South African
Computer
Journal
Number 11
May 1994

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 11
Mei 1994

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarverenging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
Email: dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof John Shochot
University of the Witwatersrand
Private Bag 3
WITS 2050
Email: 035ebrs@witsvma.wits.ac.za

Production Editor
Dr Riël Smit
Mosaic Software (Pty) Ltd
P.O.Box 16650
Vlaeberg 8018
Email: gds@cs.uct.ac.za

Subscriptions

	Annual	Single copy
Southern Africa:	R45,00	R15,00
Elsewhere:	$45,00	$15,00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Phone: +27 (11) 315-1319 Fax: +27 (11) 315-2276

Parallel Execution Strategies for Conventional Logic Programs: A Review

P E N Lutu
Department of Computer Science, University of Transkei
cspen@unitrix.utr.ac.za

Abstract

Parallel execution strategies for Prolog programs are reviewed. Three models, AOPM, APPNet and RAP are discussed in some detail. The AOPM and the APPNet are AND-OR parallel execution models and represent the two ends of the spectrum for approaches to AND-OR parallelism. The AOPM exploits fine-grained parallelism while the APPNet exploits coarse-grained parallelism. RAP is an AND-parallel execution model and is based on fine-grained parallelism. The problem of shared variables in AND-parallelism is dealt with differently by each model. APPNet uses dependent AND-parallelism where streams of results are back-unified to eliminate inconsistent combinations of results. The AOPM and RAP use different approaches to independent AND-parallelism. Both models use intelligent backtracking to avoid the generation of inconsistent results. All three models have shown significant speed ups in the execution of Prolog programs.
Keywords: *Logic programs, parallel Prolog execution, AND-parallelism, AND-OR parallelism*
Computing Review Categories: *A1, D3.1, F4.1*

Received: October 1993

1 Introduction

Research in the area of Conventional Logic Programming has developed on two fronts. One area deals with the design of parallel and concurrent logic languages, based on Horn Clauses, for modelling parallel and concurrent processes. The other area deals with parallel execution strategies for programs written in sequential Horn clause logic languages. From a programming point of view, logic programming is a convenient paradigm for fast software development. This is due to its simple declarative semantics, relational form and non-deterministic nature.

Prolog is the most successful sequential language for Conventional Logic Programming. Unfortunately, programs written in Prolog tend to run slower than equivalent applications written in C. On the other hand, when there is an obvious way to express a problem in logic, program development is fast, and the resulting program is more likely to be correct. For this reason, Prolog is often used for rapid prototyping. It has been argued that Prolog is especially suited for running on multiprocessor architectures. If this turns out to be the case, prototypes written in Prolog may be converted directly into products without first having to be rewritten in C.

Many models for the parallel execution of Prolog programs exist. These have shown that significant speed ups can be realised when Prolog programs are executed in parallel. This article is a review of the execution methods for Prolog. Section 2 discusses the two alternative strategies for the sequential execution of Prolog programs. Sections 2 and 3 discuss the parallel execution strategies.

2 Sequential execution of prolog programs

A Prolog program comprises a set of Horn clauses and a goal statement. The Horn clauses express the information to be used to solve problems. A Horn clause has the form $b :- a_1, \ldots, a_n (n \geq 0)$. b is a literal called the head and $a_1, \ldots, a_n$ is a conjunction of literals which form the body of the clause. A literal a_i has the form $r(T_1, \ldots, T_n)$ where r is called a predicate and the T_i are terms. Terms can be distinguished into constants (first symbol a lower case letter), variables (first symbol an upper case letter) or compound terms of the form $f(T_1, \ldots, T_m)$ with f an m-ary function name and the T_i again terms. A Horn clause reads: "to solve the problem b, solve the problems a_1 and a_2 and ... and a_n. The goal statement specifies the problem to be solved. It is written $:- a_1, \ldots, a_n (n \geq 1)$, where the a_i are again literals.

Execution based on the Unification Algorithm

For purposes of program execution, each Horn clause is called a procedure, and each literal, in the goal statement, a (procedure) call or a goal. If $X_1, \ldots, X_m$ are the variables occurring in the above goal statement, then it reads: "find values $X_1, \ldots, X_m$ which solve the problem a_1 and a_2 and ... and a_n. The goal statement is the initial state of execution. It can be represented by an AND-tree (proof tree) where the goals a_i are the children of the root node. Program execution is a finite sequence of derivation steps. To perform a derivation step on a goal statement $:- a_1, \ldots a_n$, the leftmost goal, in this case, $a_1 = r(V_1, \ldots V_p)$ is selected. A procedure for r is chosen and its variables are renamed to become unique. The procedure is applicable when the goal $r(V_1, \ldots V_p)$ and the head $r(T_1, \ldots, T_p)$ of the procedure have a most general

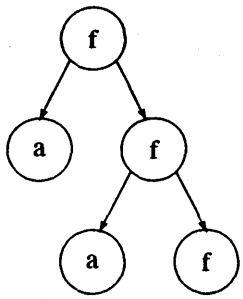

Figure 1. The Infinite Tree for the term $f(a, X)$ unified with X

unifier (m.g.u) θ which matches them. The m.g.u. θ is of the form $\theta = \{T_1/V_1, \ldots, T_p/V_p\}$. By applying the substitution θ, a new goal statement: $:- (b_1, \ldots, b_m, a_2, \ldots a_n)\theta$ is derived, where the b_j form the body of the chosen procedure. If the body of the applied procedure is empty, then the goal a_1 is resolved away. This step is the result of applying the resolution principle [18]. One derivation step corresponds to extending the proof tree by adding the child nodes $b_1, \ldots, b_n$ for the node a_1. The process of generating θ is called *unification*. If a unification failure occurs, then shallow backtracking occurs to an untried procedure. If no untried procedure exists then deep backtracking occurs to the most recent ancestor goal with untried procedures. When the goal statement is empty, the problem is solved. The set of substitutions generated during the derivation are used to construct the solution. Alternative solutions are obtained by backtracking.

The majority of Prolog interpreters are based on this approach. For efficiency purposes the interpreters use a simplified version of the unification algorithm. The simplification consists of suppressing the *occur check*, by allowing the unification of a variable with a term already containing that variable. This simplification may be unsafe, but it results in substantial gains in execution efficiency. For example, if the *occur check* is included, the concatenation of two lists requires a time proportional to the square of the length of the first list. If the *occur check* is eliminated the time becomes linear.

Execution based on infinite trees

Colmerauer [7] has proposed this alternative approach for Prolog execution[1]. A Prolog variable represents a finite tree constructed over a set F of function symbols. When a variable X is unified with a term $f(a, X)$ then X represents an infinite tree as shown in Figure 1. This way the interpreter does not need to perform the *occur check*. The tree of Figure 1 is also called a *rational tree* since it has a finite set of subtrees, namely the tree with root f and the tree with root a.

An *assignment* δ to a set of variables is defined as a finite set of ordered pairs of the form: $\delta = \{X_1 = r_1, \ldots X_n = r_n\}$ where the X_is are variables and the r_is are (possibly infinite) trees. If a given term t contains only variables from δ then δ^*, the mapping from terms to trees, denotes the tree defined by:

If $t = X_i$ then $\delta^*(X_i) = \delta(X_i)$
If $t = ft_1 \ldots t_n$ then $\delta^*(ft_1 \ldots t_n) = f\delta^*(t_1) \ldots \delta^*(t_n)$

An *equation* is an ordered pair of terms (t, s) written as $t = s$. An *assignment* δ^* is a solution of this equation iff $\delta^*(t) = \delta^*(s)$. A *system of equations* of the form: $\{X_1 = t_1, \ldots X_n = r_n\}$ is said to be in *solvable form*, and has at least one solution of the form: $\{X_1 := r_1, \ldots X_n := r_n\}$ where the X_is are variables, the t_is are terms, and the r_is are (possibly infinite) trees. For example, the system of equations:

$$\{X_1 = f(X_2, X_1), X_2 = a\}$$

is in solvable form and has a solution:

$$\delta = \{X_1 = f(a, f(a, f(\ldots))), X_2 = a\}$$

A system of equations containing an equation of the form:

$$ft_1 \ldots t_n = gs_1 \ldots s_n$$

where f and g are distinct function symbols, is said to be in *unsolvable form* and has no solution. Various transformations [7] can be performed on a system of equations to determine if it is or is not solvable. If the system is solvable, application of these transformations leads to the solution.

A Prolog program is viewed as a recursive definition of a subset A of the (possibly infinite) trees over the set F of function symbols. Each clause is interpreted as a rewriting rule[2]: $r \rightarrow r_1 \ldots r_n$; to mean: "$r$ can be rewritten by the sequence $r_1 \ldots r_n$, when $n = 0$ then r is erased". Therefore the assertions are the trees which can be erased in a finite number of steps, using the rewriting rules. The purpose of the program execution may then be stated as follows: "Given a program which is a set A of assertions, and a set of terms $\{t_0 \ldots t_n\}$ corresponding to the initial goal statement, and a set of variables $\{X_1, \ldots, X_m\}$ which appear in the set of terms, compute all tree assignments of the form: $\delta = \{X_1 := r_1, \ldots X_m := r_m\}$ such that the set of trees $\{\delta^*(t_0), \ldots, \delta^*(t_n)\}$ becomes a subset of A".

Let the state of the computation be represented by the pair (T, E) where T is the remaining sequence of terms to be erased and E is the set of equations generated since the start of the computation. The computation can then be modelled by the following three formulae:

$$(t_0 t_1 \ldots t_n, E) \quad (1)$$

$$s_0 \rightarrow s_1 \ldots s_m; \quad (2)$$

$$(s_1 \ldots s_m t_1 \ldots t_n, E \bigcup \{t_0 = s_0\} \quad (3)$$

The computation moves from the state (1) to the state (3) if the rule (2) can be used to rewrite t_0 and the system of equations: $E \cup \{t_0 = s_0\}$ has at least one solution. The initial state of the computation is represented by: $(t_0 \ldots t_n, \{\})$, where the t_is represent the initial goal statement. A solu-

[1] Colmerauer has extended this approach to PrologII and PrologIII

[2] The Marseille style has developed along slightly different lines from the more common Edinburgh style. In the Marseille style the 'if' symbol ':- ' is replaced by the arrow '→'. No punctuation is used between the terms on the RHS of a rule and each fact or rule must end with a semicolon.

tion to the initial goal statement can be provided whenever the computation reaches the state (ϵ, E'), where ϵ is the empty sequence of terms. A Prolog interpreter based on this approach has been implemented by Colmerauer [7].

Backtracking

In both schemes, alternative solutions to the initial goal statement are obtained by chronological (also known as "naive") backtracking. Whenever the computation reaches a choice point, that is, when there are more than one applicable procedure (rule) one procedure is selected and the others are tried later, on backtracking. Naive backtracking can be very inefficient as illustrated by the following example.

$$p(X, Y) :- q(X), r(Y)$$

With naive backtracking, if $r(Y)$ fails, the interpreter backtracks to $q(X)$. If X and Y are independent variables, that is, they are not aliases of each other, then backtracking to $q(X)$ is a worthless exercise since producing a new value for X cannot possibly affect the success or failure of $r(Y)$.

'Several schemes for intelligent backtracking in sequential interpreters have been proposed [2, 10], but these have been found to create unacceptable runtime overheads for sequential execution. Parallel execution schemes, however, do benefit from intelligent backtracking. Selective backtracking proposed by Pereira and Porto [17], is a more practical scheme for sequential execution, and has been implemented in a number of practical interpreters.

Semantics

The formal semantics of logic programs are defined within the context of the Herbrand Universe [21]. This is the set of all possible terms that can be formed from the constants and functions in a given program. The denotation $D(p)$ of an n-ary procedure p is a set of n-tuples of terms. There are three ways of defining D; all three methods define the same set. $D1(p)$, the operational semantics of p, is defined to be the set of all tuples $\langle t_1, \ldots, t_n \rangle$ such that $p(t_1, \ldots, t_n)$ is provable, given the clauses of the program as axioms. Definitions for the model-theoretic semantics, and the fixed point semantics may be found in [21].

3 Parallel Execution of Conventional Logic Programs

Exploitation of parallelism

The syntax of Horn clause logic programs makes the exploitation of parallelism more obvious that other programming paradigms. Two types of parallelism which can be exploited are OR-parallelism and AND-parallelism. When there is more than one procedure for solving a given goal, OR-parallelism can be exploited by having one parallel process for each procedure for the goal. If a given goal statement is a conjuction of several goals, AND-parallelism can be exploited by executing the conjucts in parallel. In addition, AND- and OR-parallelism may be combined. The parallel execution models, for Prolog, discussed in this paper do not require the programmer to modify a program in any way. For other approaches, e.g. Epilog [23] and PMS-Prolog [24] special features are added to the original Prolog so that a programmer can explicitly specify when parallelism should be exploited from a program.

OR-Parallelism

OR-parallelism speeds up the execution of highly nondeterministic problems, by generating all solutions in parallel. Ideally, an OR-parallel interpreter does not need to backtrack. If the search tree for a given problem consists of N leaf nodes then N processors are needed to implement full OR-parallelism. In practice, however, if a problem is large, it is not possible to have full parallelism. In such cases, when there are no free processors, execution degrades to sequential and backtracking must then be used.

Many OR-parallel models for Prolog have been proposed. These include: Ali's model [1], SRI [22], Delphi [6], Aurora [16] and many others. In general OR-parallel evaluation strategies use a splitting algorithm which creates new parallel processes at any point in the computation where alternatives are encountered. The AOPM and APPNet models discussed in Sections 4 are typical approaches to OR-parallelism.

AND-Parallelism

For heavily deterministic problems, no gains can be realised from OR-parallelism. AND-parallelism on the other hand can be used to speed up the computation of such problems. The major problem with AND parallelism, is how to deal with conflicting bindings of variables. The automatic scheduling of a process for every literal in the body of a clause leads to binding conflicts if the literals involved have variables in common. This can happen even when the literals involved appear not to share any variables at all as illustrated by the following Prolog example:

clause: $band(X, Y)$:- $singer(X), guitarist(Y)$
query: :- $band(X, X)$

At unification time, X and Y become the same variable, therefore $singer(X)$ and $guitarist(Y)$ cannot be executed in parallel. Several approaches are used to deal with these binding conflicts. In models which use *independent AND-parallelism*, these conflicts are detected either at run time [9], or at compile time [3, 13, 14]. In models which use *dependent AND-parallelism* [25], *back-unification* is used at run time to eliminate inconsistencies which would arise due to these conflicts. For the parallel logic languages, not discussed in this paper, the programmer is required to mark literals as either *consumers* or *producers* of values for the shared variables. Among these are: Concurrent Prolog [19], Parlog [4] and GHC [20]. The type of AND-parallelism used by the interpreters of these languages is known as *stream AND-parallelism*.

Three models which implement AND-parallelism for conventional Prolog are reviewed in the next section. The models have been selected for the way they either perform *back-unification* to eliminate conflicting bindings or the

way they perform *backtracking* to avoid conflicting bindings.

4 Parallel Execution Models

The AND-OR process model (AOPM)

The AOPM [9] implements OR-parallelism by using one parallel AND process for each alternative procedure whose head unifies with a given literal in a goal statement. AND-parallelism is realised by having one parallel OR process for each literal in the body of a procedure.

The OR processes

An OR process is an independent interpreter created to solve a goal statement consisting of exactly one literal. An OR process created to solve a literal $p(X_1 \ldots X_n)$ is expected to return every tuple in the set $D_1(p)$. It does this by constructing a proof of the goal statement consisting of only this literal. The OR process creates an AND process for every procedure with a head that unifies with $p(X_1 \ldots X_n)$. When an OR process is first created, it assumes that its parent AND process is waiting for an answer. The first result constructed by the OR process is sent via a *success* message to the parent. After this, however, the OR process saves the answers, until a *redo* message is received from the parent. The OR process acts as a message centre, deciding when to transmit results and when to store them.

The AND processes

An AND process is an independent interpreter which solves the body of a clause. It does this by creating one OR process for each literal in the body. When literals share a variable, only one, called the *generator* for the variable is allowed to bind it. All steps in the execution of the generator must complete before the consumers can start. This form of parallelism is called *independent AND-parallelism*. The implementation of AND parallelism has three major components:

An Ordering Algorithm This automatically decides on the solution order of the literals. The ordering algorithm establishes a linear ordering for the literals and dynamically constructs a *dataflow graph* for each AND process. The dataflow graph is a representation of the producer/consumer relationships for the variables of the AND process.

The Forward Execution Component This creates the descendant OR processes, handles *success* messages from these processes and determines which literals can be solved as a result. A *success* message from literal L has the form *success*$(L\theta)$ where $L\theta$ is a copy of L with some variables possibly bound.

The Backward Execution Component This handles *fail* and *redo* messages, and decides which literal(s) must be re-solved before continuing forward execution. The information in the dataflow graph for each AND process is used to implement semi-intelligent backtracking.

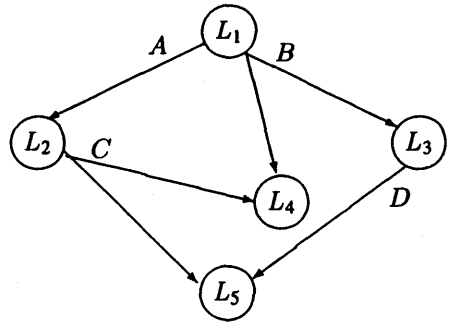

Figure 2. Dataflow Graph for the colour3(A, B, C, D) call

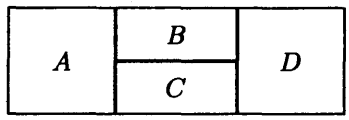

Figure 3. Regions to colour for the colour3(A, B, C, D) call

Example 1

The following augmented clause is part of a Prolog program for the three colour map colouring problem. The label against each literal is the literal's label in the dataflow graph of Figure 2.

$$
\begin{array}{lll}
colour3(A, B, C, D) & :- & \\
(L_1) & & next(A, B), \\
(L_2) & & next(A, C), \\
(L_4) & & next(B, C), \\
(L_3) & & next(B, D), \\
(L_5) & & next(C, D).
\end{array}
$$

The regions to colour are shown in Figure 3. Given the call: $:- colour3(A, B, C, D)$, the ordering algorithm would construct the dataflow graph of Figure 3 for the resulting AND process. The dataflow graph indicates that literal L_1 is the producer of values for variables A and B, literal L_2 is the producer of C and literal L_3 is the producer of D. OR processes for literals L_2 and L_3 may start executing in parallel once the OR process for literal L_1 has produced values for A and B. The OR processes for L_4 and L_5 may asynchronously start executing as soon as values for C and D become available.

Semi-Intelligent Backtracking

When several parallel OR processes exit for the body of a clause, it is not possible to talk about chronological backtracking for the whole clause. The scheme used in the AOPM is called *semi-intelligent backtracking*, and operates as follows. If the OR process for a literal fails, it means that one of the generators for its variables must have produced an unusable value. Backtracking should then be done to each of the generators in turn starting with the one which appears latest in the linear ordering. For example, if the OR process for literal L_5 fails, then the backtrack path is (L_3, L_2, L_1). However, if L_4 fails, the backtrack path is (L_2, L_3, L_1), even though L_3 does not generate any

values for L_4. The reason for including L_3 in the backtrack path for L_4 is that when L_2 fails, the *backward execution component* cannot tell whether L_2 was being executed as a result of L_4's failure and not as a result of L_5's failure. The scheme is called semi-intelligent because it is not fully intelligent, otherwise L_3 would not be retried when L_4 failed.

Algorithms for the implementation of AND-parallelism in the AOPM are given in [9, 8]. The model by Chang *et al* [3] is very similar to the AOPM. The major difference between the two models is that the model by Chang *et al* performs a static analysis of the data dependencies for each clause body and then uses the worst case results to implement AND-parallelism. This model has shown that static analysis of data dependencies results in a major reduction in the run time overheads.

Restricted AND-parallelism model (RAP)

Restricted And-parallelism [11–13] allows only independent literals in a goal statement to be executed in parallel. This ensures that only variables bound to ground terms, or variables which have been established to be independent, may be shared. This way, the problems associated with maintaining consistency for shared bindings are avoided. In Hermenegildo's RAP model [13] *Conditional Control Expressions (CGE's)* generated at compile time, are used to reduce the run time overhead of data dependency analysis. A *CGE* may be defined informally as a series of conditions followed by a conjunction of literals. i.e.

$$(\langle conditions \rangle | L_1 \& L_2 \& \ldots \& L_n)$$

where $\langle conditions \rangle$ represent any number of conjunctions or disjunctions of checks on a $\langle var\text{-}list \rangle$ and $\langle var\text{-}list \rangle$ is a collection of variable names which have their first occurrence before(i.e. "to the left of" in Prolog) the $\langle conditions \rangle$ field of the current graph expression. In this definition, CGE's can appear in the body of a clause in any position a conventional literal may appear. Therefore they may also be nested. The three types of tests which can be used inside conditions are:

- *ground*($\langle var\text{-}list \rangle$):
  will evaluate to true iff all variables in $\langle var\text{-}list \rangle$ are ground, i.e. they are bound to a term with no uninstantiated variables.
- *independent*($\langle var\text{-}list \rangle$):
  The "set of contained variables" (SCV) for each variable is defined as follows: If a variable is bound to a fully ground term, the SCV is empty. If the variable is unbound the SCV contains a single element; the variable itself. If the variable is bound to a term and some of its arguments are variables, the SCV is recursively defined as the union of the SCVs for each of those variables. The test *independent*($\langle var\text{-}list \rangle$) evaluates to true iff the intersection of all the SCVs associated with each variable in $\langle var\text{-}list \rangle$ is empty.
- The logical values *true* and *false*.

Example 2: Forward execution

The forward semantics of CGEs dictate that: "If $\langle conditions \rangle$ evaluates to true, then all the expressions inside the CGE can be executed in parallel. Otherwise, they must be executed sequentially and in the order they appear within the expression" [14].

Suppose we have the following Prolog clause:

$$f(X,Y) :- g(X,Y), h(X), k(Y)$$

In general, the literals g, h, and k cannot run in parallel since they have variables in common. However, if both X and Y are *ground* and/or *independent* then AND parallelism can be extracted from the clause. This fact can be expressed by the following CGE:

$$f(X,Y) :- \ (ground(X,Y) | g(X,Y) \& indep(X,Y) \ | h(X) \& k(Y))$$

If X and Y are ground on entry to the CGE then g, h and k, will run in parallel. If X and Y are not *ground*, then g will be executed first. As soon as g succeeds, $indep(X,Y)$ is checked. If X and Y are independent then h and k will be started in parallel.

Backward execution and Intelligent Backtracking

Consider the following annotated clause. The CGE is underlined.

$$f(..) :- a(..), b(..), \underline{(\langle cond \rangle | c(..) \& d(..) \& e(..))}, g(..), h(..)$$

Conventional backtracking is used if a failure occurs during sequential execution, e.g. if a, b or h fail. If g fails then one of two things can happen. If none of the literals inside the *CGE* has unexplored alternatives, then the interpreter backtracks to b. If at least one literal inside the *CGE* has unexplored alternatives then backtracking inside the *CGE* is done sequentially in reverse order i.e. to the "rightmost" literal with untried alternatives. If a failure occurs during the parallel execution of the literals inside a *CGE*, e.g. if c, d or e fail, then intelligent backtracking can be done by abandoning the whole *CGE*, since the variables inside the *CGE* are either ground or independent.

The APPNet

APPNet, the Distributed AND-OR Parallel Prolog Network has been designed and implemented by Wrench [25]. The APPNet is based on the Delphi Principle [5]. The Delphi principle and the corresponding Delphi machine [15], rely on dynamically splitting the proof tree into independent subtrees, with each subtree being allocated to a separate processing element (*PE*). This way, the computation of a given pre-determined path of the proof tree (from the topmost goal node to a terminal node) is isolated to a single processor. The traditional and-or proof tree is used to represent the partitioned search space. *Oracles* are used to assign individual subtrees to *PEs*. An *oracle* consists of two paths; an and-path and an or-path.

The *oracles* that specify the paths to the terminal nodes for the map colouring program of Example 3 are shown at

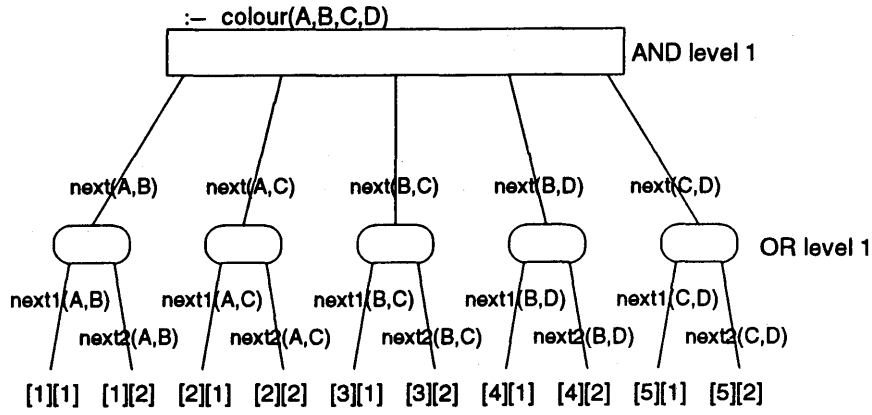

Figure 4. Proof tree for the map colouring problem

the terminal nodes of the And-Or proof tree of Figure 4. Each *oracle* is an ordered pair [and-path][or-path]. The i^{th} element in the *oracle* represents the branch to take at the i^{th} AND node (for the and-path) and the i^{th} OR node (for the or-path) on the current execution path. If each unique traversal of the tree is allocated a separate *PE*, ten partial solutions are generated.

Example 3

$$
\begin{aligned}
\textit{colour3(A,B,C,D)} \;:-\;& next(A,B),\\
& next(A,C),\\
& next(B,C),\\
& next(B,D),\\
& next(C,D).
\end{aligned}
$$

$$
\begin{aligned}
\textit{next(X,Y)} \;:-\;& next1(X,Y).\\
\textit{next(X,Y)} \;:-\;& next2(X,Y).
\end{aligned}
$$

next1(red,green).
next1(red,blue).
next1(green,blue).

$$
\textit{next2(X,Y)} \;:-\; next1(Y,X).
$$

Forward Execution

The network consists of *n PEs*, a *Command Server* and an *Answer Server*. At system start up, the *Command Server* configures the network by loading the entire user program plus the top level query into each *PE*. This creates an initial configuration suited to independent execution in each *PE* and also means that no additional loading of any source code needs to be done if a *PE* were to execute a different subtree at a later stage. As soon as a *PE* receives the go ahead, it commences the execution of its allocated portion of the search space in an attempt to find a solution. The control strategies for partitioning the search space and the means available for communicating the individual paths to each *PE* are given in [25].

Back-unification

Back-unification of partial results is needed when several *PEs* are used to execute the literals of a clause, *(dependent AND-parallelism)*. This ensures that inconsistent bindings which arise from the sharing of variables by the literals are eliminated. In the APPNet *back-unification* is done in parallel. If the solution generated by a *PE* is only a partial solution, the *PE* requests further partial solutions from the *Answer Server*. If none are available, it logs its answer with the *Answer Server* and becomes idle. On the other hand if a valid partial is found a complete copy of this partial is returned to the *PE* where a *join* takes place. The resulting partial (or complete) solution is then returned to the Answer Server. This can result in further *back-unification* if more valid partial solutions are available; if none are found, or if the solution generated is complete, the *PE* becomes idle.

The complete list of partials for the map colouring program of Example 3, using the template (A,B,C,D), is given in Table 1. The colours red, green, blue, have been abbreviated to r, g, b. Each partial is accompanied by the *oracle* used for its generation. The *oracles* are use to determine how partials may be back-unified. For this example, the solutions generated from the five And-paths at And-level 1 are all back-unified. However, the partials from the same And-path should not be back-unified with each other. For example, the partials in the second column of the table, from oracles [1][1] and [1][2] belong to the same And-path and so are alternative sets of solutions for next(A,B). A full cross product is therefore performed on columns two to six of Table 1. In general, there may be more than one And-level and more than one Or-level. In such a case, two partial solutions may be back-unified if their And-path oracles differ only in the last element. The treatment of Or-path oracles is slightly more complicated.

Table 2 gives the partials which produce the final solutions. The partials in each row will produce the final solution shown in the last column. The algorithms, necessary data structures and alternative strategies for back-unification are given in [25].

Comments The above example illustrates one of the major problems associated with *dependent AND-parallelism*. The number of partial solutions generated is quite large compared to the actual number of final solutions.

Table 1. Partial solutions for the call *colour3(A,B,C,D)*

	Literal [And-path]																			
	next(A,B) [1]				next(A,C) [2]				next(B,C) [3]				next(B,D) [4]				next(C,D) [5]			
[Or-path]	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
[1]	r	g	-	-	r	-	g	-	-	r	g	-	-	r	-	g	-	-	r	g
[1]	r	b	-	-	r	-	b	-	-	r	b	-	-	r	-	b	-	-	r	b
[1]	g	b	-	-	g	-	b	-	-	g	b	-	-	g	-	b	-	-	g	b
[2]	g	r	-	-	g	-	r	-	-	g	r	-	-	g	-	r	-	-	g	r
[2]	b	r	-	-	b	-	r	-	-	b	r	-	-	b	-	r	-	-	b	r
[2]	b	g	-	-	b	-	g	-	-	b	g	-	-	b	-	g	-	-	b	g

Table 2. Generation of solutions for the call *colour3(A,B,C,D)*

Literal [and-path]																				Final Solution			
next(A,B) [1]				next(A,C) [2]				next(B,C) [3]				next(B,D) [4]				next(C,D) [5]				colour3 (A,B,C,D)			
A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
r	g	-	-	r	-	b	-	-	g	b	-	-	g	-	r	-	-	b	r	r	g	b	r
r	b	-	-	r	-	g	-	-	b	g	-	-	b	-	r	-	-	g	r	r	b	g	r
g	b	-	-	g	-	r	-	-	b	r	-	-	b	-	g	-	-	r	g	g	b	r	g
g	r	-	-	g	-	b	-	-	r	b	-	-	r	-	g	-	-	b	g	g	r	b	g
b	r	-	-	b	-	g	-	-	r	g	-	-	r	-	b	-	-	g	b	b	r	g	b
b	g	-	-	b	-	r	-	-	g	r	-	-	g	-	b	-	-	r	b	b	g	r	b

Wrench [25] has pointed out that the use of constraints on the forward path would greatly reduce the number of unusable partial solutions which are generated.

5 Conclusions

Or-parallelism speeds up the execution of highly non-deterministic problems by executing alternative procedures in parallel. And-parallelism speeds up the execution of heavily deterministic problems by executing the literals of a goal statement in parallel. A combination of And-parallelism and OR-parallelism ensures that all possible parallelism for a given problem is exploited. The AOPM, the earliest parallel execution model, showed that decisions made at run time result in correct implementation of parallelism. Unfortunately, the AOPM suffers from huge runtime overheads involved in determining when parallelism should be exploited. Later models like the RAP and the APPNet perform a compile time analysis of the program so that the decisions to be made at run time are greatly simplified.

Currently, intensive research is going on in the area of Constraint Logic Programming (CLP). The basic idea behind CLP is to actively use constraints to prune the search space by disabling the generation of combinations of values which cannot appear together in a solution. The combination of constraint propagation and parallel execution of logic programs should result in major speed ups for logic program execution. This is our current area of research.

References

1. K A M Ali. 'Or-parallel execution of Prolog on a multi-sequential machine'. *International Journal of Parallel programming*, **12**(3):189–214, (1987).
2. M Brynooghe and L Pereira. 'Deductive revision by intelligent backtracking'. In J Campbell, ed., *Implementations of Prolog*, pp. 194–215. Ellis Horwood Ltd, Chichester, (1984).
3. J H Chang, A Despain, and D DeGroot. 'And-parallelism of logic programs based on a static data dependency analysis'. *COMPCON*, pp. 218–225, (1985).
4. K Clark and S Gregory. 'Parlog: Parallel programming in logic'. *ACM. Trans. Prog. Lang. Syst.*, **8**(1):299–308, (1986).
5. W Clocksin. 'Principles of the Delphi parallel inference machine'. *Computer Journal*, **30**(5):386–392, (1987).
6. W Clocksin and H Alshawi. 'A method for efficiently executing Horn clause programs using multiple processors'. *New Generation Computing*, **5**:361–376, (1988).
7. A Colmerauer. 'Prolog and infinite trees'. In K Clark and S Tarnlund, eds., *Logic Programming*, pp. 231–251. Academic Press Inc., (1982).
8. J Conery. 'Implementing backward execution in non-deterministic and-parallel systems'. *Proc. 4th ICLP*, pp. 633–653, (1987).
9. J Conery. *Parallel Execution of Logic Programs*. Kluwer Publishers, Boston, 1987.
10. P Cox. 'Finding backtrack points for intelligent backtracking'. In J Campbell, ed., *Implementations of*

Prolog, pp. 216–233. Ellis Horwood Ltd, Chichester, (1984).
11. D DeGroot. 'Restricted and-parallelism'. *Proc. of the International Conference on Fifth generation Computer Systems*, pp. 471–478, (1984).
12. D DeGroot. 'Restricted and-parallelism and side effects'. *Proc. 1987 International Symposium on Logic Programming*, pp. 80–91, (1987).
13. M Hermenegildo. 'An abstract machine for restricted and-parallel execution of logic programs'. *Proc. 3rd. ICLP, Lecture Notes in Computer Science*, **225**:25–39, (1986).
14. M Hermenegildo and R Nasr. 'Efficient management of backtracking in and-parallelism'. *Proc. 3rd. ICLP, Lecture Notes in Computer Science*, **225**:40–54, (1986).
15. C Klein. *Exploiting Or-parallelism in Prolog using Multiple Sequential Machines*. PhD thesis, University of Cambridge, 1989.
16. E Lusk, D Warren, *et al.* 'The Aurora or-parallel Prolog system'. *Proc. International conf. on Fifth generation Computing Systems*, pp. 819–830, (1988).
17. L Pereira and A Porto. 'Selective backtracking'. In K Clark and S Tarnlund, eds., *Logic Programming*, pp. 107–114. Academic Press, (1982).
18. J Robinson. 'A machine oriented logic based on the resolution principle'. *J. ACM*, **12**(1):23–41, (1965).
19. E Shapiro. 'A subset of concurrent Prolog and its interpreter'. Technical report, Institute for New Generation Computing Technology, Tokyo, Japan, (January 1983).
20. K Ueda. 'Introduction to guarded Horn clauses'. *ICOT Journal*, **13**:9–18, (1986).
21. M van Emden and R Kowalsi. 'The semantics of predicate logic as a programming language'. *J. ACM*, **23**(4):773–742, (1976).
22. D Warren. 'The SRI model for or-parallel execution of Prolog – an abstract design and implementation'. *Proc. 1987 International Symposium on Logic Programming*, pp. 92–102, (1987).
23. M Wise. 'EPILOG: Re-interpreting and extending Prolog for a multiprocessor environment'. In J Campbell, ed., *Implementations of Prolog*, pp. 341–351. Ellis Horwood Ltd, Chichester, (1984).
24. M Wise. 'Experiences with PMS-Prolog: a distributed, coarse-grain parallel Prolog with processes, modules and streams'. *Software Practice and Experience*, **23**(2):151–175, (1993).
25. K Wrench. 'A distributed and-or parallel Prolog network'. Technical Report 212, University of Cambridge Computer Laboratory, (December 1990).

Acknowledgements: Thanks are due to Dr. Karen Bradshaw, Department of Computer Science, University of the Witwatersrand, for her constructive advice and the intensive discussions which have enabled me to write this paper.

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for *review* should be prepared according to the following guidelines.

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets [1–3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) LaTeX file(s), either on a diskette, or via e-mail/ftp – a LaTeX style file is available from the production editor;
2. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

   Further instructions on how to reduce page charges can be obtained from the production editor.
3. In camera-ready format – a detailed page specification is available from the production editor;
4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R30-00 per final page for LaTeX or camera-ready contributions and the maximum is R120-00 per page for contributions in typed format (charges include VAT).

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 2 and 4 above will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

1. E Ashcroft and Z Manna. 'The translation of 'goto' programs to 'while' programs'. In *Proceedings of IFIP Congress 71*, pp. 250–255, Amsterdam, (1972). North-Holland.
2. C Bohm and G Jacopini. 'Flow diagrams, turing machines and languages with only two formation rules'. *Communications of the ACM*, **9**:366–371, (1966).
3. S Ginsburg. *Mathematical theory of context free languages*. McGraw Hill, New York, 1966.

South African Computer Journal

Number 11, May 1994
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 11, Mei 1994
ISSN 1015-7999

Contents