Parallel Programming in Prolog

Robert M. Keller
Quintus Computer Systems, Inc
Mountain View
California
What Prolog is:

A major landmark in language design, comparable in scope to the invention of Fortan, Algol, Lisp, Cobol.

An language usually implemented with an interactive environment.

A language in which complete applications can be programmed.

A language within several built-in features not found in other languages, including:

- Pattern-matching via unification
- Backtracking
- Integral database
- Comprehensive meta-facilities

A language which strongly relates to certain aspects of logic.

A language in which relational or entity-relation database concepts are naturally expressed.

A language with only a very modular syntax and only 2-levels of lexical scoping.
What kinds of applications are suitable for Prolog?

Database applications

Compilers

Rapid Prototyping of all kinds (including AI)

Expert systems, and other applications requiring meta facilities

Program-transforming programs
Prolog Users

- Universities
- Research institutions
- Government agencies
- Corporate
  - AI groups
  - Research and development
  - MIS
- System integrators / application developers
Prolog Application Areas

- Knowledge based systems
- Fault analysis
- Diagnosis
- Configuration
- Monitoring complex situations

- Components of traditional applications
- Design
- Intelligent front ends
- Compilers, generators
- Translators
Some Prolog Application Markets

- Manufacturing (aerospace, automobile, electronics)
- CAD (electronic, mechanical, architectural)
- Database, decision support (financial)
- CASE (software engineering)
### Some History of Prolog

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Location</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1965</td>
<td>J.A. Robinson</td>
<td>Syracuse</td>
<td>Development of Resolution principle</td>
</tr>
<tr>
<td>1973</td>
<td>R. Kowalski</td>
<td>Edinburgh</td>
<td>Predicate Logic as a Programming Language</td>
</tr>
<tr>
<td>1975</td>
<td>A. Colmerauer</td>
<td>Marseille</td>
<td>Metamorphosis-Grammar Language</td>
</tr>
<tr>
<td>1975</td>
<td>P. Roussel</td>
<td>Marseille</td>
<td>First Prolog Implementation</td>
</tr>
<tr>
<td>1977</td>
<td>D.H.D. Warren</td>
<td>Edinburgh</td>
<td>First compiler (DEC-10 Prolog)</td>
</tr>
<tr>
<td>1980?</td>
<td>K. Clark</td>
<td>London</td>
<td>MicroProlog developed</td>
</tr>
<tr>
<td>1981</td>
<td>K. Fuchi</td>
<td>Tokyo</td>
<td>Fifth-generation project announced</td>
</tr>
<tr>
<td>1982</td>
<td></td>
<td>Marseille</td>
<td>First International Conference on Logic Pgm.</td>
</tr>
<tr>
<td>1982</td>
<td>F. Pereira</td>
<td>Edinburgh</td>
<td>C-Prolog distributed</td>
</tr>
<tr>
<td>1984</td>
<td></td>
<td>Atlantic City</td>
<td>First U.S. Symposium on Logic Programming</td>
</tr>
<tr>
<td>1985</td>
<td>J.A. Robinson</td>
<td>Syracuse</td>
<td>Journal of Logic Programming started</td>
</tr>
<tr>
<td>1984</td>
<td>D.H.D. Warren</td>
<td>Palo Alto</td>
<td>Quintus founded</td>
</tr>
<tr>
<td>1986</td>
<td></td>
<td>London</td>
<td>Association for Logic Programming started</td>
</tr>
</tbody>
</table>
What is the syntax of a Prolog program?

There are two parts:

A set of *clauses* (relatively static)

A *query* or *goal* (which starts things off)

The primitive *syntactic construct* of the above is the Prolog *term*.

A *term* is either:

A numeral, *e.g.* 5, -99, 6.23E-14

An *atom*, designated by a string of characters beginning with a lower case character, *e.g.* foo, b1234, x, aFAR, or enclosed in single quotes, *e.g.* 'Abraham Lincoln', 'X'.

A *variable*, designated by a string of characters beginning with an upper case character, *e.g.* X, Foo, A21, or beginning with the special character _.

A *composite term*, which has the form

\[ \text{constructor}( \text{term}_1, \text{term}_2, \ldots, \text{term}_n) \]

where each \( \text{term}_i \) is a term and \text{constructor} is an atom, *e.g.*

\[ \text{a}(1) \]

\[ \text{baz}(c, x23) \]

\[ \text{foo}(\text{bar}(a), \text{hiss}) \]

are all composite terms.
Operator syntax:

Some functors can be alternatively expressed as *operators*, which in Prolog means that they may have infix, prefix, or postfix representations.

An example is the + functor: We can write a term as either

\[ + (X, Y) \]

or as

\[ X + Y \]

Some functors have a built-in operator syntax.

For others, operator syntax may be declared by the user.
Interpretation of Prolog terms:

Prolog terms are abstract; their meaning depends on context.

For example, they may mean:

- **Record construction and component selection.**
  
  employee(Name, Address, SSNo)

- **Expression to be evaluated (as in an arithmetic expression)**
  
  $3^*X + 5$

  (such an expression is actually evaluated only when the context so indicates)

- **Goals directing Prolog execution:**
  
  brother(john, X)

- **Prolog program fragments:**
  
  son(X, Y) :- parent(Y, X), male(X).

  (In the above term, :- is the constructor with operator syntax.)
There are many options for

**Representation of Knowledge in Prolog**

Several types of representation are possible.

**Example:** Suppose we wish to represent the following bits of information:

- A certain animal, $x$, is a rabbit.
- $x$ is white.
- $x$ has red eyes.

We could do this with various of the following terms:

- `rabbit(x)`
- `x(rabbit)`
- `is(rabbit, x)`
- `is(x, rabbit)`
- `white(x)`
- `x(white)`
- `has_color(x, white)`
- `eye_color(x, red)`
- `has_color(eyes(x), red)`
- `red(x(eyes))`
- `color(eyes, x, red)`

*etc.*

The choice of representation will depend on the rules for using the knowledge.
Prolog clauses

The bulk of a Prolog program is usually a set of *clauses*. These are the counterpart of statements in most languages, but depending on their form, take meaning as:

- Procedure definitions
- Declarations of logical relationships
- Data definitions
Facts

The simplest form of Prolog clause is called a fact (or unit clause). It has the form:

Prolog term.

The period following the term identifies this as a fact. Here is an example of several facts:

male(john).
parent(john, caroline).
parent(jackie, caroline).
female(jackie).
female(caroline).

Here we may interpret the atoms john, jackie, and caroline as individuals, and the facts assert relationships among the individuals.
Rules

The general form of Prolog clause is called a rule. A rule has the form

\[ \text{term} \leftarrow \text{term}_1, \text{term}_2, \ldots, \text{term}_n. \]

The term on the left of \(\leftarrow\) is called the head of the rule. The other terms comprise the body of the rule.

The symbol \(\leftarrow\) can be read if, or provided. The meaning of the rule is that the term on the left hand side is solvable as a Prolog goal provided that the terms on the right hand side are all solvable, in the sequence indicated.

Example:

\[ \text{father}(X, Y) \leftarrow \text{parent}(X, Y), \text{male}(X). \]

This asserts that a goal \(\text{father}(X, Y)\) is solvable if \(\text{parent}(X, Y)\) is solvable and \(\text{male}(X)\) is solvable.

We see that the commas separating the body terms can be interpreted as a logical and, while the symbol \(\leftarrow\) can be interpreted as logical if (in other words, the conjunction of the right-hand side terms logically implies the left-hand side).

Keep in mind that \(X\) and \(Y\) above are Prolog Variables, which means that they represent arbitrary entities. Furthermore, these variables are regarded as being distinct from variables of the same name in any other clauses.
Building Prolog Applications

End-User Interface

Prolog Application Code

Prototyping Console Layer

Query Interface

Prolog Kernel

"Foreign Code"
(C, Fortran, etc.)

Operating System

Processor

Quintus Computer Systems, 1988
**Prolog Database:** The collection of all clauses for a program (facts and rules) is referred to as the *database*.

**Example:** Suppose that the database is:

```
male(john).
pARENT(john, caroline).
pARENT(jackie, caroline).
female(jackie).
female(caroline).

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

**Ground terms:** A term is called *ground* if it contains no variables. When a goal is a ground term, it is either solvable, or not, i.e. there is a yes or no answer. Prolog can determine solvability of the goal relative to the current database. This is done by entering the goal following the prompt `?-`.

Consider Prolog's response to some typical ground goals:

```
?- female(jackie).
yes

?- parent(john, jackie).
no

?- mother(jackie, caroline).
yes
```

In the first two queries, Prolog simply only used facts in the database. In the third query, it had to use the rule for mother, which translates into the two goals

```
female(jackie), parent(jackie, caroline).
```

In this instance, the two goals are solvable by facts. In general, long chains of rules might be necessary to establish a goal.
Non-ground goals:

When a Prolog goal is non-ground, Prolog tries to find a way of solving it by substituting a value for the variables, if necessary. Such a substitution also called an *instantiation* or *binding*.

**Examples:** Assume again the database

```
male(john).
pARENT(john, caroline).
pARENT(jackie, caroline).
female(jackie).
female(caroline).

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

Here are Prolog's responses (in bold-face) to some non-ground goals:

```
?- male(X).
  X = john

?- female(X).
  X = jackie

?- parent(john, X).
  X = caroline

?- parent(jackie, X).
  X = caroline

?- parent(X, Y).
  X = john,  
  Y = caroline

?- parent(X, X).
  no

?- mother(X, Y).
  X = jackie.
  Y = caroline.
```
Multiple Solutions to a Goal:

In certain of the goals in the preceding example, we see that there are multiple ways in which the goal could be true in principle. For example,

?- female(X).

Here either \( X = \) jackie or \( X = \) caroline would satisfy the property.

When Prolog presents its answer to a query, if we follow by return, that is the only answer we shall see.

If we follow by semicolon, then we may see another answer (if there is one). We may repeat this response as often as desired.

?- female(X).

\( X = \) jackie ;

\( X = \) caroline ;

\( \text{no} \)
Builtin Predicates taking Predicate Arguments

Often we need to find all solutions to a goal query and collect them into a list. A special builtin predicate does this for us:

\[
\text{bagof( } \text{term, goal, answer })
\]

computes all terms of the form \text{term} which satisfy \text{goal} and collects them into the list \text{answer}.

Example:

\[
\text{bagof(X, father(john, X), Y)}
\]

collects into list \text{Y} the set of john's children.

\[
\text{bagof([X, Y], father(X, Y), Z)}
\]

collects into list \text{Z} the set of list of pairs \text{[X, Y]} such that \text{father(X, Y)} is solvable

A similar predicate which sorts the result list according to the universal order is called \text{setof}. 
Ergonomic uses of operators:

:- dynamic manages/2.
:- op(500, xfx, manages).
:- op(500, xfx, reports_to).
:- op(500, xfx, responsible_for).
:- op(400, fx, discharge).

john manages joe.
john manages jack.
john manages tim.
tim manages sally.
tim manages fred.
jack manages sue.
jack manages charles.

X reports_to Y :- Y manages X.

X responsible_for Y :- X manages Y.
X responsible_for Y :- X manages Z,
                     Z responsible_for Y.

discharge X :-
               retractall(X manages _),
               retractall(_ manages X).
Record construction and Extraction

The unification process is useful for building records and extracting their components. For example, we might have a rule:

overpaid(employee(Name, Salary, Level)) :-

\[ \text{Salary} > 10000 \times \text{Level}. \]

When we try to solve a goal such as

overpaid(employee(smith, 100000, 9))

the match which takes place instantiates the Name, Salary, and Level components of the record to smith, 100000, and 9 respectively.
A Prolog "procedure" is a collection of all clauses with a given head predicate.

The two interpretations of a Prolog procedure:

The logical interpretation: Each clause represents a logical implication. The collection of clauses, the heads of which match a given goal, define the only ways in which the goal may be true. For example, if the clauses for brother_in_law are:

```
brother_in_law(X, Y) :- brother(X, Z), spouse(Z, Y).
brother_in_law(X, Y) :- husband(X, Z), sister(Z, Y).
brother_in_law(X, Y) :- husband(X, Z), sister(Z, W), spouse(W, Y).
```

we mean

for arbitrary entities $X$, $Y$, $Z$, and $W$

```
if (brother(X, Z) and spouse(Z, Y)) then brother_in_law(X, Y)
and
if (husband(X, Z) and sister(Z, Y)) then brother_in_law(X, Y)
and
if (husband(X, Z), sister(Z, W), and spouse(W, Y))
then brother_in_law(X, Y)
```
The *procedural* interpretation:

To solve for a goal, try the first of the clauses having a head which matches the goal. Solve each of the corresponding body terms in turn. If this attempted solution fails, try the next relevant clause similarly, etc. If none of the clauses works, then the goal is unsolvable.

There are two ways in which information flows from a Prolog procedure:

Through the argument values

Through the *success* or *failure* of the procedure. Each time a procedure is called, it either succeeds or fails.

If it succeeds, then control proceeds as normal.

If it fails, then it causes Prolog to abandon the current calling sequence and seek an alternative within the next clause of the procedure. If there is no next clause, then the calling procedure also fails, etc.
A compiler for an Expression Language to a Stack Machine

Uses "Definite-Clause Grammar" notation

:- op(550, xfx, ':=').

comp((S;T)) --> {comp(S, CS, []), comp(T, CT, []), CS, CT}.

comp(A:=B) --> {atom(A)}, !, [addr(A)], comp(B), [store].

comp(A+B) --> !, {comp(A, CA, []), comp(B, CB, [])}, CA, CB, add_inst.

comp(A*B) --> !, {comp(A, CA, []), comp(B, CB, [])}, CA, CB, mult_inst.

comp(A) --> {atomic(A)}, !, [A], load_inst.

load_inst --> [load].
add_inst --> [add].
mult_inst --> [mult].
An Emulator for the Stack Machine

% calc(Instruction_List, Stack)
calc([], ).
calc([X, load | Y], Stack) :-
    integer(X), !,
    calc(Y, [X | Stack]).
calc([X, load | Y], Stack) :-
    calc(Y, [value_of(X) | Stack]).
calc([mult | X], [Y, Z | More]) :-
    compute(Y*Z, W),
    calc(X, [W | More]).
calc([add | X], [Y, Z | More]) :-
    compute(Y+Z, W),
    calc(X, [W | More]).
calc([addr(V) | X], Stack) :-
    calc(X, [addr(V) | Stack]).
calc([store | X], [Value, Addr | More]) :-
    write('store '), write(Value), write(' into '), write(Addr), nl,
    calc(X, More).
The AND-OR Tree View of Prolog Execution

Matching of a Goal with a Clause

\[ \text{head} :- \text{BT}_1, \text{BT}_2, \ldots, \text{BT}_n. \]

In general, several clauses may match a given goal

\[ \text{head} :- \text{BT}_{11}, \text{BT}_{12}, \ldots, \text{BT}_{1n}. \]
\[ \text{head} :- \text{BT}_{21}, \text{BT}_{22}, \ldots, \text{BT}_{2n}. \]
\[ \text{head} :- \text{BT}_{m1}, \text{BT}_{m2}, \ldots, \text{BT}_{mn}. \]
In general, several clauses may match a given goal

head :- BT11, BT12, ...., BT1n.
head :- BT21, BT22, ...., BT2n.

head :- BTm1, BTm2, ...., BTmn.
Backtracking

Backtracking occurs in any query where we ask to see multiple answers. In effect, we cause the answer just seen to act as if it were failure, and Prolog picks up its search where it left off.

To further see the power of backtracking, consider the following map problem:

*It is desired to color the regions from a set of colors on a map so that no two adjacent regions have the same color. Find a solution if one exists, or report that none exists.*

The simplest way to solve this problem is just to try various colorings until one is found to work, or until all have been tried. This is a complicated bookkeeping problem in a language without backtracking. In Prolog, it is simple and elegant.
Solving the map problem:

We express the map by using the regions as variables and a list of terms representing adjacencies. For example, the following map is expressed as shown:

\[ \text{map(R1, R2, R3, R4) :- adj(R1, R2), adj(R1, R3), adj(R1, R4), adj(R2, R3), adj(R2, R4), adj(R3, R4).} \]

We express the fact that each pair of adjacent regions must be colored, and must be colored by distinct colors, by:

\[ \text{adj(R, S) :- color(R), color(S), distinct(R, S).} \]

The predicate \textit{distinct} is true exactly when its arguments are not the same. In Prolog, we could define it using the built-in predicate \(\neq\) which tests that its arguments are not identical:

\[ \text{distinct(R, S) :- R \neq S.} \]

We need to give a the list of colors, e.g.

\[ \text{color(C) :- member(C, [red, blue, yellow]).} \]

Then we need only present the goal

\[ ?- \text{map(R1, R2, R3, R4).} \]

and Prolog does the rest. How? Backtracking is the key.
Prolog-in-Prolog Meta Interpreter

solve(true) :- !.
solve((Goal, Goals)) :- !, solve(Goal), solve(Goals).
solve(Goal) :- clause(Goal, Body), solve(Body).

Remember here that clause will in general return a Body with some substitutions already made, as occur in the matching of Goal.

Caution: The above does not work if an interpreted clause contains cut. It is difficult to handle this case with the tools in standard Prolog.

Meta Interpreter which gives Reasons

solve(true, []) :- !. % no reason needed
solve((Goal, Goals), [Reason | Reasons]) :- % conjunction
    !,
    solve(Goal, Reason),
    solve(Goals, Reasons).

solve(Goal, [(Goal :- Body) | Reasons]) :- % rule
    clause(Goal, Body),
    solve(Body, Reasons).
General Problems in running "dusty" Prolog code in Parallel

As with many languages,

Prolog code is typically written with sequentiality in mind;

The Prolog programmer often optimizes his code by ordering subgoals. A prime example is in the

"generate and test"

problem solving method

\[
\text{generate}(\text{Input, Candidate}), \quad \text{test}(\text{Candidate}).
\]

Logically, the two subgoals can be reversed or done in parallel, but in the Prolog execution model, there can be a variety of disasters:

- Arithmetic variables must be bound in Candidate before testing.

- test(Candidate) may be divergent if Candidate does not have a certain binding state, due to depth-first assumption.

- test(Candidate) may have an infinite number of solutions for Candidate
AND parallelism

Clause

\[ p(X) :\! -\! \ q(X),\ r(X). \]

Modes (generally dynamic)

X ground

\[ \begin{array}{c}
p(X) \\
q(X) \\
r(X)
\end{array} \]

solve both

Speculative, since if q(X) fails,

sequential execution would have been better
Wholesale AND Parallelism through Recursion

forall(Predicate, List) means the Predicate is true on each element of List

Clauses

forall(_, []).  
forall(Predicate, [A | X]) :- call(Predicate, A),  
forall(Predicate, X).

Usage

forall(p, [x1, x2, x3, ..., xn])  

can spawn parallel executions of  
p(x1), p(x2), p(x3), ..., p(xn).

Similar possibilities hold for predicates such as map:

map(Predicate, List1, List2) means that Predicate is called pairwise on elements of List1 and List2 (usually the predicate is functional).
AND parallelism (cont'd)

Clause

\[ p(X) \leftarrow q(X), r(X). \]

\( X \) not ground

\begin{align*}
\text{on success,} \\
\text{binding can "flow" forward}
\end{align*}

If multiple solutions for \( p(X) \) are desired, this flow yields "pipelining".

In principle, \( q(X) \) and \( r(X) \) could go in parallel above, but then there would be the problem of reconciling bindings.
AND parallelism (conclusion)

Knowledge (by the compiler or execution system) of two aspects of the data are essential:

Knowing whether terms are ground

Knowing whether there are any logical variables shared between terms.

Without these, the overhead to get correct execution may be too high.
OR parallelism

Clauses

\[ p(X) :- \text{body1.} \]
\[ p(X) :- \text{body2.} \]

Cases of Interest:

Find all solutions to \( p(X) \):

- solve both
- bindings aggregated in a list

\[ \text{body1} \quad \text{body2} \]
OR parallelism (cont'd)

Clauses

\[ p(X) : \text{body1}. \]
\[ p(X) : \text{body2}. \]

Cases of Interest:

Find the first solution to \( p(X) \):

solve both \hspace{1cm} \text{body1} \hspace{1cm} \text{body2}

As above, but must synchronize bindings.

Each body may itself have multiple solutions, so the overall parallelism is an unlimited "tree" of execution.

Additional parallelism possible in Anticipation of need for additional solutions.
OR parallelism (cont'd)

Clauses

\[ p(X) :- \text{body1}. \]
\[ p(X) :- \text{body2}. \]

Cases of Interest:

Find a solution to \( p(X) \):

- solve both
- \( \text{body1} \)
- \( \text{body2} \)

Truly non-deterministic result.
Speculative
Super-linear speedup possibilities

Proper Prolog semantics will have uses for the first two modes, but not the third, which would require a special directive to avoid incorrectness.
Implementation Problems for OR parallelism:

- Maintaining large numbers of separate variables (for binding in parallel) while sharing bindings wherever possible (to economize space).

- Synchronization in parallel OR branches, when side-effects are involved.
Aurora

Aurora - a prototype Prolog system exploiting
OR-parallelism

"Workers" explore the Prolog search tree in OR-parallel

• the "engine"
• the "scheduler"

The Aurora implementation environment:

• Engine-scheduler interface
• Scheduler test harness
• Instrumentation
Conclusions from Aurora

Engine overhead due to SRI model and scheduler hooks: 15-35%

This overhead defines breakeven with sequential systems

Speedups Measured under Aurora:

<table>
<thead>
<tr>
<th>Example</th>
<th>speedup for 3 workers</th>
<th>speedup for 5 workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse5</td>
<td>(2.83)</td>
<td>(4.08)</td>
</tr>
<tr>
<td>8-queens2</td>
<td>(2.97)</td>
<td>(4.88)</td>
</tr>
<tr>
<td>salt&amp;must</td>
<td>(2.87)</td>
<td>(4.82)</td>
</tr>
<tr>
<td>parse3*20</td>
<td>(2.09)</td>
<td>(2.30)</td>
</tr>
<tr>
<td>farmer*100</td>
<td>(1.63)</td>
<td>(1.69)</td>
</tr>
</tbody>
</table>

Speedups measured on a six processor Sequent Balance
Selected References on Parallelism in Prolog

A. Ciepielewski and S. Haridi. A formal model for the OR-parallel execution of logic programs. IFIP '83, North-Holland. keywords: OR-parallelism


Knowledge-Server Viewpoint

?- ancestor(X, john).
X = mary ;
X = tom ;
X = susan

Schematic View

Prolog

Knowledge Base

Stream of answers

Query input

Simplified "Bus-Oriented "Representation

Prolog

Quintus Computer Systems, 1988
Exploiting the "Knowledge Net"

Application #1

Application #2

Prolog

Application #1

Application #2

Knowledge Server A

Knowledge Server B

Knowledge Server C