Network Related Performance Issues and Techniques for MPPs

Network Related Performance Issues and Techniques for MPPs

S. Lennart Johnsson

TR-29-95

December 1995

Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

To appear in *Optoelectronic Interconnect and Packaging, SPIE International Symposium on Lasers and Integrated Optoelectronics*, 1996.

Network Related Performance Issues and Techniques for MPPs

S. Lennart Johnsson
Department of Computer Science
University of Houston
and
Division of Applied Sciences
Harvard University

Abstract

In this paper we review network related performance issues for current Massively Parallel Processors (MPPs) in the context of some important basic operations in scientific and engineering computation. The communication system is one of the most performance critical architectural components of MPPs. In particular, understanding the demand posed by collective communication is critical in architectural design and system software implementation. We discuss collective communication and some implementation techniques therefore on electronic networks. Finally, we give an example of a novel general routing technique that exhibits good scalability, efficiency and simplicity in electronic networks.

1 Introduction

Massively Parallel Processors (MPPs) are a critical resource for large–scale computation in science and engineering. Accurate modeling require large data sets. For most applications, there is an inherent high degree of fine grain parallelism (data parallelism) in those data sets. Many applications also exhibit a course grain parallelism (control parallelism) arising from different functions being applied to the data set in succession, or to different parts of the data set. Evaluating technologies and different architectural options for efficient large scale computation requires an understanding of the computational requirements as determined by problem representation, algorithms used for the solution, and implementation techniques. This paper reviews the data motion requirements of some typical operations in large–scale computations and some of the techniques available for efficient management of data references. The techniques are developed for systems with electronic interconnect with an aim towards reducing communication needs and network contention.

From an applications/mathematical perspective, the functional software architecture for the solution of scientific and engineering problems on highly parallel architectures has so far remained the same as on uniprocessors. The changes that have taken place are in the software engineering domain. Many application codes use a set of matrix operations as low level primitives in constructing equation solvers, either direct or iterative. Solvers for partial differential equations are then constructed either directly from the matrix primitives, or the equation solvers, or by using various transforms or convolution. Optimization methods also require matrix

Operation	Local storage (words)					
	Processor		Chip		Board/Machine	
	Reg.	1k	32k	1M	32M	1G
Matrix Mult.	1	16	90	512	2896	16384
3D–Relaxation	2	19	32	77	307	1024
FFT	1	20	32	45	57	70

Table 1: Number of local operations per remote reference. 3D–Relaxation: 7–point stencil, vector–length 8 ($\alpha = 8$, $\beta = 96$).

primitives and possibly also equation solvers. An important collection of matrix primitives are the BLAS [7, 8, 43] (Basic Linear Algebra Subroutines) targeted for dense and banded matrix operations. For problems originating from discretized geometric domains as well as for network problems, the common matrix representation of the relationship between variables yield so–called sparse matrices. These matrices reflect the network topology, but are not uniquely determined by it. Whereas the BLAS for dense and banded matrices form a defacto standard, there is no consensus on BLAS for so–called sparse matrices. Nevertheless, sparse matrix operations are used very frequently in scientific and engineering problem solving.

The reduction in communication bandwidth requirement offered by exploiting locality of reference for three typical computations are quantified in Table 1 [24]. It shows the number of operations (and local references) per remote reference per processor for a few local memory sizes, assuming optimal locality of reference. For matrix multiplication block algorithms are optimal [20]. With a block size of 100×100 (local storage 32k words) per processor, the reduction in the need for memory or communications bandwidth is a factor of 100 compared to no locality of reference. The table entries for 3–D relaxation assume that the ratio of operations to remote references follow the rule $\frac{1}{\alpha}(\frac{M}{\beta})^{\frac{1}{\gamma}}$ for suitable values of α, β, and γ and M being the local memory size. Exploiting locality reduces the required communication bandwidth by a factor of 8–100 at the chip boundary for these computations, by a factor of 80–5000 at the board level, and by at least a factor of 125 at the I/O interface. For the FFT, the reduction in required interprocessor communication bandwidth is a factor of 14 for a data distribution allowing a radix-16k algorithm, compared to a data distribution with no locality of reference. The proof of optimality of an algorithm with a radix equal to the local memory size can be found in [20]. Gentlemen and Sande [16] made this observation soon after the discovery of the FFT, but did not provide a formal proof of optimality. Sorting behaves like the FFT with respect to the optimal required bandwidth as a function of the local memory size [20].

The 3–D regular grid relaxation example is typical for explicit solvers for partial differential equations based on finite differences. The stated numbers for the relative frequencies of remote and local references reflects the well–known notion of minimal surface–to–volume ratio. This ratio is easy to determine for regular grids. Though much more difficult to determine, it is expected that the minimal surface–to–volume ratio for localized nonuniform grids, such as tetrahedral grids for many fluid dynamics problems, exhibit a similar behavior. For nonlocalized grids, the average number of local references per remote reference may be $O(1)$. Partitioning

of nonuniform grids such that the processing nodes receive approximately equal sized subgrids while minimizing communication is an active area of research. We will discuss configuring the address space in the context of a few applications in the next few sections, then discuss a few important collective communication primitives followed by a brief discussion of general routing techniques for electronic networks.

2 Address space – data allocation

Traditional languages have a one–dimensional or linear storage model. Multidimensional data arrays are linearized by either using a row major or a column major ordering of data array axes. However, memory systems are not truly random access in that the access time to different parts of the memory is not the same. Techniques for memory contention in banked and interleaved memories have been studied extensively, with the work on the Burroughs Scientific Processor [5, 40, 41, 42] and the Prime Memory System [42] providing early and novel examples. These techniques all assume that the network bandwidth is sufficiently high to support the full bandwidth of the memory system. However, in massively parallel processors (MPP) with electronic interconnect, the bandwidth between a processor and its local memory is often (considerably) higher than the bandwidth to remote memory. In fact, to a significant degree the ideal data distribution objectives for MPPs is exactly opposite to those for interleaved or banked memory systems. For instance, in our introductory discussion of locality of reference we showed the benefit of block algorithms with respect to communication bandwidth requirements. A block data allocation implies a multidimensional address space as opposed to the one–dimensional address space with cyclic data allocation. Issues to be addressed are the dimensionality of the address space, its shape, how elements are aggregated into subsets for the processor memories, and how the subsets are distributed among the processor memories.

2.1 The shape of the multidimensional address space

For a d–dimensional data array with equally frequent local data references along all axes, and the same reference pattern for each array element, it is well known that the number of remote references is minimized when the lengths of the segments of all axes mapped to a memory unit are the same [14, 24]. Thus, for two–dimensional arrays, the local subarray should be a square for minimum communication needs, and for three–dimensional arrays the local data set should be a cube, and so on. Consider matrix multiplication: $C \leftarrow A \times B$. The index space for the computation is three–dimensional and for A of shape $P \times Q$ and B of shape $Q \times R$, the index space is of shape $P \times Q \times R$ as shown in Figure 1, even though the index space for each of the operands is two–dimensional. For the computations, the optimal shape of the address space may be one–dimensional, two–dimensional, or three–dimensional depending upon the relative sizes and shapes of the three operands [27]. The fact that a three–dimensional address space is desirable when there is many more processing nodes than matrix elements is well known. However, a three–dimensional address space may also be beneficial with respect to performance when there are many more matrix elements than processing nodes [28, 27].

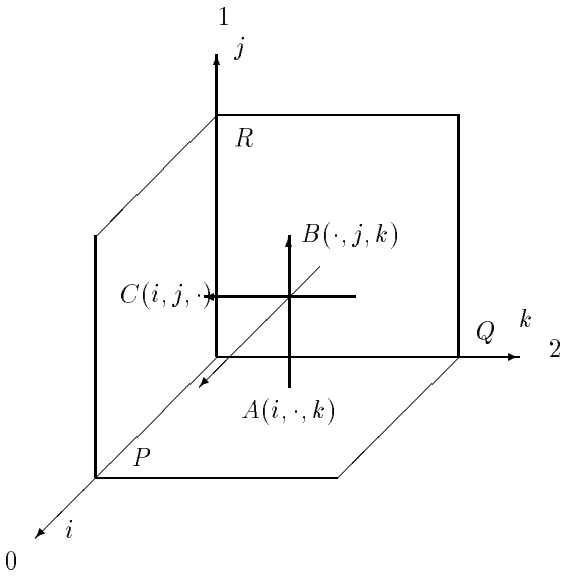

Figure 1: The index space for matrix multiplication.

We illustrate the significance with respect to performance of the shape of a two–dimensional address space in Figures 2 and 3. With the strategy of keeping the matrix with the largest number of elements stationary [27, 50] and moving the other operands as required (in order to minimize data motion), the ideal shape of the address space is such that the stationary matrix has square submatrices in each node [27, 50]. Figure 2 confirms that the optimal nodal array shape is square for square matrices. For the matrix shapes used in this experiment, a one–dimensional nodal array aligned with either the row or column axis, requires about a factor of six higher execution time than the ideal two–dimensional nodal array shape.

Figure 3 is more interesting in that all three operands have different shapes. The aspect ratio for the product matrix is four, while the aspect ratios for the multiplier and the multiplicand were varied by a ratio of up to 16. With a 256 node system, and an algorithm that keep the product matrix stationary, the results show that making the submatrices of the product matrix square tends to yield the best performance. Deviations from the predicted optimal shape is due to some restrictions in permissible data allocations on the Connection Machine system CM–2/200. For the rectangular matrices in our example, the best to worst performance ratio as a function of the nodal array shape was nearly 20.

Computations such as LU and QR factorization and the solution of triangular systems of equations, require global communication operations for selecting pivots, distributing pivot rows and columns, and related operations. No nearest neighbor operations in a Cartesian space is required. The ideal shape of the address space for the factorization is such that the subarrays are close to a square [47] (the pivot selection makes the communication along the two axes unbalanced).

For the Alternating Direction Implicit (ADI) methods, which are based on the solution of sets of tridiagonal systems of equations in alternating directions along

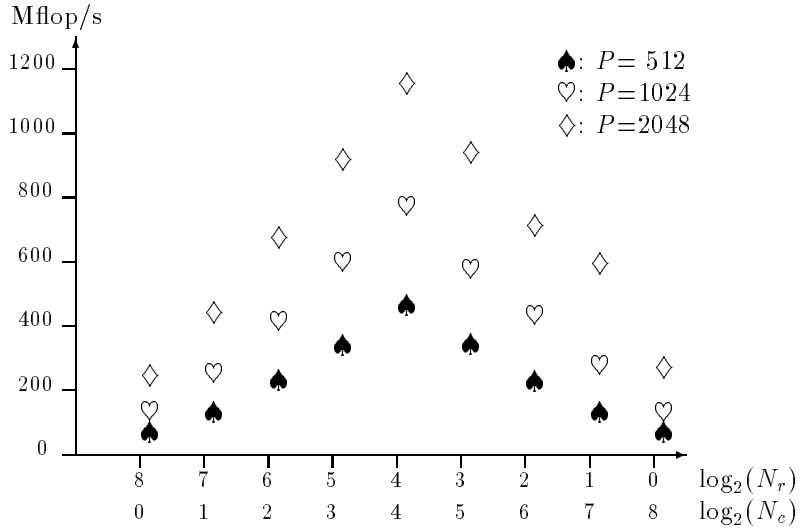

Figure 2: Influence of shared processor configuration on the performance for multiplication of square matrices of size P, 64–bit precision. The shape of the 256 processor Connection Machine system CM–200 is $N_r \times N_c = 256$.

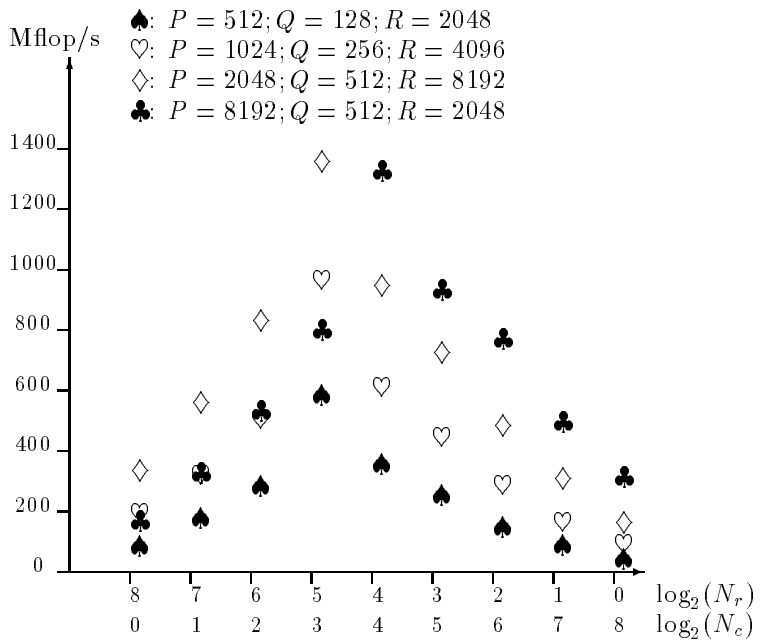

Figure 3: Influence of shared processor configuration on multiplication of rectangular matrices of shape $P \times Q \times R$, 64–bit precision. The shape of the 256 processor Connection Machine system CM–200 is $N_r \times N_c = 256$.

array axes, the ideal shape of the address space is such that the shape of the local subarrays have axes' of equal length [30, 36].

The implementation of matrix multiplication used in the above examples reflects the characteristics of a *node limited* architecture. In such an architecture, the nodal communications bandwidth is independent of the message destinations and the message routing. The Connection Machine CM–5 is an example of such an architecture. In *link limited* architectures [64, 65], the rate at which a node can send and receive data is dependent upon the destination of the messages, in that the number of channels per node that can be used effectively is a function of the destinations. The Connection Machine CM–2/200 is an example of the latter type of architecture. The matrix multiplication algorithm can be restructured such that it in fact is link limited [27, 28].

As an illustration of the performance characteristics of a link limited architecture with a suitable algorithm we use an implementation of the Cooley–Tukey FFT algorithm with pipelining of the successive butterfly stages on a binary cube network. From a computational point of view, the data set of size $M = 2^m$ for a one–dimensional FFT is best viewed as an m–dimensional data set with each axis of length two. For a link limited architecture using a binary cube network interconnect of $N = 2^n$ nodes, each node contains 2^{m-n} data elements, $n \leq m$. There are $m - n$ local dimensions and n nonlocal dimensions. The data set is bisected for each of the nonlocal dimensions, and the surface area exposed by the bisections is 2^{m-n}. The same amount of data must be communicated along all dimensions. For a multidimensional data array making the axes subject to transformation entirely local clearly minimizes the communication. When that is not possible, minimizing the surface area exposed to one link is sufficient for a minimal number of element transfers in sequence in a link limited model, whereas minimizing the surface area for all links is necessary for a node limited model. In the link limited model, it is immaterial with respect to the number of elements being transferred how the n processor dimensions are distributed among the array dimensions. Figures 4 and 5 illustrate these facts.

2.2 Aggregation

In the above discussion of the shape of the address space it was implicitly understood that data *aggregation* was of the *consecutive* [23] (block) type in which a set of successive elements along an axis are mapped to the same node. Another apparent form of aggregation is *cyclic* [23] (wrap) allocation in which element i along an axis is mapped to node $i \bmod P$, where P is the number of nodes along the axis. The two forms of data aggregation are illustrated in Figure 6. Both block and cyclic data allocation are supported in High Performance Fortran [13] through compiler directives.

The data layout may have an impact on both load–balance and total communication needs. For the FFT it can be shown that cyclic data allocation requires half as many element moves in sequence for unordered FFT as a block allocation would require [33] in a link limited model. For computations with a nonuniform use of the address space, such as factorization of dense matrices and triangular system solvers,

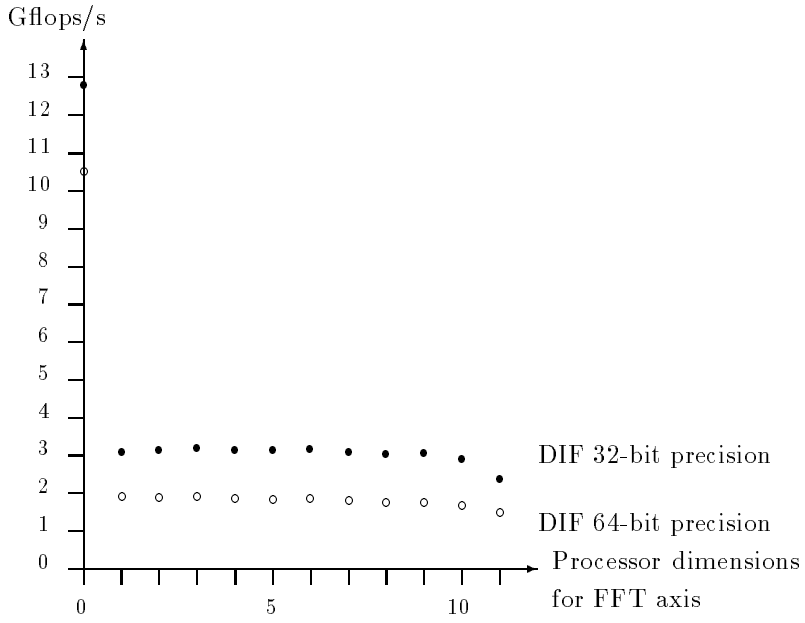

Figure 4: Total execution rate for one–dimensional, unordered, FFT on a 4096×4096 array as a function of the configuration of a 2048 processor CM–200.

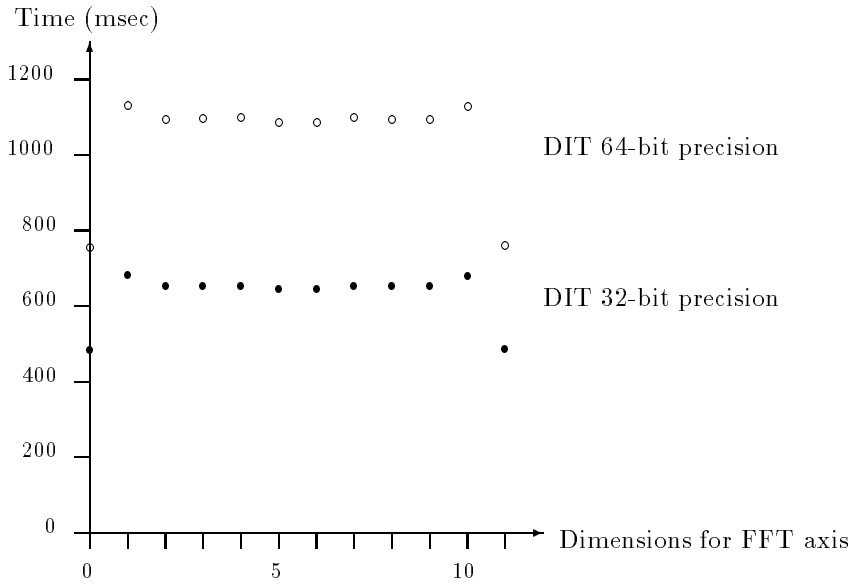

Figure 5: Total execution time for a two–dimensional unordered CCFFT on a 4096× 4096 array as a function of the configuration of 2048 CM–200 processors.

Consecutive data allocation

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

Cyclic data allocation

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31

Figure 6: Consecutive and cyclic data allocation of 32 elements to 8 processors.

load–balance is affected by the data aggregation. A cyclic data allocation is often promoted as the only way of achieving good load–balance, whereas in fact good load–balance can be achieved for either form of data allocation by simply matching the order of the traversal of the address space with the data allocation scheme [47]. This point is very important. It illustrates that scheduling and data allocation must be considered together in that data reallocation may be avoided by adjusting the scheduling of operations.

2.3 Sparse matrices – regular grids

The purpose of sparse matrix techniques is to take advantage of the zero/nonzero structure of a matrix to reduce both storage and arithmetic needs. For highly regular sparse matrices, storage schemes and address calculations can be much simplified compared to arbitrary sparse matrices. Scheduling of operations for efficient use of memory hierarchies is also much simplified for highly regular sparse matrices. Therefore, the data representation and the algorithms for sparse matrices with a "stripe structure", i.e., a structure with nonzeroes appearing on diagonals that are not necessarily adjacent, are typically different from those for arbitrary sparse matrices. On MPPs, it is advantageous to maintain this difference, but suitable representations differ from those for sequential machines. We refer to sparse matrices originating from difference approximations on regular domain discretizations as *grid sparse* matrices. Matrices capturing relationships on unstructured grids are referred to as *arbitrary sparse* matrices. In either case, the matrix entries may be single elements, vectors, or small matrices.

It is common to represent grid sparse matrices as a collection of one–dimensional arrays, or as a dense matrix with the number of columns (or rows) equal to the number of nonzero diagonals [62]. Each array, or each column in the two–dimensional representation, represents a linear ordering of all grid points. If the sparse matrix

entries are vectors or matrices, then the dimensionality of the arrays is increased by one or two dimensions. Representing grid sparse matrices in this traditional manner is not suitable for distributed memory architectures. Relaxation methods for linear system solution and explicit methods for partial differential equations typically access data in some local neighborhood of the grid point to be updated. The task of preserving locality of reference is much simplified if the adjacency in the grid is preserved in the data representation. By representing grid sparse matrices as dense arrays of the same shape as the grid they represent, the techniques for preserving proximity in mapping dense arrays to processing nodes can be employed also for grid sparse matrices. For instance, the common three–point stencil in one dimension yields a tridiagonal matrix, a five–point stencil in two dimensions yields a matrix with five nonzero diagonals with a row or column ordering of the grid points. The main diagonal and its two adjacent diagonals are nonzero. The remaining two nonzero diagonals are separated from the main diagonal by M columns or rows for a grid where the axes of stride one has extent M. Similarly, a seven–point stencil in three dimensions yields a matrix with seven nonzero diagonals.

2.4 Sparse matrices – irregular grids

For computations involving arbitrary sparse matrices and irregular grids, taking advantage of locality of reference is a much more difficult problem than for grid sparse matrices. The basic idea is still the same: assign variables and elements of the sparse matrix, or the grid, to processing nodes based on a partitioning of the underlying graph. However, the representations of the matrix and the nodal variables are quite different from the grid sparse case, and so are the partitioning techniques. As was the case for grid sparse problems, partitioning based on subdomains preserve locality of reference when the computations use data with indices in some local neighborhood of the data being updated. This property holds for most iterative methods for the solution of sparse systems of equations, and for explicit methods for the solution of partial differential equations.

Two general partitioning techniques of significant recent interest are the recursive spectral bisection (RSB) technique proposed by Pothen et. al. [59] and the geometric approach proposed by Miller et. al. [53, 54, 69]. The RSB technique has been used successfully by Simon [63] for partitioning of finite volume and finite element meshes. A parallel implementation of this technique has been made by Johan [21]. The spectral partitioning technique is based on the eigenvector corresponding to the smallest, absolute, nonzero eigenvalue of the Laplacian matrix associated with the graph to be partitioned. The Laplacian matrix is constructed such that the smallest eigenvalue is zero and its corresponding eigenvector consists of all ones. The eigenvector associated with the smallest nonzero eigenvalue is called the *Fiedler vector* [10, 11, 12]. Grid partitioning for finite volume and finite element methods is often based on a dual mesh representing finite volumes or elements and their adjacencies (or some approximation thereof) rather than the graph of nodal points. The reason for using a volume or element based graph is that the computations are naturally organized as volume or elementwise computations. These computations exhibit locality of reference within the volumes or elements and can often be performed as a (large) collection of dense matrix operations. Communication is required when

passing data between the global representation, and the representation of the collection of local elements [35, 52]. The purpose of the partitioning is to minimize this communication.

For finite element computations, the dual graph subject to partitioning typically is formed by only modeling adjacencies between elements that share faces. This adjacency accurately represents the communication requirements for face centered schemes, such as finite volume methods. However, in finite element methods, communication is also required between elements sharing edges and corners. Based on the partitioning of the dual graph a partitioning of the set of nodal values is carried out. Nodal points internal to a partition are mapped to the processing node to which the partition is assigned. Boundary nodes must be assigned to one of the partitions among which they are shared, or replicated among the partitions among which they are shared. Only boundary nodes require communication.

One advantage of the spectral bisection technique is that it is based on the topology of the graph underlying the sparse matrix. It requires no geometric information. However, it is computationally quite demanding. The geometric partitioning technique by Miller et. al. holds promise to be computationally less demanding than the spectral decomposition technique, but relies on geometric information and geometric properties of the graph [55, 56]. Geometric information is typically available for meshes generated for the solution of partial differential equations, but may not be present in other applications.

The RSB technique has been used to partition the following five tetrahedral meshes on five CM–5 systems of different sizes [22]:

– A mesh with 109,914 elements around a Falcon Jet airplane, Figure 7, partitioned on a 32–node CM–5.
– A mesh with 266,556 elements around an ONERA M6 wing, Figure 8, partitioned on a 64–node CM–5.
– A mesh with 575,986 elements around a generic commercial airplane, Figure 10, partitioned on a 128–node CM–5.
– A mesh with 1,010,174 elements around an F–18 fighter jet, Figure 11, partitioned on a 256–node CM–5.
– A mesh with 2,132,448 elements around an ONERA M6 wing partitioned on a 512–node CM–5. This is a refined version of the 266,556 element mesh above.

The number of nodes, number of elements and number of graph edges for each problem are summarized in Table 2. The number of elements per processor is not constant over those problems, but its variations are relatively small, thus allowing a useful evaluation of the RSB technique. Table 3 presents the partitioning timings on the CM–5, the number of Lanczos iterations required for each problem, and the number of edge cuts generated by the RSB algorithm. Figure 9 shows a partitioning of the M6 Wing. Figure 12 shows the scalability of the parallel RSB implementation by giving the partitioning time as a function of the number of partitions for the five meshes we consider. The increase in the partitioning time is modest despite the significant increase in problem size, number of partitions and Lanczos iterations.

Example	No. of nodes	No. of elements	No. of graph edges
Falcon Jet	19,417	109,914	217,669
M6 wing	48,011	266,556	527,966
Airliner	106,064	575,986	1,136,029
F-18	182,055	1,010,174	1,999,646
M6 wing(fine)	367,723	2,132,448	4,244,312

Table 2: Mesh characteristics of five partitioning examples.

Example	Number of processors	Number of partitions	No. of Lanczos iterations	Elapsed time (sec)	Number of edge cuts
Falcon Jet	128	128	1,156	44	22,926
M6 wing	256	256	1,413	76	57,063
Airliner	512	512	1,606	124	112,910
F–18	1,024	1,024	2,061	178	220,413
M6 wing (fine)	2,048	2,047	2,419	201	481,359

Table 3: Partitioning statistics for five examples. Lanczos tolerance is 10^{-3}.

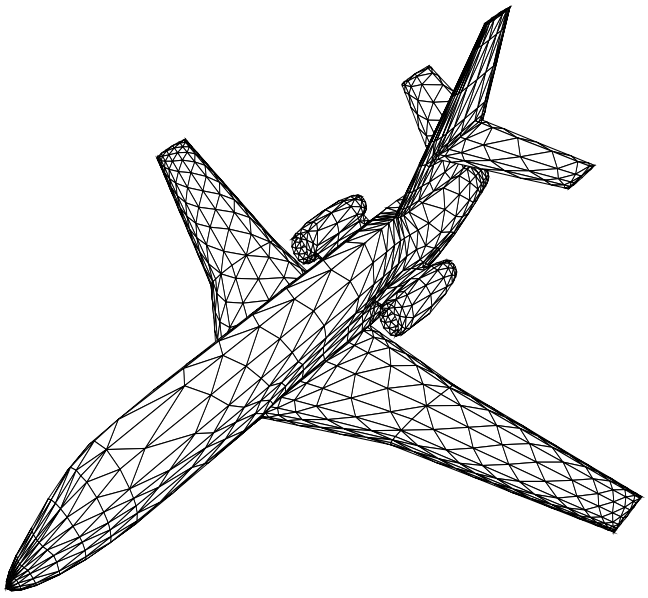

Figure 7: Falcon Jet. View of surface mesh on the airplane.

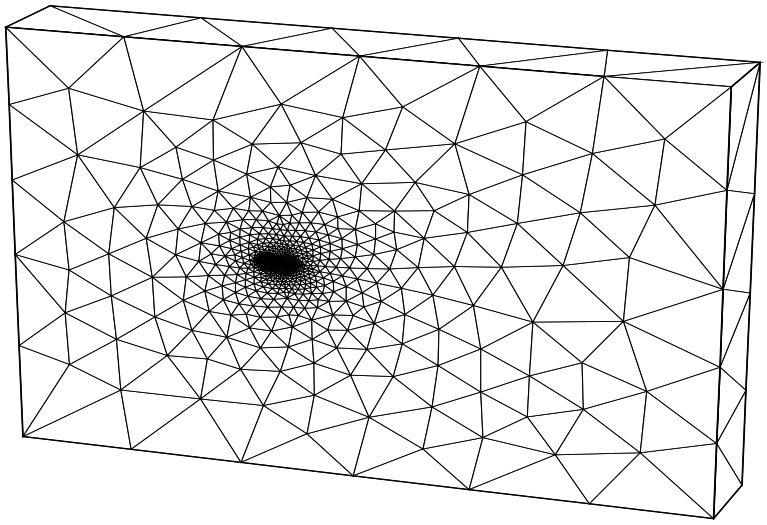

Figure 8: M6 wing. View of surface mesh on outer boundaries.

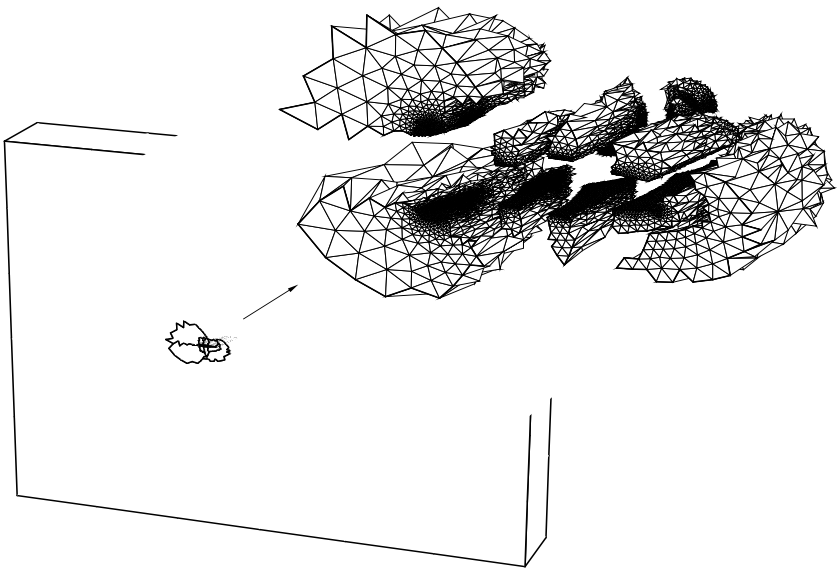

Figure 9: M6 wing. Decomposition into 16 subdomains.

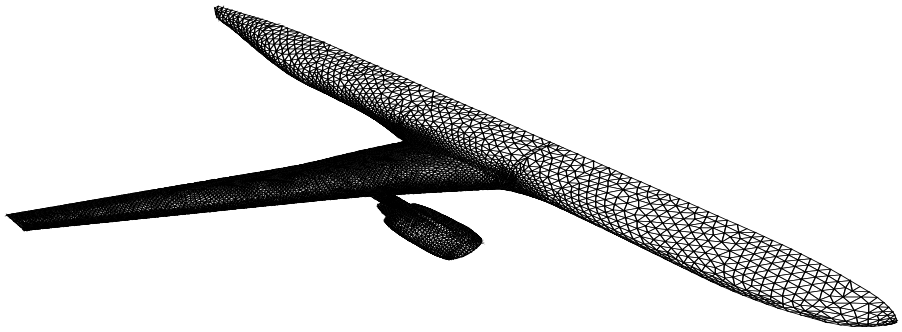

Figure 10: Commercial aircraft. View of surface mesh on the half–airplane.

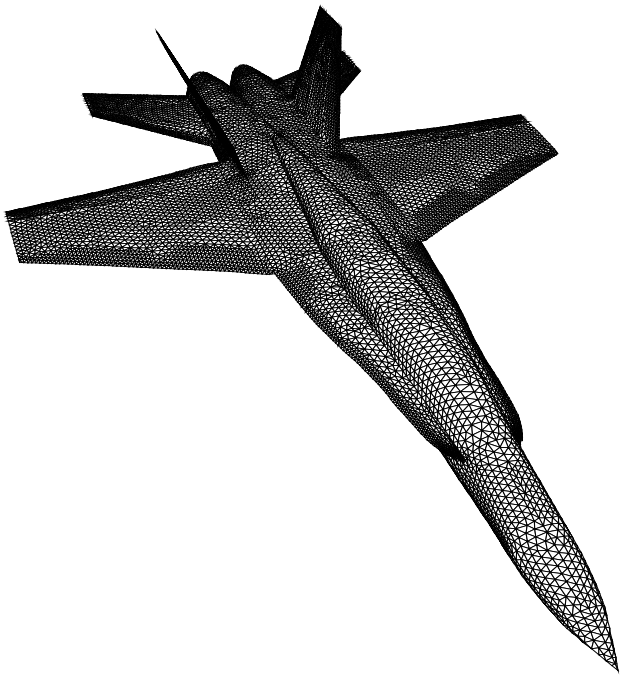

Figure 11: F–18 fighter jet. View of surface mesh.

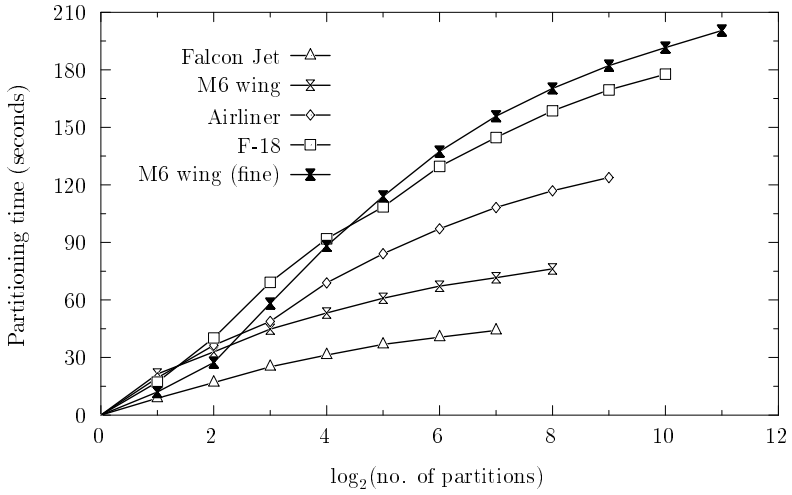

Figure 12: Partitioning time as a function of the number of partitions for five examples.

2.5 Allocation of aggregates

The consecutive and cyclic distribution schemes and the dimensionality of the address space for regular and grid sparse arrays, and spectral or geometric partitioning for arbitrary sparse matrices and irregular grids, determines which elements are grouped together for a node. The data reference pattern determines the communication needs of each node.

The total demand on the communication system is not only affected by how the data is grouped together, but also how the groups are assigned to nodes. Ideally, the groups of data are allocated to the nodes such that the contention is minimized. To accomplish this task, both the *data reference pattern* and the *network topology* must be taken into account, as well as the *routing* scheme. Optimal allocation of data is a hard problem. Moreover, the allocation may need to be dynamic to efficiently accommodate different phases of a computation. We will briefly discuss the mapping of groups of data below. Routing issues will be discussed in a later section.

For irregular grids, mapping of groups of data to processing nodes such that communication time is minimized under various load conditions is a much more difficult problem than the mapping of regular arrays. Instead of attempting to find the best possible map, it may be more profitable to search for a map that is guaranteed to have an acceptable worst case behavior. A randomized data placement [60, 61] reduces the risk for bottlenecks in the routing system. The randomized placement of data achieves the same communication load characteristics in a single (deterministic) routing phase as randomized routing achieves in two phases [74, 75].

Figures 13 and 14 give examples of the performance improvements achieved on the CM–2 through the use of randomized data allocation in a finite element compu-

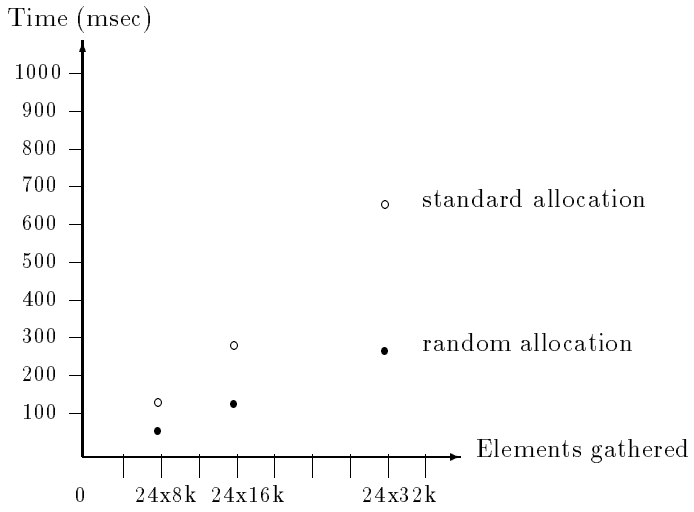

Figure 13: Gather with binary and randomized addresses. 8K CM-2.

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 4: The effect of randomization on gather and scatter performance. Times in msec on an 8K CM–200.

tation on an unstructured grid. The horizontal axis shows the number of degrees of freedom and elements, while the vertical axis denotes the execution time. Each element has 24 degrees of freedom. The performance improvement for the gather instruction due to randomization is in the range 2.1 - 2.4. The improvement is increasing with the problem size. Figure 14 shows the execution times for two methods of accumulating the product vector: using the combining features of the router, and accumulation after the routing operation. Randomization of the addresses improved the router combining time by about a factor of two, but performing the routing without combining is even more effective. Table 4 gives the gather scatter times with and without randomization for a solid mechanics application [52] on the CM–200. The performance enhancement is a factor of 1.5 – 2.2, which in our experience is typical. It is rarely the case that randomization has caused a performance degradation.

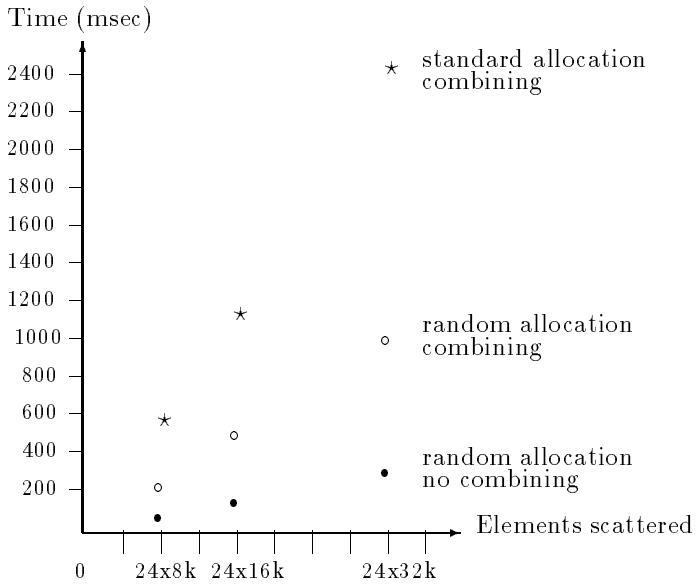

Figure 14: Accumulation of vector elements. Binary and randomized addresses. 8K CM-2.

3 Collective communications functions

The previous section demonstrated the value of data allocation with respect to bandwidth requirements and some techniques for accomplishing locality of reference in some important computations. In architectural design it is also important to know what data reference patterns must be supported. Of particular interest are so–called collective communications for which several communication actions are known and the path selection and scheduling can be chosen such that the communication time is minimized. Important collective communication functions are

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

These functions are applied to complete data arrays, or segments thereof. Below, we give a few examples of the use of some of the communication primitives above, and

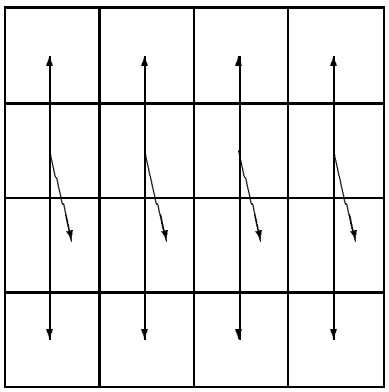

Figure 15: Broadcasting of a pivot row in LU–decomposition.

some techniques used to achieve high utilization of the communications bandwidth in some networks.

3.1 Broadcast

A novel approach to the efficient use of communications bandwidth for broadcast in networks with several paths between the source and the destinations is the use of multiple spanning trees [29]. Broadcast and reduction from a single source to subsets of nodes, holding an entire row or column, are 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 15. 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 along the different instances of an axis do not interfere with each other, and the concurrent broadcasts degenerate to a number of broadcasts within disjoint subsets of nodes. However, in other networks a data mapping guaranteeing no interference between communication operations on disjoint subsets of data 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. For instance, on the Connection Machine system CM–5 which uses a fat–tree interconnection network [45, 46], a two–dimensional array is either mapped to the nodes in row or column major order. A row major ordering implies that broadcast within rows can take place without interference between rows, since they are mapped to separate subtrees. However, for broadcast within columns contention occurs since the CM–5 fat–tree exhibits a reduction in bi–section width for the first two levels of the fat–tree.

In implementing a broadcast algorithm it is important to exploit the bandwidth. A simple spanning tree may not accomplish this task. Consider a binary cube network (used in the CM–2 and CM–200). On an n–cube, using a binomial tree to broadcast

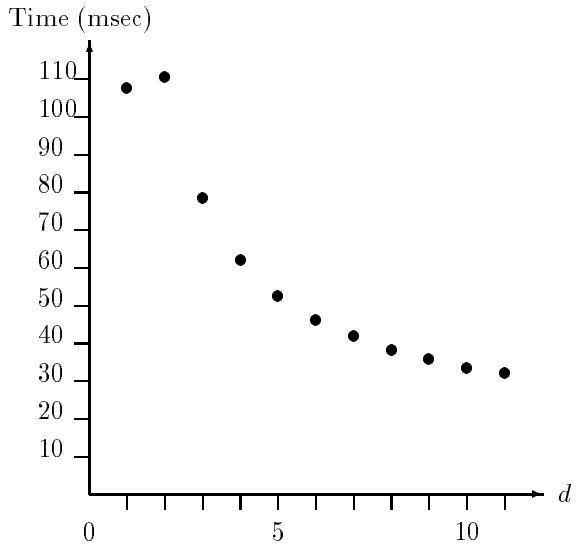

Figure 16: Time in msec for broadcast of 16K 32–bit data elements on Connection Machine system CM–200 as a function of the number of cube dimensions.

M elements from a node to all other nodes requires a time of nM with the communication restricted to one channel at a time. 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 [29]. 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 [29]. The basic idea is that 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 16 [25]. As expected, the time to broadcast a given size data set decreases with the number of nodes to which the set is broadcast. Thus, broadcast exhibits a logarithmic speedup with respect to the number of nodes.

3.2 All–to–all broadcast/reduce

Another important communication primitive is the simultaneous broadcast from each node in a set to every other node in the same set, *all–to–all broadcast* [29, 4, 51]. This communication is typical for so called direct N–body algorithms, but it is also required in many dense matrix algorithms. In all–to–all reduction, reduction operations are performed concurrently on different data sets, each distributed over

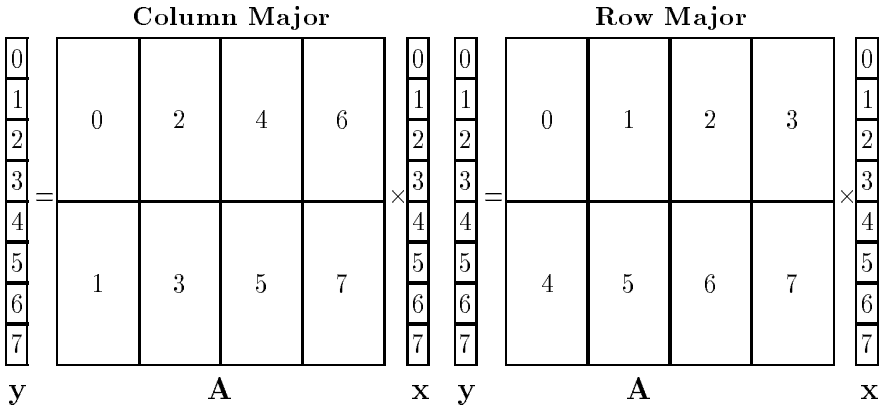

Figure 17: Data allocation on a rectangular nodal array.

all nodes such that the results of the different reductions are evenly distributed over all nodes. Here we will illustrate their use in matrix–vector multiplication.

With the processing nodes configured as a two–dimensional nodal array for the matrix, and 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 17 illustrates the data allocation for both row major and column major ordering of the matrix. For a matrix of shape $P \times Q$, allocated to a two–dimensional nodal array in column major order, an all–to–all broadcast [14, 29, 64, 65] 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 [51].

As in the case of broadcast from a single source, all–to–all broadcast on an n–

cube can be performed in a time proportional to the lower bound. With each node initially holding M elements, a time of $M(N-1)$ is required for communication restricted to a single port at a time, and a time of $\frac{M(N-1)}{n}$ is required for all–port communication [29]. A simple, yet optimal, all–port algorithm for all–to–all broadcast uses n Hamiltonian paths for each node [29, 4, 51]. For all–to–all broadcast, the Hamiltonian paths need not be edge–disjoint [4, 29].

3.3 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. Many of these gather/scatter operations are related to stencil computations. The required data motion is preferably supported through PSHIFT (for polyshift) [17], which allows the programmer to specify concurrent shift operations in one or both directions of one axis or multiple axes. PSHIFT is a generic communication primitive that can be optimized with respect to communication and local data motion. An example of the call for a seven–point stencil in three dimensions is shown below. In effect, PSHIFT provides an effective means of emulating lattices on binary cubes. (A further level of optimization for stencil evaluation is provided by the so–called stencil compiler [3], 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.)

```
DIMENSION A(512,512,512), B(512,512,512), C(512,512,512), D(512,512,512),
          E(512,512,512), F(512,512,512), G(512,512,512), X(512,512,512),
          Y(512,512,512)
    :
CALL GRID_SPARSE_MATRIX_VECTOR_MULT(ier, setup, y_axes, coeff_axes, x_axes,
          Y, X, A, B, C, D, E, F, G)
```

For gather and scatter operations on unstructured grid computations, it is generally necessary to resort to the use of a general purpose router. A discussion of some general routing techniques is given in the next section. Here we only mention that one approach toward minimizing contention for gather/scatter operations is a randomized data allocation that guarantees that the risk for severe contention is low. Randomized data allocation is supported on the Connection Machine Cm–2/200 systems. Table 5 [49] summarizes the effect of randomization of the data allocation on the performance of gather/scatter operations on a few meshes. A performance enhancement of $1.5 - 2.3$ is typical.

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 5: The effect of randomization on gather and scatter performance. Times in msec on an 8K CM–200.

3.4 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 [51]. 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 used on the Connection Machine systems CM–2/200 [34]. 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)	$(a_5a_4a_3a_2a_1\|x_0)$
(04321|5)	$(x_0a_4a_3a_2a_1\|a_5)$
(05321|4)	$(a_4x_0a_3a_2a_1\|a_5)$
(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)	$(a_4a_3a_2a_1a_5\|x_0)$

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 it is desirable to perform multisections including all local memory bits. Thus, for instance, with two local memory bits, four–sectioning should be carried out, as shown in Example 2.

Example 2.

Address	Index
(65432|10)	$(a_6 a_5 a_4 a_3 a_2 \mid \underbrace{x_1}_{j\,1}\ \underbrace{x_0}_{j\,0})$
(10432|65)	$(\underbrace{x_1 x_0}_{m}\ a_4 a_3 a_2 \mid \underbrace{a_6 a_5}_{m})$
(10652|43)	$(\underbrace{\underbrace{a_4 a_3}_{m}\ \underbrace{x_1 x_0}_{m}}_{k\cdot m}\ \underbrace{a_2}_{n-k\cdot m} \mid \underbrace{a_6 a_5}_{m})$
(10654|23)	$(\underbrace{a_4 a_3 a_2}_{n-m}\ \underbrace{x_0}_{(k+1)m-n}\ \underbrace{x_1}_{n-k\cdot m} \mid \underbrace{a_6 a_5}_{m})$
(23654|10)	$(\underbrace{a_4 a_3 a_2}_{n-m}\ \underbrace{a_6}_{(k+1)m-n}\ \underbrace{a_5}_{n-k\cdot m} \mid \underbrace{x_0 x_1}_{m})$
(23654|10)	$(\underbrace{a_4 a_3 a_2}_{n-m}\ \underbrace{a_6}_{(k+1)m-n}\ \underbrace{a_5}_{n-k\cdot m} \mid \underbrace{x_1 x_0}_{m})$

Example 2 was 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.)

In multidimensional FFT, all of local memory should be considered in performing the multisectioning [26].

The FFT produces the results in bit–reversed order with respect to the indices. Thus, establishing a normal index map in the output domain requires an unshuffle with bit–reversal. Figure 18 [26] shows an example.

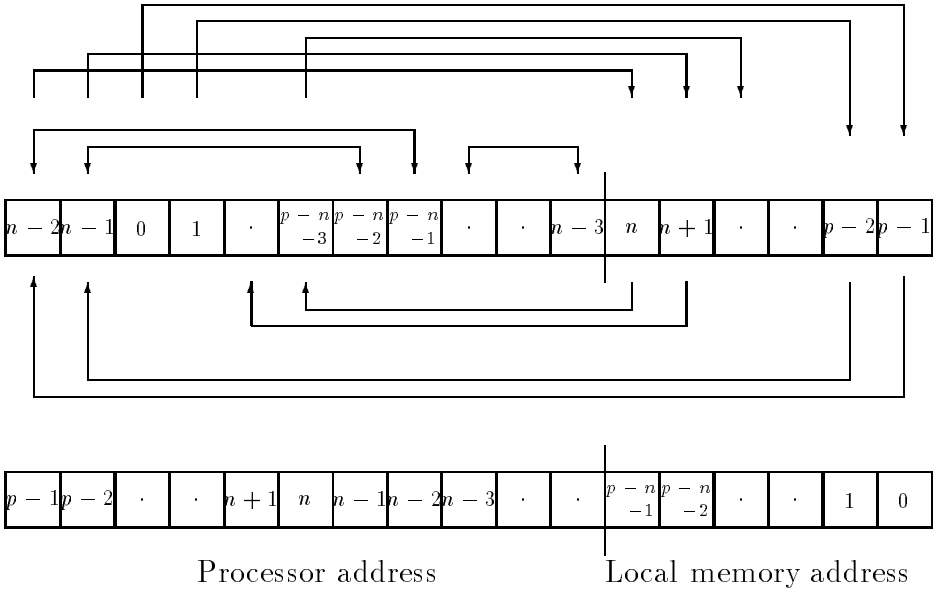

Figure 18: Two step reordering after 4–section based radix–2 FFT. First step, bit–exchange between nodes; second step, bit–exchange between local memory and node addresses. $p \leq 2n - m$.

The lower bounds for all–to–all personalized communication depends upon the network and communication system. For a binary n–cube with M data elements per node, the lower bound is $\frac{nMN}{2}$ for communication restricted to a single port per node and $\frac{MN}{2}$ for all–port communication [29]. Balanced spanning trees [19] provide for optimal all–to–all personalized communication with 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 [29].

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 spend in transition from source to destination in order to minimize pipeline delays. Algorithms with a minimal transition time are presented in [32].

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 19 [25]. 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.

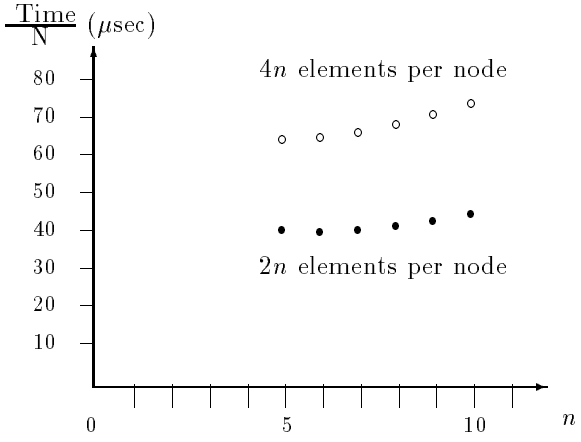

Figure 19: All–to–all personalized communication on the Connection Machine.

3.5 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 [67, 68]. 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 20. The Figure also shows that the postprocessing matching indices i and $N - i$ correspond 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 is the same as if there was only one element per node. In a cyclic data allocation, the communication for the first complex local memory location across all nodes is the same as if there was a single element per node. 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 [27].

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

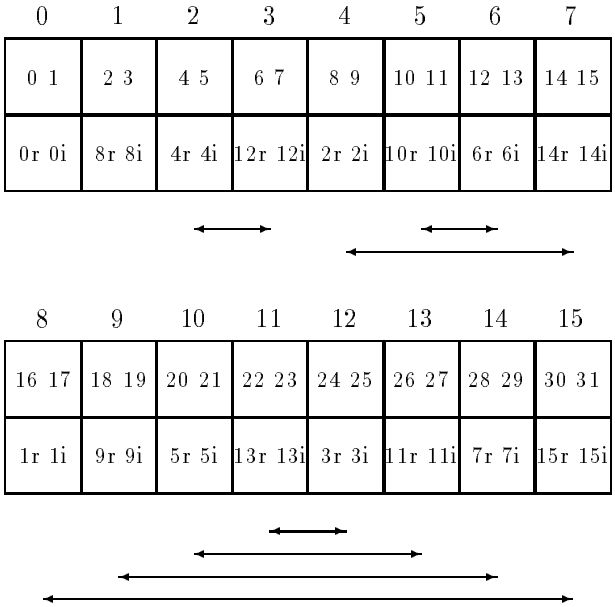

Figure 20: Postprocessing for real–to–complex FFT. Bit–inversion in subcubes.

used.

3.6 Shuffle operations on binary cube networks

Computing the FFT through multisectioning results in an m–step shuffle on the index space, where m is the number of bits encoding the first digit exchange. 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

$$\begin{pmatrix} a_3\, a_2\, m_2\, a_1\, a_0\, m_1\, m_0 \\ y_2\, y_1\, y_0\, x_3\, x_2\, x_1\, x_0 \end{pmatrix}$$

to the allocation

$$\begin{pmatrix} m_2\, m_1\, m_0\, a_3\, a_2\, a_1\, a_0 \\ y_2\, y_1\, y_0\, x_3\, x_2\, x_1\, x_0 \end{pmatrix},$$

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 communi-

cation in two cube dimensions for each step, when 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 [31] we present algorithms that are nonoptimal by two exchanges at most, regardless of the number of cube dimensions in the shuffle and the data elements per node. The algorithms 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.

4 Routing

Minimizing the potential for high congestion of communication links is an important goal in the design of routing algorithms and networks. Minimizing network load by using minimal algorithms is a sensible router design objective, though minimality in routing does not guarantee a minimal routing time, particularly under heavy load. In fact, non–minimal routing algorithms and data allocations which do not preserve locality of reference may utilize more of the available network bandwidth and decrease the congestion of communication links, without the corresponding increase in demand for network bandwidth.

There are three common approaches to the message routing problem:

1. *Oblivious* – The path taken by a message depends only on the source and destination of the message. The path is *not* influenced by other messages.
2. *Adaptive* – The path taken by a message is influenced by other messages, usually in an attempt to minimize the total routing time.
3. *Customized* – Message routing is based on global knowledge of the problem.

Routing algorithms are also *randomized* or *deterministic*. In a randomized algorithm the path selection, the scheduling of messages, or both, use randomization. In a deterministic algorithm, the paths and schedules are determined without randomization. The Intel Paragon, CalTech Mosaic, MIT J–Machine, Cray T3D, and CMU iWarp all use deterministic, oblivious routing. Such routing is relatively easy to implement and usually performs well, though performance may decay rapidly under heavy load. For *any* N–node, degree–d network, there exist permutations for which any deterministic, oblivious routing strategy requires time $\Omega(\sqrt{N}/d)$ [37]. Matrix–transpose and bit–reversal exhibit worst–case behavior on the binary cube for e–cube routing.

Two–phase randomized routing in which a random node is used as an intermediate destination [72, 73] can route *any* permutation on binary n–cubes and rank–n butterfly and related networks in about $2n$ steps [71] with high probability. The

disadvantages of the algorithm are that all data locality is lost and buffers of size $O(n)$ are required in each node. However, variants with fixed size buffers have been proposed [48, 71]. Randomization is employed in the first routing phase in the fat–tree network [44] of the CM–5 [70].

An improvement of the routing time by approximately a factor of two can be achieved by randomizing the allocation of the address space, as suggested by Ranade et al. [60, 61]. This approach is being implemented by Abolhassan et al. [1]. Fixed size buffers suffice, but data locality is still lost. Some of the experience of randomized data allocation on Connection Machine systems was discussed in Section 2.5.

Adaptive routers which do not employ randomization have been proposed by Dally and Aoki [6], Duato [9], Konstantinidou [38], Ngai and Seitz [57], and Pifarré et al. [58], among others. A survey of adaptive routing for binary cubes can be found in [15]. Deterministic, adaptive routing is used on the CM–2/200 [18]. The Chaos router [2, 39] is a randomized, adaptive router, which allows messages to follow random shortest paths from source to destination, but will misroute messages when congestion occurs.

ROMM (Randomized, Oblivious, Minimal, Multi–Phase) routing is a routing technique that combines minimality with randomization for contention minimization. ROMM algorithms route each message from source to destination in k phases. During each phase, a subset of the address dimensions that need to be routed are traversed whenever $k \leq n$, where n is the number of dimensions in the address space. The dimensions traversed in each phase are determined by randomly selecting, for each message, $k-1$ intermediate nodes $R_1, R_2, \ldots, R_{k-1}$, on some minimal path between the source and destination with R_0 and R_k being the source and destination nodes, respectively. For hypercubes (multidimensional meshes), the intermediate destinations are chosen among the corner nodes of the minimal subcube (mesh) that has the source and destination nodes as two of its corners. For a binary cube network, the number of links traversed in the i^{th} phase is simply the Hamming distance from R_{i-1} to R_i. Within each phase, the e–cube algorithm [66] is used for routing. e–cube, or *dimension order* routing, is a minimal algorithm which traverses the dimensions that must be routed in descending order. Figure 21 shows an example of ROMM routing on a binary cube network for $k = 4$.

If the number of phases exceeds the number of address dimensions to be routed, then the extent that needs to be routed in some dimensions are divided among two phases, for $n < k \leq 2n$. As k increases relative to n additional splittings of axes extents are made.

Minimality ensures that the bandwidth demand on the network is minimized given the data distribution. As k increases, more randomization becomes possible. This results in more path choices for a message, reducing the likelihood of congestion for many, but not all permutations. The scalability of ROMM routing is illustrated in Figure 22 in which routing time is plotted against network size for matrix transposition and routing to a single random destination for all packets in a node. In the figure, the plots on the right are scaled by the network bisection width. As predicted, the behavior observed on 256 node meshes scales, at least up to 1024 nodes.

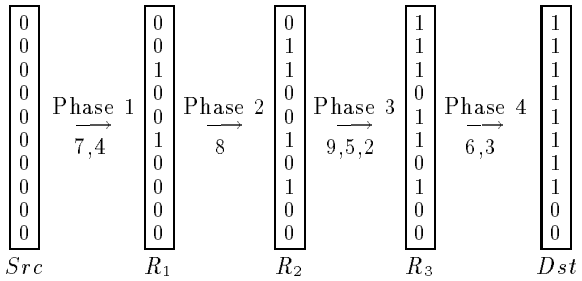

Figure 21: Using ROMM with $k = 4$ to route from 0000000000 to 1111111100 in C_{10}.

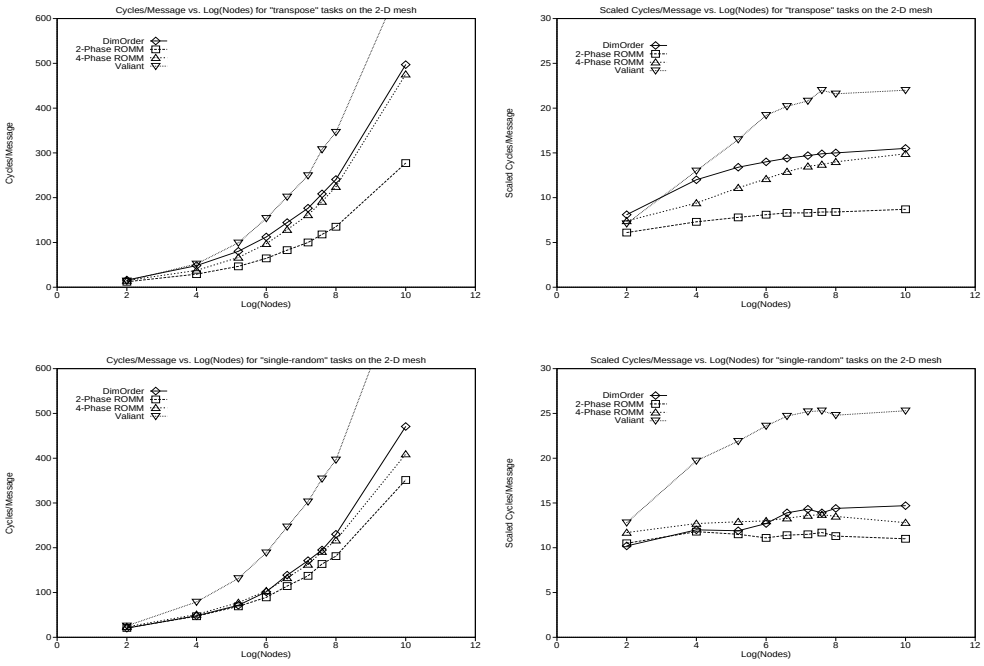

Figure 22: Graphs of cycles/message vs. mesh size, for transpose and single–random on square meshes.

5 Summary

We have reviewed some of the techniques used for limiting the bandwidth demands on interconnection networks, discussed some of the higher level communication primitives known as collective communication, and presented some of the techniques used for effective bandwidth utilization in electronic interconnection networks. The techniques allow for a very high degree of utilization of the available bandwidth in electronic networks. We believe that, despite the promise of considerably higher bandwidth in systems with optimal interconnect, some of the ideas may also be useful in such systems. The collective communication functions should be very valuable also in evaluating the design of such systems.

Acknowledgment

Much of the work reported on here was made in collaboration with several staff members at Thinking Machines Corp., in particular Kapil Mathur (now with David Shaw Inc) and Zdenek Johan (now with Centric Engineering) and with Ted Nesson while a Ph.D. student at Harvard University. We would also like to acknowledge the support of the Office of Naval Research through grant N00014-93-1-0192 and the Air Force Office of Scientific Research through grant F49620-93-1-0480 and the Los Alamos National Laboratory through grant 8283K0014-9U.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W. J. Paul, and D. Scheerer. On the Physical Design of PRAMs. *Computer Journal*, 36(8):756–762, December 1993.

[2] K. Bolding, S.-C. Cheung, S.-E. Choi, C. Ebeling, S. Hassoun, T. A. Ngo, and R. Wille. The Chaos Router Chip: Design and Implementation of an Adaptive Router. In *Proceedings of VLSI '93*, September 1993.

[3] 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.

[4] 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.

[5] P. Budnik and David J. Kuck. The organization and use of parallel memories. *IEEE Trans. Computer*, 20:1566–1569, December 1971.

[6] W. J. Dally and H. Aoki. Deadlock–Free Adaptive Routing in Multicomputer Networks Using Virtual Channels. *IEEE Trans. on Parallel and Distributed Systems*, 4(4):466–475, April 1993.

[7] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. Technical Report Reprint No. 1, Argonne National Laboratories, Mathematics and Computer Science Division, August 1988.

[8] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. Technical Report Technical Memorandum 41, Argonne National Laboratories, Mathematics and Computer Science Division, November 1986.

[9] J. Duato. A New Theory of Deadlock–Free Adaptive Routing in Wormhole Networks. *IEEE Trans. on Parallel and Distributed Systems*, 4(12):1320–1331, December 1993.

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

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

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

[13] High Performance Fortran Forum. High performance fortran; language specification, version 1.0. *Scientific Programming*, 2(1 - 2):1–170, 1993.

[14] 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.

[15] P. T. Gaughan and S. Yalamanchili. Adaptive Routing Protocols for Hypercube Interconnection Networks. *COMPUTER*, 26(5):12–23, May 1993.

[16] W. Morven Gentleman and G. Sande. Fast Fourier transforms – for fun and profit. In *Proceedings – Fall Joint Computer Conference, 1966*, pages 563–578. AFIPS, 1966.

[17] 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.

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

[19] 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.

[20] 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.

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

[22] Zdenek Johan, Kapil K. Mathur, S. Lennart Johnsson, and Thomas J.R. Hughes. Scalability of finite element applications on distributed–memory parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 119(1 – 2):61 – 72, November 1994.

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

[24] S. Lennart Johnsson. *Models of Massively Parallel Computation*, chapter 4, pages 223–389. Morgan Kaufmann, 1990.

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

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

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

[28] 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.

[29] 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.

[30] S. Lennart Johnsson and Ching-Tien Ho. Optimizing tridiagonal solvers for alternating direction methods on Boolean cube multiprocessors. *SIAM J. on Scientific and Statistical Computing*, 11(3):563–592, 1990.

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

[32] S. Lennart Johnsson and Ching-Tien Ho. Optimal communication channel utilization for matrix transposition and related permutations on Boolean cubes. *Discrete Applied Mathematics*, 53:251–274, 1994.

[33] 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.

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

[35] S. Lennart Johnsson and Kapil K. Mathur. Data structures and algorithms for the finite element method on a data parallel supercomputer. *International Journal of Numerical Methods in Engineering*, 29(4):881–908, 1990.

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

[37] C. Kaklamanis, D. Krizanc, and T. Tsantilas. Tight Bounds for Oblivious Routing in the Hypercube. In *Proc. of the 2nd Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 31–36. ACM Press, July 1990.

[38] S. Konstantinidou. Adaptive, Minimal Routing in Hypercubes. In *Proceedings of the 6th MIT Conference on Advanced Research in VLSI*, pages 139–153. MIT Press, 1990.

[39] S. Konstantinidou and L. Snyder. The Chaos Router: A Practical Application of Randomization in Network Routing. In *Proc. of the 2nd Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 21–30. ACM Press, July 1990.

[40] David J. Kuck and Richard A. Stokes. The Burroughs Scientific Processor (bsp). *IEEE Trans. Computers*, C-31(5):363–376, May 1982.

[41] Duncan H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. on Computers*, 24(12):99–109, 1975.

[42] Duncan H. Lawrie and Chandra R. Vora. The Prime Memory System for array access. *IEEE Trans. Computer*, 31:435–442, May 1982.

[43] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM TOMS*, 5(3):308–323, September 1979.

[44] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The Network Architecture of the Connection Machine CM–5. In *Proc. of the 4th Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 272–285. ACM Press, July 1992.

[45] Charles E. Leiserson. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Computers*, 34:892–901, October 1985.

[46] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Cral R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A St Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The network architecture of the Connection Machine CM–5. In *SPAA '92*, pages 272–285. ACM Press, 1992.

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

[48] Y.-D. Lyuu. *An Information Dispersal Approach to Issues in Parallel Processing*. PhD thesis, Harvard University, 1990.

[49] 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.

[50] 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.

[51] Kapil K. Mathur and S. Lennart Johnsson. All–to–all communication on the connection machine CM-200. *Journal of Scientific Progamming*, 1995.

[52] Kapil K. Mathur, Alan Needleman, and V. Tvergaard. Dynamic 3-d analysis of the charpy v–notch test. *Modelling and Simulation Materials Science and Engineering*, 1(4):467–484, July 1993.

[53] Gary L. Miller, Shang-Hua Teng, William Thurston, and Stephen A. Vavasis. Automatic mesh partitioning. In *Sparse Matrix Computations: Graph Theory Issues and Algorithms*. The Institute of Mathematics and its Applications, 1992.

[54] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *32nd Annual Symposium on Foundations of Computer Science*, pages 538–547. IEEE Computer Society, 1991.

[55] Gary L. Miller and William Thurston. Separators in two and three dimensions. In *Proceedings of the 22th Annual ACM Symposium on the Theory of Computing*, pages 300–309. ACM Press, 1990.

[56] Gary L. Miller and Stephen A. Vavasis. Density graphs and separators. In *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 331–336. ACM Press, 1991.

[57] J. Y. Ngai and C. L. Seitz. A Framework for Adaptive Routing in Multicomputer Networks. In *Proc. of the 1st Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 1–9. ACM Press, June 1989.

[58] G. D. Pifarré, L. Gravano, S. A. Felperin, and J. L. C. Sanz. Fully Adaptive Minimal Deadlock–Free Packet Routing in Hypercubes, Meshes, and Other Networks: Algorithms and Simulations. *IEEE Trans. on Parallel and Distributed Systems*, 5(3):247–263, March 1994.

[59] 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.

[60] 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.

[61] 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.

[62] Yousef Saad. *Numerical Methods for Large Eigenvalue Problems*. Manchester University Press, 1993.

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

[64] 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.

[65] 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.

[66] H. Sullivan and T. R. Brashkow. A Large Scale Homogeneous Machine. In *Proc. of the 4th International Symp. on Computer Architecture*, pages 105–124. IEEE, 1977.

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

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

[69] Shang-Hua Teng. *Points, Spheres, and Separators: a unified geometric approach to graph partitioning*. PhD thesis, Carnegie-Mellon University, School of Computer Science, 1991.

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

[71] A. M. Tsantilas. A Refined Analysis of the Valiant–Brebner Algorithm. Technical Report TR-22-89, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1989.

[72] L. G. Valiant. A Scheme for Fast Parallel Communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

[73] L. G. Valiant and G. J. Brebner. Universal Schemes for Parallel Communication. In *Proc. of the 13th Annual ACM Symp. on the Theory of Computing*, pages 263–277. ACM Press, May 1981.

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

[75] 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.