Data Permutations and Basic Linear Algebra Computations on Ensemble Architectures

S. Lennart Johnsson

Research Report YALEU/DCS/RR-367
February, 1985

Work supported by the Office of Naval Research under Contract No. N00014-84-K-0043

Data Permutations and Basic Linear Algebra Computations on Ensemble Architectures

S. Lennart Johnsson
Department of Computer Science
and Electrical Engineering
Yale University

Abstract

Ensemble architectures are interesting candidates for future high performance computing systems. The ensemble configurations discussed here are linear arrays, 2-dimensional arrays, binary trees, shuffle-exchange networks, boolean cubes and cube connected cycles. We discuss a few algorithms for arbitrary data permutations, and some particular data permutation and distribution algorithms used in standard matrix computations. Special attention is given to data routing. Distributed routing algorithms in which elements with distinct origin and distinct destinations do not traverse the same communications link make possible a maximum degree of pipelined communications. The linear algebra computations discussed are: matrix transposition, matrix multiplication, dense and general banded systems solvers, linear recurrence solvers, tridiagonal system solvers, fast Poisson solvers, and very briefly, iterative methods.

1. Introduction

High performance in future computing systems to an ever increasing extent has to come from concurrency in computation. The performance improvements over the last few decades have largely been due to technological developments. Switching speeds have been improved by about 5 orders of magnitude, but clock rates only by about 3 orders of magnitude. Architectural innovations have accounted for 2 - 3 orders of magnitude in increased floating-point capability per clock cycle, yielding a total improvement in floating-point speed of 5-6 orders of magnitude. Silicon technology is expected to offer about one order of magnitude increased switching speed before fundamental limits are reached. Other technologies, such as gallium arsenide, potentially offer a further factor of 5 to 10 in increased switching speeds.

Extreme performance in future systems must come from architectural innovations, in particular through highly concurrent systems. High real performance can also be accomplished through algorithmic innovations, and such innovations are in general necessary to obtain high *real* performance on a system of high *nominal* performance. The experience gained from current vector machine architectures, as well as of the ILLIAC IV, has made the interdependence between architectures and algorithms very clear. Some algorithms that are efficient on sequential machines are inherently limited in their ability to benefit from

realistic parallel architectures. Others can be adapted to a variety of such architectures. The adaptation may be accomplished through algorithm or program transformations.

Abstract representations of algorithms that capture the essence of their behavior are urgently needed, as are efficient transformation techniques that generate the appropriate data and control structures for the target architecture. A suitable abstract representation relieves algorithm designers from all the minute details of particular machine architectures. The variety of parallel architectures is indeed very large compared to uniprocessor architectures. In highly concurrent architectures the communication time is of prime importance. Entirely new algorithms that minimize the times of arithmetic operations and communications on classes of architectures are likely to be discovered. High real performance and the success of any architecture depends on effective programming models and systems. Algorithm transformation techniques will be an important part of such systems.

Many algorithms that are efficient on a single processor are limited in their ability to realize the potential of a parallel architecture. The computation of inner products is a prime example. For instance, a lower bound for the parallel arithmetic complexity of solving a linear system of equations by Householder transformations is $O(N\log_2 N)$, because of the need for inner product computations in each step of the algorithm. This complexity is the same as the parallel complexity of Gaussian elimination with partial pivoting [85]. The parallel arithmetic complexity of Given's rotations is $O(N)$, the same as Gaussian elimination without partial pivoting. The parallel arithmetic complexity of the conjugate gradient method is $O(N\log_2 N)$ [40], also because of the need for inner product computations/distributiions in each step of the algorithm. In estimates of the computational complexity in highly concurrent architectures it is also necessary to include the communication time. In the discussion of algorithms below we focus on the communication requirements.

An ideal architecture must be scalable both with respect to the desired performance and the technology. It is also highly desirable that the essential architectural characteristics are stable with respect to these forms of scaling. The technology, the architecture, the algorithms and the applications, and the programming models and systems, all have different dynamics. It is highly desirable that the dependence of any one of these components on any others is minimized in order that advances in any one area can be introduced at the rate they are produced.

2. Ensemble Architectures

Ensemble architectures represent a low cost alternative to future high performance systems. An ensemble architecture, in the sense used here, is composed of large numbers of relatively simple (small), mass produced processors, each equipped with local storage. The processor interconnections form a sparse, regular graph. Control, as well as data, is distributed. There are only a few shared resources where the bandwidth requirement is low, such as storage at some point in the hierarchy, and certain peripherals.

High performance requires a high rate of instruction execution, given that each instruction has a limited set of operands, and high storage bandwidth. In the ensemble architectures we use as models for algorithm development, high instruction execution rates are achieved through replication of processing units, each executing its own, distinct, instruction stream. We consider architectures classified by Flynn [21] as MIMD (Multiple Instruction streams Multiple Data streams) architectures. High storage bandwidth is achieved through a highly partitioned storage. Each processing unit has its own local storage. A high nominal performance is obtained through replication of mass produced parts implemented in state-of-the-art technology for an excellent cost/performance ratio.

A high performance design should also allow high clock rates in any given technology. Simple processors are smaller than processors with large instruction and register sets. Simple processors can be designed effectively with regular structures without severe area and performance penalties [27], [28], [74], [73], [92]. Pipelining of functional units that significantly adds to the complexity of scheduling operations can be reduced to a minimum. Wire delays are of increasing importance as feature sizes of the technology are reduced. Indeed, wire delays already are of main concern with respect to performance. The difference in the effort required for the design of a fast simple and a fast sophisticated processor will be magnified in the future.

The same argument applies to storage as well. A large on-chip storage based on an array design may actually become slower as the feature sizes are reduced. To maintain or improve the speed of storage with reduced feature sizes it ultimately becomes necessary to structure storage itself [67], which may reduce the density, and cause access time to be nonuniform. Intermingling of storage and processing reduces the average area per processing element, and increases the clock rate in a synchronous (non-pipelined) design. Both the increased ratio of processing capability per unit of storage, and the increased clock rate contribute to increasing the maximum size of the state that can change in a single clock cycle, i.e., the rate of computation.

An architecture achieving high nominal performance through replication of parts requires effective communication and synchronization. As the processors and their storage are reduced in size and increased in number, the interconnection problem increases both in complexity and importance. The speed of interprocessor communication relative to the speed of accessing local storage increases with increased locality of interconnection. Currently, interprocessor communication is generally inter-chip, or inter-board communication, which presents challenges different from on-chip communication.

The increased importance of interprocessor communication with reduced processor size also stems from the fact that with reduced granularity of the computations the amount of computation per communication decreases for large classes of problems. The sensitivity of the performance of the system

to different network configurations increases, as does the sensitivity to data allocations (and reallocations). In a system of course grain size the sequential time dominates the time for communication for most computations, reasonable communication speeds, and data allocations, regardless of the interconnection scheme.

Manufacturability, and scalability of ensemble architectures with respect to performance and the reduced feature sizes of the evolving technology, are assured by interconnecting the processing elements sparsely and regularly. The richness of the interconnect is a point of trade-off between performance, fault tolerance, and design and manufacturing concerns.

High real performance and scalability are accomplished by enforcing the concept of locality wherever possible. Control and storage are effectively distributed among the processing elements, eliminating potential bottlenecks and sources of limited scalability. High real performance is achieved by devising algorithms that minimize the time devoted to arithmetic and communication. Global communication is accomplished via a succession of local communications. Algorithms need to be devised with respect to the communication time required by the best possible embedding of the computation graph in the ensemble architecture. In some instances a static, properly embedded, data structure allows the computation to be carried out in minimum total time. Other cases require dynamic data structures (or reallocation of the data structure). The solution of tridiagonal systems of equations by cyclic reduction on ensembles configured as binary trees is an example of the latter kind of computation [41]. It is discussed in some detail later.

The need for data reallocations, as well as the ease of finding embeddings of computation graphs that have a low communication time depends on the topology of the ensemble. A topology is said to be more powerful than another if a sufficiently large class of algorithms that runs in a given time on the latter can be made to run on the former with an increase in running time by at most a constant factor, but the converse is not true. For instance, a shuffle-exchange network is considered more powerful than a binary tree, since any tree algorithm can be made to run on a shuffle-exchange network with the running time increased by at most a constant factor, but the converse is not true. The computation of the Fast Fourier Transform is a prime example. The running time of an algorithm on a shuffle-exchange network may be decreased by a constant factor if the algorithm is mapped on to a boolean n-cube. For a discussion on the computational power of some networks see [107]. Many graphs can be embedded in a boolean cube preserving proximity. A given data structure can support many types of access schemes without communication penalty, reducing the need for data reallocations. Reconfigurability, i.e., programmable interconnections, also allows a fixed data structure to efficiently support a variety of access schemes.

A high real performance for data independent computations is attained by a *compile time* mapping of the computations on to the processors of the ensemble. Whenever it is possible to find a set of rules for the mapping of a computation with data independent control flow on to the ensemble, the mapping can be made part of the compilation process. The user is relieved of the need to have detailed knowledge of the architecture. Where compile-time mapping is not used an effective *run-time* mapping is required. Such a mapping must take the processor load as well as communication needs and resources into account.

The idea of ensemble architectures is not new. Early examples are the PEPE (Parallel Element Processing Ensemble), and the ILLIAC IV. A significant difference between the early ensemble architectures and several of those currently being proposed, or built, exist in the programming model, the storage organization, and in the communication. The early machines were SIMD (Single Instruction stream Multiple Data streams) architectures. The Cosmic Cube [91] is a MIMD architecture with processors with local storage configured as a boolean 6-cube (64 processors). There is no global storage. The INTEL iPSC with up to 128 nodes is a commercial version of the Cosmic Cube. A similarly configured machine with processors of much more limited capability is the Connection Machine [30]. The Ultracomputer [89], [25] is another MIMD architecture that differs from the Cosmic Cube and the Connection Machine in that it has a global storage. Processors with very limited storage are interconnected to the shared storage via an Ω-network [58]. The Heterogeneous Element Processor (HEP) [98] is a similarly configured MIMD architecture, but with substantially more powerful processors and a corresponding lower degree of potential concurrency [90], than in the Ultracomputer. Another proposed ensemble architecture of the MIMD type is the Texas Reconfigurable Array Computer, TRAC [93]. The TRAC has processors and storage modules on opposite sides of a switch network, like the Ultracomputer. The CHiP machine [99] is also a reconfigurable architecture. It consists of processors with local storage interconnected via programmable switches. The Tree Machine [8], [9] is a tree configured MIMD ensemble. Processors have local storage. The Non-Von computer is another tree configured ensemble. It is of the SIMD type [95]. The proposed Non-Von IV is of the MIMD variety [96]. Finally, the CEDAR project [55] should be mentioned as an example of a hierarchical architecture of the MIMD type with both storage local to a processor, as well as shared storage accessible via a switch network. For further references on multiprocessor architectures see Hwang [35].

3. Ensemble Configurations

The communication capabilities are of prime importance in an ensemble architecture. The selected processor interconnection scheme affects scalability, wireability, and performance. Clearly, from a designers point of view it is desirable to minimize the amount of interconnect, whereas from a performance point of view a high bandwidth between arbitrary processors is desirable. The feasibility of different forms of interconnect from a design point of view is related to the granularity of the processors: a large number of fine grain processing elements may fit on a single chip, but there may be pin-out

difficulties in going off chip, and further difficulties in interconnecting a large number of processors on chip. The following is a brief review of some of the qualitative communication and design characteristics of a few processor interconnection networks which are considered in the section on ensemble architecture algorithms. A survey of interconnection networks is given by Siegel [97], see also Wu [112].

Configuring processors as linear arrays and binary trees requires a total number of interconnections equal to the number of processors. Both configurations scale in an excellent way. With several processors on a single chip, the required bandwidth at the chip boundary grows at the rate of the clock frequency only, regardless of the number of processors per chip and the size of the machine being built [62]. The tree has an advantage over a linear array for global communication in that the maximum number of routing steps between two processors merely grows logarithmically with the number of processing elements.

However, linear arrays and trees do present communication bottlenecks for large classes of computations. Richer interconnection networks are 2-dimensional meshes, shuffle-exchange networks, cube connected cycles [75], and boolean n-cubes. The number of interconnections per processor (the fanout), is 3 for almost all processors in a shuffle-exchange network (half the processors in a binary tree also have fanout 3), 4 for the square 2-dimensional mesh, 6 for the hexagonal mesh, and $\log_2 N$ for a boolean cube. The number of edges (communication links) in a binary tree of 2^n-1 nodes is 2^n-2, in a 2-dimensional mesh of 2^n nodes it is 2×2^n, in a shuffle-exchange graph approximately 1.5×2^n, and in a boolean cube of 2^n nodes it is $n/2\times 2^n$. The diameter of the network topology defines a lower bound for the speed of computation [23]. The diameter of a binary tree of 2^n-1 nodes is 2(n-1). For the shuffle-exchange network of 2^n nodes it is 2n-1, and for the boolean n-cube (2^n nodes) it is n. A 2-dimensional square mesh has a diameter of $2\sqrt{N}$ without end-around connections, $\sqrt{N}$ with end-around connections. Binary trees and shuffle-exchange networks offer a small diameter for few interconnections. The shuffle-exchange networks are more powerful than the binary tree, but the tree is a recursive structure, which implies several advantages with respect to construction and programming.

The power of the boolean cube and cube connected cycles networks are obtained at the price of an increased wiring complexity. The wires also require volume, or area in the event the configuration should fit in a plane. In the technology of the future a substantial number of processors will fit on a single chip, even with processors being of significant complexity. Three dimensional technologies may become feasible at some point in the future, but here we use the required planar area as a measure of the space needed by the different configurations. A linear array, 2-dimensional mesh, and a binary tree using the H-tree layout [8], each requires an area of $O(N)$. Complete binary tree layouts with all leaf nodes along the perimeter require area $O(N\log_2 N)$ [7]. Shuffle-exchange networks [61] and the cube connected cycles network [75] each requires an area of $O(N^2/\log_2^2 N)$, and the boolean cube an area of $O(N^2)$ [62]. Wire

length is another important measure of the characteristics of a layout. For the binary tree, the shuffle-exchange and cube connected cycles network, and the boolean cube, the wire lengths are nonuniform. The minimum of the maximum wire length is of order $O(\sqrt{N}/\log_2 N)$ for the binary tree [72], [82]. For the shuffle-exchange network the lower bound is $\Omega(N/\log_2^2 N)$ and the upper bound $O(N/\log_2 N)$. For a boolean cube the minmaxx wire length is $O(N)$ [61].

The number of interchip (or interboard) connections grows at a lower rate for the mesh than for shuffle-exchange networks, that in turn grows at a lower rate than the number of interchip interconnections for a boolean n-cube [6]. The n-cube has the largest fanout of the configurations considered here, and requires the largest area, but it offers higher bandwidth between arbitrary processors than any of the other alternatives. Another definite advantage of the boolean cube over shuffle-exchange networks is that it is a recursive structure. Extensions of a cube do not require a complete rewiring, but for an easy extension it needs to be anticipated at design time so that a sufficient number of ports are available.

An alternative to direct links between processors is some form of switch network, such as an Ω-network. The bandwidth of a pipelined Ω-network, like that planned for the proposed NYU Ultracomputer [25], is essentially the same as the bandwidth in a boolean cube, given that the two alternatives have an equal number of processors. However, the latency in the switch grows logarithmically with the number of processing elements. In the boolean cube the number of routing steps for interprocessor communication is nonuniform. The minimum number of routing steps is 1, the maximum is $\log_2 N$. Interprocessor communication in an architecture of the Ultracomputer type always requires $2\log_2 N$ routing steps.

For a large cube, or a large switch, the time for interprocessor communication in the cube, and the time between stages of the switch, is likely to grow with the number of processors, due to increased distance of communication. Though there is a qualitative difference in the communication capabilities, the quantitative difference depends on various implementation decisions. Furthermore, whether the nominal difference results in a real performance difference depends on the particular data dependences of the computation, and the mapping of the computations on to the architecture. With simple switching elements, and several elements on a chip, the fanout may become a problem in a switch based architecture at an earlier stage in the evolving technology than in a machine with processor-to-processor connections. One can view such an architecture as one in which the complexity of the switches is increased to yield regular processing nodes.

4. Ensemble Architecture Algorithms

An ensemble architecture of extreme concurrency is similar to systolic architectures. However, in ensemble architectures data management is an even more predominant factor. In most, but not all, systolic architectures most of the data (input and output) are stored outside the array, and the

management of such data is generally ignored. In algorithms for ensemble architectures it is generally assumed that initial data, as well as the results, are stored within the ensemble. Furthermore, the number of nodes in the ensemble is, in general, insufficient to match characteristic parameters of the problem, but the amount of storage per node is significant. The granularity of computations in ensemble architectures is often larger than in systolic architectures.

Time-space trade-offs are at the core of mapping algorithms onto ensemble architectures. Data and control structures, synchronization and communication are, in general, considerably more complex in ensemble architectures than in systolic designs. The time-space trade-off in ensemble architectures is often made in favor of minimizing data movement. Systolic designs are of fine grain and designs are often such that the communication time is comparable to the time for logic or arithmetic operations. Ensemble architectures are of a coarser grain and interprocessor communication is typically slower than the execution of arithmetic and logic operations.

Algorithms are devised both in an ad hoc manner and systematically. The first approach may lead to entirely new, efficient, algorithms. The second approach can be supported by algorithm design tools, and provides the necessary insight to develop compilation techniques that transform abstract representations of algorithms into efficient code for a variety of architectures. In the systematic approach, which is followed in the description of sample algorithms below, a *computation graph* defining the partial ordering of computations is created from the definition of the computation in a suitable notation. Then, this computation graph is mapped on to the ensemble. We often carry out this mapping in two stages. First, a mapping is carried out for the case of a sufficiently large ensemble. In such an ensemble the mapping of the computation graph can be made such that all nodes of a given level (order) are assigned to distinct processors. The situation in this case is similar to what is typical for systolic designs [47], [49], [14], [65], [70], [69], [76], [64], [16], [18]. In the second step the actual computation graph is mapped on to the ensemble. This step can often be thought of as a folding of the computation graph on to itself, until the reduced graph is of a size that fits the ensemble.

In folding the computation graph and mapping it on to a specific ensemble architecture several performance related issues arise that do not occur in designs of the systolic type. For instance, in such a design it is often sufficient for maximum utilization of resources that no two elements compete for the same communications link at the same stage in the execution of the algorithm. But, in an ensemble architecture it may be required that for maximum performance communications during different stages of the algorithm do not compete for the same communication link. The ability to establish different communication paths with a minimum number of shared edges becomes important. The size of the problem relative to the size of the ensemble also affects the optimum embedding in other ways due to restricted communication.

With larger granularity of computations operations are no longer occurring concurrently to the extent disclosed by the computation graph. A parallel algorithm that minimizes the time for arithmetic operations on an unbounded number of processors may have a higher total operations count than an algorithm minimizing the number of arithmetic operations. Bitonic sort and odd-even cyclic reduction are examples thereof. In order to minimize the required solution time on an ensemble of finite size a combination of algorithms may be needed. In some instances such combinations can be obtained through algorithm transformations. Which algorithm minimizes the execution time may also depend upon the number of problems to be solved in that some algorithms are more amenable to pipelining than others.

We will describe some basic ensemble architecture algorithms for computational linear algebra and sorting, and focus on the issues raised above. The ensemble architecture topologies used as model architectures are linear arrays, 2-dimensional meshes, binary trees, shuffle-exchange, boolean cube, and cube connected cycles networks. Before discussing the algorithms we present a few results on graph embeddings that are used in some of the algorithms.

4.1. Graph Embeddings

The computation graph defines a partial ordering of the computations. Constraints on the realization of the computation graph are imposed by the ensemble architecture, and are incorporated in the mapping process. The computations corresponding to the nodes at a given level (order) have to be spread over time if there are an insufficient number of nodes in the ensemble, or if the communication implied by directed edges may require more than one communication step. This situation occurs if the operands at the source and sink of the edge are located at nodes at a distance greater than 1 in the ensemble. In the interest of conserving storage, nodes of the computation graph are sometimes identified with a given storage location. Such a strategy results in a variety of access schemes for the same data structure. If the storage has a latency then this strategy has its cost in that the latency may determine the rate of execution during some part of the algorithm, as is the case for familiar computations such as FFT, and odd-even cyclic reduction [12] on vector architectures. The bank conflict problem in vector processors is well known, and architectural solutions [10], [58], [60], [78], as well as solutions at the application program level for particular algorithms [51], [34] have been proposed.

The situation is the same within a node in an ensemble architecture, but in addition the time of accessing storage is not uniform. In a simplified model the storage of nodes with which a given node has direct connections can be accessed in 1 unit of time, the storage of the neighboring nodes of the immediate neighbors in 2 units of time, etc. The larger the number of neighbors the larger the number of different access schemes that can be supported by a fixed data structure at a given number of communication actions. Reconfigurability through switchable interconnections gives the ensemble the same property.

Commonly appearing data structures are linear or multidimensional arrays. Rosenberg and Snyder [81], based on results of Sekanina [94], have given a procedure for a *proximity preserving* embedding of a 2^n-1 node loop in a 2^n-1 node binary tree. Let $d_L(i,j)$ be the distance between nodes i and j in the loop, and $\phi(i)$ and $\phi(j)$ be the embeddings of nodes i and j in the tree. Then, $d_T(\phi(i),\phi(i+1)) \leq 3$ for all i, where $d_T(i,j)$ is the distance between nodes i and j in the tree. They have also shown that $\sum_{i=0}^{|L|-2} d_T(\phi(i),\phi(i+1)) \leq 2(|L|-1)$, i.e., the average distance between adjacent nodes is less than 2. $|L|$ denotes the length of the loop. For any embedding of 2-dimensional arrays of n by n nodes in the leaves of a complete binary tree DeMillo, Eisenstat, and Lipton [19] have shown that there exist nodes (i,j) and (i+1,j), adjacent in the array, such that $d_T(\phi(i,j),\phi(i+1,j)) > \log_2 n - 3/2$. Rosenberg and Snyder [81] show that the average distance for the embedding of a 2-dimensional array in the leaves of a binary tree is at most $7-2^{-\lfloor \log_2 n \rfloor+1}$. Rosenberg and Snyder also consider the embedding of d-dimensional arrays with n^d nodes in the leaves of 2^d-ary and binary trees. The average distance between nodes adjacent in a d-dimensional array when embedded in a 2^d-ary tree is at most $4-2^{-\lfloor \log_2 n \rfloor}$. The bound for a binary tree is $(4-2^{-\lfloor \log_2 n \rfloor})d$. The maximum distance is at most $2d\log_2 n$ for the binary tree embedding.

Nodes in a boolean cube can be given addresses such that the addresses of adjacent nodes differ in precisely 1 bit. Furthermore, the number of adjacent nodes for any node equals the number of dimensions of the cube, i.e., the number of bits in the address. A loop embedding that preserves proximity is easily obtained for $|L|=2^n$ by encoding the indices of the nodes in the loop in a *binary-reflected* Gray code [79]. Such Gray codes have several interesting properties. For instance, it is easy to show that $d_C(G_i,G_{(i+2^j) \bmod 2^n})=2$ for $j > 0$. This property is important for algorithms such as the FFT, bitonic sort, and cyclic reduction. For the FFT and bitonic sort an embedding according to a direct binary encoding of the indices of the data elements is preferable. However, application programs typically include the use of several different "elementary" algorithms, and a Gray code embedding may be preferable for other computations.

Another important property of the binary-reflected Gray code is that for i even $d_C(G_i,G_{(i+3) \bmod 2^n})=1$. This property implies that any loop of length $2^{n-1}+2k$, $k=\{1,2,\ldots,2^{n-2}\}$ can be embedded in a n-dimensional cube (n-cube) such that $d_C(G_i,G_{(i+1) \bmod |L|})=1$ for $i=\{0,1,\ldots,|L|-1\}$. For $|L|$ odd there exists a node i in the loop such that $d_C(G_i,G_{(i+1) \bmod |L|})=2$. That the minimum maximum distance must be 2 is easily proved, since if it is equal to 1, then the cube would contain a cycle of length 3, or equivalently, that adjacent nodes would differ in 2 bits. In the following we refer to binary-reflected Gray codes simply as Gray codes.

An embedding according to the binary encoding of node indices in a loop does not preserve proximity. For i even $d_C(\phi(i),\phi(i+1))=1$, but for i odd $d_C(\phi(i),\phi(i+1))$ falls in the range $[2,n]$ $(d_C(\phi(2^{n-1}-1),\phi(2^{n-1}))=n)$. A Gray code encoding $G_i=(g_{n-1},g_{n-2},\ldots,g_0)$ can be rearranged to a binary

encoding $i=(b_{n-1},b_{n-2},\ldots,b_0)$ in n-1 routing steps. The highest order bit in the Gray code encoding of an integer, and the highest order bit in its binary encoding coincide. The encodings of the last element, N-1, differ in n-1 bits. An element needs to be routed in dimension j if $g_j \oplus b_j=1$. Routing the elements such that successively lower (or higher) order bits are correct yields paths that intersect at nodes only [44]. This form of routing amounts to reflections around certain "pivot" points in the Gray code. The pivot points are defined by the transitions in the bit being subject to routing. Figure 4-1 illustrates the sequence of reflections that convert a 4-bit Gray code to binary code. A reflection consists of an exchange of elements between a pair of processors. Since each dimension is routed only once, no two elements traverse the same edge in the same direction during the entire process of data reallocation. If there are multiple data per node the routing of elements can be pipelined without conflict. This property is important if a processor can concurrently support communication on all of its communication links.

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

Figure 4-1: Conversion of Gray code to binary code

The embedding of d-dimensional meshes with n_{d_i} nodes in dimension i is easily accomplished by partitioning the address space such that there are $\lceil \log_2 n_{d_i} \rceil$ bits (dimensions of the cube) allocated for dimension d_i of the array. For $n_{d_i}=2^k$ for some k this simple embedding is also efficient in the use of nodes in the cube.

4.2. Data Structures for Sample Problems

Next we present some algorithms for general data permutations (sorting), and routing algorithms used in forming the transpose of a matrix, multiplying matrices, and solving dense and banded systems of linear systems of equations. A natural data structure for matrix problems is a 2-dimensional array. For a boolean cube configured ensemble two obvious candidate embeddings of a 2-dimensional array are a proximity preserving embedding by separately encoding the row and column indices in a Gray code, or by a separate binary encoding of the row and column indices. For an N by N matrix and a 2k-cube with

$N^2 > 2^{2k}$, elements of the matrix have to be identified, and stored in the same node of the ensemble. We consider two schemes of identifying matrix elements with nodes of a 2^k by 2^k array.

In *consecutive* storage all elements (i,j)=$\{0,1,..,N-1\}$ of a matrix A are identified and stored in processor (p,q) if p=$\lfloor i2^k/N \rfloor$ and q=$\lfloor j2^k/N \rfloor$. Each processor stores a submatrix of size $N^2/2^{2k}$. In *cyclic* storage the matrix elements are stored such that elements (i,j) are identified with node (p,q) if p=imod2^k, and q=jmod2^k. The consecutive and cyclic storage schemes are illustrated in Figure 4-2.

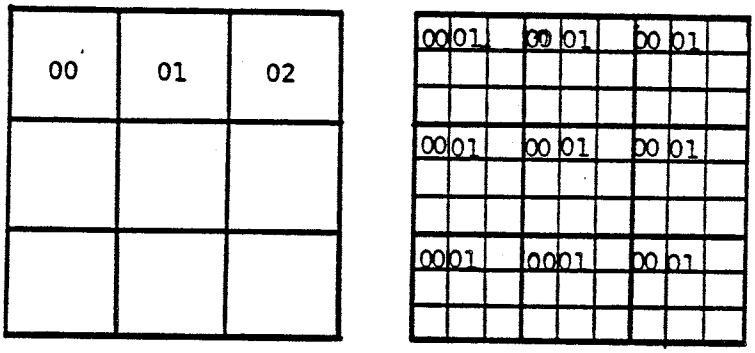


Figure 4-2: Consecutive and cyclic storage of a matrix

With the consecutive storage scheme algorithms devised for the case of N=2^k can be employed with the apparent change of granularity. Operations on single elements are replaced by matrix operations. In the cyclic storage scheme the processing elements can be viewed as forming a processing plane, and the submatrices as forming storage planes, Figure 4-3.

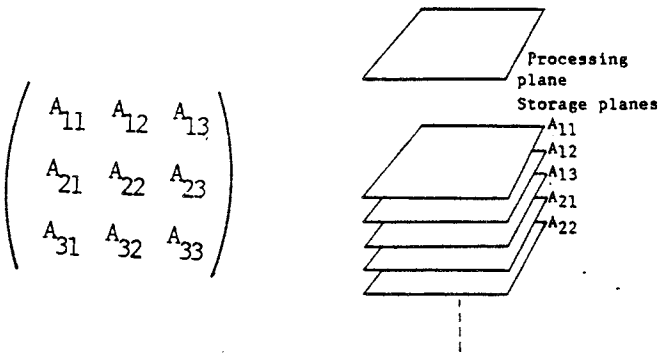


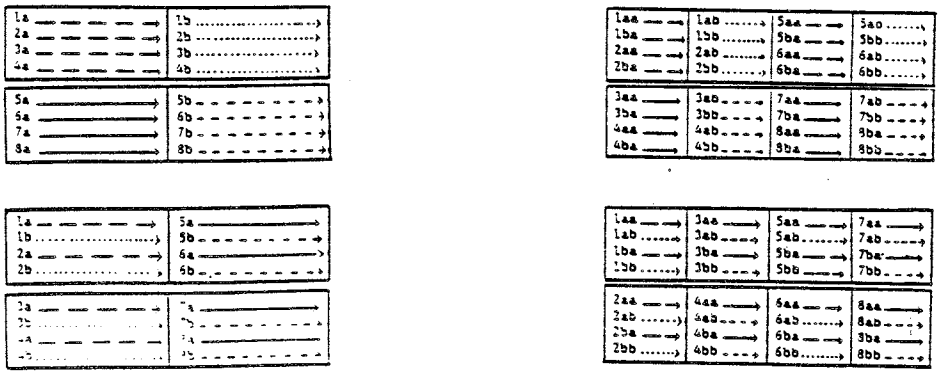
Figure 4-3: Processing and storage planes

We find that the cyclic storage scheme enforces a greater insight into the communication and storage management issues. Elemental operations are of fine grain. For an ensemble architecture with communication overhead that is nonzero, or that is not proportional to the number of elements communicated, and that has pipelined arithmetic units, operations of fine grain should, in general, be merged for optimum use. Conversely, if the consecutive storage scheme is used it may be desirable to

partition the elemental operations to increase the utilization of the ensemble. Either storage scheme allows for the same optimum use of the resources for the computations treated here. The optimization of vector length, or communication packet size does not affect the optimum allocation of data, or the choice of algorithm. The main focus here is on the data and control structures that yield a complexity of minimum order. The detailed optimization that accounts for specific architectural parameters is left out.

Rearrangement of consecutive to cyclic storage order (or vice versa) can be carried out in time $N/2^k+k$ for N elements stored in a 2^k processor boolean cube. For this communication complexity it is required that a processor can support communication on multiple ports, and that the communication for successive stages can be pipelined. We will now show that pipelining of successive stages is possible. Consider the local storage of each node as a column of a matrix that has as many columns as there are nodes in the array. Cyclic storage corresponds to storage of the elements in row major order, and consecutive storage to storage of elements in column major order. Clearly, if the number of rows is the same as the number of columns, one is obtained from the other through a matrix transpose operation. However, unlike in forming a matrix transpose the number of rows and columns does not change with the storage form.

The rearrangement can be made recursively by exchanging elements between pairs of processors differing in successively lower order address bits. The processors with addresses in the lower half of the processor address space exchange the elements of the upper half of their local address space with the contents in the lower half of the local address space of their corresponding processors in the upper half of the processor address space. The result is that the first half of the processors contain the first half of the elements. An unshuffle operation on local addresses (or shuffle operation on the data) brings the first half of the data into row major order in the processors with addresses in the lower half of the address space, and the second half into row major order in the second half of the set of processors. The procedure is repeated recursively for each half independently, and concurrently. Figure 4-4 gives an example.



Recursion step 1 Recursion step 2

Figure 4-4: Transforming cyclic storage to consecutive storage

Clearly, the local shuffle operation need not be carried out explicitly. Note that exchanges are always

performed on half of the local address space, regardless of recursion step, or the number of rows or columns. This property is not true in forming the transpose of a rectangular matrix. Note further that the data transfers can be pipelined.

Carrying out the recursion in reverse order transforms a consecutive storage order to a cyclic storage order.

4.3. Data Permutations

4.3.1. Combining sequential and bitonic sort on a boolean cube

Sorting algorithms can perform general permutations. The bitonic sort [4] fits the topology of a boolean cube well. Stone [101] observed that the bitonic sort maps well on to shuffle-exchange networks. From Stone's observations the implementation on a boolean cube is immediate for one element per node. We will describe two algorithms for sorting N evenly distributed elements on a $K=2^k$ processor boolean cube for $N > K$.

The bitonic sort merges sorted sequences recursively. With one element per node the algorithm proceeds by comparison-exchange operations on elements that are located in nodes differing in 1 address bit, say the lowest order bit. Then, two sorted sequences stored in two 1-cubes are merged into one sorted sequence in a 2-cube. The sorting order, nonascending or nondescending, is determined by a mask. The mask is a function of the processor address and the length of the subsequences being merged. In all, $\log_2 N$ sequences are merged serially. The number of sequences merged concurrently decreases from N/2 to 1. The final step merges two sequences stored in separate (k-1)-cubes into one sequence in a k-cube. The number of routing steps is k(k+1)/2, independent of the data. Each routing is performed in only one dimension. An algorithm for the bitonic sort expressed in pseudo code is as follows:

```
For i:=1,2,...,n do
  If i<n do
    nodes (a_{n-1},..., a_{i+1}, a_i, a_{i-1},..., a_0), a_i=1, set mask=1.
    nodes (a_{n-1},..., a_{i+1}, a_i, a_{i-1},..., a_0), a_i=0, set mask=0.
  end
  For j:=i-1,i-2,...,0 do
    nodes (a_{n-1}, ..., a_{j+1}, 1, a_{j-1}, ..., a_0), send their elements to
    nodes (a_{n-1}, ..., a_{j+1}, 0, a_{j-1}, ..., a_0), which compare local
    and received elements
    nodes with mask=0 keep the smaller element and
    nodes with mask=1 keeps the larger
    rejected elements are sent to (a_{n-1}, ..., a_{j+1}, 1, a_{j-1}, ..., a_0)
  end
end
```

If there are N elements to be sorted on a k-cube and $N > 2^k$, then additional sequential steps are necessary. Sorting of the elements into consecutive and cyclic order is considered. Assume for simplicity

that $N=2^n$. The last merge operation involves sequences of length N/2. The merge is accomplished through a sequence of comparison-exchange operations on subsequences that decrease in length by a factor of 2 for each step.

With cyclic storage order the first n-k comparison-exchange operations are local to a node. The result is 2^{n-k} bitonic sequences ordered with respect to each other. Each sequence has one element per node. The last k steps are separately performed on each of those sequences. Carrying out the first n-k local steps as bitonic sort yields poor performance. The operational complexity of bitonic sort is $O(N\log_2^2 N)$, compared to $O(N\log_2 N)$ for a good sequential sort. The first n-k steps can be carried out as a local merge operation concurrently in all processors, requiring at most 2^{n-k} comparisons (instead of $(n-k)2^{n-k}$), [42]. The correctness of the algorithm can be proved by observing that the first n-k steps, given the assumed storage order, realize 2^k independent bitonic sorters each for 2^{n-k} elements, and that corresponding output elements from these sorters form a bitonic sequence. The last k steps realize 2^{n-k} bitonic sorters for sequences of length 2^k. The situation is a generalization of Batcher's construction [4] of a 16-sorter out of 4-sorters, see Figure 4-4.

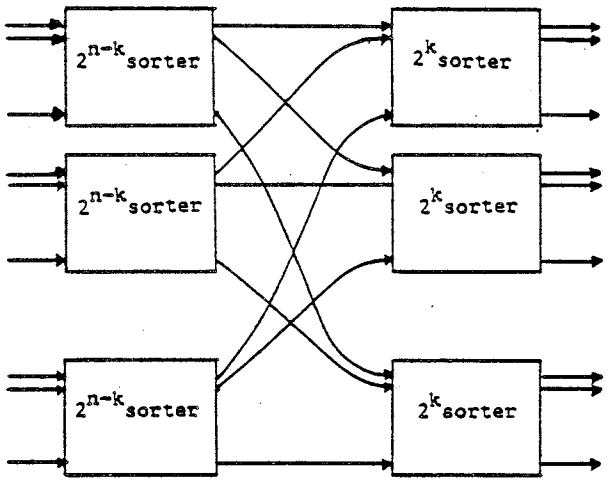


Figure 4-5: A network of 2^k 2^{n-k}-sorters followed by 2^{n-k} 2^k-sorters.

The time for cyclic sort by the algorithm outlined above is $T = 2^{n-k-1}(k(2n-k+1)(4t_c+t_{ce})/2 + 2(n-k-1)t_{ce}) + t_{ce}$, where t_c is the time for communication of an element between a pair of processors, and t_{ce} is the time for a comparison operation. If $N \gg 2^k$ then T is of order $O((N/2^k)\log_2 N$ and if $N \approx 2^k$ T is of order $O(\log_2^2 N)$. The speed-up is linear for a small cube and gradually changes to $O(N/\log_2 N)$.

With sorting into consecutive storage order the recursion can be carried out such that an increasing number of elements in a node is part of a given sorted sequence, as in the cyclic sort. It is also possible, as suggested by Baudet and Stevenson [5], to carry out the recursion over the number of nodes included in

the sorted sequence. A local sort is performed initially. In each recursion step all elements in a node belong to the same sorted subsequence. Bitonic sort is used for internode sort. The communication and comparison complexity is of the same order as for sorting in cyclic order [42].

The running time of bitonic sort does not depend on the data distribution. This property is a drawback for nearly sorted sequences. The data movement in such instances can be reduced at the expense of additional logic for determining what subsequences should be exchanged. Such a modification can be made while preserving one advantage of bitonic sort, namely that the number of elements per node is kept constant during the sorting process.

4.3.2. Distribution counting

Rank assignment in the context of distribution counting [52] with L counters, or "buckets", can be carried out in a time of at most $[N/2^k+L+k-1]2t_a + [L(1-1/2^k)3+k]2t_c$ for $2^k \leq L$, and $[N/2^k+L+k-1]2t_a + [6(L-1)+5k-3\log_2 L]t_c$ for $2^k > L$, on a boolean cube [42] (t_a is the time for an arithmetic operation). For few processors and a large number of elements compared to the number of buckets the algorithm offers linear speed-up. If the number of buckets is comparable to the number of elements to be sorted the speed-up is sublinear. For few buckets and few elements per node the speed-up is of order $O(N/\log_2 N)$. The rank assignment algorithm that yields the complexity estimates above is data independent, as are the algorithms based on bitonic sort. The rank assignment algorithm is easily modified to deal only with non-empty buckets, which is efficient if only a few buckets in each node are populated. For particular distributions of elements the data dependent version will have a complexity of order $O(\log_2 L)$ in the number of buckets, as in Hirschberg's shared storage model [31].

In the rank assignment algorithm multiple binary tree like computations are carried out concurrently. There are L binary trees with N leaf nodes each. The trees form subtrees of a tree with a total of $\log_2 NL$ levels. Each node has a local copy of each bucket. To find the total number of elements in any bucket the number of elements in each of all the different copies of a bucket have to be added. This addition is carried out by the subtrees of $\log_2 N$ levels. For the rank assignment partial sums are distributed from the root of the trees to the leaves. However, first an accumulation over all global bucket sums has to be performed. For each tree the summation/rank assignment process is carried out by recursive doubling [53].

The recursive doubling process is carried out concurrently in all subtrees of height $\log_2 N$, by rooting the subtrees in different nodes of the ensemble. The global sums of different buckets are contained in distinct nodes. The tree embedding is such that one node in the boolean cube contains $\log_2 N$-1 non-leaf nodes of a subtree, another cube node $\log_2 N$-2 non-leaf nodes of the same subtree, yet 2 other nodes $\log_2 N$-3 non-leaf nodes of the same subtree, etc. After the first step half of the subtrees are treated by distinct halves of the cube. The divide-and-conquer process is repeated recursively. The speed-up for the subtrees of height

$\log_2 N$ is of order $O(N)$ for L of at least order $O(\log_2 N)$. The top $\log_2 L$ levels of the tree are embedded similarly. Figure 4-6 illustrates the computations for N=L=4.

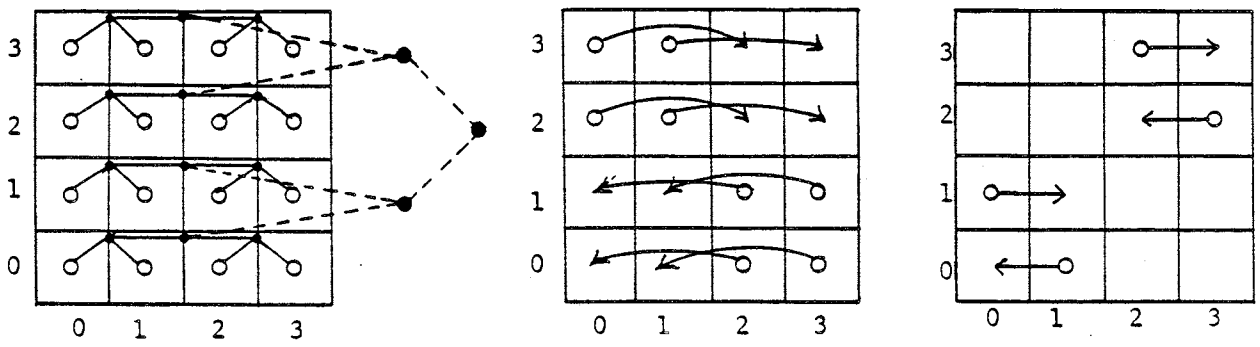


Figure 4-6: Concurrent tree computations on a boolean cube

4.3.3. Randomized Communication

Recently, several probabilistic algorithms of complexity $O(\log_2 N)$ for arbitrary permutations on a boolean cube and d-way shuffle networks have been devised. The probabilistic algorithms do not guarantee an even distribution of elements during permutation. (However, combined with a rank assignment algorithm the permutation can be specified so that the distribution on completion of the permutation is even, e.g., for load balancing). The probabilistic algorithms have two phases. First, the elements are routed to a random location, then the elements are routed to the final destination. The routing is deterministic.

During routing from the initial location to the final destination during either phase, several elements may reach a node and then be delayed because of competition for a given communications link, even in the case that the there is precisely one element per node in initial and final states. Also, several elements may reside in a single node at the end of the first phase. Valiant and Brebner [109] show that for a boolean cube with one element per node initially, and after the permutation, the queue length with high probability is at most of order $O(\log_2 N)$. Indeed, for N/K elements per node they show that the probability that the permutation will require more than $(\alpha N/K+1)\log_2 K$ routing steps is less than $(e/2\alpha)^{\alpha(N/K)\log_2 K}$. They also establish similar bounds for so called d-way shuffle networks (in- and out-degree of a node is d), which also are considered by Upfal [108] and Aleliunas [3]. It is assumed that a processor can support communication on all its ports concurrently. Valiant and Brebner also show that for a k-dimensional mesh with $N=n^k$ nodes, the probability that at least one packet has not finished in time $(2k-1)(n+\alpha n^{3/4})$ is less than $C^{\alpha n^{1/2}}$ for $C < 1$. This result compares favorably with the complexity of Batcher's bitonic sort or odd-even merge on meshes [106], [71], [56]. The Thompson and Kung algorithm yields a complexity of approximately $6\sqrt{N}$ for a 2-dimensional mesh, and $(3k^2+k)N^{1/K}$ for a k-dimensional mesh. Simulations that exhibit a behavior well within the bounds for a variety of ensemble configurations are also presented.

4.3.4. One-to-Many Distribution

One-to-many distribution is often called broadcasting. We avoid that term here since it in some architectures is reserved for particular means of accomplishing data distributions. There exist many ways of accomplishing the data distribution task in most ensemble architectures. Critical issues are termination, completeness and uniqueness, i.e., that all nodes have received the message precisely once upon termination. Furthermore, local control of the distribution algorithm is desirable. As an example we present the one-to-many distribution algorithm that is used in the Gaussian elimination algorithm described later. Pivoting is assumed to take place on the diagonal. The distribution algorithm must guarantee correct order of arrival in addition to the general properties, when used in conjunction with the elimination algorithm.

One possible algorithm is outlined below. A ticket included in the message carries information about the order in which dimensions shall be routed, and what dimensions have already been routed. The routing order is determined by the source node (but need not be). The routing order is the order in which the dimensions are used in the Gray code encoding of the indices $(i+j) \bmod (2^k)$ for monotonously increasing values of $j=\{1,2,3,..., 2^k-1\}$. Let G_i be the Gray code encoding of i. The algorithm is described in a synchronous mode of operation to simplify the arguments of the proof of completeness and uniqueness.

One-to-all distribution with node G_i as source

Node G_i computes the order in which the cube dimensions shall be routed.

Node G_i sends the message to node $G_{(i+1) \bmod (2^k)}$, and marks the appropriate dimension as being routed. Node G_i keeps a copy of the ticket.

Upon receipt of a message and the ticket a node resends the message with updated ticket in the dimension next in the routing order. A node keeps a copy of the message and the updated ticket, and keeps resending the message in dimensions not yet routed, with the ticket properly updated.

Each node that receives a message can determine the next dimension of communication, or if the distribution is terminated, from the ticket it receives, or the copy it keeps. Note that in the mode of operation described above a node only communicates on one port at a time. The control of the algorithm is clearly distributed. Termination is assured. This one-to-many distribution algorithm guarantees that all nodes of a k-cube receive a message precisely once within at most k routing steps. The cube has its origin at the source node. The distribution algorithm reaches a new dimension for each routing step. The source node participates in k distributions for a k-cube, node i+1 in k-1 distributions, etc. Uniqueness is easily established if the source node is node (00....00). Routing from any other source node is identical after relabeling of the dimensions.

The distribution paths formed by the algorithm clearly form a spanning tree rooted at the source node.

Hence, the completeness, uniqueness, and termination of the algorithm also holds for an asynchronous algorithm.

The algorithm is order preserving in the sense that elements distributed by nodes encoding consecutive integers are received in the order distributed. This property is established by noticing that the path i, i+1, i+2, ... reaches into 2 dimensions after 2 steps, 3 dimensions after 3 or 4 steps depending on i, and 4 dimensions after a minimum of 5 steps. Distributions from nodes encoding consecutive integers can be initiated every 2 routing steps.

The behavior of the algorithm for a 4-dimensional cube is shown in Figure 4-7.

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

Figure 4-7: One-to-Many Distribution in a boolean cube, with sources in sequential order

4.3.5. Matrix transpose

The formation of a matrix transpose is a particular permutation of data elements. The transpose of a matrix can be formed recursively as illustrated in Figure 4-8.

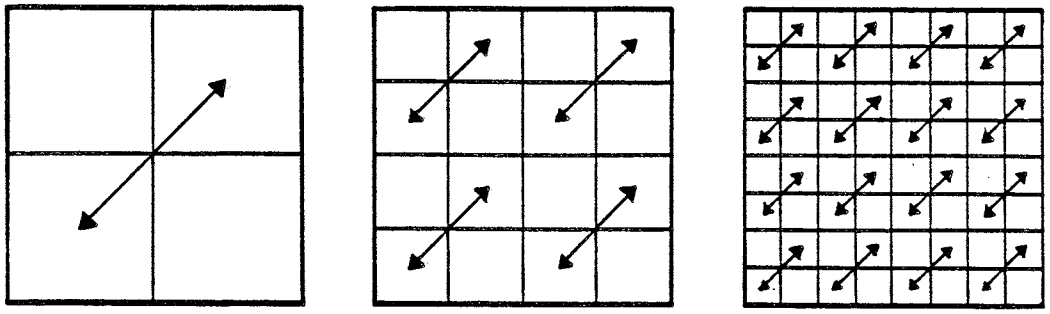


Figure 4-8: Recursive transposition of a matrix

In the first step of the recursive procedure the interchange of data is performed on the highest order bit of the row index and the highest order bit of the column index. In the second step the interchange is performed on the second highest order bit of the row and column indices, for all combinations of the highest order bits (i.e., 4 combinations). The number of index sets that differ in one bit of the row and column indices increases as the procedure progresses towards lower order bits.

For the consecutive storage scheme it is easily seen that with $N=2^n$, the first k steps imply interprocessor communication and the last n-k steps are local to a node in the 2^k by 2^k array. The last n-k steps of the recursive procedure consist of local address changes. We presume here that the transpose is needed with some other data in some computation. Otherwise, the first k steps could also be accomplished without data movement by a suitable change of processor addresses.

With the cyclic storage scheme the situation is reversed. The first n-k steps amount to local address changes, whereas the last k steps require interprocessor communication. After the first n-k steps there are $2^{2(n-k)}$ matrices of size 2^k by 2^k to transpose. All matrices are stored identically.

With cyclic storage and an embedding by binary encoding of the row and column indices two routing steps are required for each element in each of the last k steps of the recursive procedure. The required number of element transfer times is $2^{2(n-k)}+2k-1$ assuming that communication in both directions can take place concurrently on all of the ports of a processor. That at least 2k routing steps are required is seen from the encoding of the elements $(2^{k-1},2^{k-1}-1)$ (100...0 011...1) and $(2^{k-1}-1,2^{k-1})$ (011..1 100..0). With the consecutive storage model and using the apparent granularity of operations the communication time is proportional to $k\times 2^{2(n-k)+1}$, which for k large is considerably higher. The difference between the two expressions is due to pipelining of operations. The paths of elements in forming the transpose intersect only at nodes. This is easily seen for the binary encoding, since element exchanges take place along connections in different dimensions for each step of the procedure.

With a Gray code embedding of the array, successive row and column indices are always located in neighboring nodes of the cube. However, the communication required by the recursive procedure is between nodes storing elements of rows and columns whose binary encoding differs in successively higher or lower order bits. Each such communication requires the communication of elements in two dimensions, since complementing a bit in the binary encoding complements 2 bits in the Gray code encoding (except in complementing the least significant bit).

With each step of the recursive procedure requiring communication in two dimensions in both the part of the address space allocated to row addresses and column addresses, it is not immediately clear that there exists a routing such that element paths intersect only at nodes. However, Routing by Alternating Descending Order Reflections, guarantees this property. The routing proceeds from the highest order bit

of the row indices, i.e., of $G_i(k)$, to the highest order bit of the column indices ,$G_j(k)$, and then further to successively lower order bits of the encoding of i and j in alternating order. This routing scheme implies that matrix elements are subject to reflections around the main diagonal, and the anti-diagonal in alternating order. The behavior of the algorithm is illustrated in Figure 4-9. The numbers on the diagonals indicate the order of the reflection the submatrices are undergoing.

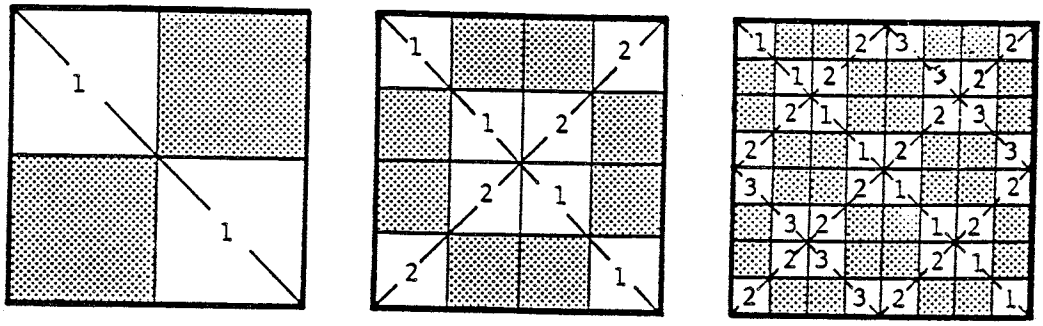


Figure 4-9: Transposing a matrix stored in a binary-reflected Gray code

The maximum path length is clearly 2k. It is also clear that paths intersect only at nodes. Hence, the transpose of a N by N matrix stored cyclicly in a 2-dimensional array that is embedded in a 2k-cube by a binary-reflected Gray code can be performed in $2^{(n-k)}+2k-1$ routing steps.

The application of the alternating descending order reflection algorithm to a 4-cube is illustrated in Figure 4-10. The routing algorithm can be made distributed by the same technique as was used in the one-to-many distribution algorithm.

In forming the transpose of a matrix half of the edges of the cube in a given dimension are used in a given step. Performing the transformation by a 2-dimensional mesh algorithm yields a considerably higher number of routing steps [44], $(2^{2(n-k)}+1)(2^{k-1}-1)/2$. The order of complexity of that algorithm cannot be reduced since N(N-1)/2 elements have to pass through $O(2^k)$ nodes, each of which has 4 ports.

4.4. Linear Algebra Computations

4.4.1. Matrix multiplication

Cannon [13] presents an algorithm for computing the product C of two 2^k by 2^k matrices A and B stored in column major order in a 2-dimensional array of identical size. The algorithm requires a time proportional to 2×2^k. It has a set-up phase followed by a multiplication phase, and is of the SIMD type. The purpose of the set-up phase is to align elements from the two matrices such that all nodes in the

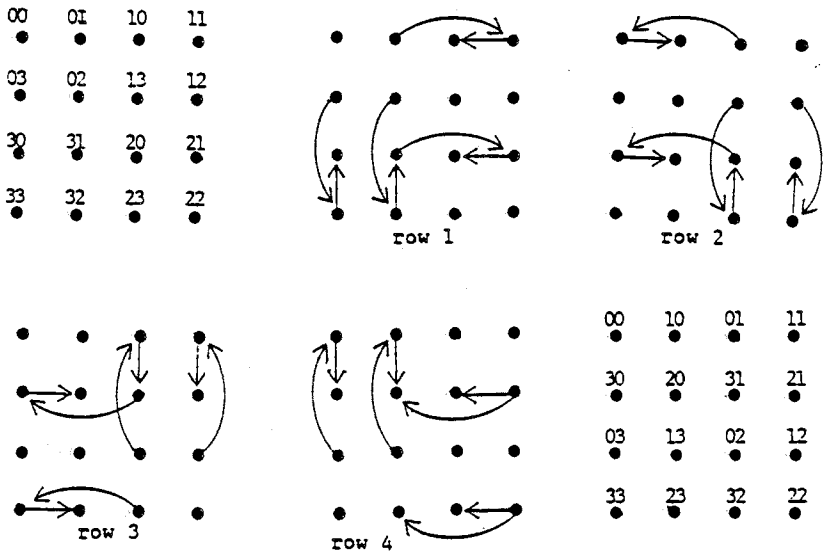


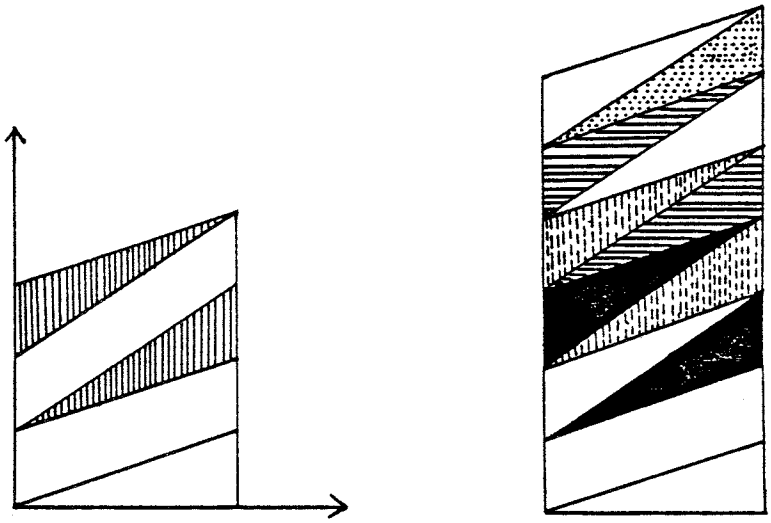
Figure 4-10: Routing paths in transposing a 4 by 4 matrix on a 4-cube

array can perform an inner product computation in every step in the multiplication phase. The alignment is accomplished by i cyclic shifts of row i of A and j cyclic shifts of column j of B. This skewing operation is the same as is seen in many systolic algorithms [57], [47].

The inner products defining the elements of C are accumulated *in-place*. Denote the storage cells for A, B and C by E, F and G. In the set-up phase the shifting yields: $E(i,j) \leftarrow E(i,(i+j) \bmod 2^k)$, $F(i,j) \leftarrow F((i+j) \bmod 2^k,j)$, $G \leftarrow 0$ for $(i,j) \in \{0,1,2,...,2^k-1\} \times \{0,1,2,...,2^k-1\}$. Clearly $E(i,j) \times F(i,j)$ is a valid product for all i and j. In the multiplication phase the following operations are carried out: $G(i,j) \leftarrow G(i,j)+E(i,j) \times F(i,j)$, $E(i,j) \leftarrow E(i,(j+1) \bmod 2^k)$, $F(i,j) \leftarrow F((i+1) \bmod 2^k,j)$, $i,j=\{0,1,2,...,2^k-1\}$. Multiplication of an $M \times R$ matrix by an $N \times M$ matrix can be accomplished in a time of $\max(\lceil N/2^k \rceil, \lceil R/2^k \rceil) \lceil M/2^k \rceil (2^k-1)t_c + \lceil N/2^k \rceil \lceil M/2^k \rceil \lceil R/2^k \rceil ((2^k-1)(t_a+\max(t_a,t_c))+2t_a)$.

A drawback of the algorithm by Cannon is that no computations are being performed during the alignment process. Some elements make almost 2 full revolutions, should only unidirectional communication be allowed. However, one revolution suffices, and algorithms can be devised such that successive matrix multiplications can be initiated every 2^k "cycles". For instance, using the outer product formulation [34] of a matrix product, and passing the columns of A along rows (one element per row) in order of increasing column indices, and rows of B along columns in the direction of increasing row indices. The distribution of columns of A and rows of B can start from the locations where the elements are stored [45]. The distribution can be pipelined, and the initiation of the distribution of the different columns and rows can be spread over time in order that no temporary storage be needed, other than for a pair of elements to be multiplied. With only unidirectional communication, and end-around connections, a total time of $5(2^k-1)$ "cycles" is required for one matrix multiplication. The complexity of the

algorithm may be improved, but with unidirectional data movement pipelining is easy to visualize. The data movement is similar to that of the dense matrix factorization algorithm (without partial pivoting) described later.



Active cycles of row 0 Pipelining of multiple matrix multiplications

Figure 4-11: Pipelined matrix multiplication by the outer product method on a torus

The complexity of the algorithm can be reduced by a term 2^k-k, if the alignment can be accomplished in time k instead of 2^k. For matrices of a size comparable to the number of processors this difference is also significant relative to the total time. Dekel et. al. [17] describes such an algorithm for 2^k by 2^k matrices embedded in a boolean cube of 2^{2k} nodes by a separate binary encoding of row and column indices. The algorithm has a set-up phase in which A and B are arranged such that $E(i,j) \leftarrow A(i,i \oplus j)$ and $F(i,j) \leftarrow B(i \oplus j,j)$. Hence, $E(i,j) \times F(i,j)$ are valid terms for C(i,j) for $(i,j) \in \{0,...,2^k-1\} \times \{0,...,2^k-1\}$. The rearrangement requires exchanges of elements in the dimensions specified by i for A and by j for B. Clearly, the set-up phase requires k steps, and no two elements traverse the same edge in the same direction. The set-up phase for multiple multiplication operations can be pipelined so that the total set-up time for P problems is P+k-1.

In the multiplication phase nodes exchange their content in an order determined by the transition sequence of the bits in a binary-reflected Gray code [17]. It follows that the time for multiplying an M by R matrix B by an N by M matrix A on a boolean 2k-cube, with A, B and $C \leftarrow A \times B+C$, embedded according to a separate binary encoding of row and column indices, modulo $2^k=\sqrt{K}$, is at most $((\lceil N/\sqrt{K}\rceil+\lceil R/\sqrt{K}\rceil)\lceil M/\sqrt{K}\rceil+\log_2 K-1)t_c + \sqrt{K}\lceil N/\sqrt{K}\rceil\lceil R/\sqrt{K}\rceil\lceil M/\sqrt{K}\rceil\max(2t_a,t_c)$ The number of submatrices of size $\sqrt{K}$ by $\sqrt{K}$ is $(\lceil N/\sqrt{K}\rceil+\lceil R/\sqrt{K}\rceil)\lceil M/\sqrt{K}\rceil$. The number of block matrix multiplications is $\lceil N/\sqrt{K}\rceil\lceil R/\sqrt{K}\rceil\lceil M/\sqrt{K}\rceil$. Cannon's matrix multiplication algorithm is devised for SIMD architectures. For mesh or boolean cube configured ensembles of the MIMD type it is possible to

devise algorithms with many different kinds of data flow and a complexity of $(\lceil N/\sqrt{K}\rceil\lceil R/\sqrt{K}\rceil\lceil M/\sqrt{K}\rceil\sqrt{K}\text{-}1)\max(2t_a,t_c)+\alpha d+2t_a$, where d denotes the diameter of the ensemble configuration and $\alpha \leq 4$ [45].

4.5. Multiplication of a full matrix by a triangular matrix

If A is an upper triangular (or lower triangular) N by N matrix, then only half of the arithmetic operations N(N+1)/2 are nontrivial. The alignment and multiplication phases of Cannon's algorithm can be interleaved such that computations start from one corner of the array and progress towards the opposite corner. The total data movement is the same. In this variation of Cannon's algorithm it is convenient to use the notion of *computational windows*. A computational window is defined by the data elements processed concurrently by the ensemble nodes. The computational window during step j for an upper triangular matrix A, $a_{ij}=0$ for $i-j > 0$, is shown in Figure 4-12. The Figure also shows the computational window for step j if A is a strict lower triangular matrix, $a_{ij}=0$, $ij \leq 0$.

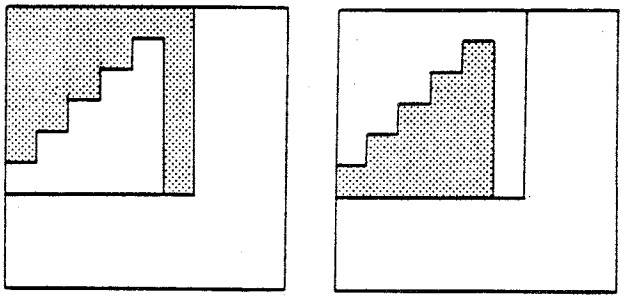


Figure 4-12: The computational window at step j for computing $C \leftarrow A\times B+C$, A upper triangular, or strict lower triangular

From Figure 4-12 it is obvious that the multiplication A×B and D×B, where A is upper triangular and D strict lower triangular of dimension 2^k by 2^k, or A strict upper triangular and D lower triangular can be performed concurrently on a torus of dimension 2^k by 2^k, or a boolean 2k-cube.

4.5.1. Matrix-vector Multiplication

The matrix multiplication algorithms described above can also be used for matrix-vector multiplication, Y=AX. However, the running time is independent of the number of columns of X, and the data movement is larger than necessary [45]. An algorithm that on a boolean cube is of a lower complexity than the algorithm by Cannon (adapted to a boolean cube), or the algorithm by Dekel et. al., for a single vector, or for a matrix X with few columns is obtained by making A stationary, distributing the elements of X to the proper ensemble nodes, and accumulating the partial products over space to yield C in the desired location.

To outline the algorithm assume A is a 2^k by 2^k matrix and x a 2^k vector with components x_i, and that both A and x are stored in row major order. First x_i is rotated i steps in direction of increasing column

index for $i=\{0,1,...,2^k-1\}$, then each x-value is distributed to all nodes in column i, and the products computed. Finally, the products are accumulated. Each of these steps can be carried out in a time proportional to k. With the matrix embedded by separately encoding row and column indices in a Gray code, the shifting is performed in different subcubes, and no communication conflicts occur. The routing of elements for a given shift s can be carried out by comparing the Gray codes of i and i+s and moving towards the desired address by one dimension at a time in any order. The one-to-many distribution algorithm can be used within columns. A similar algorithm can be used for the accumulation of inner products. Complexity estimates for algorithms computing matrix-vector products by accumulating inner products *in-space* are given in [45] for dense matrices and in [46] for banded matrices.

4.5.2. Factorization of dense matrices

The algorithms for Gaussian elimination and Gauss-Jordan elimination described below can be viewed as modifications of systolic algorithms [57], [36]. The modification of the symmetric versions of Gaussian elimination such as Cholesky's, Crout's, and Doolittle's methods can be carried out in a similar way. Systolic algorithms for mesh configured ensembles for Cholesky's method are given in [2], [37], for Given's rotations in [24], [26], [2], [38], and for Householder transformations in [39]. Given's and Householder's methods make use of unitary transformations and are numerically stable.

The factorization of a linear system $Ax=y$ into a lower triangular matrix L and an upper triangular matrix U, is carried out such that the product form of L^{-1} is computed, $L^{-1}=L_{N-1}L_{N-2}...L_1$. $A=LU$ and $Ux=L^{-1}y$. The elements of the factors are stored in the same locations as the elements of the matrix to be factored.

$$L_i = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & l_{ii} & \\ & & & \vdots & \\ & & & l_{N-1i} & 1 \end{pmatrix}$$

The non-trivial elements of a factor become known after the preceeding factors have been applied to A, i.e., l_{ki}, $k=\{i,...,N-1\}$ equals the corresponding elements of $A_i=L_{i-1}A_{i-1}$, $A_0=A$. The application of the factors can be pipelined, as is done in systolic algorithms. In Gauss-Jordan elimination the inverse is also expressed in product form, $A^{-1}=J_{N-1}J_{N-2}....J_0$. The non trivial column of J_i is determined by the corresponding column of $A_i=J_{i-1}A_{i-1}$.

$$A^{-1}=J_{N-1}J_{N-2}\cdots J_0, \qquad J_i=\begin{pmatrix} 1 & & f_{0i} & \\ & \ddots & \vdots & \\ & & 1 & \\ & & f_{ii} & \\ & & \vdots & \ddots \\ & & f_{N-1i} & & 1 \end{pmatrix}$$

We again assume that A is stored in a 2-dimensional array in row major order. We first present algorithms for 2-dimensional arrays, and then describe modifications that can take advantage of the added communication capability of a boolean cube. In the application of L_i to A_i row i (the pivot row) is distributed to rows k, $k > i$, and column i to columns l, $l > i$. In Gauss-Jordan elimination the pivot row is distributed to all other rows. If the matrix is of the same dimension as the array, i.e., there is only one matrix element per node, then Gauss-Jordan elimination can be completed in the same time as Gaussian elimination. An increasing number of processors become idle in Gaussian elimination.

For A large compared to the array only the diagonal blocks are diagonalized, with A being stored cyclicly. The storage of the pivot row and columns [45] and their distribution is illustrated in Figure 4-13. Each application of a factor is similar to performing a column by row product in the outer product matrix multiplication algorithm.

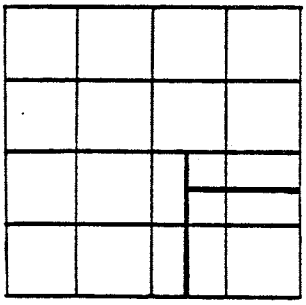


Figure 4-13: Storage and distribution of pivot row and column in dense matrix factorization

For each column elimination operation a number of elements needs to be distributed along rows, and a number along columns. The number of elements distributed along rows equals the number of submatrices on and below the diagonal. The number of elements distributed along columns is equal to the number of blocks on, and to the right of the diagonal, including the right hand sides.

A boolean cube offers a capability of carrying out the distribution of the pivot row to other rows and the pivot column to other columns in less than linear time. For the factorization of a matrix by Gaussian elimination without partial pivoting this capability does not lower the complexity of the elimination, but it does in the event of partial pivoting. However, in forward substitution on multiple right hand sides,

and in Gauss-Jordan elimination the communication capability of the boolean cube can be used to reduce the complexity of the propagation term. The order of the complexity is still linear in the size of the matrix, which is intrinsic to Gaussian elimination without partial pivoting. Faster methods for band matrix problems are described in the next section.

In performing the data distribution in Gaussian elimination, or Gauss-Jordan elimination, without partial pivoting the source nodes are consecutively indexed. A correct result is guaranteed if data arrives in the same order as its distribution is initiated. This condition is sufficient, but not necessary for correctness. The one-to-many distribution algorithm described earlier guarantees the same order of arrival as that of distribution [46]. Note that in Gaussian elimination the distribution for the last several equations only needs to cover successively smaller cubes.

4.5.3. Linear Recurrences

A number of methods for the solution of general linear recurrences $Lz=y$ (y and z are vectors, L a lower triangular matrix) on architectures with global storage have been proposed, and their complexity analyzed assuming zero communication cost. Sameh [85] gives a survey of such algorithms and their properties. We present an algorithm for mesh or boolean cube configured ensembles [44]. It is an adaptation of the binary tree algorithm by Johnsson [40], which in turn is a particular instantiation of the column-sweep algorithm described by Kuck [54]. We assume that the vectors y and z are stored in row major order, and L in column major order (L^T is stored in row major order). In the algorithm outlined below y and z are stationary, L communicated along rows, and partial inner products along columns.

The elements of a column of L are passed along rows of the array. The elements of a column are passed in order of increasing row index. The first element of a column of L is used to compute a new component of z. Subsequent elements of a row of L^T are multiplied by this z component, added to the corresponding partial inner product passed along columns in direction of increasing row indices, and the result passed to the next processor in the same column. The first partial inner product that reaches a processor is used to update the right hand side, before a new z is computed. Hence, a processor in row i when first activated computes z_i, then computes the product $l_{ki}z_i$, adds this product to $\sum_{j=0}^{i-1} l_{kj}z_j$ received from the preceeding row, and outputs the result to the succeeding row.

This algorithm for solving linear recurrences progresses from one row to the next at the rate t_d+2t_a, ignoring communication time (t_d is the time for division of two floating-point numbers). For a 2-dimensional array of 2^k by 2^k processors, the service of a processor is requested for a new row every $(t_d+2t_a)2^k$ units of time. If L is a banded matrix with m nonzero diagonals, then a processor needs a time of $t_d+2(m-1)t_a$ to complete the computations for one column of L. The time to solve the linear recurrence by this algorithm is approximately

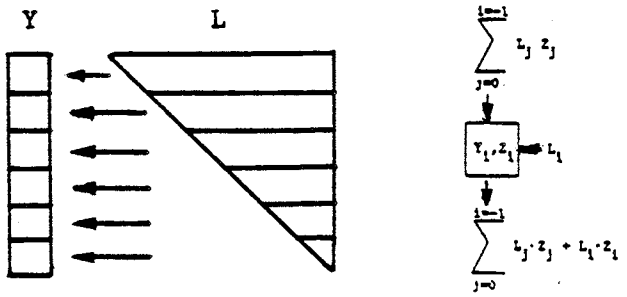


Figure 4-14: Data movement in a linear recurrence algorithm on a torus

$$\lceil (N-m)/2^k \rceil \max(t_d+2(m-1)t_a,(t_c+t_d+2t_a)2^k) + \sum_{j=0}^{\lfloor m/2^k \rfloor} \max(t_d+2(m-j2^k-1)t_a,(t_c+t_d+2t_a)2^k).$$

For banded systems a recurrence solver can also be based on the partitioning method [87]. This approach can further reduce the complexity of solving linear recurrences. The partitioning method is discussed further in the next section.

4.5.4. Banded System Solvers

Tridiagonal systems

Irreducible tridiagonal systems of equations can be solved in $2\log_2 N$ steps using $O(N)$ arithmetic operations by odd-even cyclic reduction [12]. The method has been modified by Hockney [34] to yield a solution in $\log_2 N$ steps, but at the expense of $O(N\log_2 N)$ arithmetic operations. For highly concurrent ensembles it is of interest to find mappings of the computation graph on to the nodes of the ensemble such that the communication complexity is no higher, or at least of the same order as the parallel arithmetic complexity. Binary trees, shuffle-exchange networks, and boolean cubes allow for global communication in a time proportional to k for $K=2^k-1$ and $K=2^k$ processors respectively.

The solution of tridiagonal systems on binary trees is interesting not only for the importance of efficient tridiagonal solvers, and the relative simplicity of constructing large tree ensembles, but also from an algorithm design point of view. First, there exists a mapping of the computation graph for cyclic reduction on N equations on to a binary tree of N nodes such that the communication complexity is $3\log_2 N$. A comparable communication complexity is also obtainable on shuffle-exchange networks and boolean cubes [41]. Second, for $N > K$, the mapping principle providing minimum communication complexity for $N=K$ does not yield the desired communication complexity, but the use of two different embeddings does. Rearrangement from one embedding to the other is made at a certain stage in the algorithm. Third, a poly-algorithmic approach yields a communication complexity of minimum order. The third property is also true for the shuffle-exchange and boolean cube configured ensembles.

The computation graph of cyclic reduction is shown in Figure 4-15. For $N=K$ mapping the equations

on to nodes in the tree in inorder for every level of the computation graph, yields a map with the desired order of complexity. The first reduction step requires a time proportional to k, the second a time proportional to k-1, etc. But, the reduction steps can be pipelined, and the total time is proportional to k [41], [22]. The inorder mapping is shown in Figure 4-16.

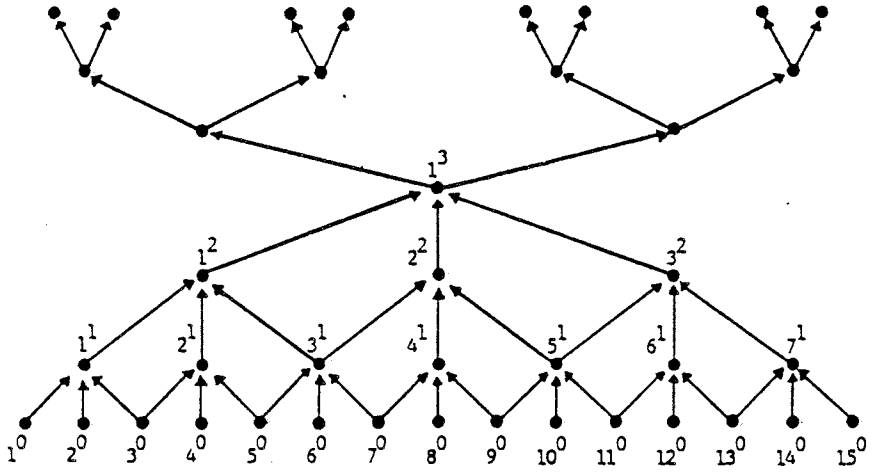


Figure 4-15: The computation graph for odd-even cyclic reduction

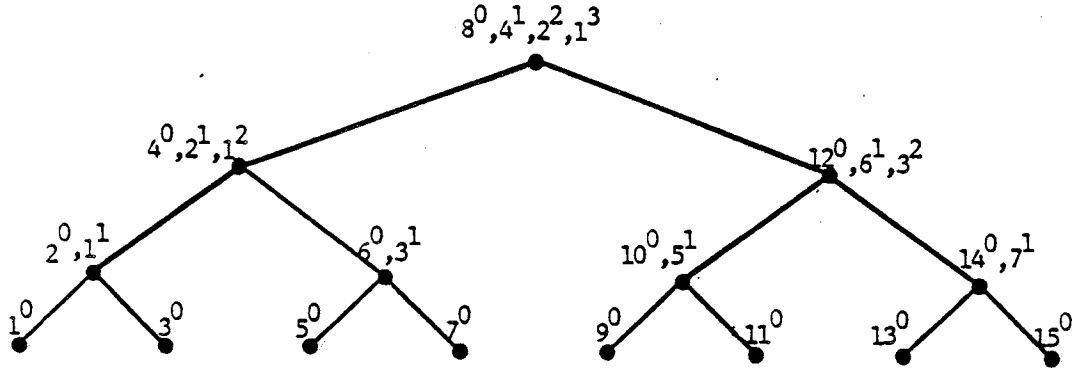


Figure 4-16: Inorder mapping of cyclic reduction on to a binary tree

For $N=2^n-1 \geq K$ it is necessary to map several equations on to a tree node. One natural approach is to first map the computation graph in inorder on to an abstract tree of size N, and then to map this abstract tree on to the ensemble tree. This mapping can be done by folding the tree on to itself, or mapping several adjacent levels of the abstract tree into one level of the processor tree. However, such an approach does not yield an efficient algorithm in the case of cyclic reduction, neither with respect to the balance of computational load, nor with respect to communication.

Another natural approach to the mapping of the computation graph on to the tree is the formation of a quotient graph from the computation graph by combining a number of successively indexed nodes at each level of the computation graph into a node in the quotient graph. This approach is similar to domain decomposition in the solution of partial differential equations. The nodes at each level of the quotient

graph are then mapped on to the processor tree in inorder. The number of quotient nodes at the leaf level of the computation graph with $\lceil N/K \rceil$ equations is $2^{n \bmod k-1}$, which corresponds to a nmodk level binary tree. In the formation of the quotient nodes the computation graph is effectively partitioned into "vertical" slices, with one quotient node per slice and level for n-k levels starting with the leaf level. In another view the same levels have been collapsed into one level. The quotient graph approach provides the best possible computational balance.

A critical observation in finding a communication efficient mapping is that the communication between some pair of partitions alternates in direction for every level (reduction step) of the computation graph. The efficiency of the inorder map relies on the fact that the communication is unidirectional, and can be pipelined. A communication (and arithmetically) efficient map in the case of cyclic reduction on a "large" set of equations on a "small" tree ensemble is obtained by a proximity preserving map of the quotient graph on to the tree, for the first several steps until there is one "active" node per quotient node, followed by an inorder map for the last k reduction steps. The remapping can be done in k-3 routing steps [41]. Figure 4-17 illustrates a proximity preserving embedding of a path in a binary tree. The embedding is a minor variation of the embedding by Rosenberg and Snyder [81].

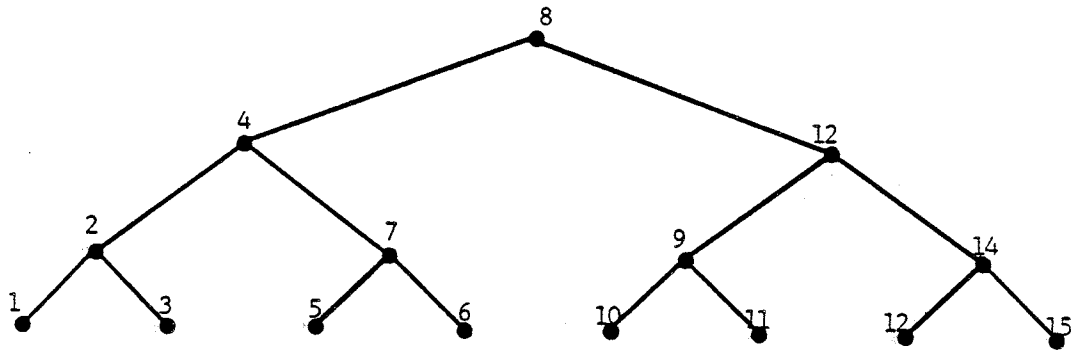


Figure 4-17: A proximity preserving path embedding in a binary tree

Hence, in the case of cyclic reduction on a binary tree ensemble, and $N > K$, the use of two different embeddings yields a communication complexity of the same order as the arithmetic complexity on a parallel processor with unbounded parallelism and shared storage, i.e., $O(\log_2 N)$, whereas for $N=K$ one embedding yields the same result.

However, cyclic reduction with a proximity preserving embedding for the first n-k reduction steps and an inorder embedding during the last k reduction steps does not yield a communication complexity of minimum order, which is $O(\log_2 K)$. A tridiagonal solver having a communication complexity of minimum order, without increased arithmetic complexity, can be obtained by combining Gaussian elimination and cyclic reduction, even though Gaussian elimination essentially is a sequential method. For $N > K$, Gaussian elimination can be used concurrently in all processors, reducing the set of equations

forming a node in the quotient graph to a single node. Fill-in is created, but the arithmetic complexity is approximately the same as for cyclic reduction [111]. In this reduction only one interprocessor communication is needed, compared to $\log_2(N/K)$ for cyclic reduction on a binary tree. The reduced system is of the form shown in Figure 4-18. The reduced system is then solved by cyclic reduction using an inorder map. The total complexity is of order $O(N/K+\log_2 K)$ [41].

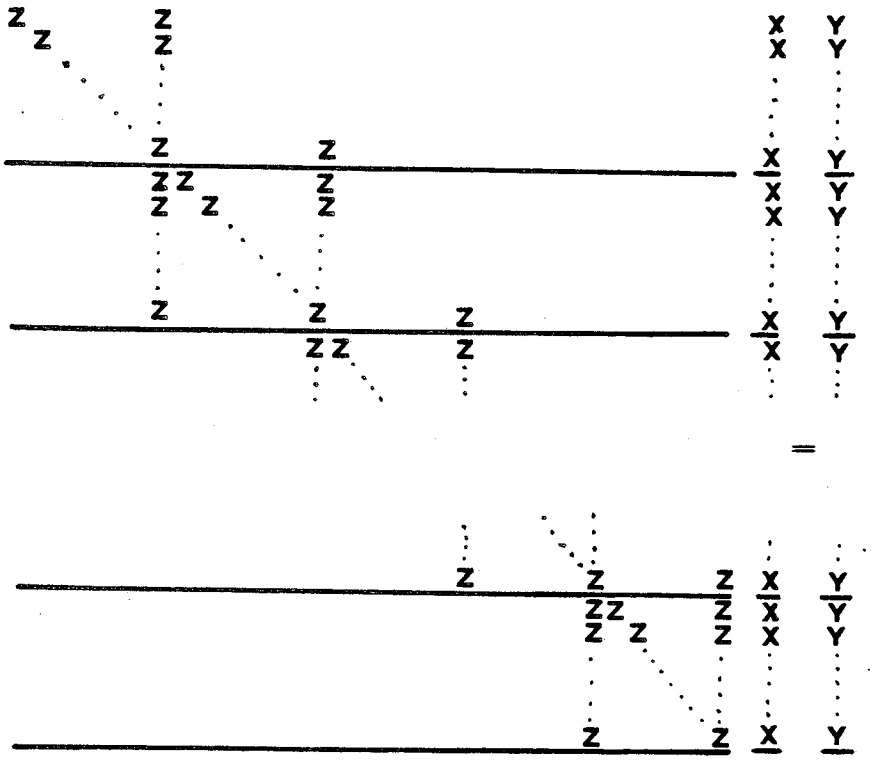


Figure 4-18: Combining Gaussian elimination and cyclic reduction for tridiagonal systems

The combined algorithm is of minimum order of complexity both with respect to communication and arithmetic. The communication time is proportional to the diameter of the ensemble. This minimum order algorithm is of the poly-algorithmic type. However, the pure cyclic reduction algorithm may have advantages in the case of truncated cyclic reduction. But, Reiter and Rodrigue [80] have showed that under certain conditions the fill-in elements decrease in magnitude, and a truncation of the reduction process may be possible for the poly-algorithmic approach as well.. The reduction in computational complexity accomplished by truncating the reduction process is relatively much more significant in a highly concurrent system than in a single processor system. For N=K the running time is proportional to the reduction steps executed, while on a single processor architecture half of the total (untruncated) execution time is spent in the first reduction step, a quarter in the second, etc. The speed-up for cyclic reduction and N=K is $O(N/\log_2 N)$, but approaches $O(N)$ if the reduction process can be terminated after a fixed number of steps, as in strongly diagonally dominant systems.

On a boolean cube a static data structure yields an implementation of cyclic reduction of communication complexity of order $O(\log_2 N)$. Combining Gaussian elimination and cyclic reduction yields a total complexity of minimum order with quotient sets mapped to nodes according to a Gray code. Unlike algorithms such as the FFT or bitonic sort each step of the cyclic reduction algorithm involves 3 nodes of the computation graph. An embedding according to a binary encoding would require k routing operations for the first step of the algorithm, k-1 for the 2nd, k-2 for the 3rd etc. The binary-reflected Gray code also allows for simple, distributed control. Each processor can determine with which neighboring processor to communicate, and what information shall be transmitted/received from its address, and the reduction step currently being executed [41]. Because of the properties of the binary-reflected Gray code each step requires only 2 routing steps. One of these routing steps can be carried out as an exchange operation (but need not be). In such a case successive levels of the computation graph are mapped into subcubes of monotonously decreasing dimensionality. For N=K all processors participate in the first reduction step, about half of which only perform communication. The equations participating in the second reduction step are moved to one half of the cube, and the process is repeated recursively. Figure 4-19 illustrates a few steps in the reduction process for N=K=8.

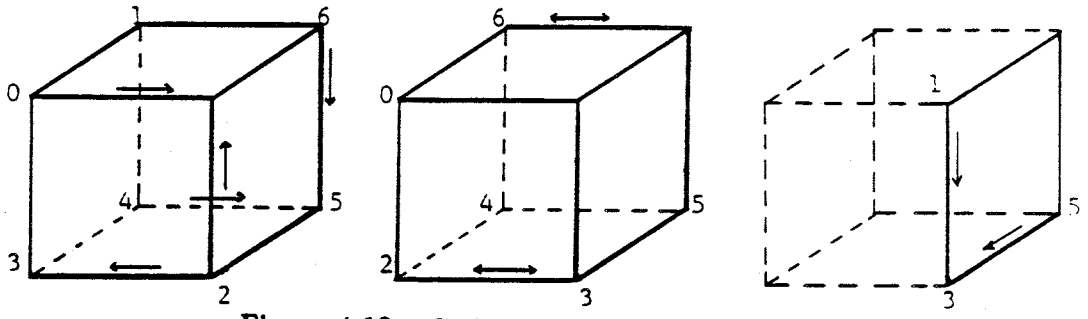


Figure 4-19: Cyclic reduction on a boolean cube

If multiple independent tridiagonal systems are to be solved, then either all problems can be distributed over the entire ensemble, or the ensemble can be logically partitioned such that each problem is solved by a partition. For tridiagonal systems and the solution methods discussed here, it is always advantageous to partition the ensemble, even in the event of negligible communication time [41].

General Banded Systems

The partitioning technique used in the communication efficient tridiagonal system solver can also be applied to banded systems. Sameh and collaborators [87], [59], [20], use the partitioning technique to reduce banded systems of bandwidth 2m+1 to dense, block pentadiagonal, systems of order 2m(P-1) for P partitions. The blocks are of size m by m. The solution of the reduced system by Gaussian elimination on a linear array is considered by Lawrie and Sameh [59], and the solution by block-Jacobi and preconditioned conjugate gradient methods by Dongarra and Sameh [20]. Reiter and Rodrigue [80], and Johnsson [43] analyze a modification that reduces the banded system to a dense, block tridiagonal system

of order mP. Reiter and Rodrigue give conditions under which diagonal dominance is preserved during the Gaussian elimination part of the algorithm. Johnsson shows that the arithmetic complexity in deriving the tridiagonal system is approximately 1/3 of that required in deriving the pentadiagonal system, and analyze the complexity of solving the reduced system by Gaussian elimination and block cyclic reduction on linear arrays, binary trees, shuffle-exchange networks and boolean cubes.

The optimum number of partitions P_{opt} depends on m, N, and the ratio of the communication and computation bandwidths. P_{opt} for Gaussian elimination on a linear array is of order $O(\sqrt{N/m})$ and the corresponding complexity is of order $O(m^2\sqrt{Nm})$. Block cyclic reduction yields a lower complexity under a variety of conditions, even on a linear array. The value of P_{opt} falls in the range $O(\sqrt{N}) \leq P_{opt} \leq O(N/m)$, and the corresponding complexity is in the range $O(m^2\sqrt{N}+m^3\log_2 N)$ to $O(m^3+m^3\log_2(N/m))$ [43]. For binary trees, shuffle-exchange networks, and boolean cubes P_{opt} is of order $O(N/m)$ and the corresponding complexity of order $O(m^3+m^3\log_2(N/m))$. For small matrix bandwidths this algorithm yields good speed-up, but as the bandwidth increases the speed-up becomes low.

The above results apply under the assumption that there is one processor per partition. The number of partitions is constrained to be at most N/m. However, it is possible to exploit concurrency in the operations also within the partitions, which for m of order $O(N)$ is the main source of concurrency. For instance, one can use $K_c \leq m^2$ processors configured as a mesh or a boolean cube during the elimination of the elements in one column, as in systolic algorithms [57] [36], but use the dual formulation in which the factors are computed *in-place*. The speed-up is of order K_c. The computations proceed in two phases: factorization with forward substitution, and backsubstitution. The factorization can proceed from the first to the last column, or from both ends concurrently. In the latter case an m by m dense system of equations must be solved for the "middle" equations before backsubstitution takes place. A dense m by m system is also solved in the 1-way elimination scheme (the last m equations). The backsubstitution consists of solving a linear, banded recurrence. The technique discussed previously can be used. Figure 4-20 illustrates an intermediate state of the factorization process.

The one-to many distribution algorithm can be used to reduce the propagation time for a boolean cube, and multiple right hand sides. The propagation time then becomes of lower order even in the case of $K_c=m^2$, and $N \approx m$. The complexity of solving banded systems by this approach is $O(m^2N/K_c)$ $(P=1)$ [46]. For symmetric matrices storage as well as time can be saved using a parallel version of Cholesky's method [2], [37].

The two methods can be combined such that K_c processors are used for each partition. Such a set of processors is referred to as a *cluster*. With P clusters of K_c processors each, intracluster connections in the form of a 2-dimensional mesh (or torus) or boolean cube, and intercluster connections forming a

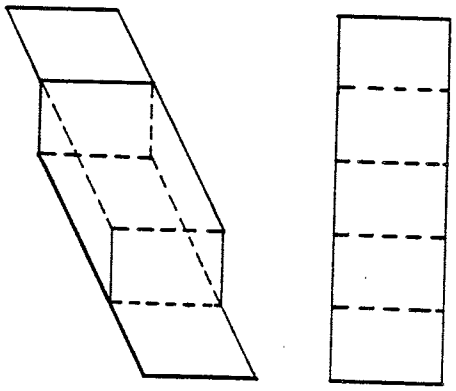


Figure 4-20: Band matrix factorization on a 2-dimensional array or boolean cube

binary tree, shuffle network, or boolean cube, the minimum time complexity is of order $O(m+m\log_2(N/m))$, and P_{opt} of order $O(N/m)$, and $K_{c_{opt}}$ of order $O(m^2)$ [46]. Note that for a boolean cube a subcube can be considered as a cluster.

The combined algorithm degenerates to the simple band matrix algorithm proceeding along the band from one corner to the other for m=N-1. For m=1 it degenerates to the tridiagonal solver described previously.

4.5.5. Fast Poisson Solvers

Fast Poisson solvers combine Fast Fourier Transforms (FFT) with tridiagonal system solvers, and block cyclic reduction to achieve a minimum arithmetic complexity of order $O(N\log_2\log_2 N)$ for an N by N grid [32], [33], [102]. Stone [101] observed that the FFT can be carried out in $\log_2 N$ steps on an N-node shuffle-exchange network. The modification of Stone's algorithm for a boolean cube is straightforward. Shuffle operations become unnecessary. The butterfly operations are simply carried out on elements residing in processors adjacent in different dimensions. This property holds for decimation-in-time (DIT) as well as decimation-in-frequency (DIF) FFT.

With multiple elements per processor the initial (and transformed) sequence can be stored in either consecutive or cyclic storage order. In either case, and depending on whether a DIT or a DIF FFT is used the first, or the last, $\log_2(N/K)$ butterfly operations are local to a node. The arithmetic complexity is of order $O((N/K)\log_2 N)$ and the communication complexity is of order $O((N/K)\log_2 K)$. The speed-up is proportional to K. FFT algorithms for linear arrays are given in a number of references [77], [47], [48]. An early description of a FFT on a 2-dimensional array is given by Stevens [100]. An analysis of the area-time aspects of the FFT on a variety of ensemble configurations is given in [105].

The solution of Poisson's problem on a rectangle can be obtained by a 2-dimensional FFT, by a number of 1-dimensional FFTs that decouple the equations into a set of independent tridiagonal systems, or by a

combination of block cyclic reduction, FFT, and tridiagonal system solvers, and the so called FACR method [32], [33], [12], [11], [102], [103], [104]. By exploiting symmetries and using real transforms the number of arithmetic operations per point is less than $2.5\log_2 N$ in the FFT computation. Using Gaussian elimination with precomputed factors [103] the number of arithmetic operations per point for the tridiagonal solvers is 4, or approximately 4 if advantage is taken of the fast convergence of the elements of the factors [88]. Hence, the arithmetic operations count is less for solving the tridiagonal systems by Gaussian elimination than by FFT, solution of a diagonal system, and inverse FFT (IFFT). A cyclic reduction algorithm could be used, but the operations count per point with precomputed factors is 6.

In a highly concurrent system the differences in complexity between the FFT and tridiagonal system solvers are much smaller. To amplify this issue consider the case with N^2 nodes in an ensemble configured as a boolean cube. The solution of Poisson's equation either by a 2-dimensional FFT, or by a combination of 1-dimensional FFT's and tridiagonal system solvers based on cyclic reduction, then requires a time of order $O(\log_2 N)$, including communication. For this extreme case Gaussian elimination is not of interest with respect to computational complexity since it is inherently sequential. The number of routing steps in the boolean cube is $4\log_2 N$ for cyclic reduction, the same as for a FFT followed by an IFFT. With a butterfly operation carried out entirely in one node, solution of the tridiagonal systems by FFT, solution of a diagonal system, followed by IFFT would require $12\log_2 N$ arithmetic operations in sequence, assuming no concurrency within an ensemble node. Sharing the computations among a pair of nodes would bring the operations count for the FFT tridiagonal solver down to $7\log_2 N$. The cyclic reduction based solver requires $6\log_2 N$ operations with precomputed coefficients. A direct 2-dimensional FFT could potentially yield a comparable or slightly lower operations count. In the FFT approach all nodes are used in all steps, but in the cyclic reduction tridiagonal system solver the number of active nodes decreases. In the FFT tridiagonal system solver $O(N\log_2 N)$ operations are performed, in the cyclic reduction tridiagonal system solver $O(N)$ operations. Furthermore, most of the tridiagonal systems are sufficiently diagonally dominant that the reduction process can be truncated. This does not reduce the total solution time in this extreme case, but a significant fraction of the processors in the ensemble can be used for other tasks.

Sameh [86] presents a method for the solution of the 2-dimensional Poisson equation on a ring of processors, and for the solution of the 3-dimensional problem on a cylinder of processors. In the 2-dimensional case FFT's are performed on data local to a processor, and the tridiagonal systems solved by a modification of the partitioning method [87]. The modification is made to take advantage of the Toeplitz form of the tridiagonal matrices. The reduced systems are solved by pipelined Gaussian elimination within the ring. In the 3-dimensional case 1-dimensional FFT's are performed, first local to a processor, then a new set of 1-dimensional FFT's are performed within a ring, resulting in N^2 independent tridiagonal systems, with each tridiagonal system spread across the rings. The tridiagonal systems are solved by Gaussian elimination.

Whether Gaussian elimination or cyclic reduction is preferable with respect to computational complexity for the solution of the tridiagonal systems depends on the ensemble topology, K, N, and α, the ratio of the arithmetic bandwidth to the communication bandwidth. The following complexity estimates can be derived: Gaussian elimination $4N^2/K+2N+\alpha\sqrt{K}$ on an ensemble configured as a $\sqrt{K}$ by $\sqrt{K}$ mesh, partitioning based entirely on Gaussian elimination $9N^2/K+4N/\sqrt{K}+(2+\alpha)\sqrt{K}$, and partitioning with cyclic reduction for the reduced system and an ensemble configured as a boolean cube $9N^2/K+N/\sqrt{K}(3+2\alpha)\log_2 K$. Precomputed coefficients are assumed for these estimates. Cyclic reduction for the reduced system becomes competitive for K approaching N on a boolean cube configured ensemble. On a linear array the logarithmic term premultiplied by α is replaced by a term linear in $\sqrt{K}$, as for Gaussian elimination. Which method is preferable on a linear array is critically dependent upon α, K, and N. An accurate comparison should also account for the fact that the truncation process can be terminated for a large fraction of the systems. With respect to performance the benefit of truncating the reduction process is particularly large on linear arrays, since the largest communication expense occurs in the last few reduction steps using an *in-place* algorithm [41].

4.5.6. Iterative methods

Conjugate Gadient Methods

The conjugate gradient method [29] is a direct method for the solution of linear systems of equations. However, it is often used as an iterative method, and combined with preconditioning is an effective iterative technique, in particular for sparse systems. The conjugate gradient method solves a linear system of N equations in N steps. Each step requires $O(NZ)$ arithmetic operations for a system $Ax=y$ in which A has NZ non-zero elements. Hence, the arithmetic complexity is of the same order as for elimination methods, Given's rotations, and Householder transformations if the matrix A is dense. But, because of fill-in in those methods, the conjugate gradient method often yields a lower complexity for sparse systems, in particular if acceptable accuracy in the solution is obtained in less than N steps (possibly much fewer steps).

The minimum time per iteration is $O(\log_2 N)$ because of global communication in each step. In each iteration an inner product including the entire state is computed, and used (distributed to all processors) in the computation of the new state. Pipelining of successive steps is not possible. The minimum parallel arithmetic complexity of the conjugate gradient method is $O(N\log_2 N)$, the same order as that of Householder's method. Preconditioning that would allow the iterative process to be terminated in less than $N/\log_2 N$ steps could possibly yield a lower complexity, but the complexity of each step has to be included. With the original system matrix used as a preconditioner one iteration suffices, but the original system of equations has to be solved in that step.

So far very few studies have been carried out for parallel versions of the conjugate gradient method.

Adams [1] has investigated the convergence of various preconditioners, and in particular their feasibility with respect to implementations on the Finite Element Machine. Saad and Sameh have investigated the conjugate gradient method on multiprocessors with shared global storage [84], and linear arrays [83]. The implementation of the preconditioned conjugate gradient method with various preconditioners has also been investigated by Kamath and Sameh [50]. They consider the solution of 2-dimensional elliptic partial differential equations on a ring of processors, and the solution of the 3-dimensional problem on a torus. The adaptation of the conjugate gradient method to binary tree architectures is described by Johnsson [40]. The effect of preconditioning on the computational complexity is analyzed. Van Rosendale [110] has proposed a modification of the inner product computation in which it is computed recursively. Only local computations are carried out in each step. However, global communication is still required in each step.

Asynchronous methods

In the classical iterative methods a number of matrix vector products are computed. Each such product requires global communication. In a highly concurrent system this global communication will limit the speed-up, unless several iteration steps can be pipelined. A large fraction of the processors in the ensemble are idle. So called asynchronous iterative methods, or chaotic relaxation, attempt to fully exploit the concurrency in multiprocessor systems by not enforcing global synchronization between each step of the iterative process. Chazan and Miranker [15] give necessary and sufficient conditions for convergence of chaotic relaxation applied to the solution of linear systems of equations. The results are extended by Miranker [68]. Baudet [5] gives necessary and sufficient conditions for convergence for nonlinear problems, and history dependent iterations, and some bounds on the efficiency, as well as some experimental results obtained on the C.mmp [113]. Recently, asynchronous iteration has also been studied by Lubachevsky and Mitra [66].

5. Summary

The power of an ensemble configuration can be measured in several different ways. One way is to measure the time required to perform arbitrary permutations. Such permutations of N elements on a binary tree of N nodes may require a time proportional to $N+O(\log_2 N)$. Arbitrary permutations can be performed on a 2-dimensional mesh in time $6\sqrt{N}$ [106], on the shuffle-exchange network in time $\log_2 N(\log_2 N-1)$, and on the boolean cube in time $1/2\log_2 N(\log_2 N+1)$ using the deterministic algorithm of Batcher, or with high probability in $c\log_2 N$ time for c a small integer using a randomized algorithm. The cube connected cycles network has the same capability of performing arbitrary permutations.

Another way to measure the power of an ensemble configuration is to determine to what extent one configuration can emulate another at an increase in running time by at most a constant factor, for certain

classes of algorithms. Of the networks discussed here the shuffle-exchange, boolean cube and cube connected cycles networks are the most powerful for so called normal algorithms [107]. The boolean cube and cube connected cycles network may yield an improved performance by a constant factor over the shuffle-exchange network for normal algorithms. The tree network is significantly less powerful in that the running time for many algorithms is higher by more than a constant factor.

The diameter of a configuration gives a lower bound for the time required for a given operation. Whether the diameter appears as an additive term or multiplicative factor in treating "large" problems on "small" ensembles depends on how communication paths with distinct origins and destinations intersect. We illustrated this point by forming a matrix transpose on a 2-dimensional mesh with end-around connections and on a boolean cube. The transpose of a $\sqrt{K}$ by $\sqrt{K}$ matrix can be formed in $(1/2\sqrt{K}\text{-}1)$ routing steps on the mesh configured ensemble, and $\log_2 K$ routing steps on the boolean cube. This difference becomes significant first for fairly large K. However, the transpose of a N by N matrix on a $\sqrt{K}$ by $\sqrt{K}$ mesh requires a time of at least order $O(N^2/\sqrt{K})$, and can be performed in $1/2(\lceil N/\sqrt{K}\rceil^2+1)(1/2\sqrt{K}\text{-}1)$ routing steps [44]. On the boolean cube the transpose can be performed in $(\lceil N/\sqrt{K}\rceil^2+\log_2 K\text{-}1)$ routing steps, an improvement by a factor of approximately $\sqrt{K}/4$ over the mesh.

The finite communication capability of ensemble configurations affects the performance adversely, sometimes significantly. The time for arithmetic operations decreases, but the time for communication may increase with the ensemble size. For most ensemble configurations and computations there exists an optimum size of the ensemble beyond which the performance decreases. For instance, in the case of the solution of tridiagonal systems of equations by combining Gaussian elimination and cyclic reduction the optimum sizes and minimum solution times are as follows for a few configurations: linear array $K_{opt}\approx\beta\sqrt{N/\alpha}$ and $T_{min}\approx\gamma\sqrt{N}$, 2-dimensional mesh $K_{opt}\approx\beta(N/\alpha)^{2/3}$ and $T_{min}\approx\gamma N^{1/3}$, binary tree, shuffle-exchange, and boolean cube networks $K_{opt}\approx\beta N/(1+\alpha)$ and $T_{min}\approx\gamma\log_2 N$, where α is the ratio between the arithmetic and communication bandwidths [41]. For band matrix solvers based on the partitioning technique the optimum number of processors configured as a linear array is of order $O(\sqrt{N/m})$ for matrices of bandwidth 2m+1, and $O(N/m)$ for binary tree, shuffle-exchange and boolean cube networks. The corresponding solution times are of order $O(m^2\sqrt{Nm})$ and $O(m^3+m^3\log_2(N/m))$, respectively [45]. For ensembles configured as boolean cubes, band matrix solvers of complexity $O(m+m\log_2(N/m))$ can be devised [41].

With insufficient number of interconnections, or with inappropriate topology, different embedding strategies may have to be applied not only for different problems, but also for different phases of a given algorithm. This need was exemplified by cyclic reduction on a binary tree. On a boolean cube a single embedding strategy yields a computational complexity of minimum order. Of relevance for many computations is the embedding of 1-dimensional, or multidimensional arrays. Linear arrays can be

embedded in binary trees preserving proximity, but for d-dimensional arrays embedded in the leaves of the tree the average distance between nodes adjacent in the mesh is $(4\text{-}2^{-\lfloor \log_2 n \rfloor})d$ when embedded in the tree. The maximum distance is of order $O(d\log_2 n)$. Both 1-dimensional and multidimensional arrays can be embedded in boolean cubes preserving proximity. If the number of elements in each dimension is slightly less than or equal to a power of 2, then this embedding is also efficient in terms of processor utilization. The impact of a given embedding on performance is in some instances determined by the average distance between array nodes, whereas in others it is determined by the maximum distance.

Ensemble architecture algorithms can be obtained by first generating a computation graph from a description of the computation in a conventional mathematical notation, and then mapping this graph on to the ensemble. This mapping process has many characteristics in common with the mapping carried out in finding efficient systolic algorithms. But, there are also several aspects of the mapping of computation graphs on to ensemble architectures that do not require attention in the systolic case. One similarity is the need to treat temporal as well as spatial aspects of computations, with a nonuniform access time to different parts of the storage. Preserving locality is also important in both architectures. However, the embedding of the computation graph in an ensemble architecture often has to satisfy additional criteria compared to what is required in the systolic case in order to yield maximum processor utilization, or minimum solution time. The need for different embedding strategies during different phases of the execution of an algorithm may depend on the size of the problem relative to the size of the ensemble, as in the case of cyclic reduction.

With the additional sequencing of operations caused by mapping several nodes of a given level of the computation graph on to the same ensemble node, instead of distinct nodes as in the systolic case, independence of communication paths becomes an issue. If communication paths with distinct origins and destinations intersect at nodes only, and the processor can support concurrent communication on all its ports, then communication actions can be pipelined to a maximum extent. In effect, the ensemble is configured optimally for the desired operation. This issue was illustrated by performing a matrix transpose on a boolean cube.

Another difference compared to algorithms of extremely fine grain is that whereas in such a case an efficient parallel algorithm may be ideal, in particular if it can be mapped on to an ensemble with only local communications without loss of efficiency, this is not necessarily true on an ensemble architecture. More operations are carried out in sequence, and the sequential operations count may be higher for an algorithm of minimum parallel complexity than for a sequential algorithm of minimum complexity. For instance, bitonic sort requires $O(N\log_2^2 N)$ operations compared to $O(N\log_2 N)$ operations for a good sequential sort. In the case of tridiagonal system solvers, cyclic reduction requires approximately twice the number of operations needed by Gaussian elimination. A combination of algorithms may yield a

lower complexity than any single algorithm. In some instances, such as in the solution of tridiagonal systems by elimination methods, it may be possible to obtain the combined algorithm by algorithm transformation techniques. Elementary rules of algebra may be used to reduce the number of arithmetic operations carried out sequentially, as in mapping the computation of the Discrete Fourier transform on to a linear array. The result is an FFT algorithm with defined data and control structure [49]. However, the most interesting aspects of algorithm transformation techniques is that a user may not have to worry about all the minute variations of algorithms and architectural details, and that for ensemble architectures more efficient algorithms may be discovered.

In the architectural model used here it is essential that the control of execution is distributed, in order to prevent bottlenecks and avoid sources of limited scalability. All of the algorithms presented here have local control, including the routing algorithms. The architecture allows each node to execute a substantially different piece of code. However, in most of the concurrent algorithms we know there is a high degree of regularity, not only in the communication pattern, but also in the instruction streams being executed. Typically there are 3 - 4 different pieces of code. In algorithms for 2-dimensional meshes boundary nodes often perform somewhat different tasks, like computing rotation factors in the case of Given's method. In binary tree algorithms the root, the leaves, and the intermediate level nodes often have their unique pieces of code [8]. This characteristic also simplifies the problem of downloading code if the ensemble serves as an attached processor. The code can be replicated within the ensemble [63], and thereby considerably reduce the potential bottleneck caused by external input/output operations.

A large class of problems not discussed here is that of computations with data dependent control flow. For data independent computations it is possible in principle to map the computations on to the nodes in the multiprocessor system at "compile time". For simple problems mappings that are optimal with respect to some criteria, like time, can be found at a small or moderate expense. However, finding optimal mappings for most problems is, in general, an NP-complete problem. For data dependent computations good strategies for run time mappings of computations on to processors are needed. To avoid potential bottlenecks it is desirable that load balancing use only local information, and that global information is gathered through a sequence of local communications.

Acknowledgement

The author is grateful to Andrea Pappas for preparing all the illustrations, and to Abhiram Ranade for his careful reading of the manuscript. The author is also indebted to the Office of Naval Research for providing financial support under contract N00014-84-K-0043.

References

1. Adams L. Iterative Algorithms for Large Sparse Linear Systems on Parallel Computers. 166027, NASA Langley Research Center, 1982.

2. Ahmed H.M., Delosme J.-M., Morf M. "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing". *Computer 15* (January 1982), 65-82.

3. Aleliunas R. Randomized Parallel Computation. ACM Symposium on Principles of Distributed Computing, 1982, pp. 60-72.

4. Batcher K.E. Sorting Networks and Their Applications. Spring Joint Computer Conference, 1968, pp. 307-314.

5. Baudet G.M. "Asynchronous Iterative Methods for Multiprocessors". *J. ACM 25*, 2 (1978), 226-244.

6. Bhatt S.N., Leighton F.T. "A Framework for Solving VLSI Graph Layout Problems". *J. of Computer and System Sciences 28* (1984), 300-343.

7. Brent R.P., Kung H.T. "On the Area of Binary Tree Layouts". *Information Processing Letters 11*, 1 (1980), 44-46.

8. Browning S.A. The Tree Machine: A Highly Concurrent Computing Environment. 1980:TR:3760, Computer Science, California Institute of Technology, 1980.

9. Browning S.A., Seitz C.L. Communication in a tree machine. Proceedings, Second Conference on Very Large Scale Integration, 1981, pp. 509-526.

10. Budnik P., Kuck D.J. "The Organisation and use of Parallel Memories". *IEEE Trans. Computer C-20* (December 1971), 1566-1569.

11. Buzbee B.L. "A Fast Poisson Solver Amenable to Parallel Computation". *IEEE Trans. Computers C-22* (1973), 793-796.

12. Buzbee,B.L., Golub, G.H., Nielson, C.W. "On Direct Methods for Solving Poissons's Equations". *SIAM J. Numer Anal 7*, 4 (December 1970), 627-656.

13. Cannon L.E. *A Cellular Computer to Implement the Kalman Filter Algorithm*. Ph.D. Th., Montana State University, 1969.

14. Capello P.R., Steiglitz K. Unifying VLSI Array Design with Linear Transformations of Space-Time. TRCS83-03, UC Santa Barbara, Dept of Computer Science, May, 1982.

15. Chazan D., Miranker W. "Chaotic Relaxation". *Linear Algebra and its Applications 2* (1969), 199-222.

16. Chen M.C. Synthesizing Systolic Designs. RR YALEU/DCS/RR-374, Dept. of Computer Science, Yale University, March, 1985.

17. Dekel E., Nassimi D., Sahni S. "Parallel Matrix and Graph Algorithms". *SIAM J. Computing 10*, 4 (November 1981), 657-673.

18. Delosme J.-M., Ipsen I.C.P. An Illustration of a Methodology for the Construction of Efficient Systolic Architecture in VLSI. 2nd InternationalSymposium on VLSI Technology, Systems, And Applications, 1985.

19. DeMillo R.A., Eisenstat S.C., Lipton R.J. "Preserving Average Proximity in Arrays". *Communicationsof the ACM 21* (March 1978), 228-231.

20. Dongarra J.J., Sameh A.H. On Some Parallel Banded System Solvers. ANL/MCS-TM-27, Argone National Laboratories, Mathematics and Computer Science Division, 1984.

21. Flynn M.J. "Very High-Speed Computing Systems". *Proc. IEEE 54*, 12 (1966), 1901-1909.

22. Gannon D., Van Rosendale J. On the Impact of Communication Complexity in the Design of Parallel Numerical Algorithms. ICASE Report 84-41, NASA Langley Research Center, August, 1984.

23. Gentleman W.M. "Some Complexity Results for Matrix Computations on Parallel Processors". *J. ACM 25*, 1 (January 1978), 112-115.

24. Gentleman M.W., Kung H.T. Matrix Triangularization by Systolic Arrays. Real-Time Signal Processing IV, Proc. of SPIE, 1981, pp. 19-26.

25. Gottlieb A., Grishman R., Kruskal C.P., McAuliffe K.P., Rudolph L., Snir M. "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer". *IEEE Trans. Computers C-32*, 2 (1983), 175-189.

26. Heller D. E., Ipsen I.C.F. Systolic Networks for Orthogonal Equivalence Transformations and Their Applications. Proceedings, Advanced Research in VLSI, 1982, pp. 113-122.

27. Hennessey J.L., Jouppi N, Baskett F., Gill J. MIPS: A VLSI Processor Architecture. VLSI Systems and Computations, 1981, pp. 337-346.

28. Hennessey J.L., Jouppi N, Przybylski S., Rowen C. Design of a High Performance VLSI Processor. Proc. of the Third Caltech Conference on VLSI, 1983, pp. 33-54.

29. Hestenes M.R., Stiefel E. "Methods of Conjugate Gradient for Solution of Linear Systems". *J. Res. Nat. Bur. Standards 49* (1952), 409-436.

30. Hillis W.D. "The Connection Machine: A Computer Architecture Based on Cellular Automata". *Physica 10D* (1984), 213-228.

31. Hirschberg D.S. "Fast Parallel Sorting Algorithms". *Communications of the ACM 21*, 8 (1978), 657-661.

32. Hockney R.W. "A Fast Direct Solution of Poisson's Equation using Fourier Analysis". *J. ACM 12* (1965), 95-113.

33. Hockney R.W. "The Potential Calculation and Some Applications". *Methods Comput. Phys. 9* (1970), 135-211.

34. Hockney R.W., Jesshope C.R.. *Parallel Computers*. Adam Hilger, 1981.

35. Hwang K. (Ed.). *Supercomputers: Design and Applications*. IEEE Computer Society, 1984.

36. Johnsson, S.L. Computational Arrays for Band Matrix Equations. 4287:TR:81, Computer Science, California Institute of Technology, May, 1981.

37. Johnsson S.L. VLSI Algorithms for Doolittle's, Crout's and Cholesky's Methods. International Conference on Circuits and Computers 1982, ICCC82, September, 1982, pp. 372-377.

38. Johnsson S. L. Pipelined Linear Equation Solvers and VLSI. Microelectronics '82, May, 1982, pp. 42-46.

39. Johnsson S.L. A Computational Array for the QR-method. Proc., Conf. on Advanced Research in VLSI, January, 1982, pp. 123-129.

40. Johnsson, S.L. Highly Concurrent Algorithms for Solving Linear Systems of Equations. Elliptic Problem Solving II, 1983.

41. Johnsson S.L. Odd-Even Cyclic Reduction on Ensemble Architectures and the Solution Tridiagonal Systems of Equations. YALE/CSD/RR-339, Department of Computer Science, Yale University, October, 1984.

42. Johnsson, S.L. Combining parallel and Sequential Sorting on a Boolean n-cube. International Conference on Parallel Processing, 1984, pp. 444-448.

43. Johnsson S.L. Solving Narrow Banded Systems on Ensemble Architectures. YALEU/CSD/RR-343, Dept. of Computer Science, Yale University, November, 1984.

44. Johnsson S.L. Communication Efficient Matrix Operations on a Torus and a Boolean Cube. YALEU/CSD/RR-361, Dept. of Computer Science, Yale University, January, 1985.

45. Johnsson S.L. Dense Matrix Operations on a Torus and a Boolean Cube. The National Computer Conference, July, 1985.

46. Johnsson S.L. Fast Banded Systems Solvers for Ensemble Architectures. YALEU/CSD/RR-379, Department of Computer Science, Yale University, March, 1985.

47. Johnsson, S.L., Weiser, U., Cohen, D., and Davis, A. Towards a Formal Treatment of VLSI Arrays. Proceedings of the Second Caltech Conference on VLSI, Caltech Computer Science Department, January, 1981, pp. 375 - 398.

48. Johnsson, S. L., Cohen, D. An Algebraic Description of Array Implementations of FFT Algorithms. 20th Allerton Conference on Communication, Control, and Computing, 1982.

49. Johnsson S.L., Cohen D. *Mathematical Approach to Computational Networks for the Discrete Fourier Transform.* Department of Computer Science, Yale University.

50. Kamath C., Sameh A.H. The Preconditioned Conjugate Gradient Method on a Multiprocessor. ANL/MCS-TM-28, Argonne National Laboratories, Mathematics and Computer Science Division, 1984.

51. Kershaw D. Solution of Single Tridiagonal Linear Systems and the Vectorization of the ICCG Algorithm on the CRAY-1. In G. Rodrigue, Ed., *Parallel Computations*, Academic Press, 1982, pp. 85-92.

52. Knuth D.E.. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, 1973.

53. Kogge P.M., Stone H.S. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". *IEEE Trans. Computers C-22*, 8 (1973), 786-792.

54. Kuck D.J. "A Survey of Parallel Machine Organization and Programming". *ACM Computing Surveys 9*, 1 (1977), 29-59.

55. Kuck D.J., Lawrie D.H., Cytron R., Sameh A., Gajski D.D. The Architecture and the Programming of the Cedar System. Laboratory for Advanced Supercompuetrs, Dept. of Computer Science, University of Illinois, August, 1983.

56. Kumar M., Hirschberg D.S. "An Efficient Implementation of Batcher's Odd-Even Merge Algorithm and Its Application in Parallel Sorting Schemes". *IEEE Trans. on Computers C-32*, 3 (1983), 254-264.

57. Kung H.T., Leiserson C.L. Algorithms for VLSI Processor Arrays. In *Introduction to VLSI Systems*, Addison-Wesley, 1980, pp. 271-292.

58. Lawrie D.H. "Access and Alignment of data in an Array Pocessor". *IEEE Trans. Computer C-24* (December 1975), 1145-1155.

59. Lawrie D.H.,Sameh A.H. "The Computational Complexity and Communication Complexity of a Parallel Banded System Solver". *ACM Trans. Math. Software 10*, 2 (June 1984), 185-195.

60. Lawrie D.H., Vora C.R. "The Prime Memory System for Array Access". *IEEE Trans. Computer C-31* (May 1982), 1435-442.

61. Leighton F.T.. *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks*. MIT Press, 1983.

62. Leiserson, C.E.. *Area-Efficient VLSI Computation*. MIT Press, 1982.

63. Li, P., Johnsson, L. The Tree Machine: An evaluation of program loading strategies. 1983 International Conference on Parallel Processing, August, 1983, pp. 202 - 205.

64. Li G.-J., Wah B.W. "The Design of Optimal Systolic Arrays". *IEEE Trans. Computers C-34* (1985), 66-77.

65. Lisper B. Description and Synthesis of Systolic Arrays. TRITA-NA-8318, The Royal Institute of Technology, Dept. of Numerical Analysis and Computing Sciences, 1983.

66. Lubachevsky B., Mitra D. A Chaotic, Asynchronous Algorithm for Computing the Fixed Point of a Nonnegative Matrix of Unit Spectral Radius. AT&T Bell Laboratories, 1984.

67. Mead C.A., Rem M. "Cost and Performance of of VLSI Computing Structures". *IEEE J. of Solid State Circuits SC-14*, 2 (April 1979), 455-462.

68. Miranker W. "Hierarchical Relaxation". *Computing 23* (1979), 267-285.

69. Miranker W.L., Winkler A. "Spacetime Representations of Computational Structures". *Computing 32*, 2 (1984), 93-114.

70. Moldovan D.I. "On the Design of Algorithms for VLSI Systolic Arrays". *Proc. IEEE 71*, 1 (1983), 113-120.

71. Nassimi, D., Sahni S. "Bitonic Sort on a Mesh-Connected Parallel Computer". *IEEE Trans. on Computers C-27*, 1 (1979), 2-7.

72. Paterson M.S., Ruzzo W.L., Snyder L. Bounds on Minimax Edge Length for Complete Binary Trees. Proc. of the 13th Annual Symposium on the Theory of Computing, 1981, pp. 293-299.

73. Patterson D.A. "Reduced Instruction Set Computers". *Communications of the ACM 28*, 1 (1985), 8-21.

74. Patterson D.A., Sequin C.H. "A VLSI RISC". *Computer 15*, 9 (1982), 8-22.

75. Preparata F.P., Vuillemin J.E. The Cube Connected Cycles: A Versatile Network for Parallel Computation. Proc. Twentieth Annual IEEE Symposium on Foundations of Computer Science, 1979, pp. 140-147.

76. Quinton P. Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations. Proc. 11th Annual Symposium on Computer Architecture, 1984, pp. 208-214.

77. Rabiner L.R., Gold B.. *Theory and Application of Digital Signal Processing*. Prentice-Hall, 1975.

78. Ranade A. Interconnection Networks and Parallel Memory Organization for Array Processing. 1985 International Conference on Parallel Processing, 1985.

79. Reingold E.M., Nievergelt J., Deo N.. *Combinatorial Algorithms*. Prentice Hall, 1977.

80. Reiter E., Rodrigue G. An Incomplete Cholesky Factorization By a Matrix Partitioning Algorithm. Elliptic Problem Solvers II, 1983, pp. 161-174.

81. Rosenberg A.L., Snyder L. "Bounds on the Costs of Data Encodings". *Mathematical Systems Theory 12* (1978), 9-39.

82. Ruzzo W.L., Snyder L. Minimum Edge Length Planar Embeddings of Trees. VLSI Systems and Computations, 1981, pp. 119-123.

83. Saad Y. Practical use of Polynomial Preconditionings for the Conjugate Gradient Method. YALEU/DCS/RR-282, Dept. of Computer Science, 1983.

84. Saad Y., Sameh A.H. Iterative Methods for the Solution of Elliptic Differential Equations on Multiprocessors. Proc. of the CONPAR 81 Conference, 1981, pp. 395-411.

85. Sameh A.H. Numerical Parallel Algorithms - A Survey. High Speed Computer and Algorithm Organization, 1977, pp. 207-228.

86. Sameh A.H. A Fast Poisson Solver for Multiprocessors. Elliptic Problem Solvers II, 1984, pp. 175-186.

87. Sameh A.H., Kuck D.J. "On Stable Parallel Linear System Solvers". *JACM 25*, 1 (January 1978), 81-91.

88. O'Donnel S.T., Geiger P, Schultz M.H. Solving the Poisson Equation on the FPS-164. YALEU/DCS/RR-293, Research Center for Scientific Computing, Dept. of Computer Science, Yale University, November, 1983.

89. Schwartz J.T. "Ultracomputers". *ACM Trans. on Programming Languages and Systems 2* (1980), 484-521.

90. Seitz C.L. Ensemble Architectures for VLSI -- A Survey and Taxonomy. 1982 Conf on Advanced Research in VLSI, January, 1982, pp. 130 - 135.

91. Seitz C.L. "The Cosmic Cube". *Communications of the ACM 28*, 1 (1985), 22-33.

92. Lutz C., Rabin S., Seitz C.L., Speck D. Design of the Mosaic Element. Proceedings, Conf. on Advanced research in VLSI, 1984, pp. 1-10.

93. Sejnowski M.C., Upchurch E.T., Kapur R.N., Charlu D.P.S., Lipovski G.J. An Overview of the Texas Reconfigurable Array Computer. Proceedings, National Computer Conference, 1980, pp. 631-641.

94. Sekanina,M. "On an ordering of the set of Vertices of a Connected Graph". *Publ. of the Faculty of Science of the University of Brno* , 412 (1960), 137-142.

95. Shaw D. The NON-VON Supercomputer. Dept. of Computer Science, Columbia University, August, 1982.

96. Shaw D. SIMD and MSIMD Variants of the NON-VON Supercomputer. Dept. of Computer Science, Columbia University, 1984.

97. Siegel H.J. "Interconnection Networks for SIMD Machines". *Computer 12* (1979), 32-48.

98. Smith B.J. Architecture and Applications of the HEP Multiprocessor Computer System. Real-Time Signal Processing IV, Proc of SPIE, 1981, pp. 241-248.

99. Snyder L. "Introduction to the Configurable Highly Parallel Computer". *Computer 15*, 1 (1982), 47-56.

100. Stevens J. A Fast Fourier Transform Subroutine for the Illiac IV. Center for Advanced Computation, Univ. of Illinois, 1971.

101. Stone, H.S. "Parallel Processing with the Perfect Shuffle". *IEEE Trans. Computers C-20* (1971), 153-161.

102. Swartztrauber P.N. "The Methods of Cyclic Reduction, Fourier Analysis, and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle". *SIAM Review 19* (1977), 490-501.

103. Temperton C. "Direct Methods for the Solution of the Discrete Poisson Equation: Some Comparisons". *J. of Computational Physics 31* (1979), 1-20.

104. Temperton C. "On the FACR(l) Algorithm for the Discrete Poisson Equation". *J. of Computational Physics 34* (1980), 314-329.

105. Thompson C.D. Fourier Transforms in VLSI. UCB/ERL/ M80/51, Electronic Research Laboratory, UC Berkeley, 1980.

106. Thompson C.D., Kung H.T. "Sorting on a Mesh-connected Parallel Computer". *CACM 20*, 4 (1977), 263-271.

107. Ullman J.D.. *Computational Aspects of VLSI*. Computer Sciences Press, 1984.

108. Upfal E. Efficient Schemes for Parallel Computation. ACM Symposium on Principles of Distributed Computing, 1982, pp. 55-59.

109. Valiant L., Brebner G.J. Universal Schemes for Parallel Communication. Proc. of the 13th ACM Symposium on the Theory of Computation, 1981, pp. 263-277.

110. Van Rosendale J. Minimizing Inner Product Dependencies In Conjugate Gradient Iteration. Proc. of the 1983 International Conference on Parallel Processing, 1983, pp. 44-46.

111. Wang H.H. "A Parallel Method for Tridiagonal Equations". *ACM Trans. Math. Software 7*, 2 (June 1981), 170-183.

112. Wu C.-L., Feng T.-Y. (Ed.). *Interrconnection Networks for Parallel and Distributed Computing*. IEEE Computer Society, 1984. Tutorial.

113. Wulf W.A., Bell C.G. C.mmp - A Multi-Mini-Processor. AFIPS 72 FJCC, 1972, pp. 765-777.