Parallel Parsing of Operator Precedence Grammars

Alessandro Barenghi, Stefano Crespi Reghizzi, Dino Mandrioli, Matteo Pradella *

DEIB – Politecnico di Milano
Piazza Leonardo da Vinci, 32, I-20133 Milano, Italy

Abstract

Operator precedence grammars, introduced by Floyd several decades ago, enjoy properties that make them very attractive to face problems and to exploit technologies highly relevant in these days. In this paper we focus on their local parsability property, i.e, the fact that any substring s of a longer one $x.s.y$ can be parsed independently of its context without the risk of invalidating the partial parsing when analyzing another portion of the whole string. We exploit this distinguishing property by developing parallel algorithms and suggest its further application to error recovery and incremental analysis. Great savings in terms of computational complexity are theoretically proved and have been reached in practice by first prototype tools.

Key words: Operator Precedence Languages, Local Parsability, Parallel Parsing, Incremental Parsing.

1 Introduction

Floyd's Operator Precedence grammars (OPGs) and their languages (OPLs) were introduced half a century ago [6,8] and are still used [12] to support efficient deterministic parsing of context-free languages, despite minor limitations in terms of generative power.

Recently the interest in this formalism has been renewed thanks to its closure properties which make it amenable to some fundamental "push-button" verification

* Corresponding author.
Email addresses: `alessandro.barenghi@polimi.it` (Alessandro Barenghi), `stefano.crespi@polimi.it` (Stefano Crespi Reghizzi), `dino.mandrioli@polimi.it` (Dino Mandrioli), `matteo.pradella@polimi.it` (Matteo Pradella).

techniques such as model checking, normally restricted to less powerful language families [5].

In this paper we exploit another distinguishing feature of OPLs, namely their *local parsability*: unlike other classical deterministic languages, whose parsing actions may depend on information arbitrarily distant from the current input string position, for locally parsable languages all actions can be deterministically taken on the basis of a bounded context of the current position. We argue that such property is a key feature to allow a natural exploitation of parallel processing, especially in modern computing systems. By contrast, previous attempts to apply parallel techniques to other traditional parsing algorithms did not produce relevant results, due to the lack of this critical property. The same property also allows to simplify and to enhance the efficiency of previous incremental parsing techniques [10].

We present two algorithmic schemata: the first one splits the input text into substrings and performs local parsing through independent "workers", then the second one recombines their partial outputs to be further processed until completion. We show the benefits of parallelism in terms of computational complexity.

In this paper we focus on the theoretical aspects enabling the proposed techniques; however, they have already been put in action in working tools: [1] complements this paper by describing the practical implementation techniques employed to tailor the algorithm to modern architectures, and includes benchmarks on real world data sets, showing speedups very close to the theoretical expectations. In the conclusions we also hint at further exploitation of the local parsability property to support incremental parsing of locally modified strings and to improve error recovery techniques.

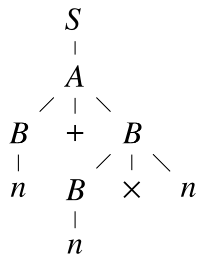
2 Preliminaries

The reader may find more details on OPGs in [5,6,8].
Let V_T be an alphabet and ε the empty string. Let $G = (V_T, V_N, R, S)$ be a *context-free* (CF) grammar, where V_N is the nonterminal alphabet, R the rule (or production) set, and S the axiom. A rule is in *operator form* if its right hand side (r.h.s.) has no adjacent nonterminals; an *operator grammar* (OG) contains only such rules. The following naming convention will be adopted, unless otherwise specified: lowercase Latin letters $a, b, \ldots$ denote terminal characters; uppercase Latin letters $A, B, \ldots$ denote nonterminal characters; letters $r, s, t, u, v, \ldots$ denote terminal strings; and Greek letters $\alpha, \ldots, \omega$ denote strings over $V_T \cup V_N$. The strings may be empty, unless stated otherwise.

For an OG G and a nonterminal A, the *left and right terminal sets* are

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

$$S \rightarrow A \mid B$$
$$A \rightarrow A + B \mid B + B$$
$$B \rightarrow B \times n \mid n$$



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

Figure 1. Arithmetic expressions without parentheses.

where $B \in V_N \cup \{\varepsilon\}$ and $\overset{*}{\Rightarrow}$ denotes the derivation relation.
The following binary operator precedence (OP) relations are defined:

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

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

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

Definition 2 *A OPG is in Fischer normal form [7] if no two rules have the same r.h.s.; no rule, possibly except one with the axiom S as the left hand side (l.h.s.), has ε as the r.h.s.; renaming rules, i.e., those with a single nonterminal character as the r.h.s., are those and only those with S as the l.h.s.*

In the sequel, we assume, without loss of generality, that OPGs are in Fischer normal form. Following the custom of sequential parsers, we enclose the input string between two # special characters, and we assume that # yields precedence to any other character and any character takes precedence over #.

Example 1 *Fig. 1 presents an OPG generating arithmetic expressions where, as usual, multiplication takes precedence over sum (left), the derivation tree of string* $n + n \times n$ *(center) and its OPM (right), where relations with # are left implicit.*

As said, OPGs enjoy a distinguishing local parsability property, unlike other deterministic families of CF languages (e.g. $LR(k)$). In fact, the definition of precedence relations is such that, if there exists a derivation $S \overset{*}{\Rightarrow} \alpha A \gamma \Rightarrow \alpha\beta\gamma$ then it must be $\alpha = \alpha' a$, $\gamma = b\gamma'$, $\beta = N_1 c_1 N_2 c_2 \ldots c_{n-1} N_n$, with $N_i \in V_N \cup \{\varepsilon\}$, $c_i \doteq c_{i+1}$, $1 \leq i < n$, $a \lessdot c_1$, $c_{n-1} \gtrdot b$ [7]. In other words any r.h.s is enclosed within a pair $\lessdot\gtrdot$ and $\doteq$ holds between consecutive terminal characters within it (nonterminals are irrelevant or "transparent" in OPGs); furthermore, being the grammar in Fisher normal form, there are no repeated r.h.s. in R; thus, a shift-reduce algorithm applied to an OPG can extract any r.h.s of any rule within a string in $(V_N \cup V_T)^*$ and reduce it to the corresponding l.h.s. with the certainty that, if the string is legal, then the chosen rule is part of the derivation and will not have to be invalidated by any backtracking action. For instance, with reference to the grammar of Figure 1, if we find a substring $+n\times n\times$ in any context, given that $+ \lessdot n \gtrdot \times$ we can reduce it to $+B\times n\times$; then, since $+ \lessdot \times$ and $n \gtrdot \times$ we further reduce it to $+B\times$ no matter what is the contents of the unspecified part of the string. More generally, if we start with a string $a.s.b$ embedded in any context (t, u), whenever we find a r.h.s. enclosed within a pair $\lessdot\gtrdot$ which in turn is in the context $(t.a, b.u)$, we can proceed with reducing it to its l.h.s. until no more pairs $\lessdot\gtrdot$ exist in the reduced string in $(V_N \cup V_T)^*$. As a consequence we obtain the following fundamental statement, which is the basis of the algorithms described in this paper.

Statement 1 *For any string asb such that* $S \overset{*}{\Rightarrow} tasbu$ *there exists a unique* α *such that* $S \overset{*}{\Rightarrow} ta\alpha bu \overset{*}{\Rightarrow} tasbu$*, and, in* $a\alpha b$*,* $\alpha = \beta.\gamma$ *and in* β *(resp.* γ*) only* $\gtrdot$ *and* $\doteq$ *relations occur (resp.* $\lessdot$ *and* $\doteq$*). If* β *or* γ *are* ε*, then either only* $\lessdot$ *and* $\doteq$ *or only* $\gtrdot$ *and* $\doteq$ *relations occur between consecutive terminal characters.*

Such a unique α is called the *minimal reduced string* deriving s in the context (a, b); in fact this condition implies that no further reduction can be applied to α without affecting its context (a, b).

We now present two algorithms for sequential and parallel parsing of OPLs, based on Statement 1. They can be used either in isolation or integrated with a compiler.

3 Parallel parsing of Operator Precedence Languages

As a first step we generalize the traditional shift-reduce parsing algorithm for OPGs to analyze strings in $(V_N \cup V_T)^*$; such strings begin and end with terminal characters but not necessarily #. This feature is needed when the algorithm is called to parse internal substrings in the parallel version.

Algorithm 1 is based on a stack $\mathcal{S}$ containing symbols (x, p), $x \in V_T \cup V_N$, $p \in \{\lessdot, \doteq, \gtrdot, \perp\}$, $\perp$ standing for undefined. The second component is used to encode the precedence relation found between two terminal symbols - thus, it is always $p = \perp$ for nonterminals. When the precedence symbols are not needed, we will use the post-fix operator $|_1$ to denote the homomorphism that erases precedence information, i.e. $(x, p)|_1 = x$. We assume that the stack grows rightwards.

When used sequentially and not as part of the parallel version outlined below, the algorithm is called with $u = \#s\#$, $\mathcal{S} = (\#, \perp)$, never performs step (4), and accepts the input only if $\mathcal{S} = (\#, \perp)(S, \perp)$.

With reference to Statement 1, string $a\alpha$ is exactly the contents of the stack of an

Algorithm 1

```
Sequential-parsing($\mathcal{S}$, $u$, head, end)
Initialization: $u = a.s.b$; $\mathcal{S} = (a, \perp)$; head, end are respectively set to the position
of the first character of $s$, and of $b$.
  (1) Read the symbol $x$ at position head in $u$, and consider its precedence relation
      with the top-most terminal $y$ found in $\mathcal{S}$.
  (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 $\lessdot$ then push $(x, \gtrdot)$; head := head + 1.
      (b) Else, let $\mathcal{S}$ be $(x_0, p_0)(x_1, p_1) \dots (x_i, p_i) \dots (x_n, p_n)$
          where $\forall j, i+1 < j \le n, p_j \ne \lessdot$; either $p_i = \lessdot$ and $p_{i+1} \ne \lessdot$, or $p_i = \perp$
          (i.e. $x_i \in V_N$) and $p_{i+1} = \lessdot$ then
          if $\exists A$: $A \to x_i \dots x_n \in R$, replace $(x_i, p_i) \dots (x_n, p_n)$ in $\mathcal{S}$ with $(A, \perp)$;
          otherwise start an error recovery procedure. [1]
  (6) If head < end or (head = end and $\mathcal{S} \ne (a, \perp)(B, \perp)$), for any $B \in V_N$, repeat
      from step (1); else return $\mathcal{S}$.
```

OPG parser that parses left-to-right asb just before shifting b. As a particular case, a and/or b can be the delimiter #.

To split the load among different workers, we divide the input string into k substrings, depending on the actual parallelism offered by the machine. The division is arbitrary, unlike other parallel parsers (e.g. [15]), which require the input to be cut in positions marked by characters (e.g. `begin`) that open syntactic constituents. We then apply Algorithm 1 to each substring. Thanks to the locality principle, its output will certainly be part of the complete parsing of the whole original string. Due to the fact that OP parsing is rooted in a look-ahead/look-back of one character to evaluate the precedence relations between consecutive terminal characters, the last character of each substring coincides with the first character of the following one. For instance, consider the grammar of Example 1. If the input string is $n + n + n \times n \times n + n \times n + n$ and $k = 3$, we may split it in three parts $\overbrace{n+n}+\overbrace{n\times n\times n+}n\overbrace{\times n+n}$ where # delimiters are left implicit and the symbols $+$, n not embraced are shared by the two adjacent substrings.

Thanks to Statement 1, after a sequential step on a substring we arbitrarily choose a point where to split the stack $\mathcal{S}$ 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 our example, we obtain the following three stacks: $\mathcal{S}_1 = (\#, \perp)(A, \perp)(+, \lessdot)$; $\mathcal{S}_2 = (+, \perp)(B, \perp)(+, \gtrdot)(n, \lessdot)$; $\mathcal{S}_3 = (n, \perp)(\times, \gtrdot)(n, \doteq)(+, \gtrdot)(B, \perp)(\#, \gtrdot)$. In this case, we have $\mathcal{S}_2^L = (+, \perp)(B, \perp)(+, \gtrdot)$, and $\mathcal{S}_2^R = (n, \lessdot)$.

To combine the results $\mathcal{S}^L\mathcal{S}^R$ and $\mathcal{S}'^L\mathcal{S}'^R$ of two adjacent workers, say $\mathcal{W}$ and $\mathcal{W}'$ to its right, respectively, we need to prepare the initial stack $\mathcal{S}$ and input string u of another run of Algorithm 1 for each worker. We take $\mathcal{S}$ as $\mathcal{S}_{combine}(\mathcal{S}^L, \mathcal{S}^R) :=$

$(a, \perp)\mathcal{S}^R$, where a is the top symbol of $\mathcal{S}^L$; while the input string u is $u_{combine}(\mathcal{S}'^L) := u'$, where u' is the suffix of $\mathcal{S}'^L|_1$ without its first symbol (which is already on the top of $\mathcal{S}^R$).
In our example, we obtain: $\mathcal{S}_{combine}(\mathcal{S}_2^L, \mathcal{S}_2^R) = (+, \perp)(n, \lessdot)$, and $u_{combine}(\mathcal{S}_3^L) = \times n + B\#$.

A few special but simple cases, e.g. when $\mathcal{S}'$ contains only $\lessdot$ and $\doteq$ precedence relations, i.e., when the string α referred to in Statement 1 is empty, are not considered here in detail for brevity.

Algorithm 2

```
Parallel-parsing($\underline{u}$, $k$)
 (1) Split the input string $\underline{u}$ into $k$ substrings: $\#u_1u_2 \dots u_k\#$.
 (2) Launch $k$ instances of Algorithm 1, where, for each $1 \le i \le k$, the parameters
     are $\mathcal{S}_i = (a, \perp)$, $u = u_i b$, head $= |u_1u_2 \dots u_{i-1}|$, end $= |u_1u_2 \dots u_i| + 1$; $a$ is the
     last symbol of $u_{i-1}$, and $b$ the first of $u_{i+1}$. Conventionally $u_0 = u_{k+1} = \#$. The
     result of this phase are $k$ pairs of stacks $\mathcal{S}_i^L\mathcal{S}_i^R$, as specified above.
 (3) Repeat: [2]
     (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 $\mathcal{S} = \mathcal{S}_{combine}(\mathcal{S}_i^L, \mathcal{S}_i^R)$, $s = u_{combine}(\mathcal{S}_{i+1}^L)$,
         head = 1, end = $|u|$.
     (b) Until either we have a single reduced stack $\underline{\mathcal{S}}$ or the computation is
         aborted and some error recovery action is taken.
 (4) Return $\underline{\mathcal{S}}$.
```

4 Complexity

Parallel parsing techniques are obviously motivated by achieving gains in efficiency. In terms of asymptotic complexity the first requirements that we state for a positive evaluation of such techniques are: a best-case linear speedup w.r.t the number of processors; a worst case performance that does not exceed the complexity of a complete sequential parsing.

To meet these requirements, it is essential that the combination of stacks $\mathcal{S}_i$ and $\mathcal{S}_{i+1}$, inside step (3)(a) of Algorithm 2, takes $O(1)$ time (hence overall $O(k)$ for k workers). This goal is easily achieved by maintaining, 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 $\lessdot$ is detected and then updated every time a reduction is applied 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 the example above, it is initialized at the position of the $+$ symbol and remains there after the three reductions $B \Rightarrow n$, $B \Rightarrow B \times n$, $B \Rightarrow B \times n$, because $+ \lessdot n$ and $+ \lessdot \times$; then, when the second $+$ (the third of the whole string) is shifted (without removing the previous one because the $\gtrdot$ between the two $+$ is not matched by a corresponding

$\lessdot$ at its left), the marker is moved to point to the position of the second + since $+ \lessdot n$ where it will mark the beginning of $\mathcal{S}_2^R$. These operations require a time $O(1)$ whether we implement the stacks by means of arrays or by means of more flexible linked lists.

5 Related Works

In principle, other formal conditions and related families of grammars capturing the notions of local parsability could be devised. For instance, Floyd himself proposed a generalization of his grammars by dropping the operator grammar form and considering a "bounded context of length k" as the one that allows to decide unambiguously the reduction to be applied [9]; OPGs fall in this category with $k = 1$, but the generalization did not prove cost-effective for serial parsing.

The local parsability property can also be found in some grammar families that are based on Church-Rosser (CR) Thue systems. The simplest family are the Nonterminally Separated grammars [3], which have a rather limited generative capacity. Other CR families include also some context-sensitive languages [13]. The ability to start reduction at any position makes CR grammars potentially attractive for parallel parsing, but we do not know of published work in that direction.

Other approaches, e.g., [14,11,2] aimed at building parallel parsers for LR grammars but, due to the lack of the local parsability property, the few people who have performed some limited experimentation on such algorithms have generally found that performances critically depend on the cut points between substrings: if a substring starts, say, with `begin`, the parser can recognize almost completely a full language block. On the contrary, starting a substring on an identifier opens too many syntactic alternatives. As a consequence such parsers have been typically combined with language-dependent heuristics for splitting the source text into substrings that start on keywords announcing a splitting friendly construct.

We also report that the efforts toward parallelizing the parsing of common LR grammars, such as the ones made by Mickunas and Schell [14] were not able to devise an algorithm able to exploit parallel architectures. Their technique is based on starting multiple parsing processes in different portions of the input and, if a reduction move needs more symbols than the ones present on the stack, it is checked whether the previous workers produced them. This implies that the parse actions done by the different workers are not final as the absence of the required nonterminals may invalidate reductions.

In summary, since OPGs are currently the only ones of any practical usage that enjoy a local parsability property, we focused our attention on this class of grammars.

6 Conclusions and Further Work

In this paper we have set the theoretical foundations to exploit the local parsability property of OPLs by means of parallel elaboration. A first practical application of the proposed techniques for parallel parsing is reported in [1] together with very encouraging experimental results which fully confirm theoretical expectations. The local parsability property, however, can be further exploited along other directions:

- First, it can be applied to incremental parsing. In many cases original input strings are modified, either to fix mistakes of previous versions or to adapt them to new requirements; most often such changes affect only a small part of the syntax tree associated with the input string, so that an incremental technique that avoids redoing parsing of unaffected portions may result in much saving in time and space complexity. Various incremental parsing algorithms have been presented in the literature, mostly based on traditional LR parsing and originated by our early work [10]. In the case of OPLs the local parsability property can further enhance the benefits of incrementality. In fact, if, for an OPG $aAb \overset{*}{\Rightarrow} asb$, then for every t, u, $S \overset{*}{\Rightarrow} tasbu$ iff $S \overset{*}{\Rightarrow} taAbu \overset{*}{\Rightarrow} tasbu$. As a consequence, if s is replaced by v in the context (ta, bu), if $aAb \overset{*}{\Rightarrow} avb$, then $S \overset{*}{\Rightarrow} taAbu \overset{*}{\Rightarrow} tavbu$, and (re)parsing of $tavbu$ can be stopped at $aAb \overset{*}{\Rightarrow} avb$.
- The capability to recover from errors is essential in any practical application, the more so for the very large texts that our parallel algorithm is able to efficiently parse. Error recovery in sequential parsing has a long history (see, e.g., [12]) but very little has been done to adapt the sequential approach to parallel parsing [4]. A risk is that traditional error recovery actions would slowdown parallel parsing. As a future development, we hint to the possibility to increase the worker pool with one or more workers devoted to error handling or to augment present workers with ad-hoc error handling procedures.
- Furthermore syntactic parallel and/or incremental analysis could be naturally paired with semantic analysis, e.g., based on attribute schemata [4].

In summary we aim at building a complete syntactic and semantic environment that assists the user in the managing of any type of long and complex documents by parallelizing lexical [3], syntactic, and semantic analysis and by supporting their evolution in an incremental way.

References

[1] A. Barenghi, E. Viviani, S. Crespi Reghizzi, D. Mandrioli, and M. Pradella. PAPAGENO: a parallel parser generator for operator precedence grammars. In *SLE*

[3] Parallelizing lexical analysis is conceptually much simpler than syntax analysis since it is “intrinsically local”.

2012, volume 7745 of *Lecture Notes in Computer Science*, pages 264–274. Springer, 2013.

[2] J. Bates and A. Lavie. Recognizing substrings of LR(k) languages in linear time. *ACM Trans. Program. Lang. Syst.*, 16:1051–1077, May 1994.

[3] L. Boasson and G. Sénizergues. NTS languages are deterministic and congruential. *J. Comput. Syst. Sci.*, 31(3):332–342, 1985.

[4] G. Clarke and D. T. Barnard. Error handling in a parallel LR substring parser. *Comput. Lang.*, 19(4):247–259, Oct. 1993.

[5] S. Crespi Reghizzi and D. Mandrioli. Operator precedence and the visibly pushdown property. *Journal of Computer and System Science*, 78:1837–1867, 2012.

[6] S. Crespi Reghizzi, D. Mandrioli, and D. F. Martin. Algebraic properties of operator precedence languages. *Information and Control*, 37(2):115–133, May 1978.

[7] M. J. Fischer. Some properties of precedence languages. In *STOC '69: Proc. first annual ACM Symp. on Theory of Computing*, pages 181–190, 1969.

[8] R. W. Floyd. Syntactic analysis and operator precedence. *JACM*, 10(3):316–333, 1963.

[9] R. W. Floyd. Bounded context syntactic analysis. *CACM*, 7(2):62–67, 1964.

[10] C. Ghezzi and D. Mandrioli. Incremental parsing. *ACM Trans. Program. Lang. Syst.*, 1(1):58–70, 1979.

[11] H. Goeman. On parsing and condensing substrings of LR languages in linear time. *Theor. Comput. Sci.*, 267:61–82, September 2001.

[12] D. Grune and C. J. Jacobs. *Parsing techniques: a practical guide*. Springer, 2008.

[13] R. McNaughton, P. Narendran, and F. Otto. Church-rosser thue systems and formal languages. *J. ACM*, 35(2):324–344, 1988.

[14] M. D. Mickunas and R. M. Schell. Parallel compilation in a multiprocessor environment (extended abstract). In R. H. Austing, D. M. Conti, and G. L. Engel, editors, *ACM Annual Conference (1)*, pages 241–246. ACM, 1978.

[15] D. Sarkar and N. Deo. Estimating the speedup in parallel parsing. *IEEE Trans. on Softw. Eng.*, 16(7):677, 1990.