Yale University
Department of Computer Science

Directions in High Performance Computations

S. Lennart Johnsson

YALEU/DCS/TR-574
June 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.

Directions in High Performance Computation

S. Lennart Johnsson
Department of Computer Science
Yale University
New Haven, CT 06520

June 1987

*Presented at American Statistical Association conference Computer Science and Statistics. 19th Symposium on the Interface.

Directions in High Performance Computation

S. Lennart Johnsson
Department of Computer Science
Yale University
New Haven[1]

Abstract

Evolving technology is driving high performance computer architecture towards highly concurrent systems. We review some of the elements of the technology influencing this direction, and discuss some of the architectural, algorithmic, and programming system consequences of this change. Finally, we briefly describe some of the essential features of the Connection Machine, a commercially available computer with an architecture and programming system that includes several of the features we expect to find in many high performance architectures in the future.

1 Introduction

There are several technological facts that drive the design of high performance computers in the direction of highly concurrent systems. Traditionally, high performance systems have been built using the fastest technology available, such as bipolar technology. However, these architectures are expensive, and they consume a large amount of power. The price has exceeded that of a standard mainframe by at least an order of magnitude, and the peak performance has been at least two orders of magnitude higher than that of a mainframe. Performance measured over complete application codes typically falls in the range of 10% - 30% of the peak performance.

Several reasons have been noted for the large descrepancy between peak and average performance. For many years, processors in high performance architectures have been faster than the storage units. Fast processors have storage that is interleaved. Balancing the storage bandwidth with the computational rate of a single processor typically requires 8-64 storage banks. Including a cache in the architecture reduces the need for a primary storage bandwidth, if the algorithms exhibit locality. Register oriented architectures have a significantly lower bandwidth to storage than to registers. Hence, if the operations have operands that reside in primary storage, then the performance is often significantly less than peak performance. With operands in storage the situation may be further aggravated, if successive requests for storage operations are directed to the same storage unit. In this case, performance is determined by the bank bandwidth, not the full storage bandwidth. A stride that equals a multiple of the number of banks is particularly unfortunate. In cache based architectures, a small stride, preferably one with respect to the machine data structure (linear ordering) is important to reduce the number of cache misses. Bank

[1]Currently on leave at Thinking Machines Corp.

conflicts and cache misses may reduce the performance by an order of magnitude. All current high performance architectures have pipelined functional units, and are known as vector architectures. Poor utilization of the pipeline contributes to the descrepancy between peak and average performance. Another reason that average performance typically is lower than the peak, even if the algorithm does allow full utilization of the architecture, is failure of the compiler to recognize these characteristics. For instance, many compilers fail to keep variables in registers between successive iterations of loops. "Loop unrolling" can increase the performance by more than a factor of two by keeping temporary variables in registers.

Vector architectures have clearly been very successful; but not without a significant investment in algorithm development, and software technology in the form of vectorizing compilers.

For continued growth in performance it is no longer possible to rely on speed improvements of standard technologies, such as bipolar technology. For highly integrated MOS technologies, a reduction in switching speeds by a factor of 5 to 10 is predicted before the limits of the technology are reached. One to two orders of magnitude will allow more parts to fit on the same die before the laws of physics may prevent further reduction in feature sizes. MOS technology is a replication technology, and parts (chips) are cheap to produce in quantities. Hence, VLSI architectures should consist of a large number of identical chips; or a few different chips, preferably easily parametrizable.

Computers are built from two elementary parts: transistors and wires. For storage one transistor cells are used for dynamic random access memory (RAM), and three transistor cells are used for static RAM. A high rate of computation requires a high storage bandwidth as well as a high processor bandwidth. Depending upon the complexity of the processor and the size of a storage unit, the processor may be slower or faster than the storage unit. It is possible for a register to be clocked at a higher rate than a chip packed with storage cells. A straight column organization results in long wires, which in MOS technologies requires a large driving power and time. Structuring storage itself [17] can improve the speed of storage compared to the column approach, however, a large amount of storage is still slower than a smaller amount. Similarly, for synchronous designs, processors with wide datapaths and large controllers are slower than processors with narrow paths, or very simple control logic. In most designs the processor tends to be faster, but not exceedingly so. Hence, to first order it is reasonable to expect VLSI systems to have approximately an equal number of storage units and processor units. The absolute number being determined by the size of the state that can be operated upon concurrently, in conjunction with the desired performance. High performance systems will have a large number of units, and the Connection Machine is a good example of such a design. The storage bandwidth of model CM-I is 32 Gbytes/sec at a clock frequency of 4 MHz compared to approximately 4 Gbytes/sec for the CRAY-2.

With a large number of processing and storage units, the interconnection network becomes a critical component of the architecture. This is also a major expense. Other

	Chip $25mm^2$ $100M\lambda^2$	Chip $25mm^2$ $200M\lambda^2$	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

important components are: programming environment, language, and compiler. We will highlight some of the issues in choosing an interconnection network, and in defining a suitable programming language amenable to compilation for highly parallel architectures.

2 VLSI technology

A rule of thumb is that a reliably working system should contain at most 20.000 integrated circuits. Hence, the simpler the units, the more of them are possible in a system. If the complexity (area) of a unit is plotted against the number of units for existing architectures, most of them fall close to a hyperbola [22]. With decreased feature sizes a given area corresponds to more storage, more processors, processors with additional hardware features or wider data paths, and/or some combination thereof. Tables 1, 2, and 3 illustrate in a simple way the capability of the MOS technology at feature sizes of one micron and a quarter micron. These predictions are based on existing designs for which a 1-bit dynamic RAM cell requires an area of $100\lambda^2$; and a 1-bit static RAM cell requires an area of $400\lambda^2$, where 2λ is the minimum feature size. Processor estimates are based on the Caltech Mosaic 16-bit processor, which is about $2.5M\lambda^2$ excluding the pad frame [16], and the RISC and MIPS 32-bit processors, which are about $12M\lambda^2$ excluding pad frame [3,4,14]. Table 3 is based on the assumption that half of the die area is devoted to storage, half to the processor, and that half of the dies are functional on a 4 inch wafer. Predictions for the Cosmic Cube are based on an estimated processor area of $140M\lambda^2$ [23], and measured performance on existing hardware. For bit-serial communication and simple switching elements for an Ω-network or a Boolean cube, the design of Knauer et. al. [15] extrapolated with respect to the scaling of such networks, and scaling of the technology, indicates that these area estimates should not be off by more than a small constant factor (2 - 3) after the interconnection is included.

Communication is the overriding concern in VLSI technologies. In MOS technologies the actual transistor area is often only $2\% - 3\%$ of the total area. The wiring area if measured in terms of wiring between functional blocks or cells, occupies 50% - 70% of the area. Wires limit the maximum rate of change of the state, since they reduce the amount

	Chip $25mm^2$ $1600M\lambda^2$	Chip $25mm^2$ $3200M\lambda^2$	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

	Capacitive Model	Resistive Model
Wire delay	$O(L)$	$O(L^2)$
Opt. wire delay	$O(logL)$	$O(L)$
Scaling: $L \rightarrow \frac{L}{\alpha}$,	$O(\frac{L}{\alpha})$	$O(L)$
opt. delay	$O(\frac{L}{\alpha})$	$O(\frac{L}{\sqrt{\alpha}})$
L fixed, other dim. scaled	$O(L)$	$O(L^2\alpha^2)$
opt. delay	$O(\frac{1}{\alpha}log(\alpha L)$	$O(L\sqrt{\alpha})$

Table 4: Wire delays in MOS technologies under scaling of feature sizes with a factor α

of logic that fits in a given area, and because there is a delay associated with the transfer of information across a wire. Moreover, the switching time of a transistor is linearly reduced with the feature size under currently accepted scaling models, but the wire delay does not scale well [17]. MOS technologies can be viewed as charge transfer technologies. The time to transfer a given charge is proportional to the size of the charge, which is determined by the capacitance. The rate of change is determined by the driving transistor and the properties of the wire. In a *capacitive model* for charge transfer, the wire resistance is ignored, and the delay is linear in the wire length or logarithmic with an optimized driver [17].

On the other hand, in a *resistive model*, wire resistance is included. Here, the delay is proportional to the square of the wire length. It is reduced to linear in the wire length when signal restoration exists along the wire.

Table 4 shows the qualitative behavior under scaling of features by a factor α such that aspect ratios are preserved. It is also assumed that the voltage is scaled to preserve the electric field in the gate region.

3 Network Characteristics

Several interprocessor connection networks have been proposed and analysed with respect to area or volume requirements, as well as with respect to maximum wire lengths. Another important characteristic is partitioning properties, in particular the bandwidth requirements at the boundary of a partition. In tables 5 and 6, we summarize some of these properties.

The diameter gives a lower bound for global communication. In most non-degenerate computations, at least one global communication is necessary. For instance, the solution of any irreducible system of equations depends on all the matrix elements and all the values of the right hand side. With the matrix coefficients distributed throughout the system, global communication is required. Similarly, sorting implies a total ordering with

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	$2(2^k - 2^{k/2})$
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	$\approx (1.5)2^k$

Table 5: Topological properties of some common networks

Configuration	Nodes	Edge len.	Area	Pin Count
Linear Array	2^k	$O(1)$	$O(2^k)$	2
2-d mesh	2^k	$O(1)$	$O(2^k)$	$4\sqrt{M}$
Binary tree	$2^k - 1$	$O(2^{k/2}/k)$	$O(2^k) - O(2^k \cdot k)$	4
Boolean cube	2^k	$O(2^{\frac{k}{2}})$	$O(2^{2k})$	$M(k - logM)$
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 6: Layout properties of some common networks

keys distributed over all nodes. This creates a situation where global communication is necessary. Hence, whether the lower bound given by the diameter is also an upper bound depends on the computation and data allocation. In order to achieve an upper bound of the same order as the lower bound, the number of global communications has to be bounded; and there must not be many conflicts. For many algorithms in numerical linear algebra and sorting, data structures and computational algorithms, deterministic routing algorithms and scheduling disciplines can be found so that few or no conflicts occur in the communication on some of the networks. For more difficult communications in higher order networks such as the Boolean cube and Cube Connected Cycles network, randomization of communication guarantees with high probability an upper bound of the same order as the lower bound [26,25,20].

4 Network utility

The utility of a network can be measured in many ways. One way is to attempt to classify the communication needs of frequently used or otherwise important computations, and determine how each such class can be implemented on the processing ensemble. *Universal* networks can simulate other networks by a slowdown of a factor $logN$ for N node networks. Thus, the problem of finding a data allocation for a given algorithm that yields a communication efficient implementation for a specific network can 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. Dilation of edges can cause a corresponding decrease in throughput (the time between successive computations of a given kind), due to long range communication, or communication conflicts due to edge sharing between different paths. An increase in latency (additional time for completion of the first computation in a set), may also do the same. Expansion is the ratio of the number of nodes in the host and guest graphs.

Many computations can be performed with total complexity, i.e. arithmetic/logic **and** communication complexity of the same order as the former on two or multidimensional lattices. Examples of such computations consist of many linear algebra operations such as matrix multiplication, LU-factorization with a variety of algorithms for dense and banded matrices, relaxation on grids for partial differential equations, and lattice calculations in physics. The purpose of these computations is to determine the global behavior. So called fast algorithms on sequential architectures use fewer operations than what is typically required by some form of relaxation method. Many of these algorithms are based on the divide-and-conquer paradigm, where parallel versions run in logarithmic time – ignoring communication. Examples, are inner products such as the Fast Fourier transform (FFT), matrix transpose, parallel prefix operations, and tridiagonal system solvers. With the exception of the tridiagonal system solvers, the familiar butterfly network is ideal for these

computations. Such networks are also good sorting networks because they perfectly support bitonic sort. The communications network that supports tridiagonal system solvers is the same as that for multigrid methods on a grid graph. It is a reduction form of a data manipulator network. Parallel cyclic reduction uses a full data manipulator network.

Broadcasting, copy-scan, and reduction operations are common. The most common reduction operations are sum/subtract and max/min of a set. Sum-scan is useful in the computation of inner products. Max-scan is useful in sorting and searching, and also LU-decomposition with partial pivoting. Scan operations can apply to either a complete set, or a subset. If it applies only to a subset, and the subset is easily defined, then the scan operation is often called a segmented scan, since it applies to a segment of the address space. Scans require spanning trees. Other forms of tree operations are split/merge operations. In the split operation, a set is divided into subsets, and each such subset communicates with a distinct processor. The merge operation is the inverse of the split. Single spanning trees can be used, but if the data volume is large, then other forms of spanning graphs may better utilize the bandwidth of the network. For instance, multiple edge-disjoint spanning trees can be used for broadcasting data in a time proportional to the lower bound on a Boolean cube [11]. For a small data volume, propagation time is more important and minimum height spanning trees are the most desirable ones. Yet, at several other reduction/broadcasting operations may take place concurrently, as in computing histograms [7]. The interleaving of these trees will determine the performance of the network [11].

Scan operations are particularly efficient on bit-serial pipelined architectures. A global summation requires a time equal to the tree height plus the number of bits in the operands, which for a reasonable machine and operand sizes is only moderatly slower than a number of independent local operations. Indeed, the summation of a large collection of numbers can be made in almost the same time as the addition of two numbers.

There are also many problems where data interaction cannot easily be described in terms of a few parameters, for which general pointer structures are required. One way of measuring the capability of a network for such data structures is to consider arbitrary permutations. In summary, a good network should be able to support communication in the form of multidimensional grids, butterfly networks, datamanipulator networks, (multiple) trees, and arbitrary permutations without a significant slowdown.

The Boolean cube and the related Cube Connected Cycles network are exceptionally versatile host graphs. Multi-dimensional arrays can be embedded in the Boolean cube with dilation 1 and expansion 1, if the number of grid points in each dimension of the array is a power of 2. A binary-reflected Gray [21] code offers such an embedding [12,8]. Many other grids can be embedded with minimum expansion and a dilation of 2 [6]. 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 [24] 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, but 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. It is also possible to embed a data manipulator network preserving proximity, but not adjacency. By using the binary-reflected Gray code, the dilation is guaranteed to be less than 2 [9]. It is also possible to construct spanning graphs that perfectly use the entire bandwidth of the Boolean cube [11].

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 [9]. The need for data reallocations in order to minimize the complexity of an algorithm decreases with the ability of the network to effectively support different access patterns to a data structure. A static embedding of a data structure can support many types of access schemes without communication penalty, reducing the need for data reallocations.

5 Programming considerations

It is desirable that a programmer sees as little as possible of the underlying architecture with respect to portability of the code. True portability requires that no significant loss of performance occurs due to the use of such a programming concept and associated language. An ideal notation allows a programmer to conveniently express computations of interest, while being conceptually focused on the problem domain. This often means that the programmer defines a few abstract objects, some of which have structure associated with them, then expresses the operations on these abstract objects with the semantics clearly defined. It is for instance natural for many lattice like computations to let the lattice be an abstract data structure defined in a Cartesian coordinate space with a given number of points in each dimension associating variables with lattice sites. Periodic boundary conditions are easily defined by defining the lattice as periodic. In the case of the solution of partial differential equations, it is necessary to compute discrete approximations to the derivatives, which can either be made completely local, or with communication according to the stencil being used. With the exception of the boundary, the same stencil is typically used for the entire domain. Hence, these computations are homogeneous. For iterative solution processes, some form of coloring scheme is often required. For a given coloring scheme the color of a lattice point is given by its coordinates. Concurrency is again high. The purpose of coloring is to sequence operations appropriately, and a sweep over the lattice requires as many sequential steps as there are colors. In each step a processor holding a lattice point interrogates the processors storing adjacent points of different color. Nat-

urally, all lattice interactions are expressed as relative operations. A good programming notation should support such a conceptual framework.

There are two types of communication operations: get-put and/or exchange. For higher dimensional networks such as Boolean cubes, butterfly networks, and Cube Connected Cycles networks, so called normal algorithms typically work by sequencing through dimensions. For instance, the FFT is easily expressed as performing a sequence of exchanges in dimensions 0 through $n-1$ ($n = log_2 N$).

In these examples there is a relation between the nodes of the logic data structures. In the two dimensional lattice a node is to the east, west, south or north of neighboring nodes, or in the Boolean cube in a 0 or a 1 subcube. This ordering is used in expressing the computations. Coloring divides the nodes into sets, with no particular ordering between the elements of a set, however, there is a definite relationship between the members of different sets. In the type of computations outlined above the regularity of the relationship can be used to carry out a mapping onto the processing network so that communication is minimized. Mapping is fairly easily computable in many cases, so that the map can be built into the compilation process. Hence, releaving the programmer of having to deal with architectural details with no loss in efficiency. For instance, if the computations are expressed as computations on a multidimensional lattice, then a possible mapping onto another lower dimensional lattice can be done through projection. This is done simply by identifying dimensions of the logic mesh with dimensions of the processor mesh; and/or by identifying all the nodes that have the same index for the processor coordinates. If the processor mesh has fewer nodes in a given dimension than the logic mesh, then identification can be made on high order bits resulting in a *cyclic* mapping, or low order bits, resulting in a *consecutive* mapping [8]. For networks that do not form a subdomain of the domain defined by the logic structure, it may sometimes be convenient to map the logic structure onto an intermediate graph, which has a known embedding in the network. In complex cases, finding an optimum map may be a NP-hard problem. An arbitrary map, or randomized mapping, are often the only practical choices.

6 Compilation - Automated Mapping

In the kind of programming model outlined above, the data structures are either one or multidimensional arrays with communication between adjacent points in the lattice, or some local neighborhood in the form of an FFT butterfly network, data manipulator network, or one or multiple trees. In addition to the conventional arithmetic/logic operators, we have also introduced scans and split/merge operators. The indicated programming model has only abstract objects natural with respect to the computational problem at hand, with no direct relation to the architecture of a buildable machine. It pushes most of the difficult implementation related issues into the compiler. Correspondence between the logic model and the architecture is traditionally established in the programming process for languages such as Fortran. Individual elements of an object are often conveniently

identified as a point in a one, two, or multi-dimensional space. Sometimes alternate representations are equally plausible and efficient. At other times, different representations result in different ease, or cost of implementation. For instance, the elements of a dense matrix are naturally identified by their row and column indices. For computations on lattices such as Ising calculations and quantum electrodynamics, variables are associated with lattice points. In the solution of partial differential equations by finite difference methods, the continuous space is approximated by regular lattices. Such lattices can also be used for some types of elements in finite element computations.

Most computations require a sequence of operations on the same data element, or derivatives thereof. Hence, time is another dimension required to define an algorithm, and there are in effect a total of $N \times T$ lattice points for a computation that requires time T on N lattice points. The nodes in this lattice are encoded by $(x_{n-1} \ldots x_0 | y_{m-1} \ldots y_0)$, where $n = log_2 N$ and $m = log_2 T$. The embedding of a lattice in the processing network in the data parallel case, where a processing element can be identified with each data element, should be made with the objective of minimizing communication required by the algorithm. For the sake of simplicity, we assume that there is one variable per lattice point. Communication occurs in space and time. In a relaxation like scheme, the communication pattern is identical for all iterations. If the intuitive mapping is made of the $N \times T$ lattice to the processing network, i.e., nodes with the same x value for different y are mapped to the same processor node, then a mapping of the original logic lattice that preserves adjacency, or at least proximity, also minimizes communication throughout the algorithm. The identification results in efficient use of the resources. With resources corresponding to an $N \times T$ grid, efficiency is of order $O(\frac{1}{T})$ for one instance of the computation. For multiple instances pipelining can be used to increase the utilization for a processor network with $N \times T$ processors.

In most cases the graph defining the algorithm has one more dimension than the data structure. In the case of relaxation, time may take on arbitrary values. In other instances the value may be directly related to the number of elements in one or more dimensions of the data structure, as in matrix multiplication. The standard algorithm for multiplication of two dense matrices has three nested loops, two of which sequence through the elements of the data structure. Different projections result in different data movements, required initial data structures and generated final data structures [1]. In the case of matrix multiplication, not all variables are defined in every point of the three-dimensional lattice, and a first step in the mapping is to define all variables at all lattice points [18,19]. This requires some form of copy-scan in the case of matrix multiplication. Different ways of performing this scan operation will yield different algorithms.

For matrices that are not dense, the row-column organization may not be the representation of choice. For instance, banded matrices are often represented most conveniently and compactly by one index for the diagonal, and another along the diagonals. The representation is still two-dimensional. Sparse matrices are most often represented as a list structure of some form. The Cartesian representation is convenient for mapping onto a lower dimensional array, or any network that efficiently simulates two-dimensional arrays.

For dense matrices there also exist other representations that are natural for recursive algorithms, such as the one by Dekel et al. [2]. Recursive algorithms often use the divide-and-conquer paradigm, and the algorithm steps correspond to operations on different bits in the binary encoding. Mapping to a Boolean cube becomes straightforward, since bit positions correspond to different dimensions in the cube.

Another example of a divide-and-conquer algorithm is the familiar radix-2 FFT. The communication for different values of T is over distances that correspond to different powers of 2, i.e., over growing distances in the linear ordering. However, in the binary encoding communication is between nearest neighbors (Hamming distance 1). Identifying corresponding nodes for all values of T, yields a Boolean cube topology, and good resource utilization for the computation of a single FFT. The execution time is $T = logN$. It is also possible to identify nodes for all values of N for each T to achieve a linear array of $logN$ units operating in time $T = N$ [13,10]. Note that in the Boolean cube case there is only one loop-level, however, in the linear array case there are two nested loops. For a uniprocessor case there are three nested loops.

In the above-mentioned examples it has been assumed that the number of processors is equal to the number of points in the data structure. In many instances each processor has an appreciable amount of storage, and it is possible to allocate more than one node of the logic data structure to each processor, should that be required. The notion of *virtual processors* is often used in such a case [5]. Typically, virtual processors correspond to some designated bits of the address space. If virtual processors correspond to the low order bits, then the mapping is *consecutive*. Virtual processors on the high order bits correspond to *cyclic* partitioning [8]. For some algorithms, either form yields the same processor utilization. For others one may be better than the other. For instance, cyclic partitioning is preferable for LU-decomposition, but consecutive for the solution of tridiagonal systems of equations.

It is not always the case that a mapping can be computed at compile time. Many efficient sequential algorithms are strongly data dependent, or adaptive, and run-time mapping or load balancing is necessary.

7 The Connection Machine

The Connection Machine is a data parallel architecture. It has $64k$ processors organized with 16 processors to a chip, and the chips are interconnected in the form of a Boolean cube. The on-chip processors are interconnected with a network effectively offering full interconnect. The interconnection network can efficiently emulate multidimensional lattices, FFT butterfly networks, data manipulator networks, and a variety of trees. For permutations and routing in arbitrary patterns, the Connection Machine is equipped with a router which routes according to the shortest path between source and destination. The router has several algorithms for resolving contention for communication channels. The bandwidth to local storage is 32 Gbytes/sec for model CM-I, and about 50 Gbytes/sec

for model CM-II. The bandwidth for communication in a two-dimensional lattice is about a factor of 5 less, and the bandwidth for random permutations yet another factor of 5 less, approximately. Model CM-II has a total of 512 Mbytes of storage distributed evenly among the processors. This model can also be equipped with hardware for floating-point operations.

The architecture is a bit-serial, with the router performing pipelining at the bit-level. Routing time is proportional to the number of address bits plus the message length. Hence, for messages of a few bytes the overhead is small. Moreover, every location is reachable in the same amount of time. Variations in access time is entirely due to contention for communication channels. Random permutations results in contention, and a factor of 5 lower bandwidth than contention free communication. The router will in most cases use the available communication bandwidth as effectively as possible. The Connection Machine can be viewed as a shared memory architecture with little or no loss in arithmetic/logic efficiency.

Scans are very efficient on the Connection Machine, given that it is a bit-serial, pipelined architecture. Global scans are almost as efficient as local operations, since there is no contention. Some contention may occur for multiple independent segmented scans. For instance, if the machine is configured as a two-dimensional lattice with a square sub-lattice per chip and a column shall be copied to all other columns. Then, each row operation can be viewed as a segmented copy-scan and all scans will use the same channels between chips.

Global and segmented scans are supported in the Connection Machine instruction set. Relative addressing in multi-dimensional lattices is supported as well. Connection Machine programming languages currently are: *Lisp and a parallel version of C called C*. These languages are extensions of the familiar Lisp and C languages. The most essential extensions are the existence of a parallel data type, and the operations thereupon. Scans are among the operators included in the extensions. Members of a set of elements forming a parallel variable are operated upon concurrently by one instruction. No enumeration of the elements is required, and one or several loop levels disappear from the code, compared to languages not supporting set operations. Thus, the code becomes more compact, simpler, and easier to debug. One source of errors has vanished.

Given that the programming languages for the Connection Machine are extensions of conventional languages, debugging tools and processes are similar to those for conventional architectures. The Connection Machine is a synchronous machine. The synchronization is handled by hardware. A user and/or programmer does not need to be concerned with this issue. Communication is implicit in references and only affects the time for a reference, or scan operation. The Connection Machine has a host processor that performs scalar operations and instruction decoding. Parallel variables are allocated in the Connection Machine and instructions with such variables as operands are sent to the Connection Machine, where another level of decoding and the control of the execution takes place.

The Connection Machine performance, like most other architectures, benefits from

Algorithm	Arithmetic op's/ element	Arithmetic ops/ Elem. comm.
Matrix-vector mpy	2	$\sqrt{M}$
Matrix-matrix mpy	$\frac{2}{3}\sqrt{N}$	$\sqrt{M}$
Relaxation 5-point stencil	1.6/iter.	$\sqrt{M}$
Radix-2 FFT	$1.25log_2N$	$1.25log_2(M/2)$
Sorting	$O(log_2N)$ - $O(log_2^2N)$	

Table 7: Number of operations per data element for a few operations on N elements in a machine with local storage M

locality of reference. Degradation is due to bandwidth limitations at the chip boundaries. Performance is measured as arithmetic or logic operations per unit time, benefits from increased granularity. For many operations arithmetic/logic operations grow faster than the communication as a function of the number of local data elements, as shown in Table 7. For matrix multiplication the variation for CM-I is in the range 60% to 90% of peak performance, whereas for matrix vector multiplication the variation is considerably larger.

8 Applications Experience

Many problems in the computational sciences have moderate to large data sets. The number of variables in many of the scientifically interesting or important engineering problems are in the range several hundred thousands to a billion. For instance, finite element computations in the geophysics, automotive, and aerospace industries are routinely made with several hundred thousand variables. These are limited by storage as well as speed of available computers. Three dimensional models have essentially been beyond reach. In physics, some of the outstanding problems require computation on lattices of ten to a few hundred million lattice sites with a few variables per site. Many of the problems that have their origin in the physical sciences are modeled from local behavior. Partial differential equations are examples thereof. Global behavior is deduced from local rules. The locality of the rules provides an excellent basis for data parallel computing. Computer architectures that allow a large part of the state to change concurrently have the potential for offering maximum performance for a given amount of hardware. Locality of the data relationships, typical in many engineering and scientific problems, makes it possible to design such systems without excessive demands on the communication system – the most expensive resource in highly concurrent systems.

We have implemented a variety of algorithms with good results both with respect to programmer productivity and performance. Examples of basic linear algebra algorithms

are matrix multiplication, matrix-vector operations, LU-decomposition, matrix transposition, the solution of banded systems of equations, sparse matrix multiplication and the solution of sparse systems of equations, FFT's, iterative equation solvers, tridiagonal system solvers, and multigrid methods. These algorithms use a variety of data structures and communication patterns. For most of these codes, the efficiency on the CM-I ranges between 50% and 90%. We have also implemented partial differential equation solvers based on the Alternating Direction Method and explicit methods with very good efficiency. Recently, a simple finite element code has also been brought up on the Connection Machine. The effort in bringing these codes up on the machine is in the order of days to a few weeks for each routine or application.

9 Summary

Technology drives the architecture of high performance computers toward highly concurrent systems. Communication becomes a fundamental issue both in devising algorithms, architectures, and in compilation. For many applications in which high performance is demanded, the number of operations per point is limited, however, the demand for high performance is due to large data sets. Describing computations on large sets of data requires suitable abstractions in order to conceptually manage the complexity. An architecture such as the Connection Machine is scalable both with respect to technology and size (number of processors), and it is an excellent vehicle for studying algorithms for future high performance systems. It has also proven to be a useful tool in developing languages, and compilation techniques for such systems. The CM-II indeed demonstrates the power of the architecture, and users of high performance computers have a new alternative with greater potential.

References

[1] Ajit Agrawal and S. Lennart Johnsson. *A Uniform representation of Matrix Multiplication Algorithms.* Technical Report , Department of Computer Science, Yale University, 1987. in preparation.

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

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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).

[9] 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.

[10] S. Lennart Johnsson and Danny Cohen. Computational arrays for the discrete fourier transform. In *Proceedings of the Twenty-Second Computer Society International Conference*, COMPCON '81, February 1981.

[11] 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.

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

[13] 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.

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

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

[16] 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.

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

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

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

[20] Abhiram Ranade. *Constrained Randomization For Parallel Communication.* Technical Report YALEU/CSD/RR-, Yale University, Dept. of Computer Science, In preparation 1986.

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

[22] 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.

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

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

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

[26] 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.