Data Parallel Supercomputing

S. Lennart Johnsson

YALEU/DCS/TR-741
September 1989

To appear in "The Use of Parallel Processors in Meteorology",
Springer Verlag.

Data Parallel Supercomputing

S. Lennart Johnsson

Thinking Machines Corporation
Cambridge, MA 02142
and
Department of Computer Science
Yale University
New Haven, CT 06520

Abstract

Supercomputers with a performance of a trillion floating-point operations per second, or more, can be produced in state-of-the-art MOS technologies. Such computers will have tens of thousands of processors interconnected by a network of bounded degree. Reducing the required data motion through a careful choice of data allocation and computational and routing algorithms is critical for performance. The management of thousands of processors can only be accomplished through programming languages with suitable abstractions.

We use the Connection Machine as a model architecture for future supercomputers, and Fortran 8X as an example of a language with some of the abstractions suitable for programming thousands of processors. Some of the communication primitives suitable for expressing structured scientific computations are discussed, and their benefit with respect to performance illustrated. With thousands of processors engaged in the solution of a single scientific problem, several subtasks are often treated concurrently in addition to the concurrent execution of each subtask. Some issues in constructing scientific libraries for such environments are discussed. Concurrent algorithms and performance data for matrix multiplication and the Fast Fourier Transform are presented. The solution of the compressible Navier-Stokes equation in three spatial dimensions by an explicit finite difference method, and the solution of a parabolic approximation of the Helmholtz equation by an implicit method are two examples of applications for which data parallel implementations are described briefly. The Helmholtz equations models three dimensional acoustic waves in the ocean.

1 Introduction

In the next decade, supercomputers are expected to have a performance of at least one trillion instructions per second, and a primary storage of tens to hundreds of Gbytes [5]. At this rate of computation and memory size, the operation code, the operand addresses, and the operands require 300–400 bits for a single instruction. The storage (including registers, or caches) must deliver 300–400 trillion bits per second, or about 16 million bits per cycle at a 25 MHz clock rate. This clock rate is somewhat conservative for

MOS technologies, but it cannot be expected to become higher by more than a small constant factor. The width of the storage needs to be several million bits. Assuming each processor can deliver 50 Mflops/sec, 40,000 processors will have a nominal peak capacity of two trillion floating-point instructions per second. A system of this complexity is entirely feasible to build. In half micron technology, 40,000 chips with on-chip floating-point units and memory are projected to have a total of about 64 Gbytes of primary storage. With the required storage bandwidth and with tens of thousands of processing units, a network is the only feasible alternative for passing data between processors and storage units. Using a technology that is an order of magnitude faster than MOS technologies, such as bipolar GaAs technology (used for the CRAY-3), would still require thousands of processing units for an architecture with a performance of a trillion floating-point operations per second.

In highly concurrent network architectures, the nominal processing capability is determined by the processing speed of a single processor and the number of processors. The real processing capability is determined by how well the individual processing units can be utilized, load balance, and how well the network supports the data motion required by the computation. The capacity available for the data motion is determined by technological constraints, and the requirements determined by data placement, computational algorithms, and routing algorithms. Of the various technological constraints that determine the performance characteristics of an architecture, the ones related to data motion are the most unforgiving with respect to performance.

In this paper we first review the communication capabilities of MOS technologies, and the communication needs of a few typical scientific applications. We then briefly discuss a programming model for architectures with a large number of processing units. The Connection Machine® architecture is presented in section four which also discusses features provided for efficient communication. Two basic computational primitives, matrix multiplication and the Fast Fourier Transform, are described in section five; and two applications, the solution of the compressible Navier-Stokes equations and the solution of a parabolic equation forming an approximation to Helmholtz equation are discussed. The last problem occurs in underwater acoustics.

2 Communication issues

In this section, we consider the communication capabilities of architectures built out of MOS technologies, the requirements of typical scientific computations, and the potential benefits of a good data placement or *address map*. The purpose of a good address map is to reduce the need for communication resources by placing data that frequently interacts close to each other.

A suitable metric for measuring locality of reference is determined either by the topology of the data set or the communications network. In solving partial differential equation, common distance measures are of the form $(\sum_{i=0}^{d} |x_i - y_i|^p)^{\frac{1}{p}}$, where d is the dimension-

ality of the problem domain and p the type of *norm*. The 2-norm (Euclidean distance) is often used in the physical domain. The 1-norm measures the distance between two points corresponding to traversals along coordinate axes. This measure is particularly interesting for Boolean cube networks. In such a network of n dimensions with x and y being processor addresses, and x_i and $y_i, 0 \leq i < n$ being the distances (0 or 1) along the coordinate axes, the 1-norm is equal to the *Hamming* distance between the two points. The Hamming distance is equal to the minimum number of communication links a data item must traverse to move from processor x to processor y in a Boolean cube network. The 1-norm is not ideal for all networks. In a completely interconnected network all points are at unit distance from each other, and the 0-norm is a relevant distance measure.

In state-of-the art MOS technologies, 10^3 - 10^4 wires fit across a chip. The total data motion capacity of 40,000 chips is 100 - 1,000 TBytes/sec at 25 MHz clock rate without sharing of on-chip channels between different data paths. Assuming current standard packaging technologies of 100 - 300 pins per chip, the data motion capacity at the chip boundary is about 10 TBytes/sec. The data transfer rate on a chip is one to two orders of magnitude higher than the transfer rate at the chip boundary. At the board boundary, assuming connectors with 500 pins, the data motion capacity for a 200 board system is about 0.16 TBytes/sec. The transfer rate at the chip boundary is one to two orders of magnitude higher than the rate at the board boundary. The transfer rate at the board boundary is two to three orders of magnitude below the required rate for a system with a performance in the Tflop/sec range. A sustained performance of this magnitude is not possible with current packaging technologies without locality of reference.

Many mathematical models of physical phenomena, such as (partial) differential equations, are derived from local interaction rules. The discrete approximation of the continuous models are typically also derived from local approximations. For instance, the difference stencils used to approximate derivatives in finite difference techniques are local approximations. Finite elements provide a different local approximation. The difference stencils in finite difference techniques and the elements in finite element techniques completely define the spatial data interactions in one step of an *explicit method* for the solution of the discretized equations. The data interaction is local in the physical domain. The classical iterative methods for the solution of linear systems of equations only require local data interaction in the index space used for the solution variables. The conjugate gradient method requires a global reduction operation for the computation of scaling factors, and a global copy, or broadcasting, operation for the distribution of these factors in addition to the same local communication as required by Jacobi's method. Though each step in the iterative methods only involves local communication in the physical domain, most problems require global communication to attain a correct solution. Elliptic problems are of this type [6].

The requirement for non-local, or global, communication is more apparent in direct methods. Factoring matrices by Gaussian elimination or Householder transformations can be performed as a sequence of rank-1 updates of the submatrix that remains to be factored. (Higher rank updates may yield better performance on some architectures.) For a dense

matrix, the pivot row is distributed to all the rows of the remaining submatrix, and the pivot column to all the remaining columns. For sparse matrices, the rank-1 update only affects the rows having non-zero entries in the pivot column, and the columns having non-zero entries in the pivot row. In Gaussian elimination one variable at a time is eliminated from the system of equations. Depending on the topology of the graph that the sparse matrix represents, the elimination process may only require local communication in the processor network, even for networks of bounded degree.

The problem of determining an address map that takes advantage of locality of reference in the physical domain, such that local communication in the processing network is possible, is often formulated as a graph embedding problem. A network of high degree local communication may be possible also when the references in the physical domain are non-local. For instance, divide-and-conquer methods for solving linear systems of equations, such as odd-even cyclic reduction, nested dissection, and multi-grid methods perform a recursive subdivision of the physical domain. For some of the recursion steps these methods require interaction between subdomains that are not adjacent in the physical domain. But, the references may still be performed by local communication in a network, such as a Boolean cube network. Any lattice can be embedded in a lattice of higher dimensionality preserving locality, but the converse is not true. It is also possible that the distance between a pair of lattice points is reduced when the lattice is embedded in a lattice of higher dimensionality. For instance, with a binary-reflected Gray code embedding of lattices in Boolean cubes [35,28] lattice points at a distance $2^j, j > 0$ in the index space are at distance 2 in the Boolean cube [16].

The communication requirements for explicit finite difference methods are determined by the grid and the difference stencils. Similarly, the elements, their order, and domain discretization determine the communication requirements for explicit finite element methods. The communication requirements for iterative solvers are determined by the adjacency matrix. The communication requirements for direct solvers can be determined by considering the graph underlying the (sparse) matrix, and by viewing variable elimination as node elimination in the graph [34]. The communication requirements are a function of the elimination order. The communications for the Fast Fourier Transform is identical to a butterfly network. Parallel cyclic reduction requires communication in the form of a data manipulator network or a PM2I (plus-minus 2^i) [37] network. Of regular communications, the emulation of multi-dimensional lattices, butterfly networks, data manipulator networks, pyramid networks, and various forms of trees for reduction and copy operations are the most common.

The potential benefits from exploiting locality of reference is illustrated by three frequently used operations: matrix multiplication, a 7-point symmetric difference stencil applied at each node in a three dimensional grid, and butterfly based computations (FFT, bitonic sort). Applying a symmetric, 7-point difference stencil at every point in a three dimensional grid with k variables per grid point and 2 operations per variable, the number of operations per remote reference is $r = \frac{1}{2d}(\frac{M}{k})^{\frac{1}{d}}$. For $d = 3$ $r = \frac{1}{6}(\frac{M}{k})^{\frac{1}{3}}$. Tables 1 and 2 give some values of r for different sizes of the local memories. In the tables, $k = 8$. If the

Computation	Registers only	4 Mbit chips	256 4 Mbit chips (board)	256 Boards
Mtx mpy	0.5	104	1600	26000
3-d Relaxation	0.17	4.27	26.7	170.7
FFT	1	18.8	28.8	38.8

Table 1: Number of operations per remote reference of a single variable.

Computation	4 Mbit 1 proc. 1 chip	256 Procs. = Board	256 boards = Machine
Mtx mpy	1	10	160
3-d relaxation	32	480	24600
FFT	3	1140	160000
no locality	300	76800	19660800

Table 2: Number of bits across the chip/board/system boundary per cycle.

local variables form matrices and the local operations imply matrix multiplications, then the number of arithmetic operations per variable is higher. Several linear algebra operations have a ratio of operations to remote references that can be modeled by the same expression as was given for the difference molecules, i.e. $\frac{1}{\alpha}(\frac{M}{\beta})^{\frac{1}{\gamma}}$ for suitable values of α, β and γ. In the Navier-Stokes flow computation, the local state vectors are of length 5 and local matrices typically of size 3×4, or 4×5, or some similar size [32]. For butterfly based algorithms, such as the Fast Fourier Transform (FFT) and sorting, the dependence is of the form $\alpha \log(\frac{M}{\beta})$. For the FFT the ratio is $1.25 log_2(M/2)$ real operations per remote reference using a radix-M algorithm, which is optimum [12].

Table 2 gives the number of bits that have to cross the chip, board, and system boundaries during a single cycle, assuming the optimum locality or no locality of reference. It is assumed that each chip has one processing unit, that a board has 256 processing units, and that all variables are in single precision.

Exploiting locality reduces the required communication bandwidth by a factor of 8–100 at the chip boundary for these computations, a factor of 80–5000 at the board level, and at least a factor of 125 at the I/O interface. A sustained performance in the Tflops/s range is possible with state-of-the-art technology only if locality is properly exploited.

3 Programming model

Architectures in which tens of thousands of operations can be performed concurrently are often referred to as *data parallel* to distinguish them from *control parallel* architectures, which offer a considerably lower degree of concurrency. Algorithms are designed based on the structure and representation of the problem domain. Objects in data parallel languages are represented by higher level data types such as the array extensions of Fortran 8X [31]. In a language with an array syntax, a number of nested loops (often equal to the number of axes in the array) disappear from the code, compared to a language without the array syntax. We illustrate this property by two examples. The first example is the implementation of a 7-point stencil in three dimensions. The second example is taken from a finite element code for stress analysis.

In the example below which defines the computation of a 7-point stencil at every point in a three dimensional grid, the operation CSHIFT defines a circular shift. The first argument is the variable to which the shift is applied, the second defines the axis along which the shift takes place, and the third argument defines the length and direction of the shift. Since there is no conditional statement in the code below, it implements periodic boundary conditions. Note that there are no explicit loops for the array axes.

```
      subroutine psolve(phi, omega, inside, n, iter)
      real phi(n, n, n), omega(n, n, n), factor
      logical inside(n, n, n)
      factor = 1.0/6.0
      do 100 i=1,iter,1
        phi = factor * (
     1     CSHIFT(phi, dim=1, shift=-1) +
     2     CSHIFT(phi, dim=2, shift=-1) +
     3     CSHIFT(phi, dim=3, shift=-1) +
     4     CSHIFT(phi, dim=1, shift=+1) +
     5     CSHIFT(phi, dim=2, shift=+1) +
     6     CSHIFT(phi, dim=3, shift=+1) ) +
     7     omega
  100 continue
      return
      end
```

In the finite element example below, the elements are brick elements of first order. There is one nodal point in each corner of an element. The state is represented by three displacements, $x = (u, v, w)$. The local interaction matrix, the elemental stiffness matrix, is a 3×24 matrix, with one row for each of the three components of the local displacement vector. The code segment also contains one compiler directive, SERIAL, which affects the data layout. The meaning will be explained later. The code fragment is from the iterative solver which requires the computation of a matrix vector product. In the particular finite

element code from which the code segment is selected, the elemental stiffness matrices are not assembled into a global stiffness matrix. Instead, a matrix vector product is performed for each element, and a total product vector assembled.

```
CMF$LAYOUT K(:SERIAL, :SERIAL, , , ), R(:SERIAL, , , ), X(:SERIAL, , , )
REAL K(3,24, 32, 32, 32), R(3,32,32,32), U(3,32,32,32), V(3,32,32,32), W(3,32,32,32), X(24,32,32,32)
CALL ALL-TO-ALL-ELEMENT-BROADCAST(U,V,W,X)
R = 0.0
DO I=1,24
  DO J=1,3
    R(J,:,:,:)=R(J,:,:,:)+K(J,I, :, :, :) * X(I, :, :, :)
  END DO J
END DO I
(WHERE (.NOT. I-RIGHT-BOUNDARY)) R=R + EOSHIFT(R, 1, 1)
(WHERE (.NOT. I-LEFT-BOUNDARY)) R= EOSHIFT(R, 1, -1)
(WHERE (.NOT. J-RIGHT-BOUNDARY)) R=R + EOSHIFT(R, 2, 1)
(WHERE (.NOT. J-LEFT-BOUNDARY)) R= EOSHIFT(R, 2, -1)
(WHERE (.NOT. K-RIGHT-BOUNDARY)) R=R + EOSHIFT(R, 3, 1)
(WHERE (.NOT. K-LEFT-BOUNDARY)) R= EOSHIFT(R, 3, -1)
```

In the above code segment, I-RIGHT-BOUNDARY, I-LEFT-BOUNDARY, etc. are boolean arrays which define the right–hand and left–hand boundaries of each finite element in the three dimensions respectively.

4 The Connection Machine Architecture

The Connection Machine [8] is a data parallel architecture. It has a total primary storage of 512 Mbytes using 256 kbit memory chips, and 2 Gbytes with 1 Mbit memory chips. The data transfer rate to storage is approximately 45 Gbytes/s at a clock rate of 7 MHz. The primary storage has 64k ports and a simple 1-bit processor for each port. The storage per processor is 8 kbytes for a total storage of 512 Mbytes and 64k bytes with 1 Mbit memory chips. The Connection Machine model CM–2 can be equipped with hardware for floating-point arithmetic. With the floating-point option, 32 Connection Machine processors share a floating-point unit, which is an industry standard, single chip floating-point multiplier and adder with a few registers. The peak performance available from the standard instruction set and the higher level languages is in the range 1.5 Gflops/s - 2.2 Gflops/s. The higher level languages do not at the present time make efficient use of the registers in the floating-point unit for operations that vectorize. With optimum use of the registers, a performance that is one order of magnitude higher is possible. For instance, for large local matrices, a peak performance in excess of 25 Gflops/s has been measured.

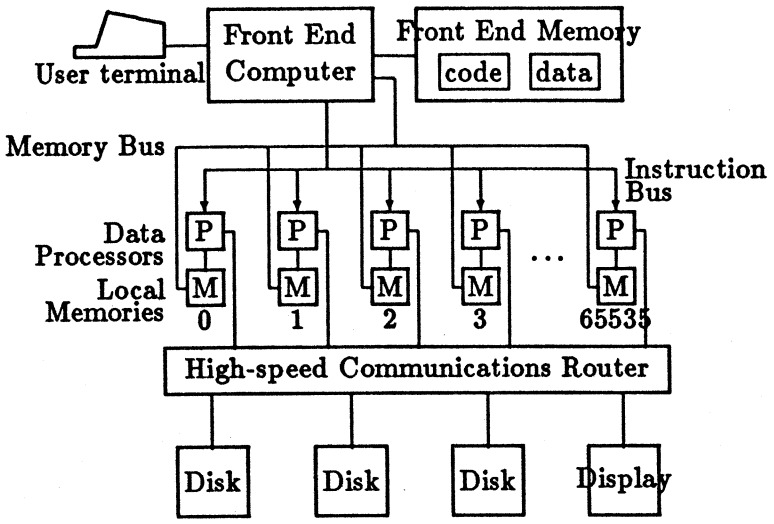


Figure 1: The Connection Machine System

The Connection Machine needs a host computer. Currently, three families of host architectures are supported: the VAX family with the BI-bus, SUN 4, and the Symbolics 3600 series. The Connection Machine memory is mapped into the address space of the host. The program code resides in the storage of the host. It fetches the instructions, does the complete decoding of scalar instructions, and executes them. Instructions to be applied to variables in the Connection Machine are sent to a microcontroller, which decodes and executes instructions for the Connection Machine. Variables defined by array constructs are allocated to the Connection Machine, unless allocation on the front-end is requested. The architecture is depicted in Figure 1. The Connection Machine can also be equipped with a secondary storage system known as the data vault. There exist 8 I/O channels, each with a block transfer rate of up to approximately 30 Mbytes/s. The size of the secondary storage system is in the range 5 Gbytes to 640 Gbytes. The Connection Machine can also be equipped with a frame buffer for fast high resolution graphics. An update rate of about 15 frames per second can be achieved.

The Connection Machine processors are organized with 16 processors to a chip, and the chips interconnected as a 12-dimensional Boolean cube. The communication is bit-serial and pipelined. Concurrent communication on all ports is possible. Through the bit-serial pipelined operation of the communication system, remote processor references require no more time than nearest neighbor references provided there is no contention for communication channels. For communication in arbitrary patterns, the Connection Machine is equipped with a router which selects one of the shortest paths between source and destination, unless all of these paths are occupied. The router has several options for resolving contention for communication channels.

4.1 Configuring the address space

The address field of the Connection Machine is divided into three parts: (off-chip|on-chip|memory). The off-chip field consists of 12 bits that encode the Connection Machine processor chips, the on-chip field encodes the 16 processors on each Connection Machine processor chip, and the lower order bits encode the memory addresses local to a processor. The lowest order off-chip bit encodes pairs of processor chips sharing a floating-point unit. On-chip communication is considerably faster than inter-chip communication. On-chip communication is a local memory reference. Off-chip communication is slower due to the limited bandwidth at the chip boundary. With the current chip (VLSI MOS) and interconnection (metal wire) technologies, such a characteristic is expected. The non-uniformity in access time impacts the optimum data allocation [15,17].

The default data allocation scheme on the Connection Machine first determines how many data elements need to be stored in each processor for an equal number of elements per processor, then stores that many successive elements in each processor, *consecutive storage* [15]. With the n highest order bits encoding the processors and the lower order bits encoding memory addresses in each processor, the consecutive assignment can be illustrated as follows.

$$\textit{Consecutive}\ \text{assignment:}(\underbrace{x_m x_{m-1} \cdots x_{m-n+1}}_{rp}\underbrace{x_{m-n} x_{m-n-1} \cdots x_0}_{vp})$$

The field denoted *rp* encodes *real processor* addresses as opposed to local *memory* addresses *vp*. For a data set of $M = 2^m$ complex points, $m+1$ address bits are required, n of which are processor address bits. There are $m - n + 1$ local storage address bits. Another frequently used address form is *cyclic* assignment, for which the lowest order address bits determine the *real processor* address.

$$\textit{Cyclic}\ \text{assignment:}(\underbrace{x_m x_{m-1} \cdots x_n}_{vp}\underbrace{x_{n-1} x_{n-2} \cdots x_0}_{rp}).$$

In the cyclic assignment, all data elements in a processor have the same n low order bits. In the consecutive assignment, the elements in a processor have the same n high order bits. The cyclic allocation scheme currently is not supported on the Connection Machine system. However, for some computations, it offers certain performance advantages [15,27].

Current implementations of the Connection Machine languages encode each axis of a multi-dimensional array separately. Each axis is extended to a length that is equal to some power of two. For an axis length P, $\lceil log_2 P \rceil$ address bits are assigned to the encoding of the elements along that axis. The consecutive allocation scheme is used for each axis. The encoding of the axes in the total address space attempts to configure each part of the address space (off-chip, on-chip, and memory) to conform with the array. To

the extent possible, all axes have a segment of each address field, and the ratio of the lengths of segments for different axes is the same as that of the length of the axes.

The default allocation of axes to off-chip, on-chip, and memory bits may not always be the preferred allocation. For a computation in which the interaction between virtual processors is equally frequent in each direction, the total amount of communication is minimized if the virtual processors assigned to a physical processor, or actually a processor chip, forms a single subdomain with an aspect ratio as close to one as possible [17]. The different Connection Machine languages provide different means for user controlled data allocation. In CM-Fortran compiler directives allow a user to specify an axis as SERIAL, which implies that the axis is allocated to a single processor. In PARIS (PARallel Instruction Set), the Connection Machine native language, a user has full control over what dimensions of the address space an axis occupies. But, only consecutive allocation of data to processors is supported.

If an array has fewer elements than the number of real processors in the configuration, the array is extended such that there is one element per real processor. In CM-Fortran an axis is added to the array with a length equal to the number of instances of the specified array that matches the number of real processors.

Array elements in the Connection Machine programming languages are often referred to as *virtual processors* [8,1]. In general, several virtual processors (array elements) are mapped to the storage of each physical processor. The number of virtual processors per physical processor is called the *virtual processor ratio* [1]. The storage of a physical processor is divided between as many virtual processors as is given by the virtual processor ratio. That many virtual processors time-share a physical processor.

4.2 Encoding of array axes

In the common binary, encoding successive integers may differ in an arbitrary number of bits. For instance, 63 and 64 differs in 6 bits, and hence are at a *Hamming* distance of 6 in the Boolean cube. A *Gray* code by definition has the property that successive integers differ in precisely one bit. The most frequently used Gray code for the embedding of arrays in Boolean cubes is a *binary-reflected* Gray code [15,28,35]. This Gray code is periodic. The code preserves adjacency for any loop (periodic one-dimensional lattice) of even length, and for loops of odd length one edge in the loop is mapped into a path of length two [15]. For the embedding of multi-dimensional arrays, each axis may be encoded by the *binary-reflected* Gray code. The embedding of an $N_1 \times N_2 \times \ldots \times N_d$ array requires $\sum_{i=1}^{d} \lceil \log_2 N_i \rceil$ bits. The *expansion*, i.e., the ratio between the consumed address space and the actual array size, is $2^{\sum_{i=1}^{d} \lceil \log_2 N_i \rceil} / \Pi_{i=1}^{d}$, which may be as high as $\sim 2^d$ [7,10]. The expansion can be reduced by allowing some successive array indices to be encoded at a Hamming distance of two. The *dilation* is the maximum Hamming distance between any pair of adjacent array indices. Every two-dimensional array can be embedded with minimum expansion and dilation 2 [3]. Minimum expansion dilation 2 embeddings for a

large class of two-dimensional arrays are given in [10], which also provides a technique for reducing the expansion of higher dimensional arrays. Minimal expansion dilation 7 embeddings are possible for all three dimensional arrays [4]. Embeddings with dilation 2 for many three dimensional arrays are given in [9].

The lattice emulation by a binary-reflected Gray code embedding is part of the standard programming environment on the Connection Machine system. In CM-Fortran, array axes are by default encoded in a binary-reflected Gray code for the off-chip segment of the address field. In the other Connection Machine languages, the Gray code encoding is invoked by configuring the Connection Machine as a lattice of the appropriate number of dimensions. The benefit of the lattice emulation feature is twofold: the virtual processors are assigned to physical processors such that the communication requirements are minimized, and lattice organized computations are often easier to express by virtue of programming constructs corresponding directly to the operations in the problem domain.

5 Scientific Libraries

A library of basic scientific routines on data parallel computers must be capable of handling two forms of concurrency: concurrent execution of multiple, independent problems, and concurrent execution of a single problem. In a single instruction stream architecture, such as the Connection Machine, the independent problems must require the same instructions for a good load balance. For instance, multiple matrix multiplications, multiple factorizations, or multiple FFT's can be performed concurrently. The need for multiple operations of the same type occurs frequently in the solution of partial differential equations on data parallel architectures, as illustrated in the next section. In this section, we give two specific examples of concurrency in a single operation by briefly describing how matrix multiplication and the FFT are performed in the library routines available on the Connection Machine.

5.1 Matrix Multiplication

The matrices to be multiplied are in general distributed across several, but not necessarily all, processors. In many cases, several different matrix products will be formed concurrently in disjoint sets of processors. Within each set, some data motion is required to compute a matrix product. Typically, each processor will have several matrix elements of each operand assigned to it. A suitable local matrix multiplication algorithm is required in addition to a global algorithm that implements the appropriate data motion given the allocation of the input and output matrices.

5.1.1 The basic algorithm

The current Connection Machine library routine for matrix multiplication is based on mesh emulation. The data motion for multiplying two matrices on a mesh can be partitioned into two phases; Alignment and Multiplication. The purpose of the alignment is to make sure that the range of "inner" indices for one matrix is a subset of the range of "inner" indices of the other matrix [15,21,20]. All processors can concurrently perform a multiplication and an addition. The data motion during the multiplication phase preserves this property. The essence of the data motion is best illustrated for the multiplication of two $P \times P$ matrices on a $P \times P$ mesh of processors [2].

forall $i, j \in \{0, 1, \ldots, P-1\} \times \{0, 1, \ldots, P-1\}$ **do**

Alignment:

$a(i, j) \leftarrow a(i, (i + j) \bmod P)$
$b(i, j) \leftarrow b((i + j) \bmod P, j)$

Muliplication:

$c(i, j) \leftarrow c(i, j) + a(i, j) \times b(i, j)$
for k:= 1 **to** $P - 1$
  $a(i, j) \leftarrow a(i, (j + 1) \bmod P)$
  $b(i, j) \leftarrow b((i + 1) \bmod P, j)$
  $c(i, j) \leftarrow c(i, j) + a(i, j) \times b(i, j)$
endfor k
endforall i, j

By a binary-reflected Gray code encoding of array indices, the data motion in the above algorithm only requires nearest neighbor communication. All array indices are extended to the nearest greater power of two. For the multiplication of matrices of arbitrary shapes on any size Boolean cube configured multiprocessor, it is necessary to generalize the algorithm to the following cases:

- The set of processors are configured with N_0 processors along axis zero, and N_1 processors along axis one, $N_0 \neq N_1$.
- A submatrix per processor instead of a single element
- Arbitrary P, Q, and R
- Parallelization of the loop on the "inner" index.

The first generalization is necessary for several reasons. One reason is that the number of processors N, to which the matrices are allocated, may not be a square. Another reason is that for small matrices and a large number of processors, the operands for a matrix product may only extend over a subset of processors, even if only a single element of each operand is assigned to each processor. In this case, as well as in the case where each operand has several elements assigned to each processor, the optimum configuration of the physical processors is an array of the same shape as that of the product matrix C [21]. The need for the second and third generalization is apparent. The last generalization is motivated by the fact that the number of processors N may be significantly greater than the number of elements PR of the product matrix C. For $N \geq PR$, all three loops in a Fortran 77 code for matrix multiplication by the standard algorithm may be parallelized. The third axis, i.e., the axis for the "inner index" can be instantiated partially, or totally, in space. With all three axes instantiated in space, the operands are assigned to orthogonal planes. For instance, the matrix A can be assigned to the plane defined by axes zero and two, the matrix B to the plane defined by axes one and two, and the matrix C to the plane defined by axes zero and one. The matrix A is copied along axis one, and the matrix B along axis zero. The matrix C is obtained by reduction along axis two.

5.1.2 Arbitrary Matrix Shapes and Sizes

We assume that a submatrix of each operand is stored in each processor to which any matrix element is assigned, and that the submatrices in different processors are of the same shape and size. We first define a matrix multiplication algorithm for a two-dimensional mesh of processors. The third axis, i.e., the axis of the inner index, is entirely instantiated in time. The set of processors are assumed to be configured with N_0 processors along axis zero and N_1 processors along axis one. If $N_0 \neq N_1$, then the range of the inner index for the two operands is clearly different in every processor. The alignment must assure that the inner index range for one operand is a subset of the inner index range of the other operand. This property must be maintained during the multiplication. In the case of the Connection Machine implementation, the ratio $\frac{N_0}{N_1} = 2^s$ for some integer s. For a small matrix and sufficiently many processors, the third axis can also be parallelized, and the set of processors are configured as an array with three axes.

Configuring the Connection Machine processors with the same number of processors along each axis of a two-dimensional array, and using Gray code encoding of the axes, allow the alignment and data motion during multiplication to be based entirely on processor addresses [18]. The data motion during the multiplication phase implements an *all-to-all broadcasting* [25] within rows for the matrix A and columns for the matrix B: $C \leftarrow C + A \times B$. The broadcasting is accomplished by cyclic rotation. Memory requirements are conserved which is an important property for the multiplication of matrices having large submatrices allocated to each processor. The minimum number of rotation steps along axis zero is $N_0 - 1$ and along axis one $N_1 - 1$. The algorithm implemented

on the Connection Machine moves a complete submatrix assigned to a processor when communication is needed. No local data motion is required. If $N_0 > N_1$, then $\frac{N_0}{N_1}$ rotation steps are performed along axis zero for every rotation step along axis one. Only a fraction of the local submatrix of the matrix subject to the fewest rotation steps is used in a local matrix-matrix multiplication for each rotation step along the longest processor axis. But, for the matrix subject to the largest number of rotation steps, the entire submatrix is used for each rotation step. Figure 2 illustrates the data motion of the matrices A and B for $N_0 = 4$ and $N_1 = 8$.

The alignment along the longest axis is performed as if the processor array was square with the number of processors along an axis equal to the number of processors along the shortest axis. The alignment along the shortest axis is performed as if the processor array was square with the number of processors along an axis equal to the number of processors along the longest axis. With this alignment, the multiplication can be accomplished by the minimum number of rotation steps along each axis [18].

With the third axis entirely instantiated in time, the matrix product requires $2\frac{P}{N_0}\frac{R}{N_1}Q$ arithmetic operations in sequence. The arithmetic time is proportional to the number of matrix elements per physical processor of the matrix C, and the length of the "third" axis, Q. The communication time for the multiplication phase is proportional to $\frac{PQ}{N_0}$ for the matrix A and to $\frac{QR}{N_1}$ for the matrix B. The alignment phase carried out as shifts along the axes of the mesh requires approximately the same amount of time as the multiplication phase if the cyclic shifts only can be performed in one direction at a time. For the Connection Machine implementation, the router is used for the alignment, and the time for large values of N_0 and N_1 is considerably less than the time required by a sequence of cyclic shifts. However, the time for alignment on a Boolean cube network can be further improved by optimizing the cyclic shifts [15].

The algorithm presented above is correct for all array shapes and sizes, and matrix shapes and sizes. However, the processor utilization can be improved when Q is small compared to P and/or R, and to N_0 and N_1. The matrix C is computed in-place. Only the set of processors to which C is allocated participate in the arithmetic operations. If $Q < R$, then only Q columns of C are computed concurrently. The set of Q columns is a function of the step of the algorithm such that at the end of the algorithm all columns of C are computed. Similarly, if $Q < P$, only Q rows are computed concurrently in any step.

By replicating A $\min(\frac{R}{Q}, \frac{N_1}{Q})$ times along axis one, and matrix B along axis zero $\min(\frac{P}{Q}, \frac{N_0}{Q})$ times, all processors to which the matrix C is allocated are used.

The optimum aspect ratio of the physical processor array is the same as the aspect ratio of the product matrix C. If there is only one matrix element of each operand per processor, then the optimum aspect ratio of the physical machine is 1, since it is desirable to minimize the maximum axis length: $N_0 = N_1 = \sqrt{N}$.

ALIGN

A

00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
22	23	24	25	26	27	20	21
32	33	34	35	36	37	30	31
44	45	46	47	40	41	42	43
54	55	56	57	50	51	52	53
66	67	60	61	62	63	64	65
76	77	70	71	72	73	74	75

B

00	11	22	33	44	55	66	77
10	21	32	43	54	65	76	07
20	31	42	53	64	75	06	17
30	41	52	63	74	05	16	27
40	51	62	73	04	15	26	37
50	61	72	03	14	25	36	47
60	71	02	13	24	35	46	57
70	01	12	23	34	45	56	67

ROTATE

←A

01	02	03	04	05	06	07	00
11	12	13	14	15	16	17	10
23	24	25	26	27	20	21	22
33	34	35	36	37	30	31	32
45	46	47	40	41	42	43	44
55	56	57	50	51	52	53	54
67	60	61	62	63	64	65	66
77	70	71	72	73	74	75	76

B

00	11	22	33	44	55	66	77
10	21	32	43	54	65	76	07
20	31	42	53	64	75	06	17
30	41	52	63	74	05	16	27
40	51	62	73	04	15	26	37
50	61	72	03	14	25	36	47
60	71	02	13	24	35	46	57
70	01	12	23	34	45	56	67

ROTATE

←A

02	03	04	05	06	07	00	01
12	13	14	15	16	17	10	11
24	25	26	27	20	21	22	23
34	35	36	37	30	31	32	33
46	47	40	41	42	43	44	45
56	57	50	51	52	53	54	55
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

↑B

20	31	42	53	64	75	06	17
30	41	52	63	74	05	16	27
40	51	62	73	04	15	26	37
50	61	72	03	14	25	36	47
60	71	02	13	24	35	46	57
70	01	12	23	34	45	56	67
00	11	22	33	44	55	66	77
10	21	32	43	54	65	76	07

Figure 2: Matrix multiplication on a 4×8 array.

$P \times Q \times R$	Gflops/s	$P \times Q \times R$	Gflops/s	$P \times Q \times R$	Gflops/s
$64 \times 4 \times 1$	7.84	$64 \times 4 \times 8$	9.20	$64 \times 4 \times 32$	9.38
$64 \times 8 \times 1$	11.98	$64 \times 8 \times 8$	13.53	$64 \times 8 \times 32$	13.71
$64 \times 16 \times 1$	16.35	$64 \times 16 \times 8$	17.74	$64 \times 16 \times 32$	17.88
$64 \times 32 \times 1$	17.35	$64 \times 32 \times 8$	18.02	$64 \times 32 \times 32$	18.11
$64 \times 64 \times 1$	19.08	$64 \times 64 \times 8$	19.55	$64 \times 64 \times 32$	19.61
$64 \times 256 \times 1$	20.38	$64 \times 256 \times 8$	20.51	$64 \times 256 \times 32$	20.52

Table 3: Performance data for the local matrix kernels.

5.1.3 Parallelizing the third axis.

The set of processors participating in the multiplication using the extended algorithm is approximately $\min(P, N_0) \times \min(R, N_1)$. If $P < N_0$, and/or $R < N_1$, only a subset of size $\sim PR$ of all N processors are used. If $N >> PR$, then substantially improved processor utilization can be achieved by also instantiating the third axis at least partially in space. To generate two instances in space, we partition the matrix A as $(A_0 A_1)$ and B as $\left(\frac{B_0}{B_1}\right)$, where A_0 and A_1 are $P \times \frac{Q}{2}$ matrices and B_0 and B_1 are $\frac{Q}{2} \times R$ matrices. Then, the products $A_0 B_0$ and $A_1 B_1$ are computed on disjoint sets of PR processors. Each set is configured as a mesh. By employing the algorithm above for each mesh, each product only requires approximately $\frac{Q}{2}$ steps. Denote the two meshes 0 and 1. Mesh 0 contains twice as many copies of A_0 and B_0 as before, but no copies of A_1 and B_1. Mesh 1 contains twice as many copies of A_1 and B_1 as in the case of a single mesh, and no copies of A_0 and B_0.

We introduce a third processor axis for the enumeration of the meshes of size $\sim PR$ each. The product $A \times B$ is obtained through a plus-reduction along the third axis, the axis of the inner index Q. The number of planes in the third dimension is $\min(Q, \frac{N_0}{P} \times \frac{N_1}{R}) = 2^{\min(q, n-p-r)}$ and the length of the "parallelized" inner axis is $\hat{Q} = 2^{\min(0, q+p+r-n)}$.

5.1.4 CM implementation issues

The local matrix multiplication kernel makes use of matrix-vector kernels. These kernels read a segment of a vector into the registers of the floating-point unit, then use it for a matrix-vector multiplication. The timings for a few matrix shapes are given in Table 3. The standard allocation scheme for arrays implies that, in general, the configuration of the physical machine is different for arrays of different shapes. However, for the algorithms described above, the operands need to be allocated assuming the same physical machine. The reconfiguration of the set of physical processors to a common shape is performed as part of the alignment. The alignment is performed by the Connection Machine router. Timings are given in Table 4.

$P \times Q \times R$	8k	16k	32k	64k
$128 \times 1024 \times 1024$	117	196		
$256 \times 1024 \times 1024$	190	321	582	968
$512 \times 1024 \times 1024$	281	496	825	1468
$256 \times 256 \times 256$	107	210	316	488
$512 \times 512 \times 512$	199	362	558	1062
$1024 \times 1024 \times 1024$	357	664	1045	1936
$2048 \times 2048 \times 2048$			1829	3463
$4096 \times 4096 \times 4096$				5814

Table 4: Performance in Mflops/s of the non-local matrix multiplication.

5.2 The Fast Fourier Transform (FFT)

The network interconnecting processor chips in the Connection Machine forms a 12-dimensional Boolean cube. In a Boolean cube of $N = 2^n$ nodes, n bits are required for the encoding of the node addresses. Every node $u = (u_{n-1}u_{n-2}\ldots u_m \ldots u_0)$ is connected to nodes $v = (u_{n-1}u_{n-2}\ldots \overline{u}_m \ldots u_0), \forall m \in [0, n-1]$.

A radix-2 butterfly network for P inputs and outputs has $P(p+1)$ nodes. Let the node addresses of the butterfly network be $(y_{p-1}y_{p-2}\ldots y_0|z_{t-1}z_{t-2}\ldots z_0)$, where $t = \lceil \log_2(p+1) \rceil$. The butterfly network is obtained by connecting node $(y|z)$ to the nodes $(y \oplus 2^{p-1-z}|z+1)$ and $(y|z+1)$, $z \in [0, p-1]$, where $\oplus$ denotes the bit-wise exclusive-or operation. For the computation of the radix-2 FFT the last t bits can be interpreted as time. The network utilization defined as the fraction of the total number of nodes that are active at any given time is $\frac{1}{t}$. During step z, the communication is between ranks z and $z+1$. Complex multiplications are made in rank z for decimation-in-time FFT and rank $z+1$ for decimation-in-frequency FFT.

By identifying all nodes with the same y value and different z values, node y becomes connected to nodes $y \oplus 2^z$, $\forall z \in [0, p-1]$, which defines a Boolean p-cube. All nodes participate in every step in computing an FFT on P elements on a p-cube. In step z, all processors communicate in dimension z. Only $\frac{1}{p}th$ of the total communications bandwidth of the p-cube is used. The full arithmetic power, instead of only half, can be used by splitting the butterfly computations between the pair of nodes storing the data. Each node performs 5 real arithmetic operations. This splitting of butterfly computations was implemented on the Connection Machine model CM-1. The parallel arithmetic complexity for computing an FFT on $P = 2^p$ complex elements on a Boolean n-cube becomes $5\lceil \frac{P}{N} \rceil \log_2 P$ real arithmetic operations, ignoring lower order terms. The speed-up of the arithmetic time is $\min(P, N)$ The communication complexity is $3\lceil \frac{P}{N} \rceil \log_2 N$ element exchanges in sequence.

In computing an FFT on P complex elements distributed evenly over $N = N < P$

processors, there are $\frac{P}{N}$ elements per processor. If the cyclic assignment is used, then the first $p-n$ ranks of butterfly computations are local to a processor. The last n ranks require inter-processor communication. For consecutive assignment the first n steps require inter-processor communication, and the last $p-n$ steps are local to a processor. If the data is allocated in a bit-reversed order, then the order of the inter-processor communication and the local reference phases are reversed.

The embedding defined above is the binary encoding of array indices. Every index is directly identified by an address in the address space. For arrays embedded by a binary-reflected Gray code array, elements that differ by a power of two greater than zero are at a distance of two, i.e., $Hamming(G(i), G(i+2^j)) = 2, j \neq 0$ [16]. Even though the elements to be used in a butterfly computation are at a Hamming distance of two, it is still possible to perform an FFT with $\min(p,n)$ nearest neighbor communications [19]. The current Connection Machine implementation assumes that the array axes are encoded in binary encoding. If the data is encoded by Gray code, then an explicit reordering to binary order is performed before the FFT computation. The Connection Machine router is currently used for this reordering. An optimum reordering is given in [15].

If there is only one element per processor, then every element is either involved in a computation or a communication. With multiple elements per processor, the communication efficiency can be increased from $\frac{1}{\min(p,n)}$ to $\frac{\min(p,n)}{n}$, which for $p > n$ is one. The increased communication efficiency is achieved by communicating concurrently in as many dimensions as possible.

5.2.1 Maximizing the communication efficiency

The radix-2 FFT implemented on the Connection Machine makes use of pipelining to achieve a high utilization of the communication (and computation) resources. For details of the implementation, and alternate implementations see [26,27]. High radix FFT's are discussed in [13]. With N processors performing $\frac{N}{2}$ butterfly computations concurrently, $\frac{P}{N}$ butterfly computations must be performed sequentially in each stage. In each of the first n butterfly stages, the lowest order $p-n$ bits are identical for the pair of data elements in a butterfly computation. The first n butterfly stages can be viewed as consisting of $\frac{P}{N}$ independent FFT's, each of size N with one complex data element per processor. This property was used in [14] for devising sorting algorithms on Boolean cubes. The independent FFT's can be pipelined. Every FFT performs communication in processor dimensions $n-1, n-2, \ldots, 0$. Each FFT is delayed by one communication with respect to the preceding one. After the n butterfly stages with inter-processor communication, the remaining $p-n$ stages are entirely local. The high order n bits identify N different FFT's of size $\frac{P}{N}$ each.

The number of complex data element transfers in sequence for the pipelined FFT is $n + \frac{P}{N} - 1$. The communication efficiency, measured as (the sum of the communication resources used over time)/((total number of available communication resources)*(time)),

Axis length	Time msec	Mflops /s
32	1.126	1455
64	2.184	1800
128	4.130	2222
256	8.326	2519
512	17.446	2705
1024	38.452	2727
2048	78.796	2928
4096	167.645	3002
8192	355.822	3065

Table 5: Performance for 2048 concurrent local radix-2 DIF FFT.

FFT	Time (msec)				Mflops/s			
	2k	4k	8k	16k	2k	4k	8k	16k
128×128	34.1	16.7	13.6	7.4	1075	1101	673	621
512×512	574	292	136	72	1315	1291	1390	1308
2048×2048			2213	1313			1668	1405
32×32× 32	99.5	50.4	24.2	13.7	791	780	811	720
64×64× 64	548	411	198	108	1378	919	956	875
128×128×128			1611	885			1093	995

Table 6: Performance for some two- and three-dimensional radix-2 DIF FFT.

for the stages requiring communication is $\frac{\frac{P}{N}}{n+\frac{P}{N}-1}$, $p \geq n$. The efficiency is approximately one for $\frac{P}{N} \gg n$.

Table 5 gives the performance for a collection of local, radix-2 FFT as a function of size for a Connection Machine system model CM–2. The same data are plotted in Figures 3 and 4. The performance data is for single precision floating point data. Both decimation-in-time and decimation-in-frequency FFT are implemented. Some sample timings for two- and three-dimensional radix-2 FFT are given in Table 5.2.1.

A higher radix FFT yields a better performance by a better utilization of the memory bandwidth, and a better load balance for the inter-processor communication stages [26,27]. Figure 5 illustrates the difference in performance for the FFT computations local to a processor for a mix of radix-4 and radix-8 kernels. The local maxima are for data sets of size 8^s for some s.

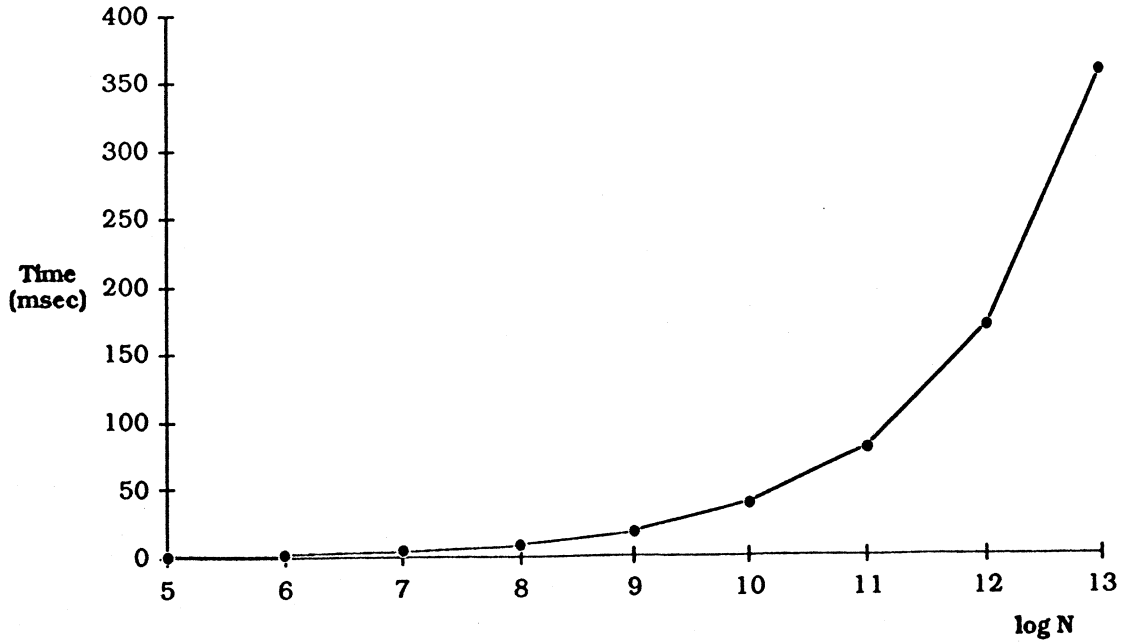


Figure 3: The execution time for local radix-2 FFT.

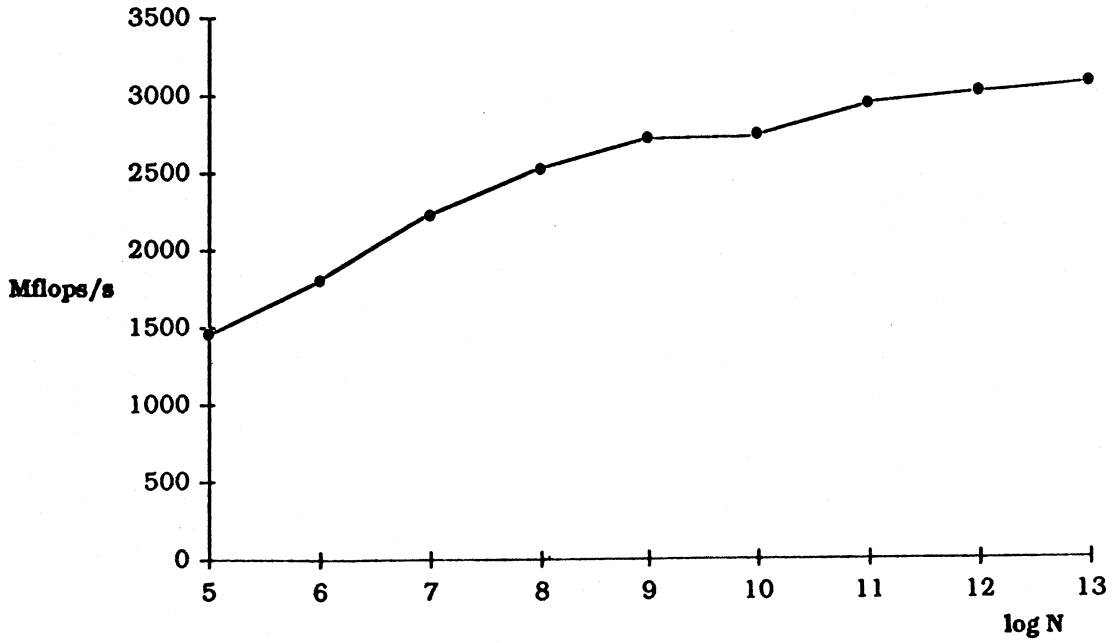


Figure 4: The floating-point rate for 2048 local radix-2 FFT's.

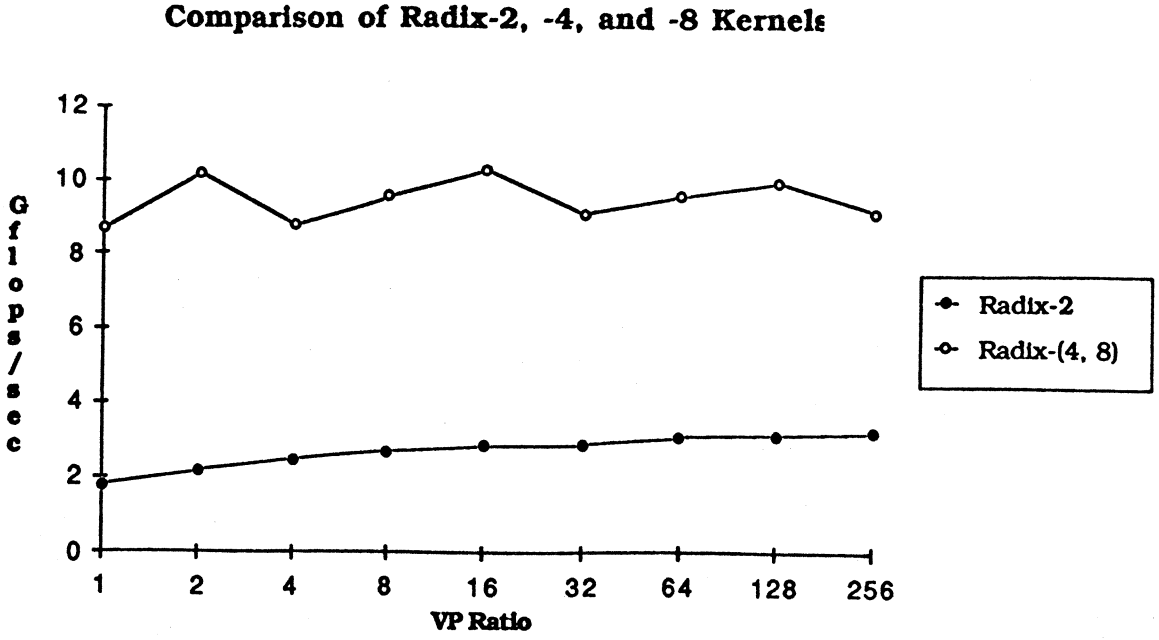


Figure 5: Performance of local radix-2 and radix-4/8 FFT computations.

6 Data Parallel Applications

In this section we present two applications. The first is the solution of the compressible Navier-Stokes equations, and the second the computation of the forward propagation of acoustic waves in the ocean. Both problems are formulated in three spatial dimensions. The purpose with these applications is to illustrate how problems are formulated for a data parallel computer and some of the functions that are needed. The examples are also intended to make it obvious that multiple concurrent instances of a computation often are both necessary and occur naturally. Both examples are based on finite difference techniques. An explicit technique is used for the Navier-Stokes problem, and an implicit technique for the underwater acoustics problem.

6.1 A compressible Navier-Stokes flow solver

The Navier-Stokes equation describes the balance of mass, linear momentum and energy, and models the turbulent phenomena that occur in viscous flow. In three dimensions, the equations in conservative form are

$$\frac{\partial \mathbf{q}}{\partial \tau} = \frac{\partial \mathbf{F} + \mathbf{F}_\nu}{\partial \xi} + \frac{\partial \mathbf{G} + \mathbf{G}_\nu}{\partial \eta} + \frac{\partial \mathbf{H} + \mathbf{H}_\nu}{\partial \zeta}, \tag{1}$$

where the variable vector $\mathbf{q}(\xi, \eta, \zeta, \tau)$ has five components: one for density, three for the linear momentum in the three coordinate directions x, y and z, and one component for the total energy. The coordinates of the physical domain are x, y and z, whereas ξ, η and

ζ are coordinates in the computational domain. $\mathbf{F}, \mathbf{G}$ and $\mathbf{H}$ are the flux vectors and $\mathbf{F}_\nu, \mathbf{G}_\nu$ and $\mathbf{H}_\nu$ are the viscous flux vectors.

For regular computational domains, the solution can be approximated by discretizing the domain by a three-dimensional grid. The grid may be stretched in order to get a good resolution of the boundary layers without an unnecessarily large number of grid points in the interior. A stretched grid is topologically equivalent to a regular grid, and any efficient embedding of such grids can be used advantageously. In [32,33], an explicit finite difference method is used. To stabilize the numeric method, artificial viscosity is introduced through a fourth order derivative. Centered difference stencils are used. The approximation of the derivatives of the flux vectors is second order accurate. There are two difference stencils being used in each lattice point, and the stencils vary for interior points, points on or close to a surface, edge, and corner. Given the directional dependence there are one interior stencil, six face stencils, 12 edge stencils, and eight corner stencils of second order accuracy. The number of different types of stencils increases with the order of approximation. A central difference stencil with $2N+1$ points in each of three dimensions gives rise to a total of $8N^3+12N^2+6N+1$ stencils because of the boundaries. For the derivates, $N=1$ in our case, and for the artificial viscosity, $N=2$, and the total number of stencils are 27 and 125, respectively. All these stencils are subgraphs of the stencil in the interior, and can be represented by a set of vectors [32]. A three step Runge-Kutta method is used for the integration.

For the computations, the Connection Machine was configured as a regular 3D-grid. There are approximately 170 variables per virtual processor (grid point). The maximum virtual processor ratio with $8k$ bytes per physical processor is 8. The subgrid for each processor is a $2 \times 2 \times 2$ grid. By using difference stencils on or close to the boundaries that are subgraphs of the stencils in the interior, the stencils can be implemented as EOSHIFT operations in Fortran 8X. With periodic boundary conditions, CSHIFT should be used. The measured bandwidth for nearest neighbor communication in this grid was on the average about 2.5 Gbytes/s for a Connection Machine with 64k processors. The execution time as a function of the virtual processor ratio and the machine size are given in Table 7. The aspect ratio of any pair of dimensions of the physical domain was either one or two, and the size ranging from $16 \times 16 \times 32$ to $64 \times 64 \times 64$.

From Table 7, it is clear that the execution time per time step is independent of the machine size, as expected. The execution time as a function of the virtual processor ratio is shown in Figure 6. The processor utilization increases by a factor of 2.75 as the virtual processor ratio increases from 1 to 8. The work increases by a factor of 8, but the execution time only by a factor of 2.9. Figure 7 shows the floating-point rate at a virtual processor ratio of 8. With this virtual processor ratio, grids with up to 524,288 points were simulated.

In the Navier-Stokes code, the operations in each virtual processor consist of stencil computations applied to vectors of length five. Each floating-point processor performs a three dimensional convolution (on vector arguments) within the physical subdomain

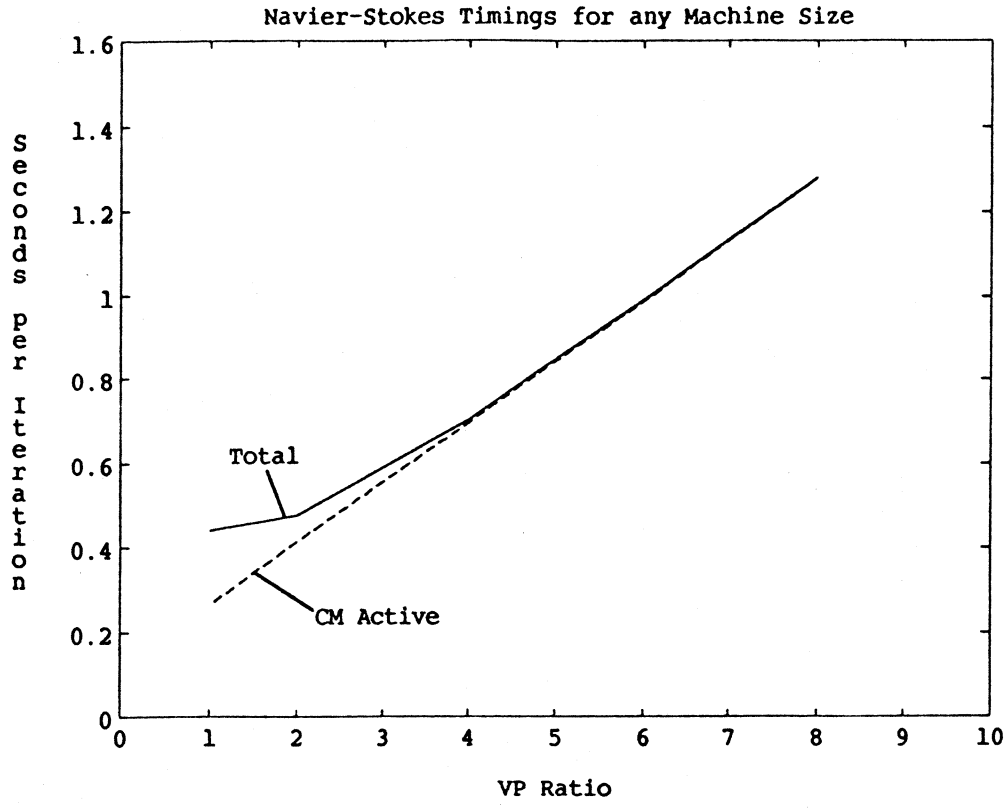


Figure 6: The Execution Time for a Single Time Step.

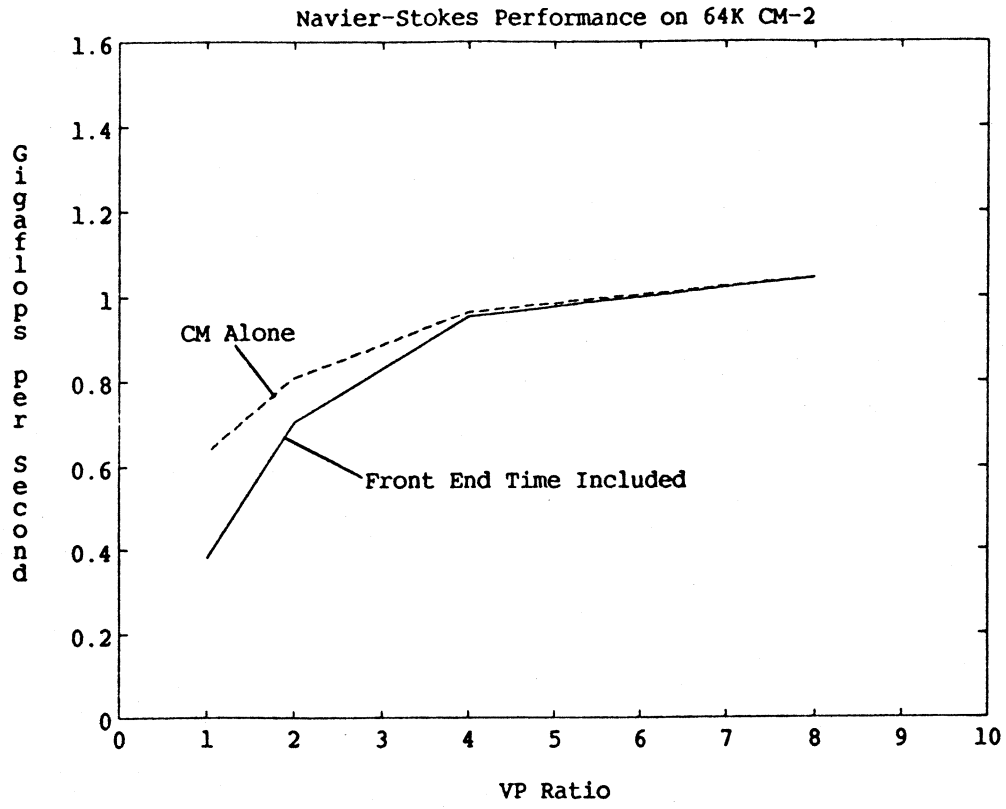


Figure 7: The Execution Speed for the Compressible Navier-Stokes Solver.

virtual processor ratio	Machine Size					
	8k		16k		32k	
	CM-time	Total time	CM-time	Total time	CM-time	Total time
1	2.61	4.43	2.61	4.42	2.61	4.42
2	4.12	4.74	4.12	4.74	4.12	4.74
4	7.07	7.08	7.04	7.05	7.01	7.02
8	12.70	12.70	12.83	12.83	12.83	12.83

Table 7: Execution Time for Different Virtual Processor Ratios.

mapped into the memories of the processors served by a floating-point unit. In addition, the computation of the flux vectors requires matrix-vector multiplication and matrix-matrix multiplication on small matrices and vectors in each virtual processor. Optimized routines for these operations were not available at the time this code was implemented and evaluated with respect to performance. Incorporating optimized routines is expected to increase the performance by a factor of about three.

6.2 Acoustic field computation by an Alternating Direction Method

The forward propagation of acoustic waves by the so called Wide Angle Wave Equation [30] implies the solution of an equation of the form

$$(1+\frac{1}{4}(1-\delta)X)(1-\frac{1}{4}Y)u(r+\Delta r) = (1+\frac{1}{4}(1+\delta)X)(1+\frac{1}{4}Y)u(r) \qquad (2)$$

where k_0 is a reference wave number, and $n(r,\theta,z) = k(r,\theta,z)/k_0$ $\delta = ik_0\Delta r$, and

$$X = \frac{1}{k_0^2}\frac{\partial^2}{\partial z^2} + (n^2(r,\theta,z)-1), \quad \text{and} \quad Y = \frac{1}{k_0^2 r^2}\frac{\partial^2}{\partial \theta^2}$$

This equation is a parabolic approximation of Helmholtz equation. The solution to the equation above can be marched out in the range (r) direction with an Alternating Direction Method [36,29]. Tridiagonal matrix-vector multiplications are performed in the θ and z directions, followed by the solution of tridiagonal systems in the same directions. Both operations consist of a number of one-dimensional problems that can be solved independently and concurrently. In addition, each system can be solved concurrently by substructuring, pipelined Gaussian elimination, partial or complete transposition of equations, and odd-even cyclic reduction, or any combination thereof [23] (which for multiple systems may be performed as balanced cyclic reduction). The communication pattern (in one dimension) of odd-even cyclic reduction is given in Figure 8. The communication pattern of balanced cyclic reduction is the same as that of parallel cyclic reduction [11].

The communication is defined by the grid and the difference stencil for the matrix-vector multiplication, but for the solution of the tridiagonal systems of equations, the communication depends on the selected algorithm. For pipelined Gaussian elimination, communication in the form of a Hamiltonian path is required. For equation transposition, the communication is equivalent to *all-to-all personalized communication* (or *all-to-some some-to-all* personal communication), which can be performed through butterfly network communication [22,25]. For balanced cyclic reduction, communication is required in the form of a data manipulator network. The communication requirements for odd-even cyclic reduction is a subtree of the data manipulator graph with the root at the top center and the leaf nodes being all nodes at the bottom level.

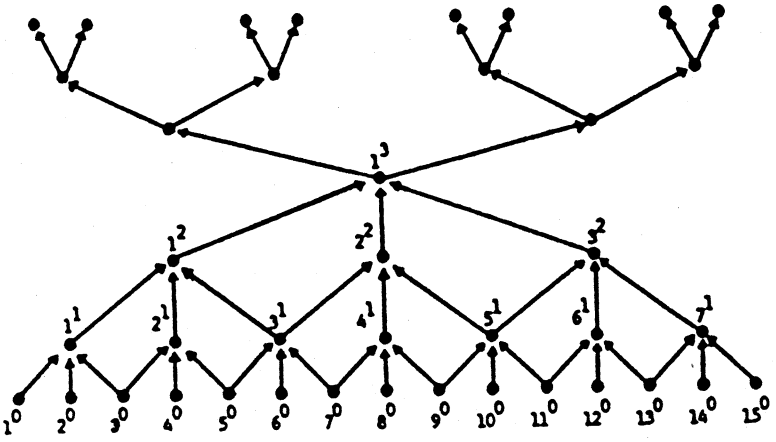
In the Connection Machine implementation, the processors are configured as a two dimensional grid. The tridiagonal systems are solved by substructured elimination followed by odd-even cyclic reduction for the reduced system of equations. The performance for the substructuring phase is about 1 Gflops/s without using any optimized library routines. The reduction phase in the current implementation uses a straightforward implementation of odd-even cyclic reduction. By using balanced cyclic reduction instead, a higher processor utilization and better performance can be achieved [24]. Note, that with the lattice emulation there is no need to perform a transposition of the data when the computations switch from one axis in the physical domain to the other. Communication time and storage accesses are the same for both directions.

7 Summary

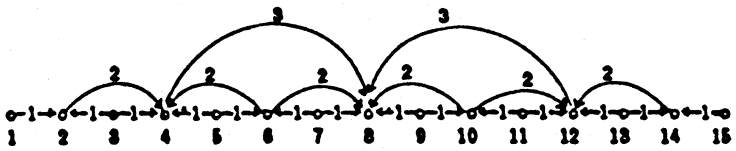
Supercomputers with a performance in the Tflops/s range are becoming technically and economically feasible to build in state-of-the-art technologies. Such computers will have thousands to tens of thousands of processing units interconnected by a bounded degree network. The most critical resources with respect to performance are the communication and memory subsystems. The efficient utilization of these resources is imperative to high performance. We have presented a few examples of how data allocation, data motion between processors, and algorithms can be chosen such that good utilization of data parallel architectures is accomplished. Performance data for matrix multiplication, Fast Fourier Transforms, a compressible Navier-Stokes solver, and an underwater acoustics code on the Connection Machine are provided.

Acknowledgement

The access to a Connection Machine system model CM –2 with 64k processors provided by the Advanced Computing Facility of the Los Alamos National Laboratories is gratefully acknowledged. The assistance provided by Ralph Brickner of the Advanced Computing Facility in carrying out the measurements was invaluable.



A computation graph for odd-even cyclic reduction.



A storage conserving computation graph for cyclic reduction.

Figure 8: The communication topology for odd-even cyclic reduction.

References

[1] *Lisp release notes. Thinking Machines Corp., 1987.

[2] L.E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm.* PhD thesis, Montana State Univ., 1969.

[3] M.Y. Chan. *Dilation-2 Embeddings of Grids into Hypercubes.* Technical Report UT-DCS 1-88, Computer Science Dept., University of Texas at Dallas, 1988.

[4] M.Y. Chan. *Embeddings of 3-Dimensional Grids into Optimal Hypercubes.* Technical Report, Computer Science Dept., University of Texas at Dallas, 1988. To appear in the Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications, March, 1989.

[5] Monty M. Denneau, Peter H. Hochschild, and Gideon Shichman. The switching network of the TF-1 parallel supercomputer. *Supercomputing Magazine*, 2(4):7–10, 1988.

[6] W. Morven Gentleman. Some complexity results for matrix computations on parallel processors. *J. ACM*, 25(1):112–115, January 1978.

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

[8] W. Daniel Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[9] Ching-Tien Ho and S. Lennart Johnsson. *Embedding Meshes in Boolean cubes by Graph Decomposition.* Technical Report YALEU/DCS/RR-689, Department of Computer Science, Yale University, March 1989.

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

[11] Roger W. Hockney and C.R. Jesshope. *Parallel Computers.* Adam Hilger, 1981.

[12] J.W. Hong and H.T. Kung. I/O complexity: the red-blue pebble game. In *Proc. of the 13th ACM Symposium on the Theory of Computation*, pages 326–333, ACM, 1981.

[13] Michel Jacquemin and S. Lennart Johnsson. *Radix-4 and radix-8 FFT on the Connection Machine.* Technical Report , Thinking Machines Corp., 1989. in Preparation.

[14] S. Lennart Johnsson. Combining parallel and sequential sorting on a Boolean n-cube. In *1984 International Conference on Parallel Processing*, pages 444–448, IEEE Computer Society, 1984.

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

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

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

[18] S. Lennart Johnsson, Tim Harris, and Kapil K. Mathur. *Multiplication of arbitrary matrices on a Data Parallel Computer.* Technical Report , Thinking Machines Corp., 1989. in Preparation.

[19] S. Lennart Johnsson and Ching-Tien Ho. *Emulating Butterfly Networks on Gray Code Encoded Data in Boolean Cubes.* Technical Report , Department of Computer Science, Yale University, 1989. in Preparation.

[20] S. Lennart Johnsson and Ching-Tien Ho. Expressing Boolean cube matrix algorithms in shared memory primitives. In *The Third Hypercube Conference*, pages 1599–1609, ACM, 1988.

[21] S. Lennart Johnsson and Ching-Tien Ho. Matrix multiplication on Boolean cubes using generic communication primitives. In *Parallel Processing and Medium Scale Multiprocessors*, pages 108–156, SIAM, 1989. (Presented at the ARMY workshop on Medium Scale Parallel Processing, Stanford University, January 1986).

[22] S. Lennart Johnsson and Ching-Tien Ho. Matrix transposition on Boolean n-cube configured ensemble architectures. *SIAM J. Matrix Anal. Appl.*, 9(3):419–454, July 1988.

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

[24] S. Lennart Johnsson and Ching-Tien Ho. *Optimizing Tridiagonal Solvers for Alternating Direction Methods on Boolean Cube Multiprocessors.* Technical Report YALEU/DCS/RR-679, Department of Computer Science, Yale University, January 1989.

[25] S. Lennart Johnsson and Ching-Tien Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers*, 38(9):1249–1268, September 1989.

[26] S. Lennart Johnsson, Ching-Tien Ho, Michel Jacquemin, and Alan Ruttenberg. Computing fast Fourier transforms on Boolean cubes and related networks. In *Advanced Algorithms and Architectures for Signal Processing II*, pages 223–231, Society of Photo-Optical Instrumentation Engineers, 1987.

[27] S. Lennart Johnsson, Robert L. Krawitz, Douglas MacDonald, and Roger Frye. *Cooley-Tukey FFT on the Connection Machine.* Technical Report , Thinking Machines Corp., 1989. in Preparation.

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

[29] S. Lennart Johnsson, Yousef Saad, and Martin H. Schultz. Alternating direction methods on multiprocessors. *SIAM J. Sci. Statist. Comput.*, 8(5):686–700, 1987.

[30] Ding Lee, Yousef Saad, and Martin H. Schultz. *An efficient method for solving the three-dimensional wide angle wave equation.* Technical Report YALEU/DCS/RR-463, Department of Computer Science, Yale University, October 1986.

[31] Michael Metcalf and John Reid. *Fortran 8X Explained.* Oxford Scientific Publications, 1987.

[32] Pelle Olsson and S. Lennart Johnsson. *A Dataparallel Implementation of Explicit Methods for the Three-dimensional Compressible Navier-Stokes Equations.* Technical Report CS-89/4, Thinking Machines Corp., February 1989.

[33] Pelle Olsson and S. Lennart Johnsson. *A Study of Dissipation Operators for the Eulers Equations and Three-Dimensional Channel Flow.* Technical Report CS-89/3, Thinking Machines Corp., February 1989.

[34] Seymor Parter. The use of linear graphs in Gaussian elimination. *SIAM Review*, 3(2):119–130, 1961.

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

[36] R. Richtmyer and K.W. Morton. *Difference Methods for Initial-Value Problems.* Wiley-Interscience, 1967.

[37] Howard J. Siegel. *Interconnection Networks for Large Scale Parallel Processing.* Lexington Books, 1985.