The Parallel Interpretation of Logic Programs in Distributed Architectures

P. CORSINI1, G. FROSINI*1 AND G. SPERANZA2
1 Istituto di Elettronica e Telecomunicazioni, Facoltà di Ingegneria, Università degli Studi di Pisa, Via Diotisalvi 2, 56100 Pisa, Italy
2 I.D.S. – Ingegneria Dei Sistemi S.p.A., Pisa

Multiple processor architectures can be used to speed up the execution of logic programs, as they allow the exploitation of appropriate forms of OR-AND parallelism. In this paper we refer to distributed systems in which nodes are connected by means of point-to-point links, and information exchanges between processes are accomplished by message passing alone. The number of nodes is supposed to be not too large, and the memory of each node not sufficient to store all the database. A parallel interpreter for logic programs is proposed in which the parallelism exploited is not the maximum one and consists of the classical OR parallelism paired with an appropriate form of AND parallelism ('backtracking parallelism'). The solutions are evaluated in the same order as a classical sequential depth-first interpreter.

Received February 1986, revised May 1987

1. INTRODUCTION

Programming in logic1, 2 consists of

- declaring certain facts about objects and their relationships;
- defining rules about objects and their relationships;
- asking goals about objects and their relationships.

Facts are clauses without a body, rules are clauses with a head and a body, and goals are clauses without a head. Facts and rules constitute the program or the database.

As an example, a simple database is the following (variables are names beginning with an upper-case letter):

(1) man(Edward).
(2) man(Albert).
(3) man(John).
(4) woman(Alice).
(5) woman(Mary).
(6) parents(Edward, Albert, Alice).
(7) parents(John, Albert, Alice).
(8) parents(Mary, Albert, Alice).
(9) brother(X,Y) :— man(X), parents(X,F,M), parents(Y,F,M).

The first 8 clauses are facts, and specify that (a) Edward, Albert and John are men, (b) Alice and Mary are women, and (c) Albert and Alice are the parents of Edward, John and Mary. The last clause is a rule, and specifies that a person X is the brother of a person Y if X is a man and the parents of X, say F and M, are also the parents of Y.

A possible goal statement is the following:

?- brother(X, Mary).

It asks if there is some person X who is the brother of Mary.

An interpreter of a logic program attempts to solve a goal statement by deriving a null clause throughout a series of inferences. More precisely, for each subgoal of the goal statement, the interpreter tries (1) to perform a match process (unification) between the subgoal and the head of a certain clause in the database, and (2) to replace the subgoal with the body (possibly void) of the unified clause, thereby obtaining a derived goal statement, and so on. Unification between a subgoal and clause head is possible if (1) they have the same predicate symbol and the same number of arguments and (2) proper instantiations of the variables to terms can be performed in order to make them syntactically identical. When an inference is performed, the instantiations resulting from the unification process are extended to all the subgoals of the actual goal statement. The bindings of the variables of the initial goal statement obtained as a result of the unification sequence performed to obtain a null clause, constitute a solution of the goal statement itself.

Referring to the previous database and goal a possible sequence of interpretation steps is the following:

- goal: brother(X, Mary)
- unification with the head of the 9th rule: Y = Mary
- new goal: man(X), parents(X,F,M), parents(Mary, F,M)
- unification of the first subgoal with the first fact: X = Edward
- new goal: parents(Edward, F,M), parents(Mary, F,M)
- unification of the first subgoal with the 6th fact: F = Albert, M = Alice
- new goal: parents(Mary, Albert, Alice)
- unification of the goal with the 8th fact
- new goal: null clause

Thus, the found solution is X = Edward.

In the general case, when there are many clauses which are unifiable with a given subgoal or when there are many subgoals, the interpretation mechanism produces a tree of derived goal statements. The leaves of the tree correspond to situations in which (1) the derived goal statement is the null clause (success leaf) or (2) the derived goal statement is not a null one, but no further unification is possible (fail leaf). So, for each success leaf a solution of the goal statement is found.

A typical sequential interpreter attempts to satisfy each subgoal in turn working from left to right. So a
reduced tree with respect to the general case is produced, and a depth-first search of this tree is performed. Moreover the interpreter backtracks every time a fail leaf is reached, or a success leaf is found but further solutions are required.

With reference to the above sample program and goal, the tree produced by the sequential interpreter is shown in Fig. 1.

```
brother(X,mary)

parent(X,F,M).
parent(mary,F,M).

parent(edward,F,M).
parent(mary,F,M).
parent(albert,F,M).
parent(mary,F,M).
parent(john,F,M).
parent(mary,F,M).

F = albert
M = alice

success
```

Figure 1. The tree produced by the sequential interpreter for the working example.

In the general interpretation mechanism, two kinds of parallelism can be exploited:

- OR parallelism: when a subgoal is unifiable with the head of several clauses, the distinct derived goal statements are independent and can be evaluated in parallel;
- AND parallelism: when a goal statements contains more than one subgoal, they can be considered as separate goal statements and can be solved with appropriate forms of parallelism that ensure the consistency of the instantiations of the variables shared by the subgoals.

The two previous forms of parallelism can be incorporated in the logic language, by providing explicit commands for their control. Two interesting proposals are concurrent PROLOG and PARLOG. The last proposal incorporate the control of both AND parallelism (mode declaration) and OR parallelism (guarded clauses).

Alternatively, the parallelism can be exploited at the interpretation level in a user transparent way, i.e. without adding any construct to the classical logic language. Such an approach has also been considered in and 7.

In the present work the last approach is considered, and a new parallel interpreter is described, that exploits the classical OR parallelism paired with a reduced form of AND parallelism (backtracking parallelism).

The main difference between the interpreter described in and that presented here concerns the AND parallelism. More precisely, Corner and Kibler propose an algorithm that considers the body of a clause and creates a data flow graph, in which a generator–consumer relationship among subgoals is established. Subgoals are then solved in the order specified by the graph. In our approach, subgoals are considered in the order they appear in a clause body, and a solution produced by a given subgoal $S(i)$ is used to solve the subgoal $S(i+1)$, while the subgoal $S(i)$ produces further solutions. The exploited parallelism is not the maximum one, and the parallel activities are conceived in such a way as to be independent, as far as possible.

The proposed interpreter fits the characteristics of a practical multiple processor architecture, that is (1) limited number of nodes, (2) limited memory size on each node, and (3) limited interconnection network bandwidth.

The interpreter is defined in a Pascal-like language, and an implementation using OCCAM & TRANSPUTERS is under development.

2. THE MULTIPLE PROCESSOR ARCHITECTURE

The aim of this section is not to describe the multiple processor architecture in any great detail. All we require is that the following properties be present:

(1) each node is a classical sequential computer element, with a processor, a private memory and communication channels;
(2) a particular node acts as an interface between the external world and the system;
(3) the interconnection network assures efficient message exchanges from a source node to a destination node on the basis of explicit node naming (no broadcasting is required).

The straightforward implementation of such an architecture provides a physical bidirectional communication link between each pair of nodes, as illustrated in Fig. 2 in the case of five nodes. An effective system that fits the structure being hypothesized can be implemented by using the INMOS Transputers as nodes. Each Transputer is able to provide four high-speed serial bidirectional links and a firmware directly handling the message exchanges.

As stated in the introduction, such a type of architecture becomes impractical for a large number of

![Figure 2. A multiple processor architecture for five nodes.](https://academic.oup.com/comjnl/article-abstract/32/1/29/341929/341929)
nodes. Moreover, it is also unrealistic to suppose that the memory size of each node will be sufficient for storing a whole copy of the database.

Therefore the following hypotheses have been put forward:

(1) the database is distributed (in a precompilation phase) between nodes so that all the clauses with the same predicate symbol in the head are allocated to a same node (such clauses constitute the procedure for that predicate symbol, and can be eventually replicated on different nodes);

(2) each node contains a copy of the whole allocation table that specifies the node (or the nodes) to which each procedure is allocated;

(3) parallel activities are modelled as concurrent processes, that communicate only by exchanging messages;

(4) communication messages are as simple as possible, and, if they are interprocessor messages, they never include the bodies of the clauses.

3. THE PROPOSED FORM OF PARALLELISM

A parallel interpreter can be modelled as a set of concurrent processes, communicating throughout the exchange of messages, and dynamically created to exploit the OR parallelism and the AND parallelism.

An OR process solves a single subgoal by attempting to reduce it to the null clause. It works on the procedure sharing the same predicate symbol with the subgoal, and for each unifiable clause, not corresponding to a fact, it creates a son AND process with the aim of solving the body of the clause itself. The OR parallelism is exploited when more than one son AND process is active at any time.

An AND process solves the body of a clause (or the initial goal statement). It creates a son OR process for solving each subgoal. The AND parallelism is exploited when more than one son OR process is active at any time. The complete AND parallelism arises if a process is independently created for every subgoal. This is impractical because the solutions of one subgoal often bind variables that are arguments in other subgoals. Moreover, and this is of the greatest practical importance, some subgoals (usually the rightmost ones) often fail if an attempt is made to solve them before proper variable instantiations are made.

In our approach two main guidelines have been resorted to:

(a) from an external point of view, the parallel interpreter is required to behave externally as a sequential interpreter, that is, it always produces the solutions in the same order, according to the depth-first principle;

(b) from an internal point of view, the number of active processes at every instant is required not to proliferate beyond reasonable limits, and the amount of information exchanged between nodes is required to be limited.

In order to respect the first guideline, when a process \( P \) activates a son process \( Q \), it provides a specific FIFO queue, in which \( Q \) must return the solutions found. So, the time ordering of the solutions found by \( Q \) is not lost.

In order to respect the second guideline, two different strategies have been adopted. First, since all the clauses constituting a procedure are statically allocated to a same node, when an OR process becomes active, it attempts to unify the head of all the clauses involved, and possibly activates a son AND process for each unifiable clause, in a sequential way. Second, the AND processes are structured in such a way as to exploit a guided form of pipeline parallelism (backtracking parallelism), as explained below. Let \( P\text{AND} \) be a process of the AND type. In order to solve the first subgoal (with given instantiations), \( P\text{AND} \) activates the first son OR process, say \( P\text{OR}_1 \), and waits for solutions from \( P\text{OR}_1 \) itself. Process \( P\text{OR}_1 \) evaluates all the solutions in an orderly fashion, sends them to \( P\text{AND} \) throughout the associated FIFO queue, and then terminates. As soon as \( P\text{AND} \) receives the first solution from \( P\text{OR}_1 \), it activates the second son OR process, say \( P\text{OR}_2 \), in order to solve the second subgoal (with instantiations updated according to the solution returned from \( P\text{OR}_1 \)). Process \( P\text{OR}_2 \) behaves in exactly the same way as \( P\text{OR}_1 \). Process \( P\text{AND} \) waits for a solution from \( P\text{OR}_2 \) in order to activate \( P\text{OR}_3 \), and so on. Solutions returned to \( P\text{AND} \) from \( P\text{OR}_2 \) (the OR process activated to solve the last subgoal) are sent by \( P\text{AND} \) to its father OR process. The last solution (possibly the only solution) returned to \( P\text{AND} \) by each son OR process is a fail. So, when \( P\text{AND} \) finds a fail from \( P\text{OR}_n \), it takes the second solution produced by \( P\text{OR}_{n-1} \) and activates \( P\text{OR}_n \) for the second time and so on. The \( P\text{AND} \) process behaves in the same way when it finds a fail as the last solution produced by \( P\text{OR}_{n-1} \), that is, it takes the second solution produced by \( P\text{OR}_{n-2} \) and activate \( P\text{OR}_{n-1} \) for the second time, and so on. Finally, when \( P\text{AND} \) finds a fail as the last solution produced by \( P\text{OR}_1 \), it returns a fail to its OR father process, and then terminates. The parallelism is exploited since an OR process \( P\text{OR} \) remains active and produces further solutions while a previously produced solution is elaborated by \( P\text{OR}_{1+1}, P\text{OR}_{1+2}, \ldots, P\text{OR} \), according to the guide pipeline form explained above.

It should be noted that the process interaction model adopted involves dynamic creation of processes. Moreover, communications between fathers and sons alone are required, and this simplifies interpreter implementation, especially if available concurrent languages are used.

Let us consider the program and the goal given in Section 1. The following processes are created:

- \( P\text{AND1} \) for solving the initial goal;
- \( P\text{OR1} \) for solving the unique subgoal: \( \text{brother}(X,Y) \);
- \( P\text{AND2} \) for solving the new goal: \( \text{man}(X), \text{parents}(X,F,M), \text{parents}(Y,F,M) \); 
- \( P\text{OR2} \) for solving the first subgoal: \( \text{man}(X) \);
- \( P\text{OR2} \) for solving the second subgoal: \( \text{parents}(X,F,M) \);
- \( P\text{OR2} \) for solving the third subgoal: \( \text{parents}(Y,F,M) \).

Figure 3 shows the genealogical tree of the above processes, together with (a) the instantiations sent by a father to each son and (b) the returned solutions. Note
that processes $\text{POR}_{2}$ and $\text{POR}_{3}$ are activated three times and twice, respectively.

Figure 4 shows a possible time evolution of the described processes. The behaviour of the most significant process, i.e. of $\text{PAND2}$, is the following:

1. it activates $\text{POR}_{1}$ with instantiation $y_{1}$, and waits for a reply message from $\text{POR}_{2}$;
2. when it receives from $\text{POR}_{2}$ the solution $x_{1}$, it activates $\text{POR}_{3}$ with instantiation $x_{1},y_{1}$, and waits for a reply message from $\text{POR}_{2}$;
3. when it receives from $\text{POR}_{2}$ the solution $f_{1},m_{1}$, it activates $\text{POR}_{2}$ with instantiation $x_{1},y_{1},f_{1},m_{1}$, and waits for a reply message from $\text{POR}_{2}$;
4. when it receives from $\text{POR}_{2}$ the solution ‘success’, it sends to its father the solution $x_{1},f_{1},m_{1}$, and waits for a further reply message from $\text{POR}_{2}$;
5. when it receives from $\text{POR}_{2}$ the solution ‘fail’, it waits for a further reply message from $\text{POR}_{1}$;
6. when it receives from $\text{POR}_{2}$ the solution ‘fail’, it waits for a further reply message from $\text{POR}_{1}$;
7. when it receives from $\text{POR}_{2}$ the solution $x_{2}$, it activates $\text{POR}_{2}$ and waits for a reply message from $\text{POR}_{2}$;
8. when it receives from $\text{POR}_{2}$ the solution ‘fail’, it takes a further solution $x_{3}$ from $\text{POR}_{2}$ (already arrived), activates $\text{POR}_{2}$ with instantiations $x_{3},y_{1}$ and waits for a reply message from $\text{POR}_{2}$;
9. when it receives from $\text{POR}_{2}$ the solution $f_{2},m_{2}$, it activates $\text{POR}_{3}$, with instantiation $x_{3},y_{1},f_{2},m_{2}$, and waits for a reply message from $\text{POR}_{2}$;
10. when it receives from $\text{POR}_{2}$ the solution ‘success’, it sends to its father the solution $x_{3},f_{2},m_{2}$, and waits for a further reply message from $\text{POR}_{2}$;
11. when it receives from $\text{POR}_{2}$ the solution ‘fail’, it waits for a further reply message from $\text{POR}_{2}$;
12. when it receives from $\text{POR}_{2}$ the solution ‘fail’, it takes a further solution from $\text{POR}_{2}$ (already arrived); since such solution is a ‘fail’, it sends to its father the solution ‘fail’ and terminates.

It should be noted that the AND parallelism is exploited in the time intervals in which almost a pair of processes among $\text{POR}_{2}, \text{POR}_{3}$, $\text{POR}_{3}$ is active. The OR parallelism is not exploited since in the example there are not two or more rules with the same head.

4. PROCESS STRUCTURE AND INTERACTION

In order to simplify the handling of messages between processes, let us suppose that each node is supplied with a manager whose aim is to capture all the messages arriving (i) throughout an internal channel, from elaborative processes (AND and OR processes) running on the same node and (ii) throughout external channels, from elaborative processes running on other nodes.

The incoming messages are typed, and three different types are provided:

1. activation message, which requires the creation of an elaborative process;
2. reply message, which contains a result returned from an elaborative process;
3. abort message, which requires the killing of an elaborative process.

An OR process requires the activation of a son AND process that will always be run on the same processor. Thus the activation message is sent to the manager via the internal channel. An AND process requires the activation of a son OR process that will be run either on the same processor or on a different one, depending on which node the pertinent procedure has been statically allocated to. Thus, by consulting the allocation table replicated in every node, the AND process identifies the channel through which the activation message is to be sent.

In sending an activation message, each elaborative process also identifies the channel through which the process will return the reply message and inserts the channel identifier into the activation message itself. Moreover, an elaborative process inserts into the activation message the identifier of the queue in which it wants the reply messages returned by the activated son to
be put. Such an identifier is echoed in the reply messages and properly used by the manager that captures the messages themselves. The formats of an activation message and of a reply message are given in Fig. 5.

An abort message occurs when no further solutions are required (for example, owing to a console command or to a cut subgoal detection). The handling of an abort message by a process is quite simple, in the sense that (i) if the process has no son, it replies by returning a fail to the father and terminates, and (ii) if the process has activated a number of sons, then it sends them an abort message in turn, waits for a fail reply message from each son, returns a fail to its father and terminates.

It should be noted that an abort message sent to an autonomously terminated son does not produce a deadlock condition, since, in any case, each elaborative process sends a fail reply message before terminating.

For sake of simplicity, the handling of abort messages will not be discussed in further detail below.

5. PROGRAM SKELETON OF ELABORATIVE PROCESSES

Table I details (in a Pascal-like self-explaining notation) the way an OR process works. Two aspects must be emphasized. First, the OR process does not activate a son AND process when the unifying clause is a fact: in this case, it simply sends the manager a reply message for itself (via the internal channel), containing the solution found. In its turn, the message is put by the manager into the pertinent queue of the OR process. It should be noted that the OR process does not immediately send the father the reply message since all the solutions should be sent back in an orderly fashion, according to the depth-first principle. This is done by the second 'for statement' in the description.

Second, the OR process could be split into two parallel tasks, one dedicated both to performing the unification algorithm and to generating activation messages, and the
Table 1

<table>
<thead>
<tr>
<th>process OR</th>
<th>begin sons := 0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i := 1 to N do (* N is the number of clauses involved *)</td>
<td>begin attempt_to_unify_clause[i];</td>
</tr>
<tr>
<td>if unifiable then</td>
<td>begin sons := sons + 1; create_queue[sons];</td>
</tr>
<tr>
<td>if (clause[i] is a_fact) then</td>
<td>begin prepare_a_reply_message_for_itself; send_message;</td>
</tr>
<tr>
<td></td>
<td>prepare_a_fail_reply_message_for_itself; send_message</td>
</tr>
<tr>
<td>else</td>
<td>begin prepare_an_AND_activation_message; send_message</td>
</tr>
<tr>
<td>end</td>
<td>end;</td>
</tr>
<tr>
<td>for j := 1 to sons do</td>
<td>begin wait_for_a_reply_message_on_queue[j];</td>
</tr>
<tr>
<td>while (solution &lt; &gt; fail) do</td>
<td>begin prepare_a_reply_message_for_its_father; send_message;</td>
</tr>
<tr>
<td></td>
<td>wait_for_a_reply_message_on_queue[j];</td>
</tr>
<tr>
<td>end</td>
<td>prepare_a_fail_reply_message_for_its_father; send_message</td>
</tr>
<tr>
<td>end;</td>
<td></td>
</tr>
</tbody>
</table>

other dedicated to waiting for solutions from sons and to return solutions to the father. This splitting is not done merely for the sake of simplicity.

A Pascal-like description of an AND process is given in Table 2. The aspects to be emphasized are those concerning the 'update instantiations' and the 'recover instantiations' operations. Such operations can be carried out by utilizing a stack (in which the instantiation status received in the activation message is initially stored) whose top stores the actual instantiation status at every instant. Each time the AND process examines a solution that is different from fail coming from a son OR process other than the Nth one, it updates the instantiation status according to both the actual status and to the solution received, and pushes the new status on the stack. Such a new instantiation status is also utilized in order to prepare an activation message for the next son OR process. If the ANA process examines a solution that is different from fail coming from the Nth son process, a new instantiation status is constructed and inserted in the reply message for the father, but no store operation is performed on the stack. Instead, when the AND process finds a fail message coming from a son other than the first, it pops from the stack and discards such a failed instantiation status and considers the instantiation status that is now contained on the top of the stack, as being the actual instantiation status.

6. BEHAVIOUR OF THE INTERFACE NODE

In the hypothesized architecture, as seen in Section 2, a node acts as the interface between the external world and the system. Besides providing a standard interactive environment for program development, the interface node will perform the precompilation phase needed to allocate procedures to nodes. Moreover, the node will accept the initial goal statement and start the interpretation process. In order to carry out this latter task and return the solutions to the console, an interface process is provided. It creates an elaborative AND

Table 2

<table>
<thead>
<tr>
<th>process AND</th>
<th>begin i := 1;</th>
</tr>
</thead>
<tbody>
<tr>
<td>examine_subgoal[i]; create_queue[i];</td>
<td>prepare_an_OR_activation_message; send_message;</td>
</tr>
<tr>
<td>wait_for_a_reply_message_on_queue[i]; repeat</td>
<td>while (solution &lt; &gt; fail) do</td>
</tr>
<tr>
<td>begin if i &lt; N then (* N is the number of subgoals *)</td>
<td>begin i := i + 1;</td>
</tr>
<tr>
<td></td>
<td>update_instantiations;</td>
</tr>
<tr>
<td></td>
<td>examine_subgoal[i];</td>
</tr>
<tr>
<td></td>
<td>create_queue[i];</td>
</tr>
<tr>
<td></td>
<td>prepare_an_OR_activation_message; send_message</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>else</td>
<td>begin prepare_a_reply_message; (* for the father *)</td>
</tr>
<tr>
<td></td>
<td>send_message</td>
</tr>
<tr>
<td>end</td>
<td>end;</td>
</tr>
<tr>
<td>while ((solution = fail) and i &gt; 1) do</td>
<td>begin i := i - 1;</td>
</tr>
<tr>
<td></td>
<td>recover_instantiations;</td>
</tr>
<tr>
<td></td>
<td>wait_for_a_reply_message_on_queue[i]</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>until ((solution = fail) and i = 1);</td>
<td>prepare_a_fail_reply_message; (* for the father *)</td>
</tr>
<tr>
<td></td>
<td>send_message</td>
</tr>
</tbody>
</table>

Table 3

<table>
<thead>
<tr>
<th>interface process</th>
<th>begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_queue;</td>
<td>prepare_an_AND_activation_message; send_message;</td>
</tr>
<tr>
<td>repeat</td>
<td>wait_for_a_reply_message_on_queue;</td>
</tr>
<tr>
<td>write_solution_on_the_console;</td>
<td>read_answer_from_the_console</td>
</tr>
<tr>
<td>until ((solution = fail) or (answer &lt; &gt; ''));</td>
<td>if (solution &lt; &gt; fail) then</td>
</tr>
<tr>
<td>begin prepare_an_abort_message_for_itsSon; send_message;</td>
<td>repeat wait_for_a_reply_message_on_queue</td>
</tr>
<tr>
<td>until (solution = fail)</td>
<td>end</td>
</tr>
<tr>
<td>end;</td>
<td></td>
</tr>
</tbody>
</table>
process for solving the goal statement and waits for solutions (coming throughout a proper queue), so behaving like a singular OR process. The interface process sends also single solutions to the console while the user requires them by entering semicolons, and sends an abort message to the created AND process when no more solutions are required.

A Pascal-like description of such process is given in Table 3.

7. CONCLUSIONS

In this work a parallel AND-OR interpreter for logic programs is proposed, and a detailed description given of the OR process and of the AND process. Externally the interpreter behaves as a sequential one, in the sense that the solutions are always produced in the same order, according to the depth-first principle.

The interpreter is intended to run on distributed architectures, made up of several nodes interconnected by means of point-to-point links. So the proposed algorithms fit the architectural requirements, that is (i) a limited number of nodes, (ii) a limited memory size on each node, and (iii) a limited interconnection network bandwidth. As a consequence of such requirements, the whole data base is distributed between nodes, the maximum parallelism is not exploited (mainly the AND parallelism) and the number and the length of the communication messages have been limited.

An architecture with the hypothesised features is under development by using the INMOS Transputers as nodes. The proposed parallel interpreter is now being written in OCCAM.8

REFERENCES