View metadata, citation and similar papers at core.ac.uk brought to you by CORE
provided by Archivio istituzionale della ricerca - Politecnico di Milano

The PAPAGENO parallel-parser generator

Abstract. The increasing use of multicore processors has deeply transformed computing paradigms and applications. The wide availability of multicore systems had an impact also in the field of compiler technology, although the research on deterministic parsing did not prove to be effective in exploiting the architectural advantages, the main impediment being the inherent sequential nature of traditional LL and LR algorithms. We present PAPAGENO, an automated parser generator relying on operator precedence grammars. We complemented the PAPAGENO-generated parallel parsers with parallel lexing techniques, obtaining near-linear speedups on multicore machines, and the same speed as Bison parsers on sequential execution.

Keywords: Parser generation, Parallel Parsing, Operator Precedence Grammars

1 Introduction

Parsing, or syntactic analysis, plays a fundamental role in a wide variety of computing applications, from compilation to browsing of structured and semi-structured data, from natural language processing to genomics, etc. In the last years all these fields have experienced increasingly demanding requirements in terms of time and energy consumption or size of the data sets to be processed, which urged for new effective parsing solutions. Some attempts have been made to devise new parsing algorithms, or obtain relevant speedups from the classical deterministic ones, by leveraging on the computing capability offered by modern multiprocessor architectures, but they had almost no success except for a few overly specific cases (as e.g. for ad-hoc parsers for XML and HTML).

The classical parsing algorithms used for deterministic context-free (DCF) languages as LR and LL, in fact, can be efficiently implemented (in linear-time) on serial machines, but they do not speedup on multicore architectures because of their inherent left-to-right sequential nature: if an input string is split into several parts, handled by different processors, the parsing actions may require communication among the different processing nodes, with considerable additional overhead. Although this work is no place for a comprehensive survey, we point out the works of Mickunas and Schell [1] and the more recent ones of [2] as an example of such issues.

Recently we focused on a subclass of DCF the *Operator precedence languages* (OPLs), and their grammars (*Operator precedence grammars*, OPGs) which have been defined by Robert Floyd a few decades ago [3] and represent a precursor of LR languages. OPLs have some limits in terms of expressive power and they had been soon overtaken by parsing techniques based on the more expressive LR family: still, OPGs are adequate for many common programming languages [4]. The remarkable – and until now unnoticed – aspect of OPLs, is that differently from the larger class of DCF languages they enjoy a property of *local parsability*, which makes them suitable for efficient parallel parsing. Local parsability means that parsing of any substring of a string according to an OPG depends only on information that can be obtained from a local analysis of the portion of the substring under processing and is, thus, not influenced by parsing of other substrings [5, 6].

In this work we present a generator of deterministic parallel parsers (PAPAGENO) for syntactic grammars specified as OPGs, which exploits their local parsability property. To our knowledge, PAPAGENO is the first general-purpose practical generator of efficient deterministic parallel parsers. It features significant speedups in parsing of both general programming languages and standard data representation languages. In this work we improve the tool features presented in [5, 6] through the effective coupling of the parallel parsing with a parallel lexical analysis. Moreover, we show that it is possible, exploiting a moderately tailored parallel lexical analysis, to describe the Lua programming language with OPGs.

2 Parallel parser generation with PAPAGENO

We first recall the essentials of OPGs and of the corresponding bottom-up parsers (more details in [4, 5, 7]).

A grammar rule is in *operator form* if its right hand sides (r.h.s.) have no adjacent nonterminals; an *operator grammar* (OG) contains only such rules. Without loss of generality, we can also assume that the rules of the grammar have no repeated r.h.s. and renaming rules are absent.

OPGs exploit three binary partial relations on the set of terminal symbols, named *precedence relations*, which can be automatically derived from the rule set of the grammar: between any two terminals the *equal in precedence* ($\doteq$), *yields precedence* ($\lessdot$), *takes precedence* ($\gtrdot$) relations may hold. An OPG is defined as an OG where between any pair of terminal symbols there is at most one precedence relation. Precedence relations are inspired by the notion of precedence between the operators of arithmetic expressions: in the same way as e.g. the precedence of product over sum controls the parsing and evaluation of an arithmetic expression, similarly the relations between the terminal symbols guide deterministically the parsing of a string.

Precedence relations, in particular, determine the local parsability property of OPGs: in any partially reduced string, any segment delimited by a pair $\lessdot$ and $\gtrdot$, where $\doteq$ holds between consecutive terminal characters within it (possibly separated by a nonterminal), corresponds to the r.h.s. of a grammar rule. Parsing of the sentence can arbitrarily start from any position in the string: when the parsing algorithm identifies a segment with this pattern by examining the precedence relations, it can reduce it to the corresponding l.h.s. (which is unique if the grammar has no repeated r.h.s.) and the reduction by means of the chosen rule will never be affected or invalidated by the processing of other portions of the whole string.

A very efficient parallel parsing algorithm can be devised. An input string can be split in different parts, each parsed in parallel by independent processors. The choice of the positions where splitting the string is totally arbitrary, differently from other proposed parallel parsing algorithms which require that each substring starts at the beginning of suitable (language-dependent) syntactic units (e.g. loops, blocks, etc.). The partial parsing trees generated by the different processors can then be pairwise combined with constant-time transformations and reduced into the final tree, possibly with a further or – rarely – multiple parallel passes, depending on the structure of the trees.

3 Tool structure, performances and applications

PAPAGENO offers a practical tool to automatically generate parallel parsers starting from the description of a grammar in a GNU Bison-like syntax. It has been conceived

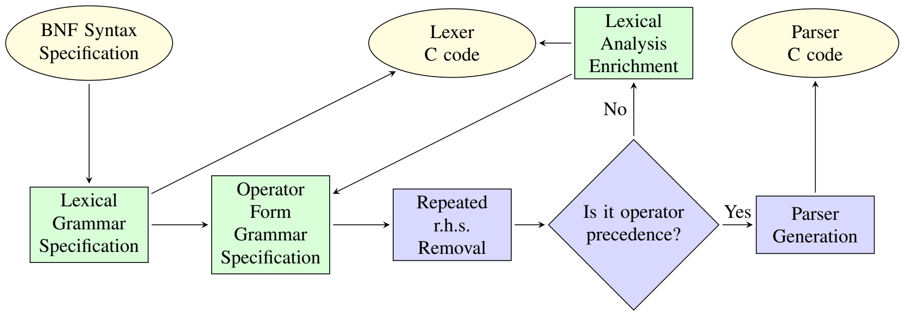
BNF Syntax Specification
Lexer C code
Lexical Analysis Enrichment
Parser C code
No
Lexical Grammar Specification
Operator Form Grammar Specification
Repeated r.h.s. Removal
Is it operator precedence?
Yes
Parser Generation

Fig. 1. Typical development flow of a parser, employing PAPAGENO. The human operator stages are marked in green, while the PAPAGENO automated staged are marked in blue.

to be a drop-in replacement to Bison-generated parsers, allowing to exploit the benefits of automatically generated parallel parsers with a minimum codebase re-engineering effort. The generated parser can thus be combined with a scanner generated by GNU Flex in the same way a Bison generated parser does, and does not rely on any external libraries, except the common C library.

The parallel workers are implemented exploiting POSIX threads, and have been successfully benchmarked with Linux and MacOS X implementations. To prevent thread interlocking due to the memory allocation performed via the `libc` allocation functions, the generated parser adopts a pooled allocation strategy to handle both the parsing stack involved in the process and the construction of the AST. As it is frequent to check whether the current symbol under analysis is a terminal or a nonterminal, its belonging to one of the two sets is bit-packed within the same integer value representing the symbol, thus yielding a fast checking strategy by means of bit-masks. To ease portability, the position of the packed bit is designer-tunable, while the tool provides a suitable default value for x86(64) and ARM architectures.

In order to optimize r.h.s. matching at reduction time, the r.h.s. of the grammar rules are stored in a prefix trie, so that the recognition of the correct reduction is performed in linear time with respect to the longest r.h.s. of the grammar and is fully independent from the grammar size. To prevent a performance loss from the scarce spatial locality of a trie, the data structure is effectively linearized into a constant vector at parser generation time, thus yielding efficient memory accesses upon look-up.

We have been able to successfully generate a full JSON parallel parser, together with a straightforward lexer, proving the practicality of parallel parsing through OPGs of data description languages. Contrary to common belief, we note that the parallelization of the lexing phase becomes relevant when dealing with operator precedence parsing, as the running times of the parser and the lexer are comparable for lightweight syntax languages such as JSON.

We have also been able to tackle the parsing of the Lua programming language, assuming some sensible, and much widespread, programming practices are employed when writing Lua sources. Parsing Lua through OPGs has been possible thanks to a proper lexing stage which allows a more natural expression of the grammar in operator precedence form through token renaming, in a fashion similar to the one proposed by Floyd for an ALGOL-like language in [8], and by De Bosschere for Prolog in [9]. We note that this enriched lexer can still be parallelized effectively: we achieved near linear improvements in our current tests.

The overall parser design workflow with PAPAGENO is summarized in Figure 1. The figure shows the novel and enriched role of lexical analysis w.r.t. to classical compilers: the lexical analysis in fact, besides being carried over in parallel, has also the goal of producing an intermediate code that is better suited for an operator precedence parsing.

4 State of the project

The current state of PAPAGENO provides a working tool to generate parallel parsers starting from the grammar description. The violations to the constraint on the absence of repeated right hand side rules in the grammar is pointed out to the parser designer and an automated r.h.s. elimination algorithm is run to assist developers. Currently, we provide the JSON sequential lexer and parallel parser with the codebase as a working example to ease the understanding of the toolchain. Interested users should thus be able to express their preferred language in an OPG compliant syntax with a limited effort. The number of parallel parsing threads can be chosen at parsing run-time, simply providing it as an input parameter to the parsing function, allowing efficient adaptation to the target platform capabilities. Moreover, we perform fully parallel lexing of JSON and Lua, obtaining further speedups. The generated parsers were tested on x86_64, ARM 926, and ARM Cortex-A architectures retaining the same performance across all the platforms. We are planning to enlarge the set of languages supported by OPGs and the corresponding lexical specifications. Further improvements involve a more methodical approach to the parallelization of the lexing stage, and the integration with incremental parsing methods such as [10], which are particularly well suited to our operator precedence parallel parsing algorithm, is also considered. The codebase of PAPAGENO is available at: https://github.com/PAPAGENO-devels/papageno

References

1. Mickunas, M.D., Schell, R.M.: Parallel compilation in a multiprocessor environment. In: Proceedings of the 1978 annual conference, New York, USA, ACM (1978) 241–246
2. You, C.H., Wang, S.D.: A data parallel approach to XML parsing and query. In Thulasiraman, P., Yang, L.T., Pan, Q., Liu, X., Chen, Y.C., Huang, Y.P., huang Chang, L., Hung, C.L., Lee, C.R., Shi, J.Y., Zhang, Y., eds.: HPCC, IEEE (2011) 520–527
3. Floyd, R.W.: Syntactic Analysis and Operator Precedence. J. ACM **10**(3) (1963) 316–333
4. Grune, D., Jacobs, C.J.: Parsing techniques: a practical guide. Springer, New York (2008)
5. Barenghi, A., Crespi Reghizzi, S., Mandrioli, D., Pradella, M.: Parallel parsing of operator precedence grammars. Inf. Process. Lett. **113**(7) (2013) 245–249
6. Barenghi, A., Viviani, E., Crespi Reghizzi, S., Mandrioli, D., Pradella, M.: PAPAGENO: a parallel parser generator for operator precedence grammars. In: 5th International Conference on Software Language Engineering (SLE). (2012)
7. Crespi Reghizzi, S., Mandrioli, D.: Operator Precedence and the Visibly Pushdown Property. Journal of Computer and System Science **78**(6) (2012) 1837–1867
8. Floyd, R.W.: Syntactic analysis and operator precedence. J. ACM **10**(3) (1963) 316–333
9. De Bosschere, K.: An Operator Precedence Parser for Standard Prolog Text. Softw., Pract. Exper. **26**(7) (1996) 763–779
10. Ghezzi, C., Mandrioli, D.: Incremental parsing. ACM Trans. Program. Lang. Syst. **1**(1) (1979) 58–70

The PAPAGENO parallel parser generator - Demo Outline

Abstract. The tool demonstration of the PAPAGENO parallel parser generator will provide a practical walkthrough of the tool use. To this end, we will implement a full parser for arithmetic expressions during the demo. We will showcase also the improvements in computation time provided by the effective parallelization of the lexing phase on JSON and Lua sources, with respect to a serial approach.

Keywords: Parser generation, Parallel Parsing, Operator Precedence Grammars

1 Introduction

In this demo we will present a typical use of the PAPAGENO tool, aimed at implementing a simple arithmetic expression grammar. To this end, we will describe how the PAPAGENO tool is embedded in the common parser development toolchain, which is sketched in Figure 1.

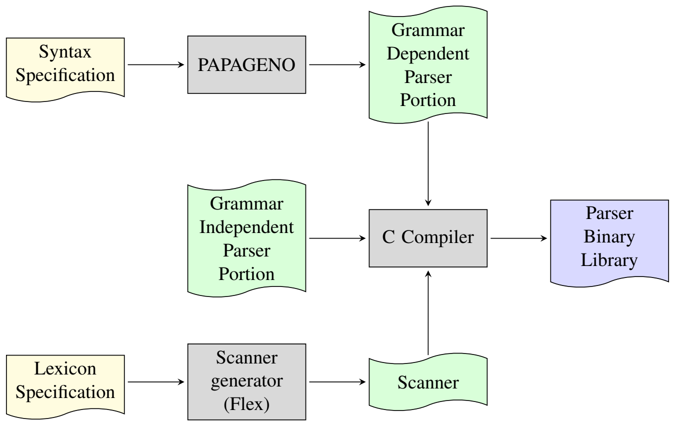

Fig. 1. Typical usage of the PAPAGENO toolchain, starting from a grammar and lexical specification, obtaining the parsing library. Specifications provided by the user are marked in yellow, generated C source code is marked in green.

PAPAGENO is designed to be a Bison replacement, minimizing the developer effort to transition to a parallel parsing approach. Consequentially, it fills the parser generator spot in the common development flow, providing a direct interface to Flex generated lexical analyzers. PAPAGENO generates C sources for the `pthread`-based library while keeping an understandable format of the generated code, allowing the developer to fine tune it, should the need arise. The tool provides a simple interface which allows the developer to select the directories where the parser implementation and headerfiles are generated, allowing a quick integration into existing projects. In this demo we will follow the common development workflow for a sample grammar, showing how it is possible to perform parallel parsing effectively and efficiently. We will take as our running example grammar the one of the common arithmetic expressions, having sum and product as arithmetic operators, parentheses, and integer numbers as operands. Briefly, the productions of the grammar are $S \to E, E \to E + T \mid T, T \to T \times F \mid F, F \to (E) \mid \text{integer}$.

2 Implementing a compliant lexer

In this section we show the key points to implement a lexer for the simple language of arithmetic expressions.

The developer can choose to implement a sequential or a parallel scanner to perform lexical analysis of the input file. In the first case, the developer has to provide a specification of the lexicon with the syntax of a Flex program. The specification file is divided into three sections (definition section, rules section, developer subroutines) separated by lines consisting of two percent signs `%%`. The first section has to include the following definition of the structure chosen to represent a token, where the `token` field denotes the token identifier and the `semantic_value` field denotes its value:

```
%{
#include "grammar_tokens.h"
struct lex_token {
        gr_token token;
        void* semantic_value;
};
extern struct lex_token* flex_token;
%}
```

Fig. 2. Data structure used to represent a token in the specification file for Flex.

The only difference with the common Flex specification format is that the semantic values of the tokens are stored as memory zones referenced by a pointer, instead of resorting to the common contract via the definition of a union.

The rule section contains the regular expression based definition for the tokens of the lexical grammar.
A sequential C scanner is then directly generated by Flex from the specification file. The lexing action simply builds the token list by calling the *yylex* routine on the input file until the end of the file is reached or an error occurs.

The developer willing to implement a parallel scanner has to provide an interface to the lexing process that initially splits the input file into different segments and then runs on each of them a parallel worker to perform the scanning. The splitting of the file into parts must not split any lexeme across distinct segments. For the arithmetic expressions grammar, where possible lexemes are numbers, arithmetic operators and parentheses, the splitting points of the input file can be easily chosen. It is sufficient to split anywhere outside a number, by first selecting equal-sized segments to divide the file and then shifting their cutting points until a non-digit character is met. For more complex lexical grammars, it may be not possible to analyze a bounded context around the initial cutting points to guarantee that the edges of the segments do not cut any lexeme in the middle. However the developer can handle efficiently this problem by allowing the cutting points unaligned with the tokens, while properly encoding the valid lexing actions for the fragment in a compact fashion.

After the splitting the file, each segment is processed by an independent worker: the parallel workers are generated as reentrant lexers through the proper Flex option. Care should be taken though, to buffer the allocation of the semantic values of the tokens to prevent interlocking on *malloc* calls coming from different threads. After each worker has completed the scanning of its file segment, the complete token list is built by concatenating the partial ones of the workers.

Except for our simple grammar example, parallelization of the lexical analysis has to deal with various issues. For instance, the lexical grammars may include arbitrary length tokens, delimited by a special character (e.g. double quotes to delimit a string). This lexical feature implies that splitting the input character stream may incur in the need to sweep a large part of the input before a proper splitting point is determined. To this end, a viable alternative is to ignore this issue during the input splitting and effectively generate two tokenized streams, according to the assumption of being or not in the middle of an arbitrary length token. Once all the lexer threads have finished, it is possible to disambiguate which tokenized streams are the correct ones, thus joining them together with a minimal cost. Practical experiments with grammars including arbitrary length tokens, such as the ones of JSON and Lua have shown that this process is effectively lightweight and leaves the actual call to the Flex generated lexer as the lexing hotspot.

3 Implementing a sample parser

In this phase, the tool demonstration will tackle the implementation of a parser for the arithmetic expression grammar. Implementing a parser employing PAPAGENO for its generation requires the developer to describe the syntactic grammar of the language with a syntax similar to the one employed by Bison. The grammar description file is split into three sections, employing a double `%` marker on a single line as separation marker between them. The first section of the grammar description is composed by a C code preamble where helper functions can be declared by the developer. The helper functions will be placed in the same compilation unit as the generated parser core. The second section of the grammar description contains the list of terminal and nonterminal symbols as reported in Figure 3. The `%axiom` keyword denotes the axiom nonterminal of the grammar. The third section of the grammar description is the one containing grammar productions, described in a Bison-like syntax: the l.h.s. and r.h.s. of the rules are separated by a colon, and the alternatives for the r.h.s. are separated by a pipe. Each rule is endowed with a semantic action, specified as a C code block where the

```
%axiom S
%nonterminal S
%nonterminal E
%nonterminal T
%nonterminal F
%terminal PLUS
%terminal TIMES
%terminal LPAR
%terminal RPAR
%terminal NUMBER
%%
S : E { };
E : E PLUS T { } | T { } ;
T : T TIMES F { }  | F { };
F : LPAR E RPAR { }  | NUMBER { };
```

Fig. 3. Description of the grammar in expressed in the PAPAGENO input format. The semantic actions associated to the rules are left blank for the sake of clarity.

developer may insert any C code at his will. We note that currently only a single tail semantic action per grammar rule is supported by PAPAGENO.

The developer can access the semantic values attached to the terminals by the Flex-generated lexer employing the same familiar *dollar-sign* convention employed in Bison syntax. Nonterminal symbols are bound to a semantic value with the same structure of terminal ones, except for the fact that the `semantic_value` pointer is NULL. In practice, the semantic values of the elements of the r.h.s. of the grammar are accessible as `$n`, where n is the position of the symbol counting from 1. For instance, in the grammar reported in Figure 3, the pointer to the semantic value of the `F` symbol in `T : T TIMES F { }` is accessible as `$3`. The semantic action is executed at rule reduction time, so the developer may safely assume that all the nonterminals belonging to the rule have been properly reduced.

The interface of the generated parser is a straightforward function call: `token_node *parse(int32_t threads, char *file_name)`, which returns a pointer to the root of the abstract syntax tree built during the parsing action, taking as parameters the input pathname of the file to be parsed and the number of threads to be spawned. The parser generated by PAPAGENO splits the input text in as many portions as the specified number of threads, and, after a first parsing pass, employs a single worker to perform the recombination of the partially parsed substrings. In case the expected structure for the AST of the input texts is a deeply nested one, it is possible for the developer to specify that the recombination action should be performed by more than one thread. This mode of operation can be selected defining the `LOG_RECOMBINATION` macro when compiling the parser code: this causes the number of threads to be halvened at each parsing pass after the first one, allowing a further exploitation of the parallelism obtained through OPG based parsers. In the demonstration, we will show both modes of operation for the generated parallel parsers.

4 Sample semantic actions and running times

After completing the implementation of the parallel parser for arithmetic expressions, we will show the speed improvements practically trying to parse an arithmetic expression encompassing the sum of the first 1000 numbers. The target platform employed for the following examples is a dual core Intel(R) Core 2 Duo L9400 laptop, absolute timings may change on a different demonstration platform. However, the reported speedups we obtain across both x86_64 and ARM based platform is consistent.

The result of such a parse action with a single thread is the following:

```
The result is:500500
Pthread 0> Correct parse
Successful parse
Parse action finished:
Lexer: 0.001687 s, Parser 0.010775 s
```

By contrast, employing two worker threads on a dual core laptop, computation produces the following result:

```
Pthread 1> Parsing in progress
Pthread 0> Parsing in progress
The result is:500500
Pthread 2> Correct parse
Successful parse
Parse action finished:
Lexer: 0.001323 s, Parser 0.004634 s
```

which is showing an effective halvening in the time required for the parsing action.

We will also show that the generated parser handles the runtime change of the number of worker threads, trying it with a large number of them (e.g. 200), producing an output similar to the following:

```
Pthread 194> Parsing in progress
the result is:500500
Pthread 200> Correct Parse
Successful parse
Parse action finished:
Lexer: 0.001346 s, Parser 0.052662 s
```

showing a natural increase in the parsing time with respect to the previous case, due to the creation of a largely superfluous amount of extra worker threads. Despite the very large amount of them, however, the performance penalty is limited, showing a good scalability of the approach.

After showing the performance results with the simple arithmetic expressions grammar we will show the practical results on JSON inputs, taking as an example a 150 KiB file, producing the following output in the serial parsing case:

```
Pthread 0> result Correct parse
Successful parse
Parse action finished:
Lexer: 0.023729 s, Parser 0.060376 s
```

while significantly reducing the parsing time with two threads as shown by the following output:

```
Pthread 0> result Parsing in progress
Pthread 1> result Parsing in progress
Pthread 2> result Correct parse
Successful parse
Parse action finished:
Lexer: 0.024487 s, Parser 0.037732 s
```

The final results which will be demonstrated during this tool demo are the ones achieved on Lua sources by means of both parallel parsing and parallel lexing. For instance, a serial parsing pass of a 60 KiB Lua file yields:

```
Pthread 0> result 0
Parse action finished:Successful parse
Lexer: 0.012889 s, Parser 0.016206 s
```

while a parallel one reports significant improvements in both the lexing and parsing phase:

```
Pthread 0> result 1
Pthread 1> result 1
Pthread 2> result 0
Parse action finished:Successful parse
Lexer: 0.006202 s, Parser 0.013914 s
```