PARALLEL TRANSLATION BASED ON GRAMMAR SYSTEMS

Stanislav ELBL, Doctoral Degree Programme (2)
Dept. of Information Systems, FIT, VUT
E-mail: elbl@fit.vutbr.cz

Supervised by: Dr. Alexander Meduna

ABSTRACT

This contribution introduces a method for parallel language translation based on grammar systems. It describes principle of parallel syntax analysis and discusses sample implementation of compiler and its results.

1 GRAMMAR SYSTEM

1.1 DEFINITION

Grammar system is a construct

$$G = (N, \Sigma, S, P_1, P_2, \ldots P_n)$$

where N is a set of nonterminal symbols, Σ is a set of terminal symbols, $S \in N$ is starting nonterminal and each P_i, $i \in \{ 1, \ldots, n \}$, is a finite set of context free productions, called i-th component of G.

Four-tuple $G_i = (N, \Sigma, S, P_i)$ is called i-th grammar of G.

1.2 DERIVATING MODES

Several derivation modes are used in grammar systems. Some of them are introduced here.

- **Terminating derivation** by the i-th component:

  $x \Rightarrow_i^t y$ if and only if $x \Rightarrow^* y$ in $G_i = (N, \Sigma, S, P_i)$ and $y \not\Rightarrow_i z$ for all $z \in (N \cup \Sigma)^*$

  where $y \not\Rightarrow_i z$ means, that y does not derive z in the grammar G_i.

- **K-step derivation, at least k-step derivation and at most k-step derivation**:

  $x \Rightarrow_i^{=k} y$ if and only if $x \Rightarrow^k y$ in G_i.

$x \Rightarrow_i^{\leq k} y$ if and only if $x \Rightarrow^j y$ in $G_i, j \leq k$.

$x \Rightarrow_i^{\geq k} y$ if and only if $x \Rightarrow^j y$ in $G_i, j \geq k$.

Using these derivating modes, the generative power of grammar system increases in comparison with context free grammars. However this contribution does not introduce this.

1.3 UTILIZATION OF GRAMMAR SYSTEM FOR PARALLEL TRANSLATION

The idea of translation parallelization is based on the fact, that one grammar of a grammar system can "work" on several places in the derivate sentence form. Following grammar system serves as an example:

$G = (\{S, A, ...\}, \Sigma, S, P_1, P_2)$

$P_1 = \{ S \rightarrow AS, S \rightarrow A \}$

$P_2 = \{ A \rightarrow ... \}$

A grammar G_1 of this system generates sentence forms A^+ - a strings of nonterminal A. Then G_2 continues derivation. Separate fragments of sentence form generated by G_2 are independent and their derivation can be processed simultaneously.

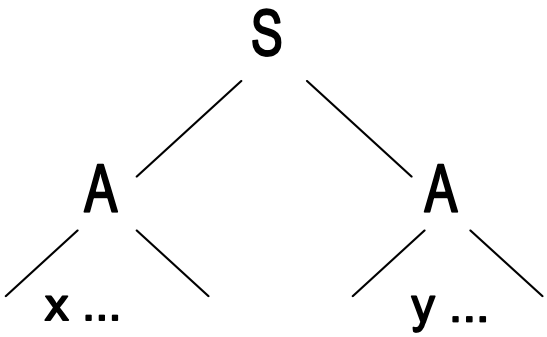

Fig. 1: *Sample derivation tree*

There is a sample derivation tree on the picture above. It corresponds to derivation $S \Rightarrow^* AA \Rightarrow^* x...y...$. Generation of substrings beginning with x and y can proceed together.

1.4 PARALLEL SYNTAX ANALYSIS

The work must be distributed to several threads to achieve parallel syntax analysis. To do this, there must be found parts in source code, which can be translated together, independently. On the picture above the substrings beginning with x and y represent these parts. In the case of any programming language, procedures, functions or code blocks can represent them.

Fast syntax analysis can be provided to find the beginnings and ends of the blocks. Its method is given by source language. For example it can look for appropriate key words (e.g. *procedure*, *function*, *endproc*, *codeend*), find end of block corresponding to its start (e.g. by

counting symbols "*{*" and "*}*"). A syntax tree is not built during this search.

Whatever method of syntax analysis can be used - recursive descent (it is used in implemented compiler), LR(1) parser based on a table and so on. It is necessary, that analysis of several inputs can be proceeded in one time. This is reason, why some automatic parser generators (as Yacc, Bison or Lex) can't be used - they often use global variables, which would cause malfunction of the compiler.

2 IMPLEMENTATION

A sample compiler was implemented to confirm this method properties and applicability. There is its implementation shortly described in this section and the results are summarized in next one.

2.1 INPUT LANGUAGE

Subset of language Pascal was chosen for testing purposes of this method. It's syntax analysis is not difficult, because it is LL(1) language. There is recursive descent method used in the implementation of compiler. Translation of basic integer and floating-point arithmetic, local variables access, function and procedure calls with parameters passing, *for* statement, *if* statement, assignment, short evaluation of logical expressions and the call of built in procedures *exit*, *write* and *writeln* are implemented.

2.2 TARGET CODE

Java class file is used as the target code of compiler. This code can be interpreted by any Java virtual machine. Class file is used for binary representation of class - this means, that any class must be created from the source program. Static methods of this class are created from procedures and functions and global variables are represented by its static attributes.

2.3 TRANSLATION

Before the translation is started, lexical analyzers and syntax analyzers are created - every thread uses one of each. The first syntax analyzer finds the beginning of the first procedure or function. All global variables, which are found during this searching, are inserted into the global symbol table. As soon as the start of function is found, analyzer finds out its end also - this is the place, where any other syntax analyzer can start reading of source code. No other thread can search for procedure start - it can either translate any previously found one, or it must wait until this thread finishes searching.

Location of the end of function consists in providing fast syntax analysis of the input. All lexical symbols expect of *begin*, *case* and *end* are omitted. These symbols are counted and the place, where the number of *end* symbols is equal to the count of *begin* and *case* symbols is marked as the end of actual function. (This is given by the syntax structure of Pascal - if the input language was another one, the rule would be different). Actual thread can begin translation of the found function and another thread can start new search.

The translation of procedure is now independent on the other threads. Analyzer returns to the procedure start and provides complete syntax analysis and creates appropriate syntax tree. When the whole syntax tree is created, local optimizations and target code generation are processed.

2.4 OPTIMIZATION

Optimization is a fundamental part of the translation. However they are not implemented in testing compiler. They are just replaced by elemental simulation - empty program loop is inserted instead of them and its duration can be changed from the command line of the compiler. Program was tested with several values of this duration.

Here are the examples of any local optimization, which can be provided: dead code elimination (detection of code, which is never executed in the program and his removing), detection of common sub-expressions, live variables detection, transfer of condition evaluation to the loop end, evaluation of constant sub-expressions etc.

3 RESULTS

This sample compiler has been tested on various source files with several settings of local optimization duration on 4 processors computer Sun Enterprise 450 (its further specification is available on the `http://www.fit.vutbr.cz/CVT/e450.html.en`).

Following tables contain times of the translation of some simple programs (it has been executed a few times in a loop to get measurable values). Optimization setting denotes a number of cycles of empty loop (just incrementing variable). Sequential translation uses distinct algorithm - it does not require searching of procedure start and end. It is not tasked with any additional computation.

	Optimization setting			
	0	10000	100000	1000000
Sequential translation	8,3	10,0	26,0	180,0
2 threads	8,3	9,1	16,5	94,0
3 threads	8,2	8,6	12,6	67,0
4 threads	8,2	8,5	10,9	52,0

Tab. 1: *Translation of sample file 1 - 6767 bytes, 26 short functions*

	Optimization setting			
	0	10000	100000	1000000
Sequential translation	4,5	4,9	9,1	51,0
2 threads	4,8	4,8	7,0	31,0
3 threads	4,5	4,7	5,8	23,0
4 threads	4,5	4,8	5,5	17,3

Tab. 2: *Translation of sample file 2 -1299 bytes, 7 short functions*

It is obvious from the table, that acceleration of parallel translation widely depends on optimization setting. This is because local optimizations of several procedures are not dependent and they can be performed together. On the other hand, input file reading must be strictly synchronized to assure correct translation - every thread must compile different part of the source program. Inserting symbols to the global symbol table must be synchronized also.

Appropriate column of the tables above must be selected to evaluate results - the column, which mostly corresponds to real compilers. The setting 100000 of optimization

extends sequential translation 2 or 3 times. This value can be optimal. The following picture summarizes attained acceleration for this setting.

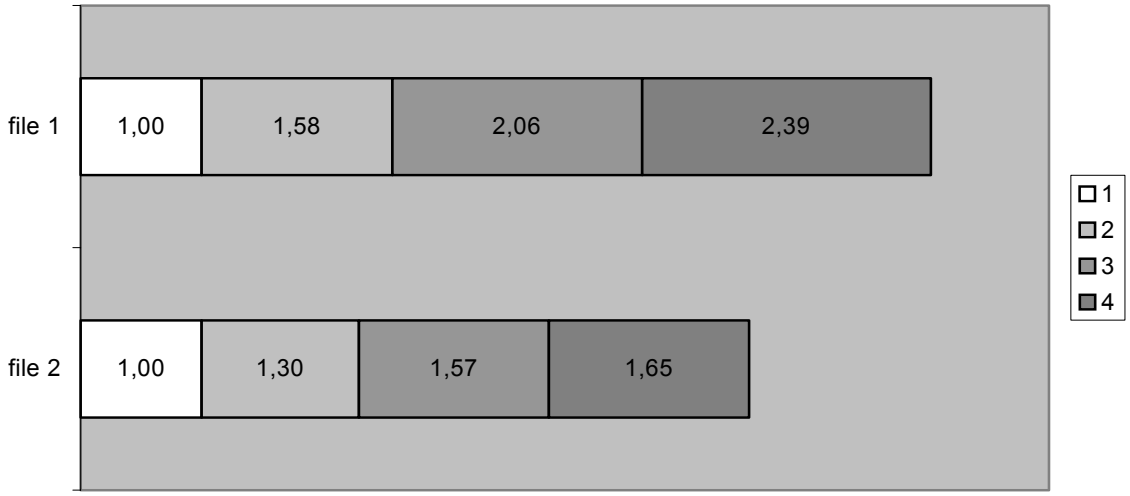

Fig. 2: *Translation acceleration*

There is an approach to improve these results. You can see, that lexical analysis is provided twice. First it is provided in combination with fast syntax analysis when finding independent parts. Second it is provided while translating code. It is obvious, that sequence of tokens can be stored in memory first time and then the second analysis is not required. In addition the lexical analysis can be provided in parallel. It is subject of present research.

REFERENCES

[1] Beneš, M., Hruška, T., Kolář, D.: Compilers, [texts for lectures], VUT Brno

[2] Beneš, M., Generování cílového programu pro JVM, [a library documentaion]

[3] Lindholm T., Yellin F., The Java™ Virtual Machine Specification, Second Edition, Document available on URL `http://java.sun.com/docs/books/vmspec/` (january 2003)

[4] Meduna, A.: Moderní teoretická informatika, [texts for lectures]

[5] OpenMP C and C++ Application Program Interface, Version 2.0, March 2002