LPS Algorithms: A Detailed Examination

Andy Lowry
Stephen Taylor
Salvatore J. Stolfo

Columbia University
Department of Computer Science

March 1984

Abstract

LPS is a Logic Programming System currently under development and specifically targeted for implementation on massively parallel architectures. We present a detailed explanation of algorithms under development for parallel execution of LPS programs. The explanation is significantly more detailed than those published previously. An abstract proof procedure is developed which encompasses these algorithms and several variants, as well as the standard sequential Prolog algorithm. This abstract procedure provides a conceptual basis for our discussion and, in a companion paper, for a critical analysis of various execution strategies.

The algorithms have been successfully implemented and demonstrated in simulation on a number of small programs. Work is currently underway to transfer this implementation to a working prototype machine based on the DADO parallel architecture.

Due to the depth of our treatment we assume that the reader has read previously published literature in the area.

This research is supported cooperatively by: Defense Advanced Research Projects Agency under contract N00039-82-C-0427, New York State Science and Technology Foundation, Intel Corporation, Digital Equipment Corporation, Valid Logic Systems Inc., Hewlett-Packard, AT&T Bell Laboratories and International Business Machines Corporation.

Table of Contents

List of Figures

1. Introduction

Logic programming has attracted a great deal of attention as a medium for the development of software for parallel execution. Two major factors contributing to this perception are the demonstrated suitability of logic programming for the expression of a wide variety of software tasks, and the identification of several sources of parallelism inherent in the logic formalism itself. Thus logic programming languages appear to offer a framework in which programs naturally lend themselves to efficient parallel execution, but in which the programmer need not be overly cognizant of this goal.

With this view in mind we have developed methods for the execution of logic programs written in a language we call LPS, under a particular parallel execution model [14, 12]. Our methods are not well characterized by any of the sources of parallelism identified by Conery [2], although they bear some resemblance to OR and AND parallelism. We unify a conjunction of goals simultaneously throughout a network of what may be considered intelligent memory devices. Each of these devices receives the entire goal list and attempts unification of each goal with every literal in its own local store. Upon completion of this activity, a series of network queries and combining operations results in the construction of a single relation representing all potential solutions of the original conjunction. The cycle repeats by selecting one member of that relation and producing from it a new conjunction to be solved.

We may view our proof search as a perusal through a tree of goal lists, where each node gives rise to children that can be obtained via resolution of one or more of its goals with clauses in the program. The structure of this tree depends on which goals are chosen for resolution in each node. In particular, we note that the standard sequential Prolog algorithm* chooses exactly one goal in each node, whereas the *current* LPS algorithms* always resolve every goal in the goal list. Both algorithms pursue a depth first search, although the LPS search tree, in comparison to the Prolog search tree, is characterized by:

- Shorter paths to leaves
- Earlier termination of unproductive paths
- Earlier consideration of most goals, causing earlier branching but not necessarily higher branching factors
- A substantially reorganized leaf structure, resulting in a different order to the construction of solutions

Although the LPS algorithms may appear to exhibit something of a breadth first nature due to the simultaneous construction of all children for whichever node is under consideration, that view is misleading. Although the children are constructed in unison, one child's subtree is searched before any other child is considered, so that the search pattern itself is purely depth first. The process may be viewed as a hill-climbing strategy in which all branches are equally favored.

---

*See [15]. We will henceforth refer to this algorithm as simply the "Prolog algorithm."

*We note that the algorithms are under ongoing development

In this paper we begin by presenting an abstract proof procedure that encompasses both the LPS and the Prolog algorithms, as well as many variations. We proceed with a specific example of the algorithm at work, followed by detailed explanation of the current LPS implementation in terms of the abstract algorithm.

For an introduction to logic programming methods the reader is referred to [7, 8, 4]. A very brief description of the Prolog language, on which much of LPS has been modeled, may be found in [9]; for complete details see [1]. A description of the computing model for which our algorithms are targeted may be found in [12]. The DADO architecture, for which a specific implementation is underway, is described in [10, 11]. The reconciliation operation which we use may have been independently discovered by Pollard [6], although we have encountered significant difficulty in obtaining this reference. Related algorithms are described in [3].

2. An Abstract Proof Procedure

2.1 Proofs

We define a *proof* for a given directive to be sequence of goal lists beginning with an instance of the directive and terminating in the empty goal list. Each goal list is composed of contributions from the individual goals in the preceding goal list, where each goal contributes any one of the following:

- Itself, as a singleton goal list. In this case we say the goal has been *retained.*

- The empty goal list, if the goal is satisfied via some fact. In this case we say the goal has been *removed.*

- The instance, under some substitution, of a rule body whose rule head, under the same substition, is identical to the goal. Here we say the goal has been *expanded.*

Our proof procedure can then be viewed as the search for such a sequence. In addition, if a proof is found, the minimal substitution that transforms the directive into the first goal list in the sequence is displayed. We call this substitution a *solution* for the directive.

Since there may be more than one way to satisfy any given goal, one goal list may give rise to more than one successor goal list, any or all of which may lead to a successful proof. Thus there may be several proofs for a single directive. In general we will want our proof procedure to be capable of pursuing all possible proofs in a systematic fashion.

The difference stated in the Introduction between the search trees traversed by the Prolog and LPS algorithms may now be restated as follows: The Prolog algorithm pursues proofs in which each proof step consists of either removing or expanding the first goal in a goal list and retaining all other goals. In the current LPS algorithms no goal is ever retained in a goal step; rather, each goal is either removed or expanded.

2.2 The Procedure

Our description of what constitutes a proof allows us to quite readily verify proofs that are handed to us, but it is substantially more difficult to discover correct proofs when they exist. Two processes allow us to identify the substitutions that give rise to proofs: *unification* and *reconciliation.*

Unification [7] provides a method for determining whether a substitution exists that will transform two terms into identical terms. Such a substitution is called a *unifier*, although in the sequel we shall use this term to refer specifically to the *most general unifier*. By "most general" we mean that if U is the most general unifier of terms T_1 and T_2, and S is any other unifying substitution, then $S(T_1)$ is an instance of $U(T_1)$.

Reconciliation [6, 3] is a procedure for determining whether two substitutions are compatible, and if so, producing the "most general" substitution that subsumes both. By this we mean that if R is the reconciliation of substitutions S_1 and S_2, then for any term T, R(T) is an instance of both $S_1(T)$ and $S_2(T)$. As with unification, by "most general" we mean that any other substitution with this property, when applied to any term T, gives rise to an instance of R(T).

Given the mechanisms of unification and reconciliation, the construction of a solution for a directive can be accomplished as shown in Figure 2-1. Starting with the directive itself as a goal list, the algorithm produces successive goal lists until either an empty goal list is constructed or a failure condition is encountered. Upon successful termination, Substitution_List contains a sequence of substitutions whose composition is a solution for the directive.

Construction of a new goal list from its predecessor proceeds as follows:

1. Each goal is analyzed individually to produce:
   - Its *contribution* to the new goal list,
   - A substitution (which we call an *instantiator*) that will be applied to the contribution before its addition to the new goal list, and
   - Another substitution comprising *constraints* on the overall solution.
2. The constraining substitutions are combined via reconciliation to produce a substitution supporting this goal step as a whole. This substitution is saved as a component of the solution that we seek.
3. All instantiators are updated through composition with the above reconciliation.
4. Each contribution is passed through its corresponding instantiator, and the results are collected into a single goal list.

2.2.1 Contributions

Contributions (in their pre-instantiated form) are determined as follows:

- A RETAINED GOAL contributes itself, verbatim.*
- A REMOVED GOAL contributes nothing.
- An EXPANDED GOAL contributes the body of the rule with whose head it unifies, verbatim.

*Keep in mind that we are presenting an abstract proof procedure which encompasses several practical strategies. Thus although we have stated that the LPS algorithms never retain a goal, we include goal retention in our abstract procedure in order to accomodate both the Prolog algorithm and several variants on the LPS algorithms.

```
Goal_List := Directive;
Substitution_List := NIL;

WHILE Not Empty(Goal_List) DO

  Constraint_Set := NIL;

  FOREACH goal G in Goal_List DO
    Decide whether G is to be retained, removed, or expanded;
    IF retaining G THEN
      Contribution(G) := G;
      Instantiator(G) := NIL;
    ELSE IF removing G THEN
      Find a fact unifying with G, call the unifier U;
        IF none can be found, FAIL;
      Contribution(G) := NIL;
      Instantiator(G) := NIL;
      Restrict U to bindings for variables in G, add
        the result to Constraint_Set;
    ELSE IF expanding G THEN
      Find a rule R whose head unifies with G, call the unifier U;
        IF none can be found, FAIL;
      Contribution(G) := rule body of unifying rule;
      Instantiator(G) := U restricted to variables in R;
      Insert bindings to new created variables into Instantiator(G)
        for all variables from R not bound by U;
      Restrict U to bindings for variables in G, add
        the result to Constraint_Set;
    FI;
  OD;

  Compute reconciliation of all substitutions in Constraint_Set,
    call the result Rec; IF reconciliation fails, FAIL;
  Add Rec to Substitution_List;

  New_Goal_List := NIL;
  FOREACH goal G in Goal_List DO
    Instantiator(G) := Instantiator(G) composed with R;
    Instantiate Contribution(G) using Instantiator(G),
      and add the result to New_Goal_List;
  OD;

  Goal_List := New_Goal_List;
OD;
```

Figure 2-1: Abstract Proof Procedure

2.2.2 Instantiators

Non-empty instantiators are only produced for expanded goals. It would be pointless to compute an instantiator for a removed goal since its contribution is always empty; in the case of a retained goal, all instantiation information comes from the constraints imposed by unification of non-retained goals, so an empty instantiator is set in place awaiting composition with the reconciliation of those constraints.

The instantiator for an expanded goal is simply the unifier that resulted from unification of the goal with a rule head. We only include bindings for variables that are contained in the rule (*rule variables*), since other bindings cannot contribute to instantiation of the rule body. We also insure that every rule variable is represented in the instantiator by binding any unbound rule variables to new created variables. Such a binding adds no information; the objective is to insure that the instantiated rule body will contain none of the original rule variables.

2.2.3 Constraints

Constraints are produced by unification of removed goals with facts and expanded goals with rule heads. Each unifier is added to a constraint set, after restricting it to variables that occurred in the goal (*goal variables*). The constraint set is used to produce a consistent substitution for the preceding goal list which supports its transformation into the succeeding goal list. Thus the only bindings of interest are those for goal variables, which is why the unifiers are pruned before adding them to the constraint set. Indeed, if the same fact or rule head is used to unify with more than one goal, inconsistent bindings for non-goal variables might result, but these must not prevent the proof from progressing. For example, consider the following program:*

Rule 1: tasty(X) :- sweet(X).
Fact 1: sweet(cookies).
Fact 2: sweet(cake).

Directive: tasty(cookies), tasty(cake).

We suppose that (as would be the case with LPS) our algorithm chooses to expand both of the original goals in its first step, using Rule 1. Unification of **tasty(cookies)** with **tasty(X)** produces the unifier **[X/cookies]**, while unification of **tasty(cake)** with **tasty(X)** produces **[X/cake]**. Reconciliation of these two unifiers cannot succeed since variable **X** cannot be bound to both **cookies** and **cake** simultaneously. Clearly, though, the directive is provable. This problem of unwanted binding interaction does not occur if we discard bindings for **X** prior to reconciliation. Note that these bindings remain in instantiators so that they may be used for instantiation of rule bodies.

Similar reasoning shows why it is necessary to include "dummy bindings" for non-unified rule variables in the instantiators for expanded goals. If this were not done, those rule variables might end up occuring in two or more goals at some point during the proof. This would cause unwanted interactions since the algorithm would insure that only mutually compatible bindings were produced for all occurrences of those variables, while the separate occurrences should in fact be treated independently.

The purpose of composing each instantiator with the constraint set reconciliation is to insure that each

*For our examples we adopt the Prolog convention that symbols beginning with a capital letter are considered variables, while all others are considered predicate and function symbols.

goal list is cast in terms of the current state of knowledge of the solution under construction. That solution is constructed as a sequence of component substitutions, where each proof step produces one component. If goal lists are not kept up to date in this fashion, the same variable may end up bound by two or more different components. During later composition of the components, all but the first of these bindings would be completely lost. For example, the composition of **[X/cookies]** with **[X/cake]** is simply **[X/cookies]**. In general, it will be the case that no goal list will ever contain a variable for which a binding exists anywhere in the component substitutions produced thus far in the proof procedure.

2.3 Some Observations

Due to the "most general" nature of unification and reconciliation, our algorithm computes the most general solution that will support the constructed proof. This translates into conciseness in the solution set reported for a directive, although it does not guarantee that no solution will be an instance of another. This may arise if there are multiple proof paths for some particular solution.

Upon failure of a particular proof path, both the LPS and Prolog algorithms backtrack to the most recent choice point and pursue an alternate path. In the LPS algorithms we find that all of these alternate paths have already been started by the simultaneous construction of all possible successor goal lists from the choice point. The Prolog algorithms do not benefit from such a head start. As mentioned in the Introduction, this feature may easily mislead one to suspect that the LPS search strategy includes some breadth first component rather than being strictly depth first.

Finally, it will be seen that in LPS the composition of the component substitutions is performed incrementally as each component is produced, rather than computing the entire composition at the end of the proof.

3. A Proof Example

Consider the following program:

Rule 1: can_eat(X) :- food_store(S), open(S,now), has_money(X).
Rule 2: has_money(X) :- friend(Y,X), has_money(Y).
Fact 1: food_store(mama_joys).
Fact 2: food_store(take_home).
Fact 3: friend(chris,andy).
Fact 4: friend(tori,chris).

Suppose the author is interested in whether or not he is currently able to eat. First, from general knowledge of neighborhood food stores, and by subtly questioning his friends, he arrives at the following additional facts:

Fact 5: open(mama_joys,now).
Fact 6: has_money(tori).

Next he invokes the algorithm with the directive **can_eat(andy)** and observes the following execution:

1. The initial goal list is {**can_eat(andy)**}. We choose to expand the single goal via Rule 1. Unification with the rule head produces the substitution [**X/andy**].

Our goal's pre-instantiated contribution is the rule-body, {**food_store(S), open(S,now), has_money(X)**}. The instantiator is [**X/andy,S/_1**], where **_1** is a created variable to which S is bound since it was not bound during unification. This expansion contributes nothing to the constraint set since no goal variables were bound during unification (indeed, there were no goal variables to be bound!).

Reconciliation of our (empty) constraint set produces an empty substitution, so our instantiator is not affected, and the next goal list is {**food_store(_1), open(_1,now), has_money(andy)**}.

2. Current goal list: {**food_store(_1), open(_1,now), has_money(andy)**}

Retain goal **food_store(_1)**:

> Contribution: **food_store(_1)**
> Instantiator: **NIL**
> Constraint: **NIL**

Remove goal **open(_1,now)** via Fact 5:

> Contribution: **NIL**
> Instantiator: **NIL**
> Constraint: [**_1/mama_joys**]

Expand goal **has_money(andy)** via Rule 2:

> Contribution: {**friend(Y,X), has_money(Y)**}
> Instantiator: [**X/andy,Y/_2**]
> Constraint: **NIL**

The overall constraint set is {[**_1/mama_joys**]}, whose *reconciliation* is just [**_1/mama_joys**]. The only instantiator that is affected by this reconciliation is the first, which becomes [**_1/mama_joys**]. Instantiating all of the contributions with their instantiators then produces the new goal list: {**food_store(mama_joys), friend(_2,andy), has_money(_2)**}.

3. Current goal list: {**food_store(mama_joys), friend(_2,andy), has_money(_2)**}

Remove goal **food_store(mama_joys)** via Fact 1:

> Contribution: **NIL**
> Instantiator: **NIL**
> Constraint: **NIL**

Remove goal **friend(_2,andy)** via Fact 3:

> Contribution: **NIL**
> Instantiator: **NIL**
> Constraint: **_2/chris**

Expand goal **has_money(_2)** via Rule 2:

Contribution: **{friend(Y,X), has_money(Y)}**
Instantiator: **[X/_3, Y/_4]**
Constraint: **[_2,_3]**

The overall constraint set is **{[_2/chris], [_2/_3]}**, whose reconciliation is **[_2/chris, _3/chris]**. This affects the instantiator for the third goal, which becomes **[X/chris,Y/_4]**. Instantiating all of the contributions with their instantiators yields the new goal list: **{friend(_4,chris), has_money(_4)}**.

4. Current goal list: **{friend(_4,chris), has_money(_4)}**

Remove goal **friend(_4,chris)** via Fact 4:

Contribution: **NIL**
Instantiator: **NIL**
Constraint: **[_4/tori]**

Remove goal **has_money(_4)** via fact 6:

Contribution: **NIL**
Instantiator: **NIL**
Constraint: **[_4/tori]**

The overall constraint set is **{[_4/tori], [_4/tori]}**,* whose reconciliation is **[_4/tori]**. All contributions are nil, so the new goal list is empty.

5. Current goal list: {}

The algorithm terminates successfully upon encountering an empty goal list.

The sequence of reconciliations that was generated by the algorithm is:

[]
[_1/mama_joys]
[_2/chris, _3/chris]
[_4/tori]

The composition of these components yields the overall substitution: **[_1/mama_joys, _2/chris, _3/chris, _4/tori]**. The sequence of generated goal lists is:

{can_eat(andy)}
{food_store(_1), open(_1,now), has_money(andy)}
{food_store(mama_joys), friend(_2,andy), has_money(chris)}
{friend(_4,chris), has_money(_4)}
NIL

If we apply the overall substitution to this sequence of goal lists, we arrive at our final proof:

---

*Of course, this constraint set is not really a *set* since it contains duplicate entries. However, the terminology is useful in a loose sense, and the current LPS implementation will in fact go through the work of reconciling two identical constraints rather than removing the duplicity.

{can_eat(andy)}
{food_store(mama_joys), open(mama_joys,now), has_money(andy)}
{food_store(mama_joys), friend(chris,andy), has_money(chris)}
{friend(tori,chris), has_money(tori)}
NIL

4. The Current LPS Implementation

The LPS algorithms that we have formulated can most easily be understood as comprising three computational phases: ***unification***, ***join***, and ***substitution***. In this section we will discuss an actual LPS implementation in terms of these components, relating each functionally to the abstract algorithm outlined above.

The implementation is based on the computing model described in Taylor et al [12]. Very briefly, we envision a network of independent *processing elements* (PE's) each equipped with a moderate local storage capacity. The network is controlled by a *contol processor* (CP) which coordinates global communication and invokes individual instructions as well as local procedures in unison throughout the PE network. Global communication consists of *broadcast* messages from the CP to the network, and *reports* solicited by the CP from individual PE's.

4.1 The Binding Set Representation

A binding set represents the result of applying a single step of our proof procedure to a goal list. It contains the following information:

- The reconciliation of the constraint set produced by unification of goals with facts and rule heads.

- A list of rule body keys by means of which rule bodies may be obtained at the CP for instantiation and inclusion in a new goal list. Note that a single rule body key may appear more than once. This will be the case if the same rule head was used to expand more than one goal in the goal list.

- An instantiator for each rule body key contained in the binding set. If a key appears more than once, each is associated with its own instantiator.

Recall that the current LPS algorithms never retain goals from one goal list to the next. Thus the above set of information includes everything required to construct the successor goal list as well as the solution component produced by this goal step.

The overall data structure may be viewed as comprising several "layers," each identified with a layer "marker." Each layer contains a substitution of some sort -- either the single reconciliation carried by the binding set or one of the possibly many instantiators. In the former case, the layer is called the *common layer* owing to its nature as a substitution that encompasses all the constraint set components contributed by the unifications. The layer marker for the common layer is the atom, **COMMON**. A layer containing an instantiator is called a *rule layer*, since a non-empty instantiator is produced only for a goal that is expanded by unification with some rule head. The marker for a rule layer is a key identifying the rule that was used in the expansion.

A binding set with no rule layers is of special interest, and we call it a *simple binding set*. Other binding sets are symmetrically termed *complex binding sets*. A simple binding set is important because it is reported only at the completion of a successful proof.

4.2 Distribution of Data

As we shall see, all unification is performed in the individual PE's that form the processor network, whereas instantiation takes place in the CP. For this reason we store all facts and rule heads, (that is, all the positive literals of our program) in the PE network itself. Each literal resides in a single PE, although any PE may contain several literals. Rule bodies, on the other hand, are kept in the CP. Each rule head in the PE network is tagged with a key which can be used to identify the corresponding rule body in the table maintained by the CP.

During execution of a logic program, goal lists are constructed in the CP, initially from the directive and subsequently from the goal list contributions carried in the binding sets. When a goal list is complete it is transmitted to the PE network where unification, reconciliation, and composition operations produce new binding sets. Of the possibly many binding sets produced, a single set is selected for transmission back to the CP, and the entire cycle is resumed while the other binding sets lie dormant in the PE network awaiting later selection. The operation is shown pictorially in figure 4-1.

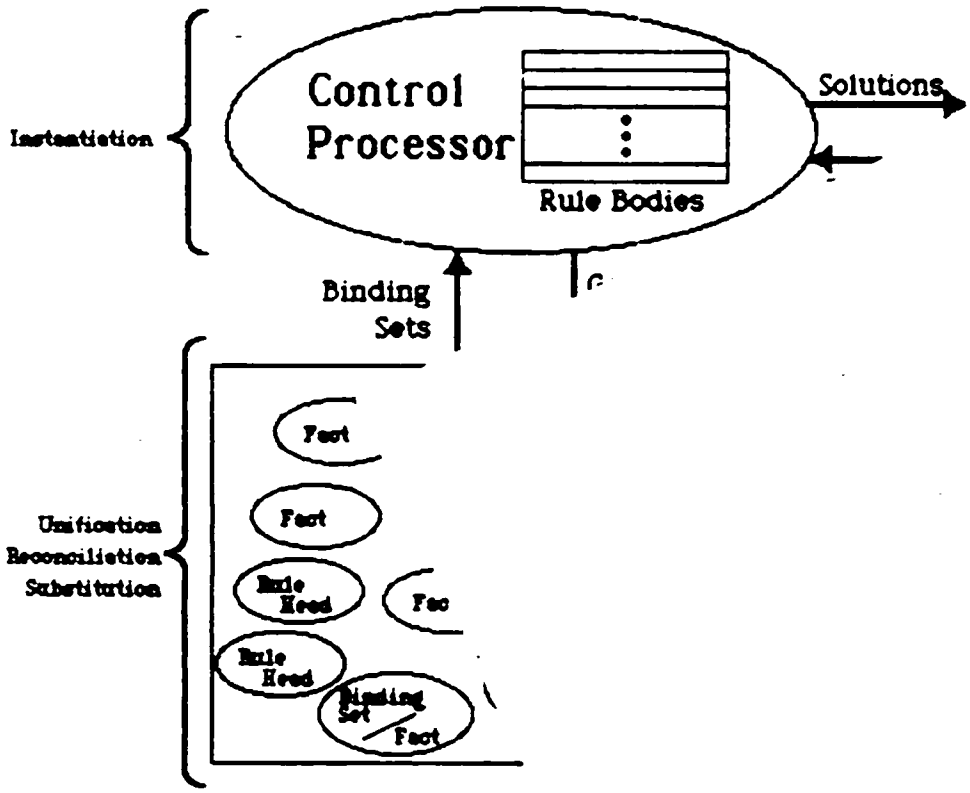

Figure 4-1: Flow of Data in LPS Execution

4.3 The Unification Phase

The first phase of the LPS algorithm begins with the transmission of a goal list from the CP into the PE network. Residing in each PE is some (possibly empty) collection of facts and rule heads that were placed there when the program was initially loaded into the machine. Once the transmitted goal list has been captured, each PE unifies every goal with as many of its resident literals as possible, producing unifiers which are stored in the PE's local storage.

Unification with a fact produces a simple binding set whose common layer is the constraint set contribution specified by the abstract algorithm for a removed goal. That is, the unifier is stripped of all bindings for variables that were not present in the unified goal, and the resulting substitution becomes the common layer.

Unification with a rule head produces a complex binding set whose common layer is the unifier stripped of its non-goal variable bindings (same as the common layer for a removed goal). The rule layer is the instantiator for the expansion, as specified in the abstract algorithm. In other words, the unifier is stripped of all bindings for non-rule variables, and supplemented with bindings to new created variables for all unbound rule variables.* The marker for the rule layer is the key associated with the unifying rule head.

Each binding set produced during the unification phase is tagged with a *level number* which identifies, via its position within the transmitted goal list, the goal whose unification gave rise to the binding set. It will become clear during the discussion of the join phase why this tagging is required.

4.4 The Join Phase

We have named the second phase of our execution loop as the "join phase" due to a useful interpretation of the basic operation as an equi-join over a set of database relations. Indeed, if we recall that each goal in the transmitted goal set gave rise, during the unification phase, to a collection of binding sets with a common level number, we see that the level number provides us with a key to the "relation" defined by the corresponding goal. The database from which the relation was produced is the collection of literals (facts and rule heads) present in the PE network.

With this interpretation in mind, one sees that joining these several relations, using reconciliation as the basic pair-wise matching operation, computes reconciliations for all compatible combinations of unifiers for the goals in the transmitted goal list. At the completion of the join phase, every one of these binding sets will reside in the PE network and will be elegible for later selection and elaboration of the particular proof path it represents. Thus the transmitted goal list can be discarded at that point.

Any matching operation performed on two binding sets will require that the two bindings sets be accessible to the same processor. In general that will not be the case at the completion of the unification phase, since each binding set is stored in the PE containing the unifying literal. The join phase thus requires communication of binding sets around the network. This communication is coordinated by the CP.

The basic step in the join phase consists of selecting two relations out of the several to be joined and

---

*Note that variables created by two different PE's must be distinguishable. This is easily done if the PE's can be assigned unique identification tags, as those tags may then be incorporated into the created variable names. Such tags may be assigned at system startup using resolve and report operations. Alternatively, many existing and proposed machines fitting our model can generate unique ID's using various highly efficient methods.

joining those two into a single relation, thus decreasing by one the number of relations to be joined. When only one relation remains, the join phase is complete.

In order to join two relations, one of the two is chosen to "feed into" the other. The CP loops over the feeder relation, extracting one member from the PE network during each iteration. As each element is obtained from the feeder it is broadcast to the entire PE network, and any PE that holds elements from the "consumer" relation attempts to reconcile the *common layer* of the feeder with each of its resident consumers (remember, the common layer is where the constraint set contributions were placed during the unification phase). Whenever reconciliation succeeds, a new binding set is created whose common layer contains the reconciliation. Any rule layer that appeared in either of the contributing binding sets is included in the new binding set, and the level number is set so as to identify the new binding set as belonging to the new joined relation under construction.

Each feeder binding set is discarded as soon as it has been matched against all possible consumers, and when the entire pair-wise join has been completed, the original consumer relation is discarded as well. Thus two relations have been discarded, and one has been produced, bringing us nearer to our goal of a single relation.

4.4.1 A Heuristic For Ordering The Join Phase

In our computing model communication should be held to a minimum since it must all be funneled through a single channel (the CP). Due to the commutative nature of the reconciliation operation, we may exercise a simple heuristic that should, under most circumstances, keep join phase communication close to minimal. Specifically, we always choose the smallest existing relation as the feeder, and the largest relation as the consumer. Cases can easily be constructed in which some other ordering turns out to be preferable, but the heuristic seems reasonable in the absense of methods for predicting the sizes of intermediate join results.

In the general case we choose to implement an approximation to the above heuristic since our computing model does not provide an efficient means of determining the size of a distributed relation.* We make use of a sequencing mechanism applied to the relation members. The idea is that within each relation the individual binding sets are assigned unique *sequence numbers* in the hope that the difference between the highest and lowest sequence numbers in a relation will generally be a useful estimate to the size of the relation.

In the current LPS implementation, sequence numbers are assigned during the unification phase according to the order in which the clauses were asserted during program loading. Thus any binding set that is produced by unification with the program's first clause is assigned a sequence number of one. Unification with the program's second clause yields sequence number two, and so on.

The assignment of sequence numbers to join results is analogous to the calculation of storage offsets to multi-dimensioned array elements. The first "dimension" is represented by the sequence number of the contributing binding set from the first relation (level number one), and so forth. The "offset" calculation can be performed efficiently by precomputing (in time linear in the number of relations) a "dope vector" similar to that used by many programming languages for array indexing. All sequence numbers are multiplied (again in linear time) by the dope vector elements corresponding to their level numbers prior to

*Note, however, that many architectures fitting our model do in fact allow for fast network-wide sums, making the heuristic viable as presented. We hope to clarify the need for such a mechanism through statistical investigations.

the commencement of the join operation. Then when two binding sets reconcile successfully, the sequence number for the new binding set is the sum of the two contributing sequence numbers.

In addition to their contribution to the join ordering heuristic, sequence numbers provide a method for ensuring a predictable perusal of the proof space by our implementation. Although from the point of view of pure theorem proving such predictability is inessential, under some circumstances such as I/O and recursion, it is crucial if the programming system is to be useful for a more general class of programs, as is the case with Prolog. Unfortunately, the sequence numbers as described here do not appear to provide an ordering that is easily comprehended or well suited for many programming tasks, so that alternatives must still be investigated.

4.4.2 Partition Of The Join Phase

For reasons that will become apparent in the upcoming discussion of variable purging, it may be desirable to impose a global constraint on the join phase ordering so that the relations arising from any single goal list contribution are fully joined among themselves prior to any attempt at combining results from different contributions. We adopt this strategy in the current LPS algorithms by conducting the join phase in two steps. First, a series of *partial joins* takes place in which each goal list contribution is reduced to a single relation in the PE network. When the partial joins have completed, a *final join* joins each of these relations into a single relation representing the successors to the goal list under consideration.

4.5 The Substitution Phase

The last task to be performed upon the discovery of a successful proof is the composition of the various substitutions that were generated along the way. As indicated in the abstract algorithm, these substitutions are the constraint set reconciliations computed to support the individual proof steps. Their composition is computed in the substitution phase of our algorithm.

As was briefly mentioned in the observations following the abstract proof procedure, we have chosen in our current implementation to compute this composition incrementally as the individual components are generated. Thus each time a new reconciliation is produced, we compute its composition with all prior reconciliations in its proof path. Once this has been computed, the individual reconciliation itself can be discarded.

In order to achieve this strategy, we store in the common layer of a binding set, not the individual reconciliation that produced the binding set, but its composition with all prior reconciliations on its proof path. This is easily implemented because all of the binding sets produced by a join phase share a common proof history, and the cumulative substitution representing that history is exactly the substitution stored in the common layer of the complex binding set that gave rise to this proof step in the first place.

In our LPS implementation, then, the substitution phase is accomplished by transmitting the prior reconciliation history to the PE network following the join phase and computing in each PE the composition of that substitution with any new reconciliations.

Three possible benefits derive from our incremental substitution strategy. First, composition computations are performed in parallel in the PE network rather than individually for each reported solution by the CP. Second, debugging is easier because the progress represented by each binding set can be read directly in terms of the original directive variables rather than an obscure collection of created variables. Finally, we avoid a bookkeeping chore in the CP which, depending upon whether certain variants on the basic algorithms are chosen, may be extremely expensive in both time and space.

4.6 Managing Created Variables

In order to keep communication and processing costs to a minimum, it is desirable to discard bindings from our binding sets whenever they are no longer needed. In general the instantiator stored in a rule layer of a binding set will contain a binding for each variable appearing in the rule body, and no other bindings. Thus rule layers are not a problem in this respect. The common layer is more complicated.

In general there are two possible reasons for keeping a binding in the common layer of a binding set:

- The binding will be required in order to construct a solution, should the current proof path succeed.

- The binding might interact with other bindings to constrain the search space, so that discarding the binding could lead to incorrect proofs.

If at any point a particular binding can be determined not to fulfill either of these conditions, we may freely discard the binding and proceed with our proof.

When we report a solution, we limit the report to a display of a minimal substitution that will transform the directive into a satisfiable goal list. In particular, the intermediate goal lists are not displayed, in either their instantiated or uninstantiated form. Recall that our substitution phase is implemented incrementally, so that common layer substitutions always represent the total accumulated current knowledge of the solution being pursued. Thus we see that our first condition demands only that we not discard bindings for variables that appear in our original directive (*top-level variables*).

Other bindings are required for their constraining effects. However, we observe that once a binding has been produced for a variable, it is immediately used to remove all appearances of the variable from the binding set. Aside from this instantiation, the only way a binding can ever act to constrain the search space is through reconciliation with another binding for the same variable. But by the instantiation itself, we are guaranteed never to see the variable in a future goal list along the same proof path, so that no future bindings for it will ever be produced. Thus no further constraint by the variable is possible. We conclude that we need never maintain bindings for a variable (other than a top-level variable) once a binding for it has appeared at the end of a proof cycle.

We do not claim that the binding would not undergo further changes were it to be maintained throughout the remainder of the proof. For instance, if we produce the binding [_1/p(_2)] we may later produce the binding [_2/a]. The overall proof substitution would then include the binding [_1/p(a)]. However, the search constraints that are represented by this refinement are accomplished by the construction and reconciliation of bindings for _2; the refinement of _1's binding is a more or less passive side-effect. Since _1 is not a top-level variable, we have no interest in this side-effect, so there is really no point in producing it in the first place.

We see, then, that when a binding set is reported to the CP from the PE network its common layer should contain bindings only for top-level variables. However, more can be said about the other variables as well. In particular, we recall the join phase partitioning strategy discussed earlier, in which the join phase proceeds by a series of partial joins involving relations produced by common goal list contributions, followed by a final join of the partial join results. It turns out that many bindings can be pruned from the binding sets before the final join takes place, thus saving in communication costs during that join.

Recall that if a rule variable is not bound during unification the resulting instantiator is augmented by binding that variable to a new created variable. The created variable will thus appear in exactly one of the goal list contributions represented by the complete binding set, and hence in exactly one of the partial

join result relations. Such a variable cannot constrain the final join, and since it is not a top-level variable, it will be discarded when the final join is complete. We can save communication costs in the final join if we discard the variable prior to the final join.

A list of such discardable variables may be computed easily by the CP during instantiation of a rule body by gathering together term sides of all variable/variable bindings in the instantiators. For example, if the binding [_34/_46] appears in an instantiator, we can safely discard all bindings for variable _46 prior to the ensuing final join.

We note here that if we are to discard bindings before the final join takes place, we must account for the possibility that some of our top-level variables are bound to terms that include discardable variables. Thus the composition operation that constitutes our substitution phase must in fact be performed prior to the final join. We may apply the operation simultaneously to all the relations that will take part in that join by waiting until all the partial joins have completed.

5. Conclusions and Future Work

It has not yet been established that the pilot algorithms presented in this paper can result in efficient interpreters for the execution of logic programs under the parallel computing model that we propose. A limited form of OR parallelism is achieved through simultaneous unification of individual goals with literals that are distributed over a large multiprocessor network, and a limited form of AND parallelism is achieved by satisfying an entire list of goals in a single algorithm cycle. Our abstract proof procedure has provided a convenient basis for comparison between the LPS algorithms and the Prolog algorithm.

Our algorithms have been implemented in order to uncover problems in parallel execution of logic programs and to discover various alternative strategies applicable under our computing model. The experience and information gained will be used in conjunction with statistical measurements to highlight fruitful areas for future research.

A companion paper [5] investigates specific alternatives to the LPS algorithms, again in the context of our abstract proof procedure, and presents a comparative analysis of the various strategies. It is found that no one strategy is optimal in all situations. Future research will further explore these and other alternatives, and will attempt to develop mixed strategies in which alternatives are chosen based on static and dynamic analysis of the program under execution.

We are currently planning an implementation of a LPS interpreter on a prototype machine based on the DADO parallel architecture. One such prototype comprising fifteen PE's is currently functioning; a 1023-node prototype is under construction. Weisberg and Lerner are working on an implementation of a parallel version of Portable Standard Lisp for the DADO machine [16]. As our simulation software was written in PSL, we expect that this effort will substantially simplify our implementation task by allowing a simple recompilation of large portions of the existing code for execution on the actual machine.

Taylor [13] describes various methods currently under development for statistical analysis of logic programs. These include static, dynamic, and data-flow analyses intended to guide algorithmic decisions in the implementation of LPS. It is hoped that these analyses will quantify the potential for parallel execution, allow accurate performance estimates to be made, and isolate various qualities of logic programs which can be used in building intelligent compilers and interpreters.

Many features must be added to the LPS language in order to make it suitable for a wide range of

applications. We intend to investigate such features as negated condition elements in rules, evaluable predicates, and condition elements with side effects. Khabaza's work [3] appears promising as a basis for the implementation of negation as failure in the LPS framework. In addition, we will explore issues relating to control of program execution, including a more useful ordering of the solution set.

References

1. Bowen, D. L., L. Byrd, F. C. N. Pereira, L. M. Pereira, D. H. D. Warren. *DECsystem-10 Prolog User's Manual.* University of Edinburgh, Dept of Artficiail Intelligence, 1982.

2. Conery, John S. *The AND/OR Process Model for Parallel Interpretation of Logic Programs.* Ph.D. Th., University of California Irvine, June 1983.

3. Khabaza, Tom. Negation As Failure And Parallelism. 1984 International Symposium On Logic Programming, IEEE Computer Society, Technical Committee on Computer Languages, Atlantic City, February, 1984, pp. 70-75.

4. Kowalski, Robert. *Artificial Intelligence Series.* Volume 7: *Logic for Problem Solving.* North Holland, New York, 1979.

5. Lowry, Andy, Stephen Taylor, Salvatore J. Stolfo. LPS Algorithms: A Critical Analysis. Columbia University, New York, NY 10027, March, 1984.

6. Pollard, G.H. *Parallel Execution of Horn Clause Programs.* Ph.D. Th., Department of Computing, Imperial College, 1981.

7. Robinson, J. A. "A Machine-Oriented Logic Based on the Resolution Principle." *Journal of the ACM Vol. 12* (1965), 23-44.

8. Robinson, J. A.. *Logic: form and function.* Edinburgh University Press, 1979.

9. Shapiro, Ehud Y.. *ACM Distinguished Dissertations.* Volume : *Algorithmic Program Debugging.* The MIT Press, Cambridge, MA, 1982.

10. Stolfo, S. J. and Shaw, D. E. "DADO: A Tree-Structured Machine Architecture For Production Systems." *Proceedings of the National Conference on Artificial Intelligence Vol. 1* (August 1982).

11. Stolfo, S. J., Miranker, D. and Shaw, D. E. Architecture and Applications of DADO, A Large-Scale Parallel Computer for Artificial Intelligence. Proceedings of the Eighth International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence, Inc., Karlsruhe, West Germany, August, 1983, pp. 850-854.

12. Taylor, S., Lowry, A., Maguire, G. Q. Jr., Stolfo, S. J. Logic Programming using Parallel Associative Operations. 1984 International Symposium on Logic Programming, Atlantic City, February, 1984, pp. 58-68.

13. Taylor, Stephen, Andy Lowry, G. Q. Maguire Jr., Salvatore J. Stolfo. Analyzing Prolog Programs. Columbia University, New York, NY 10027, March, 1984.

14. Taylor, S., C. Maio, S.J. Stolfo, D.E. Shaw. Prolog On The DADO Machine: A Parallel System for High-Speed Logic Programming. Third Annual International Phoenix Conference On Computers And Communications, IEEE, March, 1984.

15. Warren, D. H. D. Implementing Prolog - Compiling Predicate Logic Programs. Tech. Rept. D.A.I. 39/40, Department of Artificial Intelligence, Edinburgh University, May, 1977.

16. Weisberg, M. K., Lerner, M. D., Maguire, G. and Stolfo, S. J. ||PSL: A Parallel Lisp for the DADO Machine. Columbia University, New York, NY 10027, February, 1984.