TR-318

Parsing Gapping Grammars in Parallel

by

Y. Matsumoto

November, 1987

Parsing Gapping Grammars in Parallel

Yuji Matsumoto
Institute for New Generation Computer Technology
Mita Kokusai Building 21F
1-4-28 Mita, Minato-ku, Tokyo, 108, Japan
e-mail: ymatsumoto@icot.junet

1. Introduction

This paper describes a parallel parsing method for Gapping Grammars (GGs) [Dahl 84a] [Dahl 84b] which is suitable for committed-choice parallel logic programming languages such as Guarded Horn Clauses (GHC) [Ueda 85] and Parlog [Clark 84]. GG is a very powerful grammar formalism that enables grammar writers to specify grammar rules concentrating on the constituents of the input sentences that are not necessarily adjacent. The parallel parsing method presented here is based on the author's previous work on a parallel parsing algorithm for context-free grammars [Matsumoto 86].

The parsing algorithm for context-free grammars will be described first by an interpreter written in Prolog. This is a metadescription of the parsing algorithm and the final parsing program can be obtained by partially evaluating (or translating) the given grammar rules accordingly to the interpreter. Although the programs are written in Prolog, it is not essential. The Prolog program obtained through the translation is a deterministic Prolog program with arguments of specific Input/Output modes and can easily be transferable to parallel logic programming languages.

We give a slight modification to the interpreter enabling it to handle GGs. Partial evaluation of the interpreter with GG rules produces an efficient bottom-up parser for the given GG. The derived program does not involve any backtracking and all the Prolog goals are inherently operable in parallel. When a clause corresponding to a grammar rule with a gap is invoked, the parsing process splits into distinct parallel processes. However, all the other parts in the input sentence are shared by these processes. The parser inherits most of the advantages of our parallel parser for context-free grammars. The basic algorithm employs a bottom-up strategy with top-down prediction, which contributes to the reduction of parsing space. Nonterminal symbols are represented as goals of Prolog (or GHC etc) that operate as parallel processes. Each of these processes corresponds to a parse tree with the nonterminal symbol as its root. They need not be kept as side-effects. Moreover, no identical parse trees are constructed duplicatedly.

2. Metadescription of Parser

Basically, the algorithm for the parallel parsing of Gapping Grammars is an extension of our parallel parsing algorithm for context-free grammars. Appendix 1 shows the main part of the interpreter written in Prolog that explains the parsing algorithm for context-free grammars. Grammar rules are expressed in reverse order (the part normally called the right-hand side of a grammar rule is written on the left-hand side) as in the last clauses in Appendix 1. For instance,

```
rule([ np, vp, '=>', s ]).
```

corresponds to the context-free grammar rule normally written in DCG like:

```
s --> np, vp.
```

To avoid confusion, we will use the terminology, head and body (or parent and sons) of a grammar rule. In this example, 's' is the head of the grammar rule and 'np,vp' is the body.

Here is the explanation of the program shown in Appendix 1. We do not deal with terminal and nonterminal symbols differently. The input sentence represented as a list of any words and nonterminal symbols and is given to 'start'. The predicate 'parse' gives each of the symbol in the input list to 'expand'. Two jobs are allocated to the symbol, specified by the predicates 'type1' and 'type2', which comprises the main part of the algorithm.

'Type1' selects grammar rules that have the specified symbol as their leftmost son, where there are two possibilities. One case is when the son is the leftmost son of several sons in the rule, and the other is when the son is the only son in the rule. The former case is handled by 'bag_of' (the definition is the same as bagof except it returns [] when there is no answer substitution), which picks up a grammar rule whose first element of the body coincides with the given word or nonterminal symbol and returns the remaining elements in the grammar rule after taking off the first element. Top-down prediction is also taken into consideration, though we omitted it in the program of Appendix 1. Suppose that the given symbol is 'det,' it returns the structures [noun,'=>',np] and [noun,rel_c,'=>',np], (provided a noun phrase is expected by the preceding context, when the top-down prediction is taken into consideration). The meaning of these structures is straightforward, that is, after getting a determiner, a noun phrase is obtained if we have a noun or if we have a noun and a relative clause. Note that the produced fragments of grammar rules are paired with the received list, 'InS,' since it should be the input list for the head symbol when it is constructed and expanded. The latter case mentioned above is handled by 'immediate_rules,' that immediately 'expand' the head of the rules since no other element is left in the body.

The second job for 'expand' (corresponding to 'type2') is to receive the data structures such as the ones shown above and to produce another data structure or to call 'expand' with the head of the newly completed grammar rule. All the data produced by these two jobs are merged into one list, which is received by the word or the grammatical symbol that follows these processes.

Readers who are familiar with Kay's Chart Parsing [Kay 80] may have noticed the similarity between his method and ours. A call of 'expand' with a grammatical symbol coincides with the creation of an inactive arc of that symbol. Creation of a data structure in our parser corresponds to the creation of an active arc. The location of an inactive arc is represented by the locus and length of the arc, and it is represented by the list the corresponding process (the call of expand) receives and the list of the fragments of grammar rules it passes to the next process. The locus of an active arc is represented by the list which is paired with it and the list which it is put into. This shows that the complexity of our algorithm is almost the same as that of Chart Parsing. The advantage of our system is that inactive arcs are implemented as parallel processes and active arcs are localized as streams (ie, lists) received by inactive arcs so that a global table or side-effect is not necessary, which makes our parser suitable for parallel implementation. The actual implementation is done by partially evaluating or compiling the metainterpreter and the grammar rules into a specialized parallel logic program. The specialization is mainly done in two aspects. One is to encode the fragments of grammar rules that appear in the parsing process into simpler symbols (identifiers). Since the set of grammar rules is finite and the fragments of grammar rules appear only in the form of postfixes of the grammar rules, it is possible to assign an distinct symbol to each of fragments of grammar rules. The other is to specialize both types of processes (i.e. type1 and type2 processes) for each nonterminal symbols. Since the metainterpreter is fixed, we have developed a translator that produces the final program from a set of context-free grammar rules. Appendix 2 shows such a program obtained from the program of Appendix 1.

3. Extending Parser for Gapping Grammars

We show that a little modification of the context-free parser enables it to handle Gapping Grammars. A Gapping Grammar rule is, for instance, written as follows:

```
    rule([coconj,gap(1,[]),object, '=>',
                    object,coconj,gap(1),object]).
```

If we follow the original description of GG rules in [Dahl 84a][Dahl 84b], it is written like:

```
    object,coconj,gap(G),object --> coconj,gap(G),object.
```

In our bottom-up interpretation, the meaning of this rule is that a configuration in the input sentence consisting of a 'coconj' (a coordinating conjunction) and an 'object' that have any sequence of words (possibly a null sequence) in between can be rewritten to the sequence shown in the head of the rule, provided gap(1,[]) must be the same sequence of the words represented by gap(1). Note that the head of the grammar rule is not a single grammatical symbol but a sequence of symbols. The first argument in gap(1,[]) is necessary to identify the position of the specific gap when there is more than one gap in grammar rules. The second argument works as a stack to keep the skipped words and is initially an empty list.

In order to deal with such rules, 'start', 'parse' and 'type2' are redefined as in Appendix 3. There are mainly three modifications on the context-free metainterpreter.

419

(1) When heads of grammar rules are called after successful analysis of bodies of grammar rules, 'parse' is used instead of 'expand'. This is because that the head of a grammar rule is not only a single nonterminal symbol but is generally a sequence of grammatical symbols.

(2) A new predicate 'gg' is defined and added to each place where a word in the input sentence is expanded. Predicate 'gg' has four arguments, of which the first is the input stream, the second is the word it is skipping and the third and the fourth are the output streams. The third argument is the output to the process ('expand') that is dealing with the very same word, and the fourth argument is the output to the next 'gg'. For a 'gg' receiving a data beginning with a 'gap', there are two options, whether to pass over the 'gap' by skipping the word it is currently dealing with, or to finish skipping the 'gap'. A 'gg' performs both of the jobs since the parser is intended to produce all the possible parsing results. To skip the word, it adds the skipped word (ie, 'Word') on top of the stack in the 'gap' it is passing to the next 'gg'. The main definition of 'gg' is described by the second and third clauses, in which a new 'gap' with the skipped word on top of the stack is put at the fourth argument. These two clauses are for the different occurrences of 'gap' in the bodies of grammar rules. Firstly, if 'gap' appears as the last symbol in the body of a rule, the head can be immediately expanded. It is done by replacing the corresponding 'gap' in the head with the already skipped words and by calling 'parse' for the head. Secondly, if 'gap' is not the last symbol in the body, 'gap' must be also passed to the output stream at the third argument of 'gg'. This is actually the case where skipping of words are completed just before the word the 'gg' is dealing with.

(3) The last modification is to add the clauses for handling gaps to the definition of 'type2.' When 'gap' is encountered by 'type2' and if the next element to the gap is the same symbol as the one it is handling, it suppose that skipping the words by the 'gap' has completed and discards the 'gap' and the symbol next to the 'gap', after replacing the corresponding 'gap's in the head by the skipped words.

The actual parsing program is obtained by partially evaluating the metainterpreter and the grammar rules in the same way as the case of context-free grammars.

4. Gapping Grammars and Extraposition Grammars

Extraposition Grammars proposed by Pereira [Pereira 81] provide a way to [illegible] the left extrapositions. In the syntactic viewpoint, Gapping Grammars subsume Extraposition Grammars. For example, an Extraposition Grammar rule:

```
rel_marker ... trace --> rel_pro.
```

is equivalent to the following Gapping Grammar rule:

```
rel_marker,gap(G),trace --> rel_pro,gap(G).
```

which is equivalently written in our way of writing grammar rule as follows:

```
    rule([rel_pro,gap(1,[]),'=>',rel_marker,gap(1),trace]).
```

Although the metainterpreter of Gapping Grammars shown in the preceding section is able to handle this type of grammar rules, it becomes much simpler if this type of grammar rules are prohibited. When 'gap's are allowed only inside of the bodies of grammar rules, the job done by the second clause of 'gg' is no longer necessary. This restriction changes the definition of 'start' and 'gg' as follows:

```
start([],End,End).
start([Word|Rest],In,End) :-
    gg(In,Word,Out1),
    expand(Word,In,Out2),
    append(Out1,Out2,Out),
    start(Rest,Out,End).

gg([],_,[]).
gg([([gap(N,X)|Rule],In)|R],Word,
        [([gap(N,[Word|X])|Rule],In)|R1]) :- !,
    gg(R,Word,R1).
gg([_|Rest],Word,Rest1) :- gg(Rest,Word,Rest1).
```

In this new definition, the only job 'gg' does is to skip the word it is handling. The job to finish skipping is actually done by the modification on the definition of 'start', where 'expand' receives the same input stream as the one 'gg' receives.

5. Discussion

The modifications to the parallel parser enables it to deal with Gapping Grammars. We have to note that the Gapping Grammars we have been discussing are not exactly the same as the Gapping Grammars proposed by [Dahl 84a, 84b]. As is seen from the explanation in Section 2 and 3, only the surface words are skipped. Although it is not a necessary condition for the algorithm, this restriction greatly helps to reduce the search space.

There are another restriction and one augmentation to the form of grammar rule. The first element of the body must not be a 'gap' in our algorithm. This is because the parsing process operates in a bottom-up manner and the first element in the body is the key to select grammar rules. However, the first element of the head need not be a nonterminal symbol unlike the original definition of Gapping Grammars. As a matter of fact, such grammar rules are quite convenient to describe typical phenomena frequently occurring in Japanese sentences. In Japanese, modal expressions are always put at the end of a sentence. Therefore, when a sentence includes some important modal information, a certain signal can be put at an earlier place in the sentence. For example, 'kesshite' is an adverb which stresses that the speaker is uttering a negative sentence. This can be expressed by the following grammar rule.

```
rule([kesshite,gap(1,[]),neg_aux,'=>',gap(1),neg_aux(stressed)]).
```

These kinds of rules are never invoked unless the input sentence contains the key word. And once it is invoked, almost no useless process is generated when the top-down prediction is incorporated in our system.

We have omitted the discussion on incorporating top-down prediction in the algorithm. This is because we did not like to make the sample programs too complicated. The idea is found in the authors previous paper on parallel parsing of context-free grammars [Matsumoto 86], and the augmentation can be discussed independently of the main parsing algorithm.

Appendix 1: Context-Free Parser Interpreter

```
start(Sent) :-
     parse(Sent,[begin],Stream),
     fin(Stream).

parse([],End,End).
parse([Word|Rest],In,End) :-
     expand(Word,In,Out),
     parse(Rest,Out,End).

expand(Cat,InS,OutS) :- type1(Cat,InS,OutS1),type2(Cat,InS,OutS2),
     append(OutS1,OutS2,OutS), !.

type1(_,[],[]).
type1(Cat,InS,Stream) :-
     bag_of((Rule,InS),(rule([Cat|Rule]),\+ Rule=['=>'|_]),Stream1),
     immediate_rules(Cat,InS,Stream2),
     append(Stream1,Stream2,Stream).

type2(_,[],[]).
type2(s,[begin|Rest],[end|Rest1]) :- !,
     type2(s,Rest,Rest1).
type2(Cat,[([Cat,'=>',Head],InS)|Rest],OutS) :-
     expand(Head,InS,Out1), !,
     type2(Cat,Rest,Out2),
     append(Out1,Out2,OutS).
type2(Cat,[([Cat|R_Rule],InS)|Rest],[(R_Rule,InS)|Rest1]) :- !,
     type2(Cat,Rest,Rest1).
type2(Cat,[_|Rest],Rest1) :-
     type2(Cat,Rest,Rest1).

fin([]).
fin([end|Rest]) :-
     write('parsed  '), !,
     fin(Rest).
fin([_|Rest]) :-
     fin(Rest).

rule([ np, vp, '=>', s ]).
rule([ det, noun, '=>', np ]).
rule([ det, noun, rel_c, '=>', np ]).
rule([ verb, '=>', vp ]).
rule([ verb, np, '=>', vp ]).
rule([ rel_pro, vp, '=>', rel_c ]).
rule([ every, '=>', det ]).
rule([ a, '=>', det ]).
rule([ the, '=>', det ]).
rule([ man, '=>', noun ]).
rule([ woman, '=>', noun ]).
rule([ walks, '=>', verb ]).
rule([ loves, '=>', verb ]).
rule([ that, '=>', rel_pro ]).
```

Appendix 2: Transferred Parsing Program (in GHC)

```
np(X,A1,B1) :- true |
     np1(X,A1,C1),
     np2(X,C1,B1).

s([],A,B) :- true | A=B.
s([begin(X)|T],A1,B1) :- true |
     A1=[end(X)|C1],
     s(T,C1,B1).
s([_|T],A1,B1) :- true |
     s(T,A1,B1).

np1(X,A1,B1) :- true |
     A1=[id1(X)|B1].

np2([],A,B) :- true | A=B.
np2([id5(X)|T],A1,B1) :- true |
     vp(X,A1,C1),
     np2(T,C1,B1).
np2([_|T],A1,B1) :- true |
     np2(T,A1,B1).

vp([],A,B) :- true | A=B.
vp([id1(X)|T],A1,B1) :- true |
     s(X,A1,C1),
     vp(T,C1,B1).
vp([id6(X)|T],A1,B1) :- true |
     rel_c(X,A1,C1),
     vp(T,C1,B1).
vp([_|T],A1,B1) :- true |
     vp(T,A1,B1).
```

```
det(X,A1,B1) :- true |
     A1=[id2(X),id3(X)|B1].

noun([],A,B) :- true | A=B.
noun([id2(X)|T],A1,B1) :- true |
     np(X,A1,C1),
     noun(T,C1,B1).
noun([id3(X)|T],A1,B1) :- true |
     A1=[id4(X)|C1],
     noun(T,C1,B1).
noun([_|T],A1,B1) :- true |
     noun(T,A1,B1).

rel_c([],A,B) :- true | A=B.
rel_c([id4(X)|T],A1,B1) :- true |
     np(X,A1,C1),
     rel_c(T,C1,B1).
rel_c([_|T],A1,B1) :- true |
     rel_c(T,A1,B1).

verb(X,A1,B1) :- true |
     A1=[id5(X)|C1],
     vp(X,C1,B1).

rel_pro(X,A1,B1) :- true |
     A1=[id6(X)|B1].

every(X,A1,B1) :- true |
     det(X,A1,B1).

a(X,A1,B1) :- true |
     det(X,A1,B1).

the(X,A1,B1) :- true |
     det(X,A1,B1).

man(X,A1,B1) :- true |
     noun(X,A1,B1).

woman(X,A1,B1) :- true |
     noun(X,A1,B1).

walks(X,A1,B1) :- true |
     verb(X,A1,B1).

loves(X,A1,B1) :- true |
     verb(X,A1,B1).

that(X,A1,B1) :- true |
     rel_pro(X,A1,B1).
```

Appendix 3: Redefined Part for Parsing Gapping Grammar

```
start(Sent) :-
     start(Sent,[begin],Stream),
     gg(Stream,_,New_stream,_),
     fin(New_stream).

start([],End,End).
start([Word|Rest],In,End) :-
     gg(In,Word,New_in,Out1),
     expand(Word,New_in,Out2),
     append(Out1,Out2,Out),
     start(Rest,Out,End).

parse([],End,End).
parse([List|Rest],In,End) :-
     list(List),
     start(List,In,Out),!,
     parse(Rest,Out,End).
parse([Word|Rest],In,End) :-
     expand(Word,In,Out),!,
     parse(Rest,Out,End).

gg([],_,[],[]).
gg([([gap(N,X),'=>'|Head],In)|R],Word,
       New_in,[([gap(N,[Word|X]),'=>'|Head],In)|R1]) :- !,
     replace(N,Head,X,Head1),
     parse(Head1,In,Nin),
     append(Nin,Nxt_in,New_in),
     gg(R,Word,Nxt_in,R1).
gg([([gap(N,X)|Rule],In)|R],Word,[([gap(N,X)|Rule],In)|Nxt_in],
          [([gap(N,[Word|X])|Rule],In)|R1]) :- !,
     gg(R,Word,Nxt_in,R1).
gg([Id|Rest],Word,[Id|New_in],Rest1) :-
     gg(Rest,Word,New_in,Rest1).

type2(_,[],[]).
type2(s,[begin|Rest],[end|Rest1]) :- !,
     type2(s,Rest,Rest1).
type2(Cat,[([Cat,'=>'|Head],InS)|Rest],OutS) :-
     parse(Head,InS,Out1), !,
     type2(Cat,Rest,Out2),
     append(Out1,Out2,OutS).
type2(Cat,[([Cat|R_Rule],InS)|Rest],[(R_Rule,InS)|Rest1]) :- !,
     type2(Cat,Rest,Rest1).
type2(Cat,[([gap(N,X),Cat,'=>'|Head],InS)|Rest],OutS) :- !,
     replace(N,Head,X,Head1),
     parse(Head1,InS,Out1),
     type2(Cat,Rest,Out2),
     append(Out1,Out2,OutS).
type2(Cat,[([gap(N,X),Cat|Rem],InS)|Rest],[(Rem1,InS)|Rest1]) :- !,
     replace(N,Rem,X,Rem1),
     type2(Cat,Rest,Rest1).
type2(Cat,[_|Rest],Rest1) :-
     type2(Cat,Rest,Rest1).

replace(N,R,X,L) :-
     reverse(X,Y),
     rep(N,R,Y,L).

rep(N,[gap(N)|R],X,[X|L]) :- !,
     rep(N,R,X,L).
rep(N,[A|R],X,[A|L]) :- !,
     rep(N,R,X,L).
rep(_,[],_,[]).
```

References

[Clark 84] Clark, K.L. and S.Gregory, "PARLOG: Parallel Programming in Logic," Research Report DOC 84/4, Imperial College, April 1984.

[Dahl 84a] V. Dahl and H. Abramson, "On Gapping Grammars," Proc. 2nd International Conference on Logic Programming, Uppsala, Sweden, pp.77-88, 1984.

[Dahl 84b] V. Dahl, "More on Gapping Grammars," Proc. the International Conference on Fifth Generation Computer Systems, Tokyo, Japan, pp.669-677, 1984.

[Kay 80] M. Kay, "Algorithm Schemata and Data Structures in Syntactic Processing," Technical Report CSL-80-12, Xerox PARC, Oct. 1980.

[Matsumoto 86] Y. Matsumoto, "A Parallel Parsing System for Natural Language Analysis," Proc. 3rd International Conference on Logic Programming, London, 1986.

[Pereira 81] F. Pereira, "Extraposition Grammars," AJCL, Vol.7, No.4, October-December, pp.243-256, 1981.

[Ueda 85] K. Ueda, "Guarded Horn Clauses," ICOT Tech. Report TR-103, Institute for New Generation Computer Technology, Tokyo, 1985. A revised version is in Proc. Logic Programming '85, E. Wada (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, pp.168-179, 1986.