CUCS-77-83

ON THE DESIGN OF PARALLEL PRODUCTION SYSTEM MACHINES: WHAT'S IN A LIP?

SALVATORE J. STOLFO

On the Design of Parallel Production System Machines: What's In a LIP?*

Salvatore J. Stolfo
Department of Computer Science
Columbia University
New York City, N. Y. 10027

Abstract

In a general manner we discuss the appropriateness of three classes of parallel computers for one AI programming methodology, Production Systems. The three classes considered are coarse-grain, fine-grain and very fine-grain parallel processors. We conjecture that each class is suitable for different Production System formalisms. Using current technology, we further project execution rates for two of these classes in the range of 10,000 Working Memory transactions per second.

*This research has been supported by the Defense Advanced Research Projects Agency through contract N00039-84-C-0165, as well as grants from Intel, Digital Equipment, Hewlett-Packard, Valid Logic Systems, AT&T Bell Laboratories and IBM Corporations and the New York State Science and Technology Foundation. We gratefully acknowledge their support.

Table of Contents

1 Introduction

A considerable amount of interest has been generated recently in specialized machine architectures designed for the very rapid execution of Artificial Intelligence (AI) software. The Japanese Fifth Generation Machine Project, for example, promises to deliver over the next decade a functioning device capable of computing solutions of large PROLOG programs at execution rates in the hundreds of thousands to perhaps millions of *logical inferences per second* (LIPS). Such a device will require high-speed hardware executing a large number of primitive symbol manipulation tasks many times faster than today's fastest computers. This rather ambitious goal has led some researchers to suspect that a fundamentally different computer organization is necessary to achieve this performance. Thus, *parallel processing* has assumed an important position in current AI research.

Several architectures have been proposed comprising a number of concurrent processors cooperatively executing symbol manipulation tasks. These proposed machines cover a wide spectrum of design issues. In this brief note we concentrate on two such issues: the *number* of concurrent processors employed in the device and their *complexity* (relative functionality and storage capacity).

In a general manner we shall discuss the appropriateness of three classes of parallel computers for one AI programming methodology, *production systems*. The three classes we shall consider are:

- *coarse-grain* devices based on one hundred very powerful (asynchronous) processors with large local memories, in the range of one megabyte of storage capacity.

- *fine-grain* devices based on ten thousand simpler (either synchronous or asynchronous) processors each with small local memories, in the range of ten thousand bytes.

- *very fine-grain* devices based on one million very weak (synchronous) processors with tiny local memories, in the range of one hundred bytes.

Issues related to interconnection topology, shared versus local memory, cost, wirability, size and breadth of application will not be discussed in this paper although such issues are central factors in the design of a computing device. Rather, we shall consider only the suitability of each general class of device to particular types of production system programs.

Production system efficiency is typically measured in terms of the number of recognize/act cycles executed per second. Unfortunately, this measure is inappropriate when considering different possible production system formalisms. For example, some production systems permit a rather small number of actions on each cycle, whereas other systems may permit many thousands of actions. In order to compare the performance of different devices while executing various inference procedures, we have chosen to use the measure of LIPS, typically used when comparing logic-based systems.

Although production systems are generally not thought of as logic-based programming systems, such as PROLOG, the similarity of the two models of computation is sufficient to be able to make the same general comments about each. We intend to elucidate the meaning of the ill-defined term LIPS in this context. It is our conjecture that, when suitably constrained in meaning, as we propose here, for at least two of the classes, tens of thousands of LIPS is indeed achievable now. This gives us confidence that hundreds of thousands of LIPS will be achievable in the coming decade as hardware advances will undoubtedly continue to develop.

We begin with a description of the general production system paradigm in question.

2 Production Systems

In general, a *Production System* (PS) [Newell 1973, Davis and King 1975, Rychener 1976, Forgy 1982] is defined by a set of rules, or *productions*, which form the *Production Memory* (PM), together with a database of assertions, called the *Working Memory* (WM). Each production consists of a conjunction of *pattern elements*, called the *left-hand side* (LHS) of the rule, along with a set of actions called the *right-hand side* (RHS). The RHS specifies information that is to be added to (asserted) or removed from WM when the LHS successfully matches against the contents of WM.

Pattern elements in the LHS may have a variety of forms which are dependent on the form and content of WM elements. In the simplest case, patterns are lists composed of constants, variables (prefixed with an equals sign) or embedded sublists of pattern elements. An example production, borrowed from the blocks world, is illustrated in figure 1.

Figure 1: An Example Production.

```
(Goal (Clear-top-of Block))
(Isa =x Block)
(On-top-of =y =x)
(Isa =y Block)  -->
                    delete(On-top-of =y =x)
                    assert(On-top-of =y Table)

If the goal is to clear the top of a block,
  and there is a block (=x)
  covered by something (=y)
  which is also a block,
            then
                remove the fact that =y is on =x from WM
                and assert that =y is on top of the table.
```

In operation, the production system repeatedly executes the following cycle of operations:

1. *Match*: For each rule, determine whether the LHS matches the current environment of WM: each pattern element is matched by some WM element with variables consistently bound throughout the LHS. All matching instances of the rules are collected in the *conflict set of rules*.

2. *Select*: Choose exactly one of the matching rules according to some predefined criterion.

3. *Act*: Add to or delete from WM all assertions specified in the RHS of the selected rule or perform some operation.

During the selection phase of production system execution, a typical interpreter provides *conflict resolution strategies* based on the *recency* of matched data in WM, as well as syntactic discrimination.

Other resolution schemes are possible, but for the present paper such issues will not significantly change our analysis, and hence will not be discussed.

We shall only consider the parallel execution of PS programs with very large rule bases and WM's, say ten thousand productions and one thousand WM elements, with the goal of accelerating the rule firing rate of the recognize/act cycle as well as the number of WM transactions performed. We shall not consider other possible parallel activities as, for example, the concurrent execution of multiple PS programs.

On first glance it appears that each phase of the PS cycle is suitable for direct execution on parallel hardware, with the greatest opportunity for a speed-up in the match phase. (Indeed, Forgy [1979] notes that some PS interpreters spend over 90% of their time executing the match phase of operation.) This requires a partitioning of PM and WM among the available processors: some subset of processors would store and process the LHS of rules, while another possibly intersecting subset of processors would store and process WM elements. Thus, we envisage a set of processors concurrently executing pattern matching tests for a number of rules assigned to them. Similarly, once a conflict set of rules is formed, high-speed selection can be implemented in parallel as a logarithmic time algebraic operation. (For pedagogical reasons, we ignore the selection phase in our subsequent analysis.) Finally, the RHS of a rule can be processed by a parallel update of WM. We summarize this approach by the following abstract algorithm..

1. Assign some subset of rules to a set of (distinct) processors.

2. Assign some subset of WM elements to a set of processors (possibly distinct from those in step 1).

Repeat until no rule is active:

3. Broadcast an instruction to all processors storing rules to begin the match phase, resulting in the formation of a local conflict set of matching instances.

4. Considering each maximally rated instance within each processor, compute the maximally rated rule within the entire system. Report its instantiated RHS.

5. Broadcast the changes to WM reported in step 4 to all processors, which update their local WM accordingly.

end Repeat;

This very simple view of the parallel implementation of the PS cycle forms the basis of our subsequent analysis.

2.1 Constraining the PS formalism

There are many ways one can modify the general PS formalism. Each such constraint offers a number of advantages, as well as challenges, for a parallel computing device to efficiently execute the recognize/act cycle. These design modifications will be the parameters for our observations concerning the parallel implementations of production systems on various devices.

2.1.1 Temporal Redundancy and the RHS

In some formalisms the number of actions specified in the RHS of a rule is bounded by a small number. Thus, WM changes very little on each cycle, a fact which has been well exploited in systems such as OPS [Forgy 1982]. This *temporal redundancy* has led to the development of clever algorithms which efficiently link WM modifications with those rules which may be affected by such changes, thus reducing the effort in the match phase of execution.

In a less restricted formalism, a large number of changes to WM may be more appropriate for certain classes of PS applications. Thus, although the RHS of a rule may have a small number of action specifications, the resulting changes to WM may be quite large. It may be advantageous to have available a construct which deletes *all* WM data elements matching a certain specification. For example, delete all elements matching (red =x). Similarly, a constructor operator which creates a large number of data elements may also be useful. For example, for each WM element matching (red =x), create an element (edible =x).

As noted, the former approach leads to a more efficient match phase by focusing the effort on a smaller number of productions. In the latter case, however, it would seem that little opportunity exists to restrict the scope of the match operation.

2.1.2 Global WM Tests

Typical PS implementations restrict the form of pattern elements in the LHS to match only a single WM data element. Thus, pattern variables are usually considered *existentially quantified* and the resulting tests of WM are restricted to a small number of data elements. One can envisage more powerful pattern matching tests specified in the LHS of a rule, for example by including *universally quantified* pattern variables as well as tests which may compute some value based on the entire contents of WM. One may wish to specify a conditional test in the LHS of a rule which succeeds if *the number of WM elements matching some pattern is greater (less) than some value.* Note that in the temporally redundant case such tests can be computed only by repetitive application of a group of rules which calculate the desired result over many PS cycles.

2.1.3 Static WM data elements

WM data elements have also been restricted in many PS formalism. Thus, typically a given WM element must have the same form and content throughout its lifetime, i.e. it cannot contain variables which may be replaced later with more complex objects. This fact is exploited in systems such as OPS to efficiently compile a network of primitive pattern matching tests from the LHS of rules. Such primitive pattern matching tests include equivalence of constants, length of lists and consistency of bindings to variables.

In the more general case, for example logic-based formalisms such as PROLOG, WM elements are general first order literals, and thus may contain arbitrary first order terms. In this case pattern matching must be generalized to *logical unification*, rather than simple one way pattern matching.

2.2 Proposed definition of a LIP

The distinction between temporally redundant and non-temporally redundant PS formalisms, incidentally, may help to eradicate the confusion concerning the proper definition of "LIPS". A LIP may be defined as

a single change to WM (assertion or denial of a fact), or a single invocation of the RHS of a rule. In terms of logic-based programming systems, the distinction exists between the satisfaction or denial of a literal, or the satisfaction of a non-unit clause (which might require the satisfaction or denial of many literals).

In the case of temporally redundant PS programs, the difference is nearly indistinguishable: the invocation of a RHS asserts or denies a small number of WM elements. In the least restrictive case, however, a single rule invocation may lead to many thousands of WM changes. It is our preference to choose the former definition since the number of LIPS in question remains the same under both PS formalisms. Thus, in subsequent sections we consider a *single LIP to be an assertion or deletion of an individual WM element which represents a single fact.* This definition gives us confidence that devices executing tens of thousands of LIPS are indeed achievable using current technology.

3 Suitability of Three Classes of Parallel Computers

In parallel processing useful computation is performed, in general, by a number of processors calculating values which are subsequently communicated to adjacent processors for another phase of computation. Communication is relatively inexpensive when communicants are in close proximity as in fine and very fine-grain machines. We observe that in coarse-grain machines, typically, communication costs are high and should be minimized.

Furthermore, communication can only be performed between consenting parties. Thus, neighboring processors must synchronize with eachother prior to communicating. In coarse-grain machines careful thought must be given to synchronization to minimize the number of waiting processors so that scarce system resources do not go underutilized. Conversely, fine and very fine-grain machines do not require careful consideration of "load balancing" since (cheap) processors are in plentiful supply.

However, we must not underplay the role of computation in parallel processing. That is, each instruction cycle of a coarse-grain machine may perform much more useful work when executing a PS interpreter than either a fine or very fine-grain device. Thus, it may take a fine grain machine 4 instruction cycles, or 32 for a very-fine grain device, to compute the same value as a single coarse-grain instruction, assuming uniform clock periods over each class. (We choose instruction cycles of 1, 4 and 32 as a rough estimate of the relative power between 32-bit, 8-bit and 1-bit processors executing symbol manipulation instructions; for example, comparing 32-bit words.)

In the following the reader should be cognizant of the fact that the name of the game is *speed*. The question to be decided is which parallel device is best suited to capturing the inherent parallelism in each of the described PS formalisms. It should be noted that all three classes can execute each of the PS models mentioned, but for some classes performance may not differ substantially with a single sequential device. Due to the lack of empirical data in existence today,* our conjectures are based on our intuitions of the relative amounts of communication, computation and inherent concurrency involved with each formalism. Our conjectures are offered as a basis for further scientific study of this curcial area of research.

In our subsequent analysis each class has been "normalized" to include a 100 megabyte total storage capacity so that differences in machine performance depend only on the number of processors and their complexity. No figures depicting any of the device classes have been included in this paper so that no one proposed machine can be used to confuse the situation. Furthermore, the reader will note that we have purposefully included few references concerning parallel computers so that no proposed machine can be claimed to have been inaccurately classified in our analysis. We have, however, ventured to make such a classification only for our own device.

3.1 Coarse-grain

A coarse-grain parallel computer, as noted, consists of say 100 hundred powerful processors each with a megabyte of local memory. Note that with large memories, each processor can store and execute large and complicated programs.

Let us first consider how we might partition rules and WM elements on such a machine.

---

*The author is unaware of any PS programs that have been written to date consisting of 10,000 rules. However, at least one such large-scale system is known to be currently under development, but statistics do not yet exist characterizing its performance.

We may think of localizing all of WM in one processor, leaving the remaining processors with the task of matching independent sets of rules. This approach, though, may be unworkable (especially in the absence of pipelining) due to the bottleneck created when multiple processors attempt to access a single shared data structure. It seems likely that many processors would lay idle while waiting for their WM requests to be satisfied.

Alternatively, we may localize PM in one processor, while distributing WM to the other members of the ensemble. The best that can be achieved in this case is 100-way parallel access to WM, but the match phase would be sequential in nature. A speed-up of matching individual pattern elements might be achievable, but only at the expense of iterating over the rules in PM.

The most appropriate scheme, therefore, is to attempt a partitioning of both PM and WM throughout the system. Thus, in such a device, PM would be distributed among the processors in 100 distinct partitions. The number of rules in each processor's memory is therefore one hundredth the total number of rules in the system. (In our stated example of a 10,000 rule system, 100 rules exists in each partition.) A local copy of a portion of WM as well as the interpreter executing the recognize/act cycle would also be resident within each processors memory. Note that localizing WM in this fashion results in no cost for communication of match operations between different processors.

To achieve the maximum for parallel matching of rules, "similar" productions (productions with common patterns and thus likely to be active at the same time) would be resident in separate partitions. This partitioning forces a great deal of WM redundancy, however, and in the simplest case, an exact copy of WM appears in each processor's memory.

During the match phase, each processor calculates a local conflict set of rules which is subsequently processed in a distributed manner by the selection phase. Once a single rule is selected for execution, the actions specified in the RHS are communicated to all of the processors

It is our conjecture that such a device is best suited to rapdily executing *temporally redundant PS's that do not include global WM tests. To achieve maximum performance for the temporally redundant case, WM elements would necessarily be static.* Furthermore, *global WM tests would be implemented by iterative procedures and, thus, such tests would not be appropriate.*

Our case can be strengthened by considering two key points. First, since communication is expensive in coarse-grain machines, the RHS actions to be communicated must be small, favoring temporal redundancy. Secondly, each processor, as a sequential device, has access to a local WM stored as a conventional data structure, and thus would best process changes to WM that were few in number. Otherwise, a considerable amount of sequential execution would result within each processor from such large numbers of changes.

We now consider performance. Forgy's analysis [Forgy 1982] of the Rete algorithm for OPS execution notes that rule firing rates, in the *best case for serial processors*, are affected logarithmically by the number of rules in PM. Thus, 100 partitions of PM reduces the number of rules in each processor to |PM|/100. In our stated example, we may achieve (in the *worst case for parallel processors*) a speed-up of $\log_2(10{,}000)/\log_2(100)=2$. Forgy's analysis also shows a *worst case* firing rate dependency for serial machines which is linear in the number of rules in PM. Thus, for coarse-grain systems, the speed up achievable may be as high as a factor of 100. This provides a considerable range of possible performance factors, i.e. from 2 to 100. Forgy notes, though, that the *expected* performance is linear, hence favoring factors near 100.

Let us now assume that each RHS specifies 5 WM changes on average. Thus, to achieve 10,000 LIPS, a coarse-grain parallel processor would necessarily execute 2000 recognize/act cycles per second. That is to say, each processor of the ensemble must sustain a cycle rate of 2000 per second for a 100 rule system.

Statistics recently reported* for various PS programs indicate that a 3000 rule system executed on conventional minicomputers operates at 10-20 cycles per second. Hence, the expected performance of a PS with 100 rules is roughly 300-600 cycles per second. This is a factor of 3-6 slower than necessary to achieve 10,000 LIPS. It is our belief, though, that advances in hardware, as well as speed-ups obtained from improved PS compilers recently anounced (OPS83 [Forgy 1984], for example), can presently produce a factor of 5 to 10 over the kinds of general purpose machines for which statistics presently exist. This gives us confidence that indeed 10,000 LIPS is attainable for coarse-grain parallel machines.

3.2 Fine-grain

A fine-grain parallel computer consists of say 10,000 simple processors each with small local memories, in the range of perhaps 10 thousand bytes. Although memory capacity is significantly restricted, it is indeed possible to store programs for production matching, as well as a relatively small number of rules and WM elements. Note, though, that a much larger number of partitions of PM are possible for such a device.

However, in this device not only may rules be distributed in a manner similar to the coarse-grain case, but WM elements as well may be fully distributed to a distinct set of processors. This provides the opportunity of parallel matching of rules, as well as *parallel access* to WM, thus substantially improving the time to match a single pattern against, or calculate a global condition of a large store of facts. Since processors are closely coupled physically in fine-grain machines, communication of pattern matching operations is rather inexpensive. (This capability has been exploited in the DADO machine [Stolfo and Shaw 1982; Stolfo 1983], for example.)

It is our conjecture that *fine-grain devices are best suited to PS programs which allow large changes to WM on each cycle. Since WM is distributed, large global tests of conditions of WM are also efficiently handled. Furthermore, since local memories may contain substantial programs, code performing logical unification rather than simple pattern matching may also be distributed to those processors storing WM elements, allowing for rapid parallel unification.* Thus, WM elements may contain general first order terms. (Indeed, these observations form the basis of the implementation of a logic-based system, similar to PROLOG, on the DADO machine.) We note, though, that *fine-grain machines may also implement temporally redundant PS's with efficiency equal to that of coarse-grain machines.*

How might we achieve 10,000 LIPS for such a machine? We observe that if we distribute the rule set in 100 partitions, we may perform at the same rate as our coarse-grain example in the worst case for parallel processors. However, as noted 4 machine instructions may be required to compute the same result as a single processor of the coarse-grain class. In order to get back our factor of 4 in performance, we would necessarily require roughly 3300 partitions, each containing 3 rules ($\log_2(100)/\log_2(3)=\sim 4$), or 33 times as many partitions. (Recall the logarithmic dependency in Forgy's serial best case analysis.) The resulting 3300 partitions leaves a balance of at most 6700 processors storing WM elements, which will substantially improve the time to match individual patterns against WM. We have ignored this obvious performance gain in our analysis. The serial worst case analysis (again, equal to Forgy's expected performance), however, indicates that only a total of 400 partitions (each containing 25 rules) would be necessary to achieve comparable execution rates to that of coarse-grain machines. (We note further that the size of each partition is smaller than those assumed in the coarse-grain case making it more likely that such partitions will "fit" within each processor's local memory.) We find it likely, therefore, that fine-grain machines can also achieve execution rates in the range of 10,000 LIPS.

*Personal communication with various members of the technical staffs of AT&T Bell Laboratories and Digital Equipment Corporation.

3.3 Very fine-grain

A very fine-grain parallel computer consists of say 1,000,000 extremely simple processors each with 100 bytes of local memory. Since local memory is so severely restricted, we note that no substantial programs can be stored and executed by an individual processor. Such devices are typically driven by a more powerful control processor issuing a single stream of instructions to the smaller processors which operate on different data in lock-step fashion. This mode of operation has been referred to as SIMD execution in the literature on parallel processing.

It seems unlikely that distributing rules to such processors will result in any substantial gains in performance since the indeterminant nature of the match phase for each rule would be difficult to implement in SIMD mode. We note, however, that Forgy [1980] reports on a parallel implementation of PS's for the SIMD-based ILLIAC IV machine, although the ILLIAC IV can be considered a member of the coarse-grain class. Indeed, Forgy's attempt to implement PS's on the ILLIAC IV produced disappointing results, even though the ILLIAC IV processors had enough capacity to store WM elements along with partitioned subsets of rules. The poor performance reported is based on the inability of the SIMD mode of operation to concurrently execute different conditional branches while matching pattern elements. Thus, in general, very fine-grain devices cannot process PM in parallel, but rather the match phase would be implemented by the control processor while iterating over each rule.

WM, however, may be fully distributed throughout the system allowing extremely rapid access to the stored facts. Thus, global tests of WM are efficiently handled. Although each LHS is processed sequentially, individual patterns are matched very quickly. However, WM elements must be static in nature since the single stream of instructions may operate only on data of uniform size and structure.

It is our conjecture that *very fine-grain devices may not be suitable for PS applications at all, beyond those which have very small PM's and very large WM's.* The best application domain of very fine-grain machines in our mind is represented by relational data base query processing or other applications requiring fast searches of large stores of uniform data. As such, we find little reason to attempt to project the performance of very fine-grain devices for PS applications with large rule bases.

By way of summary, we list our results in tabular form as follows:

Class	Coarse	Fine	Very fine
No. processors	100	10,000	1,000,000
Type PS	Temporal Redundant	Both	Neither
Global WM tests	No	Yes	Yes
Static WM	Yes	No	Yes
No. PM partitions	100	400-3300	--
LIPS	~10,000	~10,000	--

4 Is Parallel Computation a Panacea

We conclude with a final question and observation. Recently, some researchers have asked, "Is parallel computation a panacea for AI?" Indeed, should AI be interested in parallel processing at all?

We believe that this is the wrong question to pose. Rather, we should ask "How may AI fulfill its promise in a cost-effective manner?" Parallel computers may not lead us to the promised land in and of themselves, but in our opinion they will get us there less expensively.

It is easy to confuse what is possible and practical for a parallel computer to achieve. For example, it is inviting to think of using vast parallel resources for searching a combinatorial solution space, typical of AI problem-solvers, in parallel. At best, this will only reduce the complexity of the search by a small constant decrease in the exponent. Thus, if 2^n states are to be visited while searching, with 1000 processors (or roughly 2^{10}) in the best case we can search in time proportional to $2^n/2^{10}$, or 2^{n-10}. Parallel processors continue, therefore, to be plagued by exponential search times as their serial forebears.

Rather, parallel computers can be directed to the tasks of decreasing the "cycle time" of AI programs, thus dramatically increasing the effective number of LIPS. It is our thesis that large PS programs, for example, can be significantly accelerated.

Our case in point is the DADO2 machine [Stolfo and Shaw 1982; Stolfo 1983], presently under construction at Columbia University. DADO2 is small prototype of a fine-grain device, comprising 1023 commercially available, 8-bit microprocessors. Over the next 2 years or so, DADO2 promises to execute various PS programs with at least a 10-fold speed-up used in conjunction with certain conventional machines. DADO2, however, is of the same hardware complexity as the conventional devices in question. Thus, a factor of two in hardware complexity produces a projected factor of 10 in speed. (It is interesting to note, though, that DADO2 need only produce a factor of at least 2 in performance to be cost-effective.)

As our final observation, we note that the cost-effectiveness of parallel computing will allow AI to fulfill its promise less expensively. Thus, AI should be very interested in parallel computing!

REFERENCES

Davis, R. and J. King,
An Overview of Production Systems. Technical Report, Stanford University Computer Science Department, 1975.

Forgy, C. L., *On the Efficient Implementation of Production Systems*, Ph.D. Thesis, Carnegie-Mellon University, 1979.

Forgy, C. L., *A Note on Production Systems and ILLIAC IV*, Technical Report 130, Department of Computer Science, Carnegie-Mellon University, 1980.

Forgy, C. L., Rete: A Fast Algorithm for the Many Pattern/ Many Object Pattern Match Problem, *Artificial Intelligence 19*, 1982.

Forgy, C. L., The OPS83 Reference Manual, Department of Computer Science, Carnegie-Mellon University, 1984.

Newell, A., "Production Systems: Models of Control Structures", In W. Chase (editor), *Visual Information Processing*, Academic Press, 1973.

Rychener, M., *Production Systems as a Programming Language for Artificial Intelligence Research*. Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, 1976.

Stolfo, S. and D. E. Shaw, DADO: A Tree-Structured Machine Architecture for Production Systems, *Proc. AAAI*, Pittsburgh, 1982.

Stolfo, S., *The DADO Parallel Computer*, Technical Report, Department of Computer Science, Columbia University, 1983.