数理解析研究所講究録
第 511 巻 1984 年 204-222

OR-Parallel Optimizing Prolog System : POPS
Its design and implementation in Concurrent Prolog

Hideki Hirakawa (平川秀樹), Rikio Onai (尾内理紀夫)
Kohichi Furukawa (古川康一)
Institute for New Generation Computer Technology

Abstract

This paper describes a computational model of an OR-Parallel Optimizing Prolog System(POPS) based on a graph-reduction mechanism and multi-processing. POPS has the following features; 1) Programs are executed in OR-Parallel, 2) The same sub-computations are shared, 3) Left recursive rules can be handled without entering infinite loop.
At present, POPS is implemented in Concurrent Prolog which supports AND-Parallel execution of subgoals and process synchronization mechanism.

1. Introduction

Programs based on the Horn logic (logic programs) represent two types of logical relations between predicates (AND-relations and OR-relations) and operational meanings such as computation sequencing. A logic program in a declarative sense may express an AND/OR tree. This implies a number of execution methods can be devised for logic programming language. In Prolog, a program is executed serially from top to bottom and from left to right. This corresponds to the top-down depthfirst search of an AND/OR tree. The other method is parallel execution of a logic programming language, which is basically equivalent to the approach of searching an AND/OR tree in parallel. There are two types of parallel-execution: AND-parallelism and OR-parallelism. AND-parallelism is introduced to describing concurrent processes as shown in, for example, Concurrent Prolog [Shapiro 83], while OR-parallelism corresponds to parallel execution for nondeterminism, a characteristic of the logic programming language.

We have designed a calcultion model that executes AND-relations serially and OR-relations in parallel, and implemented a system called OR-parallel Optimizing Prolog system (POPS) according to the model. In the POPS model, OR-parallelism is accomplished by multiple processes and communications among these processes. Also, the POPS model has adopted the concept of graph reduction [Turner 79] characterized by the concept of self-modification of a term and that of shared modification, graph reduction mechanism has the simularity to the execution process of logic programming languages. Modification sharing means the sharing of the same computation by multiple processes; this mechanism permits the POPS to bypass Prolog's inherent problem, re-execution of the same computation. In Prolog, executing a program that contains a derivation cycle enters an infinite loop. On the other hand the computation sharing mechanism of the POPS prevents the same derivation from being repeated.

At present, the POPS is implemented in Concurrent Prolog. Concurrent Prolog uses AND-parallelism to describe processes running concurrently, and performs interprocess communications via variables shared by concurrent processes. Concurrent Prolog can simplify the description of a multiprocess and interprocess communications; it also permits the processing system of the POPS to be created as a very compact program.

Section 2 of this paper discusses the graph reduction mechanism and computation models of POPS. Section 3 describes how the POPS is implemented in Concurrent Prolog. Finally, Section 4 discusses enhancements to the POPS and its implementation on a multiprocessor, and gives a useful application for the POPS, ie, syntax analysis in natural language processing.

2. Basic Computation Model

This section describes the graph reduction mechanism and the computation model in POPS.

2.1 Graph Reduction Mechanism

A Prolog program is executed by repeatedly generating resolvants from parent clauses, in other words, in the execution process of a Prolog program, goals modify themselves by applying reduction rules [Turner 79]. This is a definition of reduction; we find simularity between the execution process of a Prolog program and the reduction mechanism. There are several types of reductions. The graph reduction mechanism is defined as: "The evaluation of an expression is shared by pointers. Therefore, an expression is evaluated only once and the result is reported to all processes which share the expression." Applying the graph reduction mechanism to the execution process of a Prolog program permits the evaluation of a term to be shared by pointers, thereby avoiding repetitive computations of the term and detecting a derivation cycle. As described in detail in a later part of this paper, in POPS, a term is detected on the 'board' and the evaluation of the term is shared by multiple processes through communication channels of Concurrent Prolog. The reduction results of the term are reported through channels to all the processes waiting for the results.

2.2 Pure Prolog and Its Interpretation

The language targeted by the POPS is Pure Prolog generally consisting of a group of definite clauses in the following format:

(a) H <-- G1, G2,...,Gn (n>= 1)
(b) H <-- true

H and Gi (1<= i <= n) are equivalent to Prolog's literals and 'true' is a special literal representing "true." Like Prolog, (b) may be described by omitting "<-- true." Pure Prolog does not include the execution control operator 'cut' and evaluable predicates such as 'not.' POPS executes Pure Prolog serially for AND connections of subgoals and in parallel for OR connections of subgoals. For example, assume (a) and (b) above made up a program. In this case, literals forming the right side of (a) cannot simultaneously be computed, (ie, Gn+1 cannot be computed before Gn is computed), while (a) and (b) can be evaluated simultaneously.

2.3 System Components

As shown in Fig. 1, the POPS is made up of four components: processes, channels, a board, and a Horn Clause Database(HDB).

```
             +---------+-------------+      generator
             |         |  channel    |      process
             |         V             V         |
process1  process2  process3 ... processn      |
   |                                 |         +----------+
   +----------------------+          |         |          |
   |access   +------------|---------+          |          |
   |         |            |          |         V          V
+-------------+       +-------------+       consumer   consumer
|Horn Clause  |       |    Board    |       process    process
|Data Base    |       |             |
+-------------+       +-------------+
```

Fig. 1 POPS Configuration   Fig. 2 Process/Channel

A process plays a key role in computations. Any number of processes can be generated in a system. A process corresponds to a clause being computed and includes a clause, such as H<--G1,G2. The computation environment is maintained by instantiating an actual value to a variable in a clause. There are two types of processes, the active and waiting processes. The waiting process waits until it receives data from another process.

The channel is a communication path between processes and is dynamically generated during computation. Data transferred through a channel is called a message. A message is passed from a process named a "generator" to a process named "consumer." The distinction between a generator and a consumer is relative, and a single process can simultaneously play two roles. One generator process can simultaneously send a message to multiple consumer processes via a channel (See Fig. 2). Similarly, one consumer process can be connected to multiple generators.

The board is a storage area accessed by processes, and stores all the subgoals currently being computed as well as channels through which the computation results (messages) for these subgoals are transmitted. For example, suppose that, when the subgoal a(X) is invoked, a channel, C, is generated for the subgoal. In this case, a(X) and C are saved as a pair on the board. The channel C is then used to pass solutions to the subgoal a(X), for example, a(1), a(2),... The data pair consisting of a channel and a subgoal (term) is called a channel head pair, which is described in the following format:

Channel+Head

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

2.4 Execution Mechanism

In POPS, computations progress while multiple processes are exchanging messages. This subsection provides a more-detailed description of the process, shows simple examples, and presents the execution mechanism of the POPS. To simplify the discussion, all predicates are assumed to have no arguments. For arguments, see Subsection 2.8.

A process is defined by five components Status, Head, Goals, Input-Channel, and Output-Channel--and 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 before receiving 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 shows the predicates to be computed to compute the Head. For example, if the HDB includes "a<--b,c" there may be a process as follows:

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 earlier described. A process appears as a consumer for the Input-Channel, while it functions as a generator for the Output-Channel. Next, we will define the operation of a process.

(A) Active process

The operation of an active process is either derivation or termination. In derivation mode, head of the subgoal is expanded using inference rules in HDB; the active process are maintained after the derivation 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 derivation 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 references or registers to the board for the channel head pair I+P (*1).

When referencing the board:
The process changes its status to 'waiting.'

When registering to the board:
The process performs a P-related selection in 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 termination mode

There are two types of termination: success or failure termination. Success termination occurs when derivation reaches true, while failure termination occurs when selection for the HDB 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.

(*1) Reference or Register : The board stores channel head pairs C1+H1, C2+H2,..., Cm+Hm. When a desired channel head pair, C+H, has already been saved on the board (ie, H and Hi are equivalent where 1<=i<=m), C is identified with Ci. This is reference. Otherwise, C+H is added to the board (register). We define equivalent as follows;
Term T1 and T2 are equivalent if there exists a substitution S such that T1 S = T2 where S={X1/Y1,... ,Xn/Yn} , Yj (1<=j<=n) is variable and k is not equal to m implies Yk is not equal to Ym.

(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, G' with its head element removed. When G' contains only P', However, NewG contains 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 mechanism terminates, when all the existing processes enter waiting mode. This termination condition is called "deadlock termination."

2.5 Computation Examples

This subsection presents a simple example to show the way the POPS is executed. In the following figures, the active process p, waiting process q and 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 in the format H<--G. The status of the board is enclosed by { }.

Assume that the HDB is given as follows:

```
HDB = {A<--B,C  B<--D  D<--true  C<--true}
```

To compute the predicate A, the following process is generated as the initial process:

```
<--c0-- (A<--A)p0                Brd={}
```

"A<--A" semantically equals"<--A." Head A shows the message the initial process should return. A message output through c0 is the solution. Since p0 is an active process and has the Goals A, it performs selection to generate a new process, p1, and then changes the status from active to waiting. At this time, p0 registers the channel head pair to the board.

```
<--c0-- [A<--A]p0 <-c1- (A<--B,C)p1       Brd={c1+A}
```

Similarly, p1 works to generate p2,

```
                                    Brd={c1+A,c2+B}

<-c0- [A<--A]p0 <-c1- [A<--B,C]p1 <-c2- (B<--D)p2
```

and p2 works to generate p3.

```
                                        Brd={c1+A,c2+B,c3+D}

<-- [A<--A] <-- [A<--B,C] <-- [B<--D]p2 <-c3- (D<--true)p3
```

p3 has 'true' as its Goals and is equal to an active process in termination mode, it sends message D through channel c3 to p2, which in turn generates a new process, p4. Message D will be maintained in a channel head pair on the board after the process p3 has been deleted.

```
                                        Brd={c1+A,c2+B,c3+D}

<-- [A<--A] <-- [A<--B,C]p1 <-+- [B<--D]p2
                              |c2
                              +- (B<--true)p4
```

Since p4 is also a process in termination mode, it sends a message B via channel c2 to p1, which in turn generates an active process, p5.

```
                                        Brd={c1+A,c2+B,c3+D}

<-- [A<--A] <-+- [A<--B,C] <--- [B<--D]p2
              |c1
              +- (A<--C)p5
```

Subsequent computations proceed in the same manner.

```
                                    Brd={c1+A,c2+B,c3+D,c4+C}

<-- [A<--A] <-+- [A<--B,C] <--- [B<--D]
              |c1
              +- [A<--C] <-c4- (C<--true)p6

                                    Brd={c1+A,c2+B,c3+D,c4+C}

<-- [A<--A]p0 <-+- [A<--B,C] <--- [B<--D]
                |c1
                +- [A<--C]
                |
                +- (A<--true)p7
```

p7 sends message A to p0, which in turn produces p8.

```
                                        Brd={c1+A,c2+B,c3+D,c4+C}

<-+- [A<--A] <------+- [A<--B,C] <--- [B<--D]
  |c0                |c1
  +- (A<--true)p8    +- [A<--C]
```

p8 terminates and sends the message A via the channel c0. This means that a solution to A is obtained. The following figure shows the entire processing situation at this point:

```
                                        Brd={c1+A,c2+B,c3+D,c4+C}

<-- [A<--A] <-+- [A<--B,C] <--- [B<--D]
              |
              +- [A<--C]
```

Since all the existing processes are in waiting mode, the deadlock termination condition is satisfied to stop the computation. In this example, there is always only one active process throughout the computation, because no two predicates have the OR-parallel relation. If OR-parallelism is present among predicates, multiple active processes will simultaneously appear.

2.6 Derivation Cycle

When there is a derivation path containing a cycle, a normal Prolog program enters an infinite loop. By contrast, in the POPS, a predicate, once having been computed, is saved on the board; when the same predicate reappears in a derivation process, board reference is performed and no new process is generated. Therefore, instead of entering an infinite loop, the whole system detects deadlock, resulting in deadlock termination. For example, consider the Horn Database, HDB,

```
        {A<--B  B<--A...}
```

containing a cycle A -> B -> A ... In this case, the computation progresses as follows:

```
(1) <--- (A<--A)                                Brd={}

(2) <--- [A<--A] <-c1- (A<--B)p1                Brd={c1+A}

(3) <--- [A<--A] <--- [A<--B] <-c2- (B<--A)p2   Brd={c1+A,
                                                     c2+B}
```

```
(4) <--- [A<--A] <-+- [A<--B] <---- [B<--A]       Brd={c1+A,
                   |c1                |                c2+B}
                   +------------------+
```

In the course from (3) to (4), p2 references the board for A, since c1+A has already been saved on the board; c1 is used as the Input-channel of p2. (4) satisfies the deadlock condition terminating the computation.

When there are infinite number of solutions, however, the deadlock condition is naturally not satisfied and the computation continues endlessly. For example, if the HDB in the above example includes "B<--true," an infinite number of derivation paths, or solutions, are possible, and the system will not terminate. Also, when a goal contains a variable, it is possible to write a program that changes the condition every time a goal is invoked; as a result, no termination occurs. For example, "a(X)<--a([s|X])" does not terminate. Although the POPS and other systems cannot detect such infinite loop, this kind of program will be extremely special.

2.7 Computation Sharing

The board not only can assure that a program containing clauses that form a cycle will terminate computation, but also prevents the same computation from being repeated. Consider the following program:

```
HDB={...A<--B,C  D<--B,E...}
```

A and D have the same subgoal, B. Assume that, in the course of calculating a predicate, both A and D are computed. In the normal top-down-serial strategy, the computation of B is carried out twice; for A and D. By contrast, the POPS computes B only once. The computation process in the POPS is schematically shown below. When two active processes, p1 and p2, have A and D as the head element of their Goals, p1 and p2 generate p3 and p4, respectively, as follows:

```
[W<--A,U]p1 <-c1- (A<--B,C)p3        Brd={..,c1+A,c2+D,..}

[X<--D,Y]p2 <-c2- (D<--B,E)p4
```

Both p3 and p4 try to save a channel head pair to the Board. When p3 first succeeds in saving, the entire condition becomes as follows:

```
                                 Brd={..,c1+A,c2+D,..,c3+B}
[W<--A,U]p1 <-c1- [A<--B,C]p3  <-c3- (B<--...)p5

[X<--D,Y]p2 <-c2- (D<--B,E)p4
```

Then, instead of saving to the board, p4 references the board; as a result, p4 is connected to p5 via c3.

```
                                          Brd={...,c1+A,c2+D,...,c3+B}
[W<--A,U]p1 <--- [A<--B,C]p3  <-+- (B<--...)p5
                                 |c3
[X<--D,Y]p2 <--- [D<--B,E]p4  <-+
```

Then, message B output from p5 is directed to both p3 and p4, causing each process to generate a new corresponding process. Even if message B is sent from p5 before p4 and p5 are connect via c3, p4 will get the message as soon as the connection is set up, because the message remains in c3. Thus, the computation of B is carried out by p5 alone, and the result is shared by p3 and p4.

2.8 Manipulation of Variables

To execute a Prolog program containing variables on an OR-parallel basis, it is necessary to ensure the independency of variables in unifying a subgoal with OR relation clauses. For example, suppose that subgoal b(X,10) is to be computed in the following program:

```
(1)  a(X)<-- b(X,10), c(X)
(2)  b(Y,U)<-- d(Y,U)
(3)  b(Z,V)<-- e(Z,V)
```

b(X,10) is derived using (2) and (3), and unified with their head (ie, X=Y and 10=U, as well as X=Z and 10=V). Unification of variables through variable binding must be ensured in the similar manner to identify of variables with the same name within the same clause. In addition variables Y and Z in OR relation clauses must be entirely independent with each other. POPS satisfies these requirements with copy (*2) and unification functions. Thus when a subgoal is derived with OR relation clauses, POPS generates a copy of the current subgoal to perform derivation, having received a message showing the computation result, having POPS produces a copy of the Head and Goals that include the subgoal, and unifies the subgoal in the copy with the message. For the program example above, the computation proceeds as follows:

```
(a)  <--- (a(X)<--b(X,10),c(X))p1

(b)  <--- [a(X)<--b(X,10),c(X)] <-+- (b(Y,10)<--d(Y,10))p2
                                   |c1
                                   +- (b(Z,10)<--e(Z,10))p3
```

If p2 carries out a computation faster than p3 and sends the message b(1,10) to pl through cl, then pl generates a copy of the Head and Goals, "a(X')<--b(X',10),c(X')," and unifies the head element of the Goals b(X',10), with message b(1,10) to generate a new process, p4.

```
(c)  <-+- [a(X)<--b(X,10),c(X)] <-+- [b(Y,10)<--d(Y,10)]p2
       |                          |c1
       +- (a(1)<--c(1))p4         +- (b(Z,10)<--e(Z,10))p3
```

Thus, POPS can ensure the identity of variables in the same clause, variable binding during derivation, and the independency between OR relation clauses, by executing the copy and unification function when necessary.

3. POPS in Concurrent Prolog

This section describes the POPS implemented in Concurrent Prolog [shapiro 83].

3.1 Concurrent Prolog

Concurrent Prolog adopts AND-parallelism to describe concurrent processes and OR-parallelism to describe nondeterministic actions of processes. While Prolog implements OR-parallelism with backtracking, in Concurrent Prolog, once a clause is selected, the choice of other clauses is ignored. Concurrent Prolog uses variables shared by processes running in concurrent for interprocess communications. (For further details of Concurrent Prolog, refer to [Takeuchi 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, and, if 'X' is a woman, her son.

(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 "|" and "?"appearing in the program are inherent to Concurrent Prolog. The comma "," in Concurrent Prolog has the different meaning from that in Prolog. 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 another alternative clause. "," denotes parallel-AND relationship and is a logical equivalent for the ordinary AND. In an operational sense, however, it differs from the ordinary AND; "P,Q" means that P and Q

(*2) A copy of a structure (for example, S' for structure S) is generated by replacing all variables in S with different variables. If a variable is equivalent to another variable in the original term, their coresponding variables in the copy will maintain the relation.

are processed in parallel. Concurrent Prolog uses a symbol "&" to express serial-AND relation. "?" means "a variable attached with '?' should not be instantiated to a non-variable term." In the case of (a), "?" indicates that Y is instantiated by the 'daughter' process and the 'output' process uses Y for reading purpose only. "?" is called 'read-only annotation,' and a ?-attached variable is referred to as a 'read-only variable.' The read-only annotation function permits shared-variable-based communications between two concurrent processes (interprocess communications). 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 Describing a POPS Process in Concurrent Prolog

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

```
process(Status,OutputChannel+InputChannel+(Head<--Goals),Board)
```

Generation of a process is performed by parallel ANDs as with the case "process :- process1,process2," and deletion of a process is expressed by termination of the process, "process :- true." A channel is implemented with variables shared by concurrent processes of Concurrent Prolog, and process synchronization is achieved with read-only annotation. The board that is accessed by all processes is implemented with the variable 'Board' shared by all the processes. The 'Board' can be configured in any manner as long as it provides the function of referencing or registering channel head pairs. Our systems has configured it using a binary tree [Warren 80]. 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.

The descriptions of POPS processes in Concurrent Prolog are shown below.

```
(c1) process(active,OutCh+_+Cls,Brd) :-
             derivationp(Cls,NextGoal,InCh,Brd,Alt_or_not_yet) |
                     process(wait,OutCh+InCh? +Cls,Brd) ,
                     process_fork(Alt_or_not_yet,InCh+NextGoal,Brd).

(c2) process(active,OutCh+_+Cls,Brd) :-
         terminatep(Cls,Mess) |
              sendmess(Mess,OutCh).

(c3) process(active,_+_+Cls,Brd) :-
         call( (write('process killed '),
              portray(Cls),
              nl)).
```

```
(c4) process(wait,OutCh+[Terminated_Goal|C1]+Cls,Brd) :-
          newcls(Cls,Terminated_Goal,NewC) |
                    process(wait,OutCh+C1? +Cls,Brd) ,
                    process(active,OutCh+_+NewC,Brd).
```

(cl) to (c3) define the behavior of active processes, while (c4) defines that of a waiting process.

(cl) performs derivation. The predicate 'derivationp' checks whether the head element of the Goals is defined in the HDB; if so, it then references or registers to the board for the element. In referencing the board, 'already' is instantiated to 'Alr_or_not_yet,' while, in saving to the board, 'not_yet' is instantiated to it. This information is necessary for the predicate 'process_fork.' When the head element is not found in HDB, the predicate 'derivationp' fails. When the guard portion of (cl) 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' performs selection for a copy of the head element of the Goals of the original process, and generates a new active process for each of newlyfetched clauses. When 'Alr_or_not_yet' has been instantiated to 'already,' however, 'process_fork' generates no new processes, because such processes are already present.

(c2) corresponds to a process in termination mode. The predicate 'terminatep' checks that 'Cls' is in the format "X<-- true." The predicate 'sendmess' sends a message to the OutCh. This is done by adding a message to the tail of D-List, OutCh. Then, the process deletes itself.

(c3) shows the operation of active processes unable to be described by (cl) and (c2), ie, the operation in which further derivation become impossible. (c3) outputs the message 'process killed,' then deletes itself.

(c4) defines the operation of a waiting process. In (c4), the Input-Chnnel is a read-only variable; when a value is instantiated to the variable (ie, when a message is received), the process starts operating. The predicate 'newcls' generates copies, H and G, from the Head and Goals of a waiting process, respectively, and unifies the head element of G with the message. Then, using G', G with the head element removed, it generates the Head and Goals of a process to be generated. In the goal portion of the program, 'newcls' forks a copy of the original process and a new active process.

An execution example of the program is shown in the Appendix. As described above, POPS can be written in Concurrent Prolog very easily, because of the high descriptive ability of Concurrent Prolog. Also this means OR-parallelism can be implemented by AND-parallelism.

4. Discussion

The POPS provides a mechanism to process OR-parallelism of Prolog in parallel. When a solution to a program has multiple meanings, POPS tries to find the entire set of the solution. For example, in the case of "man(X)," X means all humans. In this sense, variables in POPS are universal. By contrast, existential variables are also possible. For example, when knowledge "if a person has a child, that person has got married," is expressed by the clause "married(X)<--has_child(X)," the clause outputs a single solution 'true,' irrespective of the number of children that person has, as long as he or she has at least one child. To handle knowledge like this, we can introduce two different predicates - universal and existential predicates. The universal predicate outputs the entire set of solutions, while the existential predicate outputs only a single solution in the solution set. In a program, we separate these two predicates by adding an attribute tag to a predicate or introducing a meta definition predicate.

In the computation mechanism of POPS, the two predicates can be distinguished by introducing the concept of 'channel close' and sending the message of an existential predicate through a channel while simultaneously closing the channel to disable subsequent transmission of messages. In Concurrent Prolog, this can be done by instantiating '[]' to the variable in the D-list (Channel).

POPS's advantages are particularly suitable to computations in multivocal fields; for example, syntactic analysis in natural language processing. In the syntactic analysis of a natural language, Context Free Grammar rules corresponding to Horn clauses and are embedded in Prolog in very simple, highly descriptive form as shown in DCG [Pereira 80], for example. Characteristics of natural language processing, however, require the following problems must be solved:

(1) Repetition of computation

Syntactic analysis of a natural language often produces multiple solutions and dozens of different syntactic-analysis results are obtained for an ordinary sentence. The multivocal characteristic such as this is handled by the backtracking function of Prolog. In general, backtracking based strategy lacks efficiency because it repeats the same computation. If there are two rules with the same portion, for example, "vp-->vt,np" and "vp-->vt,np,pp," backtracking computes the overlapped portion twice. Based on the backtracking strategy, the parsing time for a sentence increases in exponential order according to its length N.

(2) Derivation cycle

Natural languages have a recursive structure by nature. Reflecting this characteristic, their syntax rules tend to contain a cycle and/or left recursive rules. For example, the rule for conjunctions may be expressed by "np-->np, and, np." Interpreting rules like this by Prolog's top-down and depth-first strategy causes the processing to enter an infinite loop; the computation never stops.

POPS offers a strategy to solve these problems. We can execute a DCG program with POPS then, unlike back-track-based system, the same computation is never repeated. As described in Section 2, the function to check a derivation cycle enables the POPS to handle a grammar containing a cycle and/or left recursive rules. POPS-based execution process of a DCG program corresponds to the (pseudo) parallel parsing strategy, and is equivalent to the execution process of Active Chart Parser [Kay 80]. Active Chart Parser can be also implemented in Concurrent Prolog [Hirakawa 83].

In a system where POPS is implemented with a multi-processor, trying to simultaneously reference all terms on the board or saving a number of terms to the board causes a bottleneck of access contention. Meanwhile, the referencing-or saving-to-the-board approach, or the graph reduction mechanism, has a merit of avoiding the repetition of the same computation. Referencing or saving to the board become effective, if the following relationship holds:

$$T1 * N > T1 + T2 * N$$

Roughly speaking

$$T1 > T2$$

where T1 is the time required to process a shared term, T2 is the time required to reference or save to the board (to be exact, referencing time may differ from saving time), and N is the number of processes sharing the term. When a term has been referenced by only one process, processing the term requires an additional time of T2. Therefore, when referencing or saving to the board, it is better to choose terms which are about to be shared and can satisfy the processing relation T1 > T2 than to select all the terms. executing a Prolog program on a partial graph-reduction basis should be considered. Also, a key to natural language processing is investigation of terms which are likely to be referenced and can satisfy T1 >T2.

5. Conclusion

This paper has described the POPS, a Pure Prolog-based processing system adopting message passing and multiple processes. With a mechanism to implement OR-parallelism of

Prolog, the POPS provides advantages of avoiding repetition of the same computation and handling a program that contains a derivation cycle. POPS has been implemented in Concurrent Prolog that has AND-parallelism, interprocess communications, process synchronization, and other capabilities.

Acknowledgement

We would like to thank Mr. Takeuchi for his valuable suggestions on Concurrent Prolog and the POPS, Mr. Sakai for his discussions on argument handling, and other researchers at the Second Research Laboratory who have joined discussions. We would also like to thank Dr. Chikayama for his useful advice about preparation of this paper.

Reference

[Hirakawa 83] Hirakawa,H.: "Chart Parsing in Concurrent Prolog", ICOT Technical Report TR-008, (1983)

[Kay 80] Kay,M: "Algorithm Schemata and Data Structures in Syntactic Processing", Xerox Technical Report, 1980

[Pereira 80] Pereira,F. and Warren,D.H.: "Definite Clause Grammar for Language Analysis - Survey of the Formalism and a Comparison with Augmented Transition Networks", Artificial Intelligence, 13, (1980)

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

[Takeuchi 83] Takeuchi,A: "Let's Talk Concurrent Prolog", ICOT Technical Memo TM-0008,(1983)

[Turner 79] D.A.Turner: "A New Implementation Technique for Applicative Languages", software-practice and experiance, No.1, vol.9 (1979)

[Warren 80] Warren,D.H.: "Logic Programming and Compiler Writing", DAI Research Paper No.128

APPENDIX A

```
%
%     OR-Parallel and Optimizing Prolog System in Concurrent Prolog
%


:- op(1200,xfx,'<--').

process(active,OutCh+_+Cls,Brd) :-
        derivationp(Cls,NextGoal,InCh,Brd,Alt_or_not_yet) |
                    process(wait,OutCh+InCh? +Cls,Brd) ,
                    process_fork(Alt_or_not_yet,InCh+NextGoal,Brd).

process(active,OutCh+_+Cls,Brd) :-
        terminatep(Cls,Mess) |
                sendmess(Mess,OutCh).

process(active,_+_+Cls,Brd) :-
        call( (write('process killed  '),
             portray(Cls),
             nl) ).

process(wait,OutCh+[Terminated_Goal|C1]+Cls,Brd) :-
        newcls(Cls,Terminated_Goal,NewC) |
                process(wait,OutCh+C1? +Cls,Brd) ,
                process(active,OutCh+_+NewC,Brd).


process_fork(already,_,_).
process_fork(not_yet,OutCh+Goal,Brd) :-
        select(Goal,ClsList) |
                        forks(ClsList,OutCh,Brd).

forks([],_,_,_).
forks([Clause|Rest],OutCh,Brd) :-
        process(active,OutCh+_+Clause,Brd) , forks(Rest,OutCh,Brd).

%
%      TOP LEVEL PREDICATES
%

psolve(X) :-
                process(active,OutCh+InCh+(X<--X),Brd) ,
                write_msg(OutCh?).

do(X) :- statistics(runtime,T1),
         solve(psolve(X),Res,N),nl,nl,
         statistics(runtime,[_,T2]),
         write('Execution Time = '),write(T2),write(' ms'),nl,
         write('Computation '),portray(Res),
         nl,write('Step = '),portray(N).

write_msg([X|Rest]) :- call((write('Message = '),portray(X),nl))
                      | write_msg(Rest?).

%
%     The Compiling Part
```

```
%

:- public newcls/3.
:- mode newcls(+,+,-).
newcls(Cls,Terminated,New) :-
                copy(Cls+Terminated,G+Gt),
                (G=(Hd<--Gt,Rest),!;
                 G=(Hd<--Gt),Rest=true),
                New=(Hd<--Rest).

:- public terminatep/2.
:- mode terminatep(+,+).
terminatep(Cls,Mess) :-
                Cls=(Mess<--true).

:- public derivationp/5.
:- mode derivationp(+,-,?,?,-).
derivationp(Cls,NextGoal,InCh,Brd,Alt_or_not_yet) :-
              reducep(Cls,NextGoal),
              copy(NextGoal,NxG),numbervars(NxG,0,_),
              ref_reg(InCh+NxG,Brd,Alt_or_not_yet).

:- public reducep/2.
:- mode reducep(+,-).
reducep(Cls,CurHead) :-
        (Cls=(_<--(CurHead,_)),!;
         Cls=(_<--CurHead)),
        \+(\+(ax((CurHead<--_)))), !.

:- public ref_reg/3.
:- mode ref_reg(+,?,-).
ref_reg(ChGoal,B_Tree,S) :- var(B_Tree),!,
        B_Tree=bt(ChGoal,_,_),
        S=not_yet.
ref_reg(Ch+Goal,bt(Ch+Goal,_,_),already) :- !.
ref_reg(Ch+Goal,bt(_+Goal1,Before,_),S) :-
        Goal @< Goal1,
        ref_reg(Ch+Goal,Before,S).
ref_reg(Ch+Goal,bt(_+Goal1,_,After),S) :-
        Goal @> Goal1,
        ref_reg(Ch+Goal,After,S).

:- public select/2.
:- mode select(+,-).
select(Head,ClsList) :-
                bagof0((Head<--X),Head^(ax((Head<--X))),ClsList).

:- public sendmess/2.
:- mode sendmess(+,?).
sendmess(Mess,Channel) :- var(Channel), !, Channel=[Mess|_].
sendmess(Mess,[_|C]) :- sendmess(Mess,C).
```

APPENDIX B

Some simple programs and the execution examples.

(1) Programs

```
ap([],X,X).
ap([U|X],Y,[U|Z]) :- ap(X,Y,Z).

is_a(X,Z) :- is_a(X,Y),is_a(Y,Z).
is_a(doctor,human).
is_a(researcher,human).
is_a(human,animate).
is_a(animate,living_thing).
```

(2) Execution of the programs

```
| ?- do(ap([a,b],[c],K)).              % 'ap' is append
Message = ap([a,b],[c],[a,b,c])

Execution Time = 1060 ms
Computation deadlock
Step = 5
K = _52

| ?- do(ap(K,L,[a,b,c])).
Message = ap([],[a,b,c],[a,b,c])
Message = ap([a],[b,c],[a,b,c])
Message = ap([a,b],[c],[a,b,c])
Message = ap([a,b,c],[],[a,b,c])


Execution Time = 2538 ms
Computation deadlock
Step = 6
K = _31,
L = _52

yes
| ?- do(is_a(X,Y)).                     % 'is_a' is transitive relation,
Message = is_a(animate,living_thing)    % so Prolog will loop indefinitely.
Message = is_a(human,animate)
Message = is_a(researcher,human)
Message = is_a(doctor,human)
Message = is_a(doctor,animate)
Message = is_a(doctor,living_thing)
Message = is_a(researcher,animate)
Message = is_a(human,living_thing)
Message = is_a(doctor,living_thing)
Message = is_a(researcher,living_thing)
Message = is_a(researcher,living_thing)


Execution Time = 5863 ms
Computation deadlock
Step = 6
X = _31,
```