SOME RESULTS ON THE COMPLEXITY OF EXPLOITING DATA DEPENDENCY IN PARALLEL LOGIC PROGRAMS

ARTHUR DELCHER AND SIMON KASIF

▷ We consider several problems related to maintaining and analyzing dataflow dependencies in AND-parallel execution of logic programs. Several problems related to optimal selection of literals for parallel execution are established to be intractable (NP-complete). Most importantly, we establish intractability even when the arity of the predicates in the logic program is restricted to a small constant. This situation represents PROLOG programs used in practice. We subsequently address the complexity of maintaining data-dependency changes that occur during program execution as variables in the literals become instantiated. For this problem we propose a simple and efficient data structure to maintain the dataflow dependencies among literals during the execution of the program. These dependencies may then be used by an intelligent control to minimize backtracking. ◁

1. INTRODUCTION

In attempting to devise schemes for parallel execution of logic programs, one obvious approach is to execute, independently in parallel, all the literals in a current goal of a program. When variables are shared among literals, however, each process executing a literal must ensure that the terms it binds to its variables are compatible with the terms every other process binds to the same variables. We call literals that share a variable *data-dependent*, and those that don't *data-independent*. For example, in the goal

$$:- p(X, Y), q(Y, Z).$$

Address correspondence to Professor Simon Kasif, Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218.

Received 20 May 1987, accepted 6 December 1987.

655 Avenue of the Americas, New York, NY 10010 0743-1066/89/$3.50

if the process executing the literal $p(X, Y)$ instantiates Y to a, while the process executing the literal q instantiates Y to b, then these two processes must communicate and resolve their discrepancy. Yet at this point, it is difficult to determine which of the two candidate bindings should be retained, for the ultimate value for Y may turn out to be either a or b, or some other value entirely.

For this reason many current parallel logic programming systems employ *dataflow analysis*, namely, the determination of data dependency among literals in the program. The methods generally belong to one of two categories:

The execution of a logic program is guided by annotation that dictates the selection of executable literals.

The interpreter tries to select for execution an "optimal" set of literals that don't share any variables. In this case, at every step the interpreter is essentially facing a scheduling problem whose complexity previously has been unknown.

In either case the interpreter depends heavily on dataflow analysis.

In this paper we study two problems related to dataflow analysis of logic programs:

(1) The complexity of scheduling goals in parallel logic programs.

(2) The complexity of incremental *dynamic* dataflow analysis, i.e., the data-dependency changes that occur during program execution as variables in the literals become instantiated. It has been observed that such dynamic dataflow analysis may be computationally prohibitive. To the best of our knowledge, however, no explicit complexity results have been reported.

We study these problems in the context of the relatively simple class of function-free logic programs (sometimes referred to as datalog programs). This class has numerous applications for databases and expert systems. We show that the problem of determining an optimal set of literals for parallel execution is NP-complete for this class. Clearly, since the simplest instance of the problem is shown to be intractable, our results are immediately applicable to the class of logic programs at large.

Problem (1) and its variations are shown to be intractable (NP-hard) even for this restricted subset of logic programs. Most importantly, we establish intractability even when the arity of the predicates in the logic program is restricted to a small constant. This situation represents PROLOG programs used in practice. Thus, our findings support the intuitions conjectured in [2] and [5] which propose several heuristic approaches for the problem.

For problem (2) we propose a simple and efficient algorithm to maintain the dataflow dependencies among literals during the execution of the program. These dependencies may then be used by an intelligent control to minimize backtracking.

1.1. *Graph Representation of Data Dependency*

We assume that the current goal of our function-free logic program contains n function-free literals, whose maximum arity is m, and that the entire goal contains k different variables. Discounting constants and multiple occurrences of the same variable in a single literal (since these have no effect on data dependency), it is clear

that $m \le k \le mn$. The current goal can then be expressed in general as

$$:- p_1(A_{1,1},\ldots,A_{1,m_1}),\ldots,p_n(A_{n,1},\ldots,A_{n,m}).$$

where the p_i's are the principal functors of the literals (and by obvious identification will be used to refer to the literals), and the A_{ij}'s are the arguments, each one corresponding either to a variable V_q, $1 \le q \le k$, or to some constant, or to a null entry in the case that the literal is of arity less than m. When not otherwise mentioned, it will be assumed that there are neither constants nor multiple occurrences of the same variable in a single literal.

Data dependency in a clause can be regarded in terms of a (simple) graph whose vertices correspond to the literals in the clause, with an arc between two literals iff they share a variable. We refer to the graph in this form as the connectivity graph (CG). An equivalent representation is a bipartite graph, hereafter referred to as BP, in which each literal and each distinct variable name in the clause correspond to a unique vertex in the graph, with an arc joining every literal to each of the variables it contains. Figure 1 illustrates the CG and BP that correspond to the clause

$$:- p(X,Y,Z), q(Y,W), r(X,Y), s(Z,W).$$

BP will always have k (the number of variables in the clause) more vertices than *CG*. It is likely, though, that *BP* has fewer arcs than *CG*. An extreme case is a clause with n literals, each containing the same single variable, such as

$$:- p_1(X), p_2(X),\ldots,p_n(X).$$

Here BP has exactly n arcs, whereas CG has $n(n-1)/2$ arcs.

In the course of execution of a logic program, as the current goal changes, corresponding changes occur in the data-dependency graph. Nodes (and their incident edges) are added and deleted as a result of literals being added to and removed from the current goal through resolution. Edges must be added to existing nodes when different variables become unified, and deleted when a variable becomes bound to a constant. For example, if the goal

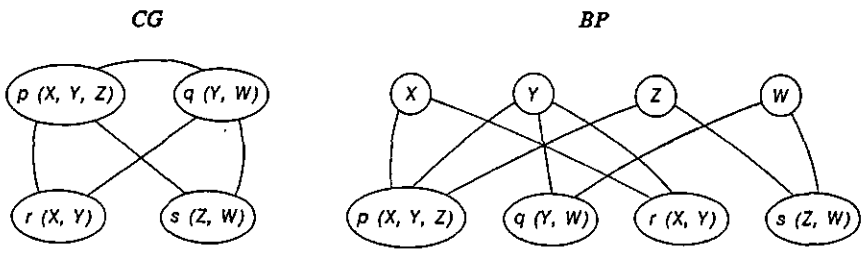
$$:- p_1(X), p_2(X),\ldots,p_n(X).$$

is resolved against a clause such as

$$p_1(a).$$

then the variable X is bound to the constant a, so that there are no shared variables

FIGURE 1. CG and BP for $:- p(X,Y,Z), q(Y,W), r(X,Y), s(Z,W).$



left in the resolvent. In any explicit representation of CG, this process would necessitate the deletion of $O(n^2)$ arcs.

A similar situation occurs in a program like

$$:- p(X, Y), q_1(X), \ldots, q_s(X), r_1(Y), \ldots, r_t(Y).$$
$$p(X, X).$$
$$\vdots$$

If p is executed, it succeeds immediately and unifies X and Y throughout the goal, which requires st new edges to be added to an explicit representation of CG.

For these reasons, we generally assume that the data dependency in a clause is represented in the form of BP, and that if CG is needed, it must be constructed from BP. Thus, for example, to find all literals dependent on a given literal (i.e., its neighbors in CG), using the BP representation, we must find the union of sets of literals—one set for each variable in the given literal, with each set consisting of all literals that contain that variable. Since all these sets might be identical, the time spent might be increased by a factor equal to the arity of the literal, compared to the time required using a direct representation of CG.

2. COMPLEXITY OF ACHIEVING MAXIMUM PARALLELISM

In the next two subsections we state some results concerning the run-time complexity of performing optimal scheduling of literals in logic programs so as to minimize total execution time.

2.1. Computing the Maximum Number of Data-Independent Literals

Let P be a goal containing n literals. The obvious strategy to exploit parallelism as much as possible without executing data-dependent literals in parallel is to select for execution the maximum number of data-independent literals. Unfortunately, the following result shows that this strategy is, in the worst case, impractical.

Proposition 1. The problem of determining a maximum-size set of literals, no two of which share a variable, is NP-complete. More formally, given N literals and a positive integer $K \leq N$, to determine whether there is a data-independent subset of K or more literals is NP-complete. (Hereafter, we shall not formally restate each problem as a decision problem.)

PROOF. The problem is clearly in NP. It is now easy to reduce the maximum-independent-set problem for graphs [8] to this problem. Turn nodes in the graph into distinct predicate names, and edges into distinct variables. Then construct a goal with one literal for each predicate name, and make it contain exactly those variables that represent edges incident on the node represented by the predicate name. It is easy to see that independent sets in the graph now correspond exactly to data-independent sets of literals. □

On the surface, Proposition 1 appears related to the main theorem in [12] that shows that the problem of finding the maximal unifiable subset of a set of unifiers is NP-complete. On close inspection, however, the problems are quite different.

In practice, the arity of the predicates in logic programs is often restricted to be less than some fixed constant. The next result shows that the problem above is still NP-hard even when the arity of the literals is restricted to be no greater than three.

Proposition 2. From a goal G consisting of N literals, none with more than three variables, the problem of determining if there is a subset with at least K data-independent literals is NP-complete.

PROOF. Identical to the proof of Proposition 1, except that the reduction is from the maximum-independent-set problem for cubic graphs [8]. □

In the case where no clause contains more than two variables we have:

Proposition 3. If all literals in a clause have arity 2, *a maximum data-independent set of literals can be selected in polynomial time.*

PROOF. Let each variable in the clause be regarded as a node of a graph, and each literal be regarded as an edge connecting its variables. Then a data-independent set of literals is equivalent to a set of edges no two of which are incident on the same node. Such a set of edges is called a *matching*. Thus, the arity-2 case can be reduced to finding a maximum matching in a graph, which can be solved in polynomial time. □

Note that in practical parallel interpreters we want a very fast scheduling algorithm. Even a quadratic-time algorithm is prohibitive unless the size of the clauses is small. In Section 2.3 we discuss simple heuristics which, though not guaranteed to deliver the best solution, are likely to give reasonable overall performance.

2.2. Computing Other Strategies

The strategy of finding the largest set of data-independent literals to execute in parallel may be far from the best strategy in many logic programs. It is easy to see that even if each literal in the current goal can succeed immediately, the maximum-independent-set strategy is not necessarily optimal. For example, in the program

$:- p(W, X), q(Y, Z), r(Q, W, X, Z), s(Q, X, Z).$

$p(a, b).$

$q(c, d).$

$r(e, a, b, d).$

$s(e, b, d).$

a maximum-independent-set strategy selects the literals p and q to execute first. They bind their variables to constants, but the remainder of the goal would still not be independent, so that two more execution steps would be needed, for a total of three parallel execution steps. But if, instead, r were executed first, it would succeed and bind all the variables except Y to constants, thereby removing all data dependencies. The rest of the goal could now be executed simultaneously, yielding a total of just two parallel executions steps.

This lack of optimality in scheduling literals for execution is not peculiar to the maximum-independent-set strategy. In fact, no simple scheduling strategy is likely to be optimal, since the following result indicates that the problem of finding an optimal strategy is intractable, even for simple cases like the above example.

Proposition 4. Let S be a logic program comprising ground assertions only, such that no predicate name occurs more than once in S. Let P be a goal. The problem of determining the optimal order of parallel execution of the literals in P, such that literals sharing uninstantiated variables may not execute simultaneously, is NP-complete.

PROOF. Since every literal must be executed, the order of the execution can be thought of as a partition of the literals into an ordered sequence of sets, where no two literals in the same set share an uninstantiated variable. In terms of the connection graph CG, this is almost a coloring of the nodes in such a way that nodes of the same color correspond to literals that are executed in parallel during the same step. The only difference is that it is possible at later steps to execute in parallel nodes which share variables, if earlier steps have already instantiated those shared variables. Thus, determining the chromatic number of CG is not equivalent to determining the optimum order of parallel execution. We can, however, reduce the problem of determining the chromatic number of a graph (which is known to be NP-complete [8]) to that of finding an optimum parallel execution strategy.

Given a graph G, we construct a set of literals as shown in Figure 2 by creating one literal for each node, and one variable for each edge, and having the literal contain a variable iff the corresponding edge is incident on the corresponding vertex. Now given a parallel execution strategy for the literals, the order of the steps in the strategy does not matter. This is because each variable is contained in exactly

Sample Graph:

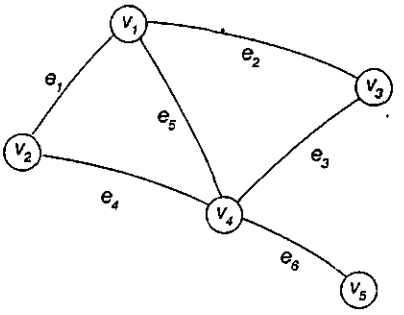


FIGURE 2. Graph and corresponding goal as in proof of Proposition 4.

Corresponding Goal:

:- v_1 (e_1, e_2, e_5),
v_2 (e_1, e_4),
v_3 (e_2, e_3),
v_4 (e_3, e_4, e_5, e_6),
v_5 (e_6).

two literals, so that when a literal instantiates a variable, it cannot affect the data dependency between any two other literals. So literals can be executed in parallel iff they share no variables at the start of execution, which is equivalent to the corresponding nodes in G not being adjacent. Thus, there is a one-to-one correspondence between node colorings of G and parallel execution strategies for the literals, where literals are executed in the same parallel step iff the corresponding nodes have the same color. Therefore the chromatic number of G is the same as the number of steps in the optimum parallel execution of the literals. Since our problem is clearly in the class NP, we are done. □

In the case where all literals have arity no greater than a constant m, we have the following easy result:

Lemma 1. Under the conditions of Proposition 4, *if* m *is the maximum arity of the literals to be executed, the parallel execution requires at most* $m + 1$ *steps.*

PROOF. Let A be a greedy algorithm which scans the goal list from left to right and selects for execution any literal that does not share a variable with an already selected literal. The time complexity of the execution sequence produced by A is no more than $m + 1$ steps. This is because, at each step, any literal not being executed must share a variable with a literal that is executing, so by the end of the step that variable will be instantiated. Thus, after a total of at most m steps, every variable has been instantiated and in one more step any remaining literals can be executed. □

Under the stated conditions, the above result guarantees a fast parallel execution for small values of m. To determine the *fastest* parallel execution, however, is NP-complete when $m = 3$, as seen in Proposition 5.

Proposition 5. The result of Proposition 4 *is still valid under the condition that all literals have arity of at most* 3.

PROOF. If all literals have arity no greater than 3, the strategy of Proposition 4 no longer works, because there is a polynomial-time solution for the problem of determining the chromatic number of graphs with vertex degrees no greater than 3. Yet, as mentioned in the proof of Proposition 4 above, the problem of determining the fastest parallel execution strategy is harder than coloring, because when variables are instantiated, dependencies among the remaining literals are "removed". We show the arity-3 case to be NP-complete by reducing from the problem of determining if a collection of 3-member sets contains an exact cover, i.e., a subcollection in which each element appears exactly once [8].

Without loss of generality, assume that no element is contained in only one of the 3-member sets (otherwise discard that 3-member set—it must be included in any exact cover). We regard the 3-member sets as literals where the elements represent variables. If there exists a two-step parallel execution strategy, then the literals executed during the first step must form an exact cover. Conversely, if the literals in an exact cover are executed, all variables become instantiated and any remaining literals can be executed together in step 2. Thus, there is an exact cover iff there is a two-step parallel execution. □

Combining Lemma 1 and Proposition 5 yields the following curiosity. For $m = 3$, using Lemma 1, we easily can obtain a 4-step parallel execution (in linear time), but the problem of determining whether there is a 2-step parallel execution is NP-complete.

The case in which no literal has more than two variables can be determined in polynomial time, as the following shows:

Proposition 6. Under the conditions of Proposition 4 with the added condition that all literals have arity 2, an optimum execution strategy can be determined in polynomial time.

PROOF. It is easy to determine if all the literals can be executed in a single step, just by seeing if there are any shared variables, and Lemma 1 guarantees the execution takes no more than three steps. Thus, the problem reduces to determining if there is a two-step execution strategy.

To determine if there is a two-step execution strategy, we convert the literals to a graph as in the proof of Proposition 3: variables corresponding to nodes, and literals corresponding to edges whose ends represent the variables contained in the literal. In this form a two-step execution corresponds to a set of edges (representing the literals to be executed during the first step) such that no two are incident on the same node (i.e., a matching) and such that every node with degree at least 2 is an end of one of the edges. The first requirement guarantees that literals executed during the first step are data-independent. The second requirement guarantees that all literals left over for the second step are data-independent, since if a degree-2 node were not an end of an edge in the matching, its corresponding variable would not be instantiated during the first execution step. Since its degree is two, there are two literals that contain it, and they could not both be executed during the second step. Thus, there is a two-step execution strategy iff there is a matching that touches each node in the corresponding graph, except possibly for degree-1 nodes.

If there are no degree-1 nodes, such a matching must touch every vertex (a perfect matching), and polynomial-time algorithms to determine the existence of perfect matchings are well known. If there are degree-1 nodes present, then the graph can be modified so that there is a two-step execution iff the modified graph admits a perfect matching. First, add a dummy node (if needed) to make the total number of nodes even. Then add a dummy edge between every two degree-1 nodes, and if there is a dummy node, add dummy edges from it to each degree-1 node. It is now clear that any suitable matching in the original graph can be extended to a perfect matching in the modified graph. Conversely, a perfect matching in the modified graph can be converted to a suitable matching in the original graph by simply disregarding all dummy edges. □

Again, it is worth noting that in this subsection we have considered just the simplest possible cases of parallel selection. It seems likely that any generalization would only increase the complexity of optimal scheduling.

2.3. *Heuristics*

For many practical purposes, a data-independent set whose size is very near the maximum might well suffice. In this case a simple heuristic method of selecting a

maxim*al* data-independent set of literals might produce a set which is often, in fact, a maxim*um*, and otherwise is close to a maxim*um*. One such method is to repeat the following steps until all literals are exhausted:

1. Select a literal with the smallest number of literals data-dependent on it.
2. Delete it and all literals dependent on it from those under consideration.

Step 2 ensures that the set of selected literals will be data-independent and maximal. Step 1 is performed to minimize the number of literals discarded in step 2, thereby leaving more literals and tending to produce a larger data-independent set. This algorithm can be executed in time proportional to the number of edges in CG. In preliminary simulations on clauses composed of randomly selected literals with bounded arity, the algorithm computed a maxim*um* independent set most of the time, while in the other cases it computed a set only one literal smaller than the maximum. For more details of these simulation results, see [6].

The algorithm can be improved somewhat by adding an extra condition to discriminate among ties in step 1. If two or more literals each have the same smallest number of literals dependent on them, select the one with the most literals dependent on its dependent literals. In terms of CG, this states that when two or more nodes have minimum degree, select the one which has the most nodes exactly 2 edges away from it. This serves to remove as much dependency as possible from the literals left after step 2 is performed. As a result, there are more possibilities for independent sets, so that larger ones can be found.

3. MAINTAINING AN ACTIVE DATA-DEPENDENCY GRAPH

As a logic program executes and literals are unified, variables can acquire bindings which affect data dependency. For example, in the goal

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

all three literals are dependent on one another. But if p is executed and binds X to a constant, the dependency between q and r is eliminated. Dependencies can also be created during the course of execution. For example, in the goal

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

the dependencies are only between p and q, and between p and r. But if p is executed and unifies with the assertion $p(Z, Z)$., both X and Y have Z substituted for them, and q and r become dependent because they now share Z.

In order to exploit dataflow analysis effectively, changes in dependency that occur in the course of program execution must be applied efficiently to the data-dependency graph representation. The general problem of incrementally updating graphs (i.e., adding and deleting edges and nodes) is studied widely in the literature (for example, see [7]). In the context of logic programming, the following changes occur in the data-dependency graph:

(1) *Node deletion* occurs when a literal succeeds and binds all its variables to constants.

(2) *Node addition* occurs when a literal unifies with the head of a clause, and the literals in the tail of that clause are added to the current goal.

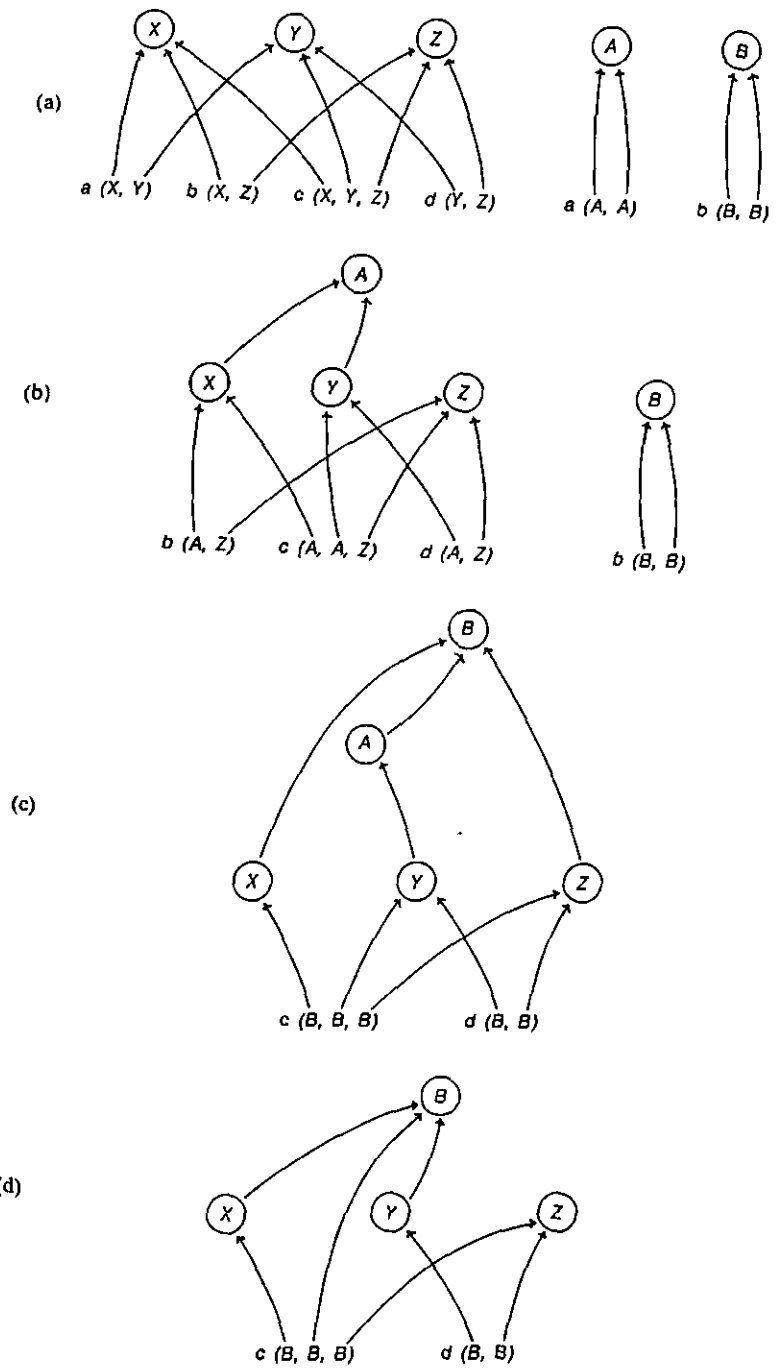


FIGURE 3. Updating the pointer representation of BP: (a) initial representation, (b) after unifying $a(X, Y)$ and $a(A, A)$, (c) after unifying $b(A, Z)$ and $b(B, B)$, (d) after a reference to the second argument in $c(B, B, B)$.

(3) *Edge addition* occurs when different variables become bound to the same variable, as in the preceding paragraph.

(4) *Edge deletion* occurs when a variable becomes bound to a constant.

These changes in dependency can be maintained in what is essentially the BP form by maintaining a chain of pointers from each argument in the literals of a clause to the current name of the argument, something like what is done in PROLOG environments. (For purposes of analyzing data dependency, chains pointing to names that are constants are not considered part of BP.) For example, in Figure 3(a) we show the pointers as they would be initially for the clauses in the program

$$:- a(X,Y), b(X,Z), c(X,Y,Z), d(Y,Z).$$

$$a(A,A).$$

$$b(B,B).$$

When a substitution is applied to an argument, the argument and each node in its chain are pointed to the new name. Moreover, for any reference to the name of an argument, each pointer in the chain is made to point directly to the end node of the chain. In this way the current names can be updated efficiently, and the pointer chains are continually compressed to prevent lengthy chain traversals. As further illustration, Figure 3(b) shows the effect of unifying $a(X,Y)$ with the simple assertion $a(A,A)$. Figure 3(c) shows what then happens when $b(A,Z)$ is unified with $b(B,B)$. Finally, Figure 3(d) shows the situation after a reference to the second argument in the literal c. (Note that the node with the name A has been dropped, since nothing now points to it.)

This pointer structure is identical to that described in [1] for UNION-FIND set operations. In our context, the sets are sets of arguments with the same current name; the FIND operation is that of determining the current name associated with an argument; and the UNION operation is that of unifying two variables. As shown in [1], for a given constant c, a sequence of cn UNION and FIND instructions can be executed in at most $c_1 n \log^* n$ steps, where c_1 depends on c and $\log^* n$ is the inverse function of $2^{2^{2^{\cdots}}}$ with n exponents of 2. Thus, BP can be maintained and processed in a way that uses essentially constant time to determine each edge. The application of UNION-FIND data structures to logic programs was also noted by Mannila and Ukkonen [10], who independently observed that the results of a sequence of unifications and deunifications can be maintained in a UNION-FIND form. Our result follows directly from the incremental graph operations generated by the execution of a logic program.

4. DISCUSSION

Originally, logic programming was proposed as a declarative method of programming that alleviated many of the low-level control considerations of other programming languages. Only recently have logic programs been used in a procedural style of programming to describe efficiently and concisely programs typical of system-level applications [3, 11]. In the context of system-level programming, it is clear that one wants to minimize to a constant the run-time complexity of analyzing dataflow

dependencies. This sometimes requires the sacrifice of flexibility and potential parallelism in favor of restricted control constructs that improve the run-time behavior [3, 5]. Our results have application to Concurrent PROLOG/Parlog-type systems. At any point during execution, a Concurrent PROLOG interpreter is trying to schedule the unification of k literals that appear in the current goal in such a way that the actual assignment of bindings to the shared variables must not be done concurrently. Now assume that we have k processors allocated to each one of the literals and they are trying to perform the unification of their respective terms in such a way that no two processors will access the same variable location simultaneously. Our results suggest that determining the optimal scheduling of this process is intractable and that the procedure currently used by the interpreter (namely first-in first-out) actually may be quite reasonable when the arity of the literals is low.

In the context of artificial-intelligence applications, where the programs are likely to be declarative (e.g. expert systems, databases), one is willing to pay a higher cost in run-time overhead to improve the logical efficiency of the system, i.e., the total number of paths in the proof tree explored by the interpreter. Thus, many researchers use dataflow analysis to minimize backtracking [4, 2, 9].

In this paper we have examined some aspects of the worst-case behavior of dataflow analysis in these kinds of systems. We have proved that dataflow analysis can be prohibitive, as previously conjectured by other researchers. We have established that scheduling goals in parallel logic programs remains NP-complete even when the arity of predicates is no larger than 3, and that it is rather difficult (but has a polynomial-time solution) for arity 2. We also have proposed several simple heuristic solutions that seem to work well in preliminary experiments.

There is much room to experiment with various restricted versions of the questions investigated here that may prove very useful in practice. We are currently investigating the application of dynamic dataflow analysis to intelligent backtracking in the context of AND-parallelism, as well as the complexity of global (interprocedural) dataflow analysis.

REFERENCES

1. Aho, A. V., Hopcroft, J. E., and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.
2. Chang, J. H. and A. Despain, Semi-intelligent Backtracking of PROLOG Based on Static Dependency Analysis, in: *Proceedings of the IEEE Symposium on Logic Programming*, Aug. 1985, pp. 10–21.
3. Clark, K. L. and Gregory, S., PARLOG: Parallel Programming in Logic, *ACM Trans. Prog. Lang. & Sys.* 8:1 (1986).
4. Conery, J. S., The AND/OR Process Model for Parallel Interpretation of Logic Programs, Ph.D. Thesis, Univ. of California, Irvine, 1983.
5. DeGroot, D., Restricted AND-Parallelism, in: *Proceedings of the* 1984 *International Conference on 5th Generation Computers*, Tokyo, 1984.
6. Delcher, A., The Complexity of Exploiting Data-Dependency in Parallel Logic Programs, M.S.E. Thesis, Dept. of Computer Science, Johns Hopkins Univ., 1986.
7. Even, S. and Shiloach, Y., An On-Line Edge Deletion Problem, *J. Assoc. Comput. Mach.* 28:1–4 (1981).
8. Garey, M. R. and Johnson, D. S., *Computers and Intractability: A Guide to NP-Completeness*, Freeman, San Francisco, 1979.

9. Kasif, S., Kohli, M., and Minker, J., Control Facilities of PRISM—A Parallel Inference System Based on Logic, in: *Proceedings of the International Joint Conference on Artificial Intelligence*, Aug. 1983.
10. Mannila, H. and Ukkonen, E., On the Complexity of Unification Sequences, in: *Third International Conference on Logic Programming*, LNCS 225, Springer-Verlag, July 1986, pp. 121–133.
11. Shapiro, E., System Programming in Concurrent Prolog, in: *Proceedings of 11th ACM Symposium on Principles of Programming Languages*, 1984.
12. Wolfram, D. A., Intractable Unifiability Problems and Backtracking, in: *Third International Conference on Logic Programming*, LNCS 225, Springer-Verlag, July 1986, pp. 107–121.