Science of Computer Programming 112 (2015) 195–226

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Parallel parsing made practical

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

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Politecnico di Milano, Piazza Leonardo Da Vinci 32, 20133 Milano, Italy

ARTICLE INFO

Article history:
Received 16 October 2014
Received in revised form 28 August 2015
Accepted 12 September 2015
Available online 30 September 2015

Keywords:
Parallel parsing algorithms
Syntax analysis
Parallel parser
Operator precedence grammar

ABSTRACT

The property of *local parsability* allows to parse inputs through inspecting only a bounded-length string around the current token. This in turn enables the construction of a scalable, data-parallel parsing algorithm, which is presented in this work. Such an algorithm is easily amenable to be automatically generated via a parser generator tool, which was realized, and is also presented in the following. Furthermore, to complete the framework of a parallel input analysis, a parallel scanner can also combined with the parser. To prove the practicality of a parallel lexing and parsing approach, we report the results of the adaptation of JSON and Lua to a form fit for parallel parsing (i.e. an operator-precedence grammar) through simple grammar changes and scanning transformations. The approach is validated with performance figures from both high performance and embedded multicore platforms, obtained analyzing real-world inputs as a test-bench. The results show that our approach matches or dominates the performances of production-grade LR parsers in sequential execution, and achieves significant speedups and good scaling on multi-core machines. The work is concluded by a broad and critical survey of the past work on parallel parsing and future directions on the integration with semantic analysis and incremental parsing.

1. Introduction

The current evolution of computing platforms led to the increase in the number of computing cores being the only way to increase their performances, beyond the power and frequency wall. As a consequence, the only way to cope with the ever increasing amount of data to be processed is to fully exploit the exposed parallelism whenever possible.

Parsing algorithms are used in all sorts of applications, with web browsing, text processing, compilation, and natural language processing being some of the most common application domains. However, in addition to being ubiquitous, they are also a significant exception to the above trend towards the exploitation of parallel architectures: the current state of the art reports no practically fruitful effort in this direction, save for an ad-hoc work tackling the parsing of HTML5 [51]. Open literature offers several historical proposals of parallel parsing algorithms, which however had no follow-up, let alone application, despite the growing amount of data which need to be processed, which has grown to such a pressing need that

* Corresponding author at: Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Tel.: +39 02 2399 3476.
E-mail addresses: alessandro.barenghi@polimi.it (A. Barenghi), stefano.crespireghizzi@polimi.it (S. Crespi Reghizzi), dino.mandrioli@polimi.it (D. Mandrioli), federica.panella@polimi.it (F. Panella), matteo.pradella@polimi.it (M. Pradella).
[1] Also with CNR IEIIT.

http://dx.doi.org/10.1016/j.scico.2015.09.002
0167-6423/

hardware accelerators have been developed to tackle it [38]. We refer the reader to Section 7 for a fairly comprehensive and critical analysis of the literature on this topic.

The most likely reason for this lack of practical parallel parsers is the intrinsically sequential nature of the classical deterministic (LR and LL) algorithms. For instance, assume to parse the language $L = L_1^*$, where L_1 is $\{1a^nb^n \mid n \geq 1\} \cup \{0a^nb^{2n} \mid n \geq 1\}$ through a deterministic shift-reduce parallel algorithm. Intuitively, it would be natural to map the parsing of each substring candidate to belonging to L_1 into a separate computation, henceforth called *worker*, and then to collect the partial results to decide whether the global sentence belongs to L or not. However, since the substrings belonging to L_1 can be arbitrarily long, any random or fixed policy to split the input into substrings is likely to be far from optimal; it may even be impossible to determine whether *ab* or *abb* groups are to be reduced in case a substring contains no 0s or 1s.

To cope with these issues, two straightforward approaches were pursued in the literature: speculative computations [41] or pre-scanning [39]. The former approach non-deterministically (speculatively) performs the parsing computation for all the possible cases: in our example, it carries on two parsing processes depending on whether the current substring is of type $1a^nb^n$ or $0a^nb^{2n}$, and discards the results of the incorrect computation as soon as possible. This approach, despite being effective, is not quite efficient: the computational effort of the parsing can be doubled or more, depending on the degree of nondeterminism. The latter approach involves a first lightweight scanning of the input to determine the viable splitting points: in the aforementioned example it would look for occurrences of 0 and 1 and split the input right before them. However, this approach introduces an overhead for the preliminary scanning which could require a pre-scanning over the whole input string. Summing up, the first approach may require an computational overhead to cope with the nondeterminism which is potentially more significant than the benefits provided by the parallelism, while the latter implies an $\mathcal{O}(n)$ worst-case preprocessing which, in case of simple languages, may take as much as a sequential parsing process.

The key point to overcome these impasses was our renewed interest in the "old-fashioned" operator precedence grammars (OPGs) invented by R. Floyd in his pioneering work [21] which laid the foundations of deterministic bottom-up parsing. After their first application in compiler construction, such grammars have been abandoned due to the advent of more powerful grammars, in terms of generative power, namely LR grammars. LR grammars [34,35] enable efficient, sequential parsing and generate all deterministic languages whereas OPGs do not. Nevertheless, we maintain the generative power of OPGs is quite adequate to formalize most languages of practical interest (Sections 4 and 6 discuss this issue) and we note that, thanks to their simplicity – and the simplicity and efficiency of their parsing algorithms – they are still used in compiler construction [28], the most notable example being the GNU Compiler Collection `gcc`, which actually employs an OP parser to handle expression parsing in C.

In this paper, we exploit the *local parsing* properties of OPGs to construct a *non-speculative* parallel parser. Intuitively, a language is locally parsable if, by inspecting a substring of bounded length, an (e.g., bottom-up) algorithm can deterministically decide whether the substring contains the right-hand-side of a production and can unequivocally replace it with the corresponding left-hand side. Local parsability is the key property that enables data-parallel parsing of isolated parts of the input so that their partial results can be recombined in a global syntax tree without backtracking: in other words, all the isolated partial syntax trees of a valid text are *final*.

In the following, we will present a systematic approach to exploit the local parsability of OPGs, providing a publicly available parser generator tool, and the results of an experimental campaign highlighting the speed-ups achievable w.r.t. popular sequential parsers, such as those generated by GNU Bison [2]. We chose as practical test-benches for our approach the JavaScript Object Notation (JSON) data description language, which offers a real-world validation for large input files, and the Lua programming language, to gauge how a language far richer than a data description language performs with the parallel lexing and parsing. We show that the minor theoretical limitations in terms of generative power of OPGs do not significantly affect the applicability of the approach: the changes needed to adapt the original BNF of the source language to OPG constraints are obtained in an original way by augmenting, and parallelizing as well, the initial phase of lexical analysis (or scanning).

The paper is organized as follows: Section 2 provides the theoretical foundations of OPGs, and their local parsability property, together with a parallel parsing algorithm. Section 3 provides our methodology for parallel lexical analysis, while Section 4 describes how we adapted the JSON and Lua languages to OP-based parsing. Section 5 describes the architecture of our parser generator, and Section 6 presents the results of the benchmark campaign. Section 7 compares our approach to previous research on parallel parsers, while Section 8 presents our conclusions and possible further research directions.[2]

2. Parallel parsing theory for OP grammars

In this section we develop a theory supporting the parallel parsing of OPLs. After summarizing their basic definitions and properties, we prove that OP grammars and languages enjoy the local parsability property, which is the key to make parsing parallel. Subsequently, we revise the sequential parsing algorithm for OPGs so that it applies both to terminal substrings and to partially processed ones (sentential forms). In this way, we devise a parallel procedure consisting of two (or more)

[2] An early description of the theory of parallel parsing for OPGs has appeared in [7] but, to make the paper self-contained, it is resumed and refined here in Section 2. On the practical side, PAPAGENO is described in the tool paper [6].

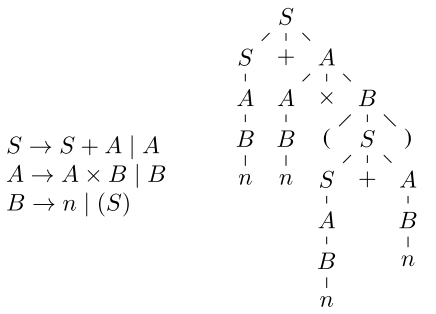


Fig. 1. A grammar generating arithmetic expressions with parentheses.

passes: at first the source string is split into chunks and each one is assigned to a separate worker, then the outputs are recombined and either a sequential or more parallel parsing passes are applied to them, computing the final parse tree.

2.1. Basic definitions and preliminary properties

Given a string x, its length is denoted as $|x|$ and the i-th character in x, for $1 \leq i \leq |x|$, is denoted as $x[i]$. The symbol ε denotes the empty string. A *context-free* (CF) *grammar* G is denoted by a 4-tuple (V_T, V_N, P, S), where V_T is the terminal alphabet, V_N is the nonterminal alphabet, P is the production set (also known as rule set), and S is the axiom. A production in P is denoted by $A \to \alpha$, where $A \in V_N$, $\alpha \in V^*$, where $V = V_T \cup V_N$ is the set of *grammar symbols*; A is called the left-hand side (l.h.s.) of the production and α the right-hand side (r.h.s.). The symbol $\Rightarrow$ denotes the immediate derivation relation between two strings in V^* and $\overset{*}{\Rightarrow}$ (resp. $\overset{+}{\Rightarrow}$) its reflexive and transitive (resp. transitive) closure. $L(G)$ denotes the language generated by G. For all omitted usual definitions of formal language concepts, we refer the reader to any textbook on formal languages and automata theory, e.g., [12]. We assume every grammar to be *reduced*, i.e, every nonterminal is reachable from the axiom and generates some string.

The following naming convention will be adopted, unless otherwise specified: lowercase Latin letters at the beginning of the alphabet ($a, b, \ldots$) denote terminals, uppercase ones ($A, B, \ldots$) denote nonterminals; uppercase Latin letters at the end of the alphabet denote grammar symbols; letters r, s, t, u, v, $\ldots$ denote terminal strings; and Greek letters ($\alpha, \beta, \ldots, \omega$) denote strings over V. All the strings may be empty, unless stated otherwise. In this paper it is important to distinguish between the terminal strings x derived by a grammar, i.e., such that $S \overset{*}{\Rightarrow} x$, and *sentential forms*, i.e., those $\alpha \in V^*$ such that $S \overset{*}{\Rightarrow} \alpha$.

In this initial part we will use arithmetic expressions, which are a small fraction of practically all programming languages, as a running example to introduce and explain the basic definitions, properties and constructions referring to OPLs.

Example 2.1. Consider the usual arithmetic expressions with parentheses, containing only additions and multiplications. Fig. 1 presents a grammar and the derivation tree for the expression $n + n \times (n + n)$. Notice that the structure of the syntax tree (uniquely) corresponding to the input expression reflects the precedence order which drives computing the value attributed to the expression.

Fig. 2 proposes a different grammar which generates the same expressions as the grammar of Fig. 1 but would associate to the same sentence the syntax tree displayed in the right part of the figure, corresponding to the fully parenthesized string $(n + (n \times (n + n)))$. Yet another (ambiguous) grammar could generate both.

A string β is in *Operator Form* (OF) if it has no adjacent nonterminals. In an OF string α two terminals are *consecutive* if they are at positions $\alpha[j], \alpha[j+1]$; or at positions $\alpha[j], \alpha[j+2]$ and $\alpha[j+1] \in V_N$. A rule is in *operator form* if its r.h.s. is in OF; an *operator grammar* (OG) contains only such rules. Both the grammars shown in Example 2.1 are OGs.

It is well known that, for every CF grammar, an equivalent OG can be obtained (see for instance [12]). For an OG, every string β that derives from a nonterminal is in OF.

Given an OG G and a nonterminal A, the *left and right terminal sets* are defined as

$$\mathcal{L}_G(A) = \{a \in V_T \mid A \overset{*}{\Rightarrow} Ba\alpha\} \qquad \mathcal{R}_G(A) = \{a \in V_T \mid A \overset{*}{\Rightarrow} \alpha aB\}$$

where $B \in V_N \cup \{\varepsilon\}$. The grammar name G will be omitted unless necessary to prevent confusion. For the grammar of Fig. 1 the left and right terminal sets of nonterminals S, A and B are, respectively:

$$\mathcal{L}(S) = \{+, \times, n, (\} \qquad \mathcal{R}(S) = \{+, \times, n,)\}$$
$$\mathcal{L}(A) = \{\times, n, (\} \qquad \mathcal{R}(A) = \{\times, n,)\}$$
$$\mathcal{L}(B) = \{n, (\} \qquad \mathcal{R}(B) = \{n,)\}$$

$$S \to A \times S \mid A$$
$$A \to A + B \mid B$$
$$B \to n \mid (S)$$

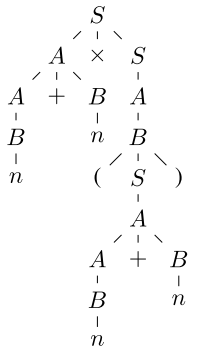


Fig. 2. A grammar generating the same arithmetic expression as that of Fig. 1 and the corresponding tree where, instead, $+$ takes precedence over $\times$.

	$+$	$\times$	$($	$)$	n
$+$	$\gtrdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\lessdot$
$\times$	$\gtrdot$	$\gtrdot$	$\lessdot$	$\gtrdot$	$\lessdot$
$($	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$
$)$	$\gtrdot$	$\gtrdot$		$\gtrdot$	
n	$\gtrdot$	$\gtrdot$		$\gtrdot$	

Fig. 3. The OPM of the grammar in Fig. 1.

The following binary *operator precedence* (OP) *relations* are defined between consecutive terminals:

$$\text{equal in precedence: } a \doteq b \iff \exists A \to \alpha a B b \beta,\ B \in V_N \cup \{\varepsilon\}$$
$$\text{takes precedence: } a \gtrdot b \iff \exists A \to \alpha D b \beta,\ D \in V_N \text{ and } a \in \mathcal{R}_G(D)$$
$$\text{yields precedence: } a \lessdot b \iff \exists A \to \alpha a D \beta,\ D \in V_N \text{ and } b \in \mathcal{L}_G(D)$$

Notice that, unlike the usual arithmetic relations denoted by similar symbols, the above precedence relations do not enjoy anyone of transitive, symmetric, reflexive properties.

The *operator precedence matrix* (OPM) $M = OPM(G)$ is a $|V_T| \times |V_T|$ matrix that maps any ordered pair (a, b) into the set M_{ab} of OP relations holding between a and b.

Fig. 3 displays the OPM associated with the grammar of Fig. 1 where, for an ordered pair (a, b), a is one of the symbols shown in the first column of the matrix and b one of those occurring in its first line.

Definition 2.2. An OG G is an *operator precedence grammar* (OPG) if, and only if, $M = OPM(G)$ is a *conflict-free* matrix, i.e., $\forall a, b,\ |M_{ab}| \leq 1$.

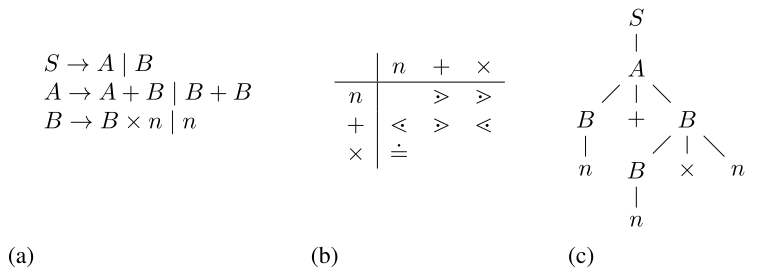
Definition 2.3. An OPG is in *Fischer normal form* (FNF) [20] if and only if the axiom S does not occur in any r.h.s.; no two rules have the same r.h.s.; no rule, possibly except one with the axiom as the l.h.s., has ε as the r.h.s.; renaming rules, i.e., those with a single nonterminal character as r.h.s., are those and only those with S as the l.h.s.

Working with OPGs in FNF removes ambiguity without affecting generality and allows us to simplify the parsing algorithm, without impairing the effectiveness and improving the efficiency of the tools that implement it.

Intuitively, an OPG parser is driven by precedence relations as outlined in the following.

- The source string s is enclosed between a pair $\langle \#, \# \rangle$, where the special symbol $\# \notin V_T$ is used for *end-marks*; by convention, # yields precedence to every terminal character and every terminal character takes precedence over #. Moreover it holds that $\# \doteq \#$; thus all r.h.s.s in a sentential form will be enclosed within a pair $\lessdot$, $\gtrdot$.
- Consider a rule $A \to \beta$, whose r.h.s. β occurs in a sentential form and is going to be reduced to A. Then β is "enclosed" between the pair $\lessdot$, $\gtrdot$, and relation $\doteq$ holds between every two consecutive terminal symbols of β. More formally, if there exists a derivation $S \overset{*}{\Rightarrow} \alpha A \gamma \Rightarrow \alpha \beta \gamma$ then it must hold that $\alpha = \alpha' a$, $\gamma = b\gamma'$, $\beta = N_1 c_1 N_2 c_2 \ldots c_{n-1} N_n$, with $N_i \in V_N \cup \{\varepsilon\}$, $c_i \doteq c_{i+1}$, $1 \leq i < n$, $a \lessdot c_1$, $c_{n-1} \gtrdot b$. Note also that a and/or b may be #. Observe that nonterminals are "transparent" in OP parsing, i.e., they are not considered when evaluating the precedence relations between *consecutive* terminals.

Continuing our running example to illustrate the properties of OPGs and of the parsing algorithms operating on them, Fig. 4 reports: a simplified version of the grammar of Fig. 1 of arithmetic expressions without parentheses (a); its OPM (b) with



(a) (b) (c)

Fig. 4. (a) OP grammar in FNF generating arithmetic expressions without parentheses, (b) its precedence matrix, (c) a sample syntax tree.

the relations with # taken as implicit; and the derivation tree of string $n+n\times n$ (c). Note that in the syntax tree the second occurrence of terminal n is enclosed by the relations $+\lessdot n$ and $n\gtrdot\times$ and can be reduced unequivocally to nonterminal B, thanks to the fact that G has no repeated r.h.s. Similarly the r.h.s. $B\times n$, in the context $\langle +,\#\rangle$ with $+\lessdot\times$ and $n\gtrdot\#$, is deterministically reduced to nonterminal B. For a detailed description of sequential OP parsing see, e.g., [28].

As previously stated, OPGs do not have the power to generate all the deterministic languages: this is practically embodied in the presence of a precedence conflict in the grammar *OPM*. For instance, there is no way to generate the language $L=\{a^nba^n \mid n\geq 1\}$ without introducing in the OPM the conflict $|M_{aa}|>1$. Indeed, matching n requires (at least) the conflicting precedences: $a\lessdot a$ and $a\gtrdot a$, because a here has intuitively both the roles of open and closed parenthesis. Despite this theoretical limitation, early experience, including our own, has shown that many real programming languages can be generated by OPGs.

2.2. Local parsability property and its exploitation for parallel parsing

Intuitively, a language L generated by a CF grammar G is locally parsable if, for every sentential form, the r.h.s. of a production to be reduced can be uniquely determined through inspecting only a bounded context of the r.h.s. For instance, the language $L=\{a^n0b^n\}\cup\{a^n1b^{2n}\}$ generated by the grammar $S\to A\mid B$; $A\to aAb\mid a0b$; $B\to aBbb\mid a1bb$, is locally parsable because the "separators" $\{0,1\}$ allow to decide the r.h.s. to be reduced. By contrast, $L=\{0a^nb^n\}\cup\{1a^nb^{2n}\}$, though being deterministic and generated by the LR grammar $S\to 0A\mid 1B$; $A\to aAb\mid ab$; $B\to aBbb\mid abb$, is not locally parsable since there is no way to decide whether to reduce a substring ab to A or abb to B without inspecting the first character of the string, which may be arbitrarily far away. The concept of local parsability has been formalized in the literature in similar ways; we adopt the definition of bounded-context CF grammar [22].

Definition 2.4. Let G be a CF grammar and $h\geq 1$. G is a *locally parsable* (or *bounded context*) grammar *with bound* h, iff for every rule $A\to\alpha$ of G, whenever

$$\#^h S\#^h \overset{*}{\Rightarrow} \zeta=\beta\gamma A\delta\eta \Rightarrow \beta\gamma\alpha\delta\eta \overset{*}{\Rightarrow} x \tag{1}$$

with $|\gamma|=|\delta|=h$, any other derivation $\#^h S\#^h \overset{*}{\Rightarrow} \vartheta\gamma\alpha\delta\phi$ where h,γ,α,δ are the same as before, can be obtained exclusively by using the same rule $A\to\alpha$ to obtain α.

Thus, h specifies the length of the left and right neighborhood, i.e., the surrounding *context*, needed to make sure that string α must be reduced to nonterminal A. Floyd proved that, besides being decidable for any given value of h, the local parsability property implies that G is deterministically parsable (therefore also unambiguous).

Theorem 2.5. *Every OPG in FNF is locally parsable with bound 1.*

Proof. Consider the step $\beta\gamma A\delta\eta\Rightarrow\beta\gamma\alpha\delta\eta$ of derivation (1), with $|\gamma|=|\delta|=1$. Then, necessarily $\gamma,\delta\in V_T\cup\{\#\}$, $\gamma\lessdot$ the first terminal of α, and the last terminal of $\alpha\gtrdot\delta$; only $\doteq$ occur within α. Then, whenever a string $\gamma\alpha\delta$ occurs in a sentential form of G, the same precedence relations hold between its terminals since G's OPM is conflict-free; thus α is the r.h.s. of some rule and, since G is in FNF, no other rule can produce α in a sentential form within the context (γ,δ). □

Although there exist contrived examples of locally parsable languages with bound 1 that cannot be generated by an OPG, such as the above language $\{a^nba^n \mid n\geq 1\}$, they are of no practical relevance when taking into account real world programming language.

The following corollary establishes the basis for parallelizing the standard OP parsing algorithm.

Corollary 2.6. *For every substring $a\,\delta\,b$ of a sentential form $\gamma\,a\,\delta\,b\,\eta$, there exists a unique string α, called the irreducible string, deriving δ such that $S\overset{*}{\Rightarrow}\gamma\,a\,\alpha\,b\,\eta\overset{*}{\Rightarrow}\gamma\,a\,\delta\,b\,\eta$, and the precedence relations between the consecutive terminals of $a\alpha b$ do not contain*

Algorithm 1: Generalized-OP-parsing(α, head, end, $\mathcal{S}$).

1. Let $X = \alpha$[head] and consider the precedence relation between the top-most terminal Y found in $\mathcal{S}$ and X.
2. If $Y \lessdot X$, push $(X, \lessdot)$; head := head + 1.
3. If $Y \doteq X$, push $(X, \doteq)$; head := head + 1.
4. If $X \in V_N$, push $(X, \perp)$; head := head + 1.
5. If $Y \gtrdot X$, consider $\mathcal{S}$:
   (a) If $\mathcal{S}$ does not contain any $\lessdot$ then push $(X, \gtrdot)$; head := head + 1.
   (b) Else, let $\mathcal{S}$ be $(X_0, p_0)(X_1, p_1)\dots(X_{i-1}, p_{i-1})(X_i, \lessdot)\dots(X_n, p_n)$ where $\forall j, i < j \le n, p_j \neq \lessdot$.
      i. if $X_{i-1} \in V_N$ (hence $p_{i-1} = \perp$), and there exist a rule $A \to X_{i-1}X_i \dots X_n$ replace $(X_{i-1}, p_{i-1})(X_i, \lessdot)\dots(X_n, p_n)$ in $\mathcal{S}$ with $(A, \perp)$;
      ii. if $X_{i-1} \in V_T \cup \{\#\}$, and $\exists A$: $A \to X_i \dots X_n \in R$, replace $(X_i, \lessdot)\dots(X_n, p_n)$ in $\mathcal{S}$ with $(A, \perp)$;
      iii. otherwise start an error recovery procedure.
6. If (head < end) or (head = end and $\mathcal{S} \neq (a, \perp)(B, \perp)$), for any $B \in V_N$, repeat from step (1);
   else return $\mathcal{S}$.

the pattern $\lessdot \left(\doteq\right)^* \gtrdot$. *Therefore there exists a factorization* $a\alpha b = \zeta\theta$ *into two possibly empty factors such that the left factor does not contain* $\lessdot$ *and the right factor does not contain* $\gtrdot$.

Proof. Consider the substring $a\delta b$: any r.h.s. contained therein is preceded by $\lessdot$ and followed by $\gtrdot$, and we reduce it to its l.h.s. Then we iterate the procedure until no pair $\lessdot \dots \gtrdot$ (with possibly $\doteq$ in between) exist. At this point, necessarily, the condition of the corollary has been reached. □

The parsing algorithm To allow its use in parallel parsing, we generalize the traditional OPG parsing algorithm in order to analyze strings that may include nonterminals: such strings must begin and end with terminals or with #, and are in OF. This generalization is needed in the parallel setting in order to parse internal text segments, and is reported in Algorithm 1. Algorithm 1 uses a stack $\mathcal{S}$ containing symbols that are pairs of type (X, p), where $X \in V_T \cup V_N$ and p is one of the precedence symbols $\{\lessdot, \doteq, \gtrdot\}$ or is undefined, denoted by $\perp$. The second component encodes the precedence relation found between two consecutive terminals – thus, it is always $p = \perp$ if X is nonterminal. To drop the precedence symbols when not needed, we introduce the projection on the first component, defined and denoted by $(X, p)|_1 = X$. As a convention, we assume that the stack grows rightwards. We define a *handle* as a candidate r.h.s., i.e. a portion of a string in OF included within a pair $\lessdot$, $\gtrdot$ and with $\doteq$ between consecutive terminals.

The algorithm takes as input the string $\alpha = \alpha[1]\alpha[2]\dots\alpha[m]$, $m \ge 3$ to be parsed: α is in OF and has the form $\alpha \in (V_T \cup \{\#\})V^*(V_T \cup \{\#\})$. It also receives as input parameters two pointers, head and end, to elements of α pointing to the second and last element of α, respectively. The last parameter taken is the parsing stack $\mathcal{S}$, initialized with $\alpha[1]$ on top of it.

Remarks.

- We will see that initially the algorithm is applied to a terminal input string that is a substring of the input text. Therefore it will be $\alpha = asb$, with $a, b \in (V_T \cup \{\#\})$, $s \in V_T^*$ and the stack $\mathcal{S} = (a, \perp)$, consequentially in this case the condition in step (4) will not be met.
- We note that Algorithm 1 behaves as a traditional sequential OP parser when $\alpha = \#s\#$ and $\mathcal{S} = (\#, \perp)$. In this case the input is accepted if, and only if, the algorithm halts having read the whole input and $\mathcal{S} = (\#, \perp)(S, \perp)$.
- If the initial stack – disregarding precedence symbols – $\mathcal{S}|_1$ is irreducible (more precisely it has the form $a\alpha b$ of Corollary 2.6), then the same property will hold for $\mathcal{S}$ upon algorithm termination (unless an error occurs); in other words the property of Corollary 2.6 is an invariant w.r.t. the algorithm execution. As a particular case, this is true when the initial stack is the singleton $(a, \perp)$. In fact, the creation of a reduction handle at runtime can only happen if a $\gtrdot$ is going to be pushed on the stack when a $\lessdot$ is already in it. However, Algorithm 1 is designed to perform a reduction in such cases thus eliminating the possibility of an non-reduced handle sitting on the stack.
- An error is detected either whenever an handle does not match any of the valid r.h.s., or if no precedence relation holds between two consecutive terminals. In these cases then an appropriate error recovery strategy can be started.

We will now describe the parallel parsing strategy mapping the computation on multiple workers and recombining their results to obtain the full parse of the input. We chose to use the term *worker* to denote the independent unit of processing in an abstract way from the chosen architecture. In principle it could even be a virtual process mapped into a mono-processor architecture, though in this case obviously there would be no benefit in terms of speed-up.

Let k be the number of available workers: the input source string is split into k substrings in an arbitrary fashion. We note that, despite there is no functional constraint concerning the splitting of the input token stream, this freedom is not incompatible with profitable heuristics for choosing the substrings, e.g., based on suitable pre-processing during lexical analysis.

Algorithm 1 is applied to each substring, obtaining a partial parse, which (thanks to the local parsability property) is a correct portion of the complete parse tree. Since OP parsing needs a look-ahead/look-back of one character to evaluate the

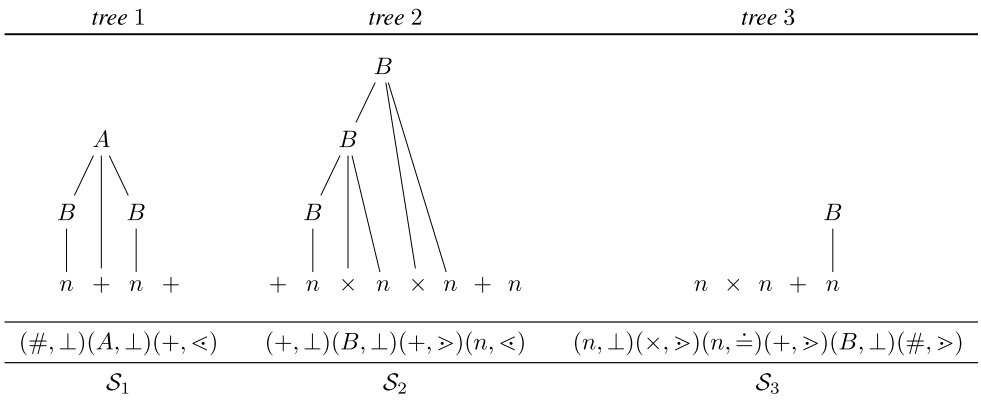


Fig. 5. Partial trees and corresponding stacks after the first parallel pass on text $n+n+n\times n\times n+n\times n+n$.

precedence relations between consecutive terminals, when the source string is split, we leave a 1-character overlap between consecutive substrings.

For the sake of clarity, we will detail this procedure by means of an example. Consider the grammar of Fig. 4, assume that $k=3$, and segment the source text: $\#\,n\,+\,n\,+\,n\,\times\,n\,\times\,n\,+\,n\,\times\,n\,+\,n\,\#$ into:

$$\#\overbrace{n\,+\,n}^{1}\;+\;\overbrace{n\,\times\,n\,\times\,n+}^{2}\;n\;\overbrace{\times n\,+\,n\#}^{3}$$

where the unmarked symbols $+$ and n are shared by the adjacent segments, and are used for look-ahead and look-back. After each parser has processed its segment, the partial trees and the stacks are shown in Fig. 5.

Thanks to Corollary 2.6, after a sequential step the stack contents $\mathcal{S}$ of each worker can be split into two parts $\mathcal{S}^L$ and $\mathcal{S}^R$, such that $\mathcal{S}^L$ does not contain $\lessdot$ relations, and $\mathcal{S}^R$ does not contain $\gtrdot$ relations (in case of several $\doteq$ between the last $\gtrdot$ and the first $\lessdot$, the separation between the two parts is arbitrary). Notice either one of $\mathcal{S}^L$ or $\mathcal{S}^R$ may be empty. In our example, $\mathcal{S}_1^L$ and $\mathcal{S}_3^R$ are empty, while the workers produce the stacks:

$(\#,\perp)\ (A,\perp)\ (+,\lessdot)$	$\mathcal{S}_1=\mathcal{S}_1^R$
$\overbrace{(+,\perp)(B,\perp)(+,\gtrdot)}^{\mathcal{S}_1^L}\ \overbrace{(n,\lessdot)}^{\mathcal{S}_2^R}$	$\mathcal{S}_2$
$(n,\perp)\ (\times,\gtrdot)\ (n,\doteq)\ (+,\gtrdot)\ (B,\perp)\ (\#,\gtrdot)$	$\mathcal{S}_3=\mathcal{S}_3^L$

To prepare the input for the next pass, we could simply concatenate the outputs, i.e., the stack contents delivered by the workers of the first pass, erase their precedence components by applying the $|_1$ projection, iterating the same schema as in the first pass, i.e. splitting the obtained string again into $k' \le k$ chunks to be assigned to k' workers. Instead we propose an heuristic approach aiming at maximizing the chance to produce a complete sub-tree or at least to include a fairly large one as soon as possible. Intuitively, we try to achieve such a goal by pairing strings containing $\lessdot$ – at the left – with others containing $\gtrdot$ – at the right.

Fig. 6 depicts the construction of the initial configuration of the stacks and inputs for pass two. Let $\mathcal{W}$ and $\mathcal{W}'$ be consecutive workers of the previous pass, and let their bipartite stacks be $\mathcal{S}^L\,\mathcal{S}^R$ and $\mathcal{S}'^L\,\mathcal{S}'^R$. We define the stack initialization function as $\mathcal{S}_{combine}(\mathcal{S}^L,\ \mathcal{S}^R) = (a,\perp)\,\mathcal{S}^R$ where a is the top symbol of $\mathcal{S}^L$. Note that the precedence value listed with a, becomes undefined since in the new stack a is not preceded by a terminal. The input string initialization function is defined as $\alpha_{combine}(\mathcal{S}'^L) := \alpha'$, where α' is the suffix of $\mathcal{S}'^L|_1$ without its first symbol (which is already on the top of $\mathcal{S}^R$).

Note that, in case $\mathcal{S}'^L$ (or, symmetrically, $\mathcal{S}^R$) is empty, $\mathcal{S}^R$ is simply concatenated with $\mathcal{S}'^R$ and the output of the following worker, say $\mathcal{S}''^L\ \mathcal{S}''^R$ is used to complete the construction of the new pair $(\mathcal{S}, u)$. Another notable exception where the aforementioned stack composition strategy cannot be applied as-is is the $\mathcal{S}^L$ component of the leftmost worker's output (symmetrically, the $\mathcal{S}^R$ component of the rightmost one), as it will always be empty. As a consequence, the initial stack of the new input for the leftmost worker will be $\mathcal{S}_1^R$ (concatenated with $\mathcal{S}_2^R$ if $\mathcal{S}_2^L$ is empty).

The complete parallel parsing schema is summarized by Algorithm 2, which provides a complete schema for parallel parsing a generic string β by means of k workers.

Remark. Algorithm 2 is just a "core formulation" obviously amenable to several variations and improvements. The most relevant one concerns the number of passes of parallel parsing: employing many workers when the whole input size is small enough to be assigned to a single worker may obviously incur in performance penalties due to the worker spawning overhead. For instance, whenever the source string exhibits a fairly balanced structure (e.g., many functions of comparable

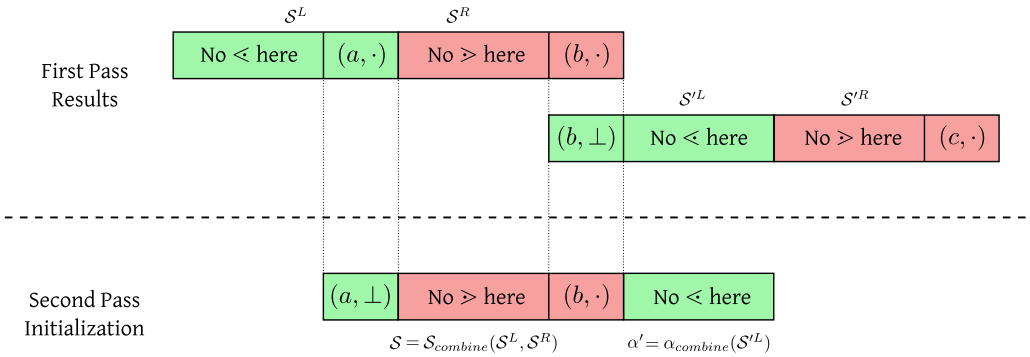


Fig. 6. Preparation of the initial stack and input string for the next parsing phase of a worker: the stack is $\mathcal{S}$ and the input string is α'.

Algorithm 2: Parallel-parsing(β, k).

1. Split the input string β into k substrings: $\#\beta_1\beta_2 \dots \beta_k\#$.
2. Launch k instances of Algorithm 1, where, for each $1 \le i \le k$, the parameters are $\mathcal{S} = (a, \perp)$, $\alpha = a\beta_i b$, head $= |\beta_1\beta_2 \dots \beta_{i-1}| + 1$, end $= |\beta_1\beta_2 \dots \beta_i| + 1$; a is the last symbol of β_{i-1}, and b the first of β_{i+1}. Conventionally $\beta_0 = \beta_{k+1} = \#$. The result of this pass are $k' \le k$ pairs of stacks $\mathcal{S}_i^L \mathcal{S}_i^R$, as specified above.
3. Repeat:
   (a) For each adjacent non-empty stack pair $\mathcal{S}_i^L \mathcal{S}_i^R$ and $\mathcal{S}_{i+1}^L \mathcal{S}_{i+1}^R$, launch an instance of Algorithm 1, with $\mathcal{S} = \mathcal{S}_{combine}(\mathcal{S}_i^L, \mathcal{S}_i^R)$, $\alpha = \alpha_{combine}(\mathcal{S}_{i+1}^L)$, head $= 1$, end $= |\alpha|$.
   (b) Until either we have a single reduced stack $\mathcal{S}'$ or the computation is aborted and some error recovery action is taken.
4. Return $\mathcal{S}'$.

size) it is likely for the first pass to produce fairly short stacks suitable to be concatenated into a unique string. By contrast, the partition feeding the first pass may generate chunks which do not correspond to large sub-trees of the complete syntax tree. In this case, the construction of the new chunks by pairing $\mathcal{S}^R$ sides with $\mathcal{S}^L$ ones should increase – if not maximize – the number of handles belonging to the same chunk, and therefore the number of reductions performed by the second – and possible subsequent – passes. Relying on our experience, we note that no more than two parallel passes are needed to produce a small enough input for the final pass.

2.2.1. Algorithm complexity

In terms of asymptotic complexity, the requirements that we state for a positive evaluation of the whole approach are: a best-case linear speedup w.r.t. the number of processors and a worst-case complexity not exceeding the one of a fully sequential parsing.

By inspecting Algorithm 1 and Algorithm 2, it is clear that the total number of elementary operations (shifts and reductions) is $\mathcal{O}(n)$ since no reduction is performed more than once exactly like the sequential case. Indeed, some terminal symbol could be shifted more than once during the various passes, but this occurs only for the few of them which have not been reduced by the previous passes, and for a number of times that does not exceed the number of passes.

To achieve a worst case parsing time not exceeding the sequential parsing, it is essential that the combination of stacks $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$, inside step (3)(a) of Algorithm 2, takes $\mathcal{O}(1)$ time (hence overall $\mathcal{O}(k)$ for k workers). A possible technique to achieve this goal consists in storing, during the execution of Algorithm 2, a marker that keeps track of the separation between $\mathcal{S}^L$ and $\mathcal{S}^R$. Such a marker can be initialized at the position where the first $\lessdot$ sign is detected and then updated every time a reduction is applied that removes the sign and a new element is shifted on the stack as a consequence of a new $\lessdot$ relation.

For instance, in the case of $\mathcal{S}_2$ in Fig. 5, the marker is initialized at the position of the first $+$ symbol and remains there after the three reductions $B \Rightarrow n$, $B \Rightarrow B \times n$, $B \Rightarrow B \times n$ since $+ \lessdot n$ and $+ \lessdot \times$. When the second $+$ (the third of the whole string) is shifted (without removing the previous one as the $\gtrdot$ between the two $+$ is not matched by a corresponding $\lessdot$ at its left), it is moved to the position of the second $+$ as $+ \lessdot n$, where it marks the beginning of $\mathcal{S}_2^R$.

These operations require $\mathcal{O}(1)$ time regardless of the stacks being implemented by means of arrays or by means of more flexible linked lists; thus, they do not affect the overall $\mathcal{O}(n)$ complexity of the whole algorithm.

It is then clear that the ideal linear speed-up w.r.t. the number of processors will be most representative of the actual one whenever most of the parsing is done during the first pass. By contrast, the worst case occurs when either only $\lessdot$ relations or $\gtrdot$ relations are present in the whole input; this is the case of regular languages respectively generated, e.g., by a left-linear or by a right-linear grammar. In such cases, only one worker (respectively the leftmost or rightmost one) performs useful parsing whereas the others leave their input unaffected. The second pass would produce a unique string – of length $(k-1)/k \cdot n$ – which would be parsed sequentially.

3. Parallelization of lexical analysis

Lexical analysis takes place before parsing and translation, and it is a common belief that it is a fairly easy and less time consuming job compared with the following phases. While this may be true in other settings, we report that lexical and syntax analysis for operator precedence languages often require comparable effort. Thus the gain in performing parallel parsing alone would be small without coupling it with parallel lexical analysis and preprocessing. Furthermore, apart from a few idiosyncrasies of some languages – which tend to go unused by programmers – lexical analysis is even better suited for parallel execution. However, to achieve this goal a few non-trivial technical difficulties must be tackled.

In this section we present a fairly general schema for parallelizing lexical analysis, which can be applied to most programming languages. A distinguishing feature of our lexical analysis is that it produces a stream of tokens which, rather than being compatible with the original BNF of the source language, is ready to be parsed according to an "OP version" of the official grammar, thus yielding an advantage from both a performance, and an adaptation to OP parsing point of view. Typically such a preprocessing allows for disambiguating some terminals which are overloaded in the language and would induce conflicting relations in the operator precedence matrix: depending on the context in which they occur, the lexer can associate them to distinct token classes, so that the resulting string of tokens can be parsed according to the OP version of the syntactic grammar. Note that such a disambiguation is complementary to the usual operations performed in the lexing phase to cope with the presence of ambiguities in the lexical grammar (as, e.g., in the presence of reserved words and identifiers corresponding to the same lexical pattern). However, this is hidden from the user who does not have to worry about the internal format.

We now provide the definitions required to describe lexical analysis parallelization.

The lexicon of the language is described by a lexical grammar, which assumes as terminal alphabet the characters present in the input stream. Often, a lexical syntax can be analyzed by means of a finite state machine (FSM) as opposed to a pushdown one, which is reserved for parsing. In this work we will tackle both a language where this assumption holds (JSON), and one where it does not (Lua) in Section 4, and will describe the issues in generalizing the approach in Section 8.

We adopt the following conventions to distinguish terminal and nonterminal symbols of syntactic and lexical grammars: terminals of syntactic grammars and nonterminals of lexical grammars are in boldface font: $\boldsymbol{n}, \boldsymbol{s}\ldots$, nonterminals of syntactic grammars are denoted by capital letters: $A, B\ldots$, and terminals of lexical grammars are in monospace font: $\texttt{if}, \texttt{+}\ldots$. We also introduce some basic terminology (as in, e.g., [1]) which tailors some general terms to the scope of this section.

Definition 3.1.

- A *lexeme* is a sequence of characters corresponding to a valid sentence of the lexicon grammar (e.g., a built-in identifier, a reserved keyword, an operator). Its form depends on the lexical part of the language definition, and it can be typically recognized by a FSM.
- A *string* is a lexeme built as a sequence of characters enclosed within a pair of delimiters, typically either single or double quotes. It cannot contain any other delimiters of the same kind without proper escaping (e.g. prefixing them with a `\` character). Strings may contain control characters (e.g. newlines).
- A *token* is a pair ⟨*token-name*, *semantic value*⟩ resulting from the analysis of a sequence of characters matching the form of a valid lexeme (*token-name* denotes a nonterminal of the lexical grammar, and there is a finite number thereof). Sample token instances are ⟨**LPAREN**, `(`⟩, or ⟨**STRING**, `"yesterday I ate too much"`⟩. Since the focus of this work is the lexical and syntactic analysis of a text, from now on we will identify the token with the first element of the pair.
- A *comment* is a sequence of characters delimited by special symbols according to language dependent rules, and does not correspond to a lexeme. A comment should be matched and discarded during the lexing process. Many languages use different markers for single-line and multi-line comments.

The goal of lexical analysis (*lexer*) is to recognize the lexemes in the source character stream and generate a sequence of tokens, removing the comments. The lexical grammar, in spite of the fact that it typically defines a regular language, may be not locally parsable in its immediate form and in most cases is ambiguous. Yet, lexical analysis can be made suitable for parallel execution and, given the typical "flat structure" of programming language lexicon, is a more natural candidate for efficient parallelization than parsing, which has to deal with the nesting of syntactic structures, as in fact it happened in practice (see the discussion in Section 7). To achieve this goal, however, two issues must be addressed. First of all, splitting the source text randomly into chunks to be processed by parallel workers may split a lexeme across different segments. Thus, the results produced by lexers working on adjacent chunks will have to be reconciled to cope with this issue.

The second issue concerns the occurrence of very long comment sections, possibly longer than the chunk assigned to a single worker. It is commonplace among some programmers to comment portions of obsolete or temporary code, effectively preventing the lexer from knowing if it is analyzing a portion of a comment or not, in the case of lack of comment delimiters in its chunk. This is further exacerbated by the fact that some languages adopt some exotic syntactic rules for comment and/or string delimiters in general. For instance, in Lua strings can be delimited either by quotes or by opening and closing symbols of unbounded number of forms.

To cope with this problem, we accept a minimum amount of nondeterminism during the lexing phase only. We run several speculative computations for each worker, corresponding to the different states of the lexing machine which are

legitimate on the splitting point of the input stream. If the worker is able to remove the ambiguity during the analysis of its chunk, the incorrect computation(s) are halted and only the correct one proceeds. We will show that the number of language dependent simultaneous computations, never exceeds 3 in our case.

Therefore, every worker will produce several candidate token lists for its chunk. Any disambiguation that cannot be performed during the single worker analysis is done when the partial token lists are joined into a single one. For instance, assume that a worker w reaches a given state s_k after analyzing completing the lexical analysis of its input chunk. If there is a computation starting with s_k among the ones performed by the worker acting on the input chunk after the one assigned to w, the corresponding candidate token list is merged with the output of w.

We now illustrate our approach to parallel lexical analysis in a similar way to the path followed in the case of parsing: we first revise normal sequential lexers to make them suitable to work on partial chunks and to produce partial outputs to be later integrated; then we will show how a parallel lexer splits the source code into chunks, assigns them to different workers, and reassembles their partial outputs. We will also make use of a running example to better illustrate the various steps of our algorithms.

Example 3.2. Consider the grammar generating arithmetic expressions presented in Fig. 4, extended to allow also for operations on strings. The resulting OPG is depicted in Fig. 7(b), whereas the lexical grammar is reported in Fig. 7(a), with ***n*** denoting an identifier and ***s*** a literal string; Fig. 7(c) depicts a FSM recognizing tokens and comments compatible with the lexical grammar; to keep its appearance easily understandable we restricted its description to the recognition of these fundamental elements of the lexicon; however, its extension to cope with a whole chunk consisting of a sequence of such elements, possibly broken at the boundaries of the chunk, is conceptually straightforward.

The lexical grammar specifies two formats for strings: single-line ones are delimited by single-quotes `' '`, multi-line ones delimited by a pair of triple-double-quotes `""" """`. The lexicon also allows for introducing both single-line and multi-line comments with a C-like syntax: a multi-line comment is delimited by `/*` and `*/`, while a single-line comment begins with `//`.

In our running example we will refer to the sample code snippet in Fig. 8.

If such a complete code snippet were supplied to the FSM of Fig. 7(c), the output produced by the lexer would be (for completeness we include also the semantic components of the tokens): ⟨***n***, `var`⟩ ⟨**+**, +⟩ ⟨***n***, `x`⟩ ⟨**×**, ×⟩ ⟨***n***, `y`⟩ ⟨**+**, +⟩ ⟨***s***, `a multi-line string interrupted here × timesToConcat +` ⟩ ⟨**+**, +⟩ ⟨***n***, `a`⟩ ⟨**+**, +⟩ ⟨***n***, `z`⟩.

Our goal is to obtain the same result by splitting the job among several workers: in our example we will use three of them.

Source character stream partitioning First, the input stream is split into segments of equal size; most likely, however, such a "blind" split may break a lexeme, typically an identifier which normally is not very long; thus, it is often sufficient to consider a look-ahead/look-back of a few characters to find a lexeme separator (i.e. a white-space): in such cases the splitting point may be conveniently set right after it and the boundaries of the chunks are updated trying to avoid splitting any token in the middle. For instance, in our example of Fig. 8, the original ending points are denoted by | (red); by analyzing a bounded context in the neighborhood of the split points we identify the occurrence of two newline characters, and update the splitting points moving them to the positions denoted by | (blue). In general it cannot be known a priori how far is the end of lexeme from a given point; thus, the length of the search should be stated on the basis of some, possibly language dependent, heuristic criteria and should not exceed a few characters in any case.

Assuming that such a separator is found, the possible ambiguity of the starting state of the lexer reduces either to the beginning of a lexeme, a position within a string, or to a position within a comment. If the search of a separator within a bounded context of the start of the segment is not conclusive, the initial state can also correspond to an internal point of a non-string lexeme. In the worst case, i.e., when a non-string lexeme is cut by the splitting, the worker will need to carry on 4 computations at once. However, we report that no such cases took place in our experimental evaluation.

In our example the worker assigned to the first chunk is deterministic and begins its analysis in the initial state of the FSM of Fig. 7(c); The second and third workers, instead must carry over three simultaneous computations each, starting at the beginning of a token, or inside a multi-line string or inside a comment; thus, the corresponding starting state of the FSM are, respectively, q_0 or q_7 or q_{18} (or q_{19}).

Parallel lexical analysis Once assigned to a given chunk, each worker carries on a computation for each possible alternative initial state of the lexing machine. Most likely, during this phase, some disambiguation among the open alternatives in the lexical analysis may occur: this happens a) if a worker meets a character forbidden in some of the active states, thus aborting the erroneous computations; b) if a sequence of characters causes the recognition of a lexeme on more than one computation, thus collapsing them into a single one. Although this collapse is quite uncommon, we will provide an example thereof when detailing the parallel lexing strategy for the Lua language.

Below we give an abstract version of the algorithm executed by each worker on the assigned chunk.

- Let M be the FSM recognizing the language generated by the lexical grammar; let Σ denote its alphabet, Q its set of states, $\Delta : Q \times \Sigma \to Q$ its transition function, q_0 its initial state and F its set of accepting states. We assume, as usual,

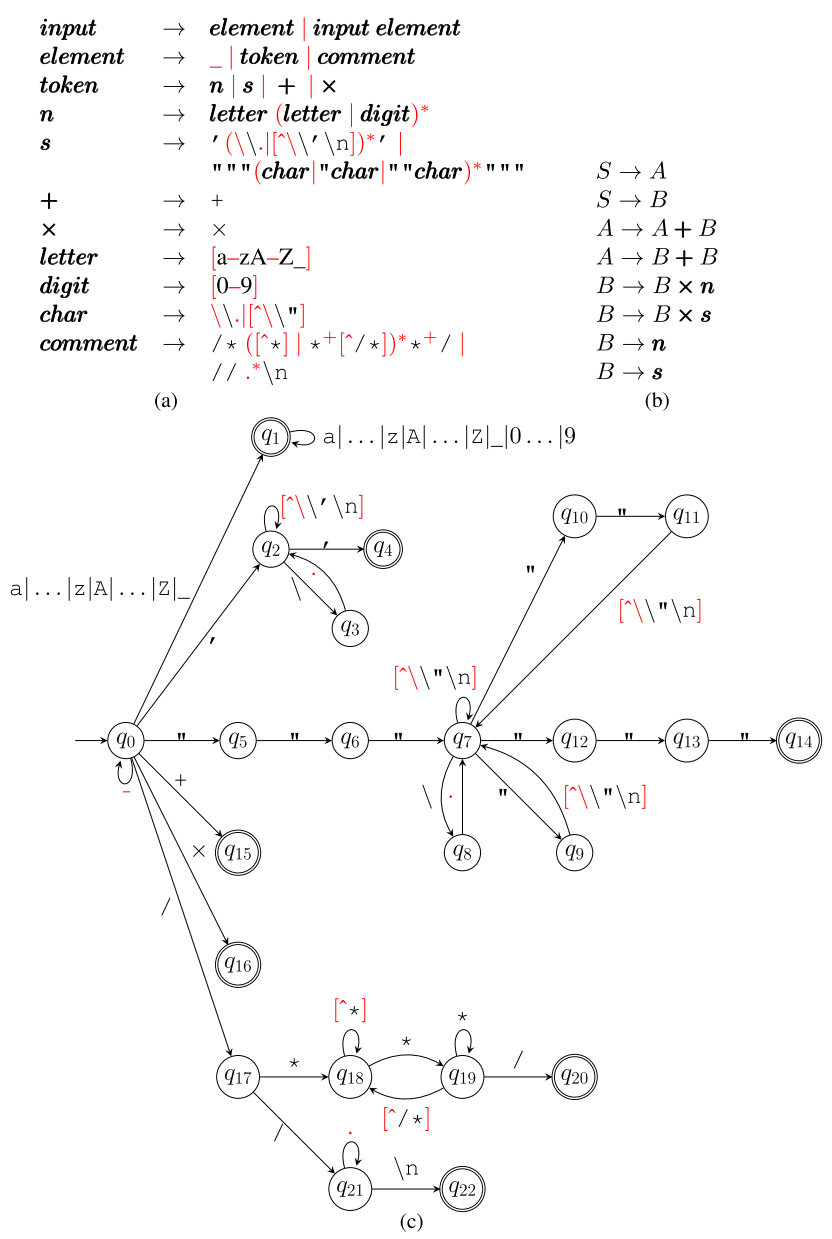


Fig. 7. Lexical (a) and syntactic (b) grammars of arithmetic expressions without parentheses, extended to deal also with strings, and a FSM recognizing tokens and comments of the lexical grammar (c). In figures (a) and (c), metasymbols used in regular expressions are in red while the terminals of the lexical grammar in black; symbol _ stands for any control character (whitespace, newline, etc.).

```
var + x /* This is a multi-line comment that
contains part of an arithmetic expressi|on:|
var + 'string*/×/*' //and this is an embedded single-line comment
+ '6' × */ y +| """a  multi-line string interrupted here|
× timesToConcat + """+a//-(5×b)""" × id
+ z
```

Fig. 8. Sample string generated by the grammars of arithmetic expressions in Fig. 7.

that when an accepting state is reached a lexeme in the input string is recognized and its semantic value is output. The transition function Δ can be partial: we use symbol $\perp$ to denote an undefined value thereof. Δ^* denotes the reflexive and transitive closure of Δ.

Algorithm 3: Sequential-lexing(γ, s).

1. Initialization: head $:= 1$; end $:= |\gamma|$; $X = \gamma$[head].
   The strings in b and the lists in T are set to empty and s is the N-tuple of initial states.
2. For each state s^j $(1 \leq j \leq N)$ such that $s^j \in Q$:
   (a) If $s^j = s^k$ for a k such that $1 \leq k < j$ then merge computations j and k, by setting $s^j := k$ and joining the end of list T^j to the end of list T^k.
   (b) Else, let $q := \Delta(s^j, X)$.
      i. If $q \neq \perp$
         A. If, while in state s^j, M is not reading a whitespace character outside a string or a comment, set $b^j := b^j X$.
         B. $s^j := q$.
      ii. Else
         A. If $s^j \in F$ and $b^j \neq \varepsilon$ then append b^j to T^j and set $s^j := \Delta(q_0, X)$ and $b^j := X$.
         B. Else $s^j := \perp$.
3. $head := head + 1$.
4. If $head \leq end$ repeat from step 2, else return the tuple s (where each state $s^j = i$ $(1 \leq i \leq N$ is updated to $s^j := s^i)$ and the tuple T of lists of tokens.

- Let N be the number of concurrent computations of M and let s be the N-tuple of states currently reached along each of the N computations of M: the undefined state value $\perp$ is used to denote that the computation has been interrupted because of an error in the input string, while a value i $(1 \leq i \leq N)$ is used to denote that the computation has been merged with computation i.
- Let b be a N-tuple of strings where b^j, for $1 \leq j \leq N$, contains the partial semantic value of the token currently under recognition along the j-th computation of M.
- Let T be a N-tuple of lists of tokens.

Example 3.2 (*continued*). Going back to our running example, the worker scanning the first chunk performs only one computation, whose initial state is the initial state the FSM of Fig. 7(c). The worker returns the tokens $\boldsymbol{n + n}$ and finishes in a state specifying the presence of a non-terminated comment. The choice of a newline separator as the ending character of the chunks excludes the possibility that their initial portion belongs to a single-line string.

The second and third chunk are scanned along three simultaneous computations starting at the beginning of a token or inside a multi-line string or inside a comment, respectively. In the case of the second worker, the first of the three computations generates a list of tokens $\boldsymbol{n + s + s \times}$ before being aborted due to the occurrence of the unexpected end of a comment `*/`. Along the second computation a multi-line string is recognized, and the computation is aborted after reading the word `multi-line`. The third computation detects the end of a comment `*/` and produces a list of tokens $\boldsymbol{\times n +}$. The run ends in a state denoting the presence of a non-terminated string, while the prefix which has been read is stored to allow for a possible concatenation with the suffix in the following chunk.

Finally, the third worker carries on three simultaneous computations: the first and second one end in a final state, generating as a list of tokens respectively $\boldsymbol{\times n + s \times n + n}$ and $\boldsymbol{s + n + n}$. The third one does not collect any token and ends in a state which signals the presence of a non-terminated comment.

List joining phase Once the various workers have processed their chunks in parallel their partial outputs must be integrated into a unique sequence of tokens to be supplied (after further partitioning) to the parallel parsing phase. This job can be done in a similar way to the case of parsing, with two important differences:

- Whereas each worker in the parsing phase delivers just one output, in general parallel lexers will produce several candidate outputs among which the integration phase will choose the right one.
- Whereas after a first parsing pass further parsing is applied, possibly in more than one pass, the partial outputs of the parallel lexical analysis need only to be selected and integrated without further analysis.

The integration of the partial outputs is carried over sequentially. Since the leftmost worker W_1 performs a deterministic computation, its final state is the correct initial one for the following worker W_2. The correct list among the ones produced by its right neighbor is thus selected by matching the final state of the first worker against one of the members of s^2. In case no match occurs, an error is signaled and error recovery strategies are enacted. The output of the following workers is handled similarly during the whole list joining phase, which has linear complexity in the number of workers h. If the lexical grammar of the language is not regular, some additional actions may be required. From our experience this case is quite uncommon in practical programming languages, and does not affect significantly the efficiency of the whole process; nevertheless we will deal with the noticeable exception of Lua.

Algorithm Parallel-lexing summarizes the coordination of the activities of the various sequential lexers.

Thus, the overall complexity of the parallel lexing phase is $\mathcal{O}(n/h) + \mathcal{O}(h)$.

Example 3.2 (*continued*). In the case of our running example, after each worker has completed the scanning of its character stream segment, the last step of the algorithm builds a single list of tokens from those generated for each chunk, eliminating

Algorithm 4: Parallel-lexing(δ, h).

1. Split the input string δ into h substrings of equal length.
2. Scan a *fixed length context of the ending points* of the substrings to check whether any lexeme is broken. If a bounded look-ahead/look-back does not suffice to determine whether the boundaries of a substring are inside a lexeme, move the substring boundary so as to reduce as much as possible the ambiguity on the starting state for its analysis. (The bound for the look-ahead/look-back and the special character to seek in this search are heuristically chosen in a language-dependent way.)
3. Let $\delta_1\delta_2\ldots\delta_h$ be the resulting substrings and, for each $1 \le i \le h$, let s_i be a tuple of states of M, such that from each state s_i^j $(1 \le j \le |s_i|)$ the scanning of substring δ_i can start.
4. Launch h instances of Algorithm 3 (sequential lexing), where, for each $1 \le i \le h$, the parameters are $\gamma = \delta_i$, $s = s_i$. The results of this pass are h tuples $q_1, q_2 \ldots q_h$ of states of M such that, for each $1 \le i \le h, 1 \le j \le |q_i|$, $q_i^j = \Delta^*(s_i^j, \delta_i)$, and h tuples $T_1, T_2 \ldots T_h$ of lists of tokens built along the corresponding computations, where, for each $1 \le i \le h$, $|T_i| = |s_i|$.
5. Build a unique list T of tokens by choosing exactly one list from each tuple T_i $(1 \le i \le h)$ and concatenating them. The selection is performed sequentially, starting from the result of the instance that processed the leftmost substring, as its computation is unambiguous (in fact, $|s_1| = 1$).
6. Return the list T of tokens.

the initial ambiguity on the start state of the lexical analysis. The partial list of the first worker is concatenated with the token list generated by the second worker along the computation starting from inside a comment. The resulting list is then concatenated with the one produced by the last worker along its second computation, updating also the semantic value of the string split across the second and third chunks.

The complete list of tokens returned to be processed by the parsing workers, together with their semantic values provided for clarity, is the same as it would have been produced by a single sequential lexer, i.e., ⟨***n***, `var`⟩ ⟨**+**, +⟩ ⟨***n***, `x`⟩ ⟨**×**, ×⟩ ⟨***n***, `y`⟩ ⟨**+**, +⟩ ⟨***s***, `a multi-line string interrupted here × timesToConcat + `⟩ ⟨**+**, +⟩ ⟨***n***, `a`⟩ ⟨**+**, +⟩ ⟨***n***, `z`⟩.

We now detail how we tailored the above general schema to JSON and Lua.

3.1. The case of JSON

JSON (JavaScript Object Notation) is a data description language, described in the Internet Engineering Task Force document RFC4627 [49], and based on a subset of JavaScript. JSON is widely employed in web applications, where it is progressively superseding XML as a more efficient and compact format for serializing and exchanging structured data.

JSON source code is not intended to be written or read by humans, but rather to be processed by machines. Consequently the JSON grammar lacks some of the typical lexical features of programming languages such as comments and does not mandate a code indentation style as some programming languages do, e.g. Python. Our purpose in selecting JSON as our first case study is to prove the practicality of parallel parsing, providing a realistic benchmark for speedups. In particular, with JSON representing a valid data description language alternative to XML in an ever increasing amount of scenarios, the average size of the JSON files to be processed is already sizeable and increasing. We now detail the steps of the algorithmic schema *Parallel-lexing*, tailoring it to the lexical features of JSON.

Source character stream partitioning The lexemes in the input stream are separated by white-space characters, i.e. spaces, tabulator characters and newlines. Thus, we split the input stream on white-space characters, which occur reasonably frequently. This choice has a drawback since spaces and tabulators are also allowed within strings, so that a string could be split across two chunks. An alternative choice is to break chunks only in correspondence of newline characters, since JSON strings are constrained to be single-line. However, automatically generated JSON code may lack newline characters, making this alternative choice unpractical. Thus, we choose to use white-spaces as chunk separators and accept the consequent limited ambiguity of the lexing. Since the JSON grammar includes only strings as arbitrary length lexemes, the ambiguity for each chunk reduces to two possible initial states only: outside a token or within a single-line string.

Parallel lexical analysis Each worker carries on at most two computations for a chunk, corresponding to the two possible starting states of the analysis. To reduce the level of ambiguity during the lexing action, we exploit the fact that the set of characters composing the non-string lexemes is a proper subset of the ones allowed to appear within a string. Thus if one of the characters which can only appear inside a string is met, the lexing action assuming to be outside a string lexeme can be stopped. To perform the token list recombination, each worker counts the number of string delimiters symbols (double quotes) occurring in the chunk.

List joining phase The last phase of the algorithm, which merges the token lists generated by the lexing workers solving the initial ambiguity, i.e., determining whether or not the chunk started within a string. This is done by checking the parity of the number of quotes read by all the workers preceding the one in need of disambiguation.

3.2. The case of Lua

Lua is a lightweight multi-paradigm programming language widely used as a domain specific language support engine, with a widespread use in video game development. Currently, Lua is the leading scripting language in this application area (as reported in [18] and in [23] which declares Lua the winner of the 2011 Game Developer Magazine Front Line Award). In particular, it has been adopted by prominent industrial game developers such as LucasArts (Grim Fandango, Escape from Monkey Island) and BioWare. Besides the video games programming application area, Lua has been used in various projects (such as Celestia [9]) and since February–March 2013 has been adopted as a template scripting language on Wikipedia [14].

Lua is a full fledged programming language and exhibits some of the syntactic "liberalities" that are fairly typical in various modern programming languages. We stress that the point of employing Lua as benchmark is to show that a richer grammar does not adversely impact on the performance gains obtainable through parsing it in parallel. In particular, the one of its key peculiarities is that the lexical grammar of Lua, unlike most programming languages, is not a regular one. This is due to a non-regular syntax for strings and comments, which requires ad-hoc solutions when tailoring the schema for parallel lexical analysis.

In Lua, strings may be delimited by the so called *long brackets*, in addition to the usual single and double quotes. An opening long bracket is defined by the character pattern `[=`^{n}`[`, with the corresponding closing long bracket being `]=`^{n}`]`. The pair of long brackets must have the same number $n \geq 0$ of = characters to be recognized as a syntactically valid pair. Opening and closing long brackets can be nested, but a valid pair of long brackets cannot contain a closing long bracket of the same type.

Similarly, comments can be single-line or multi-line. Single-line comments start with a double hyphen (`--`) and extend until the end of the line. Multi-line comments have a syntax similar to strings, as they begin with an opening long bracket, preceded by a double hyphen, and end at the corresponding closing long bracket. Strings and comments can be arbitrarily nested, except that they cannot properly contain other comments or strings delimited by brackets with the same number of = symbols, lest an ambiguity on closing long brackets should arise.

Lua's complex syntax for multi-line strings and comments may lead to an intolerable source of ambiguity when different chunks of the input character stream are scanned in parallel. In particular, the arbitrary length marker defined as a delimiter for multi-line strings and comments results in an infinite number of possible delimiters for these constructs. Since there is no way to discern via a fixed look-ahead/look-back analysis which of these delimiters, if any, is enclosing the stream chunk to be analyzed by a lexer, the possible number of starting states and token lists generated along the corresponding computations on the segment could potentially be infinite. A lexing worker, thus, cannot carry on distinct runs for all the possible alternatives for the starting point of the analysis.

To deal with multi-line strings and comments in Lua, we introduce a few constraints on the source programs that can be processed by the schema. To bound the possible degrees of ambiguity in the lexical analysis, we forbid the non-regular syntax for string delimiters: we require that opening and closing long brackets that delimit multi-line strings has $n = 0$ characters = in the patterns `[=`^{n}`[` and `]=`^{n}`]`, i.e., we admit only the `[[` and `]]` as string delimiters. Instead, we do not restrict the syntax to specify comments, so that they can be delimited by long brackets with an arbitrary number of = signs between the brackets, retaining also the non-regular constraint on the number of = characters. This choice is consistent with the common use for multi-line comments, as a container of legacy code, which in turn mandates the need to specify a different comment delimiter from the ones already in use in the enclosed portion of text. Furthermore, to limit the complexity deriving from the possible arbitrary nesting of strings and comments, we assume that multi-line comments are always ended by a newline, so that they cannot end inside a single-line string or comment.

Given these constraints on the lexical grammar of Lua, which we have verified to match widely employed programming practices, we now detail the steps in the algorithm *Parallel-lexing* left open in the general schema.

Source character stream partitioning The partitioning of the source character stream for Lua has been operated by employing the newline characters as effective splitting points. This choice is justified by Lua being a programming language which is expected to be written and read by humans, and thus endowed with proper indentation. Given the hypotheses made on the source form, the possible open ambiguities concern whether the lexing worker is acting on characters belonging to a non-string lexeme, a string, or a comment.

Parallel lexical analysis Each worker, save for the first one, starts with the initial ambiguity of being either at the beginning of a proper lexeme, or within a multi-line comment, or within a multi-line string. The worker carries on two computations: one of them handles simultaneously the first two possibilities, while the other deals with the third one.

The workers starting as if they are at the beginning of a lexeme, or in a multi-line comment, process their chunk along a single computation both matching the lexemes and keeping track of all the comment marker positions it encounters in a list. Whenever they find a closing multi-line comment symbol, they go back to the state where the beginning of a lexeme should be matched, keeping track of the comment-closing point. The position of the delimiters of multi-line comments is used in the last phase, to find which portions of the token stream should be kept and which ones should be discarded. The workers must deal with two possible causes of ambiguity:

I) Closing a multi-line comment within a string. The closing symbol of a multi-line comment may occur within a multi-line string beginning in the same chunk. In this case it is not possible to determine whether the closing symbol ends a

construct started in a previous chunk or belongs to the contents of the current string. To solve this, the worker keeps tokenizing the character stream, until a closing string symbol is met. The worker memorizes the position of the closing string token, allowing for the identification of the string end during recombination.

II) Closing a multi-line string within a comment. A second ambiguity may appear, conversely, when the ending symbol of a multi-line string which began in the same chunk occurs within a single-line comment or a single-line string (delimited by quotes) and the worker cannot ascertain whether the closing delimiter represents the end of a previously interrupted string. The worker has to start two simultaneous computations, collecting the following tokens into two corresponding lists. The two computations can be possibly merged again into a single one when – and if – the ambiguity can be solved.

The workers can also reduce the two initial computations to a single one, eliminating the ambiguity, either when an error occurs in one of the two runs and the corresponding execution is aborted or when the worker reads a symbol with an overloaded semantics (as, e.g., `]]`). In particular, a sequence of two closing square brackets `]]` may represent either the end of a multi-line comment, or the end of a multi-line string or two lexemes used to index a table (e.g., the two closed brackets in `a[b[i]]`). If this sequence occurs, the worker may continue the scanning along the first computation only, since the following characters of the chunk lie necessarily outside of such comments or strings, and may only start a new lexeme or belong to another type of multi-line comment; thus, the two possibilities are handled by a single computation, as stated above.

List joining phase Starting from the first chunk, the list of delimiters produced by each worker is scanned, matching open and closed markers of multi-line strings or comments. The actual starting state of the analysis for each chunk is thus identified by checking the presence of possible open delimiters in previous segments, and the final token list is built by concatenating the portions of token lists generated along the correct computations on the chunks.

Example 3.3. Consider again the grammar of arithmetic expressions in Fig. 7, extended with the possibility of employing the multi-line string and comment definitions of Lua, including the restrictions required to apply our parallel lexing schema. Consider the following source code chunk, which gets assigned to a worker for lexing:

```
x + [[a string]] × [[this string may end here ]=] y +
[[another string]] + z
```

The worker scans the chunk along two computations, C_{token} and C_{string}, corresponding to the hypotheses of starting at the beginning of a token or inside a multi-line string respectively. C_{token} generates deterministically the partial list of tokens $\boldsymbol{n} + \boldsymbol{s}$ $\boldsymbol{\times}$. Subsequently, while it's matching the contents of a multi-line string beginning upon the `[[` delimiter, it detects the closing comment symbol `]=]` and memorizes its position for recombination. Upon the recognition of `]=]`, the worker needs to manage the possibility of the `]=]` symbol being an actual end of a comment which began on a previous chunk. To this end, C_{token} computation diverges on two paths: $C_{token-1}$ resumes the computation from the state corresponding to the beginning of a token and moves on, while $C_{token-2}$ scans the segment following the closing comment symbol as if contained the interior of the string. Upon matching the `]]` delimiter, $C_{token-1}$ has recognized the tokens $\boldsymbol{n} + \boldsymbol{s}$ and annotates the presence of the matched `]]` symbol, while $C_{token-2}$ completes the string recognition and merges back with $C_{token-1}$ yielding the initial computation C_{token}. C_{token} then recognizes the last tokens $+\boldsymbol{n}$.

The second computation C_{string}, on the other hand, scans the segment until the first `]]` (on the first line of the chunk) is encountered, collecting the characters as they were a portion of a string token, and is merged with the ongoing computation $C_{token-1}$ from that point on.

After analyzing the code chunk, information from the previous chunks is employed to determine whether C_{token} or C_{string} has performed the correct computation, and discard the incorrectly lexed tokens accordingly.

4. Adapting grammars to parallel parsing

Almost no grammar in its original user-oriented BNF is immediately ready to be used as an input for a general-purpose deterministic parser generator. For instance, to be suitable for a classical top-down parser such as LL ones, a grammar must avoid left-recursive derivations such as $A \overset{*}{\Rightarrow} A\alpha$, which must be automatically transformed into right-recursive ones. Typically, official technical language definitions do not even comply with the requirements of an LR(k) grammar, as their syntax specification exhibits shift/reduce or reduce/reduce conflicts that require a refactoring of the grammar so that it can be properly handled by parser generation tools. Even worse, most modern programming languages tend to be highly – perhaps too much – liberal towards the users and allow for overloading some symbols, e.g. parentheses, and/or using different symbols as aliases, e.g. ';' and newline. Among those, it is also well known the extreme case of Perl whose parsing has been proved undecidable [32]. Thus, it is common practice and need, before building a compiler front-end for any new language, to carefully redesign its grammar to make it well-suited for the chosen deterministic parsing algorithm.

Such a preliminary work is needed as well to exploit an algorithm based on operator precedence. In our experience, the effort required to transform the official language specification of a technical language in OP form exceeds, but is comparable with, the one necessary to apply a standard LR or LALR-based parsing algorithm such as those used by Bison. We do not believe that such an increased difficulty hides a real impossibility due to the lesser theoretical generative power of OPGs

S → OBJECT
OBJECT → { } | { MEMBERS }
MEMBERS → PAIR | PAIR, MEMBERS
PAIR → STRING : VALUE
VALUE → STRING | number | OBJECT | ARRAY | bool
STRING → " " | " CHARS "
ARRAY → [ ] | [ELEMENTS]
ELEMENTS → VALUE | VALUE, ELEMENTS
CHARS → CHAR | CHAR CHARS
CHAR → char

CHARS → char | char CHARS

Fig. 9. Official JSON grammar. The productions for nonterminals CHARS and CHAR (highlighted in red) are replaced by the one highlighted in green to transform the grammar in OP form. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

w.r.t. LR ones: so far we did not find in real programming languages features preventing a language from being generated by an OPG, such as the ones reported in Section 2.

A drawback of the proposed approach is that the readability of the OP compliant grammar is lower than the one of the LL and LR ones. However, the programmer employing the language targeted by the syntax analyzer will not need to be aware of the employed parsing strategy, and thus of the actual grammar employed by the syntactic analyzer.

In this section we show how we managed the two languages we employed as case studies, i.e., JSON and Lua. Both have been treated with ad-hoc techniques and it turned out that the adaptation to OP form was trivial in the case of JSON, while it required more effort (a few man-days) in the case of Lua; other attempts – e.g., with JavaScript – generated more problems and suggested to resort to a more systematic and fully automated approach. We note that OPGs are already available in the literature for other programming languages, with considerable syntactic richness, such as ALGOL 68 and Prolog [21,16].

4.1. The case of JSON

The official JSON syntactic grammar, reported in Fig. 9, can be trivially transformed into operator precedence form. The only required modification is the one replacing the productions of the CHARS and CHAR nonterminals, to generate a sequence of printable alphanumeric characters, with a plain right-recursive rule.

4.2. The case of Lua

Tackling the transformation of the Lua grammar in OP form has proven more challenging than with JSON. As it happens for other standard technical definitions of classical programming languages, Lua's syntactic grammar (defined in the official reference manual [43] is not expressed in LR form. Concerning its transformation in OP form, the significant issues to be dealt with are the following ones:

1. The language statement terminator `;` is optional and can be replaced by either a white-space and/or a newline character.
2. Function definitions and calls allow one or more newline characters to appear between the function name and the parameter list. For instance:

```
a = b + c
(print or io.write)("done")
```

is to be interpreted as

```
a = b + c(print or io.write)("done")
```

3. Functions are first-class citizens in Lua, thus they can be returned as the result of a call to another function. This feature, in combination with the possibility of employing an in-place defined table as the single parameter passed to a function, allows to write the following code snippet:

```
i = SecOrderFunct{A=3,B=25}
("hazelnut","strawberry")
```

where SecOrderFunct is a call to a second order function. The resulting first order function is subsequently invoked with the `hazelnut` and `strawberry` strings as parameters.

To deal with the above issues, we imposed the following constraints on the sources to be parsed (besides the lexicon restrictions described in Section 3.2), following what we have observed to be the best programming practices in Lua.

1. Multiple statements on the same line must be separated by a `;` character.
2. Comments between statements must be preceded and/or followed by a newline.
3. Multi-line comments are always followed by a newline.
4. We forbid the presence of newline characters between the in-place table declaration of a function parameter, and the parameter lists of the possible lower-order functions returned as the result. This prevents the programmer from using the same code indentation of the example at point 3. Possible ways to reformat are:

```
i = SecOrderFunct{A=3,B=25}("hazelnut","strawberry")
```

and

```
i = SecOrderFunct{A=3,B=25}("hazelnut",
"strawberry")
```

We emphasize that, in the whole set of real world code-bases examined for regression testing purposes, no violation of these constraints has been found. Moreover, the aforementioned constraints stand well within the common best practices in programming, allowing a better readability of the source code.

Provided the aforementioned constraints are respected by the source code, it is possible to solve all the remaining issues to allow OP parsing of Lua by means of a proper token rebelling done during the lexing phase. The transformations applied by our *Parallel-lexing* algorithm are:

Token Disambiguation. The overloading of various tokens is disambiguated by emitting specialized tokens during the lexing phase. For instance, separate tokens are emitted for:

1. `;` used as a separator between statements or as a separator between fields in a table
2. `=` within assignment statements or in the initialization of table fields
3. The round parentheses enclosing a function parameter list and all the others
4. The classical ambiguity between unary and binary minus

It is easy to verify that translations (3) and (4) can be performed by a finite state automaton. Transformations (1) and (2), instead, need a stack to distinguish whether the innermost context where the symbols occur is a statement or a table; such an ad-hoc stack managing is fully integrated with our *Parallel-lexing* algorithm, and does not incur in a significant performance penalty as the lexical grammar of Lua is not regular anyways.

Semicolon Insertion When a newline is employed as a separator between a token that ends a statement, i.e, an element of the set $S = \{$`nil`, `false`, `true`, a number, a string, `...`, `}`, `)`, `]`, `name`, `end`$\}$ and `(`, `name` or any other initial keyword for a statement, (i.e. `break`, `if`, `do`, `while`, `local`, `for`, `function`, `repeat`, `::`, `goto`) we replace the newline with a semicolon: in this way all statements are separated by a semicolon. Note that this substitution does not add a semicolon between the closing parenthesis of a function parameter list and the beginning of the function body. We also insert a semicolon between the elements of these two sets whenever they are separated by a comment.

Thanks to these transformations, the output produced by our enriched *Parallel-lexing* algorithm is fully compatible with the operator precedence constraints. We were thus able to define an OPG which matches the Lua programs with the above restrictions (for completeness the OPG of Lua is reported in Appendix A). One more step is necessary, however, to enable the parallel parsing algorithm described in Section 2: the grammar must be not only an OPG but must also be in Fischer normal form. This last step is performed by the well-known algorithms to eliminate renaming rules and repeated r.h.s. While the previous transformations did not affect significantly the size of the original Lua grammar, in this case the total number of nonterminals and productions increased, respectively, from 38 nonterminals and 144 rules to 49 nonterminals and 8547 rules. However, this increase in the size of the grammar is perfectly tolerable from the point of view of the memory fingerprint of a modern system and does not affect at all neither the run-time efficiency nor the end programmer (since she can fully ignore the new syntax grammar). This last transformation does not significantly alter the shape of the abstract syntax tree (AST) corresponding to a language sentence, save for the compression caused by the elimination of the renaming rules.

As a summary of the actions taken to obtain OPG grammars for both Lua and JSON we can state that performing this step for JSON was trivial, as only one rule modification was required (see Fig. 9). Lua, on the other hand, proved to be in need of intervention not only to obtain an OPG description, but also to have an LR(1) one that could be provided as an input to Bison. The developer effort required to transform the reference Lua grammar in LR(1) form was comparable, if a bit smaller, than the one required to obtain its OPG. We note that the actual LR(1) Lua grammar, ready to be employed in

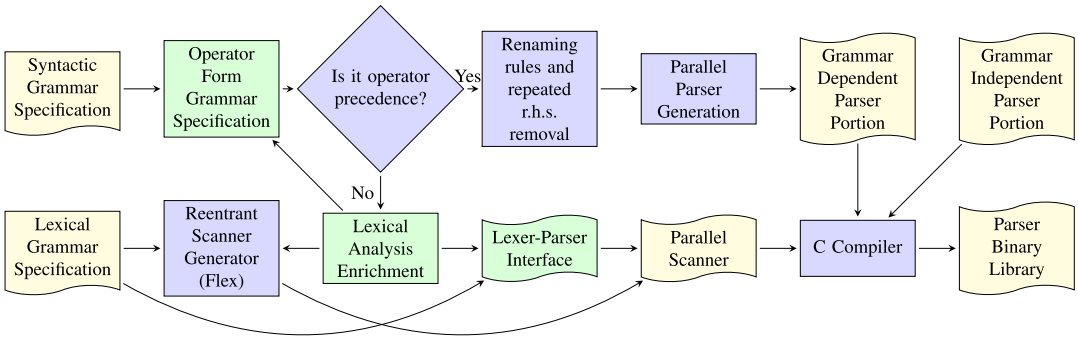


Fig. 10. Typical usage of the PAPAGENO toolchain. The human operator stages are marked in green, while the PAPAGENO automated staged are marked in blue. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Bison, is constituted of 143 productions, with the resulting LALR pilot automaton being 243 states wide, which is in itself quite a sizable one.

5. PAPAGENO toolchain

Here we describe the general architecture of the PArallel PArser GENeratOr (PAPAGENO) toolchain for parallel operator precedence language analysis, in which we implemented the algorithms described in previous sections. PAPAGENO is an open source project available under GNU General Public License and it is written in ANSI/ISO C and Python: the codebase can be downloaded at [42].

The PAPAGENO toolchain provides an automatic parallel parser generator that converts a specification of a syntactic grammar into an implementation of the operator precedence parallel parsing algorithm described in Section 2. The generated parallel parsers can be complemented with parallel scanners, hence obtaining a complete parallel lexer and parser library. For the library we have chosen C as implementation language, because it permits strict control over the computation process and memory management. For the sake of portability, all the C code generated by PAPAGENO employs fixed-size types standardized in the C99 standard; furthermore, it relies exclusively on the standard C runtime and a POSIX-compliant thread library, thus avoiding any architecture-specific optimization. The generated lexers and parsers have been run successfully on x86, x86_64, ARMv5 and ARMv7 based-platforms with no code modifications.

First, we detail the stages of PAPAGENO's workflow for parallel source analysis from the end-user standpoint. Second, we describe the design choices and optimization techniques which proved crucial in exploiting the parallelism exposed by the lexing and parsing algorithms.

5.1. Architecture of PAPAGENO toolchain

The architecture is depicted in Fig. 10. The input of the process contains the specifications of the lexical and syntactic grammars of the target language. If the syntactic grammar of the language is not in operator precedence (OP) form, the tool notifies the inconsistency in the input specification and the user is given proper diagnostics pointing out the rules where precedence conflicts or adjacent nonterminals occur. The user has thus to modify the grammar: a convenient approach to eliminate precedence conflicts consists in enriching the lexical analysis stage with proper transformations, as insertions or renaming of tokens. In Section 4 we described how to adapt into a form suitable for parallel parsing the two case study grammars of JSON and Lua.

Then PAPAGENO automatically eliminates from the OP grammar both the repeated r.h.s. rules and the renaming rules. At last, the C code of a parallel parser is generated.

The parallel parser generator in PAPAGENO has been designed as a replacement for the classical GNU Bison generator and adopts the same basic syntax conventions, allowing an easy porting of the grammar descriptions available in Bison-compliant format. The generated parallel parser is logically split into two parts, as shown in Fig. 10: a language independent support library, and a language dependent parser code portion. This choice was made to allow for easy extensions and possibly further architecture dependent optimizations of the language independent portion, while retaining the automated code generation feature.

The parsing process is invoked by means of a function call, where the developer may specify at runtime the input stream to be analyzed and the number of workers to be employed to perform the analysis. Each worker is mapped to a single POSIX thread, belonging to a thread pool initialized at the beginning of the parsing process.

The developer can choose between two parallel parsing strategies in the generated code. In the first strategy, after a first execution of the parallel parsing algorithm, the recombination of the partial stacks is assigned to a single worker which operates in sequential mode. In the second strategy, instead, the first parallel pass of the parsing algorithm is followed by parallel recombination of the partially parsed substrings along the lines described in the general Algorithm 2: the number

of initial workers is reduced by at least two, and each of the remaining workers has to recombine two partial parsing stacks generated in the first pass (the number of threads can be reduced even further if the part $\mathcal{S}_L$ or $\mathcal{S}_R$ of some partial stacks is empty). This recombination process is iterated until a single thread is left to complete the parsing. The second strategy aims at exploiting the parallelism offered by particularly deep parsing trees. We anticipate that in our case studies the input exhibits a regular or shallow tree structure, causing the difference in parsing time between the two strategies to be small. As stated in Section 2, we feel that quite seldom more than two recombination passes will be advantageous.

The PAPAGENO generated parsers can be naturally combined with either a sequential Flex generated scanner, or a parallel scanner resulting from the implementation of our algorithmic schema described in Section 3. Unlike the generation of a parallel parser, which is fully automatic, the phase of parallel lexer generation currently requires some interaction with the user.

In particular, the programmer is expected to provide the specification of the grammar in the Flex input format for reentrant lexers, write the code managing the input character stream splitting, and the one handling the token list recombinations. The input splitting code performs the actual chunking, possibly employing a fixed-width search window as described in Section 3, and inputs the data into the Flex-generated scanners. The multiple working states of the scanner are mapped onto the multi-state lexer features offered by Flex, requiring from the programmer the definition of the language-specific transitions from one state to the other. At the end of the parallel lexing process, the information on the multiple lexer is exploited by the code written by the programmer to perform the constant-time recombination of the token lists produced by the parallel lexers.

Finally, once the parallel scanner is obtained, as a combination of the output of Flex and the user's lexer-parser interface, it is possible to compile all the sources generated by the toolchain, resulting in a complete binary lexing and parsing library.

5.2. Optimization techniques

The internal architecture of PAPAGENO relies on carefully designed implementation strategies and data structures, which play a fundamental role to obtain high performances of parallel lexers and parsers. In the following, we recall the well-known bottlenecks preventing efficient parallelization and present the solutions adopted in our tool to cope with them.

Two commonplace issues in achieving practical parallelism are 1) the data representation and handling geared towards efficient memory use, and 2) a proper management of the synchronization issues, typically minimizing the use of locks. Thanks to the computationally lightweight parsing algorithm devised for OP grammars, and the minimal requirement for synchronization actions, issue 2) is less important for us, and memory management and memory allocation locality was found to be the crucial issue. Therefore, we start from a discussion of issue 1) and conclude with the synchronization requirements and thread orchestration performed by PAPAGENO generated parsers.

We describe several simple yet effective memory optimizations.

- First, we encoded terminal and nonterminal symbols as word-sized integers, taking care of employing one bit of the encoding to distinguish terminal from non-. By default, the most significant bit is used; however PAPAGENO allows to choose its position at parser generation time to allow room for further information packing. Such information packing does not prevent the definition of large target languages, as the architecture word length in modern devices is at least 32 bit, and 64 bit for most of them. Adopting this technique, we can do without a look-up table to check whether a symbol on the parsing stack is a terminal or non-.
- A second optimization towards improved data locality comes from the observation that the precedence relation between may take one out of four values ($\lessdot, \doteq, \gtrdot, \perp$). Using a bit-packed representation of the precedence matrix, we obtain significant savings for large matrices (which occur in large languages), and, moreover, we manage to fit entirely the matrix in the highest level caches, thus significantly improving the average memory access latency.
- Furthermore, in order to avoid serialization among the workers upon the system calls for dynamic memory allocation, we adopt a memory pooling strategy for each thread, wrapping every call to the `malloc` function. This strategy has also the advantage of reducing memory fragmentation, since the memory allocation is done in large contiguous segments. To evaluate the memory needed for pre-allocation during parsing, we estimate the number of nodes of the parsing tree by computing the average branching factor of the AST as the average length of the r.h.s. of the productions. Then, the parallel parser generator initially pre-allocates half of the guessed size of the AST and augments the memory pool of a worker by one fifth of this quantity, every time the thread requires more memory. A similar memory pooling strategy is employed in the lexing phase, in order to avoid serialization among the lexing threads in need for memory to allocate the token lists.
- One of the most computationally intensive parts of OP parsers is the matching of a production right-hand-side (r.h.s.) against the ones present in the grammar. By representing the r.h.s.'s as a prefix tree (trie), it becomes possible to find the corresponding left-hand side in linear time with respect to the length of the longest r.h.s. of the grammar. Furthermore, to optimize the size and the access time to the trie, we followed the technique described in [24], that represents the structure as an array, storing the pointers to the elements of the trie within the same vector. To take advantage of the trie compression provided by this technique, we assume an upper bound of 2^{16} for the total number of terminal and nonterminal symbols, which clearly does not affect applicability for any common language. The vectorized trie is fully precomputed by PAPAGENO, and is included in the generated parser as a constant vector.

For the synchronization and locking issues in OP-based parallel parsing, we used rather straightforward techniques.

- Since each parallel worker performs the parsing action on separate tokenized input chunks, it is completely independent from the other workers, and there is no need for any synchronization or communication between them. This in turn allows the proposed strategy to scale easily even in the cases where the inter-worker communication has a high cost, e.g. whenever the input is so large that they have to be run on different hosts.
- Similarly, all the lexers act independently on the input, without need of communication or synchronization while performing the lexing actions.
- The requirement for enforced synchronizations is only present in the following two cases: i) a single barrier-type synchronization point is required between the end of the lexing phase, and the beginning of the parsing one whenever the lexical grammar requires a constant-time chunk combination action to be performed by the lexer; ii) synchronizations are required to enforce data consistency if the user desires to perform multiple parallel parsing recombination passes, instead of a single one.
- While the first barrier synchronization cannot be subject to optimizations, the synchronizations between multiple parallel parsing recombination passes can be fruitfully organized hierarchically. In particular, a parsing worker from the n-th pass will only need to wait for the completion of the $n-1$ pass workers producing its own input, effectively avoiding the need of a global barrier synchronization between passes. Such a strategy allows to effectively exploit the advantages of multiple parallel passes whenever the parse tree is very high.

6. Experimental results

In this section, we present and discuss the experimental results of our parallel lexing and parsing system on both JSON and Lua languages.

The benchmarks We chose real world JSON and Lua inputs of various sizes, on which we performed the parsing and the construction of the abstract syntax tree (AST) in memory. This is the only semantic action associated to the parsing process. The rational is to evaluate the computational load of the parsing process, regardless of any subsequent use of the parsed data. Since in typical compilers the semantic actions are more computationally demanding than AST construction, it follows that even greater performance benefits can be achieved if they can be parallelized. In other words, our results evidence the speed-up that is achieved by parallelizing syntax analysis and nothing else.

For JSON, the set of inputs includes a shopping list from an online shop (30 kiB), the configuration file of AdBlocker, a common browser plugin (80 kiB), the Gospel of John (150 kiB), a statistic data-bank (1.6 MiB) on food consumption (source Italian Institute of Statistics), a file containing statistics on n-grams present in English in Google Books (10 MiB), and the index of all the documents available on the UK Comprehensive Knowledge Archive Network (75 MiB).

For Lua, benchmarks were derived, instead, from the codebase of Lucasarts's Grim Fandango, which is available together with the game. This code-base size amounts to 2.5 MiB, and, to obtain benchmarks of different sizes, a suitable number of compilation units have been concatenated together. To explore the scalability of the parallel parsing approach we tested the PAPAGENO generated analyzer with files ranging from 7 kiB to 35 MiB, with the ones larger than the whole code-base being generated by concatenating all the files from the code-base more than once.

Hardware platforms To evaluate the practical speedups obtained, we used two platforms:

1. A quad-Opteron 8378 host, thus amounting to 16 physical cores (4 cores per socket): the Opteron 8378 CPUs are endowed with independent, per CPU, L1 and L2 caches, and a chip-wide shared L3 cache. The host runs Ubuntu Linux 14.04 (`x86_64` architecture) server and is endowed with enough RAM to contain the whole AST materialized during the parsing process and token list. The purpose of the evaluation on this platform is to highlight the scalability of our approach, even in the context of a multi-socket system with a non-uniform memory access.
2. An Odroid-XU Lite board, endowed with a Exynos 5 Octa SoC, which is driven by four Cortex-A15 and four Cortex-A7 CPUs, in big.LITTLE configuration, clocked at 1.4 GHz, and 2 GB of DDR3 DRAM. The platform runs a Debian 7.6 Linux (`armv7l` architecture), and the main choice is to use the four Cortex-A15 CPUs, as the architectural constraint do not allow to employ all the 8 cores simultaneously. The benchmarks run on this platform are representative of the actual performance benefits obtainable on a high-end embedded system, such as the ones which are increasingly more common in mobile phones and tablets. Such platforms are typically characterized by a uniform memory access, and limited main memory resources with respect to desktop machines.

All the executable binaries have been produced through `gcc` 4.9.1, employing standard release grade optimizations to obtain an efficient binary (`-O3 -march=native` optimization options). All the timing results presented have been collected employing Linux real-time clock primitives, and are the average of 50 runs to reduce measurement noise.

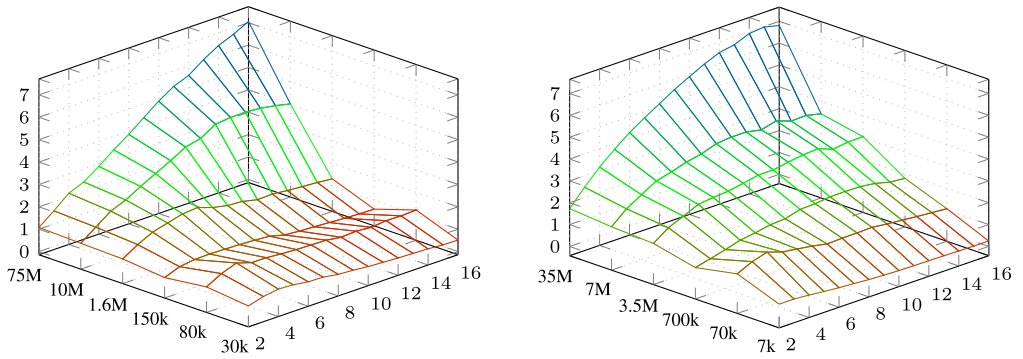
Purely sequential execution Tables 1 and 2 report the absolute processing times respectively obtained for JSON and Lua, using a purely sequential PAPAGENO lexer-parser pair: they establish a practical baseline for comparison. Notice that the

Table 1
Total text analysis times of the JSON test-bench files, for both the server and mobile platform.

Input size	Elapsed time [ms]					
	Server			Mobile		
	Lexing	Parsing	Total	Lexing	Parsing	Total
2.7 kiB	0.6	1.8	2.4	0.8	1.8	2.6
30 kiB	2.7	5.1	7.8	2.8	5.6	8.4
80 kiB	7.7	15.4	23.1	8.3	18.0	26.3
150 kiB	15.0	37.4	52.4	24.8	69.0	93.8
1.6 MiB	98.4	255.0	353.4	153.1	431.1	584.2
10 MiB	588.1	1584.7	2172.8	1033.6	2665.3	3698.9
75 MiB	3462.6	8892.2	12354.8	–	–	–

Table 2
Total text analysis times of the Lua test-bench files, for both the server and mobile platform.

Input size	Elapsed time [ms]					
	Server			Mobile		
	Lexing	Parsing	Total	Lexing	Parsing	Total
7 kiB	0.9	2.2	3.1	1.0	2.3	3.3
70 kiB	5.8	9.7	15.5	7.2	12.0	19.2
700 kiB	24.6	41.7	66.3	63.5	98.2	161.7
3.5 MiB	105.8	161.0	266.8	212.8	314.5	527.3
7 MiB	203.4	313.4	516.8	424.4	602.4	1026.8
35 MiB	998.9	1559.3	2558.2	–	–	–



(a) JSON: Lexing phase

(b) Lua: Lexing phase

Fig. 11. Speedups achieved on the lexing for JSON and Lua, taken against a sequential lexer, depicted as a function of the number of workers and the input size expressed in bytes.

absolute times for the larger files are quite important, especially on the mobile platform. Moreover, the latter is not endowed with enough memory to materialize the whole AST for the largest test cases; as a consequence the two largest benchmarks cannot be run on it. A point worth noting is that, both in the case of JSON, and in that of Lua, the time spent in the lexical analysis of the input is non-negligible: more specifically, it is around 30% for JSON and 40% for Lua. This result substantiates our claim that, for OP-based parsing, the lexical analysis accounts for a non-trivial amount of the text processing time.

Parallel execution Fig. 11 shows the speedup obtained by the parallel lexer versus a sequential run of a Flex generated lexer, while Fig. 12 reports the speedups of the parsing phase of the computation, computed against a sequential run of a PAPAGENO generated parser.

Consider the results for the JSON lexing phase reported in Fig. 11(a): for all file sizes $\geq$ 80 kiB, parallel lexing achieves a significant speedup over the plain sequential Flex generated lexer. This is more evident when the file size allows all the workers to perform a significant amount of computation.

A point worth discussing is the relatively low speedup achieved when only a few workers are employed: in this case, the lightweight computation required by JSON lexing, together with the possible thread migration from one CPU to the other in the NUMA machine employed, negatively affect the performances. More in detail, a thread migration from one CPU to another implies a significant drop in the effectiveness of the caches, as the computation is moved to a processor where the working-set is not pre-heated in cache. By contrast, a higher load on all the available CPUs will prevent the scheduler from moving the tasks in an attempt to equalize the load. Although this issue can be solved pinning the threads to a specific CPU

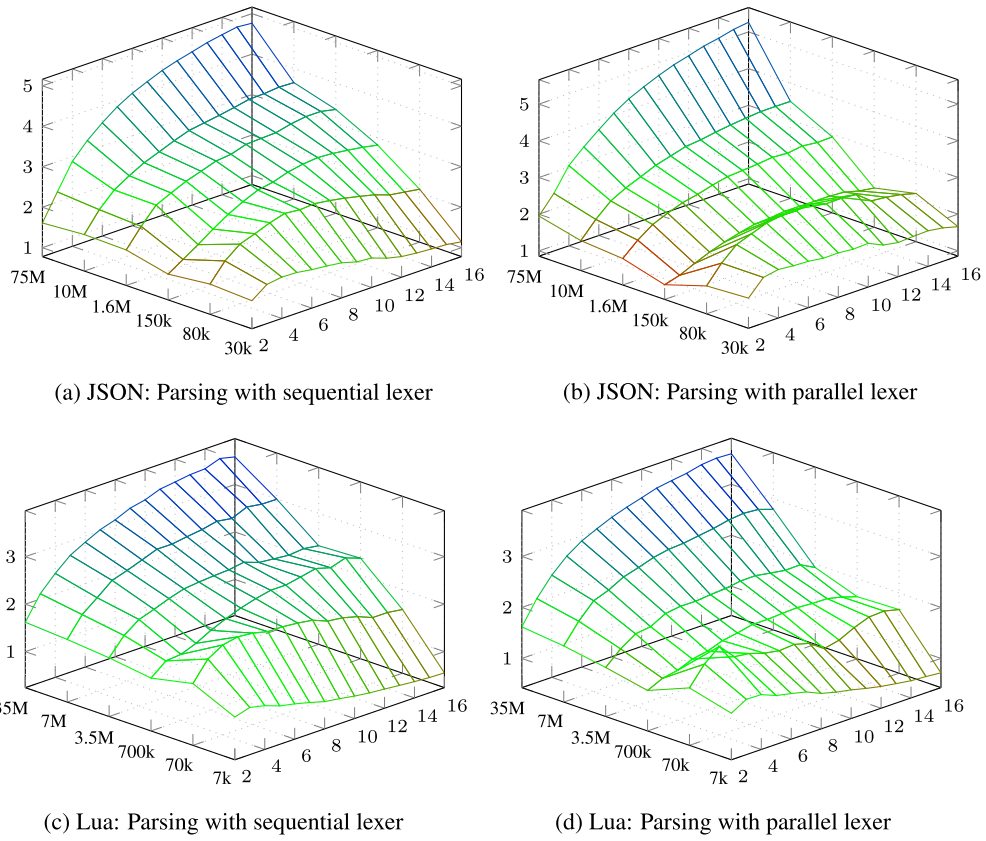


Fig. 12. Comparison of speedups achieved on the lexing and parsing phase for JSON and Lua employing either a sequential or a parallel lexer, taken against a sequential PAPAGENO parser. The speedups are represented as a function of the number of employed workers, and the input size in bytes.

through processor affinity settings, we chose not to perform the measurements with such a technique as it may yield overly optimistic results with respect to running environment where CPU pinning is forbidden (e.g. large data-centers where the computation is taking place inside virtual machines). To finish, we note that the maximum achieved speedup is 7× in the case of a 75 MiB JSON file, cutting down its lexing time from 12 sec. to less than two.

The lexing phase results for Lua (Fig. 11(b)) confirm the speedups achievable through the parallelization of the lexing stage, even in the case of a lexical grammar much more complex than JSON one. We know Lua parallel lexer needs to perform a non-trivial recombination at the end of lexing, and the results show that the recombination phase does not impact adversely performances. The reduced performance gained on the 150 kiB input file is to be ascribed to low level cache contention, as the file exceeds by a small amount the least level cache size for the involved CPUs.

The JSON parsing phase (Figs. 12(a) and 12(b)) also benefits from significant speedups (up to 5.3×) for large files, and show how a parallel parsing approach is advantageous even in the case of small files. In particular, Figs. 12(a) and 12(b) report the speedups achieved during the parsing phase only, for an implementation with a Flex-generated sequential lexer (Fig. 12(a)), and a parallel lexer (Fig. 12(b)). Comparing two situations, we get an interesting insight on the use of a parallel lexer. As it can be seen, combining the parallel parser with the parallel lexer, has a positive synergistic effect, even in the case of small files, yielding effective speedups already for the 30 kiB file. This effect is to be ascribed to the L2 and L1 cache pre-heating effect caused by having the text lexically analyzed by different independent workers. In fact, such an approach is more likely to be fetching the data which will be parsed by a worker into the dedicated L1 and L2 caches of the corresponding core, effectively reducing the memory pressure for the parsing action, and thus increasing the performances. As a further confirmation of this fact, we note that the performance boost does not take place in the case of the parsing of the 150 kiB file, which is a good cache fit already with a sequential lexing process.

Concerning the Lua parsing phase, Figs. 12(c) and 12(d) show how this can be effectively parallelized, notwithstanding the much richer structure of the language. On the other hand, the synergistic effect between parallel lexing and parsing in Lua is less evident than in JSON; we ascribe this fact to the higher memory requirements for the Lua parsing process, which in turn add extra pressure on the caches, preventing the pre-heating from having a significant impact.

Comparison with Flex and Bison Since the lexers and parsers produced by Flex and Bison are the current state-of-the-art for tool generated language processors, it is interesting to compare in Fig. 13 the performance of the parallel lexer/parser

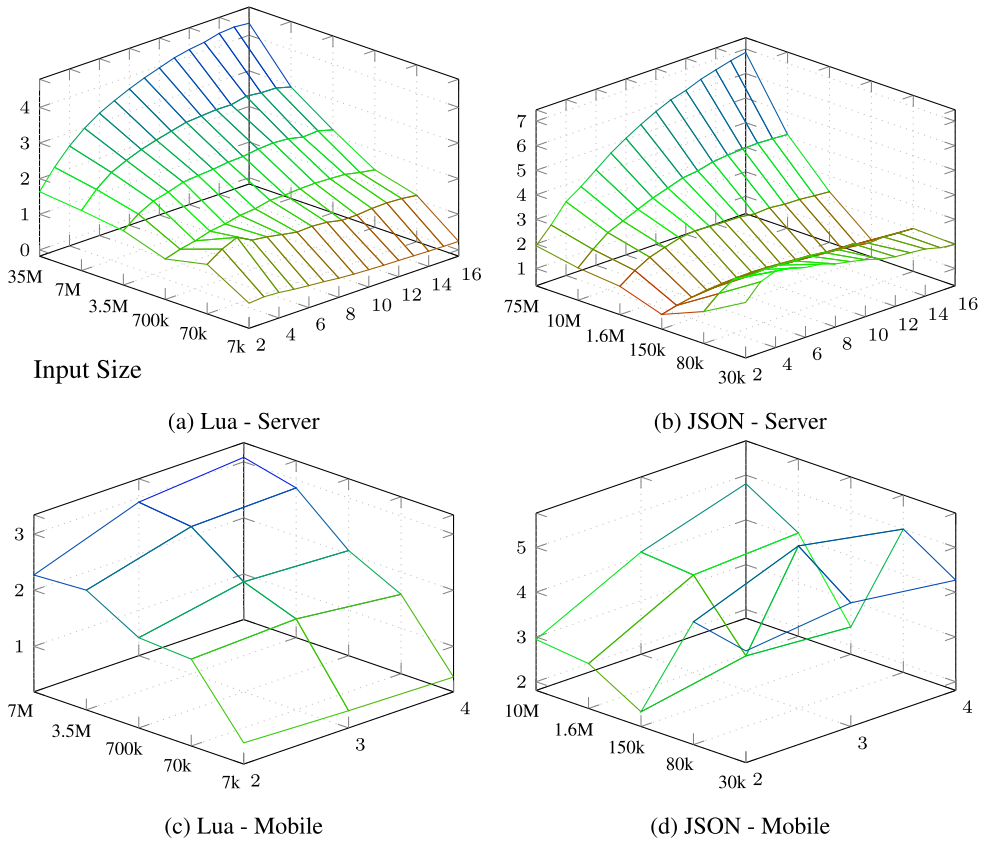


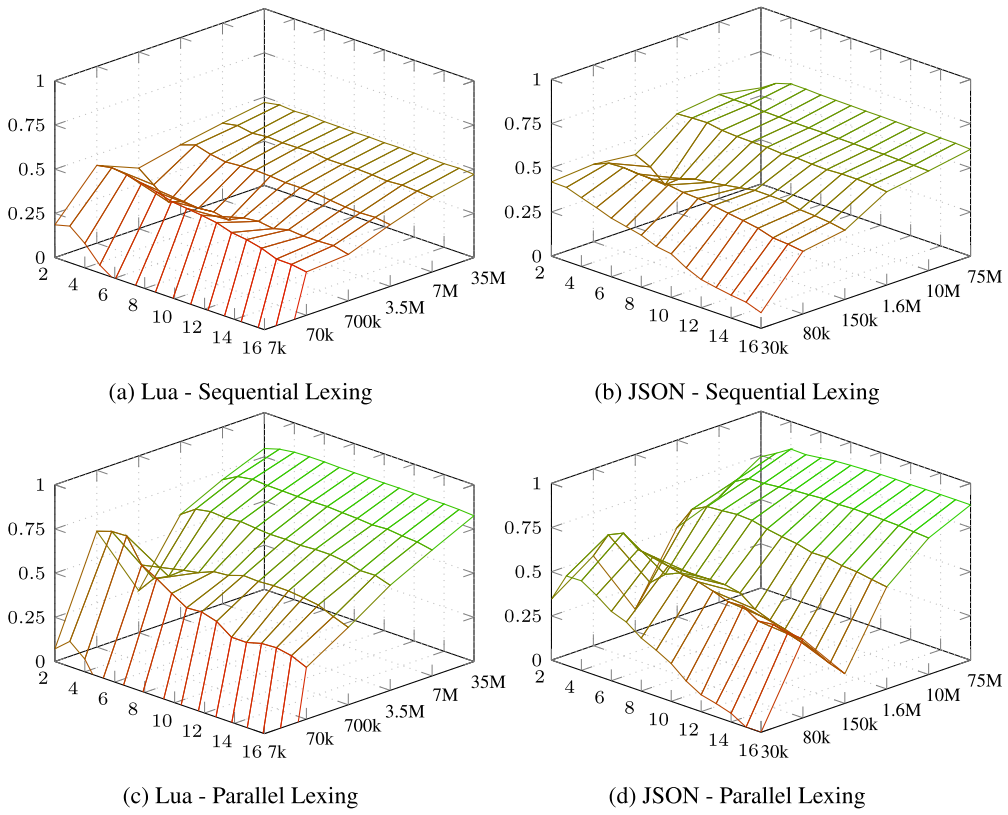
Fig. 13. Speedups obtained against a sequential Flex–Bison generated text analyzer on both the server and mobile platforms. Darker lines indicate smaller input files.

library generated with PAPAGENO, against a text analysis library produced by Flex and Bison, selecting the LALR(1) parser generation algorithm. The results evidence, in both the server and mobile platform, a significant speedup with respect to the state-of-the-art of tool generated parsers, for all but the smallest test-bench files; and a good scalability of the approach. We discuss two aspects in particular:

- The ability to exploit per-die caches in JSON parsing leads to significant benefits for small texts, whenever the parsing action is contained within a single multicore CPU on the server platform, i.e., up to 4 simultaneous workers.
- The simplicity of the OP parsing algorithm represents an effective advantage on RISC architectures, such as the one of the ARM platform, where it is able to obtain a speedup of up to 5.5×, as a combination of the parallel processing technique on multiple cores and the lesser computational requirements with respect to a classic LALR(1) parser.

Amount of parallel code We maintain that performing the lexing actions in parallel gives a substantial advantage in terms of the actual amount of code which is executed in parallel. To substantiate this claim, we computed the sequentially executed code portion e from our obtained speedups according to the Karp–Flatt metric [31] as $\frac{1/s-1/p}{1-1/p}$, with s the achieved speedup, and p the number of involved processors. We chose to employ the Karp–Flatt metric as it is designed to provide a concrete counterpart to Amdahl's law, as the latter only states the maximum achievable speedup, while assuming no interference by the operating system and runtime on which the processes are run. Fig. 14 depicts the portion of code which is executed in parallel, obtained as $1 - e$, for both JSON and Lua parsing on the server platform, with Figs. 14(a) and 14(b) reporting the serial-lexer implementation while Figs. 14(c) and 14(d) report the parallel-lexer one. It is evident that, eliminating the parallelization of the lexing action, has the effect to significantly reduce the parallel code portion, namely around 20%–25% for JSON and 40% for Lua for all the input sizes/number of workers combinations, where the operating system overhead is not dominating the computation. Moreover, the parallel portion analysis additionally shows how an input size increase determines a larger portion of the code being executed in parallel, in turn implying that the scaling of our approach on the input size does not introduce significant system overhead, thus allowing the user to fully reap its benefits.

Finally, it can be noticed that, for sufficiently large inputs, the overhead of spawning more workers is negligible: this is a consequence of the absence of communication between the workers, which only need to synchronize at the end of their computation. This in turn translates into a high efficiency in scaling to high amounts of parallel threads.



(a) Lua - Sequential Lexing

(b) JSON - Sequential Lexing

(c) Lua - Parallel Lexing

(d) JSON - Parallel Lexing

Fig. 14. Parallel code portion in both the JSON and Lua lexing/parsing process obtained as the complement to the sequential code portion obtained via Karp–Flatt metric, plotted as a function of the number of workers, and the size of the input expressed in bytes.

7. Related work

The literature on parallel parsing (and lexing) is vast and extends over half a century. The valuable survey and bibliography [3] lists some two hundred publications in the period 1970–1994, and research has continued since, though perhaps less intensively. It is worth contrasting this quantitative aspect with the paucity of existing realizations and, even more so, of tools for producing parallel parsers. We omit, as less relevant to our objectives, some categories: the work on grammar types not belonging to the context-free family, the studies based on connectionistic or neural models of parallel computations, and the large amount of work on natural language processing. We are left, roughly speaking, with the following categories:

1. *Theoretical analysis* of algorithmic complexity of parallel context-free language recognition and parsing, in the setting of abstract models of parallel computation, such as P-RAM.
2. *Parallel-parser design and performance analysis* for specific programming/web languages, sometimes combined with experimentation, or, more often, simply with demonstration, on real parallel machines.

Category 1. is mainly concerned with the asymptotic complexity of recognition/parsing algorithms on abstract parallel machines. The algorithms proposed for unrestricted CF grammars require an unrealistic number of processors: for instance Rytter's [44] recognizer has asymptotic worst-case time complexity $\mathcal{O}(\log n)$, with n the input length, and requires $\mathcal{O}(n^6)$ processors; the numbers of processors grows to $\mathcal{O}(n^8)$ if parsing, i.e., syntax-tree construction, is required. Several researchers have shown that such complexity bounds can be lowered, by restricting the language class, sometimes so much that it loses practical interest. We mention some cases, from the simplest to the more general ones, for the recognition problem. For the *input-driven* (also known as *visibly pushdown*) languages, the time complexity is $\mathcal{O}(\log n)$ and "only" $\mathcal{O}(n/\log n)$ processors are used [26]. The *deterministic CF* languages are recognized in time $\mathcal{O}(\log n)$ on a P-RAM machine using $\mathcal{O}(n^3)$ processors [33]. A series of papers (e.g., [10]) have gradually refined the complexity bounds for the case of unambiguous CF languages.

Such idealized results are, of course, not really comparable with experimental findings, as already asserted by [3], yet they offer some interesting indications. In particular, all the subfamilies of deterministic CF languages for which the theoretical complexity analysis reports a close to linear use of processors, are included in the family of OP languages we use.

Such abstract complexity studies had little or no impact on practical developments, for several reasons. First, it is known that the abstract parallel machines, such as P-RAM, poorly represent the features of real computers, which are responsible for performance improvements or losses. Second, asymptotic algorithmic time complexities disregard constant factors and mainly focus on worst cases, with the consequence that they are poorly correlated with experimental rankings of different algorithms. Last, most theoretical papers do not address the whole parsing task but just string recognition. In the following years 1995–2013 the interest for research on the abstract complexity of parsing algorithms has diminished, with research taking more practical directions.

The classical *tabular* recognition algorithms (CKY, Earley) for unrestricted CF languages have attracted much attention, and a number of papers address their parallelization. It is known that such parsers use a table of configurations instead of a pushdown stack, and that their time complexity is related to the one for matrix multiplication, for which parallel algorithms have been developed in many settings. Parallel algorithms derived from CKY or from Earley sequential parsers (*s-parsers* for brevity) may be pertinent to natural language processing, but have little promise for programming/data description languages. As tabular s-parsers are significantly slower than LR or OPG s-parsers (up to some orders of magnitude), it is extremely unlikely that the parallelization of such a heavy computational load would result in an implementation faster than a deterministic parallel parser. Moreover, as we are not aware of existing tabular parallel-parser generation tools, confirmation by experiment is not possible at present.

The comparison with previous work in category 2. is more relevant and reveals the precursors of several ideas we use in our generator. We only report on work dealing, as our own, with deterministic CF languages.

Bottom-up parsing Some early influential efforts, in particular [19] (described in [11]) and [41], introduced data-parallelism for LR parsers, according to the following scheme: a number of LR s-parsers are run on different text segments. Clearly, each s-parser (except the leftmost one) does not know in which parser state to start, and the algorithm must spawn as many deterministic LR s-parsers as the potential states for the given grammar; each parser works on a private stack. When a parser terminates, either because it has completed the syntax tree of the text segment or because the lack of information on the neighboring segments blocks further processing, the stack is merged with the neighboring left or right stack, and the s-parser process terminates; similar ideas occur in other papers too. However, the idea of activating multiple deterministic bottom-up s-parsers is often counterproductive: the processes, associated to the numerous parser states of a typical LR grammar, proliferate and reduce or nullify the speedup over sequential parsing.

Two ways of reducing process proliferation have been proposed: by *controlling the points of segmentation*, and by *restricting the family of languages* considered. An example of the first is in [45], so explained: "The given input string ... is divided into approximately q equal parts. The i-th processor starting at token ... scans to the right for the next synchronizing token (e.g., *semicolon*, *end*, etc.) and initiates parsing from the next token". If synchronizing tokens are cleverly chosen, the number of unsuccessful parsing attempts is reduced, but there are drawbacks to this approach: the parser is not just driven by the language grammar, but needs other language-specific indications, to be provided by the parser designer; thus, [45] chooses the synchronizing tokens for a Pascal-like language. Furthermore, to implement this technique, the lexer too must be customized, to recognize the synchronizing tokens.

Similar language-dependent text segmentation policies have been later adopted by other projects, notably by several developments for XML parsers; such projects have the important practical goal to speed-up web page browsing, and investigate the special complexities associated to parallel HTML parsing. Although they do not qualify as general purpose parsers, their practical importance deserves some words. The recent [51] paper surveys previous related research, and describes an efficient parallel parser, *Hpar*, for web pages encoded in HTML5. HTML5 has a poorly formalized BNF grammar and tolerates many syntax errors. A HTML5 source file may include a script (in JavaScript), which in turn can modify the source file; this feature would require costly synchronization between lexing and parsing threads, which make a pipelining scheme inconvenient. *Hpar* splits the source file into *units* of comparable length, taking care not to cut an XML tag. Each unit is parsed by an independent thread, producing a partial DOM tree; at last, the DOM trees are merged. A complication comes from the impossibility to know whether a *unit*, obtained by splitting, is part of a script, a DATA section, or a comment. The parser uses heuristics to speculates that the unit is, say, part of a DATA section, and rolls-back if the speculation fails. (In Section 3 we have described a similar approach to parallel lexing.) More speculation is needed for another reason: when a unit parser finds a closing tag, say $<$`/Table`$>$, it does not know if the corresponding opening tag occurred before in the preceding unit, or if it was missing by error. The best speedup achieved ($2.5\times$ using five threads) does not scale for the current web page sizes.

Returning to parsers purely driven by the grammar, in view of the popularity of (sequential) LR parser generation tools like Bison, the fact that no parallel-parser generators exist is perhaps an argument against the feasibility of efficient parallel parsers for LR grammars. This opinion is strengthened by the fact that several authors have developed parallel parsers for language families smaller than the deterministic CF one, but it would be too long to cover all of them, and one example suffices. The grammars that are LR *and* RL (right to left) deterministic enjoy some (not quantified) reduction in the number of initial parser states to be considered by each unit parser. Such grammars are symmetrical with respect to scanning direction: rightwards/leftwards processing, respectively, uses look-ahead/look-back into the text to choose legal moves. By combining the two types of move into a bidirectional algorithm, dead-ended choices are detected at an earlier time. We observe that Floyd's OPGs too have the property of reversibility with respect to the parsing direction and benefit from it for making local parsing decisions, which are unique and guaranteed to succeed if the input text is grammatically legal.

Indeed, thanks to the local parsability property, OP languages do not incur in the penalties that affect LR parsers; the latter, as said, need to activate multiple computations for each deterministic unit parser, since many starting states are possible. For OP parsers, in fact, all the actions can be deterministically taken by inspecting a bounded context (two lexemes) around the current position, and do not depend on information coming from the neighboring unit parsers: thus, each text unit can be processed by an OP parser instance along a single computation, without incurring on the risk of backtrack.

To complete the topic of restricted CF language families suitable for local parsability, we mention two papers not surpassing the preliminary proposal stage. A list of requirements for local parsability is in [36]. The work we consider to be closest to our choice of OPGs is [40], that uses bounded-context grammars, a grammar model [22] generalizing OPGs, which however has been rarely considered for s-parsing.

Top-down parsing Less effort has been spent on top-down deterministic LL parsers, possibly because, at first glance, their being goal-oriented makes them less suitable for parsing arbitrarily segmented text. The article [50] surveys the state-of-the art for such parsers and reports in detail a parallel (non-experimented) algorithm that works for a subclass of LL grammars, named LLP. Imagine that the text is segmented into substrings and on each segment a classical LL(k) s-parser is applied. Similar to the LR case, each s-parser does not know the result (i.e., a stack representing the prefix of a leftmost derivation) for parsing the substring to its left: therefore each s-parser has to spawn as many s-parsers working on the same segment, as there are possible initial stacks, too many to be practical. Therefore it is proposed to limit the number of possible initial stacks by imposing a restrictive condition on LL(k) grammars. The subfamily thus obtained is named $LLP(q, k)$ and is based on the idea of inspecting a look-back of length q tokens as well as the classical look-ahead of k tokens. Although not compared in the paper, $LLP(q, k)$ grammars look quite similar to the already mentioned bounded-context grammars. This and earlier studies on parallel LL parsers may be theoretically interesting but do not offer any hint on practical usability and performances.

Parallel lexing The problem of breaking up a long string into lexemes is a classical one for data parallel algorithms, well described in [29]. They assume, as such studies invariably do, that each lexeme class is a regular language, therefore the sequential lexer is a deterministic finite automaton (DFA) that makes a state transition reading a character. For a string x, the chain of state transitions define a *lexing function* that maps a state p to another state q; moreover the function for the string $x \cdot y$ obtained by concatenation is obtained by function composition. The data-parallel algorithm is conceptually similar to the one for computing all partial sums of a sequence of numbers, also known in computer arithmetic as the parallel *sum prefix* algorithm. In essence, the source text is split into pieces, and the DFA transition function is applied to each piece, taking each DFA state as a possible starting state. Then the functions obtained for neighboring pieces are composed and the cases of mismatch are discarded. Such processing can be formulated by means of associative matrix operations. This parallel algorithm is reported to be optimal from a purely theoretical viewpoint, but early simulation on fine-grained architectures with very many processing units is not conclusive. More recently, various experiments of similar algorithms on GPGPU and on multi-core architectures have been reported. A criticism is that such algorithms are very speculative, performing a significant amount of computation which may be later on discarded, thus yielding fairly poor energy efficiency. Some authors have considered the *regular expression matching* problem, instead of the *lexing* problem, and, although regular expressions and DFA models are equivalent, the parameters that dominate the experiments may widely differ in the two cases. An example suffices: [46] presents a notable new version of the mentioned [29] approach. They claim that for certain practical regular expressions that are used in network intrusion/detection systems, the size of the parallel lexer remains manageable and not bigger than the square of the minimal DFA. Then, they are able to construct the parallel scanner on-the-fly, i.e. delaying as much as possible the construction of the states. Clearly, algorithm [46] is not intended as a lexer to be invoked by a parallel parser, but as a self-standing processor for matching regular expressions – yet partially so, since it does not address the central issue of ambiguous regular expression parsing, which fortunately does not concern our intended applications.

Recently, [47] has experimented on the Cell Processor a parallel version of the Aho–Korasick string matching algorithm. This work was motivated by the good performance of that algorithm on multi-core machines for string search against large dictionaries. But a downside of that approach is that it apparently assumes that the input file can be unambiguously divided into text segments; therefore it does not apply to the case of general programming- or data-representation languages, since, for such languages, scanning cannot avoid an initial degree of nondeterminism caused by the absence of a separator between tokens (as a newline) that could be identified by inspecting a bounded portion of the segments.

Compared with the mentioned studies, our approach to parallel lexing in Section 3 addresses further critical issues. First, the approach is suitable for more general lexical grammars that involve pushdown stacks and cannot be recognized by a DFA (as the lexical grammar of Lua). Second, our approach integrates some pre-processing steps that enhance the performance of the following parsing stage. Furthermore, we address a complexity that made previous approaches such as [29,41] unpractical: splitting the input file into segments may cause ambiguity, in the sense that the lexing function associated to a segment may return multiple values (states), depending on the assumed input state. To compute such function, several workers are needed, but in our design their number does not equal the number of states of the automaton, but is limited to two or three, and does not critically affects performance, as attested by the experimental results achieved by PAPAGENO.

8. Conclusion and future research

Research on OPGs has been abandoned a long time ago due to the advent of other grammar families well suited for deterministic parsing and with greater generative power. OPGs, however, exhibit, among other nice properties not pursued in this paper, the distinguishing feature of local parsability. Among them, we recall their algebraic properties: in fact their languages (to be named OPLs) are closed w.r.t. all basic language operations, including the Boolean ones. This feature is unusual in language families larger than regular languages (to the best of our knowledge OPLs are the largest family of deterministic languages that enjoys all such closures), and is the key to the application of model-checking algorithms even outside the traditional scope of finite-state machines: for instance OPLs strictly include the *visibly pushdown* languages [13], which have been successfully proposed with the motivation of applying model-checking techniques to infinite state machines [5], and also as an abstract model of mark-up (parenthesized) languages.

In this paper we exploited the local parsability property of OPGs to produce automatically a parallel parser which is complemented by a parallel lexer. This allowed us to bring the possibility to exploit effectively non-speculative parallelism in parsing, which was up to now a considerable exception to the present tendency to exploit parallelism in practically any application.

Our prototype tool already offers considerable speed-up compared with state of the art sequential parsers such as Bison and can be used, with variable effort, to produce parallel parsers for most practical programming languages. It is available for free downloading (under GNU GPL license) for further application, experimentation, and possible extensions.

In the following, we mention various research directions to fully exploit the local parsability property; we first give a couple of hints to make our tool even more efficient and more widely applicable to a larger set of languages. Moreover, we outline other research directions aimed at exploiting local parsability, both of theoretical nature and with the purpose of applying it in other application fields than “pure” compilation.

8.1. Optimizing performances and effectiveness of the present tool

The general parallel parsing algorithmic schema defined in Section 2 is conceived in such a way that it can be iterated through several passes until the obtained result is short enough to make it convenient to apply a final sequential parsing. As a matter of fact, our present tool obtained quite satisfactory performances even with one only parallel pass immediately followed by a final sequential one. As we already noticed, however, there could be cases where splitting the input in chunks of equal length does not correspond to the overall structure of the source, e.g., if the chunks consist of the frontiers of two adjacent and large sub-trees. In such cases the further passes described in the general Algorithm 2 may produce substantial benefits. Some further experiments are planned to validate our conjecture that in most cases two passes are all that is needed to exploit at best parallel parsing.

Deterministic parsing based on OPGs has been applied to many programming languages in the past [21,16] and we added a couple of more recent ones used in our benchmark; purposely, JSON and Lua have been selected with sharply different features and, not surprisingly, adapting Lua to our OP-based approach was a considerably tougher job than the former one. Some early investigation, e.g., on JavaScript, seems to hint at even more difficulties with other modern languages: in general, in fact, as pointed out in Section 4, many modern languages offer (too?) much freedom to programmers which generates overloading of various symbols and hampers deterministic analysis, what often results into precedence conflicts. We are confident that such difficulties are not insurmountable but we are aware that more work is needed to make our approach applicable in a generalized way to most practical languages. As a first step we plan to widen the heuristic techniques developed in the case of Lua so as to apply them to other widely used languages. To be useful in practice, such techniques must be supported by automatic tools to obtain an OPG grammar equivalent to the original one and/or to produce an intermediate text – after parallel lexical analysis – that can be supplied to an OP parallel parser. Moreover, we are also looking into obtaining further validation of the advantages of parsing in parallel other data description languages, such as, for instance, XML based ones. In the longer term we also wish to investigate the theoretical aspects of this issue as hinted in the following subsection.

On the other hand we notice that recent recommended best practices tend to limit the excessive freedom allowed by the grammars of modern languages. For instance [15] proposes a series of disciplined ways to write cleaner and more understandable JavaScript programs; we verified that, if applied rigorously, they produce a subset language almost ready for OP-parsing with no need for a heavy preprocessing. Another case of well disciplined, and easily analyzable source is provided by modern compiler back-ends, which target a restriction of the JavaScript language as their assembly output such as Emscripten [4] and asm.js [48]. In both cases, the language to be analyzed has all the features to be efficiently parsed in parallel: it is characterized by very large compilation units, and has a grammar free from unusual quirks.

8.2. Related research directions

The local parsability property proposes several intriguing questions which further widen the spectrum of potential applications both within and beyond parallelism.

From a theoretical point of view, OPLs are just an example of locally parsable languages with bound 1 but many more could be worth investigating: as mentioned in Section 7, Floyd himself proposed a generalization of the family by adopting

larger bounds of the context necessary to disambiguate the r.h.s. to be reduced. At the time it was concluded that the approach was unpractical for complexity reasons, but the computational power available nowadays could question that early decision. Given that most parsability properties of CF languages are in general undecidable (see e.g. [34,27] for a summary of such results) and that it is even undecidable whether the language generated by a CF grammar is OPL [30], a few natural questions arise.

- To which languages is it possible to apply a preprocessing in the same style as we did for Lua in Section 4, so that the obtained intermediate text is an OPL?
- Furthermore, is it possible and useful to extend our proposed approach to locally parsable languages beyond the OPLs?

The local parsability property can be exploited also in further ways besides parallel parsing.

It is universally acknowledged that, thanks to its malleability, software is subject to continuous evolution, whether for corrective or evolutionary maintenance. Most often the changes applied to a large program are local, as they affect only a small fraction of its syntax tree. This asks naturally for *incremental parsing*, i.e., to modify the existing parse only in the affected part without redoing much identical work. Starting from the early work [25] a fairly rich literature on incremental parsing has been developed (see, e.g., [17,37]) which, unlike the case for parallel parsing, has also produced several practical tools. Such results, however, normally concern more widely adopted families of deterministic languages, which do not enjoy the local parsability property; OPLs instead, can add to the techniques adopted for the more general family, more specific ones directly based on the local parsability property, which may produce simpler and more efficient algorithms; a very early one is sketched in [8]. Mainly, the local parsability property can be exploited in conjunction for *incremental and parallel* parsing in the non-infrequent case of multiple, scattered changes to large pieces of software.

(Syntactic) error management can also take advantage of the local parsability property. In many cases, in fact, a syntax error may affect an unpredictable portion of code, and it is often the case that, at their early occurrence, parsing is stopped or becomes meaningless (e.g., the standard parsers generated by Bison stop their processing at the first error). The local parsability property instead, allowing for (re)starting parsing at any position, may produce, possibly by acting in parallel, large portions of syntax tree associated with correct code, even if such code is preceded by serious errors. Thus, breaking the code into many chunks can help not only to locate the source of the problem but also to fix it without redoing much useless work, by exploiting both parallelism and incrementality. Similar, though less relevant, benefits could be obtained by exploiting parallel lexical analysis.

Finally, let us remember that parsing is normally preliminary to a subsequent semantic phase. In the case of programming languages such semantic analysis consists typically in code interpretation or compilation and is often based on some attribute schema. If we assume, as it is sometimes the case in bottom-up compilation, that the attribute schema is of synthesized type, then, we obtain "for free" parallelism and incrementality for semantic analysis automatically integrated with parallel and incremental lexical and syntactic analysis. The benefits of such exploitation can be enormously extended beyond the realm of programming language compilation since most "structured" design activities can be formalized in terms of a tree-shaped syntax paired with an attribute-based semantic evaluation; furthermore, in many cases the semantic algorithms exhibit a high computational complexity so that the efficiency gained by means of parallelism and, mainly, incrementality can become really impressive. Again, a very early description of an approach pursuing the goal of incremental syntax-semantic analysis is given in [8].

Appendix A

Herein we report Lua's syntactic grammar in operator precedence form.

chunk → block | ENDFILE
block → statList |
retStat |
statList RETURN SEMI |
statList RETURN exprList SEMI |
statList RETURN |
statList RETURN exprList
statList → stat |
SEMI |
stat SEMI |
statList SEMI stat |
statList SEMI
stat → varList XEQ exprList |
functionCall |

label |
BREAK |
GOTO NAME |
DO block END |
DO END |
WHILE expr DO block END |
WHILE expr DO END |
REPEAT block UNTIL expr |
REPEAT UNTIL expr |
IF exprThen END |
IF exprThen ELSE block END |
IF exprThen ELSE END |
IF exprThenElseIfB END |
IF exprThenElseIfB ELSE block END |
IF exprThenElseIfB ELSE END |
FOR name XEQ eCe DO block END |
FOR name XEQ eCeCe DO block END |
FOR nameList IN exprList DO block END |
FUNCTION funcName LPARENFUNC parList RPARENFUNC block END |
FUNCTION funcName LPARENFUNC RPARENFUNC block END |
FOR name XEQ eCe DO END |
FOR name XEQ eCeCe DO END |
FOR nameList IN exprList DO END |
FUNCTION funcName LPARENFUNC parList RPARENFUNC END |
FUNCTION funcName LPARENFUNC RPARENFUNC END |
LOCAL FUNCTION name LPARENFUNC parList RPARENFUNC block END |
LOCAL FUNCTION name LPARENFUNC RPARENFUNC block END |
LOCAL FUNCTION name LPARENFUNC parList RPARENFUNC END |
LOCAL FUNCTION name LPARENFUNC RPARENFUNC END |
LOCAL nameList |
LOCAL nameList XEQ exprList

elseIfBlock → block ELSEIF expr THEN block |
block ELSEIF expr THEN elseIfBlock |
ELSEIF expr THEN block |
block ELSEIF expr THEN |
ELSEIF expr THEN |
ELSEIF expr THEN elseIfBlock

exprThenElseIfB → expr THEN elseIfBlock

exprThen → expr THEN block |
expr THEN

name → NAME

eCe → expr COMMA expr

eCeCe → eCe COMMA expr

dot3 → DOT3

retStat → RETURN SEMI |
RETURN exprList SEMI |
RETURN |
RETURN exprList

label → COLON2 NAME COLON2

funcName → nameDotList |
nameDotList COLON name

```
nameDotList → NAME |
              nameDotList DOT NAME
    varList → var |
              varList COMMA var
        var → NAME |
              prefixExp LBRACK expr RBRACK |
              prefixExp DOT NAME
   nameList → NAME |
              nameList COMMA name
   exprList → expr |
              exprList COMMA expr
       expr → logicalOrExp
logicalOrExp → logicalAndExp |
              logicalOrExp OR logicalAndExp
logicalAndExp → relationalExp |
              logicalAndExp AND relationalExp
relationalExp → concatExp |
              relationalExp LT concatExp |
              relationalExp GT concatExp |
              relationalExp LTEQ concatExp |
              relationalExp GTEQ concatExp |
              relationalExp NEQ concatExp |
              relationalExp EQ2 concatExp
  concatExp → additiveExp |
              additiveExp DOT2 concatExp
additiveExp → multiplicativeExp |
              additiveExp PLUS multiplicativeExp |
              additiveExp MINUS multiplicativeExp
multiplicativeExp → unaryExp |
              multiplicativeExp ASTERISK unaryExp |
              multiplicativeExp DIVIDE unaryExp |
              multiplicativeExp PERCENT unaryExp
   unaryExp → caretExp |
              NOT unaryExp |
              SHARP unaryExp |
              UMINUS unaryExp
   caretExp → baseExp |
              baseExp CARET caretExp
    baseExp → NIL | FALSE | TRUE | NUMBER |
              STRING |
              DOT3 |
              functionDef |
              prefixExp |
              tableConstructor
  prefixExp → var |
              functionCall |
              LPAREN expr RPAREN
functionCall → prefixExp LPAREN exprList RPAREN |
              prefixExp LPAREN RPAREN |
              prefixExp LBRACE fieldList RBRACE |
              prefixExp LBRACE RBRACE |
```

prefixExp STRING |
prefixExp COLON name LPAREN exprList RPAREN |
prefixExp COLON name LPAREN RPAREN |
prefixExp COLON name LBRACE fieldList RBRACE |
prefixExp COLON name LBRACE RBRACE |
prefixExp COLON name STRING

functionDef → FUNCTION LPARENFUNC parList RPARENFUNC block END |
FUNCTION LPARENFUNC RPARENFUNC block END |
FUNCTION LPARENFUNC parList RPARENFUNC END |
FUNCTION LPARENFUNC RPARENFUNC END

parList → nameList | nameList COMMA dot3 | DOT3

tableConstructor → LBRACE fieldList RBRACE |
LBRACE RBRACE

fieldList → fieldListBody | fieldListBody COMMA | fieldListBody SEMIFIELD

fieldListBody → field | fieldListBody COMMA field | fieldListBody SEMIFIELD field

field → bracketedExp EQ expr | name EQ expr | expr

bracketedExp → LBRACK expr RBRACK

References

[1] A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Compilers: Principles, Techniques, and Tools, second edition, Pearson/Addison Wesley, 2007.
[2] Akim Demaille, et al., GNU Bison, http://www.gnu.org/software/bison/, 2014.
[3] H. Alblas, R. op den Akker, P.O. Luttighuis, K. Sikkel, A bibliography on parallel parsing, SIGPLAN Not. 29 (1) (1994) 54–65.
[4] Alon Zakai, Emscripten, https://github.com/kripken/emscripten/wiki, 2014.
[5] R. Alur, P. Madhusudan, Adding nesting structure to words, J. ACM 56 (3) (2009).
[6] A. Barenghi, S. Crespi-Reghizzi, D. Mandrioli, F. Panella, M. Pradella, The PAPAGENO parallel-parser generator, in: A. Cohen (Ed.), Compiler Construction – 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5–13, 2014. Proceedings, in: Lecture Notes in Computer Science, vol. 8409, Springer, 2014, pp. 192–196.
[7] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, M. Pradella, Parallel parsing of operator precedence grammars, Inf. Process. Lett. 113 (7) (2013) 245–249.
[8] D. Bianculli, A. Filieri, C. Ghezzi, D. Mandrioli, Syntactic-semantic incrementality for agile verification, Sci. Comput. Program. (2013), http://dx.doi.org/10.1016/j.scico.2013.11.026.
[9] Celestia Development Team, The celestia space simulation, http://sourceforge.net/projects/celestia/, 2014.
[10] M. Chytil, M. Crochemore, B. Monien, W. Rytter, On the parallel recognition of unambiguous context-free languages, Theor. Comput. Sci. 81 (2) (30 Apr. 1991) 311–316, Note.
[11] J. Cohen, T. Hickey, J. Katcoff, Upper bounds for speedup in parallel parsing, J. ACM 29 (2) (Apr. 1982) 408–428.
[12] S. Crespi Reghizzi, L. Breveglieri, A. Morzenti, Formal Languages and Compilation, Springer, 2013.
[13] S. Crespi Reghizzi, D. Mandrioli, Operator precedence and the visibly pushdown property, J. Comput. Syst. Sci. 78 (2012) 1837–1867.
[14] D. Crockford, RFC4267 – the application/json media type for JavaScript Object Notation (JSON), http://www.ietf.org/rfc/rfc4627.txt, 2006.
[15] D. Crockford, JavaScript – the Good Parts: Unearthing the Excellence in JavaScript, O'Reilly, 2008.
[16] K. De Bosschere, An operator precedence parser for standard prolog text, Softw. Pract. Exp. 26 (7) (1996) 763–779.
[17] P. Degano, S. Mannucci, B. Mojana, Efficient incremental LR parsing for syntax-directed editors, ACM Trans. Program. Lang. Syst. 10 (3) (July 1988) 345–373.
[18] M. DeLoura, The engine survey: general results, http://www.satori.org/2009/03/the-engine-survey-general-results/, March 2009 [Online; accessed 5 December 2013].
[19] C.N. Fischer, On parsing context free languages in parallel environments, Technical report, Cornell University, Apr. 1975.
[20] M.J. Fischer, Some properties of precedence languages, in: P.C. Fischer, S. Ginsburg, M.A. Harrison (Eds.), Proceedings of the 1st Annual ACM Symposium on Theory of Computing, Marina del Rey, CA, USA, May 5–7, 1969, ACM, 1969, pp. 181–190.
[21] R.W. Floyd, Syntactic analysis and operator precedence, J. ACM 10 (3) (1963) 316–333.
[22] R.W. Floyd, Bounded context syntactic analysis, Commun. CACM 7 (2) (1964) 62–67.
[23] Game Developer. 14th Annual Front Line Awards, January 2012.
[24] U. Germann, E. Joanis, S. Larkin, 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, pp. 31–39.
[25] C. Ghezzi, D. Mandrioli, Incremental parsing, ACM Trans. Program. Lang. Syst. 1 (1) (1979) 58–70.
[26] A. Gibbons, W. Rytter, Optimal parallel algorithms for dynamic expression evaluation and context-free recognition, Inf. Comput. 81 (1) (Apr. 1989) 32–45.
[27] S. Greibach, A note on undecidable properties of formal languages, Math. Syst. Theory 2 (1) (1968) 1–6.
[28] D. Grune, C.J. Jacobs, Parsing Techniques: A Practical Guide, Springer, 2008.
[29] W.D. Hillis, G.L. Steele Jr., Data parallel algorithms, Commun. ACM 29 (12) (1986) 1170–1183.
[30] H.B. Hunt, D.J. Rosenkrantz, Computational parallels between the regular and context-free languages, SIAM J. Comput. 7 (1) (1978) 99–114.
[31] A.H. Karp, H.P. Flatt, Measuring parallel processor performance, Commun. ACM 33 (5) (1990) 539–543.
[32] J. Kegler, Perl and undecidability, Perl Rev. (2008).
[33] P.N. Klein, J.H. Reif, Parallel time O(log n) acceptance of deterministic CFLs on an exclusive-write P-RAM, SIAM J. Comput. 17 (1988).
[34] D.E. Knuth, On the translation of languages from left to right, Inf. Control 8 (6) (1965) 607–639.
[35] D.E. Knuth, A characterization of parenthesis languages, Inf. Control 11 (3) (1967) 269–289.
[36] J. Lampe, Local parse – a base for realistic parallelisation of compilers, Elektron. Inf.verarb. Kybern. 26 (1/2) (1990) 75–84.
[37] J. Larchevêque, Optimal incremental parsing, ACM Trans. Program. Lang. Syst. 17 (1) (Jan. 1995) 1–15.

[38] Layer7 Technology. XML Accelerator, http://www.layer7tech.com/products/xml-accelerator, 2014.
[39] W. Lu, K. Chiu, Y. Pan, A parallel approach to XML parsing, in: 7th IEEE/ACM International Conference on Grid Computing, GRID 2006, September 28–29, 2006, Barcelona, Spain, Proceedings, IEEE, 2006, pp. 223–230.
[40] R. McCloskey, J. Wang, J. Belanger, Parallel parsing of languages generated by ambiguous bounded context grammars, Mar. 18 1994.
[41] M.D. Mickunas, R.M. Schell, Parallel compilation in a multiprocessor environment (extended abstract), in: R.H. Austing, D.M. Conti, G.L. Engel (Eds.), Proceedings 1978 ACM Annual Conference, vol. I, Washington, DC, USA, December 4–6, 1978, ACM, 1978, pp. 241–246.
[42] Papageno Developers, PAPAGENO: the parallel parser generator for operator precedence grammars, https://github.com/PAPAGENO-devels/papageno, 2014.
[43] Pontifical Catholic University of Rio de Janeiro. Lua official reference manual, http://www.lua.org/manual/5.2/, 2014.
[44] W. Rytter, On the complexity of parallel parsing of general context-free languages, Theor. Comput. Sci. 47 (3) (1986) 315–321, Note.
[45] D. Sarkar, N. Deo, Estimating the speedup in parallel parsing, IEEE Trans. Softw. Eng. 16 (7) (1990) 677.
[46] R. Sin'ya, K. Matsuzaki, M. Sassa, Simultaneous finite automata: an efficient data-parallel model for regular expression matching, in: 42nd International Conference on Parallel Processing, ICPP, 2013, Lyon, France, October 1–4, 2013, IEEE Computer Society, 2013, pp. 220–229.
[47] G.U. Srikanth, Parallel lexical analyzer on the cell processor, in: SSIRI (Companion), IEEE Computer Society, 2010, pp. 28–29.
[48] The Mozilla Foundation, Asm.js, http://asmjs.org/, 2014.
[49] The Wikimedia Foundation, Wikipedia adopts Lua as its default scripting language, http://en.wikipedia.org/wiki/Wikipedia:Wikipedia_Signpost/2012-01-30/Technology_report, 2012.
[50] L. Vagner, B. Melichar, Parallel LL parsing, Acta Inform. 44 (1) (2007) 1–21.
[51] Z. Zhao, M. Bebenita, D. Herman, J. Sun, X. Shen, HPar: a practical parallel parser for HTML — taming HTML complexities for parallel parsing, ACM Trans. Archit. Code Optim. 10 (4) (Dec. 2013).