Parallel Interpretation of Logic Programs

John S. Conery*
Dennis F. Kibler

Department of Information and Computer Science
University of California, Irvine

Abstract

Logic programs offer many opportunities for parallelism. We present an abstract model that exploits the parallelism due to nondeterministic choices in a logic program. A working interpreter based on this model is described, along with variants of the basic model that are capable of exploiting other sources of parallelism. We conclude with a discussion of our plans for experimenting with the various models, plans which we hope will lead eventually to a multi-processor machine.

1. Introduction

Kowalski [3] assigned a procedural semantics to predicate calculus, so that logic could be used as a programming language. More specifically, logic programming refers to programming with Horn clauses. Prolog is a high-level applicative language based on logic programming. The language was originally implemented in Marseilles [9, 2] as a tool for building natural language front-ends. Since then the language has been implemented on a number of different computer systems (c.f. [6]) and has been used for research and application development, primarily outside the United States. Pure research using the language has been done in such diverse areas as plane geometry, learning, generalization, planning, symbolic calculus, natural language understanding, speech understanding, chess, query optimization, and robotics. Applications have included compilers, interpreters, debuggers, drug interaction prediction, architecture design aids, CAI, and small data bases.

* Authors appear alphabetically

The first compiler was written by David Warren [11] for the DEC-10 and both the interpreted and the compiled code are comparable to LISP in terms of execution speed. The DECsystem-10 interpreter [6] uses a depth first search of the AND/OR tree defined by the program. There are a number of other, more "intelligent" interpreters. IC-Prolog [1] has control annotations to help guide the search by using certain runtime information. A selective backtracking interpreter [7, 8] keeps track of where values are created, so that if a value later causes a failure, the interpreter can backtrack directly to the source of the error. David Warren [12] has written an interpreter which dynamically reorders the order of goals to be executed.

All of the above mentioned efforts aim at increasing the efficiency of logic programs executing on a computer with a single processor. In this paper we present a model for parallel interpretation of logic programs. Eventually we will design a multiple processor system to carry out this parallel interpretation, but here we confine our discussion to the issues of parallelism.

The remainder of this paper is divided into six sections:

Section 2 is a bare bones introduction to logic programs with some sample computations. This section is intended for readers who are not familiar with logic programming or the Prolog language, and may be skipped without loss of context.

Section 3 outlines four types of parallelism possible with logic programs, and in section 4 we present the abstract model for parallel interpretation of logic programs. Our philosophy in this section is to give a model of parallel computation which is as simple as possible. For example, we do not address the problems of processor allocation nor of message structures.

Section 5 discusses one concrete realization (in the form of an interpreter written in Prolog) of the abstract model. Some elaborations of the abstract model are given here.

Section 6 presents some alternatives and extensions of the basic model. These alternatives exploit sources of parallelism that are ignored in the simple model of section 4.

Finally, section 7 outlines the form that our experiments will take, the statistics that we will gather and the results we hope to obtain.

2. Logic Programming

This section contains definitions, some simple logic computations, and a brief introduction to Prolog (using the syntax of DECsystem-10 Prolog). We only discuss the procedural semantics of Prolog; the declarative semantics can be found in [4].

A logic program consists entirely of a set of clauses. Clauses can be either implications (which are described later) or assertions. The following is a simple Prolog program which has two assertions:

```
mother(peg,judy).
mother(judy,kara).
```

In this example the words "peg, judy, and kara" are constants and the word "mother" is a functor of two arguments. An atom is a constant (an uninterpreted symbol) or an integer. Constants are denoted by strings which begin with a lower case letter. Variables are denoted by strings which begin with an upper case letter. A term is a variable or an atom or a functor of n-arguments, where each argument is itself a term. An example of a more complicated term is

```
fee(Fie,foe(3,fum(2,Foo))).
```

Terms are the basic data structure provided in Prolog.

To use a program, the user supplies a term or a list of terms, which are called goals. Some example queries and responses are:

	query	response
	mother(peg,judy).	yes.
	mother(peg,kara).	no.
	mother(X,judy).	X=peg.
	mother(X,Y).	X=peg,Y=judy.
	mother(X,X).	no.
*	mother(X,Y),female(Y).	no.
	mother(X,Y),mother(X,kara).	X=judy,Y=kara.

To answer a query, the Prolog interpreter pattern-matches each goal of the query against the list of clauses in the program. This list of clauses is often referred to as the data base. The particular pattern matching algorithm used is unification. Unification accepts two terms, which may contain variables. Two terms can be unified if there is a substitution for the variables that makes the terms identical. For example, the two terms p(X,a) and p(b,Y) can be unified by the substitution {X/b,Y/a}, meaning substitute "b" for X and "a" for Y, to give the single term p(b,a). Unification is more general than most pattern matching algorithms since it allows pattern variables in both terms. The use of variables in Prolog is analogous to the use of dummy variables in mathematics, in that the "scope" of a variable is simply the clause in which it appears.

The standard Prolog interpreter solves a conjunction of goals by working from left to right, unifying each term with the data base. When a term unifies with some assertion in the data base, the substitution generated is applied to the remaining terms in the query. This continues until either no terms are left, indicating success, or until some term cannot be unified with any assertion in the data base, at which point backtracking (explained later) occurs. This process explains how queries up to query * are answered.

To understand the answer to * requires an understanding of backtracking. Prolog can solve the first goal with substitution {X/peg,Y/judy}. However, there are no assertions for female, so the goal female(judy) fails. When Prolog fails to solve a goal, it backtracks, i.e. it tries to find a different solution to the most recently solved goal. Thus, in our example the goal mother(X,Y) is retried and the substitution {X/judy,Y/kara} is found. Now the goal female(kara) is attempted, and fails since there are still no assertions for female. For the third time Prolog tries to solve mother(X,Y). It is unable to find any solution and fails, responding "no". Notice that if the query were

```
female(Y),mother(X,Y).
```

then failure would have been immediate.

Now let us extend our program by adding the clause

```
female(X):-mother(X,Y).
```

Our program now appears as

```
mother(peg,judy).
mother(judy,kara).
female(X):-mother(X,Y).
```

The third clause is an example of an implication. An implication has the form

```
head:-body.
```

where the head is a single term and the body is a (possibly empty) list of terms. Note that an implication with an empty body is an assertion. The terms in either the head or the body may contain variables. As mentioned before these variables may be viewed as being local to the clause. Using our new program, let us return to query * and redo the computation. We need to extend our understanding of the interpreter to allow for the processing of implications.

To solve a goal list, the first goal is removed from the list and matched against the heads of clauses in the data base. Variables are renamed so no clauses have variables in common. If the goal matches an assertion, the action taken is as before. If the goal matches the head of an implication, the body of the implication replaces the head in the goal list, and the substitution generated by the match is applied to the new goal list. The interpreter repeats this cycle until the goal list is empty. Whenever a goal fails, backtracking occurs. If backtracking leads back to the first goal and there are no more alternatives for solving that goal, then the interpreter reports failure.

Looking at this computation with respect to query *, we see that solving the first goal establishes the substitution {X/peg,Y/judy}. These

```
/* Here are two ways of defining the grandfather relationship:          */
/* 1)  X is the grandfather of Z if X is the father of some Y           */
/*     and Y is the father of Z.                                        */
/* 2)  X is the grandfather of Z if X is the father of some Y           */
/*     and Y is the mother of Z.                                        */

gf(X,Z) :- f(X,Y), f(Y,Z).
gf(X,Z) :- f(X,Y), m(Y,Z).

/* these assertions define our database:                                */

f(curt,elaine).            f(sam,larry).
f(dan,pat).                f(larry,den).
f(pat,john).               f(larry,doug).
m(elaine,john).            m(peg,den).
m(marion,elaine).          m(peg,doug).

/* here are some sample queries, annotated with the actions of the      */
/* DECsystem-10 interpreter                                             */

  step   goals to be solved        matching                unifying
                                     clause               substitution

  [1]    gf(sam,G)             gf(X,Z) :- f(X,Y),f(Y,Z)    {X/sam,Z/G}
  [2]    f(sam,Y),f(Y,G)       f(sam,larry)                {Y/larry}
  [3]    f(larry,G)            f(larry,den)                {G/den}
  [4]      <none>

/*  The variable G in the original list was bound to "den", thus        */
/*  the answer is "den" is a grandson of "sam".                         */

/*  This next query can be read as "is there any pair (A,B) such        */
/*  that A is the grandfather of B?"                                    */

  [1]    gf(A,B)               gf(X,Z) :- f(X,Y),f(Y,Z)     {X/A,Z/B}
  [2]    f(A,Y),f(Y,B)         f(curt,elaine)            {A/curt,Y/elaine}
  [3]    f(elaine,B)           <none>
  [4]    f(A,Y),f(Y,B)         f(dan,pat)                  {A/dan,Y/pat}
  [5]    f(pat,B)              f(pat,john)                 {B/john}

/*  The answer is A = "dan" and B = "john"                              */

/*  NOTE:  after Prolog produces an answer, the user may force it to    */
/*  backtrack, and answer the query in another way.  Thus,              */
/*  backtracking would answer G = "doug" to our first query, and        */
/*  generate all grandfather-grandson pairs in response to the last     */
/*  query.                                                              */
```

Figure 2-1: Example of a Prolog Program

bindings are transmitted to the rest of the body. The new goal list is female(judy). This matches the third clause of the data base with the first argument bound to judy and the second argument unbound. The goal list becomes mother(judy,Y′) where Y′ is an arbitrary new variable name. Since mother(judy,Y′) unifies with a member of the data base and there are no more goals remaining to be solved, the interpreter returns X=judy.

A more complicated example of a Prolog program is given in figure 2-1. Later we will show how our parallel interpreter performs the same computation.

References [1], [3], [5], and [6] are other descriptions of logic programming and various Prolog interpreters.

3. Parallelism

The opportunities for parallelism based on logic programs are manifold. To be specific, suppose we had the task of solving the goal list f(X,Y),g(Y),h(Z). By this we mean we must find terms for X, Y and Z which will simultaneously solve each of these goals. Suppose also that the data base or logic program had fl clauses whose head term began with f. Let gl and hl be defined analogously. With this context let us define several types of parallelism. We will rely on an intuitive understanding of terms such as processors, processes, and messages.

A single term may unify with the heads of many clauses in the data base. By OR parallelism we mean assigning a process to solve each body whose head unifies with the term. For instance, OR parallelism for the term f(X,Y) would lead to fl concurrent processes. Note that this form of parallelism replaces backtracking.

By AND parallelism we mean simultaneously starting processes to solve each of the goals of the body. For the example at hand, this means starting three processes, one for f(X,Y), another for g(Y), and a third for h(Z). Since all of these goals eventually need to be solved, one might try to start working on each goal as soon as possible. However since the goals are interrelated, answers to one goal may limit the number of choices for the others. For example, each answer to f(X,Y) may uniquely determine Y, while g(Y) may have a large number of solutions. More generally, if it is known that fl is small and gl is large it is best to solve f before g. Conversely if gl is large and fl small, one would want to solve g before f. Such an approach for solving a conjunct of goals is recommended by Warren [12]. Solving for the goals h(Z) and g(Y) in parallel is reasonable, and is a form of parallelism that we plan to incorporate in our future interpreters based on the model of section 4.

By stream parallelism we mean the eager evaluation of structured data, which can be treated as a stream. For example one might begin testing for membership in a list while the list was being constructed. Clark and McCabe [1] and van Emden [10] explore this form of parallelism.

By search parallelism we refer to the possibility of partitioning the data base into disjoint sets of clauses, permitting parallel searching of the data base. It could be used to initialize the OR parallelism. This would probably be best for programs that contained a large number of assertions, and might be necessary for large data bases.

One problem with many physical implementations of parallel computation is that the overhead in setting-up the parallelism and the overhead associated with communicating among the processes may outweigh the advantages of the parallel computation. In anticipation of this, we are designing our model so that all processes will have roughly the same amount of work to do.

4. The Model

In this section we present our abstract model for parallel interpretation of logic programs: the AND/OR model. The model is based on the concept of independent processes that communicate via messages. It is our long range goal to develop a multi-processor machine such that these processes can be executed in parallel.

There are two types of processes: AND processes and OR processes. It is the basic task of an AND process to produce a solution for a conjunction of goals, whereas OR processes solve single goals. A "snap-shot" of the relationship among processes during the execution of a logic program would reveal an AND/OR tree, with AND processes as the descendants of OR processes, and vice versa.

There are three types of messages: success, fail, and redo. Fail and success are always sent from a descendant to a parent; redo is always sent from a parent to a descendant. A success message contains one possible substitution that satisfies the goal(s) the process was created to solve. In the basic model, a process that sends a success message does no further work, but instead waits for instructions from its parent. A fail message indicates a final failure. A process sending this message indicates that there is no possible solution below it in the search space. After sending a fail message, a process terminates itself. A redo message is sent from a parent to a previously successful descendant; in effect this message says that the previous success message did not help the parent solve its own goal(s), and a different substitution is required.

In the following sections we describe AND and OR processes in detail, the conditions under which the messages are sent, and how each process reacts to each type of message.

4.1. OR Processes

An OR process attempts to solve its single goal by 1) finding every clause such that the head of the clause can be unified with the goal to be solved, then 2) applying the unifying substitutions to the corresponding clause bodies, and finally 3) starting up descendant AND processes to solve these bodies.

If there are no clauses with matching heads, the OR process fails immediately; it sends its parent a fail message and terminates itself. An OR process that creates descendants is in one of two modes: waiting mode or gathering mode. An OR process is in the waiting mode when its parent is waiting for an answer. An OR process is in the gathering mode when it has sent an answer to its parent and the parent is using that answer; in this mode, any further answers received from AND descendants are saved. For example, a process that creates three descendants is in waiting mode while those three descendants solve their corresponding goal lists in parallel. When a descendant sends back a success message, the OR process sends its own parent a success message and then goes into the gathering mode. Now when answers are received from the other two descendants, they are saved for future use, and not (yet) sent to the parent.

The various messages and their effects on OR processes are as follows (this is summarized in figure 4-1):

Success messages:

- A process in the waiting mode uses the bindings in this message to create a success message for its own parent, sends this message, and goes into gathering mode.

- A gathering mode process creates a success message, but instead of sending this message to its parent, the process adds the message to a list of messages waiting to be sent.

Redo messages:

- OR processes in the waiting mode will never be sent redo messages. There are three possible actions for a gathering OR process: if there are any success messages not yet sent, one is selected and sent to the parent; otherwise if there are any descendants still active (i.e. not all have sent fail messages), then they are all sent redo messages, and the process goes into the waiting mode; otherwise the process fails.

Fail messages:

- A gathering OR process simply notes the fact that the descendant failed.

- An OR process in the waiting mode does one of two things: if the descendant that sent the message was the sole remaining descendant, the process fails, otherwise the process notes that fact that the descendant failed.

4.2. AND Processes

This section is a description of the simplest possible AND process, one that does not attempt to exploit any parallelism in its goal list. In section 6.2 we discuss extensions to the model that achieve some AND parallelism. The simplest AND processes solve their conjunctions in a manner that is very similar to how a standard Prolog interpreter solves a goal list: a solution to the first goal is obtained; any variable bindings generated by this solution are applied to the remaining goals, and then the next goal is solved. This continues until all goals from the original list have been solved. If a goal cannot be solved, the previous one must be solved in a different way (so that a different set of variable bindings is produced).

A solution to a single goal is obtained by starting a descendant OR process to solve the goal. Once the descendant has been started, the AND process waits for a message (either success or failure) from that descendant. When a success message, which possibly contains a list of variable bindings, is received, the bindings are applied to the remaining goals, and a process is started for the next goal. If a fail message is received, the AND process must undo any bindings sent by the previous descendant, and then send that descendant a redo message.

If an AND process obtains solutions for each of its goals, it sends a success message back to its own parent process. At a later time, the parent may request a different solution (via a redo message). If it does, the AND process reacts as if it had received a fail message from one of its descendants: the bindings from the most recent success message are undone, and the process that sent that success message is sent a redo message.

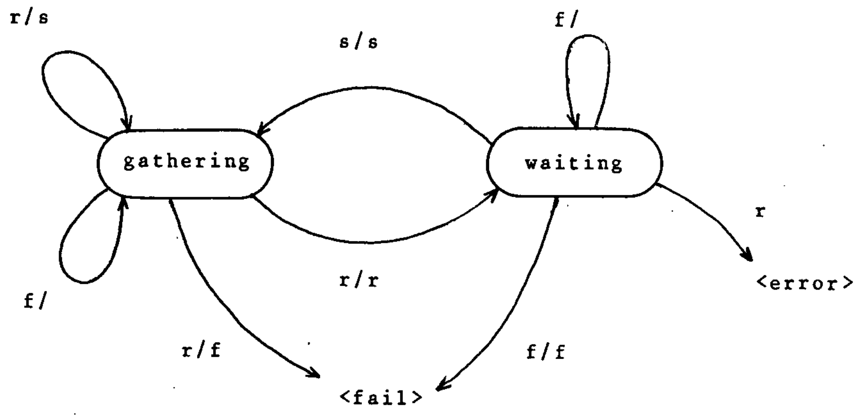
A trivial AND process is one that was created to solve an originally empty goal list, i.e. its parent OR process matched an assertion, which is a clause with an empty body. This corresponds to a goal that always succeeds, and thus this process can immediately send its parent a success; a subsequent redo message causes a failure.

If an AND process receives a fail message from the descendant that was created to solve the first goal in the conjunction, the AND process itself fails.

5. An Interpreter

We have written an interpreter based on the AND/OR model that simulates the actions of AND and OR processes as they work together on the solution of a single logic program. The interpreter is written in DECsystem-10 Prolog [6]. In this section of the paper, we will describe some of the interesting features of the interpreter, which extends the basic model, and present as an example

Figure 4-1: The Effect of Messages on OR Processes



The letters r, f, and s stand for redo, fail, and success.
A label x/y on an arc means that when the OR process receives message x it follows the transition and sends message y.

Notes: 1. If a gathering OR process has a waiting message, it is sent, else if it has descendants, they are sent redos, else the process fails.
2. If a waiting OR process receives a fail from its last remaining descendant, it is itself a failure.

the history of various processes as the interpreter solves the family tree problem presented in an earlier section.

5.1. Trivial AND Processes

In the current implementation, an OR process does not bother to start an AND descendant for a clause if the clause is an assertion, i.e. if the clause has a null body. Instead, the OR process just uses any bindings created by the unification of its goal with the head of the assertion to create a success message. In this case the process can immediately send a success message to its parent and go into the gathering mode. Another benefit in terms of performance is a reduction in the number of processes created.

5.2. Eager Evaluation

In the description of the basic model, it was stated that after an AND process sends a success message to its parent, it goes into a suspended state until the parent sends a redo message. This mode of operation is similar to "lazy evaluation", in that an AND process does not produce a result unless it knows that the result is required by the parent process. Another possibility, which has been implemented in the current interpreter, is that when an OR process receives a success message from an AND descendant, it immediately sends that descendant a redo message. We refer to this as eager evaluation. This mode is similar to the behavior of dataflow systems, where results are computed as soon as input values are present. Future versions of the interpreter will incorporate simulations of physical processing elements (see section 7), and we plan to experiment with the effect of eager versus lazy evaluation on the use of these processing elements. The expected benefit of eager evaluation is that after the first answer to the user's initial query is produced, additional answers will be produced in a shorter time than the time required when lazy evaluation is used. However, when there are a finite number of processing elements, eager evaluation may cause a large amount of unnecessary processing, which means valuable resources are being wasted; the net result might very well be a slower response to all queries.

5.3. Duplicate Answers

A third extension to the basic model that was implemented in the interpreter is the filtering of duplicate answers. Each OR process maintains a list of answers that it has sent to its parent. When a success message comes from a descendant, the OR process uses the bindings in that message to create a success message for its parent. If that message is either in the list of messages waiting to be sent, or in the list of messages already sent, then it is ignored and the descendant is sent a redo message.

By employing this filtering mechanism, a process guarantees that its parent never repeats any (presumably) useless computation, i.e. additional answers are not sent to the parent until the parent sends a redo message, and the parent presumably doesn't send a redo message unless the previous answers were unsatisfactory for some reason. It is interesting to note that the overall effect of the filtering is related to the intelligent backtracking of Pereira and Porto [7], where a backtrack point is skipped if it is known that it cannot produce any further useful information.

5.4. An Example

Figure 5-1 shows a "snap-shot" of the configuration of processes just before the top process is about to receive a success message. A process is represented as a box, where the contents of the box show the goal (for OR processes) or goal list (for AND processes) that the process was given to solve when it was created. The lines connecting boxes represent parent/descendant relationships. A process ID is in braces next to the box.

The first process that is created is a process to answer the user's query. Since users can query the system with a list of goals, this first process is an AND process. In our example, this is a list with only one goal, and thus the AND process has only one OR descendant, which is process {2}. Process {2} can match its goal with the heads of two clauses, so it starts up two AND processes, {3} and {4}, which operate in parallel.

{3} and {4} each have a goal list with two goals. They start {5} and {6} to solve the first of their goals. {5} and {6} match the heads of six assertions; one answer is sent back to the parent and the other five answers are put on a waiting list. {4} and {3} receive success messages, and start {8} and {7}. At this point, {7} fails, but {8} succeeds. {4} can send a success to {2} (since both its goals are solved), but {3} must undo the bindings for A and Y and send {5} a redo message.

When {2} receives the message from {4}, it sends a success to {1} and goes into the gathering mode. In the eager evaluation interpreter, {2} will also send a redo to {4} at this time.

As is shown in the snap-shot, process {5} responded to the redo message by removing an answer from its waiting list and sending it to {3}. {3} uses this substitution to create a new descendant, process {9}.

6. Variants of the Basic Model

In this section we will present two variants of our basic abstract model. Each variant provides for more opportunities for parallelism. We hope to experiment with these and other extensions using future versions of the interpreter.

6.1. Multiple Answers from OR Processes

Referring back to the example in figure 5-1, notice that process {5} solved its single goal immediately when that goal matched six assertions. Under the basic model, one of these answers is sent to the parent AND process, and the others are placed in the waiting list and sent back one by one in response to redo messages.

One extension of the basic model would be to have the OR process send back all answers that it computes in such situations. The parent AND process would then set up descendant OR processes for each answer in the list, where these new OR descendants would operate in parallel. Referring again to the figure, process {5} would send a list of six answers to {3}; {3} would then set up six

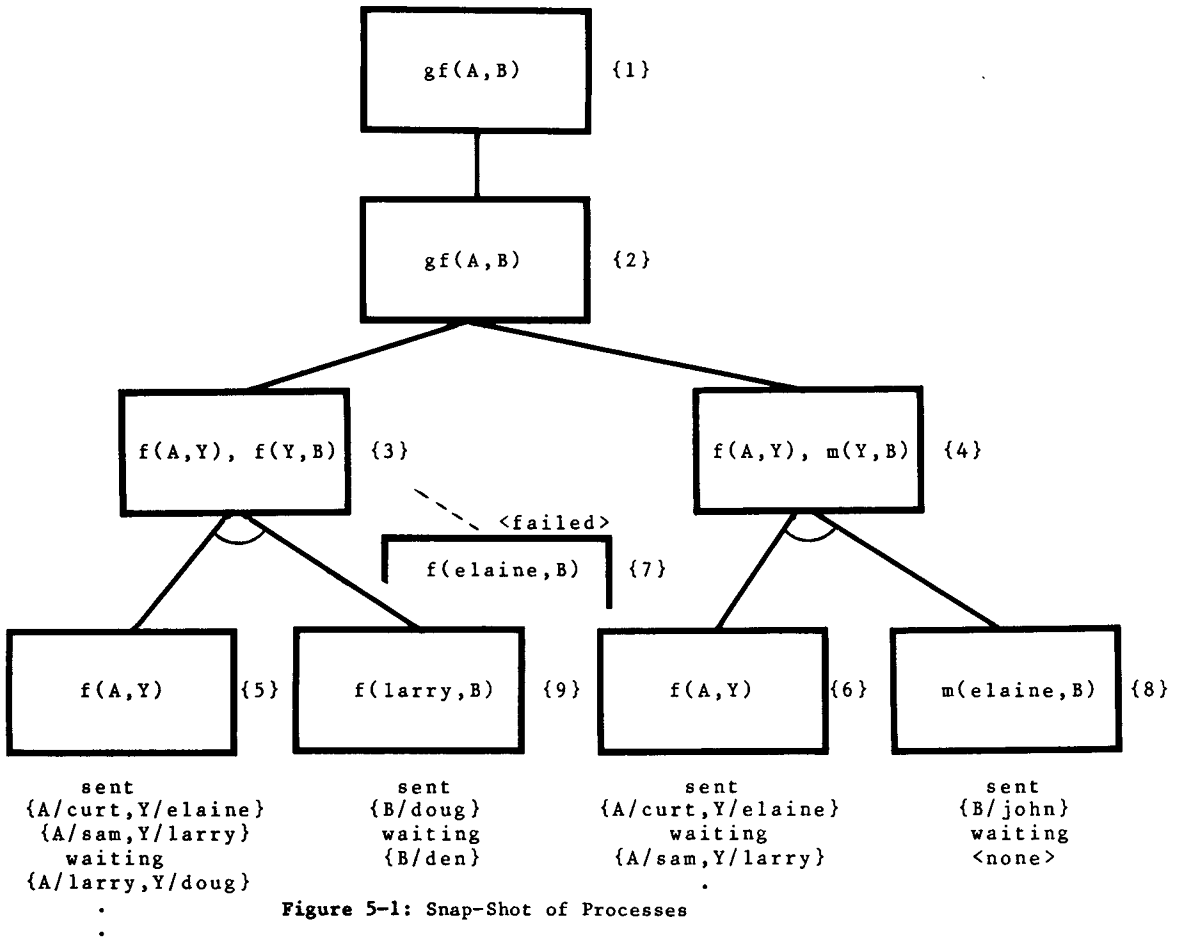


Figure 5-1: Snap-Shot of Processes

processes (including those marked {7} and {9} in the figure) all at the same time, to operate in parallel; the net result would be that process {9} would produce its part of the correct answer at roughly the same time {8} computes its answer.

6.2. AND Node Parallelism

In the basic model, AND processes produce answers to their conjunctions using the same methods employed by standard Prolog interpreters, using backtracking to produce alternate answers. Another extension to the basic model involves increasing the "level of sophistication" of the methods used by AND nodes. One approach is to adapt some of the methods of Warren [12], Pereira and Porto [7, 8], and Clark and McCabe [1]; another is to attempt to exploit parallelism within AND processes. The latter is a difficult issue, one we have not explored in great detail.

AND parallelism was defined previously, in section 3. In that section, we indicated that goals in a conjunction of the form

p(X),q(Y).

could be solved in parallel, since they have no variables in common. Note that if the process to solve p(X) sends a set pl of success messages, and the process for q(Y) sends a set ql, then the AND process must send its own parent every element from the product of pl and ql.

For clauses of the form

gf(X,Z) :- f(X,Y), f(Y,Z).

the situation is more complicated. As others have pointed out, if the query is "gf(X,a)", then the body to be solved is

f(X,Y), f(Y,a).

and it makes sense to solve the second goal first, since the binding of one variable in that term limits the search space somewhat. The IC-Prolog interpreter [1] allows one to annotate clauses, so that different orderings of subgoals are chosen depending on the pattern of variable bindings in the head of the matched clause. We hope to design methods of analyzing such patterns dynamically, so that an AND process can intelligently order its subgoals, allowing it to determine which of them can be solved in parallel, and which must be solved before others.

7. Future Research

The next step in our research is to build simulators for the models of parallel computation that we have defined. Simulation is the first step towards gaining an understanding of the various tradeoffs among the different parallel models of computation.

We have yet to address ourselves to three serious problems. We must define a means for allocation of processors (PEs) to processes. We must define metrics for comparing parallel interpreters. We must develop debugging techniques for programmers. Below we sketch our current thoughts on these problems.

7.1. Processor Allocation

Currently we are debating between a dispatcher and a management model of processor distribution. In the dispatcher model, one special process has the job of collecting and distributing free PEs. In the management model each process is given a number of PEs which it can allocate among its descendants as it sees fit. When a process fails, its PEs are returned to its parent for reallocation.

Since logic programs are inherently non-deterministic, and we are postulating an asynchronous search, the constraint that a program produce the same answer each time it is run is too strong. We define the answer set as the collection of answers one gets by repeatedly applying the same program to the same problem. Two answer sets are equivalent if they are equal as sets, independent of order of elements. We say an answer set A contains an answer set B if each element of B is an element of A. A parallel interpreter is monotonic if after the addition of PEs the new answer sets contain the old answer sets. We have not yet addressed the issue of guaranteeing monotonicity within our model as it depends on processor allocation.

7.2. Metrics for Comparing Parallel Interpreters

Some of the metrics that we plan to gather statistics on are:

1. Size and number of messages sent during the solution of a problem.

2. Measure the ratio of idle time to processing time for each PE. When there are more processes than PEs, we need to measure the amount of time processes are "blocked" versus "ready" in each PE.

3. For each PE, the costs for preparing, routing, and receiving messages.

4. Cost for data base search for each processor.

It will also be possible to measure the number of unifications attempted, and the number of successful unifications, during the solution of a problem by the various "intelligent" single processor interpreters, and compare them with the same statistics for the parallel interpreters.

7.3. Programmer Aids

As can be seen by section 5, it is difficult to explain in a sequential manner an asynchronous, parallel computation. Software engineering has determined that the most difficult programming bugs to find and remove are those that related to control. Although programming in logic will allow the programmer to identify wrong implications or assertions, it will not help him find missing implications or assertions.

By setting the number of allocatable processors to one, the programmer can gain some confidence in his program. By guaranteeing that the parallel processing interpreter's answer stream will always include those answers produced by a single processor, we give the programmer confidence in the parallel computation without requiring that he understand the details of process creation or processor allocation.

Acknowledgements

We would like to thank the anonymous referees for many helpful criticisms of an earlier version of this paper. For comments on later versions, we are indebted to our friends and colleagues, Steve Fickas, Jim Neighbors, Paul Morris, and Steve Willson.

Portions of the research presented in this paper were supported by grant N66001-80-C-0377, from the Naval Oceans Systems Center, in San Diego, California.

References

1. Clark, K. L., and G. McCabe. The Control Facilities of IC-Prolog. In D. Michie, Ed., Expert Systems in the Micro Electronic Age, Edinburgh University Press, 1979.

2. Colmerauer, A. Les Grammaire de Metamorphose. Univ. d'Aix-Marseille, Groupe de IA, 1975.

3. Kowalski, R. A. Predicate Logic as a Programming Language. Proc. IFIPS 74, 1974.

4. Kowalski, R. A. Logic for Problem Solving. Elsevier - North Holland, New York, 1979.

5. Kowalski, R. A. Logic as a Computer Language. Proc. Infotech State of the Art Conference "Software Development: Management", June, 1980.

6. Pereira, L. M., F. C. N. Pereira, and D. H. D. Warren. User's Guide to DECsystem-10 Prolog. Dept. of Artificial Intelligence, Univ. of Edinburgh, September, 1978. version 1.32

7. Pereira, L. P. and A. Porto. Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Theory. Report 2/79, Departamento de Informatica, Universidade Nova de Lisboa, October, 1979.

8. Pereira, L. P. and A. Porto. An Interpreter of Logic Programs Using Selective Backtracking. Report 3/80, Departamento de Informatica, Universidade Nova de Lisboa, July, 1980.

9. Roussel, P. Manuel de Reference et d'Utilisation. Univ. d'Aix-Marseille, Groupe de IA, 1975.

10. van Emden, M. H. and G. J. de Lucena. Predicate Logic as a Language for Parallel Programming. In K. L. Clark and S. A. Tarnlund, Ed., Logic Programming, Academic Press, New York, 1981.

11. Warren, D. H. D., L. M. Pereira, and F. C. N. Pereira. Prolog - The Language and its Implementation Compared with LISP. ACM SIGPLAN Notices 12, 8 (1977), 109-115.

12. D. H. D. Warren. Efficient Processing of Interactive Relational Database Queries Expressed in Logic. Dept. of Artificial Intelligence, Univ. of Edinburgh, September, 1981. Paper 156