PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

RESTRICTED AND-PARALLELISM

Doug DeGroot
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598

ABSTRACT

A method of compiling Prolog clauses into single execution graph expressions is presented. These graphs are capable of expressing potential and-parallelism in the clauses. The run-time support is minimal: simple and efficient tests suffice to detect potential and-parallelism and no recalculation of the execution graphs is required. Some parallelism may, however, be overlooked. An execution model is presented which hopefully overcomes these limitations.

AND-PARALLELISM

And-parallelism in logic programming involves the simultaneous execution of subgoals in a clause. Whereas or-parallelism attempts to achieve increased speed by investigating many possible solutions in parallel, and-parallelism attempts to achieve increased speed by investigating the subparts of a particular solution in parallel. Many models of parallel logic programming attempt to exploit both of these forms of parallelism.

Because subgoals within a clause can share variables, variable binding conflicts can arise in and-parallelism if all subgoals within a clause are allowed to execute unconstrained and if two concurrently executing subgoals attempt to instantiate (bind) a shared variable to two different values. Clearly these binding conflicts must be prevented. Two major approaches have generally been taken to solve this problem. The most common approach, perhaps, involves some method of annotating variables, as in Parlog [Clark], IC-Prolog [Clark2], Concurrent-Prolog [Shapiro], and Epilog [Wise]. In these schemes, variables are annotated in a variety of ways to indicate which subgoals can bind values to specific variables and which cannot. In particular, only one subgoal is allowed to bind a value to each variable. Such subgoals are called "producers" of those variables. In contrast, Conery describes a sophisticated, non-annotated, process-structured system that dynamically monitors variables and continually develops data dependency networks to control the order of execution of subgoals, never allowing two potential producers for the same variable to execute in parallel [Conery]. In this scheme, a particular subgoal may be a producer of a variable in one invocation and a consumer of that variable in another.

Without the benefit of variable annotations, and-parallel execution of logic programs can lead to highly-complex coordination problems at run-time. Because of these problems, parallel logic programming schemes that implement non-annotated and-parallelism generally incorporate sophisticated control mechanisms to ensure that these problems either do not arise or are corrected. If these control mechanisms are not compiler-related but instead are code segments that must be repeatedly executed while interpreting a logic program, they may detract from the performance gains achieved by the and-parallelism, and could even produce negative gains. It is important, therefore, to search for methods of achieving and-parallelism which rely mostly on compiler-related technology instead of run-time technology. This paper presents one such method.

NON-ANNOTATED AND-PARALLELISM

In this section, problems with achieving and-parallelism without annotations are described. The main problem is the possibility of creating binding conflicts for variables. Given a clause such as

```
f(X) <- p(X) & q(X).,
```

if f is called with an unbound variable argument, then X remains uninstantiated when we invoke both p(X) and q(X) in parallel. During execution, p and q may produce solutions with two different values of X. Clearly we cannot return either value of X as the result of f(X) until one of the values has been proven by the other goal. But this may be impossible for either of the two values, and it may be necessary to reinvoke both p and q in order to derive new values of X. In general, this scheme might be viewed as repeatedly reinvoking both p and q, deriving two complete sets of answers - one for p and one for q.

The value of f then is the intersection (join) of the two sets that are produced by p(X) and q(X). It is significant that this set can be computed either dynamically or statically, and that if it is computed dynamically, the sets of answers produced by all goals in the computation can be "streamed" back as they are computed, forming a pipelined computation mechanism.

If the set of answers for p(X) is small and contains the solution X=a but the set of answers for q(X) is very large and does not contain X=a, then this "join" scheme may waste processor time by producing the entire large set for q, while f(X), the parent goal, waits to see if q(X) is ever going to produce the value X=a. Instead, as the set of values for p(X) is produced, we can forward each value from p(X) to q(X), making sure that q works only on values produced by p. The set of answers returned by f(X) will then be the set of answers returned by q(X). The potential for execution economies is significant. In addition, this "forwarding" technique can effectively take advantage of goal reordering to achieve further economies [Warren], whereas the "join" technique cannot. This scheme of "forwarding" values from subgoal to subgoal has been extensively proposed and studied.

Notice that now subgoals within a clause do not produce conflicting sets of variable bindings. Parallelism occurs because while one goal is operating on its argument, the preceding goal is "preproducing" a subsequent value to be tested if necessary. But unfortunately, if a program consists mostly of deterministic procedures and clauses, little parallelism will be exhibited by this "forwarding" technique. In this case, the "join" technique would appear to exhibit greater parallelism as well as higher efficiency. If many answers exist, however, the "join" technique exhibits great parallelism but low efficiency. In practice, the "join" technique may prove impractical [Conery].

Notice that in the "forwarding" scheme, for a given potential solution only one subgoal is ever in execution at any one time. Many subgoals within a clause may be in execution at the same time, but if so, they will be working on different potential solutions. Consequently, production of a single, specific solution is always the result of a sequential execution. This is the reason simple "forwarding" schemes exhibit little parallelism when executing largely deterministic programs. To derive parallelism within the production of a single solution, some way must be found to decompose the list of subgoals into sets of parallel subgoals.

Conery's method does exactly this. His method involves a set of elaborate run-time algorithms which dynamically compute parallel execution graphs based on data dependencies between subgoals. It can execute in parallel both deterministic and non-deterministic programs. While Conery's scheme nearly always achieves optimal and-parallelism, it does so at considerable expense because of the complexity of the supporting run-time algorithms. This expense may be so high as to render the scheme impractical. However, if techniques can be found to shift more of the algorithmic burden to the compiler and to significantly reduce the amount of run-time computation, and-parallel "forwarding" schemes may prove to be a practical solution to the problems of and-parallelism.

VARIABLE BINDING CONFLICTS

When considering a program clause such as

f(X) <- p(X) & q(X).,

it is generally impossible at compile time to determine whether or not p(X) and q(X) can execute in parallel without creating a binding conflict. If f is called with a ground argument (any term not containing a variable), such as in the call "f(4)", then the two subgoals become at run-time p(4) and q(4). Since the argument to f contains no variables, no binding conflicts can arise by the parallel execution of p and q. If however f is called with an unbound argument or with a non-ground argument (a term containing one or more variables), then p and q will share at least one variable, and hence the potential for binding conflicts exists if p and q are executed in parallel. Consequently, in this case, p and q must execute sequentially. The important point is that some sort of run-time test is needed in order to determine whether p and q can execute in parallel or if they must execute sequentially.

Consider now the clause

f(X,Y) <- p(X) & q(Y).

Here it appears that X and Y are independent variables and that p and q can execute in parallel. This is far from certain, however. For suppose a call to f is made similar to one of the following:

```
f(Z,Z).
f(Z,g(Z)).
f(g(Z),h(2,Z)).
```

Then at run-time X and Y will be aliases of each other or will have values that share at least one variable. If p(X) and q(Y) execute in parallel, then the potential for binding conflicts exists. Run-time tests again become necessary.

The situation can be even more complicated. Consider the clause

f(X) <- p(X) & q(X) & s(X).

If at run-time f is called with a ground argument, as in the call "f(4)", then all three subgoals can execute in parallel. But if f is called with a non-ground argument, p(X) must first execute while q(X) and s(X) wait. Upon completion of p(X), if p has instantiated X to a ground term, then q(X) and s(X), the two remaining subgoals, can execute in parallel. But if X is still non-ground after completion of p, then q(X) and s(X) must execute sequentially. Thus depending on the call and execution, three different execution graphs result. Figure 1 illustrates the three graphs.

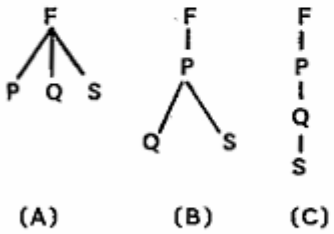

Figure 1. EXECUTION GRAPHS

Conery uses five run-time algorithms to monitor execution of subgoals in a clause and to dynamically determine the execution graphs: the Literal Ordering Algorithm, the Forward Execution Algorithm, the Backward Execution Algorithm, the And-process Algorithm, and the Or-process Algorithm. Although these algorithms are quite expensive, they do achieve nearly optimal detection of and-parallelism.

In the next section, a method is presented which determines a much more restricted form of and-parallelism. An important advantage of this method is that it requires much simpler run-time support. It will, however, occasionally fail to detect some potential for parallelism due to the fact that it computes only one execution graph, and this is done at compile-time using incomplete information. Later it will be argued that this may in fact be an advantage. Several simple algorithms are used in this scheme, the most complex of which is performed only at compile-time; the run-time tests are quite simple. Each is described below.

THE TYPING ALGORITHM

This section describes a simple algorithm for determining when two or more terms are independent (share no common variables) or are interdependent (share at least one common variable). Actually, the algorithm can only accurately determine when two or more terms are clearly independent. Otherwise, the algorithm will assume, perhaps incorrectly, that the terms are interdependent. How this affects the parallel execution of a program is described later.

Each term is allocated a special type field. A term can have one of three types:

1 - a ground term (contains no variables; this type includes integers, strings, and complex ground terms)

2 - a non-ground, non-variable term (i.e., a term with a known principle functor but which contains at least one inner variable)

3 - a variable (i.e., an uninstantiated variable)

These types and the special type field are in addition to any other type or tag fields usually found in data representations, such as integers, lists, strings, etc. [Warren2]. Most terms appearing in the source program can easily have their type codes preset by the compiler. For instance, all integers, constants, and complex ground terms can have their type codes preset to 1; local variables (those not appearing in the head of a clause) can be preset to type 3; and finally, all structured source code terms containing one or more variables will be preset to type 2. In a structured term, not only is the term itself preassigned a type, but so are all components of the term at all levels. Note that all variables appearing in the head of a clause cannot have their types determined at compile-time since their values will not be known until run-time. Variables in queries, however, will be preset to type 3.

Determining the type of a term at run-time is slightly more difficult. We clearly cannot afford the run-time expense of traversing several entire structures on each procedure invocation or exit in order to determine if a structure is ground or if two structures share a common variable. Instead, an efficient approximation technique is desired. Such a technique is now presented.

First, consider how variables (type 3) can inherit type 1 or type 2 codes. Consider the following program for append:

```
append(nil,L,L).
append(X.Y,Z,X.L) <- append(Y,Z,L).
```

and the call

```
append(nil,a.b.c.nil,R)?.
```

The compiler will have preassigned type code 1 to both nils and to the list a.b.c.nil and type code 3 to the variables R and L. Since the call unifies with the first append clause, R becomes bound to the list structure through unification, and the type code of R is then changed from 3 to 1. This example shows how output variables can inherit type 1 or type 2 codes from input variables through the normal unification procedure.

Now consider the call

append(a.nil,b.c.nil,S)?.

This call unifies with the head of the second append clause, with substitution

(a/X,nil/Y,b.c.nil/Z,X.L/S).

The compiler will have preset X.Y and X.L to type 2. At run-time X, Y, and Z will all inherit type codes of 1 and S will inherit a type code of 2. As terms are being constructed or decomposed in unification, any whose type code is still 2 after unification (meaning non-ground) has its address placed in a special "pending" list. Following the successful termination of all subgoals in the clause, all structures on the pending list are investigated to see if they have become ground terms. This is done simply: the type codes of all the top level components in a pending structure are checked to see if they are ground (or have become ground) terms (have type codes of 1). If so, then the structure on the pending list has also become ground, and its type code is now set to 1.

In the append example above, it is easy to see that when the subgoal completes, L will be bound to "b.c.nil", and since this is a ground term (whose type code was preset to 1 by the compiler), L will inherit a type code of 1, as in the first example. Before returning control from the top-level call to append, the argument S, whose address has been placed on the pending list and whose value is X.L, will be checked. Now, because both X and L will have type codes of 1, the type code of S is changed from 2 to 1, meaning it is now ground. The fact that the list "b.c.nil", the value bound to Z, is ground and has type code 1, could not only have been the result of a compile-time determination but also the result of an input action or a similar term construction sequence.

Consider this example of "finishing" a constructed symbol table, as described in [Warren]:

```
finish_st(nil).
finish_st(sym(S,Lson,Rson)) <-
        finish_st(Lson) &
        finish_st(Rson).
```

Here the leaves of the tree will all be set to nil upon completion of the program. As execution of each clause completes, the subtree represented by the head of the clause will have become ground, and the type code of the structure can be set to 1. Upon completion of the procedure, the entire tree will be ground, have a type code of 1, and contain only components whose type codes are 1.

It is of course possible that some structures might be constructed which are at first non-ground but which later become ground and yet which escape detection. For example consider:

f(X) <- find_a_var(X,V) & V=nil.

Here, assume that X has a variable in it several layers deep and that find_a_var binds it to V. When V is set to nil, we exit. X is taken off the pending list and its top level components are inspected. The top level component which contains V will still have a type code of 2 (non-ground) and so the type code of X will not be changed to 1, even though now X is ground. It is possible that later some traversal program might discover that X is in fact ground and so set the type code to 1, but this is far from certain. This escape of X's might result in some loss of parallelism, as will be explained below. However, due to the way goals are distributed for parallel execution, this loss is anticipated to be low, perhaps even insignificant.

THE INDEPENDENCE ALGORITHM

Given the clause

f(X) <- g(X) & h(X) & p(X).,

a compiler might assume all three subgoals are interdependent and so must be executed sequentially. However, to see if the three subgoals can be executed in parallel, all that must be done is to check at run-time whether or not the actual parameter to f is ground and thus has a type code of 1. If so, then g, h, and p can all execute in parallel. If not, then we assume that they must execute sequentially. However, after execution of g(X), we can again check the type code of X. If it is now 1, then we can execute the two remaining subgoals in parallel.

Consider the clause

f(X,Y) <- g(X) & h(Y).

Here, the compiler would identify g(X) and h(Y) as being potentially independent subgoals and emit appropriate type checking code which at run-time attempts to verify the independence of X and Y. If they are independent, then g(X) and h(Y) can execute in parallel. If the test fails, then g and h are executed sequentially. The Algorithm used to test for the independence of two parameters is shown in Figure 2. Tests for more than two variables are similar but involve O(n2) tests, where n is the number of arguments appearing in the clause head. Because n will usually be very small, and because the test algorithm is so very simple, the actual overhead is minimal.

```
IF TYPE(ARG1) = 1 OR TYPE(ARG2) = 1
    THEN INDEPENDENT ELSE
IF TYPE(ARG1) = TYPE(ARG2) = 3 AND
   ADDRESS(ARG1) ¬= ADDRESS(ARG2)
    THEN INDEPENDENT ELSE
(ASSUME) DEPENDENT;
```

Figure 2. INDEPENDENCE ALGORITHM.

PROGRAM EXECUTION GRAPHS

Two utility predicate routines are provided for testing terms at run-time: GPAR(X1,...,Xn) and IPAR(X1,...,Xn). The value of GPAR(X1,...,Xn) is true if all of the arguments to GPAR are ground terms (type code 1). If any argument is non-ground, the value of GPAR is false. Similarly, the value of IPAR(X1,...,Xn) is true if all its arguments are mutually independent; but if any two are interdependent, the value of IPAR is false. GPAR simply checks the type code of each of its arguments to ensure that they are all 1; IPAR uses the Independence Algorithm. Both take an arbitrary number of arguments; but IPAR requires at least two.

Six types of execution expressions are allowed:

1. G
2. (SEQ E1 . . . En)
3. (PAR E1 . . . En)
4. (GPAR(X1,...,Xk) E1 . . . En)
5. (IPAR(X1,...,Xk) E1 . . . En)
6. (IF E1 E2 E3)

G is an arbitrary goal (procedure call). An SEQ expression indicates that the following expressions are to execute sequentially, in presentation order, while a PAR expression indicates that they are to execute in parallel. A GPAR expression indicates that if all the arguments of the GPAR function call are ground terms, then the following expressions are all to execute in parallel; but if any one of the arguments is not ground, then the expressions are all to execute sequentially, in order. Similarly, an IPAR expression indicates that if all the arguments of the IPAR call are mutually independent, then the following expressions are all to execute in parallel; but if any two arguments are interdependent, then the following expressions are to execute sequentially, in order. The IF expression chooses between two alternative actions based on the evaluation of the boolean expression E1, choosing E2 if true and E3 if false.

All program graphs will be constructed from these six expression types. Consider the clause

```
f(X) <- p(X) & q(X) & s(X).
```

This clause can be compiled into several different program execution graph expressions, some of which are:

```
f(X) = (SEQ p(X) q(X) s(X)).

f(X) = (GPAR(X) p(X) q(X) s(X)).

f(X) = (GPAR(X)
          p(X)
          (GPAR(X) q(X) s(X))).
```

The first allows no possibility of parallelism and is equivalent to execution graph c of Figure 1. The second allows all three subgoals to execute in parallel if X is ground, and therefore to achieve execution graph 1a; but if X is not ground, execution reverts back to graph 1c. If X is ground, the third expression will achieve the maximal parallelism of graph 1a. If X is not ground, however, this third expression will first execute p(X) and then retest X. If X is now ground, q(X) and s(X) will execute in parallel, as in 1b; otherwise they will execute sequentially, as in 1c. Thus the third expression is capable of achieving any of the three execution graphs of Figure 1.

Several important points should be noted:

1. First, only one execution graph expression is created for the clause. This graph is created at compile-time. No reordering of goals occurs at run-time which can lead to the need to dynamically create an alternative execution graph. The tests made by IPAR and GPAR and the typing algorithm are all very simple and inexpensive.

2. To achieve execution graph 1a, the third expression above actually makes a redundant test to see if X is ground. Fortunately, the amount of additional overhead introduced by this redundant test is small since the GPAR(X) test is so simple.

3. Even though under ideal circumstances the third expression can achieve maximal parallelism, it may occasionally fail to find parallelism due to the approximation technique of the typing algorithm. Consequently, in this scheme, all three goals might execute sequentially while in Conery's scheme they would execute in parallel.

As another example, consider the clause

```
f(X,Y) <- p(X) & q(Y) &
          s(X,Y) & t(Y).
```

The compiler will produce the following execution graph expression:

```
(GPAR(X,Y)
     (IPAR(X,Y) p(X) q(Y))
     (GPAR(Y) s(X,Y) t(Y)) )
```

If X and Y are independent at run-time but not ground, the execution graph of Figure 3 will be achieved. Conery's scheme will also achieve this same execution graph. But

Conery's scheme has one advantage - as soon as q(Y) completes, t(Y) can begin execution, even if p(X) is incomplete. In the scheme presented here, t(Y) will have to wait until the first two subgoals complete. This is a result of the definition of an SEQ expression. This loss of parallelism is due to the limited execution graph expressions and is not a result of the approximation technique of the typing algorithm. Consequently, there are two ways in which the and-parallelism of a clause may be restricted.

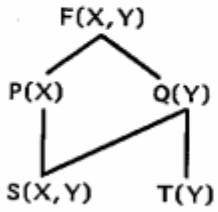

Figure 3. EXECUTION GRAPH

Finally, consider the following quicksort program and the compiled execution expression:

```
qksort(L,SL) <-
    partition(L,L1,L2) &
    qksort(L1,SL1) &
    qksort(L2,SL2) &
    append(SL1,SL2,SL).


(SEQ
   partition(L,L1,L2)
   (IPAR(L1,L2)
      qksort(L1,SL1)
      qksort(L2,SL2) )
   append(SL1,SL2,SL) )
```

A PARALLEL EXECUTION MODEL

In this section a parallel execution model is presented for executing the graph expressions. The model is intended to run on a tightly-coupled parallel architecture with each processing element containing its own large, local memory. Each processor is assumed to have a copy of the entire program resident in its memory (or at least to have rapid paging access to the program).

When all processors are busy executing pieces of a large program, it is not necessarily beneficial for one processor to decompose its own piece of work into two or more pieces since it may end up having to execute them both anyway. The busier the processors are, the less beneficial a task decomposition is likely to be. We adopt the following views in the model:

1. When all processors are busy doing useful work, it is not necessarily beneficial to distribute work among them.

2. When a processor executes a clause, it assumes it will have to execute all subgoals in the clause by itself (that is, it assumes all other processors are busy doing useful work).

3. Subgoals will be distributed to other processors only when those processors have volunteered to help (when they have asked for work). This will usually occur only when the requesting processor is idle (has no goals to execute) or at least is lightly loaded (perhaps all its goals are suspended).

4. Before distributing a subgoal set, we try to ensure that it is nontrivial.

Because it may frequently prove non-beneficial to dynamically decompose a task when all processors in the system are busy, it may similarly prove non-harmful if an occasional opportunity for parallelism is not detected. Consequently, the loss of maximal detection of parallelism due to the approximation technique of the typing algorithm and as a result of the limited execution graph expression types is believed to be acceptable. The return is in the extreme simplicity of the run-time component of the model.

Each processor maintains two expression stacks - a sequential stack and a parallel stack. The sequential stack contains expressions that must be - or have been decided to be - executed sequentially; the parallel stack contains expressions that can be executed in parallel. When a user query is read, it is converted into an execution graph expression and placed on the sequential stack of some processor. Each processor checks the top of its sequential stack to obtain its next piece of work. If the expression on top of the stack is a goal, that goal is executed as in normal Prolog systems. If it is an IPAR or GPAR expression, the specified test is performed, and if the result is true, the following subexpressions are all placed on the parallel stack. If the result is false, the sub-expressions are all placed on the sequential stack. An SEQ expression simply puts the following list of sub-expressions back on the sequential stack, while a PAR expression puts its sub-expressions on the parallel stack. The actual entries on the stacks will, of course, be pointers to expressions and not the expressions themselves.

If the sequential stack becomes empty, the processor takes an expression off the parallel stack and executes it. However, if the parallel stack is also empty, the processor can volunteer to help some other busy processor.

When a busy processor receives an offer to help from some idle processor, the busy processor can, if it wants, check to see if it has any entries on its parallel stack. If it does, it can select any one of these and assign it to the volunteering processor. It should be clear that entries on the parallel stack can be executed in any order in parallel. Entries on the sequential stack, however, must be executed sequentially, in order, in a "last-in, first-out" manner.

As the system becomes busy, requests to help from volunteers will decrease, and the number of entries on each processor's parallel stack will tend to increase. If a parallel stack reaches some "high" level, the processor may opt to process a potentially parallel expression sequentially. By so doing, any extra overhead in performing the test, decomposing the expression, pushing sub-expressions onto the parallel stack, and popping them back off at some later time can be avoided. Further, by executing a group of several related subgoals, instead of a string of separate subgoals, the architectural and software implementations can potentially take advantage of the many aspects of program locality (e.g., [Tick]).

BACKTRACKING

Backtracking will behave as described in [Chang]. Space limitations prevent a detailed description of the scheme here. A very important advantage shared by this scheme is that all backtrack points can be computed at compile-time. Each goal can have at most two backtrack points. Goals in SEQ or PAR expressions have a single backtrack point; but goals in IPAR and GPAR expressions have two - one used when the IPAR or GPAR expression is executed in parallel and one for when the expressions are executed sequentially. This particular backtracking scheme achieves some of the efficiency of intelligent backtrack schemes [Pereira].

USER-SPECIFICATION OF PARALLELISM

Because of the limitations of the execution graph expressions, certain clauses may be compiled into sequential graphs when with only minor changes they could be compiled into parallel graphs. For example, the following clause will be compiled into the given execution graph expression:

```
f(X) <- g(X) & h(X)

(GPAR  g(X) h(X))
```

This expression will execute in parallel only if X is ground. If the clause is rewritten as

```
f(X) <- g(X) & h(Y) & X=Y.,
```

the following expression is compiled

```
(SEQ (PAR g(X) h(Y)) X=Y).
```

The second clause clearly has potential performance and semantic dif- ferences from the first. But if the programmer is aware of these differences and considers them non-harmful, he may choose the second clause in an effort to establish greater parallelism at run-time. This second clause can achieve parallelism irrespective of whether X is ground or non-ground.

As another example, consider the following, more efficient quicksort program. This program uses difference lists to avoid the calls to the append program.

```
qksort(L,SL-Rest) <-
        partition(L,L1,L2) &
        qksort(L1,SL-T) &
        qksort(L2,T-Rest).
```

Unfortunately, the restricted and-parallel method presented here will fail to detect any potential for parallelism. But if the program is changed to

```
qksort(L,SL-Rest) <-
   partition(L,L1,L2) &
   qksort(L1,SL-T1) &
   qksort(L2,T2-Rest) &
   T1=T2.
```

then the following parallel execution graph is obtained:

```
(SEQ partition (L,L1,L2)
     (IPAR (SL,Rest,L1,L2)
           qksort(L1,SL-T1)
           qksort(L2,T2-Rest) )
     T1=T2)
```

How frequently the programmer will be able to assist the compiler with this type of change remains to be seen.

SUMMARY

A method for obtaining restricted and-parallelism in logic programs has been presented. The method involves the compile time creation of a parallel execution graph expression for each program clause. Only one expression per clause is created. The run-time algorithms involved are simple and inexpensive. Due to the limitations of the graph expressions and to the approximation technique of the typing algorithm, an opportunity for parallelism may occasionally be missed. However, a parallel execution model that utilizes demand-driven distribution of work may provide efficient, parallel execution of the graphs in a manner that renders these misses harmless.

BIBLIOGRAPHY

[Chang] "And-Parallelism of Logic Programs Based on Static Data Dependency Analysis," Jung-Herng Chang and Doug DeGroot, in preparation.

[Clark] "PARLOG: A Parallel Logic Programming Language," Keith L. Clark and Steve Gregory, Research Report DOC 83/5, Imperial College, March 1983.

[Clark2] "The Control Facilities of IC-Prolog," Keith L. Clark and Frank McCabe, in Expert Systems in the Microelectronic Age, D. Michie, editor, Edinburgh Univ. Press, 1979.

[Conery] The AND/OR Process Model for Parallel Execution of Logic Programs, John S. Conery, Ph.D. dissertation, Univ. of California, Irvine, Tech. Report 204, Information and Computer Science, 1983.

[Pereira] "Selective Backtracking," Luis Moniz Pereira and Antonio Porto, in Logic Programming, Keith L. Clark and Sten-Ake Tarnlund, editors, Academic Press, 1982, pp.107-114.

[Shapiro] "A Subset of Concurrent Prolog and Its Interpreter," Ehud Y. Shapiro, ICOT Tech. Report TR-003, February, 1983, Tokyo, Japan.

[Tick] "Towards a Pipelined Prolog Processor," Evan Tick and David H.D. Warren, Proc. of the 1984 International Symposium on Logic Programming, IEEE, pp. 29-40.

[Warren] Applied Logic - Its Use and Implementation as a Programming Tool, David H.D. Warren, Ph.D. dissertation, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1977.

[Warren2] "An Abstract Prolog Instruction Set," David H.D. Warren, Technical Note 309, October 1983, SRI International.

[Wise] "A Parallel Prolog: the Construction of a Data Driven Model," Michael J. Wise, Univ. of New South Wales, Australia.