A New Parallel Algorithm for Generalized LR Parsing

Hiroaki NUMAZAKI Hozumi TANAKA
Department of Computer Science
Tokyo Institute of Technology
2-12-1 Ookayama Meguro-ku Tokyo 152, Japan

Abstract

Tomita's parsing algorithm [Tomita 86], which adapted the LR parsing algorithm to context free grammars, makes use of a breadth-first strategy to handle LR table conflicts. As the breadth-first strategy is compatible with parallel processing, we can easily develop a parallel generalized LR parser based on Tomita's algorithm [Tanaka 89]. However, there is a problem in that this algorithm synchronizes parsing processes on each shift action for the same input word to merge many stacks into Graph Structured Stacks (GSS). In other words, a process that has completed a shift action must wait until all other processes have ended theirs — a strategy that reduces parallel performance. We have developed a new parallel parsing algorithm that does not need to wait for shift actions before merging many stacks, using stream communication of a concurrent logic programming language called GHC [Ueda 85]. Thus we obtain a parallel generalized LR parser implemented in GHC.

1 Introduction

To provide an efficient parser for natural language sentences, a parallel parsing algorithm is desirable. As Tomita's algorithm is compatible with parallel processing, we can easily develop a parallel generalized LR parser [Tanaka 89]. However, with respect to the performance of the parallel parsing, one of the defects of Tomita's algorithm is that it forces many parsing processes to synchronize on each shift action for the same input word. A parsing process that has completed a shift action must wait until all other processes have completed their shift actions as well; such a synchronization strategy reduces the performance of parallel parsing.

In this paper, we will present a new parallel parsing algorithm which is a natural extension of Tomita's [Tomita 86]. Our algorithm can achieve greater performance in parallel parsing for natural language sentences.

There are two major differences between Tomita's algorithm and ours. Initially, the new algorithm does not make parsing processes wait for shift actions to merge many stacks with the same top state. The process that has finished a 'shift N' action first can proceed to the next actions until a reduce action needs to pop the element 'N' from the stack. If some other parsing processes carry out the same 'shift N' actions, their stacks will be merged into the position in which the first process has placed an element by the 'shift N' action.

Secondly, to avoid duplications of parsing processes the new algorithm employs Tree Structured Stacks (TSS) instead of Graph Structured Stacks (GSS). The reason why we do not use GSS is because it is rather complicated to implement the GSS data structure in the framework of a parallel logic programming language such as GHC. The merge operation of the stacks is realized by a GHC stream communication mechanism.

In section 2 we explain generalized LR parsing, in section 3 give a brief introduction to GHC, and in section 4 decribe our new parallel generalized LR parsing algotihm. In section 5 we compare the parallel parsing performance of our algorithm with Tomita's.

2 Generalized LR Parsing Algorithm

The execution of the generalized LR algorithm is controlled by an LR parsing table generated from predetermined grammar rules. Figure 1 shows an ambiguous English grammar structure, and Figure 2 an LR parsing table generated from Figure 1.

Action table entries are determined by a 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 stack operations: shift and reduce operations. Some entries in the LR table contain more than two operations and are thus in conflict. In such cases, a parser must conduct more than two operations simultaneously.

The symbol 'sh N' in some entries indicates that the 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 the generalized LR parser has to reduce from the top of the stack the number of elements equivalent to that of the right-hand side of the rule numbered 'N'. The symbol 'acc' means that the generalized LR parser has successfully completed parsing. If an entry contains no operation, the generalized LR parser will detect an error.

The right-hand table entry indicates which state the parser should enter after a reduce operation. The LR table shown in Figure 2 has two conflicts at state 11 (row no. 11) and state 12 for the 'p' column. Each of the conflicting two entries contains a shift and a reduce operation and is called a shift-reduce conflict. When a parser encounters a conflict, it cannot determine which operation should be carried out first. In our parser, conflicts will be resolved using a parallel processing technique such that the order of the operations in conflict is of no concern.

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

Fig.1: An Ambiguous English Grammar

3 Brief Introduction to GHC

Before explaining the details of our algorithm, we will give a brief introduction to GHC, typical statements of which are given in Figure 3. Roughly speaking, the vertical bar in a GHC statement (Fig.3) functions as a cut symbol of Prolog. When goal 'a' is executed, a process of statement (1) is activated and the body becomes a new goal in which 'b(Stream)' and 'c(Stream)' are executed simultaneously. In GHC, this is called AND-parallel execution. In other words, subprocesses 'b(Stream)' and

	det	noun	pron	v	p	$	NP	PP	VP	S
0	sh1		sh2				4			3
1		sh5								
2				re5	re5	re5				
3					sh6	acc		7		
4				sh8	sh6			10	9	
5				re4	re4	re4				
6	sh1		sh2				11			
7					re2	re2				
8	sh1		sh2				12			
9					re1	re1				
10				re3	re3	re3				
11				re7	sh6/re7	re7		10		
12					sh6/re6	re6		10		

Fig.2: A LR Parsing Table obtained from Fig.1 Grammar

'c(Stream)' are created by a parent process 'a' and they run in parallel. Note that the definition of process 'c' in statement (3) is going to instantiate the variable 'Stream' in 'c(Stream)' with '[A | Stream1]'. In such a case the execution of process 'c' will be suspended until 'Stream' has been instantiated by process 'b(Stream)'. By the recursive process call in the body of definition (2), process 'b' continues to produce the atom 'x' and places it on stream. The atom 'x' is sent to process 'c' by the GHC stream communication; process 'c' continues to consume atom 'x' on stream.

```
(1) a:- true |
        b(Stream),
        c(Stream).
(2) b(Stream):- true |
        Stream=[ x|Rest ],
        b(Rest).
(3) c([ A|Stream1 ]):- true |
        c(Stream1).
```

Fig.3: Typical Statement of GHC

4 New Parallel Generalized LR Parsing Algorithm

The new parallel parsing algorithm is a natural extension of Tomita's algorithm [Tomita 86] and enables us to achieve greater parallel performance. In our algorithm, if a parsing sentence contains syntactic ambiguities, two or more parsing processes will run in parallel.

4.1 Tree Structured Stacks

To avoid the duplication of parsing processes, the new algorithm makes use of Tree Structured Stacks (TSS) instead of Tomita's Graph Structured Stacks (GSS). An example of TSS is given in the form of a list data structure in GHC. Consider the following generalized LR parsing, using for the input sentence, the grammar and the table in Figure 1 and Figure 2 respectively. After the parser has shifted the word 'with', the following two stacks with the same top state '6' will be obtained:

```
Sentence :
" I open the door with a key . "

(1) top < [ 3,s,0 ]
(2) top < [ 6,p,12,np,8,v,4,np,0 ]
```

Fig.4: Two Stacks to be Merged

We will merge these two stacks and get the following TSS:

```
(3) [ 6,p, [12,np,8,v,4,np,0],
           [3,s,0] ]
```

Figure 5 shows an image of the TSS above.

4.2 Stack Operations on Stream

In order to merge the stacks, Tomita's algorithm must synchronize the parsing processes for shift operations, thereby reducing parallel performance. To solve this problem, we have developed an improved parallel generalized LR algorithm that involves no waiting for shift operations before merging many stacks. The new algorithm is made possible by a GHC stream communication mechanism.

Through this stream communication mechanism, a process that has completed a 'shift N' first has the privilege of proceeding to subsequent actions and continuing to do so until a reduce action pops an element with state 'N'

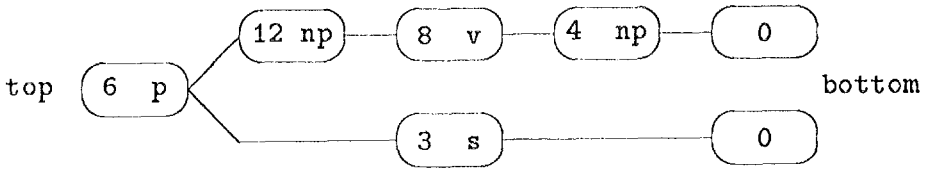

Fig.5 : A Tree Structured Stack

into the stack. If other parsing processes carry out the same 'shift N' actions, their stacks will be merged into the position in which the "privileged" process had, by the 'shift N' action, inserted an element. The merging of stacks is thus greatly facilitated by the GHC stream communication mechanism.

To begin parsing, we will create a sequence of goal processes, namely `p1,p2,...,pn,p$`, each of which corresponds to a look-ahead preterminal of an input sentence (referred to hereafter as a parent process). The stack information is sent from process p1 to process p$ using the GHC communication mechanism. Each parsing process receives the TSS from its input stream, changes the TSS in parallel according to the LR table entry, and sends the results as the output stream — which in turn becomes the input stream of the adjacent process. The stream structure is as follows:

```
[ Stack1,Stack2,...,Stackn | Stream ]
```

where `Stacki` is a TSS like (3) or a simple stack like (1).

Consider the case where a shift-reduce conflict occurs and the parent process produces two subprocesses which create stacks (1) and (2) (Fig.4). In order to merge both stacks, Tomita's parser forces the parent process to wait until the two subprocesses have returned the stacks (1) and (2). Our algorithm attempts to avoid such synchronization: even though only one subprocess has returned stack (2), the parent process does not wait for stack (1), but generates the following stack structure and sends it on to the output stream (which in turn becomes the input stream of the adjacent process). The adjacent process can then perform its own operations for the top of stack (2) on the input stream. Thus the new algorithm achieves greater parallel performance than its predecessor.

Output Stream of Parent Process :

```
[ [6,p | Tail] | Stream ]
```

where '6,p' are the top two elements of the stack (2).

Note that 'Tail' and 'Stream' remain undefined until the other subprocess returns stack (1). If the adjacent process wants to retrieve 'Tail' and 'Stream' after processing the top of stack (2), the process will be suspended until 'Tail' and 'Stream' have been instantiated by the rest of stacks (2) and (1).

This kind of synchronization is supported by GHC. Let's suppose the adjacent process receives the above output stream from the parent process. Before the parent process has generated stack (1), the adjacent process can execute 5 steps for the top two elements of stack (2) (see Figure 6). During the execution of the adjacent process, the parent process will be able to run in parallel.

As soon as the parent process receives stack (1) with the same top elements '6,p' of stack (2), it instantiates the variables 'Tail' and 'Stream' and merges '6,p', getting the same TSS shown in Figure 5:

```
Tail   = [ [ 12,np,8,v,4,np,0 ],
          [ 3,s,0 ] ]
Stream = [ ]
```

We need to consider the case where the top element of stack (1) is different from that of stack (2). For example, suppose that stack (1)

State	Symbol	Action	Stream
6	det	sh 1	[[1,det,6,p | Tail] | Stream]
1	noun	sh 5	[[5,noun,1,det,6,p | Tail] | Stream]
5	$	re 4	[[6,p | Tail] | Stream]
6	np	goto 11	[[11,np,6,p | Tail] | Stream]
10	$	re 7	[Tail | Stream]

Fig.6 The Parsing Process with an Incomplete Stack

is [8,p,3,s,0], then the variables 'Tail' and 'Stream' will be instantiated as follows:

```
Tail   = [ 12,np,8,v,4,np,0 ]
Stream = [ [ 8,p,3,s,0 ] ]
```

In this case, we have two simple stacks in the stream.

5 Comparison of Parallel Parsing Performance

In this section, we will show by way of a simple example that our algorithm has greater parallel parsing performance than Tomita's original algorithm. Consider the parallel parsing of the input sentence " I open the door with a key ", using a grammar in Figure 1 and a table in Figure 2. As the input sentence has two syntactic ambiguities, the parsing process encounters a shift-reduce conflict of the LR table and is broken down into two subprocesses. Figure 7 shows the state of the parsing process and grammatical symbols which are put into a stack. When the process has the state 12 and the look-ahead preterminal 'p', the process encounters a 'sh 6/re 6' conflict. Then it is broken down into two subprocesses: the first process performs the 'sh 6' operation and goes to state 6, and the other performs the 're 6' operation. The second process also goes to the state 6 after performing 'goto 9','re 1','goto 3', and 'sh 6' operations. The processes that run according to the simple parallel LR parsing algorithm are shown in Figure 7(a).

We can see that the two processes perform the same operations after performing the 'sh 6' operations. If we do not merge these kinds of processes, we will face an explosion in the number of processes. Tomita's algorithm (shown in Figure 7(b)) can avoid the duplication of parsing processes by merging them into one process. However, the algorithm needs a synchronization that decreases the number of processes which are able to run in parallel. On the other hand, our algorithm (shown in Figure 7(c)) does not require such synchronization as long as these processes do not try to reduce the incomplete part of a stack. In this example, two processes run in parallel after a 'sh 6/re 6' conflict has occurred. Then, an incomplete stack like [6,p|Tail] is created, with the upper process in Figure 7(c)

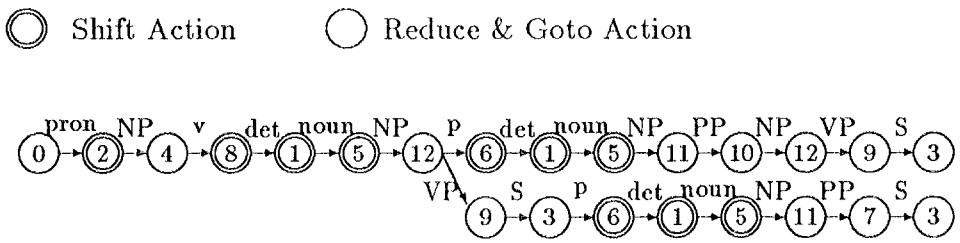

Fig.7(a): A Simple Parallel LR Parsing

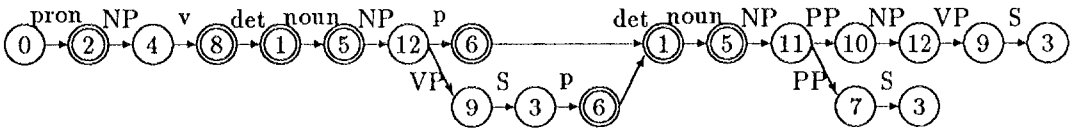

Fig.7(b): A Parallel Parsing Guided by Tomita's Algorithm

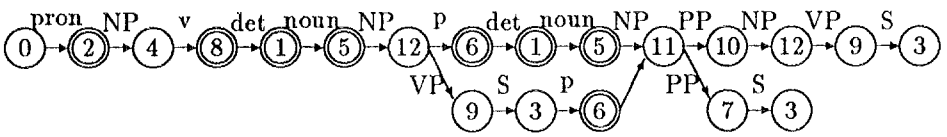

Fig.7(c): Our Parallel Parsing

performing the 'sh 1', 'sh 5', and 're 4' stack operations while the lower process calculates its incomplete part. After finishing the 'sh 6' operation of the lower process, the incomplete part `'Tail'` will be instantiated and thus we obtain the following tree structured stack:

```
[ 6,p, [ 12,np,8,v,4,np,0 ],
       [ 3,s,0 ] ]
```

It is remarkable that our algorithm takes less time to than either the simple algorithm or Tomita's to generate the first result of parsing. The reason is that our algorithm can analyze two or more positions of an input sentence in parallel, which is a merit when parsing with incomplete stacks.

The complexity of our algorithm is identical to that of Tomita's [Johnson 89]. The only difference between the two is the number of processes that run in parallel. So if we simulate the parsing of our algorithm and that of Tomita's on a single processor, the time of parsing will be exactly the same.

6 Conclusion

We have described a new parallel generalized LR parsing algorithm which enables us to achieve greater performance of parallel parsing than Tomita's algorithm. Simulations indicate that most processes run in parallel and that the number of suspended processes is very small, but the experiment must be carried out using many processors. Fortunately, ICOT (headquarters of the Japanese fifth generation project) has offered us the possibility of using the Multi-PSI machine composed of 64 processors. We are now preparing to conduct such an experiment to put our new parsing algorithm to the test.

References

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

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

[Johnson 89] Mark Johnson :*The Computational Complexity of Tomita's Algorithm* International Workshop on Parsing Technologies, pp.203-208 (1989)

[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)

[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)

[Tanaka 89] Tanaka,H. Numazaki,H.:*Parallel Generalized LR Parser (PGLR) based on Logic Programming,* Proc. of First Australia-Japan joint Symposium on Natural Language Processing, pp. 201-211 (1989)

[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)