Parallel Generalized LR Parsing based on Logic Programming

Hozumi TANAKA
Tokyo Institute of Technology

Hiroaki NUMAZAKI
Tokyo Institute of Technology

Abstract

A generalized LR parsing algorithm, which has been developed by Tomita[Tomita 86], can treat a context free grammar. His algorithm makes use of breadth first strategy when a conflict occcurs in a LR parsing table. It is well known that the breadth first strategy is suitable for parall processing. This paper presents an algorithm of a parallel parsing system (PLR) based on a generalized LR parsing. PLR is implemented in GHC[Ueda 85] that is a concurrent logic programming language developed by Japanese 5th generation computer project. The feature of PLR is as follows: Each entry of a LR parsing table is regarded as a process which handles shift and reduce operations. If a process discovers a conflict in a LR parsing table, it activates subprocesses which conduct shift and reduce operations. These subprocesses run in parallel and simulate breadth first strategy. There is no need to make some subprocesses synchronize during parsing. Stack information is sent to each subprocesses from their parent process. A simple experiment for parsing a sentence revealed the fact that PLR runs faster than PAX[Matsumoto 87][Matsumoto 89] that has been known as the best parallel parser.

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. One of parallel parsing systems we have ever known is PAX[Matsumoto 87][Matsumoto 89] that is based on Chart parser. It is well known that LR parser is the most efficient paser, since LR parsing algorithm runs deterministicly for any LR grammar which is a subset of context free grammar. Unfortunately, LR grammar is too weak to parse sentences of natural languages. When we apply LR parsing algorithm to context free grammar, it is an usual case that conflicts appears in a LR pasing table. So we need to generalize the LR parsing algorithm in order to process these conflicts. There are two kinds of strategies to resolve the conflicts, namely a depth first strategy and a breadth first strategy. Nilsson[Nilsson 86] has adopted a depth first strategy and Tomita[Tomita 86] a breadth first strategy which is called a generalized LR parsing. As it is easy for us to simulate the breadth first strategy by parall processing technique. We have developed a parallel generalized LR parsing system (PLR) based on the generalized LR parsing algorithm which makes use of a breadth first strategy.

After we will give a brief introduction of LR parsing algorithm in section 2, we will describe our PLR system details of which will be explained in section 3. PLR is implemented in a concurrent logic programming language called GHC[Ueda 85] that is developed by Japanese 5th generation computer project. One of the most significant feature of PLR is to regard each entry of a LR parsing table as a process which handles shift and reduce operations. If the process discovers a conflict in a LR parsing table, it creates and activates subprocesses in order to process shift and reduce operations in the conflict. These subprocesses run in parallel and simulate breadth first strategy. There is no need to make subprocesses synchronize during parsing. Stack information is sent to each subprocesses from their parent process. In order to understand PLR algorithm we will show a trace of actual parsing in subsection 3.6.

In section 4, we will explain some results of the experiment which parses sentences using PLR and PAX. The experiment revealed the fact that PLR runs faster than PAX that is one of the best parallel parsers.

2 Generalized LR Parsing algorithm

The generalized LR parser is guided by a LR parsing table which is generated from grammar rules given. Fig.2 shows an ambiguous English grammar. Fig.2 shows a LR parsing table generated from the English grammar. The LR parsing table is devided into two parts, an action table and a goto table.

The lefthand 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 lookahead 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 which mean that there is a conflict in the entry, and a parser should conduct more than two operations at once.

The symbol 'sh N' in some entries means that generalized LR parser has to push a lookahead 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 righthand 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 PLR explained in the next section, conflicts will be resolved using parallel processing technique and we do not mind the order of the operations in a conflict.

3 Implementation of PLR

PLR 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 are regarded as a process which will handle shift and reduce operations. If the process discovers a conflict in a LR parsing table, it activates subprocesses in order to process shift and reduce

(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

	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

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

fig.3: typical stat ·nent of GHC

operations in the conflict. These subprocesses run in parallel and simulate breadth first strategy for the generalized LR parsing. There is no need to make subprocesses synchronize during parsing. Stack information is sent to each subprocesses from their parent process.

3.1 Brief Introduction of GHC

Before explaining the details of PLR 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 corresponding to the statement (1) is activated and the body becomes a new goal in which 'b' and 'c' are excuted simultaneously, since GHC adopts AND-parallel strategy. In other word, subprocesses 'b' and 'c' are created by a parent process 'a' and they run in paralell. Although GHC has a few of synchronization mechanisms, it will not be necessary for you to understand them.

3.2 Description of PLR Algorithm

At first, PLR creates a list of preterminals of an input sentence which will be parsed. PLR parser begins activating an action process which corresponds to the LR table entry determined by the state '0' and the first preterminal in the preterminal list. The action process activates the other processes according to the comands specified in the LR talbe entry. Activated processes recieves stack information from the parent process and also perform some comands specified in the corresponding LR table entry. The process activation will continue until some processes find out an 'acc' or an 'error' entry. If we have a conflict during parsing, more than two subprocesses will be activated at once and run in parallel. There are three kinds of processes which are activated in PLR parser.

- action process:
  An action process carries out shift and/or reduce operations. In case of a shift operation, the action process pushes a lookahead preterminal on a stack and activates a new process which corresponds to new state given by the shift operation. When an action process encounters a reduce operation, a reduce process will be activated and recieve stack information from the parent action process. If an action process finds a conflict, more than two subprocesses will be activated each of which perform either a shift or a reduce operation. These subprocesses run in parallel. If an action process enconters an 'acc' operation, the action process will extract the result of the parsing and end with success. On the contrary, if all of the above conditions are not satisfied, an action process will end with failure.

- reduce process:
  Using a grammar rule specified by a reduce operation, the reduce process makes a reduction of an appropriate portion of stack, and the reduce process activates a goto process in order to enter a new state.

- goto process:
  Using stack information given by a reduce process, a goto process activates an action process to enter a new state.

In the following subsections, we will give the GHC definition of PLR processes obtained by the LR table shown in fig 2.

3.3 Definition of Action Process

Followings are examples of definitions of an action process.

1. Suppose an action process 'i0' that corresponds to the entry in fig.2 whose row and column are 0 and 'noun' respectively. As the entry contains 'sh 1', the process has to activate a subprocess which carries out a shift operation. The definition of the process 'i0' is shown below.

```
i0(noun, Stack, [noun,NextCat|List], Info) :- true |
        i1(NextCat, [[1,noun]|Stack], [NextCat|List], Info).
```

   In the above process definition, the predicate 'i0' is a process name, and its first argument is a lookahead preterminal 'noun'. The second argument is 'Stack' on which information about state, grammatical categories and the other information are pushed. The third is a list of preterminals of an input sentence. The fourth outputs 'Info', the results of parsing. The subprocess 'i1' is activated and carries out a 'sh 1' operation. The subprocess 'i1' recieves a new stack which consists of 'Stack', state '1' and a preterminal 'noun'. Note that in the third argument of the process 'i1', preterminal 'noun' is eliminated from the list of preterminals, since preterminal 'noun' should be shifted.

2. Consider an entry of state '2' and a lookahead preterminal 'v' in fig.2. The definition of action process 'i2' is given below :

```
i2(v, Stack, List, Info) :- true |
        re6(v, Stack, List, Info).
```

   In the body of an action process 'i2', a subprocess 're6' is activated in order to conduct a reduce operation. The subprocess 're6' recieves the same stack information and a preterminal list as those of the parent process 'i2'. The detail of the reduce process will be explained later.

3. Consider an entry of state '14' and a lookahead preterminal 'p' in fig.2. We will find out a shift-reduce conflict, 'sh 7/re 10'. The definition of an action process 'i14' is as follows.

```
i14(p, Stack, [p,NextCat|List], Info) :- true |
        i7(NextCat, [[7,p]|Stack], [NextCat|List], Info1),
        re10(p, Stack, [p,NextCat|List], Info2),
        merge(Info1, Info2, Info).
```

In the body of the process 'i14', both subprocesses 'i7' and 're10' carry out a shift and a reduce operation simultaneously. The 'merge' process is a built-in process which merges the output produced by the subprocesses 'i7' and 're10'.

4. Consider the entry of state '4' and a lookahead preterminal '$' in fig.2. We will find out 'acc' in the entry which indicates a success of parsing. The definition of the action process 'i4' is as follows.

```
i4($, [[_,Result]|_], _, Info) :- true |
        Info=[Result].
```

In the body of the action process 'i4','[Result]' is sent to the fourth argument 'Info', and finally the action process 'i4' terminates with success.

5. If no operation is specfied in an entry, an error handling process has to be activated. We have to define an error handling process in some states if necessary. The following is a definition of an error process in state '0' which should be placed at the end of definitions of the process 'i0'.

```
otherwise.
i0(_, _, _, Info) :- true |
        Info=[].
```

The statement 'otherwise' is a built-in statement which declares that GHC statements below 'otherwise' should be executed after all GHC stetements before 'ohterwise' fails.

3.4 Definition of Reduce Process

The following definition of a reduce process 're10' is an example of reduce actions correspondds to the grammar rule numbered 10 in fig.1((10) PP—p,NP).

```
re10(NextCat, OldStack, List, Info) :-
        OldStack=[[_,T1],[_,T2],[State,T3]|Tail]|
        pp(State, NextCat, [pp,T2,T1], [[State,T3]|Tail], List, Info).
```

In the second argument of 're10', the topmost two elements of 'OldStack' are popped and sent to a goto process 'pp' in which the third argument '[pp,T2,T1]' constructs a syntactic tree whose root is 'pp' in accordance with the grammar rule 10 in fig.1. The name of the goto process 'pp' is the name of the lefthand side nonterminal symbol in the grammar rule 10. The first argument 'State' is a new state number extracted from 'OldStack'. Note that the reduce process 're10' passes a next incomming preterminal 'NextCat' to the 'pp' process, since a reduce process does not consume any incomming preterminals.

3.5 Definition of Goto Process

After a reduce operation is carried out, a goto process is activated in order to enter a new state in which a new action process will be activated. At that time, the goto process uses both an incomming nonterminal symbol and a state number on the top of the stack.

We will give a sample definition of goto processes.

```
s(0, NextCat, Tree, Stack, List, Info) :- true |
        i4(NextCat, [[4,Tree]|Stack], List, Info).
```

The process 's' defined above is activated after 's' is constructed by a reduce process in state '0'. As the entry of row '0' and column 's' in the LR table of fig.2 includes '4', the goto process 's' activates an action process 'i4' pushing state '4' and tree information onto the stack.

3.6 An Example of PLR Parsing

Given a LR table of fig.2, a translater generates the following definitions of parsing processes.

```
i0(det,Stack,[_,NextCat|List],Info):- true|
        i1(NextCat,[[1,det]|Stack],[NextCat|List],Info).

i0(noun,Stack,[_,NextCat|List],Info):- true|
        i2(NextCat,[[2,n]|Stack],[NextCat|List],Info).

i0(pron,Stack,[_,NextCat|List],Info):- true|
        i3(NextCat,[[3,pron]|Stack],[NextCat|List],Info).

otherwise.
i0(_,_,_,Info):- true|Info=[].

i1(noun,Stack,[_,NextCat|List],Info):- true|
        i6(NextCat,[[6,noun]|Stack],[NextCat|List],Info).

        . . . . . .
```

Following is an example of PLR parsing.

```
input sentence : i open the door with a key .

Parsing begins with activating the following action process 'i0'.
 i0(pron,[[0,[]]],[pron,v,det,noun,p,det,noun,$],Info)
     -      -Stack    -List of Preterminal
    Lookahead

Activates the action process 'i3' for 'shift 3'.
 i3(v,[[3,pron],[0,[]]],[v,det,noun,p,det,noun,$],Info)

Activates the reduce process 're7' for 'reduce 7'.
 re7(v,[[3,pron],[0,[]]],[v,det,noun,p,det,noun,$],Info)
        [[3,pron],[0,[]]]=[[_,T1],[State,T2]|Tail]
Activates the goto process 'np'.
 np(0,v,[np,pron],[[0,[]]],[v,det,noun,p,det,noun,$],Info)
```

```
    ^State ^Tree    ^Stack

Activates the action process 'i5' for 'goto 5'.
 i5(v,[[5,[np,pron]],[0,[]]],[v,det,noun,p,det,noun,$],Info)

Activates the action process 'i10' for 'shift 10'.
 i10(det,[[10,v],[5,[np,pron]],[0,[]]],[det,noun,p,det,noun,$],Info)

        . . . . . .

 i16(p,[[16,[np,det,noun]],[10,v],[5,[np,pron]],[0,[]]],[p,det,noun,$],Info)
A conflict 'shift 7/reduce 8' occures.
Activates 'i7' and 're8' processes simultaneously.

 i7(det,[[7,p]|[[16,[np,det,noun]],[10,v],[5,[np,pron]],[0,[]]]],[det|[n,$]],Info)
 re8(p,[[16,[np,det,noun]],[10,v],[5,[np,pron]],[0,[]]],[p,det|[noun,$]],Info)

        . . . . . .

Both processes end with success and produce the following results in 'Info'.
 i4($,[[4,[s,[np,pron],[vp,v,[np,[np...],[pp,p,[np...]]]]]],[0,[]]],[$],Info)
    Info=[s,[np,pron],[vp,v,[np,[np,det,noun],[pp,p,[np,det,noun]]]]]

 i4($,[[4,[s,[s,[np,pron],[vp,v,[np,[np...]]]],[pp,p,[np...]]]],[0,[]]],[$],Info)
    Info=[s,[s,[np,pron],[vp,v,[np,[np,det,noun]]]],[pp,p,[np,det,noun]]]
```

4 The Results of A Experiment

We conducted an experiment to parse many English sentences with many PP attachments such as :

NP,v,NP
NP,v,NP,PP
NP,v,NP,PP,PP
NP,v,NP,PP,PP,PP
. . .

In the experiment, PLR and PAX are used to parse sentences. The number enclosed by parenthesis in fig.4 indicates the number of parsing trees. PLR runs 1.4 times faster than PAX that was known as the best parallel parser in the past. In order to get all parsing trees of a sentence with 9 PP attachments, PLR takes about 65 sec. on Sun-3/260 workstation. It means that PLR produces a parsing tree only every 4 msec.

The reader should note that the PLR which we explained in this paper does not use a graph structured stack. For comparison, the results of parsing which makes uses of the graph structured stack is shown by a solid line. The PLR parser with a graph structured stack runs 10 times slower than the one without a graph structured stack. The reason is that the former

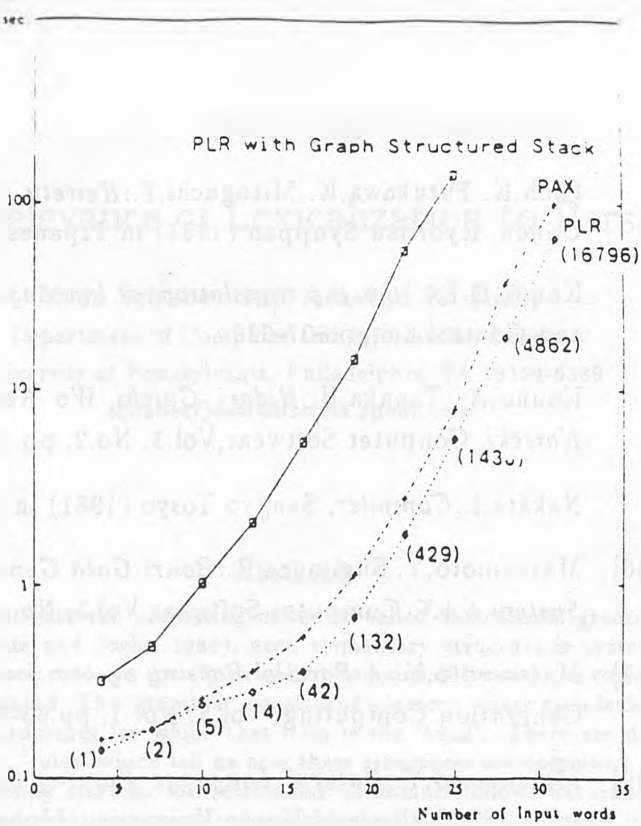

fig.4: The result of Parsing time

causes many processes to wait for synchronization. We are now considering the reason why PLR parser without the graph structured stack runs so fast. One of the reasons is that PLR parser without the graph does not cause many processes to suspend for synchronizations.

5 Conclusion

We described an exaple of the implementation of the PLR algorithm in GHC in which each entry of the LR table is regarded as a process which handles shift and reduce operations. When a conflict occurs in an entry of the LR table, the corresponding parsing process activates two or more subprocesses which run in parallel and simulate breadth first strategy of the generalized LR parsing. Each subprocess is given the stack information from the parent process and runs further to execute a shift and a reduce operation.

The experiment has revealed that PLR runs faster than PAX that has been known as the best parallel parser. PLR runs so fast that it will be a promising parser for processing many complex natural language sentences.

However, PLR has many problems to be solved, for example, handling of gapping and idiom, and integration of syntactic and semantic processing which are urgent problems to be solved in the near future.

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] Fuchi,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*, Computer 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)

[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] Percira,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