The Parallel Multipole Method on the Connection Machine

Feng Zhao and S. Lennart Johnsson

YALEU/DCS/TR-749
October 1989

The Parallel Multipole Method on the Connection Machine®

Feng Zhao
MIT Artificial Intelligence Laboratory
545 Technology Square, Cambridge

S. Lennart Johnsson*
Thinking Machines Corporation
245 First Street, Cambridge

October 23, 1989

Abstract

This paper reports on a fast implementation of the three-dimensional non-adaptive Parallel Multipole Method (PMM) on the Connection Machine system model CM–2. The data interactions within the decomposition tree are modeled by a hierarchy of three dimensional grids forming a pyramid in which parent nodes have degree eight. The base of the pyramid is embedded in the Connection Machine as a three dimensional grid. The standard grid embedding feature is used. For 10 or more particles per processor the communication time is insignificant. The evaluation of the potential field for a system with 128k particles takes 5 seconds, and a million particle system about 3 minutes. The maximum number of particles that can be represented in 2G bytes of primary storage is $\sim$ 50 million. The execution rate of this implementation of the PMM is at about 1.7 Gflops/sec for a particle-processor-ratio of 10 or greater. A further speed improvement is possible by an improved use of the memory hierarchy associated with each floating-point unit in the system.

*Also affiliated with the Dept. of Computer Science, Yale University

1 Introduction

Many physical phenomena can be modeled as an ensemble of N particles interacting in a gravitational (or Coulombic) field, known as the N-body problem. Efficient computation of the gravitational (or Coulombic) potentials and forces exerted on each other by N particles has been of great interest. Recently, a class of fast algorithms known as tree-like particle methods has been developed. These methods, notably the Barnes-Hut tree algorithm [1] and the Multipole Method [8, 20], compute the potentials (or the forces) by partitioning them into two parts:

$$\phi_{total} = \phi_{near-field} + \phi_{far-field}, \tag{1}$$

where $\phi_{near-field}$ is the potential due to nearby particles and $\phi_{far-field}$ is the potential due to faraway particles. Since $\phi_{near-field}$ can be computed directly and relatively fast, we are mostly concerned with the fast computation of $\phi_{far-field}$.

The tree-like particle methods compute $\phi_{far-field}$ by recursive decomposition of the computational domain. The decomposability of $\phi_{far-field}$ is the direct consequence of the assumed additivity of $\phi_{far-field}$ and the spatial locality in the particle distribution. The same tree data structure can be used both for the Barnes-Hut and the Greengard-Rokhlin algorithms.

An example of additive potential fields is a Newtonian particle system. The Newtonian gravitational field $\overline{E}_{x_0}(x)$ on a particle with unit mass at location x due to a particle with mass m at location x_0 is given by

$$\overline{E}_{x_0}(x) = G\frac{m}{\| \overline{x} - \overline{x}_0 \|^3}(\overline{x} - \overline{x}_0),$$

where $\| \overline{x} - \overline{x}_0 \|$ is the Euclidean distance between locations $\overline{x}$ and $\overline{x}_0$.

Greengard and Rokhlin [8] have shown that the Multipole Method solves the N-body problem in linear time on sequential machines. The focus of this paper is the implementation of the non-adaptive Multipole Method on the Connection Machine. We first review the method and its computational structure, then discuss the efficient implementation of the required communication, and conclude with experimental results on performance.

2 The Parallel Multipole Method

The Parallel Multipole Method (PMM) is the parallel version of the Multipole Method. It achieves significant speed-up on parallel machines [20, 6]. In [20] we presented a data parallel version of the PMM, and showed that its parallel complexity is $O(\log N)$, asymptotically. The asymptotic growth rate was verified by an implementation on the Connection Machine.

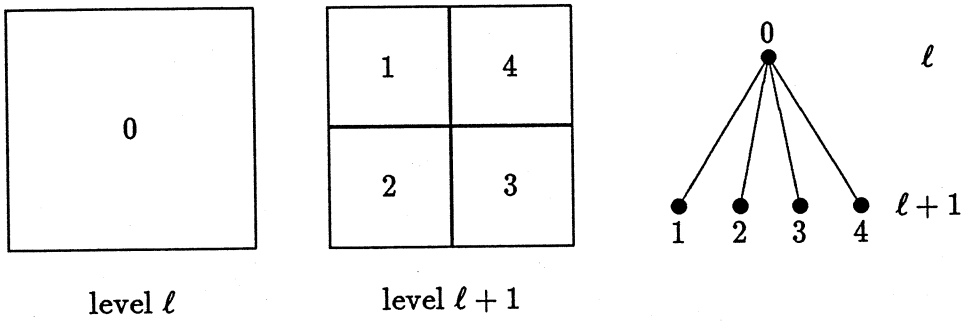

Figure 1: Recursive domain decomposition and the decomposition tree in two spatial dimensions.

2.1 Decomposition tree

The computational domain of the Multipole Method is a square in two spatial dimensions and a cube in three spatial dimensions. The domain is recursively divided into subdomains, i.e., subsquares in two dimensions and subcubes in three dimensions. A domain subject to decomposition is the *parent* of its subdomains after the decomposition. These subdomains are the *children* of the parent domain. The recursive decomposition process continues until some prespecified condition is met. Clearly, there is no reason to further decompose a subdomain containing only one, or no particles. However, as has been shown both analytically [5] and in several implementations [20, 6, 7] the direct method is faster than the Multipole Method for particle systems with sufficiently few particles. To minimize the computational complexity the decomposition of a subdomain should stop at a level where for instance using the direct method instead yields a lower comutational complexity. Subdomains that are not further decomposed are *leaves*.

The recursive decomposition is most conveniently represented by a *decomposition tree*, within which a node corresponds to a subdomain in the decomposition process. The root of the tree is the original domain under consideration. Each parent has four children in two dimensions, and eight children in three dimensions. With a uniform distribution of particles over the domain the height of the decomposition tree is at most $\log_d N$, where $d = 4$ for two spatial dimensions, and $d = 8$ for three spatial dimensions. The root is at level 0, and the leaves at level $h \leq \log_d N$. The implementation of the PMM described here is non-adaptive. The complexity estimates, as well as the implementation, assumes a uniform distribution of particles in the domain. The decomposition tree in two dimensions is illustrated in Figure 1.

The Multipole Method involves computations at all levels of the decomposition tree. In the non-adaptive case the decomposition tree is balanced. This tree is represented by a hierarchy of grids, or a pyramid. A point within the grid at level ℓ represents a subdomain corresponding to a node of the decomposition tree at level ℓ. In three dimensions the

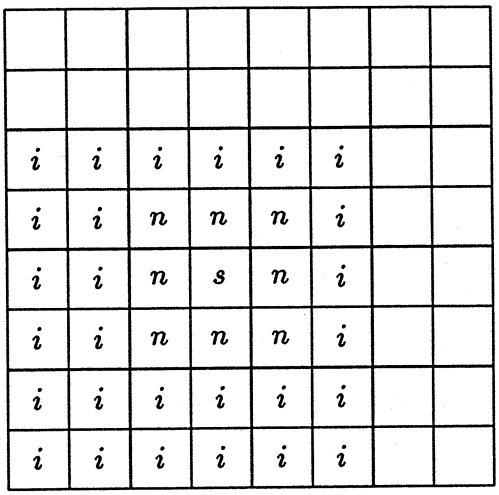

Figure 2: The near-field and the interactive-field in two dimensions.

grid size at level ℓ is $2^\ell \times 2^\ell \times 2^\ell$. The computations consist of interactions between parent/child nodes in the pyramid, i.e., between adjacent grid levels, and among grid points representing subdomains within a local neighborhood at each level of the pyramid. For the Multipole Method it is advantageous to distinguish among three regions relative to each grid point at each grid level [8, 5, 20, 18]: the *near-field*, the *interactive-field*, and the *far-field*. The *near-field* consists of those subdomains that share a boundary, or an edge, or a corner with the considered subdomain. In two dimensions the near-field consists of eight subdomains, i.e., eight adjacent grid points. In three dimensions the number of subdomains (grid points) in the near-field is 26. The near-field is represented by a 9-point stencil in two dimensions and by a 27-point stencil in three dimensions. The *far-field* of a subdomain is the entire domain excluding the subdomain and its near-field. In three spatial dimensions the number of subdomains (grid points) in the far-field is $2^{3\ell} - 27$ for a subdomain at level ℓ, $\ell > 1$. The *interactive-field* of a subdomain is the part of the far-field that is contained in the near-field of its parent. The number of subdomains in the interactive-field in two dimensions is $27 = 6^2 - 3^2$, and in three dimensions the interactive field consists of $189 = 6^3 - 3^3$ subdomains. The concepts of near-field and interactive-field are illustrated in two dimensions in Figure 2 [20]. For the subdomain s, its near-field is the union of subdomains labeled n and its interactive-field is the union of those labeled i.

2.2 The algorithm

The Multipole Method can be specified in terms of three functions G, Φ and Ψ, three translation operators T_1, T_2 and T_3, and a set of recursive equations [8, 20, 18]. G is

the potential function in explicit Newtonian formulation. T_1, T_2 and T_3 are higher order translation operators on the functions Φ and Ψ[1]

$$T_1 : (\Phi, y^{(0)}) \rightarrow (\Phi, y'^{(0)})$$
$$T_2 : (\Phi, y^{(0)}) \rightarrow (\Psi, y'^{(0)})$$
$$T_3 : (\Psi, y^{(0)}) \rightarrow (\Psi, y'^{(0)})$$

where $y^{(0)}$ and $y'^{(0)}$ are the reference centers for the corresponding functions Φ and Ψ.

Φ represents the far-field and Ψ the local field. More precisely, Φ_i^ℓ is the contribution to the potential field in the far-field region of subdomain i at level ℓ due to the particles in subdomain i, as computed by a multipole expansion centered at the center of subdomain i. Ψ_i^ℓ represents the contribution to the potential field in subdomain i at level ℓ due to particles in the far-field region of subdomain i. For the computation of Ψ_i^ℓ it is convenient to introduce $\tilde{\Psi}_i^\ell$, which represents the contribution to the potential field in subdomain i at level ℓ due to the particles in the far-field of its parent domain. In order to obtain Ψ_i^ℓ it is necessary to add to $\tilde{\Psi}_i^\ell$ the contributions of the particles in the interactive-field of subdomain i at level ℓ. The interactive-field of subdomain i at level ℓ is part of its far-field.

The Multipole Method computes $\Phi_{far-field}$ of (1) in two distinct phases. The computations performed at intermediate levels of the decomposition tree reduce the far-field interactions via an upward-pass and an downward-pass. The tree is traversed twice. In the upward-pass the far-field interactions Φ_i^ℓ are computed by shifting the multipole expansions of child nodes to the center of the parent node, and adding the result. The downward-pass through the tree distributes the results of far-field interactions, aligns them properly, and combines them to form the local expansion Ψ. The computation of $\Phi_{near-field}$ of (1) performed at leaf nodes reduces near-field interactions. In the recursive formulation by J. Katzenelson [18], the upward-pass consists of the computation

$$\text{Compute } \Phi_i^h \text{ for all nodes } i \text{ at the leaf level } h \tag{2}$$

$$\Phi_n^\ell = \sum_{i \in \{children(n)\}} T_1(\Phi_i^{\ell+1}), \tag{3}$$

and the downward-pass consists of the computation

$$\tilde{\Psi}_i^\ell = T_3(\Psi_{parent(i)}^{\ell-1}) \tag{4}$$

$$\Psi_i^\ell = \tilde{\Psi}_i^\ell + \sum_{j \in \{interactive-field(i)\}} T_2(\Phi_j^\ell). \tag{5}$$

The first term in Ψ_i^ℓ computes the local field due to the particles in the far-field of the parent node by shifting the center of the expansion from the center of the parent domain

[1] T_1, T_2 and T_3 are defined in Lemmas 3.2.1 through 3.2.3 in [20].

to that of the current subdomain. The second term converts the far-field expansions to a local expansion.

At the end of this recursion it remains to compute the potential field due to particles in the near-field of subdomain i at level h. This contribution to the potential field is computed by a direct evaluation of the Newtonian interaction with nearby particles

$$\Phi_i^h{}_{near-field} = \sum_{j \in \{near-field(i)\}} G_j(i). \tag{6}$$

The complete solution, therefore, is given by

$$\Phi_i^h{}_{total} = \Psi_i^h + \Phi_i^h{}_{near-field}.$$

In the upward-pass (3), the shifting operations of T_1 for different subdomains within each level are independent of each other and can be performed concurrently, as long as the computation at levels below is completed. Similarly, the shifting operations of T_3 in the downward-pass (5) can be performed concurrently across all levels of the grids. The conversion of the far-field multipole expansions to local expansions represented by the operator T_2 can also be performed concurrently for all subdomains at a given level. The reduction operation in the upward-pass can be performed concurrently not only for different parent nodes, but also for each parent node by performing the reduction in a binary-tree-like manner with trees of height two in two dimensions and height three in three dimensions. The distribution operation in the downward-pass has the same characteristics with respect to concurrency as the reduction in the upward-pass. The computations of (6) at different leaf nodes are entirely independent of each other, and can proceed concurrently. Moreover, the computations within any level of the grid are uniform across all grid points. The PMM exploits the high degree of independence and the regularity of the Multipole Method, and achieves significant speed-up on a large number of particles with a large number of processors. The PMM requires $O(\log N)$ time asymptotically for N particles distributed with a fixed number of particles per processor [20, 6].

The PMM algorithm performs the computations implied by the expressions (2), (3), (4), (5), and (6). Expression (2) is evaluated concurrently for all leaf nodes. In each such node a multipole expansion due to all particles within the leaf node is formed. The multipole expansions formed at the leaf nodes are shifted and reduced in parallel during the upward-pass through the tree, as described by (3). The far-field interactions are then accumulated and distributed in parallel during the down-ward pass through the decomposition tree, as defined by (5). The far-field interaction for each of the particles is obtained by evaluating the multipole expansion from (5) at the particle position. The near-field interaction at the leaf nodes is computed by direct evaluation of the Newtonian gravitational field. This evaluation is performed concurrently for all leaf nodes. The sum of the far-field interaction and the near-field interaction gives the answer.

The algorithm for PMM on the Connection Machine is given in Figure 3.

```
(1) global-initialization
  set-up arrays for Φ, Ψ, and intermediate results;
    precompute difference vectors among tree nodes;
    precompute shifting arrays by theorem-3-2-2 in [20] for
      local-field-reduction;
    precompute mathematical tables.

(2) far-field-reduction
  init-expansion at leaf nodes
    forall nodes i at leaf-level h do
      compute Φ_i^h by theorem-3-2-1 in [20];
    end
  multigrid-reduction
    for level ℓ from h to 1 step -1 do
      forall nodes i at level ℓ do
        compute Φ_i^ℓ by lemma-3-2-1 in [20];
        reconfigure CM grid;
        reduce Φ_i^ℓ with + op;
      end
    end

(3) local-field-reduction
  multigrid-distribution
    for level ℓ from 1 to h do
      forall nodes i at level ℓ do
        compute Σ_{j∈{interactive−field(i)}} T_2(Φ_j^ℓ) by lemma-3-2-2 in [20];
        compute Ψ̃_i^ℓ by lemma-3-2-3 in [20];
        compute Ψ_i^ℓ;
        distribute Ψ_i^ℓ with identity op;
        reconfigure CM grid;
      end
    end

(4) local-interaction
  forall nodes i at leaf-level h do
    for j ∈ {near − field(i)} do
      for each particle in node i do
        for each particle in node j do
          compute G_j(i);
        end
      end
    end
    compute Φ_i^h near−field;
  end

(5) final-evaluation
  forall nodes i at leaf-level h do
    for each particle p in node i do
      evaluate Ψ_i^h at p;
      compute Φ_i^h total for p;
    end
  end
```


Figure 3: The Parallel Multipole Method on the Connection Machine

3 Embedding of hierarchical grids

The Connection Machine system model CM-2 has 64k bit-serial processors divided evenly among 4096 processor chips. These chips are interconnected as a 12-dimensional Boolean cube. The Connection Machine model CM-2 can also be equipped with hardware floating-point units (fpu's). Two processor chips share such a unit. Each Connection Machine processor is associated with 8k bytes of storage with 256k bit memory chips, or 32k bytes with 1M bit memory chips. The storage per floating-point unit is 64k or 256k 32-bit words. The total storage is 512M bytes, or 2G bytes. The programming systems on the Connection Machine supports the notion of *virtual processors*. A program is written for a number of virtual processors consistent with the application. A virtual processor may for instance represent the data associated with a grid point, and carry out all the computations associated with it. The virtual processors are mapped to *real* processors. The number of virtual processors assigned to each real processor is the *virtual processor ratio*. Virtual processors time share a real processor. Each virtual processor assigned to a real processor is assigned a distinct segment of real processor memory.

Inter-processor communication can take place either by using the general routing facility, or via embedded grids. The Boolean cube network interconnecting the processor chips contains regular grids of up to twelve dimensions as subgraphs. Address encoding by a binary-reflected Gray code [19] provides a mechanism for grid emulation [17, 13]. The dimensionality and shape of the grid can be altered dynamically. The Gray code has the property that adjacent integers are at *Hamming* distance one, i.e., $Hamming(i, i \pm 1)=1$. By using this encoding successive integers are mapped into adjacent nodes of a Boolean cube. Each node of a Boolean cube of 2^n nodes has n neighbors. The address of a neighbor can be obtained by complementing a bit in the address.

For a lattice with several axes each axis can be encoded in a binary-reflected Gray code. The address space is partitioned with $\lceil \log_2 N_i \rceil$ bits assigned to the encoding of the N_i elements along axis i. All edges of the lattice are mapped to distinct cube edges. The separate binary-reflected Gray code encoding of each array axis results in an effective utilization of the nodes in the Boolean cube, if the length of the array axes is a power of two. In general, the utilization of the nodes in the Boolean cube offered by the Gray code encoding is $\frac{\lceil \log_2 \Pi_{i=1}^{d} N_i \rceil}{\Pi_{i=1}^{d} \lceil \log_2 N_i \rceil}$. For grids of arbitrary shape the utilization may be as low as $\sim \frac{1}{2^d}$. Any embedding of grids into Boolean cubes preserving adjacency will have this processor utilization [9]. In order to increase the processor utilization for grids of arbitrary shape it is necessary to allow some grid edges to be mapped into paths of a length greater than one. The *dilation* of an edge is the length of the path into which it is mapped, and the dilation of the embedding is the maximum dilation of any edge. The *expansion* of the embedding is the ratio between the number of nodes of the Boolean cube required for the embedding and the number of nodes in the grid being embedded. Any two-dimensional grid can be embedded into a Boolean cube with dilation two and minimum expansion [2].

Any three dimensional grid can be embedded with at most dilation seven and minimal expansion [4]. Grids with k dimensions can be embedded with a dilation of at most $4k+1$ and minimal expansion [3]. Several dilation two embeddings of two and three dimensional grids are given in [10, 11].

In the Connection Machine programming systems data represented by arrays configure the address space as a grid. Configuring the address space as a grid implies that the array indices are encoded in a binary-reflected Gray code. Each axis is encoded separately. The encoding is transparent to the programmer. The Connection Machine address field has three parts ($off\text{-}chip|on\text{-}chip|memory$). The off-chip address field has 12 bits, the lowest order bit of which encodes the pair of Connection Machine processor chips sharing a floating-point unit (fpu). The on-chip address field encodes the processors on a Connection Machine processor chip. The memory address field encodes the bits of the memory local to a processor. Only the off-chip part of the address field is subject to the binary-reflected Gray code encoding in the lattice emulation mode. The on-chip and memory fields are encoded in binary code.

With a non-adaptive recursive decomposition of the domain it suffices to consider grids with axis lengths being powers of two. For such grids the following two properties [12, 13] of the binary-reflected Gray code are important for the embedding of a hierarchy of grids: $Hamming(i, i \oplus 2^j) = 2, j > 0$, and $Hamming(i, i+3)$=1 for $i \bmod 2 = 0$. If the computations performed on a hierarchy of grids only take place at one grid level at a time, then the different grids can be mapped on to the same set of processors without loss of efficiency. By embedding the finest grid in a Boolean cube by a binary-reflected Gray code adjacent nodes in a coarser grid consisting of every other grid point along an axis are at distance two. The same property is true for all coarser grids obtained in the same manner. Hence, even though successively coarser grids consist of points with indices differing by increasing powers of two, the points are always in proximity when the grid is mapped to a Boolean cube by a binary-reflected Gray code. This property is easily proven [12] and apparent from Figure 4.

Communication between adjacent nodes in the finest grid is nearest neighbor communication in the Boolean cube. Communication between adjacent nodes in coarser grids implies communication between processors at distance two in the Boolean cube. All distance two communications can be arranged such that there is no contention for channels [12]. Then, the communication time for bit-serial pipelined communication, which is the communication mode on the Connection Machine system, is dominated by the message length. The path length is an additive term.

Subselecting grid points for coarse grids by choosing the grid at level ℓ to consist of all points such that $i \bmod 2^{h-\ell} = 0$ for each axis causes the grid points to be allocated to different subcubes. In [13] an exchange step was introduced to move the selected points into subcubes identified by address bits. After the exchange the subselected points are embedded by a binary-reflected Gray code in a half size subcube. For instance, in the example above, if 2 and 3, 6 and 7, 10 and 11, and 14 and 15 are exchanged, then the

Integer	Gray code
0	0000
1	0001
2	0011
3	0010
4	0110
5	0111
6	0101
7	0100
8	1100
9	1101
10	1111
11	1110
12	1010
13	1011
14	1001
15	1000

Figure 4: A 4-bit binary-reflected Gray code.

grid points with the same lowest order bit are Gray coded in a code with one less bit, i.e., at Hamming distance one. Locations with the lowest order bit zero contain all the even points, locations with the lowest order bit one contain all the odd grid points. In general, at each step $k, k \geq 0$ exchanging grid points between locations that differ only in bit k if the parity of bits $k+1$ through $n-1$ for an n-bit code is odd guarantees that the subselected grid points are at distance one. This scheme converts the binary-reflected Gray code to a binary code [13, 14]. This exchange algorithm implicitly and recursively makes use of the fact that $Hamming(i, i+3)=1$ for $i \bmod 2 = 0$. If the interaction between adjacent grid levels consists in a reduction/distribution operation the exchange step need not be performed. In the one-dimensional case and a reduction on pairs of grid points the reduction operation on nodes $i \bmod 4 = 0$ and $i \bmod 4 = 1$ can be rooted in grid point $i \bmod 4 = 0$, and the reduction on $i \bmod 4 = 2$ and $i \bmod 4 = 3$ rooted in $i \bmod 4 = 3$. This strategy is used in our implementation of the PMM. It was previously used in [16, 15].

4 Implementation

4.1 Overview

The first implementation of the PMM on the Connection Machine [20] mapped the nodes at all levels of the decomposition tree to processors by a binary encoding of the node addresses. The nodes of the decomposition tree were labeled in a breadth first order. At the time of the first Connection Machine implementation, the grid communication mechanism was not available. Communication between processors was by way of the router. The binary encoding does not preserve locality in the index space. The Hamming distance between consecutive integers may be as high as the number of bits required for the encoding. With this mapping of the grid points to processors more than 60% of the execution time is spent in communication.

The objective of the implementation of the PMM described here was to yield high performance on three-dimensional problems with a large number of particles, say 1,000,000 or more. In order to exploit the locality of reference present in the non-adaptive recursive subdivision of the domain, the grid emulation feature of the Connection Machine programming systems was used. The address space was configured as a three dimensional grid. The different grids in the hierarchy were mapped to processing units such that coarser grids were contained in subcubes identified by higher order bits. The reduction/distribution operations were performed such that the grids at any level were embedded by a binary-reflected Gray code, as described above.

In the *field-wise* programming model a word is allocated serially, i.e., in successive memory locations of a processor. The field-wise programming model is used by all programming languages on the Connection Machine. In this model each processor has its own memory. But, a floating-point unit shared by 32 Connection Machine processors can access their memories in bit-slices. Hence, with a word stored with one bit per Connection Machine processor a word can be accessed by a floating-point unit in a single cycle. This type of data representation and data access corresponds to the *slice-wise* programming model of the Connection Machine system. In this model the Connection Machine consists of 2,048 processing units interconnected as an 11-dimensional cube with two communication channels between each pair of units. The slice-wise view of the Connection Machine offers the potential for exploiting the memory hierarchy introduced through the registers within the floating-point unit. The registers and associated buses on the floating-point unit increase the effective memory bandwidth. The floating-point units are not complete processors and some operations have to be performed on the Connection Machine processors. At the time of this implementation the supported programming systems did not contain features allowing access to the internal features of the floating-point units. Many functions were implemented using a mixture of the field-wise bit-serial model and the slice-wise 32-bit wide model.

In three dimensions the near-field consists of 26 subcubes, and the interactive-field

consists of 189 ($=6^3 - 27$) subcubes. In order to simplify the implementation a $7 \times 7 \times 7$ neighborhood of subcubes (consisting of 343 ($=7^3$) subcubes), symmetrized with respect to each subcube by including a few more "null" subcubes from the far-field and the near-field, is considered, and an interactive-field mask is used to subselect from the enlarged neighborhood the subcubes of the interactive-field. The dominating operations are reduction, distribution, and convolution as defined by the near-field (a 27-point kernel with respect to grid points) and the interactive-field. The implementation is restricted to one leaf node per floating-point unit. The leaf node is represented in the field-wise model, i.e., by 32 Connection Machine processors.

In our implementation computation-intensive static data are precomputed. For example, the translation operator T_2 requires that first a set of coefficients ($b_{i,j,k}$ [20]) be computed before the convolution against the Φ's contributed by the interactive-field is carried out. For each subdomain this operation is repeated for every member of its interactive-field (in our case, for up to 189 times). This computation accounts for most of the execution time within the tree. The coefficients ($b_{i,j,k}$) are obtained from expansions on difference vectors between centers of different subcubes that can be predetermined, and thus can be computed ahead of time. The precomputation makes use of the fact that there exist a subset of kernels from which the coefficients can be computed with a modest effort [18].

4.2 Reduction and distribution functions

In the upward-pass (3) of the PMM a reduction operator is applied recursively on a three-dimensional grid data structure. The downward-pass (5) accumulates the data and distributes the results. The reduction operator and the distribution operator are very general. The same type of operators are extensively used in Multigrid type of algorithms. We have implemented these two operators as generic functions called **Multigrid-reduce** and **Multigrid-distribute**. The arguments of the Multigrid-reduce function include a reduction operator, the source grid variable indicating where to get the source data to be reduced, and the destination grid variable on a smaller grid specifying where to put the result. The operator for the Multigrid-distribute function is always a copy operator. The reduction of grid size as described above is implicit in the Multigrid-reduce function, and the expansion is implicit in the Multigrid-distribute function.

The subselection of grid points can be achieved by reshaping each axis. By adding a "dummy" axis for each array axis a plane is created for each axis. The shape of this plane is changed throughout the recursion such that the total number of grid points in the plane is preserved for every step. For instance, an axis with 32 points is represented as a 1×32 array, and for successively coarser grids is represented as a 2×16, 4×8, 8×4, 16×2, and 32×1 array. This reconfiguration can take place dynamically. The first axis requires 0,1,2,3,4 and 5 bits, respectively. The second axis requires 5,4,3,2,1 and 0 bits. If the encoding of the elements along the first axis is assigned bits starting with the lowest

order bit, then successive points along the second axis in the 2×16 grid consists of all even points in the first grid, etc.

$$(\underbrace{a_4a_3a_2a_1a_0}_{axis-1}) \qquad (\underbrace{a_4a_3a_2a_1}_{axis-1}\ \underbrace{a_0}_{axis-0}) \qquad (\underbrace{a_4a_3a_2}_{axis-1}\ \underbrace{a_1a_0}_{axis-0})\ldots$$

Applying reshaping to the on-chip part of the axis yields no performance advantage. The lowest order off-chip address bit shall be treated in the same manner as the on-chip bits, since it defines the processor chips sharing memory. An example of the representation of one reshaping step of a $16 \times 16 \times 16$ grid is the following: (1 16 1 16 1 16) $\rightarrow$ (2 8 2 8 2 8). The second, fourth and sixth dimensions are reduced. The first, third, and fifth dimensions are used to collect grid points not part of the reduced grid.

The syntax for the Multigrid-reduce function is

Multigrid−reduce(*dest src reduce−operator indices*)

The Multigrid-Reduce reduces *src* with *reduce-operator* and stores the result in *dest*. *indices* is a list of axes along which reduction takes place. Multigrid-Reduce also reduces the geometry of the *src* and returns the *dest* with the reduced geometry.

As an example, *Multigrid−reduce*(*dest source '+!! '*(2 4)) will reduce *source* with +!! along the 2nd and the 4th axes, and store the result in *dest*.

Likewise the syntax for the Multigrid-distribute function is

Multigrid−distribute(*dest src distribute−operator indices*)

4.3 Direct interaction

The computation of the direct interaction with particles in the near field at the base of the pyramid is implemented in the field-wise model, except for the data motion. The interaction between a pair of subdomains is defined by the interaction of every particle in one domain with every particle in the other domain. The communication is implemented by first performing an exchange of particle information between the domains, then rotating the received information through the Connection Machine processors representing the subdomain. After a complete revolution the interaction between all particles in one domain with all particles in the other domain has been evaluated. The rotation within a domain is implemented using some of the slice-wise features of the Connection Machine. The measured performance of this implementation is 1.46 Gflops/sec on a 64k processor Connection Machine, which is very close to the maximum possible within the high level languages at a virtual processor ratio of one.

5 Results

The first implementation of the PMM on the Connection Machine [20] required 94 seconds on a 8k CM–2 for the evaluation of the potential field due to 16,000 particles. More than half of the total time was spent in communication. The measured performance of the new implementation of the PMM on the Connection Machine is 1.67 Gflops/sec at a particle-processor-ratio of 10. The communication time is insignificant. With the new implementation the computation of the potential field for 16,000 particles requires 5 seconds on a 8k CM–2.

5.1 Timing results

In the experiments reported here we compute the potential field using third order multipole expansions. The measured root-mean-squared error E_{rms} is less than 10^{-2}. Only uniform particle distributions were used. The performance measurements were carried out on a 16k CM–2 at a *virtual processor* ratio of 1. There was one leaf node, or one grid point in the base of the pyramid, per floating-point unit associated with a cluster of 32 CM processors. With ten or more particles per CM processor the execution time is entirely dominated by the computation of the local interaction, which is made by a direct evaluation.

	Running Time			
particle-processor-ratio	1	2	3	4
init-expansion (ms)	26.6	36.6	46.6	57.0
far-field (ms)	21.7	21.7	21.5	21.5
local-field (sec)	1.78	1.86	1.76	1.79
multipole-exp-eval (ms)	7.07	14.2	21.2	28.3
local-direct (sec)	0.846	2.99	6.45	11.2
total (sec)	2.68	4.92	8.30	13.1
particle-processor-ratio	10	20	30	40
init-expansion (ms)	116	218	316	435
far-field (ms)	21.5	21.5	21.8	21.5
local-field (sec)	1.79	1.76	1.81	1.75
multipole-exp-eval (ms)	71.1	149	222	291
local-direct (sec)	66.8	264	590	1050
total (sec)	68.8	266	592	1052

Table 1: Timing results for potential field evaluation by the Parallel Multipole Method.

Explanation of the entries in Table 1:

- *init-expansion*: form multipole expansions at leaf nodes;

- *far-field*: upward-pass through the pyramid;
- *local-field*: downward-pass through the pyramid;
- *multipole-exp-eval*: evaluate the resulting multipole expansions at each particle position;
- *local-direct*: compute interactions with nearby particles directly.

5.2 Complexity and performance analysis

5.2.1 Time complexity

Using the timing results from Table 1, we can express the time complexity for each of the subroutines of the PMM in terms of the particle-processor-ratio r and with one leaf node per floating-point unit

$$\begin{aligned} T_{init-expansion} &= 16.6 + 10r \quad (ms) \\ T_{far-field} &= 21.6 \quad (ms) \\ T_{local-field} &= 1.79 \quad (sec) \\ T_{multipole-exp-eval} &= 7.1r \quad (ms) \\ T_{local-direct} &= 0.197r + 0.649r^2 \quad (sec) \end{aligned}$$

The total running time of the PMM is

$$T_{total}(r) = 1.84 + 0.214r + 0.649r^2 \quad (sec).$$

In order to gain insight into the relative importance of communication T_{comm} and computation T_{comp} for the PMM, detailed timings were conducted and summarized in Table 2.

The results of Table 2 show that for $r = 1$ the time for communication is 21.5% of the total time. The T_{comp} component grows much faster with r than the T_{comm} component. For $r = 10$, T_{comm} is only 3.4% of the total time. The computation in *far-field* and *local-field* is essentially 3D convolution, and the one in *local-direct* is Newtonian interaction.

5.2.2 Speed

The *local-field* computations require about $(343 + \frac{343}{8}) \times (\frac{40}{8} + 32)$ floating-point operations in our implementation. The computations labeled *local-direct* require $14 \times 32 \times 32 \times$

	Running Time	
time component	T_{comm}	T_{comp}
init-expansion (ms)	17.7	$10r - 1.1$
far-field (ms)	3.8	17.8
local-field (sec)	0.36	1.43
multipole-exp-eval (ms)	0	$7.1r$
local-direct (sec)	$0.197r$	$0.649r^2$
total (sec)	$0.38 + 0.197r$	$1.45 + 0.017r + 0.649r^2$

Table 2: Communication and computation time as functions of the number of particles per processor.

$40r^2$ floating-point operations. Since these two parts are dominant in terms of floating point operations, we use them as an approximation for the total number of operations. With W floating-point processors and one leaf node of the decomposition tree per floating-point unit the speed of the PMM is determined by

$$\begin{aligned} Speed_{PMM}(r) &= \frac{((343 + \frac{343}{8}) \times (\frac{40}{8} + 32) + 14 \times 32 \times 32 \times 40r^2)W}{T_{total}(r)} \\ &= \frac{(14277 + 573440r^2)W}{1.84 + 0.214r + 0.649r^2} \quad (flop/sec) \end{aligned}$$

On a 64k CM–2 $W = 2048$. For $r = 1$, $Speed_{PMM}(1) = 435$ Mflops/sec; For $r = 10$, $Speed_{PMM}(10) = 1.67$ Gflops/sec, which is close to the asymptotic speed 1.81 Gflops/sec. The performance for the arithmetic operations $+, -, \times$ is 1.5–2.4 Gflops/sec in *Lisp, with the lower figure achievable for a virtual processor ratio of one. All of our timings were made at a virtual processor ratio of one. Some code segments are implemented in lower level code, which explains a peak performance in excess of the *Lisp performance. But, an improvement in the arithmetic performance by a factor of 2–3 is expected from a complete change to the slice-wise, 32-bit wide, model.

5.2.3 Storage requirements

With a multipole expansion of degree p, the total number of terms retained in the expansion is

$$m = \frac{1}{6}(p+1)(p+2)(p+3).$$

For a pyramid of h levels the state and the problem description for a grid point at the base (leaf node) *and* its ancestors require $(h-2)m$ words of storage for Φ, $2m$ words for Ψ,

$\lceil (h-2)\frac{343}{32} \rceil$ words for the interactive-field masks, $6(h-2)$ words for cell coordinates, and about another 100 words for miscellaneous variables. In the downward-pass $343m$ words are required to store the intermediate results. Each particle takes 10 words of storage (the position, the velocity, and the acceleration require three words each plus one word for the mass). Thus, the total storage required for a grid point at the base (leaf node) and its ancestors is

$$S \doteq 88 + 343m + \lceil (h-2)\frac{343}{32} \rceil + h(m+6) + 10R \quad (words),$$

where R is the particle-node-ratio. For $p = 3$ (and therefore $m = 20$), a three level tree requires $S \doteq 7037 + 10R$ $(words)$, a four level tree $S \doteq 7074 + 10R$ $(words)$, and a five level tree $S \doteq 7111 + 10R$ $(words)$ for a grid point at the base and its ancestors. Clearly, the storage requirement per leaf node is nearly independent of the depth of the tree.

In the field-wise mode each of the 32 processors sharing a floating-point unit allocates the same storage. Our implementation uses the model of mixed field-wise and slice-wise data representation. Each CM processor represents different particles. The part of storage for $343m$ words is allocated slice-wise. The rest of the memory is allocated field-wise, that is, replicated in each CM processor, if there are fewer than 32 leaf nodes per floating-point unit. The actual storage allocated for each leaf node in our implementation is

$$S_{mixed} \doteq 343m + 32[(88 + \lceil (h-2)\frac{343}{32} \rceil + h(m+6)] + 10R \quad (words),$$

where R is again the particle-node-ratio with one leaf node per floating-point unit.

For a Connection Machine system the maximum depth of the decomposition tree that is of interest is three, four, or five, which corresponds to 512, 4096, or 32768 leaf nodes, respectively. In a 64k processor Connection Machine system the number of leaf nodes per floating-point unit is 2 for a four level tree, and 16 for a five level tree. Tables 3 and 4 summarize the number of particles that can be represented with 64k words and 256k words of storage per floating-point unit as a function of the height of the decomposition tree for the slice-wise data representation and for the mixed field-wise and slice-wise data representation. Figure 5 shows the total number of particles that can be represented in the slice-wise representation. With the increase in the height of the decomposition tree the number of leaf nodes increases and the total number of particles that can be represented in a fixed size storage decreases. The difference in the two models of storage allocation is the reason for the difference in the size of the particle systems between Tables 3 and 4.

In our implementation that uses the mixed data representation on a 16k CM, with 64k words of storage per floating-point unit the maximum number of particles per leaf node is $\sim 5,300$ for a three level tree. With 256k words of memory per floating-point unit the maximum number of particles per leaf node for a three level tree is $\sim 25,000$. The measured maximum number of particles for a three level tree using the mixed field-wise and slice-wise representation is about $4160 (= 130 \times 32)$. The difference between the

Storage per fpu (words)	Tree height	Connection Machine configuration											
		8k			16k			32k			64k		
		Particles			Particles			Particles			Particles		
		leaf	fpu	Total	leaf	fpu	Total	leaf	fpu	Total	leaf	fpu	Total
64k	$h = 3$	2,570	5,150	1.32 M	5,850	5,850	3.00 M	—	—	—	—	—	—
	$h = 4$	—	—	—	112	894	458 K	930	3,720	3.81M	2,570	5,140	10.5 M
256k	$h = 3$	12.4 K	24.8 K	6.35 M	25.5 K	25.5 K	13.1 M	—	—	—	—	—	—
	$h = 4$	931	14.9 K	3.81 M	2,570	20.6 K	10.5 M	5,850	23.4 K	23.9 M	12.4 K	24.8 K	50.8 M
	$h = 5$	—	—	—	—	—	—	108	3,460	3.54 M	927	14.8 K	30.4 M

Table 3: Estimated maximum number of particles per leaf node and total particle system size for some Connection Machine systems (slice-wise data representation).

Storage per fpu (words)	Tree height	Connection Machine configuration											
		8k			16k			32k			64k		
		Particles			Particles			Particles			Particles		
		leaf	fpu	Total	leaf	fpu	Total	leaf	fpu	Total	leaf	fpu	Total
64k	$h = 3$	2,310	4,620	1.18 M	5,300	5,300	2.71 M	—	—	—	—	—	—
	$h = 4$	—	—	—	48	381	195 K	781	3,120	3.20 M	2,250	4,500	9.21 M
256k	$h = 3$	12.1 K	24.3 K	6.21 M	25.0 K	25.0 K	12.8 M	—	—	—	—	—	—
	$h = 4$	910	14.6 K	3.73 M	2,510	20.0 K	10.3 M	5,700	22.8 K	23.3 M	12.1 K	24.2 K	49.5 M
	$h = 5$	—	—	—	—	—	—	108	3,460	3.54 M	902	14.4 K	29.6 M

Table 4: Estimated maximum number of particles per leaf node and total particle system size for some Connection Machine systems (mixed field-wise and slice-wise data representation).

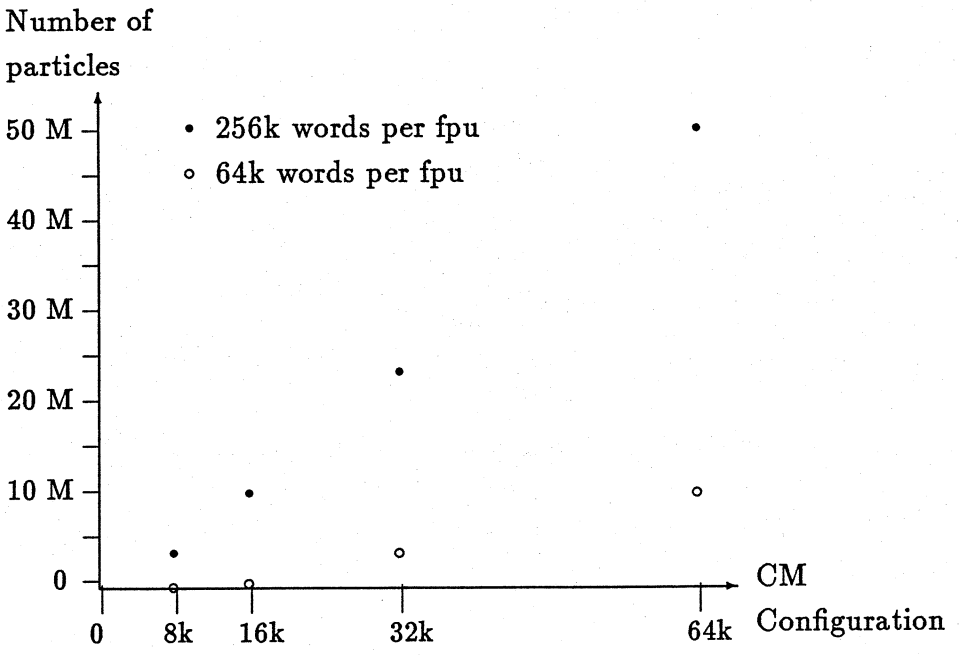

Figure 5: The maximum number of particles in primary storage for various Connection Machine systems and a four level decomposition tree (slice-wise data representation).

measured and the predicted maximum number of particles is due to the allocation of stack variables not accounted for in the expression for the required storage.

5.3 Optimizing the number of leaf nodes

The points in the grid of finest granularity represents the leaf nodes of the decomposition tree. Each leaf node holds a number of particles. The interaction among the particles in the same leaf node is computed by direct evaluation of the Newtonian gravitational field. The associated computational complexity is quadratic in the number of particles per leaf node. With a leaf node together with all its particles assigned to a single floating-point unit, the computation of the direct interactions within leaf nodes requires no communication. The computations are local and sequential for each leaf node, and "embarrassingly" parallel across different leaf nodes.

The decomposition tree of the non-adaptive Multipole Method corresponds to data interaction in the form of a pyramid with the same height h as the decomposition tree. The number of grid points at the base of the pyramid is 2^{3h}. If the number of grid points at the base is greater than the number of floating-point units 2^n, then a unit has to perform the computations of several grid points, that is, the grid points of a subpyramid of height $h - \frac{n}{3}$. The optimal value of h with respect to performance depends upon the relative overheads of different methods. Greengard and Rokhlin [5] compared the non-adaptive Multipole Method with the direct evaluation of the Newtonian field on a VAX. The cross-over point below which the direct method is faster was found to be in the range of 200 – 400 particles for a variety of two dimensional particle distributions including highly non-uniform distributions. In [7] the non-adaptive algorithm is compared with the direct method for three dimensional problems. For uniform distributions the cross-over point on a VAX-8600 is in the range of 1000 – 2000 particles. In the three dimensional implementation of the Multipole Method reported in [20] the cross-over point is at about 1000 particles. The local direct interaction at the leaf node includes the direct interaction within the same leaf node as well as that with adjacent leaf nodes. The further subdivision of a subdomain with a few thousand particles, increases the computation time for interactions within the same subdomain, however reduces the computation of the direct interactions with adjacent subdomains which are smaller in size after the subdivision. Hence, the stopping criteria for the recursive subdivision with respect to minimum execution time is at somewhat fewer number of particles per subdomain, than that indicated by the comparisons referenced above. The optimum stopping point is implementation dependent.

The stopping criterion for the subdivision of a domain represented by several floating-point processors also has to acknowledge the difference in communication needs between the methods. In the parallel implementation of the two-dimensional Multipole Method reported in [6] a comparison was made with a parallel implementation of the direct method. The two implementations were made on a shared memory machine, the Encore Multimax.

The speed-up for the direct method was almost identical to the number of processors, when the particle to processor ratio in the experiments ranged from 40 to 1250. The speed-up of the Multipole Method was about $\frac{3}{4}$ of the number of processors for 40 particles per processor, but increased to become almost identical to the number of processors for a larger number of particles per processor. We have not compared the Parallel Multipole Method with the direct method, or any other tree method for the Connection Machine. The expected speed-up for the direct method is linear in the number of processors as long as the particle-to-processor ratio is high since the communication is linear and the computations are quadratic in the number of particles. For the Multipole Method the communications and the computations are of the same order, and the speed-up is likely to be less. Assigning several processors to a subdomain favors stopping the recursive subdivision at an earlier stage.

6 Summary

The goal of the PMM implementation described here was the efficient use of the Connection Machine architecture. For the implementation reported in [20] about 60% of the total execution time was due to interprocessor communication. With a non-adaptive recursive partitioning of the domain the data interaction is defined by a pyramid in which each parent node has degree eight in three dimensions. The base of the pyramid was embedded in the Connection Machine as a three dimensional grid by making use of the lattice emulation feature of the Connection Machine programming systems. With other performance enhancements the speed-up of the new implementation compared to the old one is approximately a factor of 20. The new implementation achieves a performance in excess of 1.67 Gflops/sec for a particle-processor-ratio of 10 or higher.

The new implementation can be sped up further by more efficient coding of the arithmetic operations. All operations are currently performed as scalar operations. However, each floating-point processor can be operated as a vector processor with a performance increase of up to a factor of 10 for operations such as matrix-vector multiplication and convolution. A further speed enhancement by a factor of three is expected. By a slice-wise data representation slightly larger particle systems can be represented in the primary storage.

With 512M bytes of storage a system of about 6 million particles can be simulated without external memory. With 2G bytes of storage systems up to about 50 million particles can be simulated. The potential field evaluation for one million particles requires about 3 minutes. A field calculation for 50 million particles would require approximately 100 hours with the current implementation. Future work on the PMM includes dynamic load-balancing on particle systems with highly non-uniform distributions.

7 Acknowledgments

The authors would like to thank Ajit Agrawal for explaining the CM geometry mechanism and members of the Computational and Mathematical Sciences Groups at TMC for generous help during the course of the implementation.

References

[1] Josh Barnes and Piet Hut. A hierarchical $o(n \log n)$ force calculation algorithm. Technical report, The Institute for Advanced Study, Princeton, 1986.

[2] M.Y. Chan. Dilation-2 embeddings of grids into hypercubes. In *1988 International Conf. on Parallel Processing.* The Pennsylvania State University Press, 1988.

[3] M.Y. Chan. The embedding of grids into optimal hypercubes. Technical report, Computer Science Dept., University of Texas at Dallas, 1988.

[4] M.Y. Chan. Embeddings of 3-dimensional grids into optimal hypercubes. Technical report, Computer Science Dept., University of Texas at Dallas, 1988. To appear in the Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications, March, 1989.

[5] Leslie Greengard. *The Rapid Evaluation of Potential Fileds in Particle Systems.* PhD thesis, Dept. of Computer Science, Yale Univ., New Haven, CT, April 1987. Published by MIT Press, 1988.

[6] Leslie Greengard and William D. Gropp. A parallel version of the fast multipole method. Technical Report YALEU/DCS/RR-640, Dept. of Computer Science, Yale Univ., New Haven, CT, August 1988.

[7] Leslie Greengard and V. Rokhlin. On the efficient implementation of the fast multipole method. Technical Report YALEU/DCS/RR-602, Dept. of Computer Science, Yale Univ., New Haven, CT, February 1988.

[8] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. Technical Report YALEU/DCS/RR-459, Dept. of Computer Science, Yale Univ., New Haven, CT, 1986.

[9] I. Havel and J. Móravek. B-valuations of graphs. *Czech. Math. J.*, 22:338–351, 1972.

[10] Ching-Tien Ho and S. Lennart Johnsson. On the embedding of arbitrary meshes in Boolean cubes with expansion two dilation two. In *1987 International Conf. on Parallel Processing*, pages 188–191. IEEE Computer Society, 1987.

[11] Ching-Tien Ho and S. Lennart Johnsson. Embedding meshes in Boolean cubes by graph decomposition. *Journal of Parallel and Distributed Computing*, 7(3), December 1989. Technical Report YALEU/DCS/RR-746, Department of Computer Science Yale University, September 1989. This is a revision of Technical Report YALEU/DCS/RR-689 March 1989, Technical Report DA89-1, Thinking Machines Corp., September 1989.

[12] S. Lennart Johnsson. Odd-even cyclic reduction on ensemble architectures and the solution tridiagonal systems of equations. Technical Report YALE/DCS/RR-339, Dept. of Computer Science, Yale University, October 1984.

[13] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distributed Comput.*, 4(2):133–172, April 1987.

[14] S. Lennart Johnsson. *Optimal Communication in Distributed and Shared Memory Models of Computation on Network Architectures.* Morgan Kaufman, 1989.

[15] S. Lennart Johnsson and Ching-Tien Ho. Optimizing tridiagonal solvers for alternating direction methods on Boolean cube multiprocessors. *SIAM J. on Scientific and Statistical Computing.* Technical Report YALEU/DCS/RR-679, Department of Computer Science, Yale University, January 1989.

[16] S. Lennart Johnsson and Ching-Tien Ho. Multiple tridiagonal systems, the alternating direction method, and Boolean cube configured multiprocessors. Technical Report YALEU/DCS/RR-532, Dept. of Computer Science, Yale University, New Haven, CT, June 1987.

[17] S. Lennart Johnsson and Peggy Li. Solutionset for AMA/CS 146. Technical Report 5085:DF:83, California Institute of Technology, May 1983.

[18] Jacob Katsenelson. Computational structure of the n-body problem. Technical Report AI Memo 1042, MIT, Artificial Intelligence Laboratory, April 1988.

[19] E M. Reingold, J Nievergelt, and N Deo. *Combinatorial Algorithms.* Prentice-Hall, Englewood Cliffs. NJ, 1977.

[20] Feng Zhao. An o(n) algorithm for three-dimensional n-body simulations. Technical Report AI Memo 995, MIT, Artificial Intelligence Laboratory, October 1987.