Eager and Lazy Enumerations in Concurrent Prolog

Hideki Hirakawa, Takashi Chikayama, Koichi Furukawa

ICOT Research Center

Institute for New Generation Computer Technology,
Mita Kokusai Bldg. 21F, 4 - 28 Mita 1-chome,
Minato-ku, Tokyo 108

Abstract

Logic programming languages have inherent possibility for AND-parallel and OR-parallel executions. Concurrent Prolog designed by E.Shapiro introduces an AND-parallelism and an limited OR-parallelism, i.e, a don't-care-nondeterminism. The other aspect of OR-parallel execution, i.e, don't-know-nondeterminism is formalized as an 'eager_enumerate' operation on a set expression. This paper describes a computational model which provides the eager enumerate function to Concurrent Prolog and shows its implementation in Concurrent Prolog itself. This paper also shows a lazy enumerate function can be implemented easily by introducing a bounded buffer communication technique to the eager enumerator.

1. Introduction

A growing area of research in highly parallel processing covers computer architectures, programming languages and computational models. One of the best candidates for high level machine language for highly parallel processors is a logic programming language which represents AND and OR relations between predicates. Logic programming languages have inherent possibility of parallel processing, that is, AND-parallel and OR-parallel execution.

Based on this idea, several parallel programming languages are proposed: such as KL1 [Fur 84], Concurrent Prolog [Sha 83], PARLOG [Cla 83] and Bagel machine language[Sha 84]. Researches in parallel programming are being conducted using these languages. In these languages, AND-parallelism is used for the description of parallel processes, which is based on the process interpretation of logic [Emd 82]. OR-parallelism has two aspects so called don't-care-nondeterminism and don't-know-nondeterminism [Kow 79]. The don't-care-nondeterminism is adopted in all the languages mentioned above. However, the don't-know-nondeterminism is introduced only in PARLOG and KL1 where it is used to find multiple solutions for a query. PARLOG and KL1 use a "set expression" as the interface between

AND-parallelism and OR-parallelism (don't-know-nondeterminism).

In this paper we regard the OR-parallelism to find all solutions as 'enumerating' elements of a set in the same way as in KL1. This paper describes the 'enumeration' in Concurrent Prolog, that means the implementation of OR-parallel execution in AND-parallel one. An advantage of this approach is that both AND-parallel and OR-parallel execution can be achieved within a small basic framework of Concurrent Prolog. This implies the decreasements of the complexity of the architecture and of the amount of required hardware of the parallel machine.

Various models for parallel processing of logic programs are proposed from the computational model view points. Nitta and Conery described parallel interpretation methods based on AND/OR process model [Nit 83], [Con 83]. Haridi proposed a language based on natural deduction, which covers a wider class of statements than Horn Logic [Har 83]. Hirakawa proposed a computational model based on multi-processing and graph reduction mechanism [Hir 83]. In this paper, a computational model for Pure Prolog is introduced. This model is based on multi-processing and message communication between processes. In this model, goals are computed serially from left to right, and clauses are computed in parallel.

Based on this model, we have implemented a Pure Prolog interpreter in DEC-20 Concurrent Prolog [Sha 83] to realize the 'enumerate' function mentioned above. This interpreter works eagerly to get all solutions for a given goal. By replacing the stream communication portion used in the interpreter with bounded buffer communication implementation and by adding some small changes, a lazy interpreter which works in accordance with demands can be obtained easily.

Section 2 of this paper explains the concept of 'enumeration'. Section 3 describes the computational model and its implementation in Concurrent Prolog. Section 4 describes the modification of the interpreter from eager version to the lazy one.

2. Enumerations

An interface between AND-parallelism and OR-parallelism (a don't know determinism) is introduced using set expressions in PARLOG and KL1. A set expression has the syntax such as:

{X|Y}
where X is a term and Y is a goal sequence

In KL1, the basic operation on a set is an 'enumerate' operation. In this paper the same expression is introduced in Concurrent Prolog as in KL1. 'Enumerate' is similar to the 'bagof' operation in DEC-10 Prolog [War 81].

Prolog	Concurrent Prolog
bagof(X,Y,Collection)	enumerate({X|Y},Stream)

The meaning of 'bagof' literal above is "Collection is the collection of terms of the form X, which satisfy the goal sequence Y". In Concurrent Prolog, 'Stream' in 'enumerate' clause is the same as 'Collection' in 'bagof' logically, but it is a stream of terms rather than a simple collection. This is a natural interface to an AND-parallel process.

There are two types of streams. One is an 'uncontrolled stream' and the other is a 'controlled stream'. 'Uncontrolled' means that once 'enumerate' is called, its output stream is never stopped until all the solutions are generated. On the other hand, 'controlled' means that the generation of the solutions is invoked by a demand of a process outside of 'enumerate'. The former type of enumeration is called 'eager enumeration' and the latter 'lazy enumeration'. The eager enumeration is used for finding all solutions to a database query and generally requires many computation resources, while lazy enumeration is used for finding a part of solutions which satisfy some requirements of other processes. The following are the simple examples of lazy and eager enumerations.

Eager enumeration :

"display all the country with more than one hundred million population"

```
Goal: eager_enumerate({Nam|country(Nam,Capl,Pop),Pop>100},Str),
      display_stream(Str?).
```

Lazy enumeration :

"display three countries with more than one hundred million population"

```
Goal: lazy_enumerate({Nam|country(Nam,Cap,Pop),Pop>100},Str?),
      display(Str,3).
```

In the above examples, 'enumerate' and 'display' run in parallel (concurrently). In the former example, 'eager_enumerate' produces a stream of country names and 'display_stream' displays them in turn. In the latter example, 'display' sends three demands for solutions to 'lazy_enumerate' and 'lazy_enumerate' produces them. The details will be described later.

3. Eager Enumeration

The eager enumeration is provided by a Prolog interpreter which computes subgoals serially and clauses in parallel. In this section, a computational model for an eager interpreter and its implementation in Concurrent Prolog are described.

3.1 Computational Model

3.1.1 Components

The computational model for the eager interpreter consists of three components: processes, channels and a Horn Clause Database(HDB).

A process plays a key role in a computation. An arbitrary number of processes can be generated in a system. A process corresponds to a clause being computed, such as H<--G1,G2. There are two types of processes, that is, active and waiting. The waiting process waits until it receives data from another process.

A channel is a communication path between processes and is dynamically generated during the computation. Data transferred through a channel is called a message. A message is passed from a process called a "generator" to processes named "consumers". The distinction between a generator and a consumer is relative, and a single process can simultaneously play both roles. One generator process can simultaneously send a message to multiple consumer processes via a channel. Similarly, one consumer process can be connected to multiple generators.

The Horn Database (HDB) is a set of Pure Prolog clauses. A process can fetch the set of clauses which have the heads unifiable with a certain term. A fetching operation about term P is called "P-related fetch".

3.1.2 Process Operation

In the computational model given here, computation progress while multiple processes are exchanging messages. This subsection provides a more detailed description of the process, shows a simple example, and presents the execution mechanism of the computational model.

A process is defined by five components: Status, Head, Goals, Input-Channel, and Output-Channel, as shown in the following format:

process(Status, Head, Goals, Input-Channel, Output-Channel)

'Status' indicates the state of a process and is either 'active' or 'waiting'. An active process can carry on computation by itself, while a waiting process can perform no processing until it receives a message. 'Head' is a predicate (term) and represents what the process must eventually compute. 'Goals' is either null, 'true' or a sequence of predicates and indicates the predicates to be computed to compute the Head. For example, if the HDB includes 'a<--b,c', there may be the following process:

process(Status, a, (b,c), Input-Channel, Output-Channel)

In addition, if the predicate b has been computed, there may be a process as follows:

process(Status, a, (c), Input-Channel, Output-Channel)

'Channel' is used to transfer messages among processes as described above. A process appears as a consumer for its Input-Channel, while it functions as a generator for its Output-Channel.

Here, we will define the operation of a process.

(A) Active process

The operation mode of an active process is either reduction or termination. In reduction mode, the rightmost subgoal of a clause is expanded using inference rules in HDB; the active process is maintained after the reduction is completed. By contrast, termination means that inference reaches 'true' or the application of an inference rule fails; in both cases, the process is immediately deleted.

Operation in reduction mode

Assume process(active, H, G, I, O). If G is neither null nor 'true' and G is in the form of either P or (P,...) where P is a predicate defined in the HDB, then the process performs a P-related fetch to the HDB to obtain a clause set, S, generates active processes for all the components of S, and connects each process with itself through Channel I (each process functions as a producer). It also changes its status to 'waiting'.

Operation in termination mode

There are two types of terminations: success or failure. A success termination occurs when reduction reaches true, while a failure termination occurs when a fetch operation fails. The failure termination corresponds to Prolog's 'fail'.

Success termination
When G is either null or true, the process sends H via channel O and deletes itself.

Failure termination
The process deletes itself.

(B) Waiting process

Having received a message (term) M via channel I, a waiting process generates G', a copy of its Goals G, in the format P' or (P', P1,...), and unifies the head element P' with M (Transfer of the computation results.) Then, it establishes NewG, which is G' with its head element removed. However, when G' contains only P', NewG will be true. Then, the waiting process generates the following active process:

process(active, H, NewG, I', O)
Where I' is a new channel.

The waiting process will be maintained in the original form.

The entire computation terminates, when all the processes are deleted.

3.1.3 Simple Computation Example

This subsection presents a simple example to show the way the computational model is executed. In the following figures, the active process p, the waiting process q and the channel c are denoted by (...)p, [...]q and --c-->, respectively. (p, q and c may be omitted.) The Head H and Goals G are shown as H<--G.

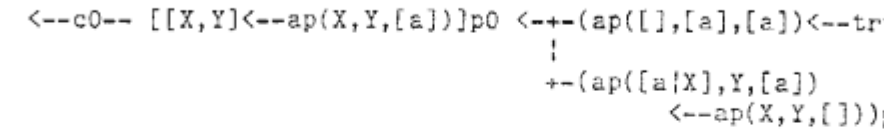
Assume that the HDB is given as follows:

HDB = {ap([],X,X). ap([U|X],Y,[U|Z])<--ap(X,Y,Z).}

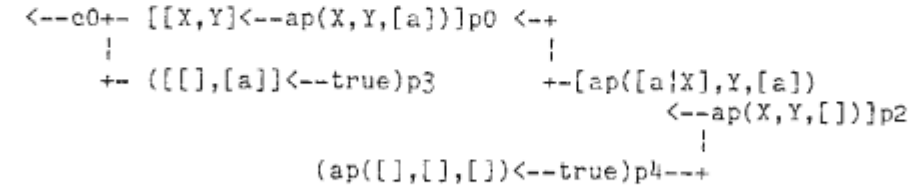
To compute [X,Y] that satisfies a goal ap(X,Y,[a]), the following process is generated as the initial process:

<--c0-- ([X,Y]<--ap(X,Y,[a]))p0

A message output through c0 is the solution. Since p0 is an active process, it performs a fetch operation and generates new processes, p1 and p2, and then changes the status from active to waiting.

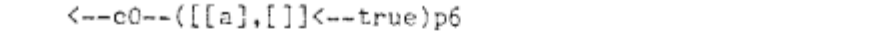


There are two active processes. Each process runs simultaneously. As p1 has a terminated clause, it sends the head of the clause and deletes itself; p0 receives message 'ap([],[a],[a])' and creates a new process p3; p2 performs a reduction mode operation and produces a new process p4.



An active process p4 sends the message 'ap([],[],[])' to p2 and deletes itself. Receiving the message, p2 creates a new process p5 and deletes itself because it has no child process; p3 sends '[[],[a]]' (the first solution) to c0 and deletes itself.

P5 sends the message 'ap([a],[],[a])' to p0 and deletes itself. p0 produces p6 and deletes itself.



P6 sends message '[[a],[]]' (the second solution) to c0 and , finally, deletes itself.

3.2 Eager Interpreter Implementation

3.2.1 Concurrent Prolog

Concurrent Prolog adopts AND-parallelism to describe concurrent processes and OR-parallelism to describe nondeterministic actions of processes (don't-care-nondeterminism). Once a clause is selected, the choice of other clauses is ignored. Concurrent Prolog uses variables shared by processes running concurrently for interprocess communications. (For further details, refer to [Sha 83]).

This subsection gives a simple program example to provide the necessary information for later discussion. The program outputs, if person 'X' is a man, his daughter's name, and, if 'X' is a woman, her son's.

```
(a) opposite_sex_child(X):- man(X) | daughter(X,Y),output(X,Y?).
(b) opposite_sex_child(X):- woman(X) | son(X,Y),output(X,Y?).
```

The symbol '|' is called a guard bar and separates a guard sequence from a goal sequence. The guard bar has the meaning similar to Prolog's 'cut', and cuts other alternative clauses. The comma ',' in Concurrent Prolog has the different meaning from that in Prolog and denotes parallel-AND relationship, it is a logical equivalent for the ordinary AND. Concurrent Prolog uses a symbol '&' to express serial-AND relation. '?' means "a variable attached with '?' should not be instantiated to a non-variable term". '?' is called 'read-only annotation', and a ?-attached variable is referred to as a 'read-only variable'. The read-only annotation permits shared-variable-based communications between concurrent processes (interprocess communication). Also, in Concurrent Prolog, a process with a read-only variable waits until another process instantiates a value to the variable (process synchronization).

3.2.2 Eager Interpreter in Concurrent Prolog

With the computational model implemented in Concurrent Prolog, a process is expressed by the following term:

```
process(Status,OutputChannel,InputChannel;Clause)
```

A generation of a process is performed by parallel AND's such as 'process :- process1,process2', and a deletion of a process is expressed by termination of the process, 'process:-true'. A channel is implemented by shared variables and process synchronization is achieved with read-only annotation. Although not shown in this paper, our system constructs the HDB using a meta representation, 'ax(Horn clause)', in the internal database of Concurrent Prolog. Fig.1 shows the program of the eager interpreter.

(p1) to (p3) define the behavior of active processes, while (p4) and (p5) defines that of a waiting process.

(p1) performs reduction. The predicate 'reduce' checks whether or not the first element of the subgoals in 'Cls' is defined in the HDB. When the first element is not found in HDB, the predicate 'reduce' fails. When the guard portion of (p1) succeeds, two predicates in the goal portion, 'process' and 'process_fork', are executed in parallel. 'process' is the original process in waiting mode, and '?' is attached to the variable 'Inch'. 'process_fork' generates a new active process for each of newly fetched clauses. 'merge' predicate is used for constructing a channel between a parent process and its child processes. Note that this merger deletes itself, when one input channel is closed.

(p2) corresponds to a process in a termination mode. The predicate 'terminate' checks that 'Cls' is in the format 'X<--true'. The second argument '[Mess]' specifies that the message is sent to 'OutCh' and the active process is terminated. Then, the process deletes itself.

(p3) shows the operation of active processes in which further reduction has become impossible. (p3) deletes itself closing the output channel.

In (p4), the Input-Channel is a read-only variable; when a value is instantiated to the variable (i.e, when a message is received), the process starts operating. The predicate 'newclause' generates a copy 'NewC' from original clause 'Cls' according to the waiting process operation definition mentioned in 3.1.2. The goal portion of the program specifies a new process generation with the new clause and the original process to be remained as it was. The output channels of these two process ('OutCh1' and 'OutCh2') are merged into the original output channel 'OutCh'.

(p5) is for a waiting process with closed message stream, which means that all the child processes have completed their jobs. The waiting process deletes itself closing its output channel.

Using this interpreter, the eager enumeration can be constructed as follows:

```
eager_enumerate({X|Y},Str) :-
        process(active,Str,(X<--Y)).
```

As described above, computational model can be written in Concurrent Prolog very easily, because of its high descriptive capability. This also shows that OR-parallelism can be implemented by AND-parallelism.

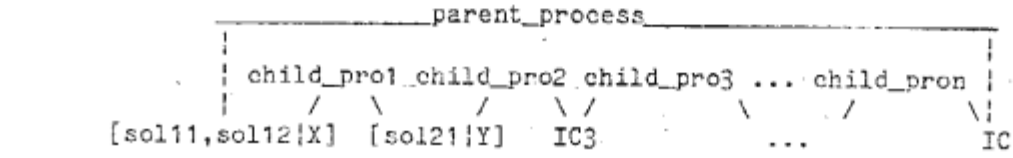
3.3 The Refined Version of The Eager Interpreter

The eager interpreter described above is the direct implementation of the computational model in section 3.1. This implementation utilizes a 'merge' network for message communication. The 'merge' predicate merges two streams nondeterministically to provide a characteristic of a channel where every child process can send a message to its parent independent of other child processes. However, the merge network has two drawbacks: it consumes a certain amount of the resources since a 'merge' is also a Concurrent Prolog process, and the message transfer takes relatively much time because the message is sent via more than one mergers. By eliminating the nondeterminacy of the message transfer, we can construct a more efficient eager interpreter without the merge network.

The basic idea of the new version is to use D-list and linearize the channel. In this version, an input channel of a parent process is the concatenation of the output channels of its child processes. To achieve this feature, a reduction of an active process is changed as follows:

```
process(active,OCs,OCe,(m<--a)) :-
              process(wait,OCs,OCe,IC1? ,IC,(m<--a)),
                  process(active,IC1,IC2,(a<--b1)),
                  process(active,IC2,IC3,(a<--b2)),
                                :
                  process(active,ICn,IC, (a<--bn)).
```

The first goal of the above clause specifies the parent process and the rest specifies its child processes. Each active process has both the output channel of its own (second argument) and its successor's output channel (third argument). After this clause is selected, each child process computes its solutions to attach them to its output channel.



The above figure shows the situation that the child_process1 produced two solutions, child_process2 produced one and the 'parent_process' has received one solution 'sol11'. When a child process puts all the solutions into its output channel, it concatenates its output channel and its successor's one. The parent process receives messages and executes its operation until the head pointer reaches the tail pointer of its input channel. When it terminates, a parent process concatenates its output channel and that of its successor because the parent process is a child process of the grand parent process. This method guarantees the ordering of solutions as well as OR-parallel execution.

The details of the interpreter is shown in [Hir 84]. The speed of this interpreter is about two times faster than the original one.

4. Lazy Enumeration

This section introduces the lazy interpreter, which is a modification of the interpreter described in section 3. This interpreter provides the lazy enumerate function.

The lazy interpreter produces a solution for a given goal sequence according to the demand from one of the other Concurrent Prolog processes. Then the interpreter suspends the computation until it receives a next demand. When the interpreter receives a "kill" message, it should release the resources and terminate itself. To implement the demand driven mechanism, the way of demand transfer and the execution suspension control should be established. These are achieved by bounded buffer communication method in Concurrent Prolog [Tak 83].

4.1 Bounded Buffer Communication

The interprocess communication is provided by the shared variables in Concurrent Prolog. Sending a message is instantiating a shared variable to the message. Since one instantiation corresponds to one message transfer, a new shared variable must be generated to continue the communication. According to Takeuchi, unbounded and bounded buffer communications can be supported in Concurrent Prolog.

The bounded buffer communication is achieved when the message receiver generates new shared variables. The following is a simple example of the bounded buffer communication with buffer length 2.

```
Goal :: integers(0,[X,Y|N?]), outstream([X,Y|N]\N).

integers(X,[X|M]) :- Y := X+1 | integers(Y,M).
outstream([X|M]\[P|Q?]) :- wait(X)&write(X) | outstream(M\Q).
```

'Integers' generates an integer stream. 'Outstream' outputs the elements of the stream. A symbol '\' is an infix operator which is used to write a head and a tail of D-list in one term. The call of 'integers' contains variables 'X,Y' which specify a buffer length of two. Process 'integers' can instantiate 'X' and 'Y' to 0 and 1 respectively, but cannot bind 2 to the variable 'N' because of its read only annotation. This process waits until the variable 'N' is bound. On the other hand, process 'outstream' waits until the 'integers' process binds the value because of the predicate 'wait(X)'. When the variable is bound to 0, 'outstream' writes the value and enters the recursive call. At this moment, a new variable 'P' is attached to the end of the communication channel because the tail of the channel (variable) is bound to '[P|Q?]' in the head of 'outstream' definition. This instantiation enables the 'integers' process to continue the processing.

The bounded buffer technique enables the receiver process to control the sender process. Attaching an uninstantiated variable to the tail of the communication channel corresponds to the demand transfer from a receiver process to a sender process. Lazy enumerator communicates with other Concurrent Prolog

processes via a bounded buffer as follows:

```
Goal :: lazy_enumerate({X|Y},[U|V?]), receiver([U|V]\V)
```

A 'kill' message to an enumerator is to close the communication channel by binding '[]' to the tail of a channel.

4.2 Lazy Interpreter Implementation

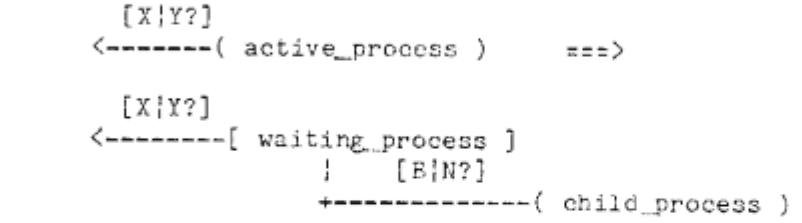
Lazy Pure Prolog interpreter is obtained by changing the characteristics of the eager one as follows:

(1) Replacing each communication channel from an unbounded buffer to a bounded buffer.
(2) Using a linearized channel instead of a merge network.
(3) Serializing process creations.
(4) Adding process operations for a kill demand.

Fig.2 shows the program of the lazy interpreter.

(p1) to (p4) define the behavior of active process. The second argument of an active process is its output channel and the third argument is its successor's output channel which is needed for linearizing a channel as mentioned in 3.3. When an active process is generated, its output channel is bound to '[B|N?]' or '[]'.

(p1) is a definition for manipulation a kill demand, which specify a termination of an active process. (p2) to (p4) correspond to the definitions in the eager interpreter. (p2) specifies the operation in the reduce mode where new child processes are generated and the active process changes its status to 'waiting' binding '[B|R?]' to its input channel. This binding is a demand for its child process. The following figure shows a demand transfer from a parent process to its child process.

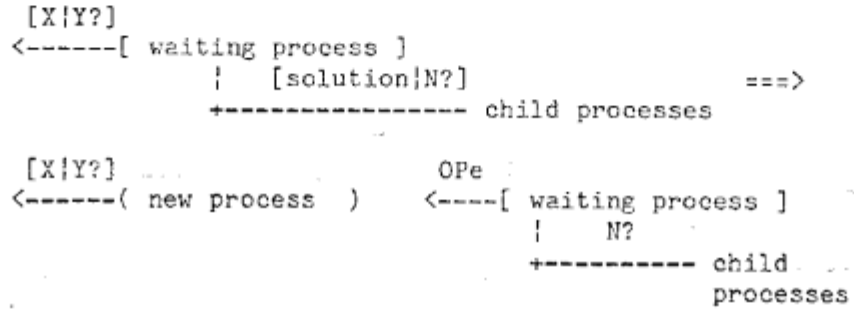


Predicate 'process_fork' executes 'clauses' and call 'forks' which is to generate child processes. This process generation is controlled by bounded buffer mechanism (the second argument of 'forks'). The second generation of child process is postponed until a next demand is detected. (f1) specifies the behavior of 'forks' when a demand is kill one. The serial-AND in (f3) specifies that a recursive 'forks' call should be tried after one process terminates. This is for only the efficient

implementation in DEC-20 Concurrent Prolog which doesn't have non-busy-wait mechanism.

(p3) and (p4) define that an active process terminates concatenating its output channel and its successor's (a unification of the second argument and the third one).

(p5) to (p7) defines the operation of waiting processes. (p5) which specify a process termination is for a kill demand. When the message sent via its input channel is 'end', a waiting process concatenates its output channel and its successor's and terminates itself. Message 'end' means that all child processes of a waiting process are terminated. (p7) specifies a waiting process operation when it received a solution. The configuration of an output channel of a waiting process and that of a new active process is as follows:



Output channel 'OPe' will be attached to the tail of the output channel of 'new process' when it terminates. Predicate 'transfer_demand' in (p7) transfers a demand, for example, the waiting process in the above figure instantiates 'N' to '[B'|N'?]' or '[]' according to a demand it receives.

Using the lazy interpreter, 'lazy_enumerate' is defined as follows:

```
lazy_enumerate({X|Goals],OPs) :-
      process(active,OPs,OPe,(X<--Goals)) & sendend(OPe?).

sendend([end_of_solution|_]).
sendend([]).
```

'Sendend' sends message 'end_of_solution' when a demand number exceeds the total element number of a set. The interface between 'lazy_enumerate' and other Concurrent Prolog process is a bounded buffer.

5. Discussion

To realize don't-know-nondeterminism, an environment of variable bindings must be maintained for multiple solutions. The interpreter described in this paper retains the environment by copying a clause, that is, a waiting processes copies its clause when it receives a message. A simple copying method has a drawbacks on both space and time efficiencies.

The space problem is that a simple method produces a whole copy of a given term which contains non-variable portions which can be shared. This problem is avoided by introducing 'rename' predicate which produces a copy of a term sharing ground term portions with its original term.

The time problem is that a copy operation should search whole part of a given term. This will increase a computation time of a waiting process according to the size of the terms it contains. One of the possible optimization methods for this problem is to determine the portion to be shared in compile time (either automatically or by giving declarations). A development of an efficient renaming method is one of the important topics for an implementation of the don't-know-nondeterminism.

6. Conclusion

This paper described an OR-parallel execution model for Pure Prolog and an implementation of enumerate function in Concurrent Prolog based on the model.

The computational model is based on multi-processing and interprocess communications. The model provides an eager Pure Prolog interpreter implemented in Concurrent Prolog. Also a lazy interpreter can be obtained easily by introducing a bounded buffer communication mechanism to the eager interpreter. The eager interpreter and the lazy interpreter provides eager and lazy enumerate functions to Concurrent Prolog, which are very important functions for a parallel logic programming.

This approach shows that both OR-parallel and AND-parallel execution of a logic program is achieved only by AND-parallel execution. This feature is very important because it decreases the complexity of the computer architecture and the amount of required hardware of a highly parallel machine.

Acknowledgement

We would like to thank Mr. Takeuchi for his valuable suggestions on the use of Concurrent Prolog and on the computational model, Mr. Sakai and Mr. Kondou and other researchers at the Second Research Laboratory who have joined discussions. We would also like to thank Dr. E.Shapiro and Dr.

K.Kahn for their useful advices about an OR-parallel execution of logic programs.

References

[Cla 83] Clark,K.L and Gregory,S:"PARLOG: A Parallel Logic Programming Language", Imperial College Research Report, (May 1983).

[Con 83] Conery,J,S: "The AND/OR Process Model for Parallel Interpretation of Logic Programs", Technical Report 204, University of California Irvine, (1983).

[Emd 82] Emden,M.H. and de Lucena Filho,G.J.: "Predicate Logic as a Language for Parallel Programming", in "LOGIC PROGRAMING", Clark,K.L. and Tarnlund,S.A. eds., Academic Press, (1982).

[Fur 84] Furukawa,K. and the Kernel Language Design Group: "Conceptual Specification of the Fifth Generation Kernel Language Version 1 (KL1)", to appear as an ICOT Technical Report, (1984).

[Har 81] Haridi,S. and Sahlin,D.: "Evaluation of Logic Programs based on Natural Deduction", The Royal Institute of Technology, TRITA-CS-8305, (1983).

[Hir 83] Hirakawa,H., Onai,R. and Furukawa,K.: "Implementing an OR- Parallel Optimizing Prolog System (POPS) in Concurrent Prolog", ICOT Technical Report, TR-020, (1983).

[Hir 84] Hirakawa,H., Chikayama,T. and Furukawa,K.: "Enumerations in Concurrent Prolog - Computational Model and Its Implementation", to be appear in ICOT Technical Report, (1984).

[Kow 79] Kowalski,R.: "Logic for Problem Solving", North Holland, New York (1979).

[Nit 82] Nitta,K., Matsumoto,Y. and Furukawa,K.:"Prolog Interpreter Based on Concurrent Programming", Proc. of 1st International Logic Programming Conference, pp.38-42, (1982).

[Sha 83] Shapiro,E.Y.: "A Subset of Concurrent Prolog and Its Interpreter", ICOT Technical Report TR-003, (1983).

[Sha 84] Shapiro,E.Y.: "The Bagel: A Systolic Concurrent Prolog Machine", to appear as an ICOT Technical Report, (1984).

[Tak 83] Takeuchi,A and Furukawa,K: "Interprocess Communication in Concurrent Prolog", Proc. of Logic Programming Workshop, (1983).

[War 81] Warren,D.H.: "Higher-Order Extensions to Prolog: Are They Needed?", D.A.I. Research Paper No. 154, (1981).

```
(p1) process(active,OutCh,Cls) :-
             reduce(Cls,NextGoal) |
                         process(wait,OutCh,InCh? ,Cls) ,
                         process_fork(InCh,NextGoal).

(p2) process(active,[Mess],Cls) :-
             terminate(Cls,Mess) | true.

(p3) process(active,[],Cls).

(p4) process(wait,OutCh,[Terminated_Goal|C1],Cls) :-
             newclause(Cls,Terminated_Goal,NewC) |
                     process(wait,OutCh1,C1? ,Cls) ,
                     merge(OutCh1?,OutCh2?,OutCh),
                     process(active,OutCh2,NewC).

(p5) process(wait,[],[],Cls).

     process_fork(OutCh,Goal) :-
             clauses(Goal,ClsList) |
                             forks(ClsList,OutCh).

     forks([],[]).
     forks([Clause|Rest],OutCh) :-
             process(active,OutCh1,_,Clause),
             merge(OutCh1?,OutCh2?,OutCh),
             forks(Rest,OutCh2).

     merge([S|X],Y,[S|Z]) :- merge(X?,Y,Z).
     merge(X,[S|Y],[S|Z]) :- merge(X,Y?,Z).
     merge([],Y,Y).
     merge(X,[],X).
```

Fig.1 Eager Interpreter for Pure Prolog

```
(p1) process(active,[],[],Cls).

(p2) process(active,OPs,OPe,Cls) :-
             reduce(Cls,NextGoal) |
                          process(wait,OPs,OPe,[B|N]\N,Cls) ,
                          process_fork(NextGoal,[B|N?]).

(p3) process(active,[Mess|R],R,Cls) :-
             terminate(Cls,Mess) | true.

(p4) process(active,OPs,OPs,Cls).

(p5) process(wait,[],[],[]\_,_).

(p6) process(wait,OPs,OPs,[Terminated_Goal|_]\_,Cls) :-
             wait(Terminated_Goal) &
             Terminated_Goal='$end$' | true.

(p6) process(wait,OPs,OPe1,[Terminated_Goal|C1]\R,Cls) :-
             wait(Terminated_Goal) &
             newclause(Cls,Terminated_Goal,NewC) |
                     process(active,OPs,OPe,NewC) &
                     transfer_demand(OPe?,R,Q) &
                     process(wait,(OPe?),OPe1,C1\Q,Cls).

     process_fork(Goal,OPs) :-
             clauses(Goal,ClsList) |
                          forks(ClsList,OPs).

(f1) forks(_,[]).
(f2) forks([],['$end$'|_]).
(f3) forks([Cls|Rest],OPs) :-
            wait(OPs) |
             process(active,OPs,OPe,Cls) &
             forks(Rest,OPe?).

     transfer_demand([_|_],[P|Q?],Q).
     transfer_demand([],_,[]).
```

Fig.2 Lazy Interpreter for Pure Prolog