Implementing $O(N)$ N-Body Algorithms Efficiently in Data-Parallel Languages

YU HU[1] AND S. LENNART JOHNSSON[1,2,*]

[1]*Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138*
[2]*University of Houston, Houston, TX 77204-3475; e-mail: johnsson@cs.uh.edu*

ABSTRACT

The optimization techniques for hierarchical $O(N)$ N-body algorithms described here focus on managing the data distribution and the data references, both between the memories of different nodes and within the memory hierarchy of each node. We show how the techniques can be expressed in data-parallel languages, such as High Performance Fortran (HPF) and Connection Machine Fortran (CMF). The effectiveness of our techniques is demonstrated on an implementation of Anderson's hierarchical $O(N)$ N-body method for the Connection Machine system CM-5/5E. Of the total execution time, communication accounts for about 10–20% of the total time, with the average efficiency for arithmetic operations being about 40% and the total efficiency (including communication) being about 35%. For the CM-5E, a performance in excess of 60 Mflop/s per node (peak 160 Mflop/s per node) has been measured.

1 INTRODUCTION

Achieving high efficiency in hierarchical methods on massively parallel architectures is an important problem. Hierarchical methods are the only feasible methods for large-scale computational problems involving many-body interactions, such as astrophysical simulations and molecular dynamics simulations including long-range forces. This article examines techniques for achieving high efficiency in implementing nonadaptive $O(N)$ N-body algorithms on massively parallel processors (MPPs). It also provides Connection Machine Fortran (CMF) [1] code fragments that illustrate how to express the techniques in a data-parallel language.

CMF was chosen because no High Performance Fortran (HPF) [2] compiler was available at the time of this project. The techniques we discuss result in high performance by

1. Reducing the need for data motion between nodes.
2. Avoiding local memory copying by specifying operations such that state-of-the-art compilers can pass arrays in place.
3. Reducing the need for memory bandwidth by organizing computations for a high degree of register reuse without a need for interprocedural analysis.
4. Increasing vector length and/or reducing the number of DRAM page faults and TLB thrashing by aggregating computations for collective function calls, increasing the degrees of freedom in scheduling operations for the function execution.

*Work conducted when author was with Thinking Machines Corporation and Harvard University, Cambridge, Massachusetts.

Received October 1994
Revised December 1995

Scientific Programming, Vol. 5, pp. 337–364 (1996)
CCC 1058-9244/96/040337-28

Efficient memory management is the most challenging issue in high-performance computing. Techniques for the automatic determination of data distributions with balanced load and efficient communication have been the focus of parallel compiler research in the last several years [3–5]. However, no general technique that balances load and generates efficient communication has emerged so far. User inputs in the form of data distribution directives and careful structuring of the code based on knowledge of the memory architecture play a much more important role in obtaining high performance in programming distributed memory machines in a high-level language than in programming uniprocessors. The added features in HPF, relative to Fortran 90, is a step toward giving users the ability to provide helpful information to the compiler.

The hierarchical $O(N)$ N-body algorithms we consider consist of near-field and far-field interactions. For the former, the interaction between groups of particles is handled by the direct, classical N-body algorithm. Far-field interactions are handled through a hierarchy of computational elements corresponding to subdomains, "boxes," with parent and children boxes having the obvious meaning. Part of the far-field close to a box is known as the interactive-field. The computations on the box hierarchy are carried out through three translation operators, two of which are used in parent–child interactions, while the third is used in handling the interactive-field. These terms are defined precisely in the next section. Our novel contributions are techniques, expressed in CMF, for

1. Very limited data motion in parent-child interactions.
2. Limited data motion in neighbor interactions for interactive-field computations.
3. Trading off redundant computation vs. communication.
4. Representing translation operations as matrix–vector multiplications.
5. Aggregating multiple independent translation operations into multiple instances of matrix–matrix multiplications.
6. Reducing the number of translation operations through a novel use of supernodes.
7. Efficiency in memory usage.

By using a careful allocation of the hierarchy of boxes, the data motion between a parent and its children is largely turned into local memory references instead of internode communication. We use one array for the leaf-level boxes and one array for all other boxes. All boxes not at the leaf-level are packed into one array of the same size as the array for leaf-level boxes. Our packing guarantees that once there is at least one box per processing node, then all its children boxes are assigned to the same processing node. The packing is described in "The Hierarchy of Boxes," a subsection in Section 5.

The virtual machine model provided by CMF (and HPF) with a global address space easily results in excessive data motion. For instance, performing a circular shift on an array causes most compilers to issue instructions that move every element as specified by the shift instruction. A more efficient way of dealing with shift instructions is to move data between processing nodes as required, but to eliminate the local memory moves by modifying subsequent local memory references to account for the specified move (that was not carried out). This issue is of particular importance in gathering boxes for the interactive-field computations. In Section 6 we show how to use array sectioning and array aliasing to implement an effective gathering of nonlocal interactive-field boxes. On the Connection Machine systems CM-5/5E [6], the communication time for the straightforward use of CMF for interactive-field computations is more than one order of magnitude higher than the communication time for the technique we use.

In Anderson's [7] version of the fast multipole method, all translation operators can be represented by matrices acting on vectors. Moreover, the translation matrices are scale invariant and only depend on the relative locations of the sources and destinations. Hence, the matrices are the same for all levels of the hierarchy, and all source destination pairs with the same relative locations use the same matrices at any level in the hierarchy. Thus, all matrices can be precomputed. The translation operators in fast multipole methods [8–10] can also be viewed as matrix–vector multiplications [11]. We discuss this arithmetic optimization in Section 7. By representing the translation operations as matrix–vector multiplications and aggregating the matrix–vector multiplications into multiple-instance matrix–matrix multiplications, many of the translation operations can be performed at an efficiency of about 80% of peak, or at a rate of 127 Mflop/s per node of a CM-5E using the Connection Machine Scientific Software Library, CMSSL [12]. Recognizing and aggregating BLAS operations and using library functions

Table 1. Efficiencies of Various Parallel Implementations of Hierarchical N-Body Methods

Author	Programming Model	% of Peak Efficiency	Machine
Salmon, Warren-Salmon [13–15]	F77 + message passing	24–28%	512-node Intel Data
Liu-Bhatt [16, 17]	C + message passing + assembly	30%	256-node CM-5
Leathrum-Board et al. [19, 20]	F77	20%	32-node KSR-1
Elliott-Board [21]	F77	14%	32-node KSR-1
Zhao-Johnsson [18]	*Lisp + assembly	12%	256-node (8k) CM-2
Hu-Johnsson [this article]	CMF	27–35%	256-node CM-5/5E

can significantly improve the performance of the computations on most architectures, including uniprocessor architectures.

For parent–child interactions three-dimensional problems require eight translation matrices in the upward and downward traversals of the hierarchy of grids. In addition, a large number of translation matrices are required for neighbor interactions in the downward traversal. For hierarchies with at least four levels, each processing node containing leaf-level boxes distant four from domain boundaries requires a copy of all matrices. We discuss the trade-off between replication and redundant computation in Section 8.

Although our implementation reported in this article is in CMF, all the CMF constructs used, with one exception, are available in HPF. The exception—array aliasing—can be achieved straightforwardly using extrinsic procedures in HPF, as described in Section 3. The array-aliasing feature has the clear advantage over extrinsic procedures that a programmer can express the optimizations in the data-parallel programming model instead of resorting to the message-passing SPMD (single program, multiple data) style programming model. An array-aliasing mechanism is being considered for inclusion in HPF II [13].

Most of our optimization techniques apply to any distributed memory machine. However, the relative merit of the techniques depend on machine metrics. We report on the performance trade-offs on the CM-5/5E.

To our knowledge, this work represents the first implementation of Anderson's method on a parallel machine as well as the first implementation of an $O(N)$ N-body algorithm in a data-parallel language. Table 1 summarizes the efficiencies of several parallel implementations, including the results reported in this article. The efficiency numbers should be viewed with some caution since the various implementations used different algorithms, different problem sizes, and parameters controlling the accuracies. The Barnes-Hut $O(N \log N)$ algorithm has been implemented using the message-passing programming paradigm by Salmon and Warren [14–16] on the Intel Touchstone Delta and by Liu and Bhatt [17, 18] on the CM-5. Both groups used assembly language for time critical kernels and achieved efficiencies in the range 24%–28% and 30%, respectively. Zhao and Johnsson [19] developed a data-parallel implementation of Zhao's method on the CM-2, and achieved an efficiency of 12% for expansions in Cartesian coordinates that results in more costly multipole expansion calculations. Leathrum and Board et al. [20, 21] and Elliott and Board [22] achieved efficiencies in the range 14%–20% in implementing Greengard-Rokhlin's method [23] on the KSR-1. Schmidt and Lee [24] vectorized this method for the Cray Y-MP and achieved an efficiency of 39% on a single processor. Singh et al. [25, 26] have implemented both $O(N \log N)$ and $O(N)$ methods on the Stanford DASH machine, but no measures of the achieved efficiency are available. Nyland et al. [27] discussed how to express the three-dimensional (3-D) adaptive version of the Greengard-Rokhlin method in a data-parallel subset of the Proteus language, which is still under implementation.

This article is organized as follows. Section 2 describes the computational structure of hierarchical methods, and details the computational elements of Anderson's method. Section 3 briefly summarizes new features in HPF and describes the array-aliasing mechanism in CMF that currently is not included in HPF. Section 4 presents the architecture of the Connection Machine systems CM-5/5E. The optimization techniques for programming hierarchical methods in CMF (HPF) are presented in Sections 5–9. Section 10 reports some performance results of our implementation. Additional performance data can be found [see 37]. Section 11 discusses load-balancing issues and Section 12 summarizes the article.

2 HIERARCHICAL *N*-BODY METHODS

Hierarchical methods [7, 8, 9, 28] for the N-body problem exploit the linearity of the potential field by partitioning the field into two parts,

$$\phi_{total} = \phi_{near\text{-}field} + \phi_{far\text{-}field}, \quad (1)$$

where $\phi_{near\text{-}field}$ is the potential due to nearby particles and $\phi_{far\text{-}field}$ is the potential due to faraway particles. The near-field is evaluated through the classical N-body technique of pair-wise interactions between all particles in the near-field. Hierarchical methods achieve their arithmetic efficiency by evaluating the far-field potential hierarchically:

1. The field induced by a cluster of particles sufficiently far away from an evaluation point is modeled by a single computational element, called far-field potential representation.
2. Computational elements for small domains are hierarchically combined into elements for large domains.
3. The far-fields due to computational elements at different levels in the hierarchy are evaluated hierarchically.

The hierarchy of computational elements is established through a hierarchy of grids (see Fig. 1). Grid level 0 represents the entire domain. Grid level $l + 1$ in a nonadaptive decomposition is obtained from level l by subdividing each region into four (in two dimensions) or eight (in three dimensions) equally sized subregions. The number of distinct boxes at mesh level l is equal to 4^l and 8^l for two and three dimensions, respectively. Subdomains that are not further decomposed are leaves. In two dimensions, the near-field contains those subdomains that share a boundary point with the considered subdomain. In three dimensions, Greengard-Rokhlin's formulation [29] defines the near-field to contain nearest neighbor subdomains which share a boundary point with the considered subdomain and second nearest neighbor subdomains which share a boundary point with the nearest neighbor subdomains. The far-field of a subdomain is the entire domain excluding the subdomain, the target subdomain, and its near-field subdomains. The far-field is said to be well separated from the target subdomain with respect to which it is defined. In the following, we often refer to a (sub)domain as a "box." The interactive-field of a target box at level l is the part of the far-field that is contained in the near-field of the target box's parent. In three dimensions, the number of subdomains in the near-field is 124 (a $5 \times 5 \times 5$ subdomain excluding the target box) and the number of subdomains in the interactive-field is 875 (a $10 \times 10 \times 10$ subdomain excluding the near-field and the target box). In two dimensions, the near-field contains eight subdomains and the interactive-field contains 27 subdomains, respectively. A discussion of a less stringent definition of the near-field in three dimensions can be found in [30].

The idea of the hierarchical combining and evaluation used in the $O(N)$ algorithms is illustrated in Figure 2. At level 2 of the hierarchy, the two subdomains marked with "i" are well separated from subdomain B. Thus, the computational elements for those two subdomains can be evaluated at the particles in subdomain B. At level 3, 15 new subdomains marked with "i" are well sep-

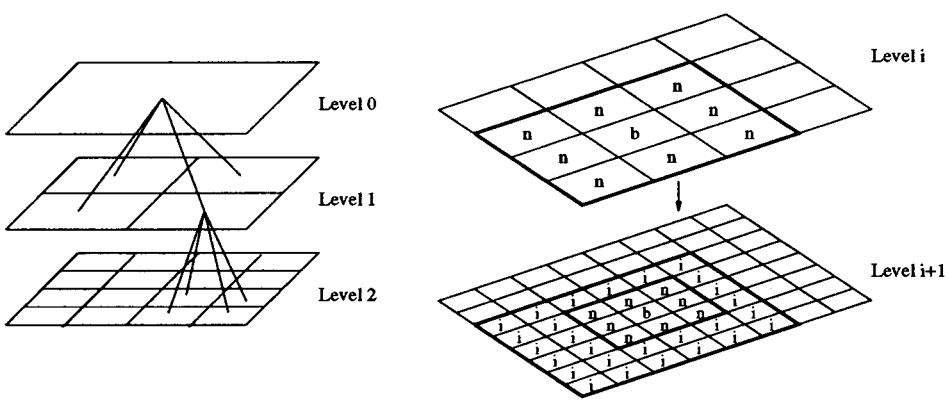


FIGURE 1 Recursive domain decompositions and the near-field and interactive-fields in two dimensions.

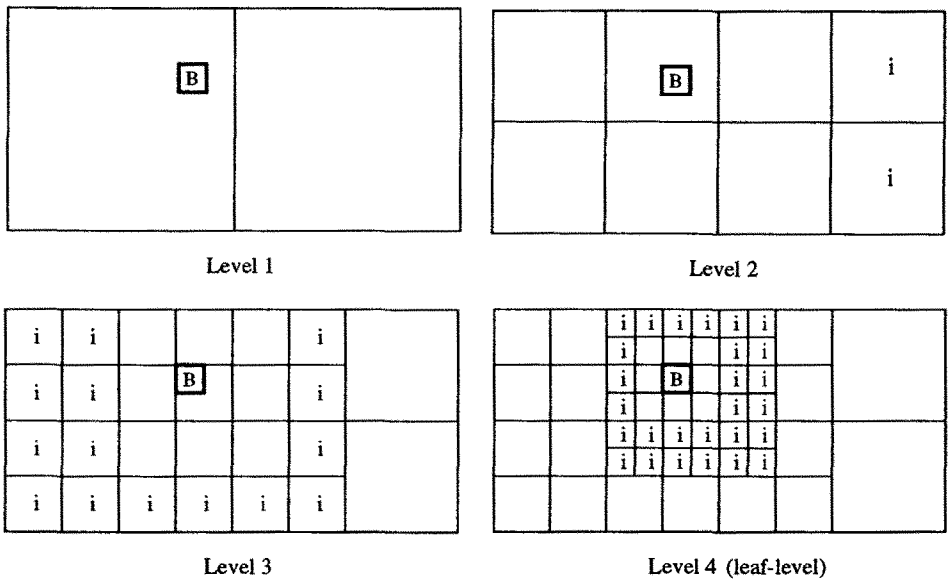


FIGURE 2 Interactive-fields for a hierarchical method.

arated from subdomain B and their computational elements can also be evaluated at the particles in subdomain B. At level 4 there are 27 new computational elements that are well separated from the particles in subdomain B. At any of the levels, the domains marked with "i" define the interactive-field at that level. If domain B would have been smaller, then it is easy to see that for all levels beyond level 4, the interactive-field will always have 27 computational elements, which is the maximum for any subdomain. The above process continues until the leaf-level is reached, at which point the far-field potential of the particles in B has been computed.

For uniform particle distributions and a hierarchy depth of $\log N$, the total number of computational elements in the hierarchy is $O(N)$ and there are only a few particles ($O(1)$) in each leaf-level computational element. For a hierarchy depth of $\log N$, the total number of computational elements evaluated at each particle is $O(\log N)$ and the evaluation of all particles' far-field potential requires $O(N \log N)$ operations. The reduction in the complexity of the interactive-field evaluation to $O(N)$ is achieved by introducing a local-field potential representation—a second kind of computational element—to combine the evaluation of computational elements that are "far away" from clusters of particles. This element approximates the potential field in a "local" domain due to particles in the far domain. The new type of computational element allows contributions from different interactive-field domains to be combined for the far-field evaluation with respect to all subdomains making up the new computational element. Conversion from the far-field potential representation (of the far domains) to the local-field potential representation (of the subdomain under consideration) is needed.

In practice, computational elements are approximated by finite length series. The accuracy is controlled by the number of terms included in the expansion. Estimates of the approximation errors as a function of the number of terms in the expansion have been derived for the different methods [7, 9, 10], and are not discussed here.

Hierarchical methods compute $\Phi_{far\text{-}field}$ of (1) in two hierarchy-traversing passes. In an upward pass, the far-field potential of computational elements is combined (T_1) to form Φ_i^l by shifting the far-field potential representation of child boxes from their respective centers to the center of their parent box, and adding the resulting representations (coefficients). Φ_i^l is the contribution of subdomain i at level l to the potential field in domains in its far-field. In a downward pass, the far-field potential of interactive-field boxes is converted into local-field potentials (T_2) which are combined with the local-field passed from a parent to its children by shifting the parent's local-field representation to the centers of its child boxes (T_3). Let Ψ_i^l represent the contribution to the potential field in subdomain i at level l due to particles in the far-field of subdomain i, i.e., the local-field potential in subdomain i at level l. Then, the computational

structure is described in the recursive formulation by Katzenelson [31]:

Algorithm: (A generic hierarchical method)

1. Compute Φ_i^h for all boxes i at the leaf-level h.
2. Upward pass: for $l = h - 1, h - 2, \ldots, 2$, compute

$$\Phi_n^l = \sum_{i \in \{children(n)\}} T_1(\Phi_i^{l+1}).$$

3. Downward pass: for $l = 2, 3, \ldots, h$, compute

$$\Psi_i^l = T_3(\Psi_{parent(i)}^{l-1} + \sum_{j \in \{interative\text{-}field(i)\}} T_2(\Phi_j^l).$$

4. Far-field: evaluate local potential at particles inside every leaf-level subdomain.

$$\phi_{k\ far\text{-}field} = \Psi_{box(k)}^h(k)$$

5. Near-field: evaluate the potential field due to the particles in the near-field of leaf-level subdomains, using a direct evaluation of the Newtonian interactions with nearby particles.

$$\phi_{k\ near\text{-}field} = \sum_{j \in \{near\text{-}field(k)\}} G_j(k),$$

where G is the potential function in an explicit Newtonian formulation.

For N, uniformly distributed particles and a hierarchy of depth h having $M = 8^h$ leaf-level boxes, the total number of operations required for the above generic hierarchical method is

$$\begin{aligned} T_{total}(N, M, p) = {} & O(Np) + O(f_1(p)M) \\ & + O((N_{int} \cdot f_2(p) + f_3(p))M) \\ & + O(Np) + O\left(\frac{N^2}{M}\right), \end{aligned}$$

where p is the number of coefficients in the field representation for a computational element: $f_1(p)$, $f_2(p)$, and $f_3(p)$ are the operation counts for T_1, T_2, and T_3, respectively; and N_{int} is the number of interactive-field boxes for interior nodes. The five terms correspond to the operation counts for the five steps of the hierarchical methods. The minimum value of T_{total} is of order $O(N)$ for $M = c \cdot N$, i.e., the number of leaf-level boxes for the optimum depth of the hierarchy is proportional to the number of particles. Since the terms linear in M represent the operation counts in traversing the hierarchy, and the term $O(N^2/M)$ represents the operation counts in the direct evaluation in the near-field, the optimal hierarchy depth balances the cost of these two phases. Thus, it is equally important to efficiently perform the hierarchical computations and the direct evaluations at the leaf-level in the hierarchical methods.

Moreover, it is worth noting that because of the large constant in the complexity of hierarchical methods, direct methods outperform Anderson's method up to about 4,500 particles, the Barnes-Hut algorithm up to about 6,000 particles, and the Greengard-Rokhlin method for up to about 9,000 particles in three dimensions and with an accuracy of an error decay rate of four in the multipole methods.

2.1 Anderson's Multipole Method without Multipoles

Anderson [7] used Poisson's formula to represent solutions of Laplace equation. Let $g(x, y, z)$ denote potential values on a sphere of radius a and denote by Ψ the harmonic function external to the sphere with these boundary values. Given a sphere of radius a and a point $\vec{x}$ with spherical coordinates (r, θ, ϕ) outside the sphere, let $\vec{x_p} = (cos(\theta)sin(\phi), sin(\theta)sin(\phi), cos(\phi))$ be the point on the unit sphere along the vector from the origin to the point $\vec{x}$. The potential value at $\vec{x}$ is (Equation 14 of [7])

$$\Psi(\vec{x}) = \frac{1}{4\pi} \int_{S^2} \left[\sum_{n=0}^{\infty} (2n+1) \left(\frac{a}{r}\right)^{n+1} P_n(\vec{s_i} \cdot \vec{x_p}) \right] g(a\vec{s}) ds, \quad (2)$$

where the integration is carried out over S^2, the surface of the unit sphere, and P_n is the nth Legendre function.

Given a numerical formula for integrating functions on the surface of the sphere with K integration points $\vec{s_i}$ and weights w_i, the following formula (Equation 15 of [7]) is used to approximate the potential at $\vec{x}$:

$$\Psi(\vec{x}) \approx \sum_{i=1}^{K} \left[\sum_{n=0}^{M} (2n+1) \left(\frac{a}{r}\right)^{n+1} P_n(\vec{s_i} \cdot \vec{x_p}) \right] g(a\vec{s_i}) w_i. \quad (3)$$

This approximation is called an outer-sphere approximation. Note that in this approximation the series is truncated and the integral is evaluated with a finite number of terms.

The approximation used to represent potentials inside a given region of radius a is (Equation 16 of [7])

$$\Psi(\vec{x}) \approx \sum_{i=1}^{K} \left[\sum_{n=0}^{M} (2n+1) \left(\frac{r}{a}\right)^{n+1} P_n(\vec{s_i} \cdot \vec{x_p}) \right] g(a\vec{s_i}) w_i \tag{4}$$

and is called an inner-sphere approximation.

The outer-sphere and the inner-sphere approximations define the computational elements in Anderson's hierarchical method. Outer-sphere approximations are first constructed for clusters of particles in leaf-level boxes. During the upward pass, outer-sphere approximations of child boxes are combined into a single outer-sphere approximation of their parent box (T_1) by evaluating the potential induced by the component outer-sphere approximations at the integration points of the parent outer-sphere approximation, as shown in Figure 3. The situation is similar for the other two translations used in the method, which are shifting a parent box's inner-sphere approximation to add to its children's inner-sphere approximations (T_3) and converting the outer-sphere approximations of a box's interactive-field boxes (T_2) to add to the box's inner-sphere approximation.

3 HIGH PERFORMANCE FORTRAN

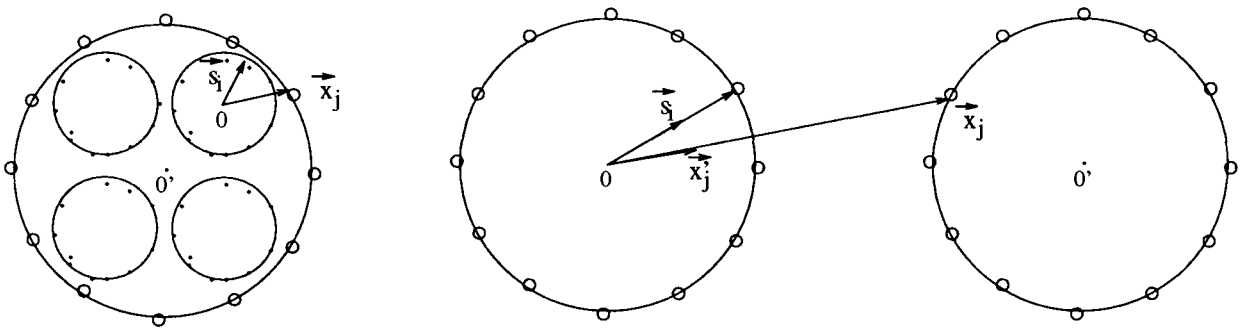
Since no HPF compiler was available when this work was initiated, we used the CMF language [1] for our implementation. All CMF constructs used, except the array-aliasing mechanism, are available in HPF. Below, we briefly summarize the new features in HPF. We then present the array-aliasing mechanism in CMF, which provides an elegant way to avoid excess data motion, and compare it to the use of extrinsic procedures for the same purpose.

HPF consists of Fortran 90 with extensions mainly for data management. The main extensions are:

1. Data distribution directives, which describe data aggregation, such as cyclic and block aggregation, and the partitioning of data among memory regions.
2. Parallel FORALL statements and constructs, which allow fairly general array sectioning and specifications of parallel computations.
3. Extrinsic procedures (local procedures), which define interfaces to procedures written in other programming paradigms, such as explicit message-passing SPMD styles.
4. A set of extended intrinsic functions, including mapping inquiry intrinsic subroutines that allow a program to know the exact mapping of an array at run-time.

HPF supports data-parallel programming with a global address space. Programs can be written without any knowledge of the architecture of the memory system. The consequence is that most compilers often generate excess data movement. Cyclic shifts are a good example already discussed in the introduction. One sensible way of avoiding excess data movement is to restructure the program in a way that even a not-so-sophisticated compiler is able to generate efficient code. This goal can be achieved by exposing the local memory and processor address spaces and giving a programmer explicit control over data allocation and data references.

In CMF, separation of the local and processor address spaces is elegantly achieved through array



(a) Translations T1 and T3

(b) Translation T2

FIGURE 3 Translations as evaluations of the approximations.

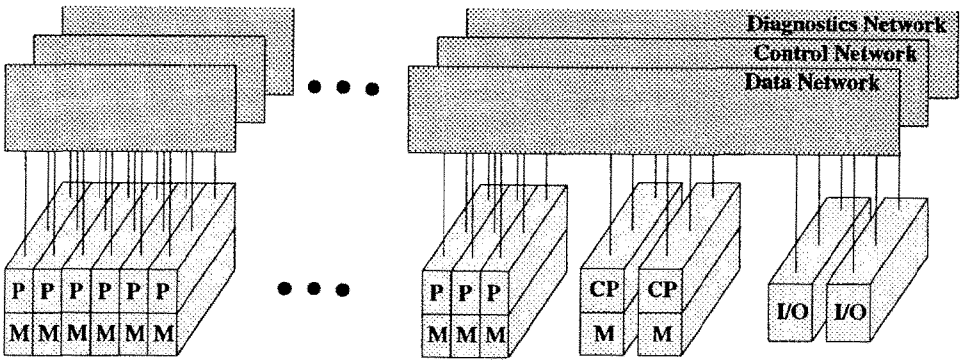


FIGURE 4 Overview of the CM-5/5E system.

sectioning and array aliasing within the global programming paradigm. Array sectioning is part of Fortran 90 and HPF, but array aliasing is not. The array-aliasing mechanism allows a user to address memory already allocated for an array, as if it were of a different type, shape, or layout. No data motion occurs. For example, let A be an n-dimensional array with extents $L_1 \times \ldots \times L_n$. Assume that after mapping A onto the physical machine, there are p_i nodes used for axis i, resulting in a subgrid of length s_i within each node for axis i, i.e., $L_i = s_i \times p_i$. Using array aliasing, we can create an array alias A_{alias}, which has extents $s_1 \times \ldots \times s_n \times p_1 \times \ldots \times p_n$, with the first n axes local to each node and the last n axes purely off-node. In this way, we have explicitly separated the local address space from the processor address space. This subgrid equivalencing feature in CMF provides a means of managing memory accesses similar to that of the EQUIVALENCE statement in Fortran 77. It is heavily used in the optimization techniques discussed in the rest of this article.

In the current version of HPF, a separation of local and processor address spaces can only be achieved through the use of extrinsic (local) procedures. Within a local procedure, a program can access directly only the memory local to a node. Access to other parts of the global memory must either be made through explicit message passing, or by returning to the global program. Hence, within HPF, optimizations based on separation of address spaces cannot be achieved within the language itself, but only by mixing programming models (data parallel and message passing). Moreover, mixing programming models and using procedure calls increase the difficulty of many forms of compiler optimizations and array aliasing is being considered for inclusion in HPF.

4 CM-5/5E ARCHITECTURE

A CM-5/5E system contains up to 16,384 parallel processing nodes (the largest configuration available today has 1,024 nodes), each with its own memory (see Fig. 4). A collection of nodes, known as a "partition," is supervised by a control processor called partition manager, although the nodes may operate independently in a multiple instruction, multiple data mode (MIMD). Each node is connected to two low-latency, high-bandwidth interconnection networks, the Data and the Control Networks. The Data Network is generally used for point-to-point internode communication, and the Control Network for operations such as synchronization, broadcasting, and parallel prefix operations. A third network, the Diagnostics Network, is used to ensure the proper operation of the system.

Figure 5 illustrates the architecture of a single node of a CM-5/5E. Each node is a SPARC microprocessor, with four virtual vector units (VUs) emulated by two physical VUs for enhanced floating-point performance. In the following we always refer to the virtual VUs simply as VUs. The VUs are memory mapped into the SPARC address space. The SPARC serves as a controller and coordinator for the four VUs. Each VU consists of an M-bus interface and an instruction decoder as well as an ALU. Each VU has its own Register File. The assembly instruction set contains vector instructions for a four-stage pipeline. The ALU can perform a floating-point or integer multiply–add or multiply–subtract operation on 64-bit operands per clock cycle. The ALUs also support 32-bit operations, but the computational rate is the same as in 64-bit precision. Each VU can address up to 128 MB of memory, giving a maximum of 512 MB/PN (PN = node). The path between each

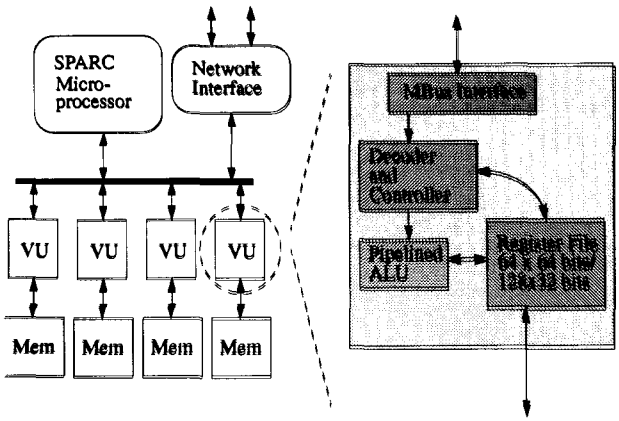


FIGURE 5 Overview of the CM-5/5E node with vector units.

VU and its memory is 64-bit wide, and the pipelined VUs can access memory on each clock cycle. The clock frequency of the node processor and the two physical VUs is 40 MHz for the CM-5E (32 MHz for the CM-5). The four VUs run at half this clock frequency, and so do the four memory banks, one for each of the four VUs. Thus, the peak performance of a VU is 40 Mflop/s and the peak performance of a node is 160 Mflop/s. The maximum bandwidth to memory is 640 MB/s/PN.

The VU memory is dynamic RAM (DRAM), with a DRAM page size of 8 KB for 4-Mbit memory chips and 16 KB for 16-Mbit memory chips. The memory per VU is 8 and 32 MB respectively, for 4- and 16-Mbit memory chips. If the VU accesses two successive memory locations which are not on the same DRAM page, a page fault occurs. If a DRAM page fault occurs, the pipeline is stalled for 5 VU cycles, and hence it is desirable to organize the scheduling of operations such that the number of DRAM page faults is minimized. In addition, only 64 DRAM pages are mapped into the SPARC address space at all times. Hence, the order in which DRAM pages are traversed may have a significant impact on performance through TLB thrashing.

5 DATA STRUCTURES AND DATA DISTRIBUTION

We start the discussion of our techniques for programming hierarchical methods in data-parallel languages with the data structures used and how they are distributed across the memories. We often refer to the distribution of array data across memories as the data or array layout.

5.1 Data Structures and Their Layout

There are two main data structures in a hierarchical method: one for storing the potential field in the hierarchy and the other for storing particle information.

The Hierarchy of Boxes

Data Representation. Far-field potentials are stored for all levels of the hierarchy, since they are computed in the upward pass and used in the downward pass. We embed the hierarchy of far-field potentials in one 5-D array that effectively consists of two 4-D arrays with the same layout. Three of the axes represent the organization of the boxes in the three spatial dimensions, while the fourth axis is used to represent data local to a box. The 5-D array representation of the potential field is quite effective with respect to memory utilization, yet can easily be made to guarantee locality in traversing the hierarchy. Moreover, the 5-D array representation is easy to use for any depth of the hierarchy; only the extent of the three spatial axes depends on the depth of the hierarchy. Representing each level of the hierarchy as a separate array can clearly be made more memory efficient, but the number of arrays depends on the depth of the hierarchy. Using arrays with one of the axes representing the levels of the hierarchy would require ragged arrays for space efficiency. But, ragged arrays are neither supported in CMF nor in HPF.

The declaration of the far-field potential array in CMF is*:

```
      REAL*8 FAR_POT(2,K,L,M,N)
CMF$LAYOUT FAR_POT(:SERIAL,:SERIAL,:,:,:)
```

The compiler directive above specifies that the rightmost three axes are parallel axes and that the two leftmost are local to each VU (specified through the attribute :SERIAL OR * in HPF). The right most three axes represent the subdomains at the leaf-level of the hierarchy along the $z-$, $y-$, and x-coordinates, respectively. The local axis of extent K is used to store the potential field values at the integration points of a subdomain in Anderson's method (or the coefficients of a multipole

* All the code examples in this article will be in CMF.

expansion in Greengard-Rokhlin's method). The leaf-level of the potential field is embedded in one layer of the 5-D array, FAR_POT(1, :, :, :, :), and levels $(h - i)$ are embedded in FAR_POT $(2, :, 2^{i-1} : L : 2^i, 2^{i-1} : M : 2^i, 2^{i-1} : N : 2^i)$ (see Fig. 6). The embedding preserves locality between a box and its descendants in the hierarchy. If at some level there is at least one box per VU, then for each box all its descendants will be on the same VU as the box itself.

At any step during the downward pass of the hierarchy, it suffices to store the local-field potential for two adjacent levels, since the goal is to compute only leaf-level local-field potentials. The data structure for local-field potentials is as follows:

```
      REAL*8 LOCAL_POT(K, L, M, N)
CMF$LAYOUT LOCAL_POT( : SERIAL, :, :, : )
```

Layout. Given an array declaration with compiler directives that only specifies whether an axis is distributed among VUs or local to a VU, the Connection Machine Run-Time System (CMRTS) as a default attempts to balance subgrid extents and minimize the surface-to-volume ratio. Since communication is minimized for nonadaptive hierarchical methods when the surface-to-volume ratio of the subgrids is minimized, the default layout is ideal.

The extents of the three parallel axes of the potential array, L, M, and N, respectively, are equal to the number of leaf-level boxes along the three spatial dimensions, and hence are powers of 2 for a nonadaptive method. The global address has p bits for $P = 2^p$ VUs and m bits for $M = 2^m$ local addresses. For a multidimensional array, such as LOCAL_POT, the VU address field and the local memory address field are each broken into segments, one segment for each axis, for a block mapping that minimizes the surface-to-volume ratio. Since the first axis is local, it is entirely allocated to local memory. For the parallel axes, both the number of VUs and the number of boxes are powers of 2. Thus, in considering the representation of the array allocation it suffices to consider their address bits. The address fields of the potential array are shown in Figure 7.

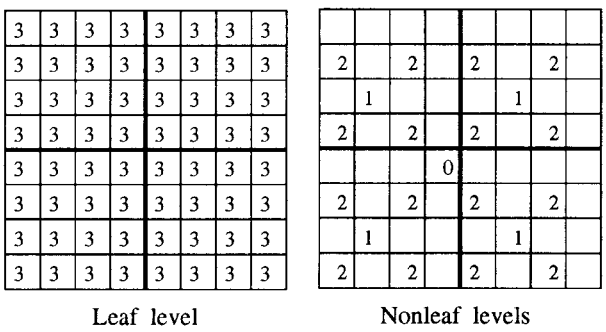


FIGURE 6 Embedding of a hierarchy of grids in two 4-D arrays.

axis	extent	VU address $b_{p+n-1}b_{p+n-2}...b_n$	local memory address $b_{n-1}b_{n-2}...b_0$
0	K		b..b
1	L	b..b	b..b
2	M	b..b	b..b
3	N	b..b	b..b

FIGURE 7 The allocation of the local potential arrays LOCAL_POT to VUs.

Particle Representation

The input to the program consists of a bounding box and relevant particle data. The particle information is given in the form of a collection of 1-D arrays: one array for each particle attribute, such as charge, mass, velocity, and coordinates.

Particle data are used in particle–box interactions in forming the far-field potential for leaf-level boxes before traversing the hierarchy, and in evaluating the local-field potential of leaf-level boxes at the particles inside these boxes after traversing the hierarchy. To maximize the locality in these computations it is desirable to allocate particles to the same VU as the leaf-level box to which they belong. For this reason, we also use 4-D arrays for each particle attribute, with a layout equal to that of LOCAL_POT and FAR_POT. The declaration of the 4-D array for the x-coordinates of the particles is

```
      REAL*8 X_4D(B, L, M, N)
CMF$LAYOUT X_4D( : SERIAL, :, :, : )
```

Note that the particle–particle interactions are defined by the collection of boxes defining the near-field. The direct evaluation can be efficiently performed using the same data structures and layouts as used in computing particle–box interactions, as detailed in Sections 6.2 and 9.

6 OPTIMIZING COMMUNICATION

By using the optimization techniques described in this section, all communication amounts to 10% of the total execution time for a sample run of 100 million particles on a 256-node CM-5E, using $K = 72$ in the field approximations on the spheres (Equations 3 and 4).

The $O(N)$ methods require three kinds of communication:

1. Particle–box: Particle–box interactions are required in forming the leaf-level boxes' far-field representation before the upward traversal of the hierarchy. They are also required in evaluating the local-field at the particles after the downward pass of the hierarchy.
2. Box–box: During the upward pass, the combining of far-field potentials of child boxes to form the far-field potential of the parent box requires parent–child box–box interactions. During the downward pass, converting the local-field potentials for parent boxes to that for child boxes also requires parent–child (box–box) interactions. In addition, the downward pass requires neighbor (box–box) interactions for the conversion of the far-field potential of interactive-field boxes to local-field potentials.
3. Particle–particle: The evaluation of the near-field requires particle–particle interactions among groups of particles contained in the near-field boxes.

We show that by maximizing the locality in allocating child boxes to the same VU as their parent, and by avoiding unnecessary local memory moves through the use of the array-aliasing feature, excessive data movement can be avoided and a high degree of communication efficiency can be achieved on box–box interactions. The efficiency in particle–box interactions for uniform, or almost uniform, particle distributions is achieved by an efficient remapping of the 1-D input arrays for particle attributes to 4-D arrays with the same layout as the leaf-level boxes of the potential arrays. This layout is also used for efficient computation (see Section 9) and communication (see Section 6.2) in the particle–particle interactions.

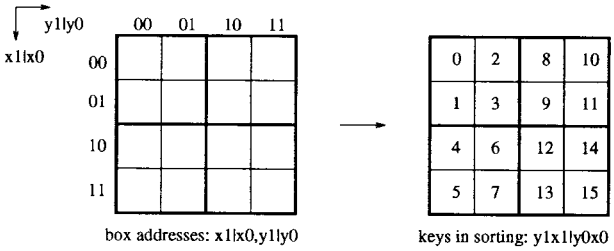


FIGURE 8 Sorting particles for maximum locality in reshaping particle arrays.

6.1 Maximizing the Locality in Particle–Box Interactions

Mapping of 1-D Particle Arrays to 4-D Arrays

The mapping of the 1-D input arrays for particle attributes to 4-D arrays is determined as follows. First, to which box a particle belongs is determined based on its coordinates and the number of boxes along different coordinate axes. Second, the particles in each box are ranked. The rank and the box number give a unique location in the 4-D array. The length of the local axis of this array is equal to the maximum number of particles in any box. The ranking of the particles in each box is made through a segmented +-scan on a 1-D input array after the particles have been sorted such that particles in the same box appear together. We use a coordinate sort (see Fig. 8) for the particle sort. The keys for the coordinate sort are determined so that for a uniform distribution of particles the sorted particles in the 1-D array are allocated to the same VU as the leaf-level boxes (in the potential arrays) to which they belong.

Algorithm: (Coordinate sort)

1. Find the layout of the 4-D potential arrays using intrinsic mapping functions, e.g., the number of bits for the VU address and the local memory address for each axis.
2. For each particle, generate the coordinates of the box to which it belongs, denoted by $xx..x$, $yy..y$, and $zz...z$.
3. Split the box coordinates into VU address and local memory address, written as $x..x|x..x$, $y..y|y..y$, $z..z|z..z$, according to the layout of the potential arrays.
4. Form keys for sorting by concatenating the VU addresses with local memory addresses, written as $z..zy..yx..x|z..zy..yx..x$.
5. Sort.

After sorting, particles belonging to the same box are ordered before any particle in any higher-ordered box. Furthermore, for a uniform particle distribution, if there is at least one box per VU, each particle in the coordinate-sorted 1-D particle array will be allocated to the same VU as the leaf-level box in the 4-D array of local-field potentials to which the particle belongs. Therefore, no communication will be needed in assigning the particles in the 1-D arrays sorted by the coordinate sort to the 4-D arrays with the same layout as the potential arrays. For a near-uniform particle distribution, it is expected that the coordinate sort will leave most particles in the same VU memory as the leaf-level boxes to which they belong.

Particle–Box Interactions

To compute the leaf-level particle–box interactions before traversing the hierarchy, the contributions of all particles in a box to each of the integration points on the sphere corresponding to the box in Anderson's method (or to each coefficient of the multipole expansion for the box in Greengard-Rokhlin's method) must be accumulated. Different boxes have different numbers of particles. Therefore, the number of terms added varies with the leaf-level box. Once the particles are sorted such that all particles belonging to the same box are ordered together, a segmented +-scan is a convenient way of adding in parallel the contributions of all the particles within each of the boxes. The segmented +-scan can be performed on either the sorted 1-D or 4-D arrays after the remapping. On the 4-D array the segmented scans are guaranteed to be local to a VU, and fast. Thus, we perform all scans required for the particle–box interactions on the 4-D arrays.

6.2 Particle–Particle Interactions

The direct evaluation in the near-field can also be carried out very conveniently using the 4-D particle arrays: Each box interacts with its 124 near-field neighbor boxes and each neighbor box–box interaction involves all-to-all interactions between particles in one box and particles in the other. If the symmetry of interaction (Newton's third law) is used, then the total number of interactions per target box is 62. This idea of reducing communication and computation in the direct evaluation in the near-field via exploiting symmetry is shown in a 2-D example in Figure 9. As box 0 traverses boxes 1–4, the interactions between box 0 and each of the four boxes will be computed. The interactions from the four boxes to box 0 are accumulated and communicated along with box 0. Using data-parallel programming, while box 0 traverses boxes 1–4, boxes 5–8 will traverse box 0 and the interactions between them and box 0 will be computed. The interactions from these four boxes to box 0 will be accumulated and stored in box 0. Finally, the two contributions to box 0 will be combined with interactions among particles in box 0. Exploiting symmetry saves almost a factor of 2 in both communication and computation. The idea of exploiting symmetry is similar to the idea used for the linear orrery by Applegate et al. [32]. Here, a linear ordering is imposed on the 62 neighbor boxes in 3-D, which contain partially ordered particles.

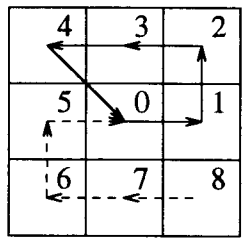


FIGURE 9 Exploiting symmetry in the direct evaluation in the near-field.

6.3 Box–Box Interactions

Excessive data movement can easily happen in programs written in data-parallel languages, such as HPF, which provide a global address space. Below, we show how to avoid excessive data movement in parent–child interactions and in neighbor interactions using the array-aliasing feature of CMF, instead of using extrinsic procedures in HPF which require the low-level, and thus more difficult, message-passing SPMD style programming.

Parent–child box–box interactions are required both in combining far-field potentials in the upward pass through the hierarchy and in local-field evaluations in the downward pass.

Neighbor box–box interactions are required for the far-field evaluation of interactive-field boxes in the downward pass of the hierarchy, and for the direct evaluation of the near-field using the 4-D array representation of the particles. In our implementation of interactive-field computations (which does not exploit the parallelism among the boxes

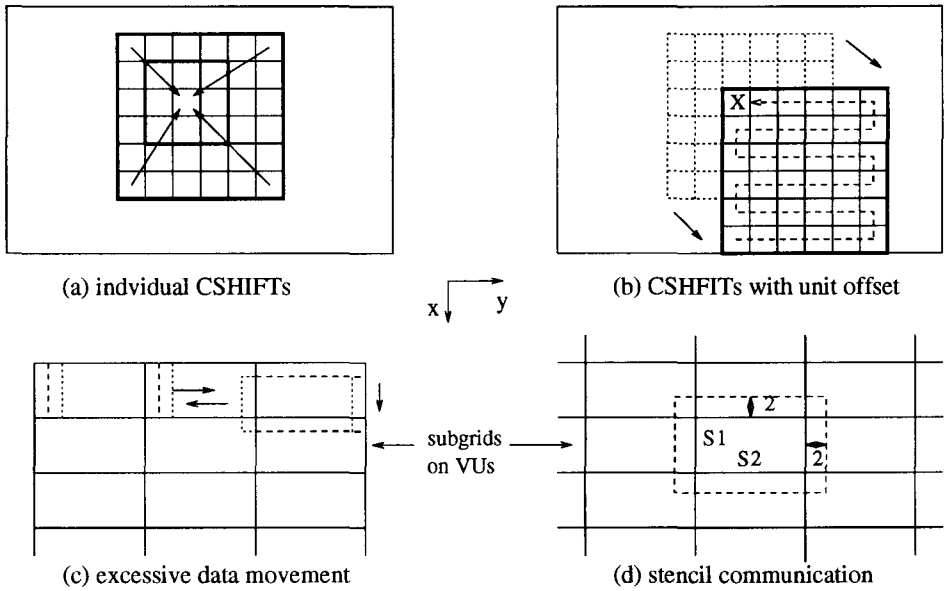


FIGURE 10 Optimizing communication in neighbor interactions. Examples are in two dimensions.

in the interactive-field), each target box needs to fetch the potential vectors of its 875 neighbor boxes (if supernodes [10] are not used).

Interactive-Field Box–Box Communication

In CMF, the simplest way to express the fetching of neighbor potential vectors for a target box uses individual CSHIFTs, one for each neighbor, as shown in Figure 10a. In the CMRTS, composite CSHIFTs are implemented as a sequence of independent shifts, one for each axis.

A better way to structure the CSHIFTs is to impose a linear ordering on the interactive-field boxes, as shown in Figure 10b. The potential vectors of neighbor boxes are shifted through each target box, using a CSHIFT with unit offset at every step. The three axes using different bits in their VU addresses. The rightmost axis uses the lowest-order bits and the leftmost axis uses the highest-order bits in the default axes ordering. Nodes that differ in their lowest-order bits are adjacent in many networks. In such networks, the best linear ordering should use CSHIFTs along the rightmost axis most often. Due to the construction of the fat-tree network on the CM-5/5E and the array layout, this shift order is advantageous in our implementation.

Unfortunately, the scheme just outlined results in excessive data motion. Assume that every VU has an $S1 \times S2$ subgrid of boxes† two dimensions, and that the CSHIFTs are made most often along the y-axis. Every CSHIFT with unit offset involves a physical shift of boundary elements off-VU and a local copying of the remaining elements. After shifting six steps along the y-axis in Figure 10c, the CSHIFT makes a turn and moves along the x-axis in the next step, followed by a sequence of steps along the y-axis in the opposite direction. All the elements in a VU, except the ones in the last row before the turn, are moved back through the same VUs during the steps after making the turn. Thus, this seemingly efficient way of expressing neighbor communication in CMF involves excessive communication in addition to the local data movement. Nevertheless, on a 32-node CM-5E it improved upon the aforementioned alternative by a factor of 7.4 for a subgrid with axes extents 16 and $K = 12$.

In order to eliminate excess data movement, we explicitly identify for all boxes in the local subgrid the interactive-field boxes that are nonlocal, then structure the communication to fetch only those boxes. Figure 11 shows a plane through a target box and its near- and interactive-field boxes in 3-D. For a child box on the boundary of the sub-

† We ignore the local axis in this section since communication only happens on parallel axes.

grid in a VU, the interactive-field box furthest away from it is at distance four along the axis normal to the boundary of the subgrid. Hence, the "ghost" region is four boxes deep on each face of the subgrid. Using the array-aliasing feature of CMF, the ghost boxes can be easily addressed by creating an array alias that separates the VU address from the local memory address. Assume the declaration for the potential array is

```
      REAL*8 POT(K, L, M, N)
CMF$LAYOUT POT( : SERIAL, : , : , : )
```

and that the subgrid of boxes has extents $S1 \times S2 \times S3$. Then, the declarations

```
      REAL*8 POT_ALIAS(K, S1, S2, S3, P1, P2, P3)
CMF$LAYOUT POT_ALIAS( : SERIAL, : SERIAL, : SERIAL, : SERIAL, : , : , : )
      REAL*8 NBR_POT(K, S1+8, S2+8, S3+8, P1, P2, P3)
CMF$LAYOUT NBR_POT( : SERIAL, : SERIAL, : SERIAL, : SERIAL, : , : , : )
```

identify the subgrids and allocate a new array NBR_POT for storing the local subgrid and the ghost boxes in a $(S1 + 8) \times (S2 + 8) \times (S3 + 8)$ subgrid. Alternatively, the ghost boxes can be stored in a separate array. The benefits of using a separate array for the boxes fetched from other VUs are that copying of the local subgrid is avoided and storage is saved by not storing twice the subgrid local to a VU. The drawback with this approach is more complex control in the interactive-field evaluation, and lower arithmetic efficiency because of shorter vectors and fewer instances for each vector operation compared to using a single subgrid. The excess storage for a single array is relatively modest; for a $8 \times 8 \times 8$ subgrid, the ghost region alone contains 3,584 boxes compared to 512 boxes for the local subgrid. On a 256-node CM-5E with 32 Mbyte/VU, the deepest hierarchy for $K = 12$ has depth eight. The largest subgrid has extents $32 \times 32 \times 16$, and the corresponding subgrid for ghost boxes has extents $40 \times 40 \times 24$. In this case, the relative memory waste due to redundant storage of the local subgrid is only 5.3%.

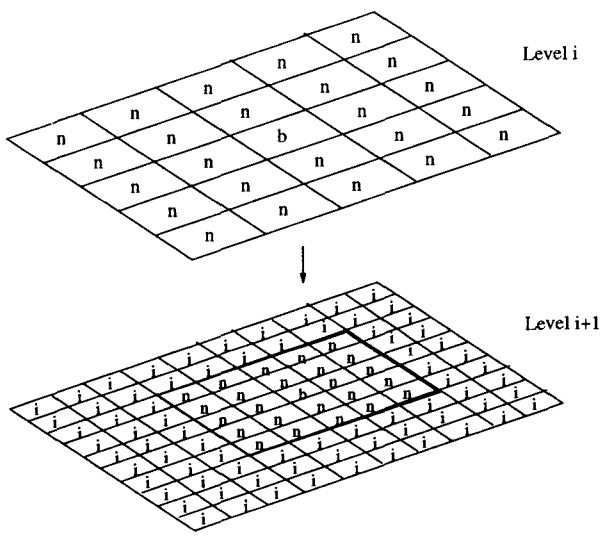


FIGURE 11 A plane of the near- and interactive-fields in three dimensions.

With the subgrids identified explicitly, fetching boxes in ghost regions requires that 6 surface regions, 12 edge regions, and 8 corner regions be fetched in three dimensions. These regions can be fetched either directly, using array sections and CSHIFTs, or by creating a linear ordering through all the VUs containing ghost boxes and using CSHIFTs to move whole subgrids. Array sectioning is performed after subgrids are moved to the destination VU. Moving whole subgrids is necessary in order to keep the continuity of the linear ordering of the subgrids. Although some redundant data motion takes place, it is considerably reduced compared to using a linear ordering on the unaliased array. Table 2 summarizes the data motion requirements for the four methods for $S1 = S2 = S3 = 8$.

The memory requirements can be reduced by prefetching fewer ghost boxes at one time. For example, instead of prefetching all the ghost boxes required by all interactive-field computations, a column of $(S1 + 8) \times S2 \times S3$ ghost boxes can be fetched and used for interactive-field computations with some fixed offsets along the y-axis and the z-axis, but different offsets along the x-axis. As the offset along the y-axis or the z-axis changes by 1, most ghost boxes fetched in the previous step can be reused. However, since in prefetching all the ghost boxes at once, the memory requirement in traversing the hierarchy is about the same as in

Table 2. Comparison of Data Motion Needs for Interactive-Field Evaluation on a 32-Node CM-5E

Method	Number of Nonlocal Boxes Fetched	Number of Local Box Moves	Number of CSHIFTS	Relative Time $K = 12$	Relative Time $K = 72$
Direct on unfactored arrays				169	178
Linearized unfactored arrays	85,888	596,608	1,333	19.4	18.2
Direct on factored arrays	3,584	7,680	54	1.63	1.48
Linearized factored arrays	4,352	6,400	10	1	1

NOTE: The local subgrid is of extent 8 and ghost boxes are stored in a 16 × 16 × 16 subgrid when using factored arrays. The unfactored and factored arrays refer to the original arrays and their aliased counterparts, respectively.

the direct evaluation in the near-field, we did not explore the partial prefetching approach (the maximum storage needs would not be reduced).

Note that for subgrid extents smaller than 4 along any axis, communication beyond nearest-neighbor VUs is required.

Near-Field Box–Box Communication

For the direct evaluation in the near-field, the fetching of near neighbor boxes can also be optimized through factored arrays as described in the previous section, which essentially trades memory requirement for efficient communication. Section 9 discusses another optimization which trades memory requirement for arithmetic efficiency also for the near-field direct evaluation. Either optimization requires similar extra memory, and makes the direct evaluation stage a memory bottleneck, but the increase in performance with the second one is much higher. To save memory, we only keep the second optimization. Thus, fetching particles in neighbor boxes is performed by using CSHIFTs on unfactored arrays with a linear ordering. Note that for the near-field the depth of the ghost region is two boxes in each direction of all axes.

Parent–Child Box–Box interaction

Using the embedding described in Section 5, the far-field potentials of boxes at all levels of the hierarchy are embedded in two layers of a 4-D array, called the base potential array. During traversal of the hierarchy, temporary arrays of a size equal to the number of boxes at the current level of the hierarchy are used in the computation.

We abstract two generic functions `Multigrid-embed` and `Multigrid-extract` for embedding/extracting a temporary array of potential vectors corresponding to some level of the hierarchy into/from the base potential array. The reduction operator used in the upward pass is abstracted as `Multigrid-reduce` operator. The distribution operator used in the downward pass is abstracted as a `Multigrid-distribute` operator. The way to implement these four functions in CMF is to use array sectioning. For example, using the embedding described in Section 5, `Multigrid-embed` at level $(h - i)$ can be expressed in CMF as

```
FAR_POT(2, :, 2**(I-1) : L : 2**I,
2**(I-1) : M : 2**I, 2**(I-1) : N : 2**I)
= TMP.
```

Unfortunately, the current CMF compiler generates a send communication for this expression, even though the corresponding boxes are allocated to the same VU for most levels.

We use `Multigrid-embed` to illustrate how the compiler-generated send can be avoided. If the array `TMP`, which stores the potential vectors for boxes at level i of the hierarchy, has at least one box per VU, `Multigrid-embed` only involves data movement within VUs and no communication is needed. The send is avoided as follows

```
      REAL*8 POT_ALIAS(2,K,S1,S2,S3,P1,P2,P3)
CMF$LAYOUT FAR_POT_ALIAS(:SERIAL,:SERIAL,:SERIAL,:SERIAL,:SERIAL,:,:,:)
      REAL*8 TMP_ALIAS(K,R1,R2,R3,P1,P2,P3)
CMF$LAYOUT TMP_ALIAS(:SERIAL,:SERIAL,:SERIAL,:SERIAL,:,:,:)

FAR_POT_ALIAS(2,:,2**(I-1):S1:2**I,2**(I-1):S2:2**I,2*(I-1):S3:2**I,:,:,:) = TMP_ALIAS
```

In the above code, we first create array aliases for the two arrays to separate their local addresses from the physical addresses. Array sectioning is then performed on the local axes and no send communication is generated.

If array TMP corresponds to a level of the hierarchy which has fewer boxes than the number of VUs, then `Multigrid-embed` is performed in two steps. First, a temporary array TMP2, corresponding to the level of the hierarchy that has the least number of boxes larger than the number of VUs, i.e., at least one box on each VU, is allocated. Then, TMP is embedded into TMP2 using array sectioning, followed by embedding TMP2 into the base potential array via local copying as in the case requiring no communication. The embedding of TMP into TMP2 requires a send communication. But this communication is much more efficient than the communication in embedding TMP directly into the much larger base potential array, although the actual amount of communication is the same. The improved efficiency is due to the smaller overhead in computing send addresses which is about linear in the array size. For array sectioning, the overhead may dominate the actual communication, which is proportional to the number of elements selected.

On the CM-5E, the performance of `Multigrid-embed` is improved by a factor of up to two orders of magnitude using the local copying or the two step-scheme, as shown in Figure 12.

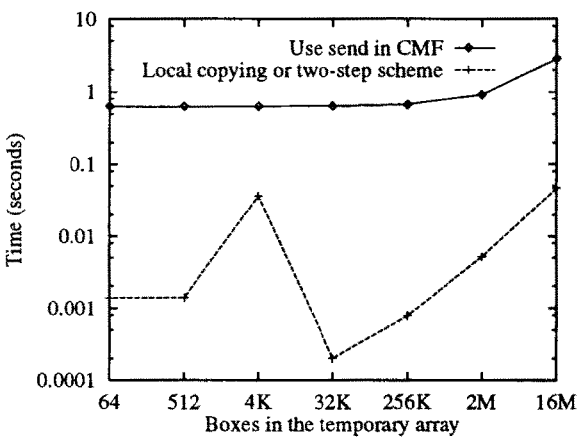


FIGURE 12 Performance improvement of `Multigrid_embed` using array sectioning and aliasing for a depth-eight 3-D hierarchy on a 256-node CM-5E. The two-step scheme was used for the first two cases and the remaining cases used only local copying.

7 OPTIMIZING COMPUTATION

Our techniques for optimizing the computations in hierarchy traversal result in an overall efficiency of 40% for $K = 12$ and a depth-eight hierarchy and 69% for $K = 72$ and a depth-seven hierarchy during the upward and downward hierarchy traversal, excluding communication. The peak arithmetic efficiency at the leaf-level of about 74% for $K = 12$ and 85% for $K = 72$ is degraded due to the following four kinds of overheads: copying, masking, overheads for the higher levels of the hierarchy, and poor vectorization in the direct evaluation in the near-field.

In Anderson's variant of the fast multipole method, each of the three translation operators used in traversing the hierarchy can be aggregated into matrices, and their actions on the potential field further aggregated into multiple-instance matrix–matrix multiplication. Since there is no other computation in the hierarchy, the entire hierarchical part takes the form of a collection of matrix–matrix multiplications, which is implemented efficiently on most computers as part of the basic linear algebra subroutines (BLAS) [33–35]. Multiple-instance BLAS forms a part of the CMSSL [12].

For $K = 12$ and a depth-eight hierarchy on a 256-node CM-5/5E, the use of CMSSL results in an efficiency of 54% and 74% (87 and 119 Mflop/s per node) for the translation operations T_1 (T_3) and T_2 at the leaf-level, respectively. Including the overhead of copying, the translation operation T_2 achieve an efficiency of 60%. Including the overhead of both copying and masking, the efficiency of translation operations T_2 drops to 44%. For $K = 72$ and a depth-seven hierarchy on a 256-node CM-5/5E, the use of CMSSL results in an efficiency of 60% and 85% (96 and 136 Mflop/s per node) for T_1 (T_3) and T_2 at the leaf-level, respectively. The efficiency in translation operations T_2 drops to 79% and 74% when the overhead of copying and the overhead of both copying and masking are included, respectively. The efficiencies are summarized in Table 3.

7.1 Translations as BLAS Operations

The translation operators evaluate the approximations of the source spheres at the integration points of the destination spheres (see Fig. 3). A sphere approximation (Equation 3 or 4) is defined by

$$\Phi(\vec{x_j}) \approx \sum_{i=1}^{K} f(\vec{s_i}, \vec{x_j}) \cdot g(a\vec{s_i}), j = 1, K, \qquad (5)$$

Table 3. Leaf-Level Arithmetic Efficiencies on a 256-Node CM-5E

Operation	$K = 12$, $h = 8$	$K = 72$, $h = 7$
T_1, T_3: arithmetic	54%	60%
T_2: arithmetic	74%	85%
arithmetic incl. copy	60%	79%
arithmetic incl. copy and masking	44%	74%

NOTE: The aggregation of T_2 translations involves copying and masking.

where $f(\vec{s_i}, \vec{x_j})$ represents the inner summation in the original approximation. $f(\vec{s_i}, \vec{x_j})$ is a function of the unit vector $\vec{s_i}$ from the origin of the source sphere to its ith integration point and the vector $\vec{x_j}$ from the origin of the source sphere to the jth integration point on the destination sphere. Due to the construction of the hierarchy of boxes and the approximation formulas used, $f(\vec{s_i}, \vec{x_j})$ is unique to each child of a parent, but is location and level independent. It is preferably precomputed. The translation of the integration points of a child to each integration point of the parent is an innerproduct computation. The translation of the integration points of the child to all of the integration points of the parent constitutes a matrix–vector multiplication, where the matrix is of shape $K \times K$. Thus, Equation 5 indeed defines a matrix–vector multiplication.

Translation Matrices for T_1 and T_3

Since in three dimensions a parent has eight children, each of the translation operators T_1 and T_3 can be represented by eight matrices, one for each of the different parent–child translations. The same matrices can be used for all levels, and for the translations between any parent and its children irrespective of location. In fact, the eight matrices required to represent T_1 are permutations of each other. One matrix can be obtained through suitable row and column permutations of another matrix.

Let the potential vectors of eight child boxes of a parent box be $f_1, \ldots, f_8$, and the translation matrix from one of the child boxes to the parent box be M. In matrix form, the application of T_1 can be written as

$$f = Mf_1 + P_2MP_2'f_2 + \ldots + P_8MP_8'f_8, \quad (6)$$

where f is the potential vector of the parent box, and $P_i, P_i', 2 \le i \le 8$ are suitable permutation matrices.

Similarly, let the potential vector of a parent box be f, and the translation matrix from the parent box to one of the child boxes be M. The application of T_3 to compute the potential vectors of child boxes can be expressed as

$$Mf, P_2MP_2'f, \ldots, P_8MO_8'f. \quad (7)$$

If the permutation property is exploited, it suffices to store one matrix for T_1 and one for T_3 in each VU, since the matrices for T_1 and T_3 are shared by all the boxes at all levels. Equation 6 can be evaluated by first permuting the potential vectors of seven of the eight child boxes, i.e., generating $P_2^Tf_2, P_3^Tf_3$, etc., then performing a matrix–matrix multiplication of the matrix M and a matrix with the eight potential vectors of the children as columns, followed by permutations of the columns of the product matrix which then are added to form f. For T_3, seven different permutations of f are generated first, then a matrix–matrix multiplication is performed as for T_1, followed by permutations of the columns of the product matrix. No reduction is required. This approach reduces the amount of computation and the storage of translation matrices and may achieve better arithmetic efficiency through the aggregated matrix–matrix multiplication. However, on the CM-5E, the time for the permutations exceeds the gain in arithmetic efficiency. In our code we store all eight matrices for each of T_1 and T_3.

Even though permutations are not used in applying the translation operators to the potential field, they could be used in the precomputation phase. Since the permutations depend on K, the number of integration points in a nontrivial fashion, using permutations in the precomputation stage would require storage of the permutations for all different Ks. To conserve memory, we explicitly compute all matrices at run-time (when K is known). We discuss redundant computation-communication trade-offs in Section 8.

Translation Matrices for T_2

The interactive-field computations dominate the hierarchical parts of the code. In three dimensions the interactive-field contains no boxes inside a $5 \times 5 \times 5$ subgrid centered at the target box. Depending on which child box of a parent is the target, the interactive-field extends two or three

boxes at the level of the child box in the positive and negative direction along each axis. Together, the target box and its near-field and interative-field boxes form a $10 \times 10 \times 10$ subgrid. This subgrid is centered at the center of the parent, and is the same for all children of the parent, although the near-field and interactive-fields of siblings differ.

Each box, except boxes sufficiently close to the boundaries, has 875 boxes in its interactive-field. Though each of the eight children of a parent requires 875 matrices, the siblings share many matrices. The interactive-field boxes of the eight siblings have offsets in the range $[-5 + i, 4 + i] \times [-5 + j, 4 + j] \times [-5 + k, 4 + k] \setminus [-2, 2] \times [-2, 2] \times [-2, 2]$, $i, j, k \in \{0, 1\}$, respectively. For illustration, see Figure 11. Each offset corresponds to a different translation matrix. The union of the interactive-fields of the eight siblings has $11 \times 11 \times 11 - 5 \times 5 \times 5 = 1{,}206$ boxes with 1,206 offsets in the range $[-5, 5] \times [-5, 5] \times [-5, 5] \setminus [-2, 2] \times [-2, 2] \times [-2, 2]$. For ease of indexing, we also generate the translation matrices for the 125 subdomains excluded from the interactive-field, or a total of $11 \times 11 \times 11 = 1{,}331$ matrices. Different ways of precomputing the translation matrices and the trade-offs are discussed in detail in Section 8.

7.2 Aggregation of Translations

Aggregation of computations lowers the overheads in computations. In addition, the aggregation of computations may allow for additional optimizations by increasing the degree of freedom in scheduling operations at a given time. The goal in aggregating translations in Anderson's method is to combine lower-level BLAS into higher-level ones, and to aggregate the highest-level BLAS that can be used into multiple-instance calls to the CMSSL BLAS. Aggregation exploits the fact that the translation matrices are the same for the corresponding child of each parent in all parent–child translations. Similarly, aggregation makes use of the fact that the matrices used for the far-field to local-field potential conversion in the interactive-field only depend on the relative locations of the source and destination boxes.

Parent–Child Interactions

Below we show how to use aggregation for an efficient implementation of the translation operator T_1. Assume that at some level of the hierarchy, the subgrid of the temporary potential array is of shape $S1 \times S2 \times S3$, with $S1, S2, S3 \geq 2$, as shown in Figure 13. Each box in the subgrid stores a potential vector and must perform a matrix–vector multiplication. As discussed in Section 7.1.1, only one copy of each translation matrix is stored, and it is shared by all the boxes on each VU. Thus, explicit looping over the boxes on each VU is needed. The loop structure is shown by the pseudo-code fragment

```
DO I=1,2
  DO J=1,2
     DO K=1,2
        DOII=1,S1,2
           DO JJ=1,S2,2
              DO KK=1,S3,2
                 CALL MATRIX-VECTOR-
                 MULTIPLY(...)
...
ENDDO
```

Each of the eight child boxes of a parent needs to use a different translation matrix. The choice of translation matrix (child) is controlled by the outer three DO loops. The inner three loops iterate through every other box along the three axes—the same child box of each parent box. The loop body contains a call to the matrix–vector multiplication subroutine with the matrix of shape $K \times K$. Since the same translation matrix is used in the inner three loops, these loops could in principle be combined into a single matrix–matrix multiplication for one matrix of shape $K \times K$ and the other of shape $K \times S1/2 \cdot S2/2 \cdot S3/2$. However, such combining is possible only if the stride for the axis of length $S1/2 \cdot S2/2 \cdot S3/2$ is constant. This condition does not hold, as shown in Figure 14. The largest number of columns that can be treated with a fixed stride is $\max(S1/2, S2/2, S3/2)$.

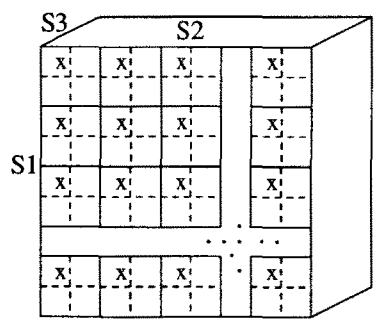


FIGURE 13 The subgrid of boxes of potential arrays on a VU.

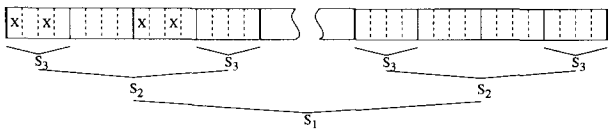


FIGURE 14 The layout of the subgrid of boxes of potentials arrays in each VU.

In aggregating matrix–vector operations into matrix–matrix operations, not only is the number of vectors being aggregated of interest, but also the stride between successive vectors, since it affects the number of DRAM page faults and TLB entry replacements in the multiple-instance matrix–matrix multiplication. With cubic or close to cubic subgrids for minimum communication, either the extents of the subgrid axes are the same or they differ by a factor of 2. For relatively small subgrids, the difference in size of the multiplicand due to the difference in subgrid axes extents has a larger impact on performance than DRAM page faults and TLB thrashing. Hence, we choose to aggregate vectors into a matrix along the axis with the largest local extent. If two or all three axes are of the same length, the vectors are aggregated along the axis with the largest local extent and with the smallest stride. For relatively large subgrids, vectors are aggregated along the axis with the smallest stride.

The remaining two loops of the three innermost loops define multiple-instance matrix–matrix multiplication, which is supported by CMSSL. The CMSSL routines fold all axes that can be folded into a single axis with constant stride for the multiple-instance computations. All such folded instance-axes are considered together with the problem-axes in determining blocking and loop orders for maximum performance.

Since the two instance-axes in Anderson's method cannot be folded into a single axis with constant stride due to the array layout (see Fig. 14), the aggregation of the matrix–vector multiplications into multiple-instance matrix–matrix multiplication is implemented as

```
DO I=1,2
  DO J=1,2
     DO K=1,2
        DO II=1,S1,2
           CALL MATRIX-MULTIPLY-MI(...)
...
ENDDO
```

The performance of the T_1 and T_3 translations improves from 58 to 87 Mflop/s/PN for $K = 12$ and subgrid of extents $32 \times 32 \times 16$ by replacing the first loop structure with the loop structure above. The matrices are of shape 12×12 and 12×8 with 16 such instances handled in a single call. For $K = 72$ and a subgrid of extents $16 \times 16 \times 8$, the performance improves from 95 to 96 Mflop/s/PN.

Far-Field to Local-Field Conversion

The conversion of the far-field to local-field potential of the boxes in the interactive-field is made using the array NBR_POT with subgrid of shape $(S1 + 8) \times (S2 + 8) \times (S3 + 8)$. For each of the eight sibling boxes of a parent box, 875 applications of the translation matrix for T_2 are required. We use three nested loops with a total of 1,000 iterations to accomplish the 875 matrix–vector multiplications; the 125 undesired iterations are skipped by a conditional test. In the loop nest below, all operations on the subgrid for a given translation matrix are performed before any operation for any other translation matrix.

```
DO I=0,1
  DO J=0,1
     DO K=0,1
        DOI1=-4,-I,5-I
           DO J1=-4-J,5-J
              DO K1=-4-K,5-K
                 DO II=1,S1,2
                    DO JJ=1,S2,2
                       DO KK=1,S3,2
                          if ((I1,J1,K1) in interactive-field)
                             CALL MATRIX-VECTOR(...)
...
ENDDO
```

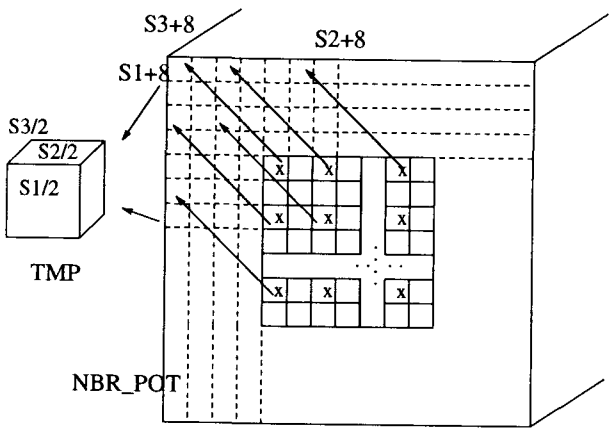


FIGURE 15 Aggregation of BLAS operations in neighbor interactions.

For parent–child interactions, aggregation of matrix–vector multiplications into multiple-instance matrix–matrix multiplications was carried out on two of the three innermost loops. For interactive-field computations we choose to copy the vectors of the array $(S1 + 8) \times (S2 + 8) \times (S3 + 8)$ referenced by the three innermost loops into a new array such that a single-instance matrix–matrix multiplication with matrices of shape $K \times K$ and $K \times S1/2 \cdot S2/2 \cdot S3/2$ is performed. The copying is illustrated in Figure 15. The loop structure for the approach using copying is shown in the following code fragment:

```
DO I=1,2
   DO J=0,1
      DO K=0,1
         DOI1=-4-I,5-I
            DO J1=-4-J,5-J
               DO K1=-4-K,5-K
                  TMP =
                  NBR_POT(:,I1+I+5:I1+I+S1+4:2,J1+J+5:J1+J+S2+4:2,K1+K+5:K1+K+S3+4:2,:,:,:)
                  CALL MATRIX-MULTIPLY(...)
...
ENDDO
```

For $S1 = 32$, $S2 = 32$, $S3 = 16$, and $K = 12$, the execution rate of the 12×12 by $12 \times 2{,}048$ matrix multiplication is 119 Mflop/s/PN. If there are no DRAM page faults, the copying requires $2K$ cycles for a potential vector for which the matrix multiplication ideally takes K^2 cycles. Thus, the relative time for copying is $2/K$. This amounts to about 17% for $K = 12$ and less than 4% for $K = 72$. With the cost of copying included, the measured performance of the translation is 85 Mflop/s/PN. For $S1 = 16$, $S2 = 16$, $S3 = 8$, and $K = 72$, the execution rate of the 12×12 by 12×256 matrix multiplication is 136 Mflop/s/PN. Including the cost of copying, the measured performance is 124 Mflop/s/PN.

The copying cost can be reduced by copying a whole column block of $(S1 + 8) \times S2/2 \times S3/2$ boxes into two linear memory blocks outside the DO-K1 loop; one for even slices of the column, and the other for odd slices. Since the axis indexed by K1 has unit stride, a sectioning with stride 2 on that axis will reside in a consecutive block of memory in one of the two temporary arrays. Each local column can be used on average 8.75 times in the DO-K1 loop. The cost of copying is therefore reduced to $4 \cdot 100/875 \cdot (S1 + 8)/(S1 \cdot K)$ of that of matrix multiplication, assuming no page faults. Including the cost of copying, the performance of translations in neighbor interactions reaches 96 and 127 Mflop/s/PN for $K = 12$ and $K = 72$, respectively.

Copying of sections of subgrids to allow for a $K \times K$ by a $K \times S1/2 \cdot S2/2 \cdot S3/2$ matrix multiplication can also be used in parent–child interactions, but the copying cost is relatively higher. In estimating the copying cost for the interactive-field computations, we ignored the small copying back cost after the accumulation. With this cost included, the total copying cost for operator T_2 is $2K + 2K/875$ cycles per matrix–vector multiplication. For parent–child interactions the total copying cost is $2K + 2K/8$ cycles, ideally, per $K \times K$ matrix–vector multiplication. The copying cost for parent–child interactions is about 10% higher than that for the interactive-field computations. By using copying in parent–child interactions, the performance for the T_1 and T_3 matrix operations drops from 87 to 82 Mflop/s/PN for $K = 12$, but increases from 96 to 123 Mflop/s/PN for $K = 72$, due to the lower cost of copying relative to that of

matrix multiplication in the latter case. Since the time for parent–child interactions accounts for only a couple of percent of the total time for the hierarchy traversal, we did not carry out the K-dependent optimization of copying in parent–child interactions.

7.3 Uniformity—Avoiding Masking

In the calls to the BLAS in the above loop-nest for neighbor interactions, masking is needed since boundary boxes have smaller interactive-fields than interior boxes. Masking is needed at all levels of the hierarchy. In CMF, the masking is handled as an unconditional matrix multiplication followed by a masked assignment, and the masked assignment is noticeably slower than an unmasked assignment.

The masking can be avoided by adding two layers of empty boxes on all sides of the domain. We evaluate this option for the leaf-level. With h levels, each axis of the physical domain is extended by a factor of $2^h/(2^h - 4)$ to create two empty boxes at each side of each axis of the domain. Using empty boxes increases the cost for the direct evaluation in the near-field. For a given hierarchy depth, using empty boxes to avoid masking at the leaf-level requires putting all the particles in the inner $(2^h - 4) \times (2^h - 4) \times (2^h - 4)$ boxes. The maximum number of particles per box increases by a factor of $\beta = [N/(2^h - 4)^3]/[N/8^h]$, and the cost of direct evaluation is increased in proportion to β^2. For a uniform distribution and $K = 12$, there are 4–16 particles per leaf-level box for the optimal hierarchy depth. The increase in the cost of the direct evaluation for these cases is shown in Table 4.

In our implementation on the CM-5E, the cost of masking at the leaf-level of a depth-eight hierarchy is about 18% of the total cost of traversing the hierarchy for $K = 12$ and depth eight. As K increases, the cost of matrix multiplication increases as K^2 and the cost of masking grows as K. Thus, the cost of masking becomes less significant. For example, for $K = 72$ and depth seven, the cost of masking is less than 4% of the cost of traversing the hierarchy. The increase in the direct evaluation, on the other hand, decreases slowly.

We conclude that on the CM-5E, by adding ghost boxes at the leaf-level of the hierarchy, the gain in avoiding masking in traversing the hierarchy is not large enough to offset the loss in the direct evaluation when using optimal hierarchy depth. Obviously, adding ghost boxes to higher levels implies adding more ghost boxes to the leaf-level and will increase the cost of the direct evaluation further. On other machines and with different compilers, the relative cost of masked assignments will most likely be different and the technique discussed here must be reevaluated.

8 REDUNDANT COMPUTATION VERSUS REPLICATION

All translation matrices are precomputed. Since the translation matrices are shared by all boxes at all levels, only one copy of each matrix is needed on each VU. Two extreme ways of computing these translation matrices are:

1. Compute all the translation matrices on every VU.
2. Compute each translation matrix only once with different VUs computing different matrices, followed by a spread to all other VUs as a matrix is needed.

In the first method the computations are embarrassingly parallel and no communication is needed. However, redundant computations are performed. In the other method there is no redundant computation, but replication is required. If there are fewer matrices to be computed than there are VUs, then VUs can be partitioned into groups with as many VUs in a group as there are matrices to be computed. Each group computes the entire collection of matrices, followed by spreads within groups when a matrix is needed. The replication may also be performed as an all-to-all broadcast [36]. The load-balance with this amount of redun-

Table 4. The Increase in the Direct Evaluation Cost at Optimal Hierarchy Depth Using Ghost Boxes

Particles/Box at Optimal Depth without Ghost Boxes	$K = 12$			$K = 72$		
	4	8	16	16	32	64
β^2 (h = 7)	1.56	1.27	1.27	1.27	1.27	1.23
β^2 (h = 8)	1.56	1.27	1.13	1.13	1.13	1.13

dant computation is the same as with no redundancy, but the communication cost may be reduced.

Let the cost of computing a translation matrix on a VU be t_1 and the cost of replicating it across P VUs be $t_2(P)$. The total cost for the above two extreme ways of computing N matrices on P VUs with each VU storing all N matrices is

$$T_1(N, P) = N \cdot t_1$$

$$T_2(N, P) = \left\lceil \frac{N}{P} \right\rceil \cdot t_1 + N \cdot t_2(P)$$

Here we assume that t_1 is independent of the number of matrices being computed on a VU, though in practice computations often are more efficient when more matrices are computed on each VU, because of more efficient vectorization. On the CM-5E, for K varying from 12 to 72, replicating a $K \times K$ translation matrix is about 3–12 times faster than computing it. Thus, computing the matrices in parallel followed by replication is always a winning choice.

For T_1 and T_3 we also implemented grouping computations and replication among eight VUs in addition to the two extreme methods. Figure 16 shows the performance of the three methods. The cost of computing the matrices in parallel followed by replication without grouping is 66–24% of that of computing all matrices on each VU, as K varies from 12 to 72. With grouping, the computation cost is the same as without grouping, but the cost of replication is reduced by a factor of 1.75–1.26 as K varies from 12 to 72. The reason for the decrease of the difference as K increases is that for larger K, the replication time is dominated by bandwidth, while for small K, latency and overhead dominate.

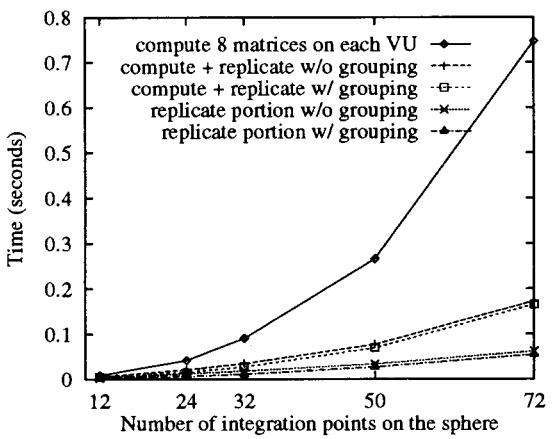


FIGURE 16 Computation versus replication in precomputing translation for $T_1(T_3)$ on a 256-node CM-5E.

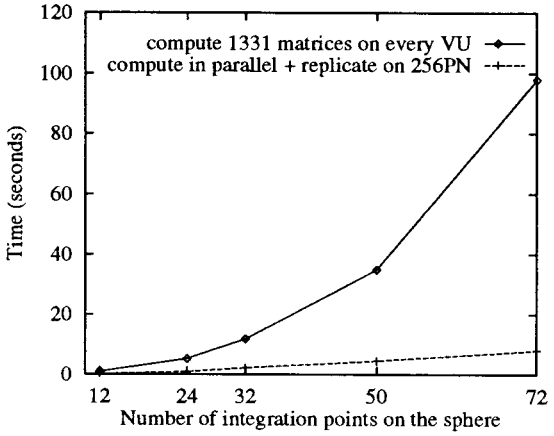


FIGURE 17 Computation versus replication in precomputing translation matrices for T_2 on the CM-5E.

For T_2, computing one copy of each of the 1,331 translation matrices and replicating them is up to an order of magnitude faster than computing all on every VU, as shown in Figure 17 for a 256-node CM-5E. The time for computing 1,331 matrices in parallel decreases on larger CM-5Es, as shown in Figure 18, while the replication time, which dominates the total time, increases about 10–20% for large K as the number of nodes doubles. As a result, the total time for the method increases at most 62% as the number of nodes changes from 32 to 512.

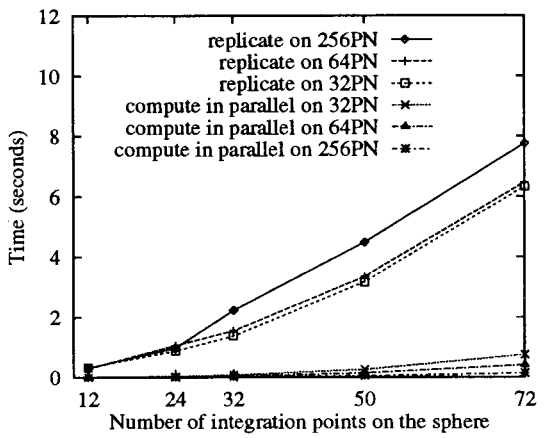


FIGURE 18 Compute in parallel and replicate in precomputing translation matrices for T_2 on various sized CM-5Es.

Storing all 1,331 translation matrices in double precision on each VU requires $(1{,}331 \cdot 8 \cdot K^2)$ bytes of memory, i.e., 1.53 MB for $K = 12$ and 53.9 MB for $K = 12$. Therefore, replication of a matrix is delayed until it is needed. The replication is made through one-to-all broadcast rather than all-to-all broadcast. The total number of replications is $1{,}331 \cdot (h - 1)$, where h is the depth of the hierarchy, since the T_2 translations are used first at level two.

9 DIRECT EVALUATION IN THE NEAR-FIELD

Since the optimal hierarchy depth that minimizes the total FLOP count of an $O(N)N$-body method balances the amount of computation in the hierarchy traversal and in the direct evaluation in the near-field, the efficiency in the direct evaluation is crucial to the overall performance. In this section, we discuss how to use the 4-D arrays of particle attributes, used for efficient particle–box interactions, for efficient evaluation of the near-field potentials.

The near-field is evaluated as a sequence of particle–particle interactions ordered with respect to the boxes to which they belong. The 124 neighbor boxes of a target box can be ordered linearly and brought to the target box through 124 single-step CSHIFTs. Another way is to fetch nonlocal near-field boxes from other VUs using 4-D arrays factored into local subgrids through array aliasing, much in the same way as in fetching nonlocal interactive-field boxes. The first method requires less temporary storage, and is used for the near-field evaluations. The CSHIFTs account for about 10–15% of the time for the direct evaluation.

Once a neighbor box has been brought to the target box, an all-to-all interaction between the particles in the two boxes is required. We investigated three alternatives for the all-to-all interaction. The simplest way is to loop through the particles in both boxes using two nested loops. Unrolling the inner loop can improve the performance of a compiler-generated code by 25% on the CM-5E. The vectorization can be further improved by replicating each particle in the neighbor box to every particle in the target box, followed by element-wise particle interactions. But the broadcast operation for each neighbor particle is relatively time consuming. A third approach, called "duplicate-and-slide," duplicates the target box, i.e., a new 4-D array with a local axis of twice the length of the original array is created. The original 4-D particle array is copied to both the first and the second half of the new array. One sequential loop over the particles in the neighbor box is used. Let b be the length of the serial axis of the original 4-D particle array, i.e., the maximal number of particles per leaf-level box. At the ith iteration, an element-wise interaction between the neighbor box and a b-long segment along the local axis of the new array starting at the ith element is evaluated. It is easy to see that the looping covers all particle interactions between the two boxes. The duplicate-and-slide approach duplicates particles once and the computations inside the loop are perfectly vectorized on each VU. On the CM-5E it is the fastest of all three approaches. However, it requires 33% more memory than the other alternatives, or a total of $4N$ memory locations for each particle attribute; N locations for the input 1-D array, $2N$ locations for the 4-D duplicated target, and N locations for the 4-D neighbor.

10 PERFORMANCE RESULTS

Our CMF implementation of Anderson's method with $K = 12$ integration points on the sphere performs the potential evaluation for 100 million particles uniformly distributed in a cubic region in 180 s on a 256-node CM-5E. The evaluation for a system of 100 million uniformly distributed particles is estimated to take around 60 s on a 1,024-node CM-5E. The overall efficiency is about 27%, and is fairly independent of machine size. With $K = 72$ integration points on the sphere, the efficiency improves to 35%. We first give a summary of the timings breakdown in computing the potential field for 100 million uniformly distributed particles on a 256-node CM-5E, then demonstrate the scalability of the implementation. A more detailed analysis of the effectiveness of the techniques is given in [37].

In considering the execution times, it should be mentioned that our implementation uses the idea of supernodes. Zhao [10] made the observation that of the 875 boxes in the interactive-field, in many cases all eight siblings of a parent are included in the interactive-field. By converting the far-field of the parent box instead of the far-fields of all eight siblings, the number of far-field to local-field conversions is reduced to 189 from 875. The supernode idea must be modified somewhat for Anderson's method, but the same reduction in computational complexity can be achieved [37].

Table 5. Weights for Floating-Point Operations in Our Three Methods for FLOP Counts

Method	FLOP Count
I Native	Always 1
II Hennessy and Patterson [38]	ADD, SUB, MULT – 1 DIV, SQRT – 4
III CM-5E/VU normalized	ADD, SUB, MULT – 1 DIV – 5 SQRT – 8

For gravitational and Coulombic fields, division and square roots represent a significant fraction of the arithmetic time. We report floating-point rates for three different weights of these operations as specified in Table 5.

The timings breakdown for the potential field calculation of 100 million particles on a 256-node CM-5E is shown in Table 6 for $K = 12$ and $K = 72$. The hierarchy depths are 8 and 7, respectively. The predicted optimal hierarchy depths based only on the number of floating-point (FLOP) operations using Method III are 7.97 and 7.10. Thus, for $K = 12$, the FLOP counts for the hierarchy and for the direct evaluation are very balanced. In fact, they differ by about 10%. Furthermore, the FLOP rates for $K = 12$ using Method III are 55.6 and 46.8 Mflop/s/PN, respectively. The overall FLOP rate is 43.7 Mflop/s/PN, with sorting accounting for most of the degradation in the overall FLOP rate. For $K = 72$, the FLOP rates for traversing the hierarchy, the direct evaluation, and overall are 81.8, 52.9, and 56.6 Mflop/s/PN, respectively.

The communication time for $K = 12$ is 22.3% of the total running time and 10% for $K = 72$, demonstrating that our techniques for reducing and managing data motion are very effective. The communication time includes the time for sorting the input particles, reshaping 1-D particle arrays to 4-D particle arrays, the multigrid functions in parent–child and neighbor interactions, the fetching of ghost boxes in neighbor interactions at all levels, replicating translation matrices for T_2 at ev-

Table 6. The Breakdown of the Communication and Computation Time for 100 Million Particles on a 256-Node CM-5E

	$K = 12$		$K = 72$	
Breakdown	Time (s)	% of Total	Time (s)	% of Total
Communication	39.75	22.3	89.01	9.99
Sort	19.60	11.0	16.04	1.80
Reshape	2.618	1.47	2.482	0.28
Upward pass – multigrid in T_1	0.107	0.06	0.092	0.01
Downward pass	8.410	4.71	56.39	6.33
Multigrid in T_3	0.215	0.12	0.162	0.02
Multigrid in T_2	0.484	0.27	0.385	0.04
Fetching ghost boxes in T_2	5.160	2.89	8.610	0.97
Replicate (T_2)	2.550	1.43	47.23	5.30
Near-field CSHIFTs	9.013	5.05	14.01	1.57
Computation	138.6	77.7	802.2	90.01
Precompute T_1 matrices	0.006	0.00	0.575	0.06
Precompute T_3 matrices	0.005	0.00	0.572	0.06
Precompute T_2 matrices	0.003	0.00	0.235	0.03
Init-potential	2.506	1.40	14.01	1.57
Upward pass-BLAS for T_1	0.783	0.44	3.459	0.39
Downward pass	63.62	35.7	166.5	18.7
BLAS for T_3	0.601	0.34	4.320	0.48
BLAS for T_2	34.98	19.6	141.6	15.9
Copy in T_2	12.90	7.23	9.990	1.12
Masking in T_2	15.14	8.49	10.53	1.18
Far-field	4.678	2.62	90.74	10.2
Near-field-direct evaluation	65.63	36.8	525.2	58.9
Near-field-masking	1.371	0.77	0.952	0.11
Total	178.4	100	891.2	100

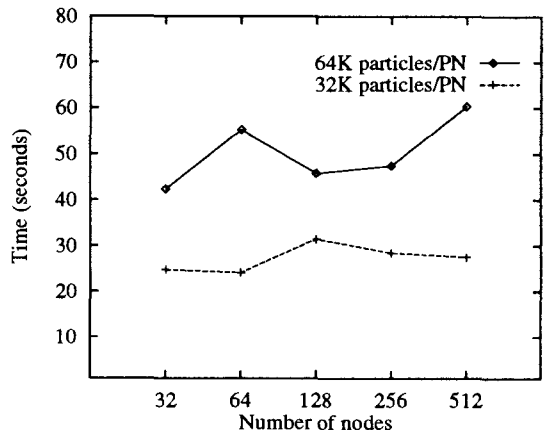


FIGURE 19 Scalarability on the CM-5s.

ery level, and the CSHIFTs in the near-field direct evaluation for fetching particles in the near-field boxes.

The computation time is 77.7% of the total running time for $K = 12$ and 90% for $K = 72$. In the computation time we include the time for forming the far-field potential for leaf-level boxes, the BLAS operations for the T_1, T_2, and T_3 translations, the copying in the aggregation of BLAS operations for better arithmetic efficiency in T_2, the masking in distinguishing boundary boxes from interior boxes in T_2, the evaluation of the potential due to particles in the far-field, and finally the direct evaluation in the near-field.

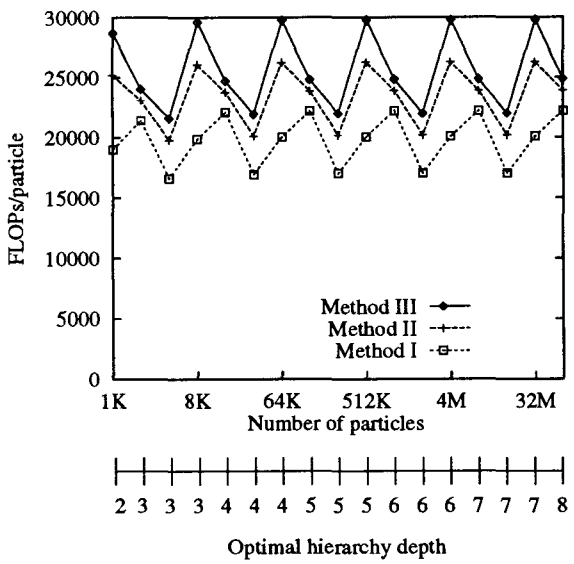


FIGURE 20 FLOP count per particle for optimal hierarchy depth, $K = 12$.

Figure 19 shows that the speed of our code scales linearly with the number of nodes and number of particles. The timings are collected on CM-5s due to the unavailability of a variety of configurations of CM-5E systems. All cases use uniform particle distribution in a 3-D cubic domain, 12 integration points per sphere, and optimal hierarchy depths. It is clear from Figure 10 that for a fixed number of particles per node, the efficiency remains independent of the number of nodes. The slight fluctuation is mainly due to the fluctuation in the number of FLOPs per particle for the optimal hierarchy depth, as shown in Figure 20.

11 DISCUSSION

Nonadaptive hierarchical methods exhibit abundant data parallelism. We have demonstrated that exploiting parallelism within each level of the hierarchy can yield high efficiency (and good load–balance). Below, we also discuss the use of a nonadaptive code for near-uniform particle distributions. For highly nonuniform particle distributions such as in typical simulations in astrophysics, an adaptive hierarchical method is needed in order to achieve good performance.

11.1 Load–Balancing Issues in Nonadaptive Hierarchical Methods

Nonadaptive hierarchical methods use nonadaptive domain decomposition, and the hierarchy of recursively decomposed domains is balanced. There are three sources of parallelism in traversing the hierarchy. First, the computations in parent–child interactions for all boxes at the same level can be performed in parallel in the upward pass and the downward pass of the hierarchy. Second, at every level of the downward pass of the hierarchy, the conversion of the far-field potential of each box's interactive-field boxes into the local-field potential of that box can be performed in parallel. Third, since neighbor interactions are between boxes at the same level, the neighbor interactions at all levels can be performed in parallel.

We only exploit parallelism among boxes at the same level of the hierarchy, which potentially could result in poor load–balance due to the limited parallelism at levels close to the root. However, hierarchical methods are advantageous compared to direct methods only when more than a few thousand particles are considered. Since for the optimal depth of the hierarchy there only are a few particles

per leaf-level box, the number of leaf-level boxes is at least about 100. For large-scale simulations, there may be several million leaf-level boxes. Hence, for most interesting simulations there is excess parallelism even for the largest of MPPs, not only at the leaf-level but also for several levels close to the leaf-level. At levels of the hierarchy close to the root, there are much fewer boxes per level, and the cost of computation is already insignificant. Hence, though the load may be unbalanced, improved load-balance will not affect the total execution time significantly. Also, a program that traverses the hierarchy level by level and sequentializes neighbor interactions requires much simpler data and control structures than one that exploits parallelism beyond just among boxes.

11.2 Nonadapative Hierarchical Methods with Near-Uniform Distributions

The simulations described in this article use uniform particle distribution as input data. In practice, nonuniform distributions are much more important. For simulations where the particle distributions are near uniform, for example in computational chemistry, a nonadaptive code may still outperform an adaptive one. A nonadaptive code may yield a more efficient implementation due to its simpler computational structure compared to an adaptive code. A nonadaptive code performs excess computations:

1. In traversing the hierarchy, excess computations are performed since the same amount of computation is carried out regardless of the number of particles a box represents.
2. In the particle-box interactions at the leaf-level, the reshaping of the 1-D particle arrays into 4-D arrays following the coordinate sort can result in extensive communication and load-imbalance because of the uneven number of particles per box. Furthermore, the memory utilization may be poor, since some boxes may contain far fewer particles than the others, but all of them occupy the same amount of memory on each node. In contrast, using 1-D arrays for particle–box interactions results in more balanced computations, since 1-D arrays will always be laid out across the processing nodes evenly. This also implies balanced memory usage. However, such an approach relies highly on efficient scan operations.
3. In the direct evaluation in the near-field, using the 4-D array representation for particles obviously leads to excess computations because the particle–particle interactions are turned into uniform computations on the boxes, but boxes may contain different number of particles. Using 1-D array representation again relies highly on efficient scan operations.

As part of the future work, we plan to investigate the impact of the uniformity of particle distributions on the efficiency of nonadaptive data structures used in this article.

12 CONCLUSIONS

We have presented optimization techniques for programming $O(N)$ N-body algorithms for MPPs and have shown how the techniques can be expressed in data-parallel languages, such as CMF and HPF. The optimizations mainly focus on minimizing the data movement through careful management of the data distribution and the data references and on improving arithmetic efficiency through aggregating translation operations into high-level BLAS operations. The most performance critical language features are the FORALL statement, array sectioning, array aliasing, CSHIFT, SPREAD, and array inquiry intrinsics. All these features, except array aliasing, are included in HPF. But, this feature is considered for inclusion in HPF-II.

The effectiveness of our techniques is demonstrated on an implementation in the CMF of Anderson's hierarchical $O(N)$ N-body method. The evaluation of the potential field of 100 million uniformly distributed particles and $K = 12$ integration points on the sphere takes 180 s on a 256-node CM-5E, with an efficiency of about 27% of the peak performance. For $K = 72$ integration points, the efficiency is about 35%. The amount of memory required for a particle at optimal hierarchy depth is about 230 bytes, independent of the error rate of the method.

For highly clustered particle distributions, an adaptive version of N-body methods is needed in order to retain $O(N)$ arithmetic complexity. We are currently investigating issues in an efficiency implementation of adaptive $O(N)$ algorithms in HPF.

ACKNOWLEDGMENTS

We thank Christopher Anderson for providing us with the sequential program and for many helpful discussions. We also thank the National Center for Supercomputer Applications at the University of Illinois at Urbana/Champaign, the Navy Research Laboratory, Washington, DC, and the Massachusetts Institute of Technology for providing Connection Machine system CM-5/5E access. The support of the Office of Naval Research through grant N00014-93-1-0192 and the Air Force Office of Scientific Research through grant F49620-93-1-0480 is gratefully acknowledged. Finally, we thank the anonymous referees whose detailed comments improved the quality of the presentation.

REFERENCES

[1] Thinking Machines Corporation, *CM Fortran Reference Manual, Version 2.1*. Thinking Machines Corp., 1993.

[2] High Performance Fortran Forum, "High Performance Fortran language specification," *Sci. Prog.*, vol. 2, pp. 1–170, 1993.

[3] J. M. Anderson and M. S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines," in *Proc. ACM SIGPLAN '93 Conference on Programming Languages Design and Implementation*, 1993.

[4] J. Li and M. Chen, "Generating explicit communication from shared-memory program references," in *Proc. Supercomputing '90*, 1990.

[5] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proc. ACM SIGPLAN '91 Conference on Programming Languages Design and Implementation*, 1991.

[6] Thinking Machines Corporation, *CM-5 Technical Summary*. Thinking Machines Corp., 1991.

[7] C. R. Anderson, "An implementation of the fast multipole method without multipoles," *SIAM J. Sci. Stat. Comp.*, Vol. 13, pp. 923–947, July 1992.

[8] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *J. Comput. Physics*, Vol. 73, pp. 325–348, 1987.

[9] L. Greengard and W. D. Gropp, "A parallel version of the fast multipole method," in *Parallel Processing for Scientific Computing*, SIAM, 1989, pp. 213–222.

[10] F. Zhao, "An O(N) algorithm for three-dimensional N-body simulations," AI Memo 995, AI Lab, MIT, Oct. 1987.

[11] K. Nabors and J. White, "Fastcap: A multipole accelerated 3-d capacitance extraction program," Boston, MA, MIT Department of Electrical Engineering and Computer Science, Tech. Rep., 1991.

[12] Thinking Machines Corporation, *CMSSL for CM Fortran, Version 3.1*. Thinking Machines Corp., 1993.

[13] R. G. Brickner, personal communication.

[14] J. K. Salmon, "Parallel hierarchical N-body methods," California Institute of Technology, Tech. Rep. CRPC-90-14, 1990.

[15] M. Warren and J. Salmon, "Astrophysical N-body simulations using hierarchical tree data structures," in *Proc. Supercomputing '92*, 1992.

[16] M. Warren and J. Salmon, "A parallel hashed oct-tree N-body algorithm," in *Proc. Supercomputing '93*, 1993.

[17] P. Liu and S. N. Bhatt, "Experiences with parallel N-body simulation," in *Proc. 6th Annual ACM Symposium on Parallel Algorithms and Architecture*, 1994.

[18] P. Liu, "The parallel implementation of N-body algorithms," PhD thesis, Yale University, 1994.

[19] F. Zhao and S. L. Johnsson, "The parallel multipole method on the Connection Machine," *SIAM J. Stat. Sci. Comp.*, Vol. 12, pp. 1420–1437, 1991.

[20] J. F. Leathrum, "The parallel fast multipole method in three dimensions," PhD thesis, Duke University, 1992.

[21] J. A. Board, Jr., Z. S. Hakura, W. D. Elliott, D. C. Gray, W. J. Blanke, and J. F. Leathrum Jr., "Scalable implementations of multipole-accelerated algorithms for molecular dynamics," in *Proc. Scalable High Performance Computing Conf. SHPCC94*, 1994.

[22] W. D. Elliott and J. A. Board, "Fast Fourier transform accelerated fast multipole algorithm," Department of Electrical Engineering, Duke University, Tech. Rep. 94-001, 1994.

[23] L. Greengard and V. Rokhlin, "On the efficient implementation of the fast multipole method," Department of Computer Science, Yale University, New Haven, CT, Tech. Rep. YALEU/DCS/RR-602, Feb. 1988.

[24] K. E. Schmidt and M. A. Lee, "Implementing the fast multipole method in three dimensions," *J. Stat. Phy.*, Vol. 63, pp. 1223–1235, 1991.

[25] J. P. Singh, C. Holt, T. Ttsuka, A. Gupta, and J. L. Hennessey, "Load balancing and data locality in hierarchical N-body methods," Stanford University, Tech. Rep. CSL-TR-92-505, 1992.

[26] J. P. Singh, C. Holt, J. L. Hennessey, and A. Gupta, "A parallel adaptive fast multipole method," in *Proc. Supercomputing '93*, pp. 54–65.

[27] L. S. Nyland, J. F. Prins, and J. H. Reif, "A data-parallel implementation of the adaptive fast multipole algorithm," in *Proc. DAGS '93 Symposium*, 1993.

[28] J. Barnes and P. Hut, "A hierarchical $O(N \log N)$ force calculation algorithm," *Nature*, Vol. 324, pp. 446–449, 1986.

[29] L. Greengard and V. Rokhlin, "Rapid evaluation of potential fields in three dimensions," Department of Computer Science, Yale University, New

Haven, CT, Tech. Rep. YALEU/DCS/RR-515, Feb. 1987.

[30] Y. Hu and S. L. Johnsson, "On the error in Anderson's fast N-body algorithm," Harvard University, Division of Applied Sciences, Tech. Rep., 1995.

[31] J. Katzenelson, "Computational structure of the N-body problem," *SIAM J. Sci. Statist. Comput.*, Vol. 4, pp. 787–815, 1989.

[32] J. H. Applegate, M. R. Douglas, Y. Gursel, P. Hunter, C. L. Seitz, and G. J. Sussman, "A digital orrery," *IEEE Trans. Comput.*, Vol. C-34, pp. 822–832, Sept. 1985.

[33] J. J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling, "A set of level 3 basic linear algebra subprograms," Argonne National Laboratories, Mathematics and Computer Science Division, Tech. Rep. Reprint No. 1, Aug. 1988.

[34] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of Fortran basic linear algebra subprograms," Argonne National Laboratories, Mathematics and Computer Science Division, Tech. Rep. Technical Memorandum 41, Nov. 1986.

[35] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM TOMS*, Vol. 5, pp. 308–323, Sept. 1979.

[36] S. L. Johnsson and C.-T. Ho, "Spanning graphs for optimum broadcasting and personalized communication in hypercubes," *IEEE Trans. Comput.*, Vol. 38, pp. 1249–1268, Sept. 1989.

[37] Y. Hu and S. L. Johnsson, "A data parallel implementation of hierarchical *N*-body methods," *Int. J. Supercomput. Appl.*, Vol. 10, 1996.

[38] J. L. Hennessy and D. A. Patterson, "*Computer Architecture: A Quantative Approach*," San Mateo, CA: Morgan Kaufmann Publishers Inc., 1990.