Towards the Parallel Execution of Rules in Production System Programs[1]

Toru Ishida[2]
and
Salvatore J. Stolfo

CUCS-154-85

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

25 October 1984

Abstract

To improve the performance of forward chaining production systems, a new parallel execution model is proposed, which fires multiple rules simultaneously on multiple processor systems. Two problems are discussed on the model, and efficient algorithms to solve these problems are proposed. The synchronization analysis algorithm determines the necessity of synchronization between rule firings, and the decomposition algorithm determines a mapping of rules on multiple processor systems. Evaluation results on an existing production system show that a speed-up of 7.5 is obtained by introducing the parallel firing mechanism.

---

[1]This research has been supported by the Defense Advanced Research Projects Agency through contact 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.

[2]Visiting from the Yokosuka Electrical Communication Laboratory, Nippon Telegraph and Telephone Public Corporation, 1-2356, Take, Yokosuka, Japan.

Table of Contents

1 Introduction

Forward chaining production systems have been widely applied in the implementation of a number of knowledge based expert problem-solving systems [1,2]. However, it is also reported that the performance of production systems is not satisfactory compared to more conventional programming languages. Although advances in the implementation of production system interpreters have provided substantial performance improvements, further speed improvements are required for very large production systems with severe time constraints [3].

Recently, several multiple processor architectures and parallel algorithms for production systems have been investigated as possible solutions to the above problem. Stolfo [4] and Miranker [5] proposed several parallel match algorithms, which are especially well suited for the tree-structured parallel machine DADO [6]. Forgy [3] reported expected effects of parallel matchings based on the evaluation of several existing production systems written in the OPS language [7]. Oflazer [8] has proposed partitioning algorithms for production systems to maximize the effect of parallel matchings.

Parallel match algorithms aim to speed-up the match process which consumes more than 90% of the total execution time [9]. The effect of improving the time to match rules thus compresses each cycle of production system execution. In this paper, we propose a new parallel execution model, the parallel firing mechanism. The mechanism aims to reduce the total number of sequential production rule cycles by executing multiple matching rules simultaneously on multiple processor systems. Thus, parallel rule firings provide more performance improvements in the execution of production systems, when combined with parallel match approaches. In our opinion, a substantial amount of parallelism can be expected to occur within sequential rule firings, because of the fundamental nature of production systems; i. e., the rules are considered as statements representing independent chunks of knowledge which may interact less than other more procedural formalisms [10].

The following two major problems are discussed to realize parallel firings.

1. *Synchronization Problem*: Production rules are written without consideration of interference with other rules. To guarantee the execution environment of a particular rule, it is necessary to determine what rules need to synchronize with the rule in question, and to suspend the firings of such rules during its execution.

2. *Decomposition Problem*: To maximize the benefit of parallel firings, efficient decomposition algorithms are required to find an optimal partition or distribution of a given production system, so that multiple rules can be fired as often as possible.

This paper is organized as follows. Section 2 provides a more detailed description of the execution model of parallel firings. Algorithms to analyze the synchronization of rule firings and to map rules on multiple processors are discussed in sections 3 and 4. Finally, some results of our preliminary evaluation of this approach is presented in Section 5.

2 Basic Definitions and Concepts

2.1 Production Systems

Before describing the details of our approach, we begin with a brief overview of production systems and their forward chaining execution.

A *production system* is defined by a set of rules or productions, called the *production memory* (*PM*), together with a database of assertions, called the *working memory* (*WM*). Each rule consists of a conjunction of condition 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 which is to be added to or removed from WM when the LHS successfully matches against the contents of WM.

A rule written in the OPS5 production system language [7] is shown in Fig. 1.

```
(p make-possible-trip
   (city ^name <x> ^state New-York)
  -(weather-forecast ^place <x> ^date tomorrow ^weather rainy)
   -->
   (make possible-trip ^place <x> ^date tomorrow))
```

Fig. 1 An Example of OPS5 Rules

In OPS5, WM consists of tuples of attribute-value pairs called *working memory elements* (*WMEs*). Two kinds of condition elements are provided in production rules; a *positive condition element*, that is satisfied when there exists a matching WME, and a *negative condition element*, that succeeds when no matching WME is found. Pattern variables are consistently bound throughout the positive condition elements. Thus the rule in Fig. 1 may be read as:

```
If
  there is a  WME in the system representing
  a city in New York state
  and
  there is no WME in the system representing that
  it will be rainy tomorrow in that city
then
  create a new WME tagging the city is a possible destination
  of tomorrow's trip.
```

Suppose the following is currently the only WME in the system.

```
(city ^name Buffalo ^state New-York)
```

where *city* represents a class name of WMEs, and attribute names are prefixed by '^' with attribute values immediately following. In this case, the LHS is satisfied, and if the rule is fired the following new WME is added to the system.

```
(possible-trip ^place Buffalo ^date tomorrow)
```

In this paper, we focus attention on an irrevocable forward chaining production system interpreter [11] including OPS5. The interpreter repeatedly executes the following cycle of operations:

- *Match*: For each rule, determine whether the LHS matches the current environment of WM.
- *Select*: Choose exactly one of the matching rules according to some predefined criterion.
- *Act*: Add to or delete from WM all assertions as specified by the RHS of the selected rule.

2.2 Execution Model of Parallel Firings

A *multiple processor system* consists of a *control processor* (*CP*) and a large set of *processing elements* (*PEs*). Each PE has its own local memory, and can execute its own program. We do not assume a particular inter-processor communication mechanism. Thus, the approach detailed in this paper is applicable to both shared memory and non-shared memory multiple processor architectures.

Production rules are distributed between the CP and all PEs. In our execution model of parallel firings, we do not assume that only one rule is chosen in the Select phase. Rather, we propose to execute multiple rules simultaneously on multiple processors. Thus, in our model, production systems are required to be written without any assumptions of particular selection algorithms.[3] However, there exists the case that the result of parallel execution of rules is different from the results of sequential executions in any order of applying those rules. For example, suppose there are two rules whose task is to reserve an air plane ticket. If both rules are fired at the same time, there is the possibility of double booking two flights. In this case, we say that there exists *interference* between rules. To avoid such an erroneous execution, rules are synchronized at each production cycle as appropriate.

[3] However, this assumption does not imply that we require commutative production systems [11].

For simplicity, we assume the same execution time for all production rules, and call the unit of time a *production cycle*. One production cycle is executed as follows.

1. *Match*: Each processor (the CP and PEs) executes the Match phase simultaneously. Each PE reports matched rules to the CP.

2. *Select*: The CP chooses one of the matched rules for each PE, so that interference does not occur between any pair of PEs.

3. *Act*: Each PE executes the Act phase simultaneously. In shared memory architectures, the WM is assumed to be allocated in a single memory and accessed from all PEs. In non-shared memory architectures, the WM is assumed to be distributed in all PEs, and the changes of the WM are reported to the CP and then broadcast to all PEs. If there is no PE which can execute a production rule in this production cycle, then the CP executes one of its matched rules. Otherwise, the rules in the CP are never fired.

3 Synchronization between Production Rules

3.1 Data Dependency Graph

To analyze the interference between production rules, we introduce a *data dependency graph of production rules*, which is constructed from the following primitives.

- A *production node* (a *P-node*), which represents a production rule.

- A *working memory node* (a *W-node*), which represents a group of working memory elements, called a *class* [7].

- A *directed edge from a P-node to a W-node*, which represents the fact that the RHS of a production rule modifies (adds or deletes) a class of WM elements. When a rule adds (deletes) WM elements of a class, the class is called as '*+changed*' ('*-changed*'), and the corresponding edge is labelled '+' ('-').

- A *directed edge from a W-node to a P-node*, which represents the fact that the LHS of a production rule refers to a class of working memory elements. When a class is referenced by a positive (negative) condition element of a rule, the class is called as '*+referenced*' ('*-referenced*'), and the corresponding edge is labelled '+' ('-').

Fig. 2 shows an example of a data dependency graph, which represents the rule illustrated in Fig. 1.

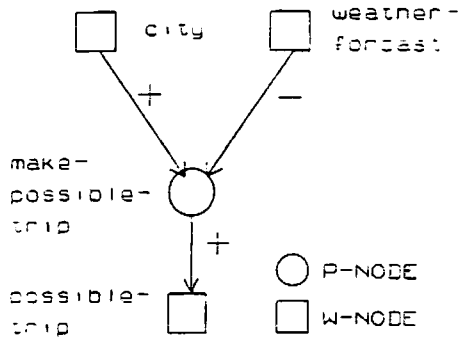


Fig. 2 An Example of a Data Dependency Graph

3.2 Synchronization Analysis

This section details a method to analyze the interference (i. e., the necessity of synchronization) of two distinct rule firings based on a data dependency analysis. The following observations can be derived from a data dependency graph.

- If all WM classes lying between rule A and rule B are '+'changed ('-'changed) by rule A and '+'referenced ('-'referenced) by rule B, then the firing probability of rule B increases monotonically by executing rule A. Thus, even if rule A is fired during the execution of rule B, interference never occurs.

- Conversely, if some WM classes lying between rule A and rule B are '+'changed ('-'changed) by rule A and '-'referenced ('+'referenced) by rule B, then the execution environment of rule B may be destroyed by the firing of rule A. (See Fig. 3 (1))

- If rule A and rule B change the same WM class, and if the class is '+'changed ('-'changed) by rule A and '-'changed ('+'changed) by rule B, then the result of simultaneous firing may be different from the result of any sequential executions of rule A and rule B. (See Fig. 3 (2))

From the above observations, we can say that synchronization between rule A and rule B is required, if there exists a WM class which satisfies any of the following conditions.

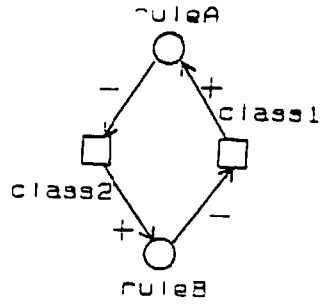
1. '+'changed ('-'changed) by rule A and '-'referenced ('+'referenced) by rule B

2. '+'changed ('-'changed) by rule B and '-'referenced ('+'referenced) by rule A

3. '+'changed ('-'changed) by rule A and '-'changed ('+'changed) by rule B

```
(p ruleA
   (class1)
   -->
   (remove class2))

(p ruleB
   (class2)
   -->
   (remove class1))
```

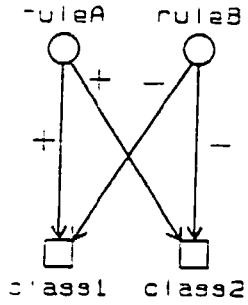



```
<WM elements>
BEFORE FIRING
  (class1, class2)
AFTER SEQUENTIAL EXECUTION
  (class1), when ruleA is
            first executed.
  (class2), when ruleB is
            first executed.
AFTER SIMULTANEOUS EXECUTION
  nil
```

(1)

```
(p ruleA
     :
   -->
   (make class1)
   (make class2))

(p ruleB
     :
   -->
   (remove class1)
   (remove class2))
```




```
<WM elements>
BEFORE FIRING
  nil
AFTER SEQUENTIAL EXECUTION
  nil, when ruleB is
       executed after ruleA.
  (class1, class2),
       when ruleA is
       executed after ruleB.
AFTER SIMULTANEOUS EXECUTION
  (class1, class2) or
  (class1) or
  (class2) or
  nil
```

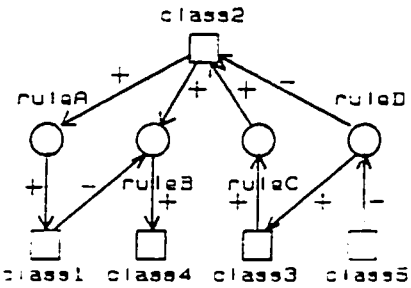
(2)

Fig. 3 An Example of Interference between Two Rules

The process of synchronization analysis produces a *synchronization set* for each rule, which contains all rules to be synchronized with the rule in question. Fig. 4 shows an example of a synchronization analysis.

```
(p ruleA
   (class2)
   -->
   (make class1))
(p ruleB
  -(class1)
   (class2)
   -->
   (make class4))
(p ruleC
   (class3)
   -->
   (make class2))
(p ruleD
  -(class5)
   -->
   (remove class2)
   (make class3))
```

(1) Sample Rules



(2) Data Dependency Graph

```
ruleA: (ruleB, ruleD)
ruleB: (ruleA, ruleD)
ruleC: (ruleD)
ruleD: (ruleA, ruleB, ruleC)
```

(3) Synchronization Sets

Fig. 4 An Example of Synchronization Analysis

3.3 Discussion of the Synchronization Analysis

Limitations of Static Analysis

The synchronization analysis algorithm is based on static analysis techniques and thus has certain limitations. In several cases the analyzer erroneously reports the necessity of synchronizing a set of rules although there is no need to synchronize their firings in practice.

- *WM classes include multiple independent subclasses*: For example, one can combine two classes, say class A and B, and create a new class, say class C, whose first attribute represents the original class names. If one rule refers to subclass A of class C in a positive condition element, and the other rule deletes some data in subclass B of class C, then the analyzer concludes that two rules should be synchronized. More accurate analysis can be performed by mapping W-nodes not to classes but to subclasses, if the information about subclasses is provided to the analyzer.

- *Class names are represented by variables*: In this case, the static analyzer cannot identify the class name which is actually referenced in the execution of a rule. To be on the safe side, the analyzer assumes that all classes are possibly referenced. If the possibly referenced class names are given to the analyzer, more accurate results would be obtained.

More accurate analysis provides a greater opportunity for parallel execution. Human help is necessary for the analyzer to produce the minimum synchronization sets.

Man-machine cooperation is also helpful to develop production systems which are suitable for parallel firings. In our evaluation process reported in section 5, the performance of an existing production system was improved many times by avoiding the interference based upon results of the synchronization analysis.

Comparison between Production Systems and Procedural Programs

Several researchers have reported on approaches to parallelizing procedural programs based on data dependency analysis ([12,13], for example). In a procedural language, conditions on parallel execution of two successive statements, say A and B, are:

1. B does not reference the variables set by A,

2. A does not reference the variables set by B, and

3. A and B do not set the same variables.

These conditions are stricter than the conditions for parallel firing of production rules, because assignment is considered as a sequence of deleting and adding. The underlying difference between procedural programs and production systems is that rules independently add/delete WMEs to/from the WM, while programs set/reference values to/from the same storage area under pre-specified controls. This comparison suggests that production systems structurally contain more possibilities for parallel execution than procedural programs as reported in [14,15].

4 Mapping Production Rules on Multiple Processors

4.1 An Overview of the Decomposition Problem

We now discuss how to decompose and allocate production systems on multiple processor systems. First of all, production systems should be decomposed into two parts; i.e. one for the CP and the other for PEs. For example, rules with I/O operations should be allocated on the CP.

The rest of this section describes an algorithm for the decomposition of production rules, which should be allocated to PEs. The aim of this decomposition is to reduce the execution time by mapping rules to a set of distinct PEs in such a way so that multiple rules may be concurrently fired as often as possible. We represent the benefit of decomposition by the number of reduced production cycles.

The decomposition problem has two fundamental difficulties. First, the number of possible decompositions is too large to take an exhaustive approach. Second, the optimal decomposition may change, when the usage of the production system changes.

To overcome the above difficulties, we use the following strategies.

- We define *parallel executability* between each pair of rules as the number of production cycles which can be reduced by allocating the two rules in distinct PEs.

- To reduce the complexity of the combinatorial problem, the decomposition is based on the parallel executability of each pair of rules. To obtain an efficient solution in an incremental manner, the most influential rule pair, which has the largest parallel executability, is first allocated.

- We use sample execution traces to calculate the parallel executability of each pair of rules. Thus, analyzing additional execution traces incrementally approaches an optimal decomposition. Furthermore, tuning is possible after the production system starts to work.

4.2 Decomposition Algorithm

Our algorithm, called the *hierarchical decomposition algorithm*, consists of two phases. In the first phase, the algorithm produces a hierarchical data structure, called a *rule tree*. In the second phase, partitions for parallel processor systems are created from the rule tree. Precise procedures in each phase are described below.

Phase1: Generating a Rule Tree

A *token* is defined as a triple, *(ruleA ruleB P(ruleA,ruleB))*, where rule A and rule B are production rules and P(ruleA,ruleB) is a parallel executability between rule A and rule B. We assume all tokens for all pairs of rules are already calculated by using sample execution traces. (The actual number of tokens is discussed later.)

The goal of this phase is to produce a rule tree in which each rule is associated with a distinct leaf node, and to maximize a sum of P(i, j) at each non-leaf node in all combinations of i and j, where i/j indicates a rule in a right/left subtree of the non-leaf node. The tree initially consists of only one root node, which contains all rules. Tokens are input to the root node in descending order of P. Each node processes tokens as follows.

- *If both rules are contained in the node*, move one of them to the root node of its left subtree, and the other to the root node of its right subtree. (If subtrees are null, create root nodes for those subtrees, and do the above operation.)

- *If one of two rules is contained in the root node and the other is contained in its right (left) subtree*, move the rule, which is contained in the node, to the root node of its left (right) subtree. (If the subtree is null, create the root node for the subtree, and do the above operation.)

- *If both rules are contained in its left (right) subtree*, pass the token to the root node of its left (right) subtree.

- *If one of two rules is contained in its right subtree and the other is contained in its left subtree*, ignore this token.

Phase2: Create Partitions for Parallel Processor Systems

This phase creates partitions of a production system for parallel processor systems from the rule tree. Because the rule pairs with a large parallel executability are decomposed in the early stage, partitions for a parallel processor system can be easily obtained by selecting a suitable layer of the rule tree. (The hierarchical rule

tree grows exponentially and thus is best mapped on to processors containing a number of PEs that is a binary power.) An example of the decomposition process is shown in Fig. 5.

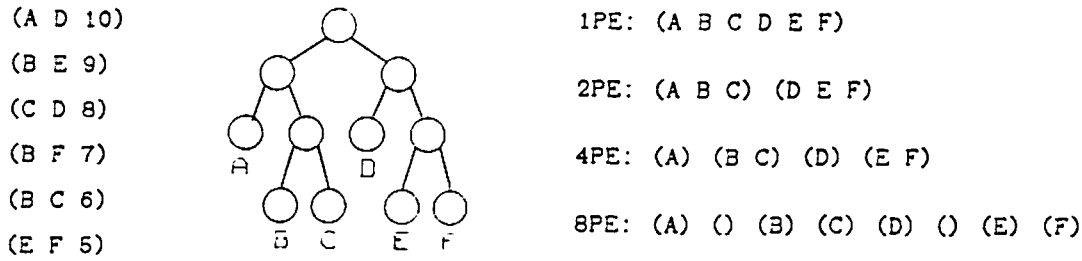


Fig. 5 An Example of a Decomposition Process

4.3 Discussions on the Decomposition Algorithm

The Number of Tokens

The number of possible pairs of rules is $n(n-1)/2$, when the number of rules is n. However, in an actual environment, the number of tokens is much smaller. The reason is that many pairs of rules will, in practice, never fire in parallel even though the production system contains a substantial amount of parallelism. For example, during our evaluation reported in section 5, only 17% of all possible pairs are processed as tokens.

Heuristics on Ordering Tokens

In the decomposition algorithm presented in section 4.2, tokens are processed in descending order. However, more heuristics for ordering tokens were applied during the evaluation reported in section 5, because partitions derived from a descending order has the following disadvantages.

- Sometimes, there exists a big difference between the numbers of rules on the right and left subtrees. (For example, see Fig. 6 (1).)
- Rule pairs are often allocated without considering the relationship with other rule pairs; i. e., the condition *'if both rules are contained in the node'*, is often satisfied. (For example, see Fig. 6 (2).)

A more sophisticated ordering, called *the linked order of tokens*, has been invented to avoid the above disadvantages. In the linked order, tokens are ordered so that rules are allocated to right and left subtrees one after the other. The linked order is obtained by the following procedures.

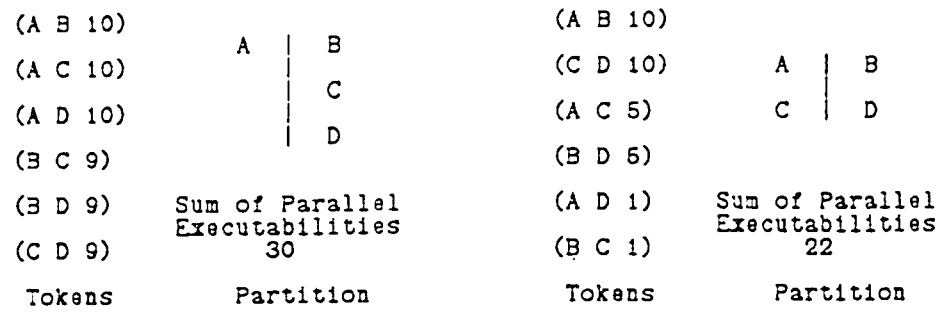


Fig. 6 Decomposition Based on a Descending Order

- Make a list of tokens, in which tokens are listed in descending order.
- Pop a token from the list and register it as the first token of a linked order. Post both rules in the first token to a common table.
- The *n*th token of a linked order is selected as follows.
    - Check tokens in the list from the top to the end, and select the first token which contains a posted rule.
        - If there is a token which contains some posted rule, then remove the token from the list and register it as the *n*th token of the linked order. Find a rule in the *n*th token which is not posted. Clear the common table and post the found rule.
        - If there is no token which contains any posted rule, then pop one token from the list and register it as the *n*th token. Clear the common table and post both rules in the *n*th token.

Fig. 7 illustrates the decomposition results obtained by using the linked order for the same problems appearing in Fig. 6. The sums of parallel executabilities in both case are improved by introducing the linked order. The linked order is applied to all layers of a rule tree by stacking and linked-ordering all tokens which are passed from a parent node. During the evaluation reported in section 5 the linked-order decomposition produced partitions which are about 20% faster than the ones obtained by the decomposition based on the descending order.

5 Implementation and Evaluation Results

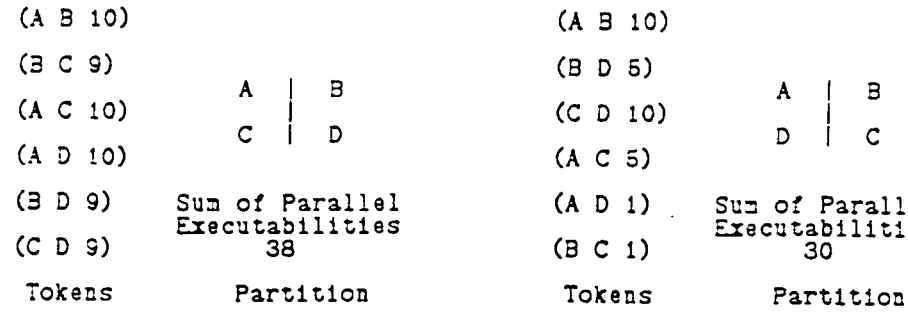


Fig. 7 Decomposition Based on a Linked Order

5.1 Simulation Environment

To evaluate the effectiveness of parallel execution of production systems, a simulation environment has been developed. The environment depicted in Fig. 8 consists of three major subsystems; i.e. an *Analyzer*, a *Simulator*, and a *Decomposer for Production Systems*.

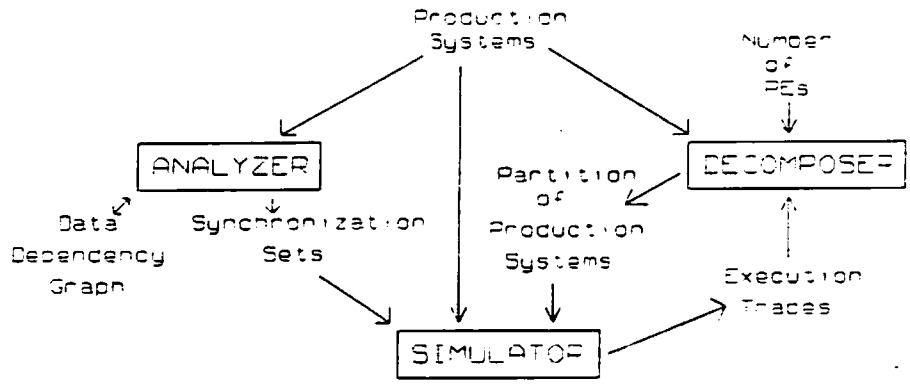


Fig. 8 A Simulation Environment

- The *Analyzer* inputs production systems written in OPS5. It constructs a data dependency graph of a given production system, analyzes the graph and outputs the synchronization sets. The synchronization analysis algorithm, which is described in section 3, is used in the Analyzer.

- The *Simulator* simulates the parallel execution of production systems. It can simulate any partition of production systems for parallel processor systems, and creates execution traces. The synchronization sets, which is the output of the Analyzer, are referenced during the simulation. For the purpose of obtaining the parallel executability between each pair of rules, special simulations are performed. Those assume that each rule is allocated on a distinct PE. The results are used by the Decomposer.

- The *Decomposer* analyzes the execution traces, which are created by the Simulator, and measures the parallel executability (the number of parallel firings) between each pair of rules. The Decomposer creates partitions of production rules based on the hierarchical decomposition algorithm described in section 4.

5.2 Evaluation Results

A preliminary evaluation has been performed on the Manhattan Mapper expert system [16], which has been developed at Columbia to provide travel schedules in Manhattan. Production rules of the Manhattan Mapper consists of two groups, i.e. rules for man-machine interface and rules for planning. We allocate the rules for man-machine interface to the CP and distribute the rules for planning (49 rules) to PEs. The following graph illustrates the main results of the preliminary evaluation.

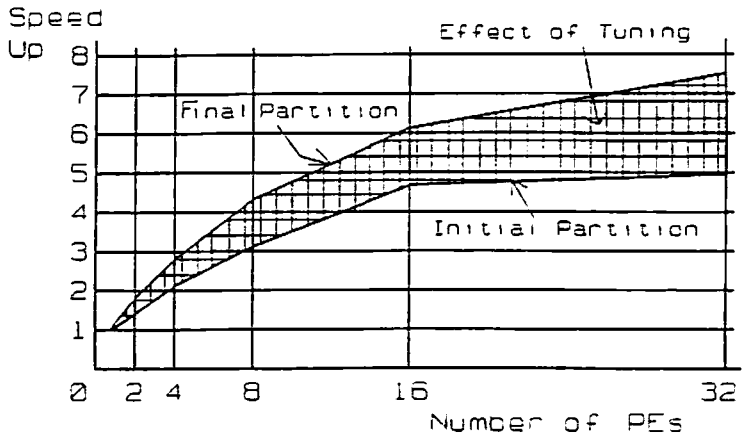


Fig. 9 Evaluation Results

- Our initial evaluation results on the original Manhattan Mapper show only a 15% reduction in production cycles resulting from parallel execution, and that up to 2 rules are fired in parallel. This result was contrary to our expectations. After further investigation of the original Manhattan Mapper, we recognized the system had been made suitable for sequential execution on single processor systems; i. e., *the execution order of rules are carefully pre-specified by embedded control information*.

- To reveal the potential parallelism in the Manhattan Mapper, we then reconstructed the system by analyzing its data dependency graph and by minimizing synchronization among rules. New results show more than an 85% reduction in production cycles and up to 13 parallel firings were achieved on some cycles of execution. Fig. 9 shows that a speed up by a factor of 7.5 is expected on 32 processor systems. *We stress that this number shows only the effect of parallel firings, i.e. it does not includes the effects of parallel matchings, which may compress the production cycle itself.* Thus more speed up than described in Fig.9 can be expected as the total effect of parallel execution.

- The evaluation is performed by gradually adding execution traces. Evaluation results on the initial partition, obtained from one execution trace, and the final partition, obtained from several additional execution traces are shown in Fig. 9. Though the final partition cannot be guaranteed to be the optimal partition, we report that better partitions than the final one are difficult to obtain even by human efforts. This result shows not only that the effect of tuning is quite large, but also that our incremental approach is efficient enough to get a satisfiable partition.

6 Conclusion

We proposed a new parallel execution model of production systems, the parallel firing mechanism, and discussed two major problems on the model, the synchronization problem and the decomposition problem. We also reported our simulation environment to evaluate the effectiveness of the model and obtained very promising results.

However, further study is underway to clarify the effectiveness of the parallel execution of production systems. The difficulty of this evaluation is that the true number cannot be obtained from a surface analysis of existing production systems. As we showed in this paper, even if existing production systems seem not to contain much parallelism, it is quite possible to reconstruct these systems to be suitable for parallel execution. Careful analysis of the original problem is necessary to determine the real effectiveness of parallel execution.

There remains the important research problem to find methods to minimize the amount of inter-processor communication on various kinds of architectures, and to find a suitable architecture for parallel firings of production systems. Our initial thinking on these problems for tree-structured machines is presented in [4] and [17]

At Columbia University, a parallel tree-structured machine, called DADO [6], is presently under construction to provide a parallel processing environment for production systems, logic programs and speech understanding tasks. Our next step is to implement this approach combined with various parallel match algorithms in the actual DADO environment.

Acknowledgments

We would like to thank Dan Miranker and Mark Lerner for their comments on an earlier version of this paper.

References

1. Stolfo, S. J. and Vesonder, G. T., "ACE: An Expert System supporting analysis and management decision making", Bell System Technical Journal, 1984.

2. McDermott, J., "R1: The Formative Years", AI Magazine, 2, 1981.

3. Forgy, C. L., Gupta, A., Newell, A. and Wedig, R., "Initial Assessment of Architectures for Production Systems", Proc. of the National Conference of Artificial Intelligence, 1984.

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

5. Miranker, D. P., "Performance Estimates for the DADO Machine: A Comparison of TREAT and RETE", Proc. of FGCS-84, 1984.

6. Stolfo, S. J. and Shaw, D. E., "DADO: A Tree Structured Machine Architecture for Production Systems", Proc. of the National Conference of Artificial Intelligence, 1982.

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

8. Oflazer, K., "Partitioning in Parallel Processing of Production Systems", Proceedings of International Conference on Parallel Processing, 1984.

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

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

11. Nilsson, N. J., Principles of Artificial Intelligence, Tioga, Palo Alto, Calif., 1980.

12. Tsuchiya, M. and Gonzalez, M. J., "Toward Optimization of Horizontal Microprograms", IEEE Trans. Comput., vol. C-25, pp.992-999, Oct. 1976.

13. Isoda, S., Kobayashi, Y. and Ishida, T., "Global Compaction of Horizontal Microprograms Based on the Generalized Data Dependency Graph", IEEE Trans. Comput., vol. C-32, pp.922-933, Oct. 1983

14. Kuck, D. J., Muraoka, Y. and Chen, S., "On the Number of Operations

Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup", IEEE Trans. Comput., vol. C-21, pp.1293-1310, Dec. 1972.

15. Shapiro, H. D., "A Comparison of Various Methods for Detecting and Utilizing Parallelism in a Single Instruction Stream", Proc. of International Conference on Parallel Processing, pp.67-76, 1977.

16. Lerner, M. and Cheng, J., "The Manhattan Mapper Expert Production System", Technical Report, Department of Computer Science, Columbia University, 1983.

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