Parallel Execution of Logic Programs

May, 1988.

Ho-Fung Leung

A Thesis submitted to
the Division of Computer Science, Graduate School,
The Chinese University of Hong Kong,
in partial fulfilment of the requirement of
the Degree of Master of Philosophy.

thesis
QA
76.6
L46

488226

Parallel Execution of Logic Programs

1988

Ho-Fung Leung

Preface

It is definitely a great pleasure to be able to do research in one's own style after a four-year undergraduate course work study. For me, perhaps there is hardly anything that is more interesting than to concentrate on the investigation and understanding of human knowledge and delve into the origin of such knowledge. Starting from Clocksin and Mellish, whom I knew as the authors of the classical Prolog text book, I found Colmerauer, Kowalski, van Emden, Loveland, Robinson, Wang, Davis, Putnam, Church, Herbrand, Godel, until Frege. It is very exciting to trace the development of logic, and the way how it becomes "logic programming," which suggests clues to the understanding of the power and limitations of human intelligence.

Logic Programming is a new, developing, robust, and challenging field of Computer Sciences. Although fruitful results have been found in both of the theoretical and practical aspects, no one shall be satisfied with the current achievements. There is still plenty of room to be explored. We are looking forward to a new era of computer science and its application, when, as all of us are expecting, logic programming will mature and play an important role.

I would like to thank my supervisor, Dr. Kam-Wing Ng, who has brought me into this interesting field. He has given me strong supports and very useful advice during this two year M. Phil. programme, in academic aspect, as well as in daily life. I would also like to thank my friends, my fellow students, and the teachers in the Computer Science Department who had discussed various problems with me. I would also like to thank the staffs of the Microprocessor & Microcomputer Laboratory and the Computing Laboratory, to whom I have given so much trouble.

"The knowledge of human being is just like an immense sea. A person can never explore the whole sea in his life. As a student, one reaches the shore and learns. But the more important thing is that, one can later pour fresh water into the sea." These are the words of a friend of mine. May I quote them here to share with the readers of this thesis.

Introduction

The chief objective of this research project is to develop a better model for the parallel execution of logic programs. The main result of the research is that a new model for the parallel execution of logic program has been designed, which is superior to the other proposed models in the same framework.

The new model presented in this thesis is called the Competition Model. It is developed in the framework of the AND/OR Process Model [Conery, 1983]. The Competition Model supports both AND- and OR-parallelism. Compared with other models in the same framework, the Competition Model allows a higher degree of parallelism, avoids the reconstruction of the data dependency graph, and causes less OR-processes to be reset.

This thesis consists of four chapters. In Chapter I the basic nomenclatures of logic programming are introduced so that this thesis can be self-contained. However, this chapter is not intended to include all definitions in logic programming (see, *e.g.*, [Lloyd, 1984] for a more complete set of definitions). Chapter II presents a brief survey on proposed parallel execution models of logic programming. Emphasis is put on the description of the other models in the same framework. The main result is presented in Chapter III. We present an informal description before the presentation of formal algorithms. The correctness of the model is proved in this Chapter. Some variations of the model are also discussed. Chapter IV summaries the contributions and possible future researches.

There are three Appendixes in this thesis. Appendix A gives the proofs of the lemmas in Chapter III. Appendix B shows some examples of the parallel execution of logic programs produced by a simulation implementation of the Competition Model. Appendix C is a brief description of the simulation implementation.

Table of Contents

CHAPTER I THEORETICAL BASIS

Logic Programming is a modern paradigm of programming. Unlike conventional imperative languages, a logic program describes the logical relationship among the objects. The user of the program rises a query concerning a possibly derived relationship among some of these objects. A logic program interpreter shall try to find out which objects, if there are, satisfy such a relationship. Consequently, apart from the interesting topic of how to write good logic programs, there are plenty of room for research on efficient query-answering algorithms.

Nowadays, Logic Programming conventionally refers to programming in the Horn Clause subset of first-order logic. Therefore, in this thesis the term *logic program* will always refer to a Horn Clause Logic Program. The execution of Logic programs is based on the *SLD-Resolution Principle* [Kowalski and Kuehner, 1971] [Hill, 1974] [Apt and van Emden, 1982]. It was developed from the *Resolution Principle* [Robinson, 1965] which is applicable to general first-order clauses.

1.1 Horn Clause Logic

1.1.1 Introduction

Logic is an important characteristic of human thoughts. However, there had been no way to write down formally the logic one uses until the last century. In 1879, Frege devised the so-called Begriffsschrift [Frege, 1879], which is now known as the predicate logic.

1.1.2 First-Order Logic

Predicate logic describes the attribute, characteristics, or nature of the objects in the universe, or the relationships among them. A *predicate* is the name of such an attribute, characteristics, nature or relationship. The notation *F(a,b)* is intended to mean that the objects *a* and *b* bear the characteristics or relationship *F*. It should be noted that although such a notation may have the intended semantics, most logicians may like to treat them simply as syntactic constructions. In traditional logic books the authors may use *Fab* to mean the same thing.

1.1.2.1 Basic Nomenclatures

The basic nomenclatures of first-order predicate logic in clausal form are defined as follows (we shall follow partly [Robinson, 1965]):

(1) <u>Variable</u>. A variable is a symbol which represents an object in the universe.

(2) <u>Functor</u>. A functor is the name of an *n*-ary function which maps *n* objects to an object. A 0-ary function hence represents a *constant*.

(3) <u>Predicate</u>. A predicate is the name of an attribute, a characteristics, a nature of an object, or the relationship among *n* objects.

(4) <u>Term</u>. A variable is a term. A constant is a term. A string consisting of, in sequence, a functor of degree $n > 0$, a left parenthesis, *n* terms separated by commas, and a right parenthesis, is a term.

(5) *Atomic formula*. A 0-ary predicate is an atomic formula. A string consisting of, in sequence, a predicate symbol of degree $n > 0$, a left parenthesis, n terms separated by commas, and a right parenthesis, is an atomic formula.

(6) *Literal*. An atomic formula optionally preceded by a "~" symbol is a literal. If A is an atomic formula, "~A" and "A" are said to be the *complement* of each other.

(7) *Well formed formula*. Terms and literals are the only well formed formulae (wff's).

(8) *Clause*. A clause is a set of literals. The empty clause is denoted $\square$.

1.1.2.2 Interpretation

Given a set of clauses, it is possible to assign different objects to the variables in this set. Consequently, according to whether the predicates correctly describe the attribute, characteristics, nature or relationships after such an assignment, the set of clauses turns out to be either true or false. Such a process is called an *interpretation*.

An interpretation I consists of the following:

(1) *Universe of Discourse*. The universe of discourse (UD) is the domain of variables.

(2) *Extent of Predicate*. The extent of each n-ary predicate is a set of n-tuples of objects in UD.

(3) *Function*. Each n-ary function maps from UD^n to UD.

The truth value (truth or falsehood) of a set of clauses can be calculated following the procedure stated below:

(1) Each variable is assigned an object in UD.

(2) The value of a functor is the image of its mapping.

(3) If a term is a variable, then its value is the assigned object of the variable. Otherwise, the value of the term is that of its outermost functor.

(4) A 0-ary predicate can be assigned TRUE or FALSE.

(5) The truth value of an atomic formula containing a 0-ary predicate is the same as that of the predicate. An atomic formula $p(t_1,...,t_n)$ is TRUE if and only if the tuple $<t_1,...,t_n>$ is a member of the extent of p of degree $n > 0$.

(6) If a literal contains an atomic formula, then its truth value is the same as the atomic formula. If a literal contains a "~" symbol and an atomic formula, then its truth value is different from that of the atomic formula.

(7) A clause is TRUE if and only if at least one of the literals it contains is TRUE.

(8) A set of clauses is true if and only if all the clauses in the set are TRUE.

It is worth noticing that an empty clause □ is always false, hence a set of clauses containing one or more empty clauses is also false.

A *model* of a set S of clauses is an interpretation under which S is TRUE.

A set of clauses is *valid* if and only if every interpretation is a model of it.

A set of clauses is *satisfiable* if and only if it has at least a model.

A set of clauses is *unsatisfiable* if and only if it has no model.

1.1.3 Horn Clause Logic

Conventionally a literal is called a *negative* literal if it contains a "~" symbol, or a *positive* literal otherwise. *Horn Clauses* are clauses that contain at most one positive literal. The Horn Clauses are often written in the following format: the positive literal, if there is any, is written on the left side, and the other (negative) literals, if there is any, are written linearly on the right side after the ":-" or "<-" symbol, dropping the "~" symbol and separated by commas. A complete Horn Clause is terminated with a period. The following clauses are examples of Horn Clauses:

```
good_book(X) :- cheap(X), easy_to_buy(X), likes(Y,X), reader(Y).
reader(john).
reader(mary).
```

1.2 SLD-Resolution

The SLD-Resolution (*L*inear Resolution with *S*election Function for *D*efinite Clauses) is a mechanical theorem proving algorithm for Horn Clauses. It is a refinement of the SL-Resolution [Kowalski and Kuehner, 1971] which is derived from the full Resolution Principle [Robinson, 1965].

1.2.1 Herbrand Universe and Herbrand Interpretation

The full Resolution Principle, and all of its derivatives, take the *Herbrand Universe* as UD. Let F be the set of functors in a set of clauses. The Herbrand Universe contains all valid terms that can be constructed from the elements in F. Since UD cannot be empty, therefore if F is an empty set, then the Herbrand Universe contains a constant a.

In a *Herbrand Interpretation*, the legal value of a variable is a term. Functors map the terms to *themselves*.

A *Herbrand Model* of a set S of clauses is a Herbrand Interpretation under which S is true.

1.2.2 Proof by Refutation

The SLD-Resolution Principle is a typical example of the *proof by refutation* scheme. Given a set of axioms, there are two ways to prove a theorem. One is to directly derive the theorem, while the other is to show, by counter examples, that the negation of the theorem is inconsistent with the given axioms. The SLD-Resolution Principle takes the latter approach.

Before describing the SLD-Resolution Principle, some terms need to be defined.

(1) *Substitution Component.* A substitution component is an expression of the form T/V, where V is a variable and T is a term.

(2) *Substitution.* A substitution S is a set of substitution components such that if T_1/V_1 and T_2/V_2 are distinct substitution components in S then $V_1 \neq V_2$.

(3) *Instantiation.* If E is a set of clauses, a clause, a literal, a term, or a variable, and $\theta = \{T_1/V_1, ..., T_n/V_n\}$ is a substitution, then $E\theta$ is an expression obtained by replacing each occurrence of V_i in E by T_i. $E\theta$ is called an instance of E by θ.

(4) *Composition of Substitution.* If $\theta = \{T_1/V_1, ..., T_n/V_n\}$ is a substitution and λ is a substitution, then their composition is $\theta\lambda = \{T_1\lambda/V_1, ..., T_n\lambda/V_n\}$.

(5) *Variant.* Literals E and F are said to be a variant of each other if and only if there exist substitutions θ and ϕ such that $E\theta = F\phi$.

(6) *Unifier.* For any non-empty set S of wff's, if there exists a substitution θ such that $S\theta$ is a singleton, then the elements in S are said to be *unifiable*, and θ is a unifier. Moreover, if θ is such a unifier that for all other unifiers ϕ_i there exist a substitution λ_i such that $\phi_i = \theta\lambda_i$, then θ is called the *most general unifier* (mgu).

(7) *Standardization.* Given a set C of literals, if the variables in C are $v_1, v_2, ..., v_n$, then ξ_x is said to be the x-standardization of C where $\theta = \{x_1/v_1, x_2/v_2, ..., x_n/v_n\}$.

(8) *Resolvent.* Given two clauses A and B. Let S_A be a non-empty subset of A and S_B a non-empty subset of B. Let N be a set of atomic formulae which are members, or complement of members, of the union of $S_A\xi_x$ and $S_B\xi_y$. If N is unifiable with an mgu θ, then

$$C = (A - S_A)\xi_x\theta \cup (B - S_B)\xi_y\theta$$

as well as its variants are called the resolvent of A and B.

Given a set of Horn Clauses, the SLD-Resolution Principle can be briefly defined as follows.

(1) *Linear-input Derivation.* Given a set S of clauses, a linear-input derivation **D** is a sequence of clauses (C_1, ..., C_n) such that C_1 is a clause in S and C_{i+1} is a resolvent of C_i and a clause C in S. C_1 is called the *top clause* and C_n is the clause *derived* by **D**.

(2) *Proof by Refutation.* Given a set S of Horn clauses as axioms, a Horn clause C is a theorem of S, if a clause C' contains all and only the complements of the literals in C, and there is a derivation **D** with C' as the top clause and the empty clause □ as the derived clause. Here C' is called the *complement* of C

1.2.3 Completeness and Soundness of the SLD-Resolution Principle

A theorem proving algorithm is *sound* if and only if all theorems it proves to be correct is really a theorem. A theorem proving algorithm is *complete* if and only if there is no correct theorem that it cannot prove to be correct.

The SLD-Resolution is sound, i.e., for all derivations deriving an empty clause, the complements of their top clauses are theorems. The SLD-Resolution is complete, i.e., for all theorems there exist derivations with the complements of the theorems as the top clauses that derive an empty clause. Further details can be found in [Kowalski and Kuehner, 1971] [Hill, 1974] [Apt and van Emden, 1982].

1.3 Logic Programming

Horn Clause Logic was proposed as a Programming Language in the early 70's [Kowalski, 1974] with the adoption of SLD-Resolution. A logic program specifies the known facts and rules as a given set of axiomatic Horn clauses. The user of a logic program provides a query, which is regarded as a theorem to be proved, to the program. An interpreter then tries to prove or disprove the theorem by SLD-Resolution. Using the negation of the query as the top clause, if there exist derivations deriving empty clauses, then the query is true, and the compositions of the substitutions used in the derivations are called the *answer substitutions*. If there is no such derivation, then the query is false.

1.3.1 Procedural Interpretation

In an SLD-Resolution procedure, an *SLD-search tree* is formed. The root of the tree is the top clause. A clause D is a child node of a clause C if and only if the clauses C, D are consecutive clauses in a derivation. The leaves of the tree are the empty clauses or a clause that is not unifiable with any input clause. A derivation corresponds to a path from the root.

The *procedural interpretation* of logic programs corresponds to an in-order traversal of the SLD-search tree. When an empty clause leaf node is encountered, an answer substitution is found. If a non-empty clause leaf node is encountered, the interpreter *backtracks* and begins to search in another subtree.

It is notable that a procedural interpretation scheme is generally incomplete. If an infinite subtree exists to the left of an empty clause leaf node, then the interpreter will be *trapped* in the infinite subtree without finding an answer substitution.

1.3.2 Prolog

Prolog (*Pro*gramming in *Log*ic) [Clocksin and Mellish, 1981] [Shapiro and Stirley, 1986] is the first logic programming language that adopts the procedural interpretation scheme of SLD-Resolution. As a programming language, it is enriched with many extra-logical features that help performing I/O, increasing execution efficiency, evaluating expressions, and manipulating terms as data structures. A typical example of Prolog is shown below. It calculates the value of *n*!.

```
factorial(0,0).
factorial(N,X) :- M is N - 1, factorial(M,X1), X is N * X1.
```

The query can be written as

```
?- read(X), factorial(X,Y), write("factorial of",X,"is",Y), nl.
```

Prolog is considered to be not a good logic programming language. Apart from its peculiar extra-logical features, Prolog loses its completeness by adopting procedural interpretation and loses its soundness due to the lack of occur check in unification (Prolog can "unify" a variable x and the term $f(x)$ with the "mgu" $\{f(f(f(f(f(...)))))/x\}$). Furthermore, there is no parallelism in Prolog. Consequently, the efficiency of logic program execution is restricted.

CHAPTER II PARALLEL EXECUTION MODELS OF LOGIC PROGRAMS

2.1 Possible Parallelism in Logic Program Execution

2.1.1 Sequential Execution of Logic Programs

Consider the following Logic Program and the query in Prolog:

```
p(a,X)  :- q(X,a),r(X,c).
p(a,c).
p(b,c).
q(b,a).
q(c,b).
r(a,c).
r(b,c).

?- p(a,X),q(Y,Z),r(X,Y).
```

In Prolog, the query will be executed sequentially from left to right. That is, the literal "*p(a,X)*" will be executed *before* the other two can start execution. However, this is not necessary. An obvious way to speed up the execution is to start the first and second literals simultaneously. The third literal starts after the first and second ones finish and instantiate the variables *X* and *Y*.

On the other hand, in executing a literal, several clauses in the program can be tried simultaneously. For example, when executing literal "*p(a,X)*", the first and the second clauses in the program are resolvable with the query. In Prolog they are tried one after the other, but there is no reason why they cannot be tried at the same time.

2.1.2 The Selection Rule and Computation Rule in SLD-Resolution

In SLD-Resolution two rules govern the execution procedure. When the last clause in a derivation D is resolvable with an input clause, a resolvent is formed. In this event, one of the negative literals in the former clause is selected according to the *selection rule*. This selected literal unifies the only positive literal in the input clause. The selection rule in Prolog is to always select the left most one of such literals.

When the last clause in a derivation is resolvable with several input clauses under the constraints imposed by the selection rule, one of these input clause is selected. The selection criterion is called the *computation rule*. In Prolog the computation rule is to always choose the first of such input clauses.

2.1.3 AND-Parallel Execution and OR-Parallel Execution

Corresponding to the selection rule and computation rule, two kinds of parallelism are possible. *AND-Parallelism* refers to exploring the possibility of selecting and executing several literals in the goal clause in parallel. *OR-Parallelism* refers to exploring the possibility of selecting several input clauses and form different resolvents in parallel.

In designing AND-Parallelism, the emphasis is the solution to the so-called *binding conflicts*. Consider the program in section 2.1.1, if the three literals are allowed to execute at the same time, they may bind different values to the same variable (for example, the literal "*q(Y,Z)*" may bind *Y* to *b*, while "*r(X,Y)*" binds *Y* to *c*.)

The emphasis in the design of an OR-Parallel Execution Model is the handling of multiple binding values. For example, the literal "*q(X,Z)*" produces the substitutions {*b/X,a/Z*} and {*c/X,b/Z*}. How these different values for the same variable are stored and usable in subsequent execution is the chief consideration of all OR-Parallel Execution Models.

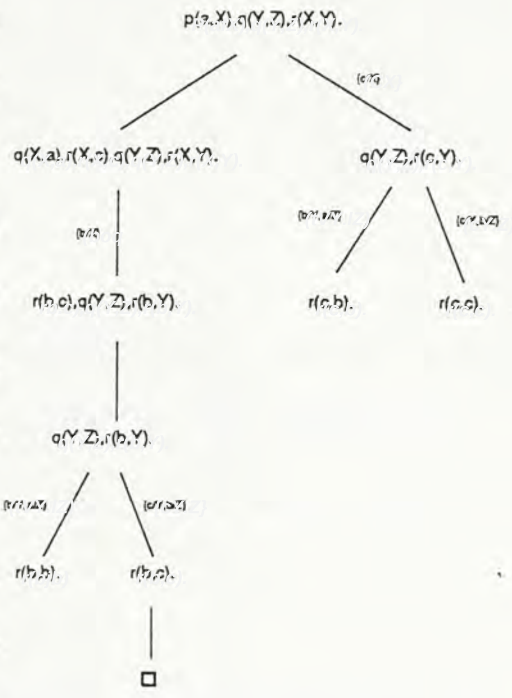
2.2 A Brief Survey

Although the parallel execution of logic programs has been an interesting topic as early as the birth of Prolog, systematic research on this topic seems to begin at the beginning of this decade. It is observed that the model proposed so far fall into several categories.

2.2.1 OR-Parallel Execution of Logic Programs

The first category is formed by the models dealing with purely OR-Parallelism. This corresponds to exploring the subtrees in the SLD-search tree in parallel. Usually these models use the same selection rule as Prolog's. Some models chiefly consider the binding environment structures. They use structures such as hash tables, directories, linked lists etc. to store the bindings. Other OR-Parallel Execution models aim at efficient processor scheduling. Since the physical number of processors is limited, an algorithm must be applied to dispatch the tasks of exploring subtrees to the processors.

In OR-Parallel Execution Models, the binding of a variable is regarded as associated with an edge in the SLD-search tree (see the diagram on the right). Every processor exploring the subtrees below this edge should be able to have access to this binding, which means that it can read or modify the binding. In order to maintain consistency, most models consider the binding as possessed by a processor below this edge. When the other processors modify the binding, new copies are made. The paper by Warren [Warren, 1987] describes the process of evolution of OR-Parallel Execution Models.



The SLD-search tree showing the bindings for the example in section 2.1.1

The other stream of research investigates the efficient allocation of processors. Roughly speaking, there are three classes of models: the "dance pool" models, the "orthodox" models and the "oracle" models. For the first class, the nodes to be explored are placed in a pool. A processor picks up a node in the pool, performs the Resolution one more step, and places the resolvent nodes back to the pool. A typical example is the model of Ciepielewski and Haridi [Ciepielewski and Haridi, 1983]. For the "orthodox" models, a processor picks up a node and generates the resolvent nodes. If there are more than one resolvents, the processor chooses one of them to explore, and calls for other processors to explore the rest. If a processor finishes, it can "steal" an unexplored subtree [Ali, 1987]. For the final class, a processor is given an "oracle" by a control processor. It follows the oracle which directs it to select which subtree to explore at a node from which several branches emerge. It is claimed that the amount of communication among the processors is reduced [Clocksin, 1987].

2.2.2 AND-Parallel Execution of Logic Programs

The other category is those which deal with AND-Parallelism. This corresponds to a simultaneous selection of several literals in a goal clause.

The framework for AND-Parallel Execution models is the AND/OR-Process Model [Conery, 1983]. In fact, the AND/OR-Process Model includes both AND- and OR-Parallelisms. However, since its OR-Parallelism is simple, we consider it as an AND-Parallel Execution model.

The AND/OR Process Model tries to define a dynamic partial order among the literals in a query so as to find out possible parallelism free from binding conflict. Given a goal clause, it is in general impossible to execute all literals in parallel, because this will result in binding conflicts. In the AND/OR-Parallel Model, if a variable is contained by several literals, then one of these literals will be responsible for finding a binding for the variable and the others check whether this binding is acceptable. The former literal is called the *generator*, and the latter literals are the *consumers*. All generators execute and finish before the consumers start. This defines a partial order in the execution sequence. The generator-consumer relation forms a *data dependency graph* among the literals. If a consumer fails, it *backtracks* to one of the generators which will then produce another binding.

The OR-Parallelism in this model occurs in the execution of a literal. Although the valid binding of a variable is required one at a time, different clauses can be tried in parallel to find the binding.

The major drawback of the AND/OR Process Model is that the data-dependency graph needs to be reconstructed whenever a generator produces a non-ground binding, because new variables are created and shared among the consumers. Subsequent researches aimed at reducing this overhead. Some of these models [Chang, Despain and DeGroot, 1985] limit the AND-Parallelism by constructing a static worst case data-dependency graph, while the others [DeGroot, 1984] [Hermenegildo and Nasr, 1986] allow a restricted AND-Parallelism in a semi-static data-dependency graph.

Most recently a model is proposed which uses a token-passing scheme to solve this problem [Lin and Kumar, 1986]. In this model, a token is created for a variable, and new tokens are created for new variables. The literals queue up, and the tokens are passed from the front towards the rear of the queue. If a literal collects all the tokens for the unbound variables it contains, the literal becomes a generator.

On the other hand, the *backward execution* algorithm in the original AND/OR Process model, which selects the backtrack literal, is found to be incorrect in certain circumstances. New algorithms have been proposed [Lin, Kumar and Leung, 1986] [Woo and Choe, 1986] to correct this mistake.

2.2.3 Parallel Logic Programming Languages

Some researchers have designed new parallel logic programming languages. Parallelism in these language needs to be specified by the programmers. Like Prolog, these languages (especially the earlier ones) emphasize efficiency, and place little concern on completeness or soundness.

Two well known examples of parallel logic programming languages are Parlog [Clark and Gregory, 1986] and Concurrent Prolog [Shapiro, 1983]. More recently there are Guarded Horn Clause (GHC) [Ueda, 1985] and Classified Horn Clause (CHC) [Yang, 1987].

These languages have some common features. To deal with AND-Parallelism, most of them use variable annotations to help determine which is the generator. For OR-Parallelism, they all adopt the GHC approach, which *commits* the execution to one of the clauses when several clauses are resolvable with the selected literal in the goal.

2.2.4 Other Approaches

The approaches of several Parallel Execution Models are quite different from the exploration of the SLD-search tree. Some embed AND-Parallelism in OR-Parallel Execution, or vice versa in some others. Recent examples include the Intelligent Channel [Kasif and Minker, 1985], the OR-forest Model [Sun and Tzu, 1986], the REDUCE-OR Model [Kale, 1987], the Sync Model [Li and Martin, 1986], *etc.*.

2.3 AND-Parallel Execution Models

The model to be presented in this thesis is an AND-Parallel Execution Model. It evolves from the AND/OR Process model, and subsumes the Lin-Kumar model. We shall therefore present a brief description of these two models. For further details, please refer to [Conery and Kibler, 1985] and [Lin and Kumar, 1986].

2.3.1 The AND/OR Process Model

The AND/OR Process Model makes use of two kinds of processes: the AND-processes and the OR-processes. An AND-process solves a clause. It finds a consistent substitution for the variables in the literals. A literal is solved by an OR-process (therefore the terms *process* and *literal* are always used interchangeably). If there are several clauses resolvable with the goal literal, the OR-process creates one AND-process for each one of the resolvents. In this way an AND-OR process tree is formed.

The AND-Parallel Execution scheme in the AND/OR Process Model consists of three algorithms: the ordering algorithm, the forward execution algorithm, and the backward execution algorithm.

The ordering algorithm constructs the data dependency graph. The literals are assumed to be in a linear list. The ordering algorithm scans from the front to the rear. When a literal is met, if all the unbound variables it contains still do not have generators, then the literal becomes the generator of all these variables. The ordering algorithm has to be run again when a generator generates a non-ground binding. This phenomenon is known as the *reconstruction of data dependency graph* Its reduction has become the goal of many subsequent researches.

After a generator has found a substitution, the consumers become active. This is the *forward execution*, which terminates only after all literals have successfully finished. On the other hand, if a literal fails, then backtracking is initiated.

The backtracking in the *backward execution* is organized according to the *nested-loop principle*. If a consumer is the child, in the data dependency graph, of several generators, these generators are regarded as the *control variables* in a nested **for**-loop with the consumer as the loop body. When the consumer fails, the last of its parent generators in the linear list is backtracked. This resembles an "increment" of the inner most control variable.

To achieve such an effect, Conery in his original model [Conery, 1983] makes use of two kinds of lists. A *redo list* is created for each literal when the ordering algorithm is applied. It contains the literal itself and all its ancestors in the data dependency graph. The *failure context* records the failures. It is initially empty.

After a literal fails, it is appended to the failure context. Then the algorithm searches to see whether the failure context is the prefix of any redo list. If there is no such redo list, the AND-process fails. Otherwise, the backtrack literal is the literal in that redo list immediately after its failure context prefix[1].

After the backtrack literal has been identified, it is asked to produce the next set of bindings. Meanwhile, all literals after it in the linear list are reset. This resembles the resetting of inner control variables when an outer control variable is increased in a nested **for**-loop. In addition, if any parent of a consumer is backtracked to or reset, then the consumer is cancelled.

Finally, when a new process starts, and it is in the failure context, it is removed from the failure context, as well as all literals to its right.

The backward execution algorithm in the AND/OR Process Model was later found to be incorrect. This was discovered by Lin, Kumar and Leung [Lin, Kumar and Leung, 1986] and Woo and Choe [Woo and Choe, 1986].

---

1 To be precise, the backtrack literal *B* is defined as

```
    backtrack_literal(B)  :- redo_list(R),
                             failure_context(F),
                             concat(F,[B|_],R).

    concat([H|T],X,[H|T1])  :- concat(T,X,T1).
    concat([],X,X).
```

2.3.2 The Lin-Kumar Model

The Lin-Kumar Model is designed in the framework of the AND/OR Process Model and is also based on the data dependency graph. It assumes that the literals are forming a linear list, and there is no need to have an ordering algorithm.

In the Lin-Kumar Model, a token is created for each variable. In the *forward execution*, the tokens are passed along the list of literals, from the front to the rear. A literal keeps a token if it contains the corresponding variable. When a literal successfully collects all the tokens of the unbound variables it contains, it becomes a generator. If a generator binds a non-ground binding, new tokens are created for the new variables. The new tokens are then passed from the generator along the list of literal towards the rear. In this way, the data dependency graph is constructed implicitly.

Corresponding to the redo lists in the AND/OR Process Model, each literal keeps a *B-list*. The B-lists are initially empty. When a consumer receives a binding from a parent literal, the parent literal is inserted to the B-list of the consumer such that the literals in the B-list are always sorted according to their order in the linear list.

When a literal fails, the head of its B-list is selected as the backtrack literal. In addition, the tail of its B-list is passed to the backtrack literal and merged into the B-list of the latter's. This operation helps memorizing the failure history and cure the flaw in the AND/OR Process Model.

After the backtrack literal has been specified, some literals are reset and some others are cancelled. The selection criteria are the same as that of the AND/OR Process Model. When a literal is reset, its B-list is re-initialized with its parents.

2.3.3 The Woo-Choe algorithm

The correction to the backward algorithm by Woo and Choe [Woo and Choe, 1986] is similar to that in an earlier version of the Competition Model [Ng, Leung and Yu, 1987].

The algorithm by Woo and Choe is a very careful and conscientious design. In the algorithm, a *parent set* and a *redo cause set* (RCS) are associated with each literal. The function of the RCS is similar to the redo list or B-list. Also there is a *failed literal set* whose function is similar to the failure context.

The parent set of a literal contains its parents. The RCS is initially empty. When a literal fails, it is added to the failed literal set. When backward execution applies, one backtrack literal is chosen for each member in the failed literal set using the follow procedure. Let S1 be the RCS of the failed literal plus the failed literal. Let S2 be the union of the parent sets of the members in S1. Let S3 be S2\S1. The backtrack literal for the failed literal is the member in S3 that is the latest in the linear order of the literals. Similar to the merging of B-lists in the Lin-Kumar model, the RCS of the backtrack literal is now replaced by the union of it and S1. If there are several backtrack literals chosen, the earliest one in the linear order is selected as the backtrack literal.

This algorithm is correct. However, when several literals failed (so-called *multiple failure*), only one literal is backtracked to. This makes simultaneous backtracking in independent subtrees impossible.

2.3.4 A Backward Execution Algorithm for Static Data Dependency Graph

One may have noticed that in the above models, some literals that are reset are innocent. Resetting these literal does not help recovering the failure. In 1987, Conery proposed another backward execution algorithm that resets less number of innocent literals [Conery, 1987]. However, this algorithm applies only to a static data dependency graph. We shall present a brief description of this algorithm because it is the basis of the backward execution algorithm of the Competition Model.

Assume that the data dependency graph is static. A linear order of the literals can be obtained by the same method as in the AND/OR Process Model. When a literal fails, it places a *mark* to all of its ancestors in the data dependency graph. The backtrack literal is then the last in the linear order of those marked by the failed literal or its successors.

In order to determine the literals to be reset, a *candidate set* is associated with each literal. The candidate set of a literal contains all its ancestors and all ancestors of its descendents, except the literal itself. After the backtrack literal has been identified, the literals to be reset are determined by the following procedure. Firstly, let *MV* contain the variables generated by the backtrack literal. All literals consuming any variables in *MV* need to be cancelled. Let *CS* be assigned the candidate set of the backtrack literal. Searching towards the rear of the linear list, if a literal is in *CS* and needs not be cancelled, the members of its candidate set is added to *CS*, and the literal is reset; if the reset literal generates new binding for its variables, these variables are added to *MV*.

The central idea of this algorithm is that a generator may belong to different nested **for**-loops. Backtracking to or resetting a generator should cause the resetting of its "inner control variables" in different **for**-loops.

CHAPTER III THE COMPETITION MODEL

The Competition Model [Ng and Leung, 1988] [Ng and Leung, 1989] is an AND-Parallel Execution Model in the framework of the AND/OR Process Model. Compared with other models in this framework, the Competition Model allows a higher degree of parallelism, avoids the reconstruction of the data dependency graph, and causes fewer literals to be reset.

3.1 The Framework

The Competition Model is designed in the framework of the AND/OR Process Model Similar to other models in this framework, the model makes use of AND-processes to solve goal clauses and OR-processes to solve single literals.

3.1.1 OR-Parallelism

The OR-Parallelism in the Competition Model is achieved by the use of OR-processes. An OR-process finds a substitution for the variables in a literal. If there are several program clauses which are resolvable with the literal, then one AND-process is created to solve each of these resolvents.

In the following description, an OR-process is assumed to return one answer substitution or report failure after a finite period of time. If an OR-process needs infinitely long time to find an answer substitution, then the Competition Model becomes incomplete. An OR-process may be *frozen*, which means that its action is suspended until the OR-process is later *melted*. An OR-process *succeeds* if any of its child AND-processes finds an answer substitution. An OR-process *fails* if all its child AND-processes fail. After an OR-process is created, it begins to find answer substitutions. An OR-process may be sent a request to produce another ("the *next*") answer substitution. An OR-process can be *cancelled.* An OR-process can be *reset,* after which the OR-process produces answer substitutions in the same sequence as if it had just been created.

Since each literal corresponds to an OR-process, we shall use the terms "a literal *P*" and "the OR-process of the literal *P*" interchangeably.

3.1.2 AND-Parallelism

The AND-Parallelism in the Competition Model is achieved by the AND-processes. An AND-process finds consistent answer substitutions for a goal clause.

In the following sections we shall present the algorithm for AND-Parallelism. Following the algorithm, one answer substitution can be found. In order to find more answer substitutions, a pseudo-literal can be added to the clause. The pseudo-literal contains all variables in the clause and fails for any bindings. After an answer substitution is found, the pseudo-literal becomes the last consumer of all bindings. Because the pseudo-literal always fails, the backtracking mechanism in the algorithm will automatically generate a new answer substitution. Such a mechanism is not included in the algorithm so that the presentation can be simpler.

The AND-Parallel Execution Algorithm consists of two parts: the *forward execution algorithm* and the *backward execution algorithm.* In the presentation of each algorithm, an informal description is given before the formal algorithm.

3.2 The Forward Execution Algorithm

3.2.1 Informal Description

Initially, a set of negative literals is given to an AND-process as the goal. All of these literals are placed in an *inactive set*. The literals are not structured. If a literal can get out of the inactive set, it starts generating the bindings for all of the unbound variables it contains. The first rule is that, every variable can have at most one generator. This is the basic idea of the algorithm.

All literals containing the same variable are potential generators of that variable. All of these literals compete for the right of generating the bindings for this variable (hence the name "Competition Model") because every variable can have at most ONE generator. Alternatively, it can be assumed that a token is associated with each variable. The literal which obtains the token then becomes the generator of that variable.

Because the literals are asynchronous processes, and it takes time to get a token, therefore deadlock may occur. For example, if there are two literals $p_1(A,B)$ and $p_2(A,B)$, then they compete to generate the bindings for the variables A and B. However, it is possible that when p_1 obtains the token of A, p_2 has obtained that of B. As a result, neither p_1 nor p_2 may start because each of them has to get both the tokens of A and B before being allowed to start. This is the second rule.

The simplest solution to this problem is to use a monitor [Hoare, 1974]. A monitor is an abstract data structure which contains both data and procedures. It is so designed that at any moment only one of the asynchronous processes can enter the monitor and use the procedures in the monitor to manipulate the data in the monitor. The tokens for each variable are placed in the same monitor so that the most "fortunate" literal can enter the monitor. Heuristics can be applied here so that it may be easier for some literals to enter the monitor than others (see section 3.4.4). If a literal *P* obtains the token of the variable *V*, all other literals containing *V* become consumers of *V*. They cannot come out of the inactive set until *P* has generated a binding for *V*.

It is worthwhile to point out that the number of variables varies during execution, because usually a non-ground binding introduces new variables. When a new variable is introduced, a token will be created for it. Of course, if the binding introducing the new variable is denied during backtracking after some time, then the token for that variable will be disposed of.

In fact, every literal generates a possibly non-ground substitution. This substitution applies only to the literals in the inactive set, but does not affect the literals outside the inactive set. This statement becomes clear if it is remembered that all the consumers are still inside the inactive set because they cannot get out, and that those literals outside the inactive set are no longer consumers of any variables with unknown binding.

Obviously, if it happens that the inactive set becomes an empty set, and at the same time all the literals (they are now all outside of the inactive set) have finished, then the AND-process succeeds, and the final substitution is the composition of the numerous substitutions produced by the generators. The data dependency graph is constructed implicitly during the time of execution.

It is more convenient to define *execution level* for a literal. A literal is said to be on level 1 if and only if it does not have any parent. Otherwise, a literal is on one level lower than the lowest level of its parents. We shall assign an *order-number* to each literal. The literal assigned an order-number n is the nth literal that becomes active. If a literal is cancelled, then later when it becomes active again, it gets a new order-number. We also define a relation '$<_l$' between two literals such that $P <_l Q$ if and only if P is on a level higher than Q; or on the same level as Q but its order-number is less than that of Q.

3.2.2 The Algorithm

<u>Forward Execution</u>

Main routine

```
1. Let the inactive set contain all the literals in the goal clause.

2. Repeat steps 3. - 5. until the inactive set becomes empty and all processes succeed.

3. Select a set G of literals from the inactive set.

4. For each literal L in G do the following:

    4.1 create an OR-process for L;

    4.2 if the OR-process succeeds in finding a substitution then

            apply the substitution

        else

            backtrack for L.

5. Continue.
```

6. The final answer substitution is the composition of the substitutions generated by the generators.

7. End.

Select operation

Select literals from the inactive set and add them to *G*. The condition is that no two literals in *G* should share common variables. These literals are said to be *active* and they no longer belong to the inactive set.

Apply operation

For each element *L* in the inactive set, if the substitution θ is applicable to *L*, then apply θ to *L*. Record that the parents of the new *L* are those of the old *L* plus the generator of θ. Note that the old *L* should be stored somewhere because θ may become obsolete some time later, and at that time the old *L* will be restored. See the **cancel** operation below.

3.3 The Backward Execution Algorithm

3.3.1 Informal Description

There may be times when literals fail. At that moment backtracking is initiated. If a literal fails, there must be one or more of its ancestors in the constructed data dependency graph which have produced a "bad" substitution. Since it is inefficient to find out the actual place where the unification conflict happens [Codognet, Codognet and File, 1986], the failed literal may mark all of its ancestors [Conery, 1987]. If a literal *P* is marked by another literal *Q*, then it is implied that *P* may be responsible for the failure of *Q* because *P* has given *Q* a "bad" substitution directly or indirectly.

Now it becomes clear that the backtrack literal must be one of the marked literals. However, an ancestor may be marked by more than one of its descendents in the data dependency graph. In addition, a literal may fail when it is backtracked by its descendents. Therefore, the criterion discussed in the previous paragraph is incomplete. The backtrack literal should be chosen from among those literals marked by the failed literal and the descendents of the failed literal.

Among the candidates of the backtrack literals, the one which is on the lowest level with the greatest order-number is chosen. That such a literal is chosen means that the most recent consistent data dependency (sub-)graph appears again. If the ordered list concept of the AND/OR Process Model [Conery, 1983] is applied here, then the literals are total ordered by the relation '$<_l$'.

After identified, a backtrack literal starts finding a new answer substitution. Although the descendents of the backtrack literal will, in general, receive new bindings, a more efficient method is not to cancel them until the backtrack literal has finished. The reason is that although the newly generated substitution is different from the previously generated one, sometimes the binding for certain variables in that substitution may remain unchanged. It is a waste of resource to cancel all the descendent literals before it is actually necessary and to reconstruct the identical sub-graph afterwards. On the other hand, however, allowing these may-be-cancelled literal to continue execution is a potential waste of resources. Under these considerations, the solution is to "freeze" the may-be-cancelled sub-graph during the execution of the backtrack literal. They may be melted, or cancelled, after the backtrack literal finished, depending on whether cancellation is necessary. If the backtrack literal fails, further backtracking will be called for, and its descendents can still be frozen. Therefore, it can be thought that the failed literals and their descendents form frozen sub-graph(s).

To assure that no answer substitution will be missed, some generators need to be reset when a backtrack literal finds a new answer [Conery, 1983]. To reset a literal means to make the literal generate answer substitutions from the beginning. We use the following method to determine the literal to be reset. Let the *mark set* of *P*, mark(*P*), be $\{\mathbf{m}_Q \mid Q$ has marked $P\}$ where *P* and *Q* are literals. Obviously, backtracking a literal *P* is to generate a new binding for the failure of one or more of the literals in mark(*P*). Therefore, those active literals (i.e. being outside the inactive set) whose mark set has a non-empty intersection with that of the backtrack literal, and which have greater order-numbers than the backtrack literal (i.e. behind the backtrack literal in the '$<_l$' order), are reset. The same is done to the members in the mark set of a reset literal.

Finally, consider the situation that a backtrack literal succeeds. There are two possible consequences. The first is that some of its frozen direct descendents melt and become its direct descendents again because the variable bindings they consume have not been changed. In this case the execution continues. The other possible consequence is that some of the original direct descendents of the backtrack literal do not become the direct descendents again for whatever reason. For examples, either a literal cannot get appropriate tokens, or the token it had been using is now thrown away as the variable disappears. In this case this literal melts all of its frozen descendents and disassembles the structure among them. All of these literals in the melted sub-graph now reside in the inactive set and become free competitors for tokens. This will be called the *cancellation* of the literals because they come back into the inactive set again.

3.3.2 The Algorithm

Backtrack for operation

1. Let the failed literal be *P*.

2. Place a mark $\mathbf{m}_P$ at the ancestors, i.e. parents, and ancestors of parents, of *P*.

3. Let *B* be a successfully finished literal such that mark(*B*) has a non-empty intersection with (mark(*P*) + $\{\mathbf{m}_P\}$). The backtrack literal is the last of such literals before *P* as determined by the '$<_l$' relation. If there is no such *B*, then the AND-process fails.

4. **Freeze** the children of *B*.

5. Reactivate the OR-process for *B* to generate the next answer substitution.

6. If the OR-process succeeds then

```
    for each child C of B do
      if the bindings C consumes are not changed then
        melt C
      else
        reset all literals R outside of the inactive set such that $B <_l R$ and the intersection of
    mark(R) and mark(B) is non-empty.
        cancel C
    apply the new bindings
else
```

```
    backtrack for $B$

7. End.
```

Freeze operation

```
1. Let the literal to be frozen be $L$.

2. If $L$ is inactive, or has been frozen, go to step 5.

3. If $L$ is active, suspend the operation of $L$.

4. Freeze all the children of $L$.

5. End.
```

Melt operation

```
1. Let the literal to be melted be $L$.

2. If $L$ is not frozen, go to step 5.

3. Resume the operation of $L$. Make the literal active.

4. If $L$ is a failed literal, backtrack for $L$; otherwise, melt all the children of $L$.

5. End.
```

Reset operation

```
1. Let the literal to be reset be $L$. If $L$ has been reset or is inactive, go to
    step 8.
```

2. Resume the operation of L if it is frozen.

3. **Reset** all literals R such that $L <_l R$ and the intersection of mark(R) and mark(L) is non-empty.

4. **Freeze** all the children of L.

5. Make L generate substitutions from the beginning. L generates the first substitution generated.

6. For each of the children C of L do

```
if the binding $C$ consumes is not changed then
  melt $C$
else
  cancel $C$.
```

7. **Apply** the substitution.

8. End.

Cancel operation

1. Let the literal to be cancelled be L. Let P be the parent of L causing this cancellation. Resume the operation of L if it is frozen.

2. Restore the old L which has not been applied with θ, and θ is the substitution generated by P. (see **apply** operation above)

3. Make L inactive. Let mark(L) be an empty set.

4. Deny the parent relationship that any other literal is holding with L. L becomes an orphan. L also denies the generatorship of any variables.

5. Cancel all children of *L*.

6. End.

3.4 Possible Variations

The proposed algorithm can be modified or improved in many ways. In this section we shall discuss some of them.

3.4.1 Generator Selection

An obvious variation is that, if several literals are the potential generators of a variable, then the one which can most quickly generate the substitution becomes the generator. That is, to make the algorithm more competitive -- all these potential generators are allowed to start, the one that finishes most quickly becomes the generator of all of the previously unbound variables it contains, and the other competitors, which lose, become consumers. This variation has an additional advantage that if there is any literal which should fail, maybe it will fail earlier. This may save a lot of resources. However, the disadvantage is that more resources are used, and wasted, because there is only one winner and many losers.

3.4.2 Token Distribution

Another variation is the algorithm by Lin and Kumar [Lin and Kumar, 1986]. In their algorithm, there is a scheme that distributes the tokens to appropriate generators. This can be regarded as a biased competition for the literals that obtain the tokens.

3.4.3 Resetting Literals

Another possible variation is that, every literal stores the substitutions it has found in a table. When the literal is reset, it can immediately get the previously found bindings from the table, instead of putting efforts to find it again. This implementation issue is similar to the idea of the *Result Cache* [Conery, 1987].

3.4.4 Use of Monitors

The fourth possible variation is that, instead of using a single monitor to store the tokens, use individual monitors (or semaphores) to control the access to the variable. The advantage is obvious: more than one literal can access different tokens at the same time. The disadvantage is also obvious: deadlock may result. Fortunately enough, there are various known algorithms (e.g. the Banker's algorithm) in the field of operating systems for solving the deadlock problem. However, this will cause much overhead.

3.4.5 Biased Competition

As another variation, it is worthwhile considering a heuristics for controlling the competition among the literals for the right(s) to generate a substitution for a variable. This can be a replacement for the first variation mentioned above. The heuristics is: the literal containing most (ground-)instantiated arguments should be selected. The main idea behind this heuristics is that the number of possible bindings for the unbound variables is significantly reduced when a ground term is involved.

3.5 The Soundness of the Competition Model

An algorithm for logic program execution is *sound* if and only if the answer it finds is always valid. It is obvious that the AND-parallel Execution algorithm in the Competition Model is sound.

If the algorithm successfully terminates, then the final answer substitution is accepted by all literals. For the generators, it is impossible for them to reject the bindings they generate. For the consumers, if they do no accept the binding, the algorithm will not terminate because backtracking will be initiated. Therefore, if the algorithm successfully terminates, all generators and consumers will accept the answer substitution.

3.6 The Completeness of the Competition Model

An algorithm for logic program execution is *complete* if and only if no answer substitution will be missed. The completeness of the backward execution algorithm in the Competition Model can be proved in much the same way as that of the Lin-Kumar-Leung algorithm [Lin, Kumar and Leung, 1986].

Let F be the failed literal. Let $X(F)$ be a set containing all the ancestors of F and the ancestors of all its ever failed descendents. It is clear that only backtracking to one of the members of $X(F)$ can cure the failure of F. To ensure that all answer substitutions are found, the active (and the finished) literals are totally ordered by the '$<_l$' relation, and backtracking is always done to the last one before F (determined by the '$<_l$' order) of literals in $X(F)$.

Lemma 1. If F fails, then B can possibly cure the failure of F if and only if $B \in X(F)$ and $B <_l F$.

Proof: See Appendix A.

Lemma 2. The backtrack literal is the last member in $X(F)$ before F as determined by the '$<_l$' relation.

Proof: Trivial by lemma 1 and preceding discussions.

Lemma 3. After F places marks on its ancestors, let

$$S(F) = \{P \mid P <_l F \text{ and } \mathrm{mark}(P) \cap (\mathrm{mark}(F) \cup \{\mathbf{m}_F\}) \neq \varnothing\}$$

then $S(F) = X(F)$

Proof: See Appendix A.

Lemma 4. If a literal B is reset or backtracked to, and it successfully finishes, it is sufficient to reset the literals R such that $B <_l R$ and $\mathrm{mark}(B) \cap \mathrm{mark}(P) \neq \varnothing$.

Proof: See Appendix A.

3.7 A Comparison with Other Models

As the final section in this chapter, we shall present a brief comparison of this model with other models in the same framework.

We shall only compare with models using dynamic data dependency graph only and we shall not compare with models using static or semi-static data dependency graph, such as [Chang, Despain and DeGroot, 1985], [DeGroot, 1984], or [Conery, 1987].

The models and algorithms we shall consider then include the AND/OR Process Model [Conery, 1983], the Lin-Kumar Model [Lin and Kumar, 1986] [Lin, Kumar and Leung, 1986], and the Woo-Choe algorithm [Woo and Choe, 1986].

3.7.1 Forward Execution Algorithms

3.7.1.1 Comparison with the AND/OR Process Model

The forward execution algorithm in the Competition Model corresponds to the ordering algorithm and forward execution algorithm in the AND/OR Process Model. Compared with the latter, the former allows a higher degree of parallelism by adopting the concept of competition. This can be illustrated by a simple example.

```
p0(A,B,C,D)  :- p1(A,B),p2(B,C),p3(C,D).

?- p0(_,_,_,_).
```

According to the data dependency graph constructed by the ordering algorithm of the AND/OR Process Model, p_1 is the generator of the variables A and B, p_2 is the consumer of B and the generator of C, and finally p_3 is the consumer of C and the generator of D. Consequently, no AND-parallelism is discovered.

On the other hand, there is no ordering algorithm in the Competition Model. The data dependency graph is constructed implicitly, and only the necessary portion of the data dependency graph is constructed. Consequently, the reconstruction of the data dependency graph is avoided.

Finally, it is obvious that the first and the third literals can start simultaneously, making the second their common child in the data dependency graph. This is possible in our model via the competition for tokens.

3.7.1.2 Comparison with the Lin-Kumar Model

The same situation appears in the Lin-Kumar Model. When the tokens of A, B, C and D are passed towards the end of the literal queue, p_1 keeps tokens of A and B, p_2 keeps that of C, and p_3 can only get the D token.

3.7.2 Backward Execution Algorithms

We shall compare our backward execution algorithm with the Lin-Kumar-Leung algorithm [Lin, Kumar and Leung, 1986] and the Woo-Choe algorithm [Woo and Choe, 1986]. They are the correction to the original backward execution algorithm in the AND/OR Process Model.

3.7.2.1 The Time Complexity of the backward execution algorithm in the Competition Model

In the Competition Model, backward execution involves three operations: to select the backtrack literal, to select the reset literals, and to freeze some literals.

To identify the backtrack literal, the first step is to place a mark of the failed literal in all its ancestors. Since this can be done by concurrent processes, the time complexity will be $O(n)$ where n is the number of literals in the clause. Afterwards, finding the last of the marked generators in the sequence determined by '<,' is also an $O(n)$ operation. (Note that set intersection and comparison are both $O(1)$ operations.) Similarly, freezing the descendent literals of the backtrack literal and resetting literals along the '$<_i$' chain are both $O(n)$ operations. Hence, it takes $O(n)$ time to perform the backward execution in the Competition Model.

3.7.2.2 Comparison with the Lin-Kumar-Leung algorithm

In the Lin-Kumar-Leung algorithm, the backtrack literal is always the head of the B-list of the failed literal. This is an $O(1)$ operation. Afterwards, the tail of the B-list of the failed literal is merged to that of the backtrack literal. Merging two lists is an $O(n)$ operation. Moreover, cancelling and resetting literals are also $O(n)$ operations. Therefore, the time complexity of this algorithm is $O(n)$.

It should be noted that more literals are reset in this algorithm.

3.7.2.3 Comparison with the Woo-Choe algorithm

The Woo-Choe algorithm is also an $O(n)$ algorithm. It is because in this algorithm the calculation of the S2 set is an $O(n)$ operation. Cancelling and resetting literals are both $O(n)$ operations.

This algorithm also reset more than enough literals. Moreover, it eliminates the possibility of performing independent backtracking in different portion of the data dependency graph due to its inappropriate handling of multiple failure.

CHAPTER IV CONCLUSION

4.1 Contributions

The Competition Model presented in this thesis is a superior one over the other proposed models in the same framework. It achieves the maximum parallelism within an AND-process, and is comparatively more efficient in backtracking.

4.2 Further Researches

Parallel execution of logic programs is an important research topic in the recent future. The reason is that this topic has a very close relation with the development of the next generation computers, both in the software and hardware aspects.

4.2.1 Parallel Execution of Horn Clauses

The framework provided by the AND/OR Process Model is a simple and efficient basis for developing efficient models. However, it is thought that the parallelism in such a model is somehow restricted. It is notable that the OR-parallelism in this framework is insufficient, and the backtracking scheme employed in AND-parallelism is definitely a very serious drawback. Some researchers think that there should not be backtracking of any form in a parallel execution model of logic programs [Li and Martin, 1986]. This may be true. We are looking forward to a formal comparison of the efficiency of different models in different frameworks.

As pointed out in Chapter 2, there are many other completely different approaches to parallel execution of logic programs. In some of them one can hardly discover any trail of the SLD-Resolution. Although it is easier to compare models within the same framework, it is not a easy task to determine which approaches are better than the others. Comparison is much easier done theoretically than practically. Many models are designed on architectures yet to be built. For the Competition Model, we have written a simulation system in C under Unix, and found that many features are not available and need to be simulated. Such limitation and the likes are definitely an obstacle in comparing the efficiency of different models.

It seems that defining a general framework and controlling the parallelism through meta-programming is a promising idea to be explored. This shall provide much flexibility and hence efficiency than a fixed model.

It is believed that there will soon be a breakthrough in the parallel execution of Constraint Logic Programming (CLP) [Jaffar and Lassez, 1987]. The *constraints* are by all means predicates. Therefore parallelism has already been implicitly embedded in the "sequential" execution of Constraint Logic Programs.

4.2.2 First Order Logic

SLD-Resolution is developed from the full Resolution principle. The latter is designed for general first-order clauses. To execute general first-order clause programs following the full Resolution principle is extremely inefficient. It should be a very interesting and challenging research topic to overcome this problem.

4.2.3 Other Logics

There are logics other than the first-order predicate logic. Their adoptions in logic programming are interesting topics. It is noted that some of these logics have had a theoretical basis of mechanical theorem proving (e.g. the Recursive Resolution method for Modal Logic [Chan, 1987]). We are expecting efficient logic programming models to be designed for these logic systems in the future.

PUBLICATIONS

AND-Parallel Execution of Logic Programs, *Proceedings of the Australian Joint Artificial Intelligence Conference*, Sydney 1987.

The Competition Model for Parallel Execution of Logic Programs, *Proceedings of the Fifth International Conference Symposium on Logic Programming*, Seattle 1988 (to appear).

Competition : A Model of AND-Parallel Execution of Logic Programs, *The Computer Journal* (to appear).

REFERENCES

Ali, K. A. M. (1987). OR-Parallel Execution of Prolog on a Multi-Sequential Machine, *International Journal of Parallel Programming*, 15:3(1987), 189-214.

ANSI X3H3. *Computer Graphics Virtual Device Interface*. March 1984.

Apt, K. R. and van Emden, M. H. (1982). Contribution to the Theory of Logic Programming, *JACM* 29:3(1982), 841-862.

Chan, M. C. (1987). The Recursive Resolution Method for Modal Logic, *New Generation Computing*, 5(1987), 155-183.

Chang, J. H., Despain, A. M. and DeGroot, D. (1985). And-Parallelism of Logic Programs Based on a Static Data Dependency Analysis, *IEEE COMPCON Spring '85*, 218-225.

Ciepielewski, A. and Haridi, S. (1983). A Formal Model for OR-Parallel Execution of Logic Programs, *IFIP '83*, 299-305.

Clark, K. and Gregory, S. (1986). PARLOG: Parallel Programming in Logic, *ACM Transaction on Programming Languages and Systems*, 8:1(1986), 1-49.

Clocksin, W. F. (1987). Principles of the DelPhi Parallel Inference Machine, *The Computer Journal*, 30:5(1987), 386-392.

Clocksin, W. F. and Mellish, C. S. (1981). *Programming in Prolog*, Springer-Verlag, 1981.

Codognet, C., Codognet, P. and File, G. (1986). A Very Intelligent Backtracking Method for Logic Programs, *Proceedings of ESOP '86*, Saarbrucken 1986.

Conery, J. S. (1983). The AND/OR Process Model for Parallel Interpretation of Logic Programs, Ph.D. Thesis, University of California, Irvine, 1983.

Conery, J. S. (1987). Implementing Backward Execution in Nondeterministic AND-Parallel Systems, *Proceedings of The Fourth International Conference on Logic Programming*, Melbourne 1987.

Conery, J. S. and Kibler, D. F. (1985). AND Parallelism and Nondeterminism in Logic Programs, *New Generation Computing*, 3(1985), 43-70.

DeGroot, D. (1984). Restricted AND-Parallelism, *Proceedings of The International Conference on Fifth Generation Computer Systems 1984*, 471-478.

Frege, G. (1879). Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens. [Translation in English "Begriffsschrift, a formula language, modelled upon that of arithmetic, for pure thought" reprinted in *From Frege to Godel: a Source Book in Mathematical Logic, 1879 - 1931*, J. van Heijenoort (Eds.), Harvard University Press, 1967]

Hermenegildo, M. V. and Nasr, R. I. (1986). Efficient Management of Backtracking in AND-Parallelism, *Proceedings of The Third International Conference on Logic Programming*, London 1986.

Hill, R. (1974). LUSH-Resolution and its Completeness, DCL Memo 78, Department of Artificial Intelligence, University of Edinburgh, 1974.

Hoare, C. A. R. (1974). Monitors: An Operating System Structuring Concept, *CACM* 17:**10**(1974).

Jaffar, J. and Lassez J.-L. (1987). Constraint Logic Programming, *Proceedings of the 14th ACM POPL Conference*, Munich 1987.

Kale, L. V. (1987). The REDUCE-OR Process Model for Parallel Evaluation of Logic Programs, *Proceedings of The Fourth International Conference on Logic Programming*, Melbourne 1987.

Kasif, S. and Minker, J. (1985). The Intelligent Channel: A Scheme for Result Sharing in Logic Programs, *IJCAI '85*, 29-31.

Kowalski, R. A. (1974). Predicate Logic as a Programming Language, *IFIP 74*, 569-574.

Kowalski, R. A. and Kuehner, D. (1971). Linear Resolution with Selection Function, *Artificial Intelligence* **2**(1971), 227-260.

Li, P. P. and Martin, A. J. (1986). The Sync Model: A Parallel Execution Method for Logic Programming, *Proceedings of Symposium on Logic Programming 1986*, Salt Lake City 1986.

Lin, Y.-L. and Kumar, V. (1986). A Execution Model for Exploiting AND-Parallelism in Logic Programs, Technical Report AI TR86-34, Department of Computer Sciences, University of Texas at Austin.

Lin, Y.-L., Kumar, V. and Leung, C. (1986). An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs, *Proceedings of the Third International Conference on Logic Programming*, London 1986.

Lloyd, J. W. (1984). *Foundations of Logic Programming*, Springer-Verlag 1984.

Ng, K. W., Leung, H. F. and Yu, C. S. (1987). AND-Parallel Execution of Logic Programs, *Proceedings of Australian Joint Artificial Intelligence Conference*, Sydney 1987.

Ng, K. W. and Leung, H. F. (1988). The Competition Model for Parallel Execution of Logic Programs, *Proceedings of the Fifth International Conference Symposium on Logic Programming*, Seattle 1988 (to appear).

Ng, K. W. and Leung, H. F. (1989). Competition : A Model of AND-Parallel Execution of Logic Programs, *The Computer Journal* (to appear).

Robinson, J. A. (1965). A Machine-oriented Logic Based on the Resolution Principle, *JACM* 12:1(1965), 23-41.

Shapiro, E. Y. (1983). A Subset of Concurrent Prolog and its Interpreter, ICOT Technical Report TR-003, Tokyo 1983.

Shapiro, E. Y. and Stirley, L. (1986). *The Art of Prolog: advanced programming techniques*, MIT Press, 1986.

Sun, C. Z. and Tzu, Y. G. (1986). The OR-forest Description for the Execution of Logic Programs, *Proceedings of The Third International Conference on Logic Programming*, London 1986.

Ueda, K. (1985). Guarded Horn Clauses, ICOT Technical Report TR-103, Tokyo 1985.

Warren, D. H. D. (1987). Or-Parallel Execution Models of Prolog, *TAPSOFT '87*, Pisa 1987.

Woo, N. S. and Choe, K.-M. (1986). Selecting the Backtrack Literal in the AND/OR Process Model, *Proceedings of Symposium on Logic Programming 1986*, Salt Lake City 1986.

Yang, R. (1987). A Parallel Logic Programming Language and its Implementation, Ph.D. Thesis, Keio University, Tokyo 1986.

Appendix A

In this appendix we shall prove the lemmas 1, 3 and 4 in Chapter 3.

Lemma 1. If F fails, then B can possibly cure the failure of F if and only if $B \in X(F)$ and $B <_t F$.

Proof: The *if* part is trivial. Now we concentrate on the *only if* part. Firstly, if B is not in $X(F)$, then B is neither an ancestor of F, nor an ancestor of failed descendents of F. Redoing B hence does not affect the success or failure of F. Therefore, B must be in $X(F)$. In addition, if $F <_t B$, then B is not an ancestor of F, which means that B is an ancestor of an ever failed descendent of F. By the algorithm, B must have been backtracked to before F. Now that F fails, it is evidence that redoing B does not prevent the failure of F. Therefore, it must be true that $B <_t F$.

Lemma 3. After F places marks on its ancestors, let

$$S(F) = \{P \mid P <_t F \text{ and } \mathrm{mark}(P) \cap (\mathrm{mark}(F) \cup \{\mathbf{m}_F\}) \neq \varnothing\}$$

then $S(F) = X(F)$

Proof: The set $X(F)$ is the union of two subsets: $A(F)$, the set of ancestors of F, and $A_D(F)$, the set of ancestors of the failed descendents of F which are before F in the '$<_t$' order. Note that $S(F)$ is also the union of two sets: $S_1(F) = \{P \mid P <_t F \text{ and } \mathrm{mark}(P) \cap \{\mathbf{m}_F\} \neq \varnothing\}$, and $S_2(F) = \{P \mid P <_t F \text{ and } \mathrm{mark}(P) \cap \mathrm{mark}(F) \neq \varnothing\}$. When F fails, it places marks on it ancestors. It is therefore evident that $A(F) = S_1(F)$. On the other hand, F is marked by D if and only if D is a descendent of F and D has ever failed, therefore $S_2(F) = A_D(F)$. Hence, $S(F) = X(F)$.

Lemma 4. If a literal B is reset or backtracked to, and it successfully finishes, it is sufficient to reset the literals R such that $B <_l R$ and mark(B) $\cap$ mark(P) $\neq \varnothing$.

Proof: This can be proved by considering different cases for every consumer C of B. We shall make use of the fact that completeness is ensured if all parents of any consumers of B, which are after B in the '$<_l$' order, are reset.

Firstly, if the consumer C has ever failed, it should have marked all its parents (including B), hence all parents of C later than B in the '$<_l$' order will be reset. Completeness is therefore ensured.

Secondly, if the consumer C has never failed, and none of its parents has been backtracked to for curing the failure of other consumer(s), then there is no need to reset the parents of C.

Thirdly, consider the case that the consumer C has never failed, and some of its parents have been backtracked to for curing the failure of consumers other than C. There are two possibilities. If B is an ancestor of some of these ever failed consumers, then the backtracked generators will be reset, and no problem arises. Otherwise, since B is not an ancestor of these failed consumers, backtracking to B or resetting B does not affect these consumers. Consequently, even if the backtracked parents of C are reset, the identical backtracking sequences will take place again and finally lead to the configuration before the resetting. Consequently such resetting is not necessary.

Hence, the lemma is correct.

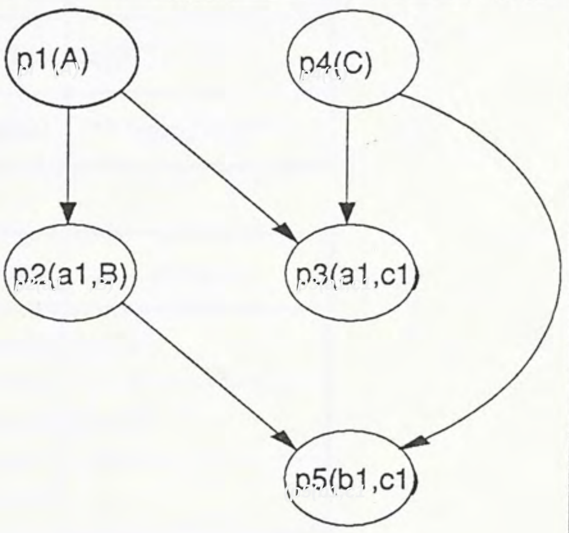
Appendix B

In this appendix we shall illustrate the execution of the Competition Model with several examples. The execution traces are actual ones produced by our simulation system. Therefore, such sequences are not unique.

Example 1

The following program is quoted from example B.1 of Appendix B of [Lin, Kumar and Leung, 1986].

```
p1(a1).

p2(a1,b1).

p2(a1,b2).

p3(a1,c1).

p3(a1,c2).

p4(c1).

p5(b1,c2).

p5(b2,c1).
```



The Data Dependency Graph up to the first failure

```
?- p1(A),p2(A,B),p3(A,C),p4(C),p5(B,C).
```

The following is a trace of the execution of the program.

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A)	p1(A)	$\{\mathbf{m}_{p5}\}$	1	1	finished	{a1/A}
p2(A,B)	p2(a1,B)	$\{\mathbf{m}_{p5}\}$	2	3	active	*** backtracked ***
p3(A,C)	p3(a1,c1)	{}	2	4	finished	
p4(C)	p4(C)	$\{\mathbf{m}_{p5}\}$	1	2	finished	{c1/C}
p5(B,C)	p5(B,c1)	{}	3	5	frozen	*** failed ***

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A)	p1(A)	$\{\mathbf{m}_{p5}\}$	1	1	finished	{a1/A}
p2(A,B)	p2(a1,B)	$\{\mathbf{m}_{p5}\}$	2	3	finished	{b2/B}
p3(A,C)	p3(a1,c1)	{}	2	4	finished	
p4(C)	p4(C)	$\{\mathbf{m}_{p5}\}$	1	2	finished	{c1/C}
p5(B,C)	p5(b2,c1)	{}	-	-	inactive	*** cancelled ***

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A)	p1(A)	$\{\mathbf{m}_{p5}\}$	1	1	finished	{a1/A}
p2(A,B)	p2(a1,B)	$\{\mathbf{m}_{p5}\}$	2	3	finished	{b2/B}
p3(A,C)	p3(a1,c1)	{}	2	4	finished	
p4(C)	p4(C)	$\{\mathbf{m}_{p5}\}$	1	2	finished	{c1/C}
p5(B,C)	p5(b2,c1)	{}	3	6	active	

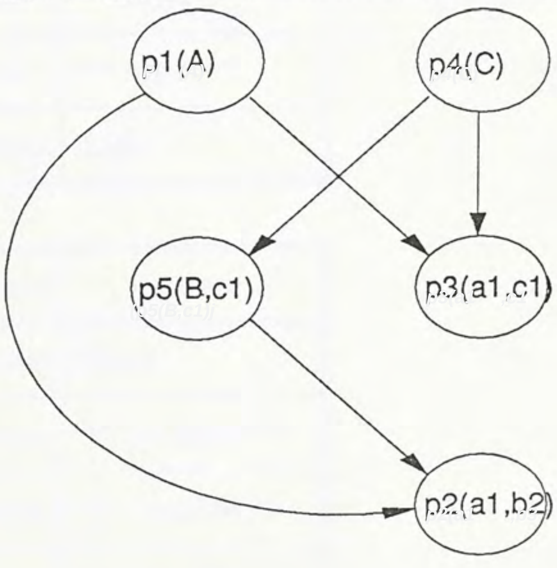
literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A)	p1(A)	{$\mathbf{m}_{p3}$}	1	1	finished	{a1/A}
p2(A,B)	p2(a1,B)	{$\mathbf{m}_{p5}$}	2	3	finished	{b2/B}
p3(A,C)	p3(a1,c1)	{}	2	4	finished	
p4(C)	p4(C)	{$\mathbf{m}_{p5}$}	1	2	finished	{c1/C}
p5(B,C)	p5(b2,c1)	{}	3	6	finished	

The final answer substitution is {a1/A, b2/B, c1/C}.

Example 2

The following program is quoted from example B.2 of Appendix B of [Lin, Kumar and Leung, 1986].

```
p1(a1).

p1(a2).

p2(a1,b1).

p2(a2,b2).

p3(a2,c1).

p3(a1,c2).

p4(c1).

p5(b1,c2).

p5(b2,c1).
```



The Data Dependency Graph up to the first failure

```
?- p1(A),p2(A,B),p3(A,C),p4(C),p5(B,C).
```

The following is a trace of the execution of the program.

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A)	p1(A)	{$\mathbf{m}_{p3}$}	1	1	finished	{a1/A}
p2(A,B)	p2(a1,b2)	{}	3	5	frozen	
p3(A,C)	p3(a1,C)	{}	2	4	frozen	*** failed ***
p4(C)	p4(C)	{$\mathbf{m}_{p3}$}	1	2	active	*** backtracked ***
p5(B,C)	p5(B,C)	{}	2	3	finished	{b2/B}

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A)	p1(A)	{$\mathbf{m}_{p3}$}	1	1	active	*** backtracked ***
p2(A,B)	p2(A,b2)	{}	3	5	frozen	
p3(A,C)	p3(A,C)	{}	2	4	frozen	*** failed ***
p4(C)	p4(C)	{$\mathbf{m}_{p3}$}	1	2	active	*** failed ***
p5(B,C)	p5(B,C)	{}	2	3	finished	{b2/B}

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A)	p1(A)	{$\mathbf{m}_{p3}$}	1	1	finished	{a2/A}
p2(A,B)	p2(a2,b2)	{}	-	-	inactive	*** cancelled ***
p3(A,C)	p3(a2,C)	{}	-	-	inactive	*** cancelled ***
p4(C)	p4(C)	{$\mathbf{m}_{p3}$}	1	2	active	*** reset ***
p5(B,C)	p5(B,C)	{}	2	3	finished	{b2/B}

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A)	p1(A)	{$\mathbf{m}_{p3}$}	1	1	finished	{a2/A}
p2(A,B)	p2(a2,b2)	{}	3	6	finished	
p3(A,C)	p3(a2,c1)	{}	2	7	active	
p4(C)	p4(C)	{$\mathbf{m}_{p3}$}	1	2	finished	{c1/C}
p5(B,C)	p5(B,c1)	{}	2	3	finished	{b2/B}

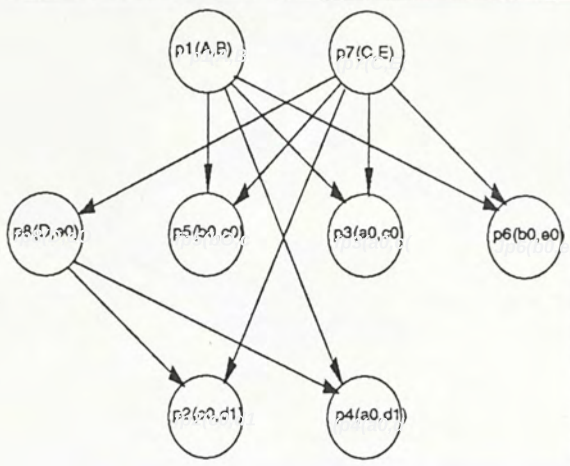
literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A)	p1(A)	{$\mathbf{m}_{p3}$}	1	1	finished	{a2/A}
p2(A,B)	p2(a2,b2)	{}	3	6	finished	
p3(A,C)	p3(a2,c1)	{}	2	7	finished	
p4(C)	p4(C)	{$\mathbf{m}_{p3}$}	1	2	finished	{c1/C}
p5(B,C)	p5(B,c1)	{}	2	3	finished	{b2/B}

The final answer substitution is {a2/A, b2/B, c1/C}.

Example 3

The following program is quoted from [Woo and Choe, 1986]. Our simulation implementation finds the answer substitution without backtracking. The data dependency graph is shown on the right.

```
p1(a0,b0).

p2(c0,d0).

p2(c0,d1).

p3(a0,c0).

p4(a0,d0).

p4(a0,d1).

p5(b0,c0).

p6(b0,e0).

p6(b0,e1).

p7(c0,e0).

p8(d0,e1).

p8(d1,e0).
```



The Data Dependency Graph

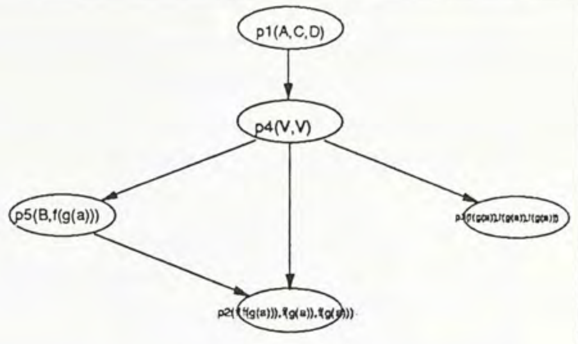
```
?- p1(A,B), p2(C,D), p3(A,C), p4(A,D), p5(B,C), p6(B,E), p7(C,E),
p8(D,E).
```

The answer substitution is {a0/A, b0/B, c0/C, d1/D, e0/E}.

Example 4

The following example shows the case when non-ground bindings are involved. It is quoted from [Ng, Leung and Yu, 1987].

```
p1(X,X,X).

p1(X,X,Y).

p2(Y,Y,X).

p2(X,Y,Z).

p3(X,f(X),_).

p4(X,f(g(a))).

p5(f(X),X).

p5(f(g(X)),g(f(X))).
```



The Data Dependency Graph up to the first failure

```
?-p1(A,C,D), p2(B,A,C), p3(C,D,A), p4(A,D), p5(B,C).
```

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A,C,D)	p1(A,C,D)	{m_{p3}}	1	1	finished	{V/A,V/C,V/D}
p2(B,A,C)	p2(f(f(g(a))),V,V)	{}	4	5	frozen	
p3(C,D,A)	p3(V,V,V)	{}	3	4	frozen	*** failed ***
p4(A,D)	p4(V,V)	{m_{p3}}	2	2	active	*** backtracked ***
p5(B,C)	p5(B,V)	{}	3	3	finished	{f(f(g(a)))/B}

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A,C,D)	p1(A,C,D)	{m_{p3},m_{p4}}	1	1	active	*** backtracked ***
p2(B,A,C)	p2(f(f(g(a))),A,C)	{}	4	5	frozen	
p3(C,D,A)	p3(C,D,A)	{}	3	4	frozen	*** failed ***
p4(A,D)	p4(A,D)	{m_{p5}}	2	2	frozen	*** failed ***
p5(B,C)	p5(f(f(g(a))),C)	{}	3	3	finished	{f(f(g(a)))/B}

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A,C,D)	p1(A,C,D)	$\{\mathbf{m}_{p3},\mathbf{m}_{p4}\}$	1	1	finished	{W/A,W/C,X/D}
p2(B,A,C)	p2(B,W,W)	{}	-	-	inactive	*** cancelled ***
p3(C,D,A)	p3(W,X,W)	{}	-	-	inactive	*** cancelled ***
p4(A,D)	p4(W,X)	{}	-	-	inactive	*** cancelled ***
p5(B,C)	p5(B,W)	{}	-	-	inactive	*** cancelled ***

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A,C,D)	p1(A,C,D)	$\{\mathbf{m}_{p3},\mathbf{m}_{p4}\}$	1	1	finished	{W/A,W/C,X/D}
p2(B,A,C)	p2(B,W,W)	{}	-	-	inactive	
p3(C,D,A)	p3(W,X,W)	{}	-	-	inactive	
p4(A,D)	p4(W,X)	{}	2	6	active	{?/W,?/X}
p5(B,C)	p5(B,W)	{}	-	-	inactive	

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A,C,D)	p1(A,C,D)	$\{\mathbf{m}_{p3},\mathbf{m}_{p4}\}$	1	1	finished	{W/A,W/C,X/D}
p2(B,A,C)	p2(B,Y,Y)	{}	-	-	inactive	
p3(C,D,A)	p3(Y,f(g(a)),Y)	{}	-	-	inactive	
p4(A,D)	p4(W,X)	{}	2	6	finished	{Y/W,f(g(a))/X}
p5(B,C)	p5(B,Y)	{}	3	7	active	{?/B,?/Y}

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A,C,D)	p1(A,C,D)	$\{\mathbf{m}_{p3},\mathbf{m}_{p4}\}$	1	1	finished	{W/A,W/C,X/D}
p2(B,A,C)	p2(f(Z),Z,Z)	{}	-	-	inactive	
p3(C,D,A)	p3(Z,f(g(a)),Z)	{}	4	8	active	{?/Z}
p4(A,D)	p4(W,X)	{}	2	6	finished	{Y/W,f(g(a))/X}
p5(B,C)	p5(B,Y)	{}	3	7	finished	{f(Z)/B,Z/Y}

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A,C,D)	p1(A,C,D)	$\{\mathbf{m}_{p3},\mathbf{m}_{p4}\}$	1	1	finished	{W/A,W/C,X/D}
p2(B,A,C)	p2(f(g(a)),g(a),g(a))	{}	5	9	active	
p3(C,D,A)	p3(Z,f(g(a)),Z)	{}	4	8	finished	{g(a)/Z}
p4(A,D)	p4(W,X)	{}	2	6	finished	{Y/W,f(g(a))/X}
p5(B,C)	p5(B,Y)	{}	3	7	finished	{f(Z)/B,Z/Y}

literal	*binding*	*mark set*	*level*	*order-no.*	*status*	*remarks*
p1(A,C,D)	p1(A,C,D)	$\{\mathbf{m}_{p3},\mathbf{m}_{p4}\}$	1	1	finished	{W/A,W/C,X/D}
p2(B,A,C)	p2(f(g(a)),g(a),g(a))	{}	5	9	finished	
p3(C,D,A)	p3(Z,f(g(a)),Z)	{}	4	8	finished	{g(a)/Z}
p4(A,D)	p4(W,X)	{}	2	6	finished	{Y/W,f(g(a))/X}
p5(B,C)	p5(B,Y)	{}	3	7	finished	{f(Z)/B,Z/Y}

The final answer substitution is {g(a)/A, f(g(a))/B, g(a)/C, f(g(a))/D}.

Appendix C A Simulation Implementation of the Competition Model

C.1 Introduction

The Competition Model [Ng and Leung, 1988, 1989] is a parallel execution model designed in the framework of the AND/OR Process Model [Conery, 1983], which supports both AND- and OR-parallelism. However, like other models in the same framework [Chang, Despain and DeGroot, 1985] [DeGroot, 1984] [Hermenegildo and Nasr, 1986] [Lin and Kumar, 1986], the OR-parallelism in the Competition Model is limited, and the aim of the design is hence an efficient AND-parallel execution algorithm.

Since the model is an abstract model, its implementation using available architecture is not straightforward. For example, in some operating systems the creation and deletion of asynchronous processes and the communication among them are quite time-consuming. On the other hand, in some operating systems such operations are not as handy as required if the program is written in high level languages. Under such consideration, a real implementation is only possible if it is written in assembly language and run on an ppropriate machine. Such an implementation project is too large to be included in this research project (*c.f.* [Conery, 1987]). Therefore, a simulation implementation with graphical illustration of execution processes is built.

The simulation system is implemented under the Sun Unix operating system of the Sun-3/110C workstation, which supports asynchronous process execution and process communication. The graphics illustration is implemented using the SunCGI package, which supports colour graphics display with the colour monitor connected to the workstation. The programs are all written in the C language since it is the natural language of Unix.

C.2 An Overall Description

The implementation only simulates the AND-parallel execution in the Competition Model. The OR-parallelism is omitted for the ease of implementation. The input program is hence restricted to consist of solely unit clauses for this reason.

An AND-process is the core of the implementation. This "AND-process" does more than required in the Model. It includes, in addition to the control mechanism described in the AND-Parallel execution algorithm, a parser of logic program and query, and the display routines for the graphical demonstration. After parsing the clauses and the query, the AND-process creates several OR-processes and controls their execution in the way specified in the algorithm.

The "OR-processes" in the implementation simulate the operation of real OR-processes. Since the program consists of solely unit clauses, there is no need for the OR-processes to create child AND-processes. However, in order that the simulation is more like the real execution, the OR-processes sleep for a random period of time before reporting the answer substitution to their common parent AND-process. In a real execution this sleeping period will be the time an OR-process takes to find the answer substitution by invoking its child AND-processes.

In order to share the program among the AND- and OR-processes, the AND-process stores the parsed program in a file, which is then opened and read by the OR-processes.

The communication between the AND-process and the OR-processes is via the pipes, which are duplex FIFO message queues. Signals and their handling is also provided by the operating system, but they are not used because signals are blocked and lost when the signalled process is handling the same signal. Signalling is therefore not considered to be a reliable way of communication.

The graphics display is implemented using the SunCGI package, which is a Sun implementation of the developing ANSI CGI standard [ANSI X3H3]. It includes certain extensions to the standard, such as the manipulation of multiple view surfaces. In the simulation implementation this extension is made use of and different information is displayed in different view surfaces.

The whole implementation is written in the C language, because it is the only available high level language supporting parallel processing in the current Sun Unix. The process manipulation and communication is performed via Unix system calls.

C.3 The AND-Process

Data Structures

The AND-process maintains three main tables: the literal table, the variable table, and the data dependency graph.

The literal table stores the information of each literals. Each entry of this table includes the literal in parsed form, its current instantiation, its execution level, its order number, its mark set, its current status (active / inactive / finished / frozen) and its current execution condition (normal / reset / backtracked). The system information, such as the process identity number and the pipe numbers are also stored in this table. In addition, the locations of its graphics display images are also recorded in this table.

The variable table stores the symbolic name and current binding for each variable, as well as its generator. For the new variables introduced during execution, their symbolic names are, by the Prolog convention, unique decimal numbers preceded by an underscore ('_').

The data dependency graph is stored as an adjacency matrix. The element e_{ij} is set to 1 if and only if literal i is a parent of literal j.

Program structure

The AND-process in the simulation implementation includes a parser for logic program, a parser for query, and the control mechanisms required by the Competition Model. The graphics display routines are embedded in the control mechanisms, and are invoked at appropriate time.

The Parsers

The parsers for the program and the query are both simple recursive descent parsers. They are very similar in structure. The reason that they are separated is that the program clauses and the query clause have different formats, and it is easier to write two different parsers than one to cater for different requirements. The program can be typed in through the terminal, or read from a file. The query is expected to be input from the terminal because it is expected that several queries can be typed in and executed after the program is parsed. (A query stored in a file can be fed into the system by a Unix '<' redirection symbol.) The parsers are called before the graphics display begins.

The Control Mechanisms

In the proposed model, the tokens of the variables are stored in a monitor [Hoare, 1974]. However, it is difficult to implement a real monitor in Unix using C. Therefore, in the simulation system the operation of monitor is replaced by the select-and-test process as described in the following paragraph.

Before the main execution begins, an OR-process is created for each literal in the literal table. This is done by the *fork()* and *execl()* system calls. Communication is through pipes created by the *pipe()* system call. When there are still inactive literals, one of them is selected randomly, and tested, so as to determine whether it is a good literal to start. If it is not, another one is tried. If it is, then the instantiated goal is sent to the corresponding OR-process together with the NEW_D direction (see section 4). This is a simulation of the events that the asynchronous processes are competing to enter the monitor.

Every time after the AND-process selects and tests a literal for starting, it takes a glance at its pipe to see whether a child OR-process returns a message. If there is no message, the AND-process goes on selecting and testing literals. Otherwise, it performs appropriate actions. Ideally, this should be done by interrupt signal and the actions should be carried out by an interrupt handler. However, in Unix this is impossible, because when the process is in an interrupt handler, future events of the same signal interrupt are blocked and lost.

An AND-process receives several kinds of message from its children. If the message is a success message from a normal literal, it will be followed by the answer substitution. The AND-process then modifies the variable table accordingly. If the message is a failure message, the failed literal is identified from the message, and the backtrack literal is determined and sent a NEXT_D direction (see section 4). Some literals are "frozen" according to the algorithm. If the message is a success message from a backtrack or reset literal, apart from that the variable will be modified according to the answer substitution, certain literals will be "cancelled" and the others "reset" as required by the algorithm.

The data dependency graph is modified accordingly during the operation of the AND-process. This adjacency matrix always reflects the current structure of the constructed data dependency graph, including the frozen subgraphs.

C.4 The OR-Processes

As mentioned above, the OR-processes in the simulation implementation are simpler than depicted in the algorithm. An OR-process in the simulation implementation maintains two main structures: the parsed program and a variable table.

The main routine of an OR-process is simple. An OR-process, after created, reads the parsed program from the file. Afterwards, it waits for directions from its parent AND-process and then performs actions accordingly.

There are three directions: the NEW_D direction, the NEXT_D direction, and the RESET_D direction. When the NEW_D direction is received, which is always accompanied by a new instantiated goal, the OR-process resets its clause pointer to point to the first program clause. Then the OR-process searches the parsed program from top to bottom to see whether there is a clause unifiable with the goal. If there is, then the substitution is reported to the parent AND-process in a success message after the OR-process sleeps for a random period of time (0 - 7 seconds). If there is no such clause found, a failure message is sent to the AND-process after a random period of time.

The NEXT_D direction is received by the literal which is backtracked to. The OR-process behaves as if it had received a NEW_D direction. However, the difference is that it will not expect to receive a new instantiated goal. nor will it reset the clause pointer.

If the RESET_D direction is reset, the operation carried out by the OR-process is same as that triggered by the NEXT_D direction, except that the clause pointer is reset.

C.5 The Graphics Display

The goal of the graphics display in this implementation is to show the execution of the algorithm by graphics images. The SunCGI package is used to display colour images on the colour monitor.

The implementation is expected to run in the SunView environment, in which SunCGI provides the necessary facilities to display with several different graphics windows called view surfaces. During execution, three different view surfaces are opened to display the images. The first one of these shows a "title page," on which static text is displayed. The second view surface shows the inactive set as well as the constructed data dependency graph. Since both of the inactive set and the data dependency graph are changing until the end of execution, dynamic display of literals is observable in this view surface. The other view surface shows text for explanation. The text explains what is happening during execution, such as activation, freezing, melting, etc..

This portion of implementation is still under development.

C.6 Discussions

The current implementation is a simulation system, but a real implementation can be built in common architecture.

It is felt that building a real implementation is very similar to building a multi-tasking operating system. Both of them need process manipulation and scheduling, communication, resource allocation, interrupt handling, memory management, etc.. In fact, if a good multi-tasking operating system is available and such functions can be called whenever it is required, a real implementation can easily be built, no matter whether the underlying architecture is a single processor or a multi-processor machine. However, usually it is only possible in the assembly or hardware level. Therefore it is concluded that a real implementation is too large a project to be included in this research.

C.7 Summaries

Although it is the desired goal that a real implementation of the Competition Model is built, the current simulation implementation is built instead due to various difficulties. However, the execution depicted by the simulation implementation resembles what can be found in a real implementation. The only difference is that the speed should be much faster in the real implementation, which is written as assembly programs running on appropriate architecture without graphics display.

Due to the above consideration, it is difficult to conclude by considering the performance of this simulation implementation whether the gain in efficiency due to parallel execution is greater than the loss due to the overhead, although it is our belief that this should be true. However, this implementation can be used to testify the correctness of the algorithm, and to make it easier to understand the algorithm because of its attractive colour graphics display of execution.