N91-20700

PARALLEL PROCESSING AND EXPERT SYSTEMS

Jerry C. Yan
Sterling Federal Systems Inc.,
NASA Ames Research Center,
MS 244-4, Moffett Field, CA 94035
(415) 604-4381
jerry@pluto.arc.nasa.gov

Sonie Lau
Information Sciences Division,
NASA Ames Research Center,
MS 244-4, Moffett Field, CA 94035
(415) 604-4944
lau@pluto.arc.nasa.gov

Abstract

Whether it be monitoring the thermal sub-system of Space Station Freedom, or controlling the navigation of the autonomous rover on Mars, NASA missions in the 90's 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 large expert systems. Speed-up 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 US and Japan. This paper surveys the state-of-the-art research in progress related to parallel execution of expert systems. The survey is divided into three major sections: (i.) multiprocessors for parallel expert systems, (ii.) parallel languages for symbolic computations and (iii.) measurements of parallelism of expert systems. Results to date indicate that the parallelism achieved for these systems is small. The main reasons are: (i.) the body of knowledge applicable in any given situation and amount of computation executed by each rule firing are small; (ii.) dividing the problem solving process into relatively independent partitions is difficult; and (iii.) implementation decisions that enable expert systems to be incrementally refined hamper compile-time optimization. In order to obtain greater speed-ups, *data parallelism* and *application parallelism* must be exploited.

1. Introduction

The science and engineering objectives of NASA missions in the 90's cannot be met without an increased level of autonomy for both onboard and ground-based systems. For example, with *Mars Rover Sample Return*, the long delays associated with signal transmission between Mars and Earth require the Rover to make intelligent decisions and operate autonomously in real-time. The day-to-day operation of *Space Station Freedom* also depends critically on real-time expert systems — whether it be operating the thermal control sub-system, or flight tele-robotic servicers. Current implementations of expert systems run too slow. Merely increasing the computational speed of uniprocessors may not be able to guarantee that real-time demands be met for large systems. Speed-up via parallel processing must be pursued alongside the optimization of sequential implementations.

Parallel expert systems has been investigated at universities and industrial laboratories in the US and 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 most cases, the degree of parallelism achieved is less than 10. In order to obtain higher speed-up 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 issues adequately, a survey of 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, CParaOPS5, concurrent PROLOG, and object-oriented languages. Section 5 reports the inherent parallelism observed in expert systems today and suggests why parallelizing expert systems is difficult. Finally, section 6 discusses how expert systems might be parallelized and what reasonable research directions might be.

2. Sequential Expert System Implementation

2.1 Software and Hardware Requirements

Unlike conventional software, expert systems 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 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, knowledge may be represented *declaratively* (e.g. using *predicate calculus*) and processed based on *resolution*, simple *rules of inference*, *backward* and *forward* chaining. Knowledge may also be encoded *procedurally* (as programs) or *structurally* (as *semantic nets*). *Frames* and *objects* combine both representation techniques by attaching procedures to structured data.

Languages proposed for symbolic computations include list processing languages (e.g. Common Lisp), object-oriented languages (such as SMALL-TALK), and logic programming languages (e.g. Prolog). 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 language feature is the ability to construct, modify and access complex data structures dynamically during run-time. In order to support dynamic data structures, storage must also be allocated/reclaimed efficiently and transparently at run-time.

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.

2.2 Lisp Machines

Lisp and object-oriented languages have been efficiently implemented on Lisp machines (such as Symbolics 3600™, XEROX 1100™ and TI Explorer™). Hardware features designed specifically to enhance the performance of symbolic computations include: tagged memory architecture and processing hardware and hardware stacks. Lists are efficiently represented using *cdr-coding* schemes. Object-oriented programs also execute efficiently because slot value access, message processing, and class inheritance and mixing are implemented with very low overhead.

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) [1]. Many of these ideas were studied and incorporated by the Japanese Fifth Generation Computer System (FGCS) project — the initial stage of which resulted in the development of the Personal Sequential Inference (PSI) machine rated at 30K LIPS (logical inferences per second). It incorporated UNIRED [2], a hardware accelerator, to increase the speed of *unification* and *reduction*.

3. Multiprocessors for Expert Systems

Given all these "state-of-the-art" enhancements mentioned in section 2, execution of large expert systems is still unable to meet the requirements of many applications such as air-traffic control, pilot's associate and real-time speech understanding. Multiprocessing must be pursued, together with innovations in software implementation, sequential hardware architecture and device technology, in order to speed-up expert systems. The next two sections summarizes the major developments in hardware and programming languages for parallel symbolic computing. Four machines are described in this section: DADO, NETL, the *connection machine*, and PIM.

3.1 DADO

The processing elements (PEs) of **DADO** [3] are connected as a binary tree. *Matches* and *updates* are processed in parallel based on simple broadcasts up and down the tree. Each PE has a special I/O device that performs three global operations (BROADCAST, REPORT, and MAX-RESOLVE) efficiently. A PE may execute instructions in its local memory and enlists its descendents by BROADCASTing to them. Each descendent executes instructions received and REPORTs back. The final solution may have to be determined by performing the MAX-RESOLVE function on the parent node's result and the two returned from its descendents.

Production systems were mapped onto DADO by dividing the binary tree into three logical layers. The top layer serves as a "decision maker"; it performs *synchronization*, *conflict-resolution* and the *act* phases. Productions are distributed across the next layer where the *match* phase and *instantiations* take place. The bottom layer holds the working memory elements. In order to reduce the communication bottleneck between peer nodes on different halves of the tree, data was duplicated wherever needed; this introduced consistency problems. Two prototypes were proposed [4] — the first of which, DADO1, consists of 15 PEs rated at 4 MIPS each. The speed up obtained on DADO1 was limited mainly because different tasks on different nodes require different processing times.

3.2 NETL

NETL [5] is a fine-grain SIMD machine. Its PEs can be logically interconnected as nodes in a semantic net. Parallel reasoning on NETL was performed via *marker passing* [6]. *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 node 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.

3.3 Connection Machine

The PEs of the **Connection Machine** [7] are connected as a hypercube. All PEs execute in a lock-step manner based on an external clock and instructions from a 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. The performance of CM depends on the size and interdependencies of the data. Because the PEs have small local memories, data can be spread out over several PEs; thereby, requiring several communication steps to process a single piece of data.

3.4 Parallel Inference Machine

As specified by Japan's FGCS project overview [8], the overall target performance of the **Parallel Inference Machine (PIM)** is 10 to 20 million reductions per second (RPS). The pilot machine PIM/P, with 128 PEs connected as a hypercube, executes 50ns cycles in a four stage pipeline. Multiple PSIs have been networked together forming multiprocessors to test parallel system software eventually to be executed on PIM [9]. These include (i.) *Kernel Language Version 1* (KL1) — a parallel Prolog-like language based on *flat guarded horn clauses*, (ii.) a *multiple reference bit* scheme for local garbage collection, (iii.) a *weighted export count* to support inter-PE garbage collection., and (iv.) a *weighted throw count* scheme for terminating remote processes. Dynamic load balancing strategies on PIM are currently being researched.

4. Parallel Languages for Expert Systems

The parallel symbolic languages surveyed in this section (parallel LISPs, PROLOG, and *object-oriented* languages) augment existent languages with parallel constructs.

4.1 Parallel LISPs

QLISP [10] (queue-based multiprocessing Lisp) was designed to execute on shared-memory architectures. A scheduler assigns new processes on a global queue to the least busy processor in a *round-robin* fashion. 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 QLAMBDA. QLET expresses parallelism that has regularity, for example, over an underlying data structure. QLAMBDA creates *closures* dynamically for expressing less regular parallel computations. QLISP runs currently on Encore multiprocessors.

MULTILISP [11] is an extension to *Scheme* with constructs 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 arguments. Any process which actually requires the value of the *future* will be suspended unless the evaluation process has completed. A "delay" construct is also provided to support *lazy-evaluation* — allowing a *future* to be evaluated only on demand. MultiLisp is implemented on the Butterfly and Concert [12].

A fine-gain version of parallel LISP called ***LISP** (previously known as CmLisp) is implemented on the Connection Machine. Operations can be performed simultaneously over each element of a large data structure. Some of the concurrent operations available are: *combine, create, modify,* and *reduce.* New SIMD-parallel operations can also be defined based on these concepts.

4.2 Parallel PROLOGs

Four sources of parallelism (and combinations of these) can be exploited in Prolog:

- *Or-parallelism*: Each rule, whose head unifies with a fact, can be solved in parallel.
- *And-parallelism*: Processes execute in parallel to solve each clause of the body.
- *Stream-parallelism* is a pipelined form of AND-parallelism. Unifications for the first sub-goal are forwarded to the process working on the next one as soon as it becomes available and so forth.

- *Search-parallelism*: 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 [13] provides a method for partitioning a logic program into small asynchronous and logically independent processes. A tree of processes is built as computation proceeds. *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*[1]. It also filters out duplicate solutions by maintaining a list of successful messages from its children and messages sent to its parent. A parallel AND-process is more complicated because distributing literals across PEs has its problems[2] — the solutions to some of which were presented elsewhere [14].

The second model, **RAP-WAM** [15], was based on DeGroot's Restricted-And-Parallelism (RAP) work [16] and parallel extensions to WAM. RAP reduces the overhead associated with managing 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 analyzed the clauses at compile-time and performed simple checks on the variables at run-time [17]. RAP-WAM also performed search with minimal backtracking by representing the problem as a condition graph to evaluate/analyze possible paths to select the best solution. This analysis also provided dependency information among goals.

4.3 Parallel *Object-Oriented* Languages

The performance of two *object-oriented* languages for distributed-memory architectures were studied by simulation by researchers at Stanford University. CAOS [18] computations consists of large grained asynchronous multiprocessing objects. Various message-sending primitives were defined, including synchronous and asynchronous SEND*s*, and SEND*s* which returned *futures*. LAMINA [19] provided extensions to LISP to support functional , object oriented, and shared variable styles of programming. Its implementation was based *stream* — a data type used to express pipelined operations by representing the promise of a (potentially infinite) sequence of values. These languages supported two concurrent problem solving frameworks developed based on the blackboard problem solving model. *Cage* and *Poligon*, were proposed for shared- and distributed-memory architectures respectively [20, 21].

5. Measuring Parallelism in Expert Systems

Parallel implementation of production systems (based on OPS5) have been extensively studied at Carnegie-Mellon University. Besides obtaining speed-up via parallel implementations of each phase, further speed-up may be obtained by allowing execution between phases to overlap (i.e. occur simultaneously). 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 [22]. Three parallel implementations were proposed [23]:

- *production parallelism* — rules fired concurrently;
- *node parallelism* — each node of the RETE-network fired concurrently;
- *intra-node parallelism* — the processing of each token to a two-input node of the RETE-network occurred concurrently;

These implementations, of decreasing granularities, subsumed one another and produced increasing levels of speed-up. Further speed-ups were obtained when changes to working memory are allowed to occur concurrently. Speed-up values of 6.3 to 12.4 were observed depending on the application.

5.1 Parallelism in Production Systems and Flat Concurrent Prolog Systems

Based on detailed measurements on six expert systems containing up to 1100 rules (written in OPS5) [23], three important observations were made:

A. Very few changes were made to working memory per recognize-act cycle. The number of RETE network nodes affected by changes to the working memory was small.

B. The total number of node activations per change was quite independent of the number of productions in the production-system program.

C. Variation in processing requirements for the (few) affected productions was large.

[1] 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 are computed if the parent should require it. If no child succeeds, a *fail* message is returned to the parent.

[2] e.g. resolving binding conflicts among the literals, idle time waiting for literals to be bound; and some literals fail if attempts are made to solve them before certain variables are instantiated

These observations were explained as follows:

A. 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 only describe a few aspects of a single object or situation; therefore, they could only be of interest to a few rules.

B. Programmers recursively divide problems into subproblems when writing large programs. The size of these subproblems are independent of the size of the original problem; it depends only on the complexity that the programmer can deal with at one time.

C. Rules accounting for different situations, formulated based on different heuristics, obviously exhibit different 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, measurements on *flat concurrent prolog* systems also revealed that although the number of goals which exist at some point during execution may exceed 1000s, the average number of goals available for concurrent processing for most of the time is much smaller (< 12) [24]. These observations suggest major obstacles as far as obtaining speed-up for expert systems from parallel processing.

5.2 Obtaining Speed-up via Parallel Processing is Difficult.

Observation *A* (presented in section 5.1) suggests that the inherent parallelism available in expert systems is small. Observation *B* further suggests that:

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

ii.) 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,

iii.) there is no reason to expect that larger production systems will exhibit more speed-up from parallelism.

Observation *C* suggests that scheduling is critical towards obtaining whatever (small) speed-up 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/optimization on expert systems cannot be performed effectively because their 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, amount of storage/communication requirements) is highly data dependent. Compile-time optimization techniques cannot be applied directly to such computations.

Synchronizations take place frequently in search problems. At the heart of many expert systems is a heuristic search problem: *given an initial state, apply knowledge to prune the search tree to arrive at the goal state.* This 2-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 [25]: 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.
- *And-parallelism* in Prolog: Common terms which occur in two clauses being worked on simultaneously must be share identical bindings. This requires tasks working concurrently on two sub-goals to communicate whenever such bindings are changed.

6. Conclusions

Many "building blocks" developed to enable parallel execution of expert systems have been surveyed in sections 3 and 4. Measurement results presented in section 5, however, seem to indicate that the inherent parallelism available in expert systems is small. Can expert system be formulation as highly parallel computations? Can these "building blocks" be put together effectively to support parallel computations? We do not have answers to these important questions. However, we would like to draw on some fundamental results concerning speed-up and parallel processing in section 6.1 and put forth some "fruit for thought" regarding future directions for research in section 6.2.

6.1 Speed-up and Parallel Processing

A small section of sequential code in an application can significantly limit its speed-up. Recall Amdahl's Law which states that the maximum speed-up S for a computation obtainable on a multiprocessor with **p** processors is governed by:

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

where **f** is the fraction of the computation that has to be executed sequentially. A simple application of this result suggests that parallel RETE-match algorithms can give at most a 10-fold improvement because only speeds-up the *match* phase which takes 90% of the execution time is affected.

When partitioning a single application into tasks, the *grain-size* of the tasks should be chosen such that: (i.) there is enough parallelism to exercise the PEs of the parallel processor and (ii.) communication and process management overhead must not outweigh the speed-up 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 [23]. Minimizing the scheduling overhead for such fine grain tasks is a major obstacle for achieving higher degree of speed-up.

A number of effective software organization structures have been proposed for multiprocessors. These include *software pipelines*, *systolic algorithms*, *divide-and-conquer* (*tree-of-processes*), and *relaxed* or *asynchronous processes* [26]. Speed-up could only be obtained, however, if certain criteria are met for each proposed organization. For example, *temporally decomposable* computations can also be arranged as software pipelines (which process data items incrementally from one stage to another). Processing at each stage may be carried out concurrently if data items can be *spatially decomposed* into (relatively independent) subsets. With *divide-and-conquer*, maximum speed-up is obtained when:

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

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

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

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. For example, KATE is an expert system for controlling the flow of conditioned air to maintain required temperatures, pressure and humidity levels within four compartments of the Space Shuttle while it resides in the Orbiter Modification and Refurbishment Facility at Kennedy Space Center. Parallelism can only be extracted by rethinking the problems KATE is trying to solve:

- monitoring sensors — data from different sensors can be processed in parallel;
- problem diagnosis — multiple fault theories and consistency checks can be pursued in parallel;
- control — alternative methods (i.e. set of commands required) for attaining a desired goal can be pursued in parallel; and
- multiple faults and complex control operations are spatially decomposable.

Researchers at the Intelligent Systems Technology Branch, Information Sciences Division of NASA's Ames Research Center are working towards a parallel version of KATE based on these dimensions of parallelism. Results should be available for publication next year.

6.2 Conclusions

WHAT IS THE BEST STRATEGY FOR BUILDING PARALLEL EXPERT SYSTEMS? Should we:

i.) *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)? or

ii.) *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)? or

iii.) *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 have an answer to this question yet. Nevertheless, we would like to suggest some research directions which seem most promising to us.

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 knowledge-based application, before choosing *concurrent objects* vs. parallel Lisp, or shared-memory vs. distributed-memory architectures. In many cases, speeding up the "knowledge-based" portion itself may not produce the overall speed-up value we require. Bottom-up approaches produce machines that *could* exhibit orders of magnitude speed-up if *suitable* applications can be found.

The most efficient parallel execution model for expert systems may not look and work anything like the way they are specified. AI programming paradigms (whether it be *knowledge sources* with *blackboards* or *productions* on *working memory*) are designed to enable knowledge to be encoded and processed in a way similar to that carried out by human beings. They are not necessarily efficient for execution on a 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.

In conclusion, we suggests that speed-up cannot come from parallelizing one particular existent paradigm or language or operating system. We must:

(i.) **understand how to break up the problem with minimal contention for accessing shared resources and reduced dependencies**; this could probably come about by considering (*macro* and *micro*) data dependencies in the system when designing its parallel implementation;

(ii.) **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

(ii.) **explore parallelism at the application level**; the nature of the application may suggest temporal or spacial decompositions; do not the portion of the application that is not *knowledge-based* (e.g. re-organizing the I/O procedures may save more time than merely replacing the sequential inference engine with a parallel one).

References

1. David H. Warren, "An Abstract Prolog Instruction Set", Technical Note 309, Artificial Intelligence Center, SRI International, October 1983.

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

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

4. Salvatore J. Stolfo and Daniel P. Miranker, "DADO: A Parallel Processor for Expert Systems", In IEEE, pp. 74 - 82, 1984.

5. Scott E. Fahlman, "Design Sketch For a Million-Element NETL Machine", Carnegie-Mellon University, Department of Computer Science, In AAAI-80, August 80.

6. James A. Hendler, *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, New Jersey, 1988.

7. W. Daniel Hillis, *The Connection Machine*, The MIT Press, Cambridge, MA, 1985.

8. Atsuhiro Goto, "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 89.

9. K. Fuchi (ICOT, Japan) and M. Nivat (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.

10. Richard P. Gabriel and John McCarthy, "Queue-Based MultiProcessor Lisp", Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, ACM, Austin, Texas, August 84.

11. Robert H. Halstead, Jr., "Parallel Symbolic Computer", Computer Magazine, 19:8, pp. 35 - 43, August 86.

12. R. Halstead, T. Anderson, R. Osborne, and T. Sterling, "Concept: Design of a Multiprocessor Development System", 13th International Symposium on Computer Architecture, Tokyo, pp. 40 - 48, June 86.

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

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

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

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

17. M. Hermenegildo and Evan Tick, "Memory Performance of AND-parallel Prolog on Shared-Memory Architectures", Proceedings of the 1988 International Conference on Parallel Processing, August 15 - 19, 1988, Volume II Software, pp. 17-21, August 88.

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

19. Bruce A. Delagi, Nakul P. Saraiya and Gregory T. Byrd, "LAMINA: CARE Applications Interface", Knowledge Systems Laboratory, Report Number KSL 86-67, Computer Science Department, Stanford University, November 87.

20. H. Penny Nii, Nelleke Aiello and James Rice, "Frameworks for Concurrent Problem Solving: A Report on Cage and Poligon", Knowledge Systems Laboratory, Report Number KSL 88-02, Computer Science Department, Stanford University, February 88.

21. James P. Rice, "Problems with Problem-Solving in Parallel: The Poligon System 1.0", Knowledge Systems Laboratory, Report Number KSL 88-04, Computer Science Department, Stanford University, January 88.

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

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

24. Leon Alkalaj, "Architectural Support for Concurrent Logic Programming Languages", PhD Thesis, Computer Science Department, University of California, Los Angeles, August 1989.

25. John E. Laird, *Soar User's Manual*, 4th Eidition, Xerox PARC, 1986.

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