6

THE MOMENT METHOD SOLUTION OF ELECTROMAGNETIC SCATTERING PROBLEMS ON MIMD AND SIMD HYPERCUBE SUPERCOMPUTERS

S. D. Gedney, A. F. Peterson, and R. Mittra

6.1 Introduction

Following the taxonomy of Flynn [1], one can describe two classes of hypercube multiprocessors, namely, Multiple Instruction Multiple Data (MIMD) hypercubes and Single Instruction Multiple Data (SIMD) hypercubes. MIMD hypercubes have processing nodes that can perform completely independent operations on data stored in their local memories. On the other hand, SIMD hypercubes are comprised of a large number of processing nodes that perform identical operations concurrently on local data. Due to the different modes of operation of MIMD and SIMD hypercubes, the methodology of developing parallel algorithms for each is much different.

Both SIMD and MIMD hypercubes can be further subclassed by their granularity [2], which refers to the number of processors employed. A coarse-grained hypercube employs a moderate number of processors (≤ 128), while a fine-grained hypercube employs a very large number of processors (≥ 1024).

In this paper, parallel algorithms performing the method of moments (MoM) solution of electromagnetic scattering problems are developed for both MIMD and SIMD hypercube architectures. It is shown that by exploiting the specific attributes of each architecture, extremely efficient parallel MoM algorithms can be developed. These attributes not only include the granularity of the hypercube, and its SIMD or MIMD operation, but also the architecture of the individual processing nodes.

In Section 6.3, the general MoM algorithm is discussed. The MoM algorithm is broken up into the *matrix fill* and the *matrix solve* subtasks. The former refers to the discretization of the integro-differential operator into a system of linear equations, and the latter is the direct method of solution of this system of linear equations. The potential for parallelizing each of these subtasks is discussed.

In subsequent sections, parallel MoM algorithms are presented for the coarse-grained MIMD hypercube (Section 6.4), the fine-grained MIMD hypercube (Section 6.5) and the fine-grained SIMD hypercube (Section 6.6). The matrix fill task is a highly parallel process, and is performed with high parallel efficiencies in all three hypercube environments. The mapping of the matrix fill algorithm is dependent on the choice of the mapping of the parallel matrix solution algorithm. Due to the requirements of the interaction and synchronization of processors, the parallel algorithms performing the direct methods of solution are

less efficient than the matrix fill. However, with the choice of an appropriate mapping, this serialism can be greatly reduced. New mapping schemes are presented for the fine-grained MIMD and SIMD environments, and a mapping scheme based on existing work is discussed for the coarse-grained MIMD hypercube. High parallel efficiencies are illustrated for each of the architectures being considered.

To illustrate the development of the MoM problem in each of the hypercube environments, two specific examples are presented. First, the electromagnetic scattering by a conducting body of revolution (BOR) is analyzed using the JPL Mark III coarse-grained hypercube. It is seen that since this problem requires the filling and solution of a series of smaller matrices, the coarse-grained hypercube provides an ideal environment for this parallel algorithm. The second example employs the fine-grained SIMD Connection Machine (CM2), to treat the scattering of an electromagnetic wave by a three-dimensional flat conducting plate. Both of these algorithms are shown to have supercomputer performance on the Mark III and the CM2.

6.2 Algorithm Parallelism

a. Parallel Algorithms

Conventional algorithms that have been optimized for a single-processor computer, or a vector-pipelined computer, are often taken for granted since they are readily available as subroutine packages. However, these algorithms may perform poorly in the parallel environment of a hypercube. To achieve superior performance, new algorithms must be developed that better conform to the hypercube architecture. Generally, the user is also free to modify the configuration of the hypercube. For example, a 5-dimensional hypercube, which has 32 processors, can be configured as a two dimensional grid of processors using a binary Gray-coding scheme. In practice, it may be necessary to conform the topology of the hypercube to be compatible with a particular parallel algorithm. In this section a methodology for designing parallel algorithms for the hypercube multiprocessors is discussed.

Any algorithm can be separated into individual subtasks, or *processes.* Similarly, the hypercube consists of a number of independent *processors.* In general, the implementation of a parallel algorithm requires the mapping of N processes onto P processors. The final goal

is to choose a mapping scheme that best exploits the hypercube's architecture and minimizes the computational time required to complete the given task.

If N equally weighted processes are independent, requiring little or no interaction with one another, and evenly distributed to all P processors, the mapping algorithm is said to have a high degree of parallelism associated with it. In other words, the P processors can perform the N processes in the same time that one processor can perform N/P processes. This parallelism will degrade, however, if the processes mapped to different processors are dependent upon one another and require interaction. This interaction takes place in the form of interprocessor communication, i.e. the exchanging of information between processing nodes via the hypercube's communication network. A large amount of interprocessor communication can result in an inefficient algorithm.

Often, output data from one process is required as the input data of another. In a parallel environment, if the two processes are assigned to different processors, their synchronization may result in one processor remaining in an idle state while waiting for the other's data. Consequently, a global task may be bottlenecked when a single processor, or a small number of processors, is executing a process vital to the others. To circumvent this inefficiency, a programmer may: (1) instruct all processors to perform the same task, introducing redundancy, or (2) optimize the synchronization of processes.

In general, any dependency between processes will deteriorate the performance of an algorithm. The term *serial computation* will be used to denote tasks that would be best performed by a single processor, avoiding dependencies between processes residing on different processors. Thus, any process that is redundant, or requires interprocessor communication, synchronization, and memory conflicts will be considered serial computation. On the other hand, the term *parallel computation* will be used to describe processes that are totally independent of one another, and can be performed completely in parallel. An ideal algorithm for multiprocessor implementation will have a high ratio of parallel to serial computation.

b. Measuring Parallelism

The degree to which an algorithm can exploit a multiprocessor is often measured in terms of its *speedup*. There are two types of speedups

described in the literature: *fixed* and *scaled* [3]. Fixed speedup is the ratio of the time a single processor takes to compute a fixed size problem to the time P processors require to compute the same problem. Let p be the time spent on the parallel portion of the program s be the time spent on the serial portion, and P be the total number of processors. The fixed speedup can then be written as

$$\text{Speedup} = \frac{(s+p)}{(s+p/P)} \tag{1}$$

where s is assumed to be a constant. This definition of speedup (also known as Amdahl's law) states that the maximum speedup over a single processor is P, which occurs when $s = 0$. There are limitations to measuring speedup in this manner. For instance, this definition assumes that s is constant. However, for many algorithms, as P grows, the amount of serialism also grows. In addition, consider an algorithm with a moderate amount of serialism. As the number of processors becomes larger, p/P quickly becomes the same order as s, resulting in a rapid decay in the fixed speedup.

Instead of studying the speedup of a fixed size problem, it may be more instructive to look at the speedup of a problem that is scaled to the number of processors. *Scaled* speedup is defined as "how long a given parallel program would have taken to run on a serial processor [3]." Scaled speedup differs from fixed speedup in that the uniprocessor program speed assumes a single hypothetical processor that has direct access to all of the RAM of the machine. Let s' and p' represent the serial portion and parallel portion of the program, respectively, when running on P processors. The hypothetical uniprocessor requires a time of $s' + p'P$ to perform the task. If s' is considered to be constant with respect to the number of processors, scaled speedup is expressed as [3]

$$\text{Speedup} = \frac{(s' + p'P)}{(s' + p')} \tag{2}$$

The advantage of looking at parallelism from this point of view is that the problem size can be chosen such that it uses the maximum capacity of each processor. Large speedups result from the fact that the ratio of p/s is typically very large. For example, the implementation of a large problem on a hypercube multiprocessor often results in a large computation to communication ratio, improving the speedup.

Parallel *efficiency* refers to the percentage of speedup achieved with P processors, and

$$E_P = \frac{\text{Speedup}}{P} \times 100\% \tag{3}$$

If there is no serial computation, the algorithm has 100% parallel efficiency.

6.3 The Method of Moments

a. Formulation of Electromagnetic Scattering Problems

Exterior electromagnetic scattering problems are often posed in terms of integro-differential equations such as the electric-field or magnetic-field equations, denoted EFIE or MFIE, respectively. In this section, we discuss the numerical solution of these equations with emphasis on the individual tasks involved in that process and the manner in which they may be implemented on serial or parallel machines.

The EFIE and MFIE have a similar form, and we restrict our attention to the general form of the surface EFIE representing an impedance scatterer [4]

$$\begin{aligned}\hat{n} \times \overline{E}^{\text{inc}}(\overline{r}) &= \mathcal{L}\{\overline{J}\} \\ &= \hat{n} \times \overline{J}(\overline{r}) Z_s(\overline{r}) - \hat{n} \times \frac{\nabla\nabla \cdot + k^2}{j\omega\varepsilon} \int \overline{J}(\overline{r}') K(\overline{r} - \overline{r}')\, d\overline{r}' \\ &\quad - \hat{n} \times \nabla \times \int \hat{n} \times \overline{J}(\overline{r}) Z_s(\overline{r}) K(\overline{r} - \overline{r}')\, d\overline{r}'\end{aligned} \tag{4}$$

where E^{inc} denotes the known excitation, Z_s denotes a surface impedance, and J represents the unknown surface current density. The kernel K represents the free-space Green's function. Equation (4) can be solved approximately for J by discretizing it into a matrix equation and subsequently solving the matrix equation. This discretization process may take several forms, but all are essentially equivalent to the *method of moments* (MoM) [5]. The process requires J to be replaced by a finite expansion having the form

$$\overline{J}(\overline{r}) \cong \sum_{n=1}^{N} j_n \overline{B}_n(\overline{r}) \tag{5}$$

where $\{j_n\}$ denote scalar coefficients and $\{B_n\}$ vector basis functions. After substitution of (5) into (4), the remaining equation may be converted into a matrix form by forcing the residual to be orthogonal to a set of "testing" functions $\{T_m\}$. Using brackets to denote some suitable inner product between vector quantities, the matrix equation has the form

$$\overline{\overline{L}}\,\overline{j} = \overline{e} \tag{6}$$

where

$$L_{mn} = \langle \overline{T}_m(\overline{r}), \mathcal{L}\{\overline{B}_n(\overline{r})\}\rangle \tag{7}$$

and

$$e_m = \langle \overline{T}_m, \hat{n} \times \overline{E}^{\text{inc}}(\overline{r})\rangle \tag{8}$$

Equation (6) can be solved to produce the unknown coefficients $\{j_n\}$.

The process of using the EFIE to analyze an electromagnetic scatterer proceeds as follows. First, a numerical model of the electromagnetic scatterer under consideration is generated. This model consists of a list of points (x_i, y_i, z_i) describing the surface of the scatterer and several pointer arrays. The arrays link these points with the associated cells or patches in order to provide information on the relative location of patches representing the surface. This model is then utilized to create the matrix equation appearing in (6). The matrix "fill" procedure requires the integrals appearing in (7) and (8) to be evaluated using, in most cases, some form of numerical quadrature. The matrix equation is then solved to produce the coefficients $\{j_n\}$. Typically, Gaussian elimination is used to solve the system. Secondary calculations are performed at this stage in order to compute the far-zone fields or radar cross-section (RCS) of the scatterer. These numerical data are then converted into the form of graphical displays and presented to the user as output data.

The procedure outlined above can be separated into several distinct parts. The *pre-processing* task consists of generating the numerical model of the geometry under consideration. (Often, this task is performed on a graphics-oriented workstation.) The preprocessing stage produces a data file containing the numerical model described above. These data are then available as input to the *central processing* task, which consists of creating and solving the matrix equation from (6) and computing secondary quantities such as the RCS. We elaborate on the central processing task below. The output from the central processing task is raw numerical data providing the equivalent current density

on the scatterer and the far fields. The *post-processing* task consists of converting the numerical data into a form more suitable for later interpretation. Often, post processing is also performed by a different computer from that used for the central computing task (perhaps the same graphics oriented workstation used for preprocessing). Generally, the pre-processing task is very time-consuming in terms of work hours, while the tasks of filling and solving the matrix equation consume the most CPU hours.

For implementation on parallel architectures, we need to examine these tasks in more detail. Our attention will focus on the matrix fill and matrix solve tasks, since these require the most computer time and are most likely to benefit from parallel implementation.

b. The Matrix Fill Procedure

The "matrix fill" task consists of evaluating the integrals appearing in (7) and (8) and organizing this information into the matrix equation (6). In general, the matrix fill process is highly parallelizable, since there is no inherent interaction required between different parts of the matrix. The system can be filled by rows, columns, or any other desirable scheme. (If the system is too large to store in directly addressable computer memory, the order of generating the matrix elements is usually dependent upon the specific storage scheme used with the out-of-core solver.) In the case of parallel implementation, the matrix can be divided into blocks that are assigned to independent processors (the manner of assigning blocks to a processor is usually done in coordination with the specific algorithm used to solve the matrix equation). Each processor will require access to the numerical model describing the part of the scatterer assigned to that processor. In order to balance the workload of each processor, each will be assigned a roughly equal portion of the matrix.

The matrix fill task usually requires a numerical evaluation of the integral appearing in (7). If performed by adaptive quadrature, some of the entries in the matrix may require a more computationally intensive integration than others. (Typically, entries representing closely-spaced cells in the scatterer model require more computation than entries representing cells that are separated by more than one third of the wavelength. In addition, diagonal entries usually require separate treatment because of the presence of a singularity in the kernel of the integral operator, which must be extracted, evaluated analytically, and added to

the numerically computed residual). Although it may be difficult to perfectly balance the matrix fill process among a large number of processors, it should be possible to achieve a nearly balanced distribution of the work.

c. The Matrix Solve Process

Once the matrix elements are computed, the solution for the unknown constant coefficients, or vector $\bar{\bar{j}}$ of (6), must be computed for different excitations. There are a number of methods available to perform the solution of the dense linear system of equations. In specialized problems, iterative techniques can be quite economical in terms of storage and required computational operation. However, iterative techniques suffer when used to treat many right-hand sides. For the purposes of this study, it is of interest to solve systems with a large number of right-hand sides (i.e., scatterers illuminated by incident fields from many spatial directions) and iterative methods will not be considered. Direct methods are preferable under these conditions, because the computationally intensive factorization or inversion need only be performed once. Therefore, the focus of this section will be on direct methods of solution of the linear systems of equations.

The matrix equation arising from the discretized integro-differential equation representing the electromagnetic scattering problem is characterized as being fully populated, complex-valued and non-Hermitian. There are many algorithms suited for solving this type of matrix equation. The first is a direct matrix inversion, which involves the explicit computation of the matrix inverse. On a sequential computer, direct inversion is seldom used because it requires $2N^3 + O(N^2)$ complex arithmetic operations, for an N-th order matrix, as well as N^2 additional complex operations to perform the matrix-vector multiplication required to compute the solution vector. (In the notation used in this paper, one arithmetic add and one arithmetic multiply of two complex floating-point numbers constitute two complex arithmetic operations.) In Section 6 it is illustrated that direct inversion is the preferable method for a fine-grained SIMD hypercube.

Probably the most widely used direct method is LU decomposition by Gaussian elimination, which involves the factorization of the matrix into a product of lower and upper triangular matrices [6]. Once the matrix is factorized, the solution vector is computed by solving the two triangular systems of equations. On a sequential computer, LU de-

composition requires $2N^3/3 + O(N^2)$ complex arithmetic operations. The solution of the triangular systems of equations requires $O(N^2)$ complex arithmetic operations. Because of the efficiency of LU factorization, a large number of user-oriented FORTRAN subroutines are available for sequential and vector computers. One example is the LINPACK routine CGEFA, which performs the factorization of a complex-valued matrix.

Of notable consideration is the error encountered when performing LU factorization by Gaussian elimination [7]. It is widely appreciated that the stability of Gaussian elimination is not guaranteed unless some form of pivoting is employed. Partial pivoting is the exchange of rows in the leading principal submatrix of $\overline{\overline{L}}$ such that the largest element in the column containing the pivot element becomes the new pivot element. This technique helps to reduce the round-off error by stabilizing the condition of the transformed matrix. The inclusion of partial pivoting in the decomposition algorithm presents additional work at each transformation. Provided that the matrix is stored in directly-addressable memory, the total additional time is small compared to the $O(N^3)$ process of the decomposition. Thus, even with partial pivoting the LU factorization algorithm is quite efficient.

Another method of factorization is to introduce a series of orthogonal transformations of the matrix. One method commonly used is the Householder transformation [6, 8], which has matured from QR factorization and the Given's reduction methods. The Householder transformation decomposes the matrix into a triangular matrix. A significant advantage of this method is that the eigenvalues of the matrix are unchanged by the orthogonal transformations. As a result, the condition of the matrix is unchanged, and the Householder transformation is a very stable numerical process. Decomposition by the Householder transformation on a single processor requires $N^3 + O(N^2)$ complex arithmetic operations, compared to $2N^3/3+O(N^2)$ required by LU decomposition by Gaussian elimination.

The following sections consider parallel algorithms that perform the matrix fill and the matrix solution. Parallel algorithms for both of these subtasks are developed for the coarse-grained MIMD hypercube, the fine-grained MIMD hypercube and the fine-grained SIMD hypercube. Due to the distinct characteristics of each hypercube architecture, the algorithms are quite different.

6.4 Parallel Implementation of the Method of Moments Algorithm Using a Coarse-Grained Hypercube (MIMD)

The first architecture considered is the coarse-grained MIMD hypercube. Each processing node of the coarse-grained MIMD hypercube is assumed to be a self-contained computer, with a sophisticated CPU that has significant computing power, a floating-point accelerator, possibly vector-pipeline capabilities, a separate chip to handle communication, as well as a large amount of local memory. Therefore, the speed of the coarse-grained MIMD hypercube is obtained by combining a few, powerful computers operating concurrently with the ability to communicate over a network with a hypercube topology. Examples of coarse-grained hypercubes include the JPL Mark III or the Intel iPSC/860 (each employs a maximum of 128 nodes). The following discusses the development of the parallel MoM algorithm for this multiprocessor environment.

a. The Matrix Fill Task

The task of computing the impedance matrix is a highly parallelizable process since each element can be evaluated independently. Therefore, a single process of the matrix fill algorithm consists of computing a single matrix element. Given a matrix of rank N, the matrix fill algorithm consists of mapping the N^2 processes onto the P processors of the hypercube. If roughly N^2/P elements are mapped onto each processor the work load is balanced and the N^2 elements can be computed in the time required by a single processor to compute N^2/P elements.

Two common techniques of mapping the matrix elements onto the hypercube are *block mapping* and *wrap mapping*. Block mapping methods distribute equally sized contiguous blocks of elements to each processor. One example of a block mapping is to assign the first N/P columns to processor 0, then assign the next N/P columns to processor 1, and so on. In contrast, wrap-mapping methods cyclically assign columns (or rows) to each processor. For example, when wrap mapping by columns, columns $1, 1+P, 1+2P, \ldots$ are assigned to processor 0, columns $2, 2+P, 2+2P, \ldots$ are assigned to processor 1, and so on.

Since the partitioning of the matrix elements is quite flexible, the choice of the mapping scheme will depend on the algorithm chosen

to perform the matrix solution. It will also depend on the amount of memory available to each processor, since both the matrix elements as well as the geometry data of the testing and basis functions required to construct the elements must be housed by the processor's local memory. For example, in order to construct a complete column (or row), the geometry data of the entire set of testing functions (or basis functions) must reside in the local memory of the processor. In the following, it is assumed that each processing node of the coarse-grained MIMD hypercube employs a sufficient amount of memory to store entire columns (or rows) of the matrix, and the entire scatterer model.

b. Direct Method of Solution

Because direct methods require $O(N^3)$ operations, as compared to the $O(N^2)$ processes of the matrix fill algorithm, the matrix solution can potentially require the largest percentage of total CPU time. Therefore, it is important to optimize the parallelism of the solution algorithm. The difficulty that arises in the parallel environment is that the direct method of solution requires a large amount of interaction and possibly synchronization between processors.

(1) LU factorization

The solution of linear systems of equations via direct methods on coarse-grained hypercubes has been a recent topic of interest [9–12]. In this multiprocessor environment, it has been found that the most efficient direct method of solution is LU factorization by Gaussian elimination. It outperforms the Householder transformation, even with the inclusion of partial pivoting [12]. Ortega and Romine [11] investigated a number of different permutations of the ijk forms of factorization by Gaussian elimination. It was demonstrated that the kji, using Column-Storage by wrap mapping and Row-Pivoting (CSRP), and the kij, using Row-Storage by wrap mapping and Row-Pivoting (RSRP), algorithms were the optimal algorithms for performing the parallel LU factorization on a coarse-grained MIMD hypercube.

The kernels of the kji/kij factorization algorithms consist of a series of $N-1$ transformations of a square submatrix $\overline{\overline{A}}_k$ of order $N-k$, during the k-th transformation (assume the order of the original

matrix to be N). Each transformation is given by the operation

$$\overline{\overline{A}}_k = \overline{\overline{A}}_k - \{l_{ik}\}\{b_{kj}\}/a_{kk} \tag{9}$$

where $\{l_{ik}\}$ is the pivot column vector, $\{b_{kj}\}$ is the pivot row vector, and a_{kk} is the pivot element. By wrap mapping the matrix by columns (or rows), it is guaranteed that the submatrix $\overline{\overline{A}}_k$ is evenly distributed among all processors during each of the $N-1$ transformations. The transformation is then performed in parallel requiring only a single broadcast of the current pivot column (or row), of length $N-k$, to all processors providing them with the necessary information to compute the matrix update.

The efficiencies of the CSRP, based on wrap mapping the matrix by columns, and the RSRP, based on wrap mapping by rows, algorithms (using kij and the kji, respectively) only differ in their application of pivoting. Partial pivoting requires the search for the largest element in the current pivot column vector $\{l_{ik}\}$. In the RSRP algorithm, the pivot column is wrap mapped about all of the processors. Therefore, the maximum pivot element can be located in parallel, first locally in each processor, then globally by comparing the maximum elements of all processors as they are communicated to a single processor along a binary tree (a process known as a *fan in*). This requires $\log_2(P)$ interprocessor communications. Then the maximum element is disseminated from the single processor to all processors (a process known as a *fan out,* or a broadcast), requiring an additional $\log_2(P)$ interprocessor communications. Once the pivot element is found, the row associated with the largest element becomes the new pivot row. This row can then either be physically exchanged with the old, or simply renumbered. The exchanging of rows will require additional communication between two processors. However, if the pivot rows are not physically exchanged, but merely renumbered, a single processor can end up with a disproportionate number of rows of the leading order submatrix. This can result in a load imbalance that severely degrades the performance of the algorithm. It turns out that the preservation of load balance at the expense of added communication is actually a more computationally efficient method [12]. This technique of preserving load balance is known as *dynamic load balancing* [12].

On the other hand, the CSRP algorithm requires a search for the pivot element along the pivot column which is stored on a single processor. Therefore, no interprocessor communication is required, how-

ever, the linear search is purely serial and degrades the efficiency of the algorithm. Unlike the RSRP algorithm, the physical exchange of rows is not necessary to preserve load balance, reducing interprocessor communication.

It was reported in [12] that both the RSRP and the CSRP algorithms will have nearly linear speedups when $N \gg P$. However, the actual performance of either of these algorithms is dependent upon the architecture. If vector pipelining is available in the processing node (the Intel iPSC2 offers vectorization), the linear search required by the CSRP algorithm is extremely efficient. Otherwise, the performance of each is determined by the speed of communication and floating point operation of the specific machine.

(2) The solution of the triangular system of equations

Once the matrix is decomposed into lower and upper-triangular matrices, the solution vector can be computed for any number of right-hand sides with the solution of two triangular systems of equations. This process is often called *forward/backward* substitution, referring to the recursive algorithm employed on a single processor computer. The solution of the triangular system of equations requires $(N^2 + N)/2$ complex-arithmetic operations, as compared to the $2N^3/3 + 2N^2$ required by the LU-factorization algorithm. Since the number of operations is $O(N)$ less, the solutions of the triangular systems are often overlooked.

In the parallel environment, the solution of the triangular systems of equations, like the LU-factorization algorithm, requires a significant amount of interprocessor communication. However, the ratio of computation to communication is much less since the total number of arithmetic operations is $O(N)$ smaller. This results in a less efficient parallel algorithm. Furthermore, due to the recursive nature of the forward/backward substitution, the potential exists for a large amount of serialism.

Heath and Romine [13] provide a comparison of a number of parallel algorithms that perform the solution of triangular systems of equations on the coarse-grained MIMD hypercube. The most promising approach appears to be their wavefront algorithm, which employs a *compute ahead* type of strategy that reduces serialism. They have evaluated the wavefront algorithm for both row-storage and column-storage schemes.

The wavefront algorithm can be implemented using either of two sequential algorithms discussed in [12]. These two algorithms, illustrated below, are known as the vector-sum algorithm and the scalar-product algorithm, respectively.

Vector-Sum Algorithm

```
for j = 1 to n
    x_j = b_j / L_jj
    for i = j + 1 to n
        b_i = b_i - x_j L_ij
```

Scalar-Product Algorithm

```
for i = 1 to n
    for j = 1 to i - 1
        b_i = b_i - x_j L_ij
    x_i = b_i / L_ii
```

Assume that the lower triangular matrix L is wrap mapped by columns onto the coarse-grained hypercube. The vector-sum algorithm requires the update of the column vector b_i on the single processor which contains column L_{ij}. Once b_i is updated, it is passed on to the next processor, which subsequently computes x_{j+1} and again updates b_i with column L_{ij+1}. The inherent serialism in this approach is that each processor must wait for b_i to be updated by the previous processor before it can continue with its work. The wavefront algorithm reduces this inherent serialism by employing a compute-ahead strategy which allows the next processor to begin its computation of x_j and b_i while the previous processors are still computing their updates of b_i. This is done as follows. The vector b_i is initially broken up into segments of length σ. The first processor will compute x_1, and then σ elements of b_i. It then communicates the σ elements of b_i to the next processor (which holds column 2 of L). The next processor then computes x_2 and updates σ elements of b_i. Meanwhile, the first processor continues to update b_i by computing subsequent blocks of σ elements and communicating each block to the second processor when finished. This strategy improves parallelism in two ways. Initially, it becomes possible to keep all processors busy using this algorithm, improving load balance. Secondly, using the wrap-mapping scheme, interprocessor communication is confined to neighboring processors only.

The amount of work being performed concurrently by all processors, implementing the vector-sum algorithm, is dependent upon the segment size σ. The segment size is an adjustable parameter and, in fact, an optimal value can be determined. At one extreme, it can be seen that if σ is chosen to be very small, the algorithm will require a much larger number of communications. This can be expensive, since the startup time associated with each communication is significant. If

σ is chosen to be quite large, fewer communications will be required, however there will be a reduction in concurrency. In fact if $\sigma = n$, the algorithm becomes purely serial. The optimal value of σ can be found experimentally, or theoretically, for a specific coarse-grained hypercube [13].

The row-oriented wavefront algorithm is performed in a similar fashion, although it is based on the scalar-product algorithm [13]. In this case, L is assumed to be wrap mapped by rows, x is assumed to be a row vector, and b is now wrap mapped among all processors. The row-oriented algorithm proceeds as follows. Initially, the first processor computes x_1 and passes it on to the second processor (which holds row 2 of L), which computes x_2. This processor sends x_1 and x_2 to the third processor. This continues until σ elements of x are computed by σ processors. This segment of x can then be broadcast to all processors, with can then start updating their b_i while the next σ elements of x are computed. Again σ is an adjustable parameter and an optimal value can be chosen.

The main difference between the row-oriented and the column-oriented wavefront algorithms is that once the segment of x is computed using the row-oriented algorithm, it can be passed on to *all* processors, whereas, in the column-oriented algorithm, each segment of b_i must be continuously updated by each processor before being passed on to the next. As a result, the segment sizes for the row-oriented algorithm can be slightly larger since a large segment size reduces communication cost without sacrificing concurrency. For example, it was found in [13] that when solving a triangular system of order 1000 on a 64-processor NCUBE hypercube, the optimal segment size for the row-oriented algorithm was 13, compared to 9 for the column oriented algorithm. Similarly, on a 64-processor Intel iPSC hypercube, the optimal segment sizes were 20 and 11.

Heath and Romine [13] illustrate theoretical benchmarks for their wavefront algorithm on both a 64-node NCUBE and a 64-node Intel iPSC hypercube. Without the aid of vector pipelining or loop-unrolling, the row-oriented wavefront algorithm performs slightly faster on both hypercubes.

c. Example: Scattering by a Body of Revolution

In this section, a parallel algorithm is presented which performs the analysis of electromagnetic scattering by a structure characterized as a

Body of Revolution. The algorithm was implemented on the Jet Propulsion Laboratory Mark III. The computational times are presented and are compared to an optimal algorithm developed for a single processor Cray X-MP.

A Body of Revolution (BOR) is a three-dimensional object that is rotationally symmetric about a single axis and can be defined as a surface that is generated by rotating a planar curve, known as the generating arc, about an axis of symmetry. Because of the axial symmetry, the original three-dimensional scattering problem can be reduced to a series of decoupled two-dimensional problems. This is accomplished by expanding all relevant quantities as a Fourier series of azimuthal harmonics. As a result, the amount of computational time and computer storage required to solve the problem is greatly reduced.

There have been a number of studies of the electromagnetic scattering by a BOR [14–20]. The parallel MoM algorithm presented in this section is based on the work presented by Gedney and Mittra [20] which was derived from original work done by Glisson and Wilton [17]. The work in [20] focuses on optimizing the computational efficiency on a sequential computer. Here, the emphasis is on developing an efficient parallel algorithm for the hypercube.

Initially, assume the BOR to be a perfectly conducting scatterer. The boundary value problem can initially be described by the electric-field integral equation (identical to (4) with a zero surface impedance). Following the MoM formulation described by Glisson and Wilton [17], the discretized EFIE becomes a superposition of matrices representing decoupled azimuthal harmonics.

$$\sum_{m=-M}^{+M} \begin{pmatrix} E_{tm}^{n^{\text{inc}}} \\ E_{\phi m}^{n^{\text{inc}}} \end{pmatrix} = \sum_{n=-M}^{+M} \begin{bmatrix} \beta_{11_m}^{nq} & \beta_{12_m}^{nq} \\ \beta_{21_m}^{nq} & \beta_{22_m}^{nq} \end{bmatrix} \begin{pmatrix} J_{tm}^{q} \\ J_{\phi m}^{q} \end{pmatrix} \tag{10}$$

where J_{tm}^{q} and $J_{\phi m}^{q}$ are the unknown coefficients of the tangential and azimuthal components of the equivalent electric-current density (corresponding to the m-th azimuthal harmonic) distributed along the generating arc. The total number of azimuthal harmonics required to represent the current is $2M + 1$, where M can be predetermined by the number of terms needed to represent the incident field on the scatterer surface. Due to symmetry about $m = 0$, only $M + 1$ linear systems of equations need be computed.

The matrix elements in submatrices $\beta_{11_m}^{nq}, \beta_{12_m}^{nq}, \beta_{21_m}^{nq}$, and $\beta_{22_m}^{nq}$ involve the computation of a double integration, with independent vari-

ables t and ϕ. (Expressions for the matrix elements are provided explicitly in the Appendix of [20].) On a sequential computer, the task of filling the matrix ordinarily requires 95% of the total computational time. However, the method presented in [20] greatly enhances the computational efficiency of the matrix fill algorithm. The Fast Fourier Transform (FFT) is used to compute the ϕ-integration, and as a result, the $M+1$ matrices need not be computed independently. Rather, the FFT provides enough information to readily compute all $M+1$ matrices simultaneously. Furthermore, using the singularity extraction method presented in [20], which also takes advantage of the FFT, the self-terms of the $M+1$ matrices are also computed simultaneously. Constructing the matrices in this manner greatly reduces the total computational time, at the expense of requiring additional memory to store all the matrices. It was shown in [20] that at least an order of magnitude in speedup can be achieved using this technique on a sequential computer.

By taking advantage of multiple processors and the FFT, an efficient matrix fill algorithm has been developed for the Mark III hypercube. An efficient mapping would involve wrap mapping the $M+1$ matrices by columns onto the hypercube in a manner that the first column of all $M+1$ matrices lies in processor 0. The elements in the columns are then computed with the use of the FFT using the above technique. The FFTs were performed independently on each processor using a sequential algorithm [21]. With the use of the FFT and the singularity extraction a dual sense of parallelism is achieved. Namely, the matrix columns are being computed in parallel on the independent processors. At the same time, within each processor the use of a single FFT provides information for the computation of an element in all $M+1$ matrices. This dual parallelism greatly enhances the efficiency of the algorithm.

One consideration of the sequential algorithm discussed in [20] was that the total number of double integrations performed numerically can be reduced due to their recurrence in other elements. In the parallel algorithm, redundancy can be avoided by sharing the computed values of recurring integrals between elements residing in the same processor. If the matrices are mapped by columns, then elements residing in the same column can easily share recurring integrations. However, information shared between row elements would require interprocessor communication and synchronization. By altering the mapping scheme,

it would be possible to exploit the recurrence of a large number of integrals between row elements arising from the overlapping support of J_t and J_ϕ without interprocessor communication. To this end, the first N columns of the $M+1$ matrices, computed using the basis functions representing J_t, are wrapped mapped by columns onto the hypercube, as was done by the original mapping. The remaining $N+1$ columns, computed using the basis functions representing J_ϕ, are wrap mapped by columns onto the hypercube, such that column $N+1$ is also located on processor 0. With this mapping the number of double integrations performed can be reduced by almost 25 percent, since the proper processors share matrix elements with recurring integrals. Any other recurring integrals in elements not shared by the same processors are not communicated, but are performed as redundant computation. Although this mapping scheme eliminates a significant amount of redundancy, it requires a reshuffling of the matrix before the LU factorization is performed. (This proved to be of little cost when compared to the time saved by eliminating the integrations.) By taking advantage of recurring integrals between row and column elements, the overall computational time was reduced by at least a factor of two.

Once the $M+1$ matrices are filled, they are each factorized separately using the CSRP algorithm discussed in Section 6.4(b).

The above algorithm was implemented on the JPL Mark III hypercube, which is a MIMD coarse-grained hypercube with a maximum of 128 processing nodes (of which 32 were available). Each processing node consisted of two 68020 processors (one dedicated to communication), a Weitek 64-bit floating-point accelerator, and four Mbytes of local memory. With the Weitek floating-point accelerator, the performance of a single node was found to be in the range of 2 to 3 MFLOPS when executing a FORTRAN program.

In Table 6.1, the benchmarked results of the parallel BOR algorithm are compared to the benchmarked times recorded on a single processor CRAY X-MP using the algorithms presented in [20] and [19] (the source code was programmed completely in FORTRAN for all three implementations). The example employed is the electromagnetic scattering from a closed, perfectly conducting cylinder with a radius of 1λ and a height of 3λ. The optimal sequential algorithm uses the FFT to compute the ϕ-integrations, computing all $M+1$ matrices simultaneously [20]. The second sequential algorithm [19] is based on the Glisson and Wilton formulation computing each matrix indepen-

$2N+1$	M	P	Mark III		Cray X-MP [20]	Cray X-MP [19]
			Matrix Fill (CPU sec)	Matrix Fact (CPU sec)	Matrix Fill (CPU-sec)	Matrix Fill (CPU-sec)
93	9	1	—	—	10.	120.
		2	78.21	5.5		
		4	39.9	4.0		
		8	20.0	2.7		
		16	10.0	2.1		
		32	6.67	1.7		

Table 6.1 Benchmarked times: Scattering by a closed PEC cylinder (1λ radius, 3λ height).

dently and using adaptive numerical integration. The benchmarked matrix-fill time refers to the total CPU time required to compute all the $M+1$ matrices. As should be expected, the matrix fill algorithm on the Mark III has a nearly linear fixed speedup. However, some degradation in efficiency is witnessed as the number of processors becomes large. This is due to the additional communication encountered when reshuffling the matrix elements, which becomes more significant as the computational time decreases. The time required to factorize all of the $M+1$ matrices on the Mark III is also recorded in Table 6.1. The parallel efficiency of this algorithm is much less than the matrix fill algorithm, as was expected. On the CRAY X-MP, the factorization of each matrix was performed in hundredths of a second using the LINPACK subroutine CGEFA.

As a second example, consider the scattering of a closed PEC cylinder with a 4λ height and a 3λ radius. The benchmarked times required to compute the solution on the JPL Mark III hypercube when the cylinder is illuminated by a plane wave incident at $\theta = 45°$ are recorded in Table 6.2 Again, the times recorded are the fill and factorization of all $M+1$ matrices. A total of 18 modes (i.e., $M = 17$) and 193 basis functions along the generating arc were required for proper numerical convergence for this problem. A 64-point FFT was used to evaluate the ϕ-integration. Since the number of unknowns and the number of modes is roughly doubled, the matrix fill task has increased

$2N+1$	M	P	Mark III	
			Matrix Fill (CPU sec)	Matrix Fact (CPU sec)
193	17	1	—	—
		2	—	—
		4	304.0	44.2
		8	155.2	25.5
		16	80.6	17.0
		32	44.3	12.9

Table 6.2 Benchmarked times: Scattering by a closed PEC cylinder (3λ radius, 4λ height).

by a factor of 8 as compared to the problem in Table 6.1. However, for this larger problem, near linear speedups are observed for the matrix fill task as the number of processors are increased for the fixed problem size, and little degradation due to interprocessor communication is observed for this case, even with 32 processors.

6.5 Parallel Implementation of the Method of Moments Algorithm Using a Fine-Grained Hypercube (MIMD)

The preceding section discussed the solution of the MoM problem on coarse-grained MIMD hypercubes. The algorithms presented assumed that complete rows or columns of the matrix and the complete numerical model of the scatterer could be stored on a single processing node. High parallel efficiencies can be achieved with the parallel factorization algorithms when $N \ll P$, even with the implementation of pivoting. However, if the multiprocessor is much more fine grained, such that $N \sim O(P)$, or $N \gg P$, one can not take full advantage of the available parallelism using these algorithms. Furthermore, due to the large number of processors, it is expected that the size of the local memory will be reduced, and it may not be desirable to store the entire model of the scatterer on each processor, especially for very large values of N. An example of a fine-grained MIMD hypercube is the NCUBE hypercube, which may employ up to 8192 processors.

Algorithms tailored to such massive parallelism are the focus of this section.

In common with coarse-grained hypercubes, efficient algorithms for the matrix fill also require an even distribution of the computational effort, as discussed in Sections 6.3.b and 6.4.a. By assigning N^2/P elements to each processor, a nearly linear speedup can be achieved. In fact, the matrix can be filled in roughly the time required to compute N^2/P elements on a single processor. When P is very large, on the order of thousands, this results in a significant time savings. The manner by which the N^2 matrix elements are mapped onto the hypercube will depend upon the mapping scheme required by the matrix solution algorithm.

a. LU-Factorization and Solution Algorithms

The RSRP and CSRP algorithms discussed in Section 6.4.b(1) assumed that $N \gg P$ and that multiple rows or columns can be stored on a single processor. However, in a fine-grained parallel environment one can no longer even assume that $N > P$. Nor can it be assumed that there is sufficient memory to store complete columns, rows, or the entire scatterer model on a single processing node. This type of an environment poses difficulty to the efficient use of the RSRP or the CSRP algorithms. Therefore, a new LU-factorization algorithm that exploits the fine-grained MIMD hypercube is developed.

Initially, the n-dimensional hypercube is configured as a two-dimensional grid. The grid is defined with P_x processors in the x-direction and P_y processors in the y-direction (where $P_x \cdot P_y = P$, and both P_x and P_y are integral powers of 2). The processors are ordered using a binary reflected Gray-coded scheme. The matrix is mapped onto the grid of processors in a manner such that the submatrix $\overline{\overline{A}}_k$ is distributed across all processors while $n - k + 1 > P_x$ and P_y. This is accomplished using a technique that is known as *two-dimensional wrap mapping* [22], in which the matrix is wrap mapped onto the two-dimensional grid of processors in two directions. Hence,

processor $(0,0)$ will receive elements

$$\begin{matrix} a_{1,1} & a_{1,1+P_x} & a_{1,1+2P_x} & \cdots \\ a_{1+P_y,1} & a_{1+P_y,1+P_x} & a_{1+P_y,1+2P_x} & \cdots \\ a_{1+2P_y,1} & a_{1+2P_y,1+P_x} & a_{1+2P_y,1+2P_x} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{matrix}$$

and so on.

An important characteristic of the two-dimensional wrap mapping is that any column of the matrix is wrap mapped about a single column of the two-dimensional grid of processors. Similarly, any row of the matrix is wrap mapped onto a single row of the grid of processors. It can also be said that any row or column of the matrix is distributed about a single row or column of the grid of processors, respectively. This results in a minimization of the communication time required to perform both the pivot search and the broadcasting of the pivot column and row. The two-dimensional wrap mapping also ensures that the leading-order submatrix is distributed among all processors while $(N-k+1) > P_x$ and P_y, thus enforcing load balance.

With the use of two-dimensional wrap mapping, the kij and kji forms of factorization are the optimal algorithms and are in fact interchangeable. Figure 6.1 illustrates the parallel kij factorization algorithm written in pseudo code. Partial pivoting is presented in this algorithm in order to treat general matrices.

Initially, for each transformation, the pivot element is determined using a *fan in* [13] within the column of processors containing the pivot column. Once the pivot element is known, it is broadcast to the column of processors such that the pivot column can be normalized.

In order to ensure load balancing, the new pivot row is physically exchanged with the original row. Geist and Romine [12] showed that on a medium-grained distributed-memory parallel processor, the time loss due to the load imbalance created by partial pivoting without exchanging rows outweighs the additional communication time required to physically exchange rows. This will also hold true in a fine-grained parallel environment because great inefficiencies occur if a large number of processors become idle (full rows of processors) while others accrue additional work.

Once the pivot row is exchanged, the matrix update is performed and is expressed as

$$\overline{\overline{A}}_k = \overline{\overline{A}}_k - l_{ik} b_{kj} \tag{11}$$

parallel (kji)

```
for k = 1, N − 1
    if (k ∈ mycols)
      find a_kmax
      copy (a_kmax to grid col.)
      for (s ∈ myrows > k)
          l_sk = l_sk / a_kmax

if (k ≠ kmax)
  if (kmax ∈ myrows)
    for (j ∈ mycols)
        copy (b_kmax,j to grid col.)
  if (k ∈ myrows)
    for (j ∈ mycols)
        copy (b_k,j to row kmax; replace b_kmax,j)
else
if (k ∈ myrows)
  for (j ∈ mycols) > k)
      copy (b_kj to grid col.)
if (k ∈ mycols)
  for (i ∈ myrows > k)
      copy (l_ik to grid row)
for (j ∈ mycols > k)
    for (i ∈ myrows > k)
    a_ij = a_ij − l_ik b_kj
```

Figure 6.1 Parallel algorithm to perform the LU factorization of a matrix of order N. Partial pivoting is included in order to treat general matrices.

In the kij/kji form, the submatrix $\overline{\overline{A}}_k$ is an $(N-k)\times(N-k)$ matrix. The product of the column vector l_{ik} and the row vector b_{kj} is also an $(N-k)\times(N-k)$ matrix and the matrix update is essentially the difference between two matrices. To compute the necessary blocks of the matrix $(l_{ik}b_{kj})$ for each processor, the vectors l_{ik} and b_{kj} must be

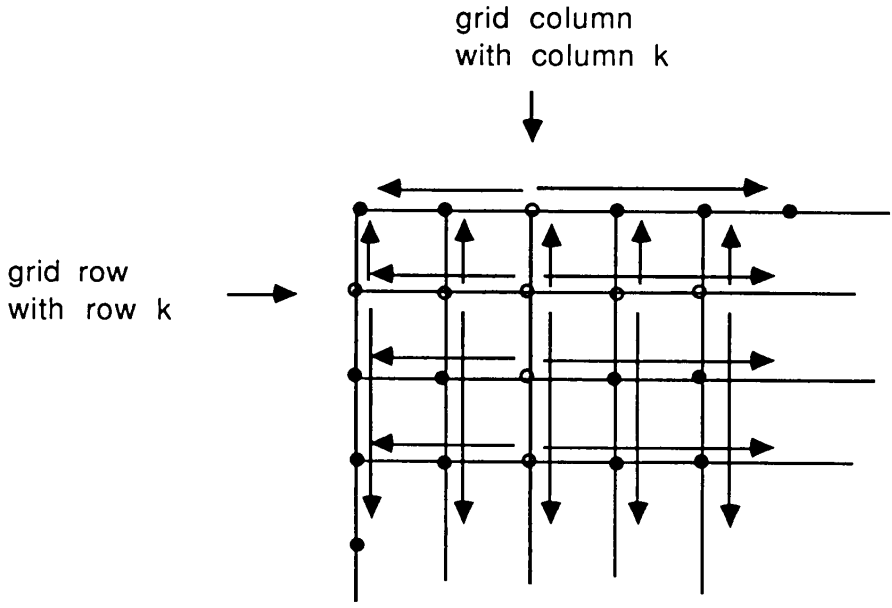

Figure 6.2 Broadcasting b_{kj} (pivot row) vertically to the processor grid, and l_{jk} (pivot column) horizontally to the processor grid.

broadcast to the processor grid. To this end, the processors that hold the column vector l_{ik} only broadcast their elements to all processors lying within the same row of the grid of processors; the processors that hold the row vector b_{kj} only broadcast their elements to all processors lying within the same grid column (see Fig. 6.2).

Let n'_x be the number of elements of b_{kj} currently stored on each processor. The broadcast of b_{kj} involves the parallel communication of n'_x elements to P_y processors. In a hypercube environment, this requires $\log_2(P_y)$ steps. Similarly, broadcasting l_{ik} involves the parallel communication of n'_y elements (the current number of elements of l_{ik} stored in each processor) to P_x processors. After these two broadcasts are performed, each processor has all of the necessary information to perform the matrix update.

The communication time required to broadcast l_{ik} and b_{kj} for all $n-1$ transformations can be expressed as

$$\sum_{k=1}^{n_x}\sum_{j=1}^{P_x}(S+\alpha(n_x-k+1))\Delta p_x|_{\text{while: } P_x(k-1)+j<n} + \sum_{k=1}^{n_y}\sum_{j=1}^{P_y}(S+\alpha(n_y-k+1))\Delta p_y|_{\text{while: } P_y(k-1)+j<n} \tag{12}$$

where $n_x = \text{int}\,|N/P_x|$ and $n_y = \text{int}\,|N/P_y|$ (both are rounded to the next highest integer). S is the startup time per communication, Δp_y and Δp_x are the largest distances between processors in the x- and y-directions ($\log_2(P_x)$ and $\log_2(P_x)$ respectively), and α is the interval time required to communicate a single complex number. The additional communication due to pivoting (including pivot search and row exchange) is given by

$$2(n-1)[S+\alpha]\Delta p_y + \sum_{k=1}^{n_x}\sum_{j=1}^{p_x}\varepsilon_p(k,j)[S+\alpha(n_x+k-1)]\Delta p_y \tag{13}$$

where $\varepsilon_p(k,j) = 1$ if an exchange of rows is necessary during the $j+(k-1)n_x$ transformation. If an exchange of rows is not necessary, then $\varepsilon_p(k,j) = 0$. This strategy assumes the exchange of a full row.

We are interested in the computational time required by this algorithm. Initially, let $2F$ be the time to perform one add and one multiply operation. If we assume that there are a total of 2^M processors, where M is an integer, then we will assume that $P_y/P_x = 2^m$ (m is an integer ≥ 0). The total computational time of the matrix update is

$$\sum_{k=1}^{n_x}\sum_{j=1}^{P_x}2F(n_x-k+1)(n_y-\varepsilon_k+1)\Big|_{\text{while } P_x(k-1)+j<n} \tag{14}$$

where $\varepsilon_k = \text{int}\left|\frac{k}{2^m}\right|$ (which is rounded to the next highest integer). There will be additional computational time due to the normalization of the pivot column with the pivot element. The computational time can be expressed as

$$DP_y(n_y^2/2 + n_y/2 - 1) \tag{15}$$

where D is the time required to perform a division.

The optimal choice for P_x and P_y is $P_x = P_y$. Furthermore the ideal load balance is obtained when $N = n_x P_x$. Under these conditions, the total computational time is

$$2FP_x\left(\frac{1}{3}n_x^3 + \frac{1}{2}n_x^2\right) + DP_x\left(n_x^2 + \frac{1}{2}n_x - 1\right) \tag{16}$$

and the total communication time is

$$2(n-1)S\Delta p_x + 2\alpha\left[P_x\left(\frac{n_x^2}{2} + \frac{n_x}{2} - 1\right)\Delta p_x\right] \tag{17}$$

plus the contribution for pivot row exchange, when necessary. Equations (16) and (17) show that the total computational time is expected to rise as the square of the order of the matrix stored on a single processor, $\frac{2}{3}n_x^2 PF + O(n_x PF)$ (assuming $P = n_x \cdot P_x$), as compared to that for the cube of the order of the full matrix as required by the serial algorithm ($\frac{2}{3}N^3F + O(N^2F)$). The parallel algorithm requires a factor of P_x^2, or P less arithmetic operations than for the serial algorithm. From another point of view, if a fixed block is kept on each processor and P_x and P_y are increased by a multiple of 2^k, which effectively increases the order of the matrix by a multiple of 2^k, the computational time will scale linearly (by 2^k). This is in comparison to the time for the serial algorithm which would scale by a factor of 2^{3k}. The communication time of the parallel algorithm scales linearly with Δp_x and Δp_y for the fixed problem.

Because of the reduced communication time and improved load balance, the above algorithm is much better suited for the fine-grained parallel environment. The efficiency is illustrated by an example. Consider a 1024 node NCUBE used to factor a 1024 order matrix. Using Ortega and Romine's *kij*-r (RSRP) algorithm [11], each of the 1024 processors will be assigned a single row. After 991 transformations, only 33 processors will be active, and the transformation will require 64 arithmetic operations performed concurrently by each of 32 processors. However, using the two-dimensional wrap-mapped algorithm, all 1024 processors will still be active, and each will only perform two arithmetic operations to compute the transformation. In fact, the two-dimensional wrap-mapped algorithm requires $[(1024)^2/3 + (1024)\cdot 32/2]$ floating-point operations, as compared to $[(1024)^2/2 + (1024)/2]$

```
for (j ∈ mycols)
    receive b_k (k ∈ {j ≤ myrow ≤ j + P_y}) from left
    if (j ∈ myrows)
        x_j = b_k / L_jj
        fan out x_j to my-proc-col
      else
        receive x_j
        b_k = b_k − x_j L_kj
        send b_k to the right
      endif

      receive b_i (i ∈ myrows ≥ j + P_y) from left
      for (i ∈ myrows)
          b_i = b_i − x_j L_jj
      send b_i (i ∈ myrows ≥ j + P_y) to right
```

Figure 6.3 Two-dimensionally wrap-mapped wavefront algorithm based on the vector-sum algorithm.

floating-point operations required by the row-storage scheme. Furthermore, the communication required by the kij-r algorithm has a total cost of $[(1024)^2 + 1024 - 2\log_2(1024)]/2$ (without pivoting) [11]. This compares to a total cost of only $32[(32)^2 + 32 - 2\log_2(32)]$ (without pivoting) as required by the two-dimensional wrap-mapping algorithm.

Once the matrix is factorized, solution vectors can be computed for any number of right-hand sides by solving the associated lower and upper triangular linear systems of equations. In the coarse-grained hypercube environment, it was illustrated in Section 6.4.b(2) that the wavefront algorithm [13] was adaptable to wrap mapping by both columns or rows. Using a compute-ahead type of strategy, much of the serialism inherent to the recursive nature of the triangular system solution algorithm was eliminated. This same compute-ahead type of strategy of the wavefront algorithm can also be used in the fine-grained hypercube environment with the matrix being two-dimensionally wrap mapped onto the two-dimensional grid of processors (assume P_x columns by P_y rows). Figure 6.3 illustrates the pseudo-code for the two-dimensionally wrap-mapped wavefront algorithm based on the vector-sum algorithm.

Parameter	NCUBE
S (microseconds)	384
α (microseconds / complex word)	20.8
F (microseconds / FLOP)	17.5

Table 6.3 Machine parameters used for the theoretical models of performance on the NCUBE hypercube.

The segment size σ of the two-dimensionally wrap-mapped algorithm can be thought of as being P_y, where P_y elements of b are computed in parallel along the processor column. The serialism inherent in this algorithm is due to processor columns being in an idle state, while waiting for the next segment of b_k. However, the remaining segments can be computed concurrently while the active segment is passed on.

b. Theoretical Predictions for the Fine-Grained MIMD Hypercube

Theoretical predictions of the performance of the above algorithms for the fine-grained NCUBE hypercube (MIMD) are presented assuming 1024 processing nodes. Each node contains a central processing unit that is similar in architecture to the VAX-11/780, and is equipped with a floating-point accelerator. It also contains eleven bidirectional DMA communication channels. Ten of the channels lead to adjacent processors of the tenth order hypercube, and the eleventh is an I/O interface. Each processing node also has 512 kbytes of local memory.

The theoretical predictions of the factorization algorithm's performance are based on the machine constants listed in Table 6.3. These values were obtained from [13], and are based on each processing node having a clock speed of 8 MHz.

For comparison, both the CSRP algorithm and the two-dimensionally wrap-mapped LU-factorization algorithms studied. The theoretical predictions of the execution times are provided in Tables 6.4 and 6.5, respectively. Note that the benchmarks assume a pivot search, but no row exchanges, and are based on the factorization of a complex-valued matrix. The order of the matrix is chosen so that the matrix block

P	N	Computational Time (sec)	Communication Time (sec)	Total Time (sec)
1	128	100.2	0.	100.2
4	256	198.6	1.96	200.6
16	512	401.1	13.3	414.4
64	1024	819.5	72.6	893.1
256	2048	1707.	367.8	2075.
1024	4096	3671.	1788.	5459.

Table 6.4 Theoretical predictions of the performance of LU decomposition of a complex-valued matrix on the NCUBE using the RSRP algorithm.

P	P_x $(=P_y)$	N	Computational Time (sec)	Communication Time (sec)	Total Time (sec)
1	1	128	100.2	0.	100.2
4	2	256	200.4	1.09	201.4
16	4	512	400.7	4.36	405.1
64	8	1024	801.3	13.08	814.5
256	16	2048	1602.	34.90	1638.
1024	32	4096	3206.	87.26	3293.

Table 6.5 Theoretical predictions of the performance of LU decomposition of a complex-valued matrix on the NCUBE using the two-dimensional wrap-mapped algorithm.

size stored in each processor is constant ($N^2/P = 128^2$). Comparing these results, it is apparent that as the number of processors becomes large (> 128), the two-dimensional wrap-mapped method is much more efficient. This is due to both improving load balance and minimizing interprocessor communication as discussed in Section 6.5.a. Figure 6.4 illustrates the scaled speedup of these two algorithms. It

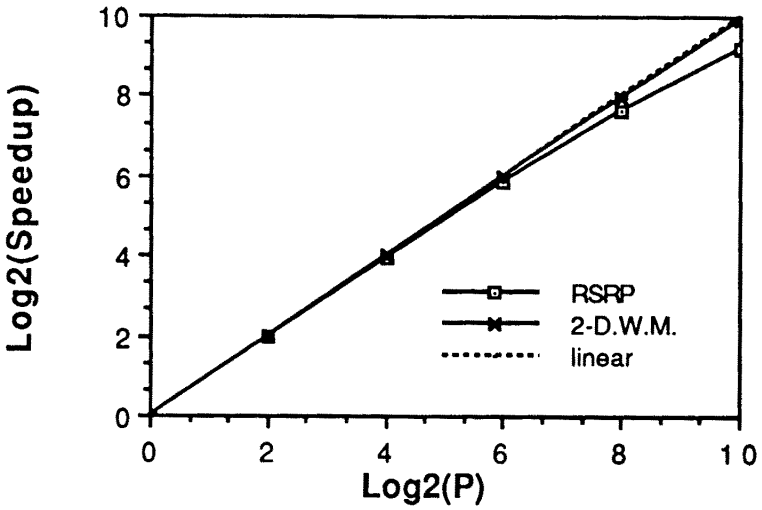

Figure 6.4 Comparison of the theoretical predictions of the scaled speed-ups of the RSRP and the two-dimensional wrap-mapped LU factorization algorithms as a function of the order of the NCUBE hypercube.

N	Solution Time (sec)
64	.233
256	.932
512	1.86
1024	3.73
2048	7.45
4096	14.9

Table 6.6 Theoretically predicted CPU time required to perform the triangular solution using the two-dimensional wrap-mapped wavefront algorithm on a 1024-node NCUBE.

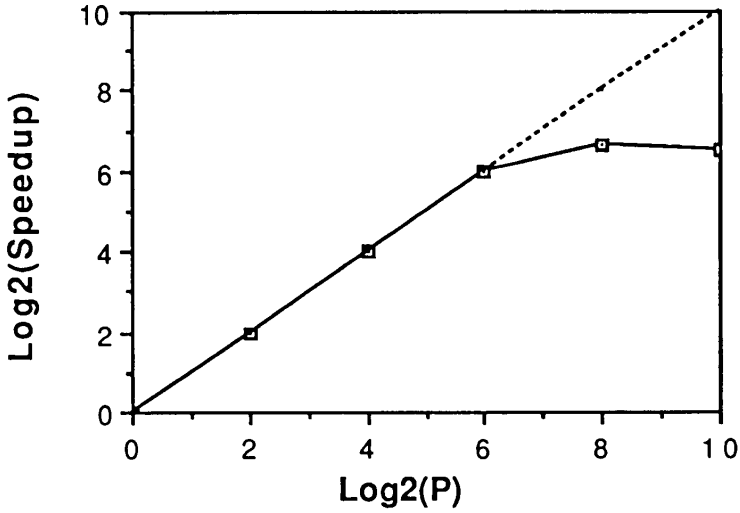

Figure 6.5 Theoretical fixed speedup of the two-dimensional wrap-mapped wavefront algorithm based on the vector-sum formulation as a function of the order of the NCUBE hypercube.

is shown that for the 4096 rank problem a parallel efficiency of 97% can be achieved using the two-dimensional wrap-mapped algorithm as compared to 59% using the CSRP algorithm.

Theoretical predictions of the two-dimensionally wrap-mapped wavefront algorithm based on the vector-sum formulation are illustrated in Figures 6.5. It is observed that the fixed speedup degrades for a large number of processors. This is due to the fixed segment size inherent within this algorithm. However, it is seen in Table 6.6 that the $O(N^2)$ serial algorithm has been reduced to an $O(N)$ algorithm in the fine-grained parallel environment when $N \gg P_x$ and Py.

6.6 Parallel Implementation of the Method of Moments Algorithm Using a Fine-Grained SIMD Hypercube

We now turn our attention to the development and application of parallel algorithms that perform the MoM solution of the electromagnetic scattering problem on Thinking Machine's CM2 (better known as the Connection Machine) [23]. Thinking Machine's CM2 has incorporated SIMD operation and the hypercube topology to obtain massive parallelism. The CM2 employs up to 65,536 (64K) bit-serial processors, which is a sixteenth-order hypercube. Each processor has access to 64 kbits of bit-addressable memory, which combines for a total of 512 M bytes of addressable memory, and an arithmetic-logic unit (ALU) that can operate on variable length operands. In addition to the standard ALU, each processor can be equipped with a floating-point accelerator (FPA). There is one 32-bit FPA (more recently, 64-bit FPAs are available) assigned to a group of 32 processors, and the FPA is accessed in a bit-serial manner. The FPA offers a ten-times speedup for floating-point operations.

Communication within the hypercube network is handled by the *router,* which consists of hardware that directs the exchange of data between all processors in the hypercube. The CM2 also offers a more structural communication mechanism called the NEWS grid, which allows processors to pass data according to a regular rectangular path. The advantage of the NEWS grid is that the overhead of an explicit specification of a destination address and path is eliminated, resulting in faster communication time. The NEWS grid is dimensioned as an $N \times M \times \cdots \times R \cdots$ grid where N, M, and R are powers of two and their product is equal to the total number of processors.

The CM2 is a SIMD multiprocessor, meaning that all processors perform the same operation at each time step. As such, individual processors do not house independent programs and must be provided with the operation to perform at each time step. These instructions are provided by the *sequencer.* The sequencer breaks down each PARIS instruction (the CM2's assembler language) from the Control Processor (CP) into a sequence of low-level data processes and memory operations. The sequencer then broadcasts these low-level instructions to the processors. There is one sequencer for each 16-k block of processors.

a. The Matrix Fill Task

In Section 6.5, it was illustrated that the computational time needed to fill the matrix on the MIMD hypercube was equal to the largest time required by a single processor to compute N^2/P elements. On the CM2, where P is as large as 65,536, the expected speedup is extremely large. However, there are obstacles that arise in the environment of the CM2 that must be overcome, namely, the SIMD operation of the CM2 and the small amount of local memory.

The SIMD operation of the CM2 requires that every processor perform the same instructions concurrently. Therefore, if a full CM2 is being used and 65,536 matrix elements are being computed in parallel, the same algorithm must be employed to compute all of these elements. On the surface, this may appear trivial; however, when developing a MoM code, difficulties may be encountered. The most obvious is that near the singularity of the integral operator kernel (the Green's function) special treatment is required. This can be performed using a singularity extraction technique. However, this operation must be performed in computing every matrix element, introducing redundant computation.

Discretizing the integro-differential operator describing the three-dimensional scattering problem will require the numerical evaluation of multidimensional integrals. Numerical integration near the singularity of the kernel will require finer sampling of the integrand than in regions that are farther away from the singularity for an accurate evaluation. In fact, as the separation between the source and observation points grows larger ($> \lambda/4$) asymptotic approximations of the integral can be made. However, on the SIMD CM2, one cannot determine special treatment for each matrix element, since all must be handled the same. Therefore, a fixed-point integration scheme (i.e., Gaussian or Gauss-Kronrod quadrature [24]) that provides accuracy for the self-term elements must be used to compute all elements.

Another concern is that when the basis or testing functions representing each component of the vector current density differ, the integrands are no longer identical. As a result, in order to utilize the SIMD operation, care must be taken to find uniformity in the operations by either choosing favorable basis and testing functions, or by introducing redundancy in the operation.

b. Direct Methods of Solution

In Section 6.5.a, it was shown that by two-dimensionally wrap mapping the matrix onto a two-dimensional grid of processors, an efficient LU-factorization algorithm could be developed for a fine-grained MIMD hypercube. In a similar fashion, this LU-factorization algorithm can be employed with high parallel efficiency on a SIMD fine-grained hypercube [25]. However, when computing the solutions for a large number of right-hand sides on the SIMD multiprocessor, it was shown in [25] that a bottleneck arises when treating the lower and upper triangular systems of equations. Because of the SIMD architecture, a compute-ahead-type strategy cannot be used, leaving a large amount of serialism in the recursive algorithm. As a result, it is not desirable to perform LU factorization on the CM2. Instead, when a large number of forcing functions need be solved for, it is more profitable to perform a full matrix inversion. Once the inverse is computed, the solution vector can be computed by a matrix-vector multiply, which is quite readily computed in the fine-grained environment.

Initially, the processors are configured as a two-dimensional grid, as was done with the parallel-factorization algorithm presented in Section 6.5.a. The parallel inversion algorithm is based on the Gauss-Jordan method. The algorithm can be thought of as a series of $N-1$ transformations of an N-th order matrix. Since the entire matrix is operated on during each transformation, load balance is easily ensured. Nevertheless, the matrix must be mapped onto the grid of processors in a manner such that interprocessor communication is minimal. Specifically, the matrix is mapped onto the processor grid such that a single row of the matrix is evenly distributed about a single row of processors, and a single column of the matrix is evenly distributed about a single column of the processor grid. This general blocking scheme satisfies the above conditions, as does the two-dimensional wrap-mapping scheme described in Section 6.5.a.

The basic parallel algorithm used to perform the Gauss-Jordan inversion is illustrated in Figure 6.6. The algorithm overwrites the original $N \times N$ matrix with its inverse. Partial pivoting is implemented to treat generalized matrices, however the physical exchange of rows is not required to maintain load balance. (The use of Gauss-Jordan inversion with partial-pivoting may result in additional round-off error as compared to LU factorization. However, if needed, full pivoting can easily be implemented in a concurrent manner with little addi-

```
for (k = 1, N)
    if (k ∈ mycols)
      find pivot element (fan in)
      if (k ∈ myrows)
        a_{k,k} = 1./a_{k,k}
        broadcast a_{k,k} to grid row
      for (myrows)
          l_i = a_{i,k}
          if (i ≠ k) a_{i,k} = 0
    if (k ∈ myrows)
      a_{k,j} = a_{k,j} · a_{k,k}
    broadcast l_i along grid rows
    broadcast a_{k,j} along grid columns
    for (i in myrows)
      if (i ≠ k)
        for (j in mycols)
            a_{i,j} = a_{i,j} − l_i · a_{k,j}
```

Figure 6.6 Parallel Gauss-Jordan algorithm for the SIMD hypercube.

tional overhead.) The broadcasting of the column vector l_{ik} and the row vector a_{kj} is performed in a similar fashion to that employed by the LU-factorization algorithm, as discussed in Section 6.5.a. A single column of processors in the processor grid will contain n_y elements of l_{ik}. Each processor holding l_{ik} concurrently broadcasts its n_y elements along its respective row of the processor grid. The row-vector is broadcast in a similar fashion. The total communication cost for broadcasting these two vectors for $N-1$ transformations is

$$\sum_{k=1}^{N-1} [(S + \alpha n_x \Delta p_x) + (S + \alpha n_y \Delta p_y)]$$
$$= 2S(N-1) + \alpha(N-1)(n_x \Delta p_x + n_y \Delta p_y) \qquad (18)$$

where $\Delta p_x = \log_2(P_x)$, $\Delta p_y = \log_2(P_y)$, S is the startup time of a communication, and α is the time to communicate one complex

floating-point number. The additional communication time required to locate the pivot element and to broadcast it is

$$2S(n-1)+\alpha(N-1)(\Delta p_x+\Delta p_y) \tag{19}$$

Furthermore, the total computational cost due to pivoting and normalizing of the pivot rows is $O(P_x n_x^2)$.

After the pivot column and row are broadcast, the matrix update is performed. Once again this is treated as the matrix operation

$$a_{ij}=a_{ij}-l_{ik}a_{kj} \tag{20}$$

where each processor contains an $n_x \times n_y$ block of a_{ij} and the corresponding $n_x \times n_y$ block of the matrix found by the vector-vector product $(l_{ik}a_{kj})$. To perform the update in parallel $N-1$ times requires a computational cost of

$$2Fn_x n_y(N-1) \tag{21}$$

where $2F$ is the time required to perform one add and one multiply (real or complex). If $n_x = n_y$, this is approximately

$$\approx 2Fn_x^3 P_x \tag{22}$$

As a result, the computational time required by the parallel matrix inversion algorithm is $O(n_x^3 \cdot P_x)$ (if $n_x = n_y$), where n_x is the order of the subblock stored on each processor. Under the same assumption, from (18) the communication time is $O(n_x^2 \cdot P_x)$. This is compared to $O(N^3)$ operations required by the serial algorithm. Therefore, the total number of operations is reduced by roughly P_x^2, or P, the total number of processors.

The parallel LU-factorization algorithm discussed in Section 6.5.a requires $2n_x^3 P_x/3 + O(n_x^2 P_x)$ complex arithmetic operations and $O(n_x^2 P_x)$ communications. The matrix inversion algorithm will obviously be more time-consuming than the LU-factorization algorithm. However, its primary advantage is that once the inverse is known, the solution vector can be computed by a matrix-vector multiply, which is performed with great efficiency on a distributed-memory computer configured as a two-dimensional mesh.

Assume that the right-hand side is stored on a single row of the processor grid. Then the matrix-vector product will have a communication cost of

$$2S + \alpha n_x \Delta p_y + \alpha n_y \Delta p_x \tag{23}$$

and a computational cost of

$$2F n_x n_y \tag{24}$$

assuming that the right-hand side is first broadcast to the processor grid. Subsequently, the row vector-vector dot products are performed in parallel.

The parallel matrix-vector multiply algorithm is significantly more efficient than the parallel forward/back substitution algorithms, since the serialism is only due to communication, which is $O(n_x)$.

c. Example: Electromagnetic Scattering by a Plate (EFIE)

In this section, the numerical example of the electromagnetic scattering by a finite-sized flat perfect conducting plate is presented. The purpose of this example is to exemplify the parallel efficiency of the algorithm when treated as a general surface scatterer. Therefore, any symmetries or Toeplitz and block-Toeplitz characteristics of the matrix that may arise are deliberately ignored by the parallel algorithm. The algorithm was implemented on the CM2, and the benchmarked times were compared with those encountered when running a sequential version of the algorithm on the CRAY-2.

The plate is assumed to be located in the $z = 0$ plane, and can be arbitrary in geometry. It is also assumed to be infinitesimally thin. The surface of the scatterer is replaced by the equivalent electric density $\overline{J}(x, y) = J_x \hat{x} + J_y \hat{y}$. The EFIE is derived by enforcing the boundary conditions on the tangential electric field at the scatterer surface. (The expression for the EFIE is similar to (4).)

The plate is discretized into rectangular cells as illustrated in Fig. 6.7. The current density is represented by a finite superposition of rooftop basis functions, which span two adjacent cells on the plate. If all the cells in the model are of size (a, b), then J_x is expressed as a superposition of the basis functions.

$$B_{xn}(x, y) = t(x; x_n - a, x_n, x_n + a)p\left(y; y_n - \frac{b}{2}, y_n + \frac{b}{2}\right) \tag{25}$$

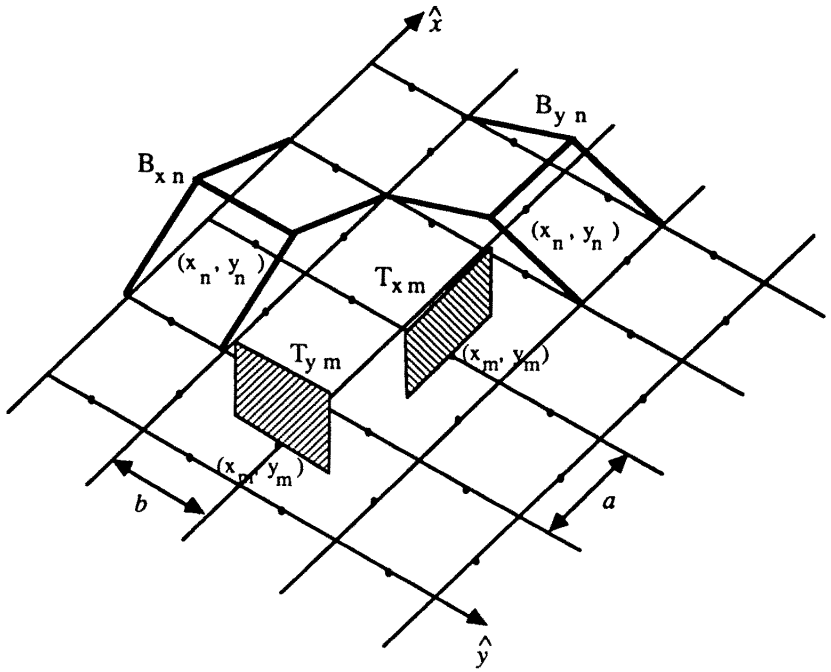

Figure 6.7 The rectangular cells representing the plate surface, the rooftop basis functions B_{xn} and B_{yn} with centers (x_n, y_n), and the razor blade testing functions T_{xm} and T_{ym} with centers (x_m, y_m).

where the centers of the basis functions are located at the interface of two adjacent cells, as illustrated in Fig. 6.7. In a similar manner, J_y is expressed as the superposition of the basis functions

$$B_{yn}(x,y) = t(y; y_n - b, y_n, y_n + b)p\left(x; x_n - \frac{a}{2}, x_n + \frac{a}{2}\right) \qquad (26)$$

The EFIE is then discretized by taking its inner product with a set of testing functions. The testing functions are chosen to be razor blade

functions, described as (Fig. 6.7)

$$T_{xm} = p\left(x; x_m - \frac{a}{2}; x_m + \frac{a}{2}\right)\delta(y - y_m) \tag{27}$$

and

$$T_{ym} = p\left(y; y_m - \frac{b}{2}; y_m + \frac{b}{2}\right)\delta(x - x_m) \tag{28}$$

The discretized EFIE is then expressed as a matrix equation

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}\begin{pmatrix} j_x \\ j_y \end{pmatrix} = \begin{pmatrix} e_x \\ e_y \end{pmatrix} \tag{29}$$

where

$$\begin{aligned}
A_{mn} &= \frac{1}{-j\omega\varepsilon_0}\int\int \frac{e^{-jkR}}{4\pi R}\left[\frac{\partial^2}{\partial x^2} + k^2\right](T_{xm} * B_{xn})\,dx\,dy \\
B_{mn} &= \frac{1}{-j\omega\varepsilon_0}\int\int \frac{e^{-jkR}}{4\pi R}\left[\frac{\partial^2}{\partial x\partial y} + k^2\right](T_{xm} * B_{yn})\,dx\,dy \\
C_{mn} &= \frac{1}{-j\omega\varepsilon_0}\int\int \frac{e^{-jkR}}{4\pi R}\left[\frac{\partial^2}{\partial y\partial x} + k^2\right](T_{ym} * B_{xn})\,dx\,dy \\
D_{mn} &= \frac{1}{-j\omega\varepsilon_0}\int\int \frac{e^{-jkR}}{4\pi R}\left[\frac{\partial^2}{\partial y^2} + k^2\right](T_{ym} * B_{yn})\,dx\,dy
\end{aligned} \tag{30}$$

where $R = \sqrt{x^2 + y^2}$.

In the above expressions, the convolution of the testing and basis functions and their second-order derivatives can be expressed analytically, thus leaving only the outer two-dimensional integration to be performed numerically.

The parallel matrix fill algorithm was implemented as follows. Initially, the hypercube was configured as a two-dimensional NEWS grid. The size of the grid was chosen to be large enough so that a single element is only mapped onto a single processor. This was accomplished using a utility of the CM2 known as virtual processing. Virtual processing allows one to partition each of the bit-serial processors and its 64 kbits of local memory into a number of virtual processors. In other words, if a virtual processor ratio of 8:1 is required, then each single-bit processor is partitioned into 8 virtual processors, each having 8 kbits of local memory. The advantage of virtual processing is the simplicity

of programming, since only a single element is computed per processor. However, additional memory is required since some local variables (mainly stack and temporary variables) must be stored multiple times on a single processor with virtual processing. This becomes significant when large virtual processor ratios are required (> 128). In light of this, it is important to reduce the number of local variables, wherever possible, and to store any variables shared by all processors on the control processor (or front end).

The first step in performing the matrix fill is to define a single process to compute the matrix elements, viz., the expressions for the integrands occurring in the double integrals of (31). There are inherent similarities in the computation of the elements in blocks A_{mn} and D_{mn}, as well as in B_{mn} and C_{mn}, and their integrands can easily be manipulated to be identical. Therefore, each of these pairs of blocks are computed concurrently. A variable change can be introduced in order to produce identical bounds of integration for the pairs of element blocks. As a result, fixed point integration can be used to evaluate the integrals, with the (x, y) positions of the integral being constant and stored on the front end. Due to the nature of the virtual processing the load still remains balanced even though these two pairs of matrix blocks are treated separately.

The numerical integration employed a fixed-point routine based on the one-dimensional 15-point Gauss-Kronrod quadrature method [24]. The method is computationally efficient and numerically accurate.

The $1/R$ singularity in the Green's function was handled with the use of a singularity extraction technique, in which the extracted term is added back in analytically. Unfortunately, the extraction must be performed in the computation of all the elements, introducing redundant computation. In order to prevent round-off errors, which can occur for entries which are actually nonsingular, the singular term is weighted by zero in elements whenever the basis and testing function centers are separated by large distances (typically more than two cells away).

Once the matrix elements are computed, the inversion of the matrix is performed using the parallel inversion algorithm described in Section 6.6.b. With the use of virtual processing, the number of elements stored per physical processor is $n_x \times n_y$, and row and column grid communication requires $\log_2(P_x)$ and $\log_2(P_y)$ communication steps, respectively, where P_x and P_y are representative of the dimensions of the actual NEWS grid and $n_x = \text{int}\,|N/P_x|$ and

$n_y = \text{int}\,|N/P_y|$.

This same algorithm was programmed for a single processor CRAY-2 for a benchmark comparison. On the CRAY-2, the same double integration routines were employed as used by the parallel algorithm. However, double integration was only used on elements whose distance between the centers of the source and observation patches is less than $\lambda/4$. Outside of this region an asymptotic approximation is used to evaluate the integrations analytically. It is also noted that neither matrix symmetry, nor Toeplitz and block-Toeplitz characteristics were utilized by the sequential algorithm, since the goal of this benchmark is to provide a comparison of computational times when performing the MoM solution for a general surface scatterer.

Once the matrix elements are computed, the matrices are decomposed into LU form using the LINPACK subroutine CGEFA, which was optimized for vectorization for the CRAY-2. The solution vectors were then computed sequentially using the LINPACK subroutine CGESL.

Tables 6.7 and 6.8 present the computational times required by the CM2 and the CRAY-2, respectively, to perform the MoM solution. The times are given as a function of the total number of unknowns N and the number of processors employed by the CM2. The CM2 program was written in C/PARIS [26], which is a language that combines PARIS (the Parallel-Assembler Instruction Set) with a C hierarchy. The floating-point operations were performed in 32-bit precision. The CM2 used is currently equipped with 32-bit Weitek-floating point accelerators, which provide a ten-times speedup for 32-bit precision. If 64-bit precision is used, the speed of floating-point computation is reduced by a factor of 20 (this is attributable to the bit-serial nature of the CM2). Numerical roundoff error was encountered in the solution vectors for the larger cases ($N > 1000$).

It is observed from Table 6.7 that the fixed speedup of the matrix fill algorithm is linear. As expected, the computational time required by the CM2 grows as N^2 for a fixed number of processors. However, the computational time required by the CRAY-2 only grows linearly. This arises due to the fact that the time consuming numerical integration is only performed on the self terms, or near-self terms, and the remaining elements are computed analytically using the asymptotic approximation.

P	Plate Size (λ)	N	Fill Time (CPU-sec)	Factor and Solve N r.h.s. (CPU-sec)
8192	1.6×1.6	480	79.73	27.68
16384	"	"	40.02	16.34
32768	"	"	20.24	9.71
8192	2.3×2.3	1012	317.89	196.04
16384	"	"	158.92	109.55
32768	"	"	79.82	71.72
32768	3.2×3.2	1984	317.80	395.82

Table 6.7 Computational times required by the parallel MoM algorithm on the CM2.

P	Plate Size (λ)	N	Fill Time (CPU-sec)	Factor Time (CPU-sec)	Solve N r.h.s. (CPU-sec)
1	1.6×1.6	480	142.67	.9984	15.20
1	2.3×2.3	1012	330.58	8.286	128.74
1	3.2×3.2	1984	717.84	59.33	927.46

Table 6.8 Computational times required by the sequential MoM algorithm on the CRAY-2.

The factorization is performed much faster on the CRAY-2, with the use of the vectorized CGEFA subroutine, than the matrix inversion performed on the CM2. However, when a large number of right-hand sides is involved ($O(N)$), the time required to both factor the matrix and solve for the right-hand sides is performed faster on the CM2. As was predicted in Section 6.6.b, if the number of processors is fixed, as $N \gg \sqrt{P}$ the matrix solution algorithm grows roughly as n_x^2 on the CM2, whereas it grows as N^3 on the CRAY-2.

The largest impedance matrix that could be computed and stored in core memory on a 32,768 processor CM2 is of order 2048. This requires 128 complex-valued elements (8192 bits) of the impedance matrix to be stored at each processor, as well as the portions of the geometry vectors that describe the block and the right-hand side vectors. A larger matrix could not be stored in core memory because virtual processing requires additional memory at each processing node for temporary storage. In fact, for a virtual processor ratio of 128:1, a single complex-valued temporary storage location assigned to each virtual processor requires the storage of 128 complex-valued numbers at each physical processor. As a result, the 65536 bits of local memory are used up rather quickly. A code can be much more memory efficient if virtual processing is not used, at the expense of a much more tedious programming task [25]. Another alternative would be to use an out-of core solution, taking advantage of the Data Vault concurrent storage available with the CM2.

Preprocessing and postprocessing have not been included in the preceding benchmarks. The above conclusions assume that these tasks are performed on the front end computer, which was a SUN 4/490. The geometry vectors indicating the positions and supports of the testing and basis functions are initially determined on the SUN. This is an easy task for the flat-plate problem; however, it may require a general purpose mesh generation program for more complicated three-dimensional bodies. Once the geometry vectors are computed, the basis function geometry vector is sent to a single row, and the testing function geometry vector is sent to a single column of the processor grid of the CM2. Subsequently, the vectors are spread to all other processors. The time to transfer the two geometry vectors to all processors of the CM2 for the flat plate problem is illustrated in Table 6.9. These vectors consist of only the center position of the basis or testing functions (x_n, y_n).

Once the solution vectors are computed they must be downloaded to the front end for postprocessing. The times required to download the N solution vectors to the front end when N is chosen to be 1012 (64-bit complex numbers) are illustrated in Table 6.10.

N	P	Upload Time (sec)
1012	8192	.064
"	16384	.041
"	32768	.029
1984	8192	—
"	16384	.1148
"	32768	.071

Table 6.9 Time required to upload the geometry vectors onto the CM2 from the Sun 4/490.

P	Download Time (sec)
8192	7.16
16384	5.75
32768	5.02

Table 6.10 Time required to download the solution vectors onto the Sun 4/490 ($N = 1012$).

6.7 Summary

Parallel algorithms performing the matrix fill and matrix solve tasks of the MoM solution of electromagnetic scattering problems have been developed for three types of hypercube multiprocessor environments: (i) the coarse-grained MIMD hypercube; (ii) the fine-grained MIMD hypercube; and, (iii) the fine-grained SIMD hypercube. In all three environments, it was shown that the matrix fill subtask is a highly parallel process, since the computation of the matrix elements inherently requires no interaction among processors. Therefore, the N^2 elements of the matrix can be computed in the time required by a single processor to compute N^2/P elements. Consequently, the larger the number of processors, the larger the speedups obtained. Special con-

siderations arise in the SIMD hypercube environment because it is necessary to employ an algorithm that computes all elements of the matrix. This algorithm must also include the treatment of the numerical integration over the singularity that occurs in the self-terms of the matrix.

The parallel algorithms performing the matrix solution were based on direct methods. The optimal algorithm for the coarse-grained hypercube is based on the *kij*-r and the *kji*-c forms of LU factorization by Gaussian elimination [11]. If the matrix is wrap mapped onto the hypercube, the parallel *kij*-r, using row storage and row pivoting (RSRP), and *kji*-c, using column storage and column pivoting (CSRP), algorithms are load balanced and interprocessor communication is minimal. When $N \gg P$, nearly linear speedups can be achieved with these algorithms. After the matrix is factorized, the solution of the triangular systems of equations can be solved efficiently using the wavefront algorithm.

A parallel algorithm performing the MoM solution of the scattering of an electromagnetic wave from a body of revolution was developed for the coarse-grained Mark III hypercube. The decoupled matrices that arise from the MoM formulation were not only computed in parallel on the Mark III, but also computed simultaneously with the use of the FFT. This dual sense of parallelism led to an extremely efficient program. A 32-node Mark III exhibited performance comparable to an optimal algorithm implemented on a single-processor CRAY X-MP.

In the fine-grained MIMD environment, one can no longer expect that the number of unknowns far exceeds the number of processors. As a result, the CSRP and the RSRP algorithms become burdened by interprocessor communication, and load balance is reduced. An algorithm based on the *kij*/*kji* forms of LU factorization by Gaussian elimination was presented using a new mapping scheme, referred to as two-dimensional wrap mapping. With the use of this mapping scheme, interprocessor communication is greatly reduced and an improved load balance is maintained. This algorithm is shown to be highly efficient when $N \gg \sqrt{P}$. The wavefront algorithm can also be employed on the fine-grained MIMD hypercube using the two-dimensional wrap mapping.

Due to the single instruction-type operation of the fine-grained SIMD hypercube and the recursive nature of the algorithm, the solution of the triangular systems of equations produced by the LU factor-

ization in this environment is prohibitive. On the other hand, after a full matrix inversion is performed, the solution vector can be efficiently produced with a matrix-vector multiply. Therefore, a parallel matrix inversion algorithm based on the Gauss-Jordan method was developed for the fine-grained SIMD hypercube. If the hypercube is configured as a two-dimensional grid using a binary reflected Gray coded scheme, and the rows and columns of the matrix are evenly distributed about rows and columns, respectively, of the grid, interprocessor communication is minimized and load balance is ensured.

A parallel algorithm that performs the MoM solution of the scattering of an electromagnetic wave by a flat conducting plate was developed for the fine-grained SIMD hypercube. The algorithm was implemented on the CM2, and was benchmarked against a sequential algorithm written for a single processor CRAY-2. A very high degree of parallelism was achieved for the matrix fill algorithm on the CM2. Due to its SIMD nature, the computational time of the parallel matrix fill algorithm is $O(N^2/P)$. Whereas, by using an asymptotic approximation to approximate the multidimensional integrals when the testing and basis functions are separated by sufficient distances, the computational time of the sequential matrix fill algorithm of the CRAY-2 is only $O(N)$. When solving for a large number of right-hand sides $(O(N))$, the computational time of the parallel matrix solution algorithm was roughly $O(n_x^2 P_x)$ on the CM2. The computational time of the sequential algorithm on the CRAY-2 was $O(N^3)$, as expected. Overall, the CM2 competed favorably against the CRAY-2 for this benchmarked case.

This study has focused on the solution of electromagnetic scattering problems via the MoM on hypercube multiprocessor computers. Using algorithms specifically tailored to the machine's architecture, it was demonstrated that computational speeds comparable to today's supercomputers are possible, and cost effective supercomputing can be provided by hypercube multiprocessor computers for this class of problems.

References

[1] Flynn, M. J., "Some computer organizations and their effectiveness," *IEEE Trans. Computers,* **21,** 948–960, 1972.

[2] Hockney, R. W., "MIMD computing in the USA — 1984," *Parallel Computing,* **2,** 119–136, 1985.

[3] Gustafson, J. L., G. R. Montry, and R. E. Benner, "Development of parallel methods for a 1024-processor hypercube," *SIAM J. Sci. Stat. Comput.,* **9,** 1988.

[4] Mitzner, K. M., "An integral equation approach to scattering from a body of finite conductivity," *Radio Science,* **2,** 1459–1470, 1967.

[5] Harrington, R. F., *Field Computation by Moment Methods,* Macmillan, New York, 1968.

[6] Strang, G., *Linear Algebra and Its Applications,* Academic Press, New York, 1980.

[7] Broyden, C. G., "Error analysis," in *The State of the Art in Numerical Analysis,* edited by D. Jacobs, Academic Press, New York, 1977.

[8] Golub, G. H., and C. F. van Loan, *Matrix Computation,* 2nd ed., The Johns Hopkins University Press, Baltimore, MD, 1989.

[9] Geist, G., and M. Heath, "Matrix factorization on a hypercube multiprocessor," in *Hypercube Multiprocessors,* edited by M. Heath, Society for Industrial and Applied Mathematics, Philadelphia, PA, 161–180, 1986.

[10] Chu, E., and A. George, "Gaussian elimination with partial pivoting and load balance on a multiprocessor," *Parallel Computing,* **5,** 65–74, 1987.

[11] Ortega, J. M., and C. H. Romine, "The *ijk* forms of factorization methods II. Parallel systems," *Parallel Computing,* **7,** 149–168, 1988.

[12] Geist, G. A., and C. H. Romine, "LU factorization algorithms on distributed-memory multiprocessor architectures," *SIAM J. Sci. Stat. Comput.,* **9,** 639–649, 1988.

[13] Heath, M. T., and C. H. Romine, "Parallel solution of triangular systems on distributed-memory multiprocessors," *SIAM J. Sci. Stat. Comput.*, **9**, 558–588, 1988.

[14] Andreasen, M. G., "Scattering from bodies of revolution," *IEEE Trans. Antennas Propagat.*, **AP-13**, 303–310, 1965.

[15] Mautz, J. R., and R. F. Harrington, "Radiation and scattering from bodies of revolution," *Applied Scientific Res.*, **20**, 405–435, 1969.

[16] Harrington, R. F., and J. R. Mautz, "Radiation and scattering from loaded bodies of revolution," *Applied Scientific Res.*, **26**, 209–217, 1971.

[17] Glisson, A. W., and D. R. Wilton, "Simple and efficient numerical techniques for treating bodies of revolution," *University of Mississippi Engineering Experiment Station* Technical Report No. 105, 1979.

[18] Glisson, A. W., and D. R. Wilton, "Simple and efficient numerical methods for problems of electromagnetic radiation and scattering from surfaces," *IEEE Trans. Antennas Propagat.*, **AP-28**, 593–603, 1980.

[19] Joseph, J., and R. Mittra, "Radar scattering by metallic bodies of revolution with or without resistive coatings," *URSI Radio Science Meeting Program and Abstracts*, 192, Philadelphia, PA, 1986.

[20] Gedney, S. D., and R. Mittra, "The use of the FFT for the efficient solution of the problem of electromagnetic scattering by a body of revolution," *IEEE Trans. Antennas Propagat.*, **AP-28**, 313–322, 1990.

[21] Bergland, G. D., and M. T. Dolan, "Fast Fourier transform algorithms," in *Programs for Digital Signal Processing*, IEEE Press, New York, 1.2.1–1.2.18, 1979.

[22] Gedney, S. D., "Solution of open region electromagnetic scattering problems on hypercube multiprocessors," Ph.D. dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, 1991.

[23] Hillis, W. D., "The Connection Machine," *Scientific American*, **256**, 108–115, 1987.

[24] Piessens, R., E. deDoncker-Kapenga, C. Uberhuber, and D. Kahaner, *QUADPACK: A Subroutine Package for Automatic Integration*. Springer-Verlag, New York, 1983.

[25] Gedney, S. D., A. F. Peterson, and R. Mittra, "The solution of electromagnetic scattering problems on multiprocessor computers via the method of moments," *Electromagnetic Communication Laboratory* Technical Report No. 89-5, University of Illinois, Urbana, IL, 1989.

[26] *Introduction to Programming in C/Paris, Version 5.0*, Thinking Machines Corporation, Cambridge, MA, 1989.