Practical Issues Relating to the Internal Database Predicates in an OR-Parallel Prolog: Extensions and Useful Hacks

by

Mats Carlsson, Ken Danhof, and Ross Overbeek

Mathematics and Computer Science Division

February 1988

*Permanent Address: Swedish Institute of Computer Science, Box 1263, S-163 13 Spanga, Sweden.

**Permanent Address: Department of Computer Science, Southern Illinois University, Carbondale, IL 62901.

This work was supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.
A major purpose of the Technical Information Center is to provide the broadest dissemination possible of information contained in DOE's Research and Development Reports to business, industry, the academic community, and federal, state and local governments.

Although a small portion of this report is not reproducible, it is being made available to expedite the availability of information on the research discussed herein.
Contents

Abstract..........................................................................................................................1

1. Introduction..................................................................................................................1

2. Implementation of an Internal Database ....................................................................2

3. Separating Sets from Keyed Access in the Multiprocessing Environment .................4

4. Implementing findall/3 in an OR-Parallel Environment .............................................5

5. Implementing AND-Parallel Execution......................................................................6

6. Implementing Pipes and Using them to Support and/2 and set/3.................................7

7. A More General Implementation of Pipes ..................................................................9

8. A Parallelized Compiler...............................................................................................11

9. Summary....................................................................................................................13

References ......................................................................................................................14
Practical Issues Relating to the Internal Database Predicates in an OR-Parallel Prolog: Extensions and Useful Hacks

by

Mats Carlsson, Ken Danhof, and Ross Overbeek

ABSTRACT

This technical report is being written to document and explain some of the insights the authors have gained during the implementation of two OR-parallel Prolog systems for shared-memory multiprocessors. We have been quite amazed by how much power and functionality can be achieved through the implementation of just a few primitive predicates. We introduce a few relatively trivial predicates which allow one to implement versions of findall/3, bagof/3, and setof/3 for the a multiprocessor environment. Then we show how these predicates can be used to implement AND-parallelism. Finally, we illustrate their use for implementing a limited notion of streams, and how such streams can be used to create a compiler that achieves excellent speedups for multiprocessors with a limited number of processors.

1. Introduction

Our goal in writing this report is to document some of the insights gained during work with two OR-parallel Prolog systems. Both Prologs were cooperatively developed; they have been referred to in the literature as ANLWAM and Gigalips Prolog. They represent the early fruits of a cooperation called the Gigalips Project. We believe that some of the detailed implementation techniques that were utilized will be of interest to others in the research community.

We begin by offering a brief description of the implementation of the predicates that manage the internal database: recorda/3, recordz/3, recorded/3, erase/1, and instance/2. These predicates are based on the notion of keyed access to sets of formulas. We have found it useful to separate the notion of "keyed access" from the set of operations that manipulate the associated "set of formulas". Hence, we introduce two additional predicates initialize_fs/1 and delete_fs/1, which can be used to create an empty set of formulas and delete a set (which may or may not be empty). We then discuss the extensions required to support concurrent manipulation of a set by different processes during OR-parallel execution. Essentially, we advocate the straightforward approach of adding lock_fs/1 and unlock_fs/1, predicates that allow a process to create a critical region.

Once these basic facilities are established, it becomes straightforward to produce a version of findall/3 that behaves as desired in an OR-parallel environment. A similar approach works for bagof/3 and setof/3, although we do not present the details here.

Once we have a version of findall/3 that works as desired, it becomes possible to implement AND-parallelism in a way that gives fairly good performance for many situations. Thus, we define and/2 in just a few lines of Prolog. We offer this straightforward implementation, not because it addresses all of the issues central to so much current research, but because it is so simple.
One can conveniently view alternative branches in an OR-parallel execution as "independent environments". Once we have facilities for sharing sets between such environments, we have created facilities for a very limited form of communication between them. It becomes possible to create "pipes" that link these environments, allowing the implementation of a crude version of stream parallelism. These ideas can be exploited to create an implementation of AND-parallelism in which execution is not delayed until all solutions of each goal are computed. Then, we illustrate the use of pipes to produce a predicate, \texttt{seti3}, which behaves like \texttt{setofi3} in cases in which all uninstantiated variables in the \texttt{Goal} also occur in the \texttt{Template}. We use \texttt{seti3} to show how the sorting phase can be overlapped with the search for solutions.

Finally, we illustrate the use of these constructs in the creation of a compiler that effectively exploits the parallelism offered in an OR-parallel Prolog. The initial tests that we have conducted indicate that the overhead introduced by these primitives is quite tolerable.

2. Implementation of an Internal Database

The internal database offers keyed access to terms. Essentially, it is necessary to create builtin routines capable of providing roughly the following functionality:

1. It is necessary to be able to associate a "set" with a "key". Here, we are using the term "set" to mean an "ordered list", which is somewhat nonstandard; however, we do so to emphasize that we are not talking about a Prolog list. To support this keyed access to a set, two builtin predicates were created. \texttt{\$user\_data(Key,Set)} can be invoked with \texttt{Key} bound to a key, and \texttt{Set} an uninstantiated variable. It has the effect of either locating an existing set with the designated key or creating an empty set with the designated key. The second builtin, \texttt{\$current\_user\_data(Key,Set)} can be invoked with \texttt{Key} bound to a key, and \texttt{Set} an uninstantiated variable. In this case, the builtin will simply fail, if no set with the designated key exists. It is also worth noting that, for functionality not relevant to this discussion, this second builtin can be invoked with \texttt{Key} as an uninstantiated variable; in this case, it represents a nondeterminate invocation in which \texttt{Set} gets bound successively to the existing sets.

2. Besides the capability of creating and accessing "sets", it is necessary to be able to insert new elements into a set. Actually, what gets inserted into the set is a reference to some external representation of a term. Insertion is accomplished using either \texttt{\$inserta(Set,Ref\_Term,Ref\_Set\_Element)} or \texttt{\$insertz(Set,Ref\_Term,Ref\_Set\_Element)}. Here \texttt{Set} and \texttt{Ref\_Term} must be instantiated (\texttt{Ref\_Term} is the reference to the external representation of the term). \texttt{Ref\_Set\_Element} is an uninstantiated variable that gets bound to a "set reference"; that is, it represents an element in a set, which must be carefully distinguished from a reference to the external representation of a term.

3. We also need to be able to create an "external representation of a term". There are two approaches to supporting this functionality. In ANLWAM[5], we implemented the ability to allocate memory and to copy a term into this newly-acquired chunk of memory (in this case, \texttt{Ref\_Term} would represent the address of the copy). In SICStus[2], the external representation is created by allocating memory and compiling the term into WAM code that is placed in the chunk of memory. Thus, the term is represented by a WAM routine that takes as input a single argument and unifies that argument with the term. This is a rather significant point that we will discuss in detail below. In any event, \texttt{\$compile\_term(Term,Ref\_Term)} is used to create a \texttt{Ref\_Term}, which corresponds to the "external representation of the term".

4. The ability to delete an element from a set is provided by \texttt{erase(Ref\_Set\_Element)}, which deallocates both the external representation of the term and the "set element" representing the position of the term in the set.

5. To move through the elements in a set, one uses \texttt{\$current\_node(Set,Ref\_Set\_Element)}, which must be called with the first argument bound to a set and the second an uninstantiated variable. This builtin will succeed once for every element in the set.

6. Finally, some means of extracting a copy of a term, given its \texttt{Ref\_Set\_Element} is required. This is supported via \texttt{instance(Ref\_Set\_Element,Term)}. In this case, \texttt{Ref\_Set\_Element} must be bound to a set element, and \texttt{Term} is unified with the term that was saved in the set. In SICStus
Prolog, as well as the Gigalips Prolog, the functionality of instance/2 is achieved by simply invoking the WAM routine corresponding to Ref_Set_Element (that is, the second argument register is properly set to reference Term and the WAM P register is set to the address of the routine saved in Ref_Set_Element).

It is important that the reader grasp this notion of representing a term by compiled WAM code and how the compiled code is utilized. To illustrate, suppose that we wish to insert the term $f(a f(X, X))$ into the set $S$. Then, the term could be compiled into the WAM code

```
get_structure f/2, A1
unify_constant a
unify_variable A1
get_structure f/2, A1
unify_variable A1
unify_value A1
proceed
```

Note that the algorithm for producing this compiled code is quite similar to the algorithm for copying the term, and that the compiled code is of similar size (that is, it may actually be as much as twice the size, but it is not exorbitantly larger). Then, the implementation of instance(Ref, Term) involves resetting the P register of the WAM to reference this code (the address of the code is accessible via Ref); register A1 will reference Term.

The choice to represent terms as compiled WAM routines is a direct consequence of the insight that it costs roughly the same overhead to copy a term into external memory as it does to compile it into the WAM code required to unify the term with an incoming argument. This basic idea has been reported elsewhere in the literature[4, 1] We found it necessary to support the unit-clause compiler in C, in order to support the claim that copying and compiling represent comparable costs. While this technique might seem a bit non-intuitive, it should be noted that we have found it extremely important in the implementation of applications involving the maintenance of a substantial database (e.g., in the automated deduction systems being constructed at Argonne National Laboratory); the performance advantage offered by rapidly compiled unification routines can be significant.

Once the functionality discussed above has been implemented, it is straightforward to implement the recorda/3, recorded/3, and recorded/3 predicates:

```
recorda(Key, Term, Ref) :-
  'Scompile_term' (Term, Ref_Term),
  'Suser_data' (Key, Set),
  'Sinserta' (Set, Ref_Term, Ref).

recordz(Key, Term, Ref) :-
  'Scompile_term' (Term, Ref_Term),
  'Suser_data' (Key, Set),
  'Sinsertz' (Set, Ref_Term, Ref).

recorded(Key, Term, Ref) :-
  nonvar(Key),
  'Scurrent_user_data' (Key, Set),
  'Scurrent_node' (Set, Ref),
  instance(Ref, Term).
```
3. Separating Sets from Keyed Access in the Multiprocessing Environment

There are two distinct ways to think about what is meant when members of the Gigalips Project discuss OR-parallel Prolog:

1. The most natural way, perhaps, is to state that an OR-parallel environment simply supports Prolog semantics, occasionally exploring the search tree in parallel. From this point of view, the Prolog semantics requires that certain builtin predicates with side-effects have the property that they "suspend" until they occur on the leftmost remaining branch of the search tree. In this case, one must carefully define which predicates suspend and which do not.

2. The other view, which was taken in the implementation of ANLWAM, is that predicates are defined as either "sequential" or "parallel". The clauses in a sequential predicate are guaranteed to be explored sequentially. That is, no clause is utilized until all paths generated by an earlier clause in the predicate have been completely explored. The clauses in a parallel predicate may be explored simultaneously.

While there are definite advantages to each viewpoint, this document is being written from the second point of view. Further, we will assume that each predicate that we define is sequential, unless explicitly stated otherwise.

In this document, we do not describe how to support versions of the builtin predicates that can manipulate the internal database from simultaneously executing branches in an OR-parallel environment. We assume that OR-parallel execution will be circumscribed to sections that do not alter the internal database (although we certainly can envision instances in which loosening such a restriction might be desirable). Given this assumption, there is very little alteration required to support the internal database predicates. Indeed, the major change is simply to alter the routines that allocate and deallocate memory to introduce the required critical sections (these changes are required due to the fact that a number of builtin predicates allocate/deallocate memory). Those changes that would normally be proposed to support simultaneous use of predicates that alter the internal database are simply not required, assuming that predicates that introduce side-effects suspend until they become "leftmost" in the execution tree. However, there is definitely a need for some new capabilities in order to support a number of useful operations appropriate to OR-parallel execution.

One basic enhancement required to effectively utilize the potentials of the OR-parallel environment centers on the ability to create sets (here, we use the term "set" in the same sense as in the above discussion) and to allow simultaneous manipulation of these sets. To accomplish this, it is useful to separate the notion of "set" from the notion of "keyed access to a set". Thus, we introduce the ability to create an initialized (empty) set, to delete a set, and to synchronize operations made against a set. The synchronization operations assume that each set has associated with it a "lock". The predicates required to support these facilities are as follows:

**initialize_fs(Set)**

This builtin predicate requires that the argument be an uninstantiated variable, which gets bound to an initialized set. The initial status of the set is "unlocked".

**delete_fs(Set)**

This builtin predicate requires that its argument be a set. It deallocates the set (including the data structure representing the set, the data structures representing elements of the set, and the associated compiled code for the terms included in the set).

**lock_fs(Set)**

This builtin predicate requires that its argument be a set. In that case, it always succeeds, but it may halt continued progress in the Prolog computation until the set attains the status of "unlocked". It has the effect of leaving the set with the status of "locked".
unlock_fs(Set)
This builtin predicate requires that its argument be a set. In that case, it always succeeds, leaving the set with a status of "unlocked".

Using these primitives, we define the following useful predicates:

% store Term as the first element of Set, returning Ref
store_fs_a(Set,Term,Ref) :-
   '$compile_term' (Term, Ref_Term),
   lock_fs(Set),
   'sinserta' (Set, Ref_Term, Ref),
   unlock_fs (Set).

% store Term as the last element of Set, returning Ref
store_fs_z(Set,Term,Ref) :-
   '$compile_term' (Term, Ref_Term),
   lock_fs (Set),
   'sinsertz' (Set, Ref_Term, Ref),
   unlock_fs (Set).

% retrieve Term (and its Ref) from Set
retrieve_fs(Set,Term,Ref) :-
   'scurrent_node' (Set,Ref),
   instance(Ref,Term).

% pop Term off the front of Set
extract_fs(Set,Term) :-
   (lock_fs(Set); % lock before extracting
     unlock_fs(Set), fail),
   retrieve_fs (Set,Term,Ref),
   erase(Ref),
   (unlock_fs(Set); % unlock after extracting
     lock_fs(Set), fail). % lock at backtracking

These rather straightforward operations allow a surprising amount of utility, as we shall try to establish in the following sections.

4. Implementing findall/3 in an OR-Parallel Environment
Clocksin and Mellish[3] state that

findall(X,G,L) "constructs a list L consisting of all of the objects X such that the goal G is satisfied."
Since findall/3 does have an understood behavior within the Prolog community (of producing the solutions in a determined sequence), it is arguable that a distinct predicate should be defined for use in the OR-parallel environment (in which the solutions are allowed to be collected in any order). We chose not to define a distinct predicate, but the decision was quite arbitrary. In any event, a version that works in the parallel environment might be defined as follows:

findall(X,G,L) :
   initialize_fs(Solutions),
   collect_solutions(X,G,Solutions),
   construct_list(Solutions,L),
   delete_fs(Solutions).
collect_solutions(X,G,Solutions) :-
  call(G),
  store_fs_z(Solutions,X,_),
  fail.
collect_solutions(_,_,_).

construct_list(Solutions,L) :-
  retrieve_fs(Solutions,X,Ref) ->
    L = [X|T], erase(Ref), construct_list(Solutions,T);
    L = [].

Here, some explanation is required to explain why this works in the OR-parallel environment offered by Gigalips Prolog. Since, collect_solutions/3 is a sequential predicate, its second clause will not be considered until all branches created by its first clause have been explored. Thus, even though the invocation of G might result in parallel execution, with store_fs/3 being invoked from parallel branches, all such branches will have been explored before collect_solutions/3 succeeds.

Finally, before leaving this section, we must point out the pitfalls of employing our version of findall/3 in an unrestricted OR-parallel environment. One central issue involves behavior when an invocation of findall/3 is interrupted due to a cut or commit. In these cases, there will be memory that gets "lost" unless some means of garbage collecting is used that can identify sets which are no longer referenced. Such issues are beyond the scope of this document. We assume within this document that the programmer using our hacks will avoid using sets within a context in which such interruptions are possible.

5. Implementing AND-Parallel Execution

Once we have a version of findall/3 that behaves as desired, it becomes straightforward to produce a version of and/2 that offers and-parallel execution of two goals:

and(G1,G2) :-
  findall(X,get_solution(G1,G2,X),Solutions),
  member(left(G1),Solutions),
  member(right(G2),Solutions).

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

:- parallel get_solution/3.
get_solution(G1,_left(G1)) :- call(G1).
get_solution(_,G2,right(G2)) :- call(G2).

Note that we declare the predicate get_solution/3 to be "parallel", which means that both clauses in the predicate may be explored simultaneously. The effect of this declaration is to allow the collection of all solutions to G1 to be computed simultaneously with the computation of all solutions of G2. The solutions are collected in Solutions. Once all of the solutions to both goals have been computed, the two invocations of member/2 are used to pick pairs of solutions, unifying them with the original G1 and G2 (which forces the solutions to be "compatible"). There are at least two drawbacks to this simple implementation of and/2:

a) The solutions are copied into external memory via the findall/3.
b) Access to the solutions does not occur until all of the solutions to each goal have been computed.

However, it should be noted that, when the two goals are determinate and the output created by each goal is relatively small, performance is nearly optimal. We believe that this happens to be the case in many situations.
There is one more point worth discussing. This version of \texttt{and/2} behaves "sequentially", in that it
does not result in multiple solutions of \texttt{and/2} being explored in parallel (although solutions to the two goals
are generated in parallel). This could easily be changed, as follows:

\begin{verbatim}
andP(G1,G2) :-
    findall(X, get_solution(G1,G2,X), Solutions),
    memberP(left(G1),Solutions),
    memberP(right(G2),Solutions).

:- parallel memberP/2.
memberP(X,[X1|_]) :- memberP(X,T).
memberP(X,[_|T]) :- memberP(X,T).
\end{verbatim}

6. Implementing Pipes and Using them to Support \texttt{and/2} and \texttt{set/3}

To this point, we have made some rather minimal assumptions about the implementation of OR-
parallelism. In particular, we chose to think of predicates as either sequential or parallel. The techniques
we are going to illustrate in the remainder of this report make somewhat more restrictive (and question-
able) assumptions. As we consider the implementation of "pipes" to link two OR-parallel environments,
we will allow the "receiving" environment (i.e., the one consuming elements from a producer) to "spin"
waiting for elements to process. This can be done only on the assumptions that

a) when multiple clauses in a predicate can be explored in parallel, they are initiated in order (i.e.,
   if a clause becomes active, then any preceding clauses have already become active), and
b) if a process begins processing a clause in a parallel predicate, then the process will continue to
   explore that subtree until it is completed.

Weaker assumptions are tenable, if one wishes to exploit particular features of a specific dispatching algo-
rithm. These assumptions are adequate for us to demonstrate the basic techniques that we have imple-
mented.

To implement pipes, we will use sets in which one process inserts elements and another removes
them. We think of these as "streams" and build three short routines to implement the concept:

\begin{verbatim}
write_to_stream(Set,Term) :-
    store_fs_z(Set,elt(Term),_).
write_end_of_stream(Set) :-
    store_fs_z(Set,end,_).
read_from_stream(Set,Term) :-
    repeat,
    extract_fs(Set,T) ->
    read_fromstream(T,Set,Term).
read_from_streamP(Set,Term) :-
    repeat,
    extract_fs(Set,T) ->
    read_fromstreamP(T,Set,Term).
read_from_stream(end,Set,Term) :-
    write_end_of_stream(Set),
    fail.
read_from_stream(elt(Term),Set,Term) :-
read_from_stream(elt(_),Set,Term) :-
\end{verbatim}
The predicate `read from streamP/2` is provided to allow the user to cause multiple values from the pipe to cause a branch in the OR-parallel search tree (i.e., `read from streamP/2` is the "parallel" version of `read from stream`).

In the following version of `andP/2`, three parallel execution environments are established. The first two compute solutions of the goals to be executed AND-parallel, while the third chooses pairs of goals from those produced from the two preceding branches. This third branch spins, waiting for solutions to appear in the set used to communicate between the three branches.

```
new_andP(G1,G2) :-
    initialize_fs(Solutions),
    (and_body(G1,G2,Solutions); delete_fs(Solutions), fail).
```

```
:- parallel and_body/3.
and_body(G1,,Solutions) :- generate_and_solutions(G1,left(G1),Solutions), fail.
and_body(_,G2,Solutions) :- generate_and_solutions(G2,right(G2),Solutions), fail.
and_body(G1,G2,Solutions) :- pair_solutions(G1,[],G2,[],Solutions,0).
```

```
generate_and_solutions(Goal,Template,Solutions) :-
call(Goal),
write_to_stream(Solutions,Template).
generate_and_solutions(_,_,Solutions) :-
write_to_stream(Solutions,end_one_goal).
```

```
pair_solutions(G1,Glsofar,G2,G2sofar,Solutions,C) :-
read_from_stream(Solutions,Term) ->
pair_solutions(Term,G1,Glsofar,G2,G2sofar,Solutions,C).
```

```
% remove declaration and replace memberP by member
% for a sequential version
:- parallel pair_solutions/7.
pair_solutions(end_one_goal,G1,Glsofar,G2,G2sofar,Solutions,0) :-
pair_solutions(G1,Glsofar,G2,G2sofar,Solutions,1).
pair_solutions(left(G1),G1,Glsofar,G2,G2sofar,_,_ioms,1) :-
memberP(G2,G2sofar).
pair_solutions(left(Term),G1,Glsofar,G2,G2sofar,Solutions,C) :-
pair_solutions(G1,[Term|Glsofar],G2,G2sofar,Solutions,C).
pair_solutions(right(G2),G1,Glsofar,G2,_,_ioms,C) :-
memberP(G1,Glsofar).
pair_solutions(right(Term),G1,Glsofar,G2,G2sofar,Solutions,C) :-
pair_solutions(G1,Glsofar,G2,[Term|G2sofar],Solutions,C).
```

While substantially more complex than our earlier version of `andP/2`, this version does allow the utilization of solutions to begin before all of the solutions to each goal have been computed (which is important only for nondeterminate goals executed AND-parallel).

We can now cover the use of a pipe to create `set3/3`, a predicate much like `setof/3` in cases in which all uninstantiated variables in the `Goal` also occur in the `Template`. Our goal in the implementation of `set3` is to show how the sorting of solutions can be overlapped with the search for solutions. Here is the code for `set3/3`:

```
:- parallel read from streamP/3.
read_from_streamP(end,Set,_) :-
    write_end_of_stream(Set),
fail.
read_from_streamP(elt(Term),Set,Term) :-
    read_from_streamP(Set,Term).
```

```
The predicate `read from streamP/2` is provided to allow the user to cause multiple values from the pipe to cause a branch in the OR-parallel search tree (i.e., `read from streamP/2` is the "parallel" version of `read from stream`).
```

```
In the following version of `andP/2`, three parallel execution environments are established. The first two compute solutions of the goals to be executed AND-parallel, while the third chooses pairs of goals from those produced from the two preceding branches. This third branch spins, waiting for solutions to appear in the set used to communicate between the three branches.
```

```
new_andP(G1,G2) :-
    initialize_fs(Solutions),
    (and_body(G1,G2,Solutions); delete_fs(Solutions), fail).
```

```
:- parallel and_body/3.
and_body(G1,,Solutions) :- generate_and_solutions(G1,left(G1),Solutions), fail.
and_body(_,G2,Solutions) :- generate_and_solutions(G2,right(G2),Solutions), fail.
and_body(G1,G2,Solutions) :- pair_solutions(G1,[],G2,[],Solutions,0).
```

```
generate_and_solutions(Goal,Template,Solutions) :-
call(Goal),
write_to_stream(Solutions,Template).
generate_and_solutions(_,_,Solutions) :-
write_to_stream(Solutions,end_one_goal).
```

```
pair_solutions(G1,Glsofar,G2,G2sofar,Solutions,C) :-
read_from_stream(Solutions,Term) ->
pair_solutions(Term,G1,Glsofar,G2,G2sofar,Solutions,C).
```

```
% remove declaration and replace memberP by member
% for a sequential version
:- parallel pair_solutions/7.
pair_solutions(end_one_goal,G1,Glsofar,G2,G2sofar,Solutions,0) :-
pair_solutions(G1,Glsofar,G2,G2sofar,Solutions,1).
pair_solutions(left(G1),G1,Glsofar,G2,G2sofar,_,_ioms,1) :-
memberP(G2,G2sofar).
pair_solutions(left(Term),G1,Glsofar,G2,G2sofar,Solutions,C) :-
pair_solutions(G1,[Term|Glsofar],G2,G2sofar,Solutions,C).
pair_solutions(right(G2),G1,Glsofar,G2,_,_ioms,C) :-
memberP(G1,Glsofar).
pair_solutions(right(Term),G1,Glsofar,G2,G2sofar,Solutions,C) :-
pair_solutions(G1,Glsofar,G2,[Term|G2sofar],Solutions,C).
```

While substantially more complex than our earlier version of `andP/2`, this version does allow the utilization of solutions to begin before all of the solutions to each goal have been computed (which is important only for nondeterminate goals executed AND-parallel).

We can now cover the use of a pipe to create `set3/3`, a predicate much like `setof/3` in cases in which all uninstantiated variables in the `Goal` also occur in the `Template`. Our goal in the implementation of `set3` is to show how the sorting of solutions can be overlapped with the search for solutions. Here is the code for `set3/3`:
set (X, G, L) :-
    initialize_fs(Solutions),
    set_body(X, G, L, Solutions),
    delete_fs(Solutions).

:- parallel set_body/4.
set_body(X, G, _, Solutions) :- generate_set_solutions(G, X, Solutions), fail.
set_body(_, _, L, Solutions) :- collect_and_sort(Solutions, [], L).

collect_and_sort(Solutions, SoFar, L) :-
    read_fromstream(Solutions, X) ->
        insert_into_set(SoFar, X, SoFar1),
        collect_and_sort(Solutions, SoFar1, L)
    ;
    (L = SoFar).

insert_into_set([], X, [X]).
insert_into_set([H|T], X, [X,H|T]) :- H @> X.
insert_into_set([H|T], X, [H|T2]) :- H @< X, insert_into_set(T, X, T2).
insert_into_set([H|T], X, [H|T]) :- H == X.

generate_set_solutions(Goal, Template, Solutions) :-
    call(Goal),
    write_to_stream(Solutions, Template).
generate_set_solutions(_, _, Solutions) :-
    write_end_of_stream(Solutions).

7. A More General Implementation of Pipes

The ideas introduced in the previous section can be carried a bit further. Indeed, it is possible to produce a general framework for constructing pipes between a list of goals. In this section, we will present this framework in detail. In the next section we will use these techniques to demonstrate the structure of a compiler that achieves near optimal performance (in the sense of utilizing multiple processes effectively).

We think of a pipe as composed of a set from which input terms are extracted, a set to which final output terms are inserted, and a sequence of stages which produce the desired output from the input elements. Each stage reads its input from a set and produces an output set. Conceptually, all stages of the pipe may be thought of as executing in parallel, although in practice they might just be executed in a left-to-right fashion. Hence, if one envisions a 3-stage pipe, there would be two intermediate sets established to handle communication of elements between stages. We begin our discussion by implementing this notion of a pipe using the predicate pipe/3:

pipe(In, Goals, Out) :-
    set_up_stages(In, Goals, Out, Stage_List),
    run_stages(Stage_List),
    clean_up_intermediates(Stage_List).

Here Goals is a list of elements, each of which has the form [Goal, Term_In, Term_Out], where

1. Goal is a normal Prolog goal which can be used to process an input element (Term_In), producing 0 or more output elements (via nondeterminant instantiation of Term_Out).
2. Term_In is an uninstantiated variable which appears as an argument of Goal, and
3. Term_Out is an uninstantiated variable which appears as an argument of Goal.
Thus,

\[
\text{pipe}(\text{Set}_\text{In}, [[p_1(\text{Tin}_1, \text{Tout}_1), \text{Tin}_1, \text{Tout}_1], \\
[p_2(\text{Tin}_2, \text{Tout}_2), \text{Tin}_2, \text{Tout}_2], \\
[p_3(\text{Tin}_3, \text{Tout}_3), \text{Tin}_3, \text{Tout}_3]], \\
\text{Set}_\text{Out})
\]

represents a pipe used to process the elements in \(\text{Set}_\text{In}\), producing \(\text{Set}_\text{Out}\) as the output. Elements go through three stages in this pipe.

The actual code to implement a simple version of \text{pipe}/3 is as follows:

\[
\begin{align*}
\text{setupstages} & (\text{In}, \text{[Goal]}, \text{Out}, \text{[stage}(\text{In}, \text{Goal}, \text{Out})]) \\
\text{set_up_stages} & (\text{In}, \text{[Goal|Goals]}, \text{Out}, \text{[stage}(\text{In}, \text{Goal}, \text{Inter})|\text{Stages}]) :- \\
& \text{set_up_stages}(\text{Inter}, \text{Goals}, \text{Out}, \text{Stages}), \\
& \text{initialize_fs}(\text{Inter}). \\
\text{run_stages} & (\text{Stage}_\text{List}) :- \text{run_stages}_1(\text{Stage}_\text{List}). \\
\text{run_stages} & (\_). \\
& :- \text{parallel run_stages}_1/1. \\
\text{run_stages}_1 & ([\text{Stage}]_\text{List}) :- \text{call}(\text{Stage}), \text{fail}. \\
\text{run_stages}_1 & ([\_], \_1\text{Stage}_\text{List}) :- \text{run_stages}_1(\text{Stage}_\text{List}). \\
\text{stage} & (\text{In}, \text{Goal}, \text{Out}) :- \text{stage_body}(\text{In}, \text{Goal}, \text{Out}). \\
\text{stage} & (\_, \_\_, \text{Out}) :- \text{write_end_of_stream}(\text{Out}). \\
\text{stage_body} & (\text{In}, \text{[G, T_in, T_out]}, \text{Out}) :- \\
& \text{read_from_streamP}(\text{In}, \text{T_in}), \\
& \text{call}(\text{G}), \\
& \text{writeto_stream}(\text{Out}, \text{T_out}), \\
& \text{fail}. \\
\text{clean_up_intermediates} & ([\_]). \\
\text{clean_up_intermediates} & ([\text{stage}(\_, \_, \text{Intermediate})|\text{Stages}]) :- \\
& \text{clean_up_intermediates}(\text{Stages}), \\
& \text{delete_fs}(\text{Intermediate}).
\end{align*}
\]

The reader should note the use of \text{read_from_streamP}/2 in the definition of \text{stage_body}/3. This use of the parallel version of the predicate causes the elements to be processed simultaneously and makes the order of elements flowing through the pipe somewhat unpredictable.

What we have presented above allows one to construct pipes in which the input is taken from a set and the output is put into a set. In some cases, however, one wishes the first stage in a pipe to take input from some other source (e.g., the compiler's first stage would read clauses from a file) or to return the output of the pipe in some other form. In this case, we wish all of the internal stages of the pipe to communicate via sets, but the initial and terminating stages must be handled in a different fashion. Hence, we introduce the following predicate:

\[
\begin{align*}
\text{pipe} & (\text{First}_\text{Stage}, \text{In}, \text{Goals}, \text{Out}, \text{Last}_\text{Stage}) :- \\
& \text{initialize_fs}(\text{In}), \\
& \text{initialize_fs}(\text{Out}), \\
& \text{set_up_stages}(\text{In}, \text{Goals}, \text{Out}, \text{Stage}_\text{List}), \\
& \text{run_stages}(\text{run_stages}([\text{First}_\text{Stage}|\text{Stage}_\text{List}]), \text{Last}_\text{Stage}), \\
& \text{delete_fs}(\text{In}), \\
& \text{delete_fs}(\text{Out}), \\
& \text{clean_up_intermediates}(\text{Stage}_\text{List}).
\end{align*}
\]
It might seem quite odd to utilize the \texttt{run\_stages/2} predicate, rather than adding \texttt{Last\_Stage} to the end of the list of goals passed to \texttt{run\_stages/1}. However, \texttt{run\_stages/1} is "failure driven"; since, \texttt{Last\_Stage} frequently constructs a data structure that would disappear due to backtracking, the approach given (in which \texttt{run\_stages/2} succeeds) must be used. It is also worth noting that, while \texttt{run\_stages/1} must be sequential, \texttt{run\_stages/2} can be (and should be) parallel; in the case of \texttt{run\_stages/2}, synchronization is achieved by passing terms through a "stream". This predicate, \texttt{pipe/5}, will turn out to be exactly what is required to implement an in-core, pipelined compiler.

\section*{8. A Parallelized Compiler}

It is possible to now create a parallelized compiler. The compiler is composed of two basic parts:

1. a pipe that reads input clauses from a file, compiles them into WAM object code (in memory), and produces a sorted list of indexing entries (each entry contains the procedure, clause number, and a reference to the compiled code for the clause) and

2. a second part that compiles the indexing for the procedures.

\begin{verbatim}
pcompile(File) :-
    statistics(runtime, _),
    pcompile_body(File, Indexed_Procedures),
    statistics(runtime, [_ , Time]),
    write('time: '), write(Time), nl,
    write(' compiled procedures: '), write(Indexed_Procedures), nl.

/* Compiling breaks into two main sections: create_sorted_index/2 reads from the input file, does clause compilation, and produces a sorted list of indexing entries (Sorted\_Index). This list contains one entry per compiled clause. generate_indexed_procedures/2 creates the indexing required for the procedures. */
pcompile_body(File, Indexed_Procedures) :-
    create_sorted_index(File, Sorted\_Index),
    generate_indexed_procedures(Sorted\_Index, Indexed_Procedures).

/* create_sorted_index is a 3-stage pipe. The first stage is defined by read\_clauses/2, which reads input clauses and puts them into the set Input\_Clauses. The second stage performs clause compilation, using clause\_compiler/2. The last stage, defined by insert_into_sorted_list/3 is used to construct the sorted list of indexing entries (Sorted\_Index) from the entries in the set Compiled\_Clauses. */
create_sorted_index(File, Sorted\_Index) :-
    pipe(read\_clauses(File, Input\_Clauses),
        Input\_Clauses,
        [
            [clause\_compiler(Clause\_In, Compiled\_Clause),
             Clause\_In,
             Compiled\_Clause
            ]
        ],
        Compiled\_Clauses,
        insert\_into\_sorted\_list(Compiled\_Clauses, [], Sorted\_Index)).
\end{verbatim}
read_clauses(File, Input_Clauses) :-
    open(File, read, S),
    repeat,
    read_clause(S, Input_Clauses, T),
    T = end_of_file,
    close(S), !.

read_clause(S, Input_Clauses, TExp) :-
    read(S, T),
    expand_term(T, TExp), !.
    TExp = end_of_file,
    write_end_of_stream(Input_Clauses), !.
    TExp = {:-_},
    write_to_stream(Input_Clauses, TExp)
    write_to_stream(Input_Clauses, (TExp:-true)) !.

insert_into_sorted_list (CompiledClauses, SoFar, SortedIndex) :-
    read_from_stream (CompiledClauses, IndexingForClause), !,
    insert (SoFar, IndexingForClause, SoFar2),
    insert_into_sorted_list (CompiledClauses, SoFar2, SortedIndex).

insert_into_sorted_list (_, SoFar, SoFar). insert([], X, [X]).
insert([H1], X, [X, H1]) :- H > X, !.
insert([H1], X, [H2, H1]) :- insert(T, X, T2).
/* An index is generated by grouping the sorted index entries into a list, each entry of which is a list of the indexing entries for a single procedure. Then, we use findall/3 to compute the set of properly indexed procedures. The use of findall/3, along with the definition of compile_an_index/2, is worth studying, since it illustrates a basic pattern for exploiting OR-parallelism to process a list of elements simultaneously. */
generate_indexed_procedures(SortedIndex, IndexedProcedures) :-
    group_clauses (SortedIndex, ProcedureList),
    findall (P, compile_an_index (ProcedureList, P), IndexedProcedures).

group_clauses([], []).

insert([H|T], X, [X, H|T]) :- H > X, !.
insert([H|T], X, [H|T2]) :- insert(T, X, T2).

group_clauses([H|T1], [H|T1], Tg)) :-
    extract_procedure (H, Proc),
    get_1_group (T, Proc, T1, Rest),
    group_clauses (Rest, Tg).

get_1_group([], _, [], [], []).

get_1_group([H|T], Proc, Hg, Rest) :-
    extract_procedure (H, P),
    get_1_group (P, H, T, Proc, Hg, Rest).

get_1_group(Proc, H, T, Proc, [H|T]), Rest) :- !,
    get_1_group(T, Proc, T1, Rest).

get_1_group(_, H, T, _, [H|T]), Rest) :- !,
    get_1_group(T, Proc, T1, Rest).
extract_procedure (Proc, _).
We coded versions of `clause_compiler/2` and `compile_index_for_one_procedure/2` that interfaced to the compilation routines used by the standard compiler used in Gigalips Prolog. This allowed us to actually compile Prolog programs and measure speedups. We chose to measure times a file (`newdic.pl`) from the distributed version of Chat. Briefly, the times we experienced are as follows:

- 1 process: 101 seconds
- 2 processes: 76 seconds
- 3 processes: 45 seconds

With more than three processes, we experienced very little improvement over these times. We found these times surprising, until we analyzed exactly where the time was being spent. When compiled using a single process, the breakdown in where time was spent is roughly as follows:

- Reading input clauses: 25%
- Compiling clauses: 31%
- Insertion of indexing data into the sorted list: 27%
- Grouping indexing data: 5%
- Generating the indexing code: 12%

There are several things worth noting about these figures:

a. The time to read in the clauses is surprisingly high. Given that 26% of the time is spent reading clauses and that 5% was spent grouping the indexing information, 31% of the time is spent doing tasks which seem inherently sequential. Unless substantial improvements can be made in the code which reads input, an upper bound for speedups of about 3 exists.

b. The insertion into an ordered list is probably a very poor idea. It would probably be more efficient to simply sort the entries after clause compilation.

c. The reason that there was not a better speedup with 2 processes is due to the fact that the code (as written) allocates 1 process to reading input an clause compilation, while the second process performs the insertion into the sorted list. The only speedup is due to "hiding" the insertion time and the relatively short period in which the indexing code is generated.

We wish to emphasize that this compiler is not viewed as a "finished product". Rather, it was written to illustrate the techniques described in earlier sections. Clearly, a number of improvements could be made.

9. Summary

This paper has presented a number of techniques that appear to us to be useful, although certainly questionable when judged by the common aesthetic standards of logic programming. We simply present them for what they are: useful hacks that others might wish to employ until proper linguistic constructs with efficient implementations remove any motivation for considering them.

We particularly wish to point out the ease of implementing constructs to support AND-parallelism, given an implementation of OR-parallelism. The versions we give do suffer from drawbacks; the most obvious is that copying solutions into external memory puts a limit on the size of computational tasks to be run in parallel. For example, our implementation would give very poor behavior if one implemented an AND-parallel version of the quicksort/2 algorithm using our constructs. On the other hand, many uses of AND-parallelism do involve fairly sizable computations that produce fairly small structures as output; for these, our approach works quite well.
References


Distribution for ANL-88-11

Internal:

J. M. Beumer (3)
F. Y. Fradin
H. G. Kaper
A. B. Krisciunas
R. A. Overbeek (50)
G. W. Pieper (50)

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (3)

External:

DOE-TIC, for distribution per UC-32 (112)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
   J. L. Bona, Pennsylvania State University
   T. L. Brown, University of Illinois, Urbana
   P. Concus, Lawrence Berkeley Laboratory
   S. Gerhart, Micro Electronics and Computer Technology Corp., Austin, TX
   H. B. Keller, California Institute of Technology
   J. A. Nohel, University of Wisconsin, Madison
   M. J. O'Donnell, University of Chicago
   D. Austin, ER-DOE
   M. Carlsson, Swedish Inst. of Computer Science (5)
   K. Danoff, Southern Illinois University (5)
   G. Michael, Lawrence Livermore Laboratory