Parallelizing the Adaptive Fast Multipole Method on a Shared Memory MIMD Machine

by

A. Greenbaum

Ultracomputer Note #162
June, 1989

ABSTRACT

The fast multipole method is a recently developed scheme for evaluating the potential and force fields in systems of particles whose interactions are Coulombic or gravitational in nature. When the distribution of particles is highly nonuniform, an adaptive version of the algorithm is needed. While the nonadaptive algorithm can be parallelized on an SIMD machine, it is shown that the adaptive version is equally well parallelized on an MIMD machine. Shared memory and an atomic fetch-and-add instruction make implementation especially easy. Results are presented on the 8-processor NYU Ultracomputer Prototype.

This work was supported in part by the Applied Mathematical Sciences Program of the US Department of Energy under contract DE-AC02-76ER03077 and by the Advanced Research Projects Agency of the Dept. of Defense under contract F49620-87-C-0065.

1. Introduction.

The evaluation of Coulombic and gravitational interactions in systems involving large numbers of particles is an important component of the numerical simulation of many physical processes. Areas in which such computations are used include celestial mechanics, plasma simulations, and the vortex method in fluid dynamics. A fast algorithm for evaluating the potential and force fields due to such interactions was recently introduced by Greengard and Rokhlin [7], and an adaptive variant, suitable for highly non-uniform particle distributions, was then described by Carrier, Greengard, and Rokhlin [2].

In this paper we report experiences in parallelizing the adaptive algorithm on a shared memory MIMD multiprocessor. Our starting point was the serial code of Carrier, Greengard, and Rokhlin [2]. This code consists of about 5000 lines of FORTRAN and was not written with parallelism as a primary concern. Results are presented on the 8-processor NYU Ultracomputer Prototype [5] -- a shared memory MIMD machine which simulates a true ultracomputer by providing an atomic fetch-and-add instruction allowing different processors to add to the same memory location simultaneously. MIMD capabilities are essential for parallelizing the adaptive algorithm, and the shared memory and atomic fetch-and-add operation made parallelization especially easy. Other work on parallelizing this algorithm has been reported for the non-adaptive version. The non-adaptive variant for both two and three dimensions has been implemented on the Connection Machine [8,10,11] and the two-dimensional non-adaptive algorithm has also been parallelized on the Encore Multimax [6].

Consider a two-dimensional physical model, and, without loss of generality, let two-dimensional space be represented by the complex plane. Let z_0 be the location of a point charge of unit strength and let z be any other point in the complex plane. The potential and electrostatic field at z due to the charge at z_0 are given by

$$\phi_{z_0}(z) = -log(\|z-z_0\|) = Re(-log(z-z_0))$$

and

$$E_{z_0}(z) = \frac{z-z_0}{\|z-z_0\|^2} = \left(-Re(\frac{1}{z-z_0}),\ Im(\frac{1}{z-z_0}) \right),$$

respectively. The potential and force fields due to a set of charges of strengths $q_1, \cdots, q_n$ at points $z_1, \cdots, z_n$ are obtained by summing the contributions of each individual charge:

$$\phi(z) = -\sum_{j=1}^{n} q_j log(\|z-z_j\|) = Re\left(-\sum_{j=1}^{n} q_j\ log(z-z_j) \right) \tag{1.1}$$

$$E(z) = \sum_{j=1}^{n} \frac{z-z_j}{\|z-z_j\|^2} = \left(-Re\left(\sum_{j=1}^{n} \frac{q_j}{z-z_j} \right),\ Im\left(\sum_{j=1}^{n} \frac{q_j}{z-z_j} \right) \right). \tag{1.2}$$

It follows that computing the electrostatic field at each of the points $z_1, \cdots, z_n$ is equivalent (except for the sign of the real part) to multiplying the vector $(q_1, \cdots, q_n)^T$ by the Hilbert matrix H, whose (i,j) element is

$$H_{i,j} = \begin{cases} \frac{1}{z_i - z_j}, & i \neq j \\ 0, & i = j \end{cases}.$$

Hence the multipole method can be viewed as a fast algorithm for applying a Hilbert matrix to an arbitrary vector. In this form, the problem has been studied by several others [e.g., 3,4,9], but the multipole

algorithm appears to be the most practical computational approach.

The multipole algorithm makes use of a multipole expansion to compute forces between groups of particles that are "well-separated" in the plane. It is based on the following theorem, a proof of which can be found in [7]:

Theorem. Suppose m charges of strengths q_i, $i=1,...,m$ are located at points z_i, $i=1,...,m$, with $|z_i|<r$. Then, for any $z \varepsilon C$ with $|z|>r$, the potential $\phi(z)$ is given by

$$\phi(z) = Q \log(z) + \sum_{k=1}^{\infty} \frac{a_k}{z^k},$$

where

$$Q = \sum_{i=1}^{m} q_i \qquad \text{and} \qquad a_k = \sum_{i=1}^{m} \frac{-q_i z_i^k}{k}.$$

For any $p \geq 1$,

$$\left| \phi(z) - Q \log z - \sum_{k=1}^{p} \frac{a_k}{z^k} \right| \leq \alpha \left| \frac{r}{z} \right|^{p+1} \leq \left(\frac{A}{c-1} \right) \left(\frac{1}{c} \right)^p, \tag{1.3}$$

where

$$c = \left| \frac{z}{r} \right|, \qquad A = \sum_{i=1}^{m} |q_i|, \qquad \alpha = \frac{A}{1-|r/z|}.$$

To see how this theorem is used, let $X = \{x_1, \cdots, x_m\}$ and $Y = \{y_1, \cdots, y_n\}$ be two sets of points in the complex plane which are "well-separated". That is, assume there exist points x_0 and y_0 and a real number $r > 0$ such that

$$\begin{aligned} &|x_i - x_0| \leq r \qquad i=1,...,m, \\ &|y_j - y_0| \leq r \qquad j=1,...,n, \qquad \text{and} \\ &|x_0 - y_0| \geq 3r. \end{aligned}$$

Suppose charges of strengths $\{q_1, \cdots, q_m\}$ are located at the points $\{x_1, \cdots, x_m\}$ and we wish to evaluate the potential

$$\phi(y_j) = \sum_{i=1}^{m} \phi_{x_i}(y_j)$$

at each point y_j. Clearly, a direct calculation using formula (1.1) would require $O(nm)$ operations. Alternatively, one can evaluate the coefficients a_k, $k=1,...,p$ of the p-term multipole expansion (1.3) with $O(mp)$ operations. This expansion can then be evaluated at the n points with $O(np)$ operations. A desired level of accuracy ε (say, the machine precision) can be obtained by taking p so that

$$A\left(\frac{1}{2}\right)^p \leq \varepsilon;$$

i.e.,

$$p \geq \log_2(\frac{\varepsilon}{A}).$$

Thus, to attain a fixed level of accuracy ε, only $O(n+m)$ operations are required.

For an arbitrary distribution of particles, the adaptive multipole algorithm requires dividing the particles into groups, most of which are well-separated, and then using the above procedure to compute forces between well-separated groups, while using a direct calculation to compute forces between nearby particles. The process of dividing the particles into mostly well-separated groups is less straight-forward than it might seem, however, and the total complexity of the algorithm is $O(\#particles \cdot \#levels)$, where *#levels* is the maximum number of times that the domain (or a piece of the domain) must be subdivided in order to obtain groups with no more than a fixed number of particles. In practical computations (ones that can be performed accurately using typical machine wordsizes for single or double precision arithmetic), the number of levels cannot be very large. If the division of the domain happens to be roughly uniform, then the time for computing the potential and force fields (after the particles have been divided into groups) is proportional only to the number of particles.

2. The Adaptive Multipole Algorithm.

As mentioned previously, there are two parts to the adaptive fast multipole method. The first consists of dividing the given particles into groups, most of which are well-separated. The second consists of using the multipole expansion to compute forces between well-separated groups and a direct calculation to compute forces between nearby particles. The complete algorithm is fairly complicated, and we will not cover every detail here. Rather we will give an outline which is sufficient for understanding the potential for parallelism. For a complete description of the algorithm, the reader is referred to [2].

2.1. Stage 1. Dividing the Particles into Groups.

The first part of the computation begins by enclosing the given particles in a box. It continues by dividing that box into smaller and smaller cells until no more than a fixed number, say, *s* particles lie in any one cell. The number *s* is supplied to the code by the user and can be adjusted for efficiency. The division is always carried out uniformly, as shown in Fig. 1. If a box contains more than *s* particles, then it is divided into four equal size boxes. If an empty box is created, it is simply ignored for the remainder of the computation. A list of the non-empty boxes at each level (stage of division) is maintained.

Each time a box is divided, it is required to loop through all particles of that box and determine in which of the four subboxes they lie. By choosing the particle positions perversely, one can construct distributions for which this strategy will be arbitrarily inefficient, as illustrated in Fig. 2. If almost all of the particles lie in one tiny subregion of the computational box, then the division may have to proceed for a very long time before actually dividing the area in which most of the particles lie. The width of the boxes after *nlev* levels of division, however, is 2^{-nlev} times the width of the original box. If this number is less than the machine precision (typically, $nlev \approx 24$ for single precision or $nlev \approx 48$ for double precision), then the computation cannot be performed accurately, anyway, using either the multipole algorithm or direct calculation. Hence for problems that can actually be solved, the number of levels is rather small, and the time for this division process is a small fraction of the total computation time. The number of operations required is $O(n \cdot nlev)$, where *n* is the number of particles and *nlev* the number of levels of division.

The division process can easily be parallelized by assigning different boxes to different processors. In our implementation, we parallelized only within levels, replacing serial *DO* loops over the boxes to be subdivided at that level by parallel *DOALL* loops [1]. The *DOALL* construct allows different iterates of a loop to be executed in parallel. Loop indices are kept in a work queue and dynamically assigned to processors as they become available. In this case, some loop indices required much more work than others, and it was impossible to determine before executing the loop whether a particular index would require much or little work. (One must count the particles in a given box before deciding whether it is to be divided or not.) Hence a static scheduling of loop indices to processors probably would have resulted in poor load balancing. When there are far more tasks than processors, however, (which, for large problems, there were in this

code) effects of the variation in work are minimized through dynamic scheduling.

2.2. Stage 2. Computing Forces Between Well-Separated Groups and Nearby Particles.

Having divided the domain into boxes in such a way that no unsubdivided box contains more than s particles, the algorithm then proceeds to compute the forces between well-separated boxes and between individual particles in adjacent boxes. Boxes which are not subdivided will be referred to as *childless*, while those which are subdivided will be called *parents* of the subboxes.

The first step is to compute the multipole expansion for each childless box. The multipole expansion for parent boxes can then be computed by shifting the expansion of each child to the center of the parent box (for details on how to do this, see [2]) and adding the results. The amount of work required is proportional to the total number of nonempty boxes, which usually is of order n but could be of order $n \cdot nlev$, for certain particle distributions of the sort shown in Fig. 2. This process must proceed sequentially through different levels of the boxes -- the expansions for each child must be computed before that of its parent can be computed -- but parallelism is possible within each level. Again, serial *DO* loops were replaced with parallel *DOALL* loops and a floating point fetch-and-add instruction was used so that children could add to the expansion of the same parent box simultaneously.

Interactions between particles in adjacent childless boxes are computed directly, using formula (1.1) or (1.2). Here, again, since each childless box contains no more than s particles and since it is usually adjacent to only about eight other childless boxes, the work for this step is normally proportional to the number of (nonempty) childless boxes, which is bounded by n. It is possible, however, that a childless box could have $O(nlev)$ childless neighbors and so the complexity of this step is again $O(n \cdot nlev)$. Interactions between different boxes can be computed simultaneously, as can interactions between different particles in the same or adjacent boxes. For small problems ($n \approx 100$), there may be very few childless boxes and so it is necessary to parallelize over individual particles to obtain good results in this case. For larger problems it is sufficient to use the larger grain parallelism and assign different boxes to different processors. We used only the larger grain parallelism of boxes, since parallelism at the particle level is probably best achieved through vectorization or SIMD computations.

The next stages of the algorithm deal with interactions between boxes that are not adjacent (touching in a least one point) but are also not well-separated, or, at least, not well-separated from each other's parent (their distance apart is less than one of their widths). For each box, lists of these "in-between" boxes are maintained and handled according to their precise relationship. This requires manipulation of multipole and local expansions, details of which are given in [2]. These computations are again easily parallelized by working on different boxes simultaneously and using a floating point fetch-and-add instruction to allow the fields or expansions from different boxes to be added to that of the same box simultaneously.

The final step of the computation is to shift the centers of the expansions of parent boxes to the centers of their children and compute the field in every childless box by summing the appropriate expansions, evaluating them at every particle position, and adding the results to the already computed fields due to nearby particles. Here it is required to loop through different levels of boxes -- from coarse to fine -- sequentially, but parallelism can be achieved within each level by working on different boxes simultaneously.

In summary, parallelism within each stage of the algorithm can be achieved by having different processors work on different boxes simultaneously. Some stages of the algorithm require sequentially looping through the different division levels of the boxes, while others allow parallelization over boxes at different levels. Part of the algorithm allows for parallelization at a finer level -- over particles -- but our implementation does not utilize this finer level parallelism and, as a result, will not be very efficient for small problems. Some of the stages of the algorithm could also be performed simultaneously; e.g., computing multipole expansions and computing interactions between particles in the same or adjacent childless boxes directly. We did not attempt to parallelize in this way since it would have required significant changes to the serial code. Our main mechanism for expressing parallelism was the *DOALL* loop, which replaced standard *DO* loops in the serial code and the *fetch-and-add* operation, which replaced ordinary addition in the serial code.

3. Numerical Results.

We first tried a problem with a uniform random distribution of charges throughout the unit square. The adaptive multipole method was used to compute the potential for distributions with 100, 500, 1000, 5000, and 10,000 particles. For all except the largest problem, the adaptive division of boxes turned out to be uniform, with one additional level of division being required for each larger problem size. The uniform division for the 1000-particle problem is pictured in Fig. 3a. For the 10,000-particle problem the division into subboxes was no longer strictly uniform, and the particle positions and adaptively defined boxes are pictured in Fig. 3b. For all problems, the maximum number of particles s in any childless box was taken to be 46.

Timings for the serial code and for the parallel code run on 1 to 8 processors are plotted in Fig. 4a. Note that the parallel code on 1 processor is somewhat slower than the serial code (about 10% slower), and speedups plotted in Fig. 4b are relative to the *serial* code time:

$$speedup\ (\ p\ ) = \frac{time\ (serial\ code)}{time\ (parallel\ code\ on\ p\ processors)}.$$

For the larger problems, good speedup is obtained, up to about a factor of 6.3 on 8 processors. Small parts of the code -- initializing arrays, etc. -- were not parallelized, although they could have been, with marginal improvement in execution time but fairly significant improvement in the speedup factor. In the parts of the subroutine that were actually parallelized, about a factor of 7 speedup was attained for the largest problem, and it is believed that this factor of 7 could also be achieved for the overall computation. Speedups on the current ultracomputer prototype appear to be limited to about this factor of 7, due to bus traffic.

There was little speedup from parallelization of the smallest problem. This is because parallelization was only over boxes and there were only four boxes in this case. A speedup of about 3.2 on 4 processors was achieved in the section of the code that computed interactions between particles in adjacent childless boxes. The overall speedup was less because the parts of the computation that loop between levels sequentially could not be parallelized efficiently and because the initialization sections (that could have been parallelized or vectorized but were not) accounted for a significant fraction of the total time for this small size problem. Problems of this size are better parallelized at the particle level, and for this a simple vectorization capability is probably more appropriate than the full MIMD capabilities of the ultracomputer. This points to the need for MIMD multiprocessors that also support vectorization.

Most of the computation time of the multipole code is spent in sections that parallelize over boxes at different levels or, at least, over all childless boxes -- computing interactions between particles in adjacent childless boxes and computing interactions between boxes at any level which are neither adjacent nor well-separated. Part of the algorithm, however, does require sequentially looping between levels and can be parallelized only over boxes at a given level. Parallelization of this part will be hampered if there are many levels of division, with few boxes at each level. To create such a situation and see how much parallelization is degraded, we took the uniform random number generator used to set the x and y components of the particles in the first problem and simply squared or cubed each component. This tends to cluster the particles near the origin. The distributions for the 1000-particle problem are pictured in Figs. 5a-b, along with the boxes determined by the adaptive division strategy. While the uniform distribution (Fig. 3a) required 3 levels of division and generated 64 boxes of equal size at the finest level, the squared and cubed distributions required much higher levels of division near the origin. There were 5 levels for the squared distribution and 7 for the cubed distribution.

Fig. 6 shows speedups for the different particle distributions (over the time for the serial code with that particle distribution) for the 1000-particle problem on 1-8 processors. Note that there is only slight degradation in speedup due to the extra levels of division. The time for the serial code applied to these different distributions was roughly the same as that for the uniform distribution. As expected, the slight degradation in parallel performance was due to the sections of the computation that parallelize only within levels. The other sections, which account for most of the computation time, parallelized as efficiently for these nonuniform distributions as for the uniform one.

4. Conclusions.

The adaptive fast multipole method is amenable to parallelization on a shared memory MIMD multiprocessor. MIMD capabilities are essential for parallelization and a shared memory relieves one of the worry of transporting data between processors and of the load balancing problems that would likely result from a fixed partitioning of the problem data to processors. With the convenience of *DOALL* and *fetch-and-add* constructs we were able to take a fairly complicated serial code and achieve reasonably good parallel performance with very few modifications to the original code.

References:

[1] W. Berke, *ParFOR -- A Structured Environment for Parallel Fortran*, Ultracomputer Note #137, Courant Institute of Mathematical Sciences, New York University, New York, NY, 1988.

[2] J. Carrier, L. Greengard, and V. Rokhlin, *A Fast Adaptive Multipole Algorithm for Particle Simulations*, SIAM J. Sci. Stat. Comput. 9, #4 (1988), pp. 669-686.

[3] A. Gerasoulis, M. Grigoriaddis, and L. Sun, *A Fast Algorithm for Trummer's Problem*, LCSR-TR-77, Dept. of Computer Science, Rutgers University, New Brunswick, NJ, 1985.

[4] A. Gerasoulis, *A Fast Algorithm for the Multiplication of Generalized Hilbert Matrices with Vectors*, LCSR-TR-79, Dept. of Computer Science, Rutgers University, New Brunswick, NJ, 1986.

[5] A. Gottlieb, *An Overview of the NYU Ultracomputer Project*, Ultracomputer Note #100, Courant Institute of Mathematical Sciences, New York University, New York, NY, 1987.

[6] L. Greengard and W. Gropp, *A Parallel Version of the Fast Multipole Method* to appear in Proceedings of Third SIAM Conference on Parallel Processing for Scientific Computing.

[7] L. Greengard and V. Rokhlin, *A Fast Algorithm for Particle Simulations*, J. Comp. Phys., 73 (1987), pp. 325-348.

[8] Katzenelson, *Computational Structure of the N-body Problem*, SIAM J. Sci. Stat. Comput. 10, #4 (1989), pp.787-815.

[9] L. Reichel, *A Matrix Problem with Applications to Rapid Solution of Integral Equations*, Report, Dept. of Mathematics, University of Kentucky, Lexington, KY, 1986.

[10] F. Zhao, *An O(N) Algorithm for Three-dimensional N-body Simulations*, AI-TR-995, MIT, 1987.

[11] F. Zhao and L. Johnsson, *The Parallel Multipole Method on the Connection Machine*, presented at the Fourth Parallel Circus, Rutgers University, 1988, report in preparation.