Three Highly Parallel Computer Architectures and Their Suitability for Three Representative Artificial Intelligence Problems

Ron Katriel
University of Pennsylvania

Three Highly Parallel Computer Architectures and Their Suitability for Three Representative Artificial Intelligence Problems

Abstract

Virtually all current Artificial Intelligence (AI) applications are designed to run on sequential (von Neumann) computer architectures. As a result, current systems do not scale up. As knowledge is added to these systems, a point is reached where their performance quickly degrades. The performance of a von Neumann machine is limited by the bandwidth between memory and processor (the *von Neumann bottleneck*). The bottleneck is avoided by distributing the processing power across the memory of the computer. In this scheme the memory becomes the processor (a "smart memory").

This paper highlights the relationship between three representative AI application domains, namely knowledge representation, rule-based expert systems, and vision, and their parallel hardware realizations. Three machines, covering a wide range of fundamental properties of parallel processors, namely module granularity, concurrency control, and communication geometry, are reviewed: the Connection Machine (a fine-grained SIMD hypercube), DADO (a medium-grained MIMD/SIMD/MSIMD tree-machine), and the Butterfly (a coarse-grained MIMD Butterflyswitch machine).

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-08.

THREE HIGHLY PARALLEL COMPUTER ARCHITECTURES AND THEIR SUITABILITY FOR THREE REPRESENTATIVE ARTIFICIAL INTELLIGENCE PROBLEMS

Ron Katriel

MS-CIS-88-08
LINC LAB 97

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

February 1988

Acknowledgements: This research was supported in part by DARPA grants NOOO14-85-K-0018, NSF-CER grant MCS-8219196 and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

Three Highly Parallel Computer Architectures and Their Suitability for Three Representative Artificial Intelligence Problems

Ron Katriel
Computer and Information Science Department
University of Pennsylvania
Philadelphia, PA 19104-6389
katriel@cis.upenn.edu

September 29, 1987

Abstract

Virtually all current Artificial Intelligence (AI) applications are designed to run on sequential (von Neumann) computer architectures. As a result, current systems do not scale up. As knowledge is added to these systems, a point is reached where their performance quickly degrades. The performance of a von Neumann machine is limited by the bandwidth between memory and processor (the *von Neumann bottleneck*). The bottleneck is avoided by distributing the processing power across the memory of the computer. In this scheme the memory becomes the processor (a "smart memory").

This paper highlights the relationship between three representative AI application domains, namely knowledge representation, rule-based expert systems, and vision, and their parallel hardware realizations. Three machines, covering a wide range of fundamental properties of parallel processors, namely module granularity, concurrency control, and communication geometry, are reviewed: the Connection Machine (a fine-grained SIMD hypercube), DADO (a medium-grained MIMD/SIMD/MSIMD tree-machine), and the Butterfly (a coarse-grained MIMD Butterfly-switch machine).

Contents

1 Introduction 5

1.1 The Need for Parallelism in AI 5

1.2 Parallelism and NP-completeness 5

1.3 Three Representative AI Domains 6

1.3.1 Why These Domains? 6

1.4 Three Representative Highly Parallel Computers 7

1.4.1 Why These Machines? 7

1.5 Outline of the Paper 8

2 Introduction to Parallelism 9

2.1 Theoretical Limitations of Parallel Computation 9

2.2 A Taxonomy of Parallel Processing 10

2.3 Practical Architectures: Narrowing the Space 10

2.4 SIMD (Array) Machines 11

2.4.1 Organization of an Array Computer 11

2.4.2 Interprocessor Communications in Array Processors 13

2.4.3 SIMD Programming 15

2.5 MIMD Machines (Multiprocessors) 15

2.5.1 Memory Organization in Multiprocessors 16

2.5.2 Synchronization, Scheduling, and Resource Sharing 18

2.5.3 MIMD Programming 18

3 Overview of the Chosen Machines 19

3.1 The Connection Machine 19

3.1.1 Introduction 19

3.1.2 The Connection Machine Design 19

3.1.3 Programming the Connection Machine 21

3.2 DADO 21

3.2.1 Introduction 21

3.2.2 DADO's Design Philosophy 22

3.2.3 Programming DADO 23

3.3 The Butterfly 24

3.3.1 Introduction 24

3.3.2 Butterfly Architecture 25

3.3.3 Programming the Butterfly Parallel Processor 25

4 Parallelism in the Selected Domains **27**

4.1 Massive Parallelism in Semantic Networks 27

4.1.1 Introduction 27

4.1.2 A Taxonomy of Massively Parallel Architectures 29

4.1.3 Shastri's Massively Parallel Representation Language 30

4.2 Parallelism in Rule-Based Expert Systems 30

4.2.1 Introduction 30

4.2.2 Production Systems 31

4.2.3 Memory-Based Reasoning 34

4.3 Parallelism in Vision 35

4.3.1 Introduction 35

4.3.2 Intermediate-Level Vision 36

5 Performance of the Chosen Machines **38**

5.1 Performance of the Connection Machine 38

5.1.1 Semantic Networks 38

5.1.2 Rule-Based Expert Systems 39

5.1.3 Intermediate-Level Vision 40

5.1.4 Comments on the Connection Machine 40

5.2 Performance of the DADO Machine 41

5.2.1 Production Systems 41

5.2.2 Intermediate-Level Vision 43

5.2.3 Semantic Networks 43

5.2.4 Comments on DADO 44

5.3 Performance of the Butterfly Machine 44

5.3.1 Intermediate-Level Vision 44

5.3.2 Semantic Networks 45

5.3.3 Rule Based Expert Systems 46

5.3.4 Comments on the Butterfly 47

5.4 Compare-and-Contrast 48

6 Architectures for the Selected Domains **49**

6.1 Knowledge Representation: Semantic Networks 49

6.1.1 Fundamental Computations 49

6.1.2 Architectural Issues 50

6.1.3 Special Requirements 51

6.2 Expert Systems: Production Systems 52

6.2.1 Fundamental Computations 52
6.2.2 Architectural Issues 52
6.2.3 Special Requirements 53
6.3 Vision (especially Intermediate-level) 54
6.3.1 Fundamental Computations 54
6.3.2 Architectural issues 55
6.3.3 Special Requirements 56

7 Conclusions **57**
7.1 The Communication Network 57
7.2 Shared Memory versus Message Passing 58

List of Figures

1 Functional structure of an SIMD array processor. Source: Hwang and Briggs, “Computer Architecture and Parallel Processing”, Page 24, McGraw-Hill, 1984 12
2 Static interconnection network topologies. Source: Hwang and Briggs, “Computer Architecture and Parallel Processing”, Page 335, McGraw-Hill, 1984 13
3 Functional design of an MIMD multiprocessor system. Source: Hwang and Briggs, “Computer Architecture and Parallel Processing”, Page 26, McGraw-Hill, 1984 16
4 Block diagram of The Connection Machine. Source: Frenkel, “Evaluating Two Massively Parallel Machines”, CACM, Volume 29, Number 8, August 1986 20
5 Functional division of the DADO tree. Source: Stolfo, “The DADO Parallel Computer”, Technical Report, Department of Computer Science, Columbia University, 1983 23
6 A 16-input 16-output Butterfly switch. Source: “The Butterfly Parallel Processor Overview”, BBN Technical Report Number 6148 (Version 1), Bolt Beranek and Newman Inc., March 1986 26
7 Example of a KL-ONE semantic inheritance network. Source: Brachman and Schmolze, “An Overview of the KL-ONE Knowledge Representation System”, Cognitive Science, Volume 9, page 196, 1985 27
8 Example OPS5 rule system. Source: Miranker, “TREAT: A Better Match Algorithm for AI Production Systems”, AAAI87 paper draft, January 1987 32
9 An example RETE match network. Source: Miranker, “TREAT: A Better Match Algorithm for AI Production Systems”, AAAI87 paper draft, January 1987 33

1 Introduction

1.1 The Need for Parallelism in AI

Virtually all current Artificial Intelligence (AI) applications are designed to run on sequential (von Neumann) computer architectures. As a result, current systems do not scale up. As knowledge is added to these systems, a point is reached where their performance quickly degrades. Thus paradoxically, programs become slower as they become smarter. As a consequence, current systems can perform "intelligently" only in limited domains and suffer from *brittleness.* Furthermore, current systems do not generally perform in real time.

Several techniques have been identified for speeding up AI programs on sequential machines. They include improved instruction set design, faster clock rates, custom VLSI hardware and the use of advanced AI language compilers. A von Neumann computer, however, does not utilize its hardware efficiently; most of the chip area is memory and only a few memory locations are accessed at a time. The performance of the machine is limited by the bandwidth between memory and processor. This is what Backus calls the *von Neumann bottleneck.* The bigger one builds machines, the worse it gets.

The von Neumann bottleneck is avoided by distributing the processing power across the memory of the computer. In this scheme the memory becomes the processor (this can be viewed as "smart memory"). Attempts to use parallelism to speed up existing AI systems have met with limited success. Often a speed-up of four times is the best that can be achieved. The major reason for this is that current AI programs (when converted to run on parallel machines) tend to have *sequential bottlenecks* which dominate in many cases the running time of the entire program.

1.2 Parallelism and NP-completeness

In a recent interview, Richard Karp said:

> "Most of the work that the theoretical computer science community has been doing on parallel computation has been concerned with making polynomial-time algorithms even faster ... [Theoreticians] may say "Focusing attention on applying parallelism to NP-Complete problems is hopeless, as you can never reduce run time from exponential to polynomial by throwing processors at a problem, unless you have an exponential number of processors. On the other hand, ..., parallelism may really help us curb combinatorial explosions." [Frenkel 86a]

Many NP-complete problems in AI are well suited to parallel computation. Most of these problems involve heuristic state-space search algorithms. Important classes of such problems are branch-and-bound algorithms (i.e., best-first search), divide-and-conquer algorithms (a.k.a. means-end analysis) and backtrack-search algorithms (e.g. depth-first search used in Prolog).

1.3 Three Representative AI Domains

1.3.1 Why These Domains?

This paper highlights the relationship between three representative artificial intelligence application domains and their parallel hardware realization. I have decided to focus my attention on the three important domains in artificial intelligence, namely *knowledge representation* (using semantic networks), *rule-based expert systems* (production systems), and *intermediate-level vision* (the Hough transform):[1]

1. *Knowledge Representation:* Semantic networks have received wide attention as a paradigm for knowledge representation. They have clean semantics (based on restrictions of First Order Predicate Calculus), are fairly expressive, and are ideally suited to take advantage of massively parallel architectures. Most existing knowledge representation systems are based on variations of semantic networks.

2. *Rule-based Expert Systems:* Production systems employ a restricted form of inference known as "forward and/or backward chaining" (a.k.a. modus ponens in formal logic). Rules have been used extensively in many systems to capture expert knowledge in areas such as medical diagnosis, mineral exploration, or speech understanding. Among their advantages are modularity, and naturalness.

3. *Vision:* Image processing is a mammoth information-processing task. Massively parallel computers are ideally suited for the low (iconic) level processing. Medium- and coarse-grained parallel computers are more appropriate at the higher (symbolic) levels of vision. The key to vision processing is a flow of communication and control both up and down through all representation levels.

The three domains span a wide range of data-structures and computations performed on them:

[1]Please note that there are many other important AI domains: natural language processing, robotics, logic programming, planning, etc. The selected domains represent areas most familiar to the author of this paper.

AI Domain	*Data Structure(s)*	*Operations*
Semantic Networks	directed acyclic graph	inheritance & categorization
Production Systems	forest of trees	database operations (join, select)
Intermediate-Level Vision	mesh or tree	statistics & pattern matching

1.4 Three Representative Highly Parallel Computers

1.4.1 Why These Machines?

Most AI-oriented machines are exploratory systems under development at universities and research labs. One can classify existing parallel computers for artificial intelligence under the following taxonomy [Hwang 87]:

- *Language-Based Parallel AI Machines:*
  1. *List-Processing (Lisp) Machines:* Symbolics, TI Explorer
  2. *Prolog Machines:* ICOT PIM, Tamura machine
  3. *Functional Programming Machines:* ALICE, Rediflow
- *Knowledge-Based Parallel AI Machines:*
  1. *Semantic Networks:* **Connection Machine**, NETL
  2. *Rule-Based:* **DADO**, NON-VON
  3. *Object-Based:* FAIM-1, SOAR
  4. *Neural Networks:* ANZA, MARK IV
- *Intelligent Interface Parallel Machines for AI-Oriented Systems:*
  1. *Speech Recognition:* Harpy, HEARSAY-II
  2. *Pattern Recognition/Image Processing:* Pyramid, CAAPP
  3. *Computer Vision:* **Butterfly**, WARP

The machines chosen for this paper, namely the Connection Machine, DADO, and the Butterfly, are **bold-faced**. They have been selected for the following reasons:

- There is a one-to-one mapping between these three machines and the three chosen AI domains

- All have been realized in hardware and extensively benchmarked and some (the Connection Machine and the Butterfly) are currently being marketed [Lim 86]
- All three machines employ conventional, off-the-shelf components for their processors and switching networks
- They have received wide attention from the AI community, and much information about their design and performance is available
- They cover a wide range of fundamental properties of parallel processors, namely module granularity, concurrency control, and communication geometry [Schwartz 83]:

    1. Thinking Machines' Connection Machine: fine-grained SIMD hypercube
    2. Columbia University's DADO: medium-grained SIMD/MIMD/MSIMD tree
    3. The BBN Butterfly: coarse-grained MIMD Butterfly-switch (shuffle-exchange)

It is important to emphasize at this point that the three machines do not exhaust the space of possible parallel architectures. Due to the lack of space (and time) many other interesting machines such as Intel's Hypercube, NYU's Ultracomputer, Schlumberger's FAIM-1, TRW's MARK IV, etc., have not been covered.

1.5 Outline of the Paper

This paper is organized in a breadth-first fashion. It is possible, however, to read it in a depth-first manner. This would be particularly useful for readers who are interested only in one of the machines or one of the AI domains. Basic familiarity with AI (especially semantic networks, production systems and vision) is assumed, but no background knowledge of parallel processing is needed – a brief but comprehensive overview of parallelism is presented in section 2. It describes and contrasts the salient features of SIMD, MIMD and MSIMD architectures. Section 3 is an overview of the architectures of the Connection Machine, DADO and the Butterfly. In section 4, an introduction to parallelism in semantic networks, rule-based expert systems and vision can be found. Section 5 presents an analysis of the performance of the chosen machines in each of the selected AI domains. In section 6 possible architectures for each of the domains are investigated and contrasted. Finally, section 7 contains a general discussion and conclusions. Readers familiar with certain sections or subsections are encouraged to skip them, particularly if short on time.

2 Introduction to Parallelism

2.1 Theoretical Limitations of Parallel Computation

At the most favorable extreme, certain problems will be totally decomposable into completely independent operations which can proceed in parallel. These can be called *constant time* concurrent algorithms. At the opposite extreme, there exist computational processes which are *completely unparallelizable.*

A general (idealized) model for parallel computation is called the *paracomputer* model: A very large number of (identical) processors share a common memory which can be read from and written to simultaneously in a single cycle (multiple writes are resolved in some way). It is useful to note that a single processor with a large enough memory can simulate an N-processor paracomputer in time $O(N)$; thus an N-processor paracomputer can *never* be more than N times faster than a serial computer.

Though theoretically useful, the paracomputer model is unrealizably powerful. A more realistic model, called the *ultracomputer*, consists of a large number N of communicating processors (each with its own local memory) connected together, where each processor communicates with a fixed number k of other processors (via a favorable interconnection pattern) [Schwartz 80]. It is clear that the N-processor ultracomputer cannot compute faster than an N-processor paracomputer. Since in an ultracomputer one cannot bring more that k quantities together in one place during one cycle of computation, at least $\log_k N$ cycles will be required. This is an attainable limit; there are many interesting $O(\log N)$ ultracomputer algorithms. Note that a paracomputer communication cycle can be simulated by an ultracomputer in time $O(\log N)$.

For highest efficiency it is often important to balance processor interconnection with local processing. In many cases the "inefficiency factor" $\log N$ can be regarded as resulting from an imbalance between the amount of work performed locally within each processor and the amount of interprocessor communication performed. One may be able to restore balance by giving each processor a larger amount of local work and only communicating "preprocessed" data. An algorithm of this sort, which uses N processors in an asymptotically efficient way, can be said to be *completely parallelizable.*

Just because an algorithm can be expressed in concurrent terms is no guarantee that it will run significantly faster on a parallel machine. The true measure of parallelism is how much faster a given program will run on n simple parallel processors compared to how fast it would run on a single processor (and for what ranges of n this is valid). The best one can hope for is a speedup of n; for most existing AI programs (written in traditional computer languages) this seems to be about 4 [Deering 84]. in practice, several factors can prevent realization of linear speedup. Among the most important are the need for interprocessor synchronization, poor algorithm design (not all parallelism inherent in the problem is exposed), contention among the processors for the same resource, and traditional I/O structures which cannot feed a high-performance processor fast enough to avoid processor idle time.

2.2 A Taxonomy of Parallel Processing

The space of parallel processing can be divided along three dimensions, namely concurrency control, module granularity, and communication geometry [Kung 80]. As will be seen later, practical parallel architectures as embodied in existing parallel computers correspond to a small subspace of the cross product of these three dimensions. The three dimensions form a taxonomy of parallel processing[2]:

1. *Module Granularity:* This refers to the maximal amount of computation a typical processor can do before having to communicate with other processors. This dimension is usually divided into three: fine-, medium-, and coarse-grain. It reflects the processing power of individual processors and the size of their memory space (the two are correlated).
2. *Concurrency Control:* Enforces desired interactions among processors so that the overall execution of parallel algorithms will be correct. This dimension can be subdivided into two orthogonal subdimensions: synchronous versus asynchronous control, and centralized control, distributed control, or control via shared data.
3. *Communication Geometry:* The spectrum of possible processor-to-processor and processor-to-memory communication geometries ranges from the simplest but most limited topologies (ring, star) through tree, mesh, hypercube, and banyan, to the ideal and most complex topology (crossbar).

2.3 Practical Architectures: Narrowing the Space

The interesting and useful parallel architectures correspond to a very small subspace of the cross product {module granularity} $\times$ {concurrency control} $\times$ {communication geometry}. It is useful at this point to introduce a classification of high-speed computer architectures, suggested by Flynn

[2]The taxonomy is crude and is by no means meant to be complete.

[Flynn 72]. Flynn has shown that computer systems fall naturally into four classes, along two dimensions: parallelism within the *instruction stream* and parallelism within the *data stream*:

1. SISD (single-instruction stream, single-data stream): A conventional serial computer, exhibiting no parallelism.
2. MISD (multiple-instruction stream, single-data stream): A generally unrealistic architecture for parallel computers (no real embodiment of this class exists).
3. SIMD (single-instruction stream, multiple-data stream): A *vector* or *array* processor in the sense that each instruction operates on a data vector rather than a single operand.
4. MIMD (multiple-instruction stream, multiple-data stream): A computer (sometimes referred to as a *multiprocessor*) composed of N processors, each of which is a complete computer.

Only the last two classes (vector processor and multiprocessor) are of interest for parallel computation. These types of computers are vastly different in how they obtain parallelism of operation. Vector processors (SIMD machines) are synchronous lock-step computers that require central controls and tend to have small granularity. The communication geometry is of major importance. Multiprocessors (MIMD machines) consist of a set of asynchronous processors with relatively large granularities. Memory organization is of major importance, as it affects control. A **hybrid** class, designed to bridge the gap between these two extremes, is referred to as *multiple*-SIMD (MSIMD). Essentially, MSIMD computers are partitioned SIMD computers where each partition is controlled by an independent host. The host machines can be organized as an SIMD, MIMD, or even MSIMD computer. MSIMD computers are particularly useful in image processing.

2.4 SIMD (Array) Machines

The central question concerning the design of array computers is how to build data access and communication facilities so that the array computer is useful for as large a variety of computations as possible (this subject area is still evolving). Most surprising, an efficient algorithm for a serial computer may lead to a relatively inefficient algorithm for a vector computer [Stone 80].

2.4.1 Organization of an Array Computer

The control unit in an array computer is itself a computer. The crucial difference between this processor and others is that it can execute conditional branch instructions. Each instruction is

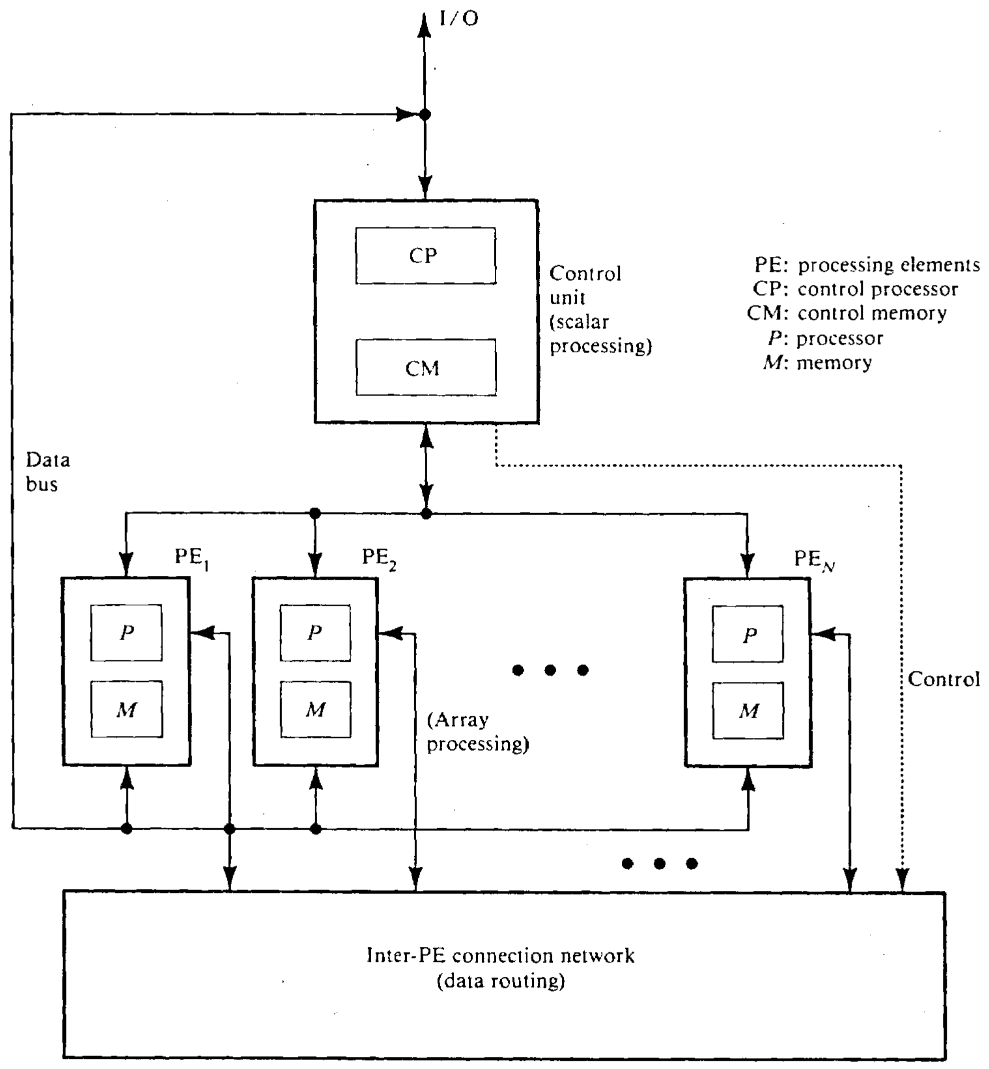


Figure 1: Functional structure of an SIMD array processor

either a control instruction, in which case it is executed entirely within the control unit, or it is a vector instruction and is executed in the processor array (but the instruction stream is much like a conventional serial instruction stream, with each instruction executed in sequence). The control processor has the exclusive privilege to determine which instruction to execute next. Figure 1 depicts the functional structure of an SIMD array processor. A few additional facilities can greatly enlarge the class of problems that can be done efficiently on an SIMD computer. They include masking for conditional branching, a suitable interprocessor communication network, and data skewing support.

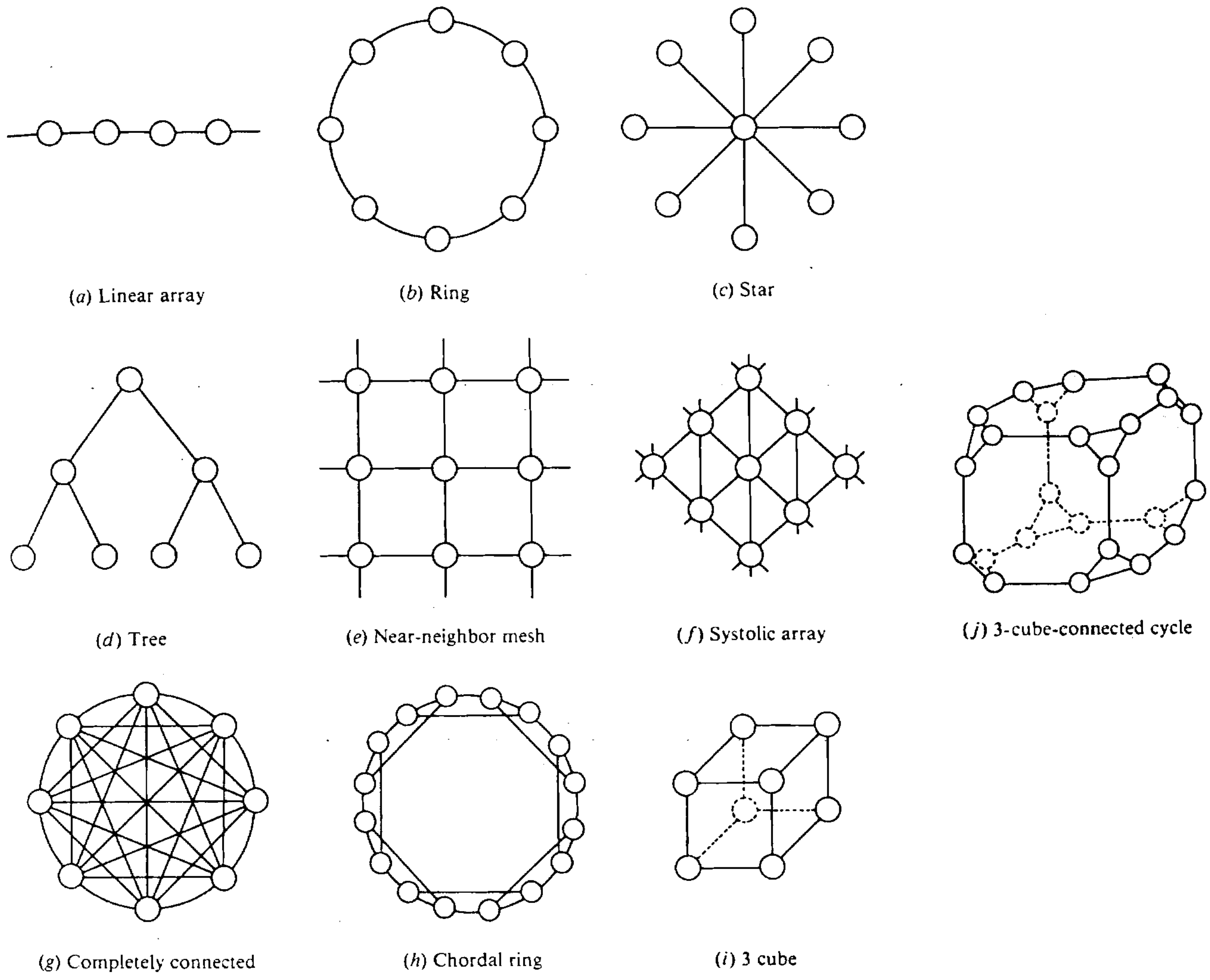


Figure 2: Static interconnection network topologies

2.4.2 Interprocessor Communications in Array Processors

The interconnection network supports data exchanges between processors in an array computer. To maximize parallelism, as much as possible of the available memory and processor bandwidths must be utilized. Data must be stored in such a way as to avoid memory-access conflicts. This calls for careful structuring of data in its storage format. Data structuring for efficient memory access frequently causes data to be fetched in such a way that operand pairs are *not properly aligned* for parallel manipulation. To solve this problem we must install a permutation network in the processor [Siegel 85]. Figure 2 depicts popular interconnection network patterns.

At the two extremes one finds the complete interconnection network (crossbar) and the ring network (linear array with wrap-around). A crossbar interconnection network eliminates contention for communication resources but does not eliminate contention for memory. It is the most general and flexible interconnection scheme, providing total connectivity between N processors at the prohibitive cost (for large N) of N^2. Ring networks are useful only where communications requirements are very small. They are extremely simple both logically (control) and hardware-wise.

One can retain full interconnection flexibility, with logic growth proportional to $N \log_2 N$, if the time to do a permutation is permitted to grow as $\log_2 N$ instead of being constant. A better approach is to reduce the number and hence flexibility of the interconnections, while maintaining as high a speed as possible. The following topologies are examples of such networks, and are presented in order of decreasing complexity (and generality):

- **Banyan Networks:** A Banyan network provides complete interconnections at a cost in switching circuitry that grows as $N \log N$. It has a unique path from each input to each output and is more general than most other cell-based networks. The binary k-cube, perfect shuffle, and tree networks are *graph homomorphic images* of the Banyan. They have lower hardware cost, but higher contention and reduced efficiency.

- **Binary k-Cube** (Hypercube): Contains a rich collection of data paths and is suitable for applications such as sorting. It is one of today's most popular designs [Wiley 87]. Each of the processors is placed at a different corner of the k-dimensional cube and is assigned a binary ID $0 \leq p \leq 2^k - 1$. Two processors are connected iff their IDs differ only in one bit. k connections per processor are required. The largest distance between any two processors is proportional to k.

- **Cube-Connected Cycles** (CCC): The CCC can be viewed as a simulator of the k-cube. Only a constant number of wires per processors is needed. The 2^k processors are arranged in 2^{k-r} rings of size 2^r at the corners of a $(k-r)$-cube. The time complexity for divide-and-conquer is $O(k-r)+O(2^r)$. If k and r are chosen so that $(r-1)+2^{r-1} \leq k \leq r+2^r$ then performance is retained while interconnection complexity is reduced [Preparata 81].

- **Perfect Shuffle** (shuffle-exchange): The shuffle is very similar to the CCC in its properties. It is ideally suited for algorithms having a recursive divide-and-conquer character. The shuffle-exchange performs a permutation of its input similarly to shuffling a deck of cards. First the deck is divided exactly in half, then the two halves are interleaved such that the lower half ends up taking even positions and the higher half odd positions in the newly formed deck (retaining their orders).

- **Mesh** (2D-array): Mesh connectivity is extensively used for image processing, matrix computations and graph algorithms, where many of the computations involve only the small neighborhood of a processor. Only four connections per processor are needed. The topology is regular and efficiently maps onto VLSI circuits. The major drawbacks of a mesh are its large delay ($O(\sqrt{n})$) and the fact that its highly blocking. The problem can be somewhat alleviated by adding wrap-around links (torus).
- **Tree**: Communication between remote leaves faces a bottleneck toward the root, but trees perform well on a rather large range of problems (sorting, matrix multiplication (in $O(n^2)$), and several NP-complete problems). Tree machines can efficiently implement exhaustive search algorithms for several NP-complete problems, but exponential complexity eventually catches up – either in compute time or in number of processors – since speed-up is, at most, linear with the number of processors.

2.4.3 SIMD Programming

The SIMD programming paradigm (which reflects the underlying architecture of an array computer) is *data-level parallelism.* One element of data is stored per processor and the front-end host computer executes a serial program, each step of which can involve computations in all the machine's processors. Such a solution seems far more natural and easy to construct than those typically proposed or implemented for coarse-grained (MIMD) parallel machines. An advantage of using a fine-grained (SIMD) machine is that it allows programmers to write programs which suppose a much larger number of processors that the machine. To quote Hillis, "It is very much like virtual and physical memory". As a fine-grained machine is expanded (by adding processors, memory and communication devices to an existing machine), programs can in many cases take advantage of the increase in computational power without any fundamental changes in design.

2.5 MIMD Machines (Multiprocessors)

The interesting problems for these machines are quite different from those of vector computers. Multiprocessors are suitable for a much larger class of computations than array computers because multiprocessors are inherently more flexible [Stone 80]. It is relatively straightforward to fit a computation to a multiprocessor, which is not the case for vector computations on an array computer. However, to attain high-efficiency computation in a multiprocessor system, one has to solve problems of *task synchronization* and *task scheduling.* In some cases, improper synchronization or

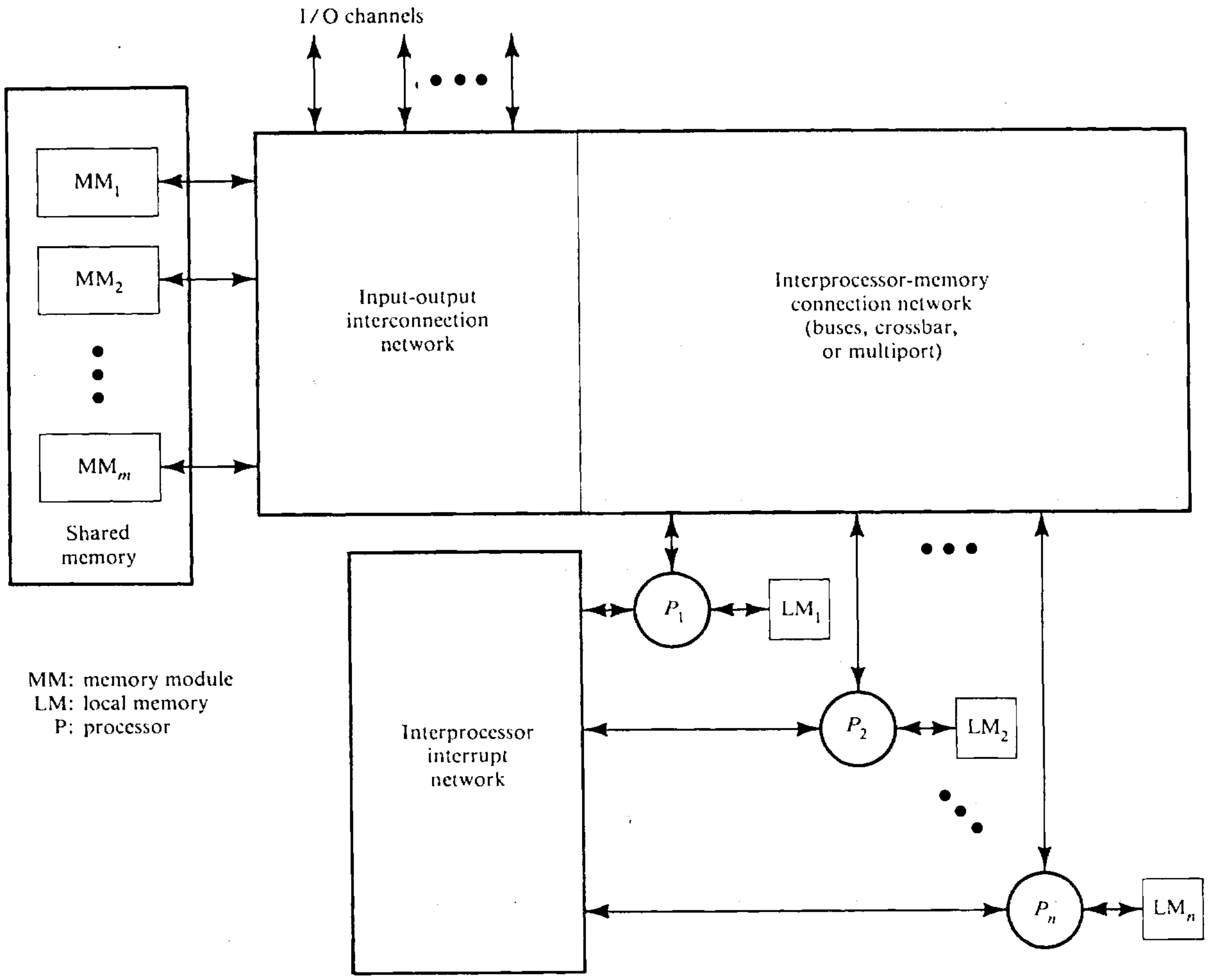

I/O channels
MM_1
MM_2
MM_m
Shared memory
Input-output interconnection network
Interprocessor-memory connection network (buses, crossbar, or multiport)
Interprocessor interrupt network
P_1
P_2
P_n
LM_1
LM_2
LM_n
MM: memory module
LM: local memory
P: processor

Figure 3: Functional design of an MIMD multiprocessor system

scheduling can leads to gross inefficiency, and in extreme situations computation may cease entirely. This is in sharp contrast to array computers. Since synchronization is automatic and scheduling unnecessary (only one task at a time is performed) for array computers, they are free from the problems that surround the multiprocessor, though at the expense of flexibility.

2.5.1 Memory Organization in Multiprocessors

Memory organization in multiprocessors is an important issue, as it most strongly affects the style of programming. Three classes of machines are presented, namely shared memory, message passing, and hybrid multiprocessors. Figure 3 illustrates the functional design of an MIMD multiprocessor system.

Shared Memory Machines A shared memory machine has a single global memory accessible to all processors. Each processor has some local memory (cache or registers), but the operating system usually presents the user with the view of totally shared memory (and often the user is not allowed to explicitly use the cache). Efficiency is the primary motivation for allowing shared memory at the user level. Each process can retrieve the data it requires using hardware primitives; no costly system software need be involved. Often, this approach ignores the problem of synchronization, which can dominate the cost of remote references. A key feature of shared memory systems is that the access time to a piece of data is independent of the processor making the request (barring memory contention, which is as important an issue as in uniprocessors). The aggregate memory bandwidth limits the number of processors that can be accommodated before the system's performance degrades due to contention.

Message Passing Machines Message passing systems are configured so that some memory is local to each processor, but **none** is globally accessible. Message-passing imposes a value-oriented semantics: processes may only communicate values, which may require the exchange of an environment in which to interpret the value (i.e., a pointer value is represented by the object to which it points). This object-oriented approach has proven quite popular for distributed systems composed of many processes. The time it takes for a processor to access data depends on its distance from the processor that currently has the data in its local memory. The design of the communication network between the processors is, therefore, a major concern in a message passing system. In contrast to the shared memory system, the performance of an algorithm will depend on how well the location of the data matches up with it use.

Hybrid Machines Hybrid systems have some of the properties of shared memory systems and some of the properties of message passing. All memory is local to a given processor, but the operating system makes the machine look like it has a single, global memory. Thus, programs are written as if for a shared memory system. However, data must be laid out as if for a message passing system if best performance is to be obtained, since access time depends on the distance between the owner of the data and the requestor (e.g., the **Butterfly**). Even though the penalties for poor data layout are often considerably smaller on hybrid systems, data layout is key to algorithm performance and the aggregate communications speed is a limit on the number of processors that can be accommodated.

2.5.2 Synchronization, Scheduling, and Resource Sharing

Synchronization and interlocking can be done relatively easily using special instructions that manipulate *semaphores.* Each semaphore has a *queue* associated with it. To obtain high-efficiency utilization of processors during a computation, it is usual to remove a suspended process from a processor (placing it in an appropriate queue) and permit another process to proceed on the newly available processor. An important problem in multiprocessing is *deadlock prevention.* A global policy is necessary for dealing with this problem. This policy, traditionally, is enforced by the operating system.

In general, multiprocessor computer systems have all of the resource allocation problems of conventional serial computers, but processor scheduling and memory allocation tend to be the dominant problems. Recent results in the study of algorithm complexity indicate that processor scheduling and memory allocation problems may be so inherently complex that there is no hope of solving them with fast algorithms. In practice, the resource allocation problem is placed in the hands of the user to solve through his assignment of relative priorities.

2.5.3 MIMD Programming

Coarse-grained parallel computers must generally be programmed by means of a method that might aptly be called *control-level parallelism.* This paradigm requires the programmer to divide a program into fragments, one for each of the processors in the machine [Waltz 87]. However, the fit of module memory capacity and the size of a typical data-element is not as natural as in the SIMD case, requiring special software mechanisms. Synchronization is another problem. In writing a program for a coarse-grained machine, one can adhere to concepts much like those used for programming sequential computers; the problems arise in attempting to coordinate the programs. Worse yet, it is often difficult to find parallelism in programs, let alone exactly the right amount of parallelism needed to distribute the work load equally among the (relatively small) number of available processors.

3 Overview of the Chosen Machines

3.1 The Connection Machine

3.1.1 Introduction

The Connection Machine was designed by Daniel Hillis to concurrently manipulate knowledge stored in semantic networks [Hillis 84]. it was designed to be fast at a few very simple operations that are important for artificial intelligence, such as property lookup in a semantic inheritance network. As it turns out, the resulting machine — built by Thinking Machines, Inc. — seems to be quite general. The Connection Machine is at the "radically fine-grained" end of the spectrum of parallel machines [Waltz 87]. It is available in configurations ranging from 16,384 up to a maximum of 65,536 (for the CM-2) processor-memory units.

The design philosophy underlying the Connection Machine is that cost should be in the network, not in its many processors, and that there should be an appropriate balance between the network and processors. A well-tested technology is employed in order to achieve simplicity and reliability. The Connection Machine can operate at its peak processing rate (close to 20 GFLOPS for the CM-2) in a wide range of applications. The key to such flexibility is a communication network that enables the multiprocessors to exchange information in the pattern best suited to the problem at hand [Hillis 87]. Furthermore, Each processor can, under software control, mimic a number of virtual processors. To accomplish this, the memory of each processor is divided up so that each "virtual processor" operates on a smaller amount of memory. The speed penalty for computation has been verified as being approximately linear (up to a million virtual processors).

3.1.2 The Connection Machine Design

Physically, the Connection Machine consists of 4096 chips of proprietary design, each containing 16 processors – each with a local memory of 64 Kbits (for the CM-2) – plus a hardware router (making for 512 Mbytes of total primary storage). All processors execute instructions from a single stream (i.e., in SIMD mode) generated by four microcontrollers under the direction of a conventional host (DEC VAXen or Symbolics 3600 Series Lisp Machines). Each operation can combine two bits from memory with one bit from a register, producing one bit to memory and one bit to a register. See figure 4 for a block diagram of the Connection Machine.

The processor-memory units are wired (via routers) as a boolean 12-cube with 4,096 (2^{12}) corners, one for each chip in the Connection Machine. 24,576 bidirectional wires are used to

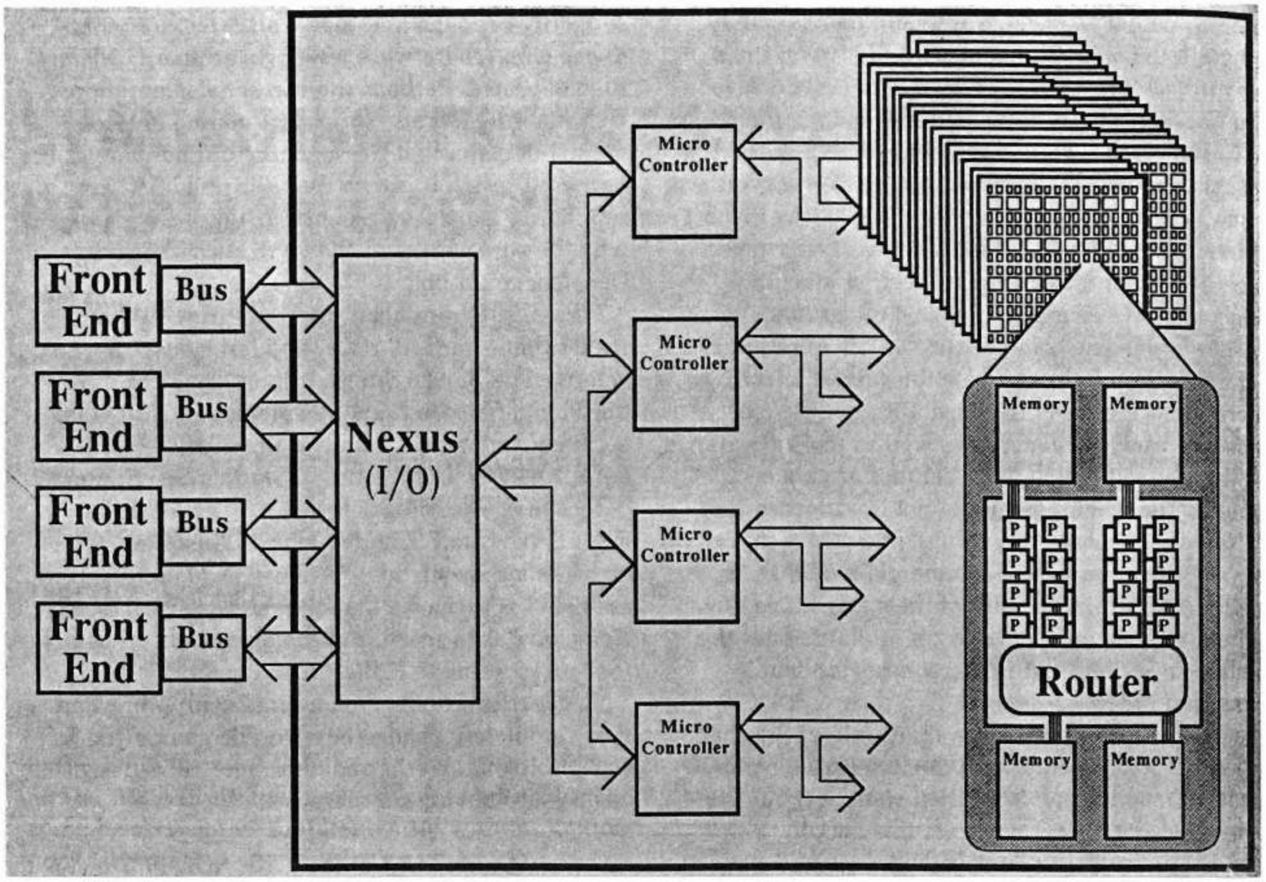


Figure 4: Block diagram of The Connection Machine

connect them. However, no processor is more than 12 wires away from any other. Each cube in the 12-cube has two subcubes, which may be designated 0 and 1 respectively. As a result each corner has a unique address specified by a string of 12 binary digits (bits). Each router can accept a message from one of the processors on the chip (or from a router on a different chip) and send it to either a processor on its chip or another router on a different chip. Each router can also buffer messages if there are no channels available over which to send them [Hillis 85].

Parallel channels support I/O rates of up to 2000 Mbps (for very-high speed devices such as disks and frame buffers). The CM-2 can be used with up to 8 Data-Vault parallel disk storage devices. Each of these contains 42 $5\frac{1}{4}$ inch winchesters that can be read from and written to in parallel. Each unit has a data transfer rate of 40 Mbytes/second, achieving a total transfer rate of approximately 320 Mbytes/second.

3.1.3 Programming the Connection Machine

The Connection Machine programming paradigm (which reflects its underlying architecture) is *data-level parallelism*: one element of data is stored per processor (or virtual processor) and the front-end host computer executes a serial program, each step of which can involve computations in all the Connection Machine's processors. Programmers interact with Connection Machine through a conventional computer, known as the *host*. The processors of the Connection Machine are connected with the host much as a conventional memory unit would be. The Connection Machine can be programmed at the front end with C* and *Lisp, which are data-parallel extensions of Common-Lisp and C respectively. Programs for the Connection Machine are surprisingly similar to conventional programs [Frenkel 86b]. However, learning to write programs for parallel machines requires thinking in ways that are quite different from those demanded by sequential computers.

The basic operations of the machine operate on sets and functions rather than on numbers or pointers. Numbers are just special nodes (concepts) that are recognized by the arithmetic instructions. Four instruction groups are supported:

1. *Set operations:* Intersection, union, difference, complement, clear, etc. These require no messages to be sent.
2. *Propagation:* Projection, restriction, and inheritance. These involve message transmission.
3. *Function manipulation:* Application, modification, composition, etc. This group of operations (which support database join) gives the Connection Machine its additional power over marker propagation machines.
4. *Arithmetic:* Associative functions such as sum, multiply, min/max, and/or, etc., and asymmetric operations such as subtract.

3.2 DADO

3.2.1 Introduction

The ultimate goal of the DADO project group is the design and implementation of a cost effective, high performance, *attached rule processor* that is driven by a conventional host machine and is capable of rapidly executing rule-based software with very large rule sets [Stolfo 86]. The essence of their approach is to execute a very large number of pattern-matching operations on concurrent hardware. Although DADO was designed specifically for the acceleration of OPS-style production systems, it has been found to suit a larger class of problems called *almost decomposable searching problems.*

3.2.2 DADO's Design Philosophy

DADO is a family of special purpose highly parallel tree-structured computers in which memory and processing are extremely intermingled [Stolfo 87]. The distinction between the members of this family is granularity, the storage capacity and processor functionality at an individual processing element (PE). Each PE is a fully programmable microcomputer with a modest amount of local memory (16-20 KBytes in DADO2). A single conventional host adjacent to the root of the DADO tree controls the operation of the entire ensemble of PEs.

DADO's execution modes are rather unique. Each PE may operate in SIMD mode whereby instructions are executed as broadcast by some ancestor PE in the tree. Alternately, a PE may operate in MIMD mode by executing instructions from its local RAM (memory). Such a PE may, however, broadcast instructions for execution by descendant PEs operating in SIMD mode. This allows DADO to be fully partitioned into a number of distinct "sub-DADOs", each executing a distinct task. This mode is usually referred to as *multiple*-SIMD (MSIMD) [Siegel 81]. Figure 5 illustrates a possible configuration of DADO for production systems execution. The PM level operate in MIMD-mode while the upper tree and the WM-subtrees operate in SIMD-mode. A fourth mode, termed SPMD (for **single-program, multiple-data** stream), provides parallel remote procedure invocation in the style of SIMD processing. The procedures are stored locally within the PEs, operate autonomously and, therefore, may take different amounts of time to complete. Machine-level instructions are not broadcast and executed in lock-step. Rather, addresses of prestored code are broadcast to PEs for local execution.

The designers of DADO introduced a special mechanism designed to accelerate certain basic computations on the machine. The *min-resolve* circuit, as it's called, uses the combinatorial hardware in the I/O switch to select one out of all the enabled PEs in a single instruction cycle [Gupta 84a]. It can be used to calculate (in one cycle) the minimum value of a set of values distributed one to a PE. Furthermore, the PE with the minimum value is marked (ties are arbitrated in hardware according to a fixed PE ordering scheme). One use of the min-resolve circuit is to identify the maximally-rated conflict set instance from all PEs storing production rules. The mechanism should also useful in certain vision applications.

DADO1 had 15 PEs connected as a complete binary tree. DADO2, the fourth prototype in

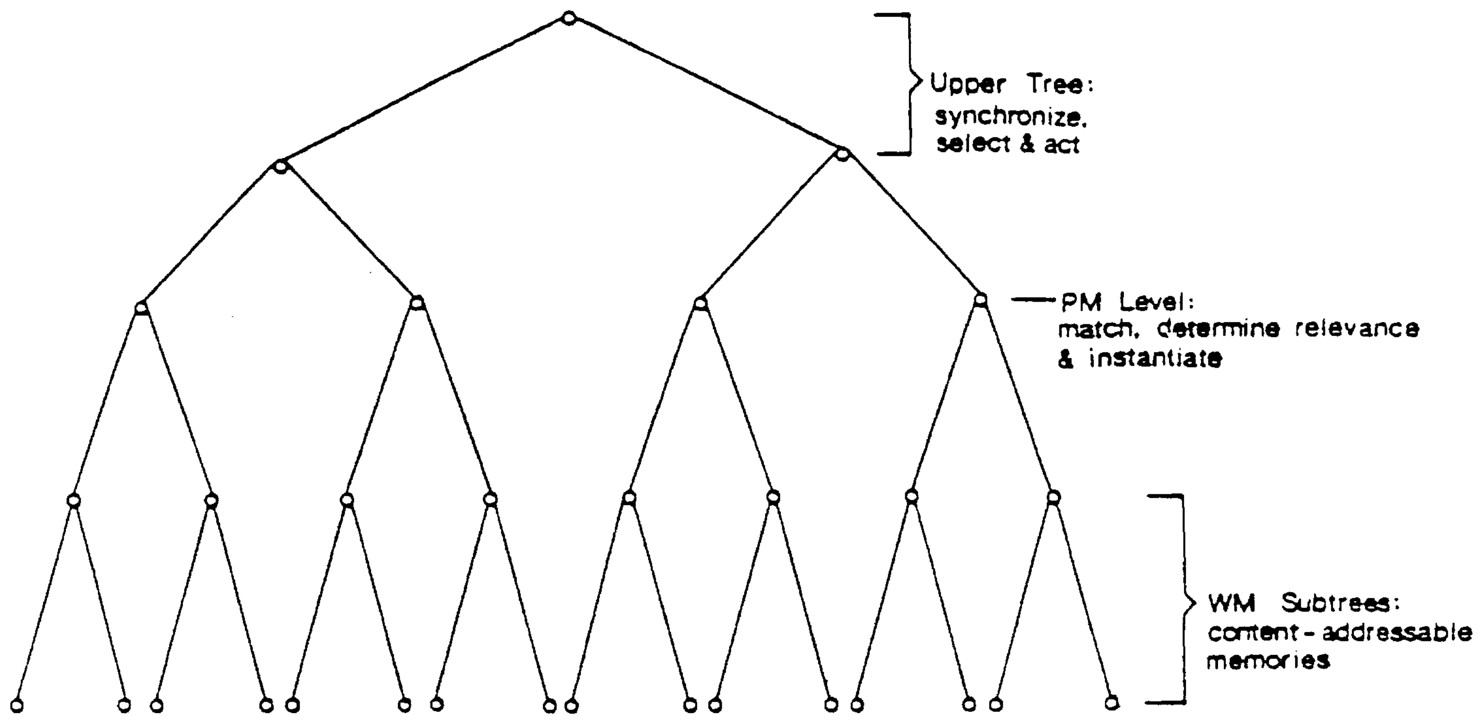


Figure 5: Functional division of the DADO tree

the DADO series (in operation since December 1985), is a medium-grained fully populated binary tree-structured multiprocessor incorporating 1023 moderately powerful processing elements (PEs). It employs two physical tree interconnections. A specialized I/O coprocessor for a DADO2 PE was designed to accelerate inter-PE communication. This also provides a measure of fault tolerance. DADO was designed around commercially available, state-of-the-art technology. The Intel 8751 was chosen at the time (1983) as the microprocessor for the PEs (it was the only commercially available single chip microcomputer in existence that provided 4 parallel 8-bit ports). Initial performance results for OPS5 rule matchings indicate that DADO2 is 2 to 31 times faster than a VAX 11/750.

3.2.3 Programming DADO

As mentioned above, DADO can be programmed efficiently for a class of problems called *almost decomposable searching problems.* The class includes Prolog-like backward chaining logic programming formalisms, relational database operations, and statistical pattern recognition. Each of these problems shares the same common programming paradigm [Stolfo 87] on DADO:

- **Distribute** an initial set of data to the processors

- **Broadcast** the data to other processors in constant time
- **Match** (in each processor) a query against the data
- **Resolve** to find the best answer in constant time
- **Report** the final answer to the root of DADO

Three high level parallel programming languages are available on DADO. All are straightforward extensions of (serial) conventional C, PL/M, and Portable Standard Lisp (PSL). These extensions involve adding SIMD variables to the languages (i.e., special vectors) and extending some operators to operate on them in parallel.

3.3 The Butterfly

3.3.1 Introduction

The Butterfly is a coarse-grained, shared-memory, expandable MIMD parallel computer built by Bolt Beranek and Newman (BBN) [BBN 86]. The computer got its name from the Butterfly switch which it uses for interprocessor communication. The switch supports a processor-to-processor bandwidth of 32 Mbits/second. One processor (Motorola MC68000) and 1 to 4 MBytes of memory are located on a single board called a Processor Node. The processor is capable of executing 500,000 instructions per second. The memory architecture, implemented by the operating system in conjunction with a micro-coded coprocessor, provides the user with the *illusion* of shared memory. Remote memory references (via the Butterfly switch) take about five times longer than local references. The speed of the processors, memories, and switch are balanced to ensure that none become a performance bottleneck.

The Butterfly's architecture scales in a flexible and cost-efficient fashion from 1 to 256 processors. When using 256 processors the computer has a raw processing power of 128 MIPS and a main memory of 256 to 1024 MBytes. For applications such as matrix multiplication, gaussian elimination, convolution, and histograms of images nearly linear speed is achieved through the entire range of processors (up to 256)[3]. To date, the Butterfly has been mainly used for research in image processing and computer vision. Symbolic processing and AI applications are currently being developed (object-oriented programming and rule-based expert systems).

[3]Note that most of these applications can be efficiently computed in parallel on mesh-connected machines.

3.3.2 Butterfly Architecture

Each Butterfly node contains an 8Mhz Motorola MC68000 microprocessor with 24 bit virtual addresses, at least 1 MByte of main memory, a 2901-based bit-slice microcoded co-processor called the Processor Node Controller (PNC), memory management hardware, an I/O bus, and an interface to the Butterfly switch. The PNC interprets every memory reference issued by the 68000 and is used to communicate with other nodes across the switching network. It also provides, in microcode, efficient test-and-set and queueing operations, a process scheduler, and communication synchronization mechanisms. The memory management unit is used to translate virtual addresses (used by the 68000) into physical memory addresses. As a result, the memory of all Processor Nodes, taken together, appear as a large single global memory to application software.

The Butterfly switch is a collection of 4×4 switch elements (4-input 4-output crossbars) configured as a "serial decision" network, a topology similar to that of the Fast Fourier Transform Butterfly. An N-processor system uses $(N \log_4 N)/4$ switches arranged in $\log_4 N$ columns. Switch operation is similar to that of a packet switching network. Figure 6 illustrates a 16-input 16-output Butterfly switch. To reduce *switch contention* a large configuration (e.g., a 128-node Butterfly) contains extra switch nodes, used to provide alternate communication paths between processors. This also makes the switch more resilient to switching node failures. Machines are configured so that the probability of message collision within the switch is relatively low (typical contention overhead is 1% to 5%). The switch supports efficient transfer of blocks of data between any pair of Processor Nodes at full switch bandwidth. The Butterfly I/O system is *distributed* among the Processor Nodes. The I/O bus on each processor supports connections to a Multibus (for fast I/O devices such as disks, and external memory and processors) and serial RS-232 lines (for terminals).

3.3.3 Programming the Butterfly Parallel Processor

The Butterfly Parallel Processor is programmed exclusively in high-level programming languages (C, Symbolics 3600 compatible Lisp, and Fortran). Editing, compiling and linking, downloading, running and debugging of programs are done from a UNIX front-end (VAX or Sun workstation). A window manager enables rapid switching between the front-end and the Butterfly system environments. The Chrysalis operating system contains application libraries for allocating memory, setting up processes, etc., and low-level subroutines (the *kernel*) callable from the user's program. The most significant application library is the *Uniform System Library* which creates an environment

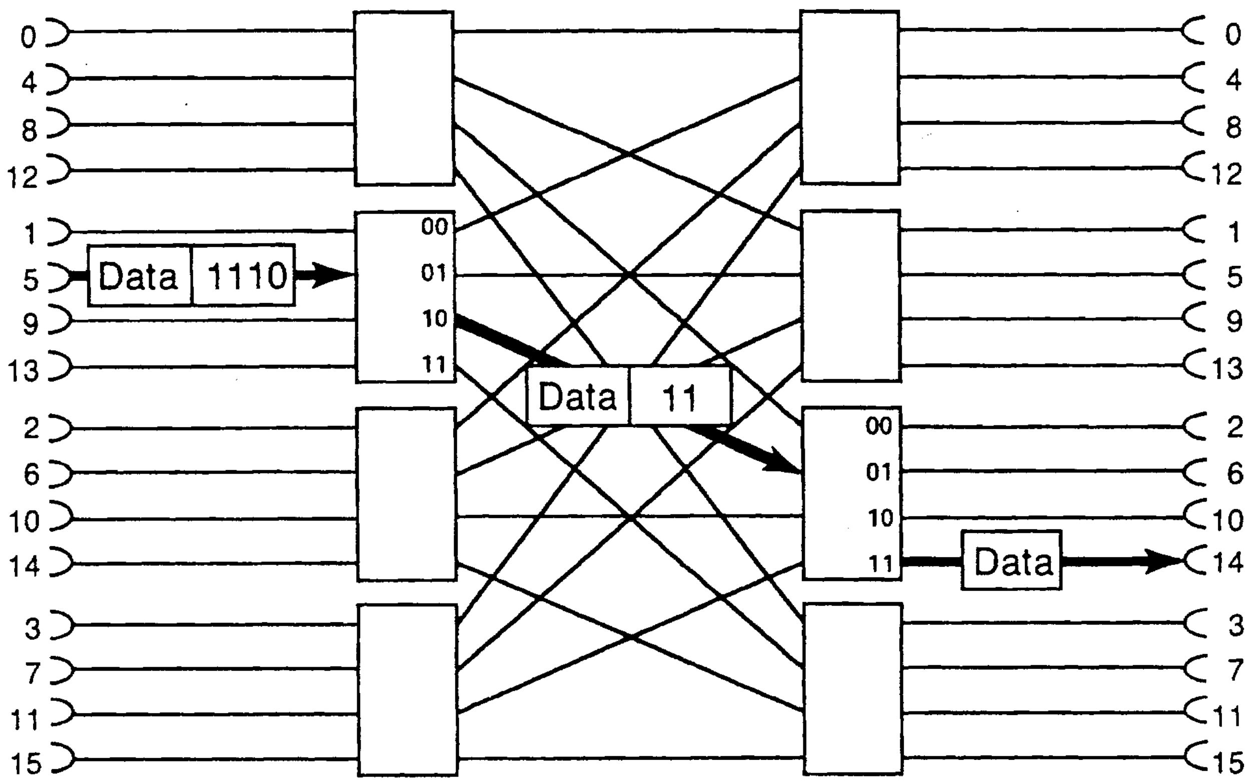
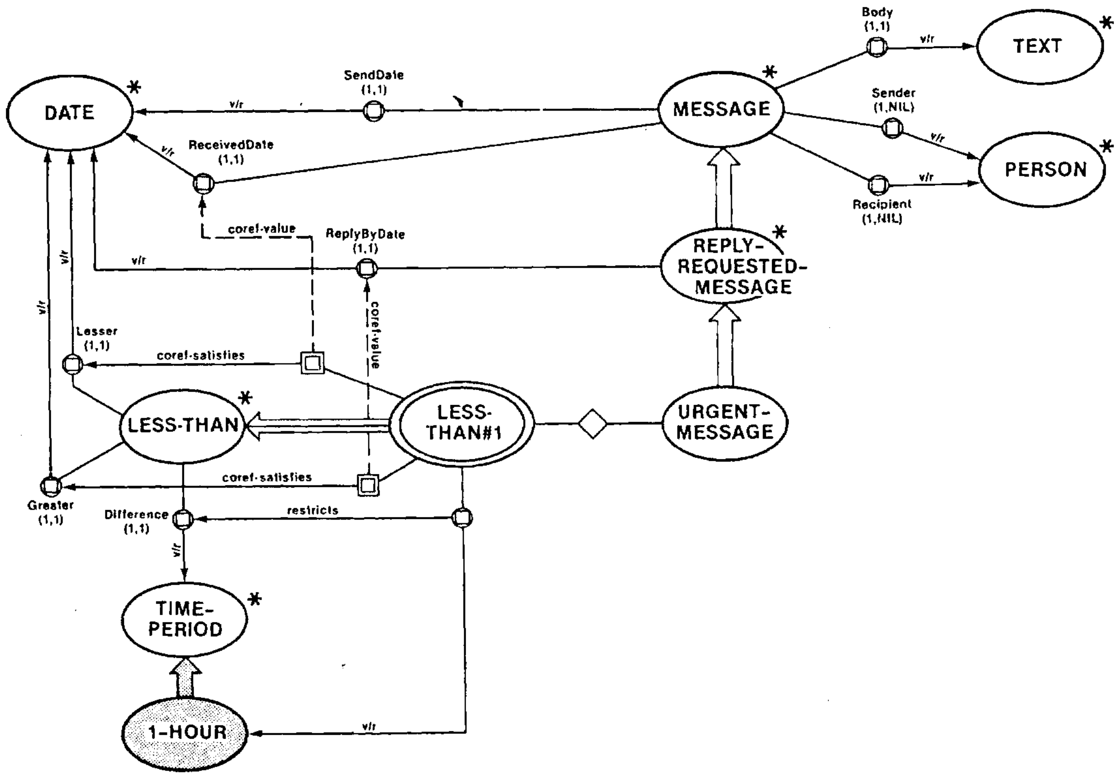


Figure 6: A 16-input 16-output Butterfly switch

where all processors share a single common address space. It provides subroutines to efficiently allocate data structures for the problem.

Two distinct approaches to programming the Butterfly have seen widespread use: message passing and shared memory. When using the message passing paradigm the programmer decomposes the application into a moderately sized collection of loosely coupled processes which from time to time exchange control signals or data (using general communication primitives). This approach is similar to programming a multiprocess application for a uniprocessor. In the shared memory approach, a task is usually some small procedure to be applied to a subset of the shared memory. A task, therefore, can be represented simply as an index, or a range of indices, into the shared memory and an operation to be performed on that memory. This style is particularly effective for applications containing a few frequently repeated tasks (e.g., scientific computing). Memory and processor management are used to keep all memories and processors equally busy.



"A REPLY-REQUESTED-MESSAGE is, among other things, a MESSAGE with a ReplyByDate, which is a DATE."

"An URGENT-MESSAGE is a REPLY-REQUESTED-MESSAGE whose ReceivedDate and ReplyByDate satisfy a LESS-THAN whose Lesser is the ReceivedDate, whose Greater is the ReplyByDate, and whose Difference is 1-HOUR."

Figure 7: Example of a KL-ONE semantic inheritance network

4 Parallelism in the Selected Domains

4.1 Massive Parallelism in Semantic Networks

4.1.1 Introduction

The building block of semantic networks are *concepts.* Concepts, represented as nodes in a semantic networks, roughly correspond to the notion of concepts in natural language, but have more precise definitions. Concepts derive their meaning from the *properties* and the corresponding *property-values* they have and from their position in a subsumption hierarchy, expressed via *IS-A links.* Concepts higher in the conceptual hierarchy denote more abstract (general) concepts while concepts at the bottom of the hierarchy are usually referred to as *individual concepts* (instances). For best economy of representation, properties should be attached at the highest appropriate level of abstraction. Figure 7 illustrates a fairly complex KL-ONE semantic network.

Semantic networks support two important forms of reasoning, namely *inheritance* and *categorization* (or *classification*). There substantial evidence that both inheritance and classification are important forms of human reasoning and that people are very good at doing both. Inheritance allows an agent to infer ("by inheritance") properties of a concept based on the properties of its ancestors. A major problem for inheritance is the presence of *exceptions* and conflicting information. Categorization (or classification) is the dual of the inheritance problem. Unlike inheritance, which seeks some property value of a given concept, categorization seeks a concept that has some specified property values [Shastri 86].

Traditional representation languages include KL-ONE [Brachman 85] and its descendants, KL-TWO [Vilain 85] and NIKL [Moser 83]. They usually have two major components: a definitional language (conceptual taxonomy) and a propositional language (fact database). Semantically, they are based on restrictions of First Order Logic with Lambda Abstraction (FOL+). These restrictions trade computational power for computational tractability. The classification issues in KL-ONE have been studied extensively and are reasonably well understood. The use of a classifier ensures that the resulting knowledge base is sound, complete, and efficiently structured. Classification has been shown to be NP-complete, even for very restricted versions of KL-ONE [Brachman 84].

As an alternative to traditional representation languages, Shastri and Feldman [Shastri 85b] [Shastri 85a] have proposed to integrate evidential reasoning into semantic networks. The evidence combination rule employed in their system is a generalization of the Dempster-Shafer evidence combination rule. Traditional representation languages do not appropriately handle incompleteness, inconsistency and uncertainty. Shastri's language offers a uniform treatment of inheritance and categorization problems, including those that involve exceptions, multiple hierarchies (lattices rather than trees), and conflicting information. A major advantage of the language is that it can be implemented on a massively parallel computer. An account, accompanied by a figure, is given below.

The Need for Parallelism in Semantic Networks Knowledge Representation in AI involves more than just looking up a fact in a table. If knowledge is stored in a semantic memory, then finding the relevant information may involve searching the entire network. This can take time that is exponentially proportional to the size of the network. In 1968, Quillian [Quillian 68] proposed that information stored in a semantic network could be manipulated by concurrently propagating markers through the network. Such a system would be able to retrieve information in a time that was essentially independent of the size of the network. The basic idea was extended considerably by Fahlman [Fahlman 82], Woods [Woods 78] and others.

The following closely follows [Shastri 86]. Ideally, each concept (node) should be allocated to a distinct processor and the interconnections between these processors should be flexible enough to represent the relations between the corresponding concepts. This requires a large number of processor elements with programmable logical connections so that the topology can be configured to suit the problem. This form of parallelism, which has been termed "massive parallelism", provides a phenomenal increase in raw computing power. Due to technological constraints, processors tend to be simple and have little local memory, and the interconnection network topology is regular. However, the memory is distributed throughout the machine and is more intelligent (can modify itself in many places simultaneously through the local processors).

4.1.2 A Taxonomy of Massively Parallel Architectures

Fahlman [Fahlman 82] has proposed a division of massively parallel systems into three classes:

1. *Marker-Passing Systems:* This is the simplest family and the most limited. Communications among processing elements is in the form of single-bit markers. Each node has the capacity to store a few distinct marker bits and perform simple boolean operations on markers. Links between nodes are, in effect, dedicated private lines, so a lot of marker traffic can proceed in parallel. Fahlman's **NETL** is an example of such as system. A major problem with his architecture was its use of an external machine to control propagation, which introduces a sequential bottleneck into the system [Bic 85].

2. *Value-Passing Systems:* These systems pass around continuous quantities or numbers and perform simple arithmetic operations on these values (e.g., traditional analog computers). They never suffer from contention. Many of the iterative relaxation algorithms that have been proposed for solving low-level image processing problems and spreading-activation models of semantic processing are ideally suited to value-passing architectures. Connectionist networks also fall under this category. Their units are patterned after neurons (though they represent a gross over-simplification). For a detailed description see [Feldman 82], [Rumelhart 86], and [Fahlman 87].

3. *Message-Passing Systems:* These are the most powerful and by far the most complex systems. Such generality has a price: individual computing elements are complex, communication costs

are high, and there may be severe contention and traffic congestion problems in the network. Many message-passing systems with well-defined semantics have been proposed (e.g., KL-ONE). Also, most parallel computers fall into this category (e.g., the Connection Machine).

4.1.3 Shastri's Massively Parallel Representation Language

Shastri [Shastri 85a] [Shastri 86] has successfully applied the connectionist paradigm to his theory of evidential reasoning in semantic networks. His system employs distributed control (i.e., spreading activation) and clever mechanisms to efficiently support inheritance and categorization. The system has been simulated on a conventional computer and performed as expected. The representation language uses special units to encode concepts, properties, and the relations among them. Units have only two states: active or inert. Active units produce an output that is equal to their potential. For concept nodes, the potential is graded and represents their level of activity.

Queries are presented to the semantic network via special routine networks. The query network is appropriately interfaced with the semantic network, activating the relevant units in it. Bidirectional relay nodes facilitate the computation along IS-A links (used in inheritance and categorization). At any given time, IS-A links are enabled in one direction only to eliminate the possibility of indefinite oscillation of the network. The network settles down after a period of time that is linearly proportional to the depth of the conceptual hierarchy (logarithmic in the size of the network, i.e., the number of concepts). A winner-take-all network (which is part of the query network) arbitrates among the possible candidate answers (if more than one concept is significantly active). The routine network has mechanisms to deal with conflicts and null answers.

4.2 Parallelism in Rule-Based Expert Systems

4.2.1 Introduction

One of the goals of AI is the creation of programs that base their "reasoning" on experience. After several decades of research in AI, rule-based production systems have emerged as one of the most important and widely employed tools for the implementation of expert systems and other AI software. A production system organization facilitates the modular, incremental growth of knowledge bases, and allows for the useful but unplanned interaction of independently-specified rules [Winston 77] [Nilsson 80]. While a few production systems have already found commercial application, their use in certain other domains (especially real-time systems) is precluded by slow execution speeds. A detailed overview of parallelism in the OPS5 production systems follows.

While there has been some progress toward the goal of automatically generating rules of inference ([Quinlan 79]) the construction of intelligent programs has proven to be a difficult task, one that depends on the ability of a highly skilled individual to create sets of rules. Waltz [Waltz 87] believes that memories of specific events, and not rules, are the key to reasoning from experience. This paradigm has been overlooked in the past because sequential (von Neumann) computers are too slow to execute some of the necessary basic operations (like associative memory search). A brief description can be found below.

4.2.2 Production Systems

In general, a production system is defined by a set of rules, or *productions*, that 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). The RHS specifies information that is to be added to (asserted) or removed from the WM when the LHS successfully matches against the contents of the WM. In operation, the Production System repeatedly executes a cycle of three major phases. In the *match* phase one determines, for each rule, whether the LHS matches the current environment of the WM. All matching instances of the rules are collected in the *conflict set* of rules. Next, a *Select* phase is initiated to choose exactly one of the matching rules according to some predefined criterion. Finally, in the *Act* phase, WM elements are added or deleted from WM as specified in the RHS of the selected rule, or some other operation (e.g., I/O) is performed. Typical conflict resolution techniques use recency of a matched data in WM, as well as syntactic discrimination.

The OPS5 Production System The production system language OPS was first described by Forgy and McDermott (1977). Several subsequent versions have appeared, with OPS5 being the most widely known. OPS5 has been evaluated favorably by many researchers and has been used to implement a large and successful commercial production system (McDermott's R1). Its static and dynamic characteristics have been measured on several OPS5 production systems, and even though the language was designed for sequential processing, its speed can be increased significantly by parallel execution. Miranker [Miranker 87] is actively engaged in the development of a production systems language specifically designed for parallel execution. Such a language may well prove better

Rule:

(P example-rule

($A < x >$)

($B < x >< y >$)

($C < y >$)

—>

;no RHS actions)

Initial Working Memory:

(A 1)

(B 1 2)

(B 2 3)

(B 2 4)

(C 3)

(C 2)

Figure 8: Example OPS5 rule system

suited to the capabilities of parallel machines. The syntax of OPS5 is very close to the description in the previous paragraph. See figure 8 for an example OPS5 rule and some relevant working memory elements.

The Development of the OPS5 Matching Algorithm Of the three steps in the production system cycle, the matching phase has proven in practice to be the most time-consuming. According to Forgy [Forgy 79], more than 90% of the execution time in a uniprocessor implementation is consumed by matching. On a sequential machine, the matching phase is an $O(\|PM\|\|WM\|)$ process that compares the LHS of each production in the PM to every statement in the WM. Worse yet, each LHS can have a number of literals in the form of a conjunction. However, careful investigation by Forgy uncovered useful heuristics which can cut down the time spent in the matching phase. Forgy's RETE Match algorithm [Forgy 82] exploits two observations about OPS5:

1. *Structural similarity:* LHS's differ (syntactically) only slightly from each other (matching is speeded-up since changes to the WM can be easily traced to the LHS's they will affect).

2. *Temporal redundancy:* The WM changes very little from production cycle to production cycle (most matches do not have to be recomputed). Furthermore, the changes to WM have few effects on the conflict set.

Hence, a computational savings results if the LHS's of all rules in the production system are compiled into an "augmented discrimination network", or *dataflow graph* [Hwang 84], with state information saved at each node during execution. See figure 9 for an example RETE network (the rule and working memory elements were introduced on page 32.) The production system

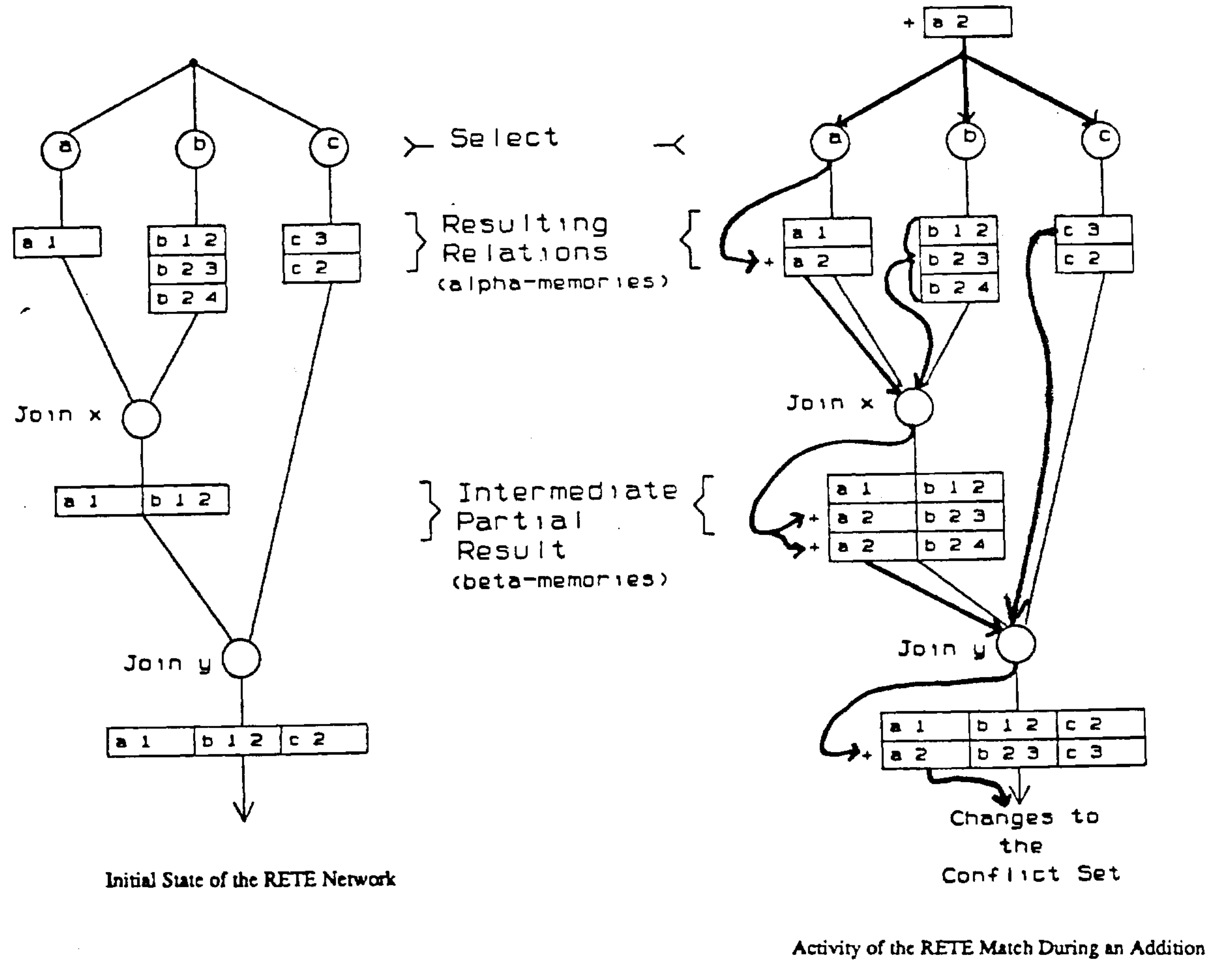

Select
Resulting Relations (alpha-memories)
Intermediate Partial Result (beta-memories)
Join x
Join y
a 1
b 1 2
b 2 3
b 2 4
c 3
c 2
+ a 2
Changes to the Conflict Set
Initial State of the RETE Network
Activity of the RETE Match During an Addition

Figure 9: An example RETE match network

interpreter can then incrementally compute the contents of the conflict set. A typical example is the R1 system [McDermott 82], which incrementally builds a solution to the VAX configuration problem. Production systems that search through large databases, such as ACE [Stolfo 82] or sensor-based systems, are not. McDermott, Newell and Moore conjectured (1978) that the cost of maintaining the state information exceeds the cost of comparisons that otherwise would have to be recomputed. Despite this, it is assumed that RETE is the best algorithm for production system matching.

The OPS-RETE Match Algorithm The RETE algorithm compiles the left-hand sides of production rules into a discrimination network. Changes to the working memory serve as the input to the network. The network, in turn, reports changes to the conflict set. The network contains two categories of nodes: *test nodes* and *memory nodes*. Additions or deletions to the WM are recorded in memory nodes (α-memories). A pointer to the change, called a *token*, is then replicated and passed to a number of entry points into the network. Once a token updates an α-memory, it continues to propagate through the network. Following the α-memories are tests nodes, called the *two-input test nodes*, which test for consistent variable bindings between two condition elements. If two tokens satisfy the test, a new token is formed at the output arc of the test node and stored in a token memory, called a β-memory. Tokens that propagate from terminal β-memories in the network (one terminal node per production) reflect changes to the conflict set.

4.2.3 Memory-Based Reasoning

The key operation in memory-based reasoning is weighted associative search for items in memory that are similar to a present case that one hopes to understand or act appropriately on [Waltz 87]. Once some matches have been found, they can suggest hypotheses. Other operations, which are based on statistics, test these hypotheses; to do so, they separate important features from unimportant ones. Waltz's example from the medical diagnosis domain should clarify things. It is assumed that the system has access to a large relational database of medical patient records. A memory-based reasoning program can find diagnostic hypotheses for a new patient as follows:

1. Load each of the processors with a database item (patient record)
2. Broadcast each feature of the new patient to all processors
3. Have each processor compute a numerical measure of closeness of its data record to each feature of the item under consideration and adding them up
4. Select from the database patients whose total scores are closest to that of the new patient

Next, a hypothesis-testing phase is run. If the results suggest that only one diagnosis is plausible, we are done; if more than one diagnosis remains, a second hypothesis-testing method is invoked, which either completes the diagnosis, or proposes tests that would differentiate among the remaining possibilities. This is repeated until only one diagnosis remains. Now, if many similar patients received the same diagnosis it is very likely that the patient under consideration should also get the same diagnosis (a rule has been discovered); if only a small number of patients are similar to the new patient this might still warrant a diagnosis (it might be a rare disease); if no patients are similar enough to the new patient the system "knows that it does not know" how to diagnose this patient.

4.3 Parallelism in Vision

4.3.1 Introduction

The image understanding problem can be thought of as an *iconic to symbolic* (or *signal to symbol*) transformation. It can be described as involving three levels of processing, namely low-, intermediate- and high-level processing. The low-level input image data is essentially an array of signal data and forms an iconic representation of the real world. To perform image interpretation the machine must transform this low level information (color and intensity) into high-level symbolic representations of objects in the scene (in terms of predefined knowledge about objects in the world). This task involves monocular static image interpretations as well as integrating information from multiple sensory sources, including stereo input and motion sequences [Weems 84]. The intermediate-level provides an interface between the low- and high-levels of representation. Much work has been done at both extremes of image processing. However, the interface between these two areas is less well understood. This grey area involves which features are to be extracted by the low-level image processing and in what format they should be presented to the image analysis level.

Many architectures deal with images in the most conventional format, namely a large two dimensional matrix of brightness values called pixels (picture elements). Image resolution ranges from real-time (b&w) video data as small as 256×256 pixels with only 6-bits of information per pixel to satellite (color) images that may involve $4,000 \times 4,000$ pixels with as many as 24-bits of information per pixel [Reeves 81]. Since in many cases there is a need for very high speed computation, it is obvious from these figures that many vision applications require parallel processing. Many special special-purpose parallel processing architectures have been proposed and implemented.

Since the focus of this paper is on AI domains, low-level image processing will not be treated (though some consider it to be a border case). The problems encountered in high-level vision are very similar to those found in many AI domains. In particular, matching and search are heavily utilized. Unfortunately, virtually no work has been done on applying parallelism to this area. Specifically, there is no literature available at the time of this writing describing high-level vision on the Connection Machine, DADO, or the Butterfly.[4] Consequently, only intermediate-level vision will be treated in this paper.

[4]This is not surprising in the case of the Connection Machine, due to its fine-grain module granularity. In the case of the Butterfly, though, one would expect differently, as its architecture is ideal for high-level vision.

4.3.2 Intermediate-Level Vision

A common problem in computer image processing is the detection of straight lines in a digitized image. The problem is to detect the presence of groups of co-linear or almost co-linear feature points. An upper-bound on the time complexity of the problem, attained by a naive algorithm on a sequential machine, is $O(n^2)$, where n is the number of feature points. The algorithm simply tests the lines formed by all pairs of points. A better method was proposed by Hough [Hough 62] and substantially improved upon by Duda and Hart [Duda 72]. It is a representative of computationally intensive intermediate-level vision problems and is therefore a candidate for execution on highly parallel machines.

The Hough Transform [5] Essentially, a Hough transform is designed to detect co-linear sets of edge pixels in an image by mapping these pixels into a parameter space (the Hough space) defined in such a way that co-linear sets of pixels in the image give rise to maxima in the Hough space. Formally, given a set of co-linear edge points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, we know that they all must satisfy the normal-angle representation of a line:

$$\rho = x_i \cos(\phi) + y_i \sin(\phi)$$

The key observation is that points lying on the same straight line in the picture plane correspond to curves through a common point in the ρ, ϕ parameter plane. Thus, the problem of finding the set of lines in the image plane is reduced to that of finding common points of intersection of sinusoidal curves in the parameter plane.

The implementation of the Hough transform involves a quantization of the parameter plane into a quadruled grid. A two-dimensional array (the *accumulator array* or the Hough array) is then used to represent the parameter plane grid. For each edge pixel, the algorithm increments the counts in all accumulator array entries that correspond to lines passing through that edge pixel (this can be though of as "voting" by the edge pixels for the parameter values of possible lines passing through these points.) On a sequential machine, the time complexity of the Hough algorithm (on a serial machine) is $O(s+mv)$, where s is the size of the grid (in pixels), m is the number of edge pixels, and v is the number of votes cast by each point (which is inversely proportional to the quantization error along the ϕ-dimension of the parameter space). Once the Hough array is computed, all maxima in the parameter plane are searched for. Since each "real" maxima is surrounded by "false" votes in

[5]This part follows closely [Chandran 86].

its neighborhood, the search is conducted in a small square neighborhood of candidates, perhaps after smoothing the Hough array.

Parallel versions for the Hough transform have been developed for mesh-connected computers, tree machines, and systolic multiprocessors. In the simplest case, the steps involved in detecting large co-linear sets of edges or feature points in a binary picture may be summarized as follows: Let ϕ_{min} to ϕ_{max} be the range of angles that we are interested in, $\Delta\phi$ be the angle resolution, and let *Accum* refer to the accumulator array. Then the simple algorithm is:

```
for each edge point $(x,y)$
        for $\phi = \phi_{min}$ to $\phi_{max}$ by $\Delta\phi$
                begin
                        $\rho = \lceil x \times \cos\phi + y \times \sin\phi \rceil$;
                        $Accum[\rho, \phi] = Accum[\rho, \phi] + 1$
                end
```

There is a high degree of data parallelism inherent in the problem. The Hough transform can be thought of in terms of mapping a three-dimensional $x \times y \times \phi$ space into a one-dimensional ρ space and then incrementing the appropriate (ρ, ϕ) cells by 1. Thus, the processors can be allocated according to one of the following schemes:

1. Distribute processing by x (or y): Assign one row (column) of the binary image to each processor, then compute ρ over the entire range of angles
2. Distribute processing by region: Assign a square shaped region of the input binary image to each processor, then compute ρ over the entire range of angles
3. Distribute processing by ϕ: Assign one angle to each processor, then compute ρ for the entire image (or sub-region of the image)

Method 3 is best as it ensures, in general, an even distribution of work among the processors. Once the accumulator array is computed, one must next find the k highest local maxima. This can be done in constant time on parallel machines equipped with special "resolve" and "broadcast" instructions. One can then construct another array which has as its entries all those elements which are the local maxima in all 3×3 neighborhoods and find the first k values. Efficient parallel algorithms for doing this exist.

5 Performance of the Chosen Machines

5.1 Performance of the Connection Machine

5.1.1 Semantic Networks

The Connection Machine's architecture is based on Fahlman's NETL [Fahlman 83], a parallel architecture for efficiently implementing semantic networks. This AI paradigm is particularly ripe for taking advantage of concurrency because data and processing is tightly coupled and control is very localized. The Connection Machine's architecture captures many of the positive qualities of marker propagation, without some of its weaknesses. Each processing element of the Connection Machine stores a chunk of data, and elements are connected in the same way as the data is (the communication connections are configured to mimic the structure of the specific problem being solved). Since it is impractical to connect elements with physical wires (which would require rewiring for every new problem), processing elements are connected through a switching network and communicate by sending messages.

Data in the Connection Machine is stored as the pattern of connections between cells (this is similar to Lisp, where data is stored as structures of pointers). Connection between nodes is dynamic – any two nodes can talk if they know each other's address. Unconnected cells can establish a connection by a mechanism called *message waves* [Hillis 84]. A similar technique may be used to connect to a cell of a particular type, rather than to a specific cell. Once contact has been established, a *cancel wave* is sent out. The cancel wave travels twice as fast as the message wave, catching up with the latter after it has travelled three times the necessary distance. This method of allocating storage may allow the machine to continue to operate with defective cells (malfunctioning cells can be cut off). None of the Connection Machine's algorithms depend on a cell existing in a specific address.

Nodes in a semantic network can be linked to an arbitrary number of other nodes. A physical cell, on the other hand, can only connect to a few other cells. The solution is to represent each node as a balanced binary tree of cells. In this scheme, each cell needs only three connections. The total number of cells needed to represent a node with c connections is $c - 1$. The overhead associated with this technique is similar to that incurred in Lisp, where 'cons'-cells are used to create lists. However, since trees are used rather than lists, the cost in time is logarithmic in the valence of the simulated vertex. The trees are kept balanced by an elegant scheme of insertion and deletion (invented independently by Carl Feynman and Browning [Browning 80] at CalTech). Links in the semantic network can be represented with a single cell (two connections plus a link type).

Although the Connection Machine has been designed for manipulating semantic networks, it has not been used in practice for this purpose at all (to the best of this authors' knowledge). It isn't surprising, though. To justify the use of the Connection Machine for semantic network processing, one would have to construct a very large knowledge base (say 10000 concepts). This is a formidable task, and there are few domains that have this many concepts. The task of representing in a computer something equivalent to human common sense would definitely qualify, but would probably take hundreds of man-years in effort.

5.1.2 Rule-Based Expert Systems

MBR versus Production Systems Memory-based reasoning, which reasons from examples rather than from rules, has several advantages over traditional Production Systems:

1. Since no rules are used, there are fewer opportunities for inadvertantly introducing inaccuracies (e.g., combining "confidence levels" for a chain of rules)
2. Rule generation (avoided in MBR) has a combinatorially explosive search space of possible rules to contend with and there is never any certainty that the resulting rules set is complete
3. No expert is required; a knowledge engineer only needs to identify database contents and mark them according to whether they are symptoms, features, or optional tests
4. MBR systems can form hypotheses on the basis of even a single precedent, something rule-based systems cannot do (rules are inherently summaries of regularities)
5. If no items in memory are closely related to the item being analyzed, then the system is able to recognize this fact and inform the user

In general, the size of the database improves the quality of the system's reasoning, without significantly changing the time needed for generating hypotheses. Learning can be added to the system by precomputing weights. The precomputed weights can then be used to generate hypotheses with a single lookup operation. Waltz and Stanfill have tested the system on a word-pronunciation task very similar to NETtalk [Sejnowski 86]. The system, called MBRtalk, achieved 65% performance on a database of 4500 randomly chosen English words [Stanfill 86]. A problem with Stanfill and Waltz's memory-based system is its inefficient use of memory space (conventional rule-based expert systems condense a lot of knowledge into every rule). His system trades off space efficiency for completeness, consistency, and ease of acquisition. Without the Connection Machine this scheme would be of theoretical interest only.

5.1.3 Intermediate-Level Vision

The Connection Machine is extremely well suited to low-level image processing. To date, the Connection Machine has been used for high resolution stereo matching, and the processing of visual information to produce contour maps (266,144 pixels) at a rate of 7000 per hour (instead of 15). Both are low-level image processing tasks (on the border of AI and signal processing). The Connection Machine is not well suited for some intermediate-level vision problems, due to its being very fine grained. The Hough transform, however, should run efficiently on the Connection Machine (see the following discussion on the Butterfly's performance on intermediate-level vision problems).

5.1.4 Comments on the Connection Machine

Although the Connection Machine is touted as a massively parallel computer for AI applications, very few AI applications have been run on it. It has been used for simulating physical systems on a very fine-grained level, data-base information (text) retrieval, and low-level image processing. This is possibly the biggest criticism that can be directed at the Connection Machine. A possible explanation for this state of affairs is that AI application design has not caught up yet with the astounding leaps in parallel hardware implementations. There is an urgent need for research in this area.

The Connection Machine differs from most array machines in its mechanisms for arbitrary communications. However, due to its fine granularity it is a special-purpose machine. A possible solution would be to introduce partitioning into the design of the Connection Machine. This would require a large number of microcontrollers, and a very high-rate interface between them and the host machine(s). A 65,536-processor Connection Machine has 128 printed circuit boards with 32 boards currently connected to each of the four microcontrollers. If each printed circuit board had its own (individually) programmable microcontroller, the Connection Machine could be viewed as a 128-processor MIMD machine, where each processor is a 512-processor SIMD parallel processor. This would enormously increase the flexibility of the machine, at a relatively modest cost. The microcontrollers could be interconnected with a dense communication pattern, such as a hypercube.

5.2 Performance of the DADO Machine

5.2.1 Production Systems

The following (very simple) view of a parallel implementation of a production forms the basis for DADO's algorithms. A parallel implementation of the production system cycle requires the partitioning of the PM and WM among available processors: some subset of processors would store and process the LHS of rules, while another (possibly intersecting) subset of processors would store and process WM elements. The basic concurrent algorithms is as follows:

1. Assign some subset of rules to a distinct set of processors
2. Assign some subset of WM elements to a set of processors (possibly distinct from those in step 1).
3. **repeat until** no rule is active:
   (a) Broadcast an instruction to all processors storing rules to begin the match phase (resulting in the formation of local conflict sets).
   (b) Considering each maximally rated instance[6] within each processor, compute (in parallel) the maximally rated rule within the entire system. Report its instantiated RHS.
   (c) Broadcast the changes due to the rule reported in step 3 (b) to all processors, which update their local WM accordingly.

In particular, I will focus on two different algorithms, referred to as "The Original DADO Algorithm" and "The Full Distribution Algorithm" [Stolfo 86]. These two algorithms point out different characteristics that may occur in various rule-based programs. The Original DADO Algorithm directly supports rule-based programs that operate upon and with large WM databases. The Full Distributed Algorithm is intended to support programs whose rules include conditions that do not match a large number of distinct WM elements.

The Original DADO Algorithm The original DADO algorithm makes direct use of the machine's ability to execute in both MIMD and SIMD modes of operation at the same point in time. The machine is logically divided into three conceptually distinct components: a *PM level*, an *upper*

[6] A maximally rated instance in a conflict set is a rule with a maximal rating based on some predefined ordering scheme.

tree, and a number of *WM subtrees.* The PM level consists of MIMD-mode PEs executing the match phase at one appropriately chosen level of the tree. A number of distinct rules are stored in each PM-level PE. The WM subtrees (sub-DADOs) rooted by the PM-level PEs consist of a number of SIMD-mode PEs collectively operating as a hardware content-addressable memory. WM elements relevant to the rules stored at the PM-level root-PE (i.e., data elements that match some pattern in the LHS of the rules) are fully distributed through the WM subtree. The upper tree consists of SIMD-mode PEs lying above the PM level, which implement synchronization and selection operations.

The Original DADO Algorithm was specifically designed for Production Systems which are non-temporally redundant, have many WM elements change and many rules affected by the WM changes in each cycle (saving state between cycles, as in RETE, has few advantages), and which have a relatively small number of production rules but have a large WM. The Original DADO Algorithm performs the match phase in time that is independent in the number of productions. To achieve this, one needs as many processors as there are productions (approximately 100 PEs per production). However, the RETE algorithm implemented on a serial computer performs the match in a time that is virtually independent of the number of productions [Gupta 84a]. The reasons for this poor performance is that while attempting to implement temporally redundant systems, this algorithm may recompute many of its matches. This can be remedied by incorporating in the algorithm many of the capabilities of the RETE match. Miranker's TREAT algorithm [Miranker 87] implements these ideas. Another problem with this algorithm is its poor use of the hardware [Gupta 84a]. This results from the fact that only a small number of WMEs are affected in each cycle, an hence only a small portion of PM-level PEs and their associated WM-subtrees will do useful work (less than 2% for a system with 2000 productions.)

The Full Distribution Algorithm: In this case, a very small number of distinct production rules are distributed to each of the DADO PEs, as well as all WM elements relevant to the rules in question. The entire DADO tree alternates between MIMD and SIMD modes of operation. The match phase is implemented as an MIMD process, whereas the selection and act phases are executed as SIMD operations. Each PE executes the match phase for its own small Production System. One such Production System is allowed to fire a rule, however, which is communicated to all other PEs.

The Full Distribution Algorithm was specifically designed for Production Systems which are temporally redundant, have many WM elements change but few rules affected by the WM changes in each cycle (saving state between cycles, as in RETE, is advantageous), and which have a relatively large number of production rules and have a large WM. It is important to note that for some Production System programs the potential speedup of the match for the Full Distributed Algorithm is not nearly as great as might be expected. In programs such as R1 (where few rules may potentially match newly asserted WM elements on each cycle) few PEs perform useful work. In other cases, certain anomalous rules may require more processing time on the average that other rules, thus producing "hot spots" of sequential execution in a distributed environment.

Reported measurements show that in an average OPS5 production system, only about 30 rules are affected by changes to the working memory during each production cycle. Furthermore, the total time spent in each match phase is the time taken by the slowest, not the average, rule. Consequently, the average speedup obtainable in the match phase from production-level parallelism is a factor of about 6 (almost irregardless of the number of production rules). A simple scheme has been devised to overcome these limitations. The essence of this approach, known as the *copy and constrain rules*, is to replicate anomalous rules and to introduce constraints within the copies that restrict them to match smaller disjoint portions of the set of potentially relevant WM elements.

5.2.2 Intermediate-Level Vision

No data is available at this time on DADO's performance on vision problems. The Hough transform, however, was implemented on NON-VON [Shaw 82], a medium- to fine-grained SIMD/MSIMD parallel tree-machine, by Ibrahim et al. [Ibrahim 86]. The description applies to DADO as well, since it shares a lot in common with the NON-VON design. Furthermore, the account ignores NON-VON's leaf mesh connections which were introduced in later versions of the machine (DADO does not employ a mesh either.) Two approaches are described, a *direct approach* and an *MSIMD approach*. The time complexity of both algorithms is proportional to the number of edge pixels (an optimal result), although the constant is much smaller for the MSIMD approach.

5.2.3 Semantic Networks

No information is currently available regarding the usefulness of DADO as an architecture for semantic networks. Forrest [Forrest 85] developed a parallel KL-ONE-like knowledge representation language, modelled on the idea of production systems, that uses a fine-grained MIMD tree-based architecture. MIMD organization was preferred over SIMD for the following reasons: it is better suited for learning, facilitates almost decomposable search problems frequently encountered in AI, and is much more plausible from a cognitive point of view as a model for human thought. She has shown that, for a tree of depth D, the worst case time complexity for a parallel algorithm is $O(D)$ for subsumption and $O(D^3)$ for classification.

5.2.4 Comments on DADO

Synchronization is a significant problem in the DADO production system algorithms. The algorithms have *three distinct sequential steps* and, therefore, there are synchronization phases during which some processors are idle, waiting for others to finish. Although matches are processed in parallel, the selection phase cannot be completed until the last match is finished. It is possible, however, to *overlap* the update phase with the next matching phase, because each PM PE can start its matching phase as soon as it has done its local updating.

Speed-up in DADO (if any), will probably come from the matching phase. The optimization in the Rete algorithm means that possible speedup due to parallelism is only proportional to the size of the conflict set, which is relatively small. In addition, the designers of the OPS system claim that with special tag bits in hardware and custom microcode routines, their programs will run *two orders of magnitude faster* than they do on current machines. In the case of production systems it seems that *DADO is not economical* compared with available alternatives.

An important issue is OPS5 suitability as a paradigm for parallelism. R1 has been *fine-tuned* for execution on serial processors, namely OPS5. Thus the inherent parallelism in R1 may bear little resemblance to the inherent parallelism in the problem R1 solves. One possible solution is to add to OPS5 parallel constructs. A language called *HerbAl* is being developed for DADO that will explore this idea.

Being a medium-grained machine, DADO2 is highly suited to intermediate-level vision problems. (e.g., Hough Transform). The lack of a mesh connectivity on DADO, however, can be an impediment to the efficient execution of certain window-based image operations.

5.3 Performance of the Butterfly Machine

5.3.1 Intermediate-Level Vision

The Butterfly, being a coarse-grained machine, is most suited for high-level vision applications. It is, however, suitable for some intermediate-level image processing tasks such as performing the

Hough transform. The following is a description of the approach taken by Chandran and Davis [Chandran 86] for implementing the Hough transform on the Butterfly.

The Hough Transform on the Butterfly The Hough transform is computed in three phases:

1. *Phase I:* The input space is allocated to the different processors. If the number of processors is small, each processor computes the transform for a set of ϕ's. If the number of processors is large, each processor is assigned a single angle and a set of rows of the input image (overlapping or "scattered"), or a part of an input row if the number of processors is even greater. Each processor then independently computes a version of the Hough array based on the subset of the problem given to it. The time complexity of this phase is $O(e \times r/p)$, where e is the number of edge points, r is the resolution of the ϕ-dimension of the Hough space, and p is the number of processors. It is independent of the size of the image.

2. *Phase II:* The Hough arrays are collapsed into the complete Hough array using the standard technique [Preparata 81]. For an $n \times n$ input image, this phase has a time complexity of $\log(n)$ on the Butterfly (or hypercube). This performance is attained when the number of processors is $n^2/\log n$.

3. *Phase III:* Maxima detection can be done on a sequential machine in time proportional to the size of the Hough array (compute the median in linear time, then use it to select the k biggest elements of the array in linear time). On a Butterfly (or hypercube) with p processors, a speedup of p can be achieved for problems of size $k \times p \times \log p$ or greater (when computing the k largest maxima).

The Hough transform has been successfully implemented on a 16-processor Butterfly machine by Chandran and Davis [Chandran 86], achieving linear speedup. No data, however, is available at this time on the performance on larger configurations. Chandran and Davis have implemented the Hough transform on NCUBE corporation's NCUBE (a hypercube) with similar results. Hence, the following section applies to the Connection Machine too.

5.3.2 Semantic Networks

Symbolic Knowledge Representation The knowledge representation language described here is part of the Parallel Expert System Execution Environment currently under development at

BBN [Quayle 86]. The system is implemented in a parallel version of Common Lisp.[7] The Parallel Expert System Execution Environment supports object oriented programming (an extended version of CommonLoops), a hybrid rule-based system, and a knowledge representation language based on KL-ONE, KANDOR, and KEE.

The knowledge representation language is a hybrid in the spirit of KL-ONE [Brachman 85] (and its descendants), KRYPTON [Brachman, Fikes, Levesque 83], and CAKE [Rich 85] (see the previous section on traditional representation languages for a detailed description). The representation language offers many opportunities for parallelism. Classification and dependency maintenance are primarily breadth-first exploration of static data structures. These can be implemented in a way that is completely hidden from the user. The precise details of how the representation language will exploit parallelism on the Butterfly has not yet been determined. Therefore, no data is available at this time about the system's performance and scalability.

Connectionist Knowledge Representation Fanty [Fanty 86] has implemented (in parallel C) an interactive parallel connectionist simulator on a 120-processor Butterfly machine. A set of routines support sequential and parallel building of networks. Efficient data structuring of network units allows the simulation of networks with as many as 100,000 units and 3,000,000 links (It takes about 3 hours to sequentially build the network, and about one minute in parallel.) Linear speedup during simulation runs was effectively achieved; for 90 processors a speedup factor of about 70 is reported.

5.3.3 Rule Based Expert Systems

As mentioned in the previous section, the Parallel Expert System Execution Environment on the Butterfly supports a rule-based system. The rule system is composed of four main parts: a processor, a context, an access interface, and a development interface. The processor is a set of programs responsible for executing the rules in a rule set (this is akin to the rule interpreter in traditional systems). The context is the Working Memory of the system. The access interface is similar in function to the Rete algorithm. The development interface supports acquisition, editing, and compilation of rules.

Two rule systems are being developed:

[7]The Butterfly uses a Lisp Machine as a front-end host. The Lisp Machine is capable of running in stand-alone mode when a multiprocessor (Butterfly) machine is unavailable.

1. The Routine Knowledge Rule System: Exploits implicit parallelism through the underlying Lisp implementation (similar in spirit to the LOOPS rule system. Its is less powerful than OPS5, which is a Turing machine-equivalent programming language).
2. The Hybrid Pattern-Matching Rule System: Exploits explicit parallelism via concurrent pattern matching, rule scheduling, and rule execution.

Performance data at the time of this writing is currently available only for the first system [Boulanger 86]. Related rules in the Routine Knowledge Rule System were organized into *rule packets*. It should be noted that that rules in the system were *totally independent of each other* (no serial dependencies between rule packets). Also, within a single rule packet, variables in the LHS of rules were disjoint from variables in the RHS of rules (Boulanger claims that this is realistic). Furthermore, the application data sets had the characteristic that very few rules actually fired for any single data set (about 10 out of 220). The system was developed on a 16-processor Butterfly, so scalability is an open question (according to Boulanger, the goal of the project was not to achieve maximum speedup, but to investigate performance bottlenecks).

The results of simulation indicate that the effective parallelism in the rule system is very low compared to the number of productions in the system. A speedup factor of about 8 was realized. The group suggests time slicing the individual processors to improve this result (especially in the case where tasks vary significantly in size).

5.3.4 Comments on the Butterfly

The Uniform System approach on the Butterfly proves to be rather problematic. To reduce *contention*, the Uniform System encourages a globally-shared memory **and** the *scattering* of data uniformly on the machine. This means that while contention for a particular memory is reduced, overall switch contention is increased because practically all data references are remote. Even when data physically resides in local memory, all data conceptually resides in shared memory and must be copied into the local workspace.

Parallelism in the Butterfly also suffers because of the overhead involved in generating the tasks. Unfortunately, in spite of the existence of a scheduler, it is not possible to generate more active processes than processors in the Uniform System. This has imposed a severe limitation on applications. Furthermore, under the Uniform System it is not possible for processors to act "independently" since the same code is loaded on all the processors (enforcing SPMD processing rather than MIMD). Hopefully, these problems should be remedied in the near future.

Leblanc [LeBlanc 86] is developing a message passing environment for the Butterfly which seems to remedy many of the shortcomings of the Uniform System approach. Process synchronization occurs only during access to message buffers, which is implemented very efficiently (micro-coded). The user does not need to be concerned with the low-level synchronization; it is implicit in the message-passing primitives. This is in sharp contrast with the Uniform System which offers no high-level synchronization primitives (each programmer must use low-level primitives to implement the appropriate synchronization).

5.4 Compare-and-Contrast

As expected, none of the three machines surveyed in this paper is suited to all the AI tasks presented above. The module granularity, concurrency control, and communications network employed by each of these computers is tailored towards the efficient execution of a certain class of algorithms.

The Connection Machine excels at traversing huge semantic networks and processing low-level image processing at blinding speeds. It is ill-suited for parallel execution of production systems, due to the coarse-grained nature of this paradigm. It may be useful for certain intermediate-level vision tasks (such as the Hough Transform) which can be finely decomposed.

The Butterfly is a general-purpose machine, but this comes at a high cost: the number of processors is quite small (at most 256 in the current version). Fine-grain problems can be hard to fit onto it. Its globally shared memory can easily become a bottleneck if care is not taken to spread data appropriately. The Butterfly is well suited for intermediate- and high-level image processing and for the execution of production systems.

DADO has great potential, due to its MSIMD flexibility, but is severely limited by its tree-structured communication network. Furthermore, it lacks mesh connections, a fact that hampers its performance in certain intermediate-level image processing. It is well suited, however, to intermediate-level image processing. Its usefulness for production system execution is still an open issue.

6 Architectures for the Selected Domains

The discussion in this section complements the investigation in the previous section. Whereas the later tried to analyze the suitability of three particular computer architectures for three AI domains, this section takes a look at each of the domains and attempts to come up with the architectural requirements for an efficient execution of these problems. A word of caution is due here. There is a lot of literature discussing the suitability of particular architectures for certain AI problems. Virtually nothing is available on the dual approach. Due to time constraints, the following discussion lacks in depth and formality. It is provided as an attempt to identify the important issues that come up when one attempts to design special purpose parallel computers for these domains.

The approach taken is as follows. First, for each AI domain, the fundamental computations that are at the heart of the problem are enumerated. Next, matching module granularity, concurrency control and communication geometry are sought. Finally, special architectural requirements for each domain are explicated.

6.1 Knowledge Representation: Semantic Networks

6.1.1 Fundamental Computations

Some fundamental computational abilities that any truly intelligent system must have (focusing on tasks that have to do with recognition and search in a very large space of stored descriptions) are:

1. *Set Intersection:* Recognition can be viewed as the process of finding, in a very large set of stored descriptions, the ones that best match a set of observed features. In its simplest form, this can be viewed as a set-intersection problem. The set-intersection operation comes up very frequently in AI knowledge-base systems. On a serial machine set-intersection takes time proportional to the size of the smallest of the sets being intersected, but frequently all of the sets are quite large. In a parallel marker-passing system, such set intersections are done in a single operation, once the members of each set have been marked with a different marker (which takes a constant time).

2. *Transitive Closure:* In knowledge-base systems it is frequently necessary to compute the closure of various transitive relations. The "is-a" relation is the most important of these transitive relations in most data bases, but other closure relations such as "part-of", "bigger

than", "later in time", etc., might need to be computed. In a serial machine, the computation of the transitive closure requires time proportional to the size of the answer set. In a parallel marker-passing system, it takes time proportional to the length of the longest chain of relations that has to be followed. For a short bushy tree (characteristic of most knowledge bases) the marker-passing system can be much faster than a serial machine algorithm.

3. *Contexts and Partitions:* In the presence of multiple overlapping partitions in a knowledge base, a serial machine must check each assertion for membership in one of the active partitions before that assertion can be used. Marker-passing systems handle this easily: mark the tree of active contexts using the transitive closure machinery, then propagate the mark to all assertions associated with these contexts, thereby activating them.

4. *Best-Match Recognition:* The set intersection computation (described above) is sufficient if the features are discrete and noise-free. Marker-passing systems are very poor at handling imperfect matches. Value-passing systems are ideal for this: they simply select the element whose activation level is the highest.

6.1.2 Architectural Issues

Although human experts may employ relatively small pools of specialized knowledge, they all seem to depend on vast amounts of what has been termed "common sense" knowledge. To represent even a fraction of this knowledge, one needs to construct large knowledge bases. This calls for massively parallel architectures, that is, fine-grained machines. As demonstrated by any dictionary, concepts are expressed in terms of many other concepts. This results in a highly connected, nonuniform graph. Since queries to a semantic network usually involve searching a large portion of the knowledge stored, one needs a fairly dense communications geometry to support the high bandwidth requirements involved. Hypercube variations (i.e., the cube-connected cycles) seem to be a good compromise between performance requirements (delay and contention) and cost (node degree and number of links).

As indicated above, a value-passing architecture seems ideal for the task at hand. This architecture is best implemented using distributed control mechanisms as indicated by Fahlman's work on marker propagation [Fahlman 82], Woods' powerful extension of marker propagation [Woods 78], Shastri's evidential reasoning massively parallel representation language [Shastri 85a] and Bic's work on dataflow architectures [Bic 85]. The MIMD paradigm seems most appropriate, in contrast with the approach taken by the designers of the Connection Machine. SIMD control has been dictated mainly by economical constraints and is still feasible for the number of processors employed. However, as the number of processors goes well into the millions, MIMD control will be the only viable control strategy. MIMD control also better supports semantic networks with many different types of nodes and links, since all types could perform different computations simultaneously. Moreover, one could simultaneously categorize many concepts by appropriately tagging each message with the name of the concept that generated it [Bic 85].

6.1.3 Special Requirements

A special problem encountered in semantic networks, involves the process of knowledge acquisition. When new concepts are added to a knowledge base, it is often necessary to add many new links and even change existing links. In the Connection Machine this is accomplished by establishing virtual connections between processors. Since physical connections are fixed, virtual communication paths can be quite long. Knight has pointed out that data objects are free to move from cell to cell, as long as they inform their acquaintances (the objects they are virtually connected to) where they are moving. This allows communicating processors to get closer to each other, thus improving communications and reducing network contention. Objects can force a swap even if it is to a less-used object's disadvantage. This would allow implementation of a virtual network, analogous to virtual memories on a conventional computer. Little-used objects would gradually be pushed away from the center of activity and eventually fall off into a secondary storage device.

A better solution involves the use of dynamically reconfigurable computers. Smitley, Lee, and Goldwasser [Smitley 85] have proposed a novel approach for dynamically reconfiguring a network to match an algorithm. Their system, called PION (Processors Interconnected with an Optical Network), is built around an optical (laser-based) communication network.[8] The network supports high-speed, non-blocking, complete interconnection communication between N heterogeneous processors. It is expandable at the reasonable cost of $O(N)$. The optical network can be configured to match the topology of the algorithm (prior to execution). Smitley has been concentrating on trying to devise strategies for determining the optimum processor-to-processor network topology for a given algorithm [Smitley 85]. Finding the optimum solution has been shown to be an *NP-complete* task, so Smitley has developed a heuristic algorithm that finds suboptimal solutions. The algorithm's best, worst, and average case performance has been analyzed; it has been shown that

[8]Others have come up with similar proposals [Tenenbaum 83] [Bell 86]

the algorithm almost always finds the optimal mapping.

6.2 Expert Systems: Production Systems

6.2.1 Fundamental Computations

A convenient way to describe the primitive operations in a production system algorithm is to make an analogy to more familiar relational database terminology [Codd 72] [Date 82] [Hillyer 84]. If the WM elements of a production system are considered to be tuples of some universal relationship in a relational database, then it becomes apparent that the LHS of a rule in a production system is analogous to a query in a relational database language. Figure 8 can be represented as the database query:

$$Join(Join(Select(A), Select(B)), Select(C))$$

The constants in a single-condition element may be viewed as a relational selection over a database of working memory. We say a working memory element *partially matches* a condition element if it satisfies the select operators in the element's pattern constraints (referred to as *intra-condition* testing). Consistent bindings of pattern variables between distinct condition elements (called *inter-condition* testing) may be regarded as a database equijoin operation on the relations formed by the selections. A collection of rules may be viewed as a collection of concurrent database queries; the conflict set as the union of the query results of each of the rules in the system. A primary optimization used when executing database queries is to perform the selects before the joins. The input layer of a RETE network contains chains of tests that perform the selection operations. Those elements that satisfy these tests (by partially matching a particular condition element) are represented by a token stored in an α-memory (deletions to WM actually remove tokens), thus forming the memory support part of the algorithm.

6.2.2 Architectural Issues

The problem of selecting an optimal grain-size for a parallel production systems accelerator is a tradeoff between generality, cost, and hardware utilization. If one increases the amount of memory, the number of distinct PEs decreases (for a fixed size machine) thus reducing the potential parallel execution of code and driving up the cost of the PEs. However, decreasing the memory size affects the size and the resultant complexity of code that may operate at an individual PE, thus restricting the scope of applicability of the architecture. Furthermore, if the inherent parallelism in a problem is not as great as the number of available PEs, the additional PEs will be underutilized [Stolfo 86]. Recent statistics reported for R1 indicate that of a total of 200 rules (and several hundred WM elements), on the average, only 30–35 rules need to be matched against WM on each cycle of operation [Gupta 84b]). This number seems to be independent of the size of the PM (Production Memory). Thus, even if 200 fine-grain PEs were available to process rules, only 30–35 PEs would perform useful work on each cycle of execution.

A tree communication geometry seems to be general enough for a production systems machine. However, since there is limited parallelism inherent in production systems and hence only a small number of processors need to be employed, one could even use a crossbar. The upshot is that the geometry is not a critical issue. We now focus our attention on the issue of concurrency control. MSIMD organization, as found in DADO, seems ideal. SIMD-mode is sufficient for the select and act phases, but MIMD-mode is preferable for the match phase. Actually, full MIMD capability is not necessary. One could do quite nicely with only SPMD-mode, that is, loading each PE with a match routine and simultaneously instructing all PE's to branch to this routine.

6.2.3 Special Requirements

Tree structures have the nice property of supporting logarithmic-time broadcast, search, and fan-in. Binary trees have a simple scheme for VLSI layout (an H-shape based fractal). Unlike array structures, the connections of a tree structure are not uniform. The (on-chip) distance between two connecting processors increases as they move up the root (this can be corrected on the chip, by adjusting delays with appropriate drivers). Off-chip communications can be conducted at the root without serious delay. Processors at high levels of the tree (close to the root) may become bottlenecks if the majority of communications are not confined to processors at lower levels.

As shown above, one can view production system execution as queries to a database. It is easy to see that additional speedup on a tree-structured parallel machine can be obtained by pipelining the queries. This calls for special mechanisms in hardware to support pipelining in both directions, that is, both up and down the tree links. A speedup of $\log N$ can be obtained for N processors. The cost is linear in the number of processors, since this involves changes to individual processors, but does not require additional links. It is important to point out here that this speedup is obtained only if flushing the pipeline occurs infrequently.

6.3 Vision (especially Intermediate-level)

6.3.1 Fundamental Computations

The computer vision problem can be described as involving three levels of processing, namely low-, intermediate- and high-level processing. Each level is characterized by different data representations and different computations performed on them. Furthermore, processing at each level is influenced by computations at the levels below and/or above it. The following is a brief exposition of these issues.

1. **Low-Level Processing:** this level consists mainly of operations on pixels and *local neighborhoods* of pixels and is characterized by an output matrix that is usually similar in size to the input. Processing involves algorithms for restoration, noise removal, geometric correction, segmentation, simple feature extraction (such as edge detection), feature enhancement, etc. Most of these algorithms are numeric in nature. The result of this level is a transformed image with labeled regions and line segments. No matchings or inferences on objects are performed at this stage.

2. **Intermediate-Level Processing:** this level provides an interface between the low- and high-levels of representation. Communication between the intermediate-level and these levels is by no means unidirectional. In most cases, recognition of an object (or part of a scene) at the high level will establish a strategy for further processing and probing the low and intermediate levels. Typical activities in this level include feature extraction for regions, lines and vertices (bottom-up activities) as well as the relations between these entities, and grouping, splitting, and labeling processes to more naturally match stored object descriptions (top-down activities). The results of this processing are two-dimensional representations of image entities such as regions, line segments and vertices.

3. **High-Level Processing:** This level controls the intermediate level of processing where the symbolic two-dimensional representations of the intermediate level must be related to object descriptions stored in a knowledge base. Typical activities involve classifying segments or features of the image into known classes which may involve combining a set of segments or features to create a total composite object. The techniques involved here are usually termed pattern recognition or AI. The result of this level is a symbolic representation of the content of a specific image in terms of the general stored knowledge of the object classes and the physical environment.

The key to vision processing is a flow of communication and control both up and down through all representation levels. In the **upward** direction communication consists of segmentation results from multiple algorithms, sets of attributes for each extracted image feature, statistical information, and the passing of actual symbols. In the **downward** direction communications inherently are commands for selecting subsets of the image, specifying further processing in particular portions of the image, and requests for additional information in terms of the intermediate representation.

6.3.2 Architectural issues

Low-level vision tasks are well suited to **SIMD** computer structures. High performance is achieved when each processor is assigned a pixel, processors are capable of bit-parallel operations, and chip interconnections are very short (communication is mostly near-neighbor, so a mesh interconnection scheme is ideal). MIMD processors are not very efficiently utilized for low-level image processing since much of the hardware is devoted to individual control units and reliable asynchronous data communication between processors. Furthermore, there is also a problem with sharing near-neighbor data.

High-level vision problems (such as classification algorithms) frequently involve a set of sequential searches for pattern matching which may be conducted independently in parallel. Such tasks, which may involve many independent operations on a common data base, are well suited to an **MIMD** parallel computer. For example, one can assign a set of processors to analyze a set of segmented objects (where each processor deals with a single object) or several processors may concurrently analyze a single object (each performing a different analysis algorithm).

For intermediate-level vision, a hybrid **MSIMD** architecture is better suited than either of the extremes (SIMD and MIMD). This scheme enables an independent group of SIMD processors to be assigned to a task. A major concern in the design of an MSIMD computer is to ensure that the worst features of both systems, the expense and inefficiency of MIMD data communication and control and the inflexibility of homogeneous SIMD structures, are not both present in the combination.

There are two possible cases where MSIMD systems might have an advantage over SIMD and MIMD systems:

1. When the ratio of low-level image processing to image analysis is very unpredictable and has great variability

2. When the SIMD low-level processing requires the processing of many relatively small sub-images and each sub-image requires a different algorithm

A mesh-based interconnection network is needed to support the great number of local neighborhood operations at the pixel (low) level. However, for certain intermediate-level vision problems a tree structured interconnection network is more suitable. A good compromise would be to have both, as the additional wiring demands for the tree connections are comparable to those needed for a mesh (the number of links per processor are four for a mesh and three for a tree). A modest amount of memory associated with each processor should be sufficient to support the data (records holding intermediate-level information about the low-level pixels) and image processing routines.

6.3.3 Special Requirements

A vision machine must be able to load (and possibly dump) a complete image very quickly (less than fraction of a second for real-time applications). Very high data transfer rates are involved here (hundreds of Megabits per second). The need to dump an image out for evaluation by a sequential program must be avoided at all costs. Most important, a general vision machine should provide efficient and fast mechanisms for communicating information and control both up and down through the three levels of representation. PASM, a partitionable SIMD/MIMD vision system developed at Purdue University [Siegel 81], is an interesting architecture that can deal with all three levels. The machine can be dynamically reconfigured as one or more SIMD and/or MIMD machines to optimize a wide range of image processing and pattern recognition. The SIMD mode efficiently supports "local" operations such as matrix arithmetic while the MIMD mode is optimized for "global" pattern matching. A multistage communication network (e.g., hypercube or k-cube) is used to connect the processors. A k-cube can be easily partitioned, as it is constructed recursively by joining together two $(k - 1)$-cubes. The system employs parallel intelligent secondary storage devices. This, in combination with double-buffering in the microcontrollers (which provide MIMD control), supports fast loading and unloading of the memory modules (which are totally distributed among the processors).

7 Conclusions

7.1 The Communication Network

Most parallel architectures are distinguished from each other by their method of inter-processor communication. Two important factors in any computer system are *communication bandwidth* and *memory contention.* Communication bandwidth is a measure of the amount of information (bytes) one can move from one location to another (via a channel) in a given amount of time. Memory contention indicates the loss in parallelism that occurs when data that could be accessed in parallel is necessarily accessed in serial. The *effective bandwidth* of a system, arguably a good measure of performance, should take both communication bandwidth and memory contention into account [Tenenbaum 87].

For certain computer applications, massively parallel processors will be more effective than uniprocessors (or coarse-grained multiprocessors), since they represent a better tradeoff between communication bandwidth and memory contention. To better illustrate this point, I shall contrast the Connection Machine (a special-purpose machine) and the Butterfly (a general-purpose computer).

The communication bandwidth between processors on the Connection Machine (typically 128 Kbps) is more than two orders of magnitude slower than the on the Butterfly (32 Mbps). However, in the Connection Machine processors that are physically close can share information almost directly. In the case of the Butterfly, all inter-processor communication must travel through the entire switching network (whose size is a function of the total number of processors)[9]. Memory contention, therefore, should be much worse on the Butterfly than on the Connection Machine.

As a result, although the maximal size of memory in the Connection Machine (256 Mbits) is more than an order of magnitude smaller than that of a 256-processor Butterfly (8000 Mbits), the Connection Machine has has an effective processing power (6000 MIPS) that is almost two orders of magnitudes faster than that of the Butterfly (128 MIPS). Therefore, for massively parallel applications (such as large semantic networks) where memory contention dominates link bandwidth, the Connection Machine's architecture is more cost-effective than the Butterfly's.

In tree structured networks, processor to processor communication can be very poor because all messages that are sent from one side of the tree to the other must pass through the root node. The algorithms designed for DADO, therefore, avoid this type of communication by allowing duplicated

[9]This is a general distinction between special-purpose and general-purpose machines.

data to insure locality of reference within the tree (not very economical). By comparison, the Butterfly and the Connection Machine have much better communication schemes.

The main cost in the Butterfly and the Connection Machine is the communication network. The binary tree topology is favored because it can be efficiently implemented in VLSI technology and can easily handle broadcasts from the root to a large number of recipients (the binary tree bottleneck is avoided in the execution of production systems, since each node talks only with its ancestors). The binary tree topology also supports algebraically commutative and associative operations (e.g., addition) in time logarithmic in the number of processors. The Butterfly machine is especially attractive for solving large-grain problems that demand general interconnects among the processors. The Connection Machine also supports general connect, but is more tuned towards fine-grain problems.

7.2 Shared Memory versus Message Passing

Since processors are still much faster than (bulk) memories and sharing data between multiple processors must be done with special communication channels, MIMD machines with a shared memory are bad as a paradigm for AI [Deering 84]. Some sort of special message passing (and forwarding) is absolutely essential for efficient handling of the traffic. The particular model of computation in use is less important than how well it is matched to the application. Shared memory is useful in applications that communicate values that must be interpreted in the context of an environment (e.g., pointers). Message-passing is a better model for value-oriented computing. Any programming environment that offers a single model of communication will not be well-matched to a large class of applications. This lesson has been learned during the development of software for the Butterfly. Initially, the operating system supported only the shared memory model. Recently a substantial effort has been undertaken to provide the butterfly with a comprehensive message passing environment. Initial benchmarks confirm that for certain applications and range of processor nodes, the message passing approach outperforms the Uniform System [LeBlanc 86].

References

[BBN 86] BBN. *The Butterfly^{TM} Parallel Processor Overview.* Tech Report 6148 (Version 1), Bolt Beranek and Newman Inc., Cambridge, Mass., March 1986.

[Bell 86] Bell, T. E. Optical Computing: A Field in Flux. *IEEE Spectrum* 23(8):34–57, August 1986.

[Bic 85] Bic, L. Processing of Semantic Networks on Dataflow Architectures. *Artificial Intelligence* 27:219–227, 1985.

[Boulanger 86] Boulanger, A. *Parallelism in the Execution of a Routine Knowledge Rule System on the Butterfly^{TM} Computer.* Report 6436, BBN Laboratories Inc., Cambridge, Mass. 02238, December 1986.

[Brachman 84] Brachman, R. J. and Levesque, H. J. The Tractability of Subsumption in Frame-Based Description Languages. *Proceedings of the Fourth National Conference on Artificial Intelligence (AAAI-84)* , 1984.

[Brachman 85] Brachman, R. J. and Schmolze, J. G. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science* 9:171–216, 1985.

[Browning 80] Browning, S. A. A Tree Machine. *Lambda Magazine* 1:31–36, 1980.

[Chandran 86] Chandran, S. and Davis, L. S. *The Hough Transform on the Butterfly and the NCube.* CS-TR 1713, University of Maryland, College Park, Maryland 20742, September 1986.

[Codd 72] Codd, E. F. *Database Systems*, Chapter on Relational Completeness of Data Base Sublanguages. Volume 6 of *Courant Computer Science Symp. Series.* Prentice Hall, London, 1972.

[Date 82] Date, C. J. *An Introduction to Database Systems.* Volume 1 of *The Systems Programming Series.* Addison Wesley, 3rd edition, 1982.

[Deering 84] Deering, M. F. Hardware and Software Architectures for Efficient AI. In *Proceedings of the Fourth National Conference on Artificial Intelligence (AAAI-84)*, pages 73–78. Morgan Kaufmann Publishers, Inc., 95 First Street, Los Altos, CA 94022, 1984.

[Duda 72] Duda, R. O. and Hart, P. E. Use of the Hough Transformation to Detect Lines and Curves in Pictures. *Communications of the ACM* 15(1), 1972.

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

[Fahlman 83] Fahlman, S. E., Hinton, G. E., and Sejnowski, T. J. Massively Parallel Architectures for AI: NETL, THISTLE and Boltzmann Machines. In *Proceedings of the Third National Conference on Artificial Intelligence (AAAI-83)*, pages 109–113. Washington D. C., August 1983.

[Fahlman 87] Fahlman, S. E. and Hinton, G. E. Connectionist Architectures for Artificial Intelligence. *IEEE Computer* 20(1):100–109, January 1987.

[Fanty 86] Fanty, M. *A Connectionist Simulator for the BBN Butterfly Multiprocessor.* Butterfly Tech Report 2, Computer Science Department, University of Rochester, Rochester, NY 14627, January 1986.

[Feldman 82] Feldman, J. and Ballard, D. Connectionist Models and Their Properties. *Cognitive Science* 6:205–254, 1982.

[Flynn 72] Flynn, M. J. Some Computer Organizations and their Effectiveness. *IEEE Transactions on Computers* C-21, September 1972.

[Forgy 79] Forgy, C. L. *On the Efficient Implementation of Production Systems.* PhD thesis, Carnegie-Mellon University, 1979.

[Forgy 82] Forgy, C. L. Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Matching Problem. *Artificial Intelligence* 19:17–37, 1982.

[Forrest 85] Forrest, S. *A Study of Parallelism in the Classifier System and its Application to Classification in KL-ONE Semantic Networks.* PhD thesis, Computer and Communications Science Dept., University of Michigan, Ann Arbor, Michigan, 1985.

[Frenkel 86a] Frenkel, K. A. Complexity and Parallel Processing: An Interview with Richard Karp. *Communications of the ACM* 29(2):112–117, February 1986.

[Frenkel 86b] Frenkel, K. A. Evaluating Two Massively Parallel Machines. *Communications of the ACM* 29(8):752–759, August 1986.

[Gupta 84a] Gupta, A. Implementing OPS5 Production Systems on DADO. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 83–91. August 1984.

[Gupta 84b] Gupta, A. *Parallelism in Production Systems: The Sources and the Expected Speed-up.* Technical Report CMU-CS-84-169, Carnegie-Mellon University, December 1984.

[Hillis 84] Hillis, D. The Connection Machine: A Computer Architecture Based on Cellular Automata. *Physica* 10D:213–228, 1984.

[Hillis 85] Hillis, D. *The Connection Machine. The MIT Press Series in Artificial Intelligence.* The MIT Press, Cambridge, Mass., 1985.

[Hillis 87] Hillis, D. The Connection Machine. *Scientific American* 256(6):108–115, June 1987.

[Hillyer 84] Hillyer, B. K. and Shaw, D. E. *Execution of OPS5 Production Systems on A Massively Parallel Machine.* Technical Report CUCS-147-83, Department of Computer Science, Columbia University, New York City, NY 10027, September 1984.

[Hough 62] Hough, P. V. C. Methods and Means for Recognizing Complex Patterns. 1962.

[Hwang 84] Hwang, K. and Briggs, F. A. *Computer Architecture and Parallel Processing,* Chapter 1, 5 and 7, pages 1–50, 325–388, 459–552. *Computer Organization and Architecture.* McGraw-Hill, 1984.

[Hwang 87] Hwang, K., Ghosh, J., and Chowkwanyun, R. Computer Architectures for Artificial Intelligence Processing. *IEEE Computer* 20(1):19–27, January 1987.

[Ibrahim 86] Ibrahim, H. A. H., Kender, J. R., and Shaw, D. E. On the Application of Massively Parallel SIMD Tree Machines to Certain Intermediate-Level Vision Tasks. *Computer Vision, Graphics, and Image Processing* 36:53–75, 1986.

[Kung 80] Kung, H. T. *Advances in COMPUTERS,* Chapter 2: The Structure of Parallel Algorithms, pages 65–112. Volume 19. Academic Press, N.Y., 1980.

[LeBlanc 86] LeBlanc, T. J. *Shared Memory Versus Message-Passing in a Tightly Coupled Multiprocessor: A Case Study.* Butterfly Tech Report 3, Computer Science Department, University of Rochester, Rochester, NY 14627, January 1986.

[Lim 86] Lim, H. S. and Binford, T. O. *Survey of Parallel Computers.* Technical Report, AI Lab, Computer Science Department, Stanford University, Stanford, CA 94305, 1986.

[McDermott 82] McDermott, J. R1: A Rule Based Configurer of Computer Systems. *Artificial Intelligence* 19(1):39–88, September 1982.

[Miranker 87] Miranker, D. P. TREAT: A Better Match Algorithm for AI Production Systems. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87).* Seattle, Washington, 1987.

[Moser 83] Moser, M. G. *An Overview of NIKL, The New Implementation of KL-ONE.* Technical Report 5421, Bolt Beranek and Newman, Inc., 1983.

[Nilsson 80] Nilsson, N. J. *Principles of Artificial Intelligence.* Tioga, Palo Alto, Calif., 1980.

[Preparata 81] Preparata, F. P. and Vuillemin, J. The Cube-Connected Cycles: A Versatile Network for Parallel Computation. *Communications of the ACM* 24(5):300–309, May 1981.

[Quayle 86] Quayle, C., Boulanger, A., Clarke, D., Thome, M., Vilain, M., and Anderson, K. *Parallel Expert Systems Execution Environment: A Functional Specification.* Report 6225, BBN Laboratories Inc., Cambridge, Mass. 02238, August 1986.

[Quillian 68] Quillian, M. *Semantic Information Processing*, pages 227–270. MIT Press, Cambridge, Mass., 1968.

[Quinlan 79] Quinlan, J. R. *Induction Over Large Data Bases.* Technical Report HPP-79-14, Compuer Science Department, Stanford University, Stanford, California, 1979.

[Reeves 81] Reeves, A. P. Parallel Computer Architectures for Image Processing. In *Proceedings of the 1981 International Conference on Parallel Processing*, pages 199–206. 1981.

[Rumelhart 86] Rumelhart, D. E., McClelland, J. L., and the PDP research group (editors). *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Volume 1. MIT Press, Cambridge, Mass., 1986.

[Schwartz 80] Schwartz, J. T. Ultracomputers. *ACM Transactions on Programming Languages and Systems* 2(4):484–521, October 1980.

[Schwartz 83] Schwartz, J. T. *A Taxonomic Table of Parallel Computers, Based on 55 Designs.* Technical Report, Courant Institute, NYU, 1983.

[Sejnowski 86] Sejnowski, T. J. and Rosenberg, C. R. *NETtalk: A Parallel Network that Learns to Read Aloud.* Tech. Report JHU/EECS-86-01, The John Hopkins University, Baltimore, MD 21218, 1986.

[Shastri 85a] Shastri, L. *Evidential Reasoning in Semantic Networks: A formal Theory and its Parallel Implementation.* TR 166, Computer Science Department, University of Rochester, September 1985.

[Shastri 85b] Shastri, L. and Feldman, J. A. Evidential Reasoning in Semantic Network: A Formal Theory. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 465–474. Los Angeles, CA, August 1985.

[Shastri 86] Shastri, L. *Massive Parallelism in Artificial Intelligence.* MS-CIS 86-77 (LINC Lab 43), University of Pennsylvania, Philadelphia, PA, December 1986.

[Shaw 82] Shaw, D. E. *The NON-VON Supercomputer.* Technical Report, Department of Computer Science, Columbia University, New York City, NY 10027, August 1982.

[Siegel 81] Siegel, H. J., Seigel, L. J., Kemmerer, Mueller, F. C., Smalley, Jr., H. E., and Smith, S. D. PASM: A Partitionable SIMD/MIMD System for Image Processing and Pattern Recognition. *IEEE Transactions on Computers* C-30(12), December 1981.

[Siegel 85] Siegel, H. J. *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies*, pages 1–32. Lexington Books, 1985.

[Smitley 85] Smitley, D., Goldwasser, S. M., and Lee, I. *IPON - Advanced Architectural Framework for Image Processing.* Tech. Report MS-CIS-85-13, University of Pennsylvania, Philadelphia, PA 19104, 1985.

[Stanfill 86] Stanfill, C. and Waltz, D. Toward Memory-Based Reasoning. *Communications of the ACM* 29(12):1213–1228, December 1986.

[Stolfo 82] Stolfo, S. J. and Vesonder, G. *ACE: An Expert System Supporting Analysis and Management Decision Making.* Technical Report, Department of Computer Science, Columbia University, 1982.

[Stolfo 86] Stolfo, S. J. and Miranker, D. P. DADO: A Tree-Structured Architecture for Artificial Intelligence. *Annual Reviews of Computer Science* 1:1–18, 1986.

[Stolfo 87] Stolfo, S. J. Initial Performance of the DADO2 Prototype. *IEEE Computer* 20(1):75–83, January 1987.

[Stone 80] Stone, H. S. *Introduction to Computer Architecture*, Chapter 8 (Parallel Computers), pages 363–423. SRA, Chicago, IL, 2nd edition, 1980.

[Tenenbaum 83] Tenenbaum, E. *A Comparison of Parallel Computer Architectures for AI Applications.* Term Paper, MIT, 1983.

[Tenenbaum 87] Tenenbaum, E. Personal communication, July 1987.

[Vilain 85] Vilain, M. The Restricted Language Architecture of a Hybrid Representation System. In 9th *Int. Joint Conf. on Artificial Intelligence*, pages 547–551. William Kaufmann, Inc., Los Altos, California, 1985.

[Waltz 87] Waltz, D. L. Applications of the Connection Machine. *IEEE Computer* 20(1):85–97, January 1987.

[Weems 84] Weems, C., Levitan, S., Foster, C., Riseman, E., Lawton, D., and Hanson, A. Development and Construction of a Content Addressable Array Parallel Processor for Knowledge-Based Image Interpretation. In *Proceedings of the Workshop on Algorithm-Guided Parallel Architectures for Automatic Target Recognition.* Xerox international Center, Leesburg, VA, July 1984.

[Wiley 87] Wiley, P. A Parallel Architecture Comes of Age at Last. *IEEE Spectrum* 24(6):46–50, June 1987.

[Winston 77] Winston, P. H. *Artificial Intelligence.* Addison Wesley, Reading, Mass., 1977.

[Woods 78] Woods, W. A. *Research in Natural Language Understanding.* Technical Report 2, BBN, Cambridge, Mass., 1978.