IMPLEMENTATION OF THE AND/OR PROCESS MODEL

by

NITIN MORE

A THESIS

Presented to the Department of Computer
and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 1986

Approved: _____
Dr. John S. Conery

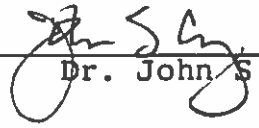
An Abstract of the Thesis of

Nitin More for the degree of Master of Science

in the Department of Computer and Information Science

to be taken June 1986

Title: IMPLEMENTATION OF THE AND/OR PROCESS MODEL

Approved: 

Dr. John S. Conery

This thesis describes an implementation of Conery's AND/OR Process Model for logic programs. The system runs on either a single processor or a network of loosely coupled UNIX systems. A goal statement is solved by creating AND-processes and OR-processes according to the rules outlined by Conery. In the multi-processor implementation, a copy of the interpreter runs on each processor where each interpreter has a copy of the source program. The processes are distributed among the processors.

VITA

NAME OF AUTHOR: Nitin More

PLACE OF BIRTH: Bombay, India

DATE OF BIRTH: March 16, 1962

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Oregon, USA
Indian Institute of Technology, Kanpur, India

DEGREES AWARDED:

Master of Science, 1986, University of Oregon, Oregon
Bachelor of Technology, 1984, Indian Institute of Technology, Kanpur, India

AREAS OF SPECIAL INTEREST:

Software Engineering, Artificial Intelligence

PROFESSIONAL EXPERIENCE:

Teaching Assistant, Department of Computer and Information Science, University of Oregon, Oregon, 1984-86

Software Consultant to Godrej and Boyce Manufacturing Company, Bombay, India, 1984

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to Professor John S. Conery, my advisor. His encouragement and guidance throughout my study have been invaluable. Special thanks are also due to Professor Stephen Fickas for his support and friendship. Finally, the faculty, staff and colleagues in the CIS department have all contributed in making my stay here at this University an educative and pleasant experience.

To my parents

Nirmala and Tukaram More

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The aim of Japan's Fifth Generation Computer System project is to build a very fast multi-processor parallel system based on non-Von Neumann architecture. An extended form of Prolog, a logic programming language, has been chosen as the kernel language for this project. Researchers have discovered many kinds of potential parallelism possible in implementing logic programs. Logic programming has been used effectively in diversed areas as plane geometry, learning, generalization, planning, symbolic calculus, natural language understanding, speech understanding, chess, query optimization, and robotics [6]. This thesis report describes an implementation of a parallel interpreter for logic programs. The interpreter is based on the AND/OR Process Model described by Conery [4,6]. The interpreter, written in C, runs on a loosely coupled network of UNIX systems and can be easily extended to run on a tightly coupled multi-processor system.

Background

Logic programming is a programming methodology based on symbolic logic, involving the use of Horn clauses. Execution of a logic program, comprising a set of Horn clauses, means finding a solution to a goal predicate supplied by the user. The solution of the goal is a set of substitutions for variables occurring in the goal predicate, obtained by solving that goal. Failure is reported when no such substitution exists.

Logic programs offer many opportunities for parallelism. One theme in parallel inference is to develop methods for extracting parallelism from standard Prolog in order to achieve faster execution on a multi-processor system. A second theme in parallel inference is to define a new logic language appropriate for programming concurrency. Epilog, an extended version of Prolog, is an example of a language developed to exploit parallelism from logic programs. A third approach, which we are taking, is to define a new logic language, not based on Prolog, but derived from pure logic. In this approach of language oriented architecture, the aim is to define machines for functional programming languages. This is also known as language first philosophy.

Types of Parallelism

Many practical logic programs have inherent large-scale parallelism. But, there are important examples of logic programs which do not have such parallelism, e.g. simple list concatenation. Among various kinds of parallelism possible in logic programming, the important ones are OR-parallelism, AND-parallelism, Search-parallelism and Stream-parallelism [6]. In OR-parallelism, several clauses matching a goal are processed concurrently. In AND-parallelism literals in the body of a clause are solved simultaneously. Search-parallelism is a technique of partitioning the data base of clauses in disjoint sets to enable efficient parallel searching. In Stream-parallelism, partial results from solution of one literal are passed to the process solving the next literal.

Issues Involved in Implementing Logic Programs on a Parallel Machine

The order in which solutions are obtained by a parallel machine need not be the same as that obtained on a sequential machine, and in almost all cases, this order is not important unless one is executing Prolog instead of a logic program. Prolog programmers rely on the order of their clauses to produce certain effects, and this will not be possible in our parallel logic system.

As the number of processors increases, the execution

time of a program should decrease. But the increase in efficiency is not directly proportional to the increase in the number of processors, because as the number of processors increase, so does the communication cost among processors. Sometimes communication cost may outweigh the benefits obtained by having many processors, hence communication should be made as efficient as possible. Also the increase in the number of processors may result in an increase in the complexity of resource sharing.

Assignment of work to processors can be done in two ways. Either work can be assigned to processors by another processor, such as a central control processor, or idle processors can be made responsible for selecting work for themselves. The choice of an appropriate scheme depends upon various factors. These factors include the distribution of source program among the processors, the number of processors et cetera. The former mechanism is a potential bottleneck when there are a very large number of processors.

How program statements (Horn clauses) should be distributed among the processors is also an important issue. Three possibilities are: a single copy of the program can be kept in a central global memory; each processor can have a copy of the program in its respective local memory; or the program can be divided into several chunks and distributed among local memories. The first scheme can lead to a

bottleneck at global memory, the second is only possible when programs are relatively small and each processor has a lot of local memory to spare. The last scheme looks very promising in terms of efficiency. Nakagawa [9] and Warren et al. [10] employ the last scheme. Nakagawa uses the concept of assertion set in which clauses with the same head are stored with same processor [9]. Tick and Warren use a distribution scheme in which clauses with the same head are kept at different processors [10]. In our system we use the second scheme, where all processors have a copy of the program in their local memories.

Two processors may need to share some data at runtime. An example is a binding environment shared during unification by many parallel OR-processes of the AND/OR Process Model [4]. The information can either be duplicated or shared by the processors. To avoid side effects, copying may be necessary. But, whenever there is no possibility of side effects, sharing of data structures is efficient, since the data structures created by logic programs can be very large.

Implementation of a true OR-parallel model, such as the system of Ciepielewski and Haridi [3], may lead to combinatorial explosion of space. This happens because many processes are invoked to solve a single literal and each invocation involves copying the existing stack. Much of the work in parallel logic programmming has been on techniques

for avoiding this. Ciepielewski and Haridi use a technique for pruning the search tree [3]. Conery talks about the use of secondary memory to store the status of blocked processes (processes at the top of the search tree), and a mechanism for inhibiting parallelism, so that fewer processes are created [4]. Other techniques are described by Warren [11], Kumon et al. [8], and Borgwardt [1]. In this implementation a new method developed by Conery, called closed environments, is used for information sharing [4].

Thesis Layout

The thesis report is organized as follows: Chapter II describes Conery's AND/OR process model, showing how logic programs can be interpreted by sets of asynchronous and independent processes instead of by one large centralized search algorithm. Implementation of Conery's model on a single processor and a multi-processor is the topic of Chapter III, which includes a detail description of data structures used. Chapter IV lists the experiments run on this system to test and measure the efficiency of the implementation. Directions for future work are discussed in the final chapter.

CHAPTER II

CONERY'S AND/OR PROCESS MODEL

Overview

In his doctoral thesis, Conery described the AND/OR Process Model, an abstract model for parallel interpretation of logic programs [4]. His model exploits both AND-parallelism and OR-parallelism. The interpretation of a goal statement is carried out by two kinds of processes, called AND-processes and OR-processes. An AND-process is created to solve a goal statement, which is a conjunction of one or more literals. The AND-process derives its name from the fact that all of its literals must be proved to be true for the goal statement to be true. An OR-process is created to solve a single literal. The number of ways in which it can solve this literal is the same as the number of clauses in the data base whose head clause can be unified with the literal. Any of these ways may lead to a number of potential solutions.

During execution, an AND-process may create one or more OR-processes to solve literals in its goal statement. Similarly, an OR-process may create one or more

AND-processes to solve the clause body of the clauses whose head unifies with the literal. In this way, a tree of processes is created with AND-processes and OR-processes on the alternate levels. A process can communicate with either its parent or its child process, but not (directly) with any other process.

The processes communicate by sending messages to each other. When a message is processed, the receiving process updates its internal state in an atomic operation, i.e., it doesn't process new messages until it completes processing the old one. This state transformation may result in the generation of one or more messages and creation of new processes.

The execution of a user's goal begins by creating an AND-process to solve that goal. It terminates successfully when the AND-process receives a "success" message from its rightmost child (i.e., from the OR-process created to solve the last literal). The execution terminates unsuccessfully if the AND-process receives a "fail" message from its first child process.

Messages

A message represents an action to be carried out by the receiving process. Messages sent to descendant processes are "start," "redo" or "cancel," and messages sent to the parent are "success" or "fail." A "start" message is sent to

a newly created process by its parent. A "success" message is sent to a process by its child when the child has been successful in solving its part of the problem. For an AND-process, this means all literals in the body have been solved. This message contains a solution, which is represented by variable bindings. A "fail" message is sent to a process by its child when the child fails to solve its goal literal(s). In some situations, after a process receives an answer from one of its child processes, it will need a different solution, so it sends a "redo" message to that child. Finally when a process doesn't need the service of a child process any more, it sends a "cancel" message to that child.

Parallel OR-process

An OR-process is created by an AND-process to solve a literal from the AND-process's goal list. The OR-process is activated when it receives a "start" message from its parent process. The OR-process will solve its literal in one of two ways. If the literal unifies with an assertion, the literal is solved immediately, and an answer can be constructed for the parent and sent back via a "success" message. If the literal unifies with the head of an implication, an AND-process is created to solve the body of that clause.

The first answer generated by the OR-process is sent to

its parent and subsequent answers are saved in a list of answers. One of these answers is sent to the parent when the parent requests another solution by sending a "redo" message. When no more solutions can be generated and the parent sends a "redo" message then the OR-process sends a "fail" message to its parent. The OR-process terminates after sending a "fail" message.

An OR-process can be either in waiting mode or in gathering mode. It is in waiting mode if its parent is waiting for an answer. It is in gathering mode when the parent is busy, using the answer sent previously. The state of an OR-process is represented by a data structure with the following fields:

1. L, the literal to be solved by the OR process.
2. WL, the waiting list, a list of answers not yet sent to the parent.
3. SL, the list of answers that have been sent.
4. DL, a list of ids of descendant processes.

State Transitions

The following sections will describe in detail how a parallel OR-process reacts to any message it receives, depending on its current state (waiting or gathering) and the type of message.

Start Message

When an OR-process receives a "start" message, it does a data base look-up to find all implications and assertions whose head matches with the goal literal, L. Two literals match when their functors are same and have same arity. The OR-process then checks if the head of the clauses can be unified with L. For all the implications for which the unification succeeds, the OR-process creates an AND-process to solve the body of the implication. The ids of AND-processes are added to the DL list. For all the assertions for which the unification succeeds, the OR-process constructs an answer to send to the parent process. All these answers are kept in the WL list of the OR-process. If the unification succeeds with at least one assertion then the WL list will contain at least one solution. If the WL is not empty, a solution is removed from the WL list and is sent to the parent via a "success" message and the OR-process goes into gathering mode. A copy of the answer is also saved in the SL list. If the unification succeeds with at least one implication but with no assertion, then WL is empty and the OR-process goes into waiting mode. If no clauses can be found whose head can be unified with L then the OR-process has failed to find a solution for the literal. Hence it sends a "fail" message to its parent and terminates.

Success Message

Whenever a parallel OR-process receives a "success" message from one of its descendant processes then it sends a "redo" message to the descendant process. The descendant immediately starts working on constructing another solution. This saves time if backtracking is required because the descendant may be able to keep an alternate solution ready before the need for backtracking arises.

The other actions taken by the OR-process after it receives a "success" message depend on whether it is in gathering or waiting mode. If the OR-process is in waiting mode then it makes a "success" message for its parent and sends it. A copy of the solution is saved in the SL list, and the process goes into gathering mode. If the OR-process is in gathering mode then it constructs an answer and saves it in the WL list. The answer is not immediately sent to the parent because the parent is busy. The answer will be sent later if the parent requests for another solution. The SL list is unchanged and the process remains in gathering mode.

Fail Message

When an OR-process receives a "fail" message from a descendant, it removes the id of the descendant process from its DL list. Further action depends on the number of active

descendants and the state of the OR-process.

If the OR-process is in waiting mode and if there are descendants still working, then no further action is required, and the mode of the process remains unchanged. But, if DL is now empty, the OR-process sends a "fail" message to the parent, since there is no way to construct another answer and the parent is waiting for an answer.

If the OR-process is in gathering mode then no further action is required. The process remains in gathering mode. Since the parent is currently busy, a "fail" message is not sent, even if the DL list is now empty. A "fail" message will be sent later when the parent sends a "redo" message.

Redo Message

A "redo" message received by an OR-process in waiting mode is an erroneous situation because waiting mode signifies the parent is waiting for an answer; when the parent is waiting, it shouldn't be sending any messages.

A gathering OR-process handles a "redo" from its parent in one of three ways, depending on the states of WL and DL lists.

If the WL list is not empty then the OR-process selects an answer from it and sends it to the parent via a "success" message. A copy of the answer is added to the SL list and the OR-process remains in gathering mode.

If the WL list is empty and if the DL list is also

empty, then a "fail" message is sent to the parent because there is no way to make another answer. After sending a "fail" message, the OR-process terminates.

If the WL list is empty and if the DL list is not empty then the process goes into waiting mode. This means that some descendants are still working to produce an answer. The SL and WL lists are not changed.

Sequential AND-process

An AND-process is created to solve a conjunction of literals. The literals are solved by creating OR-processes for each one. AND-parallelism can be achieved by creating OR-processes all at once for each literal. But, there are several issues involved in extracting AND-parallelism. It is not so easy to achieve this as OR-parallelism. In this thesis, we will be concerned only with sequential AND-processes.

An AND-process creates an OR-process to solve the first literal in the goal statement. If the OR-process succeeds, the bindings in the answer sent by the OR-process are applied to the remaining literals. An OR-process is then created to solve the next literal in the goal statement. If all the literals are solved in this manner then a "success" message is sent to the parent along with the appropriate variable bindings.

If an OR-process fails to produce an answer and sends a

"fail" message, then a "redo" message is sent to the OR-process which was created to solve the previous literal in the goal list. If there is no such previous literal, i.e., if the "fail" message is received from the first OR-process, then the AND-process sends a "fail" message to its parent and terminates.

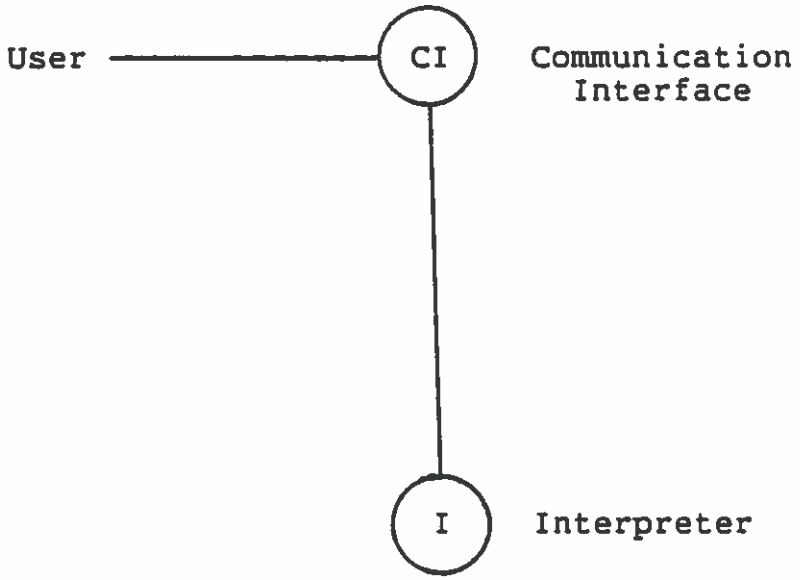
CHAPTER III

IMPLEMENTATION

Overview

This chapter describes the implementation of the AND/OR Process Model. The interpreter is written in C. The first version described executes on a UNIX system. It maintains a data base of clauses, two process pools -- one each for OR-processes and AND-processes -- a message queue, stack frames for clause environments, and other runtime data structures. The clause data base and process pools are organized as hash tables. The message queue is a simple First In First Out queue. By providing a suitable implementation of message passing, and a method for distributing new processes to other copies of the interpreters, this system will run on a network of UNIX systems. These extensions are discussed at the end of the chapter.

Figure 1 shows a pictorial view of the implementation on a single processor. CI is a Communication Interface, which acts as an interface between the user and the interpreter, I. At any given time, either I or CI is in



The Communication Interface (CI) acts as an interface between the user and the interpreter (I). CI and I are separate modules; they communicate with each other synchronously.

FIGURE 1. Configuration of the system on a single processor.

reading mode and the other is in writing mode. Initially CI is in writing mode. The user can type in clauses or load a disk file containing clauses. He can also query by typing in a goal statement. CI parses each clause and sends it to the interpreter via messages. The interpreter stays in the read mode until it receives a goal statement. After sending a goal statement, CI goes into reading mode, and waits for an answer from the interpreter. The interpreter stores all the clauses in an internal data structure. It goes into write mode when it receives a goal statement. It solves the goal and sends the result back to CI. CI then reads the result from the interpreter and reports it to the user. The user may type in more clauses or more goal statements.

When the interpreter receives a goal statement from CI, it starts solving it by creating AND-processes and OR-processes according to the rules outlined in the previous chapter. The main issues involved in the implementation of this interpreter are scheme for storing clauses, choice of the unification algorithm, handling clause environments, data structures for processes, managing process pools, inter-process communication, handling variable bindings and inter-processor communication. The unification algorithm described by Conery [5] is used in this implementation and other issues are described below.

Organization of Clause Data Base

A structure called a "cell" is used to represent a term, a variable or a constant in the clause. A cell has three fields to specify type and other information. All cells are stored in an array called Cell-memory. A clause is parsed by CI into a sequence of cells representing the clause in prefix form. Given this sequence of cells, a structure of type Clause-node is constructed, which arranges the available information in an organized manner. An example will help to make these data structures clear (Figure 2).

Figure 2(a) shows a program clause. Figure 2(b) shows the output of the parser, the prefix form of the clause. Figure 2(c) shows cells used in storing this clause in the Cell-memory. The first two fields of cells shown in Figure 2(c) are "tag" and "name." The value of the "tag" field specifies the type of the cell; it can be one of "sfun" (functor), "var" (variable) and "atom" (constant). "Name" is actually a pointer to a symbol table entry for the string shown in the Figure. Depending upon the value of the tag field, the third field is interpreted as arity of the functor if the tag is "sfun", or variable number in the clause when the tag is "var"; the third field is ignored when the value of "tag" field is "atom." Notice that "A" and "B" are the first and second variables in the clause

a

```
p(A,B) :- r(A) & s(B,e).

Note: "A" and "B" are
      variables, "e" is
      a constant, and ":-",
      "&","p","r", and
      "s" are functors.
```

b

```
:-(p(A,B),&(r(A),s(B,e)))
```

c

```
100: <sfun, ":-", 2>
101: <sfun, "p",  2>
102: <var,  "A",  0>
103: <var,  "B",  1>
104: <sfun, "&",  2>
105: <sfun, "r",  1>
106: <var,  "A",  0>
107: <sfun, "s",  2>
108: <var,  "B",  1>
109: <atom, "e",  0>

Note:100-109 are indexes
     in the Cell-memory
```

d

```
functor:      100

no-of-vars:     2

body:
        0:   101
        1:   105
        2:   107
        3:   nil
        :     :
        :     :
      max:   nil


next-clause: nil
```

Note:100, 101, 105 and 107 are indexes in the Cell-memory

FIGURE 2. Clause representations: source program clause (a); clause in the prefix form (b); cells in Cell-memory (c). Clause-node structure (d);

respectively; hence the value in the third field of cells at locations 102 and 106, representing "A," is 0, and that for cells at locations 103 and 108, representing "B," is 1. All the cells representing a single clause are stored in continuous locations in the Cell-memory.

Figure 2(d) shows the contents of Clause-node structure after this clause is loaded. The Clause-node structure has four fields. The "functor" field points to the first cell in the clause; in the above example, the value of functor is 100, meaning the clause is stored starting at location 100 in the Cell-memory. The second field, "no-of-vars," is used to store the number of variables in the clause. The third field, an array called "body," is a very important part of the Clause-node. This array is used by an AND-process in solving conjunctions of literals. Index 0 in this array points to the head of the clause. Subsequent indices point to literals in the clause body in the order in which they appear in the clause body. In the above example, index 1 points to the cell representing literal r(A) and index 2 points to cell representing literal s(B,e). This organization simplifies the work of AND-processes. Access to the cells of the next and previous literals is simplified. The fourth field in Clause-node, "next-clause," points to the Clause-node structure representing a clause with the same head functor and arity.

Clause-nodes are stored in a hash table. The functor

of the clause and its arity serve as the primary and the secondary keys respectively. Each entry in the hash table consists of a "functor," an "arity" and a "ptr-to-clause" field. Clause-nodes with the same functor and arity are linked together by the "next-clause" field in the clause-node structure.

Data Structure for Processes

In our implementation, an OR-process is represented as a data structure with the following fields:

1. self-id : id of the OR-process.
2. parent-id : id of the parent AND-process.
3. L : pointer in cell-memory to the current literal to be solved by this process.
4. WL : list of answers to be sent.
5. SL : list of answers already sent.
6. DL : list of ids of descendant AND-processes.
7. state : operating mode; "gathering" or "waiting."
8. next-OR-process: pointer to next OR-process. This field is used in garbage collection and in organizing the process pool.

An AND-process has the following fields:

1. self-id : id of the AND-process.
2. parent-id : id of the parent OR-process.
3. SG : pointer to Clause-node representing the goal to be solved.
4. CL : index in array "body" of Clause-node of the current literal being solved.
5. clause-envn : pointer to environment for variable bindings.
6. next-AND-process: pointer to next AND-process. This field is used in garbage collection and in organizing the process pool.

The id of a process is a single integer.

Process Pools

All active processes are kept in process pools. AND-processes and OR-processes are kept in different pools, each organized as a hash table. The process-id is used to determine the index in the hash table. All the processes with the same hash function value are linked together using the next-process pointer in the process structures. To access a process, the hash function is applied on its id to get the index in the hash table. The content of this index in the table points to the list of processes with the same

hash function value. A sequential search through this list is required to access the required process.

Inter-process Communication

Inter-process communication is implemented by sending messages. Whenever a message is generated by some process it is inserted at the end of a global message queue. A message has the following fields:

1. source-id : id of the source process.
2. dest-id : id of the destination process.
3. command : type of message; it can be one of "start," "redo," "success" and "fail." ("cancel" messages are not implemented)
4. ptr-to-frame : pointer to the stack frame. This is used to send a clause environment with "start" message and to send an updated environment with "success" messages.
5. next-message : pointer to the next message in the queue.

Interpreter Loop

When the user requests solution of a goal statement, the goal statement is parsed and the interpreter is invoked. The interpreter starts by setting up a root AND-process for

solving the goal and inserts it in the AND-process pool. It then inserts a "start" message for this process in the message queue. The interpreter then loops until the goal is solved or until it fails. In each cycle, it removes the first message from the message queue and gets the destination process from the appropriate process pool. The choice of a process pool is determined by the value of the process-type field in the message. The type of message is decoded and the appropriate action is taken depending upon the current state of the process. In response to this message, the state of the process will change, and it may generate new processes and messages. The new processes are inserted in the process pools and the messages are inserted in the message queue. After the state transition is complete, the interpreter loops back and removes the first message from the queue and the whole procedure is repeated again. An empty message queue is an erroneous situation unless the last message sent was a success for the user's original goal.

Variable Bindings

Variable bindings are stored on two stacks, called the "local stack" and the "global stack." These stacks are sequential arrays of cells. The structure of cells has been described earlier in this chapter. Whenever an AND-process is created a stack frame is allocated on the local stack for

the variables in the goal statement. This stack frame is called the clause environment of the goal. The size of this stack frame is equal to the number of variables in the goal statement plus a cell to store additional information about the frame.

When variables are bound to simple terms (atoms or other variables), the binding is stored in the cell in the environment. When a variable is bound to a complex term, an instance of the complex term is constructed on the global stack and a pointer to the term is stored in the variable's slot in local stack frame.

When an AND-process starts a descendant OR-process, it sends a pointer to its clause environment to the OR-process via a "start" message. The OR-process does a look up in the data base of clauses to see if the head of any clause is a potential match with the literal to be solved; a potential match has the same functor and arity. For each potential match, a copy of the parent's stack frame is created on the local stack, and a stack frame is allocated for variables in the candidate clause. Complex terms from the global stack are also duplicated. A unification algorithm is then called to check if the goal literal and the head of the candidate clause are unifiable. As a side effect of this unification algorithm, the two stack frames are updated to reflect the variable bindings.

If the unification fails then the two stack frames are

garbage collected. If the unification succeeds and if the candidate clause is an implication, then an AND-process is created to solve the body of the candidate clause. The stack frames are then "closed" to remove references to the parent stack frame from the descendant stack frame. Unification and environment closing are described by Conery [5]. The closed stack frame is passed as the clause environment for the AND-process along with the "start" message.

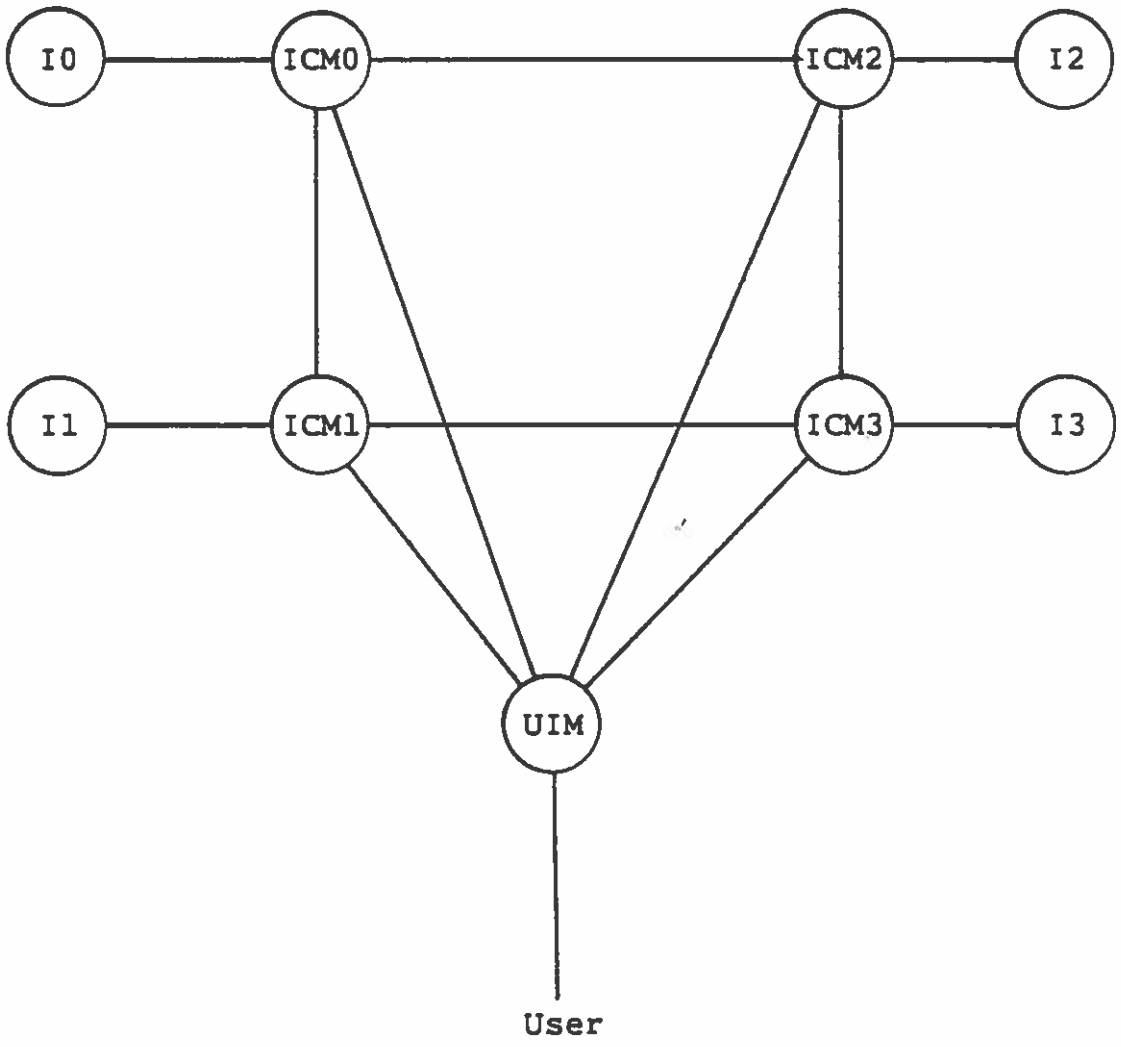
If the unification succeeds and the candidate clause is an assertion then the updated parent stack frame contains an answer for the goal literal. This stack frame is added to the WL list of OR-process and is eventually sent to the parent along with the "success" message. The descendant stack frame can be garbage collected. If no matching clauses are found or if none of the matching clauses unify with the goal literal then a "fail" message is sent to the parent AND-process.

When an AND-process receives a stack frame along with a "success" message from one of its descendants, it creates an OR-process to solve the next literal from the goal list. The stack frame received with the "success" message becomes the current environment of the AND-process and is sent to the next OR-process with the "start" message. If no more literals are to be solved, then the AND-process simply passes on the stack frame to its parent as an answer frame.

When an OR-process receives a stack frame from an AND-process along with the "success" message, it combines the stack frame with the copy of the parent's stack frame associated with this AND-process. This is done by closing the parent's stack frame with respect to the answer frame [5].

Multi-processor Implementation

Figure 3 shows the configuration of the system on five processors. This configuration can easily be extended to more processors. For the purpose of multi-processor implementation, the Communication Interface Module (CI) from the single processor implementation is separated into two modules. These modules are called User Interface Module, UIM, and Inter-processor Communication Module, ICM. In Figure 3, there are four ICM's and one UIM. The UIM runs on a separate processor. An interpreter and an ICM share a processor. All of these units can talk to each other by sending messages. UIM is an interface between the user and the system. All the clauses typed in by the user are parsed by UIM and are broadcast to all ICM's. ICM's, in turn, forward these clauses to the respective interpreter. This ensures that a copy of the program is available at each interpreter. The interpreter stores these clauses in an internal data structure. When the user types in a goal statement, UIM parses it and sends it to only one ICM. This



The User Interface Module (UIM) acts as an interface between the user and the system. The Inter-processor Communication Modules (ICMs) are responsible for inter-processor communication; the communication is asynchronous. An ICM and an interpreter (I) share a processor.

FIGURE 3. Configuration of the system on five processors.

ICM forwards the goal clause to its interpreter. The interpreter starts solving the goal by creating AND-processes and OR-processes. To reduce the load on this interpreter, some of these processes and other data structures are transported to other interpreters via ICM's. This kind of communication via messages goes on until the original goal is solved or until it fails. Once the goal is solved, its solution is sent to the UIM. The UIM, in turn, reports the solution to the user and waits for further instructions from the user.

To have distinct ids for all processes in the system for the purpose of efficient message passing, the id is a structure of three fields: "process-no," "host-machine," and "execute-machine." Each of these fields is a single integer. "Host-machine" is the processor number on which the process is created, and "execute-machine" is the processor number on which the process is running. Since each machine maintains its own counter for the "process-no" field, these ids are unique in the system.

The distribution of processes and other data structures at runtime are very important factors in the efficiency of the parallel model on a multi-processor system. We have assumed a simple mechanism of distributing processes. A simple hashing function is applied on the process-id to determine where that process should be transported. More elegant methods can be used to implement this in order to

make the system more efficient. For example, distribution of processes may depend upon the number of processes in the process pools and the number of messages in the message queue at each processor so that equal distribution of work can be achieved.

All the interpreters run concurrently on separate processors, sending messages to each other asynchronously. In the single processor implementation, the communication between the Communication Interface and the interpreter is synchronous; in this implementation, either one of them is sending messages at any given time. The ICM and the interpreter are separate units to ensure that the interpreter is independent of the communication protocol among the processors. This system can be implemented on any tightly coupled processors or loosely coupled network of processors by proper choice of ICM's.

The representation used for stack frames assures proper execution of this model no matter which interpreter in the network performs a process transition. Stack frames passed in "start" messages and "success" messages are in closed form, meaning there are no pointers to local stacks in other interpreters.

Chapter Summary

This chapter has introduced the implementation of the AND/OR Process Model on single processor and multi-processor

systems. Various data structures used for the implementation were discussed in detail. The clause data base is organized as a hash table. The head functor of the clause and its arity serve as keys for hash table access. AND-processes and OR-processes are implemented as data structures, which are kept in process pools. The process-id is used as the key for storing a process in the pool. Processes communicate with each other by sending messages. A message queue is maintained to store all messages. In each interpretation cycle, the interpreter removes the first message from the queue and takes appropriate action. This action may lead to a change in the state of a process and/or generation of new processes and messages. Closed environments are passed with "success" and "start" messages. This ensures that the environments are portable.

Implementation of the system on a multi-processor system has three kinds of modules. These modules are User Interface Module, Inter-processor Communication Module and Interpreter. An Inter-processor Communication Module is used for routing messages to appropriate destinations. Implementation of this module depends upon the type of processors connected to each other. Hence this module is kept separate from the interpreter, in order to keep the interpreter independent of the types of processors in the network and the communication protocol between these processors. The User Interface Module runs on a separate

processor and serves as an interface between the user and the system. An Inter-processor Communication Module and an interpreter share a processor. A copy of the program is available at each interpreter. The goal statement is given to only one of the interpreters. This interpreter starts solving the goal by creating AND-processes and OR-processes. To reduce the load on this interpreter, it transports some of its processes and other necessary data structures to neighboring interpreters. All the interpreters solve the goal through a combined effort by communicating asynchronously with each other via messages.

CHAPTER IV

EXPERIMENTS

This chapter discusses various experiments carried out to test the implementation of the AND/OR Process Model and lists the measurements gathered in the process. The following statistics were gathered for each test program:

1. Number of OR-processes created.
2. Number of AND-processes created.
3. Number of OR-process pool accesses.
4. Number of AND-process pool accesses.
5. Maximum size of the message queue.
6. Number of "start" messages processed.
7. Number of "success" messages processed.
8. Number of "redo" messages processed.
9. Number of "fail" messages processed.
10. Total number of messages generated.
11. Number of unifications attempted.
12. Number of successful unifications.
13. Number of successful unifications with assertions.
14. Number of clause data base accesses.
15. Size of local stack without garbage collection.

16. Size of local stack with garbage collection.
17. Size of global stack.
18. Number of local stack frames used.
19. Number of local stack frames garbage collected.
20. Percentage time spent in unification.
21. Percentage time spent in duplicating local stack frames.
22. Percentage time spent in sending messages (on the same processor).
23. Percentage time spent in allocating and deallocating (garbage collecting) memory.

Measurements

The purpose in gathering these measurements is to see how large data structures become, how often they are accessed, where the most time is spent, and to collect statistics about message passing (item nos. 5-10) and unification (item nos. 11-13). The number of processes (item nos. 1 and 2), the load on the processor (item no. 5), the number of messages (item nos. 6-10), and size of stacks (item nos. 15-18) gives an idea about the total size of data structures. It also reflects the need for an efficient garbage collection mechanism (item nos. 18 and 19). By counting the number of accesses to the process pools (item nos. 3 and 4) and to the clause data base (item no. 14), one can judge how often the data structures are accessed.

Timing analysis (item nos. 20-23) helps in identifying often used parts of the system, so more effort can be expended in the future for improving those. Other statistics may be helpful in the improvement of the basic model of the system itself.

The statistics mentioned in the previous section were gathered for three programs: "Map Coloring Problem," "Expression Grammar," and "List Reverse." The listings of the programs are given in Figures 4 through 6. Tables 1 through 3 summarize the respective measurements obtained during the experiment. These programs were selected to test the unification algorithm, message passing space requirement, backtracking, and analysis of the time spent among different modules of the interpreter.

Analysis of Measurements

After conducting the experiments, it was evident from the measurements that the size of the data structures is very large even with garbage collection. Currently garbage collection is done for almost all major data structures except the global stack. Figures obtained for the size of local stack with and without garbage collection conveys the importance of garbage collection. A lot of space can be saved if the global stack is garbage collected in the problems in which a considerable number of complex terms are created on the global stack. The garbage collection

```
color(A,B,C,D,E) <- next(A,B) & next(C,D) & next(A,C) &
                    next(A,D) & next(B,C) & next(B,E) &
                    next(C,E) & next(D,E).

next(red,blue).

next(red,yellow).

next(red,green).

next(blue,red).

next(blue,yellow).

next(blue,green).

next(yellow,red).

next(yellow,blue).

next(yellow,green).

next(green,red).

next(green,blue).

next(green,yellow).

?- color(A,B,C,D,E).
```

FIGURE 4. Map coloring problem.

TABLE 1. Measurements for "Map Coloring Problem"

Category	Measurement
No. of OR-processes	22
No. of AND-processes	2
No. of OR-process pool accesses	71
No. of AND-process pool accesses	40
Maximum message queue size	3
No. of "start" messages processed	24
No. of "success" messages processed	23
No. of "redo" messages processed	14
No. of "fail" messages processed	13
Total no. of messages generated	75
No. of unifications attempted	253
No. of successful unifications	39
Successful unifications w/ assertions	38
No. of clause data base accesses	22
Local stack size w/o garbage collection	1794
Local stack size w/ garbage collection	303
Global stack size	0
No. of local stack frames used	509
Local stack frames garbage collected	428
% time spent in unification algthm.	24.0
% time spent duplicating stack frames	19.9
% time spent sending messages	4.6
% time spent alloc. and dealloc. memory	29.3

```
expr(E,S0,S1) <- term(E,S0,S1).
expr(plus(T1,T2),S0,S1) <- term(T1,S0,S2) &
                           plus(S2,S3) & expr(T2,S3,S1).
expr(minus(T1,T2),S0,S1) <- term(T1,S0,S2) &
                            minus(S2,S3) & expr(T2,S3,S1).

term(T,S0,S1) <- factor(T,S0,S1).
term(times(F1,F2),S0,S1) <- factor(F1,S0,S2) &
                            times(S2,S3) & expr(F2,S3,S1).
term(divide(F1,F2),S0,S1) <- factor(F1,S0,S2) &
                             divide(S2,S3) & expr(F2,S3,S1).

factor(F,S0,S1) <- id(F,S0,S1).
factor(F,S0,S1) <- digit(F,S0,S1).
factor(F,S0,S1) <- lpar(S0,S2) & expr(F,S2,S3) &
                   rpar(S3,S1).

plus([43|L],L).
minus([45|L],L).
times([42|L],L).
divide([47|L],L).

lpar([40|L],L).
rpar([41|L],L).

id(a,[97|L],L).
id(b,[98|L],L).
id(c,[99|L],L).

digit(0,[48|L],L).
digit(1,[49|L],L).
digit(2,[50|L],L).
digit(3,[51|L],L).
digit(4,[52|L],L).
digit(5,[53|L],L).
digit(6,[54|L],L).
digit(7,[55|L],L).
digit(8,[56|L],L).
digit(9,[57|L],L).

?- expr(E,[40,97,47,98,43,99,41,45,40,99,43,98,42,97,41],
        []).
```

Note: The expression being parsed is "(a/b+c)-(c+b*a)."

FIGURE 5. Expression grammar.

TABLE 2. Measurements for "Expression Grammar"

Category	Measurement
No. of OR-processes	5836
No. of AND-processes	4642
No. of OR-process pool accesses	26700
No. of AND-process pool accesses	24316
Maximum message queue size	821
No. of "start" messages processed	10478
No. of "success" messages processed	4629
No. of "redo" messages processed	4616
No. of "fail" messages processed	10403
Total no. of messages generated	30136
No. of unifications attempted	20711
No. of successful unifications	5837
Successful unifications w/ assertions	1196
No. of clause data base accesses	5836
Local stack size w/o garbage collection	187429
Local stack size w/ garbage collection	72071
Global stack size	148480
No. of local stack frames used	47881
Local stack frames garbage collected	34358
% time spent in unification algthm.	23.4
% time spent duplicating stack frames	27.6
% time spent sending messages	8.4
% time spent alloc. and dealloc. memory	20.4

```
reverse([],[]).
reverse([A|L],R) <- reverse(L,Tmp) & append(Tmp,[A],R).

append([],L,L).
append([X|A],B,[X|C]) <- append(A,B,C).

?- reverse([a,b,c,d,e,f,g,h,i,j],L).
```

FIGURE 6. List reverse.

TABLE 3. Measurements for "List Reverse"

Category	Measurement
No. of OR-processes	66
No. of AND-processes	56
No. of OR-process pool accesses	366
No. of AND-process pool accesses	350
Maximum message queue size	4
No. of "start" messages processed	122
No. of "success" messages processed	121
No. of "redo" messages processed	118
No. of "fail" messages processed	116
Total no. of messages generated	479
No. of unifications attempted	132
No. of successful unifications	66
Successful unifications w/ assertions	11
No. of clause data base accesses	70
Local stack size w/o garbage collection	1654
Local stack size w/ garbage collection	957
Global stack size	1960
No. of local stack frames used	375
Local stack frames garbage collected	185
% time spent in unification algthm.	9.8
% time spent duplicating stack frames	25.0
% time spent sending messages	12.0
% time spent alloc. and dealloc. memory	22.5

algorithm for the global stack is complex because the references to the structure created on the global stack need not be to the first cell of the structure and any number of such references can be made to various parts of the same structure. So it is not easy to deduce whether a structure on the global stack can be garbage collected.

Most time is spent in unification, duplicating stack frames, and allocating and deallocating memory. The time spent in duplicating stack frames and the total number of stack frames used can be reduced if structure sharing is employed.

The maximum load on the processor is the same as the maximum size of the message queue. The maximum message queue size for "Expression Grammar" was 821, which is a large number. As the number of processors increases, the load on the processors will decrease considerably up to a certain point because the work will be divided among the processors.

The number of process pool accesses is also a large number, hence pool accesses should be made as efficient as possible. A simple hash function was used for organizing the process pools. A better hash function would increase the efficiency considerably.

Conclusion

The aim of this project was to develop a parallel

interpreter for logic programs. The first version of the interpreter, as described in this thesis report, has been implemented on a loosely coupled network of UNIX systems. This implementation is the first step in the ongoing research work in logic programming at the University of Oregon. The experiments conducted to test the interpreter have provided an insight into developing more efficient versions of the same. As of now, unification and duplication of stack frames consume a considerable chunk of the interpretation time; a coprocessor or a smart memory devoted entirely to the task of unification would probably save a lot of time. The interpreter is currently being implemented on a tightly coupled multiprocessor system.

CHAPTER V

FUTURE WORK

This chapter suggests a few extensions to the current implementation of the AND/OR Process Model. These include a graphical interface that shows the execution of processes and/or shows the status of the interpreters, sharing of runtime data structures, a dynamic load balancing mechanism for effective utilization of the processors, and exploitation of AND-parallelism. The following sections will describe these extensions briefly.

Graphics Interface

A color graphics interface would improve the user interface dramatically. The working of the model could then be graphically displayed, and this will help the user in monitoring the execution of the interpreters. Statistics about the number of messages, the number of processes et cetera could be displayed for each interpreter. Whenever a message is added to the message queue or a process is added to the pools, the display could be updated. The display could be in the form of dynamically changing bar charts or

simple numbers. Separate bar charts could be maintained for different types of messages which will give an idea about how many messages of each type are currently in the message queue. Similarly, messages for AND-processes and OR-processes could be shown separately. Instead of numbers, colors could be used. A picture of the network could be drawn, using shades of color to indicate activity.

In the second, the process tree of each interpreter could be displayed separately. The process currently running in the interpreter could be highlighted and its data structure could be displayed on the screen. For example, in the case of an AND-process, the goal being solved, the current literal being solved et cetera could be displayed. Details about the message being processed could also be displayed. This type of interface will form a basis for a debugging interface. The type of debugging features that should be provided and how they would be implemented is currently a major research topic.

Sharing of Data Structures

To keep the implementation simple, sharing of data structures was not employed in the first version. Clause data base, process pools, message queue, local stack frames and global stack could be shared. Sharing of data structures would increase the efficiency of the implementation. It would save space and time required in

duplicating the data structures.

Two types of data structure sharing are possible. Firstly, the data structures could be shared by processes on the same processor. This is not so difficult to achieve because the processes share a common memory. Secondly, the data structures could be shared by processes on different processors. This kind of sharing is not possible if the processors are loosely coupled because memory can not be shared by the processors. It is only possible if the processors are tightly coupled. To share data structures in this case, local memories of different processors or global memory need to be accessed. This accessing using the common bus may result in bus contention and an increase in processing time. Explicit copying of the data structures may be preferred in this case. The tradeoff involved in the choice between explicit copying and data structure sharing depends upon the average size of data structures to be shared, the average number of accesses to these data structures, and the architecture of the machine (which determines how expensive a global memory access is).

Since the current implementation is on a network of loosely coupled processors, we chose to share as many data structures as possible among the processes on the same processor and explicit copying of data structures is used for sharing information among the processes on different processors. However, in the tightly coupled multi-processor

implementation, some experiments can be conducted to choose between explicit copying and sharing of data structures. In this implementation, a single copy of the source program kept at a central location will suffice. Also, a pointer to the location where a data structure is stored in the global memory can be passed during inter-processor communication rather then passing a copy of the data structure.

Sharing of data structures on the same processor has been implemented in a few places in the implementation described in this thesis. Currently, clauses represented as Cell-memory and Clause-nodes (described in chapter III), process pools, message queue et cetera are shared by processes. How stack frames from the local and global stack are shared and when explicit copying is done by various processes during unification and closing operation is described by Conery [6]. However, there are a few other places where some more data structure sharing can be done. During closing operation, a lot of explicit copying of stack frames is done. This may be reduced.

Load Balancing

Devising an appropriate technique for load balancing is a very hard problem because the time required by a process to solve a literal can not be predicted beforehand. An efficient policy for load balancing would lead to the optimal utilization of the processors. This would involve

dividing the work equally among the processors. This can be done by migrating processes from a processor with heavy load to another with lesser load. The messages for these processes should be redirected appropriately to the new destination. Keller has employed a similar load balancing technique in the Rediflow Model [7]. Burton and Sleep use a method in which the processes are categorized into three sets, viz., pending, blocked and running; in this the processes which are pending can be stolen by neighboring processor [2]. The load on a processor can be calculated as a function of the number of messages in the message queue.

The cost involved in the migration of processes and redirection of messages from one processor to the other should not exceed the cost due to underutilization of a processor. This tradeoff should be considered while formulating a strategy for load balancing. The communication cost can be roughly estimated by counting the number of bytes transfered among the processors.

In the current implementation, we have employed a "hashing model." A hash function is applied on the process-id of a process to determine the processor on which it should run. Better hashing techniques can be employed. A hashing function can be applied on the functor being solved to determine the destination processor. This will not be very effective if a lot of clauses are recursive in nature; most clauses in a logic programming language tend to

be recursive. If the hashing function is history sensitive, i.e., if it remembers the previous hash value for a particular functor, then a different hash value can be provided every time the hash function is called for that functor. This will increase efficiency because the parts of a recursive clause will be executed on different processors.

The hash function can be improved by using the knowledge of how clauses are distributed among the processors. If each processor has a copy of all clauses then hash functions as described in the previous paragraph could be used. Otherwise, the value of hash function applied on a functor would depend upon where clauses with this functor are situated.

Exploiting AND-parallelism

In the current implementation, only OR-parallelism is exploited. The current model can be enhanced to exploit some AND-parallelism. This could be done by having an AND-process create OR-processes for more than one literal at a time. An effective method for achieving AND-parallelism is a problem of deciding which literals must be solved sequentially and which can be solved in parallel. Conery presented a scheme for implementing AND-parallelism by a method of ordering of literals, forward execution and backward execution [4]. This method could be added in the current implementation to achieve some AND-parallelism.

BIBLIOGRAPHY

1. Borgwardt, P. Parallel Prolog using stack segments on shared-memory multiprocessor. In Proceedings of the International Symposium on Logic Programming. IEEE, Atlantic City, N.J., Feb. 1984, pp. 2-11.

2. Burton, F.W., and Sleep, M.R. Executing functional programs on a virtual tree of processors. In Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture. ACM, Portsmouth, N.H., 1981, pp. 187-194.

3. Ciepielewski, A., and Haridi, S. Control of activities in the parallel token machine. In Proceedings of the International Symposium on Logic Programming. IEEE, Atlantic City, N.J., Feb. 1984, pp. 49-57.

4. Conery, J.S. The and/or process model for parallel interpretation of logic programs. Ph.D. Dissertation, Univ. of California, Irvine, 1983.

5. Conery, J.S. Partitioned memory representation for parallel logic programs. Tech. Rep. 86-002, Univ. of Oregon, April, 1986.

6. Conery, J.S., and Kibler, D.F. Parallel interpretation of logic programs. In Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture. ACM, Portsmouth, N.H., 1981, pp. 163-170.

7. Keller, R.M., Lin, F.C.H., and Tanaka, J. Rediflow multiprocessing. To appear in Compcon '84, IEEE, San Francisco, 1984.

8. Kumon, K., Masuzawa, H., Itashiki, A., Satoh, K., and Sohma, Y. Kabu-wake: A new parallel inference method and its evaluation. To appear in Compcon '86, IEEE, San Francisco, 1986.

9. Nakagawa, H. And parallel Prolog with divided assertion set. In Proceedings of the International Symposium on Logic Programming. IEEE, Atlantic City, N.J., Feb. 1984, pp. 22-28.

10. Tick, E., and Warren, D.H.D. Towards a pipelined Prolog processor. In Proceedings of the International Symposium on Logic Programming. IEEE, Atlantic City, N.J., Feb. 1984, pp. 29-40.

11. Warren, D.S., Ahamad, M., Debray, S.K., and Kale, L.V. Executing distributed Prolog programs on a broadcast network. In Proceedings of the International Symposium on Logic Programming. IEEE, Atlantic City, N.J., Feb. 1984, pp. 12-21.