Abstract: It is important to handle large-scale data in text formats such as XML, JSON, and CSV because these data very often appear in data exchange. For these data, instead of data ingestion to databases, ad hoc data extraction is highly desirable. The main issue of ad hoc data extraction is to serve both the programmability to allow handling various types of data intuitively and the performance for large-scale data. To pursue it, we develop Centaurus, a dynamic parser generator library for parallel ad hoc data extraction. This paper presents the design and implementation of Centaurus. The experimental results on ad hoc data extraction have demonstrated that Centaurus outperformed fast dedicated parser libraries in C++ for XML and JSON, and achieved excellent scalability with actions implemented in Python.

Keywords: LL(*) grammars, parser generators, parallel processing, ad hoc data extraction, Python

1. Introduction

Data in text formats such as XML, JSON, and CSV are commonly used for data exchange. The processing of large-scale data in text formats has recently increased in demand because of the advancement in machine learning and data science. The standard approach to managing and handling data efficiently and scalably involves storing them into databases. Database systems such as BaseX *1 for XML and MongoDB *2 for JSON are suitable for the data models of these text formats and are well developed. However, database systems commonly require ingesting the whole of the input data. This ingesting is excessively costly when it comes to the processing of large-scale data in the context of data exchange because the data of concern typically consist of only a little part of the large-scale data. Moreover, the concerned part often varies depending on our purpose. Therefore, it is crucial to extract concerned parts from raw data in text formats without expensive ingesting, i.e., ad hoc data extraction.

In the context of databases, on one hand, the efficient processing of raw data in text formats, e.g., for parallel XML streaming [10], [17], JSON parsing [15], [18], [19], and key-value stores [25] has been studied. Although these approaches were quite efficient, they were specialized to specific data models and queries. As a result, the conversion of data models and the use of external functions are difficult or often impossible on these database systems. In other words, they are designed to trade off efficiency against adaptability, which are crucial for ad hoc data extraction in the real world. The adaptability (or integration) to programming languages widely used for data processing, such as Python and C/C++, is a practical desideratum.

On the other hand, a classic and well-developed approach to ad hoc text processing is the one based on regular expressions. Regular expressions are integrated into many practical programming languages (e.g., JavaScript) as a crucial component for text data processing. They are useful for converting various types of text data into data representations in host languages. Moreover, efficient regular expression libraries based on dynamic native code generation, such as PCRE-JIT *3, and on data-parallel processing [16] are available. However, they have a limitation in expressivity. Regular expressions cannot recognize nested structures in general and are therefore insufficient for processing semi-structured data such as XML and JSON.

In a nutshell, the main issue of ad hoc data extraction is to serve both performance and adaptability. We address this issue by applying parser generators to ad hoc data extraction.

In this work, we present Centaurus, a dynamic parser generator based on LL(*) grammars [20]. Centaurus is a library that, given grammars in EBNF and semantic actions in Python or C++, performs parallel data processing on the fly, as shown in Fig. 1. A parser becomes ready to run at runtime as soon as we pass it a

```
import Context from centaurus

ctx = Context('json.cgr')  # Generate a parser
ctx.attach(JsonListener())  # Attach actions
ctx.start()  # Start with two parallel workers
result = ctx.parse('data.json')  # Process data
ctx.stop()  # Stop subprocesses
```

Fig. 1 Data processing using Centaurus.

1 Graduate School of Information Science and Technology, The University of Tokyo, Bunkyo, Tokyo 113-8656, Japan
2 Center for Technology Innovation, Hitachi Ltd., Yokohama, Kanagawa 244-0817, Japan
3 sato.shigeyuki@mi.u-tokyo.ac.jp

*1 http://basex.org/
*2 https://www.mongodb.com/
*3 https://www.pcre.org/original/doc/html/pcrejit.html
grammar and actions. The details of parallel processing are hidden from users; it is sufficient for users to specify the number of parallel workers. Moreover, the generated code for syntactic analysis utilizes SIMD instructions. It thus brings high throughput without the user's burden. The grammar descriptions supported by Centaurus are of a subset of LL(*), which is strictly more powerful than regular expressions and able to implement pattern matching briefly on semi-structured data. Because actions are merely in Python or C++, Centaurus is highly adaptable to host languages. The experimental results on data extraction from semi-structured documents have demonstrated that processing in Python had excellent scalability and the generated code outperformed fast dedicated parsers.

This work is summarized as follows:

- We develop Centaurus, a dynamic parser generator library based on LL(*) grammars that facilitates parallel ad hoc data extraction in Python and C++. We present the design and implementation of Centaurus (Sections 3 and 4), which pursue both performance and adaptability.
- The experimental results on data extraction from semi-structured documents (Section 5) have demonstrated the efficacy of our design and implementation. The throughput of the Python versions scaled up to more than 40-fold with 72 processes and that of the C++ versions in serial settings was significantly better than that of fast dedicated parsers in C++.

2. Data Extraction using Centaurus

First, we present a simple example of how to use Centaurus. Figure 2 and Fig. 3 show an example of data extraction for JSON such that it counts the occurrences of null: e.g., with these actions and grammar, the ctx.parse method in Fig. 1 returns 2 for the following JSON data:

\[\{0, \{"spam": 2, "egg": null\}, null, "ham"\}\]

The methods of the JsonListener class defined in Fig. 3 correspond to the actions for nonterminals defined in the JSON grammar in Fig. 2: e.g., parseList corresponds to the action for List. The defaultact method denotes the default action. Each action takes a parsing context object ctx and returns a semantic value of the nonterminal. The ctx objects have APIs for extracting data managed by Centaurus. ctx behaves as a type of collection of children’s values and we can iterate those values via iterators, len(ctx) returns the number of those values, and ctx[i] returns the i-th one. ctx.read() returns the interval of input that the current nonterminal spans.

One characteristic of the actions of Centaurus is that their resultant values are always optional; None is the special value that means void, as with the function calls in Python. In Fig. 3, since the default action returns None, the values of the nonterminals Number, String, False, and True, which do not have the corresponding action methods, are void. This is why the action for Object tests the number of children’s values. Given no children value, the value of Object is also void. Similarly, when children’s values for DictEntry are not a key-value pair, the value of DictEntry is void.

In the null counting case as shown Fig. 3, it might be more intuitive to define the default action as to return 0 instead of void. However, the use of void is crucial for the efficiency of handling large-scale input data. If the default action returned 0, the parsing stack including semantic values could blow up in proportion to the input size and be full of ineffective values. Centaurus is thus designed for programming data extraction from large-scale data.

Centaurus shows its true worth where we use it together with ad hoc grammars for specific datasets, rather than with the predefined standard grammars (such as Fig. 2). For example, consider a JSON dataset of a list of dictionaries, which we call the input size and be full of ineffective values. Centaurus is thus designed for programming data extraction from large-scale data.

Centaurus shows its true worth where we use it together with ad hoc grammars for specific datasets, rather than with the predefined standard grammars (such as Fig. 2). For example, consider a JSON dataset of a list of dictionaries, which we call

```
class JsonListener:
    def parseObject(self, ctx):
        return ctx[0] if ctx else None
    def parseList(self, ctx):
        return sum(ctx)
    def parseDict(self, ctx):
        return sum(ctx)
    def parseDictEntry(self, ctx):
        return ctx[0] if ctx else None
    def parseNull(self, ctx):
        return 1
    def defaultact(self, ctx):
        return None
```

Fig. 3 Actions for counting nulls.

```python
class JsonListener:
    def parseObject(self, ctx):
        return ctx[0] if ctx else None
    def parseList(self, ctx):
        return sum(ctx)
    def parseDict(self, ctx):
        return sum(ctx)
    def parseDictEntry(self, ctx):
        return ctx[0] if ctx else None
    def parseNull(self, ctx):
        return 1
    def defaultact(self, ctx):
        return None
```

Fig. 4 Ad hoc grammar of record list in EBNF.

© 2020 Information Processing Society of Japan 725
import json

class RecordListListener:
    def parseRecordList(self, ctx):
        return list(ctx)
    def parseRecord(self, self, ctx):
        if ctx:
            return json.loads(ctx.read())
    def parseRecordIdEntry(self, self, ctx):
        if ctx:
            return True # any value other than None
    def parseRecordIdValue(self, self, ctx):
        if 0 < int(ctx.read()) < 1000:
            return True # any value other than None
    def default_act(self, self, ctx):
        pass # equivalent to return None

Fig. 5 Actions for filtering records by the id entries.

Grammar ::= (Declare | Include | Rule)+
Declare ::= grammar GrammarName ;
Include ::= include "FilePath" ;
Rule ::= Nonterminal : RhsExp ;
RhsExp ::= Nonterminal
   | Literal
   | RegExpLiteral
   | RhsExp RhsExp
   | RhsExp | RhsExp
   | RhsExp UnaryOp
   | ( RhsExp )
UnaryOp ::= ? | * | +
Literal ::= 'String'
RegExpLiteral ::= /RegExp/

Fig. 6 Syntax of grammar descriptions for Centaurus.

RegExp ::= Character
    | RegExp RegExp
    | RegExp | RegExp
    | RegExp RegExpUnaryOp
    | [ CharClassUnit+ ]
    | [ . CharClassUnit+ ]
    | ( RegExp )
RegExpUnaryOp ::= ? | * | + | ++ | +++
Character ::= Alphabet | EscapedChar |
CharClassUnit ::= RawCharacter | RawClassUnit
RawCharacter ::= Alphabet | SpecialChar | Space

Fig. 7 Syntax of regular expressions supported by Centaurus.

This parser of the ad hoc grammar substantially performs scraping. Although this example reparses scraped segments as JSON, in practical settings, we could use dedicated libraries for concerned applications. Therefore, Centaurus is quite adaptable to various scraping-based tasks.

Lastly, we summarize the syntax of grammar descriptions for Centaurus. As shown in Fig. 6 and Fig. 7, grammars are basically in EBNF and are able to contain regular expression terminals. Declare is for describing the name of a grammar and is currently not interpreted; Include is for including other grammar files, which is useful for describing ad hoc grammars. The order of production rules Rule does not matter, except that the first Rule defines the start symbol. Regular expressions supported by Centaurus are also plain except for *, +, and ++, which are SIMD-annotated versions of *, and +, to be explained in Section 4.2.

3. Design of Centaurus

3.1 LL(*) Grammars

We adopt LL(*) grammars\(^4\) for the foundation of Centaurus. The main reason of this choice is the balance between programmability and scalability.

In terms of programmability, thanks to their regular lookahead of input, LL(*) grammars naturally subsume regular expressions with lookahead, which is a practical extension supported by many languages and libraries. Parsers based on LL(*) grammars, i.e., recursive descent parsers, are known to be easy to describe and control actions, which is one of the major reasons why ANTLR has adopted LL parsing. Because we design Centaurus to support grammar-based programming, as described in Section 2, the ease of controlling actions is an especially important factor.

In terms of scalability, LL grammars are very friendly with streaming parsing so that their parsers can stream out parse trees as bracket sequences like XML on the fly. Streaming parsing is practically necessary for large-scale input handling and significantly facilitates parallel processing of subsequent tasks. Assuming parsers to stream out the bracket sequence of a parse tree, we can process its fragments (i.e., possibly unbalanced bracket sequences) independently and merge the partial results into one in a common manner of parallel XML processing (e.g., Refs. [11], [17]).

We do not consider LL(*) grammars to be the best both for ease of programming and for scalability because there are grammars that are easier to write [1], [7] and another allowing efficient parallel divide-and-conquer parsing [3]. However, we find an appropriate balance between them in LL(*) grammars.

LL(*) grammars were originally the foundation of ANTLR 3. However, we do not follow the design of ANTLR 3 entirely because of the difference of purposes: ANTLR is designed for the frontends of various programming languages with complicated syntax, whereas Centaurus is designed for scalable and efficient data extraction from the text of simple but ad hoc grammars. Therefore, Centaurus deals with a subset of LL(*) grammars with which ANTLR 3 can deal.

3.2 Attribute Grammars on EBNF

To assign semantic actions to context-free grammars is formalized as attribute grammars [13]. Since we assign actions to grammars, Centaurus is also based on attribute grammars. However, we have chosen a different approach from the standard one: to assign an action to a nonterminal instead of a production rule. Intuitively, we regard the union of rules for a nonterminal using the | operator as a kind of regular expression, and assign one action to it. In other words, we have adopted S-attributed grammars.

\(^4\) Precisely, the grammar class that Centaurus supports currently is LL-regular grammars [21] because it has not supported predicates yet.
There are two reasons for this design choice. First, we suppose that the main targets of input formats are XML and JSON, and that these data often form nearly list structures. List structures are easy to specify in EBNF, and it is simple and intuitive that the values of list elements form a list in actions. Second, we suppose that the main target of computations, particularly for large-scale input, is data extraction. In implementing extractors, the results of actions should be optional for the space efficiency of intermediate data structures (e.g., stacks), as described in Section 2. The repetition of nonterminals in EBNF naturally makes the number of children unpredictable; it involves the nature of optionality desired. Therefore, the actions for nonterminals in EBNF are reasonable.

Note that this style of action assignment does not degrade programmability. If we would like to distinguish different rules of a nonterminal, it is sufficient to define the production of each rule as a separate nonterminal. We consider that this design facilitates action assignments in host languages because a symbol associated with an action (i.e., function or method) is explicit in grammars.

3.3 Dynamic Generation of Scannerless Parsers

Our aim is ad hoc data extraction similar to the use of regular expression libraries. Regular expression libraries generate finite automata dynamically, and sometimes also generate the native code of matchers for performance. By respecting this design, we have chosen the dynamic native code generation of parsers for a given grammar. In grammar-based programming, we would like to design both grammars and actions together at once. Our design choice makes editing them easier and supports lightweight grammar-based programming.

When we design the dynamic generation of parsers, it is important to simplify the parser code to be generated. Generating complicated code is generally costly in terms of generation time, code space, and implementation efforts. The standard design of parsers is to separate lexing from parsing and to make parsers perform token-level parsing. From the viewpoint of dynamic generation, it necessitates two additional runtime data structures: lexer code and token sequences. If we implement locality-aware buffering for performance, it is also required to manage lexer states and parser states. Scannerless parsing avoids these complications and enables us to generate parsers of simple brief code. Therefore, we have chosen to generate scannerless parsers from ordinary token-level grammars.

3.4 Parallel Processing

Parallel processing by Centaurus consists of streaming parsing followed by parallel application of semantic actions. We design Centaurus to avoid performing syntactic analysis (i.e., parsing in a strict sense) in parallel. The main reason for this design is the inherent difficulty of parallel LL parsing [23]. In principle, to pursue efficient parallel parsing is to sacrifice the expressivity of grammars [3], [23]. Because we adopt LL(*) grammars for programmability, serial parsing is efficient and reasonable. As mentioned in Section 3.1, Centaurus performs streaming parsing to stream out the bracket sequence of a resultant parse tree and facilitates parallel processing of the subsequent part.

As seen in Section 2, Centaurus is designed to return only the semantic values associated with the root of a parse tree of input for space efficiency in data extraction from large-scale input. For this purpose, complete abstract syntax trees of input are unnecessary and the value-decorated views of trees are redundant. Therefore, Centaurus, assuming S-attributed grammars, performs semantic actions through parallel bottom-up reduction (as in Ref. [11]) of parse trees in the form of bracket sequences.

Figure 8 illustrates a concrete workflow of parallel processing. It forms a three-stage asynchronous pipeline: 1) streaming parsing, 2) parallel partial reduction, and 3) streaming reduction. The stage-1 parser generates the bracket sequence of a parse tree and passes a chunk of the output to a stage-2 reducer, i.e., parallel worker. Each stage-2 reducer reduces the matched brackets in a given chunk to values recursively, and passes a resultant sequence (more precisely, stack) of partially reduced brackets to the stage-3 reducer. The stage-3 reducer picks up a stage-2 result in the order of its position on input and completes reduction serially. These stage processors work asynchronously to pass one unit of resultant data after another to the next stage. Therefore, there is pipeline parallelism among different stages, whereas there is embarrassing parallelism among stage-2 reducers.

4. Implementation

4.1 Parser Generation

The parser generation of Centaurus basically follows the foundation of ANTLR 3 [20], whose parser generation is outlined as follows: 1) It constructs an augmented transition network (ATN) and a set of tokens from a grammar description. An ATN is the model of recursive descent parsers to be generated and consists of AT machines, each of which models a function parsing a specific nonterminal and is a nondeterministic finite automaton (NFA) for token sequences. Next, 2) it constructs a lookahead deterministic finite automaton (LDFA) for token sequences from the ATN and the set of tokens. Lastly, 3) it generates a parser from the ATN and the LDFA.

The main difference from the above is that Centaurus translates user-defined token-level grammars into character-level grammars for scannerless implementation by constructing a character-level ATN and LDFA. A character-level ATN, which we call a composite ATN (CATN), can be constructed through the product construction of the ATN and an automaton that models the set of tokens. Character-level LDFA are constructed from the CATN as with token-level LDFA. Centaurus finally generates a scannerless parser from the CATN and the character-level
LDFA.

Note that CATN construction does not always succeed because of character-level ambiguities, which are a common problem in scannerless parsing [1], [24]. We leave disambiguation filters for future work.

As with ANTLR, LDFA construction does not always succeed. **Centaurus** tries to construct LDFA by looking ahead nonterminals up to recursion depth two. Unlike ANTLR, **Centaurus** does not fall back to a backtracking mode even if LDFA construction fails. In such a case, it simply fails parser generation. This design is for the simplicity and efficiency of dynamically generated parsers.

The core part of code generation has been implemented in 800 lines of C++ source code using AsmJit*5. It is quite brief and straightforward as a result of choosing scannerless parsing. The building blocks of assembly code that constitute parsers are very close to those for DFA matchers. The only primary extension is the call/return of recursive functions for nonterminals. Bracket generation is embedded in the body of each recursive function in assembly. Only the memory allocation for output is implemented in C++ and invoked from the generated native code.

We have limited input characters that **Centaurus** parsers can handle, to single-byte ones both for the simplicity and efficiency.

### 4.2 SIMD Annotation

Our dynamic native code generator utilizes SIMD instructions for performance. A key instruction is PCMPISTRI for string manipulation, which enables string comparison per 128-bit (i.e., 16 bytes) width. In principle, we can utilize SIMD instructions extensively for character matching. However, although they serve a significant performance gain when matching many characters, they also incur a performance penalty when matching a single character (or a few bytes). An important point is to judge which part of character matching is worth using SIMD instructions.

We can predict the number of characters to be processed in matching literals. **Centaurus** by default decides to use SIMD instructions for literals of 8 or more characters. Meanwhile, we cannot predict it in matching repetition * expressions in general. Therefore, it is difficult for **Centaurus** to decide SIMDization for repetition.

To remedy this difficulty, we have designed a way to receive hints from users through the SIMD-annotated repetition operator *+. When specified, **Centaurus** SIMDizes repetition matching code depending on the user’s judgment if it detects the corresponding loop structure in LDFA.

### 4.3 Segregated Stack for Bracket Reduction

As illustrated in Fig. 8, **Centaurus** performs the reduction of bracket sequences that represent parse trees. Each bracket has a nonterminal and a position into input, so that a matched pair of start/end brackets represents a nonterminal and the interval of input that it spans. **Figure 9** formalizes brackets and reduced ones.

**Figure 10** formalizes the reduction of bracket sequences in **Centaurus**. A stage-1 parser generates the chunks of the bracket sequence of a parse tree, say, bs0, ..., bsN. Stage-2 reducers apply reduceChunk to all the chunks and yield the corresponding reduced bracket stacks, say, rs0 = reduceChunk(bs0). A stage-3 reducer reduces all the reduced bracket stacks with @ to a single value, i.e., \( rs_1 = [V(v_i)] \) where \( v_i \) denotes the final return value. Note that function reverse in \( @ \) exists for formalization and takes no cost practically in runtime because lists are implemented with arrays. Although the stage-3 reduction can run in parallel owing to the associativity of \( @ \), **Centaurus** performs it in a serial streaming manner for space efficiency and the simplicity of implementation.

A key point of implementing bracket reduction is reducePrefix, which reduces the shortest prefix of a given stack that represents a matched pair of brackets if it exists. We should implement it efficiently, considering the arity of brackets to be unbounded and possibly very large. For efficiency, we have two desiderata:

- Semantic values are to be passed to actions in constant time and be accessed without overhead;
- Finding a matched pair of brackets (i.e., shortest prefix pattern matching) is to be in constant time.

These are crucial for efficiently handling large-scale trees of unbounded degree like XML and JSON datasets, whereas usual parser generators (particularly for programming languages) do not care seriously about that case.

To satisfy them, we design segregated stacks, as shown in **Fig. 11**. A segregated stack is a tuple of arrays that represents a list of tagged unions such that it stores each type of element

---

*5 https://asmjit.com/
into a separate array in order and manages a stack of element tags. The value stack of a segregated stack is merely an array of semantic values and, therefore, directly exposed to actions, with no access overhead. In the tag stack, value-tags (i.e., $V$) are compressed through run-length encoding, e.g., $[E, V^2, S, V^1, S]$ for $[E, V, V, S, V, S]$. The repetition length of value-tags corresponds to the number of argument semantic values for an action. Since we obtain it in constant time, we can pass a prefix array slice to the action in constant time. Finding a matched pair of brackets is obviously in constant time. We thus obtain a desired representation of stacks for bracket reduction.

With segregated stacks, assuming actions $A$ to be constant-time operations, $reduce\text{Chunk}(s)$ runs in linear time with respect to the length of $s$ and $s_1 \otimes s_2$ runs in linear time with respect to the length of $s_2$. If we implemented constant-time array append by using a list of arrays for a value stack, $s_1 \otimes s_2$ would run in linear time with respect to the length of the tag stack of $s_2$ although it would incur an additional cost for element access.

### 4.4 Process-level Parallel Processing

We would like to run Python actions in parallel. CPython, the standard implementation of Python, uses the global interpreter lock (GIL) so that threads in Python do not run in parallel. The multiprocessing module in the standard library overcomes this issue through multi-process execution. **Centaurus**, therefore, offers a multi-process implementation of parallel processing.

An important point of the multi-process implementation is that actions are implemented in Python and semantic values thus inhabit Python, while the core part of **Centaurus** is implemented in C++ for performance. In other words, stage-2 results semantically scatter both in the Python side and the C++ side.

Segregated stacks can deal naturally with this situation. It is sufficient to place the value stack of a segregated stack in Python. As a result, for each reduction event, the C++ side notifies the Python side of a bracket pair and its arity.

Moreover, this segregating design enables us to implement inter-process communication efficiently. In the Python side, only semantic values are sent via the APIs of multiprocessing. In the C++ side, input segments and the stage-2 results are shared through OS-level shared memory pages.

It is worth noting that the use of GIL is not limited to CPython (e.g., CRuby) and some languages (e.g., Lua) do not have threads themselves. Even for these languages, process-level parallel processing is feasible. Consequently, the multi-process implementation makes the core part of **Centaurus** immediately transferable to different languages.

Note that **Centaurus** simply performs multithreading when we use the C++ API.

### 5. Experiments

#### 5.1 Experimental Setting

We conducted the experiments of data extraction from XML and JSON datasets using **Centaurus**. We used a DBLP \(^5\) XML dataset and a CityLots \(^6\) GeoJSON dataset, both of which had nearly list structures. DBLP was of 2.5 GB and CityLots was of 1.8 GB, where we increased the size of CityLots by multiplying the top-level list by 10.

For **Centaurus**, both in Python and in C++, we implemented extractors **dblp** for DBLP to extract the list of the authors from each article published in 1990 (0.0014% in byte size was extracted) and **citylots** for CityLots to extract feature dictionaries on the JEFFERSON street (0.035% in byte size was extracted).

The extractors were implemented in a scraping style, as described in Section 2. The Python versions used the standard modules json and re for processing scraped segments. Similarly, the C++ versions used RapidJSON \(^8\) (v1.1.1-9-gfcec7353) and pugixml \(^9\) (v1.10.5-g6b576c), which are, to the best of our knowledge, some of the fastest dedicated parsers for JSON and XML, respectively.

We used a server equipped with four Xeon E7-8890 v4 (2.20 GHz, 24 cores) processors and 2048 GB memory (DDR4-2133), running Linux 4.15.0. We used g++ 7.4.0 for compiling **Centaurus** and Python 3.6.7 for running Python programs.

We measured the running time of Context.parse, i.e., excluded the time of initializing parsers and subprocesses. For comparison, we also measured the loading time of these datasets using the Python standard module minidom and json for Python and using pugixml and RapidJSON for C++. For pugixml and RapidJSON, we used the loaders from memory by using mmap to make settings fair to the implementation of **Centaurus**. We used the median of 15 measurements.

All the files used in the experiments are available online: https://github.com/satoshigeyuki/Centaurus.

#### 5.2 Experimental Results

Figure 12 shows the throughput of the extractors in Python and the serial loaders and Fig. 13 shows the relative speedup, where two additional processes for the stages 1 and 3 were always running for the extractors. **Centaurus** achieved excellent scalability: for **dblp**, 41.4 folds with 70 + 2 processes; for **citylots**, 45.7 folds with 84 + 2 processes. Although the throughput of the serial loader json was better than that of **Centaurus** with 6 + 2 pro-

---

\(^5\) https://dblp.org/xml/release/dblp-2019-06-02.xml.gz
\(^6\) https://github.com/entaurus/entaurus
\(^7\) https://github.com/zemirco/4f-city-lots-json
\(^8\) https://github.com/Tencent/rapidjson
\(^9\) https://github.com/zeux/pugixml
The extractors in C++ significantly outperformed the loaders of pugixml and RapidJSON. Here, it is also obvious that the extractors are more efficient than the corresponding ones based on pugixml and RapidJSON because they necessitate initial loading with the loaders for tree-based queries. Centaurus (parse) was only 11.4% slower than RapidJSON (read), which demonstrates the efficiency of the generated parser code. These results are sufficient to support the practical utility of Centaurus.

We observed no significant difference in execution time among Centaurus (parse), Centaurus (dry), and Centaurus (extract) for DBLP, which means the overhead of parallel processing was negligible. Meanwhile, we observed a little overhead of parallel processing for CityLots.

In summary, the experimental results of the extractors have validated the design and implementation of Centaurus. The Python version has proved its scalability, and the C++ version has proved its efficiency.

6. Related Work

6.1 JSON and XML Processing

In the field of databases, there is much literature on the efficient processing of semi-structured data through lightweight preprocessing (e.g., parsing and indexing). XML streaming [10], [14], [17] is an instance on XML processing. In particular, it has recently been actively investigated for JSON [9], [15], [18], [19], [25]. Although these studies dealt with JSON parsing, their problem settings differed depending on concerned data models and queries. For example, AT-GIS [18] was for GeoJSON and spatial queries; Mison [15] was for streaming almost uniform dictionaries found in Web APIs. FishStore [25] dealt with on-demand indexing of object stores from different queries. These approaches are closed under specific data models and queries. It is worth noting that the acceleration of Mison using early filtering on raw input [19] is similar to what the scraping-style data extraction using Centaurus does substantially; both filter out segments of input through lightweight pre-analysis.

Grammar-based enhancement of parallel processing [9], [10] aimed to compile queries with grammatical constraints. Our aim with Centaurus is grammar-based programming. Grammars are part of queries originally and both are designed together.

Koch and Scherzinger developed XML Stream Attribute Grammars [14], a class of attribute grammars for XML streaming. It assigns attributes to extended regular tree grammars, in which the right-hand sides of production rules may contain regular expressions. It can thus deal with an unbounded number of children in XML documents. This design is similar to the attribute grammars of Centaurus. Because each extended tree grammar there can be considered as a context-free word grammar by changing the interpretation of productions, theirs can correspond to ours.

Regarding parallel reduction of trees in the form of bracket sequences, we have built upon the work of Kakehi et al. [11]. However, we have not followed their algorithm because the re-
quirements on reduction operators (i.e., actions in \textsc{Centaurus}) do not conform to our purpose. Their algorithm, assuming algebraic conditions on operators, is guaranteed to work efficiently independently of the height of a given tree. In contrast, our algorithm is simpler and does not work well for tall trees but requires no algebraic condition on actions. \textsc{Centaurus} pursues experimental performance through well-designed implementations, such as asynchronous pipeline, segregated stacks, and SIMDization, rather than algorithmic elaboration.

It is worth noting that the serial chunking of XML documents asynchronously followed by parallel tree construction is common in DOM parsers (e.g., ParDOM \cite{22}). We can regard \textsc{Centaurus} as a generalization of them.

6.2 Parser Generators and Parallelism

In the field of formal languages, Barenghi et al. \cite{3} developed an efficient parallel parsing algorithm for operator precedence grammars (OPGs) \cite{5}. They found the local parsability of OPGs, which enables parallel parsing through the divide and conquer over input. Their algorithm thus realizes simple yet efficient parallel processing. However, the expressivity and extensibility of OPGs under parallel parsing are unclear. For example, the lookahead extension, which is natural to regular expressions and LL grammars, does not seem to fit OPGs. If OPGs were sufficiently expressive to real-world applications, there would be no reason not to adopt their algorithm.

LR grammars \cite{12} and PEG \cite{7} are more common in parser generators and practically easier to write regarding left recursion than LL(*) grammars even though the differences in formal expressive power between them are unclear. Here, we discuss them in terms of parallelism.

Parallel LR parsing and parallel PEG parsing are inherently difficult, at least as difficult as parallel LL parsing. Parallel LR parsing \cite{4} was more studied than parallel LL parsing \cite{23} probably because of its bottom-up parsing nature. However, the parsing decisions in LR parsing and LL parsing depend similarly on the current stacks even though the contents of stacks are different. This dependence on unbounded stacks makes divide-and-conquer parsing over input quite difficult. PEG parsing is formalized as top-down parsing with backtracking \cite{2}, saving the cost of backtracking through memoization. Parallel PEG parsing, therefore, suffers from the difficulty of top-down parsing as with parallel LL parsing and moreover, the difficulty derived from memoization.

Because we have chosen serial parsing, the combination of parallel reduction with them is another possible choice. However, as mentioned in Section 3.1, we consider that LL parsing (with regular lookahead) is more suited to combining with parallel reduction for processing large-scale input.

Packrat parsing \cite{6}, the standard algorithm for PEG parsing, uses a memoable of $O(mn)$ space, where $n$ denotes an input size and $m$ denotes a grammar size. As mentioned in Ref. \cite{6}, packrat parsing is inappropriate for parsing of large-scale input. Henglein and Rasmussen \cite{8} developed more space-efficient progressive tabular parsing, which still takes $O(mn)$ space for the worst case but takes $O(m)$ space for the best case as a result of its streaming behaviors. LL parsing with regular lookahead is more space-efficient and safer in processing large-scale input.

LR parsing is, assuming that the spatial cost of automata used is equivalent, as space-efficient as LL parsing with regular lookahead. However, because LR parsing is bottom-up parsing, it is inherently difficult to implement streaming parsing based on LR parsing; e.g., LR parsing cannot stream out parse trees in the form of bracket sequences. LL parsing (with regular lookahead) is naturally streamable and allows us to implement asynchronous pipeline parallel processing efficiently and simply.

7. Conclusion

In this paper, we have presented \textsc{Centaurus}, a dynamic parser generator for ad hoc data extraction, and experimentally demonstrated its excellent scalability and efficiency for data extraction.

Two extensions of \textsc{Centaurus} are promising. One is lookahead constructs, such as syntactic predicates like ANTLR’s, ordered choice like PEG’s, and negative lookahead in regular expressions. It enriches the expressive power of grammars and makes it easy to specify ad hoc matching at the grammar level. The other is parser generation aware of the void action, which guarantees no action of the associated nonterminals. Stage-1 parsers can suppress the bracket generation for void-action nonterminals and their descendants. It reduces intermediate data and thus makes both parsing and reduction efficient. We leave them for future work.

Acknowledgments We thank Akimasa Morihata for his discussion with the first author on an earlier version of this work.

This work was partially supported by JSPS KAKENHI Grant Number 16H01715.

References

\begin{thebibliography}{99}

\bibitem{1} Afroozeh, A. and Izmaylova, A.: Faster, Practical GLL Parsing, \textit{CC 2015: Compiler Construction}, Lecture Notes in Computer Science, \textbf{Vol.9031}, pp.89–108, Springer (2015).

\bibitem{2} Aho, A.V. and Ullman, J.D.: \textit{The Theory of Parsing, Translation, and Compiling}, Prentice-Hall (1972).

\bibitem{3} Barenghi, A., Reghizzi, S.C., Mandrioli, D., Panella, F. and Pradella, M.: \textit{Parallel Parsing Made Practical}, \textit{Sci. Comput. Program.}, \textbf{Vol.112}, No.2, pp.195–226 (2015).

\bibitem{4} Cohen, J., Hickey, T. and Katzoff, J.: \textit{Upper Bounds for Speedup in Parallel Parsing}, \textit{J. ACM}, \textbf{Vol.29}, No.2, pp.408–428 (1982).

\bibitem{5} Floyd, R.W.: \textit{Syntactic Analysis and Operator Precedence}, \textit{J. ACM}, \textbf{Vol.10}, No.3, pp.316–333 (1963).

\bibitem{6} Ford, B.: \textit{Packrat Parsing: Simple, Powerful, Lazy, Linear Time (Functional Pearl)}, \textit{Proc. 7th ACM SIGPLAN International Conference on Functional Programming}, ICFP ’02, pp.36–47, ACM (2002).

\bibitem{7} Ford, B.: \textit{Parsing Expression Grammars: A Recognition-Based Syntactic Foundation}, \textit{Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages}, POPL ’04, pp.111–122, ACM (2004).

\bibitem{8} Henglein, F. and Rasmussen, U.T.: \textit{PEG Parsing in Less Space Using Progressive Tabling and Dynamic Analysis}, \textit{Proc. 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation}, PEP’17, pp.35–46 (2017).

\bibitem{9} Jiang, L., Sun, X., Farooq, U. and Zhao, Z.: \textit{Scalable Processing of Contemporary Semi-Structured Data on Commodity Parallel Processors – A Compilation-based Approach}, \textit{Proc. 24th International Conference on Architectural Support for Programming Languages and Operating Systems}, ASPLOS ’19, pp.79–92, ACM (2019).

\bibitem{10} Jiang, L. and Zhao, Z.: \textit{Grammar-aware Parallelization for Scalable XPath Querying}, \textit{Proc. 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming}, PPaPP ’17, pp.371–383, ACM (2017).

\bibitem{11} Kakehi, K., Matsuzaki, K. and Emoto, K.: \textit{Efficient Parallel Tree Reductions on Distributed Memory Environments}, \textit{Scalable Computing: Practice and Experience}, \textbf{Vol.18}, No.1, pp.1–15 (2017).

\bibitem{12} Knuth, D.E.: \textit{On the Translation of Languages from Left to Right}, \textit{Inf. Control}, \textbf{Vol.8}, No.6, pp.607–639 (1965).

\end{thebibliography}
[13] Knuth, D.E.: Semantics of context-free languages, *Mathematical Systems Theory*, Vol.2, pp.127–145 (1968).
[14] Koch, C. and Scherzinger, S.: Attribute grammars for scalable query processing on XML streams, *VLDB J.*, Vol.16, No.3, pp.317–342 (2007).
[15] Li, Y., Katsipoulakis, N.R., Chandramouli, B., Goldstein, J. and Kossman, D.: Mison: A Fast JSON Parser for Data Analytics, *PVLDB*, Vol.10, No.10, pp.1118–1129 (2017).
[16] Mykytowycz, T., Musuvathi, M. and Schulte, W.: Data-Parallel Finite-State Machines, *Proc. 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’14*, pp.529–542, ACM (2014).
[17] Ogden, P., Thomas, D. and Pietzuch, P.: Scalable XML Query Processing using Parallel Pushdown Transducers, *PVLDB*, Vol.6, No.14, pp.1738–1749 (2013).
[18] Ogden, P., Thomas, D. and Pietzuch, P.: AT-GIS: Highly Parallel Spatial Query Processing with Associative Transducers, *Proc. 2016 International Conference on Management of Data, SIGMOD ’16*, pp.1041–1054, ACM (2016).
[19] Palkar, S., Abuzaid, F., Balis, P. and Zaharia, M.: Filter Before You Parse: Faster Analytics on Raw Data with Sparser, *PVLDB*, Vol.11, No.11, pp.1576–1589 (2018).
[20] Parr, T. and Fisher, K.: LL(*): The Foundation of the ANTLR Parser Generator, *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’11*, pp.425–436, ACM (2011).
[21] Poplawski, D.A.: On LL-Regular Grammars, *J. Comput. Syst. Sci.*, Vol.18, No.3, pp.218–227 (1979).
[22] Shah, B., Rao, P., Moon, B. and Rajagopalan, M.: A Data Parallel Algorithm for XML DOM Parsing, *XSym 2009: Database and XML Technologies*, Lecture Notes in Computer Science, Vol.5679, pp.75–90, Springer (2009).
[23] Vagner, L. and Melichar, B.: Parallel LL parsing, *Acta Inf.*, Vol.44, No.1, pp.1–21 (2007).
[24] van den Brand, M.G.J., Scheerder, J., Vinju, J.J. and Visser, E.: Disambiguation Filters for Scannerless Generalized LR Parsers, *CC 2002: Compiler Construction*, Lecture Notes in Computer Science, Vol.2304, pp.143–158, Springer (2002).
[25] Xie, D., Chandramouli, B., Li, Y. and Kossman, D.: FishStore: Faster Ingestion with Subset Hashing, *Proc. 2019 International Conference on Management of Data, SIGMOD ’19*, pp.1711–1728, ACM (2019).

Shigeyuki Sato is an Assistant Professor in the Graduate School of Information Science and Technology at the University of Tokyo. He received his Ph.D. from the University of Electro-Communications in 2015. His research interest is in compilers and parallel programming, especially, automatic parallelization, program synthesis, high-level optimizations, domain-specific languages, parallel patterns, and tree/graph processing. He is also a member of ACM and JSSST.

Hiroka Ihara received his B.E. and M.S. from the University of Tokyo in 2017 and 2019, respectively, and then joined the Center for Technology Innovation, Hitachi Ltd. He is currently working on enterprise storage systems.

Kenjiro Taura is a Professor in the Department of Information and Communication Engineering at the University of Tokyo. He received his B.S., M.S., and Ph.D. from the University of Tokyo in 1992, 1994, and 1997, respectively. His major research interests spread parallel and distributed computing, system software, and programming languages. He is also a member of ACM, IEEE, and USENIX.