Yale University
Department of Computer Science

Ensemble Architectures and Their Algorithms: An Overview

S. Lennart Johnsson

YALEU/DCS/TR-580
November 1987

This work has been supported in part by the Office of Naval Research under Contracts N00014-84-K-0043 and N00014-86-K-0564. Approved for public release: distribution is unlimited.

ENSEMBLE ARCHITECTURES AND THEIR ALGORITHMS: AN OVERVIEW [1,2]

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

Abstract

During recent years, the number of commercially available parallel computer architectures has increased dramatically. The number of processors in these systems varies, from a few processors up to as many as $64k$ processors for the Connection Machine. In this paper, we discuss some of the technology issues that are the underlying driving force and focus on a particular class of parallel computer architectures. This class is often called Ensemble Architectures, and they 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 communication 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

Advances in device technology have thus far been the primary factors in the evolution of high performance computing. Switching times have decreased from 1 μs for vacuum tubes to around $0.1 - 0.05ns$ for MOS technologies, and an order of magnitude less for bipolar technologies. Clock rates have increased from below 1 MHz to $10 - 40$ MHz in MOS technologies, and 250 MHz for the CRAY-2 (bipolar). While switching speeds have increased by four to five orders of magnitude (six for the CRAY-2), clock rates have increased by only two to three orders of magnitude. The instruction issue rate has increased similarly to the clock rates, but the amount of work carried out per instruction varies significantly. The rate at which floating-point operations can be performed has increased from $0.1 - 1$ kflops (floating-point operations per second) to $10 - 20$ Mflops for MOS technologies, and 250 Mflops per floating-point unit in a CRAY-2.

[1] To appear in Numerical Algorithms for Parallel Computer Architectures, Springer Verlag

[2] Presented at the Institute for Mathematics and its Applications, the University of Minnesota, Minneapolis, November 3 - 7, 1986.

	Chip $25mm^2$ $100M\lambda^2$	Chip $50mm^2$ $200M\lambda^2$	4" Wafer (50%) $166G\lambda^2$	Clock
Dynamic RAM	1 Mbit	2 Mbit	160 Mbit	
Static RAM	256 Kbit	512 Kbit	40 Mbit	
16-bit proc.	40	80	6400	54 MHz
32-bit proc.	8	16	1280	36 MHz

Table 1: Chip and wafer level integration, 1μ feature size

Of the improvement of five to six orders of magnitude in floating-point capability at most three orders of magnitude are attributed to technological and low level design improvements. Dedicated hardware for floating-point operations, extensive use of pipelining, and enhanced algorithms contribute to the remainder. In current high performance systems, a floating-point operation can be initiated every clock cycle. Silicon technologies are 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 five to ten fold increase in switching speeds.

Further dramatic increases in performance must derive from architectural innovations that exploit concurrency in computations. The critical path in most designs is determined by wire delays rather

One cannot expect to reduce the number of switching delays per clock cycle by more than a factor of 5 by comparing common microprocessor designs with a highly optimized design such as the CRAY. Reducing the clock cycle through architectural means, i.e., decreasing the number of switching delays per clock cycle, gets increasingly difficult because wire delays are ever more significant as feature sizes are reduced. Wire delays do not scale well if the geometric aspect ratios and the electric field in the gate region remain constant. While switching delays may be reduced in proportion to the scaling, wire delays may either decrease slowly, remain constant, or even increase depending on what factors (capacitive or resistive) govern the delay.

Replication of locally interconnected parts offers the highest promise for high performance architectures. Intermingling storage and processing elements reduces the average area per processing element, and increases the clock rate in synchronous (non-pipelined) designs. The increased ratio of processing capability per unit of storage, and the increased clock rate, both contribute to increasing the maximum size of the state that can change in a single clock cycle, i.e., the rate of computation.

As feature sizes are reduced, the amount of storage and logic that fits on a single chip or wafer becomes impressive. Tables 1 and 2 contain predictions for 1μ and 0.25μ feature sizes, respectively. The basis for the storage predictions are that a 1-bit dynamic RAM cell requires an area of $100\lambda^2$ and a 1-bit static RAM cell requires an area of $400\lambda^2$. Processor estimates are based on the Caltech Mosaic 16-bit processor, which is about $2.5M\lambda^2$, excluding the pad frame,[92], and the RISC and MIPS 32-bit processors, which are about $12M\lambda^2$, excluding pad frame [34,35,71].

	Chip $25mm^2$ $1600M\lambda^2$	Chip $50mm^2$ $3200M\lambda^2$	4" Wafer (50%) $256G\lambda^2$	Clock
Dynamic RAM	16 Mbit	32 Mbit	2660 Mbit	
Static RAM	4 Mbit	8 Mbit	640 Mbit	
16-bit proc.	640	1280	102400	216 MHz
32-bit proc.	128	256	20480	144 MHz

Table 2: Chip and wafer level integration, 0.25μ feature size

32-bit RISC II processors	160 Mbyte 10240 processors 370 GIPS
16-bit MOSAIC processors	160 Mbyte 51200 processors 55 GFLOPS 32-bit add 12 GFLOPS 32-bit mult
Cosmic Cube nodes ($140M\lambda^2$) (Intel 8086, 8087, 128kbyte)	256 Mbyte 2000 processors 2 GFLOPS ("measured")

Table 3: Wafer level integration, same area for processors and storage

At the 0.25μ feature size level, a large number of processors fit on a single die. The estimates in the tables are highly simplistic and ignore the area required for interprocessor communication as well as processor-to-storage communication. communication in the form of a one- or two-dimensional array of processors with local storage, the wiring area should not substantially alter the estimates. With other interprocessor and/or processor-to-storage communication networks, substantial area may be required for wiring. We obtain the figures in Table 3 by assuming bit-serial communication and simple switching elements for an Ω-network, or a Boolean cube, and the design of [73] as a base case we obtain the figures in Table 3. The predictions for the Cosmic Cube are based on an estimated processor area of $140M\lambda^2$ [118], and measured performance on existing hardware.

In any fabrication process it is expected that some of the processing cells will be defective. In a two-dimensional array of cells on a wafer in which bad cells are arbitrarily distributed, it may still be possible to use the wafer by configuring wires around the defective cells, for example, by laser-restructuring techniques. It is desirable to design wafers so that live cells can be configured in the desired pattern by "threading around" the dead cells. Independently, Leiserson and Leighton, [85] and Greene and El Gamal[32] have investigated the problem of configuring one- and two-dimensional arrays of processors on a faulty wafer. They show that if the faulty nodes are randomly, and independently distributed, then the live nodes may be

connected into a smaller two-dimensional mesh with expected maximum edge length $O(logN)$ and a channel of width $O(1)$, where N is the number of cells in the array. Simple regular structures are not costly to assemble under the presence of faults.

1.1 Ensemble Architectures

Ensemble architectures [117] represent a low cost alternative to future high performance systems. High nominal performance at a low cost is obtained by composing systems out of a large number of parts. These parts are mass produced in state-of-the-art technology. The storage may be entirely distributed among the processors, or part of the storage may be subdivided into storage modules, each of which forms a node in a network. In the generic architecture, some of the nodes in the network represent processors with storage. Others may represent storage alone. Network topology in an ensemble architecture is sparse and regular. Control and data are distributed. The notion of ensemble architectures is not new. PEPE [47,48] is an early example of an ensemble architecture of medium granularity; and cellular automata can be viewed as ensemble architectures of fine granularity.

High performance requires a high rate of operand consumption and generation. With only a few operations per operand high storage bandwidth is required. High storage bandwidth is achieved in ensemble architectures through highly partitioned storage. Associative memory is one extreme instance of this philosophy in that each storage cell is equipped with some processing logic. Systolic architectures and the Connection Machine [37,38] are close to this extreme in that there are only a few registers, or a limited amount of storage, per processing node. The storage bandwidth of the Connection Machine model CM-I is 32 $Gbytes/sec$ at 4 MHz. Model CM-II offers a peak bandwidth in excess of 50 $Gbytes/sec$. Intermediate levels of storage bandwidth are obtained by a larger granularity of computations, as in the Cosmic Cube. Partitioning of storage to yield storage bandwidth compatible to processor capacity is used in high performance architectures such as the CRAY and CYBER series of computers, and was used already in early computer designs [76]. The partitioning of the storage is much less than in, for instance, the Connection Machine, and so is the storage bandwidth, which is 4 $Gbytes/sec$ for the CRAY-2.

There are many considerations in choosing interconnection networks, processor designs, and programming models for ensemble architectures. Manufacturability and scalability with respect to performance and reduced feature sizes of the technology are assured by interconnecting the processing elements sparsely and regularly. The interconnection network chosen represents a tradeoff between communication bandwidth, fault-tolerance, and design and manufacturing considerations. The global architecture (SIMD or MIMD), and the granularity of nodes and their architectural features must be chosen so that high real performance can be achieved by minimizing communication and computation time.

1.1.1 Interconnection Networks

The choice of interconnection network determines the rate at which processors can communicate. While high bandwidth is desirable from the point of view of an algorithm designer, it may be undesirable, and in fact infeasible, from the point of view of the hardware designer. High degree

of interconnect adversely affects scalability, area and volume requirements, and clock rate. One of the most pressing problems in assembling large systems across many chips is caused by severe pin restrictions — while the number of components per chip is expected to grow by up to two orders of magnitude, the chip sizes are not expected to increase much. With pin sizes limited by mechanical considerations, the bandwidth at the boundary of a chip will only increase if the rate at which off-chip wires are driven is increased. Unless many communication channels are multiplexed per pin, thereby making the system clock longer, high fanout systems simply cannot be built.

There is an intimate relationship between processor design and the choice of interconnection network. As the capacity of local storage grows, so does the interprocessor distance, and the distance to the furthest location of local storage. With increased local storage it becomes necessary to structure storage [93] to minimize access delay. In the capacitive model for wire delays, the access time can be reduced to order $logM$ and in the resistive model to order $\sqrt{M}$ for a storage of size M. Assume for the moment that the processors with local storage are interconnected as a one- or two-dimensional array. Then, the interprocessor distance grows as $\sqrt{M}$, and the minimum time to drive the interprocessor connection increases in proportion to $logM$ for the capacitive model and $\sqrt{M}$ for the resistive model. There is no qualitative difference in the relative growths in local access time and interprocessor communication time. Small local storage allows for short wires between processors, and potentially high clock rates. For one- or two-dimensional arrays, it is desirable to keep the local storage small. Local storage can increase performance, if arithmetic is parallel and communication serial, because with local storage several such references are typically made for each remote reference.

Many interconnection networks with a small diameter, such as the shuffle-exchange, butterfly, cube connected cycles and Boolean cube, require long interconnections when layed out in a two- or three-dimensional space. Interprocessor communication will be slower than references to local storage, even if the whole network were to fit on a single wafer in submicron technology. It is important that such systems are self-timed in order for computations to make use of the higher bandwidth between a processor and its local storage, than the bandwidth for remote references. Note, that the access time for different remote references may differ. The fact that some of the networks with a high wiring area do not make effective use of the area (silicon), may indeed cause networks like a mesh to offer a higher total bandwidth than, for instance, a Boolean cube [105].

Network costs

Configuring processors as linear arrays and complete 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 only grows at the rate of the clock frequency, regardless of the number of processors per chip and the size of the machine being built [86]. The tree has the advantage over the linear array in that its diameter (the distance between the processors that are furthest apart) is $2(log_2 N - 1)$ compared to N (or $\frac{1}{2}N$ for an array with end-around connections). The diameter of the network topology defines a lower bound for the speed of computation [29]. Global communication can be accomplished faster in a complete binary tree than in a linear array. The required area for the complete binary tree is of order $O(N)$ [93], if the nodes can be placed arbitrarily in the

Configuration	Nodes	Diam	Fan-out	Edges
Linear Array	2^k	2^{k-1}	2	$2^k - 1$
2-d mesh	2^k	$2(2^{k/2} - 1)$	4	$2(2^k - 2^{k/2})$
2-d mesh of trees	$3 \cdot 2^k - 2 \cdot 2^{\frac{k}{2}}$	$2k$	6	$5 \cdot 2^k - 4 \cdot 2^{k/2})$
Tree of meshes	$(k+1)2^k$	$8 \cdot 2^{\frac{k}{2}}$	4	$2(2^k(4k-1) + \sqrt{2^{k+1}})$
Binary tree	$2^k - 1$	$2(k-1)$	3(1)	$2^k - 2$
Boolean cube	2^k	k	k	$k \cdot 2^{k-1}$
CCC	$k2^k$	$2k-1$	3	$3k \cdot 2^{k-1}$
Shuffle-exchange	2^k	$2k-1$	≤ 3	$\approx 1.5 \cdot 2^k$

Table 4: Topological properties of some common networks

plane, in which case the maximum wire length is $\frac{1}{logN}\sqrt{N}$ [99,9]. Placing all the leaf nodes of the complete binary tree along the boundary, yields an area requirement of order $O(Nlog_2N)$ [11], and a maximum wire length of order $O(\frac{1}{logN}N)$.

Linear arrays and complete binary trees have a small bandwidth and present communication bottlenecks for many important computations. The two-dimensional mesh and mesh of trees [84] offers higher bandwidth and is preferable for many matrix computations. The first two networks can be realized in small area on a wafer ($O(N)$ for the N node mesh and $O(N \log^2 N)$ for the N node mesh of trees) with wire lengths $O(1)$ for the mesh and $O(\frac{1}{\log \log N}\sqrt{N} \log N)$ for the mesh of trees. The advantage of the mesh of trees over the mesh is its logarithmic diameter ($2 \log N$ compared with $2\sqrt{N}$ for the mesh).

More sophisticated networks, such as the shuffle-exchange, Ω-networks, cube connected cycles [101], and Boolean n-cube have also been proposed because of their ability to efficiently emulate other important networks, or for high total bandwidths. They have been studied extensively in the literature. All of these networks require a layout area almost quadratic in the number of nodes, and wire lengths that grow almost linearly with the number of nodes. Correspondingly, the cost per communication is extremely high and the clock rates are decreased. Currently, a 64 input one bit wide Ω-network with simple switching elements, or 2 32-input Batcher sorting networks, fits on a single chip [73].

Large systems must be partitioned across many chips and boards. Not all the networks mentioned above are easily partitioned under the existing or predicted pin constraints. Tables 4 and 5 summarize the above discussion. The number of off-chip channels are stated in terms of the number of processors per chip, M.

Nodes in a network may either be switching elements or complete processors. The two alternatives yield architectures with slightly different properties. Multistage shuffle-exchange networks, Ω-networks, banyan networks, and butterfly networks are all closely related interconnection networks in which internal nodes typically are switching elements, possibly with a queueing capacity. These networks are used in architectures such as the HEP [10], the Ultracomputer [116,31], the RP3 [100], TRAC [119], CEDAR [78], and the BBN Butterfly [20]. In

Configuration	Nodes	Edge len.	Area	Pin Count
Linear Array	2^k	$O(1)$	$O(2^k)$	2
2-d mesh	2^k	$O(1)$	$O(2^k)$	$4\sqrt{M}$
2-d mesh of trees	$K = 3 \cdot 2^k - 2 \cdot 2^{\frac{k}{2}}$	$O(\sqrt{K} log K / log log K)$	$O(K log^2 K)$	$\approx \frac{6}{\sqrt{3}}\sqrt{M}$
Tree of meshes	$K = (k+1)2^k$	$O(k + log k)$	$O(K log K)$	$2^{\lfloor \frac{m}{2} \rfloor}$
Binary tree	$2^k - 1$	$O(2^{k/2}/k)$	$O(2^k) - O(2^k \cdot k)$	4
Boolean cube	2^k	$O(2^{\frac{k}{2}})$	$O(2^{2k})$	$M(k - log M)$
CCC	$k2^k$	$O(2^{\frac{k}{2}})$	$O(k^2 2^{2k})$	$M - \frac{M}{k} log_2 \frac{M}{k}$
Shuffle-exchange	2^k	$O(2^k/k)$	$O(2^{2k}/k^2)$	

Table 5: Layout properties of some common networks

some designs, processors with local storage are at both ends of the network. In other designs, the processors have a negligible amount of storage and storage units, and processors are at opposite sides of the network. At others again processors with a measurable amount of storage are at one side of the network and a "shared" storage at the other side. The CHiP architecture [121] represents an "intermediate" form of architecture in that it uses a switch network for communication between processors with local storage. However, the switch network is novel and any path between a restricted pair of processors goes through a fixed number of switches, like 2 or 4. The switch network yields a capability to reconfigure the ensemble into a large variety of common configurations. Recently, Leiserson has proposed the Fat-tree network [87] as a universal hardware-efficient network that also uses switches, but with rebroadcasting instead of queues at switching nodes.

Network capabilities

An attractive feature of a network is the ease and efficiency with which it can simulate other networks. If the simulation does not require much overhead, and if the processors of the network being simulated can be mapped automatically onto the existing network, then any program written for the first network can be automatically compiled to run efficiently on the second. The problem of finding a communication efficient algorithm for a specific network can then be formulated as a problem of embedding one graph, the *guest* graph corresponding to the communication needs of the algorithm in the graph describing the network, the *host* graph. Typically, edges in the guest graph are mapped onto paths in the host, and the host may have a larger set of nodes than the guest. The *dilation* of an edge is equal to the length of the path it is mapped to in the host graph, and the *expansion* is the ratio of the number of nodes in the host and guest graphs. The dilation of edges can cause a corresponding decrease in throughput (the time between successive computations of a given kind), or just an increase in latency (additional time for completion of the first computation in a set).

For elementary algorithms, common data structures and communication patterns are one- to four-dimensional meshes, complete or arbitrary binary trees, the FFT butterfly network, and the data manipulator network. As may be expected, hosts with a low connectivity and small bisection width, such as one-dimensional arrays and trees, are not efficient universal networks. Two-dimensional arrays of processors are better hosts. Thompson gives an embedding of the

FFT butterfly network in a two-dimensional mesh such that $logN$ nodes of the butterfly are mapped to one node in the mesh. The maximum edge dilation is $\sqrt{N}$, and up to $logN$ butterfly edges are mapped onto one edge of the mesh.

The Boolean cube is an exceptionally versatile host graph. Multi-dimensional arrays can be embedded with dilation 1 and expansion 1 in the Boolean cube, if the number of grid points in each dimension of the array is a power of 2. It is also well known that the FFT butterfly network can be embedded in the Boolean cube with $logN$ butterfly nodes per cube node, such that the dilation is 1 and there is a one-to-one correspondence between edges in the FFT network and the cube. A static embedding allows a normal [129] algorithm (one that proceeds from input to output without reversing direction) to execute in the same number of steps on the cube as on the FFT butterfly network. However, the throughput of the cube is lower by a factor of $logN$, since a cube node simulates $logN$ FFT network nodes. Similarly, a dynamic embedding of the FFT-butterfly network in a shuffle-exchange network yields a slowdown by a factor of 2 for normal algorithms. The throughput is degraded by a factor of $2logN$. Similar results hold for bitonic sorting networks, being recursively composed FFT networks. Recently, a number of results have been obtained on the embedding of complete binary trees, multiple complete binary trees, multiple binomial trees, balanced spanning trees, and arbitrary trees for the cube [132,8,7,50,68]. It follows from [8], that the Boolean cube can efficiently simulate the mesh of trees network. By construction of the tree of meshes network it is not difficult to see that the Boolean cube can embed this network with dilation 1. The universality of the Boolean cube follows from the fast implementation of sorting algorithms, and the randomized routing schemes of Valiant [131].

Applications rarely consist of a single type of computation. Each component of the set of "elementary" computations defining an application may have different ideal data structures. Indeed, it may even be the case that different data structures are ideal for different phases of an "elementary" algorithm, since the communication pattern may not be uniform throughout the execution of the algorithm [58]. The need for data reallocations in order to minimize the complexity of an algorithm decreases with the ability of the network to efficiently support different access patterns to a data structure. The Boolean cube can emulate, with low overhead, many of the prototypical graphs that either represent particular data structures or data dependency graphs for standard algorithms, or networks that have been proposed as VLSI computing structures. A static embedding of a data structure can support many types of access schemes without communication penalty, reducing the need for data reallocations.

For arbitrary computations one measure of the utility of a network is its total bandwidth, which is proportional to the total number of edges. Hence, the bandwidth of a pipelined Ω-network, (for example, the NYU Ultracomputer [31]), is the same as the bandwidth of a Boolean cube. However, there is a latency in the switch that grows logarithmically with the number of processing elements. In the Boolean cube, the interprocessor communication time is nonuniform. The minimum interprocessor communication time amounts to one routing. The maximum number of routing steps is log_2N. Interprocessor communication in an Ω-network includes $2log_2N$ links. If the communication width corresponds to a word, then this latency may be a significant fraction of the communication time. With bit-serial communication the difference may be negligible, and entirely dependent upon various implementation decisions. Moreover, 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.

2 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 cases, but not all, systolic architectures data (input and output) is 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. However, 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. In general, data and control structures, synchronization and communication are, 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. Most 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. These provide 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. The computation graph has a *level* or *stage* for each sequential step of the algorithm. Nodes of the graph represent computations, and the computations represented by the nodes at a given level can all be performed concurrently. Arcs between nodes represent data transfer, which with nodes of the computation graph mapped into different processors represent communication. In a sufficiently large ensemble, the mapping of the computation graph can be made such that all nodes of a given level are assigned to distinct processors. The situation in this case is similar to what is typical for systolic designs [69], [65], [17], [90], [96], [95], [102], [88], [19], [22]. If there are fewer processors than the maximum number of nodes at any one level of the computation graph, then different nodes have to be identified with the same processor. Two such schemes are *cyclic* and *consecutive* identification [50] defined precisely later.

In the identification of multiple nodes of the computation graph with processors in 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. However, 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. The algorithms have communication topologies in the form of *one- or multi-dimensional meshes, butterfly networks, data manipulator networks, and spanning trees.* We first describe the embeddings of these graphs in graphs representing the topology of the ensemble architectures.

2.1 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 is 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 the latency may determine the rate of execution during some part of the algorithm as for FFT and odd-even cyclic reduction [15] on vector architectures. The bank conflict problem in vector processors is well known, and architectural solutions [13], [81], [83], [104], as well as solutions at the application program level for particular algorithms [72], [46] have been proposed.

The situation is the same within a node in an ensemble architecture, however, 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.

2.1.1 Complete Binary Tree Hosts

We first consider guest graphs in the form of one- and multi-dimensional arrays. Rosenberg and Snyder [108] and Sekanina [120], 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 indices of the tree nodes to which nodes i and j are mapped. Then, $d_T(\phi(i), \phi(i+1)) \leq 3, \quad \forall i$, 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 [23] 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 [108] 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.

2.1.2 Boolean cube hosts

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 [106]. Such Gray codes have several interesting properties. For instance, it is easy to show that $d_C(G_i, G_{(i+2^j) mod 2^n}) = 2$ for $j > 0$ [58]. This property is important for algorithms such as the FFT, bitonic sort, and cyclic reduction. For the FFT and bitonic sort, 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 property of the binary-reflected Gray code is that for i even $d_C(G_i, G_{(i+3) mod 2^n}) = 1$. Any loop of length $2^{n-1} + 2k, k = \{1, 2, ..., 2^{n-2}\}$ can be embedded in a n-dimensional cube (n-cube) such that $d_C(G_i, G_{(i+1) mod |L|}) = 1$ for $i = \{0, 1, ..., |L| - 1\}$ [50]. For $|L|$ odd there exists a node i in the loop such that $d_C(G_i, G_{(i+1) mod |L|}) = 2$. That the minimum maximum distance must be 2 is easily proved by considering the number of bit complementations in a cycle. 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 $G_i = (g_{n-1}, g_{n-2}, ..., g_0)$ can be rearranged to a binary encoding $i = (b_{n-1}, b_{n-2}, ..., b_0)$ 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

Elem. index	Proc. index	Elem. index	Proc. index	Elem. index	Proc. index	Elem. index	Proc. index
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

Table 6: Conversion of Gray code to binary code

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 [50]. 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. Table 6 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.

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. For meshes with sides that are not powers of 2 the embedding can be made such that the expansion is minimum and the maximum dilation is equal to 2 [43].

For the **FFT butterfly network**, an identification of the corresponding nodes in different ranks with a cube processor yields a dilation 1 embedding. If $(x|y)$ is the address of a butterfly node, where x is n bits and y is $log_2 n$ bits, i.e., x gives the address within a rank of the butterfly, and y is the address of the rank, then all nodes with the same x are mapped into the same node of the Boolean cube with the scheme just suggested. Each cube node performs the task of $log_2 N$

butterfly nodes. Each butterfly communication between nodes is adjacent in the Boolean cube. If a binary-reflected Gray code encoding is applied to x, then butterfly communications are between nodes at distance two, except for the butterfly on the lowest order bit of x.

For the **data manipulator network**, an identification of nodes of the network in the same way as for the FFT butterfly network does not yield communication between adjacent nodes, since any node communicates with nodes $i \pm 2^j$. One of these two communications are with an adjacent node, but the other is not. However, if a binary-reflected Gray code encoding is applied to x, then proximity is preserved in that no communication is over a distance greater than two.

There exist many **spanning graphs** [68] in a Boolean cube. When buffer space is at a premium a Hamiltonian path may be the only choice. If the data volume is low, then the height may be more important, and a *spanning binomial tree* [27,3] may be the best choice. For maximum utilization of the bandwidth n, rotated and translated spanning binomial trees can be used. Such *edge-disjoint spanning binomial trees* [68] are optimum for large data volumes. In many instances several spanning trees are required concurrently, such as if all nodes broadcast data to all other nodes. If communication can take place on only one port at a time, then the spanning binomial tree routing is optimal. However, if communication can take place on all ports concurrently, then routing according to *Balanced Spanning n-trees*, or *Rotated Spanning Binomial Trees*, is optimum [68].

2.1.3 Aggregation of data Elements

For a P by P matrix and a $2n$-cube with $P^2 > 2^{2n}$, 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^n by 2^n array. In *consecutive* storage, all elements $(i,j) = \{0,1,..,P-1\}$ of a matrix A are identified and stored in processor (p,q) $p = \lfloor \frac{i}{\lceil \frac{P}{2^n} \rceil} \rfloor$ and $q = \lfloor \frac{j}{\lceil \frac{P}{2^n} \rceil} \rfloor$. Each processor stores a submatrix of size $\frac{P^2}{2^{2n}}$. In *cyclic* storage, the matrix elements are stored such that elements (i,j) are identified with node (p,q), $p = imod2^n$, and $q = jmod2^n$. In the *consecutive* storage scheme, elements with the same *least significant bits* are identified with the same processor; whereas, in the *cyclic* scheme the identification is made on the *most significant bits.*

With the consecutive storage scheme, algorithms devised for the case of $P = 2^n$ 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, also known as virtual processors, Figure 1.

We find that the cyclic storage scheme enforces a greater insight into 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. For matrix multiplication of square matrices there is no difference in processor

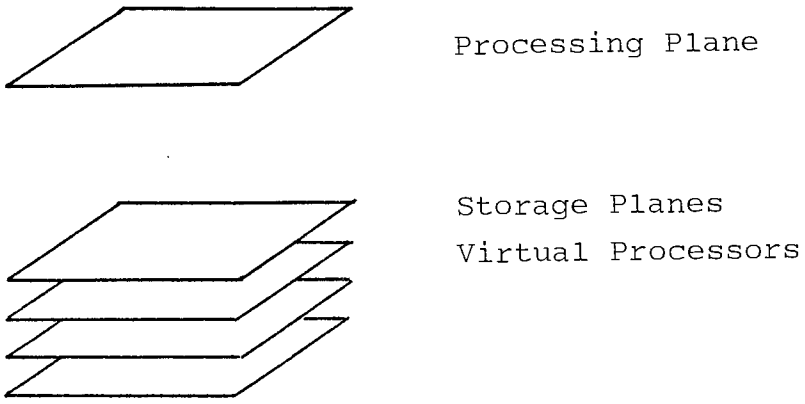

Figure 1: Processing and storage planes (virtual processors)

utilization for the two storage schemes. However, for an algorithm such as LU-decomposition where the number of matrix elements involved in a step is decreasing throughout the computation, cyclic storage may be preferable. For LU-decomposition on a dense matrix it may yield a performance improvement in the range 1.5 - 2 [24]. For the solution of tridiagonal systems it yields a *performance degradation* [58]. The optimization of vector length, or communication packet size does not affect the optimum allocation of data, or the choice of algorithm.

2.2 Data Permutations

2.2.1 Conversion between storage schemes

Rearrangement of consecutive to cyclic storage order (or vice versa) can be carried out in time $P/2^n + n$ for P elements stored in a 2^n processor Boolean cube [50]. 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. In the consecutive storage order the partitioning of the address space is $(a_{n-1}a_{n-2}\ldots a_0|b_{m-1}\ldots b_0)$, where the n highest order bits are processor addresses and the m lowest order bits are local addresses. In the cyclic storage scheme, the bit fields are exchanged to $(b_{m-1}\ldots b_0|a_{n-1}a_{n-2}\ldots a_0)$. Clearly, if $m = n$ the storage conversion is equivalent to a matrix transposition. The exchange can be performed as a sequence of *shuffle* operations, i.e., left cyclic shifts on the address, or *unshuffle* operations through right cyclic shifts. Each such shuffle operation has a maximum path length of n edges. Performing the shuffle operations one at a time results in a communication time proportional to $min(n, m)n$. By performing a bit-wise exclusive-or operation on the addresses before and after the conversion it is clear that the maximum distance an element needs to traverse is n. Since each routing operation is an exchange operation and a dimension is only routed once, it follows that the paths are edge disjoint.

The rearrangement can be made recursively by a sequence of exchanges (exclusive-or operations) on distinct bits. The order in which the bits are treated is immaterial. All exchanges imply communication if $m \leq n$. An alternative implementation of the conversion algorithm is to perform exchanges of elements between pairs of processors differing in successively lower order address bits [50,41], and to perform local shuffle operations to make the elements that are

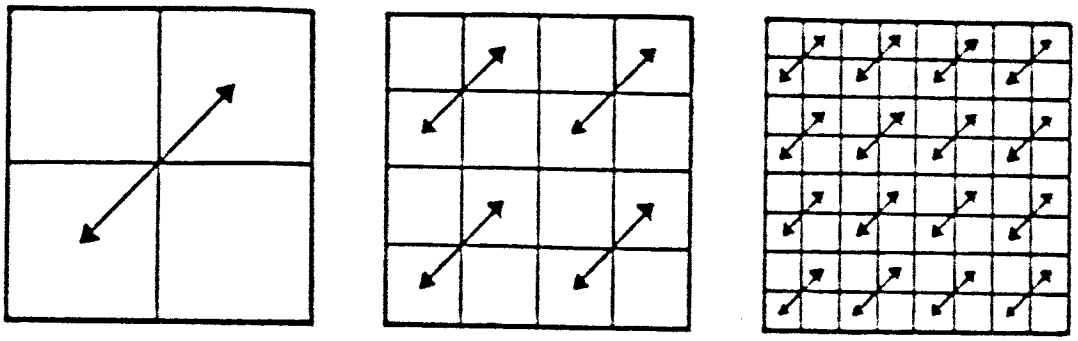

Figure 2: Recursive transposition of a matrix

being exchanged form a contiguous block. 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.

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 the recursion step, or the number of rows or columns. This property is not true in forming the transpose of a rectangular matrix.

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

2.2.2 Matrix transpose

The formation of a matrix transpose is a particular permutation of data elements. With the matrix elements stored consecutively the encoding is $(rp_{n_r-1}rp_{n_r-2}\ldots rp_0|rv_{m_r-1}\ldots rv_0||cp_{n_c-1}$ $cp_{n_c-2}\ldots cp_0|cv_{m_c-1}\ldots cv_0)$, where rp denotes the row processor addresses and rv the local addresses in the row direction (virtual row processors), cp the column processors and cv the local addresses for the column Direction. The transposition corresponds to exchanging the row and column bit fields. It can be carried out recursively [123,26,50,41] as illustrated in Figure 2 for $n_r + m_r = n_c + m_c$.

In the first step of the recursive procedure illustrated in Figure 2, 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 $n_r = n_c = n$, the first n steps imply interprocessor communication; and with $m_r = m_c = m$ the node. These last steps consist of local address changes. (We presume here that the transpose is needed with some other data in some computation). Otherwise, the first n 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 m steps amount to local address changes, whereas the last n steps require interprocessor communication. After the first m steps there are 2^{2m} matrices of size 2^n by 2^n to transpose. All matrices are stored identically.

With $n_r = n_c = n$ there is $2n$ dimensions to be routed. Indeed, all processors on the main anti-diagonal have elements that require a routing distance equal to $2n$. With row and column dimensions taken pairwise, all communications are exchanges over a distance of 2. The paths can be made edge-disjoint and communication pipelined. Moreover, constant storage per node suffices [50,41]. The element transfer time is $2^{2m} + 2n - 1$ accomplished by pipelining the 2^{2m} matrix transpositions, assuming that communication in both directions can take place concurrently on all of the ports of a processor. With the consecutive storage model and using the apparent granularity in the form of block operations the communication time is proportional to $2n \times 2^{2m}$, which for n large is considerably higher. The difference between the two expressions is due to the pipelining of element transfers in the first case. The same complexity is also attainable in the consecutive storage case by pipelining the transfers of elements of the blocks. It is possible to reduce the time further by establishing additional paths [41].

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 on row and column indices 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).

However, with $G(i)$ and $G(j)$ being the Gray code of the row index i and column index j the transpose operation for the case with $m_r = m_c = 0$ is equivalent to the communication implied by changing $(G(i)|G(j))$ to $(G(j)|G(i))$, which indeed is the same operation as in the binary encoded case. Routing row/column dimensions in descending order 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 3. The numbers on the diagonals indicate the order of the reflection the submatrices are undergoing.

The application of the alternating descending order reflection algorithm to a 4-cube is illustrated in Figure 4. The routing algorithm can be made distributed.

Performing the transformation by a 2-dimensional mesh algorithm yields a considerably higher number of routing steps [50], $(2^{2m} + 1)(2^{n-1} - 1)/2$. The order of complexity of that algorithm cannot be reduced since $P(P-1)/2$ elements have to pass through $O(2^n)$ nodes, each of which has 4 ports.

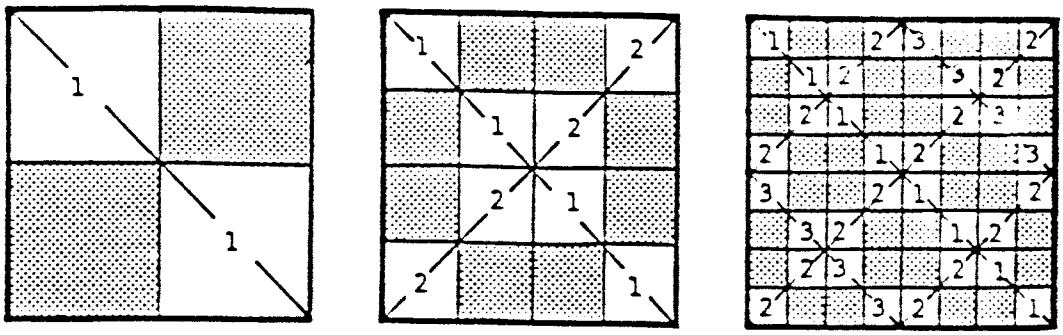

Figure 3: Transposing a matrix stored in a binary-reflected Gray code

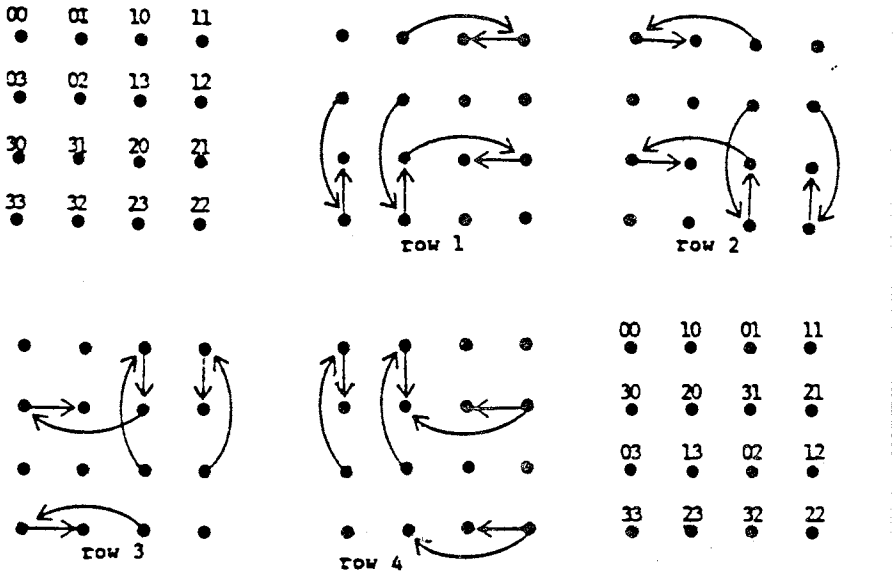

Figure 4: Routing paths in transposing a 4 by 4 matrix on a 4-cube

The results presented for square matrices can be generalized to rectangular matrices [41].

2.2.3 Randomized Communication

Recently, several probabilistic algorithms of complexity $O(log_2 N)$ for arbitrary permutations on a Boolean n-cube and d-way shuffle networks have been devised. The probabilistic algorithms do not guarantee an even distribution of elements during permutation. 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 in either phase, several elements may reach a node, then be delayed because of competition for a given communications link, even in the case that there is precisely one element per node in the initial and final states. Also, several elements may reside in a single node at the end of the first phase. Valiant and Brebner [131] 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 P/N elements per node they show that the probability that the permutation will require more than $(\alpha P/N + 1)log_2 N$ routing steps is less than $(e/2\alpha)^{\alpha(P/N)log_2 N}$. 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 [130] and Aleliunas [4]. It is assumed that a processor can support communication on all its ports currently. Valiant and Brebner also show that for a n-dimensional mesh with $M = m^n$ nodes, the probability that at least one packet has not finished in time $(2n-1)(m + \alpha m^{3/4})$ is less than $C^{\alpha\sqrt{m}}$ for $C < 1$. This result compares favorably with the complexity of Batcher's bitonic sort or odd-even merge on meshes [128], [97], [79]. The Thompson and Kung algorithm yields a complexity of approximately $6\sqrt{M}$ for a 2-dimensional mesh, and $(3n^2 + n)M^{1/N}$ for a n-dimensional mesh. Simulations that exhibit a behavior well within the bounds for a variety of ensemble configurations are also presented.

2.2.4 Scan functions

Some operations apply to sets, such as broadcasting a value to a set of processors, finding the maximum or minimum in a given set of variables, or adding all the values. Critical issues in a system with distributed control (such as in a message passing system) 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. In the Connection Machine, which is a bit-serial, synchronous, SIMD architecture, scan functions are available as operators in the programming language. The scan function requires that a spanning tree be generated for a specified set of processors. The encoding of the set of processors is most convenient if the processors form a contiguous domain in some index space, such as, a one- or multi-dimensional array. Encoding is particularly easy, if the processors correspond to a specific bit-field. Such scans are known as segmented scans in the Connection Machine terminology [39]. The segments need not correspond to all possible addresses generated by a given bit-field, but should be contiguous for ease of encoding.

Assume for simplicity that the set of processors for which the scan operation shall be per-

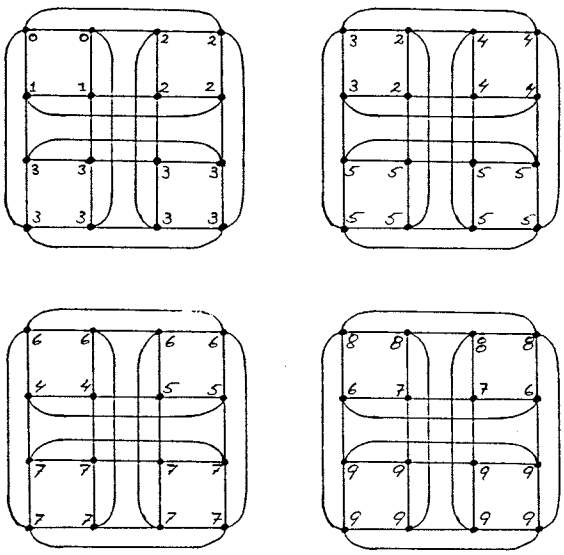

Figure 5: Spanning Binomial Trees in a Boolean cube, with sources in sequential order

formed form a subcube of dimension k. Then, a spanning binomial tree for source node s is generated by every node i performing an exclusive or operation on its address and the source node address and communicating with all its neighbors corresponding to leading zeroes of $i \oplus s$. An interesting property of this simple scan routing is the following [50,53]. If the integers are embedded in the cube according to a binary-reflected Gray code, and the dimensions are routed in the order in which they first appear in the Gray code going from i to $i-1$ in increasing order modulo the size of the subcube, then the order of arrival for every processor is the same as the order of scan initiation, if a scan operation is started at successive processors every other cycle. This situation occurs in Gaussian elimination with pivoting on the diagonal.

The order preserving property for a binary-reflected Gray code is established by observing 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. The behavior of the algorithm for a 4-dimensional cube is shown in Figure 5.

2.3 Sorting

2.3.1 Combining sequential and bitonic sort on a Boolean cube

Stone [123] observed that the bitonic sort [5] maps well onto 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 P evenly distributed elements on a $N = 2^n$ processor Boolean cube for $P > N$.

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, $log_2 P$ sequences are merged serially. The number of sequences merged decreases from $P/2$ to 1. The final step merges two sequences stored in separate $n-1$-cubes into one sequence in an n-cube. The number of routing steps is $n(n+1)/2$, independent of the data. Each routing is performed in only one dimension. An algorithm for the bitonic sort expressed in pseudo code for $P = 2^n$ 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

For $P = 2^n$ the last merge operation involves two sequences of length $P/2$. 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. If $P > 2^n$, then additional sequential steps are necessary.

With the sorted sequence to be stored in cyclic order the first $log_2 P - n$ comparison-exchange operations of the final merge are local to a node, since the merge is performed recursively on successively shorter sequences, i.e., from the high order bits to the low order bits, and the higher order bits are local in the cyclic storage scheme. After these steps the result is $\frac{P}{N}$ bitonic sequences ordered with respect to each other. Each sequence has one element per node. The last n steps are separately performed on each of those sequences. Carrying out the first $log_2 P - n$ local steps as a bitonic merge yields poor performance. The operational complexity is $O(\frac{P}{N}(log_2 P - n)$, compared to $O(\frac{P}{N} + log_2 \frac{P}{N})$ for a sequential merge including bisection to find the maximum/minimum of the local bitonic sequence. This merge can be carried out concurrently in all processors [49]. The correctness of the algorithm can be proved by observing that the first $log_2 P - n$ steps, given the assumed storage order, realize N independent bitonic mergers, each for $\frac{P}{N}$ elements. The corresponding output elements from these mergers form a bitonic sequence. The last n steps realize $\frac{P}{N}$ bitonic mergers for sequences of length N. This situation is a generalization of Batcher's construction [5] of a 16-sorter out of 4-sorters, see Figure 2.3.1.

The sorting is accomplished by recursively building longer sorted sequences, starting from the lowest order bits. The $\frac{P}{N}$ local elements belong to different sequences for the first n merges, each being a recursive merge. The time for cyclic sort by the algorithm outlined above is $T = \frac{P}{2N}(n(2log_2 P - n + 1)(4t_c + t_{ce})/2 + 2(log_2 P - n - 1)t_{ce}) + t_{ce}$, where t_c is the time for

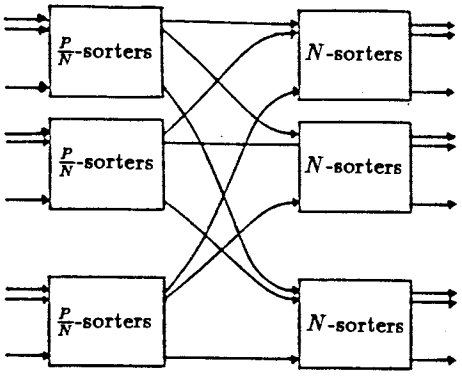

Figure 6: A network of N $\frac{P}{N}$-sorters followed by $\frac{P}{N}$ N-sorters.

communication of an element between a pair of processors, and t_{ce} is the time for a comparison operation. If $P \gg N$ then T is of order $O((\frac{P}{N})log_2P)$, and if $P \approx N$, T is of order $O(log_2^2N)$. The speed-up is $O(N)$ for $N \ll P$ and gradually changes to $O(N/log_2N)$.

With sorting into consecutive storage order and the elements initially stored consecutively, the first $log_2P - n$ merges of the bitonic sort is local to a processor, with the merge sequence progressing from low to high order bits. These first steps generate a local sorted sequence, that is more efficiently created by a good sequential sort. The last n merges require interprocessor communication, and involves sequences of $\frac{P}{N}$ elements instead of single elements. This algorithm is similar to the one proposed in [6]. The final log_2P-n steps of each of the last n bitonic merges are local to a processor and should be performed as a sequential merge. The communication and comparison complexity is of the same order as for sorting in cyclic order [49].

A cyclic storage order is generated by building sorted sequences over an increasing number of nodes. Then, when a sorted sequence extends over all nodes additional elements should be included locally in the proper way. For the consecutive storage order the sorted sequences are first built locally, then extended over the processors when all local elements are included.

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.

2.3.2 Distribution counting

Rank assignment in the context of distribution counting [74] with L counters, or "buckets", can be carried out in a time of $[\frac{P}{N} + L + n - 1]2t_a + [L(1 - \frac{1}{N})3 + n]2t_c$ for $N \leq L$, and $[\frac{P}{N} + L + n - 1]2t_a + [6(L-1) + 5n - 3log_2L]t_c$ for $N > L$, on a Boolean cube [49] (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 $O(N/log_2N)$. The rank assignment algorithm that

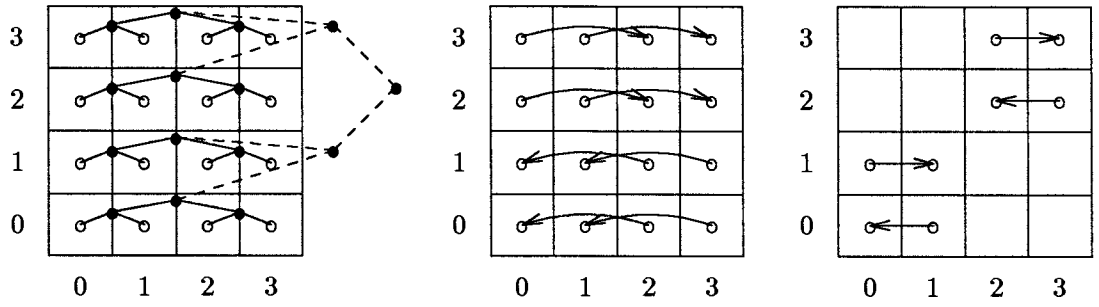

Figure 7: Concurrent tree computations on a Boolean cube

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 [40].

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 [75].

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 7 illustrates the computations for $N = L = 4$.

2.4 Linear Algebra Computations

In this section we briefly describe a mesh algorithm by Cannon [16], which also is suitable for Boolean n-cubes since a two-dimensional mesh can be embedded in a Boolean cube with edge dilation one. We also briefly mention a few variations [50] of this algorithm. A recursive algorithm [21] that maps directly to a Boolean cube is also discussed, in particular its routing paths and pipelining properties. The multiplication of matrices of arbitrary shapes is treated in [66]. We also discuss some of the concerns in solving dense, triangular, and banded and tridiagonal systems on ensemble architectures. We conclude with a discussion on the relative merits of FFT and tridiagonal solvers on such architectures.

2.4.1 Matrix multiplication

Cannon [16] presents an algorithm for computing the product C of two $\sqrt{N}$ by $\sqrt{N}$ matrices A and B stored in a 2-dimensional array of identical size. The algorithm requires $\frac{3}{2}\sqrt{N}$ communication steps, out of which $\lceil\frac{\sqrt{N}}{2}\rceil$ steps are for a set-up phase, and $\sqrt{N}-1$ are for the multiplication phase. The purpose of the set-up phase is to align elements from the two matrices such that all nodes in the 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 the alignment seen in many systolic algorithms [80,69].

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: $E(i,j) \leftarrow E(i,(i+j)mod\sqrt{N})$, $F(i,j) \leftarrow F((i+j)mod\sqrt{N},j)$, $G \leftarrow 0$ for $(i,j) \in \{0,1,2,...,\sqrt{N}-1\} \times \{0,1,2,...,\sqrt{N}-1\}$. 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: $G(i,j) \leftarrow G(i,j) + E(i,j) \times F(i,j)$, $E(i,j) \leftarrow E(i,(j+1)mod\sqrt{N})$, $F(i,j) \leftarrow F((i+1)mod\sqrt{N},j)$, $i,j = \{0,1,2,...,\sqrt{N}-1\}$. With A a $P \times Q$ matrix and B a $Q \times R$ matrix the multiplication can be accomplished in a time of $max(\lceil\frac{P}{\sqrt{N}}\rceil,\lceil\frac{R}{\sqrt{N}}\rceil)\lceil\frac{Q}{\sqrt{N}}\rceil(\sqrt{N}-1)t_c + \lceil\frac{P}{\sqrt{N}}\rceil\lceil\frac{Q}{\sqrt{N}}\rceil\lceil\frac{R}{\sqrt{N}}\rceil((\sqrt{N}-1)(t_a+max(t_a,t_c))+2t_a)$.

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 $\sqrt{N}$ "cycles". For instance, using the outer product formulation [46] 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 [54]. 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 unidirectional communication, and end-around connections, a total time of $5(\sqrt{N}-1)$ "cycles" is required for one matrix multiplication. The complexity of the algorithm may be improved [50], 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.

The complexity of the algorithm can be reduced by a term $\sqrt{N} - \frac{1}{2}n$, if the alignment can be accomplished in time n instead of $\sqrt{N}$. For matrices of a size comparable to the number of processors this difference is also significant relative to the total time. Dekel et. al. [21] describes such an algorithm for $\sqrt{N}$ by $\sqrt{N}$ matrices embedded in a Boolean cube of N 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 $(i,j) \in \{0,...,\sqrt{N}-1\} \times \{0,...,\sqrt{N}-1\}$. 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 n 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 $P+n-1$.

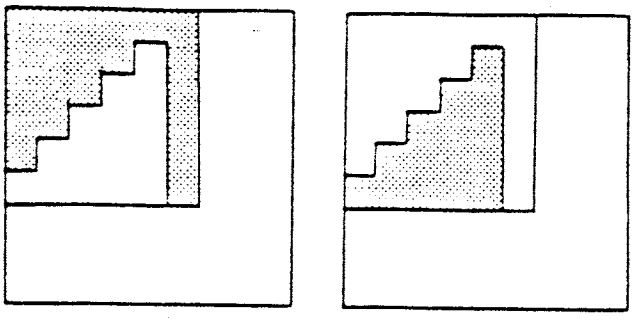

Figure 8: The computational window at step j for computing $C \leftarrow A \times B + C$, A upper triangular, or strict lower triangular

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 [21]. It follows that the time for multiplying a Q by R matrix B by an P by Q matrix A on a Boolean n-cube, with A, B and $C \leftarrow A \times B + C$, embedded according to a separate binary encoding of row and column indices, is at most $((\lceil \frac{P}{\sqrt{N}} \rceil + \lceil \frac{R}{\sqrt{N}} \rceil)\lceil \frac{Q}{\sqrt{N}} \rceil + log_2 N - 1)t_c + \sqrt{N}\lceil \frac{P}{\sqrt{N}} \rceil \lceil \frac{R}{\sqrt{N}} \rceil \lceil \frac{Q}{\sqrt{N}} \rceil max(2t_a, t_c)$. The number of submatrices of size $\sqrt{N} \times \sqrt{N}$ is $(\lceil \frac{P}{\sqrt{N}} \rceil + \lceil \frac{R}{\sqrt{N}} \rceil)\lceil \frac{Q}{\sqrt{N}} \rceil$. The number of block matrix multiplications is $\lceil \frac{P}{\sqrt{N}} \rceil \lceil \frac{R}{\sqrt{N}} \rceil \lceil \frac{Q}{\sqrt{N}} \rceil$. 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 $(\lceil \frac{P}{\sqrt{N}} \rceil \lceil \frac{R}{\sqrt{N}} \rceil \lceil \frac{Q}{\sqrt{N}} \rceil \sqrt{N} - 1)max(2t_a, t_c) + \alpha d + 2t_a$, where d denotes the diameter of the ensemble configuration and $\alpha \leq 4$ [54].

The multiplication of rectangular matrices is treated in detail in [66]. Depending on the shape of the matrices and the parameters of the machine, any matrix algorithm as outlined above, or a matrix-vector algorithm as described below, or the computation of the transpose of the product may be the optimum.

2.4.2 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 8. It also shows the computational window for step j, if A is a strict lower triangular matrix, $a_{ij} = 0, ij \leq 0$.

From Figure 8 it is obvious that the multiplication $A \times B$ and $D \times B$, where A is upper triangular and D strict lower triangular of dimension $\sqrt{N}$ by $\sqrt{N}$, or A strict upper triangular

and D lower triangular can be performed concurrently on a torus of dimension $\sqrt{N}$ by $\sqrt{N}$, or a Boolean n-cube.

2.4.3 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 [54]. An algorithm on a Boolean cube that 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 $\sqrt{N} \times \sqrt{N}$ matrix and x a $\sqrt{N}$ vector with components x_i. Assume that the vector x is aligned with the first column of A. First x_i is rotated i steps in the direction of increasing column index for $i = \{0, 1, ..., \sqrt{N}-1\}$; 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 n. 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. A copy-scan can be used within columns. A sum-scan 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 [54] for dense matrices, and in [55] for banded matrices.

2.4.4 Factorization of dense matrices

The algorithms for Gaussian elimination and Gauss-Jordan elimination described below can be viewed as modifications of systolic algorithms [80], [52]. 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], [63], for Given's rotations in [30], [33], [2], [59], and for Householder transformations in [51]. Given's and Householder's methods make use of unitary transformations and are numerically stable.

The factorization of a matrix A 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.

The non-trivial elements of a factor become known after the preceding 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 $A_i = J_{i-1}A_{i-1}$.

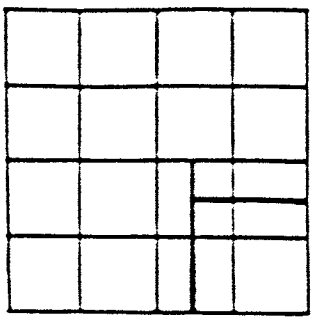

Figure 9: Storage and distribution of pivot row and column in dense matrix factorization

We assume that A is stored in a 2-dimensional array with one element per processor. We first present algorithms for 2-dimensional arrays, 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 [54] and their distribution is illustrated in Figure 9. Each application of a factor is similar to performing a column by row product in the outer product matrix multiplication algorithm.

For each column elimination operation, a number of elements need 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. However, it does in the event of partial pivoting. On the other hand, 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 scan algorithm described earlier guarantees the same

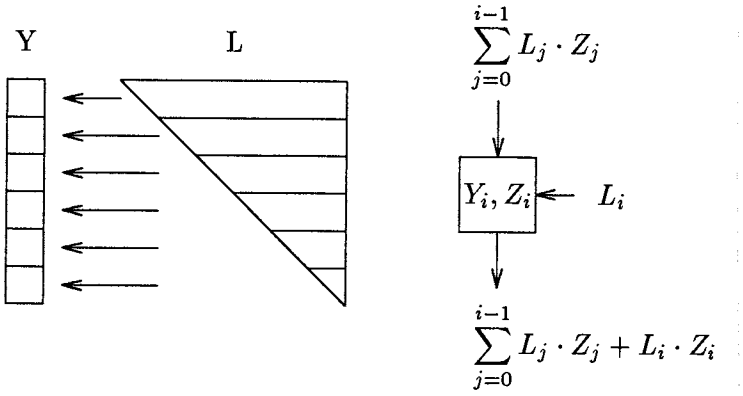

Figure 10: Data movement in a linear recurrence algorithm on a torus

order of arrival as that of distribution [55]. Note, that in Gaussian elimination the distribution for the last several equations only needs to cover successively smaller cubes.

2.4.5 Solution of Triangular Systems

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. Their complexity has been analyzed, assuming zero communication cost. Sameh [114] gives a survey of such algorithms and their properties. We present an algorithm for mesh or Boolean cube configured ensembles [50]. It is an adaptation of the binary tree algorithm by Johnsson [57], which in turn is a particular instantiation of the column-sweep algorithm described by Kuck [77]. 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 preceding 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 $\sqrt{N}$ by $\sqrt{N}$ processors, the service of a processor is requested for a new row every $(t_d + 2t_a)\sqrt{N}$ 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

$$\lceil\frac{P-m}{\sqrt{N}}\rceil max(t_d+2(m-1)t_a,(t_c+t_d+2t_a)\sqrt{N})+p\sum_{j=0}^{\lfloor\frac{m}{\sqrt{N}}\rfloor} max(t_d+2(m-j\sqrt{N}-1)t_a,(t_c+t_d+2t_a)\sqrt{N}).$$

For banded systems a recurrence solver can also be based on the partitioning method [115]. This approach can further reduce the complexity of solving linear recurrences. The partitioning method is discussed further in the next section.

2.4.6 Banded System Solvers

Tridiagonal systems

Irreducible tridiagonal systems of equations of order P can be solved in $2log_2P$ steps using $O(P)$ arithmetic operations by odd-even cyclic reduction [15]. The method has been modified by Hockney [46] to yield a solution in log_2P steps, but at the expense of $O(Plog_2P)$ arithmetic operations. For highly concurrent ensembles it is of interest to find mappings of the computation graph onto 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 p for $P = 2^p - 1$ and $P = 2^p$ 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. There exists a mapping of the computation graph for cyclic reduction on P equations onto a binary tree of P nodes such that the communication complexity is $3log_2P$ [58]. A comparable communication complexity is also obtainable on shuffle-exchange networks and Boolean cubes [58].

The computation graph of cyclic reduction is shown in Figure 11. For $P = N$ mapping the equations onto nodes in the tree in order 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 n, the second a time proportional to $n-1$, etc. However, the reduction steps can be pipelined, and the total time is proportional to $3n$ [58], [28]. The inorder mapping is shown in Figure 12.

For $P > N$ several equations must be identified with the same processor node. For load balancing it is desirable to make the division as even as possible. For simplicity we assume here that $P = 2^mN$, i.e., that m address bits are required for the local address. The consecutive storage scheme can be considered as forming a quotient graph from the computation graph by combining a successively indexed node 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 $\lceil\frac{P}{N}\rceil$ equations is $2^{pmodn-1}$, which corresponds to a *pmodn* 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 $p-n$ levels starting with the leaf level. The quotient graph approach provides the best possible computational balance.

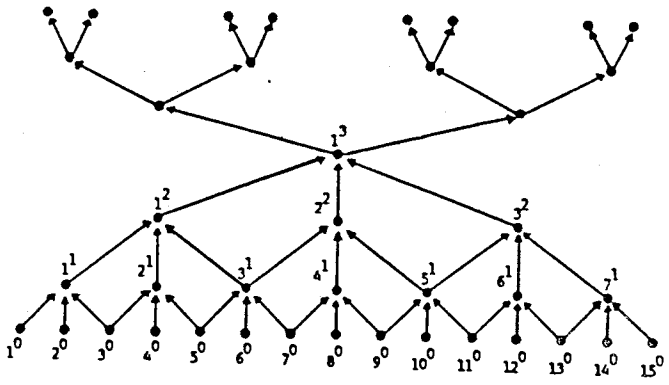

Figure 11: The computation graph for odd-even cyclic reduction

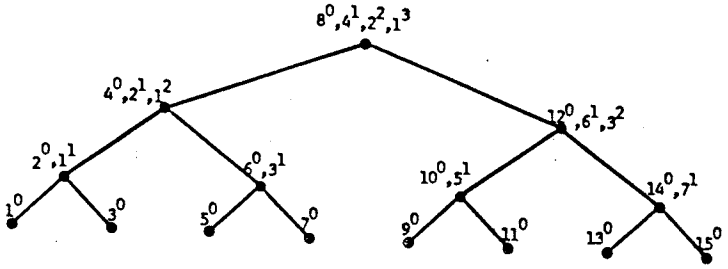

Figure 12: Inorder mapping of cyclic reduction on to a binary tree

A critical observation in finding communication efficient mapping is that the communication between some pairs 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. Hence, odd-even cyclic reduction applied to this mapping is not efficient, but by changing the elimination order to substructured elimination the mapping is effective. Only one communication is required in the substructuring phase. The amount of fill-in is approximately the same as in odd-even cyclic reduction, and so is the arithmetic complexity. The reduced system is then solved by cyclic reduction using an inorder map. The total complexity is of order $O(\frac{P}{N} + n)$ [58,62].

The substructured algorithm is of minimum order of complexity, both with respect to communication and arithmetic. For a diagonally dominant system the the diagonal dominance increases during substructuring [107] and no or only a few reduction steps may be sufficient. If a few cyclic reduction steps suffice, then a proximity preserving embedding [108] of the quotient graph may be advantages. 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 $P = N$ the running time is proportional to the reduction steps executed, while on a single processor 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 $P = N$ is $O(\frac{N}{log_2 N})$, 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, substructured elimination with odd-even cyclic reduction for the reduced

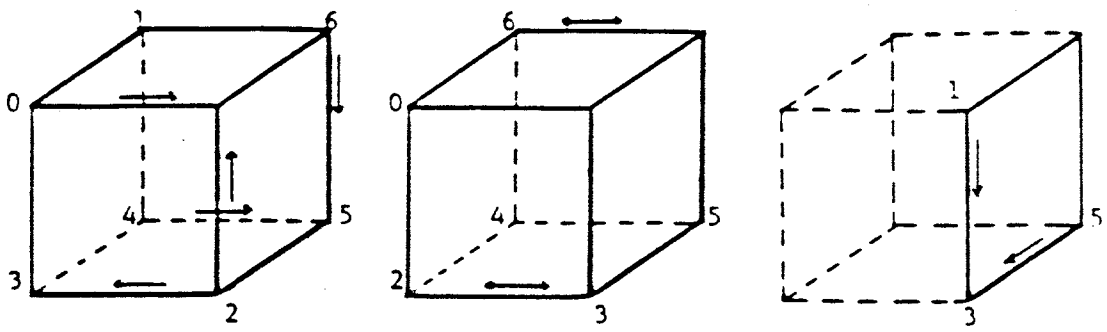

Figure 13: Cyclic reduction on a Boolean cube

system has a communication complexity $O(n)$. Each step of the cyclic reduction algorithm involves 3 nodes of the computation graph. An embedding according to a binary encoding would require communication across n edges for the first step of the algorithm, $n-1$ edges for the $2nd$ step, $n-2f$ for the $3rd$ step, 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 [58,62]. 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 monotonely decreasing dimensionality. For $P = N$ 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 13 illustrates a few steps in the reduction process for $P = N = 8$.

Odd-even cyclic reduction has a higher arithmetic complexity than Gaussian elimination, and it also requires more communications than if a transposition of the data is performed to one processor, and the result distributed back to where the equations came from. Hence, for certain combinations of arithmetic and communication capabilities it may be faster to use a transposition and a sequential algorithm, and even to avoid substructuring [112,67].

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 [58].

A detailed experimental study of optimum methods for the solution of single and multiple tridiagonal systems on the Intel iPSC is reported in [67].

General Banded Systems

The substructuring technique has also been applied to banded systems. Sameh and collaborators [115,82,25] use partitioning to reduce banded systems of bandwidth $2m+1$ to dense, block

pentadiagonal, systems of order $2mN-1$ for N partitions. The blocks are of size $m \times m$. The solution of the reduced system by Gaussian elimination on a linear array is considered in [82], and the solution by block-Jacobi and preconditioned conjugate gradient methods in [25]. Reiter and Rodrigue [107], and Johnsson [61,60,24] analyze a slightly different substructuring that reduces the banded system to a dense, block tridiagonal system of order mN. 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 N_{opt} depends on m, P, and the ratio of the communication and computation bandwidths. N_{opt} for Gaussian elimination on a linear array is of order $O(\sqrt{\frac{P}{m}})$ and the corresponding complexity is of order $O(m^2\sqrt{Pm})$. Block cyclic reduction yields a lower complexity under a variety of conditions, even on a linear array. The value of N_{opt} falls in the range $O(\sqrt{P}) \leq N_{opt} \leq O(\frac{P}{m})$, and the corresponding complexity is in the range $O(m^2\sqrt{P} + m^3 log_2 P)$ to $O(m^3 + m^3 log_2 \frac{P}{m})$ [61]. For binary trees, shuffle-exchange networks, and Boolean cubes, N_{opt} is of order $O(\frac{P}{m})$ and the corresponding complexity of order $O(m^3 + m^3 log_2 \frac{P}{m})$. For small matrix bandwidths this algorithm yields good speed-up, however, 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 $\frac{P}{m}$. However, it is possible to exploit concurrency in the operations also within the partitions, which for m of order $O(P)$ is the main source of concurrency. For instance, one can use $N_c \leq m^2$ processors configured as a mesh or a Boolean cube during the elimination of the elements in one column, as in systolic algorithms [80] [52], but use the dual formulation in which the factors are computed *in-place*. The speed-up is of order N_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 14 illustrates an intermediate state of the factorization process.

The scan 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 $N_c = m^2$, and $P \approx m$. The complexity of solving banded systems by this approach is $O(m^2 \frac{P}{N_c})$ $(N = 1)$ [55]. For symmetric matrices, storage as well as time can be saved using a parallel version of Cholesky's method [2], [63].

The two methods can be combined such that N_c processors are used for each partition. Such a set of processors is referred to as a *cluster*. With $\frac{N}{N_c}$ clusters of N_c processors each, intracluster connections in the form of a 2-dimensional mesh (or torus) or Boolean cube, and intercluster connections forming a binary tree, shuffle network, or Boolean cube, the minimum time complexity is of order $O(m + m log_2 \frac{P}{m})$, and N_{opt} of order $O(N_c \frac{P}{m})$, and $N_{c_{opt}}$ of order $O(m^2)$ [55]. Note that for a Boolean cube a subcube can be considered as a cluster.

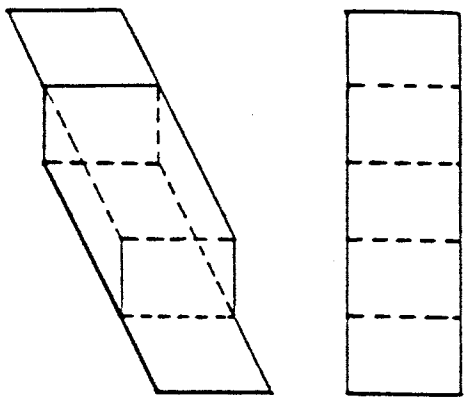

Figure 14: Band matrix factorization on a 2-dimensional array or Boolean cube

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

2.4.7 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 $O(Plog_2log_2P)$ for a $P \times P$ grid [44], [45], [124]. Stone [123] observed that the FFT can be carried out in log_2P steps on a P-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, $log_2\frac{P}{N}$ butterfly operations are local to a node. The arithmetic complexity is of order $O(\frac{P}{N}log_2P)$ and the communication complexity is of order $O(\frac{P}{P}log_2P)$. The speed-up is proportional to P. FFT algorithms for linear arrays are given in a number of references [103], [69], [64]. An early description of a FFT on a 2-dimensional array is given by Stevens [122]. An analysis of the area-time aspects of the FFT on a variety of ensemble configurations is given in [127].

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 [44], [45], [15], [14], [124], [125], [126]. By exploiting symmetries and using real transforms, the number of arithmetic operations per point is less than $2.5log_2P$ in the FFT computation. Using Gaussian elimination with precomputed factors [125],

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 [98]. 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 P^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 $O(log_2 P)$, 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 communications in the Boolean cube is $2log_2 P$ for odd-even cyclic reduction on P equations, which can be reduced to $log_2 P$ for parallel cyclic reduction [46]. Even though it seems preferable to use parallel cyclic reduction, this is not necessarily true [56]. With a binary-reflected Gray code embedding of the equations each such communication, but the first is over two edges. In a packet switched communication system the number of communications is $4log_2 P$ and $2log_P$, respectively. It suffices with $4log_2 P$ communications even if the communication system only supports one send *or* one receive operation at a time. In the case of the Poisson equation only the right hand side need to be transferred. The number of real arithmetic operations is at most $7log_2 P$, which can be reduced to $6log_2 P$ with precomputed factors.

With P^2 processors a real FFT on P points requires $5log_2 P - const$ operations per point, if the butterfly computations are split between two processors, otherwise $10log_2 P - const$. In the former case two exchanges are required per butterfly, otherwise one exchange suffice. If the communication system only supports one send or one receive operation, then each exchange requires two communications. With a binary encoding of the lattice all communications are over single edges, but if the lattice is embedded by a binary-reflected Gray code embedding then the communication is over two edges. Hence, an FFT and an inverse FFT on P points require between $2log_2 P$ and $16log_2 P$ communications depending on the communication system, the embedding, and the load balancing. Each communication involves a complex variable.

Using a tridiagonal solver instead of an FFT-IFFT saves at least $3log_P$ to $4log_2 P$ arithmetic operations and maybe also communication (depending upon the architecture). In the FFT-IFFT approach all nodes are used in all steps, but in the cyclic reduction tridiagonal system solver the number of active nodes decreases. Most of the tridiagonal systems are sufficiently diagonally dominant so that the reduction process can be truncated. This property does not reduce the total solution time in this extreme case (P^2 processors for P^2 lattice points). With fewer processors it gives rise to an interesting load balancing problem.

Sameh [113] 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 [115]. 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 P^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, N, P, and the arithmetic rate, the communication rate, and the overhead in these operations. Gaussian elimination requires $4\frac{P^2}{N} + 2P + \alpha\sqrt{N}$ for a $\sqrt{N} \times \sqrt{N}$ mesh, substructuring with Gaussian elimination for the reduced system requires $9\frac{P^2}{N} + 4\frac{P}{\sqrt{N}} + (2+\alpha)\sqrt{N}$, and substructuring with cyclic reduction for the reduced system on a Boolean n-cube $9\frac{P^2}{N} + \frac{P}{\sqrt{N}}(3+2\alpha)log_2N$. Precomputed coefficients are assumed for these estimates. Cyclic reduction for the reduced system becomes competitive for N approaching P on a Boolean cube configured ensemble [58,62]. On a linear array the logarithmic term premultiplied by α is replaced by a term linear in $\sqrt{N}$, as for Gaussian elimination. Which method is preferable on a linear array is critically dependent upon architectural parameters, N, and P. An accurate comparison should also account for the truncation of the reduction process 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 [58].

2.4.8 Iterative methods

Conjugate Gadient Methods

The conjugate gradient method [36] 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 P equations in P steps. Each step requires $O(PZ)$ arithmetic operations for a system $Ax = y$ in which A has PZ 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. However, 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 P steps (possibly much fewer steps).

The minimum time per iteration is $O(log_2P)$ 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 $O(Plog_2P)$, the same order as that of Householder's method. Preconditioning that would allow the iterative process to be terminated in less than $\frac{P}{log_2P}$ 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 [111], and linear arrays [110]. The implementation of the preconditioned conjugate gradient method with various preconditioners has also been investigated by Kamath and Sameh [70]. 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 [57]. The effect of preconditioning on the computational complexity is analyzed. Van Rosendale [109] 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 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 [18] 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 [94]. Baudet [6] 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 [133]. Recently, asynchronous iteration has also been studied by Lubachevsky and Mitra [91].

3 Summary

The capacity 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 P elements on a binary tree of P nodes may require a time proportional to $P+O(log_2 P)$. Arbitrary permutations can be performed on a 2-dimensional mesh in time $6\sqrt{P}$ [128], on the shuffle-exchange network in time $log_2 P(log_2 P - 1)$, and on the Boolean cube in time $\frac{1}{2} log_2 P(log_2 P + 1)$ using the deterministic algorithm of Batcher, or with high probability in $c log_2 P$ 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 capability of an ensemble configuration is to determine to what extent one configuration can emulate another without a substantial increase in running time. Of the networks discussed here, the Boolean cube and the Cube Connected Cycles networks are the most powerful. 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 ori-

gins 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 $\sqrt{N} \times \sqrt{N}$ matrix can be formed in $\frac{1}{2}\sqrt{N} - 1$ routing steps on the mesh configured ensemble, and $log_2 N$ routing steps on the Boolean cube. This difference becomes significant first for fairly large N. However, the transpose of a $P \times P$ matrix on a $\sqrt{N} \times \sqrt{N}$ mesh requires a time of at least order $\frac{O(P^2}{\sqrt{N})}$, and at most $\frac{1}{2}\lceil\frac{P}{\sqrt{N}}\rceil^2 + 1(1/2\sqrt{N} - 1)$ routing steps [50]. On the Boolean cube the transpose can be performed in $(\lceil P\ over \sqrt{N}\rceil^2 + log_2 N - 1)$ routing steps, an improvement by a factor of approximately $\sqrt{N}/4$ 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 $N_{opt} \approx \beta\sqrt{\frac{P}{\alpha}}$ and $T_{min} \approx \gamma\sqrt{P}$, 2-dimensional mesh $N_{opt} \approx \beta(P\ \alpha)^{2/3}$ and $T_{min} \approx \gamma P^{1/3}$, binary tree, shuffle-exchange, and Boolean cube networks $N_{opt} \approx \beta P/1 + \alpha$ and $T_{min} \approx \gamma log_2 P$, where α is the ratio between the arithmetic and communication bandwidths [58]. For band matrix solvers based on the partitioning technique, the optimum number of processors configured as a linear array is of order $O(\sqrt{P/m})$ for matrices of bandwidth $2m + 1$, and $O(P/m)$ for binary tree, shuffle-exchange and Boolean cube networks. The corresponding solution times are of order $O(m^2\sqrt{Pm})$ and $O(m^3 + m^3 log_2(P/m))$, respectively [54]. For ensembles configured as Boolean cubes, band matrix solvers of complexity $O(m + m log_2 \frac{P}{m})$ can be devised [58].

With an 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. 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 $(4 - 2^{-\lfloor log_2 n \rfloor})d$ 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. For the embedding of arbitrary meshes see [42,43]. 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 onto 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 onto 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 onto 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 $O(Nlog_2^2N)$ operations compared to $O(Nlog_2N)$ 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 [65]. 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 [12]. 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 [89], 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 compu-

tations 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 assistance with the illustrations. The author is also indebted to the Office of Naval Research for providing financial support under contract N00014-84-K-0043.

References

[1] Loyce Adams. *Iterative Algorithms for Large Sparse Linear Systems on Parallel Computers.* Technical Report 166027, NASA Langley Research Center, 1982.

[2] Hassan M. Ahmed, Jean-Marc Delosme, and Martin Morf. Highly concurrent computing structures for matrix arithmetic and signal processing. *Computer*, 15:65–82, January 1982.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[4] R. Aleliunas. Randomized parallel computation. In *ACM Symposium on Principles of Distributed Computing*, pages 60–72, ACM, 1982.

[5] Kenneth E. Batcher. Sorting networks and their applications. In *Spring Joint Computer Conference*, pages 307–314, IEEE, 1968.

[6] Gerald M. Baudet. Asynchronous iterative methods for multiprocessors. *J. ACM*, 25(2):226–244, 1978.

[7] Sandeep N. Bhatt, F.R.K. Chung, F. Tom Leighton, and Arnold L. Rosenberg. Optimal simulations of tree machines. In *Proc. 27th IEEE Symp. Foundations Comput. Sci.*, pages 274–282, IEEE Computer Society, 1986.

[8] Sandeep N. Bhatt and Ilse I.F. Ipsen. *How to Embed Trees in Hypercubes.* Technical Report YALEU/CSD/RR-443, Yale University, Dept. of Computer Science, December 1985.

[9] Sandeep N. Bhatt and Charles E. Leiserson. *Minimizing the Longest Edge in a VLSI Layout.* Technical Report MIT VLSI Memo 82-86, MIT, 1982.

[10] Smith B.J. Architecture and applications of the hep multiprocessor computer system. In *Real-Time Signal Processing IV, Proc of SPIE*, pages 241–248, 1981.

[11] Richard P. Brent and H.T. Kung. On the area of binary tree layouts. *Information Processing Letters*, 11(1):44–46, 1980.

[12] Sally A. Browning. *The Tree Machine: A Highly Concurrent Computing Environment.* Technical Report 1980:TR:3760, Computer Science, California Institute of Technology, January 1980.

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

[14] Billy L. Buzbee. A fast poisson solver amenable to parallel computation. *IEEE Trans. Computers*, C-22:793–796, 1973.

[15] Billy L. Buzbee, Gene H. Golub, and C W. Nielson. On direct methods for solving poisson's equations. *SIAM J. Numer. Anal.*, 7(4):627–656, December 1970.

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

[17] Peter R. Capello and Kenneth Steiglitz. *Unifying VLSI Array Design with Linear Transformations of Space-Time.* Technical Report TRCS83-03, UC Santa Barbara, Dept of Computer Science, May 1982.

[18] D. Chazan and Willard L. Miranker. Chaotic relaxation. *Linear Algebra and its Applications*, 2:199–222, 1969.

[19] Marina C. Chen. Synthesizing systolic designs. In *2nd InternationalSymposium on VLSI Technology, Systems, And Applications*, IEEE Computer Society, 1985.

[20] W. Crowther, J. Goodhue, E. Starr, R. Thomas, W. Milliken, and T. Blackadar. Performance measurements on a 128-node butterfly parallel processor. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 531–540, IEEE Computer Society, 1985.

[21] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM J. Computing*, 10:657–673, 1981.

[22] Jean-Marc Delosme and Ilse C.P. Ipsen. An illustration of a methodology for the construction of efficient systolic architecture in vlsi. In *2nd InternationalSymposium on VLSI Technology, Systems, And Applications*, IEEE Computer Society, 1985.

[23] R.A. DeMillod, Stanley C. Eisenstat, and Richard J. Lipton. Preserving average proximity in arrays. *Communicationsof the ACM*, 21:228–231, March 1978.

[24] Jack Dongarra and S. Lennart Johnsson. Solving banded systems on a parallel processor. *Parallel Computing*, 5(1&2):219–246, 1987. (ANL/MCS-TM-85, November 1986).

[25] Jack J. Dongarra and Ahmed H. Sameh. *On Some Parallel Bandded System Solvers.* Technical Report ANL/MCS-TM-27, Argonne National Laboratories, 1984.

[26] J.O. Eklundh. A fast computer method for matrix transposing. *IEEE Trans. Computers*, C-21(7):801–803, 1972.

[27] Michael J. Fischer. *Efficiency of Equivalence Algorithms*, pages 153–167. Plenum Press, 1972.

[28] Dennis Gannon and John Van Rosendale. On the impact of communication complexity in the design of parallel numerical algorithms. *IEEE Trans. Computers*, C-33(12):1180–1194, December 1984.

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

[30] W. Morven Gentleman and H.T. Kung. Matrix triangularization by systolic arrays. In *Real-Time Signal Processing IV, Proc. of SPIE*, pages 19–26, SPIE, 1981.

[31] Allan Gottlieb, R. Grishman, Clyde P. Kruskal, K.P. McAuliffe, L. Rudolph, and M. Snir. The nyu ultracomputer - designing an mimd shared memory parallel computer. *IEEE Trans. Computers*, C-32(2):175–189, 1983.

[32] J.W. Greene and A. El Gammal. Area and delay penalties in restructurable wafer-scale arrays. In *Third Caltech Conference on VLSI*, pages 165–184, Computer Sciences Press, 1983.

[33] Donald E. Heller and Ilse C.P. Ipsen. Systolic networks for orthogonal equivalence transformations and their applications. In P. Penfield Jr, editor, *Proceedings, Advanced Research in VLSI*, pages 113–122, Artech House, 1982.

[34] John L. Hennessey, N. Jouppi, Forrest Baskett, and J. Gill. Mips: a vlsi processor architecture. In *VLSI Systems and Computations*, pages 337–346, Computer Sciences Press, 1981.

[35] John L. Hennessey, N. Jouppi, S. Przybylski, and C. Rowen. Design of a high performance vlsi processor. In *Proc. of the Third Caltech Conference on VLSI*, pages 33–54, Computer Sciences Press, 1983.

[36] M.R. Hestenes and E. Stiefel. Methods of conjugate gradient for solution of linear systems. *J. Res. Nat. Bur. Standards*, 49:409–436, 1952.

[37] W. Daniel Hillis. *The Connection Machine.* Technical Report Memo 646, MIT Artificial Intelligence Laboratory, 1981.

[38] W. Daniel Hillis. *The Connection Machine.* MIT Press, 1985.

[39] W. Daniel Hillis and Guy L. Steel. Data parallel algorithms. *Communications of the CACM*, 29:1170–1183, December 1986.

[40] Daniel S. Hirschberg. Fast parallel sorting algorithms. *Communications of the ACM*, 21(8):657–661, 1978.

[41] Ching-Tien Ho and S. Lennart Johnsson. *Matrix Transposition on Boolean n-cube Configured Ensemble Architectures.* Technical Report YALEU/DCS/RR-494, Yale University, Dept. of Computer Science, September 1986.

[42] Ching-Tien Ho and S. Lennart Johnsson. On the embedding of arbitrary meshes in boolean cubes with expansion two dilation two. In *Int. Conf. on Parallel Processing*, pages 188–191, IEEE Computer Society, 1987. Report YALEU/DCS/RR-576.

[43] Ching-Tien Ho and S. Lennart Johnsson. *On the Embedding of Meshes in Boolean Cubes.* Technical Report YALEU/DCS/RR-, Yale University, Dept. of Computer Science, In preparation 1986.

[44] Roger W. Hockney. A fast direct solution of poisson's equation using fourier analysis. *J. ACM*, 12:95–113, 1965.

[45] Roger W. Hockney. The potential calculation and some applications. *Methods Comput. Phys.*, 9:135–211, 1970.

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

[47] Kai Hwang, editor. *Supercomputers: Design and Applications.* IEEE Computer Society, 1984.

[48] Kai Hwang and Faye A. Briggs, editors. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[49] S. Lennart Johnsson. Combining parallel and sequential sorting on a boolean n-cube. In *International Conference on Parallel Processing*, pages 444–448, IEEE Computer Society, 1984. Presented at the 1984 Conf. on Vector and Parallel Processors in Computational Science II.

[50] S. Lennart Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *Journal of Parallel and Distributed Computing*, 4(2):133–172, April 1987. (Report YALEU/DCS/RR-361, January 1985).

[51] S. Lennart Johnsson. A computational array for the qr-method. In Jr. P. Penfield, editor, *Proc., Conf. on Advanced Research in VLSI*, pages 123–129, Artech House, January 1982.

[52] S. Lennart Johnsson. *Computational Arrays for Band Matrix Equations.* Technical Report 4287:TR:81, Computer Science, California Institute of Technology, May 1981.

[53] S. Lennart Johnsson. *Data Permutations and Basic Linear Algebra Computations on Ensemble Architectures.* Technical Report YALEU/DCS/RR-367, Yale University, Dept. of Computer Science, February 1985.

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

[55] S. Lennart Johnsson. *Fast Banded Systems Solvers for Ensemble Architectures.* Technical Report YALEU/DCS/RR-379, Department of Computer Science, Yale University, March 1985.

[56] S. Lennart Johnsson. Fast pde solvers on fine and medium grain architectures. In *Advances in Computer Methods for Partial Differential Equations - VI*, pages 405–410, IMACS, 1987. YALEU/DCS/RR-583.

[57] S. Lennart Johnsson. Highly concurrent algorithms for solving linear systems of equations. In *Elliptic Problem Solving II*, Academic Press, 1983.

[58] S. Lennart Johnsson. *Odd-Even Cyclic Reduction on Ensemble Architectures and the Solution Tridiagonal Systems of Equations.* Technical Report YALE/DCS/RR-339, Department of Computer Science, Yale University, October 1984.

[59] S. Lennart Johnsson. Pipelined linear equation solvers and vlsi. In *Microelectronics '82*, pages 42–46, Institution of Electrical Engineers, Australia, May 1982.

[60] S. Lennart Johnsson. Solving narrow banded systems on ensemble architectures. *ACM TOMS*, 11(3):271–288, November 1985. (Report YALEU/DCS/RR-418, November 1984).

[61] S. Lennart Johnsson. *Solving Narrow Banded Systems on Ensemble Architectures.* Technical Report YALEU/DCS/RR-343, Dept. of Computer Science, Yale University, November 1984.

[62] S. Lennart Johnsson. Solving tridiagonal systems on ensemble architectures. *SIAM J. Sci. Stat. Comp.*, 8(3):354–392, May 1987. (Report YALEU/DCS/RR-436, November 1985).

[63] S. Lennart Johnsson. Vlsi algorithms for doolittle's, crout's and cholesky's methods. In *International Conference on Circuits and Computers 1982, ICCC82*, pages 372–377, IEEE, Computer Society, September 1982.

[64] S. Lennart Johnsson and Danny Cohen. An algebraic description of array implementations of fft algorithms. In *20th Allerton Conference on Communication, Control, and Computing*, Electrical Engineering, University of Illinois, Urbana/Champaign, 1982.

[65] S. Lennart Johnsson and Danny Cohen. *Mathematical Approach to Computational Networks for the Discrete Fourier Transform.* Technical Report, Department of Computer Science, Yale University, 1984.

[66] S. Lennart Johnsson and Ching-Tien Ho. Matrix multiplication on boolean cubes using generic communication primitives. In *Parallel Processing and Medium Scale Multiprocessors*, SIAM, 1987. (Presented at the ARMY workshop on Medium Scale Parallel Processing, Stanford University, January 1986, Report YALEU/DCS/RR-530, March 1987).

[67] S. Lennart Johnsson and Ching-Tien Ho. *Multiple tridiagonal systems, the Alternating Direction Method, and Boolean cube configured multiprocessors.* Technical Report YALEU/DCS/RR-532, Yale University, June 1987.

[68] S. Lennart Johnsson and Ching-Tien Ho. *Spanning Graphs for Optimum Broadcasting and Personalized Communication in Hypercubes.* Technical Report YALEU/DCS/RR-500, Yale University, Dept. of Computer Science, November 1986. To appear in IEEE Trans. Computers.

[69] S. Lennart Johnsson, Uri Weiser, Danny Cohen, and Al Davis. Towards a formal treatment of vlsi arrays. In *Proceedings of the Second Caltech Conference on VLSI*, pages 375 – 398, Caltech Computer Science Department, January 1981.

[70] C. Kamath and Ahmed H. Sameh. *The Preconditioned Conjugate Gradient Method on a Multiprocessor.* Technical Report ANL/MCS-TM-28, Argonne National Laboratories, Mathematics and Computer Science Division, 1984.

[71] M.G.H. Katevenis. *Reduced Instruction Set Computer Architectures for VLSI.* The MIT Press, 1985.

[72] D. Kershaw. *Solution of Single Tridiagonal Linear Systems and the Vectorization of the ICCG Algorithm on the CRAY-1*, pages 85–92. Academic Press, 1982.

[73] S.C. Knauer, J.H. O'Neill, and A. Huang. *Self-routing Switching Network*, pages 424–448. Addison-Wesley, 1985.

[74] Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley, 1973.

[75] P.M. Kogge and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers*, C-22(8):786–792, 1973.

[76] David J. Kuck. *The Structure of Computers and Computations.* John Wiley, 1978.

[77] David J. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys*, 9(1):29–59, 1977.

[78] David J. Kuck, Duncan H. Lawrie, R. Cytron, Ahmed Sameh, and Daniel D. Gajski. *The Architecture and the Programming of the Cedar System.* Technical Report, Laboratory for Advanced Supercomputers, Dept. of Computer Science, University of Illinois, August 1983.

[79] M. Kumar and Daniel S. Hirschberg. An efficient implementation of batcher's bitonic odd-even merge algorithm and its application in parallel sorting schemes. *IEEE Trans. Computers*, C-32(3):254–264, 1983.

[80] H.T. Kung and Charles E. Leiserson. *Algorithms for VLSI Processor Arrays*, pages 271–292. Addison-Wesley, 1980.

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

[82] Duncan H. Lawrie and Ahmed H. Sameh. The computational and communication complexity of a parallel banded system solver. *ACM TOMS*, 10(2):185–195, June 1984.

[83] Duncan H. Lawrie and C.R. Vora. The prime memory system for array access. *IEEE Trans. Computer*, C-31:1435–442, May 1982.

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

[85] F. Tom Leighton and Charles E. Leiserson. Wafer-scale integration of systolic arrays. *IEEE Trans. Comp.*, C-34(5):448–461, May 1985.

[86] Charles E. Leiserson. *Area-Efficient VLSI Computation.* MIT Press, 1982.

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

[88] Li.-J. Li and Benjamin W. Wah. The design of optimal systolic arrays. *IEEE Trans. Computers*, C-34:66–77, 1985.

[89] Peggy Li and Lennart Johnsson. The tree machine: an evaluation of program loading strategies. In *1983 International Conference on Parallel Processing*, pages 202 – 205, IEEE Computer Society, August 1983.

[90] Bjorn Lisper. *Description and Synthesis of Systolic Arrays.* Technical Report TRITA-NA-8318, The Royal Institute of Technology, Dept. of Numerical Analysis and Computing Sciences, 1983.

[91] Boris Lubachevsky and Debasis Mitra. *A Chaotic, Asynchronous Algorithm for Computing the Fixed Point of a Nonnegative Matrix of Unit Spectral Radius.* Technical Report, AT&T Bell Laboratories, 1984.

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

[93] Carver A. Mead and Lynn Conway. *Introduction to VLSI Systems.* Addison-Wesley, 1980.

[94] Willard L. Miranker. Hierarchical relaxation. *Computing*, 23:267–285, 1979.

[95] Willard L. Miranker and Andrew Winkler. Spacetime representations of computational structures. *Computing*, 32(2):93–114, 1984.

[96] Donald I. Moldovan. On the design of algorithms for vlsi systolic arrays. *Proc. IEEE*, 71(1):113–120, 1983.

[97] D. Nassimi and Sartaj Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Trans. Computers*, C-27(1):2 – 7, 1979.

[98] Susan T. O'Donnel, P. Geiger, and Martin H. Schultz. *Solving the Poisson Equation on the FPS-164*. Technical Report YALEU/DCS/RR-293, Research Center for Scientific Computing, Dept. of Computer Science, Yale University, November 1983.

[99] M.S. Paterson, W.L. Ruzzo, and Larry Snyder. Bounds on minimax edge length for complete binary trees. In *Proc. of the 13th Annual Symposium on the Theory of Computing*, pages 293–299, ACM, 1981.

[100] Gregory F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and J. Weiss. The ibm research parallel processor prototype (rp3); introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, IEEE Computer Society, 1985.

[101] Franco P. Preparata and J.E. Vuillemin. The cube connected cycles: a versatile network for parallel computation. In *Proc. Twentieth Annual IEEE Symposium on Foundations of Computer Science*, pages 140–147, 1979.

[102] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Annual Symposium on Computer Architecture*, pages 208–214, IEEE Computer Society, 1984.

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

[104] Abhiram Ranade. Interconnection networks and parallel memory organization for array processing. In *1985 International Conference on Parallel Processing*, IEEE Computer Society, 1985.

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

[106] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms.* Prentice Hall, 1977.

[107] E. Reiter and Gary Rodrigue. An incomplete cholesky factorization by a matrix partitioning algorithm. In *Elliptic Problem Solvers II*, pages 161–174, Academic Press, 1983.

[108] Arnold L. Rosenberg and Larry Snyder. Bounds on the costs of data encodings. *Mathematical Systems Theory*, 12:9–39, 1978.

[109] John Van Rosendale. Minimizing inner product dependencies in conjugate gradient iteration. In *Proc. of the 1983 International Conference on Parallel Processing*, pages 44–46, IEEE Computer Society, 1983.

[110] Yousef Saad. *Practical use of Polynomial Preconditionings for the Conjugate Gradient Method.* Technical Report YALEU/DCS/RR-282, Dept. of Computer Science, 1983.

[111] Yousef Saad and Ahmed H. Sameh. Iterative methods for the solution of elliptic differential equations on multiprocessors. In *Proc. of the CONPAR 81 Conference*, pages 395–411, Springer Verlag, 1981.

[112] Faisal Saied, Ching-Tien Ho, S. Lennart Johnsson, and Martin H. Schultz. Solving schroedinger's equation on the intel ipsc by the alternating direction method. In *Hypercube Multiprocessors 1987*, pages 680–691, SIAM, September 1986. Tech. report YALEU/DCS/RR-502, January 1987.

[113] Ahmed H. Sameh. A fast poisson solver for multiprocessors. In *Elliptic Problem Solvers II*, pages 175–186, Academic Press, 1984.

[114] Ahmed H. Sameh. Numerical parallel algorithms - a survey. In *High Speed Computer and Algorithm Organization*, pages 207–228, Academic Press, 1977.

[115] Ahmed H. Sameh and David J. Kuck. On stable parallel linear system solvers. *J. ACM*, 25(1):81–91, January 1978.

[116] J.T. Schwartz. Ultracomputers. *ACM Trans. on Programming Languages and Systems*, 2:484–521, 1980.

[117] Charles L. Seitz. Ensemble architectures for vlsi – a survey and taxonomy. In P. Penfield Jr., editor, *1982 Conf on Advanced Research in VLSI*, pages 130 – 135, Artech House, January 1982.

[118] Charles L. Seitz. Experiments with vlsi ensemble machines. *J. VLSI Comput. Syst.*, 1(3), 1984.

[119] M.C. Sejnowski, E.T. Upchurch, R.N. Kapur, D.P.S. Charlu, and G.J. Lipovski. An overview of the texas reconfigurable array computer. In *Proceedings, National Computer Conference*, pages 631–641, IEEE, 1980.

[120] M. Sekanina. On an ordering of the set of vertices of a connected graph. *Publ. of the Faculty of the Sciences of the Univ. of Brno*, 412():137–142, 1960.

[121] Larry Snyder. Introduction to the configurable highly parallel computer. *Computer*, 15(1):47–56, 1982.

[122] J. Stevens. *A Fast Fourier Transform Subroutine for the Illiac IV.* Technical Report, Center for Advanced Computation, Univ. of Illinois, 1971.

[123] Harold S. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. Computers*, C-20:153–161, 1971.

[124] Paul N. Swarztrauber. The methods of cyclic reduction, fourier analysis, and the facr algorithm for the discrete solution of poisson's equation on a rectangle. *SIAM Review*, 19:490–501, 1977.

[125] Clive Temperton. Direct methods for the solution of the discrete poisson equation: some comparisons. *J. of Computational Physics*, 31:1–20, 1979.

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

[127] C.D. Thompson. *Fourier Transforms in VLSI.* Technical Report UCB/ERL/ M80/51, Electronic Research Laboratory, UC Berkeley, 1980.

[128] C.D. Thompson and H.T. Kung. Sorting on a mesh-connected parallel computer. *CACM*, 20(4):263–271, 1977.

[129] Jeffrey D. Ullman. *Computational Aspects of VLSI.* Computer Sciences Press, 1984.

[130] E. Upfal. Efficient schemes for parallel computation. In *ACM Symposium on Principles of Distributed Computing*, pages 55–59, ACM, 1982.

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

[132] Angela Y. Wu. Embedding of tree networks in hypercubes. *Journal of Parallel and Distributed Computing*, 2(3):238–249, 1985.

[133] W.A. Wulf and C.G. Bell. C.mmp - a multi-mini-processor. In *AFIPS 72 FJCC*, pages 765–777, 1972.