PAPAGENO: a parallel parser generator for operator precedence grammars

Alessandro Barenghi, Ermes Viviani, Stefano Crespi Reghizzi
Dino Mandrioli, and Matteo Pradella

Dipartimento di Elettronica e Informazione - Politecnico di Milano
{barenghi, viviani, crespi, mandrioli, pradella}@elet.polimi.it

Abstract. In almost all language processing applications, languages are parsed employing classical algorithms (such as the LR(1) parsers generated by Bison), which are sequential due to their left-to-right state-dependent nature. Although early theoretical studies on parallel parsing algorithms delineated potential speedups on abstract parallel machines using a data-parallel approach, practical developments have not materialized, except in recent experiments on ad hoc parsers for large XML files. We describe a general-purpose practical generator (PAPAGENO) able to produce efficient deterministic parallel parsers, which exhibit significant speedups when parsing large texts on modern multi-core machines, while not penalizing sequential operation. The generated parser relies on the properties of Floyd's operator precedence grammars, to provide a naturally parallel implementation of the parsing process. Parsing of each text portion proceeds in parallel and independently, without communication and synchronization, until all partial parse stacks are recombined into the final result. Since Floyd's grammars can express most syntaxes with little adaptation, we have performed extensive experiments, on both synthetically generated texts and real JSON documents. The effective parallel code portion in the generated parsers exceeds 80% for most of the tested scenarios.

Keywords: Parser generation, Parallel Parsing, Floyd Operator Precedence Grammars

1 Introduction

Language parsing, also known as syntactic analysis, occurs in many situations: compilation, natural language processing, document browsing, genome sequencing, program analysis targeted at detecting malicious behaviours, and others. Although syntactic analysis is less computationally demanding than semantic analysis, it is frequently applied to very large data sets in contexts where speedups and related energy saving are often important. The common linear-time left-to-right LR(1) and LL(1) algorithms used for deterministic context-free (or BNF) languages are an important milestone of algorithmic research. Due to their ability to recognise a wider class of formal languages, they superseded earlier algorithms such as the ones employing Floyd's *operator-precedence grammars* (FG), which rely on only local information to decide parsing steps (for an introduction see e.g. [1]). The LR and LL algorithms are amenable to efficient implementations on serial computing machines, such as the ones provided in popular parser generators (e.g. Bison), but their structure hinders efficient parallelization: early attempts at it have not been successful, and appear to be almost abandoned.

In this work we describe a general-purpose optimized FG-based parallel-parser generator. As mentioned, FGs have already been used successfully to define early programming languages; they have a very fast sequential parser, which is still used by modern compilation platforms (`gcc` for instance) to parse expressions with multiple operator precedences. Recently, research on formal methods has renewed the interest for FGs, thanks to their nice closure and decidability properties [2]. In particular, the relevant feature distinguishing FG languages from LR and LL ones is the closure under the substring extraction; this property enables independent parallel parsing of substrings. Starting from a straightforward sequential parser, we have implemented a parser generator producing two versions of an associative reduction scheme, and measured consistent speedups on both synthetic benchmarks and large real JSON files. Measurements indicate good scalability on different multi-core architectures, leading to significant reductions of parsing time with respect to state-of-the-art sequential parsers.

Related research The straightforward idea of splitting a long text into chunks, to be parsed in parallel and subsequently recombined by further parsing actions, has motivated several studies in the period 1965-1990. Early theoretical studies on data-parallel algorithms such as the ones reported in [3, 4] determined the computational complexity that can be obtained in principle on certain abstract parallel machines by using a data-parallel approach.
Parallel parsing requires an algorithm that, unlike a classical parser, is able to process *substrings* which are not syntactically legal, although they occur in legal strings. A substring parser operates on a substring without any knowledge of the outcome of parsing left and right contexts. Incidentally, substring parsing algorithms have been also studied for different purposes, such as processing damaged or faulty texts. Notable examples of substring parsing research have adapted shift-reduce LR(1) parsers, by dropping the condition that the text to the left of the chunk under examination should have been parsed already. The first, and simpler approach due to Mickunas and Schell [5], modifies an LR(1) parser in order to start in several possible states and to scan the chunk as far as deterministically possible. However, a significant amount of the computation is typically left to the chunk recombination phase as this may propagate changes on all the chunks preceding the one being recombined. In other approaches, such as [6] and [7], each chunk parser carries on all possible alternative parses and subsequently, it recombines the parsed chunks: assuming the original grammar to be LR(1), this chunk parsing can be done in linear time although with constants greater than the ones for shift-reduce parsing.
The few people who have performed some limited experimentation on such algorithms have generally found that performances critically depend on the cut points between chunks: if a chunk starts, say, with `begin`, the parser can recognize almost completely a full language block. On the contrary, starting a chunk on an identifier opens too many syntactic alternatives. As a consequence such parsers have been typically combined with language-dependent heuristics for splitting the source text into chunks that start on keywords announcing a splitting friendly construct.
After a long intermission, research in the field of parallel parsing has recently resumed with more practical goals, such as to parse XML documents on multicore machines, both servers and clients. Some published works, e.g., [8] and [9], rely explicitly on the assumption that the parsed language is either XML or a subset of it, in order to devise

ad-hoc strategies to extract parallelism from the parsing process. In other words, such projects no longer qualify as general-purpose parsers. Although specialized parsers, say, for XML, may be of interest to particular communities, our project is the first to develop a general-purpose language-independent parallel parser generator, and validate it with experiments on real-world benchmarks on modern architectures. In addition to this, we provide guidelines for the implementation and optimization of the algorithm, such as the design of a unified data structure for the abstract syntax tree and parser stack representation, which has proved instrumental for obtaining the high code fraction executed in parallel, reported in the experimental results.

The paper is organized as follows: Section 2 provides the background and the definitions regarding FGs, Section 3 proposes our parallel parsing algorithm, Section 4 delineates the implementation strategies employed and reports the results of the experimental validation campaign. Finally, Section 5 draws our conclusions.

2 Definitions and background on operator precedence grammars

Since *Floyd*'s operator precedence *Grammars* (FG) and parsers are a classical technique for syntax definition and analysis, it suffices to recall the main relevant concepts from e.g. [1]. Let Σ denote the *terminal alphabet* of the language. A BNF grammar in *operator form* consists of a set of productions P of the form $A \to \alpha$ where A is a *nonterminal* symbol and α, called the right-hand side (rhs) of the production, is a nonempty string made of terminal and nonterminal symbols, such that if nonterminals occur in α, they are separated by at least one terminal symbol. The set of nonterminals is denoted by V_N. It is well known that any BNF grammar can be recast into operator form. To qualify as FG, an operator grammar has to satisfy a condition, known as absence of precedence conflicts. We will now introduce informally the concept of *precedence relation*, a partial binary relation over the terminal alphabet, which can take one of three values: $\lessdot$ *(yields precedence)* , $\gtrdot$ *(takes precedence)* , $\doteq$ *(equal in precedence)*. For a given FG, the precedence relations are easily computed and represented in the *operator precedence matrix* (OPM). A grammar for simple arithmetic expressions and the corresponding OPM are in Fig. 1. Entries like $+ \lessdot a$ and $a \gtrdot +$ indicate that, when

Grammar G consists of $\Sigma = \{a, +, \times, (,)\}, V_N = \{E, T, F\},$ axiom $= E$ and

$$P = \{E \to E + T \mid T, \qquad T \to T \times F \mid F, \qquad F \to (E) \mid a\}$$

Operator precedence matrix:

$$M_2 = \begin{array}{c|c|c|c|c|c|}
 & a & + & \times & (&) \\ \hline
a & & \gtrdot & \gtrdot & & \\ \hline
+ & \lessdot & \gtrdot & \lessdot & \lessdot & \gtrdot \\ \hline
\times & \lessdot & \gtrdot & \gtrdot & \lessdot & \gtrdot \\ \hline
(& \lessdot & \lessdot & \lessdot & \lessdot & \doteq \\ \hline
) & & \gtrdot & \gtrdot & & \gtrdot \\ \hline
\end{array}$$

Fig. 1. Example of FG for arithmetic expressions.

parsing a string containing the pattern $\ldots + a + \ldots$, the rhs a of rule $F \to a$ has to

be reduced to the nonterminal F. Similarly the pattern $\ldots + (E) \times \ldots$ is reduced by rule $F \rightarrow (E)$ to $\ldots + F \times \ldots$ since the relations are $\ldots + \lessdot (\overset{\doteq}{E}) \gtrdot \times \ldots$. There is no relation between terminals a and (because they never occur as adjacent or separated by a nonterminal. A grammar is FG if for any two terminals, at most one precedence relation holds. In sequential parsers it is customary to enclose the input string between two special characters $\perp$, such that $\perp$ yields precedence to any other character and any character takes precedence over $\perp$.
Precedence relations precisely determine if a substring matching a rhs should be reduced to a nonterminal. This test is very efficient, based on local properties of the text, and does not need long distance information (unlike the tests performed by LR(1) parsers). In case the grammar includes two productions such as $A \rightarrow x$ and $B \rightarrow x$ with the same rhs, the reduction of string x leaves the choice between A and B open. The uncertainty could be propagated until just one choice remains open, but, to avoid this minor complication, we assume without loss of generality, that the grammar does not have repeated rhs's [2].
The mentioned local properties suggest that FG are an attractive choice for data-parallel parsing, but even for sequential parsing, they are very efficient [1]: "Operator-precedence parsers are very easy to construct and very efficient to use, operator-precedence is the method of choice for all parsing problems that are simple enough to allow it". In practice, even when the language reference grammar is not a FG, small changes permit to obtain an equivalent FG, except for languages of utmost syntactic complexity. This is witnessed by the JSON grammar employed as our benchmark, described in Section 4.

3 PAPAGENO

3.1 Parallel Parsing Algorithm

As the parallel algorithm implemented by our generator stems directly from sequential FG parser, we first describe the latter, providing the extension to the parallel technique afterwards. As far as we know, this is the first design and realization of a parallel parsing algorithm for FGs. We also note that this algorithm performs an effective parse of the token stream and is thus different from parallel bracket matching algorithms such as the one presented by Cole [10]. The key idea driving Algorithm 1 is that, wherever a series of $\doteq$ precedence relations enclosed by a pair of $\lessdot, \gtrdot$ is found between adjacent tokens, the enclosed symbol string is the handle of a reduction. To find handles, the parser uses the operator precedence matrix OPM and a (pushdown) stack S to keep track of the tokens to be reduced when the next $\gtrdot$ relation is found. As the parsing algorithm needs to recognise a particular grammar rule (e.g. for building the Abstract Syntax Tree AST representation) upon reduction, the list of productions P is needed to detect the actual production to be applied. Provided that the algorithm is adapted to always shift nonterminal symbols, it is possible to reuse it for all the stages of parallel parsing with no modifications.

In the case of a serial parsing the algorithm takes an empty stack S as a parameter, and the list of input tokens as I, which can be thought as delimited by two special symbols marking the beginning and the end of the token stream. Algorithm 1 operates as follows: it obtains the symbol under the cursor and checks its precedence relation with the terminal symbol occupying the highest place on the parsing stack (lines 3–4).

Algorithm 1: Floyd Grammar Parser

```
Globals: $OPM$: Precedence matrix of $G$, $P$: List of productions of $G$
Input: $I$: Input symbol list, $S$: Parsing stack
Output: $S$: the parsing stack after the parsing action
begin
    while $I \neq \emptyset$ do
        token ← READ_CURSOR($I$)
        prec ← $OPM$(token,TOP_TERMINAL($S$))
        if prec $\neq \gtrdot$ or IS_NONTERMINAL(token) then
            PUSH($S$ , (token, prec))
            MOVE_CURSOR_FORWARD($I$)
        else
            repeat
                (token , prec) ← POP($S$)
                PUSH(rhs_rebuild , token)
            until prec= $\lessdot$
            if $\exists\, p \in P \mid$ RHS($p$) = rhs_rebuild then
                SEMANTIC_ACTION(LHS($p$))
                PUSH($S$ , (LHS($p$),$\perp$))
            else
                return NIL
    return $S$
```

If the precedence relation is not $\gtrdot$ or the symbol is not a terminal, the symbol is pushed on the top of the stack and the cursor is moved forward by one position (lines 5–7). If the parsing algorithm meets a $\gtrdot$ precedence relation, it needs to rebuild the rhs of the corresponding rule to perform the proper reduction action: this is performed through an auxiliary stack, rhs_rebuild, where the algorithm stores the elements from the top of the stack, until it finds a $\lessdot$ precedence relation. Upon finding the $\lessdot$ precedence relation, the algorithm checks if the rebuilt rhs is a valid production of the grammar and performs the reduction action if this is the case. If the rebuilt rhs is not a valid production the algorithm terminates abnormally signalling that the input string is not valid through returning NIL. If the serial parsing procedure terminates correctly, Algorithm 1 is expected to return a parsing stack containing only the axiom of G.

Exploiting the fact that the parser makes the decision whether to shift or to reduce only on the basis of the precedence matrix OPM, the current token, and the top of the stack, it is possible to divide the token stream into different chunks or substrings, and perform a substantial amount of the parsing with different workers. The essential quality of this algorithm is that all the parsing actions performed on a chunk are final, i.e. no parsing work is ever undone on a substring, thus all the parsing actions performed by the worker threads are correct and useful. To this end, the input is split into as many chunks as the desired number of workers w, and all the workers run the aforementioned algorithm returning their stacks at the end.

Since the splitting of the token stream is not constrained in any way, nor it depends on the language grammar, the result returned by a worker will likely be a nonempty

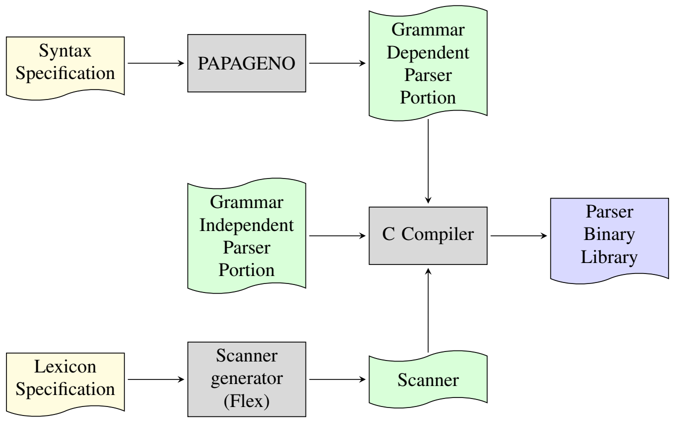


Fig. 2. Typical usage of the PAPAGENO toolchain, starting from a grammar and lexical specification, obtaining the parsing library. Specifications provided by the user are marked in yellow, generated C source code is marked in green.

stack S, since there are no warranties that an arbitrary substring of a sentence is a valid sentence of the language. Such nonempty stack can be partitioned in two parts: one containing only $\doteq$ and $\gtrdot$ relations (i.e. the lower one), and one containing $\lessdot$ and $\doteq$ (i.e. the upper one).

We thus combine the results of two adjacent parsers employing as the parsing stack of the new parser the upper part of the stack of the left worker, and as input stream the lower part of the right one. As the parsing algorithm is able to shift the nonterminal symbols on the stack, it will be able to handle the input stream even if it is not composed by tokens only.

3.2 The PAPAGENO Parser Generator

PAPAGENO, the parallel parser generator tool, produces a C implementation of the parallel parsing algorithm described before, from the specification of a grammar in operator precedence (Floyd) form. The implementation of the parser is combined with a lexical scanner obtained from the de-facto standard scanner generator Flex, to obtain a fully functional parser library, which can be linked with the main application being developed. We chose to employ serial lexers generated by Flex to enhance the ease of use of the tool, however designing a parallel lexer is a rather easy task, as lexical analysis is inherently local and provides a mean to split the input data properly. Moreover the performance loss is negligible as the lexing process is very fast. The parsing process is started by invoking the `parse` call, which receives two parameters: a reference to the input character stream, and the number of workers onto which the parsing process should be split. As depicted in Figure 2, the typical workflow to employ PAPAGENO is analogous to the one of common parser generators such as Yacc/Bison. The user writes two files: a grammar specification, describing the grammar rules and any semantic actions to be performed jointly with reductions, and a lexical specification, describing the terminals or tokens used by the grammar. PAPAGENO can thus be employed as a drop-in replacement for common parser generators, provided that the user checks the form of the language grammar, and removes the possible precedence conflicts in the rules.

For the sake of clarity, and to ease the study of the implementation and possible modifications, the C parser implementation is split into a grammar-independent part (support for data structures, parsing algorithm implementation) and a grammar dependent part (token and productions representation), residing in different compilation units. The choice of the C language as the target for the implementation was driven by the need to produce highly performing parser implementations while retaining the largest portability. To ease the adoption of PAPAGENO, the syntax of the grammar specification file follows closely the one used by Yacc/Bison, thus requiring little or no effort to port an existing grammar definition. In particular, semantic attributes of the terminals and nonterminals may be conveniently accessed through the same syntax as Bison, and the semantic actions to be triggered upon a reduction are specified in the same way. To guide the user during the process of representing the grammar in Floyd form, PAPAGENO offers diagnostic messages pinpointing any existing precedence conflicts between terminal symbols, and it outputs a printable form of the precedence matrix.

3.3 Performance tuning strategies

It would be impossible to achieve the potential advantages of parallel parsing without a careful choice of efficient programming techniques, of which we report here the most significant ones. We have found that memory access represents the bottleneck of our parsing technique, due to the computational lightweight nature of FG parsing, therefore the parsers generated by PAPAGENO exploit various techniques to relieve as much as possible the memory pressure on the target architecture. First, the terminal and nonterminal symbols are represented as integers, taking care to use the most significant bit as a flag to separate terminals or non-. In this way, it is possible to decide whether a list node should be shifted on the stack or not, through a simple check on the first bit of the value, avoiding the use of a lookup table.

In addition to this, since the precedence value can only assume four different values (namely, $\lessdot, \doteq, \gtrdot$ and $\perp$), we use a bit-packed representation of the OPM effectively reducing its size by four times against a straightforward character based representation, which allows to achieve low latency access to the table thanks to the fact that it is small enough to fit in the processor cache memories. To prevent performance losses from fragmented memory allocation, typical of pointer based structures such as the AST, we manage a preallocated userspace memory pool, wrapping the common memory allocation function (`malloc`). This technique both increases the data locality and prevents the workers, implemented as POSIX threads, from being serialized during the calls to the `malloc` function. As far as the size of the memory pool goes, PAPAGENO preallocates half of the estimated size of the AST, through computing the average branching factor from the length of the right hand sides of the grammar rules, and increases the allocated pool size by one fifth of this quantity, if the parser needs more memory.

Another performance tuning technique concerns a smart representation for the rhs of the rules, to ease the checks upon reduction. The rhs are stored in a prefix tree (*trie*), thus allowing the parsing process to find the matching production in linear time w.r.t. the length of the rhs of the productions. In order to further compress the trie, we use the

S $\rightarrow$ OBJECT
OBJECT $\rightarrow \{\} \mid \{$ MEMBERS $\}$
MEMBERS $\rightarrow$ PAIR $\mid$ PAIR , MEMBERS
PAIR $\rightarrow$ STRING **:** VALUE
VALUE $\rightarrow$ STRING $\mid$ $\boldsymbol{number}$ $\mid$ OBJECT $\mid$ ARRAY $\mid$ $\boldsymbol{bool}$
STRING $\rightarrow ''''$ $\mid$ $''$CHARS$''$
ARRAY $\rightarrow$ [ ] $\mid$ [ELEMENTS]
ELEMENTS $\rightarrow$ VALUE $\mid$ VALUE , ELEMENTS
CHARS $\rightarrow$ CHAR $\mid$ CHAR CHARS $\quad\|\quad$ $\underline{\text{CHARS} \rightarrow \boldsymbol{char} \mid \boldsymbol{char}\ \text{CHARS}}$
CHAR $\rightarrow$ $\boldsymbol{char}$

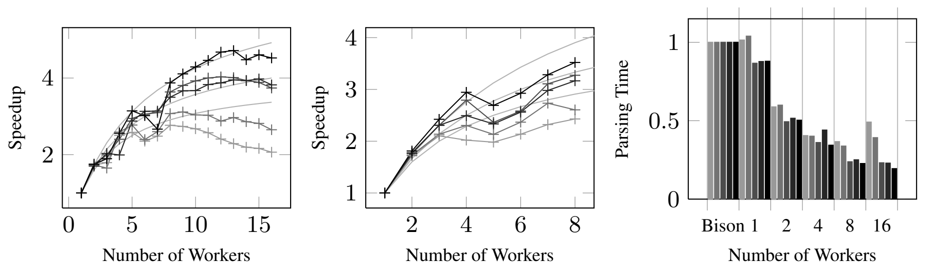
Fig. 3. Official JSON grammar. Uppercase symbols denote nonterminals, while lowercase ones are tokens; the only one modified rule needed to transform the grammar in operator precedence form is underlined.

technique described by Germann et al. in [11], which represents the trie as an array, both improving the data locality and reducing the size of the data structure, while retaining the same cost in the lookup operations. This technique involves the representation of the pointers of the trie structure as indexes stored in the same array as the trie values.

4 Benchmark application: the JSON language

We chose the *JavaScript Object Notation* (JSON) language as a case study to evaluate the performances of the generated parser. JSON is a data representation language described in the Internet Engineering Task Force document RFC4627, and widely employed in web applications as a less verbose substitute for XML. We also picked JSON as the average size of a JSON file is larger than a compilation unit of a common programming language. The official grammar of JSON, listed in Fig. 3 required only a trivial change to be put into Floyd compliant form, thus confirming the expressive power of this family of languages. The generated parser for the JSON grammar was tested on two different x86-64 Linux hosts to evaluate the achieved speedups: the first host is an Intel Core i7 920, a high end desktop CPU endowed with Simultaneous Multi-Threading (SMT) capabilities, while the second one is a quad-Opteron 8378, (16 physical cores in total, 4 cores per socket), a typical server grade platform. All the hosts were running a Linux 2.6 series kernel and were equipped with enough RAM to contain the whole AST and token list.

As a testbench, we chose real world JSON files of different sizes, in order to evaluate the speedup obtainable. The set of chosen files encompasses the configuration file of AdBlocker, a common browser plugin (80 kB), the Gospel of John (150kB), a statistic data-bank on food consumption provided by the Italian Institute of Statistics (1.6MB), a file containing statistics on n-grams present in English in Google Books (10MB), and the index of all the documents available on the UK Comprehensive Knowledge Archive Network (75MB). Figure 4(a) and 4(b) report the speedup factors over a serial parsing process achieved on the aforementioned testbench by the 16 core and 4-core-SMT platforms respectively while raising the number of workers. The tone of grey in the plot indicates the length of the string, with the lighter greys representing shorter strings. Moreover, the theoretical speedups predicted by Amdahl's law for a parallel code portion of 75%-80%-85% are plotted in background as a reference gauge. Already for small string lengths (80kB) the algorithm yields a speedup higher than $2\times$, but the full



(a) Speedups w.r.t input string length - 16 Core Platform (b) Speedups w.r.t input string length - 4 Core Platform (c) Comparison with Bison - 16 Core Platform

Fig. 4. Speedups obtained with respect to the string length (Figures (a) and (b))and comparison with Bison generated LALR(1) parsers (Figure (c)). Running times have been normalized taking as time unit the execution time of the Bison generated parser.

advantages of parallel parsing become evident from strings as small as 150kB (where the parallel execution portion reaches 75%) and are fully exploited starting from the 1.6 MB dataset. These results show that our approach is already effective and profitable for text sizes in the range of the average webpage (Google reports in its web metrics report in 2010 an average page size of 320 kB [12]). We also report that, without employing the aforementioned performance tuning techniques to optimize memory accesses and reduce inter-worker serializations (packed structures and memory pooling), the portion of code effectively executed in parallel by the architecture was significantly lower.

Overall, the parallel parsing strategy shows a promising exploitation of the multiple cores available on modern platforms. In particular, this strategy effectively exploits the advantages of simultaneous multi-threading enabled architectures, as raising the number of workers beyond the one of the physical cores of the architecture (i.e. above four in our case) still yields significant speedups. A quantitative measure of the parallel code portion shows that 95% of the instructions are performed in parallel on the four cores thanks to the interleaving of the instructions from two workers on a single core performed by the architecture. This tight instruction interleaving exploits the stalls caused by the memory load and store actions to perform computations from another worker effectively obtaining a parallel code portion close to the theoretical maximum. Finally, Figure 4(c) provides a comparison of the overall parsing times against the ones of a serial LR parser generated by Bison. The depicted data show how our approach behaves consistently better than Bison when employing at least two workers, while showing comparable running times even when employed in serial mode. In particular, the parsing time for the longest string (75MB in size) is effectively cut down from 10.51s to 2.07s, a roughly five-fold improvement.

As there is no recent open literature report on the performances of a general purpose parallel parser generator, we report the work of Lu et al. [8], who built an XML specific parser. The authors report that, exploiting selected features of the language identified via a preparsing phase, it is possible to obtain speedups ranging from 2.35× to 2.55× on a quad core machine, depending on the target XML structure. To this approach we compare favourably as we achieve higher speedups without the use of a preparsing pass or any specific knowledge about the points where the input token stream is split.

5 Conclusion

We have presented a new parallel parser generator which exploits the properties of operator precedence grammars, also known as Floyd grammars. Such grammars are expressive enough to be used a real world programming language (for instance Algol68 has a fully specified FG). The tool generates automatically a C implementation of the parser given a grammar description provided in Bison compatible syntax, and an additional parameter indicating how many threads are desired. The experimental validation of the effectiveness of the generated parsers, employing the grammar of JSON as a practical test case shows that the code portion running in parallel reaches 85% on common multicore architectures and scales well up to 16 cores.

As a future direction to enhance our tool, we foresee the development of a parallel lexer generator. This will allow a complete parallelization of the lexing-scanning process, thus allowing an easier distribution of the workload among different execution units. Also, the same distinguishing FG property of closure under substring extraction will allow us to couple parallel parsing with incremental techniques.

Acknowledgements We thank Brad Chen of Google Inc. for practical indications on the benchmarks, and in particular for suggesting to use JSON. Valerio Ponte has participated in the early development of the tool.

References

1. Grune, D., Jacobs, C.J.H.: Parsing techniques a practical guide. Ellis Horwood Limited, Chichester, England (1990)
2. Crespi Reghizzi, S., Mandrioli, D.: Operator precedence and the visibly push-down property. JCSS, Journ. Computer and System Science **78**(6) (November 2012) 1837–1867
3. Cohen, J., Kolodner, S.: Estimating the speedup in parallel parsing. IEEE Transactions on Software Engineering **11**(1) (January 1985) 114–124
4. Sarkar, D., Deo, N.: Estimating the speedup in parallel parsing. IEEE Trans. on Softw. Eng. **16**(7) (July 1990) 677
5. Mickunas, M.D., Schell, R.M.: Parallel compilation in a multiprocessor environment (extended abstract). In: Proceedings of the 1978 annual conference. ACM '78, New York, NY, USA, ACM (1978) 241–246
6. Goeman, H.: On parsing and condensing substrings of LR languages in linear time. Theor. Comput. Sci. **267** (September 2001) 61–82
7. Bates, J., Lavie, A.: Recognizing substrings of LR(k) languages in linear time. ACM Trans. Program. Lang. Syst. **16** (May 1994) 1051–1077
8. Lu, W., Chiu, K., Pan, Y.: A parallel approach to XML parsing. In: GRID, IEEE (2006) 223–230
9. Pan, Y., Zhang, Y., Chiu, K.: Hybrid Parallelism for XML SAX Parsing. In: Web Services, 2008. ICWS '08. IEEE International Conference on, IEEE Computer Society (sept. 2008) 505 –512
10. Cole, M.: Parallel programming, list homomorphisms and the maximum segment sum problem. In: Proceedings of ParCo. Volume 93. (1993) 211–230
11. Germann, U., Joanis, E., Larkin, S.: Tightly packed tries: How to fit large models into memory, and make them load fast, too. In: Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing. (2009) 31–39
12. Ramachandran, S.: Web metrics: Size and number of resources. Technical report, Google (2010)