THE REDUCE-OR PROCESS MODEL FOR PARALLEL EXECUTION OF LOGIC PROGRAMS*

LAXMIKANT V. KALÉ[†]

▷ A method for parallel execution of logic programs is presented. It uses REDUCE-OR trees instead of AND-OR or SLD trees. The REDUCE-OR trees represent logic-program computations in a manner suitable for parallel interpretation. The REDUCE-OR *process model* is derived from the tree representation by providing a process interpretation of tree development, and devising efficient bookkeeping mechanisms and algorithms. The process model is *complete*—it produces any particular solution eventually—and extracts full OR parallelism. This is in contrast to most other schemes that extract AND parallelism. It does this by solving the problem of interaction between AND and OR parallelism effectively. An important optimization that effectively controls the apparent overhead in the process model is given. Techniques that trade parallelism for reducing overhead are also described. ◁

1. INTRODUCTION

Symbolic computations have emerged as a significant class of computations in recent years. With developments in the fields such as expert systems and problem-solving systems, the complexity of such computations has been increasing. Many problems in these domains tend to be combinatorially explosive. It can be expected that their complexity will *entail* a parallel execution scheme.

Address correspondence to Laxmikant V. Kalé, Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 West Springfield Avenue, Urbana, IL 61801; telephone (217) 244-0094.

Received August 1988; accepted March 1989.

*Parts of this paper are adapted from a paper presented at the Fourth International Conference on Logic Programming, Melbourne, 1987.

[†]This research was supported in part by the National Science Foundation under grant CCR-87-00988.

THE JOURNAL OF LOGIC PROGRAMMING

©Elsevier Science Publishing Co., Inc., 1991
655 Avenue of the Americas, New York, NY 10010 0743-1066/91/$3.50

Logic-programming languages are suitable for expressing symbolic computations. It is also relatively easy tó express parallelism in a logic program.

This paper describes the development of an efficient scheme for parallel evaluation of combinatorially explosive symbolic computations expressed as logic programs. There are many research efforts directed at this broad goal. This approach is further guided by the following considerations:

(1) It is generally believed that communication cost is a dominant determinant of the performance of a parallel processing system. Therefore it is important, as far as possible, to subdivide the computation into *independent subtasks*.

(2) Given a logic program, one should strive to extract the *maximal possible degree of parallelism*, subject to the independence requirement above. In specific situations, it may be wise to trade some parallelism for a reduction in overhead. But if the basic scheme made that compromise, it would not be able to exploit another architecture with more processors, for example. Instead, methods should be developed for *dynamically* controlling the parallelism to match the resources. That is, the issue of controlling the degree of parallelism should be kept orthogonal to the basic scheme that extracts the parallelism.

(3) Broadly, the sources of parallelism in a logic program can be classified as AND-parallel and OR-parallel. Both of these sources of parallelism are significant in the symbolic computation domain. They must be exploited to effectively parallelize a wide range of problems, and also to effectively use a large number of processors, when available. Moreover, the two sources should be pursued *in concert*, which involves solving the problems of their interaction, to derive the full multiplicative benefit from both forms of parallelism.

The classes of programs that benefit most from our approach include combinatorial search computations encountered in, say, problem–reduction-based problem solving. There may be many methods for solving a subproblem, and a method may involve consistently solving many subproblems. The OR parallelism is typically present underneath AND parallel branches in these computations. Symbolic integration and many planning problems are examples of problem reduction. Although the proposed scheme is meant to work equally well for purely AND-parallel or purely OR-parallel computations, it is worthwhile keeping this class of computations in mind as one of the motivations behind this approach.

2. REPRESENTATION OF PARALLEL LOGIC COMPUTATIONS

First, the terminology used in this paper is briefly introduced.

A logic program consists of a set of Horn clauses. Each clause has the form '$\text{L} :- \text{L}_1, \ldots, \text{L}_n$.', where all of the L_i's are *literals*. A literal consists of a predicate name followed by a parenthesized list of *terms*. A term can either be a *constant* or a *variable* or a *function symbol* followed by a parenthesized list of terms. Predicate names, function symbols, and constants can be quoted strings or identifiers that start with a lowercase letter. A variable is any identifier starting with an uppercase

letter. In the above clause, L is the *head* of the clause, and '$L_1, \ldots, L_n$' is the *body* of the clause. A clause with an empty body is written simply as 'L.' and is called a *fact*. A clause that is not a fact is called a *rule*.

A (possibly empty) set of literals is called a *query*. The body of a clause C, instantiated with the substitution that unifies a literal G and the head of C, is called a *query for G using clause C*.

The clause '$L :- L_1,..L_n$.' may be understood to mean "one way to solve L is to solve the subproblems L_1 and...and L_n." A fact signifies a solved or "true" formula. A literal L is *solved* if there exists a clause C whose head unifies with L, and all the literals in the query for L using C are also solved. We define these notions of *solving* and *solution* more precisely in the next section.

2.1. *The* REDUCE-OR *Trees*

As many researchers have pointed out [5, 14], AND parallelism is an important source of parallelism because it frequently, but not always, involves "mandatory" actions as opposed to "speculative" ones, and thus relates directly to speedup. (If one of the AND-parallel branches fails, the work done in the other branch is wasted, and so AND parallelism can also be speculative in some cases.) Schemes for exploiting AND parallelism [2, 5, 17, 18, 29], whether they deal with OR parallelism or not, are typically based on the AND-OR tree [4, 28, 33] as the representation of logic computations.

There are two types of nodes in an AND-OR tree: AND nodes and OR nodes. Each node has a label. OR nodes are labeled with a single literal. AND nodes are labeled with a query (i.e. a set of literals). Given a query Q and a logic program P, the AND-OR (AO) *tree for Q w.r.t. P* is recursively defined as follows. The root of the tree is an AND node labeled with Q. Each AND node labeled with a query $\{G_1, \ldots, G_n\}$ has n children, which are OR nodes labeled with $G_1, \ldots, G_n$. Each OR node labeled with a literal L has a child AND node labeled with $Q1$ for every clause C such that $Q1$ is a query for L using C. The arc from an OR node to an AND node is called a *match arc* and is labeled with the most general unifier of the head of C and L. An AND node labeled with an empty query is a *leaf node*. A leaf node results from using a fact clause for matching an OR node. In illustrations, we sometimes label each leaf node with the corresponding fact. A candidate *solution tree* for a literal G (or a query Q) is a subgraph of the AND-OR tree for G (or Q) such that:

(1) It includes the root of the AND-OR tree.

(2) If it includes an AND node A, it also includes all the child nodes of A.

(3) If it includes an OR node O, it also includes exactly one child node of O.

A *consistent* solution tree is a candidate solution tree such that the labels of its match arcs have a unifiying composition. A query Q is said to be *solved* if the AND-OR tree rooted at Q contains a consistent solution tree. Let S be the composition of the substitutions that label the arcs of a consistent solution tree for a query Q. The projection of S onto the variables of Q is called a *solution* to Q. A binding is said to *satisfy* a literal G (or a query Q) if it is a solution to G (or Q).

Consider the AND-OR tree for the query '$p(X), q(X, Y)$' shown in Figure 1. There are four candidate solution graphs, consisting of nodes $\{1, 2, 3, 4, 6\}$,

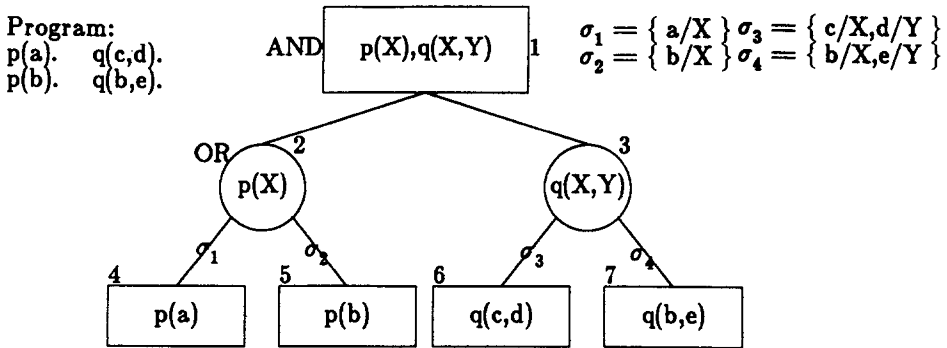


FIGURE 1. An AND-OR tree.

{1,2,3,4,7}, {1,2,3,5,6}, and {1,2,3,5,7}. Of these, only the last one is a consistent solution graph, because σ_2 and σ_4 have a unifying composition, providing $\{X=b, Y=e\}$ as a solution to the query. The other solution graphs are inconsistent because the substitutions on their match arcs are inconsistent. For example, σ_1 and σ_3 do not agree on the value of X. If the clauses for p or q were defined by rules, such inconsistencies could occur between arcs that are arbitrarily far apart in the tree. Detecting these requires a global check across the whole solution tree.

In a parallel environment such global checks lead to unacceptable levels of communication. Therefore, we wish to constrain our tree models so that each node represents a completely described subproblem—to be solved without any reference to the nodes in the tree above it. To ensure this, we associate a *partial solution set* PSS with every node in the tree. Each element of this set is a substitution that is a solution to the literal or query that labels the node. It must be computed using information only from PSSs of the children of the node.

In our example, this constraint implies that the tree beneath the node for $p(X)$ computes a set of bindings for X that satisfy $p(X)$, the tree beneath the node for $q(X,Y)$ computes a set of bindings for X and Y that satisfy $q(X,Y)$, and then a *relational join* of these two sets yields the solutions for '$p(X), q(X,Y)$'. All the consistency checks are thus localized in the join operation.

With that, though, another problem emerges: the AND-OR trees do not allow distinct representation of the true subproblems. Consider the same query '$p(X), q(X,Y)$' again. A legitimate, and frequently effective, way to solve this query is to solve $p(X)$ first, and to solve $q(x_i, Z)$ for every x_i that satisfies $p(X)$. This allows solutions to $p(X)$ to prune the search space for q. Here the real subproblems of the query are $p(X), q(x_1, Z), q(x_2, Z), \ldots$. This subdivision cannot be represented in the AND-OR tree: it has just one node for the literal $q(X,Y)$. Therefore, we propose a different representation of logic computations. As it faithfully represents the process of *reducing* a problem to its subproblems, we call it the REDUCE-OR *tree*. This representation was informally described in [19]. Its formal definition and relationship to other tree representations are examined in [23].

A REDUCE-OR tree consists of REDUCE nodes and OR nodes. See Figure 2 for a sample REDUCE-OR tree for the query of Figure 1. Each REDUCE node is labeled

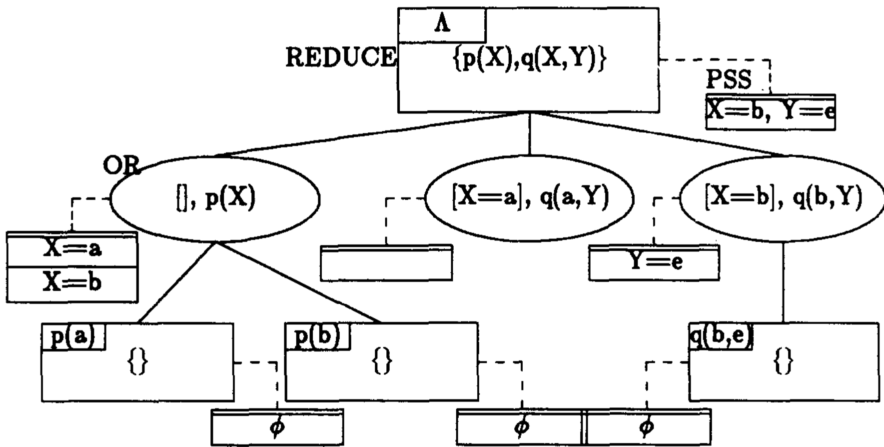


FIGURE 2. A REDUCE-OR tree.

with a query, and each OR node with a literal. If a REDUCE node is labeled with a query Q, every OR child of this node is labeled with an instance of a literal, say L, in Q. A substitution S that is a solution to some of the other literals in Q may be used to instantiate L. (We say the corresponding OR node *inherits* the substitution S.) To coordinate this, we assume a partial order on the literals in every REDUCE node. A concrete way of specifying such an order (in the form of a graph) will be discussed in the next section. The OR children corresponding to a literal can inherit substitutions only from the OR children corresponding to the predecessor literals in the partial order. In the root node of Figure 2, we assume that $p(X)$ precedes $q(X,Y)$. The substitutions $X=a$ and $X=b$ produced by the OR child for $p(X)$ are thus inherited by the OR children for $q(X,Y)$.

In text, we denote an OR node as $\mathrm{O}(\sigma, \mathrm{G}, \mathrm{PSS})$, where σ is a substitution, G is a literal, and PSS is a set of substitutions that satisfy G. σ serves a book-keeping purpose explained shortly. A REDUCE node is represented as $\mathrm{R}(H, \{G_1, \ldots, G_n\}, \mathrm{PSS})$, where H and $\{G_1, \ldots, G_n\}$ are the instantiated head and body of some clause of the program, and PSS is a set of substitutions that satisfy $\{G_1, \ldots, G_n\}$ and hence H. As in attributed syntax trees, the nodes in REDUCE-OR trees have information that is inherited from parents or siblings, and information that is synthesized from children. The first two components are inherited, and the third component, PSS, is synthesized.

The REDUCE-OR tree for a query Q, w.r.t. a logic program P, is defined using mutually recursive rules as follows. Rules (1) through (3) specify how to build the tree, and rules (4) and (5) specify how to collect the PSSS.

(1) The root of the tree is $\mathrm{R}(\Lambda, Q, \mathrm{PSS})$.

(2) *The children of a* REDUCE *node:* Let $\mathrm{R}(H, \{G_1, \ldots, G_n\}, \mathrm{PSS})$ be a REDUCE node. If $n = 0$, then it has no children. Otherwise, it has zero or more OR children corresponding to each G_k, $1 \le k \le n$, as described below. (Informally, there is an OR node for solving an instance of G_k for each substitution that satisfies all the predecessors of G_k.) If G_k has no prede-

cessors, the REDUCE node has one OR child corresponding to G_k: o (ϕ, G_k, PSS'), where ϕ is the empty substitution. Let $P_1, \ldots, P_m$ be the predecessors of G_k in the partial order. (Note: $\{P_1, \ldots, P_m\} \subset \{G_1, \ldots, G_n\}$.) The REDUCE node has one or more OR children labeled with an instance of P_i for each i, $1 \le i \le m$. Pick one OR child for each i, say $o(\sigma_i^j, P_i^j, \text{PSS}_i^j)$. ($P_i^j$ is an instance of P_i.) From each PSS_i^j, pick one substitution, say $\delta_i \in \text{PSS}_i^j$. Let $\mu_i = \sigma_i^j \cdot \delta_i$ and $\sigma = \mu_1 \cdot \mu_2 \cdot \cdots \cdot \mu_m$. For each such choice, if the resultant σ is a consistent substitution, then there is a new OR child: $o(\sigma, \sigma G_k, \text{PSS}')$. That is, the new goal literal is instantiated using a composition of one solution to each of its predecessors. σ essentially remembers the "context" of the OR node. Without such a σ, the root REDUCE node of Figure 2 would have no way of knowing that $\{X = a, Y = e\}$ is not a solution.

(3) *The children of an* OR *node:* Let $o(\sigma, G, \text{PSS})$ be an OR node. For each clause of the form '$\overline{H}_i$:– Q_i' such that $\overline{H}_i$ unifies with G, the OR node has a child $R(H_i, \theta Q_i, \text{PSS}_i)$. Here $H_i = \theta \overline{H}_i$, and θ is the most general substitution to the variables of $\overline{H}_i$ so that it matches G (i.e., $\theta \overline{H}_i = \pi G$ for some π).

We now define the PSS of a node in terms of the PSS of its child nodes.

(4) Let $o(\sigma, G, \text{PSS})$ be an OR node with children $\{R(H_i, Q_i, \text{PSS}_i)\}$. Then $\text{PSS} = \{\delta \mid \delta G = \rho_i H_i$, where $\rho_i \in \text{PSS}_i$, for some $i\}$. (Informally, any solution to a REDUCE node translated via back unification is also a solution to its parent OR node.)

(5) Let $R(H, \{G_1, G_2, \ldots, G_n\}, \text{PSS})$ be a REDUCE node with children $\{o(\sigma_i^j, G_i^j, \text{PSS}_i^j)\}$. Then $\text{PSS} = \{\rho \mid \rho = \mu_1 \cdot \mu_2 \cdot \cdots \cdot \mu_n$, where $\mu_i = \sigma_i^j \cdot \delta_i^j$ and ρ is consistent for some j, with $\delta_i^j \in \text{PSS}_i^j\}$. (Informally, a consistent composition of solutions to each literal of a clause is also a solution to the head of the clause.)

Notice the special case $R(H, \{\}, \{\varnothing\})$, i.e., the PSS for a REDUCE node corresponding to a *fact* is a singleton set with a null substitution.

2.2. *The Data Join Graph*

The body of a clause consists of a set of literals. It is not usually efficient to compute them all in parallel. Frequently, some partial ordering among the literals can be used to solve the query effectively. We assumed such an ordering in the definition of the REDUCE-OR tree in the previous section. The effectiveness of a specific ordering for a clause may depend on the rest of the program and on the query. Some analysis of the rest of the program and of the data dependencies within the clause is needed to estimate the best partial order. The data join graph (DJG) encodes the recommendation of such analysis, specifying which literals can be solved in parallel and which must wait for data from others. Many techniques for obtaining such a graph (or a variant) are proposed and being developed by researchers [2, 10, 35]. Here, we are concerned with only what the DJG is.

Each arc in the graph represents a literal of the clause. Each node denotes a joining point for the data produced by the literals on its incoming arcs. Literals on the arcs emanating from a node can be evaluated only after the node is triggered.

$h(Y,Z)$:$-g(X),p(X,T),q(X,U),r(T,U,Y),s(T,U,Z)$.

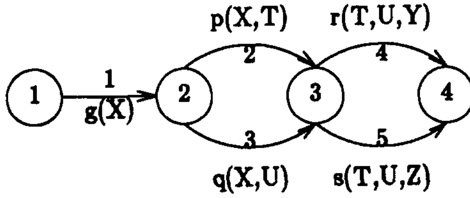


FIGURE 3. A data join graph.

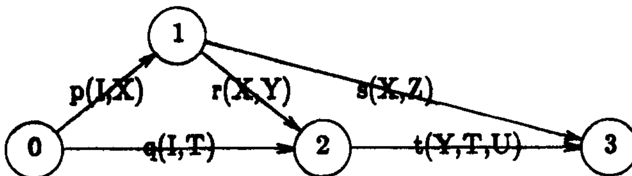
A DJG for the clause

$$h(Y,Z) :- g(X), p(X,T), q(X,U), r(T,U,Y), s(T,U,Z). \quad \text{(C1)}$$

is shown in Figure 3. Here p and q can be executed in parallel, and so can r and s, but r and s depend on at least one solution each for p and q.

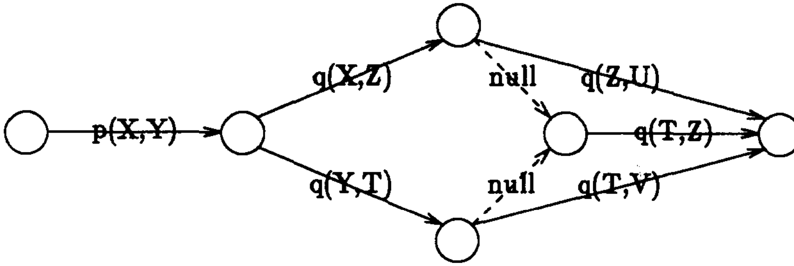
Despite some differences, the DJG is essentially equivalent to Conery's data flow graphs [4] or the data dependency graphs described by Chang et al. [2]. One difference is that we use arcs, as opposed to nodes, to denote literals. The nodes denote joining points for data. As shown later, having such explicit join points has some bookkeeping advantages for the process model. Another difference is that the DJG does not preclude parallel computation of two subgoals that share a variable. Such dependent AND parallelism is permitted by the REDUCE-OR trees, to cater to the rare situations when it is beneficial, although our emphasis is on independent AND parallelism. Conversely, the DJG can encode the dependence on pure tests as in *generate*(X), *test*(X), *process*(X,Y). Here, we do not want *process* to carry on until *test* certifies the generated value, but there is no explicit data flow from *test* to *process*. Such a dependence can also be encoded in the above schemes by adding an extraneous variable that is shared by the *test* and the *process* literal.

Consider the clause body shown in Figure 4. Here, I is assumed to be the input variable; p and q compute X and T, respectively, using the value of I. Literals r and s use X to produce Y and Z respectively, while t uses Y and T to produce U. As the figure demonstrates, the DJGs can encode certain dependencies efficiently which cannot be easily encoded in the kinds of graphs used in [1,8,16]. The legal



Clause Body: p(I,X), q(I,T), r(X,Y), s(X,Z), t(Y,T,U).

FIGURE 4. A DJG that cannot be expressed using CGEs.



Clause body: p(X,Y), q(X,Z), q(Y,T), q(Z,U), q(T,V), q(T,Z).

FIGURE 5. A DJG with null arcs.

constructs in these systems (such as CGEs and 'andP's) lead to "block-structured" (or simply "structured") graphs—constructed with single-entry, single-exit blocks. Trying to express the DJG in Figure 4 with such graphs results in loss of AND parallelism, as it introduces unnecessary dependencies. This has also been pointed out in [31]. For example, using the CGE (conditional graph expression) as defined in [16], one may write the clause body as

$$((p(I,X),r(X,Y))\,\&\,q(I,T)),(s(X,Z)\,\&\,t(Y,T,U)).$$

Here, '&' denotes parallel subexpressions. However, this expression forces the literal $s(X,Z)$ to wait for a solution to $q(I,T)$, which is unnecessary. When the predicates involved are nondeterministic, such graphs lead to even more serious performance degradation, as illustrated at the end of Section 3.3.2 with the map-coloring example.

A consequence of using the DJG representation instead of the data flow graphs is the presence of null arcs, as illustrated in the DJG of Figure 5. 'null/0' can be thought as a predicate defined by a single fact.

As DeGroot and Chang [9] point out, a static dependency graph, fixed at compile time, is bound to lose significant parallelism, because it has to be conservative in its estimate of which actions can be safely executed in parallel. For that reason, the notion of DJG has been extended to *conditional* DJGs, which are described in [20]. In this paper, for clarity, we assume a fixed DJG, obtained at compile time or at least before the REDUCE node is created.

3. THE REDUCE-OR PROCESS MODEL

Computation of a query Q with respect to a logic program can be viewed as a process of growing a REDUCE-OR tree rooted at a REDUCE node labeled with Q. There are, of course, many possible ways of growing the tree and managing the information at each of the nodes. The process model described in this section specifies one particular execution method. This process model was presented in a shorter conference paper [21] without the full description of the optimizations, etc.

In this section, we first derive the process model from the tree representation. Section 3.1 describes the REDUCE process in detail, and Section 3.2 presents an important optimization. Section 3.3 discusses two properties of the model: completeness and "full" OR parallelism. The algorithm, particularly the optimizations of Section 3.2, are somewhat complex. We would like to caution the reader against equating the conceptual complexity of the algorithm described, working in the general case, with its computational complexity—viz. the overhead. In Sections 4 and 5 we give arguments and data to show how the model can be efficiently implemented.

A process is associated with each node of the tree. Thus there are two kinds of processes: REDUCE processes and OR processes. We now describe how the information that appears in the tree is stored by the processes. In the tree, the OR nodes carry a field which stores a substitution representing its *context*. This field, denoted σ, is used only at its parent REDUCE node, for computing the parent's PSS, and to add more child OR nodes to it. Also, this context is shared by many OR nodes, if they correspond to AND-parallel branches emanating from a node in the DJG. So it is sensible to store this field at the REDUCE process. We attach this information, in the form of a tuple of bindings, to the nodes of the DJG. All the tuples attached to a node N form a *node relation* for N. The context of an OR process for a literal L is now a tuple in the node relation for the node immediately preceding L in the DJG. The OR process receives only a pointer to the context tuple, and this pointer is included along with any solutions reported to the REDUCE process. Notice that the PSS of the REDUCE node is the same as the node relation of the last node of its DJG.

The PSS of an OR node also gets used only in the parent REDUCE process. So we move that up to the parent REDUCE process. PSSS of all the OR nodes for a literal L form a *literal relation* that is attached to the arc corresponding to L in the DJG.

The entries in the PSS of an OR node are computed using rule (4) of Section 2.1. According to that rule, an entry δ_i is to be created in the PSS of an OR node for every entry ρ_i in the PSS of its child REDUCE node, by solving for δ_i in

$$\delta_i \cdot G = \rho_i \cdot H, \tag{1}$$

where H is the instantiated head of the clause used in forming the REDUCE node. Directly using this rule leads to the following procedure. When a REDUCE process finds a solution ρ_i in its PSS, it instantiates the head literal H using ρ_i, forming the instantiated literal $\rho_i \cdot H$. The REDUCE process then sends it up to its parent OR process, which can then solve for δ_i using the equation (1). However, this procedure can be optimized. Let $\bar{H}$ be the head of the clause used. Notice that H was obtained from $\bar{H}$ by unifying $\bar{H}$ with G [see rule (3)]. Thus, we have

$$\pi \cdot G = \theta \bar{H} = H$$

(H is the most general instance of G and $\bar{H}$). Substituting for H in the equation (1), we get

$$\delta_i \cdot G = \rho_i \cdot \pi \cdot G .$$

Therefore, $\delta_i = \rho_i \cdot \pi$ is the most general solution for θ. So, instead of storing the instantiated head H, the REDUCE process stores the bindings π to the goal variables produced during unification. Now, a REDUCE process R can easily

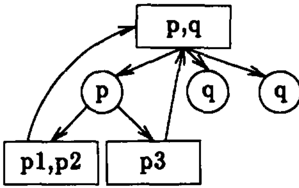


FIGURE 6. Interprocess communication.

construct the entry δ_i for the parent OR process's PSS when it makes an entry (ρ_i) in its PSS. The OR process's PSS is stored in the grandparent REDUCE process R_p, so R may directly send it δ_i as a response to R_p.

With the correlation between the REDUCE-OR tree and the process model established, we now describe the process model in detail.

An OR process, say O, is given a single goal literal, say G, and a context pointer T by its parent REDUCE process, say R. It finds from the program all the clauses whose heads unify with G. While unifying a head H with G, it creates two substitutions: θ, which is a substitution to the variables of H, and π, which is a substitution to the variables of G [see rule (3)]. When the matching clause is a fact, it simply sends π as a solution to its parent REDUCE process R. It creates a REDUCE process for each matching rule, and passes θ, π, and the DJG for the clause. It also passes the process id of the parent REDUCE process, and the context pointer T to these processes, so that they can send answers directly to R. It then terminates, as it has no part in relaying the solutions to its parent.

The REDUCE process is more complex, and is fully described in the next section. It must first identify all the literals that have no predecessors in the DJG of Q, and start an OR process to solve each such literal. It must await responses from these processes. For each response received, say R, which is a solution to one of its literals, it must check if one or more complete solutions to all literals of Q can be constructed using R. If so, it must send all the constructed solutions to the (grandparent) REDUCE process. Otherwise, some other literal(s) may have become eligible for solution with the arrival of this response. It must create an OR process for each such literal. The selection of eligible literals must be constrained by the DJG of Q. The selected literals given to the new OR processes must be instantiated using rule (2) of Section 2.1.

An example of processes and their communication patterns is shown in Figure 6. The communication structure is quite structured. A parent process communicates with each child only when the child is being created. The REDUCE processes communicate to their (grand)parent REDUCE process only when they have a solution. The subprocesses are independent to the extent that their computation, once started, does not depend on any message or information coming from the parent or siblings.

3.1. The REDUCE *Process*

Each REDUCE process, for a query Q, maintains many relations, one for each literal arc (called a literal relation) and one for each node (called a node relation) of the

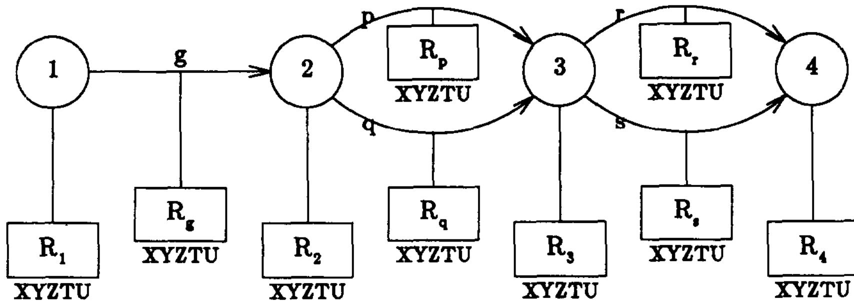


FIGURE 7. Relations maintained by the REDUCE process.

DJG of Q. Figure 7 shows the relations for the DJG of Figure 3. Each relation is over the set of all the variables of the clause; if the clause has variables $X_1, \ldots, X_n$, a tuple $T = [T_1, \ldots, T_n]$ in one of the relations represents the substitution $[T_1/X_1, \ldots, T_n/X_n]$. The semantics of these relations is as follows: A tuple in a node relation (or literal relation) is a substitution that satisfies all the literals preceding it in the DJG (including itself, in case of a literal relation). All the relations are empty when the process begins.

When started, a REDUCE process is given the process id of its grandparent REDUCE process (say ParentPID), a parent goal number (which represents the context), and a substitution π that expresses the variables in the parent in terms of its own variables. The process stores these for use while constructing a response. It is also given a DJG together with a tuple of the initial bindings of the variables of the literals in the DJG. This tuple is first inserted in the relation associated with the first node. The algorithm followed by the REDUCE process is described by the pseudocode in Figure 8. The process can be best understood by looking at what happens when

(1) a tuple is inserted in a node relation,

(2) a response, which is a solution to a literal, arrives, and

(3) a tuple is inserted in a literal relation.

Insertion of a tuple T in a node relation: If the node is *not* the last node of the DJG, an OR process is started for each literal corresponding to an arc emanating from that node. The variables of these literals are first instantiated using the substitutions from T. A unique goal number (the *context*) is passed down to each OR process being created. All responses from this process include this goal number for reference. For each goal number, the REDUCE process remembers (1) the arc number of the literal, and (2) a pointer to T, the context tuple that is being extended by the new OR process. Figure 9 shows a snapshot of the actions that follow the insertion of a tuple in a node-relation for node 3 of the example query of Figure 3.

If the node is the last node, a solution to the query has been found. The values of variables from T are used to instantiate a copy of π, the stored substitution, to obtain a substitution tuple R that belongs to its parent's PSS. A response of the

```
PROCESS REDUCE(Parent_PID, Parent_Gnum, Parent_tuple /* π*/, Init_Tuple/*θ*/,DJG)
/* The DJG corresponds to one of the clauses of the program
  Tuple has values of variables of this clause as instantiated by the 'call' */
     allocate and initialize the node_relations and literal_relations;
     insert_in_node_relation(0,Init_Tuple);
     repeat forever
          wait_for_response(R);
          process_response(R);

PROCEDURE insert_in_node_relation(Nodenum,Tuple);
  if last_node(Nodenum,DJG)
    then     Response_tuple = compose(Parent_Tuple, Tuple); /* δ_i=ρ_i·π */
             send(Parent_PID, Parent_Gnum, Response_tuple);

    else     NodeRelations[Nodenum] ← NodeRelations[Nodenum] ∪ {Tuple};
             A ← successor_arcs(Nodenum,DJG);
             for each arc ∈ A,
                 G ← instantiate( literal(arc,DJG), Tuple);
                 Gnum ← a unique number;
                 Node_tuple[Gnum] ← Tuple; arcnum[Gnum] ← arc;
                 start OR-process(Gnum,G,My_PID);

PROCEDURE process_response(RM /* response message */)
     Gnum ← RM.Parent_Gnum;      R_Tuple ← RM.Tuple;
     Arc ← arcnum[Gnum];
     NewTuple ← unify(R_Tuple, Node_tuple[Gnum]);      /* extend the context tuple */
     insert_in_literal_relation(New_Tuple,Arc);

PROCEDURE insert_in_literal_relation(Tuple,Arcnum)
     LiteralRelations[Arcnum] ← LiteralRelations[Arcnum] ∪ {Tuple};
     Node ← successor_node(Arcnum,DJG);
     A ← predecessor_arcs(Node,DJG);
     NewTuples ← {Tuple}
     for all arcs a ∈ A,
          NewTuples ← NewTuples ⋈ LiteralRelations[a]      /* ⋈: relational join */

     for each T ∈ NewTuples,
       insert_in_node_relation(T,Node);
```

FIGURE 8. Pseudocode for the basic REDUCE process.

form ⟨parent goal number, R⟩ is sent to the REDUCE process, ParentPID. The REDUCE process then resumes waiting for more responses.

Arrival of a response: A response includes a goal number and a binding tuple R (see Figure 10). The arc number and the tuple T being extended by the response R are recalled using the goal number. Note that R has bindings for variables of the current REDUCE process because of the way it was constructed. So T and R are tuples over the same domain. The tuple T (here $[a, -, -, b, c]$) is then unified with the new tuple R, and the resultant tuple ($[a, n, -, b, c]$) is inserted in the appropriate literal relation. Unifying two tuples involves unifying each member in one with

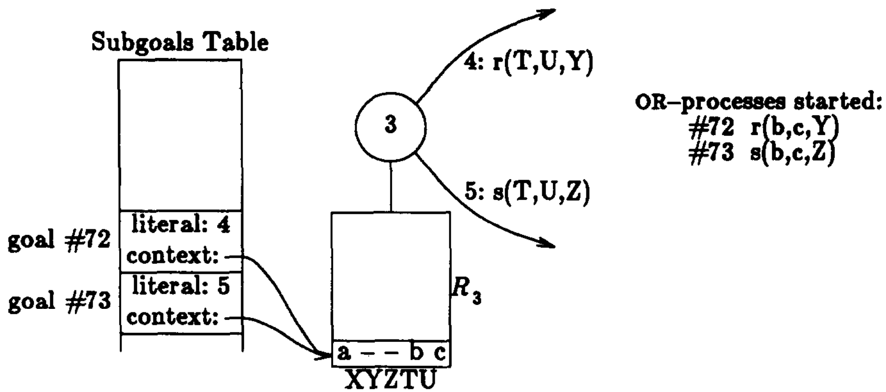


FIGURE 9. Inserting a tuple in a node relation.

the corresponding member in the other. When the DJGs enforce independent AND parallelism, the unification may be simplified to copying.

Insertion of a tuple in a literal relation: When a tuple is inserted in a literal relation, say R_L, it must be checked whether any tuples can be inserted in the node relation for the node (say N) that follows the literal. Tuples in a node relation denote the bindings that solve all the preceding literals. So the node relation associated with node N is a join of all the literal relations for the literals immediately preceding N. To update the node relation without duplicating tuples, a join of all the literal relations for the preceding literals, excluding R_L, and the singleton relation consisting of the new tuple is computed. Each tuple of the resulting relation is inserted into the node relation for N, triggering all the actions associated with insertions into a node relation as described above. Figure 11 shows an example situation taken from the processing of the query for (C1) in Section 2.2. It illustrates the actions following insertion of a tuple $T = [a, -, -, -, c]$ in the literal relation for q.

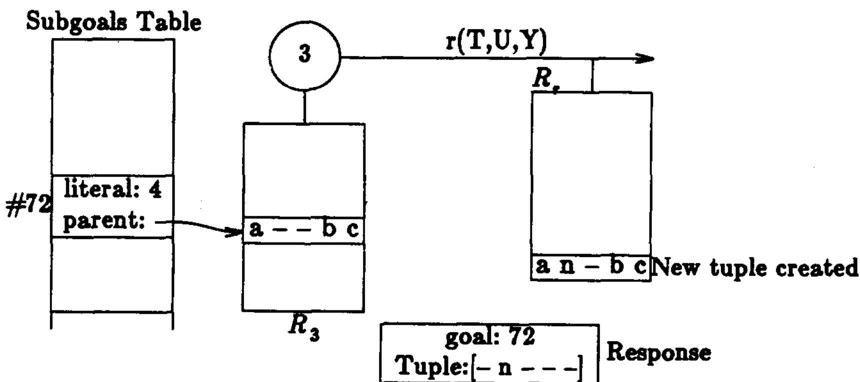


FIGURE 10. Arrival of a response.

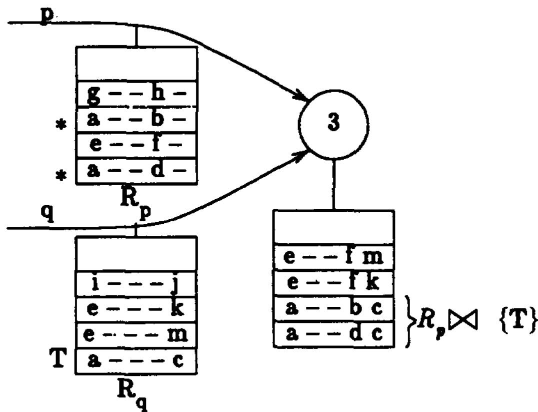


FIGURE 11. Inserting a tuple in a literal relation.

3.2. An Important Optimization

We now describe an important optimization that significantly reduces the overhead in the REDUCE process. It simplifies the join operation, and permits reductions in space usage. We consider it an essential part of the REDUCE-OR process model. The main reason for not including it in the description in Section 3.1.1 is pedagogical.

In the basic algorithm, when a tuple is inserted in a literal relation, it is "joined" with all the tuples in the other literal relation(s) that join at the same successor node. The process of combining two tuples involves first checking if the tuples are compatible (i.e. they have the same values for the variables bound in both of them) and then copying the bindings to create a combined tuple. For example, in Figure 11, when the tuple T was inserted in R_q, all the tuples in R_p were visited, and each checked to see if it has the same value for X ($= a$) as T. Thus, there are two sources of overhead in the basic algorithm: visiting tuples that are not useful, and performing the consistency checks. The optimization described next makes it possible to visit only the relevant, compatible tuples, and thereby also avoids the consistency checks.

The extended algorithm is best explained by first showing its behavior on a simple example. Consider a new DJG being used by a REDUCE process as shown in Figure 12. With every tuple inserted in a node relation, several pointers are created when the corresponding OR processes are created. In this example, one pointer is created corresponding to each of the arcs emanating from the node. (In general, there is one pointer for each arc that participates in a join in which this node is a common ancestor. See below.) Each pointer points to a list (initially empty) of tuples in the literal relations that are to be joined.

When a response for r is received, the goal number associated with it is used to locate the context tuple T. T is then extended to obtain a new tuple T_4 using the bindings provided by the response. (So far, all the actions have been the same as in the basic algorithm of Section 3.1.) T_4 is now inserted into the linked list of all tuples in R_r that are extensions of T, as indicated by the dashed line. (Explicitly inserting T_4 in relation R_r is not necessary now.) To compute the additions to the

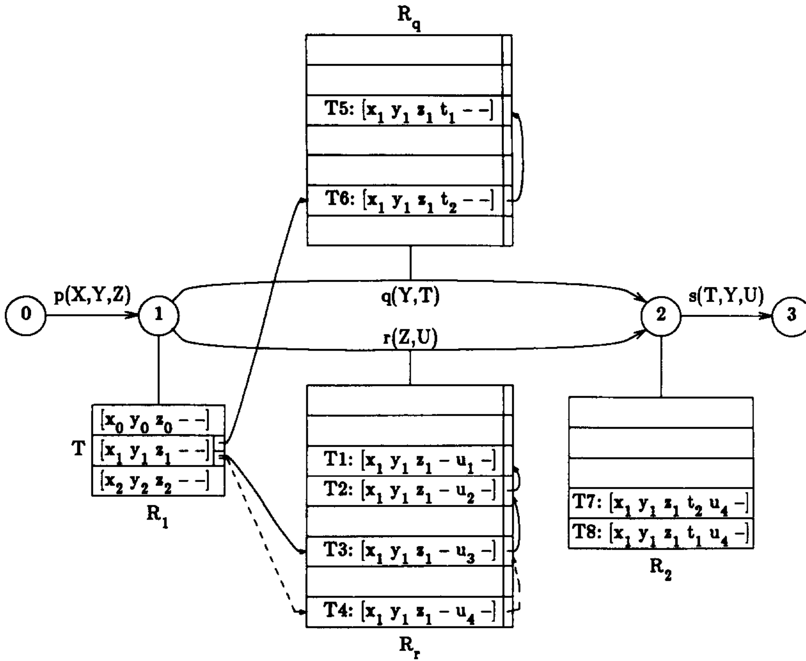


FIGURE 12. The optimized algorithm: a simple example.

joined relation, R_2, we do not join T_4 with the whole of R_q. Instead, we now follow the q-pointer from the context tuple T into the linked list of extensions of T in R_q. This leads us to tuples T_5 and T_6. The join of T_4 with T_5 and that of T_4 with T_6 result in two new tuples, T_7 and T_8, in the node relation R_{qr}. The complexity of the join operation is now reduced to the minimum. The whole literal relation (R_q here) isn't traversed; only the relevant tuples are visited. In addition, if the DJGs ensure independent AND parallelism (i.e., literals that are allowed to execute in parallel cannot bind the same variable), there is no need for any consistency checks or unifications for the tuples visited.

In general, the situation can be more complex than that of the above example, depending on the structure of the DJGs. The algorithm deals with these situations as follows.

(1) *Distant common ancestors:* In the previous example, the context tuple T also happened to be the common ancestor of tuples in both literal relation tuples that were joined. Consider the DJG of Figure 13. Here the common-ancestor tuple (T_A) is in the relation R_0. So, when a response from p arrives, its context tuple (T_1) does not have a pointer to the linked list of relevant tuples in R_q. To handle this situation, we require each node-relation tuple to carry additional *back pointers*. Here, the context tuple (T_1) in the node relation R_2 carries a back pointer to the common ancestor tuple T_A. The new tuple T_{p1} can be inserted in the linked list that begins at one of the forward pointers at T_A. Also, from T_A we follow the other forward pointer into R_q as before, to perform the join. The tuples in the node

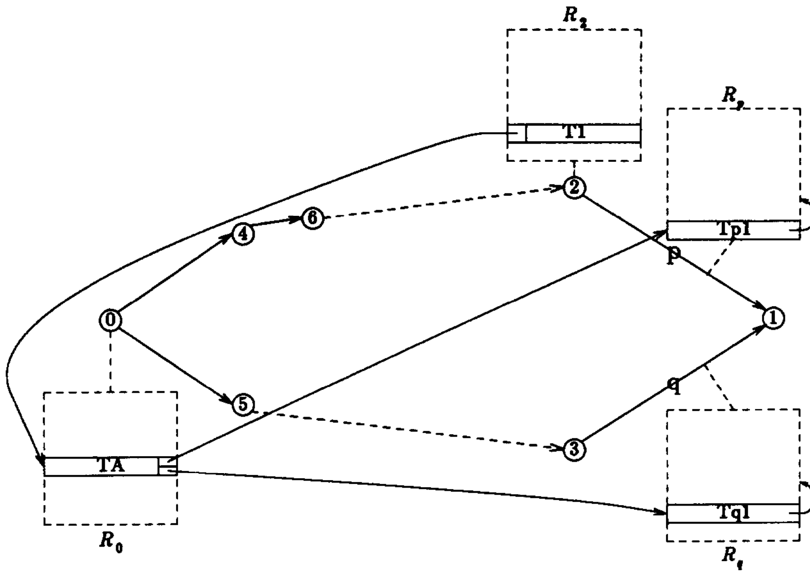


FIGURE 13. Distant common ancestors

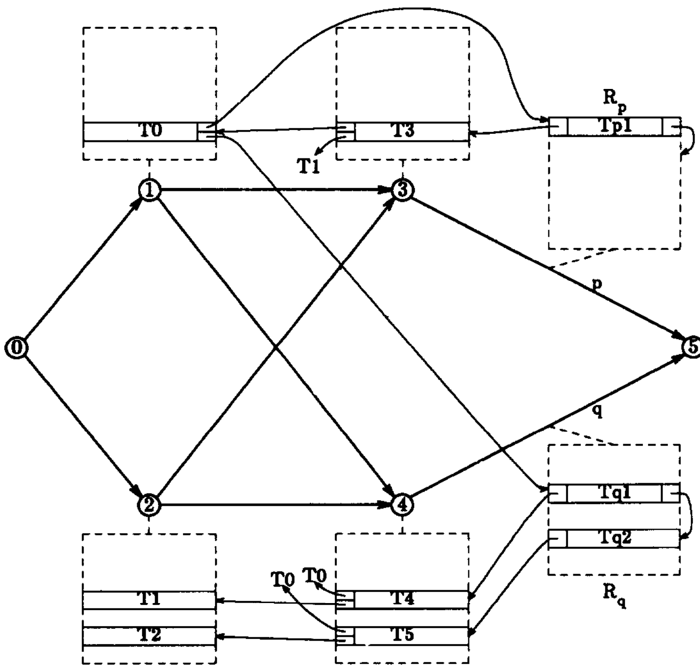


FIGURE 14. Multiple common ancestors.

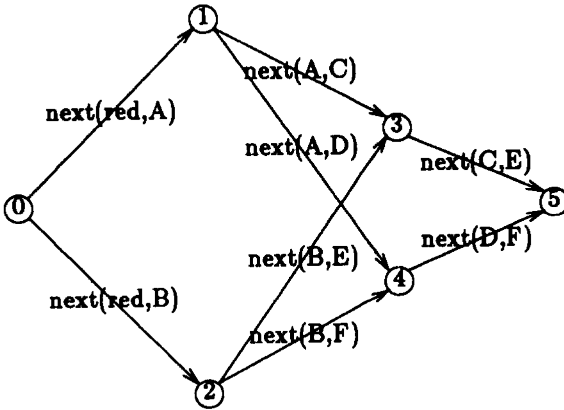


FIGURE 15. Multiple joins with the same common ancestor.

relations along the path from the common ancestor to the joining node simply copy and carry the back pointers. For example, all tuples in the relation for node 4 (R_4) carry a back pointer to their ancestor tuple in R_0. Let T_4 be a tuple in R_4. When a tuple T is inserted in node relation 6 due to a response for a goal on the arc from 4 to 6, with T_4 as its context tuple, the back pointer is copied from T_4 into T.

(2) *Multiple common ancestors:* Figure 14 shows a DJG where two joining literal relation tuples may have more than one common ancestor. When a response for p arrives, its context tuple (T_3) is located, and the extended tuple (T_{p1}) is constructed. From the context tuple T_3 we now follow the back pointers to one of the common-ancestor tuples (T_0). This is called the primary common ancestor. The extended tuple T_{p1} is inserted in the linked list that begins at the primary common ancestor, and the forward pointer from there (T_0) is followed into the other literal relation (R_q), as before. While traversing the tuples in the other relation R_q via the linked list, it is checked if their second common ancestor is the same as that of T_{p1}. If not, the tuple is skipped. For example, T_{q2}'s second common ancestor is T_2, whereas T_{p1}'s is T_1. So T_{q2} is not used.

(3) *Multiple joins with identical common ancestors:* Consider the DJG given in Figure 15. (This graph arises in a map-coloring problem.) Node 0 is the common ancestor of two joins: at nodes 3 and 4. Now, each tuple in node relation 0 must keep four forward pointers corresponding to the four literal relations that join at these nodes. The operation of the algorithm is unchanged.

(4) *Nodes with in-degree larger than 2:* This can be handled easily by extending the DJG with null arcs so that all nodes have in-degree ≤ 2. Alternatively, this situation can be handled dynamically by considering arcs two at a time while processing a response. Notice that the common ancestors for each

pair of arcs may be different, even though they all are incident on the same node.

To summarize, each literal-relation tuple carries one pointer to its context tuple. Each node-relation tuple may carry several back pointers. Also, several forward pointers are carried by node-relation tuples at nodes that are primary common ancestors for a join. Notice that which (one of the possibly many) back and forward pointers to follow from the tuples while processing a response and computing a join can be decided at compile time.

3.3. Properties of the Process Model

We now examine the properties of the proposed scheme, and compare it with others. Two of its important properties are completeness and "maximal" parallelism.

3.3.1. Completeness. Completeness is the ability to produce a solution whenever a solution exists, even if the search space is infinite. To isolate the properties of the process models from those of the scheduler, let us assume a fair scheduler (that is, it guarantees that every process that is ready to execute will be executed eventually). The ROPM searches all OR branches in parallel. So every particular solution to a given query is guaranteed to be produced eventually. So the ROPM is complete (see [20] for a proof). In contrast, the AND-OR process model (AOPM) [5] and most of other reported AND-parallel schemes are not complete (however, see the discussion at the end of this section). For example, consider the query '$p(X), q(Y), r(X,Y)$', when p and q are infinite relations. Assume that static analysis has led to a DJG where p and q generate X and Y, and r tests them. Assume r succeeds for the pair (x_5, y_7) where x_5 and y_7 are two solutions to $p(X)$ and $q(Y)$. When the ROPM starts the OR processes $p(X)$ and $q(Y)$, each of them will eventually return the appropriate component of the solution (x_5 and y_7), by the inductive assumption. When the later of the two responses (say y_7) arrives, it will be joined with all the preceding responses to the other literal (including x_5). Obviously, the set of the previous responses at that time is finite, and so the ROPM starts the process for $r(x_5, y_7)$, which eventually succeeds by the same inductive assumption. In the AOPM, the OR processes buffer the solutions and send them up to the AND process one at a time, on demand. Even if we assume the OR processes for p and q receive solutions x_5 and y_7 respectively, they may not be used by the parent process unless they both happen to be the very first values produced. Otherwise (assume, without loss of fairness, that they are both different than x_5 and y_7), the parent process receives X and Y values on which r fails, which leads it to backtrack into one of the two OR processes. Whichever one it backtracks to, it has no chance of success now, as that process will supply an infinite stream of values all of which will fail in r. The same example demonstrates incompleteness of other schemes related to Conery's AOPM [8, 17, 30, 42] and also that of Epilog [40]. The sync model does not have explicit backtracking; however, its synchronization mechanism forces it to be trapped into infinite branches without producing solutions, as illustrated by a different example [22]. The PEPSys scheme [39] may produce a solution for these examples, but it depends on

whether it has created a backtrack point or a branch point for the OR nodes for p and q. With backtrack points, it is incomplete.

There is a tradeoff involved here. Fairly searching many OR branches in parallel requires much storage and/or extra computation. So even a complete evaluation scheme may not be able to produce a particular solution if it runs out of resources. For problems with finite search spaces, an incomplete scheme may be preferred if it is more efficient. We believe that the evaluation scheme should not sacrifice completeness in *anticipation* of lack of resources, since otherwise it would not be able to produce all solutions even when resources are available. Also, a fair scheduler is not infeasibly inefficient. For example, one can use the depth-first iterative deepening techniques [27], which, under reasonable assumptions, find any specific solution in at most twice as much time, and with highly efficient space usage.

3.3.2. "Full" OR *Parallelism.* In this subsection, we show how the ROPM exploits all of the available OR parallelism in a logic program. We then compare the degree of parallelism obtained by the ROPM and some other schemes.

Any two actions (unifications) that belong to different branches of the SLD tree (OR tree) are said to be OR-parallel. A scheme exploits full OR parallelism if it can execute any two such actions in parallel.

Pure OR-parallel schemes such as those described in [3, 15, 37, 38] develop the OR tree, and obviously are able to exploit full OR parallelism. However, by ignoring independent AND parallelism, they not only miss that parallelism, but also create a potentially large amount of redundant work. In processing a query of the form '$p(X), q(Y), r(X, Y)$', where p and q are independent generators of X and Y, a pure OR-parallel scheme calls $q(Y)$ once for each solution of $p(X)$! This redundant computation must be avoided either by catching the results from previous calls, or by incorporating AND parallelism. However, adding deterministic AND parallelism alone to an OR-parallel system will not eliminate this problem, as illustrated by the same example. As P and Q are nondeterministic predicates with multiple possible solutions, such systems will not execute them in parallel.

In the context of an AND-parallel system, subtle interactions between AND and OR parallelism as exploited by a specific scheme may lead to loss of parallelism. We now define three different forms of OR parallelism, as a refinement that helps distinguish between different approaches. Note that all three forms are just manifestations of OR parallelism in different contexts; in particular, a purely OR-parallel system exploits all three forms.

Consider the query 'gen(X), test(X)'. Assume that gen is the producer of X, and may have multiple solutions.

(1) Different clauses for gen can be explored in parallel, exploiting OR parallelism *underneath* the literal gen(X). This is an example of *literal-level* OR *parallelism*.

(2) Independent of whether this parallelism is exploited or not, one may start test(x_1) for a value x_1 returned by gen in parallel with gen looking for more solutions for X. This is called *pipeline parallelism*. Notice that it is a form of OR parallelism because searching for the second value of X belongs to a different branch of the OR tree than the one that tests the first value.

(3) In addition, if two different X-values have already been returned by gen, one may start testing [test(x_2) and test(x_3), say] them in parallel. This parallelism between multiple instances of the consumer literal is called consumer-instance parallelism.

It is easy to see that the ROPM exploits all three forms of OR parallelism. The OR processes start the REDUCE processes for all the matching clauses in parallel, leading to "literal-level OR parallelism". Returned values immediately lead to starting consumer literal processes, irrespective of whether other instances have already been started. Thus it exploits both pipeline and consumer-instance parallelism. In the AOPM, only one value is returned to the parent AND process. If the second value is ready (at the OR process), it misses consumer-instance parallelism. The AOPM does exploit the other two forms. Also, Conery [7] has recently developed a method that splits the parent AND process when there are bindings available in the OR process and there are idle processors, thus exploiting consumer-instance parallelism. The sync model [29] also prevents full exploitation of consumer-instance parallelism, as the sync signals form a barrier which prevents solutions to earlier literals from being used to instantiate new consumer processes. The Epilog system of Wise as described in [40, 41] is interesting in this respect. In the above example, it waits for all the solutions to gen to arrive, and then spawns all test instances in parallel. Thus it exploits consumer-instance parallelism, but leaves the pipeline parallelism unexploited.

The PEPSys model [39] does exploit all three sources of parallelism, with a different process model based on modification of the OR tree. However, the backtrack points are established dynamically (depending on load), which may cause a loss of parallelism and lead to duplication of work (see [39] for details). This is a tradeoff, because the backtrack points are more efficient than parallel branches, which incur scheduling overhead. Only extensive performance studies can tell whether the possibility of duplication is compensated by the efficiency gained.

The following program is a "synthetic benchmark" (adapted from [22]) to test how a given scheme handles the interaction of AND and OR parallelism, and whether it can extract consumer-instance parallelism:

Program 1 (litmus-AND-OR-2).

:- gen(X), test(X,Y).

gen(X) :- $g(P_A, 1, D_A, X)$.

$g(P_A, M, M, X)$:- $p(M, X1)$, busy_work(P_A, $X1$, X).

$g(P_A, M, N, X)$:- $M < N$, Mid is $(M+N)/2$, $M1$ is Mid + 1,
  $g(P_A, M,\text{Mid}, X1)$, $g(P_A, M1, N, X2)$,
  X is $E_{X1,X2}$.

$p(M,M)$.

$p(M,N)$:- N is $M+1$.

test(X,Y) :- busy_work(N,$X1$,Y), $q(Y)$.

Busy_work is simply a sequential predicate of time proportional to its first parameter. The generate phase fans out into D_A AND-parallel calls to g in a

TABLE 1. Degree of extracted parallelism

		Pure AND	Pure OR	AOPM unmodified	ROPM
litmus-AND-OR-2	Generate phase	D_A	2	D_A	$2D_A$
	Test phase	1	2^{D_A}	1	2^{D_A}
8-queens	Maximum	28	4544	28	26400
	Average	≈ 16.9	2245	≈ 16.9	11670

divide-and-conquer fashion. Each leaf of this AND tree picks one of two possible values nondeterministically via p and processes it using busy_work. These values are combined in an arbitrary algebraic expression $E_{X1,X2}$ involving $X1$ and $X2$. Thus it generates 2^{D_A} possible values for X. The parallelism exploited by different schemes in both phases of computations is shown in Table 1. Notice that for a purely OR-parallel scheme the concurrency during the AND phase would have been 2^{D_A} if the redundancy were not eliminated as explained in the previous paragraph. Also, each invocation of the first clause for g results in two parallel calls to busy_work in the ROPM, for the two values returned by p. Thus the ROPM creates $2D_A$ parallel tasks during this phase. The AOPM and the sync model serialize the two busy_work calls and create D_A parallel tasks.

Frequently, recursion compounds the OR parallelism of a computation. A generic example of such a computation is provided by the following program:

```
solve(Selected,Selected) :- finished(Selected).
solve(Selected,[First|Rest]) :- gen(First), test(Selected,First),
    solve([First|Selected],Rest).
```

This program represents a multistage decision process typically used in solving many combinatorial problems. A simple example is a program for solving the well-known nonattacking-queens problem: At each stage, one chooses position of the queen in the current row, tests if it attacks any queens already placed, and if not, extends the solution further. This program also features AND parallelism in the test: A new queen can be tested for "no attack" with each one of the selected queens in parallel. The table shows the maximum and average parallelism obtained from this program by different schemes. Note that we considered checking a pair of queen positions for "no attack" a single operation while computing the entries in the table, rather than counting each individual test within that check. It is worthwhile pointing out that the AND parallelism in this example is speculative in nature. If the test fails for one pair of queens, there is no need to test the other pairs. In this particular example, it may be advantageous to ignore the AND parallelism. However, in general in multistage decision-process computations one frequently finds the generation of alternative steps (gen) to be nonspeculatively AND-parallel.

As another example, consider the following top-level query for finding a prime number that can be expressed as a sum of a Fibonacci number and a perfect number:

```
?- fib(F,1000), perfect(P,30), Q is F + P, isPrime(Q).
```

fib generates values for F up to 1000 in OR-parallel fashion; perfect is similar. For the given limits, an OR-parallel system (without resulting catching) finishes execution in 162 time units with the maximum parallelism of 560. The ROPM, exploiting AND parallelism between fib and perfect, performs the computation in 126 time units with a maximum parallelism of 67. Thus, the ROPM does less work and finishes first. With result caching, an OR-parallel system will perform about the same amount of work as the ROPM, but will sequentialize the calls to fib and perfect, thus leading to a longer time to completion. (These figures were obtained using the ROPM interpreter, and assuming one time unit per resolution.) The AOPM obtains the same degree of maximum parallelism (67) in this program (due to the OR parallelism within generation of Fibonacci and perfect numbers), but completely sequentializes the calls to isPrime, which constitute the bulk (more than half) of this computation. So it finishes execution in approximately 1800 time units. Note that the faster time for ROPM than for AOPM is obtained with identical grain size. The ROPM creates additional parallelism *not* by dividing the computation into finer granules than the AOPM, but by executing in parallel those processes (such as isPrime) that are executed sequentially by AOPM.

The ROPM is particularly suited for such problems, where there is OR parallelism underneath AND-parallel branches (fib and perfect) and there may be dependences among subproblems. Most of the current set of approaches in parallel logic programming, which are purely OR-parallel, or purely AND-parallel, or stream-parallel, are limited in their use in such domains. Proposals for adding a simple form of AND parallelism—that between determinate literals which would produce at most one solution—are also not pertinent here. Such problems occur frequently in AI computations. For example, consider the problem of plan construction. In its pure formulation, its goal is to produce a course of action. At each step in the planning process, an action from multiple alternatives has to be selected. For each such decision, many possible contingencies may exist. Subplans for each such contingency have to be produced recursively. The latter is the AND-parallel part. Some methods for achieving a (sub)goal of a plan may require production of plans for multiple subgoals with possible resource constraints and other dependences between them. This is another source of AND parallelism in such problems. The recursive nature of the plan construction process results in intertwining of AND and OR parallelism.

The REDUCE-OR process model can exploit arbitrary DJGs. Systems that permit only structured DJGs cannot use arbitrary DJGs with multiple common ancestors, such as the one in Figure 14, or those with multiple joins (Figure 15). The inability to exploit such DJGs may seem like just a small sacrifice of AND parallelism for the sake of simplicity and efficiency. In fact, in combination with nondeterminism, the limited DJGs lead to substantial redundant computation. A natural graph for expressing the query in Figure 15 with structured DJGs (which merges node 1 with 2, and node 3 with 4) leads to about 9 times as many calls as the one in Figure 15. (Even the best possible structured graph makes 3 times as many calls.) Schemes described in [8, 16, 39] assume structured DJGs. In most of them, it appears possible to extend them to exploit arbitrary DJGs, and our analysis indicates that they should be so extended.

In summary, the ROPM successfully extracts "full" OR parallelism from programs, in addition to the AND parallelism. Its ability to handle arbitrary (possibly

unstructured) graphs also enhances its parallelism, and ensures that it need not do redundant work. Given that many current multiprocessors have only a few processors, one may ask whether it is worthwhile attempting to exploit large parallelism. However, large-scale parallel processors are becoming commercially available. A 1024-processor Ncube hypercube has been used successfully by researchers on numerical problems [13]. Shared-memory machines with thousands of processors have also been designed [12] and prototyped [34]. In any case, the process model should extract maximal parallelism, and leave the problem of accommodating a small system to the resource allocation algorithms. Of course, the degree of parallelism must be balanced against the overhead incurred in obtaining it. In fact, the ROPM can achieve this higher degree of parallelism at marginal increase in overhead over other parallel-process models for logic programs. This partly due to the optimizations of Section 3.2, strategies for controlling overhead described in the next section (in particular, the scheduling strategy), and implementation techniques cited in Section 5.

4. FLEXIBLE CONTROL OF OVERHEAD

The total number of processes created by the ROPM is no more than that for the AND-OR process model (AOPM), at least while executing all-solutions programs. Also, with the optimization of Section 3.2, the amount of work in a REDUCE process is of the same order as that in the corresponding AND process of the AOPM. However, the number of processes in existence at a given point in time may be much larger in the ROPM than in the AOPM. Also, the amount of storage needed within a REDUCE process may be larger, because it pursues consumer-instance parallelism. Techniques for controlling this space overhead, which sometimes trade away the parallelism, are now presented.

4.1. Completion Detection

A REDUCE process does not need to detect failure or completion of its child processes to ensure its correctness. Nevertheless, significant saving can be obtained by detecting these. When a REDUCE process detects that all its child processes have terminated and all the messages sent by them are received and processed, it can terminate itself, thus releasing resources such as memory. This is very important for combinatorially explosive computations such as search and some divide-and-conquer computations. Although the number of active processes and messages can be kept proportional to the depth of the computation tree, the "dead" processes can consume exponential space, if not recovered. A simple completion-detection algorithm is described in [20]. An efficient version of this has been developed and used in an interpreter for the ROPM.

A related, but more difficult, problem is that of terminating useless computation. When, for example, one of the AND-parallel branches fails, i.e. completes without producing any solution, the other branch carries out futile work. Terminating such computations is quite tricky in an asynchronous parallel system. We have a few schemes for solving this problem, which have been tested in other non-PROLOG computations. They have not been implemented in the interpreter.

Note that completion detection by itself (without termination of useless computation) is quite powerful and effective in controlling space usage.

4.2. Limiting Active Tuples

When the overhead needs to be reduced even at the cost of partial loss of parallelism, we may pursue only one binding for each variable at a time as follows: Each node relation has at most one *active tuple* (which may now be removed from the node relation and merged with an *activation record* of the clause). Multiple responses for a subgoal are stored without firing any new subgoals. A tuple T in a node relation *fails* if there are no active goals with T as their context tuple. The failure-detection algorithm determines when a goal becomes inactive, based on the "failure" and "done" responses from the child processes. Only when a tuple fails is another tuple from the corresponding node relation activated, resulting in firing of OR processes for each literal immediately following the node in the DJG. The tradeoff involved in this decision is the balancing of the loss of parallelism with the reduction in the overhead. The clauses for which this technique is applied behave exactly the same way as in Conery's AND-OR process model [4]. Many optimizations can be carried out for this special case. For example, as there is only one tuple in each relation, all tuples may be combined in a single tuple, and bindings trailed as in a sequential implementation.

An interesting generalization of the above technique is to let the number of active tuples in a node relation be limited to some fixed number (as opposed to just one). At one extreme, with the number $= 1$, this results in the same behavior as the technique above, while at the other, it behaves exactly like a pure REDUCE process, generating maximal parallelism. Thus this mechanism can be used to control the degree of parallelism finely. This strategy is also not implemented in the interpreter, for reasons given below.

4.3. Scheduling Strategy

The scheduling strategy can also be used to control the creation of processes. A simple strategy of processing responses before processing new goals and using a stack discipline for selecting messages (goals or responses) for execution can be quite effective in many situations. For example, when the program is purely AND-parallel or purely OR-parallel, the stack discipline tends to minimize the memory requirement. We find this strategy effective enough that the techniques of the previous subsection are not needed in most situations. In programs with both kinds of parallelism interacting, this simple strategy does control memory usage effectively, but does not minimize the time to first solution. We are developing a priority-based scheme [26] for dealing with this situation. In addition to improving the time to first solution, it also reduces memory usage beyond the strategy mentioned above.

4.4. Throttling Down

If the above techniques prove inadequate in a specific situation, the following strategy, which uses a memory-sensitive mode, can be employed. In this mode,

both REDUCE and OR processes behave somewhat differently, as described below. The relative priorities of processes are also affected in this mode.

(1) Each OR process, when started, has a high priority. It first checks if there are any clauses matching its goal at all. If not, it sends a failure response and terminates. Otherwise, it selects only *facts* that match its goal, sends the corresponding solutions to its parent process, and then decreases its priority. In this low-priority status, it will later start the REDUCE processes for the matching clauses that are not facts.

(2) A REDUCE process may have many responses that it needs to process. In this mode, a REDUCE process scans the list of responses, and selects the *failure* responses for processing. A REDUCE process processing a failure response runs at a higher priority. The rationale behind this strategy is that a failure response may trigger termination of some of its subprocesses, and possibly a failure of the REDUCE process itself, thus freeing their resources.

(3) The REDUCE processes, in this mode, constrain AND parallelism. When a tuple is inserted in a node relation, instead of starting OR processes for *all* literals dependent on the node, it starts them one at a time, starting the second only after the first has finished, and so on. Notice that this behavior is different than making the DJG sequential. If the DJG were sequential, the second of the two independent literals will be unnecessarily solved multiple times, once for every solution of the first one. Here, it is called only once, but the call is delayed until there are solutions to the first one.

An extreme form of control over memory space can be obtained by simply destroying some process, say P, and all its descendents. The parent of process P must adjust its data structures so that it will re-create P eventually. Notice that by our definition of the REDUCE-OR tree, computation of each node is affected only by the nodes in its own subtree. So this strategy does not affect the correctness of the algorithm. However, a thrashing-like behavior in which the same processes get killed and re-created endlessly is possible.

5. ONGOING WORK ON EFFICIENT IMPLEMENTATION

The process model in its full generality requires dynamic creation of many processes, handling of messages between them, management of the tuple relations, and so on. Even though it seems clear that it may lead to increased parallelism, it is reasonable to ask whether it can be efficiently implemented, particularly when compared with existing highly efficient sequential implementations. In this section, we briefly discuss why we believe that an efficient implementation is possible, listing pertinent issues and how we propose to solve them, and citing some of our recent results.

In Section 3, for simplicity of presentation, we ignored the issue of how variable bindings are represented in the ROPM. Conceptually, it suffices to assume that full copies of argument terms are created and sent down to OR processes when they are created, and full copies are made when solutions are returned. Although a straightforward interpretation of the process model description in previous sections would lead to such a scheme, it will clearly be very inefficient for programs

with complex data structures. A more sophisticated *binding environment* is essential for an efficient implementation. We have developed [24] a binding-environment scheme which builds on Lindstrom's early scheme [32]. It is similar to Conery's *closed environments* [6] in its basic approach, in that it deals with a non-stack-based environment, and is suitable for message-passing machines as well as shared-memory ones. The worst-case performance for the ROPM and the binding environment occurs when large terms with deeply embedded logical variables are passed as arguments or results. We plan to work on developing techniques to improve the performance in such cases.

Creation and management of processes and messages is another source of overhead. A system for machine-independent parallel programming called the *chare kernel* simplifies this task considerably. The process-model implementation does not specify where the processes are to execute, or any other machine-specific details. The kernel's run-time support system handles dynamic load balancing, scheduling, etc. We found this separation of concerns quite useful in developing the interpreter cited below. The process creation is also quite efficient—about 200 microseconds—in the chare kernel. The dynamic load-balancing scheme can balance a large number of small-grained tasks and, according to our simulation as well as real machine experiments (on a 64-node iPSC/2 hypercube), is scalable to large multicomputers [25].

An interpreter for the ROPM has been written atop the chare kernel in the C programming language. It accepts annotated logic programs and produces all solutions. The optimizations described in Section 3.3 have not been incorporated in the interpreter. The performance figures mentioned earlier in the paper were obtained by using this interpreter running on a uniprocessor simulating infinite processors. The interpreter runs on a variety of multiprocessors, including a message-passing machine (an iPSC/2 hypercube) and shared-memory machines such as an Alliant FX/8, a Sequent Symmetry, an Encore Multimax, and ORACLE, a multiprocessor simulator that models many interconnection topologies with varying numbers of processors.

The implementation techniques used in the interpreter and the kernel, and their performance analyses, are beyond the scope of this paper. However, the early performance results [36] are encouraging. For example, the interpreter, running with the parallel code on one processor (Encore Multimax), takes about 34.9 seconds to compute 17th Fibonacci number (using the naive algorithm), compared to 34.4 seconds taken by the SBProlog interpreter on the same machine. The Quintus PROLOG interpreter takes 9.5 seconds on a Sun 3/60, which is about twice as fast as the Multimax. Thus our interpreter performance is comparable with SBProlog, and has roughly half the speed of Quintus. Performance on the 6-queens problem (21.7 seconds compared with 20.7 seconds for SBProlog, both on the Multimax, and 5.8 seconds for Quintus on a Sun 3/60) also confirms these ratios. As we do expect to pay some overhead due to our process model, this performance seems satisfactory for an interpreter. With eight processors, our interpreter finishes the Fibonacci computation in 5.1 seconds (speedup 6.84) and the 6-queens in 3.2 seconds (speedup 6.78). On Intel's iPSC/2 hypercube with eight processors, the time goes down from 15.5 seconds on one processor to 3.1 on eight, for the Fibonacci computation. The speedup tends to saturate beyond that, due to the small problem size and small grain size. We expect that with grain-size

control (see below), and with a better memory management scheme which will allow running larger problems on the interpreter, even the interpreter performance will scale up to large distributed-memory machines.

In recent years, it has been conclusively demonstrated that sophisticated compilation leads to a substantial improvement in performance of sequential logic-programming systems. The parallel systems are certainly not an exception [11, 15]. In addition to eliminating interpretive overhead, compilation allows one to specialize the code for each clause and call. Major savings can be obtained by identifying (using static analysis) situations where the full strength of the most general algorithm is not needed, and simplifying code accordingly. For example, if static analysis can determine that all calls in a particular clause return at most one solution each, a REDUCE process for that clause need allocate only one tuple, whereas the general algorithm would maintain multiple singleton relations. Another example: There may be many messages waiting for a REDUCE process. With some additional synchronization it is possible to execute them in parallel. As there is synchronization overhead, such an algorithm should be applied only to the clauses that are likely to benefit from it.

A compiler for the ROPM is being developed. One source of complexity was the fact that we could not use any significant part of the extensive body of work on the Warren abstract machine (WAM), which has been used successfully for sequential PROLOG as well as many parallel PROLOG systems. This is due to the fact that most of the assumptions in the WAM about the context in which it operates are invalid in the ROPM. The binding environment, unification algorithm, term representation, run-time process management system, process model with tuples in relations, etc. are quite different from those found in sequential as well as many parallel systems. All the requisite algorithms have been developed, and a prototype compiler is being tested. The preliminary performance data indicates that the performance of compiled programs will be on par with those compiled with SBProlog and within a small factor of Quintus on uniprocessors. (For example, the Fibonacci(17, N) computation takes 3.8 seconds on the Multimax, compared with 3.1 seconds for SBProlog and 0.58 seconds for Quintus on a Sun 3/60). As compilation improves the efficiency, the time spent on process management starts to become disproportionately large compared to the actual work inside the processes. Fortunately, this proportion can be effectively controlled by using *grain-size control*, i.e. using sequential execution techniques without any process creation etc. within portions of programs which are not worth exploring in parallel.

It should be noted that the ROPM does not need a finer grain size to obtain a higher degree of parallelism. For example, in the query for finding prime numbers that can be expressed as a sum of a Fibonacci and a perfect number (Section 3.3.2), even if we assume that each subgoal of the top level REDUCE process is lumped into a sequential process, the ROPM will lead to more parallelism than do AND-OR and other similar models, because it computes multiple isPrime literals in parallel.

The sources of overhead are thus controlled. Management of tuples and relations is carried out efficiently using the optimization in Section 3.2, a binding environment scheme that obviates the need to form full copies of arguments most of the time, and a low-overhead memory management scheme (for managing

tuples and dynamically constructed terms). The chare kernel provides very efficient process creation, message passing, and load balancing. Further, the remaining overhead is amortized with grain-size control, as the "processes" inside the sequentialized grain do not incur any significant overhead beyond the usual sequential execution cost.

6. SUMMARY

We have developed a viable scheme for parallel execution of logic programs. As its distinguishing feature, the scheme captures parallelism between multiple instances of consumer literals. Such parallelism is known to be important in many AI computations. Unlike the single-binding schemes that consider only one binding for each variable at one time, and thus support AND parallelism and a limited form of OR parallelism, our process model tackles the problems of combining them, and thus fully exploits the OR parallelism also.

We have developed techniques that make the overhead of our scheme manageable, and that allow a flexible and dynamic tradeoff between concurrency and overhead. Implementations on real multiprocessors exist. Enhancements including compilation with relevant static analysis are planned for the near future.

The implementation work cited in Section 5 is being carried out with graduate students B. Ramkumar, W. Shu, and V. Saletore. I am grateful to David S. Warren for his encouragement, support, and guidance. Finally, I would like to thank the anonymous referees for their comments and suggestions.

REFERENCES

1. Carlsson, M., Danhof, K., and Overbeek, R., A Simplified Approach to the Implementation of AND parallelism in an OR parallel environment, in: *International Conference on Logic Programming*, Seattle, 1988.
2. Chang, J., Despain, A. M., and DeGroot, D., AND Parallelism of Logic Programs Based on a Static Data Dependency Analysis, in: *Digest of Papers, IEEE Compcon*, Feb. 1985, pp. 218–225.
3. Ciepielewski, A., and Haridi, S., A formal model for or-parallel execution of logic programs, in: R. E. A. Mason (ed.), *Information Processing*, Elsevier Science, 1983, pp. 299–305.
4. Conery, J. S., The AND / OR Process Model for Parallel Interpretation of Logic Programs, Ph.D. Thesis, Univ. of California, Irvine, June 1983.
5. Conery, J. S. and Kibler, D. F., AND-parallelism and non-determinism in Logic Programs, *New Generation Comput.* 3:43–70 (1985).
6. Conery, J. S., Binding Environments for Parallel Logic Programs in Nonshared Memory Multiprocessors, in: *Proceedings of the 1987 Symposium on Logic Programming*, San Francisco, Sept. 1987, pp. 457–467.
7. Conery, J. S., Personal communication, June 1988.
8. DeGroot, D., Restricted AND-Parallelism, in: *Proceedings of the International Conference on Fifth Generating Computing Systems*, Nov. 1984, pp. 471–478.
9. DeGroot, D. and Chang, J., A Comparison of Two AND-Parallel Execution Models, in: *Proceedings of AFCET Informatique Congress on Hardware and Software Components and Architectures for the Fifth Generation*, Mar. 1985.

10. Debray, S. K., Automatic Mode Inference for Prolog Programs, in: *Proceedings of the Third Symposium on Logic Programming*, Salt Lake City, Sept. 1986.
11. Disz, T., Lusk, E., and Overbeek, R., Experiments with OR-Parallel Logic Programs, in: *Proceedings of Fourth International Conference on Logic Programming*, Melbourne, May 1987.
12. Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L, and Snir, M., The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer, *IEEE Trans. Comput.* C-32:2 (Feb. 1983).
13. Gustafson, J. L. and Montry, G. R., Programming and Performance on a Cube-Connected Architecture, presented at Compcon 88.
14. Halstead, R., Parallel Symbolic Computing, *Computer*, 19:8 (Aug. 1986).
15. Hausman, B., Ciepielewski, and Haridi, S., OR Parallel Prolog Made Efficient on Shared Memory Multiprocessors, in: *Proceedings of the 1987 Symposium on Logic Programming*, San Francisco, Sept. 1987, pp. 69–79.
16. Hermenegildo, M. V. and Nasr, R. I., Efficient Management of Backtracking in AND-parallelism, in: E. Shapiro (ed.), *Proceedings of Third International Logic Programming Conference*, London, July 1986, pp. 40–54.
17. Hermenegildo, M. V., An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel, Ph.D. Thesis, Univ. of Texas at Austin, 1986.
18. Hermenegildo, M. V., Relating Goal Scheduling, Precedence, and the Memory Management in AND Parallel Execution of Logic Programs, in: *Proceedings of Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987, pp. 556–575.
19. Kalé, L. V. and Warren, D. S., A Class of Architectures for Prolog Machines, in: *Proceedings of the Conference on Logic Programming*, Uppsala, Sweden, July 1984, pp. 171–182.
20. Kalé, L. V., Parallel Architectures for Problem Solving, Doctoral Thesis, Dept. of Computer Science, SUNY, Stony Brook, Dec. 1985.
21. Kalé, L. V., Parallel Execution of Logic Programs: The REDUCE-OR Process Model, in: *Proceedings of Fourth International Conference on Logic Programming*, Melbourne, May 1987, pp. 616–632.
22. Kalé, L. V., Completeness and Full Parallelism of Parallel Logic Programming Schemes, in: *Proceedings—1987 Symposium on Logic Programming*, IEEE, Sept. 1987, pp. 125–133.
23. Kalé, L. V., A Tree Rcpresentation for Parallel Problem Solving, in: *Proceedings of AAAI*, Morgan Kaufman, St. Paul, Aug. 1988.
24. Kalé, L. V., Ramkumar, B., and Shu, W., A Memory Organization Independent Binding Environment for AND and OR Parallel Execution of Logic Programs, in: *The Joint International Conference/Symposium on Logic Programming*, Seattle, 1988, pp. 1223–1240.
25. Kalé, L. V., Comparing the Performance of Two Dynamic Load Distribution Methods, in: *Proceedings of the International Conference on Parallel Processing*, St. Charles, Aug. 1988.
26. Kalé, L. V. and Saletore, V., Obtaining First Solutions Fast in Parallel Problem Solving, Report UIUCDCS-R-88-1481, Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign.
27. Korf, R. E., Depth-First Iterative Deepening: An Optimal Admissible Tree Search, *Artif. Intell.* 27:97–109 (1985).
28. Kowalski, R., *Logic for Problem Solving*, Elsevier North-Holland, New York, 1979.
29. Li, P. P. and Martin, A. J., The Sync Model: A Parallel Execution Method for Logic Programming, in: *Proceedings of the Third Symposium on Logic Programming*, Salt Lake City, Sept. 1986, pp. 223–234.

30. Lin, Y., Kumar, V., and Leung, C., An Intelligent Backtracking Algorithm for Parallel Execution of Logic Programs, in: *Proceedings of the Third International Conference on Logic Programming*, London, July 1986, pp. 55–68.
31. Lin, Y. and Kumar, V., An Execution Model for Exploiting AND Parallelism in Logic Programs, *New Generation Comput.* 5(4):393–425 (1988).
32. Lindstrom, G., OR Parallelism on Applicative Architectures, in: *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden, July 1984, pp. 159–170.
33. Nilsson, N. J., in: *Principles of Artificial Intelligence*, Tioga Publishing, 1980.
34. Pfister, G. F., Brantley, W. C., George, D. A., Harvey, S. L., Kleinfelder, W. J., McAuliffe, K. P., Melton, E. A., Norton, V. A., and Weiss, J., The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, presented at International Conference on Parallel Processing, 1985.
35. Reddy, U.S., On the Relationship between Logic and Functional Languages, in: D. DeGroot and G. Lindstrom (eds.), *Logic Programming: Functions Relations and Equations*, 1985.
36. Shu, W., Ramkumar, B., and Kalé, L. V., Implementation and Performance of a Parallel Prolog Interpreter, Report UIUCDCS-R-88-1480, Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, IL 61801, Dec. 1988.
37. Warren, D. H. D., OR Parallel Execution Models of Prolog, in: *Proceedings of the 1987 International Joint Conference on Theory and Practice of Software Development*, Pisa, Italy, 1987, pp. 243–259.
38. Warren, D. H. D., The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues, Invited Paper, in: *Proceedings of the 1987 Symposium on Logic Programming*, San Francisco, Sept. 1987, pp. 92–102.
39. Westphal, H. and Robert, P., The PEPSys Model: Combining Backtracking, AND and OR Parallelism, in: *Proceedings of the 1987 Symposium on Logic Programming*, San Francisco, Sept. 1987, pp. 436–448.
40. Wise, M. J., A Parallel Prolog: The Construction of a Data Driven Model, in: *Proceedings of the 1982 Conference on Lisp and Functional Programming*, 1982, pp. 56–66.
41. Wise, M. J., *Prolog Multiprocessors*, Prentice-Hall International Editions, 1986.
42. Woo, N. and Choe, K., Selecting the Backtrack Literal in the AND Process of the AND/OR Process Model, in: *Symposium on Logic Programming*, Salt Lake City, 1986.