通信情報保護學會論文誌
第 2 卷 第 1 號, 1992. 6

개선된 병렬 논리프로그래밍 시스템의 복잡도 분석

정인정* · 이흥규**

Complexity Analysis for the Improved Parallel Logic Programming Systems†

In-Jeong Chung and Heung-Kyu Lee

요 약

본 논문은 컴퓨터 시스템의 보호 및 데이타 통신의 암호등과 관계가 있는 논리 프로그래밍에 대한 것이다. 본 논문에서 우리는 병렬 논리프로그래밍 시스템에 대한 개선된 control strategy를 제안하였으며, 이에 대한 형식적인 syntax와 semantics등을 정의하였다. 병렬 논리프로그램에 대한 병렬성등을 분류하여, 이들이 어떻게 활용되는지 설명하였다. 또한 병렬 논리프로그램의 유도과정과 계산방식이 같은 alternating Turing machine이란 병렬 계산모형을 제안하여, 병렬 논리프로그램의 복잡도의 분석과 제안된 idea에 대한 타당성을 ATM을 이용하여 하였다.

Abstract

This paper deals with the logic programming which is related with the computer system security and the encryption of data communications. This paper suggests an improved control strategy for the parallel logic programming systems, shows its formal syntax and semantics. We classify the potential parallelisms of logic programs, and explain how these parallelisms can be utilized in the logic programs. It then associates parallel logic programs with the highly abstract computation model, called alternating Turing machine(ATM). ATM does capture the essence the procedural interpretation of parallel logic programs, and the complexity of computations of ATM represents a useful measure of complexity of derivations of logic programs. We analyze the complexity of parallel logic programs, and show the justification for our suggested idea utilizing ATM.

† "이 논문은 1990년도 문교부 지원 한국 학술진흥 재단의 지방대 육성 학술 연구 조성비에 의하여 연구되었음."
* 고려대학교 자연과학대학 전산학과 부교수
** 한국과학기술원 전산학과 조교수

1. Introduction

Logic programming is one of the declarative programming approaches which has the declarative semantics as well as operational semantics. Logic programming is very suitable not only to represent the nondeterministic asynchronous parallel computations, but also to address the encapsulatin of operating systems and security or encryption of communication. The abstract operations of logic programs are generally useful in a world of mutually mistrusted but cooperating agents. Cryptography is only necessary for implementing the operations of logic programs when there is no mutually trusted secure agent. In this sense, logic programming systems provide such a secure, reliable agent for the embedded computations: they can be used as a foundatin for secure multi-user systems. Shapiro[13] proposed a foundation for secure distributed multi-user operating systems using Concurrent Prolog.

The collected papers in Shapiro[15] survey recent research in parallel logic programming systems. They describe parallel logic languages: investigate how logic programming systems carry out system programming and nondeterministic parallel algorithms: how to embed high-level language: how to implement this language efficiently on sequential and parallel computers.

Volume 1 of Shapiro[15] describes parallel logic programming languages: contains the major topics of parallel logic programming systems: addresses the complexity of systoc logic programs: mentions on distributed programming methodologies: how these languages represent nondeterministic parallel algorithms: how the potential parallelism can be exploited in logic programming programs.

Volume 2 investigates the relationship between the system programming and logic languages: reports on research towards advanced techniques for programs development such as meta-interpreters, program transformations and partial bindings: describes an implementation techniques for logic languages: treats techniques for emdedding the high-level programming paradigms such as object-oriented, logic and functional programming.

It also reports on a method for secure communication in parallel logic programming systems. The control and mapping techniques draw heavily on the concepts of meta-interpretation and partial evaluation. Some logic languages such as Flat Concurrent Prolog can be used for the encapsulation and encryption. Miller et al in Shapiro[15] addressed the encryption and security problem for the banking program, using Concurrent Prolog. It presents a secure substrate for secure systems and provides an implementation of banking system as an axample.

The idea of logic programming, due to Kowalski[10], is that an algorithm is composed of two parts, logic and control: i.e. algorithm = logic + control. The logic corresponds to what the algorithm wants to solve, and the control corresponds to how it can be solved. Ideally the users would prefer to restrict their attention to the logic part only, and leave the control part to the implementation. i.e. an ideal logic programming systems would allow the user to mention only the logic part of the algorithm, the control part should be transparent to user. Unfortunately, there is not yet such an ideal logic programming system.

The reason for the nonexistence of such an ideal logic programming system is tha there is difference between logic programming and logic programming program. The difference is that (pure) logic programming concerns only the logic part of the algorithm for its declarative meaning, but logic programming systems need to include some control

strategy(CS) to affect efficiency. To paraphrase the Kowalski's dictum[10] "algorithm = logic + control", logic programming systems = logic formalism + CS.

The CS enables a logic program to be executed on a computer, having an effect on its efficiency. It determines the operational semantics of logic programs, while the declarative semantics is decided by reading the program as a logical formula. Therefore we should have reasonably good CS for the parallel logic programming system to achieve reasonably good efficiency.

This paper proposes an improved CS for parallel logic programming systems, shows its formal syntax and semantics. It then associates the logic programs with the highly abstract parallel computation model, shows their relationships, and analyzes the complexity of suggested system with that model. It also shows that the suggested CS would yield the unnecessary AND-parallelism and OR-parallelism for parallel logic programming system.

2. Alternating in AND/OR Tree of Logic Programs

2.1 AND/OR Tree of Logic Programs

In the evaluation logic programming, the logic reduction for a query Q with a program P can be regarded as the task to find a solution to an initially given problem, using a given collection of facts and rules to reduce the programs to a set of subproblems. For this task, a graphical representation, called the AND/OR tree is used to illustrate the solution of a query using a given logic program.

Suppose that we are given a logic program which is a set of clauses of the form, $A:\text{-}B_1B_2, \cdots, B_m$ and the query of the form $:\text{-}q_1 \cdots, q_n$ where m, $n \geq 0$. In the parallel logic program, ',' procedurally means the fork. i.e. a conjunction p, q indicates that goals p and q will be solved by different processes.

Now we will give sketch how we can obtain the fully parallel AND/OR computation model from logic program P using two kinds of processes, AND-process and OR-process. The computation model can be represented by AND/OR tree using these AND/OR processes. In the following discussion, AND-nodes and OR-nodes of AND/OR tree are referred to as AND-processes and OR-processes, respectively.

(i) Initially the goal statement constitutes a root node, a special case of an OR-process. The label of this root node is the initial goal.

(ii) If there are several alternative clauses to a relation p in the logic program, draw an arc form this node with predicate symbol p in the tree to all possible nodes whose labels are the same predicate symbol p, but with different arguments. Since the original node with predicate symbol p in its label can be solved if any one of its children can be solved, we call this node OR node. Each subtree of an OR node is an OR branch, representing an alternative solution to the problem.

(iii) The offsprings of an OR nodes are AND nodes, one for each clause that unifies with the given relation call. The children of this AND node represent a set of subproblems to be solved. It is called AND node, since this set of all subproblems should be solved to satisfy the AND node. AND node is marked by placing arcs(half circle) on its branches.

(iv) A node which can be exactly matched with some fact clause in the logic program has no descendants. This node is called terminal node.

(v) The solution to the initial goal can be obtained at the root of the tree by finding

compatible sets of branches. i.e. those labelled by compatible substitutions.

Note that the fully parallel AND/OR computation model described above is a graphical representation of problem solving. It will be used to illustrate the solution of a query using a given logic program.

Note also that this representation is very abstracted AND/OR computation model which reveals nothing about the control. i.e. how the problem is solved.

2.2 Parallelism of logic programs

In this section we describe the major chracteristics of the possible parallelism in logic programs. The fully parallel AND/OR model with AND/OR tree described above can be interpreted with AND-parallelism and OR-parallelism. A computation of logic program amounts to the construction of a proof of an existentially quantified conjunctive goals from the axioms.

The computation progresses via nondeterministic goal reduction: at each step it has a current goal $A_1, \cdots A_n$: it arbitrarily seletcs a goal A_i, for some $1 \leq i \leq n$: it then nondeterministically selects a clause $A' :\!-B_1 \cdots B_k$ for which A and A' are unifiable via a substitution θ, and uses this clause to reduce the goal. The reduced goal is $(A_1, \cdots, A_{i-1}, B_1, \cdots, B_k, A_{i+1}, \cdots A_n)\theta$. The computation terminates when the current goal is empty.

This description readily suggests two kinds of parallel excution: the reduction of several goals in parallel, called the AND-parallelism, and a concurrent search of the computation paths resulting form different nondeterministic choices of the unifiable clauses, called OR-parallelism.

Therefore the parallel execution on the AND-node in the fully parallel AND/OR model described above corresponds to the AND-parallelism, whereas the parallel execution on the OR-node corresponds to the OR-parallelism. Note that the term AND-parallelism refers to a computation rule in which relation calls are evaluated concurrently. OR-parallelism is parallelism in the search rule, such that several clauses are acted on concurrently. As stated in Gregory et al[2),3),6)], it is the CS that enables a logic program to be executed on a computer. The CS decides both the computation rule as well as the search rule. i.e. the CS decides the efficiency of the execution of logic program.

We can classify the AND-parallelism and OR-parallelism more in detail as below:

1) Restricted AND-parallelism: the concurrent evaluation of several calls in a conjunction which are independent. i.e. do not share variables.

2) Stream AND-parallelism: the concurrent evaluation of several calls which share a variable with the value of shared variable communicated incrementally between the calls. Here the shared variable behaves as communication channel between the processes.

3) OR-parallelism: the concurrent application of several clauses in a procedure while solving a call.

4) All-solutions AND-parallelism: the concurrent evaluation of several calls in a conjunction, each working on a different solution.

Of these four possibilities, 1), 3), and 4) are perhaps easy to implement because each of the prallel processes is working on a different solution and they are independent. Stream AND-parallelism is different from other parallelism mentioned here in two aspects. First of all, it has the advantage that solutions can be communicated incrementally by means of single assignment property. On the other hand, it has the disadvantage that at most one solution can be obtained. It contrasts with the explorative parallel

logic programming programs such as Conery's AND/OR Process Model[4].

However stream AND-parallelism is a more primitive form of parallelism. If many solutions are required to a relation, these solutions can be obtained as members of a list, which is a single solution to another relation. By the incremental construction of the solution list, the solutions can be made available one at a time[6].

Options 1) and 2) differ from the corresponding functinal concepts in that the opportunities for AND parallel evaluation are not obvious from the structure of a Horn clause program. Some control mechanism or a sophisticated proof procedure may be necessary to decide both the independence of calls and the direction of stream communication through shared variables. Moreover stream AND-parallelism is difficult to implement in the presence of nondeterminism, since there should be some synchronization mechanism between the producer of the shared variables and consumer of them. To implement stream AND-parallelism efficiently, some means of constraining the nondeterminism is necessary. The existing parallel logic programming languages such as Concurrent Prolog[13]. Parlog[6] and GHC[17] are based on concepts of stream AND-parallelism and committed choice nondeterminism(CCN) as explained in section 3. 2.

3. Related Works

In pure logic programs without consideration to the control part, the operational semantics and declarative semantics coincide. The interpreter of pure logic programs follows all possible paths in the search space and hence its search space increases exponentially. i.e. blind breadth first search without any constructs for CS prohibits the practical parallel implementation of pure logic program due to its inherent combinatorial explosion. Therefore, for an implementable programming language we need the CS and supporting constructs so that the available parallelism is controlled and thereby limits the branching factor. Note that it is the CS of a logic program that enables a program to be executed on a computer in a practical way.

The nature of a CS decides the operational semantics of a logic program, while the declarative semantics is decided solely by reading the logic program. Different CSs might be obtained by varying the computation rule and search rule. This is the reason why the CS decides the efficiency of the implementaion of logic programming programs.

The CSs logic programming languages so far have following problems.

3.1 Prolog

Prolog requies a user to provide a lot of CS information, partly by the order of literals in a clause and partly by the extra-logical control features such as cut operator. The leftmost literal in a clause would be chosen first for execution, and the nondeterministic choice of the unifiable clause is simulated by sequential search and backtracking. Since the order of clauses in a Prolog program and the order of literals in a clause are important, the effectiveness of cut is critically dependent on the order of the clauses in a Prolog program.

Moreover Prolog uses the depth first search strategy which might introduce nonterminating possibilities, and the extra-logical control feature for its CS, called the cut operator.

The cut operator, which affects the operational behaviour of Prolog program, is used to reduce the search by pruning the search space. Its aims

are to prevent Prolog programs from subsequent useless computation paths due to the nondeterminism, or from nonterminating possibilities due to its depth first search strategy. It can be used to cut off computation paths that do not yield solutions, and thereby reducing the search space. However, its use is controversal, since it has some side-effects and asymmetric properties, and can only be interpreted by operational semantics(red cut), in contrast to the declarative style of logic programs. The red cut operator can change the nature of a Prolog program in the sense that the declarative meaning and the procedural meaning of the program would not be the same.

3.2 Parallel logic programming languages based on committed choice nondeterminism (CCN) and stream AND-parallelism

In order to overcome the deficiencies of Prolog described in the above section, some parallel logic programming languages such as Concurrent Prolog[13]. Parlog[6] and GHC[17] were introduced recently. These languages have the notion of nondeterministic commitment to a single clause in the same manner as the selection of a single statement list in Dijkstra's guarded commands[5]. This approach was influenced by the elegant adoption of Dijkstra's formalism in Hoare's CSP programming paradigm[7].

The CSs of these parallel logic programming languages are based on the CCN[7] and efficient implementation of stream AND-parallelism using the commit operator as described below.

CCN is the policy of committing to the body of the clause with the first guard that evaluates to be true, disregarding all other choices. Briefly, it is the 'don't care' nondeterminism, in which the system 'doesn't care' how a solution is obtained, so does not need to search. It is the basis of Dijkstra's language of guarded commands and Hoare's CSP. Kowalski[10] indicates that it is the intelligent backtracking in the sense that it eliminates unnecessary searching.

In the parallel logic programming languages based on CCN, every clause except the goal has the form of A:-G_1, G_2, $\cdots G_m \mid B_1, \cdots B_n$ where m, n $\geq$ 0. The construct ' $\mid$ ' is called the commit operator. The left side of the commit operator is called the guard part, and the right part is called the body part. The declarative reading of the above clause is: A is true if $G_1 \cdots G_m$ and $B_1, \cdots B_n$ are true.

Operationally, the guard is a test that must be executed successfully, in addition to the input matching, before the clause is selected. The commit operator is used in Concurrent Prolog, Parlog and GHC for their control and nondeterminism without side-effect. It distinguishes the gurard part and the body part of a clause. It has a similar effect as the cut operator of Prolog, but has a cleaner semantics due to its symmetry, much the same way as the guarded command has a cleaner semantics than the conventional if-then-else construct. Besides the commit operator, Concurrent Prolog adds a new syntactic construct ' ? ', an annotation denoting a read-only variable for the synchronization of processes. The annotation on read-only variables would be used to specify communication constraints.

In Parlog, the synchronization between the processes is performed by the mode statement. The mode statement determines the communication constraints on Parlog processes by specifying which variables should be used for input variables, and which variables should be used for output variables. The use of mode statement does have significant advantages with respect to efficiency in that the safety check of Parlog is done during the

compilation statically, and Parlog does not require the multiple environment in which there may be distinct to an identical variable an needed in Concurrent Prolog. In GHC, there is no explicit syntactic construct for the synchronization: it is done by two general rules, rule of commitment and rule of suspension[17]. Since GHC has no explicit syntactic construct for synchronization, it requires a run time safety check whenever a variable is to be bound and this implementation is expensive. For the deatailed synchronization differences between these languages, see Chung and et al[3), 6)]

3.3 Epilog

Epilog[18)] is an extension of Prolog. Its purpose is the construction of a multiprocessor environment and an abstract parallel architecture based on the dataflow model of Prolog. Epilog uses some constructs, such as thresholds and annotations on variables, based on dataflow model to control the available parallelism and thereby limit the braching factor. However, Epilog is not a generic parallel logic programming language. It was designed to execute Prolog programs for the realization of dataflow model and multiple processors of Prolog programs. Epilog is not concerned with solving the concurrent nondeterministic problems: instead it is concerned with the parallel execution of Prolog programs. The apparent difference between Epilog and the parallel logic languages is in the design methodology. The motivation of Epilog is different from that of the parallel logic programming languages based on CCN and efficient implementation of stream AND-parallelism.

3.4 Conery's AND/OR process model

Conery suggested an AND/OR Process Model[4)], an abstract parallel model of logic program. It provides a framework for execution of logic programs where an interpreter solves a goal by dividing it into independent pieces for solution by other interpreters.

While the parallel logic programming languages described in the above section are based on the CCN to implement the stream AND-parallelism efficiently, the parallelism exploited by Conery's AND/OR Model is oriented towards a more explorative style of nondeterminism, replacing the backtracking method of sequential system by a semi-intelligent backward execution algorithm. The explorative nondeterminism was first developed by Pollard[11)] in 1981. In the Conery's algorithm, AND process style is designed to work with in conjunction with OR process style of OR-parallelism.

Conery's system includes algorithm to detect dependencies between calls as well as heuristics to select the optimal orderings for the sequential evaluation of dependent calls for constructing the nondeterminism. This run time detection of dependent calls might be effective in exploiting potential parallelism and restraining useless nondeterminism, because variables shared between calls in the source program may actually be instantiated by the time the call is invoked, rendering the calls independent. However, this complex algorithms involved can constitute a computational overhead.

Stream AND-parallelism takes advantages of partial bindings allowed by logical variables using the feature for object oriented programming and the other elegant features, but with the expense of exploratory nondeterminism. Conery's AND/OR Process Model is opposite: it sacrifices parallelism when partial bindings are present but is able to exploit explorative style of nondeterminism[4), 6)].

4. Improved control strategy

4.1 Improved control strategy

To restrain the unnecessary parallelism and nondeterminism in the guard part and body part of a clause, we add the following two constructs based on the dataflow model for the suggested CS, in addition to existing construct of commit operator.

Our aims with the improved CS for an effective parallel logic programming systems are as follows:

(i) Control of execution of parallel logic programs by specifying a partial order on the execution of logic programs which are based on CCN and efficient implementation of stream AND-parallelism. It is desirable that the specification is natural in the sense that sequencing constraints in the guard part, body part and clauses are expressed explicityly.

(ii) Automatic detection of concurrency in logic programs, Here, the parallel logic programming system should exploit the intrinsic parallelism.

To achieve the above aims, we represent the sequencing constraints and inherited parallelism of logic programs which are based on CCN and efficient implementtion of AND-parallelism by using the notion of a control group defined as below:

(a) A control group is either a P_group or an S_group. An element of a group is called a control unit. Here, P_group represents the control group which will be executed in parallel, and S_group represents the control group which will be executed sequentially. i.e. every pair of the two control units in S_group has partial order between them, and there is no partial order among the control units in P_group.

(b) A P_group and an S_group is a set of control units tht are either atoms or control groups.

To avoid confusions, we use parenthesis {} and brackets 〈〉 to denote the parallel and sequential execution of a P-group and an S-group, respectively.

4.2 Syntax

Since we introduce essential constructs for the above purposes, the legal syntax allowed in the suggested parallel logic systems is different from the syntax of the pure logic or conventional parallel logic systems.

```
〈clause〉  ::= 〈atomic_formula〉 '.'  |
              〈atomic_formula〉 ':-' 〈body_part〉 '.'  |
              〈atomic_formula〉 ':-' 〈guard_part〉 ' | ' '.'  |
              〈atomic_formula〉 ':-' 〈guard_part〉 ' | ' 〈body_part〉 '.'

〈guard_part〉  ::= 〈literal〉 (' || '〈literal〉)*
〈body_part〉  ::= 〈literal〉 (' || '〈literal〉)*

〈literal〉  ::= 〈atomic_formula〉  |
               〈P_group〉          |
               〈S_group〉

〈S_group〉  ::= '<' 〈literal〉 '→' 〈literal〉 '>'

〈P_group〉  ::= '{' 〈literal〉 (,〈literal〉)* '}'
```

For each clause there is at most one commit operator. In a guarded clause there may be any number of CAND or COR operations for the sequencing of literals or clauses. The legal clauses are those which consist of these constructs, atomic formulas, predicates, variables, functions and constants, etc.

Now we define the legal syntax of the suggested parallel logic programming systems using the notion of BNF notations as above.

The atomic formuals are constructed from predicate identifiers and terms in the usual fashion. Note that we can omit the symbols for '{}' (bracket for P_group) or '〈〉' (parenthesis for S_group) if there is no ambiguity.

4.3 Semantics

The CS suggested in this paper can represent a partial order of literals in the guarded clauses and the clauses, with the CAND ('→'), COR ('||') control constructs, in addition to the co mmit operator. Influenced by the elegant adoption of Dijkstra's formalism[5] in Hoare's model[7], the notion of nondeterministic commitment to a single clause is used in the parallel logic programming systems. An important consequence of adopting such a nondeterministic commitment rule is the 'single assignment property', instead of the 'generate and test'(with backtracking) paradigm of a sequential logic programming programs. The commit operator is used for this purpose.

In order to restrain the unnecessary parallelism and nondeterminism, we adopt the following two constructs based on the dataflow model[18] for the suggested CS, in addition to existing construct of commit operator of CCN. With these constructs we can represent a partial order on the execution of literals in the guard, body part, and clauses of a logic programs.

Conditional And(CAND)

The CAND operator '→', is used to control unnecessary AND-parallelism in the guard part and body part by restraining a left-to-right sequence. The literal on the left of CAND construct will be executed first: if its execution terminates successfully then the literals to the right of CAND will be executed next. If the left hand side of CAND terminates with failure, then the result of failure is reported immediately. This operator is designed to represent a partial order among literals in the guard part or body part of a clause. We can specify the order of control units in S_group by CAND.

Even if PRISM[9] is a logic programming language that provides the user with control facilities to specify a partial order on the execution of literals and clauses in a program, it only allows CAND for literal sequencing. There is neither commit operator nor COR in PRISM.

Conditional (COR)

This construct, '||', is used to control the unnecessary OR-parallelism by reimposing an ordering on the set of literals in a clause. Since in parallel logic programming programs, the collection of clauses for a relation is an unordered set (unlike the Prolog's ordered list of clauses), this new construct COR can be used to reimpose an ordering on the set of clauses in our parallel logic programming systems.

Here, one set of literals will be evaluated before another set. However, unlike CAND, the right hand side of COR will be executed only if its left hand side terminates with failure: if the left hand side of COR succeeds, then its result of success is immediately reported to the parent. This operator is designed for the clause sequencing. For example, the meaning of clause of p:-q || r.

If it fails then the subsequent literal r will be executed. whereas if q succeeds then execution halts and the result of success is reported to its parent.

With COR, we can further control the restricted OR-parallelism. Without COR, we cannot control the restricted OR-parallelism as desired. The conventional CS with commit operator only provides a limited form of OR-parallelism, called the restricted OR-parallelism. Note that if we do not use the commit operator, we assume the full OR-parallelism. If we only use the commit operator, we assume the restricted OR-parallelism. With COR, we can obtain completely flexible OR-parallelism.

In the control of OR-parallelism, there is analogue between COR and cut operator of Prolog. But COR has cleaner semantics than cut operator and is in fact very different from cut. Fot the detailed explanation on their difference, uniqueness, and for the comparison with other existing parallel logic languagues, see Chung[2)3)]. In addition to the usage to restrain the useless AND-parallelism and OR-parallelism, the syntactic constructs allowed in the improved system give many desirable properties such sufficient guard property, deadlock free property, naturalness, etc.

5. Alternating Turing machines and logic programs

5.1 Alternating Turing machines

As seen in the next subsection 5.2, the complexity of computations of Turing machine can be used as the complexity of derivations of logic programs. For that purpose, we use an extended Turing machine, called the alternating Turing machine(ATM)[1)].

Definition: An Alternating Turing machine(ATM) is an eleven tuple $M = \langle Q, \Sigma, \Gamma, \delta, *, q_0, U, E, A, R, N \rangle$,

where k is the number of working tapes

Q is a set of finite states

Σ is a set of finite input symbols

Γ is a set of finite tape symbols, including blank symbol(B)

δ is transition relation

$*$ is an endmarker

B is blank symbol

q_0 is the initial state of M such that $q_0 \in Q$

$U \subset Q$ is a set of universal states

$E \subset Q$ is a set of existential states

$A \subset Q$ is a set of accepting states

$R \subset Q$ is a set of rejecting states

$N \subset Q$ is a set of negating states

Here, U, E, A, R, N are the members of the partition of Q.

Note that the language accepted by an ATM defined as above can be also accepted by another ATM M' whose states do not include the negating states, and these two ATMs have the same computation power[1]. According to Chandra[1], these two kinds of machiness M and M' can be regarded interchangeably. Therefore, if needed, we assume the ATMs whose states do not include the negating states. M has a read-only input tape with endmarker * and k working tapes, which are set initially empty.

The operational behaviour of ATM M is described as below: An existential state a spawns a set of states by its applicable transition relation, and accepts if at least one of them accepts. i.e. an existential state a generates several possible next states $b_1, b_2, \cdots, b_n$ in one step, and a leads to acceptance if there exists a successor b_i $(1 \leq i \leq n)$ which leads to acceptance.

A universal state a spawns a set of possible next states by its transition relation, and accepts if all of them accept. i.e. a universal state a generates a set of states $b_1, b_2, \cdots, b_n$ in one step, and a leads to acceptance if all successors $b_1, b_2, \cdots, b_n$ lead to acceptance.

Definition: A *step* of M consists of reading a symbol from each $k+1$ tapes (one read-only input tape, k working tapes), writing a symbol on each of the k working tapes, then moving each of the heads left/right one square according to the transition relation δ.

Definition: A *configuration* C_M of M is defined by the set $C_M = \{\sigma \mid \sigma \in (Q \times \Sigma^* \times ((\Gamma - \{B\})^*)^k\}$, where each $\sigma = \langle q, w, \gamma \rangle \in C_M$ represents the state of M, the input, the nonblank contents of the k working tapes, and the head positions of read-only input tape and k working tapes. If $q \in U$ (E, A, R), then σ is called *universal configuration* (or *existential*, *accepting*, *rejecting configuration*, respectively).

The initial configuration of M on input w is given by $\sigma_0(w) = \langle q_0, w, \varepsilon^k \rangle$, where ε denotes the empty string. Note that each configuration represents an instantaneous description(ID) of M at some point during a computation.

A *computation path* $a_1, a_2, \cdots$, (possibly infinite) is a sequence of configurations of M where a_{i+1} is a successor of a_i. A *computation tree* T of M is a directed tree, where each node of T is a configuration of M with the property that each path of T is a computation path of M. A computation tree T accepts an input string w if it is a finite tree and its root is associated with the initial configuration σ_0.

Definition: A *characteristic function* $l_C: C_M \rightarrow \{true, false\}$ is a function which labels a configuration a with *true* if a leads to acceptance or *false* if a leads to rejection. i. e. M accepts (rejects) the input string w if and only if its initial configuration is ever labeled with *true*(or *false*, respectively).

Therer is a recursive procedure for doing this:

$$lc(\alpha) = \begin{cases} \wedge \ \vdash^* - lc(\beta) \text{ if } a \text{ is universal configuration and } \alpha \vdash^* - \beta \\ \vee \ \vdash^* - lc(\beta) \text{ if } a \text{ is existential configuration and } \alpha \vdash^* - \beta \\ true \text{ if } \alpha \text{ is accepting} \\ false \text{ if } \alpha \text{ is rejecting} \end{cases}$$

Therefore, M accepts (rejects) the input string W if and only if $lc(\sigma_0(w)) = true$ (or *false*, respectively). M halts on w if and only if it either accepts w or rejects w, and $L(M) = \{w \in \Sigma^* \mid lc(\sigma_0(w)) = true\}$

Definition: M accepts an input string in time t if $lc(\sigma_0(w)) = true$, where $C = \{\alpha \mid \sigma_0(w) \vdash^* \alpha$ in t or fewer stepts$\}$

Also M accepts w in space s if $l_D(\alpha_0(w)) = true$, where $D = \{\alpha \mid space(\alpha) \leq s\}$ and $space(\alpha)$ is the sum of nonblank contents of the k working tapes in configuration α. Here, s, t are some integer numbers.

We say that M accepts w in time T(n) (or space S(n)), provided that for any $w \in L(M)$, M accepts w in time at most T($|w|$) (or space S($|w|$), respectively).

For the relationship of time and space complexities berween ATM, deterministic Turing machine and nondeterministic Turing machine, see Chanadra and et al[1),3),12)].

ATMs accept all recursive enumerable sets because any nondeterministic Turing machine is a particular case of an ATM, consisting only existential states. For converse, for any given ATM M and an input string W, enumerate finite subsets C of C_M, construct the characteristic l_C. accept w if we get $lc(\sigma_0(w)) = true$. If such a subset C exists then M accepts w, and if M accepts w then such a subset C exists.

5.2 Relationship of logic programs and alternating Turing machines

The reason why ATM is chosen as the tools of complexity analysis of logic program is that ATM is a surprisingly good model of parallel complexity and is a generic model of parallel computations. Moreover, ATM provides a basis for evaluating the parallel complexities of logic programs.

There is a close relationship between an abstract interpreter of logic program P and the execution mechanism of ATM M. We can associate the computation tree of M with the AND/OR computation model of P. The computation of ATM does capture the procedural behaviours for the derivations of parallel logic programs, and the computation tree of ATM corresponds to the AND/OR computation tree of logic programs. As seen in the operational behaviour of ATM in the subsection 5.1, an universal state (existential state) behaves as an AND node(OR node, respectively) of an AND/OR tree of logic program.

The existential state of M corresponds to the nondeterministic choice of a clause whose head unifies with a goal, whereas the universal state of M represents the simultaneous satisfaction of the goals in the body of a clause. M is in a rejecting state when there is no applicable next transition if and only if no clause is in P whose head unifies with the goal. M is in an accepting state when the input tape is empty if and only if during the exection of P, current goal is the empty clause obtained from the goal reduction.

In fact the computations of M with its computation tree is the exactly the same as the proof procedure of P with its AND/OR computation model. An universal configuration of M can be thought as an AND process of AND/OR computation model of P. It leads to acceptance if all of its successors lead to acceptance. An existential configurtion can be thought as an OR process of AND/OR computation model of P. It leads to acceptance if any of its successors leads to acceptance. From these apparent similarities between the execution of logic program and the computation of ATM, the complexity of derivations for parallel logic program is closely related to the complexity of computations of ATM.

6. Complexity analysis

We will perform the complexity analysis for the logic programs using a general, theoretical framework for the computational complexity of logic programs.

The reason is that it is extremely difficult to come up with a precise, meaningful comparison when we do the performance evaluation for the parallel logic programs. The evaluations for parallel logic programs on a physical computing machines would be dependent on particular algorithms, systems and hardware considerations. Moreover it is not natural and reasonable to do the evalutations for parallel nondeterministic logic programs with the execution on a sequential von Neumann computing machines.

In this paper, we ignore the overheads dependent on particular algorithms, systems, and hardware considerations. In order to estimate the improvement in efficiency of the suggested system, we measure the ratio of the potential gain of suggested parallel logic programming systems to the gain of the conventional systems. This measure, called the potential parallel factor(PPF), is the maximum possible gain of the parallel logic systems over the sequential one. Since this measure provides us with the quantitative description of the effect of parallelism, it can be used to compare the complexity of different logic programming systems. For this measure, we use the ATM, which is highly abstract computation model and describes the parallel computational behaviour of logic programs, as described in the above section 5.

In section 5, we observe that the derivations of logic programs and the computations of ATMs are closely related. These results reveal similarities between logic programs and ATMs. From these similarites, we relate the complexity measures of logic programs with those of ATMs.

In addition to several desirable properties of logic programs (such as sufficient guard property, deadlock-free, etc). Chung[2,3] shows the relationship between the parallel complexity (NC-class) and the complexity of logic programs. It also discusses the relationship between the ATM's low level primitive computations (such as movements, iterations etc) and the proof procedure (such as instantiation, query size, number of variables in a clause, ect) of logic program. It shows how ATM's necessary bookkeeping is performed in its working tapes. For the properties or articles not mentioned in this paper, see Chung[3].

Since the computations of ATMs describe the derivations of logic programs and due to the close relationship betwen logic programs and ATMs, the complexity of computations of ATMs can be used to bound complexity measure of derivations of logic programs[3),14),18)].

Moreover, as stated in Chandra[1], it is desirable that when the problem of interest involves alternating quantifiers, we should classify the problem with complexity of ATMs and then translate the classification to one in terms of deterministic machines.

Therefore, we first associate different classes of logic programs with corresponding classes of ATMs, then we classify these problems by translating the ATMs to one in terms of deterministic machines.

theorem 6.1: Let A_1, A_2 and A_3 be the number of alternations of ATMs M_1, M_2 and M_3 respectively, where M_1, M_2 and M_3 are the alternating Turing machines associated with the corresponding logic programs L_1, L_2 and L_3 respectively. Here, L_1 is a pure logic program without any syntactic constructs for the efficiency, i. e. its CS is the

blind exhaustive search, and L_2 is the program with the CS of CCN only, and L_3 is the program with the suggested CS.

Furthermore, let S_1, S_2, and S_3 be the space complexity of M_1, M_2, and M_3 respectively for the input w with the length n. Then $A_1(n), \geq A_2(n), \geq A_3(n)$ and $S_1(n), \geq S_2(n), \geq S_3(n)$.

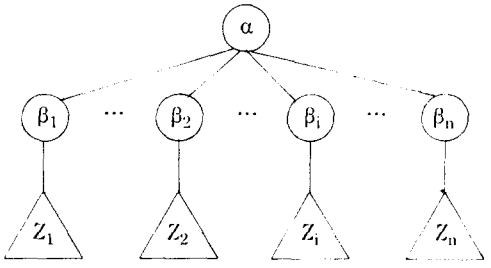


Fig 6.1 Computation tree for the exhaustive search

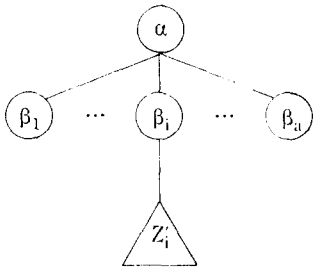


Fig 6.2 Computation tree for the previous control strategy

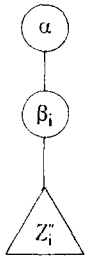


Fig 6.3 Computation tree for the suggested control strategy

proof: Let T_1, T_2, and T_3 be the computation tree of the ATMs M_1, M_2, and M_3 respectively. Since T_1 is the computation tree obtained from the exhaustive breadth first search, every possible movement between the configurations of M_1 appears in T_1, and every possible consistent solution of L_1, would appear in T_1: in this sense T_1 gives the complete solution set even if it causes the combinatorial problem, and T_1 is the fully complete AND/OR tree. T_1 is shown in figure 6.1.

T_2 is a subtree of T_1, since for the offsprings of an existential configuration node of T_1, only one path will be followed: once a clause of a procedure is committed, all the other existential configuration nodes with same parent would not be followed.

In T_2, we sacrifice the completeness of T_1 for its efficiency. In figure 6.2, one successor β_i $(1 \leq i \leq n)$ from the set of all successors $\beta_1, \cdots, \beta_n$ of an existential configuration node α is selected: all other successors $\beta_1, \cdots, \beta_{i-1}, \cdots, \beta_n$ would not be followed.

Moreover, we can cut off the unnecessary nodes by partially sequentializing the firing of literals of a clause in the universal configurations with the commit operator: if the guard part of a clause fails, the literals in its body part would not be evaluated, i. e. these offsprins of the universal configuration corresponding to the body part of the clause would not appear in T_2.

As shown in figure 6.3, suppose that the successor β_i $(1 \leq i \leq n)$ is selected from the successors $\beta_1, \cdots, \beta_n$ with the CS of CCN only, where α is the root, existential node of T_2. Furthermore let Z_i' be a subtree in which only one successor is selected from all successors of an existential configuration node in Z_i. Then

clearly Z_i' is a subtree of Z_i, hence T_2 is a subtree of T_1. i. e. $A_1(n) \geq A_2(n)$.

With the suggested idea, we could have only successor from an existential configuration node in T_1 by restraining OR-parallelism, and we can reduce the offsprings in its universal configuration nodes by restraining AND-parallelism. With the same method we can cut off unnecessary offsprings for subsequent nodes. T_3 is a computation tree whose figure is given in figure 6.3. Clearly Z_i'' is a subtree of Z_i', hence T_3 is a subtree of T_2. i. e. $A_2(n) \geq A_3(n)$. Therefore $A_1(n) \geq A_2(n) \geq A_3(n)$.

Assume that the space complexity $S_1(n)$ of M_1 is decided by a configuration node k of T_1. If k is in T_2 then $S_1(n) = S_2(n)$. If k is not in T_2, clearly $S_2(n)$ which is the space complexity of T_2 is less than S_1. (If $S_2(n) \geq S_1(n)$, then $S_2(n)$ should be a complexity of M_1, a contradiction). With the same argument, we get $S_2(n) \geq S_3(n)$. Therefore $S_1(n) \geq S_2(n) \geq S_3(n)$.

theorem 6.2: Let M_1, M_2 and M_3 be the ATMs described in the above theorem 6.1. Then M_1, M_2 and M_3 can be simulated by the deterministic Turing machines(DTMs) N_1, N_2 and N_3 with the space complexity DSPACE($S_1^2(n)$), DSPACE($S_2^2(n)$) and DSPACE($S_3(n)$), respectively. (Note that we obtain $S_1(n) \geq S_2(n) \geq S_3(n)$ in the theorem 6.1)

proof: Let's explain about the recursive procedure MAIN in the algorithm 6.1 to decide whether some configuration α leads to an acceping configuration or to a rejecting configuration where M is an ATM with space complexity $S(n)$ and there exist some useless nondeterminism in the configurations of the computation tree of M.

M takes some configuration a as its single input parameter with space(a) $\leq S(n)$ and returns the value of *true* (or *false*) if a leads eventually to an accepting configuration (or rejecting configuration respectivly). The initial input parameter a is the initial configuration of the form $\alpha_0 = \langle q_0, x, \varepsilon^k \rangle$ where q_0 is the starting state of state M, x is the input string to be scanned, ε^k denotes the initial empty k working tapes.

Let's explain how MAIN can be implemented on DTM N that uses a tape as a stack to keep local storage over a call.

When a is an acceping (or rejecting configuration), MAIN terminates immediately by returning the value of *true*(or *false* respectively).

Let $C = \{\Upsilon \mid \text{space}(\Upsilon) \leq S(n)\}$, i. e. C is the set of all configuration that can use at most $S(n)$ as the sum of the lengths of the nonblank working tapes contents. When α is an existential configuration, and $\alpha \in C$, DTM N should test each configuration β whether some β eventually leads to an accepting configuration or to rejecting configuration where $\alpha \vdash^* \beta$, $l_c(\alpha) = \vee l_c(\beta)$ and the disjunction is taken over the set $\{\beta \mid \alpha \vdash^* \beta$ and all configuration along the path in $\alpha \vdash^* \beta$ are existential configuration and elements of C, and β's are not existential configuration.$\}$

In order to do this, as described in the below algorithm 6.1, MAIN writes down each nonexistential β defined above successively until N finds out some β such that there is a path from α to β, and β would eventually lead to an accepting configuration.

If there is no such β defined as above, which leads to an accepting configuration, N terminates MAIN with returning the value of *false*, and if there is some β defined as above which leads to an accepting configuration eventually, N terminates with returning the value *true*.

Similarly, when α is a univrsal configuration and an element of C, the action of MAIN can be described as following. Since $l_c(\alpha) = \wedge l_c(\beta)$ and

the configuration is taken over the set $\{\beta \mid \alpha \vdash^* \beta$ through only universal configuration in C and β is not universal configuration$\}$. N should test for ech β whether every β eventually leads to an accepting configuration and there exist a path from α to β.

In order to do this, MAIN writes down successively all non-universal configuration β defined as above with the property space$(\beta) \leq S(n)$ and calls PATH(α, β) to check whether there is a path from α to β through only universal configuration in C. If PATH(α, β) returns *true*, it calls MAIN(β) recursively to decide whether configuration β eventually leads to an accepting configuration or not.

As shown in algorithm 6.1, recursive procedure MAIN with the single input parameter α calls procedure PATH to decide whether there is a path from α to β. If α is a universal (existential configuration), it decides whether there is a path from α to β where all configuration in the path are universal (existential respectively) and should be elements of C.

Note that the procedure PATH may take its input parametrs α, β with different association on these two configuration i. e. β may be universal (existential) configuration even if α is an existentual (universal respectively) configuration.

Even if there is some nondeterminism in the configuration of the computation tree of ATM, the decision whether $\alpha \vdash^* \beta$ will take space $S(n)$ just by guessing the correct path nondeterministically.

If there is no controlling on this nondeterministic gussing, DTM $N_i(i = 1, 2)$ will take $S_i^2(n)$ global space for this decision due to the nondeterministic guessing of the correct path among the alternative paths. The squaring of space complexity $S_i^2(n)$ from $S_i(n)$ is due to Savitch[12]. i.e. N_1(or N_2) would take $S_1^2(n)$ (or $S_2^2(n)$ respectively) global space storage for the deterministic decision form the nondeterministic guessing of the correct path. When DTM N does not have to guess nondeterministically the correct path, i.e. when there is an explicit mechanism to control this nondeterministic guessing and to specify the correct path, there would be just constant factor in the space complexity. Moreover it can be simulated by another DTM with a single tape and $S(n)$ space complexity space[8].

Since space$(\alpha) \leq S(n)$, the sum of the lengths of the nonblank working tape contents of α is bounded by $S(n)$, therefore at most $S(n)$ local storage would be needed for each call to MAIN. Moreover the depth of recursion is bounded by $A(n)$. since each recursive call to MAIN corresponds to another alternation, and when α is the parameter to a particular instantiation of MAIN, then only α need be saved over recursive calls. Therefore DTM M_i will use at most $A_i(n) * S_i(n)$ $(i = 1, 2, 3)$ space for the local storage and $S_i^2(n)$ $(i = 1, 2)$ and $S_3(n)$ global space.

```
procedure MAIN(α)
begin
  case
    α is an existential configuration: do
      write down each nonexistential β ∈C
      successfully;
      if PATH(α, β) through only existential
      configurations
        then MAIN(β) else try on another β
        defined above;

α is a universal configuration: do
  write down all nonuniversal β ∈C;
  if PATH(α, β) through only univrsal
  configuration
    then MAIN(β) else return false;

α is an accepting configuration: return true;
```

α is an rejecting configuration: return false;

end MAIN

PROCEDURE PATH(α, β)
begin
case
α is a universal configuration: do
decide whether there is a path from α to β such that all configuration appearing on the path
(with the poissible exception of β) are universal configurations and an element of *C*.

β is an existential configuration: do
decide whether there is a path from α to β such that all configurations appearing on the path
(with the possible exception of β) are existential configurations and an element of *C*.

end PATH

algorithm 6.1

7. Conclusion

This papr addresses the parallel logic programming systems and their complexity analysis. The logic programming systems play an important role in the artificial intelligence and deductive database. They are useful in a world of mutually mistrusted but cooperating agents. In this sense, logic programming systems provide the security, encryption and authentication of multi-user systems. The logic programming systems can be utilized as a secure systems in the doimain of operating systems, database and communication.

In this paper we suggest an improved control strategy for parallel logic programming systems. We present its formal syntax and semantics. The suggested idea is a combination of committed choice nondeterminism and the dataflow model. The commit operator is used for the committed choice nondeterminism. It distinguishes the guard part and the body part of a clause. The constructs based on the dataflow model are used to control the unnecessary AND-parallelism and OR-parallelism by specifying the partial order of literals in a guarded clause or the clauses for a relation. The suggested idea reduces the search space by pruning the unnecessary offsprings. With the suggested control strategy, we can exploit the maximum useful parallelism by obtaing optimal granularity.

In order to justify that the suggested idea is reasonable, we need the potential parallel factor for comparing the performance evaluations. To do so, parallel logic programs are analyzed by means of alternating Turing machines which constitute the parallel computation models that capture the essence of the procedural interpretation of parallel logic programs.

The complexity of ATMs does represent a useful measure of the potential parallel factor. To show this we associate the alternating Turing machines with the parallel logic programs. The computation trees of ATMs exactly represent the AND/OR computation trees of parallel logic programs.

It is suggested that when the problems of interest intrinsically involve alternaing quantifiers, we classify the prblems in terms of the complexity of ATMs and then analyze performance by the complexity of deterministic Turing machines.

We show that a natural reduction of the alternating Turing machines for our suggested idea yields a deterministic Turing machine whose complexity is linearly related to that of the ATM. However such a reduction for the conventional

committed choice nondeterminism only has complexity tht is polynomially related to that of the ATM.

References

1. A. K. Chandra, D. C. Kozen and L. J. Stockmeyer: Alternation, JACM, Vol. 28 No. 1, pp. 114-133, Jan. 1981

2. I. J. Chung: Improved control strategy for parallel logic programming, Proc. of IEEE interantional conference on tools for AI, pp. 702-708, 1989.

3. I. J. Chung: Improved control strategy for parallel logic programming, Ph. D. dissertation, Univ. of lowa 1989.

4. J. S. Conery: Parallel execution of logic programs, Kluwer Academic Pub. 1987.

5. E. W. Dijkstra: Guarded commands, nondeterminacy and formal derivations of programs. CACM Vol. 18 No. pp, 453-457, 1957.

6. S. Gregory: Parlog, Addison Wesley 1987.

7. C. A. R. Hoare: Communicating sequential processes, CACM Vol. 21 No. 8, pp. 666-677, 1978.

8. J. E. Hopcroft and J. D. Ullman: Introduction to automata therory, languages and computation, Addison Wesley 1979.

9. S. Kasif: Control and data driven execution of logic programs: a comparison, Internationl J. of parallel programming Vol. 15, No. 1, pp. 73-99, 1986.

10. R. A. Kowalski: Algorithm = logic + control, CACM Vol. 22, No. 7, pp. 424-436, 1976.

11. G. H. Pollard: Parallel execution of Horn clause programs, Ph. D. dissertation, Dept. of computing, Imperial College, 1981.

12. W. Savitch: Relationship between nondetrministic and deterministic tape complexities, J. of computer sytems and sciences, Vol. 4, No. 2, pp. 177-192, 1970.

13. E. Shapiro: Subset of Concurrent Prolog and its interpreter, Techncal report TR-003 Feb, 1983.

14. E. Shapiro: Alternation and the computation complexity of logic programs, J. of logic programming No. 1, 1984.

15. E. Shapiro: Concurrent Prolog, Collected papers, Vol. 1, and Vol. 2, MIT Press, 1987.

16. P. Stepanek: Logic programs and alternation, third international conference on logic programming, LNCS Vol. 225, pp. 99-106.

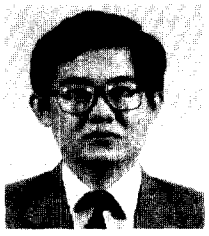
17. K. Ueda: Guarded Horn Clauses, LNCS Vol. 221, Springer Verlag pp. 168-179, 1985.

18. M. J. Wise: Prolog multiprocessors, Prentice Hall of Austrailia 1986.

□ 著者紹介

정 인 정(종신회원)

1978년 2월 : 서울대학교 자연과학대학 계산통계학과 졸업(이학사)
1980년 2월 : 한국과학기술원 전산학과 졸업(이학석사)
1980년 3월－1983년 2월 : 삼성전자 컴퓨터 사업부 근무
1981년 3월－1983년 2월 : 홍익대학교, 동국대학교 강사
1983년 3월－1984년 8월 : 이화여자대학교 전자계산학과 전임강사
1989년 12월 : 미국 Univ. of Iowa 대학원 졸업, 이학박사(Ph.D.)
1992년 3월－1992년 2월 : 고려대학교 자연과학대학 전산학과 조교수
1992년 3월－현재 : 고려대학교 자연과학대학 전산학과 부교수



이 홍 규

1978년 2월 : 서울대학교 공과대학 전자공학과 졸업(공학사)
1981년 2월 : 한국과학기술원 전산학과 졸업(이학석사)
1984년 8월 : 한국과학기술원 전산학과 졸업(공학박사)
1986년 8월 : 미국 미시간 대학 Manufacturing Center 연구원
1986년 9월－현재 : 한국과학기술원 조교수
1987년 8월 : 영국 Imperial College in London 교환교수
주관심분야 : 병렬처리, VLSI설계, 실시간 및 고장허용 시스템등임.