MANIP-2: A MULTICOMPUTER ARCHITECTURE FOR EVALUATING LOGIC PROGRAMS

Guo-jie Li and Benjamin W. Wah
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

ABSTRACT

Logic programs are conventionally evaluated by brute-force depth-first search. To avoid unnecessary searching, an intelligent search strategy that guides the search by heuristic information is desirable. In this paper, the evaluation of a logic program is modeled as the search of an AND/OR tree. A heuristic function using the ratio of the success probability of a subgoal to the estimated overhead of evaluating the subgoal is found to be useful in guiding the search of logic programs. An optimal search strategy that minimizes the expected overhead is proposed and analyzed. The optimal strategy requires a large run-time computational or storage overhead. An efficient heuristic search strategy that can be implemented on a multiprocessor architecture is presented.

INDEX TERMS: AND/OR tree, AND parallelism, heuristic search, logic programming, OR parallelism, pruning, success probability.

1. INTRODUCTION

Logic programming is a programming methodology based on Horn-clause resolution [Kow79]. The efficiency of solving a logic programming problem depends strongly on the many ways of representing the logic program. Evaluating a logic program can be considered as the search of an AND/OR tree [Kow79,CoK83]. The root is labeled by the initial problem to be queried; the OR nodes represent (sub)goals; and the AND nodes represent clauses.* All subgoals in the same body of a clause are children of an AND node. A (sub)goal (OR node) and its children display the nondeterministic choices of clauses with the same head. The terminal nodes denote clauses or subgoals that cannot be decomposed. Figure 1(a) shows an example of the AND/OR-tree representation of a logic program. In general, a logic program without any inference loop can be viewed as an acyclic AND/OR graph. A finite AND/OR tree is obtained from the AND/OR graph by duplicating common descendent nodes.

The AND/OR tree in Figure 1(a) can be represented more clearly in Figure 1(b) as a high-level OR tree involving the selection of all combinations of clauses, and multiple low-level AND trees representing the solution tree resulted from clauses selected in the OR tree. Parallel processing can be applied to evaluate the multiple solution trees in parallel (OR-parallelism), or can be applied to search a solution tree in parallel (AND-parallelism). Note that the number of edges in this representation could be much large than that of an AND/OR-tree representation.

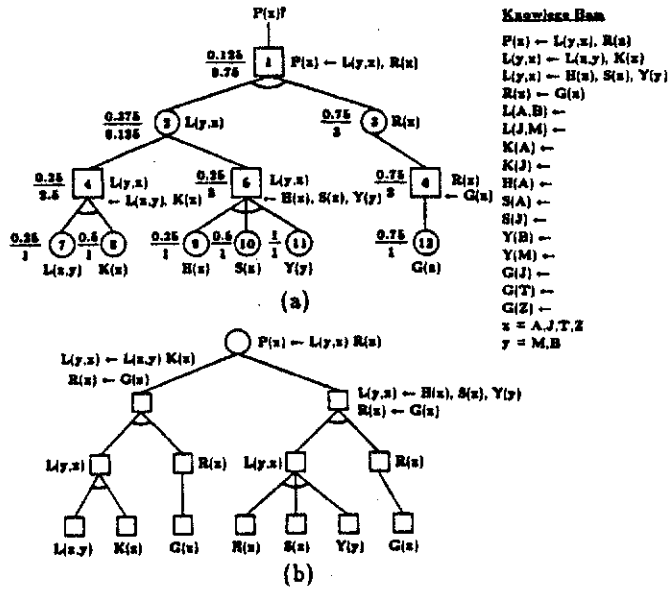

Figure 1. (a) An example of a logic program represented as ar AND/OR tree. (OR nodes are represented as circular nodes; AND nodes are represented as squared nodes. In the logic program, P: happy; L: like; R: rich; K: kind; H: handsome; S: strong; Y: young; G: goodjob; A: Ares; B: Betty; J: John; M: Mary; T: Tom; Z: Zeus. The numbers outside each node are the ratio of success probability to expected search cost.) (b) The same logic program represented as a hierarchy of OR tree followed by AND trees.

AND-parallelism involves the simultaneous execution of subgoals in a clause. Since subgoals within a clause can share variables, the binding of variables of concurrently executing subgoals must be coordinated to avoid conflicts of a shared variable being bound to more than one value. AND-parallelism is limited by the measures to avoid conflicts. One approach to avoid conflicts is to annotate variables to indicate which subgoals can bind values to specific variables and which cannot [CIG83]. In particular, only one subgoal, called *producer*, is allowed to bind a value to a variable. Conery proposed a complex, non-annotated, process-structured system that dynamically monitors variables and continually develops data dependency networks to control the order of execution of subgoals, never allowing two potential producers for the same variable to execute in parallel [CoK83]. DeGroot described a method to obtain restricted AND-parallelism by compile-time creation of a parallel execution-graph expression for each program clause [DeG84].

Research supported by CIDMAC, a research unit of Purdue University, sponsored by Purdue, Cincinnati Milicron Corporation, Control Data Corporation, Cummins Engine Company, Ransburg Corporation, and TRW.

International Conference on Parallel Processing, 1985.

* The definitions of AND and OR nodes are taken from [MaM73]. The roles of the AND and OR nodes are reversed in Nilsson's definition [Nil80].

In OR-parallelism, all subgoals are independent of each other, and consistency checks for shared variables needed in AND-parallelism are avoided [Kow79,Mot84]. However, an OR-tree representation is inefficient due to the large number of branches needed as compared to that of an AND/OR tree. Given an initial query, ← A, B, with n ways of solving A and m ways of solving B, the OR tree contains n×m branches, whereas an AND/OR tree contains n+m. To improve the efficiency of an OR-tree search of logic programs, several models that modify pure OR-parallelism, such as introducing process bundles [YaN84] and bagof [CiH84], are proposed.

In this paper, we study parallel processing for an AND/OR-tree representation and exploit both AND-parallelism and OR-parallelism. The search strategy developed can be extended to the corresponding AND/OR-graph representation. The search algorithm is generally considered different from an AND/OR game-tree search for the following reasons. First, in contrast to combinatorial-extremum searches that find the best solution, solving a logic program corresponds to finding any (or all) solution(s) satisfying the given conditions, the implicative Horn clauses, and the consistent binding of variables for the subgoals. Second, the value of a node in the AND/OR tree for a logic program is either TRUE (success) or FALSE (failure). The selection of a node for evaluation is usually based on a fixed order, and heuristic information to guide the search is not available. Third, a variable in a logic program can be bound to several values, and some subgoals may share a common variable. For example, in Figure 1, some subgoals share variable x, and x can be bound to any of Ares, John, Tom, and Zeus. For a particular variable in a subgoal, a subset of its possible values may be allowed. In contrast, the nodes in a game tree are independent. Lastly, pruning rules for evaluating the AND/OR tree of a logic program are different from α-β pruning due to the binary values returned by the terminal nodes.

Since the search space of a logic program is large, an intelligent search strategy that guides the search is very important [Pea84]. Parallel processing is not useful here because it is generally used to improve the computational efficiency of solving a given problem, but *not to extend the solvable problem space of the problem*, especially when the problem space is exponentially large [WLY85]. The efficiency of a search strategy can be improved by guiding the search with heuristic information, and reducing the search space by pruning.

Heuristic information to guide the search, such as the success probability of each subgoal or clause and the estimated overhead (or *cost*) of searching a subgoal or clause, remain an open problem. In this paper, a heuristic search based on the information of the ratio of success probability to estimated cost is studied. This search strategy is aimed to minimize the expected search cost and the dynamic run-time overhead of evaluating logic programs.

Pruning is used to eliminate unnecessary expansions when it is determined that a solution tree cannot be found from a subgoal or clause. Two kinds of pruning exist. In *AND-pruning*, if one successor of an AND node for a given binding of values to variables is known to be FALSE, then all its remaining successors for the same binding can be pruned. Likewise, if one solution is sought, then *OR-pruning* can be applied to prune other successors of an OR node once one of its successors is known to be TRUE. In this paper, it is assumed that one solution tree is sought from an AND/OR-tree representation of a logic program, and hence both AND-pruning and OR-pruning can be applied.

Pruning and parallelism are conceptually illustrated in Figure 2(a). In a sequential depth-first search, if Node 1 fails, then Node 2 will be examined next, otherwise Node 3 will be examined. Similarly, the traversal of Node 5 depends on the results of traversing Nodes 1, 2, 3, and 4. This dependence

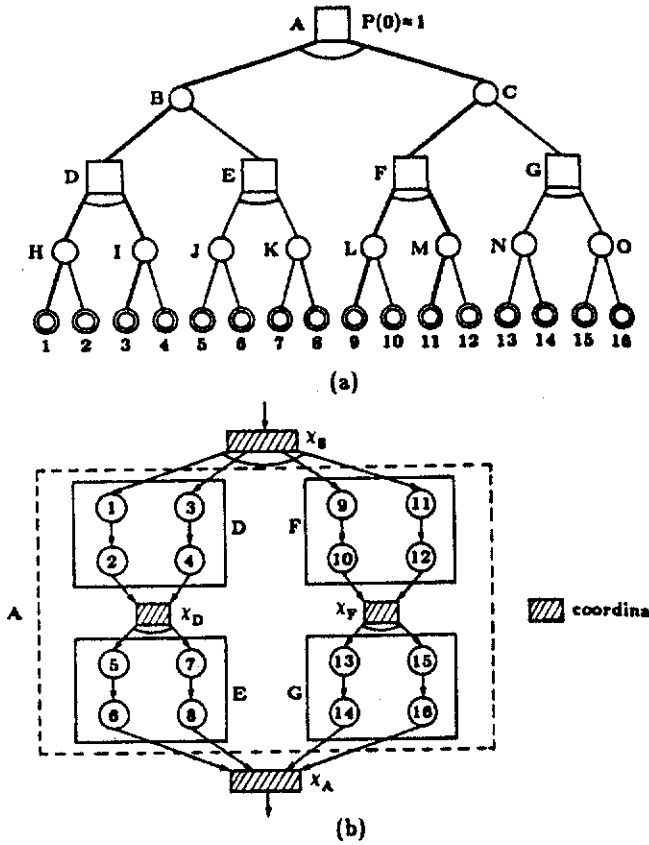

Figure 2. (a) A binary AND/OR search tree with high success probability (circular nodes represent OR nodes; squared nodes represent AND nodes). (b) The corresponding fail-token-flow graph, G_f.

information can be represented in a *fail-token-flow graph*, G_f, as depicted in Figure 2(b) for the tree in Figure 2(a). A node (circle) in the graph will be active only if it receives a fail-token from an incident edge. When a terminal node is found to be FALSE, a fail-token is sent along the direction of the corresponding edge. The coordinator (shaded box) in the graph coordinates the activities of the connected blocks. When a fail-token is received from any incident edge of a coordinator, fail-tokens are sent to *all* directly connected nodes. At the same time, any node searched in the block directly connected to this coordinator can be terminated because it does not belong to the solution tree. For example, when Node 1 is found to be FALSE, then a fail-token is sent to Node 2. If Node 2 is found to be FALSE, then a fail-token is sent to Coordinator x_D. At this time, any node concurrently searched in Block D can be terminated. When a solution tree is found, there is one node in each column of G_f that returns TRUE. G_f can be used to represent pruning in AND-parallelism when the success probability is high and most of the terminal nodes are TRUE.

On the other hand, when the success probability of the goal is low, most of terminal nodes are FALSE. The search for the inexistence of a solution tree in OR-parallelism can now be represented by the *success-token-flow graph*, G_s. G_s is the dual of G_f in the sense that a fail-token is replaced by a success-token, and the columns in G_f are transposed to become the rows in G_s. Referring to Figure 2(a), Nodes 1, 2, 5, and 6 are assigned to four processors. If any of Nodes 1, 2, 5, and 6 succeeds, then a success-token is generated, and the next connected node is assigned to an idle processor. Since

most of the terminal nodes are FALSE, the search will be completed when a small number of nodes have been searched in each column of G_p.

When two AND/OR subtrees are searched in parallel, more work than necessary might be performed if the pruning information of one processor is unavailable to other processors. The extra work that must be carried out due to a lack of pruning information is called the *information-deficiency overhead*. Pruning information can be exchanged by messages or through a common memory. This increased communication overhead needed for passing this information is called the *information-transfer overhead*. In general, a tradeoff exists between the information-deficiency and information-transfer overheads. If better pruning is obtained by increasing the information-transfer overhead, then the information-deficiency overhead will decrease. A good parallel search algorithm should consider these tradeoffs and reduce the run-time overheads by proper assignments of tasks to processors.

Several parallel models of logic programs and the corresponding multiprocessing architectures have been studied [FuN82,KKM83, Mot84,DeG84,CiH84,HaA84,StM84]. Nearly all these architectures were based on ad hoc search strategies and scheduling methods. In this paper, we propose MANIP-2**, a multicomputer architecture to evaluate logic programs. However, *the goal here is not in describing the details of an architecture, but in giving the theoretical foundation of the necessary search algorithm upon which the architecture is based.* The emphasis of this paper is in showing the reasonableness of heuristic searching and the feasibility of an effective scheduling method.

2. HEURISTIC INFORMATION FOR SEARCHING AND/OR TREES

The useful heuristic information to guide the search include the predicted success probability of a solution tree being found from a subgoal or clause, and the associated average cost of finding the solution tree.

The success probability of a node (or alternatively a branch) in an AND/OR-tree representation of a logic program is an a priori probability that reflects the possibility of finding a solution tree over all unifications from this node. These probabilities are used to guide the search initially, and will be improved after more dynamic pruning information is obtained.

For a terminal node with variable x, its success probability is $m(x)/n(x)$, where $n(x)$ is the total number of values that variable x can acquire, and $m(x)$ is the number of values acquired by x in this terminal node. It is assumed that all values in the domain of a variable are equally likely to be assigned to a subgoal. When a subgoal shares more than one variable with other subgoals, its success probability can be computed as the product of the success probabilities of the variables if all variables are independent. In general, the success probability of a node cannot be directly determinated by the success probabilities of its immediate descendents, which may be correlated (the descendent nodes may contain shared clauses or subgoals that renders them dependent). It may have to be evaluated from actual information in the knowledge base. For example, in Figure 1(a), four values can be bound to x, and two can be bound to y. For the eight combinations of values of x and y, only two of them exist in the knowledge base. Hence, the success probability of L(x,y) is 0.25.

Assuming that a nonterminal node K has two immediate descendents, K_1 and K_2, let $P(\cdot)$ be the a priori success probability of a node. Then

$$P(K) = \begin{cases} P(K_1)\cdot P(K_2|K_1) & (K \text{ is AND}) \\ P(K_1)+P(K_2)-P(K_1)\cdot P(K_2|K_1) & (K \text{ is OR}) \end{cases} \quad (1)$$

Eq. (1) can be generalized to nodes with more than two descendents. The computation of the a priori conditional probabilities can be complex due to the shared variables.

The success probabilities of a logic program can also be either assigned initially by the designer, or determinated by statistic collected during execution. In the latter case, no a prior probability is available before the program is executed, and all branches are assumed to have equal possibility of success. A deterministic search strategy, such as a depth-first search, has to be used initially. For example, in Figure 1, the success probability of the clause $L(y,x) \leftarrow L(x,y)\cdot K(x)$ is the probability that both L(x,y) and K(x) succeed. However, L(x,y) and K(x) are dependent, and the success probability of L(y,x) would be difficult to compute. Statistic collected show that the success probability is 0.25. Other success probabilities in Figure 1 were computed by enumerations of all combinations of binding of variables.

Since unification has a linear complexity [PaW78, MaM82], the search cost can be defined by the number of nodes traversed before a solution tree is found to exist or not exist. The cost for searching a subtree depends on the structure and dependence of the subtrees, the query used, and the partial search results already obtained. One way is to define an average search cost based on the estimated probabilities of success. For node K with descendents K_1 and K_2 and assuming that K_1 is searched first, the average search cost is

$$C(K) = C(K_1) + \begin{cases} C(K_2|K_1)\cdot P(K_1) & (K \text{ is AND}) \\ C(K_2|K_1)\cdot(1-P(K_1)) & (K \text{ is OR}) \end{cases} \quad (2)$$

Simulations have shown that the average search cost depends on the structure of the tree, but is quite insensitive to changes in the success probability. A complete binary AND/OR tree with unitary search cost at the terminal nodes was assumed. The simulation results depicted in Figure 3 indicate the relationship between the success probabilities of the root of trees with height 12 and 16, respectively, and the associated average search cost. The average cost is the smallest when p is either zero or one, and is maximum when p is around 0.5. Moreover, the difference between the maximum

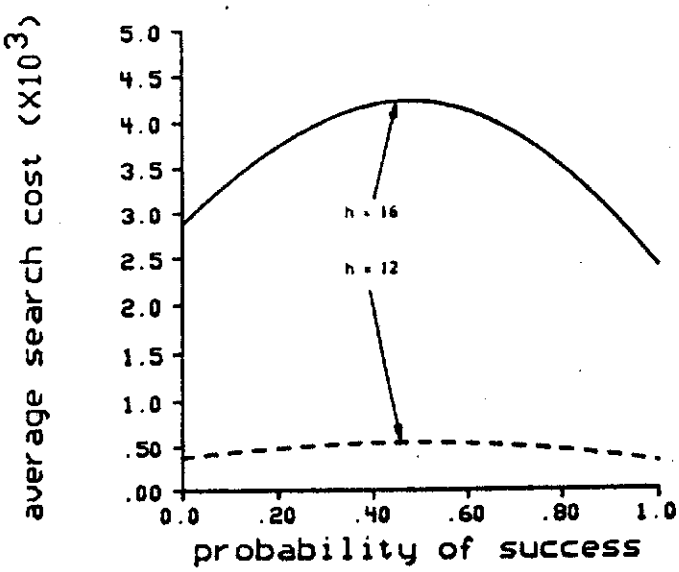

Figure 3. A plot of the search cost versus the success probability of the root.

** MANIP is a multicomputer architecture proposed earlier to evaluate parallel branch-and-bound algorithms with a best-first search, which is an OR-tree heuristic search guided by lower bounds of subproblems [WaM84].

and the minimum costs is relatively small, reflecting the insensitivity of the expected search cost with respect to the success probability.

The average search cost of a subgoal is difficult to be formulated mathematically because it is related to the dependence of descendent subgoals (due to the shared clauses and subgoals). Moreover, the average search cost depends on the search strategy and the order that values are bound to variables, which in turn are driven by the average search costs. Hence the search costs would have to be initially estimated from statistic collected by a given search strategy. As better search costs are obtained, the search will become more efficient, and better estimates on the search costs can be obtained. For example, in Figure 1, the average search cost for L(y,x) was computed by averaging the search cost to verify the result of L(y,x) for all combinations of values of x and y. The search strategy used in computing the costs in Figure 1 assumed a left-to-right traversal of the descendents.

The expected search cost of a subgoal represents an average over all possible queries, all possible paths leading to this subgoal, and all possible combinations of values of variables for a given search strategy. However, when the path leading to a given subgoal is known, the minimum cost and the associated success probability of obtaining a solution tree are better measures to guide the search. Of course, this will result in an enormously large amount of stored information for each subgoal that renders the scheme impractical.

3. HEURISTIC SEARCH FOR LOGIC PROGRAMS

Let P(x) and C(x) be the probability of success and the associated average search cost for node x. Define the criteria Φ_s and Φ_o for any node x as

$$\Phi_s(x) = \frac{P(x)}{C(x)} \quad \text{(x is descendent of an OR node)} \quad (3)$$

$$\Phi_o(x) = \frac{1-P(x)}{C(x)} \quad \text{(x is descendent of an AND node)} \quad (4)$$

Simon and Kadane have studied the optimal OR-tree search and have proved that the search sequence $b = b_1, ..., b_n$ is optimal iff $\Phi_s(b_i) \geq \Phi_s(b_{i+1})$, where the b_is are descendents of an OR node b with preccdence relationships [SiK75]. Barnett has extended their results to optimal search from AND nodes with the assumption that all immediate descendents are independent [Bar83]. Garey has proved that the optimal sequence of performing a set of tasks until one of them fails or all tasks are fulfilled is in descending order of Φ_o [Gar73].

For a pure OR-tree or AND-tree search, the search order is well defined by a single heuristic function. However, when an AND/OR tree is searched, there are two criteria, Φ_s and Φ_o, to order the AND and OR nodes, respectively. Hence a complete order cannot be defined for all active nodes. To resolve this problem, we can decompose an AND/OR tree into a hierarchy of a single OR tree, each terminal of which is an AND tree (Figure 1(b)). The following theorem relates the criteria Φ_s and Φ_o and defines an optimal search order for an AND/OR tree with dependent nodes.

Theorem 1: Suppose that an OR node K has n immediate descendent AND nodes, $K_1, ..., K_n$, and that the AND node K_i, $1 \leq i \leq n$, has i_m immediate descendent OR nodes, $K_{i_1}, ..., K_{i_m}$. If $\Phi_s(K_i) \geq \Phi_s(K_{i+1})$, then the expected search cost C(K) is minimum when all descendents of K_i are searched before K_{i+1}.

Proof: Suppose that K_{i_1} is found to be TRUE, then the conjunction of the remaining subgoals of K_i, namely, $K_{i_2}, ..., K_{i_m}$, forms a new AND node, K_{i_2}'. The conditional probability of $K_{i_2}, ..., K_{i_m}$ being TRUE, given that K_{i_1} is TRUE is

$$P(K_{i_2}') = P(K_{i_2}, ..., K_{i_m} \mid K_{i_1}) = \frac{P(K_i)}{P(K_{i_1})} \quad (5)$$

To get the optimal strategy, we need to compare the probability-to-cost ratios in respect to node K_{i_2}' and K_{i+1}. Using Eq's (2) and (5),

$$\Phi_s(K_{i_2}') = \frac{P(K_{i_2}')}{C(K_{i_2}')} = \frac{P(K_i)/P(K_{i_1})}{C(K_{i_2}, ..., K_{i_m} \mid K_{i_1})} \quad (6)$$

$$\geq \frac{P(K_i)}{C(K_i)} \geq \frac{P(K_{i+1})}{C(K_{i+1})} = \Phi_s(K_{i+1})$$

Eq. (6) implies that the remaining subgoals $K_{i_2}, ..., K_{i_m}$ should be unified first before K_{i+1}. If subgoal K_{i_2} is found to be TRUE, then the above proof can be applied again to show that the remaining subgoals of K_i should be unified before K_{i+1}. □

Theorem 1 shows that the optimal strategy of searching an AND/OR tree is to select the most promising solution tree with the largest Φ_s value among all possible solutions trees, and to examine AND nodes in this solution tree in descending order of Φ_o values. The search is switched to the next best solution tree (with the next largest Φ_s value) if the first solution tree fails.

The key issue in performing the above optimal strategy is to find the most promising solution tree with the largest Φ_s value, and for the solution tree selected, the node with the largest Φ_o value. If h, the height of a complete AND/OR tree, is taken as a measure of the problem size, then it is unlikely that a polynomial-time algorithm exists for finding the most promising solution tree. Let B_o and B_s be the numbers of branches of each OR and AND node, respectively. If $B_s = 1$, then there are $B_o^{h/2}$ possible solution trees, each of which consists of one node. To get the maximum Φ_s over all solution trees, at least $B_o^{h/2} - 1$ comparisons are needed. This is the lower-bound complexity for computing the largest Φ_s when $B_s = 1$. In general, if $B_s > 1$, then before Theorem 1 can be applied, the AND/OR tree has to be transformed into a hierarchy of an OR tree, each terminal of which is an AND tree. The number of terminal nodes in the transformed tree has a lower bound of $O((B_o^{B_s})^{B_o^{h/2-1}})$. Hence to select the largest Φ_s and Φ_o values would require an exceedingly large amount of computational time. Another approach is to store these values associated with each node in the AND/OR tree and to retrieve the decisions in real time. Unfortunately, this would require a large amount of storage space with a lower bound of the same complexity as stated above. Therefore, it is unrealistic to apply the optimal search strategy in respect to an AND/OR-tree search.

Owing to the intractable complexity of propagating the Φ_s and Φ_o values of all terminal nodes to the root in implementing the optimal AND/OR-tree search, an efficient top-down heuristic search is proposed here. As stated in Section 2, each node in the AND/OR tree can be assigned an estimated success probability and the associated expected search cost. These values, together with the information of the path leading from the root to this node, can be used to compute a heuristic value for the node. The search will be guided by the heuristic values.

A *solution tree* is a tree such that all nonterminals are AND nodes and all terminals are OR nodes. To minimize the search effort for a *solution tree*, it is necessary to first find one of the FALSE terminals in this tree, since the search can be terminated once this node is found. This method will be used to compute the heuristic values for AND nodes. From the duality between AND and OR nodes, a *failure tree* is a tree such that all nonterminals are OR nodes and all terminals are AND nodes. A failure tree is the dual of a solution tree, such that the entire AND/OR tree is FALSE if all nodes in the

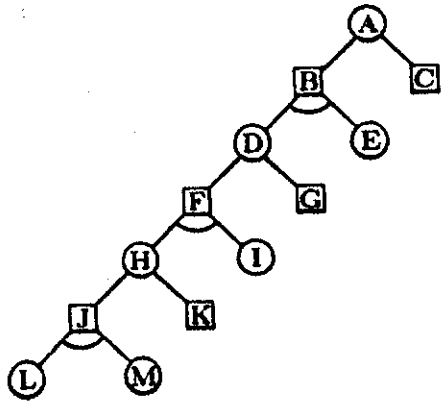

Figure 4. Computation of the heuristic values (OR nodes are represented as circular nodes; AND nodes are represented as squared nodes).

failure tree are FALSE. To stop the search of a failure tree as early as possible, it is necessary to verify that one of the terminals is TRUE. This method will be used to compute the heuristic values for OR nodes.

Referring to the partial binary AND/OR tree in Figure 4, suppose that AND-node F is to be searched. For the goal to be TRUE, Nodes E and F must be TRUE. This is the information that can be extracted directly from the path leading from the root to F. The heuristic value of Node F will be the ratio of the probability of verifying Node A to be TRUE to the associated search cost. This can be computed as the ratio of the probability of success of Nodes E and F to the total average search cost of verifying that Nodes E and F are TRUE or any of them is FALSE. Note that Nodes E and F may be dependent. Other information, such as nodes searched from E, would be also crucial to computing the heuristic value for F. However, this information cannot be extracted directly from the path, and would require a high run-time overhead to maintain, hence will not be considered here. Similarly, the heuristic value of AND-node J is based on information about Nodes E, I, and J. In general, the heuristic value $\Phi_s(x)$ of an AND node x is based on information extractable from the path leading from the root to x such that the goal can be verified to be TRUE. The information on the complete path from the root to any AND node must be maintained with each AND node.

To compute $\Phi_s(x)$ for an OR node x, the information extractable from the path leading from the root to x such that the goal can be proved to be FALSE will be used. Complete information on the path from the root must also be maintained for each OR node. Referring to Figure 4, in computing the heuristic value for OR-node H, the goal will be FALSE if Nodes C, G, and H are FALSE. Note that these nodes may be dependent.

The following heuristic-search algorithm, BAO, is a top-down best-first search algorithm that uses heuristic information about a particular set of solution trees for a given AND node and a particular set of failure trees for a given OR node. It is assumed that the values of Φ_s are available for each set of clauses with the same head (an OR node) in the logic program. Similarly, the values of Φ_s for each subgoal in a clause (an AND node) are also available. It is further assumed that the values bound to a variable are independent and in a fixed order, and that the values of Φ_s and Φ_s are applicable to all instantiations. The following procedure is applied iteratively by binding each variable to a constant value (or to a set of constant values that could be the domain of the variable) until the goal is proved to be TRUE or FALSE. There is an Active List containing tasks in decreasing order of heuristic values. Without loss of generality, assume that the goal node, S, is an OR node.

BAO--Heuristic Search Algorithm for one solution tree:

(1) (*Initialization*): Initiate search from S. If S is known to be TRUE or FALSE, then stop. Otherwise, create a task for S with the information that S is an OR node, and compute its heuristic value. The task is inserted into the Active List.

(2) (*Decomposition*): Select a task T from the Active List with the maximum heuristic value. If T is a ground node, then go to Step 3. If T contains a variable that is not bound, a constant value (or a set of constant values that could be the domain of the variable) is bound to the variable. Decompose the task (an AND or OR node) into its immediate descendent tasks. The new tasks, with their heuristic values, the information about the path from S, and the values bound to variables are inserted into the Active List in the proper order. Go to Step 2.

(3) (*Pruning*): For Task T under consideration, the following steps will be carried out based on values returned by the ground node.
   (a) The subgoal or clause (T_s) on the path from the root to T that is nearest to the root and becomes TRUE or FALSE is found.
   (b) The information on T_s is incorporated into all tasks in the Active List. Subtasks in some tasks can be removed based on T_s. If all subtasks within a task W are removed, then pruning will be carried out recursively on W.
   (c) The values successfully or unsuccessfully bound to variables are broadcast to all active tasks.
   (d) The heuristic values for all active tasks are updated. The active tasks are reordered if necessary.

(4) (*Termination*): If S is terminated, then return success with the bound values. If S cannot be bound to any new value, then return failure. Otherwise, go to Step 2.

There are several considerations when Algorithm BAO is implemented. First, a single list of active tasks is kept, and the AND and OR nodes are not distinguished. Depending on the heuristic values and whether the node concerned is an AND or OR node, the objective is to either prove that the goal is TRUE or prove that it is FALSE. Second, pruning performed in Step (3) requires a large overhead because the entire Active List has to be updated and reordered. However, the merits on the number of tasks eliminated and the better heuristic values generated are doubtful. To avoid this run-time overhead, pruning can be performed after a task is selected from the Active List (Step 2). In this case, all pruning information will be recorded in a common list. Of course, this may result in some unnecessary tasks in the Active List. Third, the computation of the heuristic values could be complex when all dependency of subgoals and variables are considered. In practice, some simplifying assumptions may be made in combining the heuristic values together. For example, a heuristic function to compute the success probability and cost of a conjunction of subgoals could be the product of the success probabilities and the sum of the associated costs of the subgoals. Lastly, the algorithm investigates many possible solution trees in parallel by switching from one to another based on the heuristic information obtained. This results in a large number of tasks in the Active List, which is a fundamental problem in heuristic searches. To reduce the storage space required, static analysis can be performed to arrange the clauses with the same head and the subgoals in each clause in a proper order, and to use a depth-first strategy to search the AND/OR tree. Of course, the order defined will be based on the average over all possible queries and all possible ways of reaching a particular subgoal.

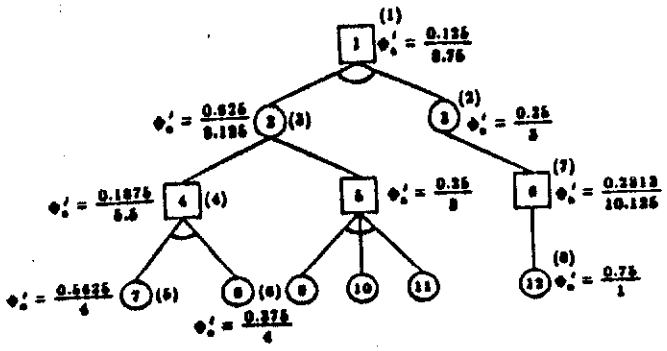

Figure 5. Illustration of the BAO procedure using the same example shown in Figure 1. (OR nodes are represented as circular nodes; AND nodes are represented as squared nodes. The number in parenthesis outside each node shows the order that this node is examined in the BAO procedure.)

The BAO algorithm is illustrated in Figure 5 by explaining snapshots of solving the logic program in Figure 1. The cost of each unification is assumed to be unity. The query is "Who is happy?" Node 1 is unified first. For simplicity, suppose that all successful bindings are equally likely, and that x is first bound to John. Let the updated heuristic values be Φ_a' and Φ_o', respectively. After decomposition, subgoals L(y,x) (Node 2) and R(x) (Node 3) with the corresponding heuristic values are inserted into the Active List. Since $\Phi_o'(3) > \Phi_o'(2)$, Node 3 is selected, and Node 6 is created. In computing $\Phi_a'(6)$, the goal is expected to be TRUE, implying that Node 2 and 6 are TRUE. Hence the heuristic information of Node 2 must be included. The correlation between Nodes 2 and 6 may be complex. For simplicity, Nodes 2 and 6 are assumed to be independent. $P'(6)$ (resp. $C'(6)$), the new success probability (resp. new average cost) of Node 6, is the product (resp. summation) of the original success probabilities (resp. original average costs) of Nodes 2 and 6. $\Phi_a'(6)$ becomes 0.2813/10.125. As $\Phi_o'(2) > \Phi_a'(6)$, Node 2 is selected in the next iteration, and Nodes 4 and 5 are created. Intuitively, Node 6 is likely to be TRUE, but to terminate the search for the goal as soon as possible, the active node that may fail first should be investigated. In computing $\Phi_a'(4)$ and $\Phi_a'(5)$, the heuristic information on Node 3 must be included. For example, $\Phi_a'(4)$ is computed as $(0.75 \times 0.25)/(2.5+3)$. Suppose that variable y is first bound to Mary. Node 4 is next selected, as $\Phi_a'(4)$ is the largest among all active nodes. This means that the possible solution trees involving Node 4 is the most promising. Nodes 7 and 8 are inserted into the Active List in iteration 4. Note that in computing $\Phi_o'(7)$ and $\Phi_o'(8)$, $\Phi_a(5)$ rather than $\Phi_a'(5)$ is used. The reason for this is that the goal is expected to be FALSE if Node 7 or 8 is selected, and the failure probability and cost of Node 5 are needed in this case, but the information on Node 3 (which is included in $\Phi_a'(5)$) does not affect the decision on Node 7 or 8. $\Phi_o(7)$ is computed as $(0.75 \times 0.75)/(1+3)$. Once Nodes 7 and 8 are instantiated in Iterations 5 and 6, respectively, Node 2 is known to be TRUE by values returned from the ground terms. Node 5 can thus be pruned by OR-pruning if a single solution tree is sought. In the last two iterations, Nodes 6 and 12 are unified, and the solution is "John is happy."

4. PARALLEL HEURISTIC SEARCH OF LOGIC PROGRAMS

In this section, we study the problems associated with the parallel processing of the AND/OR-tree representations of logic programs. By minor modifications of MANIP [WLY84],

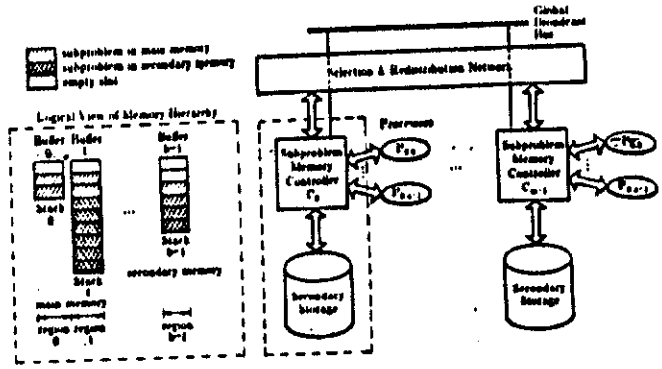

Figure 6. The architecture of MANIP-2, a multiprocessor for parallel heuristic search of logic programs and AND/OR graphs.

parallel heuristic search of logic programs can be carried out efficiently. The architecture of MANIP-2 is shown in Figure 6.

The first problem in parallel processing is the unification of shared variables. When AND-parallelism is involved, variable binding conflicts must be prevented. A lot of efforts have been devoted to solving this problem [ClG83,CoK83, DeG84,Sha83]. We do not attempt to propose a new method to overcome this problem. Instead, we assume one or more of the following conditions: (a) that the subgoals do not share variables; (b) that the shared variables are uninstantiated; (c) that a producer subgoal has bound one or more values to a shared variable and forwards them to many consumer subgoals. In any of these cases, processors can carry out tasks independently.

The second problem is the granularity of parallelism. If the granularity is small, then the information-transfer overhead will be high. In contrast, if the granularity is large, then the information-deficiency overhead will be high, and the degree of parallel processing may be small. The proper granularity depends on the capacity of the communication network, the number of processors, and the relative overhead between the top-down unification of variables and the bottom-up return of solution values. In searching a logic program, if the cost of a subgoal selected is smaller than the defined granularity, then the subgoal is considered as an indivisible operation and processed by a single processor.

Once the granularity is determined, the next problem is to determine the scheduling of the k parallel processors. Given a list of active tasks ordered by decreasing probability-to-cost ratios, the problem is to determine the number of processors to evaluate each task in parallel. To minimize the expected completion time, the problem can be formulated into a complex integer-programming optimization problem. Moreover, the processors have to be rescheduled again once a given solution tree is found to be TRUE or FALSE. Considering the facts that the scheduling algorithm is complex and that static scheduling is not feasible, we decide to assign a free processor to each task in the Active List. When a task has reached its minimum granularity, it will be processed by a single processor until completion.

Another problem is on resource sharing. For a set of processors evaluating a solution tree, it is necessary to search those subtrees with the largest failure-probability-to-cost ratios. Likewise, for sets of processors examining different solution trees, the possible solution trees with the largest success-probability-to-cost ratios must be evaluated. Further, load balancing must be carried out to keep all processors busy. The ring network in MANIP is adequate for load balancing the processors.

The parallel selection problem has been studied thoroughly in MANIP. In the parallel evaluation of branch-

and-bound algorithms, a set of subproblems with the minimum lower bounds must be expanded. The selection overhead has been found to be high; furthermore, the selection rule is based on a fallible lower-bound heuristic. Therefore, it might be more efficient not to follow the selection rule strictly. A no-wait policy has been proposed [WaM84,WLY85]. Instead of waiting for one of the k subproblems with the smallest lower bounds, each processor would expand the "most promising" subproblem in its local memory and initiate a fetch of the "most promising" subproblem from its neighbors. In this case, the "most promising" subproblem is the one with the minimum lower bound. Evaluations for MANIP have shown that, when the k most promising subproblems are randomly distributed among the processors, the average fraction of processors containing one or more of the most promising subproblems is at least 0.63, resulting in a speedup proportional to 0.63k. Further, with occasional redistribution of subproblems with small lower bounds using a ring network, the performance is almost as good as that of a complete selection.

The no-wait policy can be applied here to schedule processors for evaluating subgoals in its local memory. Subgoals with large failure-probability-to-cost ratios (resp. large success-probability-to-cost ratios) and a suitable granularity can be sent to neighboring processors connected on the ring network when AND-parallelism (resp. OR-parallelism) is considered. Load balancing is, therefore, carried out automatically with the shuffle and selection of subgoals.

Yet another problem is on the communication of pruning information from one processor to another processor. When the result on a common subgoal is found, it must be communicated to other processors to stop the processing of a subset of the eliminated solution trees and allow the heuristic values of related tasks to be updated. Similarly, when either a solution tree is found or the goal is proved to be FALSE, all processors should stop further processing. These pruning information are more complicated than the incumbent in the parallel branch-and-bound algorithm implemented in MANIP. In this case, a bus is necessary to broadcast to all processors a subgoal or clause found to be TRUE or FALSE. To minimize the information broadcast, the subgoal or clause should correspond to the nonterminal node closest to the root in the AND/OR tree. Other processors receiving this information must update all tasks in its Active List by reordering the tasks according to the new heuristic values computed and by eliminating tasks that cannot lead to a solution tree. The complete path from the root to each active task must be maintained to allow the pruning information to be incorporated. The overhead for the propagation of pruning information is extensive and may not be beneficial because the probability-to-cost ratios may be fallible. The no-wait policy is again applied here to continue the evaluation of tasks according to previously computed heuristic values. Information received on subgoals will be used to eliminate unnecessary work when the task is selected.

The last problem on the implementation of a heuristic search lies in the management of the large memory space required. In our study of MANIP, it was found that a direct implementation involving an ordered list of pointers to the subproblems results in poor locality of access, because the subproblems are not ordered by lower bounds in the secondary memory. A specially designed virtual memory that tailors its control strategies to the access behavior of the algorithm was found to be inflexible. The inadequacies of these approaches are due, again, to the strict adherence to the selection rule. A better solution is to use the no-wait policy to implement a modified heuristic search in each processor. In the modified heuristic search, the range of possible lower bounds (in this case, the range of probability-to-cost ratios) is partitioned into b disjoint regions (Figure 6). The subproblems in each region are maintained as a separate list. The top portion of each list resides in the main memory, and the rest resides in the secondary memory. Due to the high overhead of secondary-storage accesses, subproblems in a list are expanded in a depth-first manner. Only subproblems in the main memory are candidates for selection. The modified algorithm is identical to a depth-first search when one list is used, and is identical to a pure heuristic search when infinity lists are used. In general, as the number of lists increases, the number of subproblems expanded decreases and the overhead of the secondary-memory accesses increases. The number of lists should be chosen to maximize the overlap between computations and secondary-memory accesses. Experience on branch-and-bound algorithms showed that two to three lists are adequate.

5. CONCLUSIONS

In this paper, we have extended the architecture of MANIP for the parallel processing of logic programs. A logic program is assumed to be represented in the form of an AND/OR tree. The results that we have obtained can be summarized as follows.

(1) OR-parallelism and AND-parallelism have been unified into OR-parallelism. The objective of an OR-tree is to select a solution tree and to prove that the goal is TRUE; whereas in an AND-tree search, the objective is to find a set of subgoals to prove that the goal is FALSE. Both types of searches require only one of the correct descendents to be selected and can be considered as OR-tree searches. When the goal is likely to be TRUE, OR-parallelism should be used. In contrast, when the goal is likely to be FALSE, AND-parallelism should be used.

(2) Heuristic information using success probabilities and average overheads of evaluation have been defined to guide the search of logic programs. These information can be generated statically; however, they represent a prior information that do not take into account the query used, the dynamic pruning information obtained, and the values of variables being bound. They are useful to roughly differentiate between tasks that are likely to lead to solution trees and those that might not. Moreover, they define whether the goal is likely to be TRUE or FALSE.

(3) An optimal heuristic search strategy that minimizes the expected overhead of obtaining one solution tree is derived. The search is guided by the probability-to-cost ratios of subgoals.

(4) The architecture of MANIP, proposed earlier for the parallel evaluation of branch-and-bound algorithms, has been extended to implement a heuristic search of logic programs. Problems on selection and virtual-memory support have been considered. The strict adherence to the heuristic search is found to be unrewarding because the probability-to-cost ratios may be fallible, and the overheads of selecting tasks according to these ratios are high. These overheads include the update of the ratios when new pruning information is received, and the selection of tasks from other processors or the secondary memory according to these ratios.

REFERENCES

[Bar83] J. Barnett, "Optimal Searching from AND Nodes," *IJCAI*, pp. 786-788, 1983.

[CiI84] A. Ciepietewski and S. Haridi, "Execution of Bagof on the OR-parallel Token Machine," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 551-560, 1984.

[ClG83] K. Clark and S. Gregory, "PARLOG: A Parallel Logic Programming Language," Research Report DOC 83/5, Imperial College, March 1983.

[CoK83] J. Conery and D. Kibler, "AND Parallelism in Logic Programming," *IJCAI*, pp. 539-543, 1983.

[DeG84] D. DeGroot, "Restricted AND-Parallelism," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 471-478, 1984.

[FuN82] K. Furukawa, K. Nitta, and Y. Matsumoto, "Prolog Interpreter Based on Concurrent Programming," *Proc. of First Int'l Conf. Logic Programming*, 1982.

[Gar73] M. Garey, "Optimal Task Sequencing with Precedence Constraints," *Discrete Mathematics*, Vol. 4, pp. 37-56, 1973.

[HaA84] R. Hasegawa, M. Amamiya, "Parallel Execution of Logic Programs Based on Dataflow Concept," *Proc. of Int'l Conf. on Fifth Generation Computer Systems*, pp. 507-516, 1984.

[KKM83] S. Kasif, M. Kohli, and J. Minker, "PRISM: A parallel Inference System for Problem Solving," *Proc. IJCAI*, pp. 544-546, 1983.

[Kow79] R. Kowalski, *Logic For Problem Solving*, Elserier, North Holland, 1979.

[MaM73] A. Martelli and U. Montanari, "Additive AND/OR Graphs," *IJCAI*, pp. 1-11, 1973.

[MaM82] A, Martelli and U. Montanari, "An Efficient Unification Algorithm", *ACM Trans. on Prog. Lang. and Systems*, Vol. 4, pp. 258-282, 1982.

[Mot84] T.Moto-Oka, et al., "The Architecture of a Parallel Inference Engine--PIE," *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pp. 479-488, 1984.

[Nil80] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.

[PaW78] M.Paterson and M. Wegman, "Linear Unification", JCSS, Vol. 16, pp. 158-167, 1978.

[PaW79] E. Page and L. Wilson, *Introduction to Computational Combinatorics*, Cambridge University Press, 1979.

[Pea84] J. Pearl, *Heuristics*, Addison-Wesley, 1984.

[Sha83] E. Shapiro, *A Subset of Concurrent Prolog and its Interpreter*, ICOT Tech. Report TR-003, Feb. 1983, Tokyo, Japan.

[SiK75] H. A. Simon and J. Kadane, "Optimal Problem-solving Search: All-or-None Solutions," *Artificial Intelligence*, Vol. 6, pp. 235-246, 1975.

[StM84] S. J Stolfo and D. P. Miranker, "DADO: A Parallel Processor for Expert Systems," *Proc. Int'l Conf. on Parallel Processing*, pp. 74-82, 1984.

[WLY84] B. W. Wah, G.-J. Li, and C. F. Yu, "The Status of MANIP--A Multicomputer Architecture for solving Combinatorial Extremum-Search Problems," *Proc. 11'th Annual Int'l Symp. Comput. Architecture*, pp. 56-63, 1984.

[WLY85] B. W. Wah, G.-J. Li, and C. F. Yu, "Multiprocessing of Combinatorial Search Problems," to appear in *IEEE Computer*, June 1985.

[WaM84] B. W. Wah and E. Y. W. Ma, "MANIP--A Multicomputer Architecture for Solving Combinatorial Extremum Search Problems," *IEEE Trans. on Comput.*, Vol. C-33, No. 5, pp. 377-390, May 1984.

[YaN84] H. Yasuhara and K. Nitadori, "ORBIT: A Parallel Computing Model of Prolog," *New Generation Computing*, Vol. 2, pp. 277-288, 1984.