FIVE PARALLEL ALGORITHMS FOR PRODUCTION SYSTEM EXECUTION ON THE DADO MACHINE*

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

Abstract

In this paper we specify five abstract algorithms for the parallel execution of production systems on the DADO machine. Each algorithm is designed to capture the inherent parallelism in a variety of different production system programs. Ongoing research aims to substantiate our conclusions by empirically evaluating the performance of each algorithm on the DADO2 prototype, presently under construction at Columbia University.

1 Introduction

In this paper we outline five abstract algorithms specifying parallel execution of production system (PS) programs on the DADO machine. Each algorithm offers a number of advantages for particular types of PS programs. We expect to implement these algorithms on the DADO2 prototype and critically evaluate the performance of each on a variety of application programs. Software development is presently underway using the DADO1 prototype that has been operational at Columbia University since April, 1983.

We begin with a brief description of PS's and identify various possible characteristics of PS programs which may not be immediately apparent from a general description of the basic formalism. These characteristics lead to different algorithms which will be discussed in the remaining sections of this paper.

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 and variables (prefixed with an equals sign), while WM elements are simple lists of constant symbols (corresponding to *tuples* of the *relational algebra*). An example production, borrowed from the blocks world, is illustrated in figure 1.

```
(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
        and assert that =y is on the table.
```

Figure 1: An Example Production.

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 the goal of accelerating the rule firing rate of the recognize/act cycle as well as the number of WM transactions performed. In a later section of this paper, we shall consider other possible parallel activities as, for example, the concurrent execution of multiple PS programs.

*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.

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. Finally, the RHS of a rule can be processed by a parallel update of WM. We summarize this approach by the abstract algorithm illustrated in figure 2.

```
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).
3. Repeat until no rule is active:
   a. 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.
   b. Considering each maximally rated instance
      within each processor, compute the
      maximally rated rule within the entire
      system. Report its instantiated RHS.
   c. Broadcast the changes to WM reported in
      step 3.b to all processors, which update
      their local WM accordingly. end Repeat;
```

Figure 2: Abstract Production System Algorithm.

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

3 Characteristics of Production System Programs

In this section we enumerate various characteristics of PS programs in general terms. The reader will note that these characteristics are less indicative of a specific PS formalism, but rather are characteristics of various problems whose solutions are encoded in rule form. It should be noted, though, that *the "inherent parallelism" in some problems may not be represented by the particular PS formalism used for their solution.*

1. **Temporal Redundancy.** Few WM changes are made on each cycle. Thus, by saving state between each cycle, previous matching operations need not be repeated. The Rete algorithm [Forgy 1982] is probably the best example of a PS interpreter incorporating this strategy.
2. **Few Affected Rules.** Few rules are affected by changes to WM on each cycle, and thus relatively few rules need be matched against the new state of WM. Note, however, that temporal redundancy alone does not guarantee this to be always the case.
3. **Many Affected Rules.** Many rules are affected by the changes to WM on each cycle. This may arise, for example, in situations where similar pattern elements appear in many rules.
4. **Massive changes to WM** (non-temporally redundant). In this case, action specifications in the RHS of a rule may have large global effects on WM. Thus, restricting the scope of the match operation seems unlikely, i.e., saving state is not appropriate.
5. **Restricted scope of pattern matches.** The number of WM elements which may potentially match each rule is relatively small. Thus, a single rule may not need access to all of WM but to a relatively small subset of data elements.
6. **Global tests of WM.** Pattern elements in the LHS of a rule may test conditions requiring access to large portions of WM, rather than individual elements (for example, tests which compare the number of WM elements against some constant threshold value). This case may be viewed as the converse of characteristic 5.
7. **Multiple rule firings.** On each cycle of operation, a number of conflict rules may be executed prior to initiating the match phase of the next cycle.
8. **Small PM.** The number of rules is restricted to only a few hundred.
9. **Small WM.** Similarly, WM may consist of only a few hundred elements.
10. **Large PM.** A PS may consist of several thousands of rules in PM.
11. **Large WM.** Similarly, WM may consist of thousands of data elements.

4 Five Algorithms

In this section we outline five different algorithms suitable for direct execution on the DADO machine. Each will be independently discussed leading to various conclusions about which characteristics they are most appropriate for capturing. Ongoing research aims to verify our conclusions by empirically evaluating their performance for different classes of PS programs.

The reader is assumed to be knowledgeable about the Rete match algorithm (see [Forgy 1979] and [Forgy 1982]). We will thus freely discuss the details of the Rete match when needed without prior explication. We begin with a brief description of the DADO architecture. (The reader is encouraged to see [Stolfo 1983] and [Stolfo and Miranker 1984] for complete details of the system.)

4.1 The DADO Machine

DADO is a fine-grain, parallel machine where processing and memory are extensively intermingled. A full-scale production version of the DADO machine would comprise a very large (on the order of a hundred thousand) set of *processing elements* (PE's), each containing its own processor, a small amount (16K bytes, in the current design of the prototype version) of local random access memory (RAM), and a specialized I/O switch. The PE's are interconnected to form a *complete binary tree*.

Within the DADO machine, each PE is capable of executing in either of two modes under the control of run-time software. In the first, which we will call *SIMD mode* (for single instruction stream, multiple data stream), the PE executes instructions broadcast by some ancestor PE within the tree. (SIMD typically refers to a single stream of "machine-level" instructions. Within DADO, on the other hand, SIMD is generalized to mean a single stream of remote procedure invocation instructions. Thus, DADO makes more effective use of its communication bus by broadcasting more "meaningful" instructions.) In the second, which will be referred to as *MIMD* mode (for multiple instruction stream, multiple data stream), each PE executes instructions stored in its own local RAM, independently of the other PE's. A single conventional coprocessor, adjacent to the root of the DADO tree, controls the operation of the entire ensemble of PE's.

When a DADO PE enters MIMD mode, its logical state is changed in such a way as to effectively "disconnect" it and its descendants from all higher-level PE's in the tree. In particular, a PE in MIMD mode does not receive any instructions that might be placed on the tree-structured communication bus by one of its ancestors. Such a PE may, however, broadcast instructions to be executed by its own descendants, providing all of these descendants have themselves been switched to SIMD mode. The DADO machine can thus be configured in such a way that an arbitrary internal node in the tree acts as the root of a tree-structured SIMD device in which all PE's execute a single instruction (on different data) at a given point in time. This flexible architectural design supports multiple-SIMD execution (MSIMD). Thus, the machine may be logically divided into distinct partitions, each executing a distinct task, and is the primary source of DADO's speed in executing a large number of primitive pattern matching operations concurrently.

Our comments will be directed towards the DADO2 prototype consisting of 1023 PE's constructed from commercially available chips. Each PE contains an 8 bit Intel 8751 processor, 16K bytes of local RAM, 4K bytes of local ROM and a semi-custom I/O switch. The DADO2 I/O switch, which is being implemented in semi-custom gate array technology, has been designed to support rapid global communication. In addition, a specialized combinational circuit incorporated within the I/O switch will allow for the very rapid selection of a single distinguished PE from a set of candidate PE's in the tree, a process we call *max-resolving*. (The max-resolve instruction computes the maximum of a specified register in all PE's in one instruction cycle, which can then be used to select a distinct PE from the entire set of PE's taking part in the operation.) Currently, the 15 processing element version of DADO performs these operations in firmware embodied in its off-the-shelf components.

4.2 Algorithm 1: Full Distribution of PM

In this case, a very small number of distinct production rules are distributed to each of the 1023 DADO2 PE's, as well as all WM elements relevant to the rules in question, i.e., only those data elements which match some pattern in the LHS of the rules. Algorithm 1 alternates the entire DADO tree between MIMD and SIMD modes of operation. The match phase is implemented as an MIMD process, whereas selection and act execute as SIMD operations.

In simplest terms, each PE executes the match phase for its own small PS. One such PS is allowed to "fire" a rule, however, which is communicated to all other PE's. The algorithm is illustrated in figure 3.

```
1. Initialize: Distribute a simple rule matcher to
   each PE. Distribute a few distinct rules to each
   PE. Set CHANGES to initial WM elements.
2. Repeat the following:
3. Act: For each WM-change in CHANGES do:

      a. Broadcast WM-change (add or delete a
         specific WM element) to all PE's.
      b. Broadcast a command to locally match.
         [Each PE operates independently in MIMD
         mode and modifies its local WM. If this
         is a deletion, it checks its local conflict set
         and removes rule instances as appropriate.
         If this is an addition, it matches its set of
         rules and modifies its local conflict set
         accordingly].
      c. end do;

4. Find local maxima: Broadcast an instruction to
   each PE to rate its local matching instances
   according to some predefined criteria (conflict
   resolution strategy (see [McDermott and Forgy,
   1978]).
5. Select: Using the high-speed max-RESOLVE
   circuit of DADO2, identify a single rule for
   execution from among all PE's with active rules.
6. Instantiate: Report the instantiated RHS actions.
   Set CHANGES to the reported WM-changes.
7. end Repeat;
```

Figure 3: Full Distribution of Production Memory.

4.2.1 Discussion of Algorithm 1

We have left the details of the local match routine unspecified at step 3.b. Thus, a simple precompiled Rete match network and interpreter may be distributed to each processor. However, it is not clear whether a simple naive matching algorithm may be more appropriate since only a very small number of rules is present in each PE. Memory considerations may decide this issue: the overhead associated with linking and manipulating intermediate partial matches in a Rete network may be more expensive than direct pattern matching against the local WM on each cycle.

Performance of this algorithm varies with the complexity of the local match. In the best case, the time to match the rule set is bounded by the time to match only a few rules. The worst case is dependent on the maximum number of WM elements accessed during the match of the rules. If a simple naive match is used at each PE, this may require a considerable amount of computation even though the size of the local WM's is limited. Simple hashing of WM may dramatically improve a local naive matching operation, however.

We conclude that this algorithm is probably best suited to implementing PS programs characterized by:

1. Temporal redundancy, since massive changes to WM would require a considerable amount of sequential execution at each PE to modify its local WM.
3. Many rules are affected on each cycle. Thus, depending on the initial distribution of PM, it would be best to partition similar rules separately. Note, though, that characteristic 2 may also be suitable, but a relatively small number of PE's would be actively computing new match results on each cycle.
5. Restricted scope of pattern matches. Clearly, each rule is required to potentially match against a relatively small local WM. Hence, global tests of WM would not be particularly appropriate.
9. Large PM is possible. Given the above characteristics, three or four rules stored at each PE make it possible for a PM consisting of 3000-4000 rules.
11. Similarly, depending on the average number of common pattern elements between rules, WM may be quite large. Even if an average of one unique WM element is resident in each PE (while a significant number of additional local WM elements are replicated in other PE's), a minimum of 1000 individual elements may be stored in WM.

The most serious drawback of this algorithm is the case where a local WM is too large to be conveniently stored in a PE. Clearly, characteristic 5 is appropriate for this algorithm only in the presence of characteristic 9, small WM.

Multiple rule firings (characteristic 7) are indeed possible. A discussion of this case is deferred to a later section.

4.3 Algorithm 2: Original DADO Algorithm

The original DADO algorithm detailed in [Stolfo 1983] makes direct use of the machine's ability to execute in both MIMD and SIMD modes of operation at the same point in time. The machine is logically divided into three conceptually distinct components: a *PM-level*, an *upper tree* and a number of *WM-subtrees*. The PM-level consists of MIMD-mode PE's executing the match phase at one appropriately chosen level of the tree. A number of distinct rules are stored in each PM-level PE. The WM-subtrees rooted by the PM-level PE's consist of a number of SIMD mode PE's collectively operating as a hardware content-addressable memory. WM elements relevant to the rules stored at the PM-level root PE are fully distributed throughout the WM-subtree. The upper tree consists of SIMD mode PE's lying above the PM-level, which implement synchronization and selection operations.

It is probably best to view WM as a distributed *relation*. Each WM-subtree PE thus stores relational tuples. The PM-level PE's match the LHS's of rules in a manner similar to processing relational queries. In terms of the Rete match, ***intracondition*** tests of pattern elements in the LHS of a rule are executed as relational *selection*, while *intercondition* tests correspond to *equi-join* operations. Each PM-level PE thus stores a set of relational tests compiled from the LHS of a rule set assigned to it. Concurrency is achieved between PM-level PE's as well as in accessing PE's of the WM-subtrees. The algorithm is illustrated in figure 4.

4.3.1 Discussion of Algorithm 2

This algorithm was specifically designed for PS programs characterized as:

4. Non-temporally redundant. Indeed, the ability to distribute WM elements in a content-addressable memory allows not only parallel access to WM for matching, but large changes to WM may also be efficiently implemented. For such an environment, saving state between cycles has few advantages.
3. Many rules are affected by WM-changes on each cycle. Since massive changes to WM may be permitted on each cycle, many rules may potentially be affected. The concurrency achieved at the PM-level would allow many rule matchings to be achieved efficiently.
6. Global tests are also efficiently handled by the WM-subtrees operating as an SIMD mode parallel device.
8. PM is, unfortunately, rather restricted in size. Since only one level of the tree is used for rule storage, the full capacity of the machine for PM is underutilized. In DADO2, for example, we envisage a PM-level at level 4 of the machine. Thus, 32 PE's would each store roughly 30 rules for a thousand rule system, potentially decreasing performance. Rule systems with a few hundred rules are more appropriate.
11. WM may be quite large, however. For example, the DADO2 configuration noted above would allow for 32 WM-subtrees, each consisting of 32 PE's. Since each DADO PE has considerable storage capacity, many thousands of WM elements may be easily stored. Furthermore, this allows a 32-way parallel access to WM for each PM-level PE. In total, nearly 1000 WM elements may be accessed in parallel at a given point in time.

While attempting to implement temporally redundant systems, Algorithm 2 may recompute much of its match results calculated on previous cycles. This indeed may not be the case if we modify Algorithm 2 to incorporate many of the capabilities of the Rete match.

```
1. Initialize: Distribute a match routine and a
   partitioned subset of rules to each PM-level PE.
   Set CHANGES to the initial WM elements.
2. Repeat the following:
3. Act: For each WM-change in CHANGES do;
   a. Broadcast WM-change to the PM-level
      PE's and an instruction to match.
   b. The match phase is initiated in each PM-
      level PE:
        i. Each PM-level PE determines if WM-
           change is relevant to its local set of
           rules by a partial match routine. If
           so, its WM-subtree is updated
           accordingly. [If this is a deletion, an
           associative probe is performed on the
           element (relational selection) and any
           matching instances are deleted. If
           this is an addition, a free WM-
           subtree PE is identified, and the
           element is added.]
       ii. Each pattern element of the rules
           stored at a PM-level PE is broadcast
           to the WM-subtree below for
           matching. Any variable bindings that
           occur are reported sequentially to the
           PM-level PE for matching of
           subsequent pattern elements
           (relational equi-join).
      iii. A local conflict set of rules is formed
           and stored along with a priority
           rating in a distributed manner within
           the WM-subtree.
   c. end do;
4. Upon termination of the match operation, the
   PM-level PE's synchronize with the upper tree.
5. Select: The max-RESOLVE circuit is used to
   identify the maximally rated conflict set
   instance.
6. Report the instantiated RHS of the winning
   instance to the root of DADO.
7. Set CHANGES to the reported action
   specifications.
8. end Repeat;
```

Figure 4: Original DADO Algorithm.

Simple changes may dramatically improve the situation. For example, rather than iterating over each pattern element in each rule as in step 3.b.ii, we may only execute the match for those rules affected by new WM changes. The selection of affected rules can be achieved quickly using the WM subtree as an associative memory. By distributing pattern elements as relational tuples in a manner similar to WM, associative probing (relational selection) can be used to select rules for matching (perhaps faster than hashing).

Consideration of these techniques led us to investigate Rete for direct implementation on DADO2. Algorithms 3 and 4 detail this approach.

4.4 Algorithm 3: Miranker's TREAT Algorithm

Daniel Miranker has invented an algorithm which modifies Algorithm 2 to include several of the features of the Rete match for saving state. The *TRE*e *A*ssociative *T*emporally redundant (TREAT) algorithm [Miranker 1984] makes use of the same logical division of the DADO tree as in Algorithm 2. However, the state of the previous match operation is saved in distributed data structures within the WM-subtrees.

TREAT views the pattern elements in the LHS of rules as relational algebra terms, as in Algorithm 2. Thus, the evaluation of such relational algebra tests is also executed within the WM-subtrees. State is saved in a WM-subtree in the form of distributed Rete *alpha memories* corresponding to partial selections of tuples matching various pattern elements. Rule instances in the conflict set computed on previous cycles are also stored in a distributed manner within the WM-subtrees. These two additions substantially improve the performance of Algorithm 2. (We note that Anoop Gupta of Carnegie-Mellon University independently analyzed a similar algorithm in [Gupta 1983]. Compared to Algorithm 2, TREAT should perform substantially better for temporally redundant systems. We note that Gupta's analysis of algorithm 2, however, depends on certain assumptions that derive misleading results.)

Another aspect of TREAT is the clever manner in which *relevancy* is computed. Pattern elements are first distributed to the WM subtrees. When a new WM element is added to the system, a simple match at each WM-subtree PE determines the set of rules at the PM-level which are affected by the change. Those identified rules are subsequently matched by the PM-level PE restricting the scope of the match to a smaller set of rules than would otherwise be possible with Algorithm 2.

The TREAT algorithm is outlined in figure 5.

4.4.1 Discussion of Algorithm 3

The TREAT algorithm is a refinement of Algorithm 2 incorporating temporal redundancy. Hence, TREAT is best suited for PS programs characterized as:

1. Temporally redundant.
3. Many rules are affected on each cycle.
6. Global tests of WM are also efficiently handled.
8. Small PM.
11. Large WM.

We note, though, that minor changes allow TREAT to implement Algorithm 2 directly (by setting L to all of the rules at the PM-level in step 3.d.ii and ignoring step 3.d.i). Thus, TREAT may also efficiently execute:

4. Non-temporally redundant systems.

In step 3.d.iii, TREAT also implements a useful

```
1. Initialize: Distribute to each PM-level PE a
   simple matcher (described below) and a compiled
   set of rules. Distribute to the WM-subtree PE's
   the appropriate pattern elements appearing in
   the LHS of the rules appearing in the root PM-
   level PE. Set CHANGES to the initial WM
   elements.
2. Repeat the following:
3. Act: For each WM-change in CHANGES do;
     a. Broadcast WM-change to the WM-subtree
        PE's.
     b. If this change is a deletion, broadcast an
        instruction to match and delete WM
        elements and any affected conflict set
        instances calculated on previous cycles.
     c. Broadcast an instruction to PM-level PE to
        enter the Match Phase.
     d. At each PM-level PE do;
          i. Broadcast to WM-subtree PE's an
             instruction to match the WM-change
             against the local pattern element.
          ii. Report the affected rules and store in
             L.
          iii. Order the pattern elements of the
             rules in L appropriately.
          iv. For each rule in L do;
               1. Match remaining patterns of the
                  rules specified in L as in
                  Algorithm 2.
               2. For each new instance found,
                  store in WM-subtree with a
                  priority rating.
               3. end do;
          v. end do;
     e. end for each;
4. Select: Use max-RESOLVE to find the
   maximally rated instance in the tree.
5. Report the winning instance.
6. Set CHANGES to the instantiated RHS of the
   winning rule instance.
7. end Repeat;
```

Figure 5: The TREAT Algorithm.

strategy. When iterating over each of the rules in L affected by recent changes in WM, those pattern elements with the smallest alpha memories are processed first. This technique tends to process the join operations quickly by filtering out many potentially failing partial joins.

As noted above, Gupta's analysis of a TREAT-like algorithm, as well as subsequent analysis performed by Miranker [1984], show TREAT to be highly efficient compared to Algorithm 2 executing temporally redundant systems. (The implementation, study and detailed analysis of TREAT forms a major part of Daniel Miranker's Ph.D. thesis.)

4.5 Algorithm 4: Fine-grain Rete

A Rete network compiled from the LHS's of a rule set consists of a number of simple nodes encoding match operations. Tokens, representing WM modifications, flow through the network in one direction and are processed by each node lying on their traversed paths. Fortunately, the maximum fan-in of any node in a Rete network is two. Hence, a Rete network can be represented as a binary tree (with some minimal amount of node splitting).

This observation leads to Algorithm 4 whereby a logical Rete network is embedded on the physical DADO binary tree structure. In the simplest case, leaf nodes of the DADO tree store and execute the initial linear chains of one-input test nodes, whereas internal DADO PE's execute two-input node operations. The physical connections between processors correspond to the logical data flow links in the Rete network. The entire DADO machine operates in MIMD mode while executing this algorithm, behaving much like a pipelined data flow architecture.

Algorithm 4 is illustrated in figure 6.

4.5.1 Discussion of Algorithm 4

Since this algorithm is a direct implementation of the Rete match, it is most suitable for PS programs characterized as:

1. Temporally redundant
2. Few rules are affected by WM changes. This observation is noted in [Forgy 1979].
10. Large PM. We may, for instance, believe that only 1023 Rete nodes may be processed by DADO2. However, a straight forward overlay technique can be implemented where several Rete networks are embedded in the tree and processed in turn. Thus, large PM may be achievable.
9. Small WM. However, since Rete network nodes require significant storage for intermediate partial match results (stored at alpha and beta memories), the limited storage capacity of a DADO2 PE may require restricting the size of WM.

Although overlayed Rete networks would be processed sequentially on DADO2, significant performance improvements can be achieved by a natural pipelining effect. Immediately following a successful match and communication at a node, the next two-input test from the overlayed network is initiated. Thus, while the parent node is processing the first network node, its children are proceeding with their tests of the second overlayed network node.

A second source of pipelining can improve performance as well. In this case, the entire RHS action specification is broadcast at once to the DADO leaf PE's at step 3.a. Immediately following the conclusion of the first match operation and communication of the first WM

```
1. Initialize: Map and load the compiled Rete
   network on the DADO tree. Each node is
   provided with the appropriate match code and
   network information (see [Forgy 1982] for
   details). Set CHANGES to initial WM
   elements.
2. Repeat the following:
3. Act: For each WM-change in CHANGES do;
     a. Broadcast WM-change (a Rete token) to
        the DADO leaf PE's.
     b. Broadcast an instruction to all PE's to
        Match. (First, the leaf processors execute
        their one-input test sequences on the new
        token. The interior nodes lay idle waiting
        for match results computed by their
        descendants. Those tokens passing the
        one-input tests are communicated to the
        immediate ancestors which immediately
        begin processing their two-input tests. The
        process is then repeated until the physical
        root of DADO reports changes to the
        conflict set maintained in the DADO
        control processor).
     c. end do;
   Select: The root PE is provided with the chosen
   instance from the control processor. Set
   CHANGES to the instantiated RHS.
4. end Repeat;
```

Figure 6: Fine-grain Rete Algorithm.

token, the leaf PE's initiate processing of the second WM token. Hence, as a WM token flows up the DADO tree, subsequent WM tokens flow close behind at lower levels of the tree in pipeline fashion.

4.6 Algorithm 5: Multiple Asynchronous Execution

In our discussion so far, no mention was made about characteristic 7, multiple rule firings. We may view this as

- multiple, independently executing PS programs, or
- executing multiple conflict set rules of the same PS program concurrently.

In this regard we offer not a single algorithm, but rather an observation that may be put to practical use in each of the abovementioned algorithms.

We note that any DADO PE may be viewed as a root of a DADO machine. Thus, any algorithm operating at the physical root of DADO may also be executed by some descendant node. Hence, any of the aforementioned algorithms can be executed at various sites in the machine concurrently! (This was noted in [Stolfo and Shaw 1982].) This coarse level of parallelism, however, will need to be controlled by some algorithmic process executed in the upper part of the tree. The *simplest* case is represented by the procedure illustrated in figure 7, which is similar in some respects to Algorithm 2.

```
1. Initialize. Logically divide DADO to incorporate
   a static Production System-Level (PS-level),
   similar to the PM-level of Algorithm 2.
   Distribute the appropriate PS program to each
   of the PE's at the PS-level.
2. Broadcast an instruction to each PS-level PE to
   begin execution in MIMD mode. (Upon
   completion of their respective programs, each
   PS-level PE reconnects to the tree above in
   SIMD mode.)
3. Repeat the following.
     a. Test if all PS-level PE's are in SIMD
        mode.
   End Repeat;
4. Execution Complete. Halt.
```

Figure 7: Simple Multiple PS Program Execution.

In the cases where various PS-level PE's need to communicate results with eachother, step 3 is replaced with appropriate code sequences to report and broadcast values from the PS-level in the proper manner. Each of the programs executed by PS-level PE's are first modified to synchronize as necessary with the root PE to coordinate the communication acts, at, for example, termination of the Act phase.

In addition to concurrent execution of multiple PS programs, methods may be employed to concurrently execute portions of a single PS program. These methods are intimately tied to the way rules are partitioned in the tree. Subsets of rules may be constructed by a static analysis of PM separating those rules which do not directly interact with each other. In terms of the *match* problem-solving paradigm, for example, it may be convenient to think of independent subproblems and the *methods* implementing their solution (see [Newell 1973]). Each such method may be viewed as a high-level subroutine represented as an independent rule set rooted by some internal node of DADO. Algorithm 1, for example, may be applied in parallel for each rule set in question. Asynchronous execution of these subroutines proceeds in a straight forward manner. The complexity arises when one subset of rules infers data required by other rule sets. The coordination of these communication acts is the focus of our ongoing research. Space does not permit a complete specification of this approach, and thus the reader is encouraged to see [Ishida 1984] for details of our initial thinking in this direction.

5 Conclusion

We have outlined five abstract algorithms for the parallel execution of PS programs on the DADO machine and indicated what characteristics they are best suited for. We summarize our results in tabular form as follows:

Algorithm	**PS Characteristics**
1. Fully Distributed PM	1, 3, 5, 7, 9, 11
2. Original DADO	3, 4, 6, 7, 8, 11
3. Miranker's TREAT	1, 3, 4, 6, 7, 8, 11
4. Fine-grain Rete	1, 2, 5, 7, 9, 10
5. Multiple Asynchronous	Applies to all cases.

Of the five reported algorithms, only the original DADO algorithm (number 2) has been carefully studied analytically. The performance statistics of the remaining four algorithms have yet to be analyzed in detail. However, much of the performance statistics cannot be analyzed without specific examples and detailed implementations. Working in close collaboration with researchers at AT&T Bell Laboratories, in the course of the next year of our research we intend to implement each of the stated algorithms on a working prototype of DADO.

In this paper, we have outlined our expectations concerning the suitability of each of the algorithms for a variety of possible PS programs. We expect our reported findings to substantiate our claims, and intend to demonstrate this with working examples in the near future.

References

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

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

Forgy C. L. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem. *Artificial Intelligence*, 1982, *19*, 17-37.

Gupta, A. *Implementing OPS5 Production Systems on DADO.* Technical Report, Department of Computer Science, Carnegie-Mellon University, 1983.

Ishida T., and S. J. Stolfo. *Simultaneous Firing of Production Rules on Tree-structured Machines.* Technical Report, Department of Computer Science, Columbia University, 1984.

McDermott, J. and C. Forgy. Production System Conflict Resolution Strategies. In Waterman and Hayes-Roth (Eds.), *Pattern-directed Inference Systems,* Academic Press, 1978.

Miranker D. P. *Performance Estimates for the DADO Machine: A Comparison of TREAT and RETE.* Technical Report, Department of Computer Science, Columbia University, April 1984.

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

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

Stolfo S. J. *The DADO Parallel Computer.* Technical Report, Department of Computer Science, Columbia University, August 1983. (Submitted to AI Journal).

Stolfo S. J., and D. E. Shaw. *DADO: A Tree-structured Machine Architecture for Production Systems.* Proceedings National Conference on Artificial Intelligence, Carnegie-Mellon University, August, 1982.

Stolfo S. J., and D. P. Miranker. *The DADO Production System Machine: System-level Details.* Technical Report, Department of Computer Science, Columbia University, 1984. (Submitted to IEEE Transactions on Computers).