# Plex: Scaling Parallel Lexing with Backtrack-Free Prescanning

1st Le Li
*The University of Tokyo*
Tokyo, Japan
lile@eidos.ic.i.u-tokyo.ac.jp

2nd Shigeyuki Sato
*The University of Tokyo*
Tokyo, Japan
sato.shigeyuki@mi.u-tokyo.ac.jp

3rd Qiheng Liu*

4th Kenjiro Taura
*The University of Tokyo*
Tokyo, Japan
0000-0001-5224-382X

2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 978-1-6654-4066-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPS49936.2021.00079

***Abstract*—Lexical analysis, which converts input text into a list of tokens, plays an important role in many applications, including compilation and data extraction from texts. To recognize token patterns, a lexer incorporates a sequential computation model — automaton as its basic building component. As such, it is considered difficult to parallelize due to the inherent data dependency. Much work has been done to accelerate lexical analysis through parallel techniques. Unfortunately, existing attempts mainly rely on language-specific remedies for input segmentation, which makes it not only tricky for language extension, but also challenging for automatic lexer generation.**

**This paper presents Plex — an automated tool for generating parallel lexers from user-defined grammars. To overcome the inherent sequentiality, Plex applies a fast prescanning phase to collect context information prior to scanning. To reduce the overheads brought by prescanning, Plex adopts a special automaton, which is derived from that of the scanner, to avoid backtracking behavior and exploits data-parallel techniques. The evaluation under several languages shows that the prescanning overhead is small, and consequently Plex is scalable and achieves 9.8-11.5X speedups using 18 threads.**

***Index Terms*—Lexical Analysis, Finite Automaton, Parallelism**

## I. Introduction

As the first phase of parsing, lexical analysis (*lexing*) dominates many important applications, such as data analytics and compilation [1]–[3]. One significant stage of lexing is *scanning* breaking input text into a list of lexical units *(lexeme)*, whose patterns are specified in regular expressions (*regex*). Lexers can be built by hand or automated tools — lexer generators (e.g., Lex [4] and Flex [5]), which ease programmers and provide many advanced features, although manual optimizations are sometimes essential. Taking as input a user-defined grammars, these tools emit a lexer facilitating the implementation of an analyzer. The metalanguage of a grammar can be described in Backus-Naur form, as Listing 1 defining four types of lexemes — `INT`, `FLOAT`, `DOT` and `ELLIPSIS`.

It is reported that parsing is a key bottleneck in querying raw data [6], [7], and lexing dominates 25-41% of the total cost based on the measurement [1]. Recent data explosion trend motivates parallelization on both of them [1]–[3], [8]. However, the scanning phase is usually based on an automaton,

```
digit    := '[0-9]' ;
dot      := '.' ;

INT      := {digit}+ ;
FLOAT    := {digit}+{dot}{digit}+ ;
DOT      := {dot} ;
ELLIPSIS := {dot}{dot} ;
```

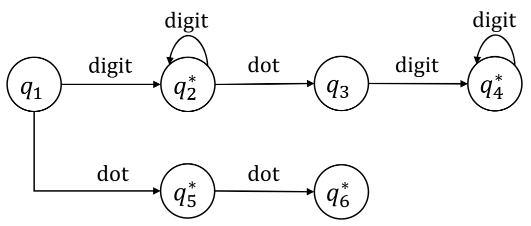
Listing 1: Lexical Specification

which is known as an inherent sequential computational model [9]. Motivated by its broad applicability, researchers proposed many methods to parallelize automaton computation, such as *speculative simulation* [10]–[13]. These efforts also contribute significantly to parallel lexing [14], [15], in which an inevitable issue is input segmentation that forces programmers to design language-specific remedy, although they have an automated tool [1]–[3]. These tools can generate parallel parsers, while their lexers are made from Lex/Flex instances requiring manual modifications to segment input. Particularly, their segmentation strategies rely on likely delimiters declaring the boundary of lexemes, which can be tricky when delimiters are allowed as part of a lexeme [1]. Therefore, remaining a question that whether there is an automatic procedure of generating parallel lexers.

In this work, we generalize the key issue in parallel scanning and present Plex (Parallel lex), whose biggest merit is getting rid of the language-specific design for input segmentation. Specifically, Plex overcomes the sequentiality by prescanning the input sequence over a special automaton to collect context information, which is then used as a guidance for the scanner. The idea of applying prescanning is similar to the approaches performing two-passes run over the input for parallel prefix computation and HTML tokenization [14], whereas Plex specialized it for parallel scanning. The prescanning automaton is derived from that of the scanner for two reasons; 1) it builds a transfer function to concatenate chunks; 2) it eliminates backtracking behavior of scanning to perform prescanning as a simple DFA computation and lower prescanning overheads. We show that the prescanning overheads of Plex is low and can be further alleviated through data-parallel techniques [14], [16], [17]. Consequently, Plex is scalable and achieves 9.8-11.5X speedups using 18 threads. The major contribution of this work is the first code generator for parallel lexers, which improves productivity for not requiring manual modifications.

This work was supported by JSPS KAKENHI Grant Numbers 16H01715 and 18K18032.

*This work was done at The University of Tokyo.



(a) $\mathcal{D}_{lex}$ for Listing 1: $q_2^*, q_4^*, q_5^*, q_6^*$ accept INT, FLOAT, DOT, ELLIPSIS

| $\tau$ | digit | dot | # |
|---|---|---|---|
| $q_0$ | $q_0$ | $q_0$ | $q_0$ |
| $q_1$ | $q_2^*$ | $q_5^*$ | $q_0$ |
| $q_2^*$ | $q_2^*$ | $q_3$ | $q_0$ |
| $q_3$ | $q_4^*$ | $q_0$ | $q_0$ |
| $q_4^*$ | $q_4^*$ | $q_0$ | $q_0$ |
| $q_5^*$ | $q_0$ | $q_6^*$ | $q_0$ |
| $q_6^*$ | $q_0$ | $q_0$ | $q_0$ |

(b) $\tau$ for Figure 1(a)

```
1: $input = c_1c_2...c_n$
2: $state = q_1$
3: for $i = 1 \to n$ do
4:     $state = \tau(state, c_i)$
5: end for
```

(c) DFA Sequential Computation

Figure 1: Example DFA (Symbol $^*$ marks an accepting state)

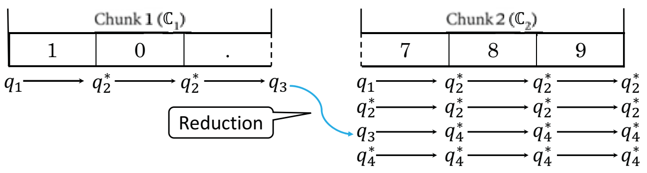

Chunk 1 ($\mathbb{C}_1$)
Chunk 2 ($\mathbb{C}_2$)
1
0
.
7
8
9
$q_1$
$q_2^*$
$q_2^*$
$q_3$
Reduction
$q_1$ $q_2^*$ $q_2^*$ $q_2^*$
$q_2^*$ $q_2^*$ $q_2^*$ $q_2^*$
$q_3$ $q_4^*$ $q_4^*$ $q_4^*$
$q_4^*$ $q_4^*$ $q_4^*$ $q_4^*$

Figure 2: Example of Speculative Simulation

## II. Background

This section gives definitions and notations for automata, and illustrations of scanning. Along with examples, we show the difficulties in parallel scanning.

### A. *Finite Automaton and Its Parallelization Problem*

Finite Automaton (FA) is widely used in many applications, such as Huffman decoding, regex matching, etc. A scanner incorporates an FA to recognize lexeme patterns.

**Definition 1.** *A* Nondeterministic Finite Automaton (NFA) *can be represented by a quintuple $\mathcal{N} = (Q, \sum, \mathcal{T}, q_1, \mathcal{A})$, where $Q$ is a finite set of states; $\sum$ is a finite set of input symbols; $\mathcal{T}$: $Q \times \sum \to \mathcal{P}(Q)$ is a transition function; $q_1 \in Q$ is an initial state; $\mathcal{A} \subseteq Q$ is a finite set of accepting states. Notation $\mathcal{P}(Q)$ denotes the power set of $Q$; Given the next input symbol $i$ for current state $q$, notation $\mathcal{T}(q, i)$ denotes a set of all next states.*

**Definition 2.** *A* Deterministic Finite Automaton (DFA) *is a special case of NFA, denoted by $\mathcal{D} = (Q, \sum, \tau, q_1, \mathcal{A})$. The transition function is a singleton, $\tau$: $Q \times \sum \to Q$. It can be represented by a two-dimensional array of states. The row and column entries specify the current state $q$ and the next input symbol $i$, respectively. Notation $\tau(q, i)$ denotes the next state.*

Fig. 1(a) and 1(b) are the $\mathcal{D}$ and $\tau$ accepting the grammar in Listing 1. The symbol # denotes any input symbol other than digits and dot. In the remainder of this paper, the DFA used in lexing is denoted as $\mathcal{D}_{lex}$. In addition, $q_0$ is a special state, called *the invalid state*. All the transitions from $q_0$ goto

Table I: Procedure of Maximal Munch Principle

| Step | Description (★: example of generating the first lexeme in Fig. 3) |
|---|---|
| (1) | The scanner begins with $q_1$ from a text postion $p_s$ (initially, $p_s$=0), meanwhile memorizing $p_{\mathcal{A}}$ (the position of the last accepting state). ★ *Beginning with $q_1$, the scanner reaches $q_2^*$ after consuming '10'* |
| (2) | If reaching the invalid state $q_0$ at a position $p_e$, it backtracks to $p_{\mathcal{A}}$. ★ *It forwards to the first dot, speculating '10.' is the prefix of a* FLOAT*. Then, it reaches $q_0$ at the second dot, and backtrack to '0'* |
| (3) | After emitting $c_{p_s}...c_{p_{\mathcal{A}}}$ as a lexeme, it restarts (1) from $p_s$=$p_{\mathcal{A}}$+1. ★ *After emitting* INT*(10), it restarts (1) from the first dot.* |

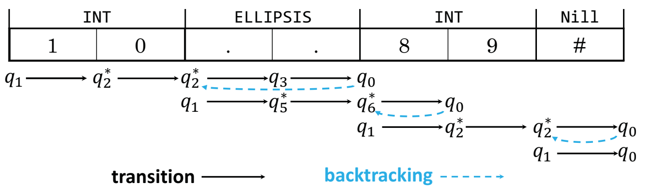

INT
ELLIPSIS
INT
Nill
1
0
.
.
8
9
#
$q_1$ $q_2^*$ $q_2^*$ $q_3$ $q_0$
$q_1$ $q_5^*$ $q_6^*$ $q_0$
$q_1$ $q_2^*$ $q_2^*$ $q_0$
$q_1$ $q_0$
transition
backtracking

Figure 3: Example of Maximal Munch Principle

itself. For simplicity, $q_0$ and transitions involving it are hidden. Fig. 1(c) shows a straightforward implementation of DFA computation, which is inherently sequential due to the data dependency between iterations, since the entry *state*.

Among proposals for parallelizing DFA computation, a typical representative is *speculative simulation* (SpecSim) that speculates likely start states for each chunk of the input. Fig. 2 shows how to match a FLOAT(10.789) in parallel. The text is split into two chunks, of which $\mathbb{C}_1$ starts with $q_1$ naturally. On the other hand, four states ($q_1$,$q_2^*$,$q_3$,$q_4^*$) are speculated as likely start states for $\mathbb{C}_2$. After both are processed, a phase called *reduction* matches states of each two adjoining chunks, as the $q_3$ between $\mathbb{C}_1$ and $\mathbb{C}_2$. In case of a speculation failure occurs, a re-execution is inevitable. For example, re-running $\mathbb{C}_2$ by starting from $q_3$, if $q_3$ was not speculated.

### B. *Scanning and Maximal Munch Principle*

Scanning is similar to regex matching in the sense that the former essentially matches the input with a repetition of regexes representing a single lexeme. For the example in Listing 1, scanning is similar to match the input text against regex '(INT|FLOAT|DOT|ELLIPSIS)∗'. However, it is usually ambiguous, e.g., '10.789' can be partitioned in several ways, such as FLOAT(10.789) and INT(10); DOT(.); INT(789). To resolve this ambiguity, a widely used principle is *maximal munch* (or *longest match*) [18]–[20] declaring that "*a lexeme should be formed from the longest matched prefix of the remaining input, independently from others*". For example, given rules A:='aaa' and B:='aa', input 'aaaaaaa' is taken as A(aaa); A(aaa); unknown(a), regardless of any potential best match, such as A(aaa); B(aa); B(aa). It is applicable in vast majority of cases [5], and has been studied in [20] proposing methods to warn designers about potential ambiguity that violates the principle. Table I describes the procedure summarized in three steps, (1)*maximal munch*, (2)*backtracking* and (3)*restarting*. Fig. 3 shows an example having INT(10); ELLIPSIS(..); INT(89) as the only correct result.

Accordingly, scanning is formed of a series of Fig. 1(c) separated by backtracking, which changes the scanning scope (Line 3) to $p_s \rightarrow p_e$. The co-effect of maximal munch and backtracking is *invalid matching* (e.g., $q_2^* \xrightarrow{\cdot} q_3 \xrightarrow{\cdot} q_0$), which does not emit any lexeme but costly due to duplicate character consumption (e.g., the two dots are consumed twice). For an input of length $n$, this repetition may result in an $O(n^2)$ of time complexity in the worst case [19], which is highly different from the linear case for a large $n$. Memoization is an approach to overcome this overhead by memorizing <*position*,*state*> pairs that finally become invalid [19], [21]. Therefore, it can backtrack directly if reaching the same *state* at the identical *position* again.

To parallelize scanning, existing proposals try to exactly discover lexeme boundaries, which mainly relies on language-specific delimiters guaranteeing to end a lexeme. Alessandro et al. discussed approaches for JSON and Lua [1]. JSON can be split at newline character, which is a perfect delimiter for not allowed in any lexeme. Note that, this is infeasible for automatically generated JSON text that usually lacks newline characters. Unfortunately, this does not work for Lua, which supports string literals and comments spanning multiple lines. In this case a lookahead is necessary to discover delimiters closing them. Similar schemes are also seen in a parallel tokenizer for APL [2]. Besides, data-parallel approaches also contribute significantly. Chandra et al. proposed *associative parallel lexing* counting the lengths of delimiter-separated lexemes with byte-level shifting [8]. Benefit from bitwise techniques [23], Yinan, Langdale et al. proposed high performance analyzers recognizing JSON-specific constructs [6], [22].

### C. Problem Formalization

The idea of discovering delimiters for input segmentation is practical if the language specifies some perfect delimiters, like a newline in JSON. Even so, many datasets are high volume and may lack delimiters for lowering size and not aiming to be directly viewed by human. Moreover, there are no guarantees that actual datasets have delimiters frequently enough so that the scanner can find delimiters quickly from an arbitrary text position without affecting load balancing (chunk size). To give a general solution, especially for generating parallel lexer automatically, we formalize the problem in parallel scanning.

Assume a given input contains $n$ lexemes $\mathcal{L}_1 ... \mathcal{L}_n$. To match $\mathcal{L}_i$ ($1 \leq i \leq n$), the main question is "how to correctly find out its start position $p_s$", unlike the sequential case naturally starting from leftmost, i.e., $p_s$=0. The ideal situation is that the chunk's boundaries coincide with the lexemes', so that threads can stop at the end of chunks; otherwise, they should scan ahead until reaching the invalid state $q_0$. Unfortunately, this is hard to guarantee for user-defined grammars, even for a specific language allowing delimiters in lexeme patterns, like Lua discussed previously. Similar to SpecSim, one may expect to obtain result through reduction. However, backtracking poses challenges, since SpecSim is originated to parallelize pure automaton computation. In addition, the reduction must be carefully handled, since there could be many states at an identical text position As shown in Fig. 3, at the text position between the two dots, $q_3$ belongs to an invalid matching when emitting `INT`(10), while the reduction for `ELLIPSIS` is given from $q_5^*$. Besides, backtracking may cross chunks, e.g., taking '.89#' as a chunk with speculated state $q_3$, it reaches the invalid state $q_0$ and attempts to backtrack, while the position of the last accepting state is out of the chunk's scope.

Therefore, if there is no way to find out lexeme boundaries, a parallel scanner must be able to handle the two cases, 1) *the scanning scope may cross chunks for obtaining a longest prefix*; 2) *the backtracking position may be in one of previous chunks*. The former is easy to handle by loading input into continuous addresses of shared-memory. The latter is arduous, since such position is usually unknown before the completion of all preceding chunks. In addition, avoiding wasting time in enumerating all likely start states is valuable.

Unlike existing work tackling with specific languages to discover appropriate splitting points [1], [8], Plex gives a general parallel scanning scheme based on prescanning, which uses two sets of states to build a transfer function mapping the start to the end of a chunk. Combining all transfer functions together, it determines all reduction states for each chunk. The prescanning is followed by the parallel scanning phase, in which each thread starts with the states in the result of a chunk's transfer function, one after another. If a particular state leads the scanner to the invalid state $q_0$ that would have to backtrack to a previous chunk, it simply chooses the next state from the result of transfer function to obey maximal munch.

Our idea *backtrack-free* originates from above insights: (1) It enables SpecSim for prescanning; (2) It reduces computation by removing duplicate character consumption; (3) It facilitates the data-parallel design (described in Section IV-C).

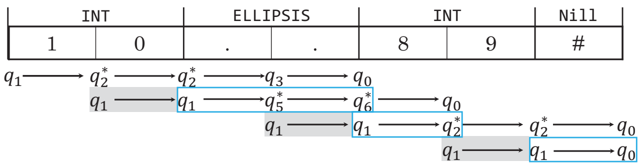
## III. Backtrack-Free Prescan FA

In order to get rid of such language-specific design, Plex guides the scanner by collecting context information with a prescanner, which adopts a special DFA (denoted by $\mathcal{D}_{prescan}$) derived from $\mathcal{D}_{lex}$. This section introduces the necessity and construction of $\mathcal{D}_{prescan}$. The prescanning and scanning phases are illustrated in Section IV.

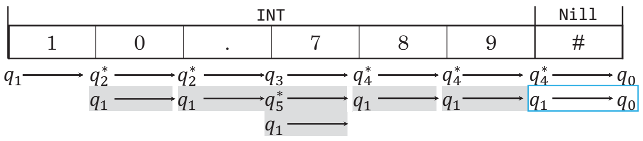
### A. Overview

The necessity of $\mathcal{D}_{prescan}$ originates from two points. First, the mission of prescanning is collecting context information. Specifically, taking '.89#' of Fig. 3 as a chunk, the result should involve three possibilities of the chunk's start position: (1) $q_3$: within a `FLOAT` with prefix '10.'; (2) $q_5^*$: within an `ELLIPSIS` with prefix '.'; (3) $q_1$: at the end of a `DOT`. Besides, they should be examined orderly to obey maximal munch; otherwise, the generated result may be incorrect. For example, if taking (3), the scanner emits `DOT`(.) instead of `ELLIPSIS`(..).

Second, the prescanning overhead is critical. As introduced, due to the backtracking behavior, the worst time complexity of scanning is $O(n^2)$. Besides, the necessity of runtime checks (if the current state is an accepting state or the invalid state in Algorithm 4) results in unfavorable branches. Therefore,



(a) Speculation Cancellation for Continuous Accepting States



(b) Speculation Cancellation for Non-continuous Accepting States

Figure 4: Backtrack-Free with Speculation

eliminating them is essential for implementing a fast prescanning scheme, which does not need to make choice to emit any lexeme. In another word, making it backtrack-free.

The $\mathcal{D}_{prescan}$ is built upon these considerations. To resolve the above two issues, all backtracking cases are embedded into the automaton itself by *speculating backtracking positions* to conduct an automaton transformation ($\mathcal{D}_{lex}\rightarrow\mathcal{N}_{prescan}$), which is termed as *speculation embedding* in this paper. The resulting $\mathcal{N}_{prescan}$ is re-transformed to a deterministic one (i.e., $\mathcal{D}_{prescan}$) through *ordered subset construction*, which borrows basic idea from *subset construction* but imposing more restrictions.

## B. Speculating Backtracking Positions

Maximal munch leads to an uncertainty when reaching an accepting state, i.e., the text position could be either within or at the end of a lexeme, as $q_1\xrightarrow{1}q_2^*$ and $q_2^*\xrightarrow{0}q_2^*$ in Fig. 3. Based on this insight, we could always speculate that every accepting state ends a lexeme, which in turn starts another. As in Fig. 4(a), after consuming '1', the scanner is at an accepting state $q_2^*$. Thus, a new path restarting from $q_1$ is created by speculating '1' is an `INT`. However, the speculation is turned out to be failed for reaching an accepting state again ($q_2^*$ after '0'). Therefore, the speculated path should be canceled, and another one is created with the identical procedure. Then, speculating '10' as an `INT` is turned out to be a speculation success after consuming the two dots.

A speculation success contracts invalid and valid matchings, e.g., $[q_2^*, q_1]\overset{.}{\mapsto}[q_3, q_5^*]\overset{.}{\mapsto}[q_0, q_6^*]$ combines invalid matching for obtaining a longest prefix, and valid matching for `ELLIPSIS`. Hence, it is backtrack-free with such speculation. To involve the speculation-based parallelism, we conduct an automaton transformation to embed speculated transitions into automaton.

## C. Speculation Embedding ($\mathcal{D}_{lex}\rightarrow\mathcal{N}_{prescan}$)

Algorithm 1 shows the procedure, which only operates the $\tau$ of $\mathcal{D}_{lex}$. Fig. 5 is the result derived from Fig. 1(a). As illustrated, a speculation is triggered by an accepting state. The transformation only concentrates on transitions, whose current states are accepting states, i.e., $q_2^*$, $q_4^*$, $q_5^*$ and $q_6^*$ (Line 2). For any accepting state $q^*$ and input symbol $i$ (Line 2-3);

**Algorithm 1** Speculation Embedding

```
Notation: $\mathcal{S}_1 \setminus \mathcal{S}_2$        ▷ Remove $e$ from $\mathcal{S}_1$ (if $e \in \mathcal{S}_2$)
Require: $\mathcal{D}_{lex} = (Q, \sum, \tau, q_1, \mathcal{A})$
Ensure: $\mathcal{N}_{prescan} = (Q, \sum, \mathcal{T}, q_1, \mathcal{A})$
 1: $\mathcal{T} \leftarrow \tau$
 2: for all $q^* \in \mathcal{A}$ do
 3:     for all $i \in \sum$ do
 4:         if $\tau(q^*, i) = q_0$
 5:             $\mathcal{T} \leftarrow \mathcal{T} \setminus \{q^* \xrightarrow{i} q_0\}$        ▷ Removal
 6:             $\mathcal{T} \leftarrow \mathcal{T} \cup \{q^* \xrightarrow{i} \tau(q_1, i)\}$
 7:         else if $\tau(q^*, i) \notin \mathcal{A}$
 8:             $\mathcal{T} \leftarrow \mathcal{T} \cup \{q^* \xrightarrow{i} \tau(q_1, i)\}$
 9:     end for
10: end for
```

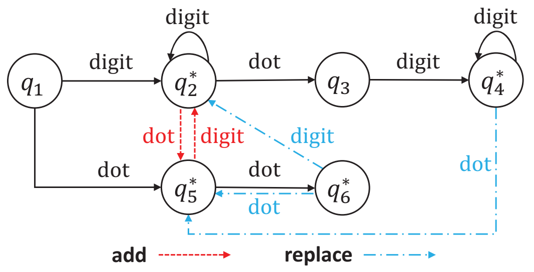


Figure 5: Result of Speculation Embedding ($\mathcal{N}_{prescan}$)

- **`replace`** (Line 4-6): if $\tau(q^*, i){=}q_0$, replace $q^*\xrightarrow{i}q_0$ with $q^*\xrightarrow{i}\tau(q_1, i)$. e.g., since $\tau(q_6^*, digit){=}q_0$, replace $q_6^*\xrightarrow{digit}q_0$ with $q_6^*\xrightarrow{digit}q_2^*$.
- **`add`** (Line 7-8): if $\tau(q^*, i){\notin}\mathcal{A}$ and $\tau(q^*, i){\neq}q_0$, add a new transition $q^*\xrightarrow{i}\tau(q_1, i)$. e.g., because $\tau(q_2^*, dot){=}q_3$, add $q_2^*\xrightarrow{dot}q_5^*$.

The condition $\tau(q^*, i) \in \mathcal{A}$ is not taken into account, since speculation failures can be found immediately under continuous accepting states, as the $q_2^*\xrightarrow{0}q_2^*$ shown in Fig. 4(a).

Behavior `replace` avoids short backtrackings from position $p{+}1$ to $p$ (i.e., $q_0$ is reached right after an accepting state). For example, when matching the `ELLIPSIS` in Fig. 3, $q_6^*\xrightarrow{8}q_0$, $q_0\xrightarrow{backtrack}q_1$ and $q_1\xrightarrow{8}q_2^*$ are contracted to $q_6^*\xrightarrow{8}q_2^*$ to eliminate the repetition over '8'. Behavior `add` enables speculation by building corresponding transitions. As shown in Fig. 4(b), when consuming the first dot: (1) $q_2^*\xrightarrow{.}q_3$ assumes '1.' is the prefix of a `FLOAT`; (2) $q_2^*\xrightarrow{.}q_5^*$ speculates '1' is an `INT`, while '.' starts another lexeme (e.g., a `DOT` or `ELLIPSIS`).

**Lemma 1.** *For any state $q$ and input symbol $i$ in $\mathcal{N}_{prescan}$, $|\mathcal{T}(q, i)|$ is two at most, since only the behavior* `add` *increases out-degree, by adding one speculated transition.*

The speculated transition is correct only if the original path goes to invalid matching immediately after speculating (after triggering a speculation, it does not reach another accepting state before the invalid state $q_0$); otherwise, speculation fails and should be canceled. However, due to basic automaton theory, transitions are unprioritized. As such, it is uncertain

**Algorithm 2** Ordered Subset Construction (ordered-SC)

```
Notation: $S_{[n]}, O_{[n]}$          ▹ $n$-th element of ordered set $S$ and $O$
          $O^\frown S$                ▹ append $e$ to $O$ (if $e \in S$ and $e \notin O$)
Require: $\mathcal{D}_{lex} = (Q, \sum, \tau, q_1, \mathcal{A})$, $\mathcal{N}_{prescan} = (Q, \sum, \mathcal{T}, q_1, \mathcal{A})$
Ensure: $\mathcal{D}_{prescan} = (Q', \sum, \tau', [q_1], \mathcal{A}')$
 1: $Q' = \emptyset$, $Q_{tmp} = \{[q_1]\}$, $\tau' = \emptyset$, $\mathcal{A}' = \emptyset$
 2: while $Q_{tmp} \neq \emptyset$ & $S \in Q_{tmp}$ do                ▹ Choose any
 3:     $Q_{tmp} \leftarrow Q_{tmp} \setminus \{S\}$,   $Q' \leftarrow Q' \cup \{S\}$
 4:     for all $i \in \sum$ do
 5:         $O = [\ ]$                                      ▹ New ordered set
 6:         for $n = 1 \rightarrow |S|$ do
 7:             $O \leftarrow O^\frown(\{\tau(S_{[n]}, i)\} \setminus \{q_0\})$        ▹ Original path
 8:             $O \leftarrow O^\frown(\mathcal{T}(S_{[n]}, i) \setminus \{q_0\})$      ▹ Speculated path
 9:             if $O_{[|O|]} \in \mathcal{A}$
10:                 $O \leftarrow O^\frown\{q_1\}$                    ▹ Speculation
11:                 break                                   ▹ Cancellation
12:         end for
13:         $Q_{tmp} \leftarrow Q_{tmp} \cup \{O\}$
14:         if $\exists e \in O$, $e \in \mathcal{A}$ then $\mathcal{A}' \leftarrow \mathcal{A}' \cup \{O\}$
15:         $\tau' \leftarrow \tau' \cup \{S \xrightarrow{i} O\}$
16:     end for
17: end while
```

| Rename | Subset | digit | dot | # |
|---|---|---|---|---|
| $O_1$ | $[q_1]$ | $[q_2^*,q_1]$ | $[q_5^*,q_1]$ | [] |
| $O_2^*$ | $[q_2^*,q_1]$ | $[q_2^*,q_1]$ | $[q_3,q_5^*,q_1]$ | [] |
| $O_3^*$ | $[q_5^*,q_1]$ | $[q_2^*,q_1]$ | $[q_6^*,q_1]$ | [] |
| $O_4$ | [] | [] | [] | [] |
| $O_5^*$ | $[q_3,q_5^*,q_1]$ | $[q_4^*,q_1]$ | $[q_6^*,q_1]$ | [] |
| $O_6^*$ | $[q_6^*,q_1]$ | $[q_2^*,q_1]$ | $[q_5^*,q_1]$ | [] |
| $O_7^*$ | $[q_4^*, q_1]$ | $[q_4^*,q_1]$ | $[q_5^*,q_1]$ | [] |

(a) Construction of $\tau'$

| $\tau'$ | digit | dot | # |
|---|---|---|---|
| $O_1$ | $O_2^*$ | $O_3^*$ | $O_4$ |
| $O_2^*$ | $O_2^*$ | $O_5^*$ | $O_4$ |
| $O_3^*$ | $O_2^*$ | $O_6^*$ | $O_4$ |
| $O_4$ | $O_4$ | $O_4$ | $O_4$ |
| $O_5^*$ | $O_7^*$ | $O_6^*$ | $O_4$ |
| $O_6^*$ | $O_2^*$ | $O_3^*$ | $O_4$ |
| $O_7^*$ | $O_7^*$ | $O_3^*$ | $O_4$ |

(b) Renamed $\tau'$

Figure 6: Result of ordered-SC ($\mathcal{D}_{prescan}$)

that which transition represents speculation. Therefore, the cancellation procedure cannot be correctly guaranteed. In other words, the transitions must be examined in a specific order to obey maximal munch, for which we propose *ordered subset construction* to guarantee the cancellation procedure.

### *D. Ordered Subset Construction ($\mathcal{N}_{prescan} \rightarrow \mathcal{D}_{prescan}$)*

Based on above discussion, the cancellation procedure can be formalized as: *for any speculation path $\mathcal{P}_j$ created by the reached accepting state on path $\mathcal{P}_i$, if $\mathcal{P}_i$ reaches an accepting state again before the invalid state $q_0$, $\mathcal{P}_j$ should be canceled.*

**Lemma 2.** *Recursive Cancellation: suppose $\mathcal{P}_i$ speculates $\mathcal{P}_j$; if $\mathcal{P}_i$ is canceled, then $\mathcal{P}_j$ should be canceled, too.*

For example, Fig. 4(b) shows two speculation paths created due to reached accepting states ($q_2^*$ after '10' and $q_5^*$ after '.' by speculating they are `INT` and `DOT`, respectively). However, both of them should be canceled after consuming '7', since the original path reaches another accepting state $q_4^*$ again. We show that this can be guaranteed in a simple way with the proposal — *ordered subset construction (ordered-SC)*. Specifically, each subset contains several ordered states, which corresponds to a column of states at an input position, as Fig. 4. Comparing to *subset construction* (SC) and existing work [24] constructing a deterministic equivalent for any $\mathcal{N}$, ordered-SC imposes more restrictions during subsets enumeration. Therefore, the resulting automaton (i.e., $\mathcal{D}_{prescan}$) may be a deterministic nonequivalent. As the name implies, in each resulting subset, a distinct feature is the *state-order-sensitivity*. Algorithm 2 shows the procedure, along with an example result shown in Fig. 6. Fig. 6(a) demonstrates construction procedure, and Fig. 6(b) is to facilitate readers. As a distinguishment to common automaton theory, notation $O_i$ is used to emphasize its key feature — order-sensitivity.

Similar to SC, the ordered-SC initializes a subset $[q_1]$ only containing the initial state $q_1$ (Line 1). From an existing subset $S$ (Line 2) and any input symbol $i$ (Line 4), it enumerates a new subset $O$ involving the next states of the current states in $S$. The property state-order-sensitivity indicates $O$ holds the same order of $S$, with two implicit sub-rules;

***Speculation-order*** **(Line 7-8):** For any state $q$ and input symbol $i$, the next state $\tau(q, i)$ originated from $\mathcal{D}_{lex}$ is appended before the speculated one. As in Fig. 6(a), $q_3$ is appended before $q_5^*$ in $\tau'([q_2^*, q_1], dot)$=$[q_3, q_5^*, q_1]$, since $q_2^* \xrightarrow{dot} q_3$ is originated from $\mathcal{D}_{lex}$, while $q_2^* \xrightarrow{dot} q_5^*$ is a speculation. In addition, due to Lemma 1, each is one at most. By keeping this order, the speculation chain can tracked in a simple way that *"in any subset, j-th state relies on i-th state, for any i < j"*.

***Cancellation-order*****:** For any two states $S_{[i]}$ and $S_{[j]}$ in $S$, if $i < j$, the next states of $S_{[i]}$ are appended prior to those of $S_{[j]}$ (Line 6). For instance, when enumerating $\tau'([q_3, q_5^*, q_1], digit)$, it first takes the next states of $q_3$, which are $[q_4^*]$ containing an accepting state. Due to Theorem 2, speculation should be canceled recursively. Therefore, the other two states $q_5^*$ and $q_1$ are speculation failures and should be canceled. This can be implemented by simply stopping the enumeration (Line 11).

The invalid state $q_0$ is excluded in an ordered subset (Line 7-8). Therefore, any $\mathcal{D}_{prescan}$ may contain an empty set (e.g., $O_4$) that can be used to recognize invalid input. Comparing to SC, which enumerates all next states disorderly (such as $[q_3, q_5^*, q_1] \xrightarrow{digit} [q_2^*, q_4^*]$), ordered-SC obeys the maximal munch principle through state-order-sensitivity. Recall that, $\mathcal{D}_{prescan}$ is used to collect context information. It builds transfer functions, while does not need to make choice to emit lexemes. As a result, the situation in Fig. 4 can be represented as follows;

Fig. 4(a): $O_1 \xrightarrow{1} O_2^* \xrightarrow{0} O_2^* \xrightarrow{.} O_5^* \xrightarrow{.} O_6^* \xrightarrow{8} O_2^* \xrightarrow{9} O_2^* \xrightarrow{\#} O_4$

Fig. 4(b): $O_1 \xrightarrow{1} O_2^* \xrightarrow{0} O_2^* \xrightarrow{.} O_5^* \xrightarrow{7} O_7^* \xrightarrow{8} O_7^* \xrightarrow{9} O_7^* \xrightarrow{\#} O_4$

where, each subset corresponds to a column of states at a specific input position. Besides, the order of states is ensured, too. In addition, it is backtrack-free for removing duplicate character consumption. Therefore, the computation over $\mathcal{D}_{prescan}$ is the same as that of DFA shown in Fig. 1(c).

## IV. Parallel Scanner

This section introduces the two-passes run (prescanning and scanning) over the input sequence. The prescanner drives

**Algorithm 3** Directive Backtracking (DB)

```
Require: $\mathcal{D}_{lex} = (Q, \sum, \tau, q_1, \mathcal{A})$
         $input = c_1c_2...c_n$, position ($p_{start}$,$p_{end}$)
Ensure: $p_{first}$          ▷ the starting position of the first lexeme
 1: procedure DIRECTIVE BACKTRACKING(ordered subset $O$)
 2:     $p_{first} \leftarrow 0$
 3:     for $e = 1 \rightarrow |O|$ do          ▷ Traverse subset orderly
 4:         $s \leftarrow O_{[e]}$, $p_{first} \leftarrow 0$
 5:         for $i = p_{start} \rightarrow n$ do
 6:             if $s \in \mathcal{A}$ then $p_{first} \leftarrow (i-1)$
 7:             $s \leftarrow \tau(s, c_i)$
 8:             if $s = q_0$ then
 9:                 if $p_{first} = 0$ then break          ▷ Try next
10:                 else goto Line 13          ▷ Found
11:         end for
12:     end for
13:     if $p_{first} < p_{end}$ then Scanning($p_{first}$)
14: end procedure
```

**Algorithm 4** Scanning

```
Require: $\mathcal{D}_{lex} = (Q, \sum, \tau, q_1, \mathcal{A})$
         $input = c_1c_2...c_n$, position ($p_{start}$, $p_{end}$)
Ensure: $List_{lexeme}$
 1: procedure SCANNING(position $p_{first}$)
 2:     $state \leftarrow q_1$, $s \leftarrow p_{first}$, $e \leftarrow p_{first}$, $List_{lexeme}$ = [ ]
 3:     for $i = p_{first} \rightarrow n$ do
 4:         if $state \in \mathcal{A}$ then $e \leftarrow (i-1)$
 5:         $state \leftarrow \tau(state, c_i)$
 6:         if $state = q_0$ then
 7:             append lexeme $(c_s...c_e)$ to $List_{lexeme}$
 8:             if $c_e > p_{end}$ then return          ▷ exceed boundary
 9:             $i \leftarrow e$, $s \leftarrow e$
10:     end for
11: end procedure
```

$\mathcal{D}_{prescan}$ over the input to collect context information, which involves contexts at each splitting point. With the context, every chunk can be scanned independently by an instance of the scanner. In addition, we describe a data-parallel implementation for accelerating prescanning.

### *A. Parallel Prescanning*

Above discussion reveals two properties of $\mathcal{D}_{prescan}$; First, It is backtrack-free, since the invalid and valid matchings are contracted by speculation. Comparing to the worst time complexity of scanning, i.e., $O(n^2)$, the computation over $\mathcal{D}_{prescan}$ is a single-pass, i.e., $O(n)$. Second, it does not need runtime checks, since the prescanner does not make choices when reaching any accepting state and the invalid state $q_0$.

Therefore, prescanning can be done in a fast way through common parallel techniques, e.g., SpecSim. As shown in Fig. 7, the input is split into three chunks, and the two discussed cases are involved: first, the boundary of $\mathbb{C}_1$ and $\mathbb{C}_2$ coincides with that of `INT`(10) and `ELLIPSIS`(..); second, the boundary of $\mathbb{C}_2$ and $\mathbb{C}_3$ breaks the `ELLIPSIS`(..). Each chunk is prescanned independently, after which the reduction phase

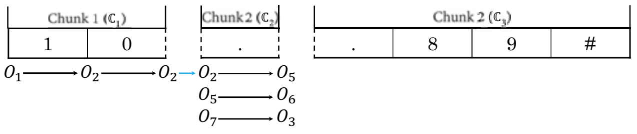
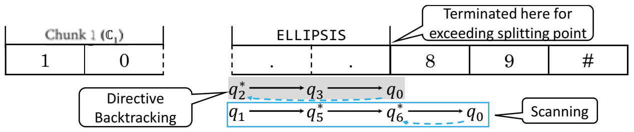
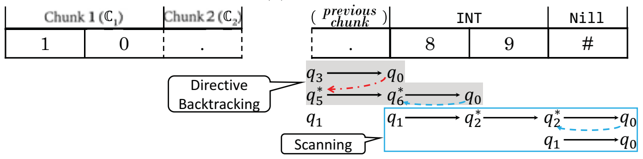


Figure 7: Parallel Prescanning



Terminated here for exceeding splitting point
Chunk 1 ($\mathbb{C}_1$)
ELLIPSIS
1
0
.
.
8
9
#
Directive Backtracking
$q_2^* \rightarrow q_3 \rightarrow q_0$
$q_1 \rightarrow q_5^* \rightarrow q_6^* \rightarrow q_0$
Scanning

(a) Chunk II



(b) Chunk III

Figure 8: Directive Backtracking & Parallel Scanning

reveals that the start subsets of $\mathbb{C}_2$ and $\mathbb{C}_3$ are $O_2 = [q_2^*, q_1]$ and $O_5 = [q_3, q_5^*, q_1]$, respectively.

### *B. Directive Backtracking & Parallel Scanning*

For each chunk, the prescanner outputs an ordered subset, which involves context information before the chunk's starting position. For example, known from $\mathbb{C}_1$ and $\mathbb{C}_2$, $O_5$ of $\mathbb{C}_3$ is a context implying the three possibilities discussed in Section III-A. Obeying the maximal munch principle, the only correct one can be found by examining Chunk III itself (termed as *directive backtracking*, DB). Fig. 8 shows how $\mathbb{C}_2$ and $\mathbb{C}_3$ are scanned in parallel; Algorithm 3 and 4 show the procedures of DB and scanning, respectively. Each chunk has two parameters ($p_{start}$,$p_{end}$) specifying its boundary, and an ordered subset $O$ obtained from prescanner. For each thread assigned to a particular chunk, all the input after $p_{start}$ is accessible, and the parameter $p_{end}$ is used to bound the beginning of the chunk's last lexeme. In other words, any chunk $\mathbb{C}_j$ is accessible for the thread processing chunk $\mathbb{C}_i$ ($i \leq j$), which is necessary if a lexeme spans several chunks. This is natural if loading into continuous addresses of shared-memory.

Algorithm 3 — the aim is to obtain correct state from subset $O$, even though the backtracking position may be unknown when it crosses chunks. Otherwise, an invalid matching would have to backtrack to an unknown position, as the path starting with $q_3$ in Fig. 8(b). States in $O$ are examined orderly until the only correct one is found (Line 5-11). The position of the last reached accepting state is memorized to discover the end of a lexeme belonging to one of the previous chunks (Line 6). No matter whether a backtracking crosses chunks or not, the next state in $O$ always indicates the correct one at the splitting point (Line 9). The examination will be terminated once the first lexeme after $p_{start}$ is found (Line 10), after which scanning starts, unless the whole chunk is part of a long lexeme (Line 13). The algorithm obeys the maximal munch principle

and avoids generating incorrect results. As in Fig. 8(b), given $O_5 = [q_3, q_5^*, q_1]$ for $\mathbb{C}_3$, $q_3$ implies that it is within the context of a `FLOAT` with prefix '10.'. Upon failure at the next dot, it examines the second state $q_5^*$, which is turned out to be correct for reaching an accepting state $q_6^*$ after reading the next dot. Thus, the third state $q_1$ is skipped. On the correct path, the last accepting state $q_6^*$ implies that the dot is the suffix of a lexeme belonging to one of previous chunk, i.e., $\mathbb{C}_2$ in this case. Therefore, the start position of this chunk is found. As a result, each chunk can be scanned independently.

Algorithm 4 — scanning begins from $p_{first}$ obtained from the DB phase, i.e., the start position of a chunk's first lexeme (Line 3). It is similar to the sequential case with a different stopping condition (Line 8: if the end position of a lexeme exceeds the chunk's $p_{end}$). As in Fig. 8(a), scanning stops at the second dot after emitting `ELLIPSIS`. As a result, for a lexeme spanning chunks $\mathbb{C}_i$ to $\mathbb{C}_j$ ($i < j$), it will only be matched by the thread processing $\mathbb{C}_i$, and skipped by others.

## *C. Data-parallel Implementation*

As an extra phase, the prescanning overhead is critical, for which a data-parallel approach is applied to execute multiple transitions simultaneously, using SIMD instructions. For the $\mathbb{C}_2$ in Fig. 7, it is similar to contract the three transitions to a vector-like transition $[O_2, O_5, O_7] \xrightarrow{.} [O_5, O_6, O_3]$. This paper assumes a SIMD vector has four cells, or length $L = 4$.

For any input symbol $i$, the $\tau(q, i)$ can be represented by vectors. As shown in Listing 2, $\tau(q, dot)$ is loaded into two vectors $v_1$ and $v_2$, where the last 0 of $v_2$ is a padding (Line 1). Similarly, $s_c$ is used to store the speculated states (Line 2).

Unlike existing work [14] utilizing instructions $\mathtt{I}_{shuffle}$ and $\mathtt{I}_{blend}$, we propose an alternative achieving higher performance through a different set of instructions, for their lower latency and cycles-per-Instruction (CPI) [25]. Specifically, they are three wordwise operations $\mathtt{I}_{shuffle}$, $\mathtt{I}_{cmplt}$, $\mathtt{I}_{sub}$, and two bitwise ones $\mathtt{I}_{and}$, $\mathtt{I}_{or}$. Table II shows the details of the wordwise ones. Listing 2 is an example describing the procedure.

$\mathtt{I}_{shuffle}(s,v)$ shuffles cells in $v$ according to *masks* stored in $s$, with two special cases. First, if $s[i] \geq L$, the resulting cell is that at the corresponding offset. For instance (Line 4), mask $s_c[1]=5$ represents offset 1 by taking the remainder of $5\%L$, and thus $n_1[1]=\tau(5\%4, dot)$. Second, if $s[i]<0$, it is taken as an invalid mask by returning 0 as the result. For example (Line 8), the 1st, 4th cells of $\mathtt{I}_{shuffle}([-2, 1, 3, -4], v_2)$ are 0's. The first case makes $\mathtt{I}_{shuffle}$ return a wrong vector if there is any $s[i] \geq L$. The correct next state of such mask is stored in another vector, e.g., that of state $s_c[1]=5$ is the second cell of $v_2$, since $5\%L = 1$.

$\mathtt{I}_{cmplt}(s,base)$ compares corresponding cells between $s$ and $base$ by $s[i]<base[i]$, and returns $-1/0$ (i.e., $0b11...1/0b00...0$) if true/false. The procedure is as follows;

- **(Line 4-6)** $\mathtt{I}_{cmplt}$ recognizes the cells, whose states are in the correct range ($< L$); along with $\mathtt{I}_{and}$ extracting correct cells from $n_1$, while setting others to 0's.
- **(Line 7-8)** $\mathtt{I}_{sub}$ subtracts $s_c$ by a vector storing $L$'s, so that states in range $[0, L-1]$ become negative; while others are set into $[0, L-1]$. Consequently, another $\mathtt{I}_{shuffle}$ is able to shuffle states from the second transition function vector $v_2$.
- **(Line 9)** Finally, $\mathtt{I}_{or}$ mixes $n_1$ and $n_2$ — the results extracted from the first and second $\mathtt{I}_{shuffle}$'s.

Table II: Details about Wordwise Instructions

| Instruction | Result of $i$-th cell | Example in Listing 2 |
|---|---|---|
| $\mathtt{I}_{shuffle}(s,v)$ | $\begin{cases} 0 & (s[i] < 0) \\ v[s[i]\%L] & (s[i] \geq 0) \end{cases}$ | Line 4 & 8 |
| $\mathtt{I}_{cmplt}(s,base)$ | $\begin{cases} -1 & (s[i] < base[i]) \\ 0 & (s[i] \geq base[i]) \end{cases}$ | Line 5 |
| $\mathtt{I}_{sub}(s,l)$ | $s[i] - l[i]$ | Line 7 |

```
1  $v_1$  =  $[0,5,3,0]$ ,  $v_2$  =  $[0,6,0,0]$ ,  $L$ = 4
2  $s_c$  =  $[2,5,7,0]$                              # $[O_2, O_5, O_7]$
3
4  $n_1$  =  $\mathtt{I}_{shuffle}(s_c, v_1)$          → $[\ 3,5,0,\ 0]$
5  $c$    =  $\mathtt{I}_{cmplt}(s_c, [L,L,L,L])$     → $[-1,0,0,-1]$
6  $n_1$  =  $\mathtt{I}_{and}(n_1, c)$               → $[\ 3,0,0,\ 0]$
7  $s_c$  =  $\mathtt{I}_{sub}(s_c, [L,L,L,L])$       → $[-2,1,3,-4]$
8  $n_2$  =  $\mathtt{I}_{shuffle}(s_c, v_2)$          → $[\ 0,6,0,\ 0]$
9  $s_c$  =  $\mathtt{I}_{or}(n_1, n_2)$               → $[\ 3,6,0,\ 0]$
```

Listing 2: Data-parallel Implementation

A repetition of Line 5-9 compensates arbitrary amount of $v_i$'s (>2), which remains a drawback that its efficiency relates to the vector's amount (consequently, that of states). An alternative is $\mathtt{I}_{gather}$ [16] having larger latency and CPI, but could be more efficient if the amount of $v_i$'s exceeds an experimental threshold. Plex adopts the $\mathtt{I}_{shuffle}$-based approach for observing that the number of states is not large in practice.

This data-parallel design benefits from the *backtrack-free* $\mathcal{D}_{prescan}$, since $\mathtt{I}_{shuffle}$ takes the transition function vector related to a specific character. Backtracking would break this rule, if arbitrary path (i.e., a cell) backtracks to consume a different character, and others do not.

## V. Evaluation

This section evaluates Plex through real-world applications. The main purpose is to show that 1) the overhead brought by prescanning is tolerable; 2) Plex achieves considerable scalability. Specifically, we show the speedup of Plex over sequential scanning, along with comparison with existing lexer generators and language-specific lexers.

Plex comprises two components — *generator* and *runtime*. By analyzing user-defined grammars, the generator emits corresponding codes to be compiled with the runtime to build a complete parallel lexer. A critical issue is workload balancing, since the total distance of backtracking and the number of emitted lexemes vary from chunk to chunk. Therefore, Plex adopts *Massivethreads* [26] — a lightweight thread library, for task parallel implementation. Table III shows the experimental setup. In terms of data-parallel prescanning, Plex exploits SSSE3 for observing a worse performance under AVX2.

Taking advantage of the flexibility as a lexer generator, three languages listed in Table IV are used. In $\mathcal{D}_{prescan}$'s,

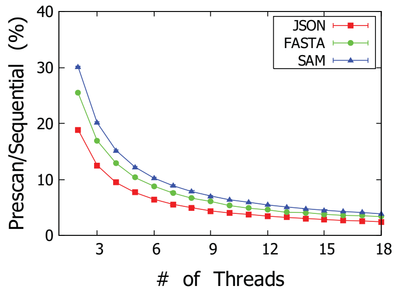


Figure 9: Prescan Overhead

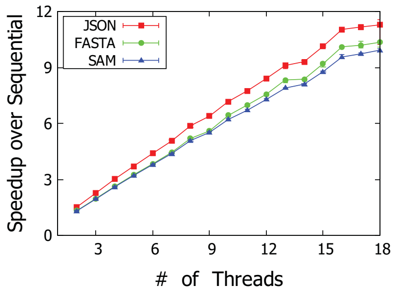


Figure 10: Speedups

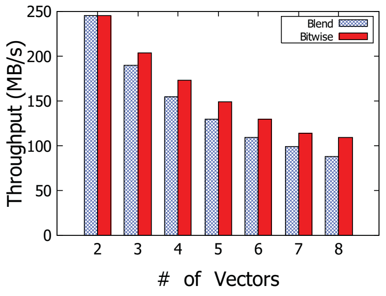


Figure 11: Parallel Table Lookup

Table III: Experimental Setup

| | |
|---|---|
| **CPU** | Intel Xeon E5-2695v4 Broadwell-EP 2.10 GHz (18 cores) |
| **Compiler** | GCC 9.2.0 (with "`-flto -O3`" optimization flags) |
| **Library** | MassiveThreads v1.00 (for task parallel implementation) |

Table IV: Description of Languages (★: Grammar Details)

| | |
|---|---|
| JSON | A popular lightweight format, widely used in data-exchange. |
| FASTA | Common in bioinfomatics for storing nucleotide sequences and amino acids, each of which is preceded by a header line beginning with a ">" to give identifier and additional information. ★ *The header line is specified with regex:* '`>[0-9]+`' *for observing that the datasets simply use numbers as identifiers.* |
| SAM | A tab-delimited format for storing biological sequences [27]. ★ *The quantifier specifications (e.g.,* '`{1,254}`'*) are disabled.* |

the numbers of states of JSON, FASTA and SAM are 65, 7 and 72, respectively. Input files, whose sizes range from hundreds of megabytes to several gigabyte, are collected from DATA.GOV and AWS S3 Buckets. Except for explicitly noted, all the results exclude the time spent in loading data from disk.

## A. Prescanning Overheads and Speedups

The first evaluation shows the performance of prescanning to weight its overheads, with which we discuss the speedups of Plex over sequential scanning. Fig. 9 shows the ratio of the elapsed time of parallel prescanning to that of sequential scanning, related to the number of threads. The ratio ranges from 18% to around 30% when using two threads. However, with the increase of available threads, it keeps reducing and reaches 2%-4% when using 18 threads.

Fig. 10 shows the speedups: it is proportional to the number of threads and reaches $9.8X$–$11.5X$ with 18 threads. However, the slopes are less than 0.65, since the ratio of *prescanning*:*scanning* could be large when both are executed in parallel. Based on our measurement, the ratios of JSON, FASTA and SAM are 39.92, 51.85 and 64.85%, respectively. It is varied by (1) the amount of $\mathcal{D}_{prescan}$'s states; (2) the total distance of backtracking; (3) the number of emitted lexemes. For example, the JSON's is smaller than the FASTA's, since JSON texts contain lots of short numeric lexemes, while FASTA texts usually contain long nucleotide sequences. Therefore, the scanning phase of JSON accounts for a larger proportion in the total elapsed time. On the other hand, the SAM's is larger for having relatively more states in $\mathcal{D}_{prescan}$, thus prescanning for SAM is more expensive. This indicates the reason why parallel JSON lexing achieves higher speedups over others.

## B. Prescanning Efficiency under Parallel Table Lookup

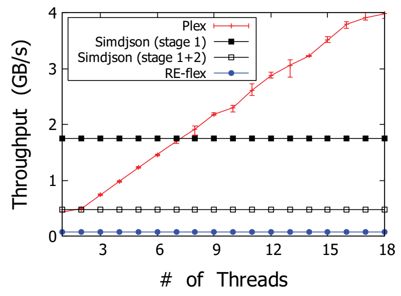
Section IV-C discusses the data-parallel implementation for accelerating prescanning performance. Differing from $\mathtt{I}_{blend}$ [14], we use a different set of instructions to mix the results of several $\mathtt{I}_{shuffle}$'s. Their single thread performances, and the effect brought by the amount of $v_i$'s are shown in Fig. 11. Our proposal is named as *Bitwise* for using bitwise operations $\mathtt{I}_{and}$ and $\mathtt{I}_{or}$. Apparently, *Bitwise* is superior for exploiting instruction-level parallelism by using instructions having lower CPI and latency. However, the performance of both become terrible with the increase of the amount of $v_i$'s (i.e., a large number of states), in which case an alternative is $\mathtt{I}_{gather}$ [16], although it forces to encode states with 32/64-bits.

## C. Comparison with Existing Tools

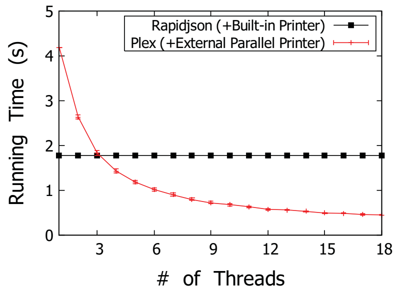
To show the usability of Plex in real world applications, we compare it with existing tools — a lexer generator called RE-flex [28] and other tools used for language-specific studies for JSON and SAM. For RE-flex, it is compiled with '`--fast`' with identical grammars as used in Plex. In addition, semantic actions are removed to ensure identical behavior. Note that, the prescanning phase of Plex is skipped when using one thread.

For JSON, we only count the number of lexemes in RE-flex. Another representative is *Simdjson* [22] — a JSON-specific parser making use of bitwise techniques and can be divided into two stages. Stage 1 builds bitmaps for the given input, and stage 2 recognizes lexeme types. Fig. 12(a) shows the throughput: the stage 1 of *Simdjson* achieves a high throughput around 1.75 GB/s, which is much higher to the other two when the number of threads is small. However, Plex outperforms it when using more than seven threads.

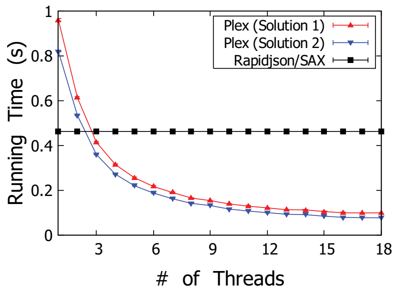
SAM is line-based and can be matched by many widely used tools, such as *grep*. In RE-flex, we counts the number of matched lines. In addition, data loading time is included, since *grep* is usually bounded by I/O performance. Fig. 12(b) shows the throughput: Plex outperforms *grep* when using more than two threads. However, the scalability is not proportional, since the data loading time could be significant when lexing takes less proportion with the increase of parallelism.



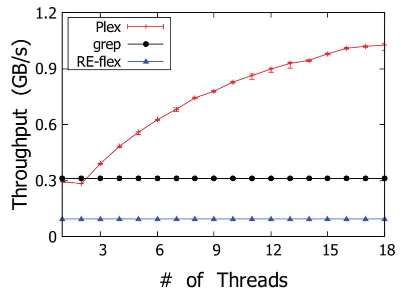
(a) JSON



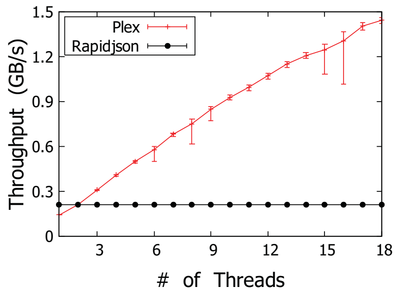
(a) Pretty Printer



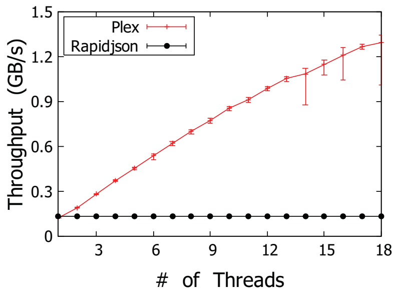
(b) Matching Counter



(b) SAM



(c) Rating Distribution



(d) Hottest Categories

Figure 12: Comparison

Figure 13: Applications

### D. Use Cases

Making use of Plex's output, users are capable of doing parallel processing. JSON is still used as the target for being supported by many tools that provides lots of use cases.

The first one is a parallel pretty printer. During scanning, a user-defined counter counts the amounts of structural lexemes — `LBrace({)`, `RBrace(})`, `LBracket([)` and `RBracket(])` of each chunk. Then, a thread processing chunk $C_i$ determines the level of indentation by calculating their amounts in chunks $C_1$ to $C_{i-1}$. We compare it with the built-in pretty printer of the fastest FA-based JSON parser — *Rapidjson* [29]. Fig. 13(a) shows the running time: with the increase of parallelism, Plex achieves remarkable performance with an external parallel pretty printer. Note that, *Rapidjson* validates both lexicon and syntax, while Plex only guarantees the former.

The second one is a widely used application — *matching counter* that counts the amount of key/value pairs if the values are `NUMBER`'s. Plex's flexibility provides two solutions for this task. First, users can simply use a common JSON grammar and make an external counter program to count the appearance of consecutive `STRING`; `COLON`; `NUMBER` from Plex's output. The second way is combining the patterns of them as a singleton (e.g., `COUNT := {STRING}{COLON}{NUMBER}`) in the grammar, and then trigger the counter by the appearance of each `COUNT`. This increases the amount of $\mathcal{D}_{prescan}$'s states to 130, but decreases that of lexemes. Besides, we use the *Rapidjson*/SAX API, which publishes lexemes as events sequentially and only guarantees the correctness of lexicon. Fig. 13(b) shows the running time: both of the Plex's implementations outperform Rapidjson/SAX with more than three threads, and the second solution of Plex performs better. The reason is that, even though Solution 2 increases the number of states in $\mathcal{D}_{prescan}$, it emits less lexemes that relates to memory bandwidth.

The third and forth use cases are two data extraction tasks, which make use of dataset from Yelp [30]. It is a collection of business reviews, each of which contains *rating of stars* and the *category of the business*, etc. The two data extraction tasks are *rating distribution* (summarizing the distribution of stars) and *hottest categories* (finding out the ten hottest business categories). Fig. 13(c) and 13(d) show the results by throughput, along with Rapidjson's DOM parser as a performance reference. Obviously, Plex can outperform Rapidjson if there are more than two/three available threads in the third/forth use case. One of the reasons is the introduced dataset that allows these two simple tasks to be completed with the correctness of lexicon, while the mechanism of Rapidjson's DOM parser force to fully parse the input.

## VI. Related Work

**Parallelization of Automata** attracts attentions for its broad applicability. Among proposals, a popular one is *speculative simulation* [10]–[13], whose performance is usually affected by the speculation technique. Making use of the convergence property of FA's, Youwei et al. proposed *convergence set enumeration* to improve speculation success rate, by organizing states into convergence sets using pre-training [31]. Zhijia et al. proposed *principled speculation* looking back an appropriate number of characters to narrow speculation scope with offline [32] or online [15] training. Ryoma et al. proposed *simultaneous finite automata* (SFA) [24], which represent a set of transfer functions found through speculative simulation. The idea of speculation embedding and Ordered-SC originates from SFA construction, whereas it deals with maximal munch in lexing rather than regular expression matching. In addition, data parallelism also show significant contributions. *Parallel table lookup* is helpful for executing several transitions simultaneously [14], [16]. *Parallel prefix computation* parallelizes FA as a random access machine [17], [33].

**Parallel Lexing** is a conventional topic, but re-aroused recently, due to data explosion. A parallel lexer is usually a language-specific design, since the divide-conquer strategy suffers from the difficulty in determining the context of a chunk [1]–[3]. Particularly, JSON [1], [6] and Lua [1], [3] are popular targets. In these designs, a language-specific lookhead strategy is usually inevitable. For example, splitting JSON/Lua at white-spaces/newlines [1]. Besides, data-parallel techniques also contribute significantly. Yinan, Langdale et al. proposed JSON-specific parsers *Mison* [6] and *Simdjson* using bitwise operations, [34]. Chandra et al. proposed *associative parallel lexing* (APL), which counts the lengths of delimiter-separated lexemes using byte-level shifting [8]. Lexing can be parallelized with parallel prefix computation [17], [35].

## VII. Conclusion

This paper presents a generator for parallel lexer — Plex, which is based on prescanning. Its prescanner guides the scanner by collecting context information, which is fast for backtrack-free and using data-parallel techniques. The cost of prescanning is one of our focus, for which we design a special automaton involving all backtracking cases through speculation. In addition, the prescanning phase is carried out in both data- and thread-level parallelism.

The evaluation results show that the prescanning overhead is tolerable with the increase of parallelism. Consequently, Plex accelerates lexing by up to $9.8X - 11.5X$ under 18 threads. In addition, as an automated tool, Plex's applicability is revealed through case studies under different kinds of languages.

## References

[1] A. Barenghi, S. Crespi Reghizzi, D. Mandrioli, F. Panella, and M. Pradella, "Parallel parsing made practical," *Science of Computer Programming*, vol. 112, pp. 195 – 226, 2015.

[2] R. Bernecky, "An spmd/simd parallel tokenizer for apl," in *Proceedings of the 2003 Conference on APL: Stretching the Mind*, ser. APL '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 21–32.

[3] A. Barenghi, E. Viviani, S. Crespi Reghizzi, D. Mandrioli, and M. Pradella, "Papageno: A parallel parser generator for operator precedence grammars," 01 2013, pp. 264–274.

[4] T. Mason and D. Brown, *Lex & Yacc*, 1990.

[5] J. Levine and L. John, *Flex & Bison*, 1st ed., 2009.

[6] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann, "Mison: A fast json parser for data analytics," in *The 43rd International Conference on Very Large Data Bases (VLDB 2017)*, June 2017.

[7] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia, "Filter before you parse: faster analytics on raw data with sparser," *Proceedings of the VLDB Endowment*, vol. 11, pp. 1576–1589, 07 2018.

[8] C. R. Asthagiri and J. L. Potter, "Associative parallel lexing," in *Proceedings Sixth International Parallel Processing Symposium*, 1992, pp. 466–469.

[9] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, p. 56–67, Oct. 2009.

[10] J. Holub and S. Štekr, "On parallel implementations of deterministic finite automata," vol. 5642, 07 2009, pp. 54–64.

[11] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Multi-byte regular expression matching with speculation," 09 2009, pp. 284–303.

[12] D. Luchaup, R. Smith, C. Estan, and S. Jha, "Speculative parallel pattern matching," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 2, pp. 438–451, 2011.

[13] Y. Ko, M. Jung, Y.-S. Han, and B. Burgstaller, "A speculative parallel dfa membership test for multicore, simd and cloud computing environments," *International Journal of Parallel Programming*, vol. 42, no. 3, p. 456–489, Aug 2013.

[14] T. Mytkowicz, M. Musuvathi, and W. Schulte, "Data-parallel finite-state machines," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 529–542.

[15] Z. Zhao and X. Shen, "On-the-fly principled speculation for fsm parallelization," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 619–630.

[16] P. Jiang and G. Agrawal, "Combining simd and many/multi-core parallelism for finite state machines with enumerative speculation," *SIGPLAN Not.*, vol. 52, no. 8, p. 179–191, Jan. 2017.

[17] W. D. Hillis and G. L. Steele, "Data parallel algorithms," *Commun. ACM*, vol. 29, no. 12, p. 1170–1183, Dec. 1986.

[18] R. G. G. Cattell, "Formalization and automatic derivation of code generators." Ph.D. dissertation, USA, 1978, aAI7815197.

[19] T. Reps, ""maximal-munch" tokenization in linear time," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 2, p. 259–273, Mar. 1998.

[20] W. Yang, C.-W. Tsay, and J.-T. Chan, "On the applicability of the longest-match rule in lexical analysis," *Computer Languages, Systems & Structures*, vol. 28, no. 3, pp. 273 – 288, 2002.

[21] R. Smith, C. Estan, and S. Jha, "Backtracking algorithmic complexity attacks against a nids," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006, pp. 89–98.

[22] G. Langdale and D. Lemire, "Parsing gigabytes of json per second," *The VLDB Journal*, vol. 28, no. 6, p. 941–960, Oct 2019.

[23] D. E. KNUTH, *BITWISE TRICKS AND TECHNIQUES*, ser. 1A. Addison-Wesley, 1968, vol. 4.

[24] R. Sinya, K. Matsuzaki, and M. Sassa, "Simultaneous finite automata: An efficient data-parallel model for regular expression matching," in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 220–229.

[25] Intel Intrinsics Guide, "https://software.intel.com/sites/landingpage/IntrinsicsGuide/," 2020.

[26] Massivethreads, "https://github.com/massivethreads/massivethreads."

[27] Sequence Alignment/Map Format Specification, "https://samtools.github.io/hts-specs/SAMv1.pdf," July 2020.

[28] Re-flex, "https://www.genivia.com/reflex.html."

[29] Rapidjson, "https://rapidjson.org/."

[30] Yelp Dataset, "https://www.kaggle.com/yelp-dataset/yelp-dataset."

[31] Y. Zhuo, J. Cheng, Q. Luo, J. Zhai, Y. Wang, Z. Luan, and X. Qian, "Cse: Parallel finite state machines with convergence set enumeration," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 29–41.

[32] Z. Zhao, B. Wu, and X. Shen, "Challenging the "embarrassingly sequential": Parallelizing finite state machine-based computations through principled speculation," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 543–558.

[33] R. E. LADNER and M. J. FISCHER, "Parallel prefix computation," in *Journal of the Association for Computing Machinery*, vol. 27, no. 4, 1980, pp. 831–838.

[34] G. Langdale and D. Lemire, "Parsing gigabytes of json per second," *The VLDB Journal*, vol. 28, no. 6, p. 941–960, Oct 2019.

[35] J. M. Hill, "Parallel lexical analysis and parsing on the amt distributed array processor," *Parallel Computing*, vol. 18, no. 6, pp. 699 – 714, 1992.