AND Parallelism in Logic Programs

John S. Conery
Dennis F. Kibler

Department of Information and Computer Science
University of California. Irvine
Irvine. Ca. 92717[1]

Abstract

An interpreter for logic programs is defined which executes some goals in parallel. OR parallelism exploits the parallelism defined from nondeterministic choices, and is essentially a replacement for backtracking. AND parallelism comes from solving goals in the body of a single clause in parallel, and is the only way to exploit parallelism in deterministic functions written as logic programs. A unique feature of our model is that it allows both forms of parallelism for the same computation.

1• Introduction

In this paper we present a method for exploiting AND parallelism within the context of the AND/OR Process Model [4]. This paper focuses on the division of a logic program into pieces that can be executed in parallel. We are concerned with the management of abstract processes -- their creation. termination, and communication. We postpone dealing with the significant problems of effective distribution of processes to processing elements (PEs) and the physical structure of the interconnection of PEs and the resulting communication times. At this point, we assume only that each PE will have a large amount of local memory, sufficient to store some number of processes and its own (unchanging) copy of the entire program. A more complete description of the AND/OR Process Model and AND parallelism can be found in Conery's dissertation [5].

[1]This research was supported by the Naval Ocean Systems Center under Contract N00123-81-C-1165.

2. Sources of Parallelism

A logic program is a set of clauses of the form p_0 <- P1 ^ . . . A p_n, where each p_i is a literal. The literals $p_{i\#}$ i > 0. form the body of the clause and the literal p_Q is the head of the clause. A computation in a resolution based logic programming system is a sequence g_0 . . • g_m. of goal statements. each of which is a set of literals. The derivation of statement g_{i+1} from g_i requires two choices: first, select a literal p_g in g_i. then find a clause which has a head that can be unified with p_g using unifier 0. Statement g_{i+1} is the resolvent of g_i and the selected clause, and contains all literals (except p) from g_i. plus the literals from the body of the selected clause [8].

Normally, the choice of p_g is fixed. For the second choice, if there is more than one clause with a head that unifies with p . then the current statement is a choice point, and a tree of goal statements is defined. The typical sequential control in logic programming systems (e.g. [6]) is a depth first search for an occurrence of the null clause in this tree.

One possible parallel control method is a parallel search of the tree: at each choice point. distribute the goal statements at descendant nodes to interpreters running on independent computers. The expected speedup in execution will be obtained if one of these interpreters derives the null clause more quickly than an interpreter that performs a simple depth first search. The amount of time required (ignoring inter-computer communication times) will be directly proportional to the length of the shortest path in the search tree from the root to a null clause. The amount of time required by a depth first interpreter is proportional to the sum of path lengths in all branches to the left of the first branch that contains the null clause.

The search tree for a deterministic program has exactly one null clause* which is typically at the end of the leftmost branch. Parallel search cannot speed up the execution of one of these programs. An interpreter based on AND parallelism is able to effectively shorten the length of this chain of derivations* and as a result obtains the same sort of parallelism for deterministic logic programs that is seen in other functional programming languages [1]. The outline of one such interpreter is presented here* in the context of the AND/OR process model.

3. The AMD/OR Process Model

Our parallel model is based on the notion of independent processes which communicate via messages. An OR process is basically an independent interpreter, created to solve a goal statement that contains exactly one literal. When solutions are obtained, they are communicated to the process that created the OR process in a success message. For example* if the literal to be solved is * sum(3,4.X)', the response will be 'success(Bum(3,4,7))'. If no solutions are possible, or after all solutions have been sent* the OR process sends a fail message.

An AND process is created to solve a conjunction of goals p_1 A , , , A p_n, where n > 1. When all of the goals in the conjunction have been solved, the AND process sends its creator a success message; if one of the goals cannot be solved then a fail message is sent.

An AND process solves the individual goals in its conjunction by creating OR processes for each one. An OR process solves its literal L by (1) creating AND processes to solve the bodies of nonunit clauses with heads that are unifiable with L, or (2) sending 'sucess(C0)• when L unifies with a unit clause C.

The relationship between processes defines an AND/OR tree. The user's original goal statement is an AND process at the root of the tree. Success and fail messages pass only upward in the tree* from a process to its parent. Messages that can be sent from a process to one of its offspring are redo, reset, and cancel. A redo message instructs the descendant to re-solve its problem. If there is another solution, a success message (presumably with a different set of bindings for variables) will be sent. If there is not another solution* a fail message is returned. A reset message sent by an AND process to an OR descendant tells the OR process to start over; the OR process will retransmit all previous answers in response to redo messages. A cancel message indicates that the descendant is no longer needed, and that it should terminate.

By OR parallelism we mean the parallel execution of one or more children of an OR process, i.e. OR parallelism is the simultaneous execution of several clause bodies where the heads match a single goal. A straightforward (sequential) implementation of an AND process just solves the goals left to right: when the OR process for literal p_i sends a success message, then an OR process will be created to solve literal P_{i+1} if p_i cannot be solved (i.e. a fail message is received) then the process for literal p_{i_1} is sent a redo message. AND parallelism is the parallel execution of one or more children of an AND process, i.e. a parallel AND process creates more than one OR process simultaneously. The remainder of this paper describes our current implementation of AND parallelism.

Note that our definition of OR parallelism, which originally appeared in [4], is not the same as that used by some other authors (for example [7]). What they refer to as OR parallelism is the parallel search described in an earlier section and called Goal List parallelism in [3].

4. AMD Parallelism

A guiding principle in our work is that the programmer should not be concerned with the details of parallel interpretation. This implies that the programmer should not be forced to annotate his program with control information* as is done in IC-Prolog [2]. Moreover he is guaranteed that the answers produced by parallel interpretation include those achieved by depth first interpretation.

There are three independent components in our implementation: An ordering Algorithm determines which literals must be solved before others, and which can be solved in parallel; the forward execution component that handles success messages, and decides if other literals can be solved as a result; and the backward execution component that handles fail and redo messages* and decides which literal(s) from the body must be re-solved.

4.1. Ordering Algorithm

The only constraint on whether or not two literals can be solved in parallel is the sharing of uninstantiated variables. If a set of literals have an uninstantiated variable V in common* one of the literals is designated as the generator of values for V* and is solved first. Its solution is expected to instantiate V. When the generator has been solved* the other literals* the consumers. may be scheduled for solution. If a generator instantiates a variable to a non-ground term (one that contains new variables)* then the ordering algorithm must be applied again in order to find generators for the new variables.

The execution order of the literals* as determined by the ordering algorithm* can be expressed as a dataflow graph [1]. There is a node in the graph for every literal* including the head. An arc* labeled with a variable name* is drawn from every generator to literals that consume the variable.

The ordering algorithm uses three rules when deciding which literal should be designated as the generator of a variable. Mode declarations may indicate that a certain literal must be or cannot be a generator. For example* using the syntax of the DEC-10 Prolog compiler [9]. the predicate 'mode(sum(+*+»-))' means that the first two arguments to a call to 'sum' must be terms or instantiated variables* and that the third argument must be an uninstantiated variable. Thus* when the ordering algorithm encounters a literal 'sum(3,X,Y)', for example* it infers that this literal cannot be the generator of X and must be the generator of Y. The interpreter has mode declarations built in for some of the known evaluable predicates (primitive procedures) and the user can add mode declarations to his program if he wishes.

The connection rule states that if there is a literal p(X»Y) in the body* and a generator has already been found for the variable X* then p(X,Y) is a good candidate to be the generator of Y. This is just one of many possible heuristics* or general policies* that can be used to select generators. Other heuristics are described in [11] and [10].

The last rule is that the leftmost literal that contains an uninstantiated variable should be the generator of that variable; this rule is used if the other two fail to identify a generator.

An implementation of this algorithm produced the dataflow graphs shown in Figures 5-2 and 5-4. Note that except for evaluable predicates or user predicates with mode declarations* any literal can be a generator. The ordering algorithm is used primarily to ensure that mode constraints are not violated* and secondarily to produce an efficient ordering.

4.2. Forward Execution

After the ordering algorithm creates an ordering of literals* an AND process enters the forward execution phase. Forward execution is a graph reduction algorithm: when a literal is resolved away from the body of a clause (i.e. when the AND process receives a success message from the OR process created to solve the literal)* the corresponding node and all of the arcs leaving it are removed from the dataflow graph.

Recall that the only arcs in the graph are those that connect generators to consumers. An OR process will be created for a literal only when there are no arcs leading in to the corresponding node in the graph* i.e. when all preceding generators have been solved. Finally* when a generator creates a non-ground term (containing a new variable V')* the ordering algorithm must be invoked again* so that a generator is determined for V. Note that this will add arcs to the graph* and may delay solution of some literals that contained the original variable V.

4.3. Backward Execution

When an AND process receives a fail message from an OR process* it must re-solve some previously solved literal. The last parent (as defined by a linear ordering of literals* obtained by a depth first traversal of the dataflow graph) for the failed process is sent a redo message. If an OR process with no parents fails* the AND process itself fails.

To be more specific* consider the clause body

gl(X) A g2(Y.Z) ^ p(X.Y) A q(Y)

where gl(X) is the generator of X and g2(Y,Z) is the generator of Y. When p(X.Y) fails* the OR process for g2(Y,Z) is sent a redo message.

In a simple case, that process will return another success message, with a different value for Y» and the AND process can resume forward execution. Note that the process for q(Y) must be canceled when p(X,Y) fails* and replaced when the new value of Y it generated.

A more complicated case arises when the OR

process for g2(Y,Z) fails, signaling that there are no more ways to instantiate Y. At first it would appear that the AND process should fail, since there are no predecessors for g2(Y,Z) in the dataflow graph. However, since p(X,Y) may succeed with a different value for X. the appropriate action is to send a redo to the process for gl(X) and to reuse all values of Y by sending a reset message to g2(Y,Z). In other words, p(X,Y) has to be tested with all possible combinations of X and Y.

The method for deciding which generat6r(s) must be redone. and in what order, is quite complicated. The AND process maintains for each literal a list, called a redo list, containing all of the literal's ancestors in the dataflow graph. A failure context is used to determine which of the ancestors in a redo list have in fact been sent redo messages. The AND process must keep multiple failure contexts, for those situations when a number of literals fail* and redo messages have to be sent to ancestors of each failed literal [5].

5. Example Computations

We illustrate AND parallelism by describing the execution trace printed by our interpreter as it solved two sample problems.

Fast factorial is an example of a deterministic function, where the subgoals will not fail* and the backward execution component is not required. The map coloring problem, used by Pereira and Porto to illustrate intelligent backtracking [10], is a problem that does require backward execution.

5.1. Fast Factorial

The Fast Factorial program (Figure 5-1) typifies the divide and conquer approach to solving large problems. The procedure is called with two integers (named H and L) as inputs* and instantiates F to the result. Note that the head of the clause is the generator of H and L and the consumer of F. The divide and conquer version of this function is to compute the midrange between H and L» find the products F_1 and F_2 of all integers in these smaller ranges, and then multiply the results. As the graph shows, the computation of F_1 and F_2 can proceed independently* as soon as the midpoints are computed.

A depth first interpreter must solve the subproblems sequentially. This is an example of a deterministic function for which the only answer is found at the end of the leftmost branch in the search tree, and thus an example of a program for which a parallel search will not gain any speedup over a depth first search. In our interpreter, however, the recursive calls are solved by independent interpreters (OR processes), and the time complexity on a multiprocessor, under the right circumstances, could be reduced to $0(\log_2 T$ from 0(T), where T is the number of steps taken by a sequential interpreter.

```
mode(ff,[+,+,-]).
mode(is,[-,+]).

fact(N,F) ← ff(1,N,F).

ff(N,N,N).
ff(L,H,F) ←
        L < H ∧
        M1 is (L + H)/2 ∧
        M2 is (L + H)/2 + 1 ∧
        ff(L,M1,F1) ∧
        ff(M2,H,F2) ∧
        F is F1*F2.
```

Figure 5-1: Fast Factorial Program

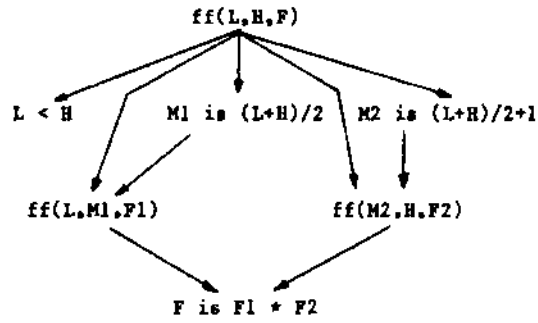

Figure 5-2: Dataflow Graph for Fast Factorial

5.2. Map Coloring

A map coloring problem, its representation as a logic program* and the dataflow graph produced for it are shown in Figure 5-3. The forward execution phase of the computation was: 'next(A*B)^{v} is the only node without incoming arcs, so it was the only one for which an OR process was created in the first step. As soon as $^{>}$next(A,B)' was solved, processes to solve $^{'}$next(A,C)', ^{v}next(A'D)^{v}* and ,next(B*E)^{f} were created. When $^{'}$next(A,C)' succeeded (instantiating C to a color), a process for 'next(B,C)' was created to verify the values of B and C. Similarly, after ''next(A*D)' and 'next(B,E)' were solved' processes for the remaining three literals were created.

The generators for B and C instantiated those

```
color(A,B,C,D,E) ←
    next(A,B) ∧
    next(C,D) ∧
    next(A,C) ∧
    next(A,D) ∧
    next(B,C) ∧
    next(B,E) ∧
    next(C,E) ∧
    next(D,E).

next(X,Y) ←
    color(X) ∧ color(Y) ∧ (X ≠ Y).
color(X) ←
    member(X,[red,blue,green,yellow]).
```

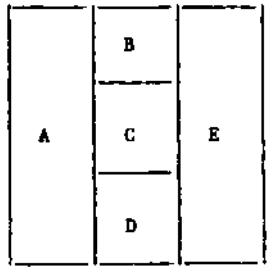

Figure 5-3: Map Coloring Example

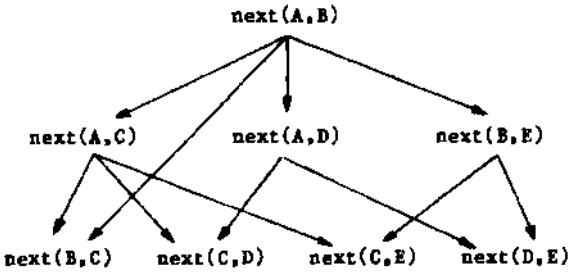

Figure 5-4: Dataflow Graph for Map Coloring

variables to the same color, so the process for 'next(B,C)' failed, thus triggering backward execution. The redo list (sorted list of ancestors) for this literal is [next(A,C),next(A»B)]. so the process for 'next(A»C)' was sent a redo message* in order to get a different value for C. Since the literals •next(C,D)[f] and 'next(C»E)[v] also consume C_9 the corresponding processes were canceled. When the process for [v]next(A»C)[v] sent the next success message* processes for the three literals that consume C were started (using this new value), and all four literals at the bottom of the graph were solved successfully.

In this example the AND process behaved very much like the intelligent backtracking interpreter of Pereira and Porto* In general, however, our backtracking algorithm will not be as smart as the theirs, since ours must be executed by a number of processes as opposed to a single process with global information, and our fail messages contain no information about xhg an OR process failed.

6. Conclusion

We have outlined an algorithm for the parallel execution of the body of a clause. The algorithm relies on the use of generators to order the solution of literals in a clause body so that a dataflow-like computation is performed. The algorithm allows generators to be nondeterministic, meaning they can produce a number of different answers. A major source of complication is in the coordination of these nondeterminiótic generators [5].

REFERENCES

1. Arvind, K. P. Gostelow, and W. E. Plouffe. An Asynchronous Programming Language and Computing Machine. Tech. Rep. 114a, Dept. of Info, and Computer Sci., UC Irvine, December, 1978.

2. Clark, K. L., and F. McCabe. The Control Facilities of IC-Prolog. In D. Michie, Ed., £x£££i Systems in JJie Micro Electronic Ag£# Edinburgh Univ. Press, 1979.

3. Conery, J. S.» P. H. Morris, and D. F. Kibler. Efficient Logic Programs: A Research Proposal. Tech. Rep. 166, Dept. of Info, and Computer Sci., UC Irvine, April, 1981.

4. Conery, J. S. and D. F. Kibler. Parallel Interpretation of Logic Programs. Proc. of the Conf. on Functional Programming Languages and Computer Architecture, October. 1981, pp. 163-170.

5. Conery, J. S. The AND/OR Model for Parallel Interpretation Q£ Logic Programs. Ph.D. Th., Dept. of Info, and Computer Sci., UC Irvine, 1983.

6. van Emden, M. H. An Interpreting Algorithm for Prolog Programs. Proc. of the First International Conf. on Logic Programming, Faculte dee Sciences de Luminy, Marseille, Sept, 1982, pp. 56-64.

7. Furukawa, K., K. Nitta, and Y. Matsumoto. Prolog Interpreter Based on Concurrent Programming. Proc. of the First International Conf. on Logic Programming, Faculte des Sciences de Luminy, Marseille, Sept, 1982, pp. 38-44.

8. Kowalski, R. A. Predicate Logic as a Programming Language. Proc. IFIPS 74, 1974.

9. Pereira, L. M., F. C. N. Pereira, and D. H. D. Warren. Users Guide to DECsystem-10 Prolog. Dept. of Artificial Intelligence, Univ. of Edinburgh, September, 1978.

10. Pereira, L. P. and A. Porto. Intelligent Backtracking and Sidetracking in Horn Clause Programs - the Theory. Report 2/79, Departamento de Informatica, Universidade Nova de Lisboa, October, 1979.

11. Warren, D. H. D. Efficient Processing of Interactive Relational Database Queries Expressed in Logic. Paper 156, Dept, of Artificial Intelligence, Univ. of Edinburgh, September, 1981.