MORE RULES MAY MEAN FASTER PARALLEL EXECUTION.

Salvatore J. Stolfo, Daniel M. Miranker
& Russell C. Mills

CUCS-175-85

More Rules May Mean Faster Parallel Execution[1]

Salvatore J. Stolfo
Daniel M. Miranker
and
Russell C. Mills

CUCS-175-85

Columbia University
New York, N.Y. 10027

April 16, 1985

Abstract

In this brief paper we report a simple scheme to extract implicit parallelism in the low-level match phase of the parallel execution of Production System programs. The essence of the approach is to replicate rules while introducing new constraints within each copy to restrict each individual rule to match a potentially smaller set of data elements. Speed up is achieved by matching each copy of a rule in parallel. Variations of this approach may be applicable to logic-based programming systems, such as PROLOG, executed in a parallel environment. Indeed, sequential implementations of OPS-style production systems based on the RETE match algorithm may enjoy performance advantages as well. This scheme may be implemented by a simple preprocessing stage which requires no modification to the underlying match algorithms.

1 Introduction

We have previously reported a number of parallel algorithms to accelerate the execution of characteristically different *Production System* (PS) programs [Stolfo 1984]. The simplest, called the *Full Distribution Algorithm*, is based on allocating each rule, as well as the *Working Memory* (WM) elements relevant to its left-hand side, to a single processing element (PE) of a large-scale, fine-grain multiprocessor, such as the DADO machine. In essence, the original PS is converted into a large number of "one-rule" PS's each processed concurrently.

For some PS programs, however, the potential speed-up of the match phase for the Full Distribution Algorithm is not nearly as great as might be expected. In programs such as R1, where few rules may potentially match newly asserted WM elements on each cycle, few PE's may perform useful work, while in programs such as ACE that work with large databases, local requirements for WM elements may exceed the capacity of some PE's. In other cases, certain anomalous rules may require more processing on average than other rules, thus producing "hot spots" of sequential execution in a distributed environment.

In this brief paper, we report a simple scheme to extract implicit parallelism in the low-level match phase of an individual rule which has the potential to improve greatly the performance of

---

[1]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.

the Full Distribution Algorithm while mitigating the problem of PE memory overflow and reducing the effects of "hot-spot" rules. The essence of the approach is to replicate anomalous rules and to introduce constraints within the copies which restrict them to match smaller, disjoint portions of the set of potentially relevant WM elements. This scheme is particularly advantageous since the Full Distribution Algorithm is the simplest of the entire set of reported algorithms and requires the least inter-PE communication. Before describing the load balancing scheme, we detail the operation of a production system program, as well as the Full Distribution Algorithm for the parallel execution of production system programs.

2 Production Systems

In general, a *Production System* [Newell 1973, Davis and King 1977] 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. An example production, borrowed from the blocks world, is illustrated in Figure 1 in the syntax style of OPS5 [Forgy 1981].

Figure 1: An Example Production.

```
(p clear-block
 (Goal ^name Clear-top-of Block ^status ON)
 (Physical-object ^name <x> ^type Block)
 (On-top-of ^bottom-object <x> ^top-object <y>)
 (Physical-object ^name <y> ^type Block)  ->
                 (delete 3)
                 (assert (On-top-of ^bottom-object Table
                                 ^top-object <y>)))
```

```
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. 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. Rules matching data elements that were more recently inserted in WM are preferred, with ties decided in favor of rules that are more specific (i.e., have more constants) than others.

In the *Full Distribution Algorithm*, one or a very small number of distinct production rules are distributed to each of the DADO 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. In simplest terms, each PE executes the match phase for its own small production system. Only one such production system is allowed to "fire" a rule, however, which is communicated to all other PE's. The algorithm is illustrated in Figure 2.

3 Copy and Constrain Rules

Measurements reported in [Gupta and Forgy 1983] show that in an average OPS5 production system, only about 32 rules are affected by changes to working memory during each production cycle. Furthermore, even if all active rules are assigned to different PE's, some PE's will take longer than others to complete the match phase. Since the select phase of the Full Distribution Algorithm cannot begin until all rules have finished matching, the total time spent in the match phase is the time taken by the slowest, not the average, rule. Simulations on OPS5 programs [Gupta 1984] indicate that because of the large variation in processing time among the affected rules, the average speed-up obtainable in the match phase from production-level parallelism is a factor of about 6. The approach we present below has the potential for transcending these limitations and increasing parallel speed-up by augmenting the number of affected rules and decreasing the variance of their processing times.

The scheme is best introduced by means of a simple example. Consider the stylized rule

$$P_1 (C_1 C_2 \ldots C_n \rightarrow A_1 \ldots A_m),$$

where the C_i ($i=1,\ldots,n$) are condition elements, and the A_i are actions. If we interpret WM_1 (the set of working memory elements relevant to P_1's left-hand side) as a large relation, or set of tuples, then we may view each condition element C_i as a *relational selection*. The set of instantiations (matches) of P_1 is the equijoin of the relations R_i selected by the condition elements C_i subject to the restriction that variables be consistently bound across all conditions. The local memory requirements and execution time to match P_1 are thus bounded by (and indeed may achieve) the size of the full Cartesian product of the individual relations R_i.

Figure 2: Full Distribution of Production Memory.

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 match locally. [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).

5. Select: Using DADO's high-speed I/O circuit, 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;

Suppose for concreteness that C_2 is a relational selection of a large number of physical objects, represented by the OPS5-style pattern:

(PHYSICAL-OBJECT ^Name <x> ^Color <y> ^Shape <z>),

and that the domain of the Color attribute of the relation WM_1 is {RED, GREEN}, that is, that physical objects are either RED or GREEN. To speed up the match of rule P_1, we split the set of working memory elements associated with P_1 and set two PE's to the concurrent tasks of matching constrained versions of P_1. We thus construct two new condition elements:

C'_2 (PHYSICAL-OBJECT ^Name <x> ^Color RED ^Shape <z>)
C''_2 (PHYSICAL-OBJECT ^Name <x> ^Color GREEN ^Shape <z>),

two new rules:

P'_1 (C_1 C'_2 ... C_n --> A_1 ... A_m)
P''_1 (C_1 C''_2 ... C_n --> A_1 ... A_m),

and two new working memories:

WM'_1 = { w in WM_1: Color(w) = RED if w is a PHYSICAL-OBJECT}
WM''_1 = { w in WM_1: Color(w) = GREEN if w is a PHYSICAL-OBJECT},

and assign them to distinct PE's, PE'_1 and PE''_1. P'_1 and P''_1 may clearly be matched in parallel, and the set of instantiations of P_1 is exactly the disjoint union of the instantiations of P'_1 and P''_1.

In the best case, half the tuples selected by the original condition C_2 of rule P_1 reside in each of PE'_1 and PE''_1, and the processing time required to match P_1 decreases by half, since the two new rules can be matched in parallel. Local processing requirements and storage of WM elements for each rule decrease significantly as well. In the worst case, of course, all the tuples selected by C_2 of rule P_1 reside in one of the two PE's, PE'_1 and PE''_1, and partitioning buys nothing. If more PE's are available, the scheme can be applied repeatedly, producing many copies of rules, each constrained to match a smaller range of distinct WM elements. Thus, with many PE's available, it should be possible to reduce the inter-PE variation in processing times, balancing the execution load over the entire system and increasing overall performance dramatically.

If a small finite domain of attribute values is not known a priori (as in the above example with RED and GREEN objects), two variations on the technique are possible. The first is *hash partitioning*. Suppose that in the above example, the domain of the Color field is not {RED, GREEN}, but is some (possibly infinite) domain D. If there is an easily computable function

f: D -> {1,....,k},

we can split WM_1 into k partitions

$WM_1(j)$ = { w in WM_1: f(Color(w)) = j if w is a PHYSICAL-OBJECT}.(j=1.....k),

and assign each of the k partitions to a separate PE along with a suitably constrained version of the rule P_1. In the best case, again, the processing time for the match phase of P_1 is divided by k, the number of partitions. Thus, our scheme is similar in many respects to hash partitioning tuples in a single relational query executed iteratively on relations exceeding the size of main memory. However, in our case hash partitioning is applied in parallel to a large number of concurrent queries operating on a large number of small relations.

The second variation of the basic technique applies when the domain is a totally ordered set: we

can simply split it into disjoint subranges. Continuing with the above example, suppose that working memory elements have the form

(PHYSICAL-OBJECT ^Name <x> ^Reflected-Wavelength <y> ^Shape <z>),

and that a set of values

$$v_{min}=v_0, v_1, \ldots, v_k=v_{max}$$

of Reflected-Wavelength are given. We can again split working memory, this time into

$WM_1(j) = \{$ w in WM_1:
Reflected-Wavelength(w) >= v_{j-1} and Reflected-Wavelength(w) < v_j
if w is a PHYSICAL-OBJECT$\}$, $(j=1,\ldots,k)$,

and assign each to a separate PE together with constrained versions of the original rule P_1. This disjoint-subranges scheme is a special case of hash partitioning that bypasses the explicit computation of a hashing function.

4 Implementation Outline

The scheme outlined can be implemented by a simple preprocessor supplied with information on how to partition the domains of working memory attributes. These pragmas, or hints, can take the form of explicit values or ranges provided by the programmer (derived from knowledge of the problem or from previous executions of the production system program), or can include hashing functions. The preprocessor's role is simply to generate new productions incorporating the value, hashing function, or subrange tests, one for each value, function value, or subrange supplied.

OPS-style productions are easily modified by adding partitioning information to *literalize* declarations (see the OPS5 manual [Forgy 1981]). The declarations for PHYSICAL-OBJECT in our first example (when a small finite domain of attribute values is known) might be

```
(literalize PHYSICAL-OBJECT
    Name
    Color (symbol RED GREEN)
    Shape)
```

To specify a hashing function defined on the Color field with range {1,....k}, we could write

```
(literalize PHYSICAL-OBJECT
    Name
    Color (hash hash-function-name k)
    Shape)
```

Specifying k subranges of a totally ordered domain is just as easy:

```
(literalize PHYSICAL-OBJECT
    Name
    Reflected-Wavelength (range v1 ... vk-1)
    Shape)
```

The preprocessor should split each rule containing non-constant tests on the partitioned fields into the appropriate set of more specialized rules.

This approach has the potential of greatly improving the performance of a variety of PS programs including those with thousands of rules and hundreds of WM elements, as for example

R1, and conversely those with hundreds of rules and thousands of WM elements, as for example ACE. Other coarse-grained approaches to the parallel execution of PS programs may make effective use of this scheme as well in addition to conventional uniprocessor implementations. The Rete match algorithm, for example, would thus compile additional match nodes for the introduced partitioning constraints which would effectively reduce the size of the Beta memories of the original condition element. The resultant sequential search of the Alpha and Beta memories to compute partial match results would thus be quicker since fewer data elements would be compared with eachother.

Indeed, logic-based programming systems may also make use of similar approaches to load balance execution by introducing copied and constrained clauses. In a PROLOG environment, the idea may be used as follows: If in a rule

a:- b,c,d

it is known via a pragma that goal b dominates the computation for satisfying goal a, then rewrite the rule as two rules:

a:-b',c,d
a:-b'',c,d

where b' and b'' are copied versions of goal b with constraints on the first order terms to unify with a smaller and distinct set of terms that would match b. Since goals b' and b'' would be executed in parallel in an OR-parallel environment, a speed up is achievable. One approach to figuring out the constraints for goals b' and b'' is to execute the PROLOG program symbolically in order to identify the range of unit literals that terminate the goal tree emanating from the posting of goal b. The range of first order literals so identified may then be partitioned by suitably constraining the first order terms of literal b.

References

Davis, R. and J. King, "An Overview of Production Systems", Machine Intelligence, 8, J. Wiley and Sons, New York, pp. 300-332, 1977.

Forgy, C. L., "OPS5 User's Manual", Technical Report CS-81-135, Department of Computer Science, Carnegie Mellon University, 1981.

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

Gupta, A. and C. L. Forgy, "Measurements on Production Systems", Tech Report, CMU, 1983.

Gupta, A, "Parallelism in Production System: The Sources and the Expected Speed-Up", Tech Report, CMU, 1984.

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

Stolfo, S. J., "Five Parallel Algorithms for Production System Execution on the DADO Machine", Proc. of the National Conference of Artificial Intelligence, 1984.