CUCS-97-84

Unification in a Parallel Environment

Stephen Taylor
Daphne Tzoar
Salvatore J. Stolfo

Columbia University
Department of Computer Science
January 1984

Submitted to: The 1984 National Conference on Artificial Intelligence.

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, Bell Laboratories and International Business Machines Corporation.

Abstract

This paper presents a method which allows standard data structure sharing techniques to be applied in a parallel environment. The techniques used are explained in the context of many parallel processors executing the same *unification* algorithm on different data concurrently. This operation can create complex structures to be transmitted between processors and naive algorithms could take an exponential amount of time to achieve the communication. The approach is compared to that advocated for the FFP machine.

To demonstrate the techniques, we describe how a linear unification algorithm may be executed locally at each parallel processor and a single chosen resolvent communicated through an interconnection network to other processors. The structures are transmitted without applying substitutions and can be recreated at the receiver.

The algorithms were developed as part of basic research related to the parallel Logic Programming System (LPS) under design at Columbia University. They have been implemented on a working prototype parallel machine, DADO1.

1. Introduction

Resolution, as first described by Robinson [13], provides a single, machine-oriented inference mechanism combining the classical inference rules and axioms of the first order predicate calculus. The Unification Algorithm [13, 14] is an important step in the refutation of a system of clauses and is of central importance in the development of programming languages based on logic formalisms [16, 12, 3].

Given two clausal expressions, unification computes a substitution of terms for variables which when applied to the expressions makes them identical. This substitution, or *unifier*, consists of a list of pairs. Each pair contains a variable

and a term which must be simultaneously substituted for the variable in order to make the expressions identical.

Consider the following example (for consistency with our other work, we use the syntax associated with Prolog [16]):

$$< g(X, Y), \quad g(f(a, b), a) >$$

The resulting unifier which makes the above terms identical is:

$$\{ X/f(a, b), \quad Y/a \}$$

Thus, unification may be viewed as a general pattern matching operation with special significance given to the logical variable. If logic formalisms are viewed as programming languages (where procedure declarations are represented by logical clauses and theorem provers are seen as interpreters [6]) the use of unification leads to pattern-directed invocation of clauses.

A particularly interesting consequence of the use of unification in logic programming languages is that partially instantiated structures may be carried along with a computation and utilized as needed. This presents a large degree of independence between goals that are set up during a computation. Coupled with the independence of logical clauses in a program, this has led to considerable interest in the field of parallel architectures [5, 11].

Provided that each processor is of sufficiently large granularity, parallel architectures, consisting of many processing elements may carry out the unification algorithm locally at each processor and communicate structures (i.e., literals, clauses and unifiers) between processors. Facilities to achieve this communication may be important to a number of proposed logic-based, parallel languages. LPS, a logic programming system under development by the DADO [15] project at Columbia University, is one such language.

2. The Cost of Unification

Unfortunately the unification algorithm may be extremely expensive to compute and the structures it generates may require an exponential amount of space to represent using a naive syntactic form. Consider the unification of the following terms [10]:

$$< g(f(X_1,X_1),\ f(X_2,X_2),\ldots\ f(X_{n-1},X_{n-1})),\ g(X_2,\ X_3,\ \ldots\ X_n) >$$

If the substitutions are represented explicitly as character sequences, the unification process will produce a result in which the length of terms grows exponentially. The value of X_n after unification contains 2^{n-1} occurrences of X_1. It should be noted that this problem bears no relation to, and is independent of, the variable renaming which is sometimes carried out in logic programming languages in order to distinguish semantically distinct variables. To emphasize this point, consider the following example:

$$< h(A,\ A),\ \ h(g(f(X_1,X_1),\ldots\ f(X_{n-1},X_{n-1})),\ g(X_2\ldots\ X_n)) >$$

The unification of the two 'h' functions involves unifying the two terms shown in the previous example.

A number of algorithms have been proposed to improve efficiency of the unification operation (e.g., [8, 9]). To economize on space, techniques for the sharing of structure can be used. Paterson and Wegman [10] propose the use of directed acyclic graphs (DAG's) in which common subexpressions are represented by a single subgraph. The algorithms they propose require a linear amount of space and take time which is linear in the number of nodes and edges in the DAG.

In order to improve the representation of logical clauses, Boyer and Moore [2] have suggested another structure sharing technique. This allows a clause to be represented as a *tuple* of information referencing and sharing structure with its parent clauses in a refutation. An essential feature of this technique is that a clause is represented without applying substitutions. Instead, a binding environment is established during unification and when a variable value is needed it is retrieved

from this environment. The binding environment for a clause is referenced from the tuple representing the clause in order to carry out this operation. The tuple representation for a clause achieves significant economy of representation compared to lists or arrays of characters.

3. Parallel Unification

Linear unification algorithms can be executed locally at each parallel processor provided that the clauses involved can be represented in the form of DAG's. This can be achieved by creating the DAG as a structure is received at a processor. This allows the creation of resolvents in linear time and space; however, if structures must be transferred between processors two problems occur:

1. The required transmission time may grow exponentially; if a naive algorithm is used and the structure of the n^{th} binding (in the previous example) is traversed and explicitly communicated.

2. The referencing environment changes; when a structure is transferred between processors, the referencing environment used for pointers within the structure changes. As a consequence, on arriving at another processor, pointers within the structure have no meaning.

Transmission can be carried out, in linear time, using a simple algorithm which applies the techniques used for structure sharing (described above) to communication. The central motivation is to only traverse and transmit a substructure once; for all following instances, simply the name of the substructure is sent. This is analogous to representing common substructures only once and not applying substitutions until necessary, in the sequential case. The following abstract recursive algorithm outlines the transmission of a logical term:

```
Sendterm(Structure)
  { if (Structure is ATOM) then transmit(Structure)
    else if (Structure is VARIABLE) then
      { send(name(Structure))
        if (bound(Structure) and not sent(Structure)) then
          { mark_as_sent(Structure)
            Sendterm(Structure.Binding)    % send VARIABLE's Binding
          }
      }
    else if (Stucture is FUNCTION) then
      { transmit(Structure.Predicate)       % send FUNCTION's Predicate symbol
        foreach Argument in Structure do
          Sendterm(Argument)
      }
  }
```

At the receiving processor, the stuctures may be recreated in a form which allows bindings to be retrieved as appropriate. The above procedure can be used to transmit terms in a resolvent, literals and clauses.

There are two primary mechanisms that may be employed to overcome the problem caused by a change of referencing environment:

1. send a symbolic representation of the structure as a stream of characters (e.g., a list)

2. send a relocatable structure

The first alternative is simple to implement but has the disadvantage that it is difficult to use additional structure sharing which may be present due to replicated ground sub-structures. For example:

f(g(a, b, c), g(a, b, c))

If a relocatable structure is sent, pointer values will not be affected by the change

of referencing environment. In this case it is possible to use Hash-Consing techniques [4, 1] to reduce the space used by structures and the information which must be communicated. When dealing with list representations, Hash-Consing places the responsibility for maintaining distinct structures with the Cons function. If the result of a structure manipulation results (using a hash function) in a structure already present in the system, a pointer is used rather than creating a duplicate structure. This mechanism would cause the above 'g' structure to be represented and communicated only once.

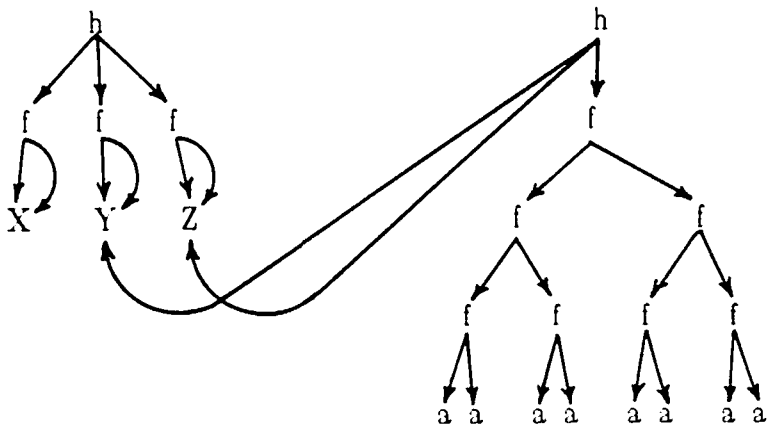
As an expedient, in the implementation we describe, character sequences were used to communicate structures. The use of relocatable structures is the subject of ongoing research by researchers building system tools for the DADO project [17].

4. Unification Example

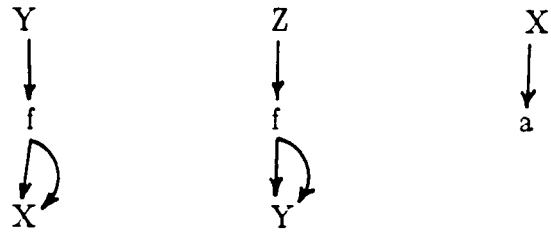
To illustrate how the above techniques are applied, consider the unification of the following expressions:

$$< h(f(X,X),\ f(Y,Y),\ f(Z,Z)),\ h(Y,\ Z,\ f(f(f(a,a),\ f(a,a)),\ f(f(a,a),\ f(a,a)))) >$$

The following DAG's are created locally at a processor to represent these structures:



The binding structures constructed as a result of unification are:



When the above structures (DAGS) are transmitted between processors they are traversed and sent using the 'sendterm' algorithm described previously. In the case of a literal the algorithm takes the literal as an argument, in the case of a set of bindings, it is used to transfer each binding individually. The following character stream would result when the above resolvent is communicated (we omit typing information for clarity, ':' is a separator):

{ Xa:Y(f X X):Z(f Y Y): }

If the structure had been traversed and sent explicitly the following character stream would have been communicated:

{ Xa:Y(f a a):Z(f((f a a) (f a a))) }

When the result arrives at the receiving processor it is reconstructed and stored in the same form as used by the sender. If the bindings for variables are needed (e.g., if they are to be printed or retransmitted) they would be retrieved and used in substitutions appropriately. For example, the following bindings would be printed as a result of the above example:

X = a
Y = f(a,a)
Z = f(f(a,a),f(a,a))

The algorithms have been successfully implemented and verified on a working prototype parallel machine, DADO1, using the Paterson and Wegman unification algorithm.

5. Related Research

Mago [7] has described an interesting approach to the problem of unification in a parallel environment for the FFP machine. The FFP machine is fine-grain parallel system obtained by interconnecting many simple processing elements in a regular pattern (binary tree). It uses the computational model of string reduction and disperses both code and data in a program to leaf cells of the machine, one symbol per processor. The Paterson and Wegman unification algorithm may be implemented on the FFP machine by applying various functional programming machine primities in order to reduce the expressions in linear time.

The essential difference between the approach advocated by Mago and that described in this paper is one related to the granularity of the systems. In the FFP machine a single instance of a unification algorithm operates by the cooperative actions of a number of processors thus achieving parallelism. Since DADO uses significantly larger processing elements, many instances of the unification algorithm operate concurrently, each at an individual processor. These use a conventional sequential algorithm and all data structures are held locally. Parallelism is achieved by executing many unifications in parallel and these are carried out at arbitrary processing elements in the system.

6. Conclusions

We have outlined simple techniques that allow a unification algorithm to be executed on multiple processors in parallel. The central problems relate to the passing of structures between processors. By applying commonly used structure sharing paradigms to the domain of communication this can be carried out in linear time. The essential techniques are to transmit common substructures only once and not to apply substitutions to expressions.

For the purposes of expediency, a number of improvements to the algorithms relating to the nature of the representation used have not yet been made. Improvements require the use of relocatable structures, which form part of our ongoing research.

The method has been demonstrated on a working, 15 processor, prototype machine, DADO1. The algorithms used are part of basic research aimed at the implementation of a logic based programming system (LPS) for the machine.

Acknowledgments

This work has greatly benefited from the ideas of Gerald Maguire Jr. who although absent during the implementation was an integral part of its inception. The authors also wish to extend their thanks to the other members of the DADO project whose help and comments have been received and are greatly appreciated.

References

1. Allen, J. *Computer Science Series.* Volume : *Anatomy of Lisp.* McGraw Hill, 1978.

2. Boyer, R. S. and Moore, J. S. The sharing of structure in theorem-proving programs. In *Machine Intelligence*, Edinburgh University Press, 1972, pp. 101-116.

3. Clark K. L., McCabe F. G. and Gregory S. IC-PROLOG Language Features. In *Logic Programming*, Academic Press, 1982, pp. 243-266.

4. Goto, E. Monocopy and Associative Algorithms in an Extended Lisp. Information Science Laboratory, May, 1974.

5. Doug DeGroot (Program Chairman) (Ed.). *1984 International Symposium on Logic Programming.* IEEE Computer Society Press, Bally's Park Place Casino, Atlantic City, New Jersey 08401, 1984.

6. Kowalski, R. A. Predicate Logic as a Programming Language. IFIP Congress, 1974, pp. 569-574.

7. Mago, G. "Data Sharing in an FFP Machine." *Conference REcord of the 1982 ACM Symposium on Lisp and Functional Programming Vol. 1* (August 15-18 1982), pp. 201-207.

8. Martinelli, A. and Montanari, U. "An Efficient Unification Algorithm." *ACM Transactions on Programming Languages and Systems 4*, 2 (April 1982), 258-282.

9. Martinelli, A and Montanari, U. Theorm proving with structure sharing and efficient unification. Tech. Rept. S-77-7, Istituto di Scienze del'Informazione, University of Pisa, February, 1977.

10. Paterson, M. S. and Wegman, M. N. "Linear Unification." *Computer and System Sciences Vol. 16* (1978), pp. 158-167.

11. Pereira, L. M., Porto, A., Monteiro, L. and Filgueiras, M. (Ed.). *Proceedings of Logic Programming Workshop '83.* Universidade Nova De Lisboa, Praia da Falesia, Algarve/Portugal, 1983.

12. Robinson , J. A. and Sibert, E. E. LOGLISP: Motivation, Design and Implementation. In *Logic Programming*, Academic Press, 1982, pp. 299-313.

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

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

15. 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.

16. 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.

17. 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.