The PAPAGENO Parallel-Parser Generator

Alessandro Barenghi[1], Stefano Crespi Reghizzi[1,2], Dino Mandrioli[1], Federica Panella[1], and Matteo Pradella[1,2]

[1] Dipartimento di Elettronica, Informazione e Bioingegneria - Politecnico di Milano
[2] National Research Council - Institute of Electronics, Computer and Telecommunication Engineering (CNR-IEIIT)
{alessandro.barenghi,stefano.crespireghizzi,
dino.mandrioli,federica.panella,matteo.pradella}@polimi.it

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, ranging from compilation to browsing of structured and semi-structured data, natural language processing and genomics. 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 classic deterministic ones, by exploiting 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, such as LR and LL, can be efficiently implemented (in linear-time) on sequential machines, however they do not achieve speedups on multicore architectures due to their inherent 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.

A. Cohen (Ed.): CC 2014, LNCS 8409, pp. 192–196, 2014.
© Springer-Verlag Berlin Heidelberg 2014

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. The parsing of the sentence can start from an arbitrary position in the string: when the parsing algorithm identifies a segment with the aforementioned pattern through examining the precedence relations,

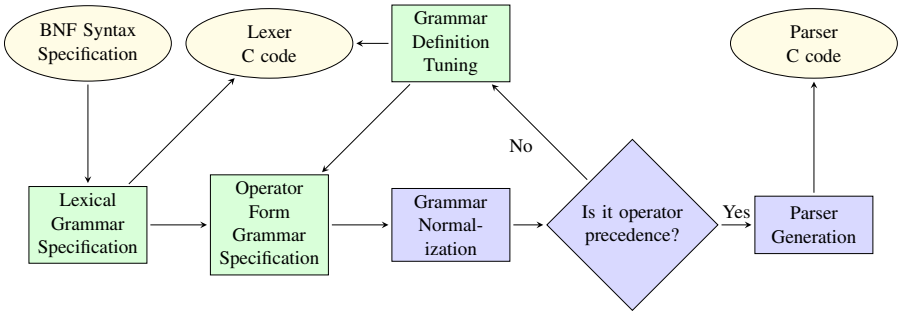

Fig. 1. Typical development flow of a parser, employing PAPAGENO. The human operator stages are marked in green, while the PAPAGENO automated staged are marked in blue.

it reduces 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 from this parsing strategy: the input string is split in different parts, each one parsed in parallel by independent processors. The choice of the positions where the string is split is fully arbitrary, differently from other proposed parallel parsing algorithms, f.i. [8], which require each substring to start 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 – seldom – multiple parallel passes, depending on the structure of the syntax 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 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. For instance, parsing a 10 MB JSON file with 8 workers, we obtain a $3.18\times$ speedup ($3.6\times$ against Bison) employing a parallel lexer coupled to the PAPAGENO generated parser, while the speedups drop to $2.08\times$ ($2.29\times$ against Bison) when employing a sequential lexer.

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 [9], and by De Bosschere for Prolog in [10]. 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 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 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 [11], which are particularly well suited to our operator precedence parallel parsing algorithm, is also considered. In addition, we are considering the possibility of tackling other data description languages: among them restricted XML documents may offer a viable topic for further research. In particular, we note that current parallel XML parsers, such as [2] employ a language specific approach to tackle the problem, often resorting to linear-time sequential preprocessing passes. 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, pp. 241–246. ACM, New York (1978)
2. You, C.H., Wang, S.D.: A data parallel approach to XML parsing and query. In: HPCC, pp. 520–527. IEEE (2011)
3. Floyd, R.W.: Syntactic Analysis and Operator Precedence. J. ACM 10(3), 316–333 (1963)
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), 245–249 (2013)
6. Barenghi, A., Viviani, E., Crespi Reghizzi, S., Mandrioli, D., Pradella, M.: PAPAGENO: A parallel parser generator for operator precedence grammars. In: Czarnecki, K., Hedin, G. (eds.) SLE 2012. LNCS, vol. 7745, pp. 264–274. Springer, Heidelberg (2013)
7. Crespi Reghizzi, S., Mandrioli, D.: Operator Precedence and the Visibly Pushdown Property. Journal of Computer and System Science 78(6), 1837–1867 (2012)
8. Sarkar, D., Deo, N.: Estimating the speedup in parallel parsing. IEEE Trans. on Softw. Eng. 16(7), 677 (1990)
9. Floyd, R.W.: Syntactic analysis and operator precedence. J. ACM 10(3), 316–333 (1963)
10. De Bosschere, K.: An Operator Precedence Parser for Standard Prolog Text. Softw., Pract. Exper. 26(7), 763–779 (1996)
11. Ghezzi, C., Mandrioli, D.: Incremental parsing. ACM Trans. Program. Lang. Syst. 1(1), 58–70 (1979)