Parallel Generation of GLR parsers

Thanaruk Theeramunkong　　Hozumi Tanaka
Department of Computer Science　Tokyo Institute of Technology

1 Introduction

Recently, there has been a trend to use a so-called *precompilation* approach in parsing natural language. Instead of directly using a grammar, this approach first compiles the grammar to a parsing table (a push-down transition automaton) and utilizes this table in parsing. Due to this precompilation, a parser can analyze input sentences in deterministic way and as the result the parsing speed is improved. A typical and well-known parsing method of this approach is generalized LR (GLR) parsing [5]. In the last three decades, previous works on parallel LR parsing have mainly focused on parallelizing the activity of **parsing** (e.g.,[4]) but there has been much less attention given on parallelizing the activity of **parser generation** [2]. At present time, there exist parsers that analyze input sentences in parallel systems, but there are no environments in such systems, that support the generation of a parser itself.

The GLR parsing algorithm first precompiles a grammar into a parsing table (called LR table), and guided by this table, it performs parsing efficiently. Thus the generation of LR parsers is closely involved in computing LR tables. At this point, a keen reader may deem that when a certain grammar is given, the parsing table needs to be generated only one time before using it for parsing input sentences. However, unlike a computer language, there is still no general grammar for a natural language on hand at present. The practical grammar is still under development. When there is a defect in the grammar, it is necessary to recompile the LR tables each time the grammar is revised. Furthermore, there is a trend of an enormous number of grammar rules to realize highly accurate recognition (especially, for speech recognition area[3]). Hence, the performance of table generation becomes a significant issue. In this paper, we propose a parallel table generation algorithm for reducing the computational time of GLR parser generation.

2 LR Table Generation

The computation of an LR table[1] is equivalent to generating a finite automaton (FA) which consists of a set of LR(1) items (called *closures*) as its states and transitions between each pair of the closures as its directional arcs. The table generation algorithm is composed of three processes shown as follows.

Closure Process : The closure process calculates the set of nonkernel items for each set of kernel items[2]. This process yields the following rule to add new nonkernel items. For each item $[A \rightarrow \alpha \cdot B\beta, a]$, if there is a grammar rule, $B \rightarrow \gamma$, then add an item $[B \rightarrow \cdot\gamma, b]$ when it is not previously added. Here, b is in FIRST(βa)[3]. Note that the number of the generated nonkernel items is finite because each newly added item is checked whether it previously exists or not.

Goto Process : The goto process calculates the *goto* sets of a closure. This process yields the following rule: if there is an item $[A \rightarrow \alpha \cdot B\beta, a]$ in the closure then the process generates a goto set including items $[A \rightarrow \alpha B \cdot \beta, a]$. The number of elements of the goto set is the number of items which have B as their leftmost elements before a dot. Note that all items in the goto set are kernel items.

Sifter Process : Some goto sets, the output of the goto process, may be similar to the goto sets which were previously generated. The sifter process filters out such goto sets. This process provokes a sequence of *filters* (subprocesses) corresponding to new goto sets. Each filter removes out the succeeding goto sets when they are identical to the goto set to which the filter corresponds. The sifter process also assigns each new goto set a unique state number.

The table generation starts from the computation of a closure of a single *kernel* items, $[S' \rightarrow \cdot S]$, where S is a start category. Based on this closure, the *goto* sets (each of which is a set of kernel items) are computed.

[1]The full account of LR table generation is given in [1].
[2]The combination of kernel and nonkernel items composes a *closure*.
[3]FIRST(α) is a set of all the terminals which can be the leftmost element of α in the derivation process.

These goto sets will be checked whether they were previously generated. In the next step, for each new goto set, its closure is computed. The computation of closures and goto sets is occurred in iterative way until no more closure can be generated. Here, each closure is assigned a state number. Figure 1 illustrates the computation of the closure and its goto sets (for the initial state (0-th state)).

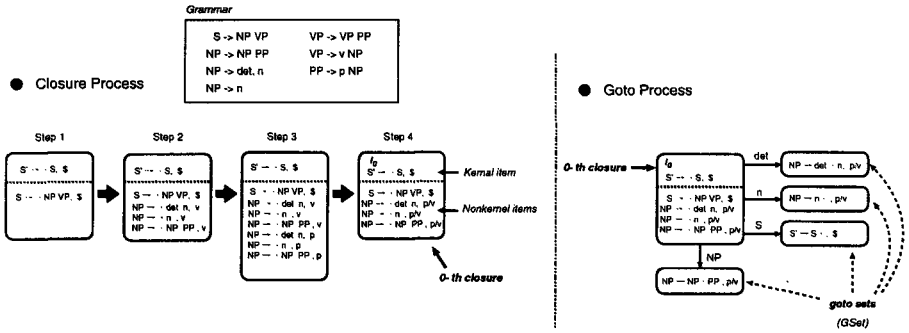

Figure 1: An example of *Closure* and *Goto* Processes

3 The Parallel Algorithm

The performance of a parallel algorithm depends on the distribution of tasks among processors. The closure and goto processes are concerned with the calculation of each individual closure, which is independent of the calculations of the others. On the other hand, the sifter process checks newly generated goto sets to remove the duplication. Then it is not the case that the sifter process can investigate all the goto sets simultaneously. It is necessary to enumerate them and examine them one by one. Taking account of these aspects, we introduce two strategies of load balancing: *on-demand dynamic* and *pipeline* load balancing methods.

3.1 On-demand Dynamic Load Balancing

Due to the independency of the tasks of the closure and goto processes, on-demand dynamic load balancing seems appropriate for parallelizing these processes. The overview of the on-demand load balancing method is shown in figure 2. In this method, a processor, called master processor, is selected from available processors to control load balancing. The master processor (MP) holds a task queue and a request queue for storing pending tasks[4] and pending requests. Each of the remaining processors, called working processor (WP), possesses two basic routines, closure and goto processes.

Initially, each WP dispatches a request for a task to the MP. By utilizing data flow synchronous mechanism, when there are some tasks in the task queue of the MP, the task at the top of the queue will match the request in the top of the request queue. This task will be transferred to the WP, which possesses that request. Here, the task signifies a goto set. For this goto set, the WP performs closure and goto routines. As the result of these routines, some new goto sets (i.e., tasks) may be generated. These goto sets are transferred to sifter process. The sifter process functions as a filter to remove the duplication of goto sets by checking with the previously generated goto sets. Only the new goto sets can pass through this process. These new goto sets are kept in the task queue (of the master processor) before they are dispatched to idle processor.

3.2 Pipeline Load Balancing

The second method, pipeline load balancing, is applied to the sifter process. The sifter process generates subprocesses (filters) corresponding to previously generated goto sets. These subprocesses function as filters for

[4]Each pending task corresponds to a goto set.

removing out the redundant goto sets. The filters are linked in a sequence. The pipeline load balancing is to distribute the chain of the filters among processors.

The most straightforward method is to distribute each filter to a processor. When the number of filters is larger than the number of processors, the filters will be assigned in round-robin fashion. This method dispatches a filter to a processor as soon as possible to gain parallelism in the early stage. However, the method has a problem in the communication cost. The task of a filter is to check the equality between an incoming goto set and its responsible goto set. If the result is false, the processor will send it to the next processor to check with other filters. In this method, there is a trend of very large communication overhead when the checking time is less than the communication time. One of the possible improvements is to group a certain number of filters to one processor(figure 2). The improved method does not dispatch a filter to a processor immediately. After the first N filters are implemented in the first processor, the following filters will be dispatched to the next processors. The motivation of this method is to improve the ratio of the checking time to the communication time. The result of grouping the filters affects the increase of checking time as well as the decrease of communication time. However, compared with the former method, this method has less parallelism. This problem arises from the tradeoff between communication time and parallelism. We will show some experiments to examine the tradeoff of parallelism and communication cost in the next section.

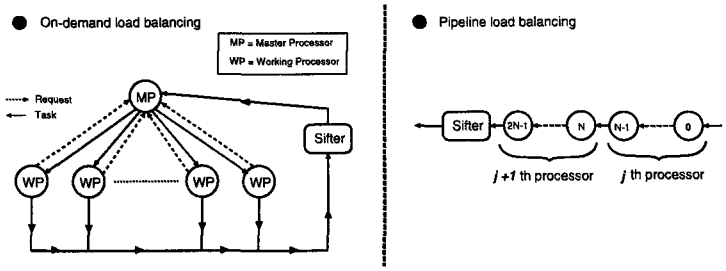

Figure 2: On-demand and Pipeline load balancing

4 Experimental Result and Evaluation

The parallel algorithm is implemented by using KL1 language on PIM/m with 256 processors available. We examine the efficiency of our parallel algorithm with a speech recognition grammar composed of 312 rules with 128 terminal, 115 nonterminal categories. The LR table generated by this grammar consists of 1710 states with 2193 shift actions, 9442 reduce actions and 422 goto actions[5]. 1128 states out of 1710 have only reduce actions (called *leave* states). In an LR automaton, a leave state is a state which has no outgoing arcs. A *balancing factor* of an automaton can be calculated as the number of non-leave states divided by the number of transition arcs. This factor implies the degree of the opportunity of parallelism. For this experimental grammar, the balancing factor is 4.49. The speedup of parallel LR table generation is shown in figure 3. Speedup is the ratio of the computational time of multiple processors to that of single processor. We investigate the speedup of the parallel algorithm using varying numbers of grouped states (N) (i.e., grouped filters in the sifter process) ranging from 1 to 10. Each datum of computational time is an average of four times of execution. From this result, we can make some observations as follows:

- There are very small speedups gained in the case of $N = 1$. As described in subsection 3.2, the filters are realized in different processors. This result shows that the checking time seems trivial and the communication cost becomes dominant. Therefore no advantage is gained from the parallelization.
- The speedup increases along with the number of processors. The maximum speedup gain is nearly 21 when the number of processors is 256 and N equals 8.

[5]The shift and goto actions are transition arcs in an LR automaton.

- The speedup increases in accord with the size of N. However, when N is more than a certain number (in this experiment, N=8), the speedup tends to decrease. As described above, although increasing the number of grouping filters (N) will lessen the communication cost, at the same time it decreases the degree of parallelism. Figure 3 illustrates this tradeoff where $N = 8$ is the turn point.

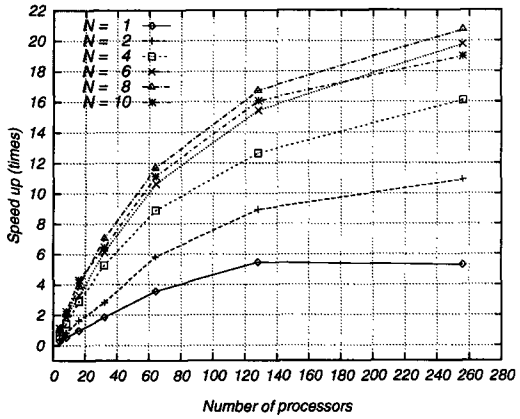

Figure 3: Speedup rate of parallel LR table generation

5 Conclusion

In this paper, we propose a parallel table generation algorithm for reducing the computational time of GLR parser generation. Our algorithm acquires parallelism by introducing two load balancing strategies for two main operations of the table generation algorithm. The parallel algorithm can succeed the speedup of 21 when the number of processors is 256. This speedup is not so large due to the *non-parallel* characteristic of the problem (table generation) itself. In the table generation, when each state is generated, it is indispensably checked with previous states. Basically, the equality check has to be carried out in one-by-one fashion. This indicates the high dependency of subtasks and then affects the total speedup. At present, there are many problems to be solved and investigated as the future work, for example, testing our idea with larger grammars and findind out a more efficient load balancing, especially for the sifter process.

References

[1] A. V. Aho and J. D Ullman. *The Theory of Parsing, Translation, and Compiling*, Vol. I & II. Prentice Hall, 1972.

[2] Manuel E. Bermudez. Parallel generation of LR parsers. In *Proc. of the 1988 International Conference on Parallel Processing*. Vol.3: Algorithms and Applications , pp. 151–155, 1988.

[3] K. Itou, S. Hayamizu, and H. Tanaka. Continuous speech recognition by context dependent phonetic hmm and an efficient algorithm for finding n-best sentence hypotheses. In *ICASSP92*, pp. 21–24, 1992.

[4] Hozumi Tanaka and Hiroaki Numazaki. Parallel generalized LR parsing based on logic programming. In Masasu Tomita, editor, *Generalized LR Parsing*, chapter 6, pp. 77–92. Kluwer Academic Publishers, 1991.

[5] Masaru Tomita. *An Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, Boston, MA, 1985.