Towards A Virtual Parallel Inference Engine

Howard E. Shrobe, John G. Aspinall and Neil L. Mayle
Symbolics Cambridge Research Center
11 Cambridge Center, Cambridge, MA 02142
Howard Shrobe is also a Principal Research Scientist
at the MIT Artificial Intelligence Laboratory.

Abstract

Parallel processing systems offer a major improvement in capabilities to AI programmers. However, at the moment, all such systems require the programmer to manage the control of parallelism explicitly, leading to an unfortunate intermixing of knowledge-level and control-level information. Furthermore, parallel processing systems differ radically, making a control regime that is effective in one environment less so in another. We present a means for overcoming these problems within a unifying framework in which 1) Knowledge level information can be expressed effectively 2) Information regarding the control of parallelism can be factored out and 3) Different regimes of parallelism can be efficiently supported without modification of the knowledge-level information. The *Protocol of Inference* introduced in [Rowley *et al.*, 1987] forms the basis for our approach.

1 Introduction

Even though there are a variety of parallel computers now in existence, using parallelism to accelerate AI programs remains a difficult art. One major problem which must be addressed is that there are a variety of different parallel processing environments and these differ markedly [Hillis, 1981; Stolfo, 1982; Davis, 1985; Forgy *et al.*, 1984; Singh, 1985]. Even when we fix our attention on a single, general purpose hardware framework (for example, shared memory multiprocessors) and a single style of computational task (such as rule-based inference) there is a wide diversity of critical parameters that determine how much parallelism of what grain-size the programmer should try to obtain. These parameters include:

1. The cost of initiating a new task or process.
2. The cost of maintaining locks and other facilities for mutual exclusion.
3. The cost of switching tasks.
4. The number of processors.
5. The bandwidth and latency of the communication path connecting the processors.

Variations in these parameters lead to quite different strategies for optimizing a parallel program. For example, when there are a large number of processors and the cost of initiating a new task is low, the obvious strategy is to create as many fine-grained tasks as possible. On the other hand, a higher cost of task initiation leads one to pick a larger grain-size for tasks, so that the useful work done in a task dominates the overhead of initializing its data structures and scheduling it for execution. Similarly, if the system provides only expensive means for mutual exclusion, then one might be inclined to aim for a strategy that employs fewer locks but which also leads to less parallelism.

To be concrete, in the environment of our research (which consists of Symbolics 3600 processors:

1. Task initiation requires a minimum of 45 microseconds for "light weight" processes.
2. A simple lock requires roughly 30 microseconds to be seized and freed.
3. Switching tasks requires in excess of 100 microseconds.
4. In an experimental multiprocessor under development in our laboratory there are between 8 and 16 processors.
5. The bandwidth of the bus connecting these processors is over 100 Megabytes per second and the latency is under 100 nsec.

It is clear that, given these specific parameters, one should not try to create a parallel task whose execution time is smaller than 45 microseconds since what is saved by parallel execution is lost in task initiation. However, there are algorithms of interest for parallel AI systems that contain such tasks. For example, we have studied the Rete [Forgy, 1982] algorithm in some detail and our metering reveals that one type of significant step (the two input merge step) takes less than 16 microseconds on average for certain benchmark programs. Attempting to reduce the grain size of parallelism to this level is, therefore, fruitless. The distribution of time consumed in this step is bimodal (there is one peak for successful merges and a second peak for failed attempts) and the relative weighting of the two peaks is application specific.

A different implementation of the Rete algorithm which used this smaller grain-size would be warranted in an environment with cheaper initiation or if the application had a different profile. In summary, the interaction between the knowledge-level task and the system environment (hardware and core system software) dictates the appropriate strategy for introducing parallelism.

It is desirable therefore to build applications using a *Virtual Parallel Inference Engine*, an AI programming system that allows the programmer to separate the knowledge-level description of the problem from the machine-specific control-level description and that allows the same knowledge-level description to effectively execute in a variety of different system environments without modifying the knowledge-level description.

In the rest of this paper we proceed as follows:

1. First we review the notion of a Protocol Of Inference, first introduced in the Joshua system [Rowley *et al.*, 1987], and show how it can serve as a Virtual Parallel Inference Engine.
2. We then look at how the protocol can implement forward and backward rule-based paradigms in both loosely and closely coupled system environments.
3. We describe two simulation environments implemented on the Symbolics 3600, which allow us to study the effectiveness of different approaches to parallelism under a variety of different assumptions about the nature of the parallel processing environment. We present results from these environments.
4. Finally, we summarize and examine weaknesses of our approach, pointing out directions for further work.

2 The Solution: The Protocol of Inference

Our solution is based on the notion of a *Protocol of Inference* introduced in the *Joshua* language [Rowley *et al.*, 1987]. An Inferential Protocol organizes the work of the AI language around the manipulation of a virtual database of assertions, whose interface is specified functionally by the two *generic functions*, ASK and TELL. TELL inserts an assertion into the database and triggers antecedent inferential processes (e.g. forward chaining rules). ASK queries the database, triggering consequent inferential processing (e.g. backward-chaining rules or interrogations of the user). ASK takes two arguments, the first is the QUERY; the second, called the CONTINUATION, is a procedure that is invoked once for every assertion or rule that matches the query.

We refer to the collection of assertions as a *virtual database*. This is because the interface to it is syntactically uniform and consists of only of the two functions ASK and TELL. However, every significant step involved in processing a statement (e.g. ASK, TELL, and all their subroutines) is implemented as a generic function, i.e. by a procedure whose behavior is determined by the datatypes of its arguments. This allows the programmer to provide diverse implementations of ASK and TELL, each customized to a particular type of statement.

In the Joshua Protocol of Inference, statements are implemented as INSTANCES whose CLASS corresponds to the predicate of the statement. The behavior of the generic functions such as ASK and TELL consists a invoking a METHOD (e.g. the ASK or TELL method) associated with the CLASS of the statement. METHODS for any step of the protocol may be defined for any predicate, thereby customizing the behavior of any aspect of the database of assertions.

2.1 Description of the Protocol

The fine-grained structure of ASK, TELL and the RULE-COMPILER are part of the protocol and a small customization of behavior can be effected by providing a small method for a lower level protocol step. Since the object-oriented programming paradigm used (e.g. the Common Lisp Object System or New Flavors) in Joshua supports a multiple inheritance class hierarchy, *mixins* of implementation behavior can be predefined and then inherited by any predicate which can exploit its capabilities.

We will show how the use of the Protocol of Inference can allow a programmer to exploit the parallelism available in a particular system environment. Our strategy is to identify protocol steps whose implementation allows parallel execution and then to provide predefined *predicate-mixins* that exploit this opportunity. Predefined mixins can be provided to support different styles and degrees of parallelism; when a system environment is decided on the programmer simply includes the appropriate mixins in the predicate definitions.

The Protocol allows a clean separation between the knowledge-level description of the task and the control-level description used to exploit parallelism. The knowledge is captured in rules, statements and the routines that manipulate these. The control is specified in protocol methods which specify how to capture parallelism. Ideally, the programmer should be able to use a ready made protocol mixin suitable for his particular system environment; porting the application to a different system environment should involve no more work than mixing in a different set of methods which are more tailored to the new environment.

Because of this separation of concerns, Joshua constitutes the *Virtual Parallel Inference Engine* we seek.

3 Providing a Plethora of Parallel Processing Paradigms

In this section we will briefly describe how the protocol of inference can be used to support distinct paradigms of parallel processing. We have chosen the following four examples merely to exhibit a spread of styles in both hardware environment and software technique. The implementations described are intentionally naive and oversimplified due to space limitations.

3.1 Loosely Coupled Systems

In the next two sections we will assume a system environment consisting of an ensemble of conventional uniprocessors connected by an Ethernet. This allows modest scale parallelism with a very high cost for initiating tasks due to the long latency of the network.

3.1.1 Loosely Coupled Forward Chaining

There are AI tasks which can exploit such a structure. For example, in distributed blackboard systems the information is decomposed into several discrete levels (e.g. signal, feature, interpretation). Similarly, most of the knowledge sources manipulate information stored in only one such level. It therefore makes sense to localize a blackboard level and its associated knowledge sources to a particular machine.

The distributed blackboard is implemented as a forward-chaining rule-based system distributed across the ensemble of machines; statements that occur at a particular level of the blackboard are statically assigned to a unique machine as is each rule that manipulates information at that level. Other systems that exhibit a similar high degree of decomposability can also exploit such an architecture.

Implementing such a system using the protocol is straightforward and involves developing methods for the following protocol steps:

- INSERT
- LOCATE-FORWARD-TRIGGER
- MAP-OVER-FORWARD-TRIGGERS

INSERT is the subroutine of TELL which decides where an assertion should be stored. In the uniprocessor world, INSERT serves as the hook to create data specific indexers; in the parallel processing world, it is also responsible for deciding which machine an assertion should be stored on.

We implement two INSERT mixins. The first of these contains the normal INSERT method that stores the assertion on the local machine. The second contains an INSERT method that forwards a request to the remote machine where the assertion should be stored. Each predicate definition is specified separately for each machine; if the predicate is actually stored on that machine, the first mixin is used. Otherwise the second one is employed.

Rules are handled analogously. LOCATE-FORWARD-TRIGGER is the method responsible for indexing the Rete network nodes used to trigger forward rules. MAP-OVER-FORWARD-TRIGGERS is the subroutine of TELL that is responsible for finding these triggers and invoking the forward chaining rules triggered by an assertion. As with the data indexing methods we implement two versions of this method. For each machine, we mix in the first version if the rule is stored locally and the second version otherwise.

This approach is a minor modification of the techniques used for sequential programs. The major difference is that some work is distributed across the network to remote machines. The protocol allows us to do this simply by mixing in the appropriate methods.

3.1.2 Loosely Coupled Backward Chaining

A backward chaining system can exploit this loosely coupled environment if it can be decomposed into nearly independent modules of expertise. Each of these modules runs independently, but when it needs services from a remote specialist, it must send a request through the network.

Such a Loosely coupled backward chaining system is also easy to capture within our protocol. As in the previous model, we assign the statements and the backward-chaining rules associated with a particular packet of expertise to a particular machine.

The new protocol methods involved in this approach are:

- LOCATE-BACKWARD-TRIGGER
- MAP-OVER-BACKWARD-TRIGGERS

To gain parallelism, we modify the ASK-RULE part of the ASK protocol to undertake two operations in parallel. The first is the processing of those rules stored locally. The second is the sending of the query to each remote machine containing relevant rules. This behavior is captured in the MAP-OVER-BACKWARD-TRIGGERS protocol step which is a subroutine of ASK-RULE.

LOCATE-BACKWARD-TRIGGER is the routine that indexes backward-chaining rule triggers; this method decides which machine should store each rule and either indexes the rule-trigger locally or sends a message to the remote machine requesting it to do so.

When a rule has satisfied a query it must call the continuation of the query to process the result. If the query had been posted in a remote machine, this involves sending a variable binding environment across the network to the requesting machine.

3.1.3 Comments on the Approach

Sending a message through the network is very slow by comparison to the time it takes to implement any protocol step on a local machine. Thus, this approach only gains performance if: 1) There is a natural coarse-grained partitioning of the problem and 2) The processing performed in response to remotely triggering a rule is quite large. This will be true if the body of the rule invokes a major computation, or if it triggers a large number of rule firings localized to the remote machine. Failing this, the approach will lead to overall system degradation rather than speedup. There is no inherent reason for assigning a rule to a specific machine. In [Singh, 1985] rules are replicated in each machine, allowing any machine to apply a rule immediately if it isn't busy. Otherwise it broadcasts the goal, providing work for other machines.

The implementation of the backward chaining system must be careful about its treatment of logic-variable bindings since the standard shallow-binding scheme used in Prolog is incompatible with the Or-parallelism introduced here; however, space does not allow us to discuss this in detail here.

3.2 Closely Coupled Systems

In the next two sections we will assume a hardware environment consisting of a shared-memory multiprocessor. This greatly reduces the cost of sending a task to a remote processor since this involves only adding and removing entries from a task queue. This allows much greater opportunity for parallelism and for smaller grain-sized tasks.

3.2.1 Closely Coupled Backward Chaining

Our model for backward-chaining in this environment achieves both *and-parallelism* and *or-parallelism* and is similar to [Singh, 1986]. For each backward-chaining rule, we create a Rete network whose initial nodes correspond to the subgoals in the IF part of a backward-chaining rule. After the rule is triggered and the THEN part of the rule has been matched to the query, a task is created for each of the subgoals in the IF part of the rule. Each task instantiates its subgoal with the variable bindings of the match and then posts a query for solutions to the instantiated subgoal.

Since the only variables instantiated in the posted subgoals are those of the THEN part of the rule, it is possible to receive solutions to two of the subgoals that inconsistently instantiate the other variables. This is the point of the Rete network. A solution to a particular subgoal is sent to the corresponding node of the Rete network; the Rete algorithm then finds all sets of mutually consistent solutions to the subgoals. The continuation of the query is called for each solution that emerges from the terminal node of the Rete network, producing the *and* parallelism.

Implementing this model involves use of the following protocol steps:

- MAP-OVER-BACKWARD-TRIGGERS
- COMPILE-BACKWARD-ACTION

The first of these is modified to trigger the relevant rules in parallel.

The second of these methods is used to customize how the rule-compiler treats the IF part (or the right-hand side) of the rule. It builds the Rete network and emits code for each subgoal which posts the instantiated query for the subgoal and feeds each solution to the appropriate Rete network node.

3.2.2 Closely Coupled Forward Chaining

This approach has been discussed widely in the literature [Okuno & Gupta, 1988; Stolfo, 1982], particularly in the context of OPS-5 implementations. OPS-5 imposes a sequential bottleneck in order to perform the *conflict resolution* step. We remove this restriction in our model.

The IF part of Forward chaining rules is normally compiled into a Rete network. The THEN part of the rule is normally compiled into a sequence of TELL statements, one for each pattern. Parallelism can be achieved by compiling the THEN part into parallel TELL statements. In addition, the Rete network implementation can introduce parallelism by creating separate tasks to handle the processing of individual statements. Further parallelism can be introduced by creating separate tasks to handle the substeps of processing an individual statement. The relevant protocol steps are:

- MAP-OVER-FORWARD-TRIGGERS
- COMPILE-FORWARD-TRIGGER
- COMPILE-FORWARD-ACTION

The first of these is the run-time routine used to fetch relevant rules when a statement is asserted; this protocol step introduces the opportunity to exploit parallelism within the process of rule lookup. The second two methods are called by the rule compiler during the compilation of forward rules. COMPILE-FORWARD-TRIGGER is the interface to the part of the compiler that builds the Rete network. The last method controls how the THEN part of a forward-chaining rules is compiled; it provides the opportunity to make the actions on the right hand side execute in parallel.

3.2.3 Comments on the Approach

The approach requires a shared memory multiprocessor in which separate processes share address space. This approach can lead to lots of parallelism, particularly if the Rete network implementation is designed to maximize parallelism. However, as we stated in the introduction, one must be careful. If the grain size of a task is reduced to the point where its startup cost is comparable to its execution time, nothing is gained; as we mentioned earlier, some of the primitive steps of the Rete algorithm exhibit this problem. In addition, parallelism in the Rete network requires us to enforce mutual exclusion in critical regions. The cost of locking may be high and should be carefully considered. Okuno and Gupta [1988] describe a parallel OPS-5 implementation with a parallel Rete algorithm similar to ours.

These concerns make a simulation environment extremely valuable to help understand how a particular system environment matches a particular detailed approach to parallelism.

4 The Simulation Environments

A simple simulator of Multilisp has been written that allows us to investigate questions about the maximum available parallelism in Lisp applications. [1]

In addition, a second type of simulation environment is provided by the multiprocessing capability of the Symbolics Genera operating system. In this environment, multiple processors sharing a single memory are straightforwardly simulated by separate processes running on a uniprocessor.

4.1 Simulating Varying Degrees of Parallelism

The Multilisp simulator runs in a single process and consists of two parts. The first part simulates the execution as if every future that is created immediately finds a processor available to run it. This shows the maximal amount of parallelism available to the program.

The simulator works by keeping track of an imaginary "simulation clock". Between requests to create and evaluate futures (that is when normal, serial, Lisp is running) the simulation clock simply tracks the normal process time. When futures are created, the creation time of the future is recorded in the future. When the future is run, the simulation clock is "backed up" to the creation time of the future, and advances from that point. The use of resources for which there may be contention is also recorded by noting the time periods for which a resource is "locked" against competing users.

The result of the simulation is a history, which has the structure of a graph. Each arc in the history corresponds to the serial execution of some piece of Lisp. A grapher tool allows us to display the history in graphical form, and extract certain statistics (e.g. processor utilization, speedup over serial execution).

The history of program execution contains sufficient information to construct other histories under conditions of limited parallelism. This is what the second part of the simulator does. The re-simulator takes a maximally parallel history, a number of processors, and an argument describing the scheduling policy. It performs an event-driven simulation and returns another history which represents the execution of the same program under those new conditions.

4.2 Simulation by Multiprocessing

Simulation by multiprocessing is done by writing programs using the normal techniques of scheduling and contention avoidance, and then creating several processes (each emulating one processor in a multiprocessor) which then look for tasks in a shared queue. Each process can then be metered separately using standard tools.

[1] Strictly speaking, the language we simulate is not Multilisp, which is based on Scheme, but an equivalent language based on adding Multilisp constructs to Common Lisp. The distinction is basically a syntactic one and unimportant to the investigation.

4.3 Comparison of the Two Techniques

Each of the two simulation techniques has advantages over the other. The Multilisp simulator has the advantage that one simulation running real code can be resimulated under varying conditions in a controlled fashion. Metering the original simulation is easy, since it happens in one process, and resimulation is free of variations induced by scheduling policy, paging overhead, and the like. On the negative side, it is easy to perform a resimulation that does not obey causality constraints. If lisp forms ever produce side effects that will be seen by another future, it is mandatory to time-stamp such values. It is easy to overlook such side effects and so care must be taken to ensure that results reflect some potential version of reality.

Simulation by multiprocessing is a much closer simulation of the reality of a shared-memory multiprocessor. Effectively, the operating system is performing fine-grained time-slicing where in the Multilisp simulator above, the resimulator performed coarse grained time-slicing. Causality effects are almost completely eliminated because of the fine-grained time-slicing, and shared resource contention must be handled properly or else the program will produce wrong results which are immediately apparent.

The two techniques are complementary.

5 Simulation of Parallel Rete Networks

Rete networks are an important technique for achieving parallelism in both our models of closely coupled parallelism; they have been studied [Gupta, 1984; Okuno & Gupta, 1988] in the context of OPS5 execution model. Conflict Resolution is an important part of the control structure of the OPS5 model but it is a bottleneck that limits available parallelism; for tasks that merely compute the deductive closure of an initial set of facts (theorem proving or simulation) Conflict Resolution is unnecessary (since rule execution order is irrelevant). Our studies involve programs for which Conflict Resolution is an artificial bottleneck, in particular, a rule-based circuit simulator.

5.1 Parallelism in the Rete network

Joshua uses a standard Rete network consisting of match and merge nodes. The nodes store states that hold consistent sets of variable bindings. As matching/merging proceeds states propagate through the Rete network. The need for mutual exclusion arises if a new state reaches each of the parents of a merge node at the same time. It is necessary that only one of the tasks merges the two new states by employing some form of mutual exclusion.

We have studied a relatively fine grain locking scheme that allows more parallelism. The exact steps that occur when a new state comes in to a parent node are as follows:

1. Grab the lock of merge node directly below.
2. Push the new state into the state list at the parent.
3. Grab a pointer to the head of the brother node's state list.
4. Unlock the lock
5. Do merges with states in the previously grabbed state list.

5.2 Task size

Within the matching/merging process there are many different ways to break up the work into separate parallelizable tasks. Here is a partial list ordered by decreasing task size:

1. Each individual firing of a rule spawns a separate task in which the all the work associated with body of the rule gets executed.
2. A rule body may do several TELLs; each of these can be spawned as a separate task. Each task handles all of the TELL protocol - both the data indexing and the rule indexing (matching/merging).
3. Each state created by a rete node (match or merge) can be spawned as a task. All of the merges of the state with other states (in brother nodes) happens within this task.
4. Each individual merge operation can be spawned as a task. Every merge between two states happens in its own task.

Our metering tools show that the last of these is below the threshold for successful parallel execution. Figure 1 shows the processor utilization charts resulting from simulating the parallel execution of a rule-based circuit simulator using the first three of the above options.

As can be seen, speedup continues up to 32 processors although cost effectiveness decreases somewhere between 8 and 16 processors. Larger simulations would effectively utilize more processors.

6 Conclusions

Our initial explorations suggest that the Joshua Protocol of Inference can be effectively used to build a *Virtual Parallel Inference Engine*. It cleanly separates the control of parallelism from the expression of task knowledge and allows the same rule-base to be executed in both sequential and parallel environments without modification. Furthermore, it allows the same rule base to be executed in a variety of different parallel environments, tailoring the strategy to the detailed nature of the system. Again this does not involve modifying the knowledge-level structures.

However, there are still many difficult problems to be confronted. So far we have conducted limited studies of programs which can be correctly executed without enforcing ordering constraints between the rules. There are natural classes of problems (such as deductive closure and simulation) where this is allowable. But there are many problems for which this is not true and some control must be placed over rule execution. The conflict resolution step of OPS-5 imposes a serial bottleneck after every rule's execution, artificially limiting the ability to exploit parallelism. We are searching for other control techniques that are more explicit and less limiting.

We also recognize that our descriptions of the use of the Protocol to implement parallelism are limited and naive. There are obviously many other ways to capture parallelism. We believe that our approach has one distinct advantage, namely its ability to include and experiment with any new technique that arises.

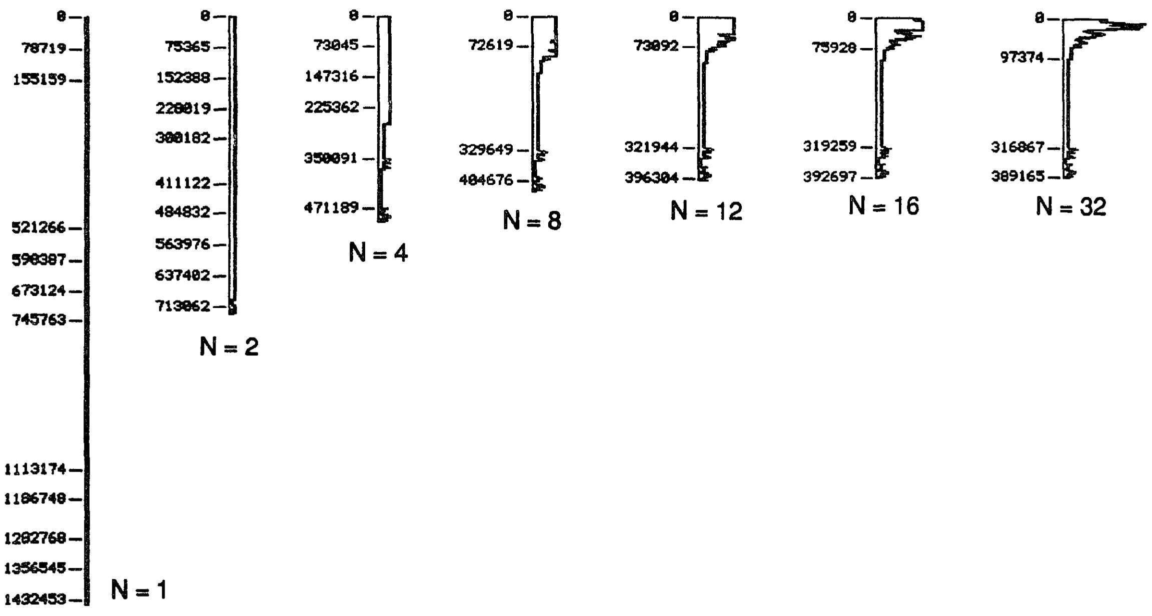

Figure 1: Processor utilization as a function of N, the number of processors available, while performing a rule-based circuit simulation. Time increases downward; the graphs show the number of processors active as a function of elapsed time.

Acknowledgements

This work was supported by NASA contract NAS2-12616.

References

[Davis, 1985] A. Davis and S. Robison. The Architecture of the FAIM-1 Symbolic Multiprocessing System. In *Proceedings IJCAI-85*, pages 32–38. International Joint Committee for Artificial Intelligence, Los Angeles, August 1985.

[Forgy *et al.*, 1984] C. Forgy, A. Gupta, A. Newell and R. Wedig. Initial Assesment of Architectures for Production Systems. In *Proceedings AAAI-84*, pages 116–120. American Association for Artificial Intelligence, Austin Texas, August 1984.

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

[Gupta & Forgy, 1983] A. Gupta and C. L. Forgy. Measurements on Production Systems. Carnegie-Mellon University, 1983.

[Gupta, 1984] A. Gupta. Implementing OPS-5 Production Systems on DADO. *International Conference on Parallel Processing*, August 1984.

[Hillis, 1981] W. D. Hillis. *The Connection Machine* MIT AI Laboratory T.R. 646, Cambridge Mass, 1981.

[Okuno & Gupta, 1988] H.G. Okuno and A. Gupta. High-Level Language Approach to Parallel Execution of OPS-5 *Proceedings of the Fourth IEEE Conference on Artificial Intelligence Applications*, March 1988, San Diego.

[Rowley *et al.*, 1987] S. Rowley, H. Shrobe, R. Cassels and W. Hamscher. Joshua: Uniform Access to Heterogenous Knowledge Structures, or Why Joshing is Better than Conniving or Planning. In *Proceedings AAAI-87*, pages 48–52. American Association for Artificial Intelligence, Seattle, July 1987.

[Singh, 1985] V. Singh and M. Genesereth. A Variable Supply Model for Distributing Deductions. In *Proceedings IJCAI-85*, Vol. 1, pages 39-45. International Joint Committee for Artificial Intelligence, Los Angeles, August 1985.

[Singh, 1986] V. Singh and M. Genesereth. PM: A Parallel Execution Model for Backward-Chaining Deductions. Stanford Knowledge Systems Laboratory, Report No. KSL-85-18, Stanford CA, June 1986

[Stolfo, 1982] S. Stolfo and D. Shaw. DADO: A Tree-Structured Machine Architecture for Production Systems. In *Proceedings AAAI-82*, pages 242–246. American Association for Artificial Intelligence, August 1982, Pittsburg.