A Parallel Asynchronous Distributed Production System*

James G. Schmolze and **Suraj Goel**
Department of Computer Science
Tufts University / Medford, MA 02155 / USA
E-mail: schmolze@cs.tufts.edu

Abstract

To speed up production systems, many researchers have turned to parallel implementations. We describe a system called PARS that executes production rules in parallel. PARS is novel because it (1) executes many rules simultaneously, (2) runs in a highly asynchronous fashion, and (3) runs on a distributed memory machine. Item (1) improves available concurrency over systems that only perform the MATCH step in parallel. Item (2) reduces bottlenecks over synchronous parallel production systems. Item (3) makes the techniques more available given the lower cost of distributed versus shared memory machines. The two main problems regarding correctness, namely serialization and the maintenance of consistent distributed databases, are addressed and solved. Estimates of the effectiveness of this approach are also given.

Introduction

Production systems have been effective vehicles for implementing expert systems. The terminology of production systems and the usual serial execution model are presented in Figure 1. Unfortunately, large production systems are often slow and many will require substantial speed improvements. To speed them up, researchers have studied parallel implementations, with much of that research focusing on OPS5 [Forgy, 1981].

Since most cpu time in OPS5 is spent in the MATCH step (over 90% according to [Forgy, 1979]), many efforts (e.g., [Gupta, 1983; Stolfo, 1984; Gupta, 1986; Miranker, 1987; Oflazer, 1987]) have tried to make parallel that one step while leaving the system to continue executing only one rule at a time. Overall, the maximum speedups realized by these approaches are about ten times the fastest sequential version of OPS5, no matter how many processors are used [Gupta, 1986].

To gain additional speedup, several researchers have investigated systems that execute many rules simultaneously. We call these systems ***multiple rule execution systems.*** When two or more rule instantiations execute simultaneously in one of these systems, we say they ***co-execute.*** Multiple rule execution systems have two important problems. (1) The order of rule execution is difficult to control. (2) The results produced may not be producible under any serial execution scheme, i.e., the final results may not be ***serializable.***

Production Memory: A set of rules or productions.

Working Memory (WM): A set of facts.

Working Memory Element (WME): Each fact in the WM is a WME. Each is a ground literal consisting of a class name plus a sequence of arguments.

Condition Element (CE): A test found in a rule.

Left-Hand Side (LHS): A sequence of CEs associated with a rule that determine when the rule matches against a specific sequence of WMEs.

Action: An operation performed when the rule executes. Usually adds or removes a WME.

Right-Hand Side (RHS): A sequence of actions associated with a rule that determines what to do when the rule is executed.

Basic Loop: The following loop is executed until no matches are found in the first step.

MATCH: For each rule, determine all sequences of WMEs that match its LHS. Each match results in an *instantiation*, which consists of the rule plus the sequence of WMEs matching the unnegated condition elements (CEs) of its LHS. A single rule may have many instantiations.

SELECT: Choose exactly one instantiation according to some predefined *conflict resolution strategy.*

ACT: Perform the actions on the RHS of the rule for the selected instantiation.

Figure 1: Terminology and Serial Execution Model for Production Systems

[Boulanger, 1988] and [Morgan, 1988] each take this approach and each offers a way to control partially the order of rule execution. However, neither guarantees

*This work was supported in part by the National Science Foundation under grant number IRI-8800163.

serializable results. In contrast, [Ishida and Stolfo, 1985; Moldovan, 1986; Pasik, 1989; Schmolze, 1989; Ishida, 1990] do not address the issue of controlling the order of rule execution, but each guarantees serializability. In addition, each takes advantage of parallel matching algorithms, thereby gaining additional speedup.

All of the above multiple rule execution systems are designed to operate *synchronously*, i.e., all processors execute each step together. To obtain good speedup, the work must be effectively distributed for each of the three steps. This is likely to prove difficult since resources must be allocated to solve three different distribution problems. Unfortunately, no efforts along this line have been reported to date. Only the MATCH step has so far been studied in this fashion.

An alternative is to execute a multiple rule execution system in a highly *asynchronous* fashion. This eliminates the time wasted in synchronous systems where some processors are forced to wait for other processors to finish the current step. The distribution problem is still challenging in an asynchronous system, but it is simplified to one distribution problem. Some preliminary work on an asynchronous parallel production system is reported in [Miranker *et al.*, 1989].

Herein, we address the problem of executing a multiple rule execution system that operates asynchronously on a distributed memory machine. The advantages of this approach follow.

- We obtain speedup over serial and parallel-match-only systems by executing all steps in parallel.
- We obtain speedup over synchronous parallel systems by a reduction of bottlenecks and a simplification of the distribution problem.
- We take advantage of distributed memory technology, which is less expensive and more accessible than shared-memory technology given today's market.[1]

This work is related to distributed database (DDBMS) technology, the main difference being the source of the transactions. For DDBMSs, they come from users, and are highly unpredictable. For production systems (PSs), they come from the rules and WM, and are more predictable. Thus, special techniques that take advantage of off-line rule analysis can yield considerable improvement. Sellis *et al* [Sellis *et al.*, 1987; Raschid *et al.*, 1989] take a database approach by delaying certain transactions using locks in order to guarantee serializability. However, they do not take strong advantage of off-line rule analysis, and their system can cause deadlock, whose detection and resolution may require considerable resources. Given that, it is appropriate to pursue other approachs such as ours.

[1]Furthermore, Gupta and Tambe [Gupta and Tambe, 1989] have argued that distributed memory machines are good choices for production systems.

The remainder of this paper describes the design problems we faced, solutions to those problems, and estimates of the effectiveness of our solutions.

The Execution Model

We assume a distributed memory machine with a set of processors, $\mathcal{P}$, and a fast interconnection network between them. Any processor can communicate with any other, either directly or through intermediaries. We are given a set of rules, $\mathcal{R}$, a set of class names, $\mathcal{C}$, that comprise all the classes used in $\mathcal{R}$, and an initial working memory, WM_0. Each WME belongs to exactly one class in $\mathcal{C}$. Each processor uses the same execution model and each is capable of storing rules and WMEs.

For simplicity, we use a static distribution. Each rule in $\mathcal{R}$ is assigned to exactly one processor from $\mathcal{P}$. For each rule r on each processor p, we store on p all classes of WMEs that could match a CE of r's LHS. If a CE in r has a variable appearing as the class, then all classes in $\mathcal{C}$ are stored on p. By storing a class c on p, we mean that all WMEs whose class is c are stored on p. Thus, classes that appear on the LHS of several rules may be stored in several processors.

Each processor p executes a variant of the basic loop for serial systems, as explained below. Each executes the loop at its own pace in an asynchronous fashion.

MATCH: In performing the MATCH step, p only examines the rules and WMEs stored locally. Since all WMEs that could affect the matching of each rule (i.e., all WMEs affecting its LHS) are stored on the same processor as the rule, no matches are missed. We use the TREAT match algorithm [Miranker, 1987].

SELECT: The SELECT step is also performed using only local data. As we will explain later, certain rules may be temporarily disabled and instantiations of such rules are ignored here. Since we do not guarantee any particular overall order of rule execution, each processor arbitrarily selects an instantiation whose rule is not currently disabled. If there are none, p returns to the MATCH step. Otherwise, p goes to the DISABLE step. Assume for now that p selects an instantiation of rule r.

DISABLE: Before executing the selected instantiation, some synchronization must occur. As we will discuss later, we guarantee serializability by prohibiting the co-execution of certain pairs of rules. Thus, in this step, p requests certain other processors to *disable* certain other rules, namely, those rules that cannot co-execute with r. If some of those rules are currently executing, then their execution is completed, after which their processor disables them. p waits until it receives messages verifying that all appropriate rules are disabled. While waiting, it is possible that p received new actions from elsewhere that invalidated the selected instantiation or that another processor disabled r (p responds immediately to such requests). Thus, p checks this, and if so, p does not execute the instantiation and jumps to ENABLE. Otherwise, p goes to ACT.

ACT: The ACT step also requires communication since some of the actions may add or remove WMEs that are stored on other processors. In such cases, the actions are forwarded as appropriate. After actions are sent, the processor waits for acknowledgements from all recipients before re-acknowledging and proceeding. The reasons for this will become clear when we later explain our solution to the inconsistency problem. Thus, at the completion of this step, all processors who store the affected WMEs have taken the actions into account. Actions that affect the local WM are taken immediately.

ENABLE: Rules on other processors that were previously disabled in the DISABLE step are re-enabled. p now returns to MATCH.

Processors accept messages from other processors at the start of the MATCH step and while waiting during the DISABLE step. Even though some synchronization is needed in the DISABLE and ACT steps, it comprises a reasonably small portion of the basic cycle time (MATCH consumes the most) and thus leaves the system highly asynchronous.

We define the *period of the execution* of an instantiation i as extending from the time that the commitment to execute i is made to the later of either (1) the time that i's local actions are taken, or (2) the time that all acknowledgements have been received regarding the delivery of i's actions. (A commitment to execute i is made in the DISABLE step, not the SELECT step.) We say that a set of instantiations *co-execute* iff their periods of execution overlaps.

In addition, we have an algorithm that is guaranteed to determine when all processors have no instantiations. When this occurs, all processors are terminated.

The Serialization Problem

The parallel execution of a set of instantiations is serializable if the result produced by the parallel system could be produced by executing the same instantiations in some serial order. The goal of serialization is to ensure that every execution sequence by our parallel system is serializable.

There are three causes of non-serializability in our parallel model. The first cause is *disabling*. We say that one instantiation *disables* another if executing the first causes the second to no longer match. This occurs iff the first adds (removes) a WME that the second matched negatively (positively). If we co-execute a pair of instantiations where each disables the other, the result is not serializable because, in a serial system, one instantiation would execute but not both. In general, when given a set of instantiations, not all should co-execute if there is a cycle of disabling relations among them.

P1: +(A <x>), −(C <y>) → +(C <x>), +(D <x>)
P2: +(B <x>), −(C <y>) → +(C <x>), +(E <x>)

For example, assume that we execute rules P1 and P2 rules from an initial WM of {(A 1),(B 2)}. Here, both rules match and, if co-executed, produce {(A 1),(B 2),(C 1),(D 1),(C 2),(E 2)}. However, no serial execution can produce this result. If P1 alone were executed from the original WM, the result would be {(A 1),(B 2),(C 1),(D 1)}, in which P2 no longer matches. Similarly, if P2 were executed alone from the original WM, the result would be {(A 1),(B 2),(C 2),(E 2)}, in which P1 no longer matches.

We solve this problem by preventing the co-execution of any set of instantiations that have a cycle of disabling relations among them.

The second cause is *clashing*. We say that one instantiation *clashes* with another if, when executed, one would add a WME that the other would remove. If (a) two instantiations clash and (b) one rule can disable the other, then we might obtain non-serializable results.

P1: +(A <x>), −(C <y>) → +(C <x>), +(D <x>)
P3: +(D <x>), −(C <y>) → −(D <x>), +(E <x>)

For example, rules P1 and P3 would match in {(A 1),(D 1)}, but executing P1 will disable P3. Thus, there are two possible serial orders. We can execute P1 and not P3, or we can execute P3 followed by P1. If the two are co-executed, then the result must be identical to that resulting from executing P3 followed by P1. Since P3 deletes (D 1) while P1 adds it, (D 1) must be in the final WM. Therefore, if these two rules are co-executed, the result must be {(A 1),(C 1),(D 1),(E 1)}.

In our parallel model, we cannot make this guarantee because our control of the order of rule execution is not sufficiently fine. Instead, we solve the problem by prohibiting the co-execution of such pairs of instantiations.

The third cause arises from temporary inconsistencies in the distributed WM. We deal with this problem two sections hence.

Solution to the Serialization Problem

If the execution of two instantiations do not overlap in time, i.e., they do not co-execute, then serialization is guaranteed. Namely, an appropriate serial order simply has the first before the second. We therefore concern ourselves only with sets of instantiations that co-execute. For now, we assume a consistent WM.

We solve the serialization problem by: (1) determining pairs of rules whose instantiations should not co-execute (we say that any such pair of rules are to be

1. Collect all pairs of rules from our set of rules, $\mathcal{R}$, that possibly clash and, for each pair, add an edge to an undirected graph called *RSynch*. *RSynch* has a node for each rule in $\mathcal{R}$ and will eventually have an edge between each distinct pair A and B iff A and B must be synchronized.
2. Construct *RDO*($\mathcal{R}$). Then, construct a modified subgraph of *RDO*($\mathcal{R}$), called *M-RDO*. We start with *RDO*($\mathcal{R}$) and, for each edge from A to B in *RSynch*, remove any edge from A to B or from B to A in *RDO*($\mathcal{R}$).
3. Construct an acyclic sub-graph of *M-RDO* called *A-RDO*. The goal is to remove only edges from *M-RDO* in producing *A-RDO*, and to remove as few edges as possible since each edge removed will create a rule synchronization. We begin *A-RDO* with a node for each rule but no edges. We maintain a reachability matrix, M, for *A-RDO* such that $M[r_j, r_k]$ is true iff there is a directed path from r_j to r_k. We examine each edge, e, from *M-RDO* in turn. Let e go from r_j to r_k. If adding e to *A-RDO* would not create a cycle, we add e to *A-RDO* and update M. Otherwise, we add an edge from r_j to r_k in *RSynch* and, if there is an edge from r_k to r_j in *M-RDO*, we remove it. Afterwards, *A-RDO* is an acyclic sub-graph of *M-RDO* and, for all edges e in *M-RDO* but not in *A-RDO*, the undirected version of e appears in *RSynch*.
4. We synchronize each pair of distinct rules that have an edge between them in *RSynch*.

Figure 2: Off-Line Algorithm to Identify Rules to Synchronize

synchronized [2]), and (2) enforcing these rule synchronizations. We work at the rule, not the instantiation, level because the latter requires too much communication for our hardware model.

We say that one rule *possibly disables* another if any instantiation of the first can disable any instantiation of the second. We define *possibly clashes* in a corresponding fashion. Off-line tests for these two inter-rule relations are given in [Schmolze, 1989].[3] We define a directed graph, called *RDO*($\mathcal{R}$), that represents the *rule disabling order*. For each rule in $\mathcal{R}$, there is a node in *RDO*($\mathcal{R}$). For each A and B in $\mathcal{R}$ where B possibly disables A, there is an edge from A to B in *RDO*($\mathcal{R}$) (note the reversal of A and B). This comprises all of *RDO*($\mathcal{R}$). Also, we define *Rules*($\mathcal{I}$) as the set of rules that have an instantiation in $\mathcal{I}$.

Theorem 1: [4] *The co-execution of a set of instantiations, $\mathcal{I}$, in our parallel model, is serializable if*

1. *RDO(Rules($\mathcal{I}$)) is acyclic except for self loops,*
2. *rules that have self loops in RDO(Rules($\mathcal{I}$)) have only one instantiation in $\mathcal{I}$,*
3. *no two distinct rules in Rules($\mathcal{I}$) can possibly clash,*
4. *rules that possibly clash with themselves have only one instantiation in $\mathcal{I}$, and*
5. *the WM is consistent.*

Since each rule appears on exactly one processor in our model, we cannot have two instantiations of the same rule co-executing. Thus, conditions 2 and 4 are met. We meet condition 3 by synchronizing every pair of rules that possibly clash. We meet condition 1 by identifying a near-minimum number of pairs of rules to synchronize such that, for any set $\mathcal{I}$ of instantiations to co-execute, *RDO*(*Rules*($\mathcal{I}$)) is acyclic. The rules requiring synchronization are identified using the off-line algorithm in Figure 2. The output is a set of pairs of rules that must be synchronized. Condition 5 is assumed for now and dealt with in the next section.

Our method of enforcing rule synchronization was given in the explanation of our execution model. To reduce the work in enforcing these synchronizations, and to prevent deadlock, we take several additional steps. First, we remove from this set of pairs of rules to synchronize any pair where both are on the same processor. No synchronization is needed between such pairs as only one can execute at a time.

Second, we recognize that in order for two rules to synchronize, only one of them needs to initiate the needed disabling. Let A and B be such a pair and let A be the one to initiate the disabling. When B's processor wishes to execute B, it simply does so as long as B is not disabled. When A's processor wishes to execute A, it first disables B, then executes A, and finally re-enables B.

If we think of the pairs of rules to synchronize as an undirected graph, this simplification has the effect of directing each edge. In this case, the undirected edge between A and B becomes a directed edge from A to B, indicating that A can execute only if B is disabled. However, cycles in this directed graph can lead to deadlock. For example, let A require that B be disabled, B require that C be disabled and C require that A be disabled. Let all three are on separate processors and let A, B and C each be selected for execution at the same time. Disabling messages go out from each processor and each waits for acknowledgements before responding to the disable request it receives, leading to deadlock.

[2]We borrow this term from [Ishida and Stolfo, 1985] who first proposed this approach to solving the serialization problem. Our solution represents an improvement over their method along with an adjustment made for our parallel model, which differs from theirs. See [Schmolze, 1989] for a complete discussion.

[3]In essence, r_1 possibly disables r_2 if an add (remove) action of r_1 unifies with a negative (positive) CE of r_2. r_1 possibly clashes with r_2 if an add (remove) action of r_1 unifies with a remove (add) action of r_2.

[4]The proof can be found in [Schmolze, 1989].

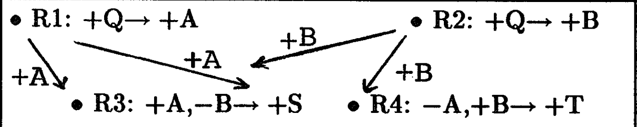

Figure 3: Temporary Inconsistencies Can Cause Non-Serializable Effects

Fortunately, this is easily solved because we can always take an undirected graph that has no self loops and direct each edge such that there are no cycles. To do so, number the M nodes from 1 to M. Go through each edge and direct it from the lower numbered node to the higher numbered one. Since there no self loops, this is an unambiguous choice.[5]

As a result, synchronization is simplified and deadlock free. Moreover, the results of our parallel system are serializable with respect to both disablings and clashes. In fact, we expect only very small delays from the needed synchronizations.[6]

Temporary Database Inconsistencies

Given that our WM is distributed, temporary WM inconsistencies can occur while actions are being sent, which can lead to non-serializable effects. For an example, we offer a case with four rules as shown in Figure 3. Here, R1 and R2 match if there is a WME of class Q and they add WMEs of classes A and B, respectively. R3 matches if there is some WME of class A and no WMEs of class B, and it adds a WME of class S. R4 matches if there are no WMEs of class A and some WME of class B, and it adds a WME of class T. We assume there are no other rules.

Let us examine all serial orders that can arise from an initial WM of {Q}. Both R1 and R2 match, but R3 and R4 do not. Let R1 execute, which adds an A WME. Now, R3 matches as well as R2. Let R3 execute, which adds an S WME. Only R2 matches, so we execute it, which adds a B WME. R4 never matched and still does not due to the presence of the the A WME. Thus, our first serial order is (1) R1, R3 followed by R2. There are three other serial orders, namely, (2) R2, R4 followed by R1, (3) R2 followed by R1, and finally, (4) R1 followed by R2. However, there is no serial order in which all four rules execute.

Unfortunately, with temporary WM inconsistencies in our parallel model, all four rules *can* execute. Assume that each of these rules is on its own processor. Let R1 and R2 co-execute and let each send out its respective action at the same time. Furthermore, let the time of message delivery be small between R1 and R3 and between R2 and R4. Conversely, let the time of delivery be large between R1 and R4 and between R2 and R3. Thus, R3 will receive the +A action from R1 before receiving the $+B$ action from R2. Let us assume that after receiving the +A and before receiving the +B, R3's processor commits to execute R3 (remember, the processors run asynchronously). Note that no rule synchronization is necessary here since there are no possible disabling cycles and no possible clashes. In a similar fashion, R4 receives the +B action before the +A action, and during the interim period, R4's processor commits to executing R4. As a result, we get all four rules executing, which is not serializable.

[5]If there were a loop, it must involve at least two nodes, and thus must have an edge from a higher numbered node to a lower numbered node, which is impossible.□

[6]We note that I/O should also be serialized. We accomplish this by assigning all rules that perform I/O to a single processor. Thus, only one I/O rule executes at a time.

Solution to the Inconsistency Problem

We solve this problem by incorporating a double acknowledgement protocol when sending actions. The protocol for the *sender* of actions is as follows.

1. Send actions to all appropriate processors.
2. Wait for all recipients to acknowledge receipt of the actions.
3. Send re-acknowledgements to all recipients.

The protocol for the *receiver* of actions is as follows.

1. Send an acknowledgement to the original sender.
2. Disable all rules affected by the actions.
3. Continue with the basic loop.
4. Upon re-acknowledgement from the original sender, re-enable the rules disabled by the actions.

The basic idea here is that the original receipt of actions from another processor indicates that certain WMEs will be added or removed. However, to prevent inconsistency problems, the receiver prevents the use of this information until it is sure that *all* appropriate processors have received it. It is sure of this upon receipt of the re-acknowledgement.

The rules disabled by the receiver are those rules having CEs that might test either positively or negatively for the WMEs affected by the action. The reason the receiver disables and re-enables rules is that, while waiting for re-acknowledgements, it can continue with its basic loop.[7]

For an example, we return to Figure 3 when R1 and R2 are about to send out their respective actions to both R3 and R4's processors. R3's processor receives the +A action, disables rule R3 since it tests for a WME of class A, and sends an acknowledgement to R1's processor. At about the same time, R4's processor receives the +B action, disables rule R4, and

[7]In an alternative design, the processor could simply wait for the re-acknowledgements instead of continuing on with its basic loop. It thus would not need to disable and re-enable rules. However, this alternative is wasteful of time, so we reject it.

	No. Non I/O Rules	I&S	PARS
Mapper	77	13.2	6.9
ToruWaltz	31	15.2	6.9
Weighted Average		13.8	6.9
Avg. Improvement Over I&S		1.0	2.0

Table 1: Comparison of Number of Synchronizations per Rule

sends an acknowledgement to R2's processor. Similar activities occur when R3's processor receives the +B action and when R4's processor receives the +A action. Soon thereafter, R1's processor has received the two acknowledgements it wants, and so it sends out re-acknowledgements. At about the same time, R2 does the same. Note that each of the rules R3 and R4 have been disabled twice. Thus, neither is re-enabled until both re-acknowledgements are received. By the time this occurs, both R3 and R4's processors have received both actions so, in this case, neither R3 nor R4 execute.

As a result, we have a protocol that prevents non-serializable effects due to temporary inconsistencies in our distributed WM. The cost of this guarantee is that additional messages must be sent and, for certain periods of time, certain rules are disabled. However, these periods are short as they last only long enough for an acknowledgement and re-acknowledgement messages to be sent. Overall, we estimate this to be a reasonably small fraction of the basic cycle time, leading to a small cost for this guarantee.

Assessment of Effectiveness

An implementation of PARS is currently under construction, so we are not able to report on actual speedups realized. Instead, we estimate speedup in terms of increased numbers of co-executing instantiations.

We applied our rule analysis method to two production systems that are commonly used as benchmarks for parallel production systems. These systems are (1) the Manhattan Mapper [Lerner and Cheng, 1983], which provides travel schedules for trips through Manhattan, and (2) ToruWaltz, which applies Dave Waltz's constraint propagation algorithm to identify objects from line drawings [Waltz, 1975]. First, we measured the number of pairs of rules to be synchronized. Second, we examined a number of conflict sets from serial execution traces and determined how many instantiations would co-execute in our system if it were running synchronously.

We compare our system to that of Ishida and Stolfo's (I&S) [Ishida and Stolfo, 1985] as their's is the most influential work in the area of multiple rule execution

	No. of CS's	Avg. CS Size	I&S	PARS N=8	PARS N=32
Mapper	340	20.3	1.2	2.6	3.1
ToruWaltz	207	47.2	1.0	2.2	4.7
Weighted Average	273.5	30.5	1.1	2.4	4.1
Improvement Over I&S			1.0	2.2	3.7

Table 2: Concurrency Estimates using Number of Co-Executions Available

systems. Thus, we are comparing a synchronous version of our approach to another synchronous system.

Table 1 compares the number of rules to be synchronized for the method of I&S versus ours. As shown in the table, we improve upon their method by having each rule synchronize with, on the average, about half the number of rules that I&S require. Table 2 compares the number of instantiations that could co-execute in our sample traces. For 8 processors, our system co-executes over twice as many instantiations than does I&S. For 32 processors, our system co-executes almost four times as many. While this demonstrates improvement, it does not take into account the cost of our differing execution model, including message passing times. We are thus optimistic that our system will show notable improvements in speedup. However, final results must await actual run times.

Conclusions

We presented a parallel production system called PARS. Novel about PARS is that it (1) executes many rules simultaneously, (2) runs mostly asynchronously, and (3) uses a distributed memory machine. Its advantages are that it (1) improves the available concurrency over serial and match-only parallel approaches by having multiple rule executions, (2) reduces a number of bottlenecks found in synchronous multiple rule execution systems, (3) simplifies the distribution problem over these synchronous systems, and (4) makes the techniques more generally available by using lower cost distributed versus shared memory machines.

PARS guarantees that the final results it produces are serializable. However, like other similar systems, PARS does not guarantee any particular order of rule execution. To help with this, we have added non-OPS5 control constructs. Unfortunately, space does not allow their presentation here (but see [Schmolze, 1988]).[8]

Overall, the speedups estimated for PARS are better than those for other similar systems. However, these speedups are not spectacular. This arises from the

[8]Space also does not allow us to present our distribution algorithm (but see [Goel, 1990]).

benchmarks having been written for serial systems. In the future, we will write and use more appropriate benchmarks. We will experiment with variations of the strategies presented herein, including dynamic distribution schemes. We will incorporate rule decomposition strategies (e.g., [Pasik and Stolfo, 1987; Pasik, 1989]). Finally, we will continue to design new control strategies for multiple rule execution systems.

Acknowledgements

Many thanks go to the students who have contributed either ideas or programs to the development of PARS, which includes David Cheung, Michael Futeran, Eric Gothelf, Kiran Shah, David Yates, and the 1987 class of CSC 280. In addition, special thanks goes to Dan Miranker et al for making the OPS5C system available to us for use in the PARS implementation.

References

[Boulanger, 1988] Albert Boulanger. The Modification of a Rule-based Diagnostic System for Routinized Parallelism on the Butterfly Parallel Computer. Technical Report 6713, BBN Laboratories Inc., Camb., MA, January 1988.

[Forgy, 1979] C.L. Forgy. *On the Efficient Implementation of Production Systems.* PhD thesis, Dept. of Computer Science, Carnegie Mellon Univ., Pittsburg, PA, 1979.

[Forgy, 1981] C.L. Forgy. OPS5 User's Manual. Technical Report CMU-CS-81-135, Dept. of Computer Science, Carnegie Mellon Univ., 1981.

[Goel, 1990] Suraj Goel. Rule Partitioning for Parallel Production Systems. Technical Report 90–2, Dept. of Computer Science, Tufts Univ., Medford, MA, 1990. In preparation.

[Gupta and Tambe, 1989] A. Gupta and M. Tambe. Suitability of Message Passing Machines for Implementing Production Systems. In *Proc. of AAAI-88*, St. Paul, Minnesota, August 1989. American Assoc. for Artificial Intelligence.

[Gupta, 1983] A. Gupta. Implementing OPS5 Production Systems on DADO. Technical Report CMU-CS-84-115, Dept. of Computer Science, Carnegie Mellon Univ., December 1983.

[Gupta, 1986] A. Gupta. *Parallelism in Production Systems.* PhD thesis, Dept. of Computer Science, Carnegie Mellon Univ., March 1986. (Also appears as Technical Report CMU-CS-86-122).

[Ishida and Stolfo, 1985] T. Ishida and S.J. Stolfo. Towards the Parallel Execution of Rules in Production System Programs. In *Proc. of the International Conference on Parallel Processing*, 1985.

[Ishida, 1990] Toru Ishida. Methods and Effectiveness of Parallel Rule Firing. In *Proc. of the 6th IEEE Conference on Artificial Intelligence Applications*, March 1990.

[Lerner and Cheng, 1983] M. Lerner and J. Cheng. The Manhattan Mapper Expert Production System. Technical report, Dept. of Computer Science, Columbia Univ., May 1983.

[Miranker *et al.*, 1989] Daniel P. Miranker, Chin-Ming Kuo, and James C. Browne. Parallelizing Transformations for a Concurrent Rule Execution Language. Technical Report TR-89-30, Dept. of Computer Science, Univ. of Texas at Austin, Austin, Texas, October 1989.

[Miranker, 1987] D. P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* PhD thesis, Dept. of Computer Science, Columbia Univ., 1987. (Also appears as Report No. TR-87-03, Dept. of Computer Science, Univ. of Texas at Austin, 1987.).

[Moldovan, 1986] Dan I. Moldovan. A Model for Parallel Processing of Production Systems. In *Proc. of IEEE International Conerence. on Systems, Man and Cybernetics*, pages 568–573, Atlanta, GA, 1986. IEEE.

[Morgan, 1988] Keith Morgan. BLITZ: A Rule-Based System for Massively Parallel Architectures. In *Proc. of 1988 ACM*, Snowbird, Utah, July 1988. ACM Conference for Lisp and Functional Programming.

[Oflazer, 1987] K. Oflazer. *Partitioning in Parallel Processing of Production Systems.* PhD thesis, Dept. of Computer Science, Carnegie Mellon Univ., 1987. (Also appears as Tech. Rep. CMU-CS-87-114, March 1987.).

[Pasik and Stolfo, 1987] Alexander J. Pasik and Salvatore J. Stolfo. Improving Production system Performance on Parallel Architectures by Creating Constrained Copies of Rules. Technical Report CUCS-313-87, Columbia Univ., New York, New York, 1987.

[Pasik, 1989] Alexander J. Pasik. *A Methodology for Programming Production Systems and its Implications on Parallelism.* PhD thesis, Columbia Univ., New York, 1989.

[Raschid *et al.*, 1989] Louiqa Raschid, Timos Sellis, and Chih-Chen Lin. Exploiting concurrency in a DBMS Implementation for Production Systems. Technical Report CS-TR-2179, Dept. of Computer Science, Univ. of Maryland, College Park, MD, January 1989.

[Schmolze, 1988] James G. Schmolze. An Asynchronous Parallel Production System with Distributed Facts and Rules. In *Proc. of AAAI-88 Workshop on Parallel Algorithms for Machine Intelligence and Pattern Recognition*, St. Paul, Minn., August 1988.

[Schmolze, 1989] James G. Schmolze. Guaranteeing Serializable Results in Synchronous Parallel Production Systems. Technical Report 89-5, Dept. of Computer Science, Tufts Univ., Medford, MA, October 1989.

[Sellis *et al.*, 1987] Timos Sellis, Chih-Chen Lin, and Louiqa Raschid. Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms. Technical Report CS-TR-1960, Dept. of Computer Science, Univ. of Maryland, College Park, MD, December 1987.

[Stolfo, 1984] S.J. Stolfo. Five Parallel Algorithms for Production System Execution on the DADO Machine. In *Proc. of the AAAI-84.* American Assoc. for Artificial Intelligence, 1984.

[Waltz, 1975] D.L. Waltz. Understanding Line Drawings of Scenes with Shadows. In P. Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw Hill, New York, NY., 1975.