Massively Parallel Computing: Data distribution and communication

Massively Parallel Computing: Data distribution and communication

S. Lennart Johnsson

TR-29-92

December 1992

Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

This work has in part been supported by the Air Force Office of Scientific Research under contract AFOSR-89-0382, and in part by NSF and DARPA under contract CCR-8908285.

To appear in *Proceedings of the First International Heinz Nixdorf Symposium on Parallel Architectures and their Efficient Use*, Springer–Verlag, 1993.

Massively Parallel Computing: Data distribution and communication

S. Lennart Johnsson
Division of Applied Sciences
Harvard University
Cambridge, MA 02138
and
Thinking Machines Corp.
johnsson@harvard.edu

Abstract

We discuss some techniques for preserving locality of reference in index spaces when mapped to memory units in a distributed memory architecture. In particular, we discuss the use of multidimensional address spaces instead of linearized address spaces, partitioning of irregular grids, and placement of partitions among nodes. We also discuss a set of communication primitives we have found very useful on the Connection Machine systems in implementing scientific and engineering applications. We briefly review some of the techniques used to fully utilize the bandwidth of the binary cube network of the CM–2 and CM–200, and give some performance data from implementations of communication primitives.

1 Introduction

Massively parallel computing systems today all have the memory distributed among the processors. The processors with their memory modules and communication circuitry, collectively referred to as nodes, are interconnected by a network of some type. Networks currently in use include two– and three–dimensional meshes, binary cubes, two levels of rings, and fat–trees. Some systems are programmed as a shared memory machine, others have a shared address space but are programmed in a single program multiple data mode, while others still are programmed as a collection of local address spaces with message passing libraries for data interchange and synchronization. Much of the motivation for massively parallel architectures is their promise to deliver extreme performance. Locality of reference and routing are key issues in accomplishing this goal. The mapping of the index space to the memory units, the data allocation, determines the communication need. Below we briefly discuss some of the techniques we have employed for effective data placement and routing on the Connection Machine systems [60, 61].

Many problems in science and engineering are solved using regular discretizations in two, three, or more dimensions. Discretizing differential operators leads to difference *stencils* forming a weighted sum of values at grid points in a local neighborhood. The size and shape of the neighborhood is a function of the differential operator being approximated by a discrete representation and the order of the approximation. Stencils in the solution of (partial) differential equations serve the same role as convolution kernels in signal and image processing. Solving the resulting set of algebraic equations by an iterative method, such as Jacobi and SOR with various coloring schemes [8, 20], or carrying out the convolution through direct evaluation (as opposed to using Fourier transforms), requires communication in a local neighborhood of each grid point as defined by the stencil or convolution kernel (which may vary from grid point to grid point). The source of the communication requirement in these methods is a matrix–vector multiplication. Other methods, such as the conjugate gradient method, in addition, requires global communication in each step (through inner products). Hierarchical or divide–and–conquer methods, such as the multigrid method and the fast multipole and related methods [2, 6, 21], require communication over successively increasing (decreasing) distances in the index space. The communication is also often represented as quad–trees (two spatial dimensions) or oct–trees (three spatial dimensions). The Fast Fourier Transform also requires communication over successively increasing (decreasing) distances, as represented by a butterfly network. Well–known algorithms such as Gaussian elimination and QR factorization require global communication among (decreasing) subsets of nodes. Techniques for maximizing the size of the subsets (for load balance) for as large a part of the computation as possible are presented in [41, 42, 43].

Many problems in scientific and engineering computations yield solutions that cover a wide range of scales in the spatial domain, the time domain, or in both. Multiscale problems in the spatial domain are usually handled by nonuniform, and often unstructured, grids. Multiple scales in the time domain are handled by dynamic grids. Thus, the efficient handling of arbitrary discretizations, for instance, of complex three–dimensional geometries that may change over time is very important in many real scientific applications, such as the computation of the airflow around complete aircraft.

Matrix multiplication, using the standard algorithm requiring $2N^3$ arithmetic operations, is often used for evaluating compilers for conventional computers. It is a simple computation that also is a good example for understanding some of the critical operations on distributed memory architectures. Unlike the FFT, or relaxation methods, matrix multiplication involves three operands that typically are of different shapes. The relative allocation of the indices from the different operands has a profound impact on the performance. We will discuss this issue in some detail; in particular, we will consider how generic communication primitives may be employed.

The outline of this paper is as follows. We will first justify a simplified model of the memory hierarchy in massively parallel computer systems, then present some of the techniques used on the Connection Machine systems to choose a suitable data allocation with respect to the memory hierarchy and locality of reference. Then, we discuss some of the communication primitives we have found useful in programming the Connection Machine systems for performance and some of the techniques used to achieve high performance for these primitives.

2 Locality of reference – packaging technology

It is well–known that exploiting locality of reference may enhance performance significantly in many architectures. For vector architectures, such as the Cray YMP series, the use of techniques such as loop unrolling may yield a performance enhancement by a factor of two to five [11]. The experience on many cache–based architectures is that loop unrolling, loop partitioning and loop skewing may yield similar performance enhancements [40]. The reason for these performance enhancements is the underlying memory hierarchy, and the ability of the mentioned techniques to exploit this hierarchy.

The memory hierarchy in computer systems is largely determined by the storage and packaging technologies used. The latter introduces bottlenecks in the system. Today, a processor with a moderate size register set, floating–point arithmetic, and a small cache, all fit on a single chip. The ability to move data on a chip is considerably higher than the ability to communicate off chip. Chip packages typically have a few hundred pins, while there may be several thousand wire channels on each of a few layers on the chip. Each channel across the chip may be shared by several wires on different parts of the chip. Thus, the data motion capacity on a chip may be two orders of magnitude higher than the ability to move data between a chip and its environment. The situation is similar with respect to printed circuit boards. The dimensions of wires, pins, and boards are larger, but the ratio of on–board data motion capacity to off–board capacity is similar.

Thus, one set of issues a designer faces in choosing an interconnection network is the tradeoff between few but wide channels per node, or many narrow channels [9, 50]. The communications bandwidth of nodes with few, wide channels is often relatively easy to exploit, except that the associated network may exhibit severe contention for important computations, as, for instance, in computing the FFT on a mesh of low dimension, in particular a one–dimensional array. A higher dimensional array may support such computations with significantly less contention, but the capacity of each channel is less. The mapping of high dimensional arrays to lower dimensional arrays was studied, for instance, in [53]. Whichever design is preferable is a tradeoff between latency and bandwidth.

For fine grain architectures, hardware techniques have been devised to create low latency communication systems, such as in the Caltech Mosaic system [44, 15] and the MIT J-machine [10]. In systems of coarser granularity, there is often sufficient excess parallelism, or slack, to use pipelining and lookahead techniques to diminish the significance of the latency issue. We focus on data allocation for preservation of locality of reference, and thus, reduced need for communications bandwidth, and the efficient exploitation of the communications bandwidth in high degree networks, such as large binary cubes, through the use of multiple embeddings.

3 Data allocation

The data motion requirements, and the performance, depend strongly upon the data allocation. The Connection Machine systems are programmed with a global address space. Below, we first discuss the techniques used for allocating arrays used in matrix–

like computations, then we report on some early experiences with using a technique for allocation of irregular grids.

3.1 Regular arrays

On the Connection Machine systems, each array is by default distributed as evenly as possible over all nodes. In the default array allocation mode, the collection of nodes is configured for each data array as a nodal array with the same number of axes as the data array. The ordering of the axes is also the same. When there are more matrix elements than nodes, consecutive elements along each data array axis (a block) are assigned to a node. The ratios of the lengths of the axes of the nodal array are approximately equal to the ratios between the lengths of the axes of the data array [62]. The lengths of the local segments of all axes are approximately the same, and the number of off–node references minimized when references along the different axes are equally frequent, i.e., the surface area for a given volume is minimized. The default array layout is known as a *canonical layout*.

The canonical layout minimizes the data to be sent or received by each node in LU and QR factorization [43]. The canonical layout is also optimal with respect to communication for the standard matrix multiplication algorithm parallelized with respect to two of the three index axes [47]. The "standard" matrix multiplication algorithm in this context is the familiar textbook algorithm requiring $2N^3$ arithmetic operations for the multiplication of two $N \times N$ matrices. The multiplication can be performed by keeping either one of the three operands stationary. The communication requirements are minimized when the matrix with the largest number of elements is stationary. The nodal array shape for two–dimensional nodal arrays shall be congruent to the stationary matrix in order to minimize the off–node references [47]. In parallelizing the computations along all three axes, the optimal nodal array shape is congruent to the shape of the index space [32], i.e., the index space allocated to each node is as close to a cube as possible. For relaxation methods with an equal number of references along the different axes, the number of off–node references is minimized for a consecutive mapping [28] in which the lengths of the segments of the different axes mapped to a node are as equal in length as possible.

Remark 1: Note that the shape of the nodal array is important not only for communication operations but also for entirely local operations, since it effects strides and the lengths of axes, which are important with respect to loop overhead, vector lengths, cache hit ratios, and DRAM page faults. The arithmetic efficiency may vary by a factor of two or more.

Remark 2: Allocating contiguous segments of axes to a node, i.e., *consecutive allocation*, is a preferred way of aggregating data with respect to communication for computations in which nearest neighbor references dominate, such as in relaxation. In others, the consecutive allocation may yield poor load balance due to computations being nonuniform across the index space, or it may result in excessive communication. Thus, in the proposed High Performance Fortran standard [16], *cyclic* and *block–cyclic* allocation are also supported. In computations such as factorization and triangular system solution, the traversal of the index space can be matched with the chosen method of aggregation, thus creating an

equal load balance for consecutive and cyclic allocation [27, 43]. But, for computations where the order of traversal of the index space is fixed, such as the FFT, a cyclic allocation may reduce the communication needs by a factor of two [37, 38, 58, 64].

For a number of important computations on regular arrays, the canonical layout indeed minimizes the number of off–node references for a given number of data elements per node. However, when references along the different axes are not uniform, other nodal array shapes may result in a reduced number of off–node references. On the Connection Machine systems, the canonical layout can be altered through compiler directives. An axis can be forced to be local to a node by the directive SERIAL, if there is sufficient local memory. The length of the local segment of an axis can also be changed by assigning *weights* to the axes. High weights are used for axes with frequent communication and low weights for axes with infrequent communication. A relatively high weight for an axis increases the length of the local segment of that axis at the expense of the lengths of the segments of the other axes. The total size of the subarray is independent of the assignment of weights for sufficiently large arrays. Only the shape of the subarray assigned to a node changes.

In many computations, more than one array is involved and the relative allocations of the arrays are often important. For instance, in solving a linear system of equations, there are at least two arrays involved: the matrix to be factored and the set of right–hand sides. The ALIGN compiler directive may be used to assure that different data arrays are assigned to nodes using the same nodal array shape for the allocation. Alignment corresponds to a reshaping of the nodal array (compared to the canonical layout). The associated data motion corresponds to a generalized shuffle operation.

3.2 Irregular grids

The consecutive, cyclic, and block–cyclic allocation schemes are easily evaluated and implemented for computations with regular data reference patterns on one or multidimensional arrays. For data structures corresponding to irregular grids, partitioning the grid for preservation of locality of reference is a much more difficult task. Partitioning unstructured grids is still a very active area of research. On the Connection Machine systems we have recently implemented the so called spectral decomposition technique [12, 13, 14, 48, 54]. Fiedler showed that the second largest eigenvalue and the ordering of the associated eigenvector can be used for partitioning of a mesh.

Johan [25, 26] has used this technique for finite element computations on unstructured meshes. As model problems, Johan used a planar triangular mesh between an outer ellipse and an inner double ellipse and a nonplanar grid of tetrahedra between concentric cylinders. The planar grid had 8,307 nodes, 16,231 triangles, and 24,537 edges. The numbers of shared nodes and edges as a function of the number of partitions are given in Table 1. The grid for the concentric spheres consisted of 20,374 nodes, 107,416 tetrahedra, and 218,807 faces. Some of the data for the spectral decomposition of this mesh are summarized in Table 2.

The spectral decomposition technique is now being applied to large–scale finite element meshes with a million to several million elements. The results will be reported elsewhere.

Number of partitions	Number of shared edges	%of total	Number of shared nodes	% of total
8	188	0.8	195	2.4
16	381	1.6	396	4.8
32	752	3.1	773	9.3
64	1483	6.0	1479	17.8
128	2154	8.8	2101	25.3

Table 1: Partitioning of a planar mesh with inner boundary in the form of a double ellipse.

Number of partitions	Number of shared edges	%of total %of total	Number of shared nodes	% of total
8	5186	2.4	2735	13.4
16	8005	3.7	4095	20.1
32	11553	5.3	5747	28.2
64	16055	7.3	7721	37.9
128	21502	9.8	9827	48.2

Table 2: Partitioning of a tetrahedral mesh between concentric spheres.

3.3 Allocation of partitions

The consecutive, cyclic, and block–cyclic allocation schemes [28] select subsets of data elements to be assigned to the same node. Compiler directives, such as axis weight, SERIAL and ALIGN, address the issue of choosing the nodal array shape.

Another data layout issue is the assignment of partitions to nodes. The network topology and the data reference pattern are two important characteristics in this assignment. The nodes of the Connection Machine system CM–200 are interconnected as a binary cube with up to 11 dimensions. A binary cube network of n dimensions has 2^n nodes. It is well–known that regular grids are subgraphs of binary cubes and that binary–reflected Gray codes [52] generate embeddings of arrays into binary cube networks that preserve adjacency [28]. The binary–reflected Gray code is efficient, both in preserving adjacency and in node utilization, when the length of the axes of the data array is a power of two [22]. For arbitrary data array axes' lengths, the Gray code may be combined with other techniques to generate efficient embeddings [7, 24].

The binary–reflected Gray code embedding is the default embedding on the Connection Machine system CM–200 and is enforced by the compiler directive NEWS for each axis. The standard binary encoding of each axis is obtained through the compiler directive SEND. The binary encoding may be preferable for computations which require that elements differing in a single bit be operated upon together, such as in the FFT. However, in this particular case, the code conversion can be integrated into the algorithm at no increase in the communication time [36].

For unstructured grids, finding an optimal placement of the blocks is much less apparent, even if data references are predominantly local in the physical grid as in explicit methods for partial differential equations. Instead of attempting to preserve locality of reference, minimizing the contention in the communication system through randomized routing [65, 66] or randomized data allocation [49, 51] may be a viable strategy. In the Connection Machine Scientific Software Library [63], CMSSL, we provide randomization of the data allocation as an option, for instance, in sparse matrix–vector multiplication. The effectiveness of a random allocation with respect to performance is evaluated on gather/scatter operations discussed in Section 4.3.

3.4 Summary – data allocation

In summary, on the Connection Machine systems, consecutive data allocation is used by the compilers to aggregate data for a node. In addition to the consecutive allocation, High Performance Fortran will also support cyclic and block–cyclic allocation. The address space is treated as a multidimensional address space with as many axes as there are axes in the data array. The default shape of the local address space has local axes of approximately equal lengths. A binary–reflected Gray code encoding preserves adjacency in the index space when mapped to the binary cube network of the CM–2 and CM–200.

The shape of the local address space can be controlled through compiler directives. Such directives also allow for the choice of a binary axis encoding instead of the Gray code encoding for the CM–2 and CM–200. The CM–5 only supports binary encoding [61].

The spectral decomposition technique is provided as a means of partitioning unstructured grids for preservation of locality of reference for many computations on such grids. Randomization of the data allocation is supported as a means of reducing the contention in the communication system. Spectral decomposition and randomized allocation is offered in the form of library routines in the CMSSL. Randomized routing is used on the CM–5.

4 Communication primitives

The following is a list of communication primitives that we have found important in the programming of the Connection Machine systems.

- One–to–all reduction/copy
- All–to–all reduction/copy
- Gather/scatter
- One–to–all personalized communication
- All–to–all personalized communication
- Dimension(index) permutation

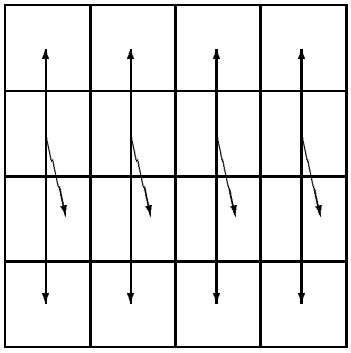

Figure 1: Broadcasting of a pivot row in LU decomposition.

- Generalized shuffle permutations
- Scan/parallel prefix
- Lattice emulation
- Butterfly emulation
- Data manipulator network emulation (PM2I network emulation)
- Pyramid network emulation
- Bit–inversion
- Index reversal ($i \leftarrow N - -i$)

Below we will discuss the need for some of the communication primitives above and the techniques used to achieve high bandwidth utilization.

4.1 Broadcast

Broadcast and reduction from a single source to subsets of nodes, holding an entire row or column, is critical for the efficiency of computations such as LU and QR factorization. In fact, many concurrent broadcast (and reduction) operations are desired in these computations as illustrated in Figure 1. Whether or not these broadcast operations imply communication that interfere, depends upon the network topology and how the index space is mapped to the nodes. On a binary cube network, entire subcubes are often assigned to a data array axis. In such a case, the broadcasts within the different columns in the index space do not interfere with each other, and the concurrent broadcast operation degenerates to a number of broadcasts within disjoint subsets of nodes. However, in other networks such a data mapping may not be feasible, and the simultaneous broadcast from several sources to distinct subsets of nodes may require a more complex routing for optimal bandwidth utilization.

Global broadcast and reduction is used, for instance, in the conjugate gradient method. But, even for the conjugate gradient method, several simultaneous broadcast operations

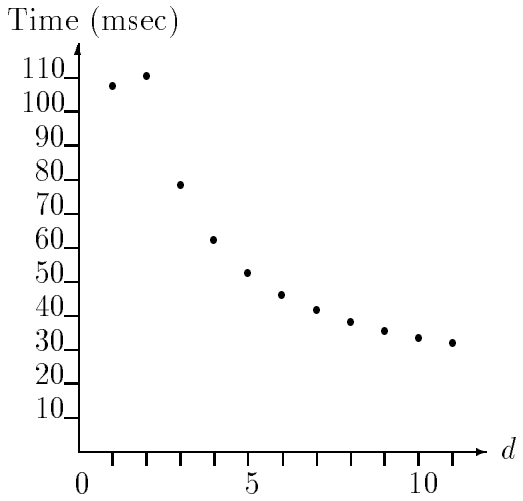

Figure 2: Time in msec for broadcast of 16k 32–bit data elements on Connection Machine system CM–200 as a function of number of cube dimensions.

may be required, since for massively parallel machines it is quite common that many computations of the same kind are performed concurrently, so called *multiple instance computation* [31].

Using a binomial tree to broadcast M elements from a node to all other nodes requires a time of nM with the communication restricted to one channel at a time, while the time is proportional to M with concurrent communication on all channels of every node, *all–port* communication. However, the lower bounds for the two cases are M and $\frac{M}{n}$, respectively [33]. Thus, the binomial tree algorithm is nonoptimal by a factor of n in both cases.

Multiple spanning trees rooted at the same node can be used to create lower bound algorithms [33]. The basic idea is: the source node splits its data set into $\frac{M}{n}$ disjoint subsets and sends each subset to a distinct neighbor. Then, each of these neighbor nodes broadcasts the data set it received to all other nodes (except the original source node) using spanning binomial trees. By a suitable construction of the trees, the n binomial trees are edge disjoint, and the full bandwidth of the binary n–cube is used effectively.

The multiple spanning binomial tree algorithm is used for broadcasting on the Connection Machine systems CM–2 and CM–200. The performance is illustrated in Figure 2 [29]. As expected, the time to broadcast a given size data set decreases with the number of nodes to which the set is broadcast.

4.2 All–to–all broadcast

Another important communication primitive is the simultaneous broadcast from each node in a set to every other node in the set, *all–to–all broadcast*. This communication is typical for so called direct N–body algorithms, but it is also required in many matrix algorithms. Here we will illustrate its use in matrix–vector multiplication.

With the processing nodes configured as a two–dimensional nodal array for the matrix, but as a one–dimensional nodal array for the vectors, both all–to–all broadcast and all–to–all reduction are required in evaluating the matrix vector product. Figure 3 illustrates the data allocation for both row major and column major ordering of the matrix. The data allocation shown in Figure 3 is typical on Connection Machine systems.

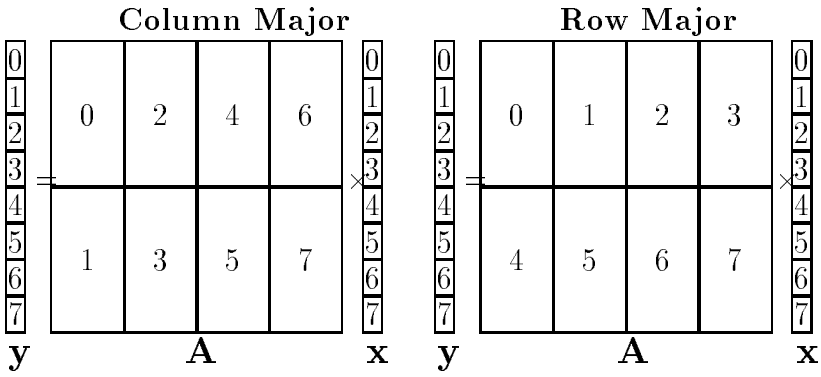

Figure 3: Data allocation on a rectangular nodal array.

For a matrix of shape $P \times Q$ allocated to a two–dimensional nodal array in column major order, an all–to–all broadcast [18, 33, 55, 56] is required within the columns of the nodes for any shape of the nodal array and for any length of the matrix Q–axis.

After the all–to–all broadcast, each node performs a local matrix–vector multiplication. After this operation, each node contains a segment of the result vector y. The nodes in a row contain partial contributions to the same segment of y, while different rows of nodes contain contributions to different segments of y. No communication between rows of nodes is required for the computation of y. Communication within the rows of the nodes suffices.

The different segments of y can be computed by all–to–all reduction within processor rows, resulting in a row major ordering of y. But, the node labeling is in column major order, and a reordering from row to column major ordering is required in order to establish the final allocation of y. Thus, for a column major order of the matrix elements, matrix–vector multiplication can be expressed as:

All–to–all broadcast of the input vector within columns of nodes
Local matrix–vector multiplication
All–to–all reduction within rows of nodes to accumulate
partial contributions to the result vector
Reordering of the result vector from row major to column major order.

All–to–all broadcast or reduction is required also when a one–dimensional nodal array configuration is used for the matrix [45].

In all–to–all broadcast, the lower bound for an n–cube where each node initially holds M elements is $M(N-1)$ for communication restricted to a single port at a time and $\frac{M(N-1)}{n}$ for all–port communication [33]. A simple yet optimal, all–port, algorithm for all–to–all broadcast uses n Hamiltonian paths for each node. For all–to–all broadcast, the Hamiltonian paths need not be edge–disjoint. Finding and constructing edge–disjoint cycles is a complex problem [1].

A Hamiltonian cycle can be constructed by moving across cube dimensions according to the transition sequence in a binary–reflected Gray code. Translating such a cycle to another source node by performing an exclusive–or operation on every node address by the source node index, assuming the first cycle has node zero as its source, creates $2^n = N$

Problem	Gather		Scatter	
	std alloc.	random alloc	std alloc.	random alloc
3200 20–node brick elements	75	50	124	55
864 8–node brick elements	5.6	3.7	7.2	3.4

Table 3: The effect of randomization on gather and scatter performance. Times in msec on an 8k CM–200.

paths. These paths cannot be edge–disjoint. But, it can be shown, that for all–to–all broadcast, there is no contention between data packets moving along the paths generated by a binary–reflected Gray code. Moreover, it can be shown that paths generated by rotating the address bits in the Gray code can be used without contention [33], thus providing n paths for each node. The initial data set M in each node is divided into n packets of approximately equal size. Each subset is then exchanged with packets in all neighboring nodes in each step. Figure 4 illustrates the idea. The use of n Hamiltonian paths is only one of several routing schemes that yield the same complexity for all–to–all broadcast [3, 17, 33, 55, 56]. The n Hamiltonian path algorithm has been implemented on the Connection Machine systems [5, 45]. Figure 5 shows the performance of the CM–200 implementation.

4.3 Gather/Scatter

Gather and scatter operations on regular grid data represented as one or multidimensional arrays, as well as irregular grid data, is critical for the performance of many scientific and engineering applications. On the Connection Machine Systems, gather and scatter operations on regular grids are supported through PSHIFT (for polyshift) [19, 63], which allows the programmer to specify concurrent shift operations in one or both directions of one or multiple axes. On the CM–2 and CM–200 with Gray coded axes, PSHIFT concurrently performs all the data exchanges requiring communication between nodes. In effect, PSHIFT provides an effective means of emulating lattices on binary cubes. A further level of optimization is provided by the so called stencil compiler [4], which in addition to maximizing the concurrency in internode communication (using PSHIFT), avoids unnecessary local memory moves and uses a highly optimized register allocation in order to minimize the number of load and store operations between local memory and the register file in the floating–point unit.

For gather and scatter operations on unstructured grid computations, the general router on the Connection Machine systems is used. However, two means of improving the performance is provided, one being randomization of the data allocation, and a second being savings of the routing information for repetitive gather scatter operations. Table 3 [46] summarizes the effect of randomization of the data allocation for a few meshes.

In these examples, the performance enhancement is a factor of 1.5 – 2.25, which in our

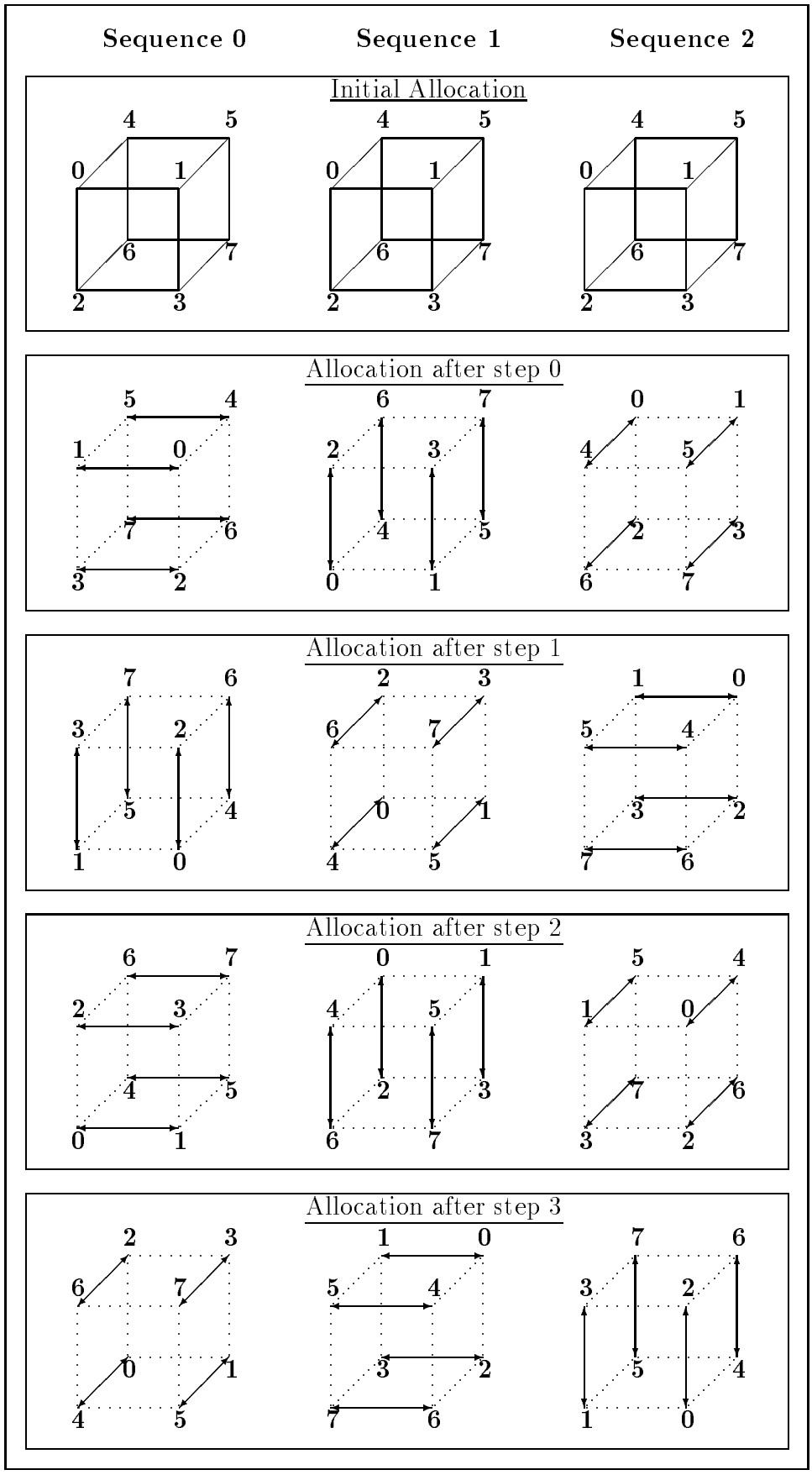

Figure 4: Three concurrent exchange sequences in a 3-cube.

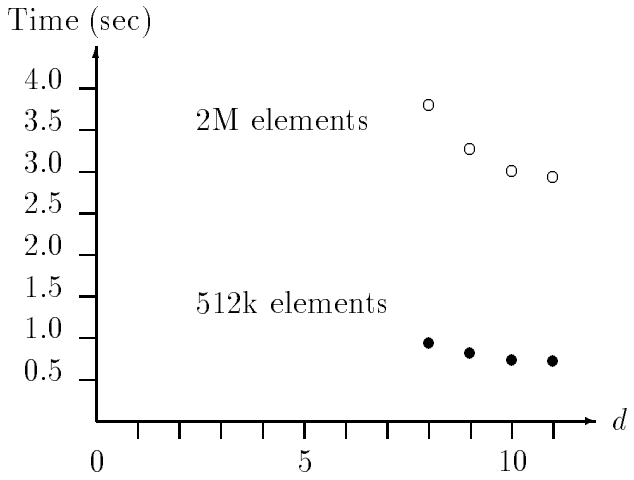

Figure 5: Time for physical all–to–all broadcast on a Connection Machine system CM–200. Array sizes are $2M = 2^{21}$ and $512K = 2^{19}$ 32–bit elements.

experience is fairly typical. In some cases, the performance improvement has been larger. It is rarely the case that randomization has caused a performance degradation.

4.4 Personalized communication

In *one–to–all personalized communication* a node sends a unique piece of data to every other node. An example is matrix computations where a node holds an entire column, which may need to be redistributed evenly over all the nodes, as in some algorithms for matrix–vector multiplication [45]. In all–to–all personalized communication, each node sends unique information to all other nodes. Personalized communication is not limited to matrix transposition but encompasses operations such as bit–reversal, transposition or bit–reversal combined with a code change (such as the conversion between binary code and binary–reflected Gray code) and bit–inversion. We now illustrate the significance of personalized communication in computing the FFT on a multiprocessor.

In computing the FFT on distributed data, one possibility is to exchange data between nodes, and have one of the nodes in a pair compute the "top" and the other compute the "bottom" of the butterfly requiring data from the two nodes. This type of algorithm is currently used on the Connection Machine systems CM–2 and CM–200 [39]. When there are two or more elements per node, then an alternative is to perform an exchange of data between nodes such that each node in a pair computes one complete butterfly. The sequence of exchanges required for the FFT amounts to a shuffle, as illustrated below, where the | separates node address bits to the left and local memory address bits to the right:

Example 1.

Address	Index
$(54321\|0)$	$(54321\|0) = (54321\|x_0)$
$(04321\|5)$	$(x_0a_4a_3a_2a_1\|a_5)$
$(05321\|4)$	$(a_4x_0a_3a_2a_1\|a_5)$

$$
\begin{array}{ll}
(05421|3) & (a_4a_3x_0a_2a_1|a_5) \\
(05431|2) & (a_4a_3a_2x_0a_1|a_5) \\
(05432|1) & (a_4a_3a_2a_1x_0|a_5) \\
(15432|0) & (a_4a_3a_2a_1a_5|x_0)
\end{array}
$$

Thus, the end result of the sequence of exchanges is a shuffle on the node address field. Each step is equivalent to the transposition of a collection of 2×2 matrices.

In practice, for a one–dimensional transform, there are typically several local memory bits. For performance, under many models for the communication system, minimizing the number of exchange steps is desirable, i.e., instead of performing bisections as in the example above, it is desirable to perform multisections including all local memory bits. Thus, for instance, with two local memory bits four–sectioning is being used as shown in Example 1. With three local memory bits, eight–sectioning is used as shown in Example 3.

Example 2.

$$
\begin{array}{rl}
\text{Address} & \text{Index} \\
(65432|10) & (65432|10) = (65432|\underbrace{x_1}_{j1}\underbrace{x_0}_{j0}) \\
(10432|65) & (\underbrace{x_1x_0}_{m}a_4a_3a_2|\underbrace{a_6a_5}_{m}) \\
(10652|43) & (\underbrace{\underbrace{a_4a_3}_{m}\underbrace{x_1x_0}_{m}}_{k\cdot m}\underbrace{a_2}_{n-k\cdot m}|\underbrace{a_6a_5}) \\
(10654|23) & (\underbrace{a_4a_3a_2}_{n-m}\underbrace{x_0}_{(k+1)m-n}\underbrace{x_1}_{n-k\cdot m}|\underbrace{a_6a_5}_{m}) \\
(23654|10) & (\underbrace{a_4a_3a_2}_{n-m}\underbrace{a_6}_{(k+1)m-n}\underbrace{a_5}_{n-k\cdot m}|\underbrace{x_0x_1}_{m}) \\
(23654|10) & (\underbrace{a_4a_3a_2}_{n-m}\underbrace{a_6}_{(k+1)m-n}\underbrace{a_5}_{n-k\cdot m}|\underbrace{x_1x_0}_{m})
\end{array}
$$

Example 3.

$$
\begin{array}{rl}
\text{Address} & \text{Index} \\
(6543|210) & (6543|210) \\
(2103|654) & (\underbrace{x_2x_1x_0}_{m}a_3|\underbrace{a_6a_5a_4}_{m})
\end{array}
$$

$$(2106|354) \qquad (\underbrace{a_3}_{n-m}\ \underbrace{x_1x_0}_{(k+1)m-n}\ \underbrace{x_2}_{n-k\cdot m}\ |\ \underbrace{a_6a_5a_4}_{m})$$

$$(3546|210) \qquad (\underbrace{a_3}_{n-m}\ \underbrace{a_6a_5}_{(k+1)m-n}\ \underbrace{a_4}_{n-k\cdot m}\ |\ \underbrace{x_1x_0x_2}_{m})$$

$$(3546|210) \qquad (\underbrace{a_3}_{n-m}\ \underbrace{a_6a_5}_{(k+1)m-n}\ \underbrace{a_4}_{n-k\cdot m}\ |\ \underbrace{x_2x_1x_0}_{m})$$

Examples 2 and 3 were deliberately chosen such that the exchanges cannot simply be treated as digit exchanges with increased radix for the digit but must indeed be treated as exchanges with digits of different radices. Moreover, the last few exchange steps were made such that the final order represents an m–step shuffle on the nodal address bits, where m is the number of bits used to encode the first exchange. This node address ordering requires a local memory shuffle to restore the original local memory ordering. In practice, it may, in fact, be preferable to avoid the local memory reordering by performing the last exchange such that local memory is normally ordered, which would leave the node addresses in an order corresponding to two shuffles: one m–step shuffle on all n node address bits, one n mod m shuffle on the last m bits.

All the above illustrations are made for the case where the indices are encoded in a binary code. With the part of the data index assigned to the node address field encoded, for instance, in binary–reflected Gray code, the actual communication pattern must account for the fact that the butterfly computations are made on bits in binary encoding, while the data allocation uses Gray code. In [36] we show how the butterfly emulation based on multisectioning of Gray coded data on a binary cube can be performed in the same time as if the data had been binary coded.

In multidimensional FFT, all of local memory should be considered in performing the multisectioning. For details see [30].

The FFT example above makes use of a sequence of all–to–all personalized communications. Before briefly discussing some algorithms for this type of communication in binary cubes, we consider the communication required to restore the original index order for the FFT. In the case of the FFT, the bits in the encoding of the output indices are computed in reverse order. Thus, what is required to establish a normal index map in the frequency domain is an unshuffle with bit–reversal. Figure 6 [30] shows a particular example.

The lower bounds for all–to–all personalized communication with M data elements per node is $\frac{nMN}{2}$ for communication restricted to a single port per node and $\frac{MN}{2}$ for all–port communication [33]. The corresponding bounds for one–to–all personalized communication are $(N-1)M$ and $(N-1)M/n$, respectively. Balanced spanning trees [23] provide for optimal one–to–all and all–to–all personalized communication in communication systems allowing all–port communication. A balanced spanning tree has N/n nodes in each of the n subtrees of the root. The use of n rotated spanning binomial trees rooted in each node also yields the desired complexity. Algorithms for both one–to–all and all–to–all personalized communication are discussed in [33]. In our FFT example above, several all–to–all personalized communications were performed in succession. In such a case, it may be of interest to minimize the time elements are in transition from source to destination in or-

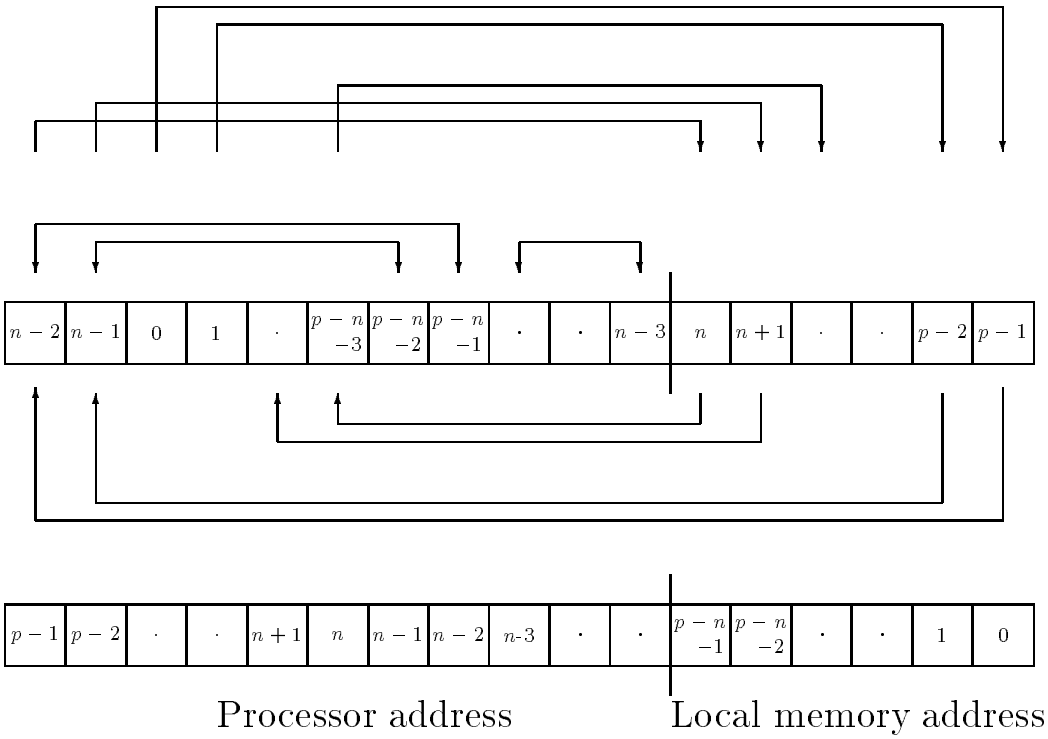

Figure 6: Two step reordering after 4–section based radix–2 FFT. First step, bit–exchange between nodes; second step, bit–exchange between local memory and node addresses. $p \leq 2n - m$.

der to minimize pipeline delays, if the communication system allows successive all–to–all personalized communications on different nodal dimensions to be pipelined. Algorithms with a minimal transition time are presented in [35].

Bit–reversal with an equal number of dimensions assigned to the node address field and the local memory address field constitutes one form of all–to–all personalized communication. The performance on various sizes of the CM–2 is shown in Figure 7 [29]. As expected, the execution time is almost independent of the machine size for a fixed size data set MN. The increase in the execution time is largely due to the fact that local memory operations cannot be performed in parallel. Thus, there is a term proportional to n in addition to the constant term.

4.5 Index reversal – bit–inversion

Index reversal is another important permutation used for instance in the computation of real–to–complex FFT. For this computation the standard algorithm requires that data with indices i and $N-i$, $0 \leq i < N$, be operated upon in a preprocessing or postprocessing step for the FFT [57, 59]. In binary–coded data, the index reversal required for the FFT corresponds to a two's–complement subtraction (bit complement plus one).

However, in the case of the real–to–complex FFT on a one–dimensional array with binary–coded data, the first step in one of the most common algorithms is to perform a complex–to–complex FFT on the array viewed as a half–size, one–dimensional, array of complex data points. The result is shown in Figure 8. The Figure also shows that the postprocess-

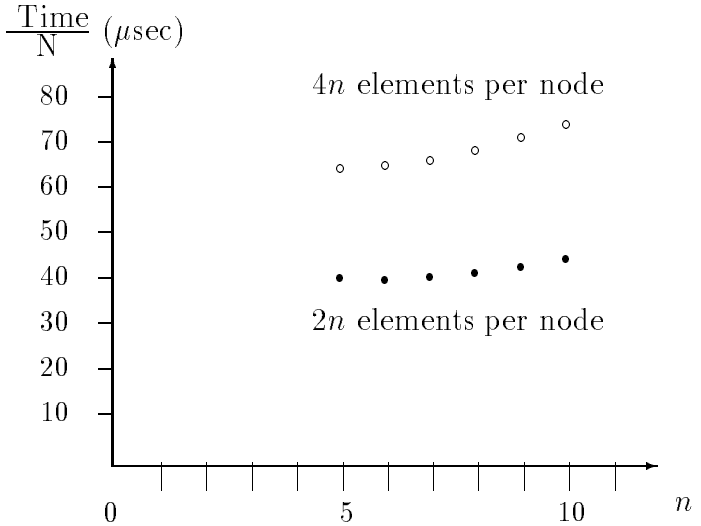

Figure 7: All–to–all personalized communication on the Connection Machine.

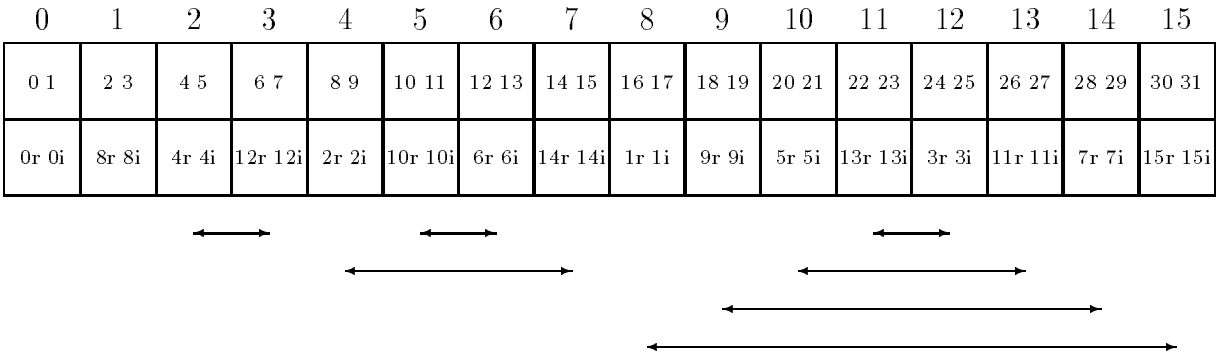

Figure 8: Postprocessing for real–to–complex FFT. Bit–inversion in subcubes.

ing, matching indices i and $N - i$, corresponds to bit–inversion in subcubes of the form $00 \ldots 01xx \ldots x$, with the inversion being performed on the bits denoted by x.

If there are more than one complex data point per node, then the communication requirements depend upon how the indices are aggregated to the nodes. In consecutive data allocation, the communication pattern between nodes remains the same. In a cyclic data allocation, the communication for the first complex local memory location is as outlined above; the communication for the second and all subsequent complex local memory locations is bit–complement on the entire node address.

Bit–inversion also occurs in the alignment of the operands in matrix–matrix multiplication on three–dimensional nodal array configurations. For details see [32].

Concurrent communication for bit–inversion on binary cubes is straightforward. For instance, multiple exchange sequences starting in different dimensions and progressing through the dimensions in increasing (or decreasing) order cyclicly can be used.

5 Shuffle operations on binary cube networks

In considering FFT computations through the use of multisectioning, we noticed that the effect on the original index space corresponds to a m–step shuffle, where m is the number of bits encoding the first digit exchange. For the FFT computation the various digit exchanges are interleaved with computation. Restoring the original index order corresponds to an unshuffle (except for the FFT which in itself implements a bit–reversal). Reshaping the nodal array for a given data array also represents a general shuffle operation. For instance, changing the allocation

$$(\overset{a_3}{y_2}\overset{a_2}{y_1}\overset{m_2}{y_0}\overset{a_1}{x_3}\overset{a_0}{x_2}\overset{m_1}{x_1}\overset{m_0}{x_0})$$

to the allocation

$$(\overset{m_2}{y_2}\overset{m_1}{y_1}\overset{m_0}{y_0}\overset{a_3}{x_3}\overset{a_2}{x_2}\overset{a_1}{x_1}\overset{a_0}{x_0}),$$

where x_i and y_i denote bits encoding an x–axis and y–axis respectively, and a_i denotes nodal address bits and m_i local memory address bits, constitutes a generalized shuffle, or dimension permutation. The dimension permutation is: $a_3 \leftarrow a_1 \leftarrow m_1 \leftarrow a_2 \leftarrow a_0 \leftarrow m_0 \leftarrow m_2 \leftarrow a_3$. In this example, the reshaping resulted in a single cycle on the dimensions. In general, the reshaping may result in several cycles, just as the m–step shuffle in general can be decomposed into several cycles.

A shuffle can be implemented as a sequence of successive pairwise dimension exchanges starting in any position. In a binary cube, such exchanges imply communication in two cube dimensions for each step if both dimensions in an exchange are nodal address dimensions. However, it is also possible to use a fixed memory dimension for each exchange. If the first exchange is repeated as a last exchange, then the result is a shuffle on all bits but the fixed exchange dimension. For a shuffle on n bits, the first alternative requires $n-1$ exchanges while the last requires $n+1$ exchanges. Thus, at the expense of two additional exchanges, each exchange only involves one nodal address dimension. In [34] we present algorithms that are nonoptimal by at most two exchanges, regardless of the number of cube dimensions in the shuffle and data elements per node. The idea for the algorithms is to use multiple exchange sequences (embeddings), exploiting the fact that a shuffle can be performed as a sequence of exchanges starting at any bit and proceeding in order of decreasing dimensions cyclicly.

6 Summary

We have discussed some of the data allocation techniques used to preserve locality of reference on the Connection Machine systems, namely data aggregation, nodal array shape, unstructured grid partitioning, and placement of partitions among the nodes. We have also demonstrated the use of several communication primitives, discussed optimal implementations of these with respect to utilizing the communications bandwidth of binary

n–cubes, and given a few results from their implementation on the Connection Machine systems CM–2 and CM–200. Scheduling the communications knowing the global communication needs has resulted in significant speedups for some communication patterns on the CM–2 and CM–200. For PSHIFT, a speedup of up to a factor of four has been observed, while for bit–reversal a speedup of close to two orders of magnitude has been observed in extreme cases. The choice of nodal array shape has been observed to affect the performance of matrix multiplication by more than one order of magnitude [47]. Randomization of the data allocation typically yields a performance improvement by a factor of 1.5 – 3 on the CM–2 and CM–200 for gather/scatter operations on irregular grids.

References

[1] B. Alspach, J.-C. Bermond, and D. Sotteau. Decomposition into cycles i: Hamilton decompositions. In G. Hahn et. al., editor, *Cycles and Graphs*, pages 9–18. Kluwer Academic Publishers, 1990.

[2] Christopher R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM J. Sci. Stat. Comp.*, 13(4):923–947, July 1992.

[3] D. P. Bertsekas, C. Ozveren, G.D. Stamoulis, P. Tseng, and J.N. Tsitsiklis. Optimal communication algorithms for hypercubes. *J. of Parallel and Distributed Computing*, 11:263–275, 1991.

[4] M. Bromley, Steve Heller, Tim McNerny, and Guy Steele. Fortran at ten Gigaflops: The Connection Machine convolution compiler. In *Proceedings of ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. ACM Press, June 1991.

[5] Jean-Philippe Brunet and S. Lennart Johnsson. All-to-all broadcast with applications on the Connection Machine. *International Journal of Supercomputer Applications*, 6(3):241–256, 1992.

[6] J. Carrier, L. Greengard, and V. Rokhlin. A fast adaptive multipole algorithm for particle simulations. *SIAM J. of Scientific and Statistical Computations*, 9(4):669–686, July 1988.

[7] M.Y. Chan. Embedding of grids into optimal hypercubes. *SIAM J. Computing*, 20(5):834–864, 1991.

[8] G. Dahlquist, Å. Björck, and N. Anderson. *Numerical Methods*. Series in Automatic Computation. Prentice Hall, Inc., Englewood Cliffs, NJ, 1974.

[9] William J. Dally. *A VLSI Architecture for Concurrent Data Structures*. PhD thesis, California Institute of Technology, 1986.

[10] William J. Dally. The J-Machine: A fine–grain concurrent computer. In *Proc. IFIP Congress*, pages 1147–1153. North-Holland, August 1989.

[11] Jack. J. Dongarra and Stanley C Eisenstat. Squeezing the most out of an algorithm in Cray Fortran. *ACM Trans. Math. Softw.*, 10(3):219–230, 1984.

[12] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23:298–305, 1973.

[13] M. Fiedler. Eigenvectors of acyclic matrices. *Czechoslovak Mathematical Journal*, 25:607–618, 1975.

[14] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25:619–633, 1975.

[15] Charles M. Flaig and Charles L Seitz. Inter–computer message routing system with each computer having separate routing automata for each dimension of the netwrok, 1988. U.S. Patent 5,105,424.

[16] High Performance Fortran Forum. High performance fortran language specification, version 0.4. Technical report, Department of Computer Science, Rice University, November 1992.

[17] Geoffrey C. Fox and Wojtek Furmanski. Optimal communication algorithms on the hypercube. Technical Report CCCP-314, California Institute of Technology, July 1986.

[18] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors.* Prentice-Hall, 1988.

[19] William George, Ralph G. Brickner, and S. Lennart Johnsson. Polyshift communications software for the Connection Machine systems CM–2 and CM–200. *Scientific Programming*, 3(1):83–99, Spring 1994.

[20] Gene Golub and Charles vanLoan. *Matrix Computations.* The Johns Hopkins University Press, 1985.

[21] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348, 1987.

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

[23] Ching-Tien Ho and S. Lennart Johnsson. Spanning balanced trees in Boolean cubes. *SIAM Journal on Sci. Stat. Comp*, 10(4):607–630, July 1989.

[24] Ching-Tien Ho and S. Lennart Johnsson. Embedding meshes in Boolean cubes by graph decomposition. *J. of Parallel and Distributed Computing*, 8(4):325–339, April 1990.

[25] Zdenek Johan. *Data Parallel Finite Element Techniques for Large–Scale Computational Fluid Dynamics.* PhD thesis, Department of Mechanical Engineering, Stanford University, 1992.

[26] Zdenek Johan and Thomas J. R. Hughes. An efficient implementation of the spectral partitioning algorithm on the connection machine systems. In *International Conference on Computer Science and Control*. INRIA, 1992.

[27] S. Lennart Johnsson. Dense matrix operations on a torus and a Boolean cube. In *The National Computer Conference*, July 1985.

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

[29] S. Lennart Johnsson. Performance modeling of distributed memory architectures. *J. Parallel and Distributed Computing*, 12(4):300–312, August 1991.

[30] S. Lennart Johnsson. Data ordering in multisection FFT. Technical report, Thinking Machines Corp., 1992. In preparation.

[31] S. Lennart Johnsson. *Compilation Techniques for Novel Architectures*, chapter *Language and Compiler Issues in Scalable High Performance Libraries*. Springer Verlag, 1993. Harvard University Technical Report TR-18-92.

[32] S. Lennart Johnsson. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Computing*, 19(11):1235–1257, 1993.

[33] 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.

[34] S. Lennart Johnsson and Ching-Tien Ho. Generalized shuffle permutations on Boolean cubes. *J. Parallel and Distributed Computing*, 16(1):1–14, 1992.

[35] S. Lennart Johnsson and Ching-Tien Ho. Optimal communication channel utilization for matrix transposition and related permutations on Boolean cubes. *Discrete Applied Mathematics*, 1992.

[36] S. Lennart Johnsson and Ching-Tien Ho. Boolean cube emulation of butterfly networks encoded by Gray code. *Journal of Parallel and Distributed Computing*, 20(3):261–279, 1994. Department of Computer Science, Yale University, Technical Report, YALEU/DCS/RR-764, February, 1990.

[37] 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*, volume 826, pages 223–231. Society of Photo-Optical Instrumentation Engineers, 1987.

[38] S. Lennart Johnsson, Michel Jacquemin, and Robert L. Krawitz. Communication efficient multi-processor FFT. *Journal of Computational Physics*, 102(2):381–397, October 1992.

[39] S. Lennart Johnsson and Robert L. Krawitz. Cooley-Tukey FFT on the Connection Machine. *Parallel Computing*, 18(11):1201–1221, 1992.

[40] Monica S. Lam, Edward E. Rothenberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *The Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74. ACM Press, 1991.

[41] Guangye Li and Thomas F. Coleman. A parallel triangular solver for a distributed memory multiprocessor. *SIAM J. Sci. Statist. Comput.*, 9(3):485–502, 1988.

[42] Guangye Li and Thomas F. Coleman. A new method for solving triangular systems on a distributed memory message-passing multiprocessor. *SIAM J. Sci. Statist. Comput.*, 10(2):382–396, 1989.

[43] Woody Lichtenstein and S. Lennart Johnsson. Block cyclic dense linear algebra. *SIAM Journal of Scientific Computing*, 14(6):1257–1286, 1993.

[44] Christoffer Lutz, Steve Rabin, Charles L. Seitz, and Donald Speck. Design of the mosaic element. In *Proceedings, Conf. on Advanced research in VLSI*, pages 1–10. Artech House, 1984.

[45] Kapil K. Mathur and S. Lennart Johnsson. All–to–all communication. Technical Report 243, Thinking Machines Corp., December 1992.

[46] Kapil K. Mathur and S. Lennart Johnsson. Communication primitives for unstructured finite element simulations on data parallel architectures. *Computing Systems in Engineering*, 3(1 – 4):63–72, December 1992.

[47] Kapil K. Mathur and S. Lennart Johnsson. Multiplication of matrices of arbitrary shape on a Data Parallel Computer. *Parallel Computing*, 20(7):919–951, July 1994.

[48] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, 1990.

[49] Abhiram Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual Symposium on the Foundations of Computer Science*, pages 185–194. IEEE Computer Society, October 1987.

[50] Abhiram Ranade and S. Lennart Johnsson. The communication efficiency of meshes, Boolean cubes, and cube connected cycles for wafer scale integration. In *1987 International Conf. on Parallel Processing*, pages 479–482. IEEE Computer Society, 1987.

[51] Abhiram G. Ranade, Sandeep N. Bhatt, and S. Lennart Johnsson. The Fluent abstract machine. In *Advanced Research in VLSI, Proceedings of the fifth MIT VLSI Conference*, pages 71–93. MIT Press, 1988.

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

[53] Arnold L. Rosenberg. Preserving proximity in arrays. *SIAM J. Computing*, 4:443–460, 1975.

[54] Horst D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[55] Quentin F. Stout and Bruce Wagar. Intensive hypercube communication I: prearranged communication in link-bound machines. Technical Report CRL-TR-9-87, Computing Research Lab., Univ. of Michigan, Ann Arbor, MI, 1987.

[56] Quentin F. Stout and Bruce Wagar. Passing messages in link-bound hypercubes. In Michael T. Heath, editor, *Hypercube Multiprocessors 1987*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.

[57] Paul N. Swarztrauber. Symmetric FFTs. *Mathematics of Computation*, 47(175):323–346, July 1986.

[58] Paul N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.

[59] Clive Temperton. On the FACR(l) algorithm for the discrete Poisson equation. *J. of Computational Physics*, 34:314–329, 1980.

[60] Thinking Machines Corp. *CM–200 Technical Summary*, 1991.

[61] Thinking Machines Corp. *CM–5 Technical Summary*, 1991.

[62] Thinking Machines Corp. *CM Fortran optimization notes: slicewise model, version 1.0*, 1991.

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

[64] Charles Tong and Paul N. Swarztrauber. Ordered Fast Fourier transforms on a masively parallel hypercube multiprocessor. *Journal of Parallel and Distributed Computing*, 12(1):50–59, May 1991.

[65] Leslie Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11:350–361, 1982.

[66] Leslie Valiant and G.J. Brebner. Universal schemes for parallel communication. In *Proc. of the 13th ACM Symposium on the Theory of Computation*, pages 263–277. ACM, 1981.