Toward Semantic-Based Exploration of Parallelism in Production Systems *

Shiow-yang Wu, Daniel P. Miranker, and James C. Browne

Department of Computer Sciences
The University of Texas at Austin
Austin,TX 78712-1188

TR-94-23 October 1994

Abstract

We propose a new approach for the parallel execution of production system programs. This approach embodies methods of decomposition abstraction using declarative mechanisms. Application semantics can then be exploited to achieve a much higher degree of concurrency. In this paper we present the underlying object-based framework of production systems and discuss the ensuing semantic-based dependency analysis technique. In particular, we define a new notion of functional dependency to characterize associative relationships among data objects, which can be used to determine concurrently executable rules.

A byproduct of this research is a new technique for rapid system development and evaluation. The technique eliminates the possible inaccuracy of simulation as well as the high cost of full-fledged system implementation.

1 Introduction

A *production system* is composed of a *working memory*, a set of *rules*, and an *inference engine*. Working memory is a global database composed of data objects called *working memory elements* (WMEs) representing the system state. A rule is a condition-action pair. The inference engine provides a three-phase cyclic execution model of *condition evaluation (matching)*, *conflict-resolution* and *action firing*. A rule with a set of WMEs satisfying the conditions is called an *instantiation*. The set of all instantiations constitutes the *conflict set*. In a sequential environment, conflict-resolution selects one instantiation from the conflict set for firing. In a parallel environment, multiple rule instantiations can be selected for firing simultaneously subject to proper correctness constraints. Firing an instantiation executes the selected actions that may add, delete, or modify WMEs in the working memory. The cycle then starts over again.

Initial implementation of production systems suffered from poor performance which prohibited their use in large scale applications [7]. On the other hand, production systems have been assumed to encompass a high degree of parallelism [9], opening the opportunity of performance improvement through parallel processing. However, after over a decade of extensive research effort [19, 30], the speedup achieved by systems with real implementation is quite limited, only about 10-fold, no matter how many processors are used.

In a recent paper [33], we analyzed several commonly used benchmark programs and pointed out that the rather limited success in the past was primarily due to the failure to properly exploit parallelism embedded in the application domains and program structures. Contrary to the conclusion drawn by previous work that the true performance gain from parallelism is quite limited [10], we showed that massive and scalable speedup was indeed achievable with a set of explicit parallel structuring mechanisms.

*This work is partially supported by DARPA under grant DABT63-92-C-0042.

In this paper, we propose a semantic-based approach for the analysis of rule interference based on associative relationships among data objects. First, a general object-based framework of production systems is proposed both for a formal basis and for the expression of parallelism in a language independent way. Then, a notion of *functional dependency* is introduced to derive information about whether a rule is self-interfering and about the interference between different rules. We show how the combination of explicit parallel structuring mechanisms and semantic-based analysis technique achieve much higher level of concurrency than traditional techniques.

For early evaluation of the effectiveness of our approach without the high cost of implementing a full-fledged system, we built a parallel rule execution engine and the associated work load generator. The execution engine actually fires multiple rules simultaneously on a shared memory multiprocessors. All synchronization and communication operations necessary for the correctness of multiple rule firing are actually performed. The load generator generates work load from sequential execution trace files. With this new technique we can experiment with alternative implementation strategies in early stage of the system development and have accurate pictures of the run-time system behavior with much less initial implementation and evaluation cost. Replacing the load generator with a parallel match engine gives us a full-fledged system.

On the parallel execution engine, we conducted several sets of experiments on three commonly used benchmark programs. We show how granularity and scheduling strategies can significantly affect the performance of a parallel rule system.

2 Related Work

Early research on parallel production systems focused almost exclusively on *parallel matching* [8, 32, 9, 22, 10, 14]. These systems parallelized only the match phase. The speedup is therefore limited by the sequential execution of rules. *Multiple rule firing systems* parallelize not only the match phase, but also the act phase by firing multiple rules in parallel [13, 12, 17, 18, 28]. Some systems even fire rules asynchronously [29, 16]. Compile-time syntactic analysis of *data dependency graph* [13] is used to detect possible interference between rules. Instantiations of *compatible rules* [18] can be fired in parallel. For dependencies that can not be resolved at compile-time, run-time analysis is applied to increase the parallelism. The *copy-and-constraint* (C&C) technique proposed by Pasik [27] proved to be quite effective in reducing the variance in rule processing time and improving the parallelism as well. However, the rules and data attributes for applying C&C are selected manually.

All techniques above are domain insensitive since parallelism specific to the application domains is not exploited. The benefit of firing multiple rules can easily be overwhelmed by the cost of synchronization and run-time interference analysis [25]. As a result, only limited speedup was achieved.

On the other hand, the SPAM/PSM system [11] exploits *task-level parallelism* which is essentially functional decomposition of the original problem into a hierarchy of tasks and subtasks. The PARULEL language [31] employs a meta-level rule system to select compatible rule instantiations for parallel execution. These systems achieved better results by exploiting application specific parallelism. However, the techniques employed tend to be ad hoc or incur excessive overhead.

Our main contribution is to provide domain independent abstraction mechanisms and semantic-based analysis techniques which effectively exploit application parallelism without the high cost of run-time interference detection or instantiation selection.

3 A General Object-Based Framework of Production Systems

The concepts and techniques presented in this paper are language independent. We propose a general object-based framework to abstract away minor details and to capture just the essential features of production systems. Thus the results presented in this paper are generally applicable to any rule language.

3.1 Object Model

We have built our framework on top of a unified object model which can be used to characterize all entities in a rule system. The basic object model is inspired by [1, 4] and is comprised of the following sets of symbols:

$\mathcal{A}$: attribute names,

$\mathcal{C}$: class names,

$\mathcal{I}$: identifiers,

$\mathcal{M}$: method names,

$\mathcal{R}$: rule names,

$\mathcal{V}$: variable names.

Definition 1 (Method) A *method definition* is a triple (M, P, B) where M is a method name, P is a set of parameter specifications, and B is the definition of operations performed by the method. A *method invocation* is a method name with necessary parameters fully supplied. □

We have deliberately left out the details of how a parameter or body of a method is actually specified. Nor do we restrict the way actual arguments are passed in a method invocation. These issues are not essential to our discussion.

Definition 2 (Class) A *class* defines a set of objects with similar structure and behavior.

- **INT**, **FLOAT**, and **STRING** are *primitive classes* representing the set of integers, floats, and character strings, respectively.
- An *attribute definition* is a pair (a, C) where a is an attribute name and C is a class name.
- A set-valued attribute is defined by adding a $*$ at the end of an attribute name.
- A class is a triple (C, A, M) where C is a class name, A is a set of attribute definitions, and M is a set of method definitions. □

Definition 3 (Object and WME) *Objects* are defined to model WMEs. They are the basic units of information and behavior encapsulation.

- Integers, floats, and character strings are *primitive objects*.
- If $a_1, a_2, \ldots, a_n$ are the attribute names of a class C and $O_1, O_2, \ldots, O_n$ are objects, then:

  $$O = (\ a_1 : O_1,\ a_2 : O_2,\ \ldots,\ a_n : O_n\ )$$

  is a *structural object* or simply a *tuple*. The object is an *instance* of the class C.
- Tuples are the generalization of WMEs. Each tuple has a unique identifier associated with it. *Working memory* is a set of tuples.
- If $O_1, O_2, \ldots, O_n$ are objects of a class C, then

  $$\{\ O_1, O_2, \ldots, O_n\ \}$$

  is a *set object*. O_i's are *elements* of the set object. Note that elements of a set object must be instances of the same class. □

Definition 4 (Rule) A *rule* is a condition-action pair. Conditions can be positive or negative.

- An *expression* is a quantifier-free first order formula.

- If v is a variable name, C is a class name and E is an expression, then $(v : C :: E)$ is a *positive condition* and $-(v : C :: E)$ is a *negative condition*.
- If P is a condition, then $v(P)$, $C(P)$, and $E(P)$ denote the variable, class, and expression components, respectively, of the condition.
- A *rule* is a triple (P, N, M) where P is a non-empty set of positive conditions, N is a set (possibly empty) of negative conditions, and M is a set of method invocations.
- A positive or negative condition is termed a *condition element*. The set of all condition elements is called the *antecedent*. The set of method invocations is called the *consequent*. □

Definition 5 (Program and System) A *program* is a pair **(C, R)** where **C** is a set of class definitions and **R** is a set of rule definitions. A *rule system* is a pair **(O, P)** where **O** is a set of tuples and **P** is a rule program. □

3.2 Execution Model and Semantics

We characterize the semantics by considering rule antecedents as queries to the working memory for selecting a consistent set of objects. The execution of a rule system is defined in terms of state transitions between working memory states.

Definition 6 (State) The *state* of a rule system is the set of tuples in working memory. □

Definition 7 (Instantiation) Pattern matching is modeled by object selection. The following definitions are defined assuming a given state S.

- A positive condition element $(v : C :: E)$ is *satisfied* in S if there exists an object of class C such that E is evaluated to true. The object (which can be referenced by the variable v) is said to be *selected* by the condition element.
- A negative condition $-(v : C :: E)$ is satisfied in S if there does not exists any object of class C such that E is evaluated to true.
- A rule is satisfied in S if there exists at least one set of objects in S such that all condition elements in the antecedent are satisfied. The set of objects selected by the positive condition elements is called an *instantiation* of the rule. The set of all instantiations of a rule r is denoted by $\mathbf{Inst}(r)$. □

Operationally, a rule can be considered as a query to the working memory. The result of the query is a class whose instances are instantiations of the rule.

Definition 8 (Rule Firing) If S is a state, r is a rule which is satisfied in the state, the result of *firing* the rule is a new state S' obtained from S by invoking the methods in the consequent of r on the set of objects i which is an instantiation of r. We denote such a rule firing by $S' = S(i)$. □

Definition 9 (Execution) An *execution* of a rule system is a sequence of rule firings that transforms the system from a state to another state. A state is a *terminal state* if no rule is satisfied under that state. An execution is a *terminal execution* if the last state in the sequence of rule firings is a terminal state. □

It is important to note that in the definitions of rule firing and execution, no restriction is placed on how objects are selected or on which rule instantiation to pick. In other words, no matching technique or conflict resolution strategy are assumed. An execution is not required to be a terminal execution. Thus allowing nonterminating systems.

The framework and execution model above characterize the core concepts and essential features of a sequential production system. We now extend the model to allow simultaneous firing of multiple rule instantiations.

Definition 10 (Interference) If i_1 and i_2 are instantiations of two (possibly the same) rules that are satisfied in a state S, then i_1 *interferes with* i_2 if any one of the following conditions is true:

1. The execution of i_1 prevents i_2 from being an instantiation in the new state resulting from i_1's execution, or vice versa.

2. There exists methods invoked by i_1 and i_2 that modify the same object.

Since a newly created object is always assigned a unique identifier, object creations do not contribute to any interference except when Condition 1 is true. Identical objects with different identifiers are allowed to coexist in our model, which is consistent with most rule languages.

We note that it is possible to weaken Condition 2 above since we need only to avoid conflicting methods to be invoked on the same object. However, such fine-grained parallelism can be easily overwhelmed by the potential complexity. We reserve this issue for future research.

Definition 11 (Compatibility) Two instantiations are said to be *compatible* if they do not interfere with each other. A set of instantiations is compatible if the instantiations are pair-wise compatible. □

Since compatible instantiations do not interfere with each others, they can be executed in parallel. Our definitions of interference and compatibility are similar to the corresponding definitions in [13, 16, 18, 28] which are all essentially originated from database concurrency control theory [2]. However, we extend it to a general object-based context which allows any type of method instead of just the add, delete, and modify operations as in most previous work on parallel production systems.

Definition 12 (Parallel Rule Firing) The result of *parallel firing* of two compatible instantiations in a state is a new state obtained by invoking all methods on corresponding objects of the two instantiations. Likewise, the parallel firing of a set of compatible instantiations I in a state S is to invoke all methods on corresponding objects of all instantiations. The parallel firing is denoted by $S' = S(I)$. □

Because of the non-interference requirement between parallel executable instantiations, the resulting state of the parallel firing is the same as the result of execution of the set of instantiations in sequence following any order. To state it more precisely, if $I = \{i_1, \ldots, i_n\}$ is a set of parallel executable instantiations in a state S, then

$$S(I) \;=\; S(i_{j_1})(i_{j_2}) \ldots (i_{j_n})$$

where $j_1, j_2, \ldots, j_n$ is any permutation of n.

4 Class Relationships and Rule Compatibility

Under the general framework presented in last section, we now describe our semantic-based interference analysis technique. Our approach is motivated by the observation that, more often than not, class relationships provide valuable hints on data decomposition patterns that actually happen at run time but are not necessarily clear at design or compile time. We show that this information can often be used in determining the semantic compatibility (i.e. parallel executability) of instantiations of the same rule or between different rules.

As an intuitive example, consider the following rule from the corporation application domain.

rule $Team\_Fairness$ {
$(\ t : Team\ ),$
$[\ e : Employee :: e.team == t.name\ \wedge\ e.salary < t.min\_wage\ ]$
$\rightarrow$
$e.salary = t.min\_wage$
}

As we presented in [33], a positive condition enclosed in square brackets is a *set selection condition* which is to select all objects satisfying the condition. The rule is to raise the salary of all under-paid employees in a team. In general, different instantiations of this rule can not be executed in parallel because the same employee may be a member of different teams. On the other hand, if each team is associated with a unique and disjoint set of employees, then different instantiations will select different teams with disjoint set of employees. Apparently, all such instantiations can be fired in parallel. The key point is on the relationship between instances of the *Team* and the *Employee* class. We call such relationship *functional dependency* which is formally characterized in the following definitions.

Definition 13 (Class Relation and Scheme) A *class relation scheme* (or simply *scheme*) is an ordered set of class names. A *class relation* on a class relation scheme with n class names is an n-ary relation among instances of the corresponding classes. □

For a class relation A, we denote the scheme on which A is defined by $Sch(A)$. A class relation can be considered as a collection of classes with certain relationship. Note that an element of an n-ary class relation is an ordered set of n objects, one from each corresponding class in the scheme. An object here can be either a structural object or a set object. For an element $a \in A$ and a scheme $X \subseteq Sch(A)$, the notation $a(X)$ denotes the ordered collection of objects in a which are from classes in X. We note that from the definition above, $a(X) \subseteq a$ and $a(Sch(A)) = a$.

Since an instantiation can also be considered as an ordered set of objects (one for each positive or set selection condition), a rule r actually defines a class relation whose elements are exactly the set of instantiations of the rule, i.e. $\mathbf{Inst}(r)$. The scheme of $\mathbf{Inst}(r)$, denoted by $\mathbf{Scheme}(r)$, is the ordered set of class name components of positive and set selection conditions of r.

Definition 14 (Functional Dependency) Let X and Y be the schemes of two class relations R_x and R_y. The *functional dependency*

$$X \rightarrow Y$$

holds on R_x and R_y if

1. Each element in R_x is associated with a unique element in R_y.
2. For all a_1, a_2 in R_x and the associated b_1, b_2 in R_y,

$$a_1 \neq a_2 \Rightarrow b_1 \cap b_2 = \emptyset. \quad \square$$

As an example, in the corporation application domain discussed earlier, the functional dependency $\{Team\} \rightarrow \{Employee\}$ holds because each team is associated with a unique and disjoint set of employees.

Definition 15 (Rule Specific Functional Dependency) Let X and Y be two class relation schemes, and r be a rule. The *rule specific functional dependency*

$$X \rightarrow Y \ in \ r$$

holds if both $X \subseteq \mathbf{Scheme}(r)$ and $Y \subseteq \mathbf{Scheme}(r)$ and for all i, j in $\mathbf{Inst}(r)$,

$$i(X) \neq j(X) \Rightarrow i(Y) \cap j(Y) = \emptyset. \quad \square$$

It is *rule specific* because the dependency only needs to hold on all instantiations of r. It may or may not hold on collections of objects that are not instantiations of r.

Definition 16 Let R be a set of rules, X and Y be two class relation schemes, then

$$X \rightarrow Y \ in \ R$$

holds if $X \rightarrow Y \ in \ r$ holds for each rule r in R. □

Except for borrowing the terminology, functional dependency as defined here is quite different than in databases [21]. In database systems, the notion of functional dependency is defined at the attribute level and is used primarily in the normalization process. We generalize the concept to the class level and use it to identify the parallelism in rule systems. Functional dependencies are considered as specifications of data decomposition across class boundaries, which are shown below to play a crucial role in determining the compatibility between instantiations of the same or different rules.

Definition 17 Let r be a rule. The *access set* of r, denoted by $\mathbf{Access}(r)$, is the set of all class names referenced in the antecedent of r. The *write set* of r, denoted by $\mathbf{Write}(r)$, is the set of class names with objects that are modified (including creation and deletion) in the rule. □

Definition 18 (Dominant Set) Let r be a rule and C be a class relation scheme. C is a *dominant set* of r if:

1. $C \subseteq \mathbf{Scheme}(r)$,
2. for all $i, j \in \mathbf{Inst}(r)$ $(i \neq j \Rightarrow i(C) \neq j(C))$. □

A dominant set of a rule is simply a set of class names sufficient to discriminate between different instantiations of the rule.

Theorem 1 (Self Compatibility) *Let r be a rule and A, B, C be three class relation schemes that are subsets of* $\mathbf{Scheme}(r)$ *satisfying the following conditions:*

1. *C is a dominant set of r and $C \subseteq A$*
2. *$A \rightarrow B$ or $A \rightarrow B$ in r*
3. *$\forall c \in \mathbf{Write}(r)(c \in B \vee c \notin \mathbf{Access}(r))$*

then all instantiations of r are compatible (i.e. parallel executable).

Proof sketch: In any given state, let i and j be instantiations of r such that $i \neq j$.

	$i \neq j$	
$\Rightarrow$	$i(C) \neq j(C)$	(* C is a dominant set *)
$\Rightarrow$	$i(A) \neq j(A)$	$(* \ C \subseteq A \ *)$
$\Rightarrow$	$i(B) \cap j(B) = \emptyset$	(* Condition 2 *)
$\Rightarrow$	$i(\mathbf{Write}(r)) \cap j(\mathbf{Write}(r)) = \emptyset$	(* Condition 3 *)
$\Rightarrow$	i and j do not interfere with each other	(* Condition 3 and Definition 10 *)
$\Rightarrow$	i and j are parallel executable	(* Condition 3 and Definition 10 *). □

The central idea of this theorem is that functional dependency implies disjoint decomposition of objects selected by the instantiations of a rule. As long as the objects modified in the consequent belong either to the decomposition or to classes which do not affect the satisfiability of the rule, no instantiations will interfere with each others. We will have examples later in this section. We first generalize this idea to the analysis of interference between multiple rules.

Definition 19 (Partially Mutual Exclusion) Let p, q be rules and C be a class relation scheme. We say that p and q are *partially mutual exclusive on C*, denoted by $p \gtrless_C q$, if

1. $C \subseteq \mathbf{Scheme}(p)$ and $C \subseteq \mathbf{Scheme}(q)$
2. For any two instantiations i, j of p and q respectively, $i(C) \neq j(C)$. □

Partially mutual exclusion simply means that p and q can not have instantiations containing the same set of objects of classes in C. The simplest and most common case is when C contains a single class referenced in both p and q but tested on disjoint values of the same set of attributes. Since the values are disjoint, p and q can not select the same object in C.

Note that no requirement is placed on selected objects that are not of the classes in C. Therefore, partially mutual exclusive rules may still interfere with each other. However, in many cases, partially mutual exclusive rules can be determined to be parallel executable with the help of functional dependencies as indicated by the following theorem.

Theorem 2 (Pair-Wise Compatibility) *If p, q are two distinct rules, and A, B, C are class relation schemes that are subsets of both* **Scheme***(p) and* **Scheme***(q) such that the following conditions are satisfied:*

1. $p >\!\!<_C q$ *and* $C \subseteq A$
2. $A \rightarrow B$ *or* $A \rightarrow B$ *in* $\{p, q\}$
3. $\forall c \in \mathbf{Write}(p)(c \in B \vee c \notin \mathbf{Access}(q))$
4. $\forall c \in \mathbf{Write}(q)(c \in B \vee c \notin \mathbf{Access}(p))$

then p and q are compatible and therefore parallel executable.

Proof sketch: In any given state, let i be an instantiation of p and j be an instantiation of q.

	$p >\!\!<_C q$	
$\Rightarrow$	$i(C) \neq j(C)$	(* Definition 19 *)
$\Rightarrow$	$i(A) \neq j(A)$	$(* \; C \subseteq A \; *)$
$\Rightarrow$	$i(B) \cap j(B) = \emptyset$	(* Condition 2 *)
$\Rightarrow$	$i(\mathbf{Write}(p)) \cap j(\mathbf{Write}(q)) = \emptyset$	(* Condition 3 and 4 *)
$\Rightarrow$	p and q are parallel executable	(* Definition 10 *). $\square$

Again, the central idea of this theorem is that as long as objects modified in p and q can be determined as non-overlapping with the help of functional dependency, instantiations of p and q do not interfere with each others.

Even with their general applicability to many cases, the two theorems above are less complicated than they appear. Continuing with our examples in the corporation application domain, if a team is associated with a set of disjoint employees as team members, then the functional dependency $\{Team\} \rightarrow \{Employee\}$ holds. We note that this semantic information can be easily supplied by the programmer (similar to the identification of key attributes in database systems). With functional dependency and the fact that a team can be uniquely identified by its name, we can immediately determine that all instantiations of the *Team_Fairness* rule can be fired in parallel using Theorem 1.

As another example, the following two rules can be determined to be parallel executable by Theorem 2.

rule $Facilities\_Research$ {
($t : Team :: t.dept == \text{“}research\text{”}$)
[$e : Employee :: e.team == t.name$]
$\rightarrow$
$e.equipment = \text{“}AXP500X(Alpha)\text{”}$
}

rule $Facilities\_Sales$ {
($t : Team :: t.dept == \text{“}sales\text{”}$)
[$e : Employee :: e.team == t.name$]
$\rightarrow$
$e.equipment = \text{“}PowerBook\text{”}$
}

In this case, the two rules are partially mutual exclusive on *Team*. With the help of functional dependency, they can be statically determined to be parallel executable.

As simple and natural as it may seem to be, without the knowledge of functional dependency between the *Team* and the *Employee* classes, it is very difficult, if not impossible, for a parallelizing compiler or any other static transformation technique to identify the parallelism underlying these rules.

In general, any type of class relationship which implies certain pattern of association or partitioning in the application domain is of great help in the determination of proper decomposition for parallel processing. Mechanisms for expressing these relationships are therefore of great value for capturing application parallelism in program development.

5 Effective Implementation Strategies

The new approach we propose can be summarized as follows.

- The central idea is to exploit application semantics and program structures. We found these to be the most valuable sources of parallelism in production systems.
- For the specification of application semantics, we proposed in [33] a set of parallel structuring mechanisms.
- In this paper, we introduce a semantic-based technique to identify parallel structure from functional dependency specification among data objects.
- All the mechanisms and techniques above are essentially to identify decomposition patterns and to raise the level of abstraction to the application domain instead of the implementation domain.

With this approach, we introduce a new stage into program development which we called "decomposition abstraction" [34]. We also showed in the paper that, with our mechanisms, programmers are completely alleviated from the need to worry about synchronization or communication details that are critical to most explicit parallel programming languages.

However, just like any explicit parallel languages, the actual performance gain depends heavily on the implementation strategy adopted to realize the parallelism expressed by the programmers. This is especially the case in production systems because rules tend to have large variation in processing requirements [10]. Systems that fire multiple instantiations in parallel tend to incur large run-time overhead [26]. Granularity control and proper scheduling strategy is therefore of crucial importance to a multiple rule firing production system with decomposition abstraction mechanisms. In this section, we discuss several alternative implementation strategies and their performance on a parallel rule execution engine we built to provide a test bed and a guide to our actual implementation of the proposed decomposition mechanisms.

5.1 A Parallel Rule Execution Engine

To be as close as possible to the real execution environment, we develop a parallel rule execution engine that actually executes multiple rule instantiations in parallel on our target machine, the Sequent Symmetry shared memory multiprocessor. A work load generator generates work from sequential execution trace files. This approach is unique in that, WMEs are actually added, removed and modified. All rule actions are executed as they would be in a real parallel environment. All communication and synchronization operations needed to maintain the correct parallel execution are actually performed. With the same set of data, the parallel rule execution engine will terminate with exactly the same result as in a sequential execution (to make sure that the execution is correct) except, of course, the execution time. The only important factor which is not accounted for is the time spent on matching. To include matching would make it a real parallel inference machine which will take much longer to develop before any experiment can be done on it. With the already existing good results on both sequential and parallel algorithms for matching [22, 23, 20], this exclusion is considered justifiable. Actually, the performance of the real system may be even better with parallel matching since the effect is at least additive, if not multiplicative.

Program	No. Rules	Description
LIFE	16	A simulation program implements Conrad's LIFE.
WALTZ	33	A constraint satisfaction problem using Waltz's algorithm for scene labeling.
MANNERS	8	A combinatorial search problem for seat assignment.

Table 1: Benchmark programs used in the experiments.

No. Iterations	Time (ms)
0	0.00
1000	2.57
3000	7.64
5000	12.71
7000	17.78
10000	25.38
20000	50.75

Table 2: Actual time delay introduced by the dummy loop.

The parallel rule execution engine is a set of programs built on top of a C++ based object-oriented light weight thread package called PRESTO [3]. A sequential rule program[1] and its execution trace are translated into a PRESTO program which is the parallel version of the program integrated with the PRESTO run-time libraries. To experiment with the effect of granularity of rules on system performance, a controllable dummy loop is added to the action part of each parallel rule in the PRESTO program. The time to execute the action part of a rule can then be controlled by varying the number of iteration for the dummy loop. Also by varying the number of threads created for processing parallel instantiations, and the granularity of work (number of instantiations) assigned to each thread, we have measured the performance of four alternative scheduling strategies.

- **Strategy 1: Maximal Parallelism** Create a new thread for each parallel instantiation.
- **Strategy 2: Fixed Granularity** Same as Strategy 1 except that each thread is assigned a fixed number of instantiations to execute sequentially. This is to increase the granularity of work assigned to each thread so as to reduce the total number of threads.
- **Strategy 3: Master-Slave** Create a fixed number of worker threads and a scheduler thread. The scheduler thread keeps dispatching instantiations, one instantiation for each worker, as long as there are idle workers. If no idle worker exists, the scheduler executes the instantiation itself.
- **Strategy 4: Master-Slave with Chunking** Same as Strategy 3 except that each time an idle worker is given a fixed number of instantiations to work on sequentially. The number is called the *chunk size* whose purpose is to increase the granularity of work assigned to a worker thereby reducing thread management and scheduling overhead.

To evaluate the effectiveness of different scheduling strategies, we collect the performance results from the execution of three benchmark programs drawn from the Texas OPS5 Benchmark Suite [5] and listed in Table 1. All three programs are executed with increasing number of processors on different problem sizes and different chunk sizes. We then compare their relative performance, scalability, as well as sensitivity to granularity change. The actual time delay introduced by the dummy loops are also measured and listed in Table 2.

[1] At this moment, the parallel rule execution engine takes only OPS5 programs. However, the same approach can be applied on any sequential rule language.

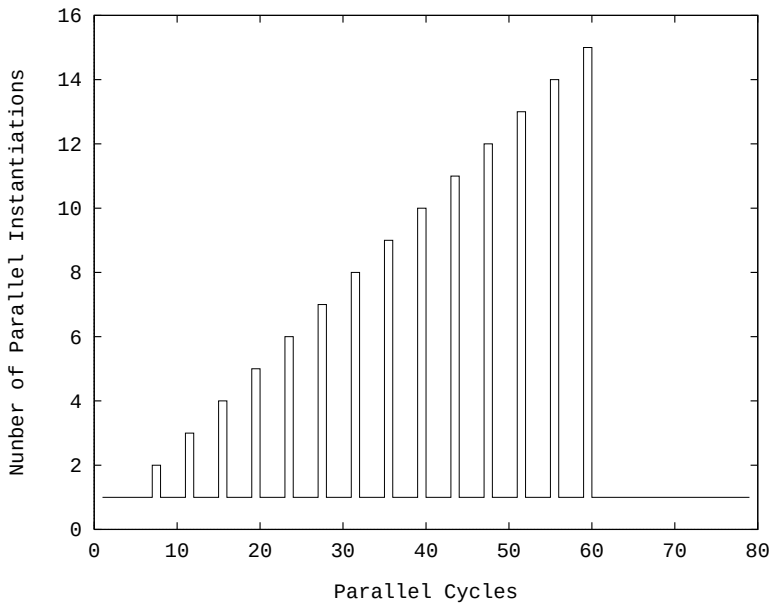

Figure 1: MANNERS16 Concurrency Profile.

5.2 Characteristics of the Benchmark Programs

In our approach, identifying the characteristics of an application is of crucial importance. In this section, we analyze the concurrent behavior of each benchmark program and point out key issues to the successful application of proposed mechanisms.

MANNERS

MANNERS was derived from an example program in [15] which employed a combinatorial search for solving a seat assignment problem among a number of guests. The seats must be assigned such that neighbors are of opposite sex and share at least one common hobby. This simple program, containing just 8 rules, is a very good test program for evaluating how effective a parallel production system is. It consists of a hot spot rule that fires repeatedly and consumes over 90% of sequential execution time for problem size of 64 guests or more. The larger the problem size, proportionally more time is consumed by this rule which is used to maintain partial solution. Although all instantiations of this rule can be fired in parallel, it is very difficult, if not impossible, to recognize this fact by pure syntactic technique at compile time without the information provided by our mechanisms. The concurrency exhibited is also very interesting. It is regular but not evenly distributed. The execution starts with a very low degree of parallelism which then gradually increases toward the end. More specifically, the program starts with only 2 instantiations that can be executed in parallel, then 3 instantiations, then 4, 5, ... etc. Figure 1 is the concurrency profile of MANNERS with 16 guests. This is highly challenging since a parallel production system must not only detect the hot spot rule, but also exploit effectively a rather peculiar pattern of parallelism.

LIFE

LIFE is a simulation program that simulates the existence of bacteria in a rectangular grid of cells for a specified number of generations. Whether a cell stays alive across a generation is determined by the number of neighbors it has. A living cell is born in an empty cell if it has exactly 3 neighbors. Since all decisions can be made locally, LIFE exhibits a high degree of data level parallelism. However, the available concurrency has not been effectively exploited in previous research. The difficulty of detecting it by syntactic analysis alone is again the key reason. Another probably even more important reason is that a parallel production system must have the ability to perform set-oriented and aggregate operations to exploit the available concurrency. Figure 2 is the concurrency profile of a 10x10 LIFE execution trace without showing the sequential printing

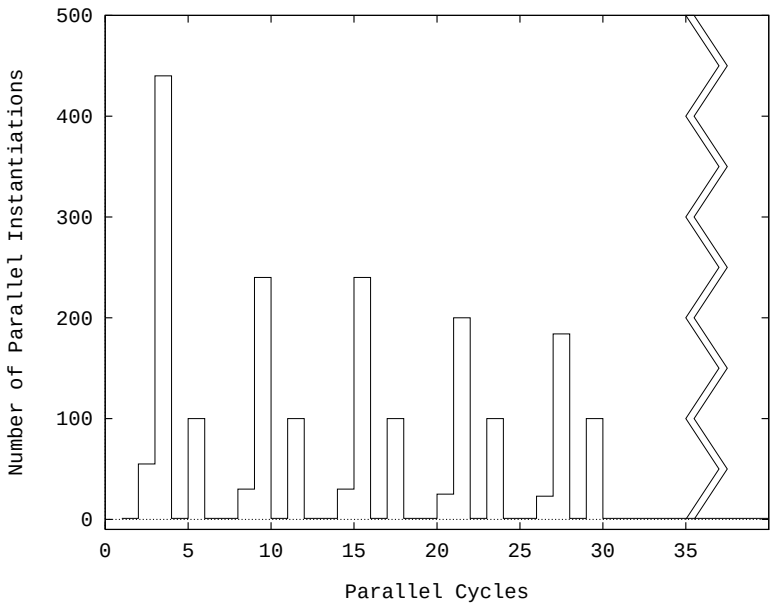

Figure 2: LIFE10 Concurrency Profile (without Showing Sequential Printing at the End of Execution).

at the end of execution. Because of the rather evenly distributed pattern of parallelism, keeping processors busy doing useful work at all time is the primary issue.

WALTZ

The frequently studied WALTZ program is also selected here to serve both as a test program to evaluate the effectiveness of our mechanisms and as a benchmark program to compare our results with others. This is a constraint satisfaction problem that implements Waltz's algorithm for labeling of line drawing scene. The algorithm propagates labels based on local decision and therefore exhibits both SPMD- and MIMD-style of parallelism (i.e. parallel instantiations of the same or different rules working on different part of the scene). The available parallelism is again quite high as depicted in Figure 3 which is the concurrency profile of a 10 regions execution trace with sequential printing of results at the end excluded.

Because, syntactically, a number of similar constraints each appears in many rules, the rules are highly interfering with themselves. Most of existing parallel production systems end up handling this at run-time resulting in excessive overhead. However, the constraints are disjoint, most of the run-time overhead is actually superfluous. The functional dependency declaration and our **Disjoint** combinators [33] express this to reveal a compile time parallel structure. A system capable of forming disjoint partition of consistent data objects can then effectively exploit this available concurrency without the overhead of run-time interference detection.

5.3 Performance Results

In this section, we demonstrate and analyze the performance results of three OPS5 benchmark programs on our parallel rule execution engine. Extensive experiments are conducted on various dimensions affecting the selection of implementation strategies. The speedup is measured against the execution time of the execution engine with single processor instead of an optimized uniprocessor OPS5 compiler such as OPS5c [24] because the former is two to three orders of magnitude faster than the later. The difference is primary due to the additional matching performed by the uniprocessor compiler. As a remedy to this lack of match phase, we artificially increase the per rule processing time using dummy loop as described earlier.

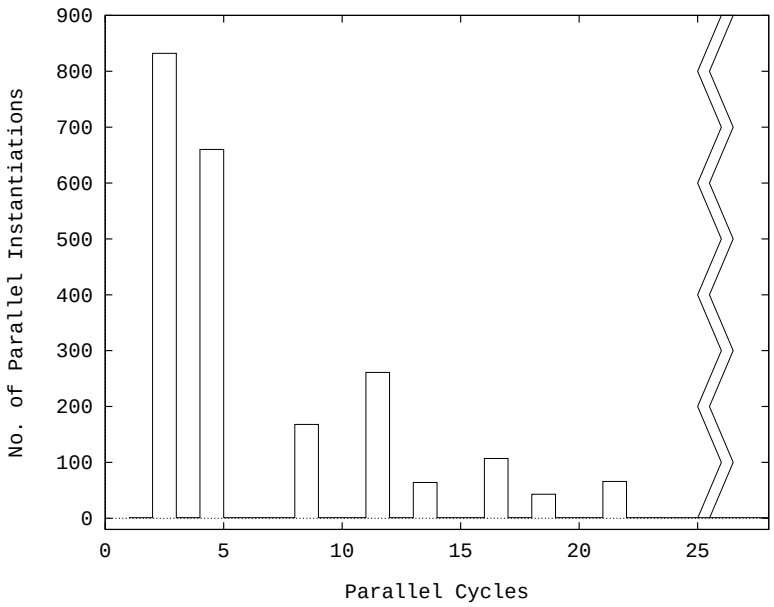

Figure 3: WALTZ10 Concurrency Profile (without Showing Sequential Printing at the End of Execution).

The Effect of Rule Granularity

To understand the effect of rule granularity (i.e. the time to process a rule) on the system performance, we select the master-slave with chunking scheduling strategy while varying the granularity of rule by changing the number of iterations in the dummy loop. Figure 4 shows the results on MANNERS512 (i.e. 512 guests). The performance improves significantly with larger granularity. Nearly ideal speedup is achieved when the granularity per rule is increased to 20000 (i.e. 50.75ms). As a comparison, the average cycle time of the same program and data set running under OPS5c on a much faster CPU (SUN4 workstation vs. the Intel 80386 on Sequent Symmetry) is about 210ms.[2] This implies that the overhead of scheduling and thread management is very low, and as long as we can keep it low in a real implementation it is very likely to get even better results since the granularity per rule is expected to be much higher than 50ms when matching is included.

Figure 5 and Figure 6 are the results of similar experiments on LIFE (40x40) and WALTZ (30 regions), respectively. In both cases, performance improvement is observed with increasing granularity.

Among all three test cases, LIFE achieves the highest speedup with granularity 20000. This is plausible because the run-time behavior of LIFE exhibits the highest and the most regular pattern of concurrency as depicted earlier in Figure 2. On the other hand, WALTZ requires larger granularity to achieve the same level of performance. We were puzzled by this unexpected result at first since from the characteristics of the Waltz's algorithm, there should not be that much difference. Later on we found that the available parallelism of a WALTZ program execution depends heavily on the data set (i.e. the scene to be label). The data generator we use (and used by other researcher as well) introduces a sequential factor that severely restricts the available parallelism. The generated scene consists of two arrays of rectangular blocks growing linearly according to the given problem size parameter. This linear factor contributes to the performance difference between WALTZ and the other two programs. We plan to develop a new data generator that generates scenes without this linear factor.

Scalability: The Effect of Problem Size

An important criterion when evaluating the effectiveness of a parallel system is its scalability. When the available parallelism increases, a parallel processing system must be able to effectively exploit it and achieve better performance. For the three benchmark programs, a common characteristic is that available parallelism

[2] For interesting readers, a LISP based implementation of OPS5 takes forever to run the program on the same data set.

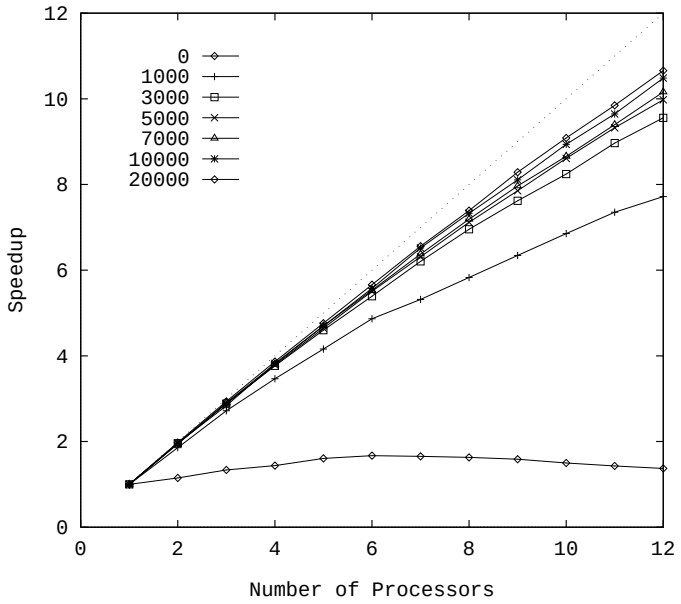

Figure 4: MANNERS512 Speedup with Varying Granularities.

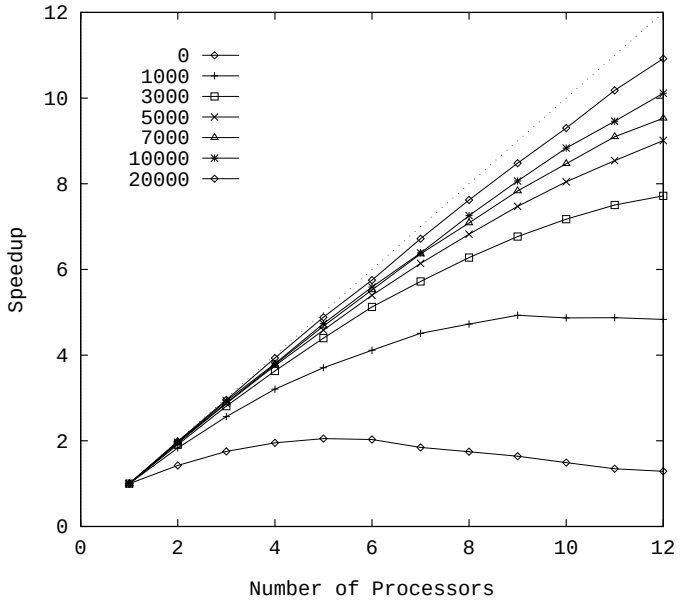

Figure 5: LIFE40 Speedup with Varying Granularities.

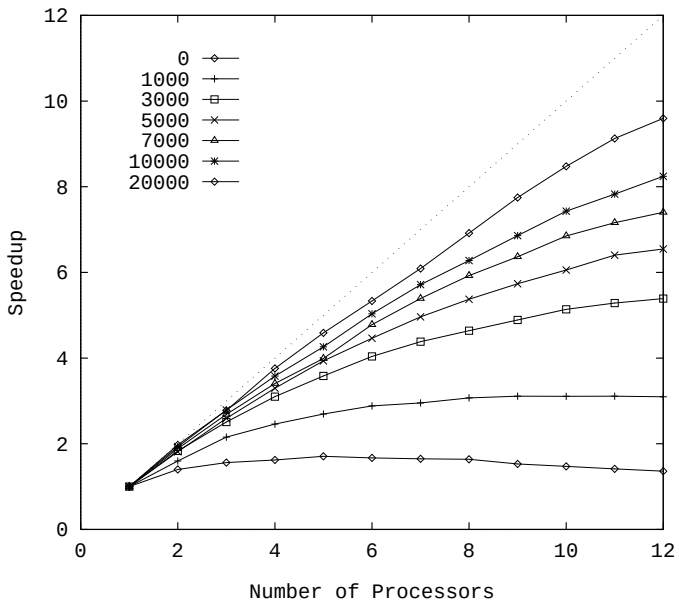

Figure 6: **WALTZ30** Speedup with Varying Granularities.

increases with the problem size. Therefore, we tested our mechanisms and system on three programs with increasing problem sizes where the problem size to MANNERS, LIFE and WALTZ are the number of guests to be assigned, the grid size, and the number of regions[3], respectively. All programs were tested under the master-slave with chunking scheduling strategy. The chunk size was set to 5 with rule granularity fixed at 20000 (i.e. 50.75ms).

Figure 7 illustrates the performance results of MANNERS on different problem sizes and vividly displays scalable speedup of our scheme. Figure 8 and Figure 9 are the results of similar experiments on LIFE and WALTZ. The speedup achieved on smaller problems is lower because the available parallelism is not enough to keep all processors busy. When problem size becomes larger, processor utilization increases and so is the speedup achieved.

Controlled vs. Unrestricted Parallelism

Too much water drowned the miller. If the available parallelism is not exploited appropriately, the benefit of parallel processing can easily be overwhelmed by the scheduling and synchronization overhead. In our case, since the embedded parallelism in the application is fully expressed, the key issue comes down to efficiently process the collection of parallel instantiations on available computation resources. For a thread-based implementation like ours, this issue manifests itself in a trade-off between parallel processing of as many instantiations as possible and controlling the number of concurrent threads. If a new thread is created for each parallel instantiation (i.e. Strategy 1), we get maximal parallelism on the one hand but highest thread management overhead on the other hand. Using the master-salves scheduling strategy, the number of threads is fixed but the communication and synchronization cost increase because of the need to partition and dispatch parallel instantiations to the worker threads.

To understand the effect of thread management overhead on system performance, we compare the results between applying Strategy 1 (maximal parallelism) and Strategy 4 (master-slave with chunking). Figure 10 is a 3-D display of two sets of experiments on WALTZ10. The timing curves on the base plane are the execution time while the B-spline surfaces are to demonstrate the performance differences. It is quite evident that master-salves with chunking outperforms maximal parallelism by a substantial margin. Figure 11 presents the results of similar experiments on LIFE30. The difference is smaller but still perceptible. This suggests that throttled parallelism is much better than unrestricted parallelism.

[3] In the scene generated by the data generator for WALTZ, a region consists of 72 line segments.

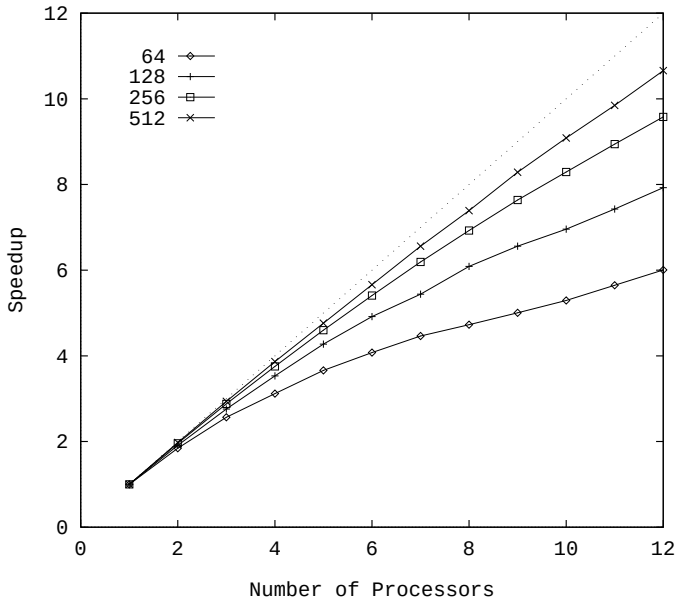

Figure 7: MANNERS Speedup on Different Problem Sizes.

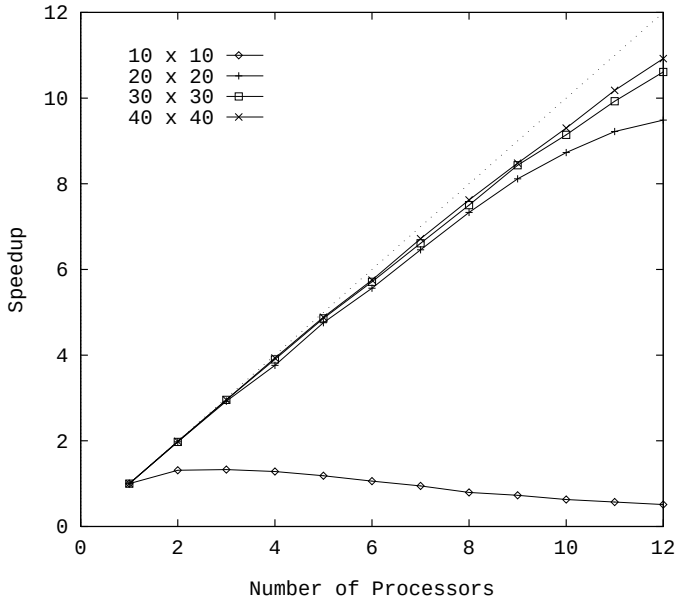

Figure 8: LIFE Speedup on Different Problem Sizes.

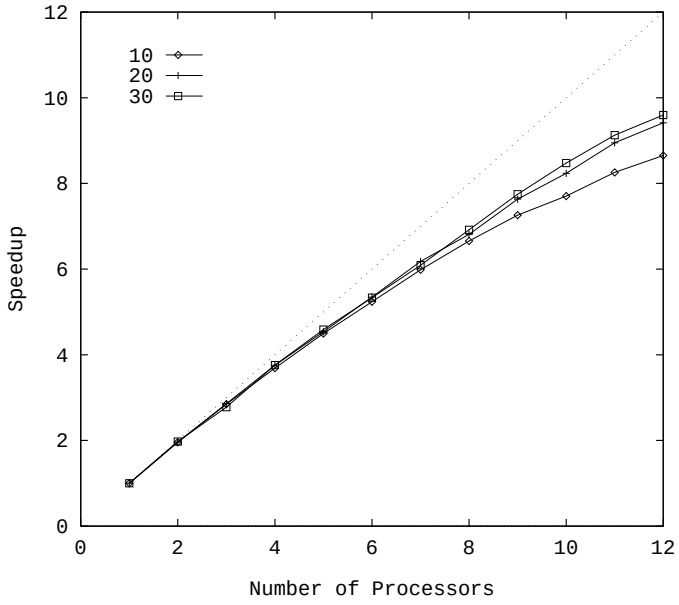

Figure 9: WALTZ Speedup on Different Problem Sizes.

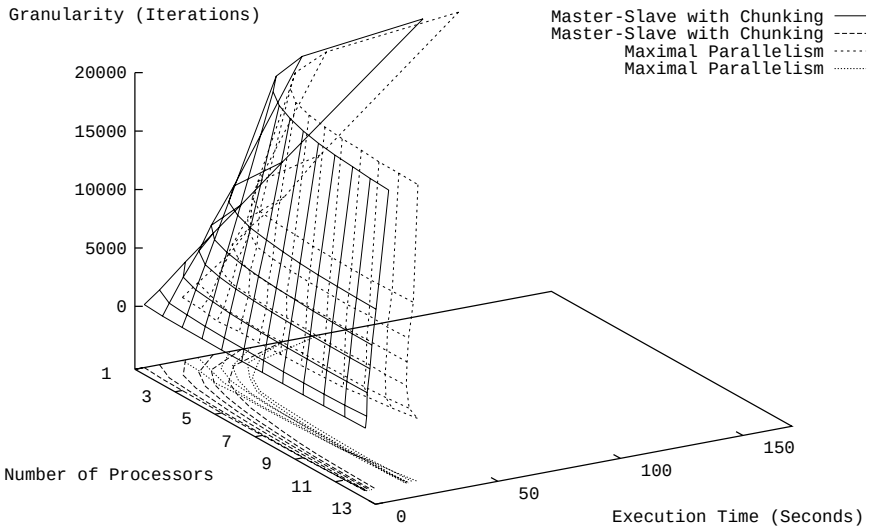

Figure 10: 3-D Display of Controlled vs. Maximal Parallelism on WALTZ10.

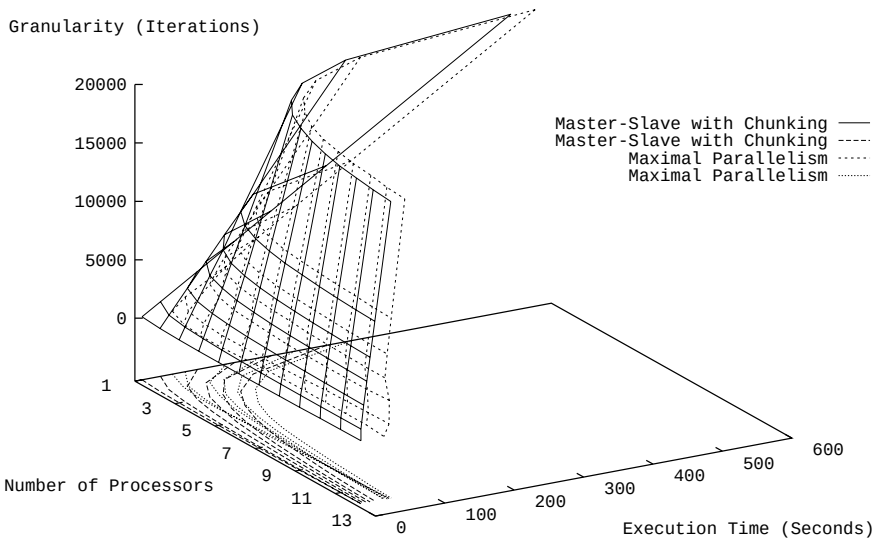

Figure 11: 3-D Display of Controlled vs. Maximal Parallelism on LIFE30.

Just when we expect to observe similar type of performance difference on MANNERS program, it does not happen to be the case. Figure 12 is the 3-D graph of the similar experiments as above. The performance of maximal parallelism is not only comparable to that of master-slave with chunking, it actually performs better when the granularity of the rules and the number of processors increase. A closer look at the problem reveals a pattern of concurrent behavior peculiar to the MANNERS program. As depicted in Figure 1, the number of parallel executable instantiations is quite small in the early stage of the execution. When chunking is applied, the available parallelism is left unexploited while under maximal parallelism strategy these instantiations are always processed in parallel.

As a summary, the master-slave strategy which creates only a limited number of threads is, in general, better than the maximal parallelism strategy which creates as many threads as the number of parallel instantiations. However, the actual performance gain still depends on the characteristics of the underlying program. When the degree of concurrency in an application is low, the maximal parallelism strategy is likely to be better.

The Effect of Chunking

In last section, master-salve with chunking appeared to be the winner in overall performance. The question of determining appropriate chunk size follows. It is unlikely to have a single chunk size that is optimal for every program. We need, at minimum, to determine if performance as a function of chunk size is behaved well enough to offer a system default. We test the benchmark programs with different chunk size of 1 (i.e. no chunking), 5, 10, and 20. Each one of them is tested with a fixed level of rule granularity. To understand the correlation of chunking with respect to rule processing time, the same set of experiments is carried out with different levels of rule granularity.

Figure 13 is the results on MANNERS512 with rule granularity set to 1000. We can observe quite clearly that chunking is always better than no chunking at this level of granularity. A chunk size of 5 provides the best performance. On the other hand, with the granularity of rule raised to 20000, the best performance is obtained when chunking is not applied as shown in Figure 14. Similar results can be observed on both LIFE and WALTZ except that chunk size of 5 is not necessary a clear winner over other chunk sizes. Because of the space limit, the results are not presented in here. See [34] for complete experiment results.

As a summary, reducing the granularity of rules improves the results we get from chunking. This is

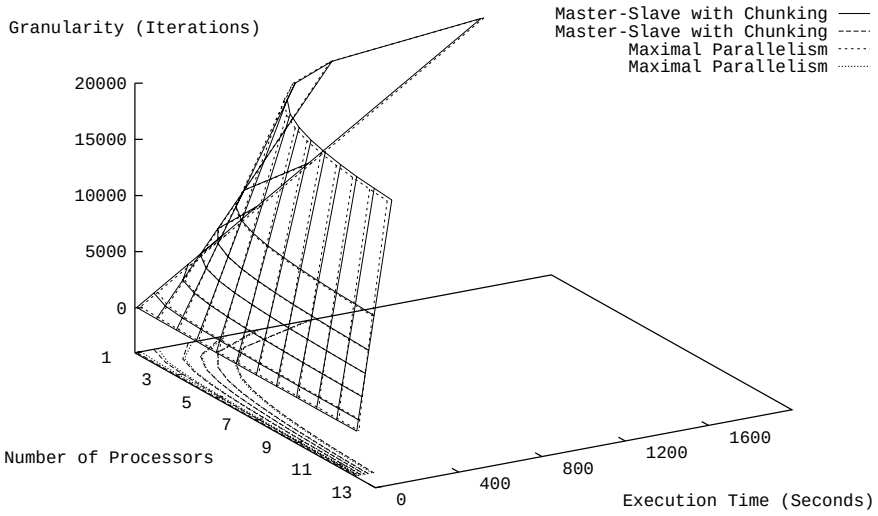

Figure 12: 3-D Display of Controlled vs. Maximal Parallelism on MANNERS256.

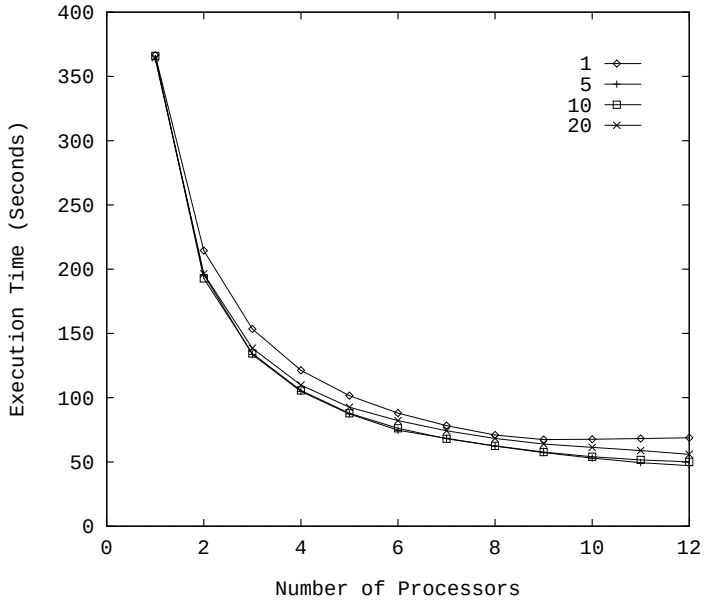

Figure 13: MANNERS512 Execution Time on Different Chunk Sizes with Rule Granularity Fixed at 1000.

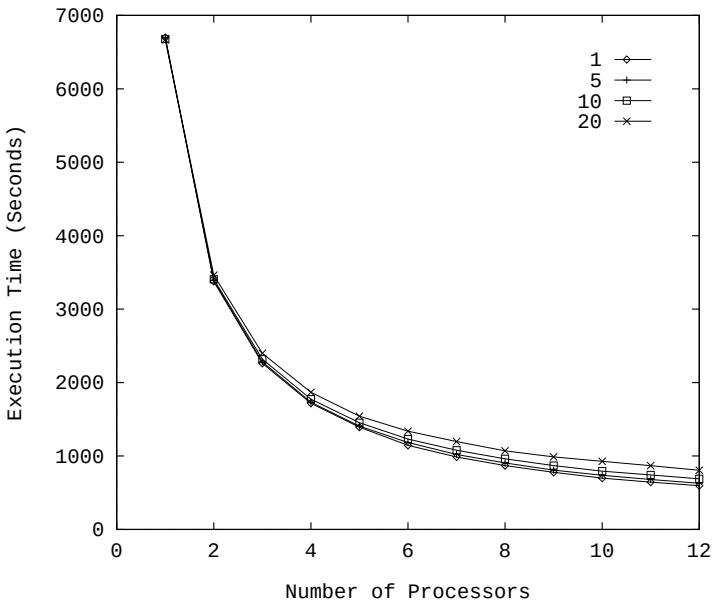

Figure 14: MANNERS512 Execution Time on Different Chunk Sizes with Rule Granularity Fixed at 20000.

primarily because of the reduction in thread management, communication, and synchronization overhead. However, when the average granularity of the rules or the chunk size become larger, the loss of parallelism offsets the benefit of chunking. In general, chunking with chunk size 5 provides the best performance when the rule granularity is smaller than 5000. When the granularity is larger than 5000, it is better to do without chunking. In other words, chunking is good for cases where the per rule scheduling overhead is comparable or larger than that of the granularity of rule. This result suggests a dynamic scheduling strategy with either user-specified granularity assignment to each rule or an automatic estimation done by the system. We are investigating this issue in our real implementation of an object-based parallel rule language equipped with decomposition abstraction mechanisms.

6 Conclusions and Future Work

We have shown in this paper that, contrary to the conclusion drawn by previous research, production systems and rule-based programs do encompass high degree of concurrency and it is possible to effectively exploit such level of parallelism. These results are obtained when a programmer is provided with decomposition abstraction mechanisms to specify the natural parallelism inherited in the application domain. The novel approach of using data relationships (functional dependency in particular) in the derivation of information for parallelism is a promising direction that has never been recognized before. Based on a new technique of rapid system development for early evaluation and experimentation, we show that the proposed set of mechanisms can be efficiently implemented on shared-memory multiprocessors.

Even with effective implementation, the granularity of rule is still critical to the actual performance gain. If this information is available to the run-time scheduler, then a strategy of master-slave with dynamic chunk size is very likely to perform better than the strategy with fixed chunk size or no chunking at all. Obviously, a user-specified granularity level assigned to each rule (similar to a priority assignment or certainty factor) is an immediate solution. We plan to provide this feature in our actual implementation. However, unlike other proposed mechanisms in our approach, proper granularity assignment is in general not readily available from the application semantics since it involves an estimation of processing time for each rule. An alternative way is to have the system estimate the right granularity for a rule either from its syntactic structure or from an execution profile.

We are embodying our decomposition abstraction mechanisms on the Venus/C++-based modular rule language [6] targeting Sequent Symmetry shared-memory multiprocessors. We choose Venus because it is

probably the first sequential rule-based programming language to provide both a declarative syntactic and semantic mechanism to support top-down modular design of rule-based programs. The embodiment of our mechanisms instantly converts the language into a parallel rule language, which is called Venus/DA. The language system is not only intended to serve as a test-bed for semantic-based exploration of parallelism in production systems, but also to be used for the planned extension of rule-based technology into database applications.

References

[1] Francois Bancihon and Setrag Khoshafian. A calculus for complex objects. In *Proc. 15th ACM Symp. on Principles of Database Systems*, pages 53–59, 1986.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[3] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. PRESTO: A system for object-oriented parallel programming. *Software - Practice and Experience*, 18(8):713–732, August 1988.

[4] Elisa Bertino, Mauro Negri, Giuseppe Pelagatti, and Licia Sbattella. Object-oriented query languages: The notion and the issues. *IEEE Trans. on Knowledge and Data Engineering*, 4(3):223–237, June 1992.

[5] David A. Brant, Timothy Grose, Bernie Lofaso, and Daniel P. Miranker. Effects of database size on rule system performance: Five case studies. In *Proc. 17th Intl. Conf. on Very Large Data Bases*, pages 287–295, 1991.

[6] James C. Browne and et. al. A new approach to modularity in rule-based programming. In *ICTAI'94: IEEE Intl. Conf. on Tools for Artificial Intelligence*, 1994.

[7] C.L. Forgy, Anoop Gupta, A. Newell, and R. Wedig. Initial assessment of architectures for production systems. In *Proc. 4th National Conference on Artificial Intelligence (AAAI-84)*, pages 116–120, Austin, TX, August 1984.

[8] Anoop Gupta. Implementing OPS5 production systems on DADO. Technical Report CMU-CS-84-115, Department of Computer Science, Carnegie Mellon University, Pittsburgh, December 1983.

[9] Anoop Gupta. *Parallelism in Production Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, Pittsburgh, March 1986.

[10] Anoop Gupta, Charles Forgy, and Allen Newell. High-speed implementation of rule-based systems. *ACM Trans on Computer Systems*, 7(2):119–146, May 1989.

[11] Wilson Harvey, Dirk Kalp, Milind Tambe, David McKeown, and Aleen Newell. The effectiveness of task-level parallelism for production systems. *Journal of Parallel and Distributed Computing*, 13(4):395–411, December 1991.

[12] T. Ishida. Methods and effectiveness of parallel rule firing. In *Proc. IEEE 6th Conference on Artificial Intelligence Applications*, pages 116–122, 1990.

[13] T. Ishida and S. Stolfo. Toward the parallel execution of rules in production system programs. In *Proc. IEEE Intl. Conf. on Parallel Processing*, pages 568–574, 1985.

[14] M.A. Kelly and R.E. Seviora. An evaluation of DRete on CUPID for OPS5 matching. In *Proc. 11th International Joint Conference on Artificial Intelligence (IJCAI-89)*, Detroit, MI, August 1989.

[15] G. Kiernan, C. de Maindreville, and E. Simon. Making deductive database a practical technology: A step forward. In *ACM SIGMOD Intl. Conf. on Management of Data*, pages 237–246, 1990.

[16] Chin-Ming Kuo, Daniel P. Miranker, and James C. Browne. On the performance of the CREL system. *Journal of Parallel and Distributed Computing*, 13(4):424–441, December 1991.

[17] S. Kuo, D. Moldovan, and S. Cha. Control in production systems with multiple rule firings. In *Proc. IEEE Intl. Conf. on Parallel Processing, 1990, Vol. II*, pages 243–246, 1990.

[18] Steve Kuo and Dan Moldovan. Implementation of multiple rule firing production systems on hypercube. *Journal of Parallel and Distributed Computing*, 13(4):383–394, December 1991.

[19] Steve Kuo and Dan Moldovan. The state of the art in parallel production systems. *Journal of Parallel and Distributed Computing*, 15(1):1–26, May 1992.

[20] Ho Soo Lee and Marshall I. Schor. Match algorithms for generalized Rete networks. *Artificial Intelligence*, 54(3):249–274, April 1992.

[21] David Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, Maryland, 1983.

[22] Daniel P. Miranker. *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. PhD thesis, Department of Computer Science, Columbia University, 1987. Also published by Morgan-Kaufmann, 1990.

[23] Daniel P. Miranker, David A. Brant, B.J. Lofaso, and David Gadbois. On the performance of lazy matching in production systems. In *National Conference on Artificial Intelligence*, pages 685–692, July 1990.

[24] Daniel P. Miranker and B.J. Lofaso. The organization and performance of a TREAT based production system compiler. *IEEE Trans. on Knowledge and Data Engineering*, 3(1):3–10, March 1991.

[25] Daniel E. Neiman. *Design and Control of Parallel Rule-Firing Production Systems*. PhD thesis, University of Massachusetts at Amherst, September 1992.

[26] Daniel E. Neiman. Issues in the design and control of parallel rule-firing production systems. *Journal of Parallel and Distributed Computing*, 1994. To appear.

[27] Alexander J. Pasik. *A Methodology for Programming Production Systems and Its Implications on Parallelism*. PhD thesis, Columbia University, 1989.

[28] James G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13(4):348–365, December 1991.

[29] J.G. Schmolze and S. Goel. A parallel asynchronous distributed production system. In *Proc. AAAI-90*, pages 65–71, 1990.

[30] A. Sohn and J-L Gaudiot. A survey on the parallel distributed processing of production systems. *International Journal on Artificial Intelligence Tools*, 1(2):279–331, June 1992.

[31] Salvatore J. Stolfo, Ouri Wolfson, Philip K. Chan, Hasanat M. Dewan, Leland Woodbury, Jason S. Glazier, and David A. Ohsie. PARULEL: Parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing*, 13(4):366–382, December 1991.

[32] S.J. Stolfo. Five parallel algorithms for production system execution on the DADO machine. In *National Conference on Artificial Intelligence*, pages 300–307, 1984.

[33] Shiow-yang Wu and James C. Browne. Explicit parallel structuring for rule based programming. In *Proc. 7th International Parallel Processing Symposium*, pages 479–488, 1993.

[34] Shiow-yang Wu, Daniel P. Miranker, and James C. Browne. Decomposition abstraction in parallel rule languages. Submitted for publication, 1994.