MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. Memo No. 646 September, 1981

The Connection Machine

(Computer Architecture for the New Wave)

by W. Daniel Hillis

ABSTRACT: This paper describes the *connection memory*, a machine for concurrently manipulating knowledge stored in semantic networks. We need the connection memory because conventional serial computers cannot move through such networks fast enough. The connection memory sidesteps the problem by providing processing power proportional to the size of the network. Each node and link in the network has its own simple processor. These connect to form a uniform locally-connected network of perhaps a million processor/memory cells.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Artificial Intelligence Laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under contract with the Office of Naval Research contract N00014-80-C-0505. The author is supported by a fellowship provided by the Fannie and John Hertz Foundation. © MASSACHUSETTS INSTITUTE OF TECHNOLOGY 1981

THE CONNECTION MACHINE

This paper describes the *connection memory*, a machine for concurrently manipulating knowledge stored in semantic networks. We need the connection machine because conventional serial computers cannot move through such networks fast enough. The connection memory sidesteps the problem by providing processing power proportional to the size of the network. Each node and link in the network has its own simple processor. These connect to form a uniform locally-connected network of perhaps a million processor/memory cells.

The connection memory is not meant to be a general-purpose parallel computer. It is fast at a few simple operations that are important for artificial intelligence, such as property lookup in a semantic inheritance network. I will discuss the need for such a machine, what it will do, and how it will work. I describe progress already made toward its design and a plan to actually build a hundred-thousand-cell prototype.

Our Machines are Too Slow

On a serial machine, the time required to retrieve information from a network often increases with size of the network. Thus paradoxically, programs become slow as they become smart. Today, we write artificial intelligence programs that use a few hundred facts. We would like to increase this to a few million, but the programs already take minutes to make decisions that must be made in seconds. Scaled up, they would take years. Von Neumann machines, even if they are built of exotic ultrafast components, are unlikely candidates for solving these problems, since they are limited by the speed of light. A supercomputer inside a six-inch cube would take one nanosecond to send a single signal from one corner to the other. A nanosecond cycle time is less than a factor of a hundred better than currently available machines, not nearly enough to solve our million-scaled artificial intelligence problems.

The Potential Solution is Concurrency

The light at the end of the tunnel is concurrency. Integrated-circuit technology makes it economically feasible to produce millions of computing devices to work on our problems in parallel. Artificial intelligence mechanisms have been proposed that are suitable for such extreme parallel decomposition [Fahlman, Minsky, Shank, Rieger, Winston, Steels, Steele, Doyle, Drescher, etc.]. These systems represent information as networks of interconnected nodes. Many of their operations are dependent only on local information at the nodes. Such operations could, potentially, be performed in parallel on many nodes at once, making the speed of the system independent of the size of the network.

Unfortunately, the word-at-a-time von Neumann architecture is not well suited for exploiting such concurrency. When performing relatively simple computations on large amounts of data, a von Neumann computer does not utilize its hardware efficiently; the number of interesting events per second per acre of silicon is very low. Most of the chip area is memory and only a few memory locations are accessed at a time. The performance of the machine is limited by the bandwidth between memory and processor. This is what Backus [1] calls the *von Neumann Bottleneck*. The bigger we build machines, the worse it gets.

The bottleneck may be avoided by putting the processing where the data is, in the memory. In this scheme the memory becomes the processor. Each object in memory has associated with it not only the hardware necessary to hold the state of the object, but also the hardware necessary to process it.

A Few Specific Operations Must be Fast

Knowledge retrieval in Artificial Intelligence involves more than just looking up a fact in a table. If the knowledge is stored as a semantic network, then finding the relevant information may involve searching the entire network. Worse yet, the desired fact may not be explicitly stored at all. It may have to be deduced from other stored information.

When retrieving knowledge, programs often spend most of their time repeating a few simple operations. These are the operations that we want to be fast:

- We need to **deduce** facts from semantic inheritance networks, like KLONE[2], NETL[6], OWL[21] or OMEGA[9].

- We need to **match** patterns against sets of assertions, demons, or productions. If there is no perfect match we may need the best match.

- We need to **sort** a set according some parameter. For instance, a program may need to order goals in terms of importance.

- We need to **search** graphs for sub-graphs with a specified structure. For instance, we may wish to find an analogy to a situation.

Tools have already been developed for describing for these operations in terms of concurrent processes. In Codd's relational database algebra, [4] database queries are specified in terms of a few simple, potentially-concurrent primitives. Another example, more directly connected to artificial intelligence, is Fahlman's [6] work on marker

propagation. Fahlman has shown that many simple deductions, such as property inheritance can be expressed in terms of parallel operations. Schwartz [17] has developed a language based on set operations. Woods has developed a more powerful extension of marker propagation. By providing a few powerful primitives that can be evaluated concurrently, each of these descriptive systems allows a programmer to express concurrent algorithms naturally. The connection memory is designed to exploit the parallelism inherent in these operations.

Marker Propagation was a Good First Step

In 1968, Quillian [25] proposed that information stored in a semantic network could be manipulated by concurrently propagating markers through the network. Such a system would be able to retrieve information in a time that was essentially independent of the size of the network. This basic idea was extended considerably in the late 1970's by Fahlman [6] and by Woods, [24] who worked out ways of controlling the marker propagation to perform deduction and retrieval operations on inheritance networks. Fahlman also proposed hardware for actually implementing his system concurrently.

Unfortunately, many of the marker propagation strategies are just heuristic. In complicated cases they give the wrong answers. [6,12] Systems with well-defined semantics, like OWL [21] and OMEGA [8], have never been successfully expressed in terms of markers. I believe that marker propagation systems, while on the right track, are not sufficiently powerful to implement these systems.

The Connection Memory

The connection memory architecture captures many of the positive qualities of marker propagation, without some of its weaknesses. It is a way of connecting together millions of tiny processing cells so that they can work on a problem together. Each cell can communicate with a few others through a communications network. **The communication connections are configured to mimic the structure of the specific problem being solved.** For a particular semantic network, the cells are connected in the same way as the data in the network. Thus, each chunk of data has its own processor, connected to processors of related data.

If the connections were physical wires, the machine would have to rewired for every problem. Since this is impractical, the processing cells are connected through a switching network. They communicate by sending messages. Receiving a message causes a cell to change its state, and perhaps to transmit a few more messages. As in Hewitt's *actor systems*, all computation takes place through the exchange of messages.

Below I describe how this all works: the communication network, the algorithms for computation and the formation of connections, and the operation of the cells. The most important features of the connection memory are:

- o It is fast. Most of the chip area is usefully active during a computation. The system may execute several million operations at a time.

- o It is wireable. The communication network is locally connected. All wires are short and pack efficiently into two dimensions. The ratio of wires to active elements can be independent of the size of the system.

- o It is useful. The connection memory seems to be able to implement all of the operations of the relational algebra, as well as structured inheritance networks such as KLONE [2], OMEGA [8], and OWL [21].

Structures in the Machine at Different Levels of Abstraction

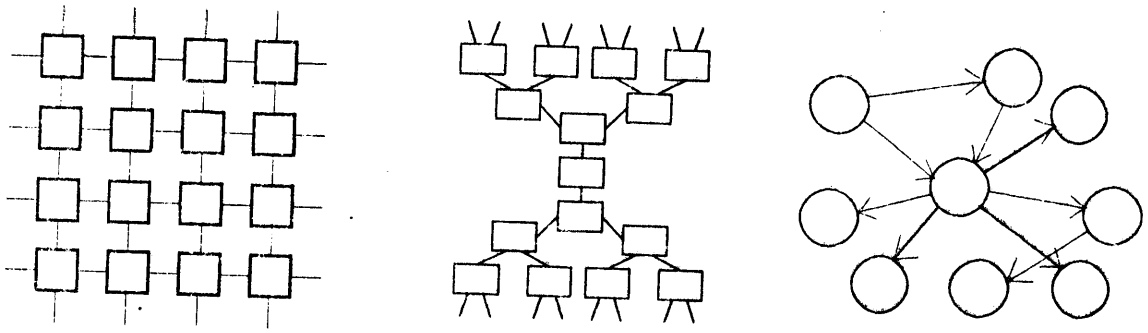

Figure 1.

All Communication Is Local

At the lowest level, the connection machine is a uniform array of *cells*, each connected by physical wires to a few of its nearest neighbors. Each cell contains a few words of memory, a very simple processor, and a communicator figure 2. The communicators form a packet-switched communications network. Cells interact through the network by sending messages. Each cell knows the addresses of a few other cells. When two cells know each other's addresses, they can communicate. This establishes a virtual **connection** between the cells. Connected cells behave as if they were linked by a physical wire, although messages actually pass through the network.

Each cell contains a simple processor.

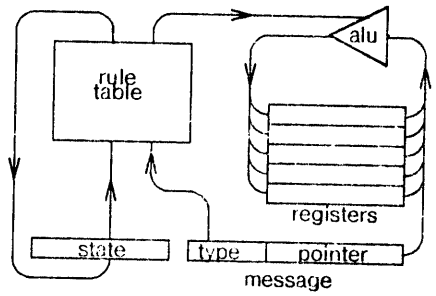

Figure 2.

Since the physical wires are all short, message must reach their destinations in incremental steps, through intermediate communicators. A cell addresses a message by specifying the relative displacement of the recipient (example: up two and over five). This does not specify the route the message is to take, just its destination. When a communicator receives a message it decides on the basis of the address and local information which way the the message should go next. It modifies the address and sends it to the selected neighbor. For example, a communicator receiving a message addressed "up two and over five" can change it to "up one and over five" and send the message to the communicator above. When the address is all zeros, the message is at its destination and can be delivered. A

single message step is illustrated in Figure 3.

A Single Step of a Message toward its Destination.

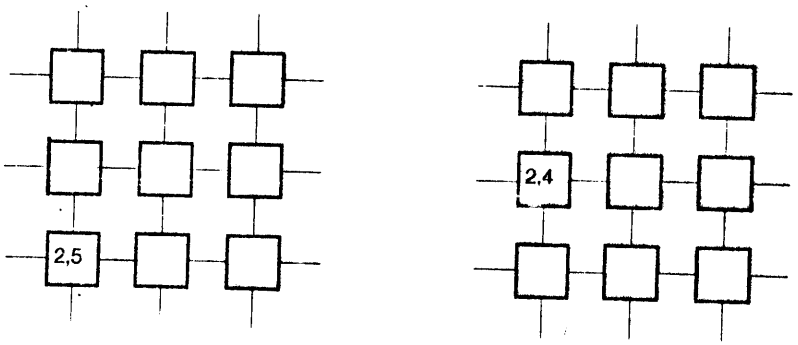

Figure 3.

Cells are Simple

Most of the hardware in a cell is memory. Each cell has a few registers, a state vector, and a rule table. The rule table is identical for all cells, so a single table can be shared among multiple cells on a chip. The registers and state vector are duplicated for each cell. Registers hold relative addresses of other cells. A cell normally has three virtual connections, so three registers are needed. There are also two or three extra registers for temporary storage of addresses and numbers. The state vector is a vector of bits. It stores markers, arithmetic condition flags and the type of the cell. A cell may have 10 to 50 bits of state vector. Addresses in a million word machine are 20 bits long, so there will be a total of about 150 bits per cell, not including the shared rule table.

The rule table tells the cell how to behave when it receives a message. Each message contains an address or number and a type field. The way a cell responds to a message depends on the state of the cell and the type of the message. When a message is received, the state and the message type are combined and used as an index into the rule table. The appropriate response is determined from the table entry. It may involve changing the cell's state vector, originating new messages, or performing an arithmetic operation, or some

combination of these operations. The cell's state vector usually changes as a result of receiving a message.

If a cell is to transmit a message, the rule table must indicate the type of the message, the pointer of the message, and the address of the recipient. The pointer and the address normally come from the registers, although they may also be loaded with numerical constants, such as the cell's own address. Since the addressing scheme is relative, the cell's own address is always zero. The addresses of immediate neighbors are also simple constants.

Arithmetic operations take place on the contents of the pointer registers, and the result can be stored back into a register. The state vector has condition-code bits which are set according to the result. For instance, there are bits indicating a zero result, a negative result, and a carry overflow. Since these bits are treated as part of the state vector, they can influence the future behavior of the cell. This is useful for numerical sorting operations.

Storage is Allocated Locally

Data in the connection memory is stored as the pattern of connections between cells. This is similar to Lisp, where data is stored as structures of pointers. The connections represent the contents of the memory.

Unconnected cells can establish a connection by a mechanism called *message waves.* Assume cell JOHN wants to get a pointer to cell MARY, but has no idea where cell MARY is. JOHN can get such a pointer by broadcasting a message wave through the network, searching for MARY. Each message in the wave contains the address of the cell that originated the wave. The wave is propagated by the individual cells, each cell forwarding the wave to its neighbors, incrementing or decrementing the backpointer appropriately. The is illustrated in figure 4. When the wave reaches cell MARY, MARY sends her address back to JOHN, using JOHN's address as specified in the wave. JOHN then sends out a second wave to cancel the still spreading request. The cancel wave travels at twice the speed of the request wave, so it overtakes the request and prevents it from propagating further.

A similar technique may be used to connect to a cell of a particular type , rather than to a specific cell. This happens most often when building new structures from unused cells. In this case handshaking is necessary to insure that only a single cell is found, even though several satisfactory cells may have replied to the request before it was canceled. A unused cell which sees a request wave transmits an AVAILABLE message back to the originator. The originator replies to the first such message with an ACCEPT, and to all subsequent messages with REJECT messages.

A Message Wave

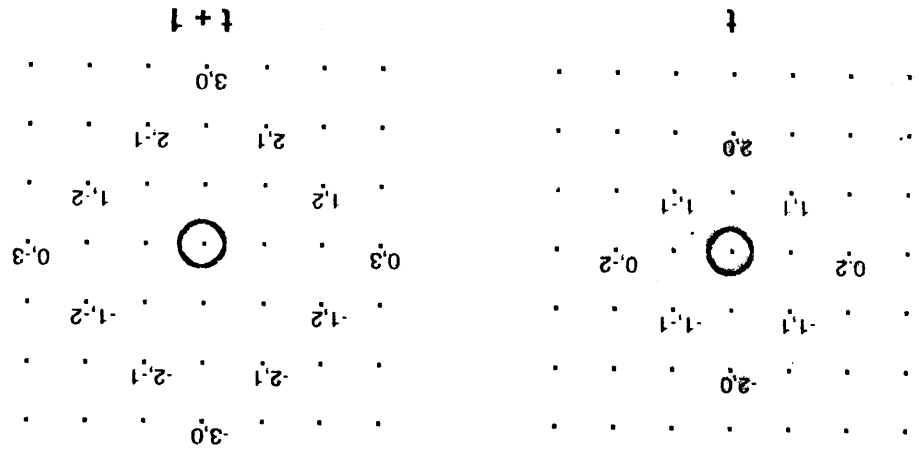

Figure 4.

It is possible to calculate just how far the request message travels before the cancel wave catches up. The space-time diagram in figure 5 shows how far each message must travel. If the request wave propagates at half the rate of the other messages, it will travel twice the necessary distance before it is canceled. This means that when connecting to an unused node, if we assume that the free nodes are uniformly distributed, it will be necessary to refuse about three AVAILABLE messages per connection.

This method of allocating storage may allow the machine to continue to operate with defective cells. Cells are connected on the basis of availability, not address, so bad cells need never be built into the network. Assume each cell has some way of knowing which of its neighbors are functioning properly. Since a cell only interacts with the system through its neighbors, a malfunctioning cell can be cut off from the rest of the system. The neighbors never route a message through the bad cell and ignore any messages it tries to transmit. None of the connection memory's algorithms depend on a cell existing at specific addresses. A system with a few faulty cells could continue to function, with a slight degradation in performance.

[I have not yet studied this defect-tolerance scheme in detail, so there may be bugs. It will become important if we ever need to build very large machines or very large (wafer-sized) chips.]

Space-time Diagram of Storage Allocation.

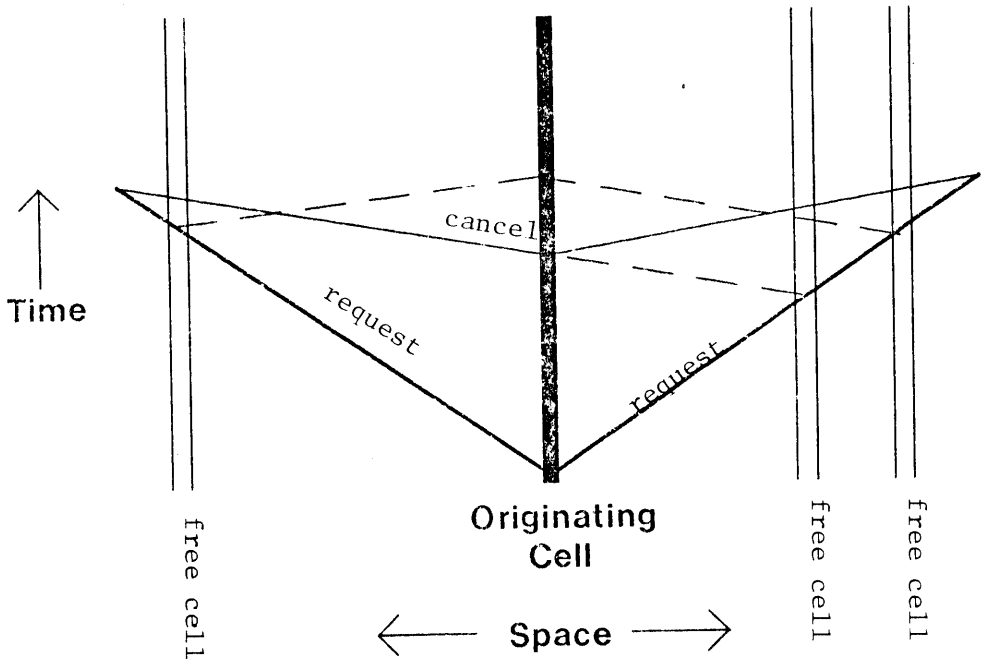

Figure 5.

Trees Represent Nodes

A node in a semantic network can be linked to an arbitrary number of other nodes. A cell, on the other hand, can only connect to a few other cells. Since the network is to be represented as a structure of connected cells, there must be some way of representing nodes with an arbitrary number of connections. This is accomplished by representing each node as a balanced binary tree of cells.

In this scheme, each cell only needs three connections. One connection links the cell to those above it in the tree and the other two connections link to the subtrees below. Each node is a tree of cells. The depth of the tree is equal to the logarithm of the number of connections to the node. The total number of cells required to represent a node is equal to the number of connections minus one.

The links in the network are also represented as connected cells. In this case, there is no fanout problem. Each link connects to exactly three nodes: the two linked nodes, plus the type of the link. Thus, a link can be represented by a single cell, that connects leaves of the

appropriate node trees. The representation of a small net is shown in figure 6.

Representing Nodes in terms of Cells.

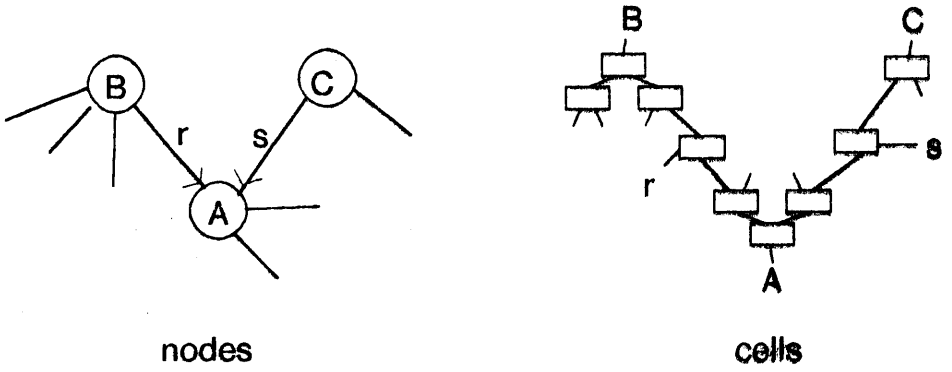

Figure 6.

Operations which add connections to the node tree must leave it balanced. To help with this, each cell carries a bit indicating if new connections should be added to the left or right side of the cell. This bit is set if the tree below the cell is left-heavy, clear if it is right-heavy, and may be either if it is perfectly balanced. When adding a new connection, a message starts at the top of the tree and move left or right as it goes down according to the balance bit. As it passes though, it complements the bit, as shown in figure 7. This operation not only selects the correct terminal of the tree, but also leaves the balance bits in a consistent state, ready for the next insertion. A similar algorithm must be used for deletion. (This elegant algorithm was invented by Carl Feynman and independently by Browning at the California Institute of Technology.)

The algorithm can be generalized to make a number of connections simultaneously. To do this, we send the number of connections to be made to the top cell of the tree. The cell divides this number by two and passes the result to the left and right sub-cells. If the number does not divide evenly the extra count is passed to the lean side of the tree. If each

The Feynman/Browning Tree-Balancing Algorithm

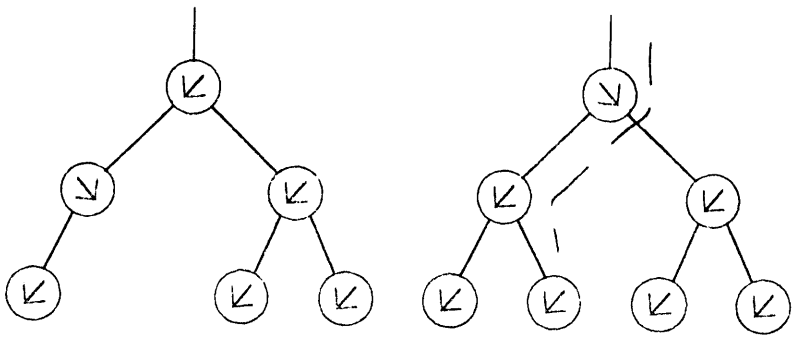

Figure 7.

---

node repeats this process the numbers that reach the terminal nodes will indicate how many connections are to be made to those points. Again, the balance bit must be toggled as the numbers pass through.

Objects Can Move to Shorten Distances

It is sometimes useful to make a distinction between the hardware of a cell and the computational object that is stored in a cell. I will call the object a *cons*, by analogy to Lisp. A cell with no cons is *free*, and may be used to build new structures.

Connections are all bidirectional, so each cons knows the address of all conses that know its address. Knight has pointed out that a cons is free to move from cell to cell, as long as it informs its acquaintances where it is moving. This would allow conses with frequent communication to move nearer. Conses in the configuration shown in figure 8 could swap places. Conses that do not wish to swap could act as intermediaries, negotiating swaps between conses on either side (fig 8 c). If conses keep track of their utilization, an often used cons may force a swap even if it is to a less-used cons's disadvantage. This would allow implementation of a *virtual network*, analogous to virtual memories on conventional computers. Little used conses would gradually be pushed away from the center of activity and eventually fall off into a secondary storage device. As in virtual memory, there could be several layers of successively slower and less expensive memories, say NMOS, magnetic bubbles, and disk.

Conses Swapping to Shorten Path I engths

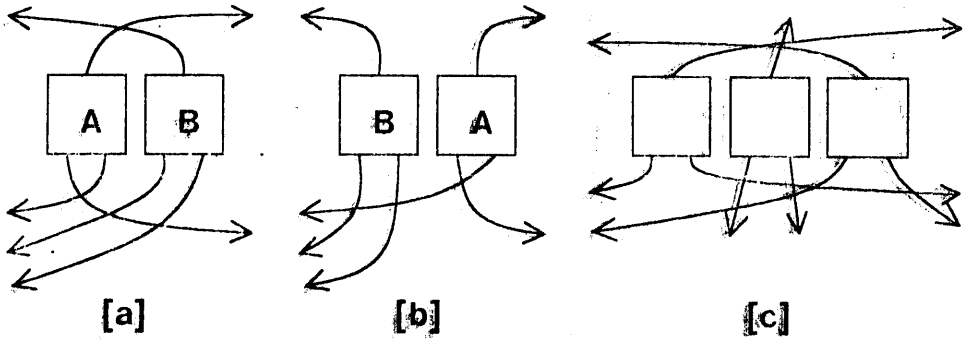

Figure 8.

I have not yet studied these migration schemes in detail. Whatever system we use, memory management in a connection machine should be easier than in conventional systems because each object is referenced only by a small, well-defined set of acquaintances. It can be safely moved after informing those acquaintances.

The Connection Memory Operates on Sets

In this section I present a register-machine description of the connection memory. This is only one possible interface between the connection memory and the outside world. It is included here because it shows specifically how the connection machine can perform certain retrieval operations.

This model does not capture the full power of the connection memory. The instructions described below are implemented by loading the rule tables of the cells, starting the machine, and waiting for the calculation to complete. This mode of operation fails to take full advantage of the memory's parallelism.

The connection memory is connected to a conventional computer in the same way as any other memory. Its contents can be read and written with normal array-like read and write operations. There are also other ways of accessing and modifying the contents. To take advantage of these additional functions, the programmer must follow certain conventions for the format of stored data. The machine treats the data as as set of named nodes, connected by named links. In artificial intelligence programs the nodes of such a network

usually represent concepts and the links represent relations between those concepts. The connection memory, however, knows nothing about the semantics of networks, only their structure.

The abstract machine has several registers. Unlike the registers of a serial machine, which hold numbers or pointers, the connection memory registers hold sets or functions.

Set-registers contain sets of nodes in the network. These sets can be arbitrarily large. The basic operations of the machine take place on every member of a set simultaneously, which accounts for most of the machine's concurrency. The letters A, B, C, and so on, will refer to set-registers. Each set-register is implemented using one bit in the state vector of every node. A set-register contains contains exactly those nodes that have the corresponding bit set.

There are also function-registers. These contain functions mapping nodes to nodes, nodes or to numbers. The letters F, G, H, and so on will be used to refer to function-registers. Each function-register is implemented by storing an address in every node. The address indicates where that node is mapped under the corresponding function. It is relatively expensive to store an address at each node, so there are only a small number of function-registers.

The instructions of the register machine fall roughly into four groups: set operation, propagation, function manipulation and structure modification, and arithmetic. Instructions in the first two groups give the machine the power of a parallel marker propagation machine such as Fahlman's. The other instructions give the machine additional capabilities involving function manipulation, pointer passing and arithmetic. Each instruction group will be discussed separately below.

Group I: Set Operations

Since the set-registers of the connection memory hold sets of objects, natural register-to-register operations are the standard set operations. In the connection memory,

```
A ← INTERSECT(B,C)
```

represents a single instruction, where "←" indicates that the value on the right is deposited into the register on the left. This particular instruction intersects the contents of two set-registers and loads the result into a third. The other standard set operations (UNION, DIFFERENCE, COMPLEMENT) are also single instructions. "Complement" in this case means complement with respect to the set of all of the nodes in the network.

Registers may be initialized to the empty set with the CLEAR instruction.

These set instructions all operate simply by performing the appropriate Boolean operations on the state vectors of all the nodes in the network. No messages need to be sent.

Group II: Propagation

Consider the following equivalent descriptions of links in a network:

- Each link is a directed connection between two nodes, with a label specifying the type of link. There are no redundant connections, i.e. no two connections with the same label start and end at the same nodes.

- Each link type is a predicate on pairs of node, selecting pairs that bear the specified relationship.

- Each link type is a relation which maps each node to a (possibly empty) set of nodes. Specifically it maps a node into the nodes to which it is connected by a link of that type.

- Each link type is a function that maps sets of nodes into sets of nodes connected by that type of link. The function is additive in the sense that if A=B U C then F(A)=F(B) U F(C). Thus, the function is defined by its behavior on the singleton sets.

These descriptions are all equivalent, in that they all describe the same mathematical object: an arbitrary set of ordered pairs of nodes. Let us call such an object a *relation*, but when we speak of applying a relation to a set, the last description is most useful in understanding what is really happening. I will be careful to *not* call this object a *function*, because that would confuse it with the things kept in function registers.

As an example, assume that the network contains nodes representing physical objects and nodes representing colors. Each object node has a color-of link connecting to the node that represents the object's color. Given such a network, we may find the color of an object by applying the color-of relation to a set containing the object. When we apply a relation we are treat it as a function from sets to sets, as in the last viewpoint above. For instance, if register A contains the singleton set {apple} then,

```
B ← APPLY-RELATION(color-of,A)
```

will load register B with {red}. Of course, the registers do not need to be loaded with singleton sets. If A had contained {apple, banana, cherry} the same instruction would have put {red, yellow} into B. Here both apples and cherries are red, so both nodes would map into the same color node.

The applied relation may map several sets into one. color-of, for example, will map both {apple} and {cherry} into {red}. This means that the relations do not always have inverses when viewed as functions. There is however always a *reverse*, which corresponds to moving backwards along the link in the same way that the standard relation correspond to moving forward along the link. For example, if A contains {red} then

```
B ← APPLY-REVERSE-RELATION(color-of,A)
```

will load B with set of all red things. The inverse relation has the property that it will always get back at least what you started with:

```
A ⊂ APPLY-REVERSE-RELATION(relation,APPLY-RELATION(relation,A))
```

Another useful associated relation is the transitive closure. This does not make much sense with respect to the color-of relation, so instead imagine a genealogy network in which nodes representing individual people are connected by parent-of links. In such a network, if register A contained {John},

```
B ← APPLY-RELATION-CLOSURE(parent-of,A,U)
```

would load B with the set of all of the ancestors of John. The third argument U, specifies the set over which the relation is closed. In this case, U specifies the set of all nodes. If we are interested only in John's matriarchal ancestry, this third argument would be the set of females. There is also an APPLY-REVERSE-RELATION-CLOSURE instruction, which would find all of John's descendants. All of the instructions in this section work by transmitting messages from node to node containing selected bits from the node's state vector. Thus, for example, the APPLY-RELATION instruction works by having all nodes in the specified set (that is, all nodes with a specific bit in their state vector set) transmit messages to this effect through color-of links. Nodes receiving such messages can then set the appropriate bit indicating that they are a member of the answer set.

Example: Property Inheritance in a Virtual-Copy Hierarchy.

Assume that colors and types of objects are represented in a network. The are two types of links in this network, `color-of` links and `virtual-copy` links. The `virtual-copy` links represent class membership. This is a transitive property: crab-apples are a kind of apple, apples are a kind of fruit, so crab-apples are fruit. The `color-of` links connect an object to its color. If there is no explicitly stored `color-of` link then the color is inherited though the `virtual-copy` hierarchy; crab-apples are red because crab-apple is a virtual copy of apple.

Here is a sequence of connection memory operations that finds all of the red things stored in such a virtual copy network:

```
A ← APPLY-REVERSE-RELATION(color-of,{red})  ;A is all explicitly red things.
B ← COMPLEMENT({red})
B ← APPLY-REVERSE-RELATION(color-of,B) ;B is all explicitly non-red things.
B ← COMPLEMENT(B)                      ;B is all red or possibly red things.
C ← APPLY-REVERSE-RELATION-CLOSURE(virtual-copy,A,B) ;C gets all red things.
```

This code will properly inherit the color of all super-types. It will also allow inherited properties to be explicitly overridden.

Group III: Instructions for Manipulating Functions

The instructions mentioned so far, allow the machine to do anything that can be done with a content-addressable memory or a marker-propagation machine. Marker programs that use *n* marks can always be translated into a connection-memory program using *n* set-registers. Unfortunately, not all easy-to-partition algorithms can be expressed in terms of set operations. For example, in the genealogy network above it is would be impossible to find every man who is his own father. To compute this function the machine must consider each node independently. A marker-propagation machine would require a separate marker for each individual. In relational database terms, a marker propagation or a set machine can concurrently compute projections and restrictions, but not joins.

This motivates the introduction of the next group of instructions, which give the connection memory additional power for handling these sorts of problems. The source of this additional power is the connection memory's ability to manipulate arbitrary functions. Such functions, from nodes to nodes, are held in the function-registers. In the sample instructions below, the letters `F`, `G` and `H` represent function registers.

The easiest way to load a function register is from a relation stored in the network. Since functions must be single valued and a relation can be multiple valued, they cannot always

be loaded directly. The connection memory handles the problem by selecting among the multiple values by an "indexing" operation. For example, if r is a single-valued relation, then

```
F ← FUNCTION(r,1)
```

will load function register F with the function that maps each node onto its r-related node, if there is one. If there is more than one, it will choose a single value according to the index. This second argument indexes the choice among the multiple values by using it to determine a unique path through the various fan-out trees in the representation of the network. The exact details of this algorithm are unimportant, except in that it guarantees that the FUNCTION instruction executed twice with the same index will return the same result. This allows a k-valued relation to be treated as a k-long vector of functions.

One thing to do with a function is to apply it, so there are APPLY-FUNCTION and APPLY-FUNCTION-CLOSURE, which are analogous to the APPLY-RELATION and APPLY-RELATION-CLOSURE instructions for applying relations.

A function may also be used to modify the structure of the network. This is the only available mechanism for building structure concurrently. For any relation r, the instruction

```
INSERT(F,r)
```

will add to r all pairs in the contents of function-register F. Similarly DELETE will delete pairs from a relation.

Since functions can be viewed as sets of ordered pairs, they may also be combined using INTERSECT-FUNCTIONS and DIFFERENCE-FUNCTIONS. UNION-FUNCTIONS may also be used if the result is actually a function, as in the union of functions with disjoint domains.

The COMPOSE instruction can be used to compose a relation with a function. Since such a composition is multiple valued in general, it too takes an index like the FUNCTION instruction:

```
G ← COMPOSE(r,F,n)
```

composes the relation r with the function F and chooses a function from the result using the index n.

The final way to create one function from another is to delete portions of it with the RESTRICT instruction. This instruction restricts the domain of function to a set contained in one of the set registers. For example,

```
F ← RESTRICT(G,A)
```

will load F with the portion of the function in G that maps from the contents of A.

A function register may be initialized to the null function with the CLEAR-FUNCTION instruction, or to the identity function with the IDENTITY-FUNCTION instruction.

The instructions in this section are the first ones that require nodes to send pointers in messages. An instruction like COMPOSE, for example, works by passing the contents of one register in each node backwards through selected links. Other instructions, such as INSERT, must actually allocate new cells and splice them into the existing network, by the message-wave mechanism described earlier.

Instructions like UNION-FUNCTIONS which do not send messages at all. Instead, they are implemented by register-to-register operations *within* each node. These instructions are similar to those in the first group (Set Operations).

Example: Relational Join

Given a genealogy network with parent-of and sex-of links, we wish to insert grandfather-of links between appropriate nodes. We assume that each person has only one sex and two parents (one of each sex).

```
A ← APPLY-REVERSE-RELATION(sex-of,{male})  ;A gets the set of all males.
F ← IDENTITY-FUNCTION()
F ← RESTRICT(F,A)           ;F is the identity function for males only.
F ← COMPOSE(parent-of,F,1) ;F is now the father function.
G ← COMPOSE(parent-of,F,1) ;G is one of the grandfather functions.
INSERT(G,grandfather-of)    ;build G into the network.
G ← COMPOSE(parent-of,F,2) ;G is now the other grandfather function.
INSERT(G,grandfather-of)    ;build your other grandfather into the network.
```

This example is a special case of the relational database equi-join operation. The code takes advantage of the fact that grandfather-of is a two-valued relation. Join on an n-valued relational would require repeating an operation n times. This is to be expected, since in the worst case the equi-join operation produces the Cartesian product of its inputs.

Group IV: Arithmetic Instructions

The arithmetic instructions manipulate functions from nodes to numbers. Numbers are just special nodes. The only thing that distinguishes them from ordinary nodes is that they are recognized by the arithmetic instructions. Thus node-to-number functions can be held in function-registers and manipulated by all of the function manipulation instructions mentioned above. They can also be manipulated by the arithmetic instructions.

The first set of arithmetic instructions are similar to the FUNCTION instruction. Like FUNCTION, they load a specified function register from a relation. The function instruction derives a single value from the potentially many-valued relation by choosing among them according to its index argument. The arithmetic instructions derive a single value by combining the values with an arithmetic operation. Thus,

```
F ← SUM(r,I)
```

will load F with the function that maps each node into the sum of all its r-related nodes. Another way of saying this is that it associates with each node a number, which is the sum of the nodes that can reached from it over r-links. The second argument to SUM indicates how to get a number from the node. In the example, I (for identity) indicated that the node itself is to be used as the value. This make sense, of course, only if these nodes are numbers. Otherwise an error condition would be flagged.

MAXIMUM and MINIMUM are two other instructions that require the r-mapped nodes to be numbers. These instructions have the same format as SUM, but instead of adding the numbers, they reduce the set to a single value by choosing either the largest or the smallest value.

AND and OR are classified as arithmetic instructions because they operate on and produce numbers. These instruction perform bit-wise logical operations on the binary representations of numbers. They have the same format as SUM, and produce a function in a similar manner.

These five instructions (SUM, MINIMUM, MAXIMUM, AND, OR) are just examples of plausible arithmetic instructions. Any function which turns a set of objects into a single number would make sense as an instruction. Any symmetric and associative arithmetic operation will do. There could be a MULTIPLY instruction, for instance. Asymmetric functions, like subtract, do not make sense in this context because it would not be obvious what should be subtracted from what.

This first class of arithmetic instructions operate by utilizing the fan out trees to actually perform the required arithmetic. They are thus similar to the pointer passing functions of the last section, except instead of selecting a single answer from those arriving at a fan out tree based on an index, the answers are all combined in some manner.

There is a second class of arithmetic instruction for which asymmetric operations make sense. These instructions combine two functions into a single functions, or to put it another way, they associate with each node a value that depends on other values already associated with the node. So, for example,

```
F ← FUNCTION-SUBTRACT(G,H)
```

will load F with the function that maps each node to the difference of the values of the G and H functions applied to that node. Similar instructions are FUNCTION-SUM, FUNCTION-MAXIMUM, FUNCTION-MINIMUM, FUNCTION-AND, and FUNCTION-OR.

This class of arithmetic instruction involves no message passing. These instructions are all executed as register-to-register operations at each node.

How To Connect A Million Processors

The most difficult technical problem in constructing a connection memory is the communications network. The memory's speed is limited by the bandwidth of the network. This bandwidth depends on the topology of the network, which is limited by physical layout and wiring constraints. Highly connected structures, such as the Boolean n-cube, are difficult or impossible to wire for such large numbers of nodes. Constraints on wiring density suggest simple tessellated structures, such as the grid or the torus. These grid-like structures are easy to wire, but the large average distance between nodes slows communication.

Instead of choosing either of these extremes, I have developed a compromise that allows us to take best advantage of the available wiring density. It is a family of connection patterns that spans the gap between the low-performance grid, and the unwireable n-cube. Given a set of engineering numbers, such as the number of pins on available connectors or the maximum wire density, we can choose from the family the highest performance connection pattern that satisfies the constraints.

A method for generating the family connection patterns is shown in figure 9. I illustrate here only the one-dimensional case. The two or three-dimensional layout is generated by repeating this pattern in each dimension independently. The first member of the family is

the torus. In two dimensions this is just a grid with opposite edges connected, as in the ILLIAC IV. [19] This pattern can easily be projected into a line, as shown. The second member of the family is generated from the torus by connecting each node to the node farthest away as shown. The nodes may be rearranged for efficient wiring by first twisting the torus and then folding it, so that each node is adjacent to the node half-way around the torus from itself. This pattern may now be projected into a line as shown.

Generating the Folded Torus

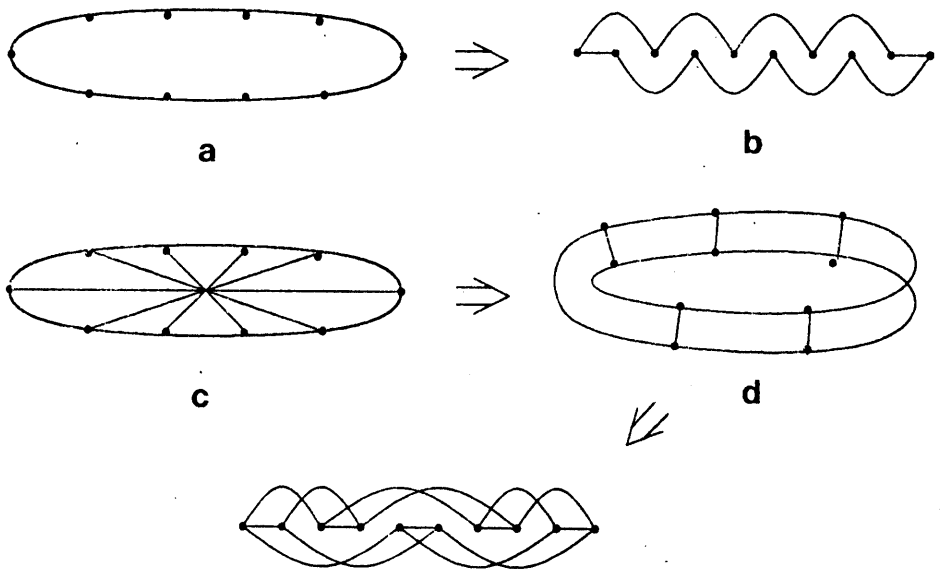

Figure 9.

This operation of connecting, twisting and folding results in a connection pattern with one half the maximum distance and twice the density of wires. The procedure may be repeated as many times as necessary to achieve an optimal tradeoff between performance and wireability. If the torus is twisted log(n) times, where n is the number of nodes, the resulting structure will be an augmented Boolean n-cube. The number of parallel wires in the connecting buses may also be varied, generating a two-parameter family of interconnection patterns.

The resulting connection pattern has the following desirable properties:

- **Uniformity.** The network looks similar from the viewpoint of each node.
- **Extensibility.** More nodes can be added by plugging more cells on at the edges.
- **A maximum wire length.** Short wires allow synchronous operation.
- **A maximum wiring density**, chosen to match available technology.
- **A maximum number of pins per module**, chosen to match available technology.

For an integrated circuit or a printed-circuit board the pattern would be repeated in two dimensions. It is also extendable to three dimensions if such a technology becomes available.

According to our initial calculations, the maximum performance network built with off-the-shelf 1981 components is a twice-folded torus with five-bit data paths.

What Can the Machine Do?

One goal of the proposed research is to formalize just what the connection memory can and cannot do. There already exists one well-worked-out formalism for describing retrieval operations: relational database theory. Codd's *relational calculus* allows queries to be described the form of a predicate calculus. The *relational algebra* provides a set of operations for computing these queries. **[4]**

We do not expect to convert artificial intelligence knowledge representations to relational databases, because they do not provide a natural way of expressing artificial intelligence knowledge manipulation. But relational database theory does address a well-specified set of problems that are similar to those that we must solve for semantic networks. I believe that relational database formalisms will provide theoretical tools for describing the operations of the connection memory.

The notion of *relational completeness*, for example, provides a measure of the expressive power of a retrieval language. If a machine can concurrently process all of the operations of the relational algebra, which is relationally complete, we know that it can compute any query that is expressible in the relational calculus. This gives us confidence that our system has no hidden weaknesses.

Comparison with Other Concurrent Architectures

A useful way to characterize the machine is to contrast it with other systems that are similar in form or purpose. Here is a list of such near misses, several of which have been important sources of ideas.

- It is not a way of hooking together a collection of general-purpose computers as in [19,7,11, 3,20,23,18,8]. The connection memory shares many features with these systems, such as extensibility, concurrency, and uniformity, but the individual processing elements in the connection memory are smaller. Since each connection-memory cell contains only a few dozen bytes of memory there can many more of them, allowing for a higher degree of concurrency. The penalty is that the connection memory is less general-purpose; it must be used in conjunction with a conventional machine.

- It is not a marker-propagation machine, as proposed by Fahlman. [6] The connection memory is able to execute marker-type algorithms, but its pointer manipulation capabilities give it additional power.

- It is not a simple associative memory. [15] The elements in content addressable memories are comparable in size to connection memory cells, but the connection memory's processing operations are far more general, due to its ability to communicate between cells.

- It is not a systolic array [14,13]. In the connection memory, cells may operate asynchronously. Uniformity is not critical; some cells may be defective or missing. The connection memory is also more flexible than a hard-wired systolic-array, although for problems that can be done on both it is likely to be slower. Systolic array algorithms can all be executed efficiently on the connection memory.

- It is not a database management machine like RAP [16] or CASSM. [5] They are designed to process a more restricted class of queries on a much larger database.

- It is not a cellular array machine [22,10] Like these machines, the connection memory has a regular repetitive layout, but unlike them it also has a mechanism for arbitrary communication.

The machine is designed for symbol manipulation, not number crunching. It does have limited parallel arithmetic capabilities because they are often useful in symbol manipulation, for example, in computing a score for a best-match retrieval. Similar

architectures may have application in numeric processing, but we do not at this time plan to investigate these possibilities.

What We Have Done so far

- We have specified an algebra for expressing network pattern matching operations, and we have shown that all expressions of the algebra can be efficiently evaluated on the connection machine. One result is that the machine can concurrently search a graph for an acyclic subgraph matching a specified pattern. This may be a first step toward a theory of the connection machine's operations.

- We have written several simulation programs of various portions of the machine. These simulations have allowed us to discover and correct weaknesses in the machine's instruction set. We have run a few simple test programs on the simulators, although we have not yet written a complete simulation of the machine.

- We have extensively simulated the communication network. We have used these simulations to measure the performance of various routing algorithms. Specifically, we have tested six different algorithms on a grid, plus one algorithm for a twice-folded torus. All of these algorithms performed well as long as the number messages in transit remained significantly less than the number of message buffers. Algorithms that used several buffers per cell performed best.

- We have designed a message-routing chip for the machine. This was mostly an exercise to give us some design experience, but we did work out circuit techniques which should be useful in the construction of an actual machine. Specifically, the chip included a crossbar and a novel incrementer/decrementer. We received chips, through MOSIS, in January. The chips function correctly, in spite of a design-rule error. We also learned things by measuring the timing of the actual chips that should allow us to make a faster chip the next time around.

We Plan to Build a Prototype

In 1967 the MIT Artificial Intelligence Laboratory commissioned the construction of the world's first 256K-word core memory. The cost was approximately half a million dollars, or about two dollars a word. The "old moby" is actually still in use, although it is now flanked by 256K words of semiconductor memory that cost literally one hundredth as much.

The proposed 128K connection memory will cost about as much per processor as the core cost per word. Part of this represents a one-time tooling cost, but by far the largest expense

is the fabrication of the chips. These fabrication estimates assume the low yields and short runs appropriate for a first-time project. If the architecture proves successful and is duplicated on a larger scale, the per-cell costs would drop dramatically. Fundamentally, a connection memory should only cost a constant factor more than a similar-sized semiconductor random access memory. If, say, half of the area of a connection memory chip is pointer memory, then storing a given amount of data would take twice as many connection memory chips as RAM chips. The RAM, of course, would only store the data, not process it.

We plan to design in detail a million-element connection memory, and then actually build and program one 128K slice of it. This is enough to to let us test the concept without needlessly replicating the inevitable mistakes of a first-time design. Because the connection memory is incrementally extendable, like ordinary memory, it would be possible to build a million element machine by simply plugging together eight duplicated sections, although we will probably never actually do this with this first machine. We will try, however, to actually solve the problems that would be encountered in constructing a larger version. Since packaging problems are significantly different for a larger machine, we will actually build the mechanical package for a million element machine. Address sizes, communication protocols and clock speeds will all be designed for a million cells.

According to our current plans, the million-element machine will fit into a single rack. The rack will contain eight card cages, four on the front and four on the back. Each cage will contain sixteen cards, each twenty-one inches wide by fourteen inches deep. One-hundred twenty-eight chips will be mounted on each card, in socketed sixty-eight-pin square ceramic packages. Each chip will contain sixty-four cells. The cells on a chip will share a single off-chip communicator, arithmetic unit and rule table.

Acknowledgments

Many of the ideas in this paper came directly from discussions with Tom Knight, Alan Bawden, Carl Feynman, Gerald Sussman and Hal Abelson. Scott Fahlman (through his thesis) and Ivan Sutherland (through a talk) started me thinking about the problem in the first place. Carl Feynman, Dan Weinreb, Neil Mayle and Umesh Vasirani wrote the initial simulations. For discussions, suggestions and encouragement I would also like to thank John Batali, Howard Cannon, David Chapman, Gary Drescher, Michael Dertouzos, Richard Feynman, Richard Greenblatt, Carl Hewitt, Neil Mayle, David Marr, Margaret Minsky, David Moon, Brian Silverman, John Taft, Patrick Winston and especially, Marvin Minsky.

1. Backus, J., "Can Programming be Liberated from the Von Neumann Style?", Communication of the ACM, Vol. 21 no. 8. (August 1978) 613-641

2. Brachman, R.J. "On the Epistemological Status of Semantic Networks" Report No. 3807, Bolt Beranek and Newman Inc., Cambridge, MA, (April 1978)

3. Browning, S. A. "A Tree Machine" Lambda Magazine, April 1980. Vol. 1. No. 2. pp. 31-36.

4. Codd, E.F. "Relational Completeness of Data Base Sublanguages" in Rustin, R. (ed) "Database Systems" Courant Computer Science Symp. Series, Vol. 6, Prentice Hall, 1972

5. Copeland, G.P., Lipovski, G.J. and Su, S.Y.W. "The Architecture of CASSM: A Cellular System for Non-numeric Processing" Proc. 1st Annual Symp. Com. Arch. 1973, pp. 121-128

6. Fahlman, Scott, "NETL: A System for Representing and Using Real-World Knowledge", MIT Press (1979)

7. Gritton. E.C. et all "Feasibility of a Special-Purpose Computer to Solve the Navier-Stokes Equations" Rand Corp. r-2183-RC (June 1977)

8. Hewitt, C. E., "The Apiary Network Architecture for Knowledgeable Systems", Proceedings of Lisp Conference Stanford. (August 1980) pp. 107-118.

9. Hewitt,C., Attardi, G. and Simi, M. "Knowledge Embedding in the Description System Omega" Proc. First Nation Conf. on A.I. (August 1980) pp. 157-164

10. Holland, John H. "A Universal Computer Capable of Executing an Arbitrary Number of Sub-Programs Simultaneously" Proc 1959 E.J.C.C. pp 108-113

11. Halstead, R.H., "Reference Tree Networks: Virtual Machine and Implementation", MIT/LCS/TR-222, MIT Laboratory for Computer Science, Cambridge, MA. (June 1979)

12. Koton, P.A., "Simulating a Semantic Network in LMS", Bachelor Thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, MA. (January 1980)

13. Kung, H.T. abd Lehman, P.L. "Systolic (VLSI) Arrays for relational database operations" Int. Conf. on Management of Data, May 1980

14. Kung, H.T. and Leiserson, C.E. "Systolic Arrays", In Itro. to VLSI Systems by C.A. Mead and L.A. Conway, Addison-Wesley, 1980, Section 8.3

15. Lee, C.Y. and Paul, M.C. "A Content-Addressable Distributed-Logic Memory with Applications to Information Retrieval" IEEE Proc. 51:924-932, June 1963

16. Ozkarahan, S.A., Schuster, S.A. and Sevcik, K.C. "A Data Base processor" Tech. Rep. CSRG-43, Comp. Sys. Res. Group, U. of Toronto, Sept 1974

17. Schwartz, J.T., "On Programming, An Interim Report on the SETL Project" Computer Science dept., Courant Inst. Math. Science., Ney York University. (1973)

18. Rieger. C. "ZMOB: A mob of 256 Cooperative Z80a-based Microcomputers" Univ. of Maryland C.S. TR-825, College Park, MD (1979)

19. Slotnick, D.L., Et.Al. "The ILLIAC IV Computer", IEEE Transactions on Computers. Vol. C-17, No. 8, (august 1978), pp. 746-757

20. Swan, R. J., Fuller, S. H., and Siewiorek, D. P. "Cm*--A Modular, Multi-Microprocessor" AFIPS Conference Proceedings 46. 1977.

21. Szolovitz, P., Hawkinson, L., and Martin, W.A. "An Overview of OWL, a Language for Knowledge Representation", MIT/LCS/TM-86, MIT Laboratory for Computer Science, Cambridge, MA. (June 1977)

22. Toffoli, Tommaso, "Cellular Automata Mechanics," Tech. Rep. No. 208, Logic of Computers Group, CCS Dept., The University of Michigan (November 1977)

23. Ward, S. A. "The MuNet: A Multiprocessor Message-Passing System Architecture" Seventh Texas Conference on Computing Systems. Houston, Texas. (October 1978)

24. Woods, W.A. "Research in Natural Language Understanding, Progress Report No. 2" Report No. 3797, Bolt Beranek and Newman Inc., Cambridge, MA, (April 1978)

25. Quillian, M., "Semantic Memory," in Minsky (ed.) "Semantic Information Processing," MIT Press (1968)