A Data-Parallel Implementation of the Adaptive Fast Multipole Algorithm[1]

by

Lars S. Nyland[2]
nyland@cs.unc.edu

Jan F. Prins[2]
prins@cs.unc.edu

John H. Reif[3]
reif@cs.duke.edu

June 1, 1993

Abstract

Given an ensemble of n bodies in space whose interaction is governed by a potential function, the N-body problem is to calculate the force on each body in the ensemble that results from its interaction with all other bodies. An efficient algorithm for this problem is critical in the simulation of molecular dynamics, turbulent fluid flow, intergalactic matter and other problems. The fast multipole algorithm (FMA) developed by Greengard approximates the solution with bounded error in time $O(n)$. For non-uniform distributions of bodies, an adaptive variation of the algorithm is required to maintain this time complexity.

The parallel execution of the FMA poses complex implementation issues in the decomposition of the problem over processors to reduce communication. As a result the 3D Adaptive FMA has, to our knowledge, never been implemented on a scalable parallel computer. This paper describes several variations on the parallel adaptive 3D FMA algorithm that are expressed using the data-parallel subset of the high-level parallel prototyping language Proteus. These formulations have implicit parallelism that is executed sequentially using the current Proteus execution system to yield some insight into the performance of the variations. Efforts underway will make it possible to directly generate vector code from the formulations, rendering them executable on a broad class of parallel computers.

1. Introduction

Solutions for N-body problems are important for solving molecular dynamics simulations, astrophysical simulations, or any other simulation where the elements of a simulation all have a mutual effect on one another (light reflected between surfaces, for instance). Straightforward solutions of the N-body problem lead to the computation of all pairwise interactions, of which there are $n(n-1)/2$, leading to an asymptotic computational complexity of $O(n^2)$. While this solution may be feasible for small solutions ($n < 1000$), it grows too rapidly for larger problems to be solved in reasonable computation times.

One technique to improve performance is to find better algorithms, and indeed several approximate solutions have been discovered that perform better than the all-pairs method. The initial alternatives were divide-and-conquer algorithms, and have been applied to yield algorithms with $O(n \log n)$ solutions, one of which is commonly referred to as the Barnes-Hut method [App85, BH86]. These algorithms rely on hierarchical decomposition and approximations that treat many bodies as one. The Barnes-Hut algorithm is very popular, as it is not too complex to implement, and tends to have fast execution.

In 1987, a new algorithm was developed for solving the N-body problem by Greengard with a complexity of $O(n)$ [Gre87]. It is called the Fast Multipole Algorithm (FMA) and relies on rather complex field mathematics to solve the problem. The mathematics are substantially more complex in three-dimensions than those used for two. The extreme complexity of the algorithm leads to both a large constant hidden by the $O(n)$ notation, as well as an extremely long development time (there are only a few known implementations of the three-dimensional solution). Like the Barnes-Hut solution, it relies on a hierarchical decomposition of space, but composes information for each of the sub-spaces from the next larger spaces, performing a constant amount of work at each node, leading to the $O(n)$ time complexity.

In addition to algorithmic improvements, another way to reduce the computation time is to look toward solutions on parallel computers. Since the sequential version of the FMA provides us with a lower bound solution to the N-body problem, we have ruled out seeking parallel tree-code solutions, such as Barnes-Hut. The task of parallelizing the FMA is to obtain a work-efficient solution. Even though the FMA has better complexity than tree-codes, we realize that the constants involved may make it more costly than other algorithms for any feasible problem size. This

1. This work supported by a grant from ARPA/ONR, N00014-91-C-00114.
2. Department of Computer Science, University of North Carolina, CB#3175, Chapel Hill, NC, 27599-3175
3. Department of Computer Science, Duke University, Durham, NC 27710

Presented at the 1993 DAGS/PC Symposium, Dartmouth College, Hanover, NH, June 23, 1993.
Copyright 1993 DAGS

in one question we are attempting to answer.

To perform algorithmic exploration, we rely upon the *Proteus* programming language. *Proteus* is a recently developed, high-level language for prototyping parallel algorithms [Nyl91, MNP+91, MNPR92, NP92], and can be used for expressing data-parallel algorithms. It supports several styles of parallelism, as well as high-level data types, such as sequences, sets and maps. One facet of our research in creating Proteus is the development of source-to-source transformations of programs, to yield code that is either more efficient or easier to translate to a particular model of parallel computation. A model currently under development specifies transformation and translation rules that apply to a data-parallel subset of Proteus, and the result is a program for a vector model of computation.

By defining an automatic translation path from data-parallel Proteus programs to programs of vector operations, we provide the programmer with a high-level model in which to program and a mechanism to achieve parallel execution. Even though the subset of Proteus that defines the data-parallel operations excludes parts of the language, it is still quite powerful. When targeting a vector model, the remainder of Proteus is not excluded, it simply does not translate to parallel operations. In contrast to other data-parallel languages, which tend to be strictly functional, the data-parallel subset of Proteus augmented with the rest of Proteus provides a programming model that is extremely powerful. We intend to demonstrate the power by expressing variants of the FMA while staying mostly in the confines of the data-parallel subset.

This paper discusses some of the solutions for the N-body problem, focusing in particular on the FMA, and an adaptive variation written in a particular style to accommodate data-parallel execution. We discuss the Proteus programming language, its support for data-parallel expression, and how programs in this form will be executed on parallel computers. We examine the performance of other published solutions, and discuss some conclusions.

2. N-body Algorithms

In this section, an overview of different N-body algorithms is presented. They rely on divide-and-conquer techniques to achieve the reduced time-complexity. Variations are also discussed that apply to the algorithms. The Barnes-Hut algorithm is included, as it is the logical successor to the all-pairs method, and precursor to the FMA.

Barnes-Hut

The Barnes-Hut algorithm splits space into 8 equal regions recursively until there are fewer than some constant k bodies per region. This creates a hierarchical decomposition, represented as a tree, where each node in the tree encompasses the regions of its children. A center of mass and total mass can be calculated for each region, with the calculations of each larger region based on its direct descendants. To compute the forces upon each body in a given region, nearby forces are calculated directly, while further and further forces are computed using the center and total mass of larger and larger regions as an approximation. The error of the simulation can be controlled by relating the distance to the size of the regions. The time complexity of the Barnes-Hut method is $O(n \log n)$, since the calculation for each body depends upon a constant amount of information from each of its ancestors.

FMA

The Fast Multipole algorithm is a linear-time algorithm, and is similar to the Barnes-Hut method in that it performs a computation over a hierarchically decomposed space, and has distinct computations for “near” and “far” bodies. It differs in the respect that the far-field forces for each body are calculated by evaluating one fixed-size multipole expansion, rather than performing a leaf-to-root tree-traversal to compute the effect of the far-field interactions. All interactions from nearby bodies are calculated directly. It also differs in that the error can be bounded by controlling the size of the multipole expansion.

The simulation space of the FMA is decomposed hierarchically, typically in a static manner to a preset level. In three dimensions, this yields an oct-tree where the root encompasses the entire space with each of its children encompassing equal-size octants of the space, repeatedly down to the leaves. The number of levels of decomposition suggested by theoretical results is $\log_8 n$, yielding an average of one body per leaf region. Virtually all interactions with this amount of decomposition are computed with multipole expansions, where the evaluation of each is expensive. Alternatively, if there is no decomposition of space, then all bodies are in the same region, leading to all interactions being calculated directly. Neither of these decompositions is adequate, the former has an extremely large constant hidden by order notation, while the latter grows too rapidly to be useful. If the number of bodies in each leaf of the decompositions is limited to no more than k, then there will be no more than $O(k^2)$

direct interactions for each of n/k regions yielding $O(nk)$ work for the near-field calculation of the algorithm. We can vary k, changing the balance of direct and multipole calculations in an attempt to improve the overall performance of the algorithm [RT93]. The execution time is balanced with the equation

$$t \propto C_1 \frac{N}{k} + C_2 Nk$$

since there are $O(n/k)$ nodes in the tree, each of which participates in the far-field calculation, and $O(nk)$ work to do for all the near interactions.

To convey the complexity of the FMA, we present the mathematics for the first phase. It is a phase that computes the coefficients of the multipole expansions at each node in the oct-tree decomposition, starting with the leaf regions and proceeding to the root. The expansions for the leaves are calculated with the potential function

$$\Phi(r,\theta,\phi) = \sum_{n=0}^{\infty} \sum_{m=-n}^{n} \frac{M_n^m}{r^{n+1}} Y_n^m(\theta,\phi)$$

$$\text{where} \quad M_n^m = \sum_{i=1}^{k} q_i \rho_i Y_n^{-m}(\alpha_i, \beta_i) \text{ for } k \text{ bodies}$$

Only a fixed number of the coefficients M are calculated (Y is the standard spherical harmonic function [Jac75]). All coordinates in the FMA are in a spherical coordinate system, as it reduces certain computational costs (see [Jac75] for details).

The expansions for the internal nodes are calculated using the expansions of their children by translating each child's multipole expansion from the center of its region to the center of the parent's region and simply adding the coefficients together where they have identical indices. The translated coefficients of a multipole expansion are computed by the function

$$M_j^k = \sum_{n=0}^{j} \sum_{m=-n}^{n} \frac{O_{j-n}^{k-m} \cdot J_m^{k-m} \cdot A_n^m \cdot A_{j-n}^{k-m} \cdot \rho^n \cdot Y_n^{-m}(\alpha,\beta)}{A_j^k}$$

where M is the new set of coefficients, and O is the old set belonging to a child. The terms A and J are functions (possibly precomputed), and (ρ, α, β) is the translation vector.

The second phase of the FMA is to compute what are called *local expansions* for each node in the oct-tree. A node's local expansion can be used to calculate the force from all bodies outside of a given radius. Its value is dependent on its parent's local expansion and the multipole expansions of some close, but not adjacent, neighbors. Similar translation formulae exist for translating local expansions and can be found in [Gre87].

The third phase is to compute the force and potential for each body in the simulation. The force on a body is the summation of the forces directly calculated from bodies in adjacent regions and the single evaluation of the local expansion for the region in which the body is contained. The computed forces can be applied, yielding new positions and velocities for each body.

Adaptive FMA

The adaptive FMA (AFMA) is a variation on the static FMA, where the fundamental difference is in the decomposition of space. Rather than decomposing the space to a preset depth everywhere, decomposition of any sub-space is continued until each region contains fewer than k bodies (somewhere in the range of 10..1000). This creates an uneven decomposition, and has a major effect only on the second phase of the algorithm.

Recall that in the second phase of the algorithm, when the local expansions are being computed, each local expansion depends on the multipole expansions of nearby, but not adjacent neighbors. In the static case, this is a constant set (possibly precomputed as a compiled-in constant). It is defined as the set of non-adjacent regions that are children of the parent's adjacent regions (some can be merged into their parent space if all the descendants exist).

The adaptive case is quite a bit more complex. Nearby spaces are categorized into one of three possibilities. The first are those spaces that are too close to use any multipole calculations, and will participate in direct interactions. The second case are those spaces that are distant enough to be considered part of the local expansion (and not included in the parent's local expansion). The last case are those regions that are in between the first two: they are the regions that are smaller than the region under consideration, but too close to be part of the local expansion. Since they are smaller, they are remote enough that their multipole expansions can be used to compute the forces on the particles in the region being examined. Figure 1 shows an examples of the three different lists.

Variations

These two algorithms, Barnes-Hut and the FMA, are very similar, they only differ in the error control and that the FMA accumulates information that can be used later on to make the algorithm linear. They both depend on a hierarchical decomposition of 3D space, and approximation methods for far interactions. This leads to the conclusion that alternative methods of decomposition can be explored that apply to both algorithms, but whatever the

result is, it must support several operations, such as quick access of neighboring regions, and the geometry of the regions created must support the mathematics involved (an important criterion for the FMA).

One decomposition that has been explored and implemented within the Barnes-Hut framework is that of *orthogonal recursive decomposition*, or ORB [WS92]. This is a decomposition of each dimension where the splitting operation places half of the bodies in each sub-space rather than half of the space. The ORB-decomposition can either cycle through each dimension in the same order, or the longest dimension of a region can be chosen as the one to split, to attempt to maintain regions that are close to cubic. What the effect of an ORB-decomposition on the FMA is unknown, but due to the success of its application in Barnes-Hut, it is well worth exploring as an alternative.

It is evident that there are many variants of the algorithm to explore, but due to the complexity of programming the variants, only a very small number have even been attempted. This type of exploration requires a development environment where it is relatively simple to develop and measure the performance of parallel algorithms, which is the true thrust of our research. We are also exploring high-level program transformation techniques to yield programs that execute in parallel.

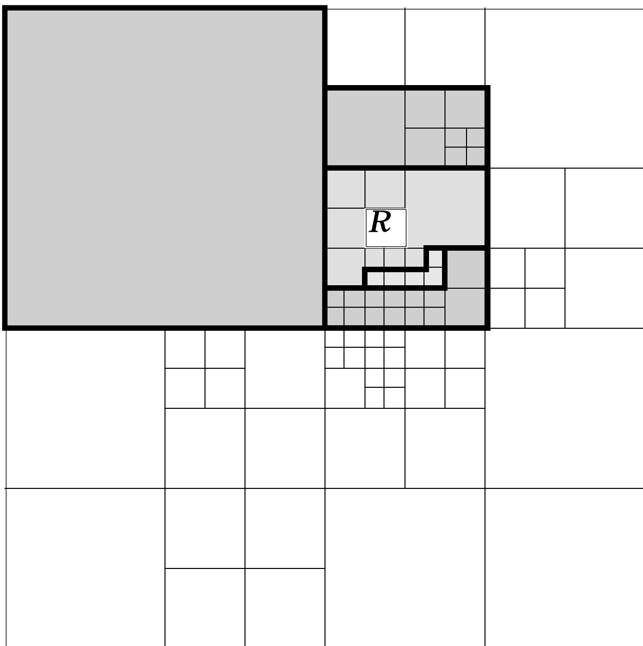
3. Proteus

The language

Proteus is a language for prototyping parallel algorithms. It supports several styles of parallelism, but for this work, we rely on only the data-parallel subset. Transformation and translation rules are under development that apply to a data-parallel subset of Proteus, and the result is a program for a vector model of computation (see [PP93] for more details).

The data-parallel subset of Proteus

Data-Parallel operations are sequences of expressions, where each expression is parameterized, and has no side-effects. These collections of expressions are usually generated with an iterator construct, creating identical code that applies to different data. A simple example is the element-by-element addition of the members of a sequence,



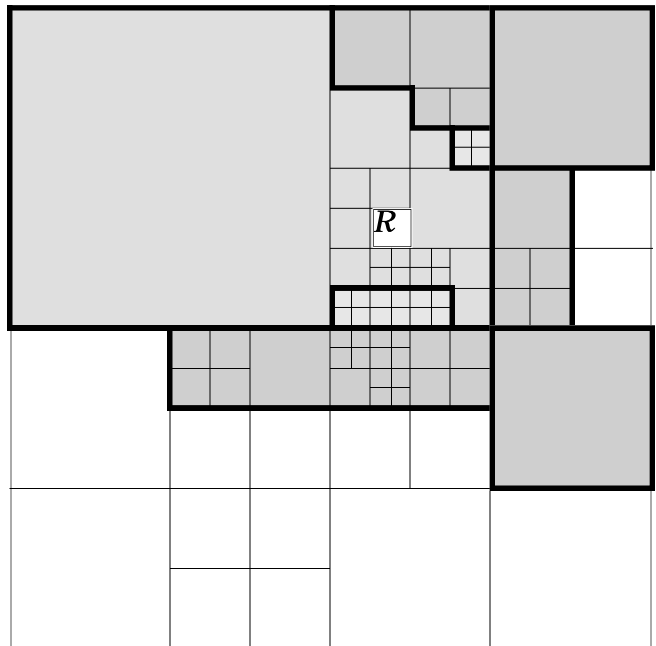


Figure 1. This figure demonstrates an example of the members of the different interaction lists in adaptive decompositions for a given box R, showing examples for separations of 1 and 2.

Notation	Meaning
`. . .` *any text*	Ellipsis start a comment, all text afterwards to the end of line is ignored
identifiers	Identifiers (names) are rather common, but the character set includes ' ' ', for names such as `f'`, to mimic mathematical notation.
$(e_\lambda)(e_1, e_2, \ldots, e_n)$	Application of a function. This is a λ-value applied to a list of expressions which are the parameters to the function.

`func` $(x_1, \ldots, x_n)$ `(`
`return` e
`)`
	λ-abstraction of an expression e with a list of parameters.

e_1 `where` $(x = e_2;$ ...)	*let*-expressions, defined as an expression with local variables whose values are defined prior to the execution of the expression.
`if` b `then` e_1 `else` e_2	A conditional expression. Both the *true* and *false* clauses must exist, since it is an expression, and therefore must have a value.
+, -, =, <, `not`, etc.	Scalar functions, unary and binary
$[e_1, \ldots, e_n]$	Sequence construction by value
$e_s[e_1]$	Sequence indexing
$\#e_s$	Sequence length
$[e_1..e_2]$	Sequence construction with a range generator
[v `in` $e_S : e_1$] [v `in` e_s | e_b: e_1]	Sequence generation with an iterator Same with boolean control
$\%+e_s$	Sequence reduction with +, other operators allowed as well
`lo`(e_s), `hi`(e_s)	The lowest and highest indices of a sequence
e_1 @ e_s	Remapping of a sequence to start at index value e_1

Table 1. The allowable constructs of Proteus for data-parallel expression

expressed as `[i in [1..#a] : a[i]+b[i]]`. This model supersedes typical vector models, in that we allow function calls, function definition, *let* clauses with single assignment variables, and nesting of data-parallel expressions. We also allow reduction and scan operations over any user-defined function (binary, associative). Nested sequences (sequences that have sequences as elements) allow us to express parallel operations not only across the elements of a sequence, but among the elements of the nested sequences as well, achieving high levels of concurrency.

Programming in a functional style using nested sequence and set generators is not only a clear style of programming, it is also amenable to parallel execution. Blelloch has pointed this out in several manuscripts [Ble93, BS90, Ble92] with the languages NESL and PARALATION LISP, and we wish to strengthen the argument with the data-parallel implementation of the FMA using Proteus. The FMA is a complex algorithm, regardless of the implementation language chosen, and we hope to demonstrate that Proteus is an excellent language for expressing the complex methods involved.

The allowable types in the data-parallel subset of Proteus are scalars such as numbers and booleans, and aggregates such as sequences and tuples. Sequences may be nested, having sequences as members. The strict definition of the allowable types is:

$$T ::= Number \mid Bool \mid Seq(T) \mid (T_1 \times \ldots \times T_n) \mid (T_D \to T_R)$$

This definition describes the types of *number*, *boolean*, *sequence*, *tuple*, and *lambda-expression*. It forces sequences to be homogenous (the type *seq*(*T*) implies a sequence of exactly one type), implying an equivalent depth when nested.

Statements such as assignment and looping are not included in this subset. This does not mean that they cannot be used, only that they are not translated into parallel execution. The data-parallel transformations are applied to programs, but only the subset defined is transformed. Then the entire program is translated, creating a program with parallelism over the subset defined, while the remainder is relegated to sequential execution.

4. Data Parallel Transformations

The general idea of the data-parallel transformations is to move the iterators deeper into expressions, and to rewrite functions that apply to sequences of their former types. This can be done repetitively yielding lengthy, nested sequences on which to operate. To give an intuitive feeling of what is being attempted, consider the simple sequence expression of

```
[i in [1..n] : sqrt(i)];
```

This is a sequence of expressions that yields a sequence whose elements are the square roots of the first *n* integers. This expression creates *n* calls to `sqrt`, incurring a certain amount of overhead for each function call.

The goal is to transform the sequence of expressions to an expression over sequences, by moving the iterators inwards. A source-to-source transformation that does this and yields a faster program is to create a new function, called `sqrt'()` that takes a sequence of numbers and returns a sequence of their square roots (typically a primitive operation on vector computers). The transformed sequence expression would then be

```
sqrt'( [i in [1..n] : i] );
```

or

```
sqrt'( [1..n] );
```

after optimization.

Only one function call is required, saving function-call overhead, even in the non-parallel case. The real benefit of this transformation comes from the subsequent translation to a vector-model, where several operations exist that take vectors as parameters and return vectors as values, yielding a high degree of concurrent execution.

The creation of a function `f'()` from the text (or abstract syntax tree) of `f()` is an automatic process. If the definition of a function is

```
f := func(x)( return expr-over-x );
```

then `f'()` is defined as

```
f' := func(x')(
    return [x in x' : expr-over-x]
);
```

As a simple example of automatic program transformation, consider the expression used to calculate the variable `RootTerm` listed in the code in section 5. The specification is

```
RootTerm := [ n in [0..P] : [ m in [-n..n] :
    sqrt(fact(n-abs(m))/fact(n+abs(m))) ]];
```

This is a doubly nested sequence, where all the individual values can be computed concurrently. The arithmetic functions are transformed to `add`, `sub`, `div`, and `neg` for '+', '-', '/' and unary '-'. Pushing the inner iterator as far into the expression as possible, we automatically derive:

```
RootTerm := [ n in [0..P] : sqrt'(
    div'(fact'(sub'([m in [-n..n]:n],
              abs'([m in [-n..n]:m]))),
    fact'(add'([m in [-n..n]:n],
            abs'([m in [-n..n]:m]))))))];
```

Continuing with the transformation, the outer iterator is pushed in, yielding:

```
RootTerm := sqrt''(
div''(
fact''(sub''(
        [n in [0..P]:[m in [-n..n]:n]],
        abs''([n in [0..P] : [m in [-n..n]: m]]
        ))),
fact''(add''([n in [0..P] : [m in [-n..n] : n]],
    abs''([n in [0..P] : [m in [-n..n] : m]]
    )))));
```

The result is code with simple data generators (range functions and distribution functions) with calls to functions that apply to nested sequences.

The transformations provided allow us to automatically rewrite the data-parallel version of the AFMA into a Proteus program that relies on vector operations and vector intrinsic functions (range, dist, extract, etc.). Once transformed, the program can be compiled producing code that relies on a well-defined model of vector operations. The vector model gives parallelism in the data-parallel expressions throughout the code.

5. Data-Parallel Variants of the FMA

In this section, we present the detailed steps of the FMA. They are presented primarily in English, hopefully in a descriptive style that conveys the operations as nested sequence operations. We also show Proteus code for one step of the algorithm to give a concrete example.

The FMA/AFMA can be summarized by outlining some initial steps to compute the decomposition which are followed by a series of steps that are repeatedly executed. The preliminary steps are:

1. Compute the decomposition of space such that there are no more than *k* bodies in each region. During the decomposition, record it by storing the center of each region created, and whether the region is a leaf, an internal node (further split), or neither (outside the decomposition).

2. Compute all the interaction lists (defined subsequently). Each region in the decomposition has one or more lists of regions that it interacts with in some way. The leaves have a different set of interaction lists than the internal nodes.

If the decomposition is used for multiple iterations, it will slowly become less than ideal, as it will no longer adequately reflect the density of bodies. The incorrect nature of the decomposition does not affect the values computed, only the time it takes to achieve them. When the decomposition is inadequate, the initialization steps can be re-executed.

Several steps of the iteration depend on the interaction lists that are created by step 2 of the initialization. These lists are defined here, and an example for one region is shown in figure 1. The calculation of these lists is not trivial, and is as important to express in a data-parallel manner as any other step of the algorithm.

The notion of what is nearby and what is not depends on the definition of *well-separated*. If the separation constant is d, two regions, r_1 and r_2, are well separated if the (Manhattan) distance between them is at least dx, where x is the length of a side of the smaller of r_1 and r_2. The separation, d, is usually 1 for two dimensions, and 2 for three dimensions (these values reasonably bound error).

1. **The direct interaction list, $\boldsymbol{D_R}$.** For a region R which is a leaf, this list contains the regions that are not well-separated from R. The proximity to R disallows the use of the multipole expansions from the regions in D_R.
2. **The far interaction list, $\boldsymbol{F_R}$.** For a region R, this list contains regions in $\boldsymbol{D_{parent(R)}}$ or any of their direct descendants that are well-separated from R.
3. **The intermediate interaction list, $\boldsymbol{H_R}$.** For a region R that is a leaf, this list contains all of the regions that fall between the direct interaction list and the far interaction list. This only has members when the direct interaction list contains regions that are smaller than R.

The steps for each iteration using the given decomposition of the simulation are:

1. Compute the multipole expansion coefficients for all leaves in the tree decomposition.
2. Compute the multipole expansion coefficients for all internal nodes in the tree with depth ≥ 2.
3. Compute the local expansion coefficients for a region R by summing R's parent's local expansion (shifted from the parent's center to the center of R) with the sum of all of the multipole expansions in R's far list, $\boldsymbol{F_R}$ (converting the multipole expansions to local expansions and shifting to the center of R) for all regions with depth ≥ 2.
4. For each body, b, in each leaf region, R, compute all the direct forces on b from all the bodies in the regions in R's direct interaction list ($\boldsymbol{D_R}$).
5. For each body, b, in each leaf region, R, compute the far force on b by evaluating the local expansion for region R at b's position.
6. For each body, b, in each leaf region, R, compute the intermediate force by evaluating the multipole expansion at b's position for each region in R's intermediate interaction list ($\boldsymbol{H_R}$).
7. Sum the 3 components of the force and potential for each body.
8. Apply the forces, updating the positions and velocities, and move the bodies to their proper regions as indicated by boundary crossing.

The representation of the hierarchically decomposed space is critical. A tree structure is a possibility, but leads to several problems. The operations on the data structure that are crucial are those of finding neighboring multipole expansions, and finding expansions belonging to parent and child nodes. A tree does not lend itself well to obtaining "all the neighbors within a distance of 2", for example, but an array, or doubly-nested sequence does. To represent the multipole expansions for all the regions, we use a sequence of 3D arrays, one for each level in the decomposition. The 3D arrays at level l are $2^l \times 2^l \times 2^l$ in size, and the multipole expansion for a region at level l has an offset $[x, y, z]$ where $0 \le x,y,z \le 2^l - 1$. A multipole expansion is stored at each point in the array, which is a depth-2 nested sequence of complex numbers. Complex numbers are represented as length-2 sequences of type `real`. The types can be listed as follows.

```
complex = seq(real);
expansion = seq(seq(complex));
grid = seq(seq(seq(expansion)));
space = seq(grid);
```

One of the interesting points to note about Proteus is that the size of the sequence (and its starting point) are not part of the type. This allows the multipole expansions to be "triangular", and it also allows the grids in the `space` type to be of different sizes. Pictorial representations of the multipole expansions and the space are shown in figures 2 and 3.

By using this structure, neighbors are easy to find ($\pm i$ in each dimension). The parent of the region at level *l* and offset [*x*, *y*, *z*] is at level *l*-1, offset [*x*/2, *y*/2,*z*/2]. The children are at *l*+1, offsets

$$\{b_x, b_y, b_z \in \{0,1\} \,:\, [2x+b_x, 2y+b_y, 2z+b_z]\}.$$

This representation also aids in the data-parallel expression, as the data structure has a type signature that fits the requirements for transformation and translation.

Computing the multipole expansions is an obvious place to begin the explanation of a data-parallel implementation, since it is important and a comparatively simple part of the algorithm. For every leaf in the tree, the coefficients of a multipole expansion must be computed. This is performed for each region with the calculation

$$M_n^m = \sum_{i=1}^{k} q_i \rho_i Y_n^{-m}(\alpha_i, \beta_i) \quad \text{where} \quad 0 \le n \le P, -n \le m \le n$$

where there are *k* bodies in the region, at the locations $(\rho_i, \alpha_i, \beta_i)$ with charge q_i (in spherical coordinates, where the origin is located at the center of the region).

The function *Y* is the standard spherical harmonic function defined as

$$Y_n^m(\theta, \phi) = \sqrt{\frac{(n-|m|)!}{(n+|m|)!}} P_n^{|m|}(\cos\theta) e^{im\phi}$$

where P_n^m is the associated Legendre polynomial, another standard function defined in [Jac75, Gre87, PFTV88].

While this seems rather complex (and it is), the point of this is that when expressed in Proteus, there are multiple nested functions applied within a sequence expression. Since nested sequence expressions can be translated to parallel code, there is no reason for a person to attempt the error-prone step of rewriting functions for vector execution.

At each level in the decomposed simulation space, there may or may not be regions containing bodies. For all of the populated regions at a given level, the multipole expansions must be computed. For level *l*, the computation is

```
multipole[l] := 0@[x in R: 0@[y in R: 0@[z in R:
        compute_M(population[l][x][y][z])
    ] ] ] where (R := [0..2^l-1]);
```

The right-hand side of this assignment statement represents a 3D array by nesting sequences to depth 3. The variable *population* is a depth-5 nested sequence, so the value of `population[l][x][y][z]` is a sequence of bodies in the region at level *l*, at offset *x*, *y*, *z*.

$$\text{expansion} = \begin{bmatrix} & & & M_{n,-n} \\ & M_{1,-1} & & \vdots \\ M_{0,0} & M_{1,0} & \cdots & M_{n,0} \\ & M_{1,1} & & \vdots \\ & & & M_{n,n} \end{bmatrix}$$

Figure 2. The triangular structure of the coefficients of a multipole expansion.

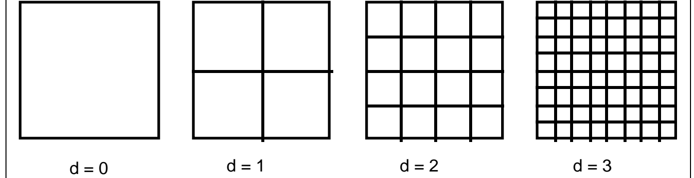


Figure 3. The change in density of grids (shown as a static 2D decomposition) as the space is further decomposed.

Realizing that all the multipole expansions, regardless of level, can be calculated independently, we can place the previous calculation for level *l* in a sequence calculation over all levels, yielding the expression

```
multipole :=
    2@[l in [2..L]:
        0@[x in R: 0@[y in R: 0@[z in R:
            compute_M(population[l][x][y][z])
    ]]] where (R := [0..2^l-1]) ];
```

The subsidiary functions needed are

```
compute_M := func(b's) (
    return
        if #b’s > 0
        then [n in [0..P]:
                [m in [-n..n]: M(n, m, b's) ] ]
        else ZeroExpansion();
);

M := func(n, m, b's) (
    return %+[b in b's :
        b.q * b.r * Y(n, -m, b.a, b.b) ];
);

... Precompute the sqrt term of Y.
... This is a global assignment.
RootTerm :=
    0@[ n in [0..P] : -n@[ m in [-n..n] :
        sqrt(fact(n-abs(m))/fact(n+abs(m))) ]];

Y := func(n, m, th, ph) (
    return
        RootTerm[n][m] *
        Legendre(n, abs(m), cos(th))
        complex(cos(m*ph), sin(m*ph));
);
```

The expression to compute the multipole expansions specifies many function applications, each of which itself specifies many more function applications, down several levels. The idea behind the transformations is to create long, segmented vectors to which vector operations may be applied, yielding a high degree of concurrency. The number of evaluations of the function Y to compute the multipole expansions for the leaves depends upon the number of bodies, N, and the index of the multipole expansions, P (typically in the range $8 \leq P \leq 20$). This leads to $N(P+1)^2$ applications of the function Y. In a typical simulation, reasonable values are $N = 10{,}000$ and $P = 9$. With these values, the number of evaluations of the function Y is 1,000,000, all of which may be done concurrently. The point of all this is to show that even though the program is not expressed within a flat vector model, there operations over long vectors even for the simple portions of the algorithm.

Rather than give explicit detail about the remaining steps with actual Proteus code, we'll rely on high-level descriptions to convey the data-parallel expression of the rest of the AFMA.

The second step is to compute the multipole expansions for all the internal nodes in the decomposition. An expansion for any internal node depends on the expansions of its children, so the levels of the tree must be done in sequence from deepest to shallowest. The data-parallel expression for each level is

for each internal node at this level, translate all of its children's expansions from their center to their parent's center, and add together the coefficients with identical indices.

The third step is to compute the local expansion for all the leaf regions, but the local expansion depends on the local expansion of the parent region, so all the local expansions for all regions at depth 2 or deeper must be computed. There is a dependence between levels, downward this time, from level 2 to the maximum depth of the decomposition. The data-parallel expression for each level is:

for every region at this level, translate its parent's local expansion from its origin to the center of this region and add it to the sum of all the multipole expansions that are converted to local expansions and translated from this region's far interaction lists.

The remaining steps actually perform the calculation of the potentials and/or forces on each body in the simulation. They each apply some calculation to each body, and the result is the sum of the three distinct calculations. The data-parallel expression of this work is not only that all the forces for each body can be computed concurrently, but that the three different force calculations are independent as well. This step does not have data dependencies , it can be expressed over all of the regions that are leaves in the decomposition.

for every body b in every region R that is a leaf, add together the force calculations from 3 sources: 1) the direct interactions of all the bodies in all the regions in R's direct interaction list, D_R, 2) the evaluation of R's local expansion at b's location, and 3) the sum of the evaluations of the multipole expansions for each region in R's intermediate interaction list, H_R.

At this point, we've done nothing more than express the algorithm in such a way that it can be written in a data-parallel fashion, using the data-parallel subset of Proteus. This algorithm has a tremendous amount of independent work to be done at several steps in the algorithm. No extra computations were introduced expressing th FMA this way, thus it is work-efficient and the complexity analysis of the AFMA remains intact.

The result of applying the transformations to the data-parallel AFMA is a Proteus program that relies on vector operations to execute data-parallel expressions. That program is then translated, yielding a vector program for a particular target architecture. The program has had several additional functions created from the existing functions by promoting the parameters and return value to sequences of their former types.

Unlike other implementations of the AFMA, our data-parallel version is load balanced, simply by the nature of expressing it in a data-parallel form. Sequences are passed across function calls, resulting in extremely large vectors and vector operations which can be run in parallel. The work specified is independent, and can therefore be evenly divided among all available processors for execution, leading to good load-balancing.

To aid us in analyzing the performance of parallel programs, we can determine some characteristics of the program. Specifically, if we can determine the number of vector steps (typically 1 or the log of the length of the vector) and the amount of work (length of the data vector) at each step, then we can state that the sequential time complexity is proportional to the work while the parallel time on a computer with an unbounded number of processors is based upon the step complexity. The estimated time complexity on a computer with p processors is $\Omega(w/p) \leq t \leq O(w/p+s)$, where w is the total work, and s is the total number of steps. This is simply explained by noting that if all the work can be evenly divided

Author	Parms	N	t	P	CPU	N/P/t	N/t
Leathrum & Board	3D, p=8	20,000	269s	1	IBM RS-6000	74	74
	3D, p=8	10,000	86s	1	IBM RS-6000	118	118
	3D, p=8	24,000	138s	16	KSR-1(preliminary)	10.9	173
Schmidt & Lee	3D, p=8	40,000	94s	1	Cray Y-MP 8/864	423	423
	3D, p=16	40,000	311s	1	Cray Y-MP 8/864	129	129
Warren & Salmon	3D, p=1	8.78M	77s	512	Touchstone Delta	222	114,025
Zhao & Johnsson	3D, p=3	16,000	5s	8k	CM-2	0.4	3200

Table 2. This table shows data for three dimensional implementations of N-body simulations. The two dimensional case of the FMA is much less complex and not relevant here. Note that the tree code is defining the upper limit on what is possible to simulate today. The FMA must outperform Barnes-Hut methods if it is ever to be adopted.

among the steps and processors, then it will take $\Omega(w/p)$ time, and if the computation is not as evenly balanced, the time complexity will be larger (but bound).

But it is possible to be even more precise about the performance of a parallel program. We can collect the step and work information about individual categories of vector instructions (e.g. arithmetic, reduction, and permute), and then in conjunction with derived machine parameters (instruction execution times), the predicted execution time of the vector code in a data-parallel program can be computed for a wide variety of parallel computers without having to run the program. The total time of the vector operations is simply the sum of the times for vector operations in each category. Therefore, an approximation of the time to execute the vector instructions is

$$\sum_i c_i \frac{w_i}{p} \le t \le \sum_i c_i\left(\frac{w_i}{p} + s_i\right) \quad i \in I$$

where I is the set of categories of instructions and c_i is the cost of executing an instruction in category i on the parallel computer of choice. This approximation becomes tighter as w becomes larger than either s or p. The result of running a program is the data for w and s for all categories of vector instructions.

The collection of work and step counts allows us to accurately determine answers to key questions about the AFMA. We can run the program over several different values of the maximum population per box, over several distinguishable distributions, yielding answers about the range of values in which the good choices fall. We can also measure the overhead of calculating the decomposition and associated lists, and determine for what distributions the AFMA performs less work.

As this paper describes work in progress, we are still evaluating the generated code from the transformations and translations. We are therefore unable to currently compare our results to others, but will be able to do so soon.

6. Comparison

Since the execution of the FMA (or any N-body simulation) is time-consuming and critical, multiple efforts have been made to achieve good parallel versions. Greengard and Gropp [GG90] parallelized the two-dimensional static FMA for an Encore Multimax, using 16 processors and gaining almost linear speedup, but the execution data is not included here, since their results only apply to the two-dimensional case. Zhao parallelized the static three-dimensional FMA on the CM-2 [Zha87], then rewrote it to reduce communications cost [ZJ91]. The precision is lower than others (error $< 10^{-2}$), thus they report very fast execution times, 5 seconds per iteration for a system of 16,000 bodies on a 16k processor CM-2. They also used expansions expressed with Cartesian coordinates, resulting in more costly multipole expansion calculations [Jac75]. Schmidt and Lee [SL91] developed a vectorizing version for the Cray Y-MP (using one processor). They claim that by using Fortran, they achieve at least a factor of 10 improvement over other programming languages (C, Pascal, Scheme), but state that the other programming language structures would be helpful (recursion, structure declarations). Leathrum and Board have a version for three-dimensional static FMA for several multicomputers, both shared and distributed memory, achieving good speedup (near linear) [LB92, Lea92]. An example system of theirs runs in 269 seconds for 20,000 bodies and eighth order multipole expansions on an IBM RS-6000 workstation. Board [Boa93] also has preliminary results of a factor of 10 increase on 16 processors on a KSR-1, with no optimization of memory use. Warren and Salmon [WS92] created a Barnes-Hut implementation augmented with multipole expansions (quadrapole) on an Intel Touchstone Delta and achieve

impressively fast simulations of immense particle systems. Execution times are 77 seconds per iteration for a system with 8.78 million bodies on a machine with 512 processors. Their precision is lower than the others, and a time factor of 8 is attributed to writing key functions in i860 assembly code. Despite the hand-coding and lower precision, their amortized cost (number of forces per second) is not substantially faster than FMA methods. The actual data is summarized in table 2.

Due to the complexity of the adaptive FMA, very little has been published about a parallel implementation. All N-body simulations that use an adaptive decomposition appear to rely on the Barnes-Hut algorithm [BH86], possibly incorporating multipole expansions [BCL92, SHG92]. There are claims that the AFMA is too complex to ever be realistic for three-dimensional simulations. Load balancing seems to be a key issue in N-body simulations using adaptive decompositions, regardless of the underlying algorithm (Barnes-Hut vs. FMA). These claims come in direct contradiction with Greengard's claim that his two-dimensional AFMA is at least 30% faster, even on uniform distributions (results in [Gre87]).

Load balancing issues are examined in [SHT+92]. They implement a two-dimensional AFMA for the DASH multicomputer, and set the maximum number of bodies to 40 (as suggested by Greengard) and later to 5 to create smaller tasks for better load balancing. In both cases, they report substantial performance penalties due to load imbalance, thus a low efficiency of parallel execution. They also report on the load-balancing of the Barnes-Hut algorithm, and get good speedup (nearly linear).

The efforts so far have shown the feasibility in relying on the FMA for N-body simulations, but they are difficult to use as a basis for exploration of the variants of the algorithm. This is where we intend to have an impact, by examining multiple variants of a complex algorithm, showing what choices are reasonable.

7. Conclusions

There are several points to be made about implementing the AFMA in a data-parallel manner. They are:

- data-parallel programming with nested sequences is a flexible and usable model for writing parallel programs.
- in using this model, automatic translations to parallel code is possible with a deterministic set of rules.
- by choosing a class of parallel architectures, the rules developed for the translation will apply to many parallel computers for years to come.
- some adaptive, unbalanced problems can be expressed in a data-parallel manner, yielding a solution that is automatically load-balanced.

The claim that the programming model provided yields better code in less time is supported by the notion that the programmer is spending very little effort thinking about the parallelization of the program or underlying machine model. The model of expressions over sequences, possibly nested, eliminates the notion of side-effect in concurrent operations, thereby making the concurrent execution non-interfering and independent. By thinking about the data on a point-by-point basis (as a programmer would if unconcerned about performance), there is no confusion introduced by attempting to write vectorized code nor the mapping of the data to processors. Since the rules allow parallel execution across function calls, the model is much more powerful than a vector model, where function calls tend to disrupt vector execution.

The transformation and translation of data-parallel Proteus programs to a vector-model is straightforward, but is still under development (only small examples are currently coming through). The translations are deterministic, and are applied as long as there are structures in the program that match the input conditions of the rules. There is no extensive analysis of the program that must be performed to use this technology, tools are currently under development that perform the transformations and translations, and are applicable to any language that is similar to the subset of Proteus chosen, such as NESL.

The resulting output, namely the parallel program, can concurrently apply parallel instructions to more data than a user could specify in a simple manner. The translation of nested sequences allows non-nested sequences to be strung together as a larger vector to which a single vector operation can be applied. The sub-vectors are unlikely to be the same length or have any particular regularity, but never-the-less, concurrency may be achieved. Acquiring this kind of concurrency "by hand" is extremely difficult to write correctly, and since it is achievable with nested sequences, we once more show the power of using nested sequences for data-parallel programming.

The target model chosen is that of vector-class machines. Certainly this is a general model, and must ignore specific features of each machine in the class, as all machines in the class do not all have the same capabilities. But by choosing the more general model, we can be assured that our results may be

applied for some time to come, as machines that represent a vector model will continue to be built much longer than any current member of this class. By choosing a general model such as this, Proteus programs will be translatable as long as the model is an effective one.

The net result is that an adaptive variant of the FMA has been written, and automatic transformation tools are currently under development that will allow it to run on several vector-class computers. It currently runs in a sequential environment, allowing us to explore and measure more variations of the algorithm.

8. References

[App85] Andrew W. Appel, An Efficient Program for Many-Body Simulations, SIAM J. Sci. Stat. Comput., Vol 6, No. 1, January, 1985.

[BCL92] S. Bhatt, M. Chen, and P. Liu. Abstractions for parallel N-body simulations. Technical Report DCS/TR-895, Yale University, 1992.

[BH86] Josh Barnes and Piet Hut. A hierarchical O(n log n) force-calculation algorithm. *Nature*, 324:446, Dec. 1986.

[Ble92] Guy E. Blelloch. Programming Parallel Algorithms, Proceedings of DAGS'92, Dartmouth College, Hanover, NH, June, 1992

[Ble93] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CMU-CS-92-103 (Updated version), Carnegie Mellon University, 1993.

[Boa93] John Board, *personal communications.*

[BS90] Guy E. Blelloch and Gary W. Sabot. "Compiling Collection-Oriented Languages onto Massively Parallel Computers," *J. of Parallel and Distributed Computing*, 8, 119-134. 1990.

[CGR88] J. Carrier, L. Greengard, and V. Rokhlin. "A Fast Adaptive Multipole Algorithm for Particle Simulations," *SIAM J. Sci. Stat. Comput.,* Vol. 9, No. 4, pp. 669-686, 1988.

[GG90] L. Greengard and W. D. Gropp. A parallel version for the fast multipole method. *Computers Math. Applic.*, 20(7):63, 1990.

[GR87] L. Greengard and V. Rokhlin. "Rapid Evaluation of Potential Fields in Three Dimensions." Yale University Research Report YALEU/DCS/RR-515, 1987.

[Gre87] Leslie Greengard. *The rapid evaluation of potential fields in particle systems.* MIT Press, 1987.

[HE81] R. W. Hockney and J. W. Eastwood, *Computer Simulation Using Particles*, McGraw Hill, 1981.

[Jac75] John D. Jackson. *Classical Electrodynamics.* John Wiley & Sons, 1975.

[LB92] James F. Leathrum and John A. Board. The parallel fast multipole algorithm in three dimensions. Technical report, Duke University, April 1992.

[Lea92] James F. Leathrum. *The Parallel Fast Multipole Algorithm in Three Dimensions.* PhD thesis, Duke University, 1992.

[MNP+91] Peter Mills, Lars Nyland, Jan Prins, John Reif and Robert Wagner. "Prototyping Parallel and Distributed Programs in *Proteus.*" In *Proc. of Third IEEE Symposium on Parallel and Distributed Processing*, 1991.

[MNPR92] P. Mills, L. Nyland, J. Prins, and J. Reif. "Prototyping N-body Simulation in *Proteus*", Proceedings of the Sixth International Parallel Processing Symposium (Beverly Hills, California, March 23-26, 1992) IEEE.

[NP92] L. Nyland and J. Prins. "Prototyping Parallel Algorithms," Proceedings of DAGS'92, Dartmouth College, Hanover, NH, June, 1992. Also available from anonymous ftp at cs.duke.edu as /pub/proteus/dags92.ps

[Nyl91] L. S. Nyland. *The Design of A Prototyping Programming Language for Parallel and Sequential Algorithms.* PhD thesis. Duke University, 1991.

[PFTV88] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.

[PP93] J. Prins and D. Palmer. Transforming High-Level Data-Parallel Programs into Vector Operations. *Proc. ACM Conf. on Principles and Practices of Parallel Programming*, ACM, May 1993. Also available as anonymous@cs.duke.edu:/pub/proteus/ppopp93.ps

[PRT92] V. Pan, J.H. Reif and S. Tate. "The Power of Combining the Techniques of Algebraic and Numerical Computing: Improved Approximate Multipoint Polynomial Evaluation and Improved Multipole Algorithms." Symp. on Foundations of Computer Science, IEEE, Pittsburgh, PA, Oct. 1992.

[RT93] J.H. Reif and S. Tate, "The Complexity of N-body Simulation," 20th Annual Colloquium on Automata, Languages and Programming (ICALP'93), Lund, Sweden, July, 1993.

[Sal90] J. Salmon. *Parallel hierarchical N-body methods.* PhD thesis, CalTech, 1990.

[SHG92] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of hierarchical N-body methods for multiprocessor architecture. Technical Report CSL-TR-92-506, Stanford University, 1992.

[SHT+92] Jaswinder Pal Singh, Chris Holt, Takashi Totsuka, Anoop Gupta and John L. Hennessy. "Load Balancing and Data Locality in Hierarchical N-body Methods." Technical report, CSL-TR-92-505, Stanford University, 1992.

[SL91] K. E. Schmidt and Michael A. Lee, "Implementing the Fast Multipole Method in Three Dimensions," *J. Stat. Phys.*, Vol. 63, Nos. 5/6, 1991.

[vGB+84] W. F. van Gunsteren, H. J. C. Berendsen, F. Colonna, D. Perahia, J. P. Hollengerg and D. Lello, "On searching neighbours in computer simulations of macromolecular systems," *J. Comp. Chem.*, 5, 272-279, 1984.

[WS92] Michael S. Warren and John K. Salmon. "Astrophysical N-body Simulations Using Hierarchical Tree Data Structures," *Proc. of Supercomputing '92.* 1992

[Zha87] Feng Zhao. *An O(n) algorithm for three-dimensional n-body simulations.* Master's thesis, MIT, 1987.

[ZJ91] Feng Zhao and S. Lennart Johnsson. The parallel multipole method on the connection machine. *J. Sci. Stat. Comput.*, 12(6), Nov. 1991.