NASA Technical Memorandum 103886

Parallel Processing and Expert Systems

Sonie Lau and Jerry C. Yan

May 1991

National Aeronautics and Space Administration

NASA Technical Memorandum 103886

Parallel Processing and Expert Systems

Sonie Lau, Ames Research Center, Moffett Field, California
Jerry C. Yan, Sterling Federal Systems, Ames Research Center, Moffett Field, California

May 1991

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

SYMBOLS

AAP	advanced architecture project
AI	artificial intelligence
cdr	[no definition] contents of the decrement register
CGE	conditional graph expression
CM	connection machine
FGCS	first generation computer system
FGHC	flat guarded horn clauses
ICOT	Institute for New Generation Computer Technology
KL1	Kernel language version 1
LHS	left hand side
LIPS	logical inferences per second
MIMD	multiple instruction multiple data
MITI	Ministry of International Trade and Industry
MPC	message passing computers
MRB	multiple reference bit
PE	processing element
PIM	parallel inference machine
PSI	personal sequential inference
RAP	restricted and parallelism
RHS	right hand side
RPS	reductions per second

SIMD	single instruction multiple data
WAM	Warren abstract machine
TREAT	TREe associative temporal
WEC	weighted export count
WME	working memory element
WTC	weighted throw count

PARALLEL PROCESSING AND EXPERT SYSTEMS

Sonie Lau and Jerry C. Yan*

Ames Research Center

SUMMARY

Whether it be monitoring the thermal subsystem of Space Station Freedom, or controlling the navigation of the autonomous rover on Mars, NASA missions in the 1990s cannot enjoy an increased level of autonomy without the efficient implementation of expert systems. Merely increasing the computational speed of uniprocessors may not be able to guarantee that real-time demands are met for larger systems. Speedup via parallel processing must be pursued alongside the optimization of sequential implementations. Prototypes of parallel expert systems have been built at universities and industrial laboratories in the U.S. and Japan. This paper surveys the state-of-the-art research in progress related to parallel execution of expert systems. The survey discusses multiprocessors for expert systems, parallel languages for symbolic computations, and mapping expert systems to multiprocessors. Results to date indicate that the parallelism achieved for these systems is small. The main reasons are (1) the body of knowledge applicable in any given situation and amount of computation executed by each rule firing are small, (2) dividing the problem solving process into relatively independent partitions is difficult, and (3) implementation decisions that enable expert systems to be incrementally refined hamper compile-time optimization. In order to obtain greater speedups, *data parallelism* and *application parallelism* must be exploited.

1. INTRODUCTION

The science and engineering objectives of future NASA missions cannot be met without an increased level of autonomy for both onboard and ground-based systems. For example, with *Mars Rover Sample Return*, significant amounts of information must be digested in real time to decide where to collect samples, how to analyze and which samples to return. The long delays associated with signal transmission between Mars and Earth require the Mars Rover to make intelligent decisions and operate autonomously. These scenarios demands the design and implementation of complex real time expert systems able to carry out a variety of tasks such as: stereogrammetic mapping of Mars from orbit, safe landing to a predetermined site, navigation about unknown terrain, site selection, sample acquisition/analysis, and docking control functions.

Space Station Freedom is expected to remain operational for many years. The onboard computer systems (consists of many interacting, physically-distributed intelligent subsystems) must be

*Sterling Federal Systems Inc., Palo Alto, CA.

coordinated smoothly, utilized effectively and remain continuously operational. Whether it be automating the operation of the thermal and power subsystems, or flight telerobotic servicers, the day-to-day operation of Space Station Freedom depends critically on the successful use of expert systems.

Current implementations of expert systems run too slowly. Merely increasing the computational speed of uniprocessors will not be able to guarantee that real-time demands be met for large expert systems. Speedup via parallel processing must be pursued along with the optimization of sequential implementations.

Parallel expert systems have been investigated at universities and industrial research laboratories around the U.S. and abroad (notably in Japan). Prototypes of multiprocessors specifically designed for expert systems have been built. Results to date indicate that only certain applications are amenable to parallelization. In many cases, the degree of parallelism achieved is less than 10. In order to obtain higher speedup values, we must understand why expert systems are difficult to parallelize, how they should be written and partitioned to obtain maximum parallelism, and how they can be effectively mapped onto parallel architectures.

In order to address these questions adequately, a survey of the current state-of-the-art in parallel processing for expert systems has been carried out. Section 2 begins with a description of well known symbolic computation paradigms and state-of-the-art sequential implementation for them. Section 3 surveys four parallel hardware architectures specifically proposed for symbolic computation: DADO, NETL, the connection machine, and PIM. Section 4 surveys various parallel extensions to existing symbolic programming languages—parallel Lisps, Parallel OPS5, parallel PROLOGs, and parallel *object-oriented* languages. Section 5 discusses some of the mapping strategies used to implement parallelism. Section 6 describes the inherent parallelism observed in expert systems today and suggests why parallelizing expert systems is difficult. Finally, section 7 discusses how expert systems might be parallelized and some feasible or productive research directions.

1.1 An Introduction to Expert Systems and Symbolic Computation

Artificial Intelligence (AI) is the area of computer science concerned with the study of intelligence in human behavior. Many computer programs capable of representing and processing knowledge have been constructed to support a wide range of applications. These applications include natural language understanding, robotics, learning, and reasoning, as well as problem solving in specific domains such as chemistry, geology and medicine. Unlike conventional software, these AI programs operate on *symbols*, as well as *numbers*. Problem state information and problem solving knowledge are represented by data *structures* (or *shapes*) as well as *values*. As the problem solving process (e.g., by resolution and refutation, forward and backward chaining, hypotheses testing, or constraint propagation) proceeds, arithmetic operations as well as *pointer manipulation* are performed by the hardware—creating new data structures, discarding old ones and changing the values, sizes and shapes of existent structures.

Many paradigms have been proposed to represent problem solving knowledge and state information for this kind of computation. For example, *predicate calculus* employs sequences (or *lists*) of

symbols connected by the connectives: "$\wedge$" (and), "$\vee$" (or) and "$\Rightarrow$" (implies). Reasoning can be implemented based on *resolution* (Nilsson, 1982) or *rules of inference*, e.g., *modus ponens*. *Rule-based systems* distinguish knowledge (represented as *if-then* rules or *productions*) and database (or "*working memory*") explicitly. Rules are *activated* when their *left-hand-sides* match entries in the database. These rules may modify, delete or generate new entries to the data. At least two kinds of deduction can be performed under this paradigm: *backward* and *forward* reasoning.

Production systems such as CLIPS (Giarratano, 1989) and OPS5 (Forgy, 1981) implement *forward chaining*. Productions are repeatedly applied to the working memory to deduce new facts (or *working memory elements*, WMEs). As shown in figure 1, each production application cycle consists of three phases: *match*, *resolve*, and *act*. First, all productions are *matched* against the working memory. All productions whose left-hand-side (LHS) are satisfied are gathered together into a *conflict set*. One production will be selected from this *conflict set* for execution. Conflict resolution may be based on several criteria such as: *weighted productions* and *time-stamps*. Either the production with the highest weight or one matching the most recently added WME is chosen (Forgy, 1981). The *firing* of the right-hand-side (RHS) of a selected production may create new WMEs, or modify or destroy old ones. Productions which matches the current working memory are then selected for *conflict resolution* again. This cycle is repeated until no more productions can be *fired* (i.e., until the *conflict set* is empty).

Other logic programming systems (e.g., Prolog) support *backward* reasoning (or *goal-directed deduction*). A problem is solved by starting at the *goal* state, working towards the *initial* state. The *hypothesis* (or *goal*) to be proven is first put into the *goal-list*. If the goal cannot be matched with facts in the data-base, productions will be matched against it. A rule whose RHS *unifies* with the goal may generate subgoals (from its LHS) which replace the original goal in the *goal-list*. Unification determines whether two terms can be made textually identical by finding a set of substitutions for variables in the terms. All occurrences of each variable are replaced by its substitution. Because both terms are allowed to contain variables, unification can be thought of as a bidirectional pattern matching operation. This process iterates until all subgoals are verified.

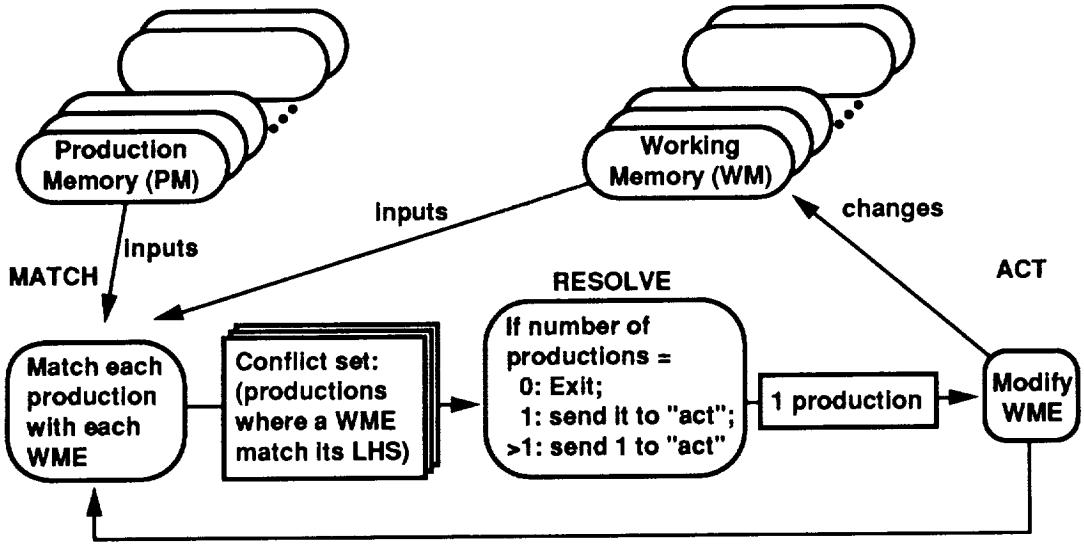

Figure 1. Production system's three-phase cycle: *match, resolve, and act.*

Besides *declarative* representations, knowledge may also be encoded *procedurally* and *structurally*. Heuristic knowledge which describes sequences of actions to be performed in well-specified situations can be represented naturally as small programs (or *procedures*). Specialized data structures (e.g., *semantic nets*) may be used to explicitly link important facts and concepts together. *Frames* and *objects* combine both representation techniques by attaching procedures to structured data. Reasoning proceeds via message exchange and processing. In response to messages received, an object may create other objects, modify its internal states, or send messages to other objects.

2. SEQUENTIAL EXPERT SYSTEM IMPLEMENTATION

2.1 Software and Hardware Requirements for Symbolic Processing

Languages proposed for symbolic computations include list processing languages (e.g., Common Lisp (Steele, 1984), object-oriented languages (e.g., Small-Talk, Goldberg and Robinson (1983)), *Flavors* with Symbolics Lisp (Cannon, 1982), and logic programming languages (such as OPS5 (Forgy, 1981), and Prolog (Clocksin and Mellish, 1981). In order to implement these languages efficiently, new requirements are placed on compilers, operating systems and hardware architectures originally optimized to support arithmetic operations on data cells. Perhaps the most demanding feature of an AI language is its ability to construct, modify and access complex data structures dynamically during run time. These structures can be built up as *lists* or *objects*—which allow structural and behavioral properties to be *inherited* implicitly through a complex hierarchy of class structures. Primitives such as "+" or ">" of these languages must be able to operate on different data types and structures.

In order to support dynamic data structures, storage must be managed efficiently and transparently at run-time. Because AI languages assume an inexhaustible supply of storage cells, storage cells that are no longer in use (or "garbage") must be identified, collected and recycled. Furthermore, the run-time system configuration must be able to support dynamic data typing (or even code modification on the fly!).

The von Neumann computer does not support this kind of (symbolic) computation directly. Hardware features supporting run-time type checking, garbage collection and pointer manipulation/arithmetic have been incorporated into Lisp and PROLOG machines to facilitate the efficient implementation of expert systems.

Finally, the process of developing AI applications is unique in that the algorithm to solve the problem (and sometimes the problem itself) is not necessarily well defined at the beginning. Program performance and solution method have to be incrementally refined. This, in turn, creates a demand for sophisticated program development environments which include

Debuggers that enable program execution to be traced, stepped and backtracked;

Inspectors that allow complex data structures to be browsed and displayed; and

System management tools that help maintain multiple code generations consistent, and perform incremental compilation for large software systems.

Since the invention of Lisp in 1959, symbolic processing has become more efficient via advances in hardware architectures as well as compiler technology. Following are examples of single-user Lisp and PROLOG machines, and new algorithms (e.g., RETE) for implementing production systems.

2.2 Lisp Machines

Lisp and object-oriented programs have been efficiently implemented on Lisp machines (such as Symbolics 3600s, XEROX 1100s and TI Explorers). Hardware architectural features designed specifically to enhance the performance of symbolic computations include:

Tagged memory architecture—A few bits of each data word (called the *tag* field) is reserved for encoding information about the word (such as its type or whether it is garbage). Lisp machines contain hardware that operates on the *tag* field in parallel with the ALU to perform run-time type-checking and garbage collection efficiently.

Hardware stacks—There are three hardware stacks in the Symbolics 3600, the *control stack*, *binding stack* and *data stack*, that are used to support tail recursion, shallow binding and reduce garbage collection overhead. Because Lisp is basically a functional language, efficient implementation of stacks reduces the time spent in function calls and returns.

Large and fast local disks—This provides support for fast virtual memory.

Large real memory—Symbolic computation generates garbage. As memory becomes more and more fragmented, memory references become nonlocalized. In order to reduce page-fault, large real memory is required.

Single user machine—The user has complete control over the machine. Machine idle time is used for system activities such as garbage collection.

cdr-coding—A compact internal representation scheme for lists that also eliminates recursion when traversing the list structure is implemented.

Object-oriented programs execute efficiently on Lisp machines

Slot value access—a single (Lisp operation) "let" provides the correct bindings,

Message processing—a "let" first provides the proper context, the method (message handler) is then retrieved via a hash table, the *method* is then applied with the arguments supplied, and

Class inheritance and mixing—various object classes can be combined to construct a new class, which inherits structural and behavioral features from its component classes.

Symbolics 3600s employs the copy-swap (Moon, 1984) garbage collection algorithm. They have since introduced refinements such as *ephemeral* garbage collection (Hewitt and Lieberman, 1983) and special hardware such as the Oracle (Moon, 1984). An Oracle is a special-purpose hardware table responsible for keeping track of references to and creations/destructions of ephemeral objects. It helps reduce the number of nodes to be traversed (therefore, the time spent) during garbage collection. XEROX's 1100 uses *reference count* (Bobrow, 1980) to collect garbage at run time. Because not all garbage can be identified this way, the entire memory has to be *marked-and-swept* (McCarthy, 1960) every now and then.

2.3 PROLOG Machines

Sequential execution of logic programs such as Prolog have been greatly improved by the concept of the Warren Abstract Machine (WAM) suggested by David Warren (Warren, 1983). WAM introduced the following features:

Retrieval of all used space on backtrack—During execution, a stack of data structures is maintained whose space must be reclaimed if backtracking is necessary.

Last call optimization—During execution, the environment of each clause is placed on the stack so that future calls can refer to it. This environment would include subclauses, variables, bindings, and so on. However, for the last clause, there is no need for its environment to be stored because no remaining clauses (that would refer to it) exist.

Environment trimming—This reduces the memory required to store the problem state and the search space.

Instructions to index clauses based on the first argument—This reduces the time needed to search for a matching clause.

Reordering of goals prior to execution—Logic programs execute according to the order the goals were asserted by default. Reordering may help minimize the backtracking and/or failures that may occur.

Many of these ideas were studied and incorporated by Japanese scientists working under the Fifth Generation Computer System (FGCS) project. FGCS is managed under the Institute for New Generation Computer Technology (ICOT) established in 1982 by the Ministry of International Trade and Industry (MITI) (Kawanobe, 1984). FGCS aims to produce extremely efficient knowledge information processing systems by addressing three key technologies: VLSI architecture, parallel processing and pattern matching hardware. They take the view that current computer systems must be redesigned for symbolic computations. Better performance measurement tools and environments for experimentation must also be developed.

The initial stage of the (three-staged) FGCS project resulted in the development of the Personal Sequential Inference (PSI) machine. The prototype was rated at 30K LIPS (logical inferences per second). It incorporated UNIRED, a hardware accelerator developed at the University of Tokyo

(Moto-oka, 1984), to increase the speed of unification and reduction in logic programs. Other features designed for efficient execution of logic programming include:

tagged architecture

horizontal microprogrammed control

high speed local memories

multiway jump capability according to the contents of a register

dereference of variables by the hardware

dedicated internal busses for control and internal communications

2.4 The RETE Algorithm

Besides designing custom hardware systems and a complete set of software tools encompassing compilers, operating systems and debuggers, certain symbolic computation paradigms can be implemented efficiently by exploiting observed behavioral characteristics. Recall that a production system executes in a three-phased cycle: *match*, *resolve* and *act*. Three observations can be made:

1. Approximately 90% of processing time is spent in the *match* phase.

2. Very few working memory changes are made every cycle.

3. Many productions share common match-patterns in their left-hand-side (LHS).

The RETE (the Latin word for network) algorithm (Forgy, 1982) (currently implemented in CLIPS, OPS5 and R1) makes use of these observations and compiles a *match-network* from the LHS of all the productions prior to execution. Working memory elements (and changes to them) propagate down branches of the network as *tokens*. Branches which fail the match (or are not affected by the change) are not touched. Because common match-patterns share the same branch, the number of *matches* performed is reduced. Many improvements have been proposed to the original algorithm; these include (Schor et al., 1986):

a single *modify* operator with a new trigger algorithm to replace "delete-and-add-with-changes";

arbitrary grouping of pattern condition elements: this allows common patterns to be shared even if the shared text is in the middle or end of the LHS;

incremental pattern match on demand: new rules may be added after the production system has begun to execute; and

the TREAT (TREe Associative Temporal redundancy) algorithm (Stolfo, 1985).

The RETE algorithm requires more memory during execution because the intermediate results of rule firing have to be saved in the nodes. The "old" *modify*, consisting of a delete and add function, triggered the same rule to fire because the *add* portion made it look like a new working memory element. The new trigger algorithm triggers on a *modify* only if that is the desired effect.

2.5 Summary: Sequential "AI Applications" Are Still Too Slow

Given all the "state-of-the-art" solutions mentioned above, execution of large expert systems is still unable to meet the requirements of many applications. For example, it was estimated that an equivalent of 1 trillion (i.e., 10^{12}) von Neumann computer operations per second is required to perform the vehicle-vision task at a level that would satisfy the long-range objectives of DARPA'S Strategic Computing Program. Current technology achieves 100 million (i.e., 10^8) operations per second at best. This implies that, at least, a 10^4 times speedup have to be achieved in order to perform tasks such as vehicle vision in the Autonomous Land Vehicle Project (Wah and Li, 1988b) in real time. Many AI applications, such as air traffic control, pilot's associate program and speech understanding, cannot be used at all if they cannot execute in real-time.

A 10,000-fold speedup is unlikely to be achieved on a single processor system based on innovations in software implementation, sequential hardware architecture and device technology alone. Software optimizations for many basic symbolic operations have been nearly exhausted. Conventional computing system architecture, on the other hand, has been pushed to their limits of operation as applications grow in size and scope. Electronic computer systems based on the traditional von Neumann architecture cannot be made orders of magnitude faster than the current systems because of at least two fundamental limitations, namely:

1. the speed at which electrical signals propagate between components; and

2. the wavelength of the light used for (and, therefore, the resolution of) the lithographic process in device manufacturing—which limits the size of devices that can be made.

Note: Multiprocessing must be pursued in order to speed up expert system applications.

3. MULTIPROCESSORS FOR EXPERT SYSTEMS

In order to speed up the execution of any application via parallel processing, three elements are needed: a multiprocessor, a parallel formulation of the application, and a resource management system that maps the application onto the multiprocessor.

Multiprocessors can be classified into two major classes: shared-memory versus distributed-memory architectures (as shown in fig. 2). Shared-memory machines consist of an aggregate of processor modules and a (logically speaking) global memory connected via some communication network. In practice, this global memory may consist of multiple memory modules —each of which is equally accessible by all processor modules (fig. 2(a)). Computing processes may communicate via shared data-structures. Distributed-memory computers, on the other hand, do not possess any (logical or physical) global memory. Each processing element (or *site*) consists of a processor unit and some local memory (fig. 2(b)). The processor at each site has exclusive access rights to its own memory. Computing processes communicate by message passing alone. Both multiprocessors shown in figure 2 are homogeneous —in that all memory modules, processor modules and processing elements are identical. The exact topology of the interconnection network is not specified; examples of topologies proposed to date include

1. The Omega Network for the Ultracomputer (Goto, 1983) and a shared bus for the Sequent Balance 21000 (Using the Sequent Balance 8000, 1986) for shared-memory architectures; and

2. Binary N-cube connections (e.g., Caltech Cosmic Cube (Seitz, 1985)) for distributed-memory architectures.

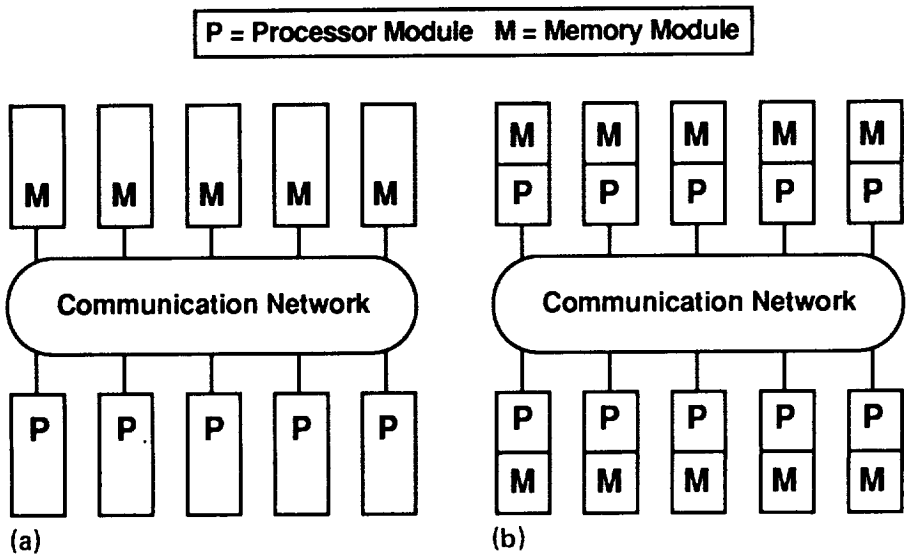

Figure 2. Architectures. (a) Shared-memory, (b) distributed-memory.

Given an ideal multiprocessor, with thousands of processing elements and extremely fast access times between these elements, speedup cannot be obtained unless the application is formulated such that most of these processing elements are doing useful work most of the time. This implies that a single computation must be partitioned into thousands of subtasks. Furthermore, unless these subtasks are *fairly independent*, much time will be wasted on waiting for synchronization.

If we have a parallel formulation of an application and a real multiprocessor, the speedup actually achieved still depends critically on how the application is actually mapped onto the machine. In other words, appreciable speedup cannot be obtained unless the resource management system of the mutiprocessor is able to do the following:

1. *properly trade off conflicting optimization subgoals*. For example, in order to minimize communication, the whole program should be placed on one site. On the contrary, maximizing concurrency suggests distributing the program over the entire multiprocessor.

2. *adapt to program behavior variations*. The resource management system must be able to detect and exploit various behavioral characteristics of application programs —both across different programs and fluctuations within a single execution.

3. *take advantage of specific hardware characteristics* (such as physical locality between a certain pair of processing elements) to reduce program execution time.

In order to help evaluate multiprocessor architecture proposed for symbolic computation, we must understand the impact of various architectural parameters on parallel program execution, scalability and performance:

1. The number of processors bounds the maximum (ideal) speedup.

2. The connection topology (and routing algorithms) affects the communication latency between different (processing) sites. Highly-connected topologies incur expensive hardware costs and do not scale; sparsely-connected topologies, on the other hand, impose long delays to most destinations.

3. The architecture of individual processing elements can influence the overall parallel architecture performance. On one end of the spectrum, the connection machine has ten thousands of (fine-grained) one-bit processors operating in SIMD mode. At the other extreme, the Intel iPSC/2 "Touchstone γ" has hundreds of RISC processors (Intel i860) operating in coarse-grain MIMD mode.

4. Shared-memory architectures are less scalable than distributed-memory architectures. Scheduling and communication, on the other hand, are simpler on shared-memory architectures.

The machines surveyed in this section include the CMU's DADO and NETL, MIT's *connection machine*, and FGCS's PIM.

3.1 DADO

The processing elements (PE) of DADO (Stolfo et al., 1983) are connected as a binary tree. Speedup is achieved through the distribution of storage and the parallel execution of *matches* and *updates* based on simple broadcast up and down the tree. Each PE has a special I/O device that can perform three global operations efficiently:

1. *BROADCAST*—send message to all descendents,

2. *REPORT*—send message to ancestor, and

3. *MAX-RESOLVE* —determine the maximum value among the current node and its two descendents.

Because of its tree structure, messages can be broadcasted to all nodes in *log n* time. Each PE can operate as a master (in MIMD mode) or a slave (in SIMD mode). A master PE executes instructions in its own local memory and uses its descendents as needed by *BROADCASTing* to them. A slave PE executes instructions *BROADCASTed* from its ancestor and then *REPORTs* back. The final solution of a computation can be determined by performing the *MAX-RESOLVE* function on the current node's value and the two results returned from its descendents. Hardware was implemented to support the functions: *maximum*, *minimum* and *average*.

Production systems were mapped onto DADO by dividing the binary tree into three logical layers. The top layer, called the uppertree, performs synchronization, conflict-resolution and the act phases (it serves basically as the decision maker). Productions are distributed across the next layer, the PM-level. At this level, the match phase and instantiations take place. The bottom layer, the WM-subtrees, holds the working memory elements (WMEs) at its leaves. Variations based on this algorithm include the following (for more detail, see section 2.4):

1. full distribution of production memory—distribute productions to processors in groups along with relevant working memory elements;

2. Miranker's TREAT algorithm (TREe Associative Temporal redundancy)—same as basic algorithm except that certain states are saved in memory;

3. fine grain RETE—RETE network compiled into binary tree; and

4. multiple asynchronous execution—for handling multiple rule firings.

In order to reduce the communication bottleneck between peer nodes on different halves of the tree, data were duplicated wherever needed. This introduced consistency problems. Furthermore, it was difficult to coordinate a large number of PES for full-speed operation because, in some cases, some nodes were slower than others and the links joining them are of different lengths. Buffers were considered; however, in other cases, the buffers created bottlenecks. There were two prototypes proposed (Stolfo et al., 1984):

DADO1: operational since April 25, 1983 with 15 PEs executing at 3.5 MHz and rated at 4 MIPS each. The speedup obtained was limited mainly because different tasks on different nodes required different processing times.

DADO2: gate-array technology; 1023 PES; runs at 12 MHz; 570 MIPS

3.2 NETL

NETL (Fahlman, 1980) is a fine-grain SIMD machine designed by Fahlman in 1979 and later refined by Touretzky in 1984. It implements a *semantic net formalism* which encodes knowledge and

concepts into graphic representations. Its processing elements are (logically) interconnected as nodes in a semantic net. The (logical) interconnections acts as the arcs. Messages are exchanged along these interconnections during computation. The NETL hardware, as shown in figure 3, consists of a number of PEs connected to a common bus and a switch which provides the logical connection between processors. A processor requiring a connection (to establish a relation) to another processor would send a request to the switch.

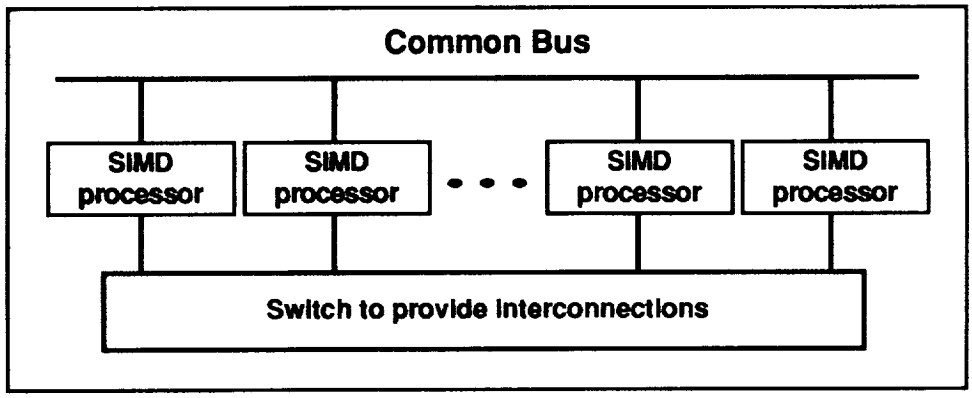

Figure 3. The NETL architecture.

Parallel reasoning on NETL is performed via *marker passing* (Hendler, 1988). *Tokens* are sent through *nodes* (i.e., PEs) that lead to the solution. When a token goes through a node, a bit at the node is set. When the goal is reached, the nodes with the bit set constitutes the search space. For example, a node satisfying all the preconditions of a production could be located by propagating the preconditions concurrently through the network. The node with a bit set for each precondition would be the one that satisfies the rule. Marker passing is basically a "nondeductive" search. It depends on an associative network which allows it to traverse the links, regardless of the format of the link, in order to mark the relevant endpoints (i.e., being able to reach the node associated with the current node). Because binding information can be ignored, marker passing is faster than deductive search. Unfortunately, marker passing can return incorrect paths because binding information is not taken into consideration. A path evaluator with a global view could help eliminate this problem.

Another important feature of marker passing is that there is virtually no contention. Several markers arriving at the same node are simply logically OR'ed (Way and Lee, 1988a). Unfortunately, the network controller in NETL (which is essentially, a single serial computer) can be a bottleneck—especially when a fact is being connected with every object that it is related to. NETL has been simulated only with software.

3.3 Connection Machine

The Connection Machine (CM) (Hillis, 1985), originally proposed by Hillis in 1981, also implements a semantic net formalism. It attempts to overcome the problems NETL faced 1) an expensive and non-scalable switch, 2) the lack of information passed between processors (markers only), and 3) the small local memory available on each processor. NETL passes only *markers* whereas CM is

able to perform reasoning based on the exchanged messages of arbitrary length between cells, manipulation of address pointers, and dynamic construction of structures.

This computer was originally designed with a fine-grain MIMD architecture in order to allow larger memory sizes but the commercial implementation, considering feasibility tradeoffs (cost for larger memory size versus cost of more processors), used a SIMD architecture. The PEs connect in a *hypercube* configuration. Each PE is given a fixed number of connections to other PEs. All PEs execute in a lock-step manner based on an external clock and instructions from the front-end host computer. A set of flags on each PE can be selectively set—thereby giving more flexibility and expressiveness in the host computer's control. Users interact with the CM via the front-end host computer (e.g., Sun workstation or Symbolics Lisp machine). What appears to be memory locations that stored their working values are, in actuality, separate processing elements.

The performance of this computer depends on the quantity of data used and the interdependencies of the data. Because the PEs have small local memories, data can be spread out over several PEs, thereby requiring several communication steps in order to process a single piece of data. This computer is in commercial use today and also supports various applications other than those simply intended for marker-passing with an inheritance hierarchy.

3.4 Parallel Inference Machine

The construction of multiprocessors for logic programs is, in fact, a major goal of Japan's Fifth Generation Computer System project (Goto, 1989). The overall target performance of the Parallel Inference Machine (PIM) is 10M to 20M reductions per second (rps). The pilot machine PIM/P, with 128 PEs connected as a *hypercube*, executes 50 nsec cycles in a four stage pipeline. PIM will be developed based on lessons learned from building sequential personal PROLOG machines (PSI). Multi PSIs were networked together forming multiprocessors —Multi-PSI I (1986-1987) and Multi-PSI II (1987-1989)—to test parallel software schemes eventually to be executed on PIM (Fuchi and Nivit, 1988). Multi-PSI II incorporates faster PEs and a more intelligent network than Multi-PSI I. These schemes include

1. The PIM will be programmed in *Kernel Language Version 1* (KL1). KL1 includes parallel extensions to KL0 (the language for programming PSI). The parallelism in KL1 is based on the syntax and semantics of Flat Guarded Horn Clauses (FGHC). FGHC was chosen because of its clear and simple semantics. Language modifications, new data abstractions and meta-inference mechanisms were also introduced.

2. The *Multiple Reference Bit* (MRB) and copying schemes are used to manage multiple references, for recognizing reclaimable data for local garbage collection and for detection of shortage in memory space.

3. The Weighted Export Counting (WEC) scheme is used for interprocessor incremental garbage collection. WEC is a specialized version of reference count —each object keeps a record of whether it is referenced by processes at remote sites. Instead of simply "counting" references, external references are weighted; so, instead of increasing/decreasing the count by 1 each time, the

(integer) weight of a particular reference is added/subtracted when it is created or no longer needed. When this count is zero, objects can be reclaimed locally (assuming that there were no local references).

4. The Weighted Throw Count (WTC) scheme is used for controlling the termination of a parent before the termination of all its children processes. It behaves similarly to the WEC except that it applies to processes rather than objects.

5. A meta-programming facility, known as SHOENS, is provided for better resource and task management and larger grain of computational units. SHOENS are metaprogramming capabilities that supports termination detection. A SHOEN signals termination once all its goals complete. Goals are solved *depth-first* and suspended with a *non-busy waiting* scheme.

Dynamic load balancing strategies are being researched. Currently, idle processors will request work from busy processors.

3.5 Summary

Production systems were mapped onto DADO's binary tree structure directly and executed in parallel. The speedup obtained, however, was small because the many processing elements remain idle most of the time. NETL introduced a *marker passing* approach to reasoning but the network switch (which services all connection requests) was a bottleneck. The connection machine improved upon the NETL design by eliminating this network switch. Unfortunately, its small local memories tend to fragment much of the data and code. The performance of PIM is yet to be seen with its new garbage collection techniques, termination detection techniques and meta-programming capabilities with SHOENS. These hardware approaches introduce new possibilities as well as further areas of research. All of the machines surveyed, with the exception of FGCS's PIM, were either never built (NETL), not used (DADO), or are now marketed for non-AI applications (CM II).

4. PARALLEL LANGUAGES FOR EXPERT SYSTEMS

Parallel languages can be constructed by extending an existing language or defining a completely new language. Extensions to an existing language are desirable for those who want to "parallelize" existing program codes with minimal effort whereas new languages can incorporate a style of programming that (potentially) improve the utilization of underlying hardware. The languages surveyed in this section include parallel Lisp, parallel OPS5, parallel PROLOG, and *object-oriented* languages.

4.1 Parallel Lisps

QLisp (queue-based multiprocessing Lisp) (Gabriel and McCarthy, 1984) was designed to execute on shared-memory architectures. A scheduler assigns new processes on a global queue to the

least busy processor based on a *round-robin* algorithm. The degree of multiprocessing can be controlled explicitly at run time. Very few extensions are made to Lisp although some existent constructs take on new meanings in a multiprocessing setting. Processes are created using two constructs: QLET and QLAMBA.

(**QLET** *pred* ((x_1 arg_1)... (x_n arg_n)) . *body*)—QLET expresses parallelism that has regularity over, for example, an underlying data structure. A predicate "*pred*" is evaluated before any other action regarding this function is taken. If *pred* evaluates to (), then QLET acts exactly like a (sequential) LET; in other words, arg_1 ... arg_n are evaluated sequentially and their results are bound to x_1 ... x_n, respectively. If *pred* returns EAGER, QLET does not wait; it proceeds to evaluate the functions in the body concurrently – with processes spawned for each arg_i. It blocks only if the value of an arg_i is actually required and not yet available. Otherwise, processes are spawned, one for each arg_i. The process evaluating the QLET waits until all arg_i are available and their values bound to x_i before it resumes and executes the *body*.

(**QLAMBDA** *pred* (*lambda-list*) . *body*)—QLAMBDA can be used to create *closures* dynamically for expressing less regular parallel computations. Again, when *pred* evaluates to (), QLAMBDA behaves exactly like LAMBDA. If *pred* returns EAGER both the parent process and the QLAMBDA-closure executes. The parent blocks only if it actually requires the result of the closure. Otherwise, the closure executes as a separate process. The parent process (caller of QLAMBDA) suspends until the spawned process finishes.

QLisp runs currently on Encore multiprocessors.

MultiLisp (Halstead, 1986) is an extension to *Scheme* (Abelson and Sussman, 1984) with constructs for supporting parallel execution. It provides lexical scoping as well as "first-class citizenship" for Lisp functions—which enables functions to be passed and returned as values (to other functions which may reside on other processors), or stored as part of a data structure. The construct "(***future*** *body*)," creates a process to evaluate *body* and returns a *future* which acts as a *place holder* for (or a *promise* to deliver) the result of the evaluation. While the evaluation proceeds, the *future* can be used for constructing data structures or passed around as an argument. Any process which actually requires the value of the result will be suspended unless the evaluation process has completed. A "delay" construct is also provided which implements *lazy evaluation*—allowing a *future* to be evaluated only on demand. MultiLisp is implemented on the Butterfly machine and Concert, an experimental shared-memory multiprocessor at MIT (Halstead et al., 1986).

QLisp and MultiLisp were both designed for shared-memory architectures. Execution is performed sequentially if desired. Only a few constructs are needed to initiate parallel processing. In both cases, information about the importance and requirements of tasks cannot be specified. Proposed solutions include associating *sponsors* (Theriault, 1983) and *priorities* (Halstead, 1986) with tasks.

Communicating Lisp processes have also been proposed as a simple and inexpensive approach to implement a parallel Lisp environment on distributed architectures Model, 1980). Lisp processes can be coordinated to work on one particular problem. Tasks can be dynamically created and passed

around because Lisp allows program code to be constructed and interpreted during execution. Communicating Lisp processes have been implemented on transputer systems (Smith, 1983).

A fine-grain version of parallel Lisp called ***Lisp** (previously known as CmLisp) is also implemented on the Connection Machine (CM) (Hillis, 1985) CmLisp is an extension of Common Lisp specifically designed to support the parallel operations of the hardware architecture. Parallelism is achieved by allowing an operation to be performed simultaneously over each element of a large data structure. A new data structure, called **Xector**, allows (each value of) a set of values to be stored by a set of processing elements. This enables entire data sets to be operated on simultaneously. Some of the concurrent operations available are: *combine, create, modify,* and *reduce*. New SIMD-PARALLEL operations can also be defined based on these concepts. *Lisp was designed to hide implementation details and the CM architecture from the user.

4.2 Parallel PROLOGs

Recall that logic programs consists of facts and rules. Facts (or *clauses*) describe the attribute of an object (fig. 4(a)) or the relationship among objects (fig. 4(b)). Rules, however, are procedural interpretations. It consists of a *head* and a *body*. The head of the rule is the clause to the left of the ":-" while the body is everything to the right (fig. 4(c)). The body may contain more than one clause. Note that for variables that exist in the rules (e.g., ***X***, ***Y***, and ***A***), the binding of all occurrences of the same literal must share the same bindings (e.g., ***A*** in the first clause of the body must be the same as ***A*** in the second clause). To begin execution of a logic program, an initial goal is supplied and matched against the facts and rules. If it is in the facts list, then it is satisfied and execution is complete. Otherwise, it may match the head of a rule. In this case, each of the clauses in the body must be satisfied (i.e., they become subgoals that must be satisfied) in order for the entire rule to be satisfied. This continues until all the goals/subgoals are satisfied. Failure occurs if a goal cannot be satisfied.

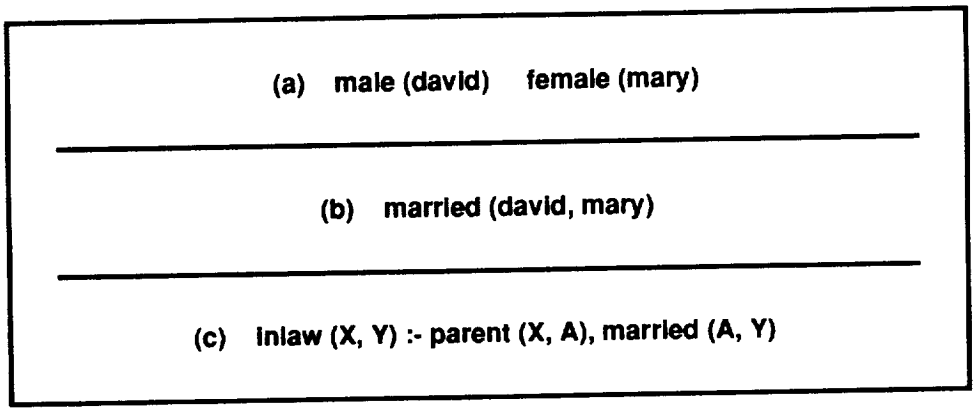

Figure 4. Logic program syntax.

Based on this simple execution model, four sources (and combinations of these) of parallelism can be exploited:

1. *Or-parallelism*: Each rule, whose head unifies with a fact, can be solved in parallel.

2. *And-parallelism*: Processes execute in parallel to solve each clause of the body. This may also involve communication and coordination among the processes to resolve variable binding conflicts between the clauses.

3. *Stream-parallelism*: Eager evaluation of structured data is treated as a *stream*. This is a pipelined form of AND-parallelism. Unifications for the first subgoal are forwarded to the process with the second subgoal as soon as it becomes available, and so forth. In this manner, the other subgoals may execute in a somewhat parallel fashion. But as with most pipelines, there will be a latency overhead (in filling the pipeline).

4. *Search-parallelism* (also known as *parallel unification* (Quinn, 1987): Assertions are grouped so that search may proceed in parallel without contention to a single resource.

Two models which exploit some of these sources of parallelism have been proposed.

The **AND/OR** parallel execution model (Conery and Kibler, 1981) was first implemented as an interpreter using the Dec-10 Prolog. This model provided a method for partitioning a logic program into small asynchronous and logically independent processes that communicate via messages. The parameters which affected the actual speedup obtained include: size and number of messages sent during the solution of a problem; the ratio of idle time to processing time for each PE—when there are more processes than PEs, the amount of time processes are *blocked* vs. *ready* in each PE are important; for each PE, the costs for preparing, routing, and receiving messages; and cost for database search for each PE.

This **AND/OR** model builds a tree of processes as computation proceeds. Messages are exchanged only between the *parent* and *children* processes—not between siblings and peers. *Start, redo* and *cancel* messages are sent from parents to children who reply either with *success* or *fail* messages. In this model, an OR-process replaces the backtracking in sequential computation by acting as a *message center*. It distributes work among its own children and sends the first successful tuple received back to its parent. Meanwhile, its other children continue working and *success* messages collected are only sent up to the parent if a *redo* message is received. Eager evaluation is implemented by sending *redo* messages to successful children so that more solutions exist if the parent should require it. If no child succeeds, it returns a *fail* message to the parent. The OR process also filters out duplicate solutions since it maintains the list of successful messages from its children and a list of messages it has sent up to its parent. A parallel AND-process is not as straight forward as it may seem because distributing literals across PEs has its problems:

1. Binding conflicts among the literals need to be resolved.

2. Not much computation can be done while waiting for literals to be bound.

3. Some literals fail if attempts are made to solve them before certain variables are instantiated.

It was proposed that the literals be ordered (based on a data dependency graph) in order to determine which must be performed sequentially and which could be computed in parallel (Conery, 1983). During execution, literals that become eligible for processing are identified and incorporated into the graph. If a failure occurs within the graph (i.e., failure occurs when a child process concludes that there is no possible solution below it in the search space) the **backward execution** algorithm would be used to select the candidate literal which may cure the failure (of the AND-process).

The second model, the **RAP-WAM** model for concurrent PROLOG proposed by Hermenegildo (Hermenegildo, 1985), was based on DeGroot's Restricted-And-Parallelism (RAP) work (DeGroot, 1984) and parallel extensions to WAM. RAP reduces the overhead associated with runtime management of variable binding conflicts between goals. Previous approaches were unsatisfactory—compile-time approaches required user input on the variables while run time approaches, such as the AND/OR model, were complex and expensive. RAP's technique combined both compile time and run time analysis. RAP was able to solve the binding conflict problem by analyzing the clauses involved at compile time and performing simple checks on the variables at run time. The combined analysis is implemented in the extensions made to WAM (Hermenegildo and Tick, 1988) as follows.

1. *conditional graph expressions* (CGE): CGEs allow users to express potential parallelism in the form of condition statements that would generate either a parallel or sequential graph depends on the result of the condition test. A parallel graph would contain a point in the graph where several goals may be satisfied in parallel.

2. *"goal stacks" to support on-demand scheduling*: A goal stack is located on every PE and contains goals that must be satisfied (i.e., work to be done). These goals are generated during a *parcall* (i.e., parallel call) which places goals that can execute in parallel onto a stack for either the local processor or a remote processor. Remote idle processors can take a goal off a local goal stack as needed. This eliminates the need to have busy processors schedule work for idle processors.

3. *message buffers*: Because some processors may take longer than others, a buffer was used for pending messages. A processor with longer processing time would check its buffer, once it finishes its current computation, for messages that may have arrived during its non-idle time.

4. *two new types of stack frames*: *parcall frames*—coordinate and synchronize execution of parallel calls by keeping track of all the goals, especially those taken off by remote processors in case backtracking is necessary; and *markers*—supports backtracking by marking the point at which backtracking should begin; appropriate register contents are saved before another clause is executed.

This model performed search with minimal backtracking by representing the problem as a condition graph to evaluate and analyze the possible paths to select the best solution. This analysis also provided dependency information among goals. The abstract (RAP-WAM) model has been studied through simulations and it is being implemented on a Sequent Balance 21000 computer.

4.3 Parallel *Object-Oriented* Languages

A number of *object-oriented* languages were studied by the Advanced Architecture Project (AAP) at Stanford University's Knowledge Systems Laboratory. The project's primary goal was to improve the performance of expert systems through parallel processing. Because the design space was prohibitively large, it was decided that only a few options would be explored at each implementation layer of the system: application, problem solving framework, resource management, programming language, and hardware architecture.

All experiments were performed on simulated multicomputers where processing sites were connected as a toroid. Besides a CPU, each site also had a separate communications controller which supports dynamic cut-through routing (Dally, 1987) and nonblocking message sending. Memory usage, code distribution and garbage collection were not simulated. Two programming languages were designed for/supported on this simulated parallel architecture:

1. A concurrent asynchronous object-oriented system (CAOS) (Brown et al., 1986)—CAOS objects were large grained asynchronous multiprocessing objects. Various message-sending primitives were defined, including synchronous and asynchronous SENDS, as well as SENDS which returned *futures*.

2. LAMINA (Delagi et al., 1987)—LAMINA provides extensions to Lisp to support functional programming, object oriented, and shared variable styles of programming. The implementation is based on the notion of a *stream*—a data type used to express pipelined operations by representing the promise of a (potentially infinite) sequence of values.

The concurrent problem solving frameworks developed were based on the blackboard problem solving model (Nii, 1986). Domain knowledge is represented as a number of knowledge sources—each of which consists of *if-then* rules. The problem state is represented on a globally shared data structure known as the blackboard. Knowledge sources can make changes to the blackboard by creating, destroying or modifying existent blackboard nodes. A scheduler governs the operation of different knowledge sources. The blackboard model has demonstrated success in many areas of real-time expert system applications such as situation analysis (Spain, 1983) as well as speech understanding (Erman et al., 1980). Parallelism may be extracted from the blackboard via: 1) *knowledge parallelism*—multiple knowledge sources can execute concurrently; 2) *pipeline parallelism*—information at different levels of abstraction are processed simultaneously; and 3) *data parallelism*—different parts of the blackboard can be can be operated on concurrently.

Two implementations, *Cage* and *Poligon*, were proposed for shared- and distributed-memory architectures, respectively. With *Cage* (Concurrent AGE) (Nii et al., 1988), a centralized scheduler is responsible for the parallel execution of rules and knowledge sources. This serializing control mechanism was discarded in *Poligon* (Rice, 1988). Blackboard nodes are distributed over the entire multiprocessor network. Modifying a slot of a blackboard node invokes the rule directly attached to the slot. These invocations created processes on different processors for execution. This reduces the length of the critical sections on the processors holding blackboard nodes, and enable multiple rule

invocations on the same blackboard node. Extra mechanisms had to be implemented to help the nodes iterate (in a distributed hill-climbing fashion) towards a coherent and correct answer.

Two major applications have been implemented based on these concurrent blackboard architectures to evaluate their performance. They are ELINT and AIR TRAC. ELINT is a expert system for interpreting processed, passively acquired, real-time radar emissions from an aircraft (Brown et al., 1986). AIR TRAC attempts to understand and interpret radar tracks (Nakano and Minami, 1987) in real time. Dependence graphs are used to decide on the decomposition scheme. This simulation, modelling a distributed memory system, has been able to achieve up to 100 times speedup over a single processor. Where there were bottlenecks, replication was used.

Major problems encountered include: bottlenecks due to memory contention and a central scheduler; race conditions with locking mechanisms and consistency problems with atomic operations.

4.4 Summary: Parallel Languages for Expert Systems

Most of the examples of parallel languages for symbolic computation were extensions to existing languages. This enabled users to parallelize their application with less effort than if a new language were defined. Extensions to the Lisp dialect include QLisp, MultiLisp, *Lisp, and LAMINA. They provide constructs that will allow for parallel execution such as the spawning of tasks onto a global queue or through the use of ***futures***. CPARAOPS5, on the other hand, actually changes OPS5 programs into parallel C programs. Other work focused on parallel execution models for logic programs. The AND/OR model created a tree of processes to provide parallelism. The RAP-WAM model used *conditional graph expressions* to denote times at which work could be performed in parallel. Most of these approaches were based on shared-memory architectures because they are simpler, involving fewer overhead issues.

5. MAPPING COMPUTATIONS TO MULTIPROCESSORS—IMPLEMENTATION OF PARALLELISM

Recall that in order to speed up the execution of any application via parallel processing, three elements are needed: a multiprocessor, a parallel formulation of the application, and a resource management system that maps the application onto the multiprocessor. This section deals with the third element—the mapping of expert system applications onto parallel architectures. In particular, the models cited here involve production systems; i.e., the mapping of *if-then* rules and working memory elements onto the DADO and multicomputers.

5.1 Mapping Production Systems onto DADO

Five algorithms were designed to map production systems onto the DADO binary tree architecture. They are described below.

1. The *original DADO algorithm*: Recall (as outlined in section 3.1) that the binary tree of PEs are divided conceptually into three levels: Uppertree, PM-level, and WM-subtrees. The uppertree performs the conflict-resolution and act phases. Productions are stored in the PM-level where the match phase also takes place. The WMEs are stored at the (leaves of the) bottom layer, the WM-subtrees. Computation begins by propagating (WME) *changes* down the tree and matches back up with conflict resolution performed at each level using *max-resolve* (*Max-resolve* is capable of comparing the values of specific registers on a specified set of PEs in one machine cycle. This improves the speed of the conflict resolution phase of the production system cycle.) until the root is reached and one rule is chosen to be fired.

2. *Full distribution of production memory*. PEs alternate between MIMD and SIMD mode dynamically. First, rules are divided into small groups and distributed to each PE along with a predefined rating criteria together with the WMEs that match some patterns of the LHSs of some these productions. Every PE (in MIMD mode) performs the match for its local rules based on changes broadcasted to all PEs. Then, in SIMD mode, each PE performs a rating on its matches using the given criteria. Once a rule has been selected, using *max-resolve*, its RHS actions are broadcasted as WME changes and the match phase begins again. The performance of this algorithm depends mainly on the complexity of the local match function and on the size of the local WMEs.

3. *Miranker's Tree Associative Temporal Redundant (TREAT) algorithm*: This algorithm improves upon the first algorithm by saving the state of the matches of the previous cycle. During the next cycle, *delete* actions will eliminate the matches that no longer apply and *add* actions will select new matches. This eliminates the need to repeat matching unaffected rules on each cycle.

4. *Fine grained RETE*: The RETE algorithm is mapped logically onto the DADO architecture. The leaves of the DADO binary architecture implement *constant-test* nodes (fig. 5). *Matches* will propagate up the tree to its ancestors that may represent *two-input* nodes. While the leaves are executing, the *two-input* nodes are idle waiting for results. A *match* reaching the top of the tree indicate the production selected to be executed. Changes to WMES are then broadcasted to the leaves of the tree and the cycle repeats. This behaves much like a pipelined architecture where every processor works in MIMD mode.

5. *Multiple asynchronous execution*: This algorithm attempts to allow for the execution of several production system programs or of several conflict set rules of one production system program concurrently.

At this point, the performance of each algorithm relative to the others have not been studied. However, each algorithm has its own drawbacks and features: Algorithm 1 tends to repeat many of its matches from the previous cycle. Associative memory would help by identifying the rules that are affected by a WM change and eliminating unnecessary matches. The performance of algorithm 2 would be limited if the local WM is too large for a PE to store conveniently. Performance varies with complexity of the local match although this may be reduced by the hashing of the WM. Algorithm 3 is simply a refinement of algorithm 1 taking temporal redundancy into consideration. Algorithm 4 provided performance improvements through a pipelined effect. It can also support the overlay of a second network since the leaves would be idle once its matches are broadcasted to its ancestors. Finally, algorithm 5 makes suggestions to support multiple rule firings whether it be from separate

production systems or from a single production system. This was something not considered in the previous algorithms. Each algorithm targets a different characteristic in the production system being implemented and the comparison between them would not be straightforward.

5.2 Mapping RETE Networks onto Multicomputers

Production systems can be mapped onto Multicomputers in an "obvious" manner by representing nodes in the RETE network as objects and tokens as messages. A single processor will be used for each object in order to avoid placing any sequential constraints on the objects. Unfortunately, because very few changes are made in each production system cycle, processor utilization would be low. Furthermore, changes made to the same object have to be processed sequentially (producing a possible bottleneck in that processor). Finally, the overhead involved in message sending and local scheduling is relatively high because the messages being exchanged are extremely short and the processing associated with each arriving message is also very small (on the order of 50 instructions according to Gupta's thesis (Gupta, 1986)). This approach would not provide any performance improvements.

Gupta (Tambe et al., 1989) proposed another approach centered around two *globally distributed hash tables*. Furthermore, the processors were partitioned based on *functionality* rather than by node boundaries. The partitioning included (see fig. 5 for an example of these nodes)

1. a *control* processor,
2. several *constant-test node* processors,
3. several *conflict-resolution* processors, and
4. remaining processors used for *matching* (mainly two-input nodes).

Recall that matching is the major bottleneck in production system execution and the RETE algorithm has greatly improved its performance. In particular, within the *match* phase, the most time-consuming activity involves the two-input node activations. As a token arrives at either the right or left input, it must be matched against the list of tokens at the other input. In this hash-table approach, tokens are stored in global hash-tables. One hash table contains tokens destined for the right input of the two-input node while the other contains the left input. This enabled referencing of the values at the two inputs in one step (based on the key to reference both tables simultaneously). Also, since the table was distributed, tests on several two-input nodes can be performed in parallel.

The performance of this approach depends on the discriminability of the hash function. The current hash function was based on 1) the variable binding being tested and 2) the unique "node-id" of the destination two-input node. Speedup depends on the length (M) of the chain of dependent node-activations. Based on initial simulation, this approach reached a 26-fold speedup in the simple case with M = 5, a 15-fold speedup in some special cases where M = 10 and speedup of 9-fold for

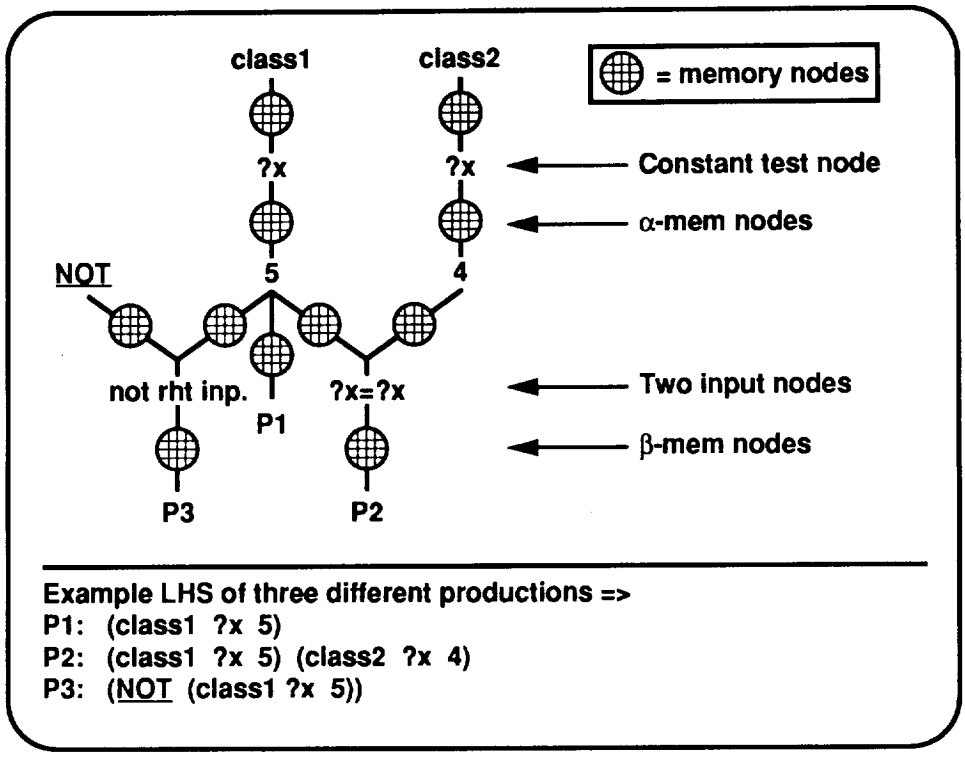

Figure 5. RETE network.

M = 15. Mapping onto a MPC eliminated the bottleneck potential of a centralized task scheduler found in shared memory architectures. However, it is possible that the hash function could potentially serialize two distinct tokens by placing them on the same processor.

5.3 Distributing Rules on Multicomputers

Moldovan views the problem of mapping production systems onto multiprocessors as a performance optimization problem (Moldovan, 1989). Moldovan makes the following definition—A rule (or production) is defined by $\mathbf{P_i} \equiv \mathbf{L_i} \rightarrow \mathbf{R_i}$ where L_i and R_i are the *left-hand-side* (*LHS*) and *left-hand-side* (*RHS*) of P_i respectively. $\mathbf{K_i}$, the *intersection* of L_i and R_i (of P_i) is defined as the set of WME(s) which is required to satisfy P_i and still remains unaltered (or true) after P_i fires. Based on the above definition of P_i, L_i, R_i and K_i, two rules (P_1 and P_2, say) can be input dependent or output-input dependent:

input dependence: P_1 is input dependent on P_2 if $L_1 \cap (L_2 - K_2) \neq 0$; i.e., P_1's *LHS* has elements common with those eliminated from the WM by firing P_2.

output-input dependence: P_1 output-input dependent on P_2 if $R_1 \cap (L_2 - K_2) \neq 0$; i.e., P_1's *RHS* has elements common with those eliminated from the WM by firing P_2.

Having defined these dependency relationships between rules, he was able to select

1. *compatible* rule subsets that can be fired in parallel (independent of one another) because they produce the same results when executed in any order (where two rules that are neither *input dependent* nor *output-input dependent* in both directions are said to be *compatible*.); and

2. *communicating* rule pairs such that the firing of one may (i) cause the other to fire or (ii) make the conditions that enable the other to fire no longer true.

These two relationships are characterized by two matrices—**P** (parallelism matrix) where

$$p_{ij} = \begin{cases} 0 \text{ if rules i and j are } \textit{compatible} \\ 1 \text{ otherwise} \end{cases}$$

and **C** (communication matrix) where

$$c_{ij} = \begin{cases} 1 \text{ if rule i is input or input - output dependent on rule j} \\ 0 \text{ otherwise} \end{cases}$$

. By inspecting the **P** matrix, he was able to reduce the size of the conflict set because firing rules in a *compatible* subset in any order produce equivalent results. Based on the **C** matrix, he proposed a *quadratic assignment* formulation that, when it is solved, will yield an optimal assignment of rules to processing sites such that communication across sites is minimized. Unfortunately, the quadratic assignment problem has been proved to be NP-complete.

5.4 Summary: Mapping Expert Systems to Multiprocessors

In this section, examples of three approaches to the mapping of expert system applications onto parallel architectures were discussed. These efforts are important in providing a resource management system to speed up the execution of an application via parallel processing. The first effort focused on production systems for the DADO architecture. Five algorithms were discussed, each of which targeted different characteristics in the production system such as temporal redundancy and multiple rule firings.

The second effort mapped the RETE network onto multicomputers. Because the RETE network concentrated on the match phase of a production system cycle, hash tables were utilized to improve the search bottleneck of two-input nodes. Speedup through this effort is limited to the amount of time generally consumed by a match phase, approximately 90%. Improvements on the two-input nodes ranged from 9-fold for general case to 15-fold for special cases and 26-fold for very simple cases.

The final effort took a different approach. The mapping problem was viewed as a performance optimization problem which, unfortunately, proves to be NP-complete. Research is still being done on the mapping problem and attempts like the ones described in this section would help identify appropriate mappings of specific applications to certain architectures as well as identifying features and drawbacks of an architecture.

6. MEASURING PARALLELISM IN EXPERT SYSTEMS

Parallel implementation of production systems (based on OPS5) have been extensively studied at Carnegie-Mellon University since the early 1980s under Forgy (1982). Recall that a production system operates in three-phased cycles: *match-resolve-act*. Besides obtaining speedup via parallel implementations of each phase, further speedup may be obtained by allowing execution between phases to overlap (i.e., occur simultaneously): e.g., the *conflict-resolve* phase could begin as soon as one rule successfully enters the conflict set; the *match* phase of the next cycle can begin as soon as an action is carried out. Nevertheless, because of the observation that 90% of processing time is spent in the *match* phase, their efforts focused on parallel implementations of the RETE-match algorithms. Three parallel implementations were proposed (Gupta, 1986):

1. *production parallelism*—rules fired concurrently;

2. *node parallelism*—each node of the RETE-NETWORK are allowed to fire concurrently;

3. *intranode parallelism*—the processing of each token to a two-input node of the RETE-network are allowed to occur concurrently;

These implementations of decreasing granularities subsume one another and produce increasing levels of speedup. Further speedups were obtained when changes to working memory are allowed to occur concurrently. CParaOPS5 (Gupta et al., 1988), a parallel version of OPS5, was written and executes on an Encore Multimax multiprocessor (a shared-memory architecture). A global hash table was used to store memory nodes of the RETE-NETWORK for fast lookup. Speedup values of 6.3 to 12.4 has been obtained depending on the application.

6.1 Parallelism in Production Systems and Flat Concurrent Prolog Systems

Perhaps the most detailed measurements on expert systems were carried out by Dr. Anoop Gupta for his dissertation research (Gupta, 1986) His results were based on six expert systems containing up to 1100 rules (written in OPS5). Three important observations were made:

1. Very few changes are made to working memory per recognize-act cycle. The number of RETE network nodes affected by changes to the working memory (and, therefore, the number of productions requiring significant processing) is also small. For the applications selected only two to three changes are made to the WM per cycle and less than 30 productions are affected for changes to the WM.

2. The total number of node activations per change is independent of the number of productions in the production-system program.

3. Variation in processing requirements for the (few) affected productions is large.

These observations were explained as follows:

1. Firstly, an expert system contains a large body of knowledge about many different types of objects and diverse situations. The amount of knowledge (therefore, number of rules) associated with any specific situation is expected to be small. Secondly, most working-memory elements describe only a few aspects of a single object or situation; therefore, they could could apply to only a few rules.

2. There are two probable causes for the small and independent size of the affected rule-sets:

Programmers recursively divide problems into subproblems when writing large programs. The size of these subproblems is independent of the size of the original problem—it is determined by the complexity that the programmer can deal with at one time.

"A large body of knowledge may be organized hierarchically for easy comprehension and reasoning." (Gupta, 1986).

3. Rules accounting for different situations, formulated based on different heuristics, obviously exhibit different levels of complexity and require different amount of processing.

These observations (and explanations) are not only specific to systems written in OPS5—they transcend all expert systems. For example, Leon Alkalaj's measurements on *flat concurrent prolog* systems (Mierowsky et al., 1985) reveals that although the number of goals which exist at some point during execution may exceed several thousand, the average number of goals available for concurrent processing for most of the time is much smaller (< 12) (Alkalaj, 1989). These observations suggest major obstacles to obtaining speedup for expert systems from parallel processing.

6.2 Obtaining Speedup via Parallel Processing is Difficult

Observation ***A*** (presented in section 6.1) suggests that the inherent parallelism available in expert systems is small. Observation ***B*** further suggests that:

1. smaller production systems do not necessarily run faster than larger ones;

2. allocating one processing element to each RETE node (or production) is not a good idea because most of them will be idle most of the time; furthermore,

3. there is no reason to expect that larger production systems will exhibit more speedup from parallelism.

Observation ***C*** suggests that scheduling is critical to obtaining whatever (small) speedup is available in the system. Unfortunately, dividing production systems into partitions which require similar amount of processing is difficult because good models are not available for estimating the processing required by productions and it varies over time.

Compile-time analysis and optimization cannot be performed effectively on expert system applications because run-time behavior is highly data-dependent. An expert system contains a large body of knowledge capable of dealing with different situations. The actual situation to be tackled is not known until program execution time. Therefore, program behavior (such as frequency of procedure calls, and number of storage and communication requirements) is highly data dependent. Compile-time optimization techniques cannot be applied directly to such computations. The computation must be partitioned such that many processors can be kept busy for most (if not all) situations the expert system is likely to face.

Synchronizations take place frequently in search problems. At the heart of many expert system applications is a heuristic search problem: *given an initial state, apply knowledge to prune the search tree to arrive at the goal state*. This two-phase cycle of knowledge application and problem state modification can be parallelized in many ways—each of which requires frequent synchronization. Consider the following examples:

The RETE algorithm (OPS5): the *conflict-resolution* phase must complete before the *act* phase can begin. Even though the *conflict-resolution* phase could begin as soon as each rule successfully enters the *conflict set*, the *best* rule to be applied next cannot be determined until all candidates (including the slowest ones) have arrived.

The Soar algorithm (Laird, 1986): Computation is divided into an *elaboration* phase and a *decision* phase. Within each phase all productions satisfied may be fired concurrently. However, the *elaboration* phase must finish completely before the *decision* phase may proceed and vice versa. Because Soar systems usually go through a few loops internally within each phase, the serializing effect of these two synchronization points is not as bad as that between the *conflict-resolution* and *act* phases in OPS5.

And-parallelism in PROLOG: common terms which occur when two clauses being worked on simultaneously must share identical bindings. This requires that tasks working concurrently on two subgoals communicate whenever such bindings are to be changed.

Parallel evaluation of alternatives (i.e., *or-parallelism*) does not necessarily speed up the time required to obtain a solution. Consider a search tree of depth d and branching factor b when the solution lies at one of the leaves of the tree. The best parallel algorithm (with sufficient information supplied) still requires the same amount of time (d steps) as the best sequential algorithm to locate the solution.

Finally, it is not clear whether the reasoning process can be naturally decomposed based on spatial (e.g., the solution of differential equations governing heat conduction over a metal plate) or temporal (e.g., real-time image enhancement) considerations in general.

7. PARALLEL PROCESSING FOR EXPERT SYSTEMS

Many "building blocks" developed to enable parallel execution of expert systems have been surveyed in sections 3 and 4. Measurement results presented in section 6, however, seem to indicate that the inherent parallelism available in expert systems is small. Can an expert system be formulated as a set of as highly parallel computations? Can these "building blocks" be put together effectively to support parallel computations? We do not have answers to these questions. However, we would like to draw on some fundamental results concerning speedup and parallel processing in section 7.1 and put forth some "food for thought" regarding future directions for research in section 7.2.

7.1 Speedup and Parallel Processing

Three principles are discussed in this section:

1. A small section of sequential code in an application can significantly limit its speedup.

2. When partitioning a single application into ***n*** tasks, ***n*** should be chosen with at least two things in mind: (1) the amount of intertask synchronization required and (2) the architectural parameters of the hardware.

3. A number of software organization structures have been proposed for multiprocessors. Speedup could be obtained, however, only certain criteria are met for each proposed organization.

Recall Amdahl's Law which states that the maximum speedup S for a computation obtainable on a multiprocessor with **p** processors is governed by

$$S \leq \frac{1}{f + (1-f)/p}$$

where **f** is the fraction of the computation that has to be executed sequentially. For example, in order to obtain a 100-fold speedup with 1,000 processors (no overhead, no communication latencies), 99.1% of the computation must be performed in parallel. In this scenario, each site of this multiprocessor spends only 10% of the total time doing useful work. A simple application of this result suggests that parallel RETE-match algorithms can give at most a 10-fold improvement because it speeds up only the *match* phase which takes 90% of the execution time.

When partitioning a single application into tasks, the *grain size* of the tasks should be chosen such that: (1) there is enough parallelism to exercise the PES of the parallel processor and (2) communication and process management overhead must not outweigh the speedup obtained from parallel processing. With production systems, it seems that extremely fine-grained tasks (of the order of 100 machine instructions) are needed for effective parallel execution (Gupta, 1986). Minimizing the scheduling overhead for such fine grain tasks is a major obstacle to achieving higher degree of speedup.

A number of effective software organization structures have been proposed for multiprocessors. These include *software pipelines*, *systolic algorithms* (Kung, 1982; Kung, 1984), *divide-and-conquer* (*tree-of-processes*), and *relaxed* (Quinn, 1987) or *asynchronous processes*.

Pipeline structures are routinely employed in the design of factory production lines, photocopiers, memory subsystems and CPU's for computers. The *speedup* obtained from pipeline structures depends on 1) how many pipelines there are (i.e., the width ***w*** of the pipeline); 2) the number of stages with each pipeline (i.e., the length ***l*** of the pipeline); and 3) the (variation of) service times at each pipeline stage.

Maximum speedup (***w*l***) is obtained when all pipeline stages require the same service time; and there is a large number of requests—making the *setup* and *trail-off* times negligible.

Certain distributed computations can also be arranged as pipelines. Each stage of the software pipeline consists of a number of computing processes. Data items are processed incrementally by passing them from one stage to another. This characteristic, known as *temporal decomposability* gives *length* to the pipeline. The processing at each stage may be carried out concurrently by partitioning the data to be processed into (relatively independent) subsets. This characteristic, known as *spatial decomposability* gives *width* to the pipeline.

Systolic algorithms can be regarded as a special kind of software pipeline because data are also exchanged regularly and rhythmically between processing nodes. However, there is usually more than one direction of flow. Maximum efficiency is again obtained when the computations performed at each processing node takes essentially the same amount of time.

Unlike pipelining where processes are responsible for different stages of a computation, divide-and-conquer algorithms attack complex problems by (recursively) ***partitioning*** a single computation down into smaller manageable sub-problems. These (relatively independent) sub-problems can be sent to different processing sites to be processed concurrently. The results obtained are later combined to obtain the complete solution. Partitioning as well as scheduling can be performed either at compile-time or run-time. With *divide-and-conquer*, maximum speedup is obtained when the following occurs:

1. the *setup* (task creation) and *trail-off* (recombination of results) times are small compared to the computation performed by each task;

2. the number of tasks created is appropriate for the multiprocessor (given its task creation and management overhead); and

3. tasks are effectively scheduled (mapped) onto the multiprocessor.

An algorithm is said to be ***relaxed*** or *asynchronous* if processes can work with the most recently available data and, essentially, never have to block and wait for a specific piece of data from another process.

Whether an expert system can be spatially or temporally decomposed is application dependent. Decomposition boundaries can be identified based on a careful analysis of the nature of the input data set and the reasoning process. Sometimes, these boundaries may not be obvious from first inspection.

7.2 Conclusions

What is the best strategy for building parallel expert systems? We have several choices.

1. *Define a specific class of hardware architecture, then study the mapping of programs to these architectures* (e.g., **Lisp* for the Connection Machine, *marker passing* on NETL and *MultiLisp* for the BBN Butterfly)

2. *Focus on a specific class of software architecture, construct a multiprocessor that best matches the program* (e.g., DADO for RETE, PIM for concurrent PROLOG)

3. *Establish a unified model to construct hardware and software architectures such that subsequent mapping between them can be easy and effective* (e.g., *CParaOPS5* for the Encore Multimax)

We do not yet have an answer to this question. Nevertheless, we would like to offer some promising research directions.

Requirements for parallel implementation should begin at the top of the software hierarchy and driven top-down—from problem solving paradigm design to programming language implementation to operating system to machine architecture. We should decide the macro software organization most likely to extract parallelism from the expert system application before choosing *concurrent objects* vs. parallel lisp, or shared-memory vs. distributed-memory architectures. In many cases, speeding up the "expert" portion itself may not produce the overall speedup value we require. Bottom-up approaches produce machines that *could* exhibit orders of magnitude speedup if *suitable* applications can be found.

The most efficient parallel execution model for expert systems may not look or work in any way like the way they are specified. AI programming paradigms (whether they be *knowledge sources* with *blackboards*, *productions* on *working memory*, recursive goal-driven deduction based on PROLOG clauses or list processing) are designed to enable knowledge to be encoded and processed in a way similar to that carried out by human beings. Since we cannot, up to this day, fully understand how we represent and solve problems in our heads, these models are shallow and incomplete; they are applicable within a certain domain and behave intelligently only to a certain extent. They are not necessarily efficient for execution on a von Neumann computer. However, when we stop asking how computers can be modified to execute these paradigms directly, efficient execution models may follow. The RETE algorithm for sequential execution is a very good example for the following reasons:

1. Knowledge, stated as productions, is actually implemented via a RETE network which does not even distinguish individual rules;

2. The process of reasoning, which involves the application of productions to facts, is actually implemented by the continuous flow of bits and pieces of facts (called tokens) through a complicated sieve (called the RETE network).

RETE is successful because redundancies in knowledge representation (which enhance the readability of the code and facilitates the incremental modification of the knowledge-base) are eliminated by intelligent compilation techniques. The same holds for parallel processing: instead of asking "How can we build a parallel processor for expert systems" or "What is the most effective way to parallelize such and such a paradigm?"; maybe we should ask "Among all the combinations of parallel hardware and software structures known to produce high speedup values, which of them can be adapted to process and represent knowledge?"

In conclusion, we suggest that speedup cannot come from parallelizing one particular existent paradigm or language or operating system. We must do the following:

1. Understand how to break up the problem with minimal contention for accessing shared resources and reduced dependencies. This could probably be done by considering (*macro* and *micro*) data dependencies in the expert system when designing its parallel implementation;

2. Re-examine problem solving and representation schemes (such as rules, blackboards, procedures, or logic programming) and be open-minded about efficient parallel execution models that may not resemble the *human problem solving process*; and

3. Explore parallelism at the application level; the nature of the application may suggest whether it can be parallelized temporally or spatially; also consider the portion of the application that is not *knowledge-based* (e.g., re-organizing the input/output procedures may save more time than merely replacing the sequential inference engine with a parallel one).

REFERENCES

Alkalaj, Leon: Architectural Support for Concurrent Logic Programming Languages, PhD Thesis, Computer Science Department, Univ. of Calif., Los Angeles, Calif., Aug. 1989.

Abelson, H.; and Sussman, G.: Structure and Interpretation of Computer Programs, M.I.T. Press, Cambridge, Mass., 1984.

Bobrow, Daniel G.: Managing Reentrant Structures Using Reference Counts, ACM Transactions on Programming Languages and Systems, vol. 2, no. 3, July 80, pp. 269–273.

Brown, Harold D.; Schoen, Eric; and Delagi, Bruce A.: An Experiment in Knowledge-based Signal Understanding Using Parallel Architectures, Knowledge Systems Laboratory, Report Number KSL 86-69, Computer Science Department, Stanford University, Stanford, Calif., Oct. 1986.

Cannon, Howard I.: Flavors, A Non-Hierarchical Approach to Object-Oriented Programming, 1982.

Conery, John S.; and Kibler, Dennis F.: Parallel Interpretation of Logic Programs, Communications of the ACM, May 81.

Conery, John S.: The AND/OR Process Model for Parallel Interpretation of Logic Programs, Dissertation for Ph.D in Information and Computer Science at Univ. of Calif. at Irvine, University Microfilms International, Ann Arbor, Mich., 1983.

Clocksin, William F.; and Mellish, Christopher S.: *Programming in Prolog*, Springer-Verlag: Berlin Heidelberg, 1981.

Dally, William J.: Wire-Efficient VLSI Multiprocessor Communication Networks, in Advanced Research in VLSI, In Advanced Research in VLSI: Proceedings of the 1987 Stanford Conference, Stanford, Calif., Paul Losleben, Ed., 1987, pp. 390–415.

DeGroot, Doug: Restricted AND-Parallelism, Proceedings of the International Conference on Fifth Generation Computer Systems, OHMSHA, Tokyo, 1984, pp. 471–478.

Delagi, Bruce A.; Saraiya, Nakul P.; and Byrd, Gregory T.: LAMINA: CARE Applications Interface, Knowledge Systems Laboratory, Report Number KSL 86-67, Computer Science Department, Stanford Univ., Stanford, Calif., Nov. 1987.

Erman, D. L.; Hayes-Roth, F.; Lesser, V. R.; and Reddy, D. Raj: The HEARSAY-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty, ACM Computing Survey, vol. 12, 1980, pp. 213–253.

Fahlman, Scott E.: Design Sketch For a Million-Element NETL Machine, Carnegie-Mellon Univ., Department of Computer Science, AAAI-80, Aug. 1980.

Fuchi, K. (ICOT, Japan); and Nivat, M. (INRIA, France): editors, Programming of Future Generation Computers, in the Proceedings of the First Franco-Japanese Symposium on Programming of Future Generation Computers, Tokyo, Japan, 6-8, October, 1986, Elsevier Science Publishers B.V., The Netherlands, 1988.

Forgy, Charles L.: OPS5 User's Manual, Carnegie-Mellon Univ., Pittsburgh, Penn., Report Number CMU-CS-81-135, July 1981.

Forgy, Charles L.: RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, Artificial Intelligence, vol. 19, 1982, pp. 17–37.

Giarratano, Joseph C.: CLIPS User's Guide Version 4.3 of CLIPS, Artificial Intelligence Section, Lyndon B. Johnson Space Center, June 1989.

Gabriel, Richard P.; and McCarthy, John: Queue-Based MultiProcessor Lisp, Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, ACM, Austin, Texas, Aug. 1984.

Gottlieb, A.; Grishman, R ; Kruskal, C. P.; McAuliffe, K. P. M.; Rudolph, L.; and Sniv, M.: The NYU Ultracomputer—Designing a MIMD Shared Memory Parallel Computer, IEEE Transactions on Computers, C-32, vol. 2, no. 175, Feb. 1983, p. 189.

Goto, Atsuhiro: Research and Development of the Parallel Inference Machine in the Fifth Generation Computer System Project, Technical Report: TR-473, Institute for New Generation Computer Technology (ICOT), Minato-Ku, Tokyo, Japan, April 1989.

Goldberg, A.; and Robson, D.: SmallTalk-80: The Language and Its Implementation, Addison-Wesley, Reading, Mass., 1983.

Gupta, Anoop; Tambe, Milind; Kalp, Dirk; Forgy, Charles; and Newell, Allen: Parallel Implementation of OPS5 on the Encore Multiprocessor: Results and Analysis, International J. Parallel Programming, vol. 17, no. 2, Apr. 1988, pp. 95–124.

Gupta, Anoop: Parallelism in Production Systems, Ph.D Thesis, CMU-CS-86-122, Department of C.S., Carnegie-Mellon Univ., March 1986.

Halstead, Robert H., Jr.: Parallel Symbolic Computer, Computer Magazine, vol. 19, no. 8, Aug. 1986, pp. 35–43.

Halstead, R.; Anderson, T.; Osborne, R.; and Sterling, T.: Concept: Design of a Multiprocessor Development System, 13th International Symposium on Computer Architecture, Tokyo, June 1986, pp. 40–48.

Hendler, James A.: *Integrating Marker-Passing and Problem-Solving: A Spreading Activation Approach to Improved Choice in Planning*, Department of Computer Science, The University of Maryland, Lawrence Erlbaum Associates, Inc., Hillsdale, NJ, 1988.

Hermenegildo, M. V.: An Abstract Machine for Restricted AND-Parallel Execution of Logic Programs, Univ. of Texas at Austin, Austin, Texas, 1985.

Hillis, W. Daniel: *The Connection Machine*, The MIT Press, Cambridge, Mass., 1985.

Hewitt, Carl; and Lieberman, Henry: Design Issues in Parallel Architectures for Artificial Intelligence, Mass. Institute of Technology, Cambridge, Mass., Artificial Intelligence Laboratory, A.I. Memo No. 750, Nov. 83.

Hermenegildo, M.; and Tick, Evan: Memory Performance of AND-Parallel Prolog on Shared-Memory Architectures, Proceedings of the 1988 International Conference on Parallel Processing, Aug. 15–19, 1988, Volume II Software, Aug. 1988, pp. 17–21.

Kawanobe, Kazukiyo: Current Status and Future Plans of the Fifth Generation Computer Systems Project, Proceedings of the International Conference on Fifth Generation Computer Systems, 1984, pp. 3–17.

Kung, H. T.: Why Systolic Architectures? IEEE Computer, January 1982, pp. 37–46.

Kung, S. Y.: On Supercomputing with Systolic/Wavefront Array Processors, Proceedings of the IEEE, vol. 72, no. 7, July 1984, pp. 867–884.

Laird, John E.: Soar User's Manual, 4th Edition, Xerox PARC, 1986.

McCarthy, John: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, Communications of the ACM, vol 3, no. 4, 1960, pp. 184–195.

Model, Mitchell L.: Multiprocessing via Intercommunicating Lisp Systems, Physics Department, Brandeis Univ. Waltham, Mass., 1980.

Moldovan, Dan I.: RUBIC: A Multiprocessor for Rule-Based Systems, IEEE Transactions on Systems, Man and Cybernetics, vol. 19, no. 4, July/Aug. 1989, pp. 699–706.

Moon, David A.: Garbage Collection in a Large Lisp System, Transactions of the ACM, March 1984, pp. 235–246.

Moto-oka, Tohru; Tanaka, Hidehio; Aida, Hitoshi; and Maruyama, Tsutomu: The Architecture of a Parallel Inference Engine - PIE -, Proceedings of the International Conference on Fifth Generation Computer Systems, 1984, pp. 479–488.

Mierowsky, C.; Taylor, S.; Shapiro, E.; Levy, J.; and Safra, S.: The Design and Implementation of Flat Concurrent Prolog, Weizmann Institute of Science, Rehovot, Israel, Technical Report CS85-09. July 1985.

Nii, H. Penny; Aiello, Nelleke; and Rice, James: Frameworks for Concurrent Problem Solving: A Report on Cage and Poligon, Knowledge Systems Laboratory, Report Number KSL 88-02, Computer Science Department, Stanford Univ., Stanford, Calif., Feb. 1988.

Nii, H. Penny: Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures, The AI Magazine, 1986, pp. 38–53.

Nilsson, Nils J.: *Principles of Artificial Intelligence,* Springer-Verlag, New York, 1982.

Nakano, Russell; and Minami, Masafumi: Experiments with a Knowledge-Based System on a Multiprocessor, Knowledge Systems Laboratory, Report Number KSL 87-61, Computer Science Department, Stanford Univ., Stanford, Calif., Oct. 1987.

Quinn, Michael J.: *Designing Efficient Algorithms for Parallel Computers,* Univ. of New Hampshire, McGraw-Hill Book Company, 1987.

Rice, James P.: Problems with Problem-Solving in Parallel: The Poligon System 1.0, Knowledge Systems Laboratory, Report Number KSL 88-04, Computer Science Department, Stanford Univ., Stanford, Calif., Jan. 1988.

Schor, Marshall I.; Daly, Timothy P.; Lee, Ho Soo; and Tibbitts, Beth R.: Advances in RETE Pattern Matching, IBM T.J. Watson Research Center, New York, March 26, 1986.

Seitz, C. L.: The Cosmic Cube, Communications of the ACM, vol. 28, no. 1, Jan. 1985, pp. 22–33.

Using the Sequent Balance 8000, ANL/MCS-TM-66, Mathematics and Computer Science Division, Argonne National Laboratory, 1986.

Stolfo, Salvatore J.; and Miranker, Daniel P.: DADO: A Parallel Processor for Expert Systems, IEEE, 1984, pp. 74–82.

Smith, K.: (Ed.). New Computer Breed Uses Transputers for Parallel Processing, Electronics International, Feb. 24, 1983, pp. 67–68.

Stolfo, Salvatore J.; Miranker, Daniel P.; and Shaw, David Elliot: Architectures and Applications of DADO: A Large-Scale Parallel Computer for Artificial Intelligence, Columbia University, New York, Jan. 18, 1983.

Spain, David S.: Application of Artificial Intelligence to Tactical Situation Assessment, Proceedings of the 16th EASCON 83, Sept. 1983.

Steele, Guy: *Common Lisp: The Language*, Digital Press, 1984.

Stolfo, Salvatore J.: Five Parallel Algorithms For Production System Execution on the DADO Machine, Columbia University, Computer Science Department, 1985.

Theriault, D.: Issues in the Design and Implementation of Act2, M.I.T. Artificial Intelligence Laboratory Technical Report AI-TR 728, Cambridge, Mass., June 1983.

Tambe, Milind; Acharya, Anurag; and Gupta, Anoop: Implementation of Production Systems on Message Passing Computers: Techniques, Simulation Results and Analysis, Carnegie-Mellon Univ., New York, NY, CMU-CS-89-129, April 20, 1989.

Warren, David H.: An Abstract Prolog Instruction Set, Technical Note 309, Artificial Intelligence Center, SRI International, Oct. 1983.

Wah, Benjamin W.; and Li, Guo-Jie: A Survey on the Design of Multiprocessing Systems for Artificial Intelligence Applications, IEEE Transactions on System, Man and Cybernetics, 1988.

Wah, Benjamin W.; and Li, Guo-Jie: Design Issues of Multiprocessors for Artificial Intelligence, Chapter 4 in Scientific Supercomputers and Artificial Intelligence Machines, K. Hwang and D. DeGroot, eds., McGraw-Hill, 1988, pp. 107–155.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM- 103886		

4. Title and Subtitle	5. Report Date
Parallel Processing and Expert Systems	May 1991
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Sonie Lau and Jerry C. Yan (Sterling Federal Systems Inc. Palo Alto, CA)	A-91077
	10. Work Unit No. 505-64-54

9. Performing Organization Name and Address	11. Contract or Grant No.
Ames Research Center Moffett Field, CA 94035-1000	
	13. Type of Report and Period Covered Technical Memorandum

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Washington, DC 20546-0001	

15. Supplementary Notes

Point of Contact: Sonie Lau, Ames Research Center, MS 244-7, Moffett Field, CA 94035-1000
(415) 604-4944 or FTS 464-4944

16. Abstract

Whether it be monitoring the thermal subsystem of Space Station Freedom, or controlling the navigation of the autonomous rover on Mars, NASA missions in the 1990s cannot enjoy an increased level of autonomy without the efficient implementation of expert systems. Merely increasing the computational speed of uniprocessors may not be able to guarantee that real-time demands are met for larger systems. Speedup via parallel processing must be pursued alongside the optimization of sequential implementations. Prototypes of parallel expert systems have been built at universities and industrial laboratories in the U.S. and Japan. This paper surveys the state-of-the-art research in progress related to parallel execution of expert systems. The survey discusses multiprocessors for expert systems, parallel languages for symbolic computations, and mapping expert systems to multiprocessors. Results to date indicate that the parallelism achieved for these systems is small. The main reasons are (1) the body of knowledge applicable in any given situation and amount of computation executed by each rule firing are small, (2) dividing the problem solving process into relatively independent partitions is difficult, and (3) implementation decisions that enable expert systems to be incrementally refined hamper compile-time optimization. In order to obtain greater speedups, *data parallelism* and *application parallelism* must be exploited.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Parallel processing Parallel hardware Parallel software Expert systems	Unclassified-Unlimited Subject Category – 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	39	A03

For sale by the National Technical Information Service, Springfield, Virginia 22161