First Australia-Japan Joint Symposium on Natural Language Proces

Proceeding of the Symposium
27-29 November 1989
The University of Melbourne
Parkville, Victoria, Australia

Parallel Generalized LR Parser (PGLR) based on Logic Programming

Hozumi TANAKA
Tokyo Institute of Technology

Hiroaki NUMAZAKI
Tokyo Institute of Technology

Abstract

The Tomita's parsing algorithm[Tomita 86] which adapted LR parsing algorithm to context free grammars makes use of a breadth first strategy to handle conflicts occured in a LR parsing table. As the breadth first strategy has a good compatibility with parallel processing, we have developed a parallel generalized LR parser PGLR that has been implemented in GHC. GHC is a concurrent logic programming language developed by Japanese 5th generation computer project. As PGLR uses Tomita's Graph Structured Stack (GSS), the stack top elements with the same states have to be merged. The simplest way to implement GSS is to use side effects, but side effects are not desirable for us to simulate GSS. The reason is that we are going to implement GSS in a framework of a logic programming language, GHC. The details of our implementation of GSS, which does not make use of side effects, will be shown in section 3.

1 Introduction

As the length of a sentence becomes longer, the number of parsing trees increases and it will take a lot of time to parse a sentence. In order to achieve fast parsing, we should look for a parallel parsing system based on the most efficient and general parsing algorithms. It is well known that LR parser is the most efficient parser, because it runs deterministicly for any LR grammar which is a subset of context free grammars. Unfortunately, LR grammar is too weak to parse most sentences of natural languages. When we apply LR parsing algorithm to a context free grammar, it is a usual case of having conflicts in a LR parsing table. So we need to generalize the LR parsing algorithm which can handle these conflicts. To resolve such conflicts, there are two kinds of strategies: (1) a depth first strategy, and (2) a breadth first strategy. Nilsson[Nilsson 86] has adopted a depth first strategy and Tomita[Tomita 86] a breadth first strategy. Using these strategies, we will be able to handle a context free grammar in the framework of the LR parsing algorithm. Such a parser is called a generalized LR parser. It is easy for us to simulate the breadth first strategy through parallel processing technique, and we have developed a parallel generalized LR parser PGLR based on a breadth first strategy.

To avoid recomputations, Tomita has devised a Graph Structured Stack (GSS) in which stack top elements with the same states will be merged. The simplest way to implement GSS is to use side effects, but side effects are not desirable for us to simulate GSS. The reason is that we are going to implement GSS in a logic programming language called GHC that has developed by Japanese 5th generation computer project.

After we will give a brief introduction of LR parsing algorithm in section 2, we will describe PGLR parser in section 3. One of the most significant feature of PGLR is to regard each entry of a LR parsing table as a process. The process keeps a stack on which shift and reduce operations are conducted. If the process discovers a conflict in a LR parsing table, the process copies its own stack and sends it to subprocesses which will perform reduce operations in the conflict. After finishing the reduce operations, a merge process will be activated if necessary. In order to understand PGLR parser, we will give an example of a trace of parsing in subsection 3.6. In section 4, we will explain some results of experiments.

(1)	S	→	NP, VP.
(2)	S	→	S, PP.
(3)	NP	→	NP, RELC.
(4)	NP	→	NP, PP.
(5)	NP	→	det, noun.
(6)	NP	→	noun.
(7)	NP	→	pron.
(8)	VP	→	v, NP.
(9)	RELC	→	relp, VP.
(10)	PP	→	p, NP.

fig.1: Ambiguous English grammar

2 Generalized LR Parsing algorithm

The execution of a generalized LR parser is controled by a LR parsing table which is generated from grammar rules given in advance. Fig.1 shows an ambiguous English grammar and fig.2 a LR parsing table generated from fig.1. The LR parsing table is divided into two parts, an action table and a goto table.

The left-hand side of the table is called 'action table', the entry of which is determined by a pair of generalized LR parser's state (the row of the table) and a look-ahead preterminal(the column of the table) of an input sentence. There are two kinds of operations, a shift and a reduce operations. Some entries of the LR table contains more than two operations and thus have conflicts. In such a case, a parser should conduct more than two operations simultaneously.

The symbol 'sh N' in some entries means that generalized LR parser has to push a look-ahead preterminal on the LR stack and go to 'state N'. The symbol 're N' means that generalized LR parser has to reduce several topmost elements on the stack using a rule numbered 'N'. The symbol 'acc' means that generalized LR parser ends with success of parsing. If an entry doesn't contain any operation, generalized LR parser recognizes an error.

The right-hand side of the table is called a 'goto table' which decides a state that the parser should enter after every reduce operation. The LR table shown in fig.2 has 4 conflicts at the state 14 (row number 14) and state 16 for the column of 'p' and 'relp'. Each of four entries, which have a conflict, contains two operations, a shift and a reduce operation. Such a conflict is called a 'shift-reduce conflict'. When a parser encounters a conflict, it cannot determine which operation should be carried out first. In PGLR explained in the next section, conflicts will be resolved using parallel processing technique and we do not care the order of the operations in a conflict.

3 Implementation of PGLR

PGLR is implemented in GHC that is a concurrent logic programming language developed by Japanese 5th generation computer project. In our system, each entry in a LR parsing table is regarded as a process which will handle shift and reduce operations. If the process discovers a conflict in a LR parsing table, the process copies its own stack and sends it to subprocesses which conduct reduce operations. After finishing the reduce operations, a merge process will be activated if necessary. The details of handling a conflict will be explained later.

To avoid recomputations, Tomita has devised a Graph Structured Stack (GSS) in which stack top elements with the same states will be merged. However, GSS is not so a simple data structure that the simplest way to implement GSS is to use side effects. But side effects are not desirable for us to simulate GSS, because we are going to implement GSS in the framework of a logic programming language, GHC[1].

In our implementation of PGLR, in order to simulate GSS, we use a Tree Structured Stack(TSS). Consider the following two stacks:

[1]In the paper [Tanaka 89], we have shown a generalized LR parser without using GSS.

	det	noun	pron	v	p	relp	$	NP	PP	VP	RELC	S
0	sh1	sh2	sh3					5				4
1		sh6										
2				re6	re6	re6	re6					
3				re7	re7	re7	re7					
4					sh7		acc		8			
5				sh10	sh7	sh9			12	11	13	
6				re5	re5	re5	re5					
7	sh1	sh2	sh3					14				
8					re2		re2					
9				sh10						15		
10	sh1	sh2	sh3					16				
11					re1		re1					
12				re4	re4	re4	re4					
13				re3	re3	re3	re3					
14				re10	sh7/re10	sh9/re10	re10		12		13	
15				re9	re9	re9	re9					
16				re8	sh7/re8	sh9/re8	re8		12		13	

fig.2: LR parsing table obtained from fig.1 grammar

(top) $[7, P, 4, S, 0]$ (bottom)
(top) $[7, P, 16, NP, 10, V, 5, NP, 0]$ (bottom)

The top element of two stacks being the same, '7,P', we will be able to merge these top two elements and will get the following TSS.

$[7, P, [4, S, 0], [16, NP, 10, V, 5, NP, 0]]$

3.1 Brief Introduction of GHC

Before explaining the details of PGLR algorithm, we will give a brief introduction of GHC. Typical GHC statements are given in fig.3. Roughly speaking, the vertical bar in a GHC statement of fig.3 works as a cut symbol of Prolog. When a goal 'a' is executed, a process of the statement (1) is activated and the body becomes a new goal in which 'b(X)' and 'c(X)' are executed simultaneously.This is called AND-parallel execution in GHC. In other word, subprocesses 'b(X)' and 'c(X)' are created by a parent process a and they run in parallel. Note that the definition of process 'c' in the statement (3) is going to instantiate the variable 'X' in 'c(X)' with 'a'. In such a case the execution of the process 'c' will be suspended until 'X' is instantiated by the process 'b(X)'.

```
(1) a:- true| b(X),c(X).
(2) b(X):- true| X=a.
(3) c(a):- true| true.
```

fig.3: Typical Statement of GHC

3.2 Description of PGLR Algorithm

Consider that an input sentence is consisted of a sequence of preterminals, 'p1 p2 ... pn'. PGLR begins activating a sequence of action processes, 'p1,p2,...,pn,p$' for each preterminals of an input sentence. Here, the process 'p$' corresponds to the end of the input sentence. At first, only an action process 'p1' becomes active, and all the other processes are suspended until stack information is sent by an active process. The action process 'p1' is initially given a stack with state '0'. It carries out a shift operation specified by a LR table, sends the new stack information to 'p2', and then terminates. The activated action process 'pi' performs shift or reduce operations and sends new stack information to the process 'pi+1' which will be newly activated. If the rightmost action process 'p$' is activated and finds out an 'acc' entry in the LR table, PGLR ends with success. If we have a conflict during parsing, more than two subprocesses will be activated simultaneously and run in parallel. There are four kinds of processes which are activated during parsing.

1. action process:
   Action processes perform the operations specified by an entry of the 'action table'. The details of action processes will be explained later with sample definitions.

2. reduce process:
   A reduce process pops the appropriate portion of the stack and creates a partially parsed tree by applying a grammar rule specified in a reduce entry. And then the reduce process activates a goto process to shift a new state. Suppose, in the course of a reduce operation, the reduce process finds out many branches on a stack, the reduce operation for every branch should be carried out. In order to do so, the reduce process creates a subprocess for the reduce operation.

3. goto process:
   A goto process performs a shift operation specified in the entry of a 'goto table'. The goto process is activated after the reduce operation.

4. merge_stack process
   A merge_stack process receives a number of stacks from both action processes and goto processes. In case of having many stacks whose top elements are same, the merge_stack process merges them into a tree structured stack. The definition of 'merge_stack' will be shown in appendix A.

3.3 Definition of Action Process

Followings are examples of definitions of action processes which carry out a shift, a reduce,an accept, and an error operation.

- activating action processes
  In order to parse a sentence, PGLR begins activating AND-parallel action processes which corresponds to a sequence of preterminals in an input sentence. If the input sentence is 'Doors open'. the initial goal becomes as following :

```
?- noun_0([[0]], [noun,doors], Stacks1),
   v_0(Stacks1,[v,opens], Stacks2),
   $_0(Stacks2,[ ],Result).
```

  The first argument of an action process is a set of stacks sent by the preceding action processes. The second argument is a pair of a preterminal and a word which will be an element pushed on the stack by shift operations. The third argument is a new stack calculated by this action process. The following is a definition of the first process 'noun_0' in the initial goal.

```
noun_0([ ], _, Out):- true |
     Out = [ ].
noun_0([[No | Stack] | Rest], T, Out):- true |
     noun(No, [No | Stack], T, Stacks1),
     noun_0(Rest, T, Stacks2),
     merge_stacks(Stacks1, Stacks2, Out).
```

  In the body of the second clause, a process 'noun', 'noun_0', and 'merge_stacks' are activated in parallel. Depending on the value of 'No', the process 'noun' performs a shift or reduce operation. The reason why the process 'noun_0' is called recursively is that every stack in the first argument should be examined. The process 'merge_stacks' merges all stacks calculated by the process 'noun' and 'noun_0'.

- shift operation
  An action process creates a pair of a look-ahead preterminal and a new state specified by a shift entry, pushes it on a stack, and finally terminates by itself. Suppose an entry of the 'action table' in fig.2 , namely the column 'noun' and the row '0'. As the entry contains 'sh 2', an action process has to perform a shift '2' operation. The definition of the action process is :

  ```
  noun(0, Stack, T, NStacks) :- true |
       NStacks=[[2,T| Stack]].
  ```

  In the above process definition, the first argument of 'noun' is a state of the top element of 'Stack'. The third argument 'T' is a leaf of parsing tree, namely a pair of a preterminal and a terminal symbol of the input sentence. The fourth argument 'NStacks' is a set of stacks calculated by this action process. In the body of this definition, a pair of state 2 and 'T' is pushed onto 'Stack'.

- reduce operation
  An action process activates a reduce process which is given a copy of stack information by the action process. The reduce process returns a reduced stack to the action process. After getting a reduced stack, the action process activates the same action process recursively which looks for another actions. Consider the entry of state '2' and a look-ahead preterminal 'v' in fig.2. The definition of an action process 'v' is :

  ```
  v(2, [_,T1| Stack], T, NStacks) :- true |
       reduce(1, 6, Stack, [T1], NStacks1),
       v_0(NStacks1, T, NStacks).
  ```

  In the body of an action process 'v', two processes, 'reduce' and 'v_0' are activated simultaneously. The process 'reduce' conducts a reduce and goto operation by which the action process moves to a new state. The reason why the action process 'v_0' is activated recursively is that any look-ahead preterminal remains the same (namely 'v') after a reduce process runs.

- shift/reduce operation
  At first, a shift operation and reduce operations are carried out, and then the action process activates a merge process which is going to merge two stacks, each of which is obtained by the shift and reduce operations.

  Consider an entry of state '14' and a look-ahead preterminal 'p' in fig.2. We will find out a shift-reduce conflict, 'sh 7/re 10'. The definition of an action process 'p' is :

  ```
  p(14 , [14 , T1| Stack] , T , NStacks):- true |
       reduce(1 , 10 , Stack , [T1] , Stacks1),
       p_0(Stacks1 , T , Stacks2),
       merge_stack([7,T,14,T1|Stack],Stacks2,NStacks).
  ```

  In the body of the process 'p', subprocesses 'reduce', 'p_0', and 'merge_stack' are activated simultaneously. The definition of 'merge_stack' will be shown in appendix A.

- reduce/reduce operation An action process activates reduce processes and sends them stack information, each of which is a copy of the stack kept by the action process. After finishing reduce processes, the action process will activate a merge process to merge several stacks sent by reduce processes. Finally, in order to look for another actions, the action process activates the same action process recursively.

- accept operation
  After an action process gets a result of parsing, it ends with success.

  Consider an entry of state '4' and a look-ahead preterminal '$' in fig.2, we will find out 'acc' which indicates a success of parsing. The definition of the action process '$' is :

  ```
  $(4 , [_,Tree|_] , _, Result):- true |
       Result=Tree.
  ```

'Tree' becomes the final output ('Result') of parsing.

- error operation
  If no operation is specified in an entry, an error handling process will be activated. We have to define an error handling process in some states if necessary. The following is a definition of an error process for a look-ahead preterminal 'noun'. The following definition has to be placed at the end of 'noun' processes.

```
otherwise.
det(S, Stack, _, NStacks):- true |
            NStacks=[ ].
```

GHC statements below 'otherwise' will be executed if all GHC statements above 'otherwise' fails.

3.4 Definition of Reduce Process

The definition of a reduce process is as follows:

```
reduce(0, N, Stacks, T, NStacks):- true|
     re(N, Stacks, T, NStacks).
reduce(M, N,[S,T1|Stack],T, NStacks, Tail):- integer(S)|
     M1 := M-1,
     reduce(M1, N, St1, [T1|T], NStacks).
otherwise.
reduce(_, _, [ ], _, NStacks):- true|
     NStacks = [ ].
reduce(M, N, [[_,T1|Stack]|Rest], T, NStacks):- true|
     M1 := M-1,
     reduce(M1, N, Stack, [T1|T], NStacks1),
     reduce(M, N, Rest, T, NStacks2),
     merge(NStacks1, NStacks2, NStacks).
```

In the body of the first reduce process, a subprocesses 're' is activated. In the body of the second reduce process, a subprocess 'reduce' is activated recursively to get a reduced stack. The fourth reduce process deals with a Tree Structured Stack which has many branches. In the body of the process, every branch is brought one by one and is sent to a subprocess 'reduce'. The first argument of reduce processes is the number of elements to be reduced. The second argument is a rule number which is used for the reduce operation.

After finishing all reduce operations, a process 're' is activated to create a partially parsed tree using a rule for the reduce operation. Following is a sample definition of the process 're'.

```
re(1,[S |Stack],T,NStacks):- true |
     s(S,[S |Stack],[sentence |T],NStacks).
re(2,[S |Stack],T,S,NStack):- true |
     s(S,[S |Stack],[sentence |T],NStacks).
re(3,[S |Stack],T,NStacks):- true |
     np(S,[S |Stack],[np |T],NStacks).
re(4,[S |Stack],T,NStacks):- true |
     . . . . .
```

The number in the first argument is a rule number that is used by the reduce operation. In the body, a goto process is activated to shift a new state.

3.5 Definition of Goto Process

After a reduce operation is finished, a goto process is activated to push a new element on the stack. The element is a pair of a partially parsed tree and a new state specified by a goto table. We will give a sample definition of goto processes.

```
np(0,Stack,T,NStacks):- true|
    NStacks = [[5,T|Stack]].
np(7,Stack,T,NStacks):- true|
    NStacks = [[14,T|Stack]].
np(10,Stack,T,NStacks):- true|
    NStacks = [[16,T|Stack]].
otherwise.
np(_, [ ], _,NStacks):- true |
    NStacks = [ ].
np(_, [[S|Stack] | Rest], T, NStacks):- true |
    np(S, [S|Stack],T,Stacks1),
    np([ ], Rest, T, Stacks2),
    merge_stack(Stacks1, Stacks2, NStacks).
```

3.6 An Example of PGLR Parsing

Following is a trace of parsing by PGLR parser.

input sentence : i open the door with a key .

Parsing begins with activating AND-parallel action processes each of which corresponds to a preterminal of the input sentence. However, only the first action process 'pron_0' will be executed and the other processes will be suspended until 'Stack1','Stack2',...,'Stack8' are instantiated one by one.

```
?- pron_0([[0]],[pron,i],Stack1), v_0(Stack1,[v,open],Stack2),
   det_0(Stack2,[det,the],Stack3), noun_0(Stack3,[noun,door],Stack4),
   p_0(Stack4,[p,with],Stack5), det_0(Stack5,[det,a],Stack6),
   noun_0(Stack7,[noun,key],Stack8), $_0(Stack8,[],Result).
```

Following is the actual output of tracing.

```
CALL  pron_0([[0]],[pron,i],Stack1)
CALL    pron(0,[0],[pron,i],Stack11)
EXIT    pron(0,[0],[pron,i],[[3,[pron,i],0]])
CALL    pron_0([],[pron,i],Stack12)
EXIT    pron_0([],[pron,i],[])
CALL    merge_stacks([[3,[pron,i],0]],[],Stack1)
EXIT    merge_stacks([[3,[pron,i],0]],[],[[3,[pron,i],0]])
EXIT  pron_0([[0]],[pron,i],[[3,[pron,i],0]])
CALL  v_0([[3,[pron,i],0]],[v,open],Stack2)
CALL    v([3,[[pron,i],0],[v,open],Stack21)
CALL      reduce(0,7,[0],[[pron,i]],Stack22)
CALL        re(7,[0],[[pron,i]],Stack22)
CALL          np(0,[0],[np,[pron,i]],Stack22)
EXIT          np(0,[0],[np,[pron,i]],[[5,[np,[pron,i]],0]])
EXIT        re(7,[0],[[pron,i]],[[5,[np,[pron,i]],0]])
EXIT      reduce(0,7,[0],[[pron,i]],[[5,[np,[pron,i]],0]])
CALL      v_0([[5,[np,[pron,i]],0]],[v,open],Stack21) : skip
EXIT      v_0([[5,[np,[pron,i]],0]],[...],[[10,[v,open],5,[np,...],0]])
                ..............................
EXIT  v_0([[3,[pron,i],0]],[...],[[10,[v,open],5...]])
CALL  det_0([[10,[v,open],5...]],[det,the],Stack3) : skip
EXIT  det_0([[10,[v,open],5...]],[det,the],[[1,[det,the],10,[v,open]...]])
CALL  noun_0([[1,[det,the],10,[v,open]...]],[noun,door],Stack4) : skip
EXIT  noun_0([[1,[det,the]...]],[...],[[6,[noun,door],1,[det,the]...]])
```

```
CALL  p_0([[6,[noun,door],1...],[p,with],Stack5)
CALL    p(6,[6,[noun,door],1...],[p,with],Stack51)
CALL      reduce(1,8,[1,[det,the],10...],[[noun,door]],Stack52) : skip
EXIT      reduce(1,8,[1...],[...],[[16,[np,[det,the],[noun,door]],10...]])
CALL      p_0([[16,[np...],10,[v...]...]],[p,with],Stack51)
CALL        p(16,[16,[np...],10,[v...]...],[p,with],Stack52)
CALL          reduce(1,8,[10,[v...]...],[[np,[det...],[noun...]],Stack53) : skip
EXIT          reduce(1,8,[10...],[...],[[11,[vp,[v,open],[np...]],5...]])
CALL          p_0([[11,[vp...],5,[np...],0]],[p,with],Stack64)
CALL            p(11,[11,[vp...],5,[np...],0],[p,with],Stack65)
CALL              reduce(1,1,[5,[np...],0],[[vp,[v...],[np...]]],Stack66) : skip
EXIT              reduce(1,1,[5...],[...],[[4,[sentence,[np...],[vp...]],0]])
CALL              p_0([[4,[sentence,[np...],[vp...]],0]],[p,with],Stack65)
CALL                p(4,[4,[sentence,[np...],[vp...]],0],[p,with],Stack67)
EXIT                p(4,[4...],[...],[[7,[p,with],4,[sentence...],0]])
            .............................
EXIT              p_0([[4...]],[...],[[7,[p,with],4,[sentence...],0]])
EXIT            p(11,[11...0],[...],[[7,[p,with],4,[sentence...],0]])
            .............................
EXIT          p_0([[11...]],[...],[[7,[p,with],4,[sentence...],0]])
CALL          merge_stack([7,[p,with],16,[np...]...],
                          [[7,[p,with],4,[sentence...],0]],Stack52)
EXIT          merge_stack([7,[p,with],16...],[[7,[p,with],4...]],
                          [[7,[p,with],[16...],[4...]]])
EXIT        p(16,[16...],[...],[[7,[p,with],[16...],[4...]]])
            .............................
EXIT      p_0([[16...]],[...],[[7,[p,with],[16...],[4...]]])
EXIT    p(6,[6,[noun,door],1...],[...],[[7,[p,with],[16...],[4...]]])
            .............................
EXIT  p_0([[6,[noun,door],1...],[...],[[7,[p,with],[16...],[4...]]])
            .............................
CALL  $_0([[6,[noun,key],1,[det,a],7,[p...],[16...],[4...]]],[],Stack8)
CALL    $(6,[6,[noun,key],1...],[],Stack81)
CALL      reduce(1,5,[1,[det,a],7...],[[noun,door]],Stack82)
            .............................
EXIT      reduce(1,5,[1...],[...],[[14,[np,[det,a],[...]],7,[p,with]...]])
CALL      $_0([[14,[np,[det,a],[...]],7,[p,with]...]],[],Stack81)
CALL        $(14,[14,[np...],7,[p,with],[16...],[4...]],[],Stack83)
CALL          reduce(1,10,[7,[p,with],[16...],[4...]],[[np...]],Stack84)
CALL            reduce(0,10,[[16...],[4...]],[[p,with],[np...]],Stack84)
CALL              re(10,[[16...],[4...]],[[p,with],[np...]],Stack84)
CALL                pp([16...],[[16...],[4...]],[pp,[p...],[np...]],Stack84)
CALL                  pp(16,[16...],[pp...],Stack85)
EXIT                  pp(16,[16...],[pp...],[[12,[pp...],16...]])
CALL                  pp([],[[4...]],[pp...],Stack86)
CALL                    pp(4,[4...],[pp...],Stack87)
EXIT                    pp(4,[4...],[pp...],[[8,[pp...],4...]])
            .............................
EXIT                pp([...],[[16...],[4...]],[...],[[12...],[8...]])
            .............................
EXIT          reduce(1,10,[7...],[...],[[12...],[8...]])
CALL          $_0([[12...],[8...]],[],Stack83)
CALL            $(12,[12,[pp...],16...],[],Stack88) : skip
EXIT            $(12,[12...],[...],
                  [[sentence,[np...],[vp,[v...],[np,[np...],[pp...]]]]])
CALL            $(8,[8,[pp...],4...],[],Stack89) : skip
EXIT            $(8,[8,[pp...],4...],[],
                  [[sentence,[sentence,[np...],[vp...]],[pp...]]])
```

4 The Results of an Experiment

We used a Sun-3/260 workstation and GHC. The CFG grammar rules are shown in appendix B. Sample sentences to be parsed are:

1. I open the window.
2. Diagram is an augmented grammar.

3. The structural relations are holding among constituents.
4. It is not tied to a particular domain of applications.
5. Diagram analyzes all of the basic kinds of phrases and sentences.
6. This paper presents an explanatory overview of a large and complex grammar that is used in a sentence.
7. The annotations provide important information for other parts of the system that interpret the expression in the context of a dialogue.
8. For every expression it analyzes, diagram provides an annotated description of the structural relations holding among its constituents.
9. Procedures can also assign scores to an analysis, rating some applications of a rule as probable or as unlikely.

The elapsed time of parsing is shown in fig.4.

Sentence No.	Time (ms)	Number of Tree
1	280	2
2	180	1
3	680	15
4	940	14
5	2600	30
6	4420	56
7	27340	192
8	40760	200
9	5600	186

fig.4 : elapsed time of parsing

5 Conclusion

It is a straightforward task for us to implement PGLR parser in GHC. The reason is that GHC has a very good mechanism for synchronization of processes. The significant feature of PGLR is that each entry of LR table is regarded as a process which handles shift and reduce operations. When a conflict occurs in an entry of LR table, the corresponding parsing process activates two or more subprocesses which run in parallel and simulate breadth first strategy. Each subprocess is given a stack information by the parent process.

The experiment has revealed that PGLR runs so fast that it will be a promising parser for processing many complex natural language sentences.

However, PGLR has many problems to be solved. For example, (1) How to handle gaps and idioms? (2) How to integrate syntactic and semantic processing? (3) Is there a good algorithm to distribute many processes among limited number of processors?

As the PGLR explained in this paper strictly scans input words from left to right, the number of processes which run in parallel will be limited in nature. It is valuable for us to look for better algorithm which enables us to extract more parallelism.

References

[Aho 72] Aho,A.V.and Ulman,J.D.: *The Theory of Parsing,Translation,and Compiling,* Prentice-Hall,Englewood Cliffs,New Jersey (1972)

[Aho 85] Aho,A.V.,Senthi,R.and Ulman,J.D.: *Compilers Principles,Techniques,and Tools,*Addison-Wesley (1985)

[Fuchi 87] Fuch.K. Furukawa.K. Mizoguchi.F.:*Heiretu Ronri Gata Gengo GHC To Sono Ouyou*, Kyoritsu Syuppan (1987) in Japanese

[Knuth 65] Knuth,D.E.: *On the translation of languages from left to right*.Information and Control 8:6,pp.607-639

[Konno 86] Konno.A. Tanaka,H.:*Hidari Gaichi Wo Kouryo Shita Bottom Up Koubun Kaiseki*, Conputer Softwear,Vol.3, No.2, pp.115-125 (1986) in Japanese

[Nakata 81] Nakata.I.:*Compiler*, Sangyo Tosyo (1981) in Japanese

[Matsumoto 86] Matsumoto,Y. Sugimura,R.:*Ronri Gata Gengo Ni Motodsuku Koubun Kaiseki System SAX*, Computer Softwear,Vol.3, No.4, pp.4-11 (1986) in Japanese

[Matsumoto 87] Matsumoto,Y.:*A Parallel Parsing System for Natural Language Analysis*, New Generation Computing, Vol.5, No. 1, pp.63-78 (1987)

[Matsumoto 89] Matsumoto.Y.:*Natural Language Parsing Systems based on Logic Programming*, Ph.D thesis of Kyoto University, (June 1989)

[Mellish 85] Mellish,C.S.:*Computer Interpretation of Natural Language Descriptions*, Ellis Horwood Limited (1985)

[Nilsson 86] Nilsson,U.: *AID:An Alternative Implementation of DCGs*, New Generation Computing, 4, pp.383-399 (1986)

[Tanaka 89] Tanaka,H. and Numazaki,H. :*Parallel Generalized LR Parsing based on Logic Programming* International Workshop on Parsing Technologies, pp 329-338 (1989)

[Okumura 89] Okumura,M.:*Sizengengo Kaiseki Ni Okeru Imiteki Aimaisei Wo Zoushinteki Ni Kaisyou Suru Keisan Model*, Natural Language Analysis Working Group,Information Processing Society of Japan,NL71-1 (1989) in Japanese

[Pereira 80] Pereira,F.and Warren,D.: Definite Clause Grammar for Language Analysis-A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artif. Intell*, Vol.13, No.3, pp.231-278 (1980)

[Tokunaga 88] Tokunaga,T. Iwayama,M. Kamiwaki,T. Tanaka,H.:*Natural Language Analysis System LangLAB*, Transactions of Information Processing Society of Japan,Vol.29, No.7, pp.703-711 (1988) in Japanese

[Tomita 86] Tomita,M.:*Efficient Parsing for Natural Language*, Kluwer Academic Publishers (1986)

[Tomita 87] Tomita,M.: *An Efficien Augmented-Context-Free Parsing Algorithm*, Computational Linguistics, Vol.13, Numbers 1-2, pp.31-46 (1987)

[Ueda 85] Ueda,K.:*Guarded Horn Clauses*, Proc. The Logic Programming Conference, Lecture Notes in Computer Science, 221 (1985)

[Uehara 83] Uehara,K. Toyoda,J.: *Sakiyomi To Yosokukinou Wo Motsu Jutugo Ronri Gata Koubun Kaiseki Program : PAMPS*, Transactions of Information Processing Society of Japan, Vol.24, No.4, pp.496-504 (1983) in Japanese

Appendix A : Definition of merge_stack Process

```
merge_stacks([ ],Stacks,NStacks):- true |
        NStacks = Stacks.
merge_stacks(Stacks,[ ],NStacks):- true |
        NStacks = Stacks.
merge_stacks([Stack|Rest],Stacks,NStacks):- true |
        merge_stack(Stack,Stacks,NStacks1),
        merge_stacks(Rest,NStacks1,NStacks).
merge_stack(Stack,[ ],NStacks):- true |
        NStacks = [Stack].
merge_stack([S,T,S1|Stack1],[[S,T,S2|Stack2]|Rest],NStacks):- integer(S1) |
        merge_stack1([S1|Stack1],[S2|Stack2],Stack3),
        NStacks = [[S,T|Stack3]|Rest].
otherwise.
```

```
merge_stack([S,T,S1|Stack1],[[S,T,S2|Stack2]|Rest],NStacks):- true |
      merge_stack2([S1|Stack1],[S2|Stack2],Stack3),
      NStacks = [[S,T|Stack3]|Rest].
merge_stack(Stack,[Stack1|Rest],NStacks):- true |
      merge_stack(Stack,Rest,NStacks1),
      NStack = [Stack1 | NStacks1].
merge_stack1(Stack1,[S2|Stack2],Stack3):- integer(S2) |
      Stack3=[Stack1,[S2|Stack2]].
otherwise.
merge_stack1(Stack1,Stack2,Stack3):- true |
      Stack3=[Stack1|Stack2].
merge_stack2(Stack1,[S2|Stack2],Stack3):- integer(S2) |
      Stack3=[[S2|Stack2]|Stack1].
otherwise.
merge_stack2(Stack1,Stack2,Stack3):- true |
      merge(Stack1, Stack2, Stack3).
```

Appendix B : Grammar Rules

```
advp → adv.
advp → p,sdec.
ncomp → vp2.
ncomp → ncomp,pp.
modalp → modal,nt.
vp3 → nt,vp3.
vp2 → v.
vp2 → vp2,advp.
gerund → nt,vp.
pp → p,np,paraconj,np.
aux → bep.
adjp → adj.
adjp → adjp,paraconj,adjp.
pred → np.
pred → pred,pp.
nomhd → v,nomhd.
ddet → nt,all.
ddet → nt,all,det.
vp → v,p,advp.
vp → v,adjp.
vp → v,sdec.
vp → v,obj,adjp.
vp → v,obj,advp.
vp → vp,advp.
sdec → subj,adv,vp.
sdec → subj,adv,auxd,vp.
sdec → subj,bep,adv,pred.
sdec → subj,aux,bep,pred.
sdec → subj,aux,adv,bep,adv,pred.
sdec → subj,adv,be.
sdec → sdec,comma,vp3.
np → nomhd.
np → a,nomhd,ncomp.
np → a,ncomp.
np → pron.
np → ddet,adjp,nomhd,ncomp.
np → gerund.
np → relpro,sdec.
sentence → sdec.
sentence → pp,comma,sentence.
srel → relpro,bep,pred.

advp → as,advp,ascomp.
ncomp → pp.
ncomp → srel.
ncomp → infinitrel.
vp3 → vp.
infinitrel → vp.
vp2 → v,advp.
vp2 → vp2,pp.
pp → p,obj.
bep → be.
aux → modalp.
adjp → ddet,adj.
adjp → adv,adjp.
pred → pp.
nomhd → n.
ddet → det.
ddet → all,det.
obj → np.
vp → v,advp.
vp → v,vp.
vp → v,np,relpro,sdec.
vp → v,obj,vp.
vp → v,obj,be,pred.
vp → vp,paraconj,adjp.
sdec → subj,auxd,vp.
sdec → subj,adv,auxd,adv,vp.
sdec → subj,adv,bep,pred.
sdec → subj,aux,bep,adv,pred.
sdec → subj,adv,aux,be.
sdec → subj,be.
sdec → sdec,advp.
np → nomhd,ncomp.
np → ddet,nomhd.
np → ddet.
np → pron,ncomp.
np → ddet,adjp,ncomp.
np → as,adj,of,np.
np → np,paraconj,np.
sentence → sdec,paraconj,sdec.
sentence → advp,comma,sentence.
srel → subj,vp2.

advp → as,advp.
ncomp → of,np.
ncomp → adjp.
modalp → modal.
vp3 → pred.
infinitrel → be,vp.
vp2 → v,adjp.
gerund → vp.
pp → p,np,of,np.
bep → be,nt.
auxd → aux.
adjp → as,adjp.
pred → adjp.
pred → vp2.
nomhd → adjp,nomhd.
ddet → all.
ddet → all,of,det.
vp → v.
vp → v,np.
vp → v,obj.
vp → v,relpro,sdec.
vp → v,advp,obj.
vp → vp,pp.
sdec → subj,vp.
sdec → subj,auxd,adv,vp.
sdec → subj,bep,pred.
sdec → subj,adv,bep,adv,pred.
sdec → subj,aux,adv,bep,pred.
sdec → subj,aux,be.
sdec → vp3,comma,sdec.
sdec → sdec,comma,advp.
np → a,nomhd.
np → ddet,nomhd,ncomp.
np → ddet,ncomp.
np → ddet,adjp,nomhd.
np → det,gerund.
np → as,adj,np.
np → np,comma,np.
sentence → sdec,comma,paraconj,sdec.
srel → relpro,vp.
subj → np.
```