Multi-Parser Architecture for Query Processing

Kui Xu*, Fuliang Weng*, Helen Meng**, and Po Chui Luk**

*Intel China Research Center, Intel Corp, Beijing, China
**Human-Computer Communications Laboratory
The Chinese University of Hong Kong, Hong Kong SAR, China.
{kui.xu, fuliang.weng}@intel.com; {hmmeng, pcluk}@se.cuhk.edu.hk

Abstract

Natural language queries provide a natural means for common people to interact with computers and access to on-line information. Due to the complexity of natural language itself, the grammar that describes the query language can be very complex. The traditional way of using a single grammar leads to an inefficient, fragile, and often very big parsing system. Multi-Parser Architecture (MPA) intends to alleviate these problems, and the modularized MPA also has the advantage of easier portability to new domains and distributed computing. In this paper, we investigate the effect of using different types of parsers on different types of query data in MPA. Three data sets and two types of sub-parsers, particularly a predictive cascading composition for pre-compiled Earley parsers, have been examined. Results show that partitioning grammars leads to superior speed performance for the Earley-style parser across the three data sets. GLR parser is faster than Earley parser in the partitioned case, but it can lead to an excessive memory usage for the un-partitioned case.

1. Introduction

Natural language queries, sometimes as the back-end to speech recognizers, provide a natural means for common people to interact with computers and access to on-line information. Due to the complexity of natural language itself, the grammar that describes the query language can be very complex. The traditional way of using a single grammar leads to an inefficient, fragile, and often very big processing system. These problems become more apparent, with the increasing demand for natural language applications over the inter-net.

Various methods to deal with these parsing issues have been studied, e.g., [1],[2],[10],[5]. The proposed MPA by [10],[11],[4]intends to alleviate these problems simultaneously by partitioning a single grammar into multiple sub-grammars and composing sub-parsers for the corresponding sub-grammars. The MPA is highly modularized, and has the advantage of re-usable sub-grammars, which is conducive towards distributed computing as well as portability to new application domains. Our current work presents several enhancements: (i) a methodology for automatic grammar partitioning; (ii) an improved parser composition method known as predictive cascading; and (iii) the use of an Earley parser in addition to the GLR parser used previously. Experiments were conducted with three data sets – a keyword list, a semantic grammar and a syntactic grammar. Results show that grammar partitioning with composition of Earley parsers can speed up processing substantially for both semantic and syntactic grammars, compared to the un-partitioned grammars. Furthermore, while the GLR parser is faster than Earley; the latter utilizes memory more efficiently.

2. The Multi-Parser Architecture

MPA involves the two main processes of grammar partitioning and parser composition. Grammar partitioning divides a grammar into multiple sub-grammars. Each sub-grammar is used by its corresponding sub-parser, and their parsing results are composed to produce an overall parsing output for a natural language query. The interaction among sub-grammars/sub-parsers is achieved by using a virtual terminal technique. The virtual terminal is essentially a non-terminal, but acts as if it were a terminal. The INPUT set to a sub-grammar is a set of virtual terminals that were previously parsed by other sub-grammars. The OUTPUT set of a sub-grammar is a set of non-terminals that are parsed based on this sub-grammar, and used by other sub-grammars as their INPUT sets. Hence a partition (subset) of production rules can be viewed as a multi-valued function – it takes the virtual terminals in the INPUT set as input, and returns a set of non-terminals in the OUTPUT set as output. Results on manually partitioned grammars were presented previously [11],[4]. In this paper, we will present a method and results for automatic grammar partitioning.

Two methods for parser composition were also presented: composition by cascading and composition by predictive pruning [11],[4]. Cascading is a bottom-up parsing procedure starting from the terminal level to and moves to the SENTENCE level. It begins by converting the input sentence into a lattice called LMG, and invokes parsers at each lattice position. During parsing, newly created virtual terminals (vt) are added dynamically to the stack and LMG. A stack stores the terminals and virtual terminals according to the topological order of the LMG. Cascading has the advantage of parser robustness, but is relatively slow due to excessive parser invocation at each lattice (LMG) position. To avoid this, predictive composition is a top-down procedure where the caller sub-parser invokes a callee sub-parser only if the latter satisfy the constraint that the input node must be its left corner. The ones that do not satisfy the constraint are pruned. While the top-down approach has the advantage of execution efficiency, it may at times be too constrained to be robust. In subsection 2.2, we present our improved parser composition method known as predictive cascading.

2.1. Automatic Grammar Partitioning

To see the parser effect on automatically partitioned grammar, we use a syntactic grammar derived from the ATIS subset in

---

1 From hereon all the occurrences of Earley parsers refers to the pre-compiled Earley parser.
the Penn Treebank, which is different from the semi-
automatically derived semantic grammar [6] used in our
previous work [11],[4].

The Penn Treebank contains a subset of the ATIS-3
corpus2, which is a set of 577 parsed queries including Class
A (self-contained) and Class D (dependent on discourse
context) queries. Parse trees of these queries are provided,
with the tree terminals being part-of-speech (POS) tags3. A
parse tree example is shown in Figure 1. Our grammar rules
are extracted from the parse trees, and for simplicity we
ignore the null elements XXX as well as co-indexing in the
grammar rules (hence WHNP-1 is the same as WHNP, see
Figure 2). As we extract the grammar rules from the training
parse trees, we also record the frequency of invocation
between rules, e.g. Rule 2 in Figure 2 (PP-DIR\rightarrow IN NP) has
called Rule 1 (NP \rightarrow NNP) once. Here the INPUT set of Rule
2 is the non-terminal NP and the OUTPUT set is PP-DIR.

Our automatic grammar partitioning procedure begins
with the set of finest grammar partition, each of which
contains exactly one grammar rule. Then we attempt to
cluster the sub-grammars to form larger ones based on the
frequency of their interaction. In other words, we want to
cluster grammar partitions that have frequent caller/callee
interactions into a larger sub-grammar. The procedure
references a calling matrix, where the entry at row i and
column j is the frequency of sub-grammar i calling sub-
grammar j (see Figure 3).

0: NP \rightarrow DT NNS
1: NP \rightarrow NNP
2: PP-DIR \rightarrow IN NP
3: PP-DIR \rightarrow TO NP
4: WHNP \rightarrow WDT
5: PP-LOC \rightarrow IN NP
6: VP \rightarrow VBP PP-LOC
7: S \rightarrow VP
8: SBAR \rightarrow WHNP S
9: NP \rightarrow NP PP-DIR PP-DIR SBAR
10: VP \rightarrow VB NP

Figure 4: Updated calling matrix of the grammar in
Figure 2.

0 1 2 3 5 6 7 9 10 11
0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
2 0 1 0 0 0 0 0 0 0 0
3 0 1 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0
5 0 1 0 0 0 0 0 0 0 0
6 0 0 0 0 1 0 0 0 0 0
7 0 0 0 0 0 0 1 0 0 0
8 0 0 0 0 0 0 0 1 0 0
9 1 0 1 1 0 0 1 0 1 0
10 0 0 0 0 0 0 0 0 1 0

Figure 2: Grammar rules extracted from the parse tree in
Figure 1.

0 1 2 3 4 5 6 7 8 9 10
0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
2 0 1 0 0 0 0 0 0 0 0
3 0 1 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0
5 0 1 0 0 0 0 0 0 0 0
6 0 0 0 0 1 0 0 0 0 0
7 0 0 0 0 0 0 1 0 0 0
8 0 0 0 0 0 0 0 1 0 0
9 1 0 1 1 0 0 1 0 1 0
10 0 0 0 0 0 0 0 0 1 0

Figure 3: Calling matrix of the grammar in Figure 2.

0 1 2 3 4 5 6 7 9 10 11
0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0
2 0 1 0 0 0 0 0 0 0 0
3 0 1 0 0 0 0 0 0 0 0
5 0 1 0 0 0 0 0 0 0 0
6 0 0 0 1 0 0 0 0 0 0
7 0 0 0 0 1 0 0 0 0 0
9 1 0 1 1 0 0 0 0 1
10 0 0 0 0 0 0 1 0 0 0
11 0 0 0 0 0 0 1 0 0 0

Figure 4: Updated calling matrix of the grammar in
Figure 2.

• 2. The next step in our clustering procedure is an
iterative step. We find the maximum value in the calling
matrix, and merge the two sub-grammars into a new
grammar. Merging is conditioned upon two thresholds:
(i) the frequency in the matrix must be greater than the
minimum count of X (set at 3) ; and (ii) the size of the
merged grammar must lie below the threshold Y (set at
100) to avoid infinite growth. Grammar size is measured
according to Equation (1): where P is the set of
production rules, and length(x) is the length of string x.

}\text{length(x)}
3. Experiments

For experiments, two different styles of parsers has been used. One is Earley parser, where the prediction step is pre-compiled. The other parser is GLR parser \[6,7\]. These two parsers have their own different characteristics. Earley parser does not encode all the prefix paths in a table, while GLR parser tries to do it as much as possible. This usually makes GLR faster but bigger, compared with Earley parser.

Three different data sets are used for our experiment. The first one is a Chinese keywords list grammar used for key words extraction; the second one is semi-automatically constructed ATIS semantic grammar \[4\]; the third one is the fully annotated Penn Tree Bank ATIS grammar. All the grammar has two versions, one is partitioned, and the other is un-partitioned.

Two sets of experiments are conducted. One is to compare the speed between GLR parser and Earley parser on partitioned grammars. The other is to compare the speed between partitioned approach and un-partitioned approach for different grammar sets. Due to excessive GLR parsing table size in Penn Treebank ATIS grammar, we only use Earley parser for the effect on speed. Both experiments use Intel Pentium III 933MHz, 512 Mbytes memory and Windows 2000 Professional.

The three grammars and their partitioning methods are explained below.

The Chinese keywords list is obtained from an international news corpus, which has 15426 keywords (rules). For partitioning, all the keywords were first sorted in alphabetical order. Then, the list of keywords is split into 78 sub-grammars with 200 keywords each. The top non-terminals are named as KEYWORD\[i\], where \(i\) is the index of the sub-grammar. Here are some examples of the keyword rules.

The ATIS semantic grammar is a set of context-free rules, but contains both semantic and syntactic structures. The low level grammar rules are mainly semantic concepts of ATIS type, such as CITY-NAME, CLASS-TYPE, MONTH-NUMBER, etc. They are obtained by a semi-automatic grammar induction algorithm \[4\]. In this experiment, we partitioned the 1297 semantic rules into 64 sub-grammars. Examples of English ATIS-3 rules are given below.

The automatically partitioned syntactic grammars, are derived from the hand-bracketed Penn Tree bank, ATIS subset (577 sentences including class-A and class-D). We use sentences, which are extracted from the Penn TreeBank LDC, POS input file as our input for parser. In this experiment, we partitioned the 416 rules into 46 sub-grammars. Examples of partitioned grammar are listed below.

For the keyword grammar experiment, test sentences are hand-crafted queries. 200 Chinese queries are used as the test sentences. Average sentence length is 10.6 characters. Max length is 19 characters. In the semantic grammar case, ATIS 1993 test data is used. 1564 ATIS training set has been used as the test sentences. Average sentences length is 11.2 words. Max length is 46 words. For the Penn Treebank ATIS case, most of the ATIS sentences, i.e. 500 out of 577 POS tags sentences, are used as the test sentences. The average sentences length is 6.5 tags, and the max length is 13 tags. The full parse coverage is 99.7% in both partitioned grammar and no partitioned grammar.

The first set of the experiments is to compare their speed for the partitioned grammar case. The difference is shown in table 1 below. In this experiment, all the grammars are partitioned and tested in both pre-compile earley parser and GLR parser. Results have shown that GLR parser runs faster in most cases, especially in more complicated grammars.
Earley parser runs faster for the partitioned grammar. Results have shown that Earley parser runs faster for the partitioned case than for the un-partitioned case (Table 2).

<table>
<thead>
<tr>
<th>Grammar Type</th>
<th>No Partitioning # of Sentences per Second</th>
<th>Partitioning # of Sentences per Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword list</td>
<td>0.016</td>
<td>40.00</td>
</tr>
<tr>
<td>ATIS semantic grammar</td>
<td>0.29</td>
<td>42.30</td>
</tr>
<tr>
<td>ATIS Penn Tree Bank grammar</td>
<td>0.22</td>
<td>1.99</td>
</tr>
</tbody>
</table>

Table 2: Comparison of Earley parser’s speed for partitioned vs. un-partitioned grammars.

A third experiment was conducted to see the effect of left-corner prediction for Earley parser. The result has shown that using prediction in cascading gives 40 times of speed up, as compared with the previous cascading composition.

4. Comparison and Conclusion

This paper systematically examines the speed effect using three different grammar sets, from a simple keyword list to a complicated syntactic grammar, with two different styles of parsers. The experimental results have shown that in general, the GLR parser is faster than the Earley parser. In case of Earley parser, the partitioned case is much faster than the un-partitioned one. The results are consistent for simple grammars and complicated grammars, and for manually partitioned grammars as well as automatically partitioned grammars.

Therefore, there is no partitioning in the architecture. [12, 13] systematically examined the effect of different left corner constraints on speed. Our results have shown a more significant improvement when using grammar partitioning. We attribute the difference to the block effect, i.e., when a sub-grammar does not meet a left corner constraint, the whole sub-grammar is pruned.

5. References