A Survey of Specialized Parallel Architectures Designed to Support Knowledge Representation

Daniel P. Miranker

AI TR87-43 January 1987

*This report describes work done by the author while at Columbia University.

This work has been supported by the Defense Advanced Research Projects Agency through contract N00039-82-C-0427, as well as grants from Intel Corp., Digital Equipment Corp., Hewlett-Packard Corp., Valid Logic Systems Inc., and IBM Corp., and the New York State Foundation for Advanced Technology. Their support is gratefully acknowledged.
A Survey of
Specialized Parallel Architectures Designed to Support
Knowledge Representation

Daniel P. Miranker

January 28, 1986

Columbia University
New York, New York

Abstract

While modern data processing systems have evolved from arithmetic calculating machines, knowledge representation schemes, developed for artificial intelligence programs, have evolved from cognitive models of intelligent behavior. Instead of the basic arithmetic operations used in data processing, knowledge representation schemes require extensive set manipulation, pattern matching and graph theoretic abilities. Thus, the implementation of artificial intelligence programs on conventional computers has suffered from a mismatch of basic abilities.

This paper surveys computer architectures that attempt to overcome this mismatch by developing computer organizations that intrinsically support the structure and basic operations required to manipulate artificial intelligence knowledge bases. Common to all these proposals is the introduction of large scale parallelism in the form of many simple processing elements.

This research has been supported by the Defense Advanced Research Projects Agency through contract N00039-82-C-0427, as well as grants from Intel Corp., Digital Equipment Corp., Hewlett-Packard Corp., Valid Logic Systems Inc., and IBM Corp. and the New York State Foundation for Advanced Technology. Their support is gratefully acknowledged.
# Table of Contents

1 Introduction 1

2 Knowledge Representation 2
   2.1 Logic 2
   2.2 Production Systems 3
   2.3 Semantic Nets 4
   2.4 Frames 6

3 Characteristics of Computer Architectures 7
   3.1 MIMD and SIMD 7
   3.2 Shared or Distributed Memory 8
   3.3 Granularity 9

4 NETL 10
   4.1 The NETL Architecture 10
   4.2 A Possible NETL Implementation 13
   4.3 Effectiveness of NETL 14

5 The Connection Machine 18
   5.1 The Connection Machine Concept 18
   5.2 Connection Machine Implementation 19
   5.3 Programming the Connection Machine 22
   5.4 Effectiveness of the Connection Machine 23

6 DADO 24
   6.1 The DADO Architecture 25
   6.2 The DADO Prototypes 26
   6.3 Programming DADO 28
   6.4 Low Level Systems Languages 29
      6.4.1 PPL/M 29
      6.4.2 PPSL and Parallel C 30
   6.5 Production System Execution 30
   6.6 Logic Programming 32
   6.7 Effectiveness of the DADO Architecture 33

7 Neural Models 33
   7.1 The Perceptron 34
   7.2 The Boltzmann Machine 34
      7.2.1 The Boltzmann Model 34
      7.2.2 Representation and Search in the Boltzmann Model 36
      7.2.3 A Symbolic Inference Engine, Based on the Boltzmann Model 37
      7.2.4 Prospects for Hardware Implementation 39

8 Summary and Conclusion 39
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>An Example Prolog Program</td>
<td>3</td>
</tr>
<tr>
<td>Figure 2</td>
<td>An Example Production Rule</td>
<td>4</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Example of a Semantic Net</td>
<td>5</td>
</tr>
<tr>
<td>Figure 4</td>
<td>The Generalized Structure of a Shared Memory Machine</td>
<td>8</td>
</tr>
<tr>
<td>Figure 5</td>
<td>NETL Hardware Organization</td>
<td>11</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Demonstrating a NETL weakness</td>
<td>15</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Finding Mothers-in-Law Using NETL</td>
<td>15</td>
</tr>
<tr>
<td>Figure 8</td>
<td>The Organization of a NETL Simulation Engine</td>
<td>17</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Fanout Tree Solution of the Buffering Problem</td>
<td>20</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Organization of a Connection Machine Processing Element</td>
<td>21</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Alpha Operator Applied to Xectors</td>
<td>23</td>
</tr>
<tr>
<td>Figure 12</td>
<td>The Beta Operator</td>
<td>24</td>
</tr>
<tr>
<td>Figure 13</td>
<td>The DADO1 Prototype Processing Element</td>
<td>27</td>
</tr>
<tr>
<td>Figure 14</td>
<td>The DADO2 Prototype Processing Element</td>
<td>27</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Software Layering for the DADO Architecture</td>
<td>28</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Functional Division of the DADO Tree</td>
<td>31</td>
</tr>
<tr>
<td>Figure 17</td>
<td>A Boltzmann Network for Computing XOR</td>
<td>36</td>
</tr>
<tr>
<td>Figure 18</td>
<td>The Structure of a Boltzmann Production System Interpreter</td>
<td>38</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Table Summarizing Machine Characteristics</td>
<td>40</td>
</tr>
</tbody>
</table>
1 Introduction

Since the early 60s, researchers have been developing computer programs to demonstrate artificial intelligence (AI). Though AI programs have been written for conventional computers and could be redeveloped for any of the parallel computers now proposed, the basic operations necessary for AI are quite different than those readily supported by general-purpose computers. General purpose computers have evolved from primitive, arithmetic calculating machines and have been optimized for basic data processing and scientific processing tasks. Rather than arithmetic operations, AI programs require extensive set manipulation, pattern matching and graph theoretic abilities. The development of new computer architectures better suited for executing AI programs has recently come into the forefront of computer architecture research. Indeed, a new term fifth generation computers, has been coined to describe these architectures.

Fifth generation research and development has focused on two classes of machines. First, specialized architectures have been developed for executing general purpose AI languages such as LISP or Prolog. A variety of LISP processors have been proposed [Moon 85, Myers 81, Batali 82]. Several have become commercially available as personal AI workstations. The Japanese fifth generation project has incorporated many of the LISP machine concepts into a Prolog processor [Uchida 82, Nishikawa 83]. It is expected that Prolog based AI workstations will soon be on the market. To facilitate the execution of general purpose AI languages, these computers contain, wide memory organizations with built in tag fields, circuitry to facilitate garbage collection and large fine tuned microstores. Nevertheless, these machines are very similar in organization to conventional, sequential computers.

The second class of architectures contains machines that are able to manipulate large amounts of low level data. This class includes machines that perform low level sensory processing [Lyon 81, Dyer 81] as well as machines capable of manipulating large databases [Ozkarahan 75, Murakami 83, Kitisuregawa 84, Su 75]. These machines employ a wide variety of novel architectural principles and use organizations that are radically different than conventional computers. The machines for sensory tasks often incorporate many of the principles of systolic design, (described below). The database machines have explored the use of sorting networks, intelligent switching networks, logic per track devices as well as associative processing methods.

Neither class of architecture, though each is well suited for their respective tasks, represent an effort to explore alternative architectures capable of higher level reasoning. Other than the work towards building fast LISP and Prolog processors, which are classified as general purpose AI systems, there are only a few
proposals for machines whose organizations attempt to exploit the intrinsic structures created by AI methodology. For that reason, this paper seeks to highlight attempts to develop architectures that directly support AI knowledge representation.

Organized into seven sections, the paper first reviews concepts in knowledge representation and parallel computer architecture and then describes each of four proposed computer architectures. The last section contains a comparative discussion of the architectures and conclusions.

2 Knowledge Representation
The two main themes of AI problem solving are knowledge representation and search. Barr and Feigenbaum [Barr 81] identify four types of knowledge:

- **Objects.** Facts about physical things, their composition, their color, their location. i.e. nouns and the adjectives that describe them.

- **Events.** The interaction of objects, such as "the car crashed into the bus".

- **Performance.** How to do something. Also known as procedural knowledge. For example, the steps required to make an apple pie.

- **Meta-knowledge.** Knowledge about knowledge. For example if it's known that the knowledge base is complete and it has been determined there is a single solution to a problem then it may be concluded that the solution is unique.

Knowledge representation and search are not independent issues. A particular problem may be more easily solved if a particular search method is used, and a certain method of knowledge representation may represent the problem more easily than another as well as more efficiently support the operators required by the search strategy. For a particular problem, different combinations of the methods for knowledge representation and search may yield more or less effective means for solving the problem.

AI researchers have employed four primary methods of knowledge representation. These are described below.

2.1 Logic
Logic based formalisms are the most cognizable method of knowledge representation. Knowledge in logic based representations is formed by small, independent declarative statements denoted by first order predicates. This is best illustrated by the following Prolog [ClocksinMellish 81] fragment. (See Figure 1.) A collection of simple facts, called *ground literals*, is entered into the environment. A predicate using logical variables succinctly defines grandmothers as females who are parents of parents. If the Prolog interpreter is asked to enumerate all the grandmothers, the definition of grandmother is
looked up, and the logical variables in its definition are substituted with the other expressions in the environment. For each consistent and complete set of substitutions the interpreter returns the value bound to the variable X. The process of searching for a consistent set of substitutions is called resolution or unification [Nilsson 80].

parent(betty, john).
parent(john, frank).
parent(terry, dan).

female(betty).

grandmother(X):= female(X), parent(X,Y),parent(Y,Z).

Figure 1: An Example Prolog Program

New knowledge in logic based representations may easily be added incrementally. In the example above, more names of people and their relationships may be added without considering the data already present. Further advantages of logic based schemes are the ability to use formal methods to verify consistency and completeness of the knowledge base as well as the capability of formal proof techniques to derive new knowledge.

The same lack of structure that facilitates knowledge acquisition in logic based systems is a great disadvantage for the search methods employed in inferencing. Since little or no structure is apparent, search methods must either blindly consider a combinatorially explosive number of search paths or rely on added information supplied by the user to direct the search [Sussman 72, Hewitt 69].

2.2 Production Systems
In general, a production system [Newell 73, Rychener 76, McDermott and Forgy 78] is defined by a set of rules, or productions, which form the production memory (PM) together with a database of assertions, called the working memory (WM). Each production consists of a conjunction of pattern elements called the left-hand side (LHS) of the rule, along with a set of actions called the right-hand side (RHS). RHS actions include updating the WM, by either adding or removing facts, and affecting external side effects, such as reading or writing an I/O channel.

In operation, a production system repeatedly executes the following cycle of operations:
1. Match. For each rule, compare the LHS against the current WM. Determine if the WM satisfies the LHS. Each set of WM elements satisfying a rule's LHS is called an
instantiation. All instantiations are collected to form a conflict set of rules.

2. Select. Some subset of the conflict set is chosen according to some predefined criteria. In practice a single instantiation is selected from the conflict set on the basis of the recency of the matched data in the WM, as well as syntactic discrimination.

3. Act. The actions in the RHS of the selected rules are executed.

An example rule using the OPS5 production system language [Forgy 81] is shown in figure 2.

```
(p categorize-job-sizes
 (message 'job <x> 'size <y> 'status new)

 (class-definition 'size <y> 'class-name medium)
 ->
 (make job 'job-name <x> 'class medium)
)
```

This rule says if
there is a WM-element in the system representing a message about a new job,
and the job's size matches the class definition for medium size jobs,
then
create a new WM-element tagging the job with the class name medium.

Figure 2: An Example Production Rule.

Production systems have many of the advantages of logic based systems in that knowledge may be added incrementally, their strong syntactic constraints facilitate automatic generalization [Laird 84] of new knowledge, and formal proof techniques may be used to verify completeness and consistency of the knowledge base. Production systems permit only a single level of substitution, i.e., a variable must be bound to a ground fact not to another expression. This restriction limits the potential combinatorial explosion possible in logic systems, but simultaneously forces the programmer to add more structure in his programs to carefully direct the search process.

2.3 Semantic Nets
Quillian [Quillian 68] is credited with the first description and use of semantic nets. A semantic net is a graph representation of knowledge. Although there are many variations of the semantic net formalism, typically the vertices of the graph represent objects, while edges connecting two vertices represent a relationship between the connected objects. The knowledge in a semantic system resides not so much in the vertices but in the pattern of interconnecting links.
Using semantic nets to describe a goldfish named Charley we may have a vertex labeled Charley and one labeled goldfish. (See Figure 3.) An edge connecting the two would be labeled *is-a*, to indicate the proposition that Charley is a goldfish. We may extend the taxonomy to include vertices for fish and animal. These vertices are also connected with *is-a* links. Such a collection is known as an *is-a hierarchy*. Other properties may be attached to nodes by links of other types. The example indicates that fish live in water and have fins and that Charley is gold colored.

![Figure 3: Example of a Semantic Net](image_url)

Attributes are represented by unique vertices. Objects with a common attribute have individual links to a shared, unique vertex representing the attribute. A system may find information about Charley by traversing the links in the graph. By following the appropriate path through the graph, we may find all the goldfish in the system or all the gold colored objects.

An advantage of semantic nets is that not only inferencing based on property inheritance may be performed by traversing the edges between nodes, but reasoning by analogy may also be performed by comparing the structural similarity of parts of the network [Winston 72]. Suppose that beside Charley the fish, the network contains information about Eliza the dolphin. The network representing Eliza would be nearly identical to the network for Charley, for both live in the water and have fins. It would then be
possible for an AI program to notice the similarity and generalize that all animals with fins live in water.

2.4 Frames

Frames [Minsky 75] are similar to semantic nets in that it is a network representation of knowledge, where related knowledge is be linked together by edges. However, the vertices are no longer simple objects, but frames. A frame is a typed structure. Fixed positions within the structure form a set of closely related objects and attributes. When a new frame is created, space is allocated according to an initial template. The template reserves a slot for each data item to be placed in the frame. Often these slots contain initial default values.

Another distinguishing feature of frames is *procedural attachment*. Slots in a frame may also be filled with pointers to procedures. Depending on its attachment, a particular procedure may be called when a frame is created or a value within the frame is changed.

Let's examine, for example, a set of frames developed for the sport fisherman. If you were fishing and you just caught a trout, a frame would have to be allocated for the new trout. Since brown trout is the most common species, the frame would have an initial "species" value saying you just caught a brown trout. The frame system, upon creating the frame, would discover and execute a procedure encoding the steps necessary to reel the fish in. Once you had landed the fish, the species would be verified and the species slot updated accordingly.

A well known example of frame based representation is the SAM system developed by Schank [Schank 77] to understand stories that take place in a restaurant.

AI systems may be classified by identifying which form of knowledge representation they employ. Most will fall into one of the four knowledge representation paradigms described above. The next section is an overview of the characteristics of parallel computer architectures. The AI machines subsequently described represent efforts to map the intrinsic structures of a particular method of knowledge representation to a computer organization that exploits the structure in a way to rapidly execute the primitive steps required for problem solving.
3 Characteristics of Computer Architectures

Many efforts have been made to form a taxonomy of parallel computer architectures [Flynn 72, Schwartz 83, Seitz 84]. These efforts have been frustrated by the lack of a succession of properties with which to form a hierarchical description of machines. There are, however, a variety of issues, each with a set of solutions, that may be used to form a multidimensional space of machine types. These issues, to-be discussed below, tend to be independent of one another. Although the following discussion generalizes trends in architectures, a sufficiently thorough investigation of the literature probably would uncover a machine proposal for every possible combination of properties.

3.1 MIMD and SIMD

An early attempt at forming a taxonomy of computer architectures, due to Flynn [Flynn 72], formalized the notion of single instruction streams, multiple instruction streams, and single and multiple operands. Single instruction stream implies a single instruction fetch unit delivering instructions either to a single processing element (PE) or multiple PEs. If a single PE performs a data fetch and executes the instruction then it is said to be operating on a single datum. This type of machine, typical of conventional computers, is called a single instruction stream, single data stream architecture (SISD). If multiple PEs receive and execute the instruction on independently fetched data, the machine is said to be operating on multiple data. Machines of this type, typical of many array processors on the market today, are known as single instruction stream multiple data stream architectures (SIMD).

If multiple PEs independently execute different instruction streams, the architecture is said to be a multiple instruction stream architecture. If the PEs also operate on different data the architecture is known as a multiple instruction stream, multiple data stream architecture (MIMD).

A fourth class of machines, multiple instruction stream single data stream is sometimes defined (MISD). Cellular or systolic arrays [Kung 76] are sometimes placed in this category. A systolic architecture is one where a collection of cells are connected in a regular topology and data moves in pipeline fashion through the collection of cells. At each step, as data flows through the cells, a function is applied to the data and the result advanced to the next cell.
3.2 Shared or Distributed Memory

Another characteristic that distinguishes parallel computers is whether the memory is distributed to the processors or resides in a common, shared pool.

Shared memory architectures contain a collection of processors, a collection of memory modules and an interconnection network between them. (See Figure 4.) Architectures that use shared memories may communicate by reading and writing data into predefined areas of the memory. Thus, communication between processors appears to be as simple as accessing memory. Further, since all processors may read and write anywhere in the memory, then any processor may communicate directly with any other. By fully connecting the processors, developers of parallel algorithms need not consider the topology of the communication channels, simplifying the development of algorithms for this class of machine.

![Diagram of a shared memory machine](image)

**Figure 4:** The Generalized Structure of a Shared Memory Machine

The organization of the interconnection network is the focus of a great deal of study [Wu 84]. The many design alternatives that must be considered for the interconnection networks may itself be the subject of a survey [Thurber 74]. Briefly these alternatives include the topology of the interconnect, the amount of intelligence and buffering at the switch points and whether the switching technique should be packet switched, circuit switched or message switched.

In all cases, the switch in this class of machine presents two major problems. The complexity of the
switch is at minimum $O(n\log n)$ switch contact pairs, where $n$ is the number of processors. For all but the smallest machines, the switch becomes very large and expensive. Further, the switch increases the distance between the processors and the memory. To compensate for the delays in memory access time introduced by the switch, designs for shared memory machines are now incorporating cache memories as part of the processor modules. Though effective in reducing memory latency, using multiple independent caches introduces the coherence problem [Dubois 82].

Distributed memory architectures are much simpler to implement than shared memory architectures. In distributed memory schemes, each processor is directly associated with a portion of memory that only it may access. A PE has no direct knowledge of the data in another PE. Generally PEs are connected by fixed communication paths. Processors may exchange data only through some form of message passing, performed along direct PE to PE communication links.

The study of the topology of the interconnect has drawn great interest. The interconnection may take many forms; tree's, n-cubes, meshes, toroids, to name a few. Each topology has its strengths and weaknesses with respect to different classes of algorithms, as well as the ability to construct the interconnection circuitry.

Other properties of distributed memory machines related to communication include, the method of communication supported by the interconnections as well as how much circuitry a PE should have to support that communication method. Communication may be synchronous or asynchronous, strictly to the nearest neighbors or packet switched, performed by the processor or by a separate I/O section.

### 3.3 Granularity

The granularity of a parallel processor is measured in many ways. Granularity takes into consideration the number of processors, the size of the processors, the total amount of memory and the amount of memory assigned to an individual processor. Machines are classified in three categories: fine, medium and coarse. Fine grain architectures contain many PEs on a single chip. In today's technology this implies very simple, bit wide data paths and very modest memories, on the order of a thousand bits per PE. Proponents of fine grain architectures call for machines with millions of PEs. Due to the modest storage capacity of each node, machines of this kind are necessarily SIMD machines. In coarse grain machines the processors are very large with great functional and throughput capacities. A coarse grain computer would have at most a few 10s of PEs. If a coarse grain architecture were based on distributed memory, each PE would be associated with megabytes of memory. Medium grain architectures are all
those in between. Proposals may call for 50 to 10,000 PEs, each composed of conventional microprocessor chips, and from several thousand to several hundred thousand bytes of memory per PE. Sometimes the fine grain category is called very fine grain, and the medium category is broken into two categories, fine and medium, that have no clear demarcation between them.

Each of the next four sections describe a machine proposal along with a discussion of the AI paradigm and how the different architectural characteristics support the AI paradigm. All the proposals advance a notion of intelligent memory, that is a distributed memory organization as well as fine or medium granularity.

4 NETL
A semantic net formalism, called NETL, was developed by Fahlman [Fahlman 79], and later refined by Touretzky [Touretzky 84]. A major contribution of the NETL system was to determine correct and simple ways to represent such concepts as virtual copies, a method by which multiple objects could share descriptive information, and exception handling, a method by which multiple objects could inherit slight variations of shared descriptive information. An essential goal for NETL was to create a semantic net formalism that could ultimately be captured directly in very simple hardware in a way that would allow parallel execution of retrieval searches. At all times the representation methods chosen for NETL were constrained by the goal to create a parallel computer to expedite queries. The primary types of retrieval supported by NETL are property inheritance, transitive closure, and set intersection [Fahlman 83a]. Several prospective hardware implementations of NETL were cursorily explored in an appendix of the thesis. Later, a detailed design of a NETL machine was developed [Fahlman 83b] but never built.

4.1 The NETL Architecture
As in semantic nets, NETL is composed of a large collection of object nodes, at least one for each object to be represented, and a large collection of bidirectional links organized so that each object node can be linked arbitrarily to any number of other nodes in the system. Signals may be propagated along arcs from many nodes simultaneously. Multiple signals arriving at a single node are merged together with a logical OR. Therefore NETL physically supports many to many mappings of nodes to nodes.

The nodes of a NETL machine are very simple. Each contains a unique id-number for addressing purposes, 32 bits of state and a bit wide boolean logic unit. The state bits are composed of 16 write-once bits that can hold type information for the node, and 16 read-write marker bits that serve as the short term scratch pad memory for processing search queries. A bit can be set by the controller either directly by the
state of an incoming link or by the result of a Boolean operation on two other bits within the node.

The links are formed by link nodes. Each link node has state bits as in object nodes. Associated with each link node are four wires. The node type is expressed by connecting a parent wire to a node containing the link type information. The two objects in an assertion are connected by a pair of wires called A and B. A fourth wire is used to represent the context of the assertion. (See Figure 5).

![Diagram of NETL Hardware Organization]

**Figure 5:** NETL Hardware Organization

All the nodes sit on a common bus wired to a conventional computer that serves as a control processor. The control processor can broadcast instructions to either an individually addressed node, or to a set of nodes based on their internal state. A NETL machine is therefore a fine grain, SIMD machine.

Instructions for the object nodes of a NETL machine instigate the following actions:
- address a single node or a set of nodes based on their internal state
- transfer a bit to or from a marker bit and a link
- or perform a simple Boolean manipulation of the internal state.

An instruction may for example, address all PEs with bit 0 on, set bit 1 to, bit 2 AND bit 3. Instructions also are provided to address a set of nodes and have them sequentially report their state to the controller. Link nodes can be individually addressed and their contact points set to create new links between object
nodes.

To perform queries, NETL uses a method known as marker passing. The node or nodes that form the starting point of a search are marked with a token. The token is then propagated along a series of links. As the tokens pass through the nodes in the graph, the nodes are marked by setting a particular state bit.

Suppose Charley the goldfish in the example above is represented in a NETL machine and we wish to determine his color. The search procedure is as follows: The node representing Charley is addressed and marked by setting one of the state bits. All the is-a links are activated. On each cycle the token, represented by the set state bit, is propagated up one more layer in the arcs of Charley’s is-a hierarchy, leaving a trail of marked nodes. Once the token has propagated to the top of the hierarchy, the has-color links are activated. The machine is then told to attempt to propagate a new token from the marked nodes along the now activated has-color links. The controller asks the NETL machine to report the id of any nodes now marked by a has-color link. Since goldfish is the only marked node in the system with a has-color link, that has-color link is the only one to propagate a token. The controller receives the id and looks in its own symbol table to discover that Charley is gold.

The above algorithm shows how property inheritance is performed in a NETL machine. The time required is on the order of the longest chain in the is-a hierarchy. In a sequential machine, if we know a priori what attributes we will be searching for, an indexing scheme can be used to organize the data to perform such a query in constant time. But, if this is not known, or if there are a sufficient number of attributes that indexing schemes become infeasible then a full sequential search of the database may be necessary.

Another important feature of an AI machine is the ability to recognize an object based on a number of features. Suppose for example we wish to identify an object that has fins, is gold and is an animal. This query is computed by the set intersection of those objects displaying each of the features. To perform this in NETL, the node representing each feature would be marked. Sequentially for each feature, a token would be propagated to all nodes representing an object containing that feature. The nodes would be marked with a specific marker bit for each feature. Once markers had been propagated for all features, the node having all the corresponding marker bits set represents the object being sought.

The fourth wire of the link nodes, the context wire, makes it possible to gate the propagation of markers depending on the context. For example, if we need to deduce the color of leaves, the context wire may gate the has-color links so that if it is summer, leaves may be determined to be green, but if it is fall they
may be a variety of colors. To make further use of context, the system must easily and correctly handle exceptions to such general statements as all animals with fins are fish, except dolphins. Fahlman does this by adding a link type called a cancel link, which in combination with careful algorithms [Touretzky 84] will inhibit the propagation of a “fish-marker” and with it some subset of its properties when we are discussing dolphins.

4.2 A Possible NETL Implementation

Fahlman asserts that a million element NETL system is 10 to 20 times the size of the largest, (1978), AI knowledge base and should be sufficient to store “enough knowledge for substantial expertise in ... specialized domains” [Fahlman 83b] though not necessarily sufficient to store enough knowledge to perform common sense story understanding. With this in mind, and by trading off physical hardware limitations and performance requirements, Fahlman was able to design a million element NETL machine. Though the machine was never built the design was not physically demanding. It needed only 7000 components, 4000 of which were 64K RAM chips.2

The most demanding aspect of the design and construction of a NETL machine is the switching network. The switching network must be capable of incrementally creating a large number of permanent links between nodes. The nodes themselves are sufficiently simple to enable many hundreds to be placed on a single silicon die. However, the pinout limitation of even the most aggressive packaging technology can provide connections for only a small number of nodes on a single die.3 Fahlman solves this problem in two ways. One is the use of probabilistic arguments to create a much simpler and smaller switching network than was first assumed to be necessary. The second observes that the parallelism available so effectively solves the queries that the switch connections may be multiplexed in time so a machine built around a smaller but time multiplexed switch will still easily match human performance.

The switching requirements of NETL suggest that it is necessary to use a cross bar switch, the most general switching network. With a cross bar, a new link may be added at any time without any existing links blocking the establishment of a new one. Cross bar switches require O(n^2) contact pairs, which is impractical for a NETL machine where n must be at least a million. It has been shown that networks may

---

1 The algorithms presented in Touretzky’s thesis supersedes those presented in [Fahlman81 81]

2 The technology available in 1979.

3 1985 technology permits several hundred thousand transistors to be placed on a chip yet only 100 to 200 signals may be connected externally
be constructed from $O(n \log n)$ contact pairs such that any mapping may be embedded, but the links must be known a priori and a particular mapping of links to hardware must be constructed. If such a network were used in a NETL machine, each time a link is added, the entire network might have to be reorganized. Fahlman asserts links are added to the NETL network too frequently for such networks to be practical for a NETL machine.

Fahlman instead adopts the notion of a seldom blocking network, which permits the addition of new links most of the time. It has been shown that such a network may be built using $O(n \log n + \log 1/P_b)$ [Fahlman 80] contact pairs, where $P_b$ is the probability of being incapable of adding a new link.

By itself, a seldom blocking switching network is still too large to build a million element machine. To overcome this problem, Fahlman asserts that he is only interested in building a machine capable of human like performance, implying a response time of roughly 0.1 to 0.3 seconds. He calculates that a typical query requires 20 to 60 marker propagation cycles; therefore, a machine that performs a marker propagation cycle in 5 milliseconds is adequate. In comparison to the switching speed of digital logic 5 milliseconds is quite long. The final NETL design proposes that one thousandth of the switch be built, and that the switch be time multiplexed a thousand fold. Such a switch is both physically realizable and attains the desired performance.

4.3 Effectiveness of NETL

Though the NETL hardware can capture the NETL semantic net paradigm, the paradigm itself is unable to compute certain kinds of queries in parallel. Fahlman uses the example of determining all the sons whose fathers hate them. To illustrate this, consider the following database, its associated NETL representation and the associated query. (See Figure 6.)

We can activate the father links and mark all the nodes representing sons. Similarly we can activate the hate links and mark all the nodes representing hated objects. The nodes containing both markers, (consider node c), are both sons and “are hated”, but we don’t know if they are hated by their own fathers. By doing such an operation, we can reduce the candidate set of sons, but each hated son will have to be queried sequentially to see if it is his own father who hates him.

Adopting database terminology, Fahlman identifies the operations for which NETL fails as joins [Fahlman 82]. However, that claim is too general. In fact, simple join queries in the form of joining on a single common attribute can indeed be done. It is only more complicated joins, such a cyclical joins
Figure 6: Demonstrating a NETL weakness

[GoodmanShmueli 82], and joins in multiple attributes, that cannot be processed in parallel. Consider the following database and representation as well as the query to determine the set of mothers-in-law.
(See Figure 7.) We may mark all those people who are married, then look for their mothers.

Figure 7: Finding Mothers-in-Law Using NETL

The source of the restriction is the limited amount of information passed between nodes. A marker is a single bit, and even that information may be occluded when several markers collide at a node at once and they are simply ORed together. One solution to the latter problem is to define markers as numeric
quantities rather than single bits. Then if two marker collide they may be combined algebraically. Fahlman briefly describes a machine called Thistle with this property [Fahlman 82]. But no results for that paradigm have been published at the time of this writing.

Other questions about NETL remain to be answered. Nothing has been published indicating the effectiveness of the complex piece of hardware designed to implement a NETL machine. Fahlman concedes that it is not feasible to directly build a switch capable of supporting a million node machine, but suggests that one thousandth of such a switch be time multiplexed a thousand times. The justification for this tradeoff is that the switch becomes simple enough to build, but still able to solve problems no more slowly than human beings. However, a variety of space time tradeoffs can be made, and there is no study indicating that the particular tradeoff selected is optimal. In fact the following alternate implementation suggests that Fahlman’s implementation may not be optimal. At minimum, much more study is necessary.

Consider taking the tradeoff to the extreme. Let’s build one millionth of the switch and time multiplex it a million times. In other words, we will build a large table of all the switch points and look them up one at a time as we need them. Such an implementation would contain no parallelism, so we will refer to it as a simulation engine. Such a simulator could be built from a large table indexed by node id. Each row of the table would contain four node id’s each representing the node at the end point of each link. Each row also would contain the state (marker bits) normally associated with a node.\(^4\) Cycling through a million element table for each marker propagation cycle even with specialized hardware clearly does not compete with the proposed NETL machine. If we assume 100nsec. memory access times, to simply read all entries of the table would require 0.1 seconds.

It is common for graph algorithms for conventional computers to maintain a list of the currently active vertices. We can similarly extend the simulation engine by keeping a set of lists, each one containing the node ids of those nodes with a particular marker bit set. When propagating a marker rather than cycling through the whole table, only those nodes in the list must be retrieved. (See Figure 8.) The NETL instructions, “place marker a on outgoing link 2, set marker b to the value on your incoming link,” would be executed by taking the list of nodes with its a marker set and using it to index the table. The node ids output from the table then would be written to a new list representing nodes with their b markers set.

\(^4\)One more state bit may be needed to simulate the ORing of colliding markers
Figure 8: The Organization of a NETL Simulation Engine

The switching lookup table and marker state bits require roughly 1,800 64K RAM chips; each of 1 million rows contain 4, 20 bit fields to represent links and 33 bits of state. The trivial worst case for the lists of marked nodes is that all nodes are marked with all markers.\footnote{Clearly the worst case in practice is a fraction of that.} This requires 5,100 64K RAM chips. Only a handful of MSI logic parts are required to complete the design. With realistic assumptions about the number of nodes marked in the worst case, the simulation engine will be much simpler than Fahlman’s proposed NETL machine.\footnote{It is worth noting that the simulation engine consists almost entirely of memory which, since the time of Fahlman’s writing, has scaled 16 fold, while packaging has only allowed at most a four fold expansion of I/O ability. Using 1985 technology, the simulation engine would be even smaller compared to Fahlman’s proposal, and as VLSI continues to scale the discrepancy in size would increase.}

With appropriate pipelining, the simulation engine would be limited by memory access time. The simulation engine could propagate a single marker in 100nsec; Fahlman’s suggested implementation requires 5msec. per marker propagation cycle. This implies that if less than an average of 50,000
markers are propagated per marker passing cycle, (i.e. the available parallelism available in a NETL query is less than 50,000,) then the simulation engine will outperform Fahlman's proposal in both time and space. Since 50,000 marker propagations implies a significant portion of the nodes must be actively propagating markers, 5% on average, the actual communication requirements of a NETL must be carefully calculated before drawing conclusions about the adequacy of the design.

In summary the NETL paradigm permits parallel manipulation of AI knowledge bases in semantic net form. A strength and weakness of NETL is that searches are performed by communicating a minimal amount, a single bit, of information along each link. Although most searches can be performed in parallel, the limited communication between nodes forces certain queries to be solved sequentially. It is feasible to build specialized hardware to directly support NETL, however, careful determination of the true amount of parallelism displayed by NETL systems must be made before the adequacy of a particular hardware implementation can be evaluated.

5 The Connection Machine

5.1 The Connection Machine Concept
The original goal of the Connection Machine project was to develop a piece of hardware to support semantic net operations, yet overcome the deficiencies of the NETL Machine. Specifically, the developers wanted to create a machine capable of setting up arbitrary bidirectional links between nodes, with each link capable of passing a message of arbitrary length. These properties would give the Connection Machine sufficient generality to overcome the limitations of the NETL machine. The Connection Machine retains the fine granularity of the NETL machine. That is, each object in a knowledge base is mapped to a single very simple PE. However, the point to point switching network used in the NETL machine has been replaced by a packet switching network.

From the perspective of a computational model, the intent was to make communication between any two PEs logically equivalent. The first proposal describing the Connection Machine [Hillis 82] implied simple, fine grain, MIMD processing elements capable of supporting a packet switched communications network. A packet switched network is a network where the messages contain their destination address. The messages may have to pass through intermediate nodes, however routing may be performed as an

---

7Small amounts of coarse grain parallelism in the form of memory interleaving could significantly improve the performance of the simulation engine at only modest hardware expense, see [Miranker-impreparation]
operation local to each node in the network. A PE assembles a message complete with destination address and places it in the network. Regardless of the address, the network will insure that the message will arrive at its destination. The packet network accomplishes the goal of making any two links logically equivalent. Though packet switching networks are usually associated with large MIMD computer networks, using engineering ingenuity and simulation methods, a large, 64K PEs, SIMD based Connection Machine has been built by Thinking Machines Inc.

In packet switching networks an arbitrary number of packets may arrive at a single node. The designers of the network must determine the amount of memory that will provide sufficient buffer space for the arriving packets. This is a central problem for the Connection Machine since it seeks to be a very fine grain machine. An algorithmic alternative to providing sufficient buffering is for a node to simply refuse additional messages when its buffers become saturated. Nodes adjacent to the saturated node must then reroute any messages destined for the saturated node, even if the first step of the new route is counterproductive. However, when developing a routing algorithm for a packet switched network, it is important to show that the network will converge, i.e. all packets eventually will arrive at their destination. The above alternative to the buffering problem, which was adopted for the current Connection Machine, makes it difficult, if not impossible, to show convergence. A major contribution of the Connection Machine project that solves both the buffering and convergence problems, is the adaptation of the idea of a fanout tree to packet networks.

The notion of a fanout tree stipulates that a single node may communicate only with a fixed number of other nodes. If the number of desired connections is greater than the permitted fanout, the node must use intermediate nodes to amplify the number of permitted connections. (See Figure 9.) The allowable fanout of a single node is balanced against the memory capacity of a node. Before a pair of nodes may communicate, an allocation mechanism must inform them of their respective network addresses. Additionally, the allocation mechanism must check if the new link will exceed the fanout of a node. If so, a new level must be added to the fanout tree. Thus, the extra nodes in the fanout tree guarantee sufficient buffering area to accommodate the maximum number of arriving messages [--- 85].

5.2 Connection Machine Implementation
The focus of the Connection Machine project was to develop as large a machine as possible. Though the original proposal for the Connection Machine suggested a MIMD implementation, a 64K node SIMD machine has been built. The reason for the SIMD based architecture is that SIMD PEs share a single
Figure 9: Fanout Tree Solution of the Buffering Problem

instruction fetch mechanism and program store. The resulting PEs are much simpler and more compact than MIMD PEs would have been. Using simulation arguments, it can be shown that a SIMD based machine can execute the same programs as a MIMD machine at a cost of a constant factor. It was assumed the branching factors of applications considered for the Connection Machine are sufficiently small that a SIMD implementation is more effective than a MIMD implementation. This assumption remains to be proven for AI applications.

The PEs of the Connection Machine are very simple. The data paths are 1 bit wide. The PEs contain a bit wide ALU and 8 bits of internal state in the form of general purpose bit flags. 4K bits of memory are associated with each PE. (See Figure 10.)

The primary component of the Connection Machine is a semicustom gate array chip containing the logic for 16 PEs and 1 router circuit. The router supports the packet switching network. Associated with the chip are 4, 4K x 4 static memory chips. 4K copies of this ensemble, 20K chips, form most of the circuitry of the Connection Machine. The physical volume of the machine is about a 5 foot cube. The machine dissipates 12,000 watts.

The 4K routers are connected by a 12 dimensional n-cube. In addition to the n-cube connections the PEs are connected in a two dimensional mesh called the NEWS network (for North, East, West and South). The NEWS connections do not involve the router and are used for diagnostics and the development of low level vision algorithms.

Instructions are broadcast to all PEs simultaneously and executed in lock step, synchronized to an external 4 MHz clock. Most SIMD machines have the ability to selectively disable processors from executing
Figure 10: Organization of a Connection Machine Processing Element

instructions. An innovation in the design of the Connection Machine is instead of a single enabling bit, typical of most SIMD machines, the Connection Machine has a condition field in its instruction format that may select any flag from the flag set as the enabling bit, as well as the polarity of the bit. A PE executes an instruction only if the the contents of the specified flag match the polarity of the condition. This feature greatly facilitates mimicking a MIMD machine. For example, given the program:

;Program to set c =max(a,b)

    a,b,c int:

    where a>b
        c:= a;
    else c:=b;

In a parallel MIMD machine, each PE would execute a conditional jump and an assignment. A SIMD machine such as the ILLIAC IV would compare a and b, disable a subset of the PEs according to the result of the comparison and then do the first assignment. The ILLIAC would then have to reenable all the PEs, disable the complementary subset and do the second assignment. The effect of the Connection Machine’s condition field is to reduce the number of primitive steps required to execute branches. To execute the above program the Connection Machine performs the comparison. The result is stored in a conditional flag. The first assignment instruction is broadcast with its condition field set so it is executed
only in PEs where a is greater then b. The second assignment instruction is broadcast with its condition inverted from the first, so it executes in those PEs where b is greater then or equal to a. No cycles are expended explicitly enabling and disabling PEs. The total number of instruction cycles is much closer to the number of cycles that would be required by a MIMD machine executing the same program.

5.3 Programming the Connection Machine

Programming language development for the Connection Machine has taken two paths. Connection Language 1 (CL1) [Bawden 84a], and its follow on Connection Graph Language (CGL) [Bawden 84b], maintain the expanded view of the Connection Machine as a graph oriented, MIMD, message passing machine. CM-LISP supports a much simpler view more closely reflecting the fine grain SIMD implementation.\(^8\)

All of these languages are LISP based. Connection Machine code is embedded within conventional LISP. A set of special declarations distinguish host code from Connection Machine code.

CL1 and CGL contain explicit functions for allocating new cells and setting up links between them. The amount of memory at each Connection Machine PE is slightly more than typically found in the register set of a general purpose processor. As a result, the size of a process executed at a node is limited to roughly a single function. Processes communicate with each other with explicit communication primitives. In a CL1, a small collection of data becomes tightly associated with a single function. This data may be accessed only by sending the function a message. In turn the function may have effects beyond its local data only by sending messages to other functions necessarily located in other cells. The combination of message passing and the limited amount of state a process may use makes CL1 similar to such object oriented programming languages such as ACT1 [Lieberman 81] or Smalltalk [GoldbergRobson 83].

Connection Machine-LISP is organized around a new data structure, the xector (pronounced zector). A xector is a collection of elements each of which is a pair of LISP objects. The first object of an element is called the index, the second the value. The set of indices of a xector is called its domain, the set of values its range. The xector model stipulates that at most one element of a xector is stored in a single Connection Machine PE. This allows all items in a xector to be operated on in parallel. Functions are provided to create xectors and to examine and set individual values of a xector. Parallel manipulation of

\(^8\)Thinking Machines is also considering a C based parallel language for the machine. But no details are available.
xectors is accomplished using two operators alpha, and beta.

The alpha operator takes as arguments two xectors and a binary function. The alpha operator returns a new xector. The two argument xectors are combined by finding pairs of elements with matching indices. The resulting xector contains elements whose index is the same as in the matched pair and whose value is the result of the argument function applied to the values of the matched pair.

A xector whose domain spans the integers corresponds to a one dimensional vector indexed numerically, typical of conventional programming languages. Alpha and plus applied to two such xectors executes the parallel adding of two vectors. (See Figure 11a.) Since the values of both the domain and range of a xector are defined to be LISP objects, much more complicated operations are permitted. (See Figure 11b.)

\[
\alpha+ \text{ domain } '(1->12 \ 2->6 \ 3->0) \ '(1->8 \ 2->16 \ 3->32) \Rightarrow (1->20 \ 2->22 \ 3->32)
\]

(a)

\[
\alpha\text{cons } '(a->\text{red} \ b->\text{blue}) \ '(a->\text{block} \ b->\text{cube}) \Rightarrow (a->(\text{red } . \text{ block}) \ (b->(\text{blue } . \text{ cube}))
\]

(b)

Figure 11: Alpha Operator Applied to Xectors

The second operator, beta, is a reduction operator similar to the reduction operator in APL [Iverson 62]. Beta takes a single xector as an argument and an associative function. The function is applied, in parallel, to the range values of the xector. In logarithmic time, a single value is returned. (See Figure 12a.) Beta may also take two xectors as arguments and return a new xector whose range values are taken from the first xector and whose domain values are taken from the second.

5.4 Effectiveness of the Connection Machine

At the time of this writing, the Connection Machine has only been operational a few months. The primary software efforts have been aimed at the development of low level vision algorithms and compilers for CL1 and CM-LISP. Very little AI software has been implemented to date.
\[(\beta + \{1->12 \ 2->6 \ 3->0\}) \Rightarrow 22\]

(a)
\[(\beta \{1->8 \ 2->16 \ 3->32\} \{1->A \ 2->B \ 3->C\}) \Rightarrow \{A->8 \ B->16 \ C->32\}\]

(b)

**Figure 12:** The Beta Operator

The NETL system has been implemented on the Connection Machine. NETL does not require the generality of the packet switching network and therefore it was not used in the implementation. Marker propagations are performed in 5usec., considerably faster than Fahlman’s machine, though the Connection Machine is considerably more complex.

Due to the fine grain SIMD architecture, the effective utilization of the Connection Machine PEs is open to question. The packet switching network requires roughly 100usec. to perform a delivery cycle, the time it takes to move a set of packets from their source to their destination. Source and destination do not include the intervening nodes within a fanout tree. Each full communication cycle requires several delivery cycles, thus many hundreds of microseconds, during which the processors are idle. Though communication between PEs may be logically simple, the time required is three orders of magnitude greater than a memory access. The fine granularity, 4K bits per PE, implies that data structures will be spread over many PEs. Several communication steps may be required to process a single piece of data. This suggests a great a imbalance between memory size and communication bandwidth.

6 **DADO**

DADO is a medium grain, tree-structured machine, designed to provide significant performance improvements in the execution expert systems written in rule based form [Hayes-Rothetal 83]. Though originally intended for expert programs written as production systems [Stolfo and Miranker 84], recent work has shown its utility for expert programs written in logic based programming languages [Taylor 84].

---

9From conversations with Danny Hillis

10"DADO" is a name, not an acronym.
Two prototypes have been built, the DADO1 composed of 15 8 bit wide processing elements each with 12K bytes of memory, and the DADO2 composed of 1,023 8 bit wide processing elements each with 20K bytes of memory.

6.1 The DADO Architecture

DADO is similar to the two previous machines, in that it is attached to a conventional host computer and serves as a backend symbolic processor. All these machines embrace the notion of intelligent memory; their architectures consisting of many processors extensively distributed around local memories. But the granularity of the DADO machine is larger. Where the proposals above suggest machines with a million PEs each with very modest memories, a full-scale production version of the DADO machine would comprise on the order of tens to a hundred thousand PEs, each containing its own fully capable processor, and a larger amount of memory.

The PEs of DADO are connected in a complete binary tree and contain a specialized I/O section that can execute three global I/O operations, broadcast, report, and max-resolve. Any node in the DADO tree can broadcast data to all of its descendants. Conversely, any node can report data to an ancestor.

The third operation supported by the I/O section is a high speed, hardware maximizing network. During a resolve operation, all PEs offer a candidate value. At each node of the tree, a three way comparison is performed between the value at the node and two values computed by the children. The maximum of the three is then passed up the tree until, in log time, the top of the tree contains the maximum of all candidate values. Simultaneously, also in log time, the PE with the winning value is told it has won. Ties are arbitrated in the hardware.

This ensemble of global operations is very powerful for implementing many algorithms. Consider an algorithm that evaluates in parallel all the points at the frontier of a search space in an AI problem solver. Each PE evaluates a different point in the frontier. The resolve operation computes the best point, the one for which the evaluating function returns the largest value, in log time. The PE with that point can report details of its state to the root of the tree, which in turn can broadcast the information to all its descendants. The PEs can then expand the search from there and, using local criteria, determine the next point they should evaluate. Many points of the search space are evaluated in parallel and the next stage of the search is set up in three log time operations.

Within the DADO machine, each PE can execute in either of two modes, as master or slave. The mode
may change dynamically during run time under software control. When a PE is acting as a slave, or is operating in SIMD mode, it executes instructions broadcast down the tree by an ancestor PE. In master, or MIMD, mode, a PE has its logical state changed in such a way as to effectively "disconnect" it and its descendants from all higher-level PEs in the tree. A MIMD PE executes instructions stored in its own local RAM, independently of the other PEs. In particular, a PE in MIMD mode does not receive any instructions from any of its ancestors. A MIMD PE also becomes the master of any connected SIMD descendants and may broadcast instructions embedded within its own program for SIMD execution.

The DADO machine can be configured such that an arbitrary set of nodes may each act as the root of an independent tree-structured SIMD device. Thus, the machine can be logically divided into distinct partitions, each executing a distinct task. This flexible architectural design supports multiple-SIMD execution (MSIMD) [Siegel 81] and is the primary source of DADO's speed in executing a large number of primitive pattern matching operations concurrently.

6.2 The DADO Prototypes

Two DADO prototypes have been built at Columbia University. For rapid prototyping purposes a very conservative processor technology, the Intel 8751, was chosen as the core of the processing element. The 8751 is an 8 bit, single chip computer. Its functionality includes, 4K bytes of EPROM, 128 bytes of RAM and most importantly 4 parallel I/O ports that make it very convenient to build communication paths.

A 15-element DADO1 prototype has been operational since spring 1983. Communication is achieved by directly wiring the ports of the 8751s to each other. The global communication primitives are simulated using the tree edges with EPROM resident software. Each PE also has 8K bytes of RAM. (See Figure 13.) The entire DADO1 prototype was built on two wire-wrap boards. It is housed in a chassis roughly the size of a small stereo receiver. The processors are clocked at 3.5 megahertz producing 4 million instructions per second (MIPS) [Miranker 84]. The effective usable MIPS is considerably less due to the significant overhead incurred when simulating global interprocessor communication. The DADO1 served as a software development environment and as a test bed providing performance data used to design the larger DADO2.

The DADO2 is identical to DADO1 except for the doubling of the RAM size, now 16K bytes, and the addition of a semicustom gate array used to perform the global communication instructions and memory support. With memory parity support, a DADO2 PE is composed of 5 chips. (See Figure 14.)
A 1,023 node DADO2 became operational in fall 1985. The 1,023 nodes of DADO2 have an aggregate capacity of 570 MIPS and 16 Mbytes of RAM. DADO2 is built from 32 large, multiwire boards, 32 PEs on each. The circuit boards are mounted in a 24 inch rack. The volume of the circuitry is less than a two foot cube. The entire machine dissipates less than 1,200 watts.

Figure 13: The DADO1 Prototype Processing Element.

Figure 14: The DADO2 Prototype Processing Element.
6.3 Programming DADO

Software development for the DADO machine is divided into 4 layers. (See Figure 15.) The guiding philosophy of the DADO project is that end users of the machine should program in such traditional rule based expert system languages as OPS5 [Forgy 81] or Prolog, with little or no knowledge that a parallel machine is executing the program. The user’s expert applications programs form the top, or fourth, software layer. The interpreters for the expert system languages form the third software layer.

![Software Layering for the DADO Architecture](image)

Figure 15: Software Layering for the DADO Architecture

Though end users are supposed to be isolated from the confusion introduced by parallel programming, it is desirable for system programmers, the implementors of the expert system language interpreters, to be aware of the parallelism being sought in a program and even the underlying architecture for which the program is targeted. Very specific algorithms are used to extract parallelism from the application programs. Therefore, the low level system programming languages contain no automatic extraction of parallelism. If they were to do so, they might have a different model than the one the system programmer had in mind and would only serve to add confusion and inconsistency. These languages, with very explicit control over the parallelism, form the second software layer.

The lowest software layer is called the *kernel*. It is a program resident in the EPROM of each PE. The kernel is tightly coupled to the hardware of the machine and causes the machine to execute the primitives that define a DADO machine. However, the primitive operations of DADO include communications primitives as well as operations controlling the partitioning and synchronization of the machine. The
kernel simultaneously takes on aspects of being an operating system program as well as the microcode of
the DADO machine.

6.4 Low Level Systems Languages

6.4.1 PPL/M

A superset of Intel's PL/M language [Intel 82], which is called PPL/M, has been implemented as the first
system-level language for the DADO prototypes. PPL/M provides a set of facilities to specify operations
performed by independent PEs in parallel. PL/M is a conventional block-oriented language providing a
full range of data structures and high-level control statements. The following two syntactic conventions
have been added to PL/M for programming the SIMD mode of operation of DADO. The design of these
constructs was influenced by the methods employed in specifying parallel computation in the GLYPNIR
language [Lowrie 75] designed for the ILLIAC IV parallel processor. The SLICE attribute defines
variables and procedures that are resident within each PE. The second addition is a syntactic construct,
the DO SIMD block, which delimits PPL/M instructions broadcast to descendant SIMD PEs. (In the
following definitions, optional syntactic constructs are represented within square brackets.)

The SLICE attribute:

```
DECLARE variable[(dimension)] type SLICE;

name: PROCEDURE[(params)] [type] SLICE;
```

Each declaration of a SLICEd variable will cause an allocation of space for the variable to occur within
each PE. SLICEd procedures are automatically loaded within the RAM of each PE by an operating
system executive resident in DADO's coprocessor.

Within a PPL/M program, an assignment of a value to a SLICEd variable will cause the transfer to occur
within each enabled SIMD PE concurrently. A constant appearing in the right hand side will be
automatically broadcast to all enabled PEs. Thus, the statement

```
X=5;
```

where X is of type BYTE SLICE, will assign the value 5 to each occurrence of X in each enabled SIMD
PE. (Thus, at times it is convenient to think of SLICEd variables as vectors which may be operated upon,
in whole or in part, in parallel.) However, statements which operate upon SLICEd variables can only be
specified within the bounds of a DO SIMD block.
DO SIMD block:

DO SIMD;
  r-statement_0;
  ...
  r-statement_n;
END;

The r-statement is restricted to be any PL/M statement incorporating only SLICEEd variables and constants.

In addition to the full range of instructions available in PPL/M, a DADO PE in MIMD mode will have available to it a set of built-in functions to perform the basic tree communication operations, in addition to functions controlling the various modes of execution.

6.4.2 PPSL and Parallel C

PPL/M was the first systems language to be developed for DADO. It has served as the basis for the subsequent development of PPSL, (parallel LISP [van Biema et al. 84]) and parallel C.

The substantive change between PPL/M and its successors is that PPL/M is a byte oriented language and its successors are structure oriented, freeing the system programmers from a considerable amount of housekeeping. Rather than assuming all data structures are flat and of fixed length, the LISP and C versions are able to move arbitrary structures around the tree. For example, the statement

(Broadcast dest-id source-s-expr)

will move an arbitrary LISP s-expression from a MIMD PE to all it descendant SIMD PEs, and bind it to "dest-id," regardless of its size or depth of nesting.

6.5 Production System Execution

There are a large number of algorithms for the execution of production systems on the DADO machine [Stolfo 84, Stolfo 85]. For pedagogical reasons, only the simplest one is presented here. First assume there are an infinite number of PEs. This assumption will be removed later.

1. A single rule is assigned to each PE, executing in MIMD mode, at a (logically) fixed level within the tree. Each rule is matched concurrently. (This fixed level within the tree is referred to as the PM-level. See Figure 16.) Thus, the time to calculate the conflict set of rules on each cycle is independent of the number of productions in the system.

2. Assign a data item from WM to a single PE executing in SIMD mode, lying below each PM-level PE. Since at most a single WM-element is in a single PE the SIMD subtree behaves as as a true hardware content addressable memory. Thus, the time required to
match a single pattern element in the LHS of a rule is independent of the number of facts in WM. Note that partially matching the new WM with the rule in the PM-level PE can determine if the WM is relevant to the rule. Only the relevant WM need be stored in the subtree; i.e. if the rule pertains to red blocks, WM representing blue blocks need not be copied into the subtree.

3. Lastly, the selection of a single rule for execution from the conflict set is also performed in parallel using associative operations that merge the data as it rises in the tree. Thus, the logarithmic time lower bound of comparing and selecting a single item from a collection of items is achievable on DADO as well.

Figure 16: Functional Division of the DADO Tree

To remove the infinite PE assumption, consider that the nodes of a DADO machine are not simple primitive PEs, as in the machines described above. Each is a fully capable computer, able to become a MIMD PE. Rather than putting a single WM-element in a single PE, there are a variety of indexing and hashing schemes [Knuth 69] that permit many WM-elements to be placed in a single PE without significant loss of performance. The SIMD PEs may be quickly and temporarily switched into MIMD PEs while they execute the local indexing and matching code. The subtrees of DADO are able to synergistically combine algorithmic properties of both sequential and parallel computation. Thus, we may eliminate the requirement for infinitely large subtrees. Similarly there are algorithms that permit multiple rules to be placed in a single PE without significant loss of performance. A comparative
evaluation of these methods with various allocation schemes has been reported elsewhere (see [Stolfo 83, Miranker 84, Ofazer 84]).

6.6 Logic Programming

A Logic Programming Language (LPS) [Taylor 84, Taylor et al. 84, Lowry 84] is under development at Columbia. The key aspects of LPS are unification and reconciliation. Unification finds a substitution to transform two terms into identical terms. The unification processes can be performed in many processors, each one working on a particular goal. There is a most general unifier such that all other unifiers are instances of it.

Reconciliation is a process that determines if two substitutions are compatible, and produces the most general substitution if one exists. It is an essential part of the LPS algorithms and makes the various unifiers consistent. In this manner, the independent unifications give rise to one answer.

We may view the proof search as a perusal of goal lists. The LPS search tree, in comparison to the standard sequential algorithm used by Prolog, is characterized by:

- Shorter paths to leaves.
- Earlier termination of unproductive paths.
- Earlier consideration of most goals, causing earlier branching but not necessarily higher branching factors.
- A substantially reorganized leaf structure, resulting in a different order to the construction of solutions.

The unification phase of the LPS algorithm places different facts into each PE, and then transmits a goal list from the host processor into the PE network. Each PE unifies with many goals, producing unifiers that are tagged with a level number to identify the goal whose unification gave rise to the binding set.

The second phase is the reconciliation phase. It is also known as a "join phase" due to similarity with the operation of an equi-join over a set of relational database operations. A heuristic for ordering the join phase can, in most circumstances, keep join phase communication close to minimal. The join phase is coordinated by the host processor, to allow communication of binding sets around the network.

The last task to be performed upon discovery of a successful proof is the composition of the various substitutions that were generated along the way. In the LPS implementation, the substitution phase is accomplished by transmitting prior reconciliation history to the PE network, and computing in each PE the composition of that substitution with any new reconciliation.
6.7 Effectiveness of the DADO Architecture
The larger granularity of the DADO machine permits several copies of complete data structures to reside in individual PEs. The MIMD/SIMD behavior of the machine provides easily used communication and synchronization mechanisms yet permits PEs to independently execute code segments. The combination of these properties along with the use of smart algorithms for the partitioning and distribution of data and programs simultaneously keeps the utilization of processing elements high while reducing communication requirements.

The bottleneck introduced by the tree topology has not been a problem. If multiple PEs need the same information, rather than communicating the data, the algorithms arrange for copies to be placed in multiple PEs. Thus, a principle of the DADO architecture is to expend an inexpensive resource, memory; rather than develop complicated communication mechanisms.

Basic differences between the DADO machine and the semantic net machines above are due to the differences between the rule based paradigms and semantic nets. In semantic nets, when a new fact is entered into the database, a link must be established between it and every other object for which a relation holds. Therefore, the semantic net machines must incorporate expensive switching mechanisms to represent the high connectivity of information. In rule based paradigms, both production system based and logic based, the fact is simply entered and the relationships are not discovered until a query is processed.

No experimental performance data for DADO machine is available at the time of this writing. Several studies have been conducted to estimate its performance [Gupta 84, Miranker 84]. Although the performance of the DADO2 prototype is hindered by the use of an 8 bit processor, it is expected that the DADO2 will execute the OPS5 production system language many times faster than a DEC VAX 11/750.

7 Neural Models
The distinction between the study of machine intelligence, AI, and the study of human intelligence, cognitive psychology, is sometimes blurred. Researchers whose interests straddle the border between the two disciplines have proposed models of computation based on networks of primitive neuron-like processing elements. The purpose of these models is to provide insight into the workings of the human mind, as well as to develop models where computation is performed largely by high bandwidth communication. Although the motivation for developing these models is not to support directly any of the four familiar schemes of AI knowledge representation, it is appropriate for this paper to describe the
developments in this area of research. A variety of such models exist. Hinton [Hinton 84] provides a hierarchical taxonomy of these models.

### 7.1 The Perceptron

The Perceptron model [Rosenblatt 61, Minsky 69] is probably the most well known neural model. It is briefly described here since many consider it to be the grandfather of AI machines. A single perceptron is a primitive processing element with multiple inputs, a threshold value and a single binary output. Each input line of a perceptron has associated with it a real-valued weight and is connected to the output line of a sensor or another perceptron. The perceptron computes the dot product of the vectors formed by the weights of the input lines and the binary activity values placed on the input lines by other perceptrons or sensors. If the product is greater than the threshold, the output of the perceptron becomes active and outputs a 1. Otherwise, the perceptron becomes inactive and outputs a 0.

The utility of perceptrons has been studied extensively. A learning algorithm was invented to adaptively adjust the threshold value of a perceptron as well as the weights on its input lines so that the perceptron would become active only if a particular feature appeared in a pattern. The perceptron has a number of shortcomings. The learning algorithm is applicable only for a single layer of perceptrons, and systems of perceptrons with feedback are not amenable to formal analysis [Minsky 69].

### 7.2 The Boltzmann Machine

Neural models have been proposed frequently for pattern recognition tasks. The potential of VLSI design has sparked a renewed interest in these models [Hinton 81] and has uncovered methods to support higher level symbolic computation. The most developed of these models is the the Boltzmann Machine [Hintonetal 84]. No hardware implementations for the Boltzmann model have been proposed.

#### 7.2.1 The Boltzmann Model

The Boltzmann model assumes a large network of binary nodes are connected to each other with symmetrically weighted, bidirectional links. A node may be on or off as a probabilistic function of the states of the connected nodes and the weights on the connecting links. Weights are real valued. A negative weight on a link creates inhibitory behavior between two nodes- if one of the two nodes is on, the other is more likely to be off. A positive weight creates supportive behavior- if one node is on, the other is more likely to be on.

For each possible combination of states in the network a measure may be made. Due to the similarity
between Boltzmann networks and systems of particles studied in statistical mechanics, this measure is called the *energy* of the *global* state and is defined as:

\[ E = - \sum_{i<j} w_{ij} s_i s_j + \sum_i s_i \Theta_i \]

where \( w_{ij} \) is the weight of the link between nodes \( i \) and \( j \), \( s_i \) is 1 if node \( i \) is on and 0 otherwise. \( \Theta_i \) is a threshold value associated with each node.

The threshold values may be eliminated by considering a new network, such that every node has a new node attached to it that is always on and the link has a weight equal to the threshold, but is otherwise identical to the old network.

In this case the energy equation simplifies to:

\[ E = - \sum_{i<j} w_{ij} s_i s_j \]

The input to the Boltzmann network is a distinguished set of nodes, whose state may be clamped on or off according to external stimuli. The output is the state induced by the network on a second set of nodes. If the changes in state of the individual nodes occur asynchronously and the communication time is negligible, then a simple algorithm [Hintonetal 84] will cause the network to settle into a local, minimum energy level for each possible input. But, to be useful, the same input must cause the network to consistently settle into the same state. The global energy minimum is such a state. In order to prevent the system from being caught in a local minima, Hinton and his colleagues incorporate the recent work in simulated annealing algorithms [Kirkpatrick 83] into the settling algorithm for the network. Simulated annealing is a stochastic process that adds noise to a system, permitting it to jump out of local minima. To draw the physical analogy, the algorithm is much like the process of heating and cooling a piece of metal to reduce its internal stress. The system starts at a high temperature where individual nodes may make large unlikely transitions, enabling them to leave local minima. The temperature is slowly reduced, making large state changes less likely, until the system reaches equilibrium at its global minimum energy.

Figure 17 shows a Boltzmann network capable of computing an XOR operation. Due to the symmetry of the links in the Boltzmann model, the three nodes across the bottom may serve as either inputs or outputs. The table to the right of the diagram indicates the energy levels for all possible states. Note that the zero values correspond to the correct computation of XOR. In operation, the network would settle into the low energy state outputting the correct value regardless of which two nodes were selected as the input cells.

One of the most significant contributions of the Boltzmann work is the development of a domain-
Figure 17: A Boltzmann Network for Computing XOR

independent learning algorithm. Given an arbitrary input, the algorithm will adjust the weights of the links such that the network will settle into a known state.

7.2.2 Representation and Search in the Boltzmann Model

As discussed in the introduction, a problem solving architecture must support both representation and search. In the Boltzmann model, representation is distributed. Rather than having the representation of a single piece of knowledge wholly contained by and local to a single processing entity, the Boltzmann model represents a fact as a set of states in a collection of Boltzmann elements. Since the state of an individual Boltzmann element is determined by a probability function, the fact may not appear as a particular element’s state but as a pattern of states in the network. The same collection of elements may simultaneously contribute to the representation of many pieces of knowledge. Only upon clamping the set of input cells will the network settle into a state representing a stored piece of knowledge.

The use of strong unbreachable constraints to direct searches, is an effective problem solving technique [Winston 72, Waltz 75]. The Boltzmann organization is adept at solving problems that involve large numbers of weak constraints. Given an input, the Boltzmann model will settle into a configuration most compatible with the known constraints and thus discover best fit solutions. The Boltzmann model is able to solve problems where some constraints must be violated.
Two important capabilities arise naturally in the Boltzmann model. The model has content-addressable-memory-like abilities. If a partial description of an object is input to the network, the machine will fill in the rest of the description.

The second capability is that the model has an implicit model of generalization. Rather than storing different facts in an orthogonol fashion [Hinton 81], the learning algorithm for the Boltzmann model discovers similarities in facts and represents their commonality by sharing an activation pattern in a common set of nodes. If one fact has properties associated with it, a second fact will inherit these properties in proportion to the strength of the similarity of the two facts. Using the example in the NETL section to illustrate this, if we have a collection of animals with fins that live in water, then the Boltzmann model will create a strong correlation between animals with fins and living in water. When a new animal with fins is added to the knowledge base, it is natural to assume it lives in the water. The Boltzmann machine model will automatically make this association. The more animals in the knowledge base known to have fins and to live in water, the stronger "living in water" will be associated with the new "finned" animal. The aspects of search and knowledge retrieval are completely unified in the Boltzmann model.

7.2.3 A Symbolic Inference Engine, Based on the Boltzmann Model

Neural models have been almost exclusively explored for low level perception tasks. It has been argued that to reason, systems of higher intelligence must be able to manipulate objects symbolically [Newell 72]. To show that neural models can support intelligent behavior, it has been incumbent on proponents of these models to show that the models are capable of symbolic manipulation.

This challenge recently was answered by a paper by Touretzky and Hinton [Touretzky 85]. A modest production system interpreter has been implemented using the Boltzmann Model. The rules are limited to the conjunction of two patterns with at most a single variable. This is analogous to the join operation that NETL is able to perform. The working memory is limited to triples of a 25 letter vocabulary. Nevertheless, the system demonstrates the plausibility of using a neural model to perform symbolic computations.

The structure that implements the production system interpreter requires the localization of functional portions of the interpreter to different parts of the network. (See Figure 17.) This is consistent with higher level abstractions of how intelligent systems may be organized [Minsky 79]. Though functions are localized, the representation of them is distributed.
The interpreter requires seven to eight thousand cells to represent and execute production systems of five to 10 rules and a similar amount of working memory. The space of possible working memory elements, $25^3$ or 15,625 possible triples, is represented by a network of 2,000 cells. An average of 28 cells must be active to indicate the storage of a particular triple. The network may accurately store about 12 triples simultaneously. The clause spaces each also contain 2,000 cells, and are used to map working memory to currently satisfied conjuncts. The 2,000 remaining cells are split evenly between rule representation and variable binding. It is worth noting that the fanout of each of these cells is on the order of a thousand connections.$^{11}$

In order to simulate the serialization of the parts of the production system cycle, the implementation of the production system interpreter requires the gating of signals to and from the working memory. This feature requires altering some basic assumptions for the Boltzmann model. The new model can still be shown to be consistent. However, the sequencing of these operations is a byproduct of the development of production systems on sequential computers. It is likely that future production system paradigms will not require the parts of the production system cycle to be sequenced [Tenorio 85, vanBiema 86, Ishida and Stolfo 84].

$^{11}$From private correspondence with Dave Touretzky.
7.2.4 Prospects for Hardware Implementation
Currently there are no proposals for a hardware implementation of the Boltzmann Machine. Several issues must be addressed before a Boltzmann Machine can be built. These may be especially problematic if the physical realization is very fine grained; that is, if a single processing element models the behavior of a single neuron. This section explores these issues.

The convergence criteria of the settling algorithm for the network creates two control issues. One, the cells must switch state asynchronously. Asynchronous behavior is more easily achieved using MIMD processing. Yet, fine granularity typically implies SIMD processing since it allows factoring out both control logic and control store from the PE circuitry. If SIMD processing is used, some quantization of the asynchronous behavior will be introduced as well as a reduction in utilization of the PEs and communication paths. As the quantization of the asynchronous behavior becomes finer, a smaller proportion of the PEs will be busy communicating their change of state. The trade off of utilization and correct functioning of the settling algorithm must be weighed.

The second requirement is the variation of the temperature variable for the annealing algorithm. Though the difficulty of synchronously distributing global information throughout a machine is overblown, it would be convenient if it were shown that the annealing algorithm works correctly even if the network does not “cool” uniformly.

To build a general purpose Boltzmann machine, either a regular pattern of interconnect must be developed to which all Boltzmann networks may be mapped or a communication switch must be included as part of the design. Since the fanouts of elements in the Boltzmann model are so much larger than those supported by NETL and the Connection Machine, -thousands instead of four and fourteen respectively,- we can only assume that a switching network for the Boltzmann model would be much more difficult. Given the poor performance of communication switches, as demonstrated by both NETL and Connection Machines, it appears paramount that a universal interconnection pattern be developed for the Boltzmann model.

8 Summary and Conclusion
This section summarizes the similarities and differences of the surveyed machines. Some generalizations obviously will not apply to the Boltzmann Machine, since it has no suggested implementation. The table in figure 19 summarizes the architectural characteristics of the surveyed machines.
<table>
<thead>
<tr>
<th>Machine</th>
<th>MIMD/SIMD topology</th>
<th>switching</th>
<th>AI paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Netl</td>
<td>SIMD</td>
<td>hashnet</td>
<td>semantic net</td>
</tr>
<tr>
<td>CM</td>
<td>SIMD</td>
<td>circuit</td>
<td>semantic net</td>
</tr>
<tr>
<td>DADO</td>
<td>MSIMD</td>
<td>packet</td>
<td>production system and logic</td>
</tr>
<tr>
<td>Boltzmann</td>
<td>Not applicable</td>
<td>message</td>
<td></td>
</tr>
</tbody>
</table>

Figure 19: Table Summarizing Machine Characteristics

The four machines surveyed have much in common. They are all specialized, seeking to address a particular problem solving paradigm. All designs include a conventional host computer as a front end, acting as a user interface and as the controller for the machine. Knowledge is distributed to a large number of processing elements, each having a local memory. Processing knowledge occurs both by processing within a node and by high bandwidth communication.

The three machines exemplify the alternatives in allocating resources to perform communication. The NETL machine contains a switch that supports point to point, circuit switched communication. The Connection Machine supports a packet switching network, creating the appearance of full connectivity between processors. The DADO supports the simplest form of communication, messages passed along fixed communication paths.

The semantic net machines represent efforts to create an isomorphism from the knowledge base to the processing elements. The large size of the knowledge bases require the PEs to be extremely simple, SIMD based with bit wide data paths and very modest local stores. The utilization of the PEs is then directly proportional to the amount of knowledge in the knowledge base that is active in the search. The absence of any studies measuring the activity in semantic net based searches prevents the proper assessment of the granularity for these machines. However, the fine granularity does force the construction of switching networks to establish logical links between individual facts represented by single PEs. The switching network is an expensive component of these machines. In order to design feasible switching networks either the functionality of the network must be severely reduced, as in the case of NETL, or the speed of the network made questionably slow, as in the case of the Connection Machine.
Relations between facts in Production Systems and Logic representations are not determined until the knowledge base is queried. This, in combination with algorithms that create multiple copies of data, permits the DADO machine to compute using a very simple topology. As an effective rule based problem solver, the DADO machine has a smaller number of more complex processing elements, each processor capable of an autonomous search, able to determine relationships on the fly.

The coarser granularity of the DADO machine provides two contrasting advantages. Using indexing techniques, disjoint information can be placed in a single PE, increasing processor utilization. Placing related information in a single PE can reduce communication requirements.

Though there are no proposals for the construction of the Boltzmann machine, we can expect that proposals suggesting a fine grain implementation, single neurons mapped to single PEs, to suffer the problems displayed by the semantic net machines. Indeed, due to the greater fanouts required of neurons than of semantic net knowledge bases, these problems may be greatly exacerbated.
References

The Handbook of Artificial Intelligence. 

The SCHEME-81 Architecture- System and Chip. 

[Bawden 84a] Bawden, A. 
A Programming Language for Massively Parallel Computers. 

[Bawden 84b] Bawden, A. and Agre, P.E. 
What a Parallel Programming Language has to Let You Say. 

[ClocksineMellish 81] Clocksin, W.F., Mellish, C.S. 
Programming in Prolog. 
Springer-Verlag, 1981.

[Dubois 82] Dubois, M. and Briggs, F.A. 
effects of cache coherency on multiprocessors. 

[Dyer 81] Dyer, C.R. 
A VLSI Pyramid Machine for Hierarchical Parallel Image Processing. 
IEEE, 1981.

[Fahlman 79] Fahlman, S. E. 
NETL: A system for representing and using real-world knowledge. 

[Fahlman 80] Fahlman, S.E. 
The Hashnet Interconnection Scheme. 

[Fahlman 82] Fahlman, S.E. 
Three Flavors of Parallelism. 
In Fourth National Conference of the Canadian Society for Computational Studies of Intelligence. 
Canadian Society for Computational Studies of Intelligence, Saskatoon, Saskatchewan, 1982.

Massively parallel architectures for AI: NETL, Thistle, and Boltzman machine. 

[Fahlman 83b] Fahlman, S.E. 
Design Sketch for a Million Element NETL. 
Cancellation in a parallel semantic network.  
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*.  
IJCAI, 1981.

[Flynn 72] Flynn M. J.  
Some Computer Organizations and Their Effectiveness.  
*The Institute of Electrical and Electronic Engineers Transactions on Computers* v-21,  
September, 1972.

[Forgy 81] Forgy C. L.  

[GoldbergRobson 83] Goldberg, A. and Robson, D.  
*Smalltalk-80, The Language and its Implementation*.  
Addison-Wesley, 1983.

[GoodmanShmueli 82] Goodman, N. and Shmueli, O.  
Tree Queries: A Simple Class of Relational Queries.  

[Gupta 84] Gupta, A.  
*Implementing OPS5 Production Systems on DADO*.  

*Building Expert Systems*.  
Addison Wesley, 1983.

[Hewitt 69] Hewitt, C.E.  
PLANER: A Language for Proving Theorems in Robots.  
In *International Joint Conference on Artificial Intelligence*.  
IJCAI, 1969.

[Hillis 82] Hillis, W.D.  
*The Connection Machine*.  

[--- 85] Hillis, W.D.  
*The Connection Machine*.  

*Parallel Models of Associative Memory*.  

[Hinton 84] Hinton, G.E.  
*Distributed Representations*.  
Boltzmann Machines: Constraint Satisfaction Networks that Learn.

[Intel 82] Intel Corporation.
PL/M-51 User's Guide for the 8051 Based Development System.
Order Number 121966.

[Ishida and Stolfo 84] Ishida T., and S. J. Stolfo.
Simultaneous Firing of Production Rules on Tree-structured Machines.

[Iverson 62] Iverson, K.E.
A Programming Language.

Optimization by Simulated Annealing.

[Kitsuregawa 84] Kitsuregawa, M., Tanaka, H. and Moto-oka, T.
Architecture and Performance of Relational Algebra Machine GRACE.

[Knuth 69] Knuth, D.E.
Sorting and Searching.
Addison Wesley, 1969.

[Kung 76] Kung, H. T.
Synchronized and Asynchronous parallel Algorithms for Multiprocessors.
New Directions and Recent Results in Algorithms and Complexity.

[Laird 84] Laird, J.E., Rosenbloom, P.S., and Newell, A.
Towards Chunking as a General Learning Mechanism.

[Lieberman 81] Lieberman, Henry.
Thinking About Lots of Things at Once Without Getting Confused: Parallelism in Act 1.
Technical Report Memo 626, Massachusetts Institute Technology Artificial Intelligence Laboratory, 1981.

A Programming Language for Illiac IV.

LPS Algorithms: A Detailed Examination and Critical Analysis.
[Lyon 81] Lyon, R.F.
The Optical Mouse, and an Architectural Methodology for Smart Digital Sensors.

[McDermott and Forgy 78]
McDermott, J. and C. Forgy.
Production System Conflict Resolution Strategies.
*Pattern-directed Inference Systems*.

*Perceptrons*.

[Minsky 75] Minsky, M.
A framework for representing knowledge.
*The Psychology of Computer Vision*.

[Minsky 79] Minsky, M.
*K-Lines: A Theory of Memory*.

[Miranker 84] Miranker D. P.
Performance Estimates for the DADO Machine: A Comparison of TREAT and RETE.

[Moon 85] Moon, D.S.
Architecture of the Symbolics 3600.

*A Relational Database Machine: First Step to Knowledge Base Machine*.

[Myers 81] Myers, E.
Machines that LISP.
*Datamation*, September, 1981.

*Human Problem Solving*.

[Newell 73] Newell, A.
Production Systems: Models of Control Structures.
*Visual Information Processing*.

[Nilsson 80] Nilsson N. J.
*Principles of Artificial Intelligence*.
The Personal Sequential Inference Machine (PSI): Its Design Philosophy and Machine
Architecture.

[Oflazer 84] Oflazer, K.
Partitioning in Parallel Processing of Production Systems.
In *Proceedings of the IEEE International Conference on Parallel Processing*, pages

RAP- Associative Processor for Database Management.

[Quillian 68] Quillian, M. Ross.
Semantic Memory.
*Semantic Information Processing*.

[Rosenblatt 61] Rosenblatt, F.
*Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*.

[Rychener 76] Rychener, M.
*Production Systems as a Programming Language for Artificial Intelligence*.
Technical Report, Carnegie-Mellon University, Department of Computer Science,
1976.
Ph.D. Thesis.

[Schank 77] Schank, R., and Abelson, R.P.
*Scripts, Plans, Goals, and Understanding*.

[Schwartz 83] Schwartz, J.
*A Taxonomic Table of Parallel Computers, Based on 55 Designs*.

[Seitz 84] Seitz, C.L.
Concurrent VLSI Architectures.

D. S. Smith.
PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern
Recognition.

[Stolfo 83] Stolfo S. J.
Knowledge Engineering: Theory and Practice.
In *Proceedings from the Institute of Electrical and Electronic Engineers Trends and
Applications*. Proceedings from the Institute of Electrical and Electronic Engineers

[Stolfo 84] Stolfo S. J.
Five Parallel Algorithms for Production System Execution on the DADO Machine.
In *Proceedings of the National Conference on Artificial Intelligence*. AAAI, Austin,
Texas, August, 1984.
[Stolfo 85] Stolfo, S.J., Miranker, D.P. and Mills, R.
A Simple Preprocessing Scheme to Extract and Load Balance Implicit Parallelism in
the Concurrent Match of Production Rules.

[Stolfo and Miranker 84]
Stolfo S. J., and D. P. Miranker.

CASSM: A Cellular System for Very Large Data Bases.

Why Conniving is Better than Planning.
Technical Report Memo 255A, Massachusetts Institute Technology Artificial
Intelligence Laboratory , 1972.

[Taylor 84] Taylor S.
LPS, A Logic Programming System: Motivations and Goals.
(In preparation).

Logic Programming Using Parallel Associative Operations.
International Logic Programming Symposium, Atlantic City, N. J., February,
1984.

Mapping Production Systems into Multiprocessors.

[Thurber 74] Thurber, K.J.
Interconnection networks- A survey and assessment.

[Touretzky 84] Touretzky, D.S.
The Mathematics of Inheritance Systemss.

Symbols Among the Neurons: Details of a Connectionist Inference Architecture.
In Proceedings of the Ninth International Joint Conference on Artificial Intelligence.

[Uchida 82] Uchida, S.
The Personal Sequential Inference Machine.
Technical Report TR/A-001, Institute for New Generation Computer Systems,
November, 1982.


Figure 18: The Structure of a Boltzmann Production System Interpreter

The interpreter requires seven to eight thousand cells to represent and execute production systems of five to 10 rules and a similar amount of working memory. The space of possible working memory elements, 25³ or 15,625 possible triples, is represented by a network of 2,000 cells. An average of 28 cells must be active to indicate the storage of a particular triple. The network may accurately store about 12 triples simultaneously. The clause spaces each also contain 2,000 cells, and are used to map working memory to currently satisfied conjuncts. The 2,000 remaining cells are split evenly between rule representation and variable binding. It is worth noting that the fanout of each of these cells is on the order of a thousand connections.¹¹

In order to simulate the serialization of the parts of the production system cycle, the implementation of the production system interpreter requires the gating of signals to and from the working memory. This feature requires altering some basic assumptions for the Boltzmann model. The new model can still be shown to be consistent. However, the sequencing of these operations is a byproduct of the development of production systems on sequential computers. It is likely that future production system paradigms will not require the parts of the production system cycle to be sequenced [Tenorio 85, vanBiema 86, Ishida and Stolfo 84].

¹¹From private correspondence with Dave Touretzky.
Figure 17: A Boltzmann Network for Computing XOR

independent learning algorithm. Given an arbitrary input, the algorithm will adjust the weights of the links such that the network will settle into a known state.

7.2.2 Representation and Search in the Boltzmann Model

As discussed in the introduction, a problem solving architecture must support both representation and search. In the Boltzmann model, representation is distributed. Rather than having the representation of a single piece of knowledge wholly contained by and local to a single processing entity, the Boltzmann model represents a fact as a set of states in a collection of Boltzmann elements. Since the state of an individual Boltzmann element is determined by a probability function, the fact may not appear as a particular element’s state but as a pattern of states in the network. The same collection of elements may simultaneously contribute to the representation of many pieces of knowledge. Only upon clamping the set of input cells will the network settle into a state representing a stored piece of knowledge.

The use of strong unbreachable constraints to direct searches, is an effective problem solving technique [Winston 72, Waltz 75]. The Boltzmann organization is adept at solving problems that involve large numbers of weak constraints. Given an input, the Boltzmann model will settle into a configuration most compatible with the known constraints and thus discover best fit solutions. The Boltzmann model is able to solve problems where some constraints must be violated.
match a single pattern element in the LHS of a rule is independent of the number of facts in WM. Note that partially matching the new WM with the rule in the PM-level PE can determine if the WM is relevant to the rule. Only the relevant WM need be stored in the subtree; i.e. if the rule pertains to red blocks, WM representing blue blocks need not be copied into the subtree.

3. Lastly, the selection of a single rule for execution from the conflict set is also performed in parallel using associative operations that merge the data as it rises in the tree. Thus, the logarithmic time lower bound of comparing and selecting a single item from a collection of items is achievable on DADO as well.

Figure 16: Functional Division of the DADO Tree

To remove the infinite PE assumption, consider that the nodes of a DADO machine are not simple primitive PEs, as in the machines described above. Each is a fully capable computer, able to become a MIMD PE. Rather than putting a single WM-element in a single PE, there are a variety of indexing and hashing schemes [Knuth 69] that permit many WM-elements to be placed in a single PE without significant loss of performance. The SIMD PEs may be quickly and temporarily switched into MIMD PEs while they execute the local indexing and matching code. The subtrees of DADO are able to synergistically combine algorithmic properties of both sequential and parallel computation. Thus, we may eliminate the requirement for infinitely large subtrees. Similarly there are algorithms that permit multiple rules to be placed in a single PE without significant loss of performance. A comparative
6.3 Programming DADO

Software development for the DADO machine is divided into 4 layers. (See Figure 15.) The guiding philosophy of the DADO project is that end users of the machine should program in such traditional rule based expert system languages as OPS5 [Forgy 81] or Prolog, with little or no knowledge that a parallel machine is executing the program. The user's expert applications programs form the top, or fourth, software layer. The interpreters for the expert system languages form the third software layer.

![Diagram of software layering for the DADO Architecture]

**Figure 15:** Software Layering for the DADO Architecture

Though end users are supposed to be isolated from the confusion introduced by parallel programming, it is desirable for system programmers, the implementors of the expert system language interpreters, to be aware of the parallelism being sought in a program and even the underlying architecture for which the program is targeted. Very specific algorithms are used to extract parallelism from the application programs. Therefore, the low level system programming languages contain no automatic extraction of parallelism. If they were to do so, they might have a different model than the one the system programmer had in mind and would only serve to add confusion and inconsistency. These languages, with very explicit control over the parallelism, form the second software layer.

The lowest software layer is called the *kernel*. It is a program resident in the EPROM of each PE. The kernel is tightly coupled to the hardware of the machine and causes the machine to execute the primitives that define a DADO machine. However, the primitive operations of DADO include communications primitives as well as operations controlling the partitioning and synchronization of the machine. The
A 1,023 node DADO2 became operational in fall 1985. The 1,023 nodes of DADO2 have an aggregate capacity of 570 MIPS and 16 Mbytes of RAM. DADO2 is built from 32 large, multiwire boards, 32 PEs on each. The circuit boards are mounted in a 24 inch rack. The volume of the circuitry is less than a two foot cube. The entire machine dissipates less than 1,200 watts.

![Diagram of DADO1 Prototype Processing Element](image1)

**Figure 13:** The DADO1 Prototype Processing Element.

![Diagram of DADO2 Prototype Processing Element](image2)

**Figure 14:** The DADO2 Prototype Processing Element.
Figure 10: Organization of a Connection Machine Processing Element

Instructions. An innovation in the design of the Connection Machine is instead of a single enabling bit, typical of most SIMD machines, the Connection Machine has a condition field in its instruction format that may select any flag from the flag set as the enabling bit, as well as the polarity of the bit. A PE executes an instruction only if the the contents of the specified flag match the polarity of the condition. This feature greatly facilitates mimicking a MIMD machine. For example, given the program:

;Program to set c = max(a,b)

a,b,c int:

where a>b
    c:= a;
else c:=b;

In a parallel MIMD machine, each PE would execute a conditional jump and an assignment. A SIMD machine such as the ILLIAC IV would compare a and b, disable a subset of the PEs according to the result of the comparison and then do the first assignment. The ILLIAC would then have to reenable all the PEs, disable the complementary subset and do the second assignment. The effect of the Connection Machine’s condition field is to reduce the number of primitive steps required to execute branches. To execute the above program the Connection Machine performs the comparison. The result is stored in a conditional flag. The first assignment instruction is broadcast with its condition field set so it is executed
Instruction fetch mechanism and program store. The resulting PEs are much simpler and more compact than MIMD PEs would have been. Using simulation arguments, it can be shown that a SIMD based machine can execute the same programs as a MIMD machine at a cost of a constant factor. It was assumed the branching factors of applications considered for the Connection Machine are sufficiently small that a SIMD implementation is more effective than a MIMD implementation. This assumption remains to be proven for AI applications.

The PEs of the Connection Machine are very simple. The data paths are 1 bit wide. The PEs contain a bit wide ALU and 8 bits of internal state in the form of general purpose bit flags. 4K bits of memory are associated with each PE. (See Figure 10.)

The primary component of the Connection Machine is a semicustom gate array chip containing the logic for 16 PEs and 1 router circuit. The router supports the packet switching network. Associated with the chip are 4, 4K x 4 static memory chips. 4K copies of this ensemble, 20K chips, form most of the circuitry of the Connection Machine. The physical volume of the machine is about a 5 foot cube. The machine dissipates 12,000 watts.

The 4K routers are connected by a 12 dimensional n-cube. In addition to the n-cube connections the PEs are connected in a two dimensional mesh called the NEWS network (for North, East, West and South). The NEWS connections do not involve the router and are used for diagnostics and the development of low level vision algorithms.

Instructions are broadcast to all PEs simultaneously and executed in lock step, synchronized to an external 4 MHz clock. Most SIMD machines have the ability to selectively disable processors from executing...
Figure 8: The Organization of a NETL Simulation Engine

The switching lookup table and marker state bits require roughly 1,800 64K RAM chips; each of 1 million rows contain 4, 20 bit fields to represent links and 33 bits of state. The trivial worst case for the lists of marked nodes is that all nodes are marked with all markers.\(^5\) This requires 5,100 64K RAM chips. Only a handful of MSI logic parts are required to complete the design. With realistic assumptions about the number of nodes marked in the worst case, the simulation engine will be much simpler than Fahlman’s proposed NETL machine.\(^6\)

With appropriate pipelining, the simulation engine would be limited by memory access time. The simulation engine could propagate a single marker in 100nsec; Fahlman’s suggested implementation requires 5msec. per marker propagation cycle. This implies that if less than an average of 50,000

---

\(^5\)Clearly the worst case in practice is a fraction of that.

\(^6\)It is worth noting that the simulation engine consists almost entirely of memory which, since the time of Fahlman’s writing, has scaled 16 fold, while packaging has only allowed at most a four fold expansion of I/O ability. Using 1985 technology, the simulation engine would be even smaller compared to Fahlman’s proposal, and as VLSI continues to scale the discrepancy in size would increase.
Father(a,b) Hates(a,b)
Father(b,c) Hates(d,c)
Join(Father(X,Y), Hates(X,Y))

Figure 6: Demonstrating a NETL weakness

[GoodmanShmueli 82], and joins in multiple attributes, that cannot be processed in parallel. Consider the following database and representation as well as the query to determine the set of mothers-in-law. (See Figure 7.) We may mark all those people who are married, then look for their mothers.

Mother(a,b)
Mother(c,d)
Married(b,c)

Mother-in-law(X,Y):= Mother(X,Z), Married(Z,Y)

Figure 7: Finding Mothers-in-Law Using NETL

The source of the restriction is the limited amount of information passed between nodes. A marker is a single bit, and even that information may be occluded when several markers collide at a node at once and they are simply ORed together. One solution to the latter problem is to define markers as numeric
state of an incoming link or by the result of a Boolean operation on two other bits within the node.

The links are formed by link nodes. Each link node has state bits as in object nodes. Associated with each link node are four wires. The node type is expressed by connecting a parent wire to a node containing the link type information. The two objects in an assertion are connected by a pair of wires called $A$ and $B$. A fourth wire is used to represent the context of the assertion. (See Figure 5).

![Figure 5: NETL Hardware Organization](image)

All the nodes sit on a common bus wired to a conventional computer that serves as a control processor. The control processor can broadcast instructions to either an individually addressed node, or to a set of nodes based on their internal state. A NETL machine is therefore a fine grain, SIMD machine.

Instructions for the object nodes of a NETL machine instigate the following actions:
- address a single node or a set of nodes based on their internal state
- transfer a bit to or from a marker bit and a link
- or perform a simple Boolean manipulation of the internal state.

An instruction may for example, address all PEs with bit 0 on, set bit 1 to, bit 2 AND bit 3. Instructions also are provided to address a set of nodes and have them sequentially report their state to the controller. Link nodes can be individually addressed and their contact points set to create new links between object
3.2 Shared or Distributed Memory

Another characteristic that distinguishes parallel computers is whether the memory is distributed to the processors or resides in a common, shared pool.

Shared memory architectures contain a collection of processors, a collection of memory modules and an interconnection network between them. (See Figure 4.) Architectures that use shared memories may communicate by reading and writing data into predefined areas of the memory. Thus, communication between processors appears to be as simple as accessing memory. Further, since all processors may read and write anywhere in the memory, then any processor may communicate directly with any other. By fully connecting the processors, developers of parallel algorithms need not consider the topology of the communication channels, simplifying the development of algorithms for this class of machine.

![Diagram of a shared memory machine]

**Figure 4:** The Generalized Structure of a Shared Memory Machine

The organization of the interconnection network is the focus of a great deal of study [Wu 84]. The many design alternatives that must be considered for the interconnection networks may itself be the subject of a survey [Thurber 74]. Briefly these alternatives include the topology of the interconnect, the amount of intelligence and buffering at the switch points and whether the switching technique should be packet switched, circuit switched or message switched.

In all cases, the switch in this class of machine presents two major problems. The complexity of the
Using semantic nets to describe a goldfish named Charley we may have a vertex labeled Charley and one labeled goldfish. (See Figure 3.) An edge connecting the two would be labeled is-a, to indicate the proposition that Charley is a goldfish. We may extend the taxonomy to include vertices for fish and animal. These vertices are also connected with is-a links. Such a collection is known as an is-a hierarchy. Other properties may be attached to nodes by links of other types. The example indicates that fish live in water and have fins and that Charley is gold colored.

![Figure 3: Example of a Semantic Net](image)

Attributes are represented by unique vertices. Objects with a common attribute have individual links to a shared, unique vertex representing the attribute. A system may find information about Charley by traversing the links in the graph. By following the appropriate path through the graph, we may find all the goldfish in the system or all the gold colored objects.

An advantage of semantic nets is that not only inferencing based on property inheritance may be performed by traversing the edges between nodes, but reasoning by analogy may also be performed by comparing the structural similarity of parts of the network [Winston 72]. Suppose that beside Charley the fish, the network contains information about Eliza the dolphin. The network representing Eliza would be nearly identical to the network for Charley, for both live in the water and have fins. It would then be
instantiation. All instantiations are collected to form a conflict set of rules.

2. Select. Some subset of the conflict set is chosen according to some predefined criteria. In practice a single instantiation is selected from the conflict set on the basis of the recency of the matched data in the WM, as well as syntactic discrimination.

3. Act. The actions in the RHS of the selected rules are executed.

An example rule using the OPS5 production system language [Forgy 81] is shown in figure 2.

\[
(p \text{ categorize-job-sizes} \quad (\text{message} \quad \text{job} <x> \quad \text{size} <y> \quad \text{status new}) \quad \text{rule name}
\]
\[
(\text{class-definition} \quad \text{size} <y> \quad \text{class-name medium}) \quad \text{condition element,}
\]
\[
\quad <x>, <y> \quad \text{are pattern variables}
\]
\[
\quad \text{condition element}
\]
\[
\rightarrow \quad (\text{make job} \quad \text{job-name} <x> \quad \text{class medium})
\]

This rule says if
there is a WM-element in the system representing a message about a new job,
and the job's size matches the class definition for medium size jobs,
then
create a new WM-element tagging the job with the class name medium.

Figure 2: An Example Production Rule.

Production systems have many of the advantages of logic based systems in that knowledge may be added incrementally, their strong syntactic constraints facilitate automatic generalization [Laird 84] of new knowledge, and formal proof techniques may be used to verify completeness and consistency of the knowledge base. Production systems permit only a single level of substitution, i.e., a variable must be bound to a ground fact not to another expression. This restriction limits the potential combinatorial explosion possible in logic systems, but simultaneously forces the programmer to add more structure in his programs to carefully direct the search process.

2.3 Semantic Nets
Quillian [Quillian 68] is credited with the first description and use of semantic nets. A semantic net is a graph representation of knowledge. Although there are many variations of the semantic net formalism, typically the vertices of the graph represent objects, while edges connecting two vertices represent a relationship between the connected objects. The knowledge in a semantic system resides not so much in the vertices but in the pattern of interconnected links.