Communications Architecture in the Connection Machine® System

C. Stanfill

Thinking Machines Corporation

Technical Report Series

HA87-3

Communications Architecture in the Connection Machine™ System

Craig Stanfill
Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts, 01242
USA

March 17, 1987

ABSTRACT

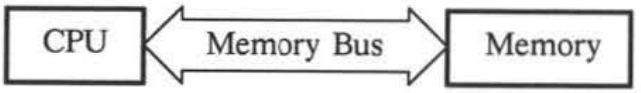
To build computers which are fundamentally more powerful than those currently available, it is necessary to build systems incorporating large numbers of processors. At this point, communications architecture becomes the dominant problem facing the machine designer, for if the processors cannot communicate they cannot cooperate in solving a problem. Many different architectures have been proposed, but most scale poorly, so that the number of processors they can support is limited either by interconnect bandwidth or by escalating hardware costs. Other architectures scale well, but are limited to specialized applications. However, the family of architectures including omega networks, hypercubes, and fat-trees scales well both in terms of hardware required and in terms of communications time. One such architecture, the hypercube, was chosen as the basis of a new parallel computer, the Connection Machine System.

1. Computer Architecture as Communications Architecture

The recent revolution in micro-electronics has opened up an opportunity to build computers which are many times more powerful than their predecessors. By linking together tens of thousands of inexpensive microprocessors, it is possible to build machines which can perform over 10^9 operations per second, compared with the 10^7 which is typical of large uniprocessor machines. The key to realizing this potential lies in building a communication system which allows information to be transferred from one processor to another. For this reason, communications architecture has become an inseparable part of computer architecture.

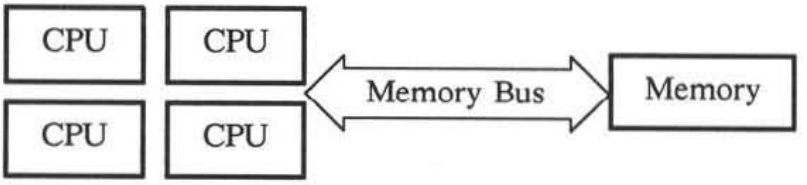
When electronic computers were developed in the 40's and 50's, the bulk of the system cost was in the central processing unit and primary storage. The transfer of informa-

tion between the two was accomplished by running a modest number of wires (the *memory bus*) from one part of the machine to another. In each computational cycle, the processor may read a value from memory or write a value to memory. This is the well known von Neumann architecture.



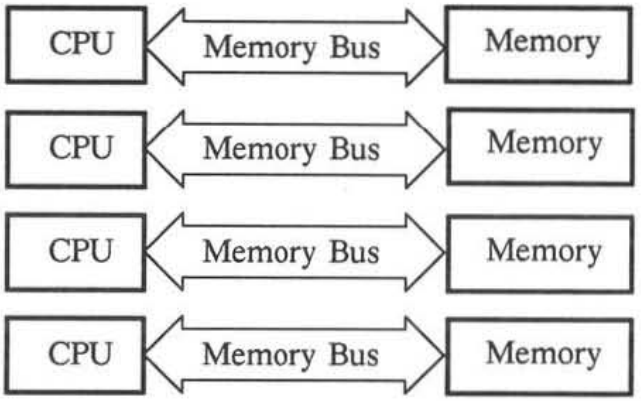
The von Neumann Architecture

When trying to build more powerful computers, the simple memory bus becomes a fundamental limitation. Suppose the single CPU in the von Neumann machine is replaced with four CPUs:



A Four-Processor von Neumann Machine

The resulting four-processor machine is fundamentally no more powerful than the single-processor machine because, although the processing component of the system is now four times more powerful, the communications component has not changed. A possible remedy to this problem is using four busses and four memories.



Adding Busses and Memories

This design has four times the computational power of the original von Neumann machine, but is no longer a single machine. In order to cooperate in performing a computation, the CPUs must be able to communicate not only with their own memories but among themselves. When such a communication system has been added, the result is no longer a von Neumann machine, but something completely new: a parallel computer. The

architecture of communications systems is therefore an important issue in the design of parallel computers.

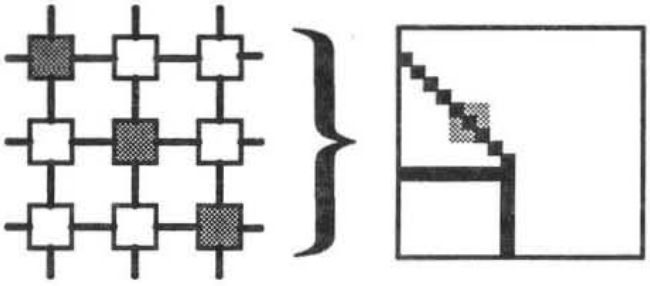
2. Requirements for a Communications Architecture

Instantaneous communications with infinite bandwidth among arbitrary sets of processors is obviously desirable and, just as obviously, impossible. This section will discuss the key tradeoffs which must be made in the communications architecture and how various compromises are likely to affect overall system utility. This discussion will be based on work by Hillis [1].

2.1. Communication Patterns

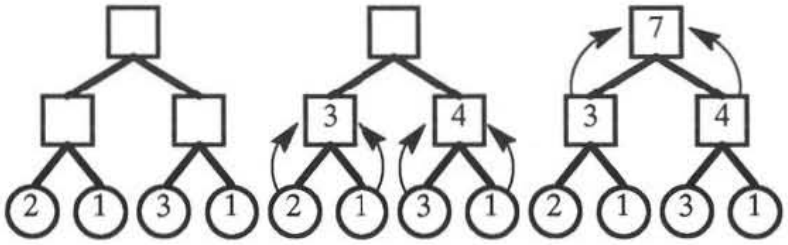
The first step is to characterize the communication patterns of the algorithms to be run on a parallel computer. Relevant features include locality, regularity, fan in, and fan out. Note that this discussion pertains to patterns of data movement for abstract algorithms, and not to any particular communication architecture. Any of the algorithms discussed below could be run on a serial von Neumann computer, or on a variety of parallel machines.

Many algorithms have highly localized communication patterns: if the problem is embedded in a two- or three-dimensional grid, then a given computation will only need to access data in a small region of the grid. Algorithms for computer vision, such as line detectors, are particularly likely to exhibit this sort of behavior. For example, if an image is represented as a grid of dots (pixels), lines may be detected by finding all dark pixels which have exactly two dark neighbors.



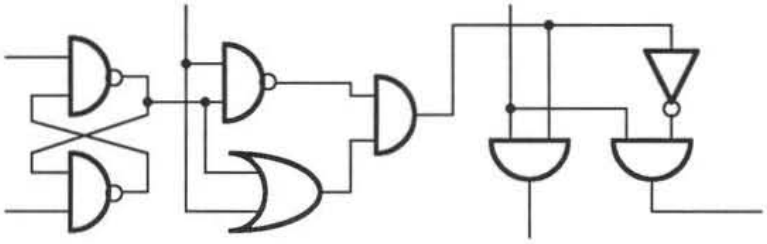
Local Communications on a Grid

Many algorithms generate very regular communication patterns, even if that communication is not strictly local. A good example is an algorithm which finds the total of a set of numbers represented as a binary tree. In this algorithm, there are two types of structures: *leaves,* which contain numbers, and *nodes,* which can refer to either two leaves or to two other nodes. To total up the values of the leaves, the nodes of the tree repeatedly ask each of their two children for their total, until the process terminates at the root of the tree.



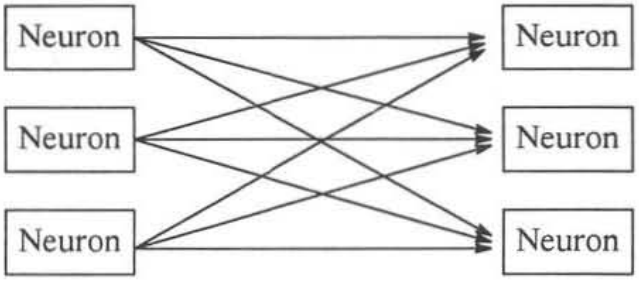
Regular Communications

In the general case, an algorithm may exhibit neither local nor regular communication patterns. For example, the components making up a silicon chip can be connected in an arbitrary pattern. To simulate the behavior of such a chip, a computer must allow information such as voltage and current to be moved around in arbitrary patterns. The difficulty of implementing a communications subsystem which can handle these arbitrary patterns is the primary reason why the widespread use of parallel computers has been slow in coming.



A Problem with Arbitrary Connectivity

Communications patterns can also differ in the number of destinations to receive a given message (fan-out) and the number of destinations attempting to send a message the same place (fan-in). The simplest case is when both fan-in and fan-out are 1 (i.e. one-to-one). The most difficult case is when fan-in and fan-out are both high (i.e. each sender is broadcasting to many destinations, and each receiver gets many incoming messages. An example of this is a neural network simulation, where an individual neuron has a large number (several hunderds) of inputs and outputs.



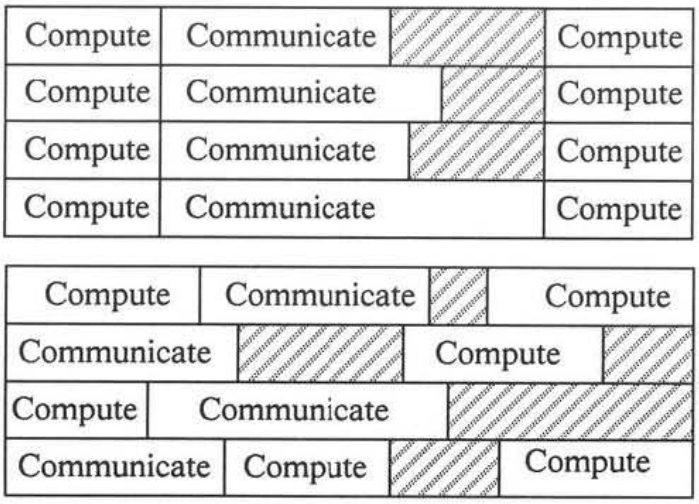
Combined Fan-in and Fan-out

In summary, algorithms can be characterized by their communications pattern. Some algorithms use only local communications, while other communicate in very regular patterns. In many cases, however, an algorithm will require the unrestricted flow of information between different structures taking part in the computation. Further complicating the situation, communications are not always one–to–one; there are many cases where the pattern is many–to–one (fan–out), one–to–many (fan–in), or even many–to–many (combined fan–in and fan–out). If the communications architecture of a parallel computer does not efficiently support all these modes of communication, the system's usefulness may be sharply limited.

2.2. Program Behavior

A single communications task is not an isolated episode, but part of a larger computation. The way in which these tasks are linked together into programs affects the load which is placed on the communications subsystem. One variable is the degree of coupling and sparseness: the uniformity of the communications needs throughout the system, and the proportion of the processors needing communications services at any moment. A second variable is the size of the packets which need to be moved around, whether they are short messages or sustained bursts of information. Finally, the degree to which the communications pattern is dynamic affects system performance; if a pattern remains uniform throughout a long computation, then the cost of an expensive setup phase may have negligible effect overall, whereas with a highly dynamic pattern setup costs may dominate the computation.

One important aspect of a parallel computation is the degree to which the actions taking place in the various processors is coupled. In a tightly coupled computation, all processors will be performing the same action at the same time. Periods of high communications activity will alternate with periods when the communications system is inactive. Furthermore, no processor will be able to proceed until every message in the communications pattern has reached its destination. In this situation the time required to perform a communications task, from start to finish, is the primary measure of system performance. In a loosely coupled system, the various processors will be performing dissimilar activities, so that communication and computation will overlap. Here the most relevant measure of system performance is the total effective bandwidth of the system.



Tight vs Loose Coupling

Another important aspect of a computation is sparseness: the proportion of processors trying to communicate at a given time. A communications system which is adequate when only 10% of the processors are trying to communicate may be woefully inadequate when the load approaches 100%; if a communications system is not carefully designed, putting too many messages into it at once may cause a traffic jam to develop, so that the movement of information comes nearly to a halt.

A third important characteristic of a computation is the size of the information packets being exchanged. If packets are large, then it is sensible to establish a complete path from sender to receiver before transmitting the data. The machine would then have a *circuit switching* architecture. However, if the packets are small, then it is sensible to send the data in a series of short jumps, buffering it at intermediate points. This yields a *packet switching* scheme. Using a circuit switching system with short messages may be inefficient because bandwidth may be wasted while waiting to establish the circuit. On the other hand, using packet switching with long messages may result in inefficiency if the messages are longer than the size of the intermediate buffers, which would necessitate breaking the message into several smaller units.

Finally, there is a distinction between a static communication pattern and a dynamic one. Establishing a communication pattern may involve significant setup time. If the pattern is unchanging, then this setup time does not matter very much. If, on the other hand, the pattern changes from one communications phase to the next, then setup time becomes a significant issue.

In short, the behavior of a program over time significantly affects the performance of the communications system of a parallel computer. If the various computations are strongly coupled, then the time to completion for a communications pattern (rather than bandwidth) is the best measure of performance. If every processor tries to communicate simultaneously, there is a danger of a traffic jam. If the size of the packets is small, then a circuit switching network may suffer from excessive overhead in establishing connec-

tions. Finally, if the communication pattern is highly dynamic, then a system requiring a high setup cost will be inefficient.

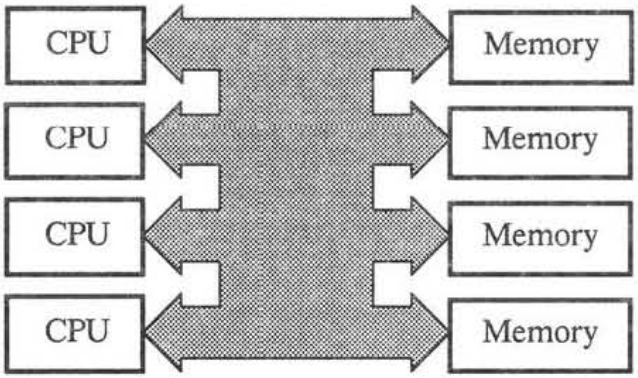
3. Communications Systems

Over the years, many different ways of combining processors, memories, and communications systems have been suggested or tried. Some methods work quite well for modest numbers of processors, but become impractical as the number of processors climbs into the thousands. This section will discuss some of these methods and point out their relative strengths and weaknesses. Of particular interest is how the cost building the system and the time needed to deliver a set of messages grows as the number of processors (N) increases.

This discussion is based on work by Hillis [1]. For a review of communications topologies, see Broomel and Heath [2], Thompson [3], or Benes [4]. For a taxonomy of parallel architectures, see Schwartz [5]

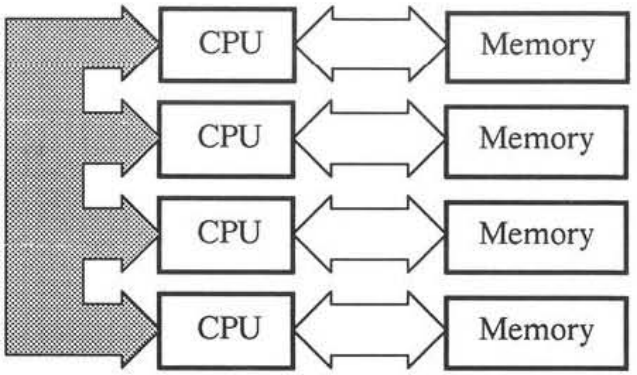
3.1. Shared vs Local Memory

There are two general ways of using a communications system in a parallel computer. The first is to interpose it between the CPU and the memories. In this case, the computer's memory will accessible to all processors, and the system will have a *shared memory* architecture.



A Shared Memory Computer

It is also possible to use a communications system for direct communications between CPUs, yielding a *local memory* architecture. The main difference between the two styles of machine is that in a shared memory computer, data flows in only one direction during one communications activity (from the CPU to the memory, or vice–versa), while in a local memory computer data may enter or exit the communications system through any port.



A Local Memory Computer

3.2. Shared Busses

One simple communications architecture is a single communications bus which is shared by all processors. This scheme suffers from several major problems. First, the bandwidth of the bus limits the total communications activity of the system. One way of compensating for this is by increasing the bandwidth of the bus. If the size of the packets is large (e.g. several hundred bits), then this can be done by making the bus wider. Another strategy is to use a higher level of technology in the bus than in the processors; it might then be possible to couple processors with a cycle time of 100 nanoseconds with a bus having a cycle time of 10 nanoseconds. This strategy suffers because the cost of building faster and faster busses escalates out of control; and in any event there is a limit to how fast a bus may be built. A second limitation with a shared bus is the time needed to arbitrate access to the bus; the speed of light places fundamental limits on how quickly ownership of the bus can be changed. A third limitation stems from the electrical fan-out of the bus; as the number of processors tapping into the bus increases, difficulties arise in supplying sufficient energy to drive all the taps.

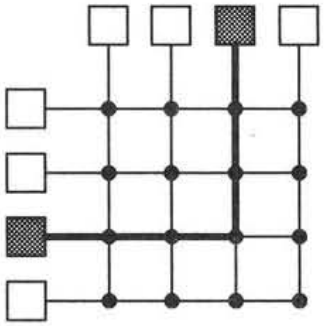
In summary, a shared bus limits the total amount of information that can be exchanged (due to bandwidth restrictions) as well as the total number of messages that can be exchanged (due to arbitration restrictions). This limits the usefulness of shared busses to architectures with relatively small numbers of processors.

For a discussion of a bus structured machines, see Davidson [6] (AMP-1*).

3.3. Crossbars

The simplest way of constructing a communications system is to connect every node to every other node, producing what is called a crossbar switch. A crossbar has certain advantages: communications time is independent of the number of processors, and it does not matter how many processors are simultaneously trying to communicate. In addition, if several connections out of a node are simultaneously active, then it is possible to implement fan-out in an efficient manner (this gives no help, however, in handling fan-in). The disadvantage of a crossbar is that the number of connections increases as the square of the number of processors: a 1000 processor machine would need 1,000,000 connec-

tions. This makes crossbars useful only for relatively small machines (up to a hundred processors or so). For a discussion of some crossbar machines, see Buehrer [7] (EMPRESS) or Trujillos [8] (Multimicrocomputer).



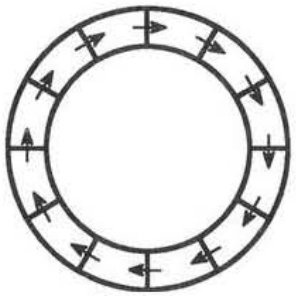
A 4x4 Crossbar

3.4. Clos Networks

If connections are one-to-one, then multi-layer networks with many of the same properties as a crossbar can be constructed with far fewer switches. These are called *Clos* networks. For example, a 5 layer Clos network with 1000 inputs requires only 146,300 switches, as opposed to 1,000,000 for a full crossbar. Nevertheless, costs still grow so fast as to preclude their use in machines with large numbers of processors. Clos networks are described by Benes [4]

3.5. Rings

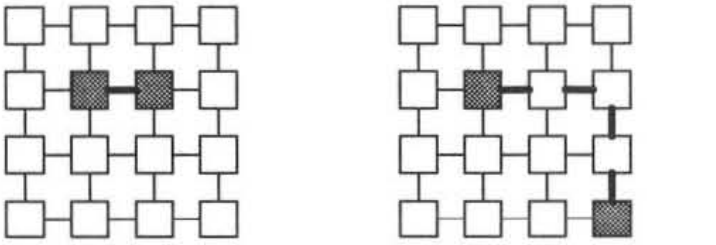
A ring network consists of a set of communications nodes arranged in a circle. At regular intervals, each node transmits information to the node to its right, and receives information from the node to its left. The nodes then examine the messages, and remove those which have reached their destination. Rings suffer from the same bandwidth limitations as shared busses, as a finite amount of information that can be moved from one station to the next on any cycle. Arbitration and electrical fan-out problems do not arise, but in their place are latency problems: the time needed to deliver a message is proportional to the number of nodes in the network. Again, this architecture is not feasible for large numbers of processors. For a discussion of the ZMOB, a 256 processor machine using a ring network, see Rieger [9].



A 12 node Ring Network

3.6. Grids

It is also possible to structure a machine as a two dimensional grid. Such a machine has several advantages. First, it is easily laid out on two dimensional boards and chips. Second, it allows very fast local communications. Third, it scales indefinitely: it is practical to build a grid machine of nearly unlimited size. The disadvantage of a grid structure is that communications between non-adjacent processors may have to pass through a large number of intermediate processors; this limits the applicability of grid machines to algorithms using only local communications. For a discussion some actual grid machines, see Slotnick [10] (ILLIAC IV), Batcher [11] (STARAN) or Batcher [12] (MPP).

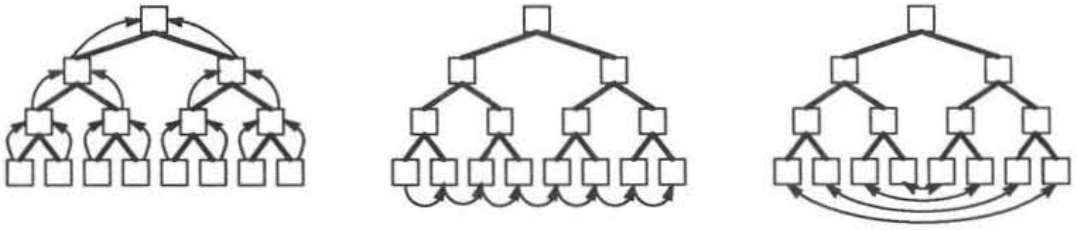


Local Communications Non-Local Communications

A 16 Processor Grid

3.7. Trees

The simplest topology that supports non-local communications for machines of arbitrary size is the tree. Trees have the advantage that for a machine with N processors, the distance between two nodes is never more than $\log_2 N$, and the cost grows linearly in N. In addition, it supports some of the most common regular communications patterns quite well. The disadvantage is that the root of the tree is a communications bottleneck. Furthermore, in the worst case every message in the system must pass through the root, and for an average random pattern half the messages will pass through it. This limits the applicability of tree machines to algorithms with one of a few regular communications patterns. For discussions of tree machines, see Shaw [13] (NON-VON) or Stolfo [14] (DADO).



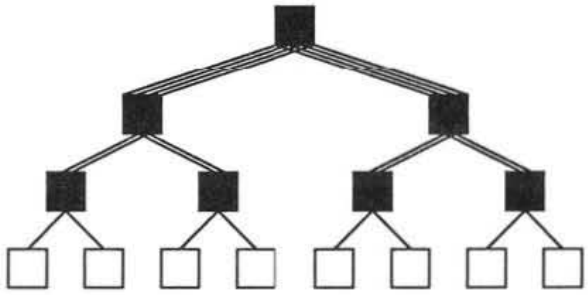
Local Best Case Worst Case

15 Node Binary Tree

3.8. Fat Trees

The problem of root congestion in tree machines may be remedied by adding extra communications paths at the higher levels of the tree. For example, the first and second

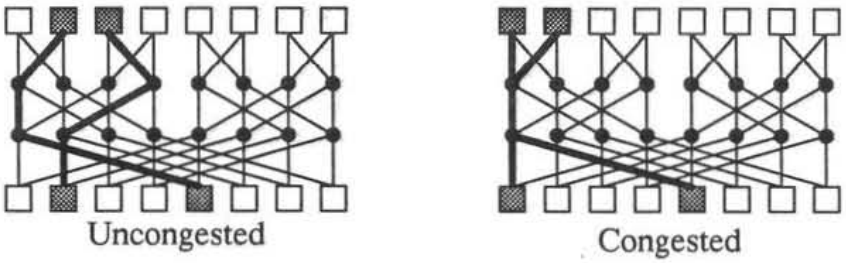
levels might be connected by single wires, the second and third levels by double wires, the third and fourth by quadruple wires, and so forth. All computation takes place at the leaves of the tree. The interior nodes switch signals between the various channels connecting to it. One advantage of fat trees is that, by varying the number of wires at each level, a family of architectures suitable for a variety of different applications can be generated. If the total number of wires at each level is kept constant (i.e. level N has 2^N wires), then the total cost of the network will be N $\log_2$ N, and the average time for a random one–to–one communications pattern will be $\log^2$ N. Both these numbers grow slowly enough to allow the construction of fat–tree machines with tens of thousands, even millions, of processing elements. Leiserson [15] proves that fat trees are universal, in the sense that a fat tree can simulate any physically buildable machine with no more than a $\log_2$ N factor slowdown.



A Fat Tree

3.9. Omega Networks

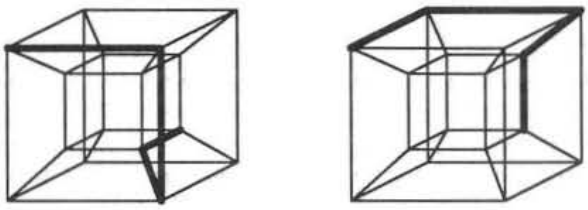
An omega network (or shuffle exchange or perfect shuffle) is a multi–layer switching network such that layer *k* allows signals to either propagate straight through or to be swapped with a signal 2^k wires away. As was the case with fat trees, the hardware cost grows as N $\log_2$ N in the number of processors, and the communications time is $\log^2$ N. One property of the omega network is that it has a bipartite topology: messages go from a set of input ports to a disjoint set of output ports. This makes butterflies well suited to shared memory architectures, where memories may be placed at the bottom of the switch and processors at the top. Another feature of an omega network is that there is exactly one path between any input node and any output node. This simplifies the routing of messages through the network. There are, however, patterns where a traffic jam develops and routing takes much more time. One remedy to this problem is to send messages via a randomly chosen intermediate processor, converting the problem to one with two random routings. Another remedy is to add additional data paths to allow for alternative routes. For discussions of some machines using omega networks, see Bolt Beranek and Newman Inc. [16], Rettberg and Thomas [17] (Butterfly™), or Schwartz [18] (Ultracomputer). For a demonstration of the equivalence of many omega–like networks, see Parker [19] or Snir [20].



A 3-layer Omega Network

3.10. Hypercubes

A hypercube (or N-cube) is a topology in which each of 2^k nodes is connected to k other nodes (the details of how this connection takes place are easier illustrated, below, than described). As with an omega network, a machine with N processors requires on the order of $N \log_2 N$ components to construct, and has $\log^2 N$ communication time. Its other properties are somewhat different. First, it has a uniform topology, in that messages may be both received and sent by one node in a single cycle. For this reason, hypercubes are particularly useful in local memory machines. Second, there are many routes connecting any pair of nodes. This leads to added flexibility in avoiding congestion and traffic jams, provided the routing algorithm is able to take advantage of this added flexibility. It should be noted that in a hypercube, there are N wires coming out of each node. In order to completely use this available communications bandwidth, it is therefore desirable to put approximately N processors at each node. This also facilitates the use of the redundant data paths by allowing up to N messages to converge on a single node. For a discussion of a machines based on hypercubes, see Hillis [1] (Connection Machine™ System).



A 4-Cube, showing two independent paths

3.11. Summary

A parallel computer requires a communications network to move information between its processors. Some communications architectures, such as shared busses and ring networks, have limited bandwidth, placing a strict ceiling on the number of processors which they can support. Others, such as crossbars and Clos networks, have communication times which remains essentially constant as the size of the machine grows, but costs which grow so fast that they are economically infeasible for large numbers of processors. Grid and tree machines can be build to arbitrary sizes, but their topology limits the com-

munications patterns they can efficiently execute to either local or certain regular patterns. Fat trees, butterflies, and hypercubes are a good compromise between escalating cost and deteriorating performance; their cost per processor grows as N $\log_2$ N in the number of processors, and their communication time increases as $\log^2$ N. This allows fat-tree, omega network, and hypercube machines of arbitrary size to be constructed. The choice between these architectures is governed by engineering and board layout concerns (e.g. compactness, uniformity, and ease of wiring) which are beyond the scope of this paper.

4. Architecture of the Connection Machine™ System

The previous section described the various interconnect topologies available to the computer architect. This section will describe how one such topology — the hypercube — was used as the basis of the Connection Machine™ System. The explanation will include a discussion of factors governing the construction of the nodes, the routing of messages, and some problems associated with fan-in and fan-out. Programming issues will not be discussed except as they impact on communications issues.

4.1. Communications Architecture

The primary design constraint on the Connection Machine System was that it have several orders of magnitude more computational power than a conventional machine. This immediately ruled out any sort of von Neumann machine, as well as parallel architectures (such as shared bus, ring, and cross bar) which are infeasible for large numbers of processors. An additional constraint was that it handle non-local and non-regular communications patterns. This ruled out grid and tree machines, leaving the choice between omega networks and hypercubes (design of the Connection Machine predates Leiserson's work on fat trees). Eventually, the hypercube was chosen.

A full explanation of why the hypercube was chosen over the omega network is beyond the scope of this discussion, but a few notes are in order. First, a hypercube has redundant data paths. This, it was felt, would reduce delays in routing due to contention for wires. Second, a hypercube has only computational nodes, rather than a mixture of computational and switching nodes; this reduces the number of different component types In addition, it makes the computational facilities of the individual processors available to assist in the routing of messages. Finally, every node in a hypercube is topologically equivalent; this means that only one type of node, and one type of board to carry those nodes, is ever needed. Based on economic and engineering limits, a hypercube with 4096 (2^{12}) nodes was chosen.

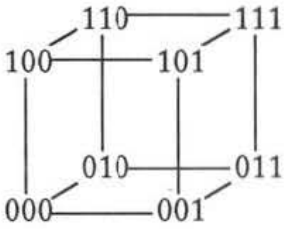
There are advantages to placing more than one processor at each node of a hypercube. First, for the regular patterns alluded to above, it is optimal to have one processor attached to each wire of a node. Second, for non-regular patterns, doubling the number of processors at each node causes the communications time to increase by a factor of less than two. This effect is due to the efficient utilization of the interconnecting wires. A

hypercube has sufficient bandwidth to route messages in $\log_2 N$ time. However, because it is impossible to keep all wires busy all the time, the actual communication time is log2 N. Putting several processors at each wire and queuing messages waiting to use the wire increases the utilization of the system's raw bandwidth. Partly on the basis of these considerations, each node was given 16 processors. This gives a total of 65,536 (2^{16}) processors.

4.2. The Router

Each node is contained in a single chip, which contains the 16 processors, the hypercube node, and connections to 12 wires. The processors' local memory is located elsewhere. The router performs 5 functions: *injecting*, *routing*, *buffering*, *referring*, and *delivering*. Injecting is removing a message from a processor and placing it in the hypercube network. Routing is switching a message to a wire. Buffering is temporarily storing a message when a wire is being used. Referring is sending a message over a random wire when buffer space is exhausted. Delivering is removing a message which has reached its destination out of the hypercube system and placing it in a processor.

It is possible to assign binary numbers to the node of a hypercube in such a way that the numbers assigned to two nodes differ by exactly one bit if and only if they are connected by an edge. Furthermore, every dimension of the hypercube corresponds to a bit position in the address. In the figure below, the left–right dimension corresponds to the rightmost bit, the in–out dimension corresponds to the middle bit, and the up–down dimension corresponds to the leftmost bit. This yields the routing algorithm for the Connection Machine System. First, for every message, find the *relative address* by taking the exclusive–OR of the addresses of the sending and receiving processors. Second, send the message over *any* unused wire corresponding to a 1 in the relative address. If no such wire can be found, put the message in a buffer. If no buffer is available, send the message over any free wire. Third, whenever a message crosses a wire, the corresponding bit in the relative address is inverted. Finally, when the relative address contains all 0's, the message has reached its destination.



Numbering of Nodes on a 3–Cube

This scheme works quite well for one–to–one communications patterns, so that a pattern of 65,536 32–bit messages can be routed in 800 microseconds. A modification of this scheme suggested by Blelloch [21] allows patterns with fan–in and fan–out to be routed in approximately twice the basic message cycle time. For fan–out, this is done by using a fast ($\log_2 N$ time) algorithm to make an appropriate number of copies of each message, then using the routing algorithm shown above the deliver the copies. For fan–

in, the above process is inverted. The two may be used together to implement combined fan-in and fan-out. The only disadvantage of Blelloch's methods is that they require a substantial amount of time (16 milliseconds) to set up, and are thus poorly suited to applications with highly dynamic communications patterns having fan-in or fan-out. The implications of these techniques have been investigated by Hillis and Steele [22].

5. Summary

In summary, computer architecture has become, in large measure, communications architecture. This is because the only way to build computers which are fundamentally more powerful than those currently available is to use thousands or tens of thousands of processing elements. At this point, communications rather than computation becomes the primary preoccupation of the computer architect. Many communications schemes have been suggested. Some of these are unsuitable for large architectures, either because their bandwidth does not increase quickly enough or because their cost escalates too quickly. Other communications schemes are indefinitely scalable, but are limited to specialized applications by restrictions on the sorts of message patterns they can support. Finally, there is a group of schemes, including fat trees, omega networks, and hypercubes, that are indefinitely scalable, both in terms of their cost and the time needed to deliver a set of messages.

One such scheme, the hypercube, was chosen as the basis of the Connection Machine System. The communications system it contains supports 65,536 processors, can transmit 65,536 messages in 800 microseconds, and can be adapted to patterns with high degrees of fan-in and fan-out. The result is the state-of-the-art in computer communications architecture.

References

[1] Hillis, D., *The Connection Machine,* (MIT Press, Cambridge Massachusetts, 1986).

[2] Broomel, G., and Heath, J.R., "Classification Categories and Historical Development of Circuit Switching Topologies," *Computing Surveys* 15 (2) (1983) pp 95-133.

[3] Thompson, C., "Generalized Connection Networks for Parallel Processor Intercommunication," *IEEE transactions on Computers,* C-27 (12) (1978).

[4] Benes, V. E., *Mathematical Theory of Connecting Networks and Telephone Traffic* (Academic Press, 1965).

[5] Schwartz, J., "A Taxonomic Table of Parallel Computers, Based on 55 Designs," Courant Institute, New York University (1983).

[6] Davidson, E., "A Multiple Stream Microprocessor Prototype System: AMP-1*," Coordinated Science Laboratory, University of Illinois, Urbana, IL, and *IEEE* (1980).

[7] Buehrer, R. E., et al, "The ETH–Multiprocessor EMPRESS: A Dynamically Configurable MIMD System," *IEEE Transactions on Computers* C–31 (11) (1982) pp 1035–1044.

[8] Trujillo, V., "System Architecture of a Reconfigurable Multimicroprocessor Research System," *1982 International Conference on Parallel Processing* (1982).

[9] Rieger, C., "ZMOB: A Mob of 256 Cooperative Z80A–Based Microcomputers," Computer Science Tech. Rep. Series TR–852, University of Maryland, College Park, MD, (1979).

[10] Slotnick, D. L, et al., "The ILLIAC IV Computer," *IEEE Transactions on Computers* C–17 (8) (1978) pp 746–757.

[11] Batcher, K. E., "STARAN Parallel Processor System Hardware," *AFIPS Conf. Proc.* 43 (1974) pp 405–410.

[12] Batcher, K. E., "Design of a Massively Parallel Processor," *IEEE Transactions on Computers,* C–29 (9) (1980).

[13] Shaw, D.E., "The NON–VON Supercomputer," Department of Computer Science, Columbia University (1982).

[14] Stolfo, S., and Shaw, D., "DADO: A Tree–Structured Machine Architecture for Production Systems," Department of Computer Science, Columbia University (1982).

[15] Leiserson, C.E., "FAT–TREES: Universal Networks for Hardware–Efficient Supercomputing," *1985 International Conference on Parallel Processing,* IEEE Computer Society (August 1985).

[16] Bolt Beranek and Newman Inc., "Development of a Butterfly Multiprocessor Test Bed," Report 5872, Quarterly Technical Report No. 1 (1985).

[17] Rettberg, R., and Thomas, R., "Contention Is No Obstacle to Shared–Memory Multiprocessing," *Communications of the ACM,* 9 (12) (December 1986) pp 1202–1212.

[18] Schwartz, J., "Ultracomputers," *ACM Transactions on Programming Languages and Systems* 2 (4) (1980) pp 484–521.

[19] Parker, D., "Notes on Shuffle/Exchange–Type Switching Networks," *IEEE Transactions on Computers* C–29 (3) (1980) pp 213–222.

[20] Snir, M., "Comments on Lens and Hypertrees — or the Perfect–Shuffle Again," Ultracomputer Note 38, Computer Science Department, New York University (1982).

[21] Blelloch, G., "Parallel prefix versus concurrent memory access," Technical Report, Thinking Machines Corporation, Cambridge MA (June 1986).

[22] Hillis, D., and Steele, G., "Data Parallel Algorithms," *Communications of the ACM,* 29 (12) (December 1986) pp 1170–1183.