An Implementation of a Parallel Bidirectional Parsing Algorithm

Yevhen Chepurnyy,
Blackburn College,
Computer Science Department,
Carlinville, IL, U.S.A.
euyork@gmail.com

Stefan Andrei,
Lamar University,
Computer Science Department,
Beaumont, TX, U.S.A.
Stefan.Andrei@lamar.edu

Abstract. With the growing prevalence of multi-core architectures, parallel parsing has been an important subject of research. Lam, Ding, and Liu claim in 2008 that among all important phases of XML (e.g. parsing, access, modification, and serialization), parsing is the most time-consuming one. Many parallel parsing algorithms have been created to improve the traditional sequential parsing algorithm. In this paper, we present a unique parallel parsing algorithm based on the bidirectional parsing approach described by Stefan Andrei in 2009 [1]. The algorithm takes full advantage of the multiprocessor architecture, and, as our experimental data shows, offers a significant improvement of the parsing speed versus the sequential parsing algorithms.

1. Introduction

Parsing has been a subject of extensive research since the 70s. It is an important part of every compiler, and as most of the programming languages are subject to the compilation phase, the importance of parsing cannot be overlooked. Parsing also has applications in other areas of computer science, such as natural language processing, speech recognition, translations to other languages, automatic error correction, and so on. Lam, Ding, and Liu claim that among all important phases of XML (e.g. parsing, access, modification, and serialization), parsing is the most time-consuming one [9].

There are two types of parsers: top-down and bottom-up. A top-down parser begins to process the input looking at the starting production, then examines the productions immediately derived from the starting one, then looks at the productions immediately derived from the ones derived from the starting production, and so on, recursively. Another way to describe top-down and bottom-up parsing in comparison to each other is representing a context-free grammar as a tree structure. The root of the said tree would be the grammar's starting production, while the leaf nodes would be terminals. The Left-to-right Leftmost parsing, usually abbreviated as LL, and the Right-to-left Rightmost parsing, dubbed RR, are examples of top-down parsing methods [4]. Left-to-right, or right-to-left, means the direction of parsing the input word, and leftmost, or rightmost, means the directions of processing the particular grammar rules. LL and RR parsers are usually coded by hand, with a few exclusions. ANTLR, short for ANother Tool for Language Recognition, is an example of an LL parser generator. Bottom-up parsers are an exact opposite of the top-down. A bottom-up parser identifies the most basic units, and works its way up to the starting production. The Left-to-right Rightmost parsing, or LR [3], and the Right-to-left

Leftmost parsing or RL [2], represent two general ways to use this approach. Bottom-up parsing methods are more complex than top-down parsing, and ought to be described in greater detail.

LR(0) and RL(0) parsers are the most basic type, which can be represented by a simple state-transition machine with input and output tapes. Each state of this machine is a collection of grammar productions, with a dot on its right hand side indicating which tokens of that production have been recognized by the grammar. Each of the machine's transitions would be one of the tokens. The process of LR(0) or RL(0) parsing may be described as examining the input symbols, comparing them to the production rules' right hand sides, and replacing the matched sets of symbols with the respective production's left hand side, until the starting symbol has been derived in this way. The state transition machine emulates that as taking a state transition which corresponds to the currently examined input symbol, and performing one of the following actions: *shift*, which consists of copying the examined symbol to the output tape; *reduce*, which removes x symbols from the output tape (where x = length of a certain production's right hand side), and copies the left hand side of that production to the output; *accept*, which designates a successful end of the parsing; and *reject*, which signals that the examined input word contains an invalid combination of input tokens.

LR(1) and RL(1) are an improved version of LR(0) and RL(0). They offer more flexibility and are able to parse more languages. This is achieved by introducing a lookahead – a terminal that could precede, for RL, or follow, in case of LR, the particular rule's left hand side token in the input word. The parenthesized one beside the grammar name represents the number of lookahead terminals. LR(1) and RL(1) contain much more states than the zero-lookahead versions – for each of the lookaheads, a new production needs to be introduced into the automaton, often leading to the creation of new states. A typical RL(1) item looks like this (Figure 1):

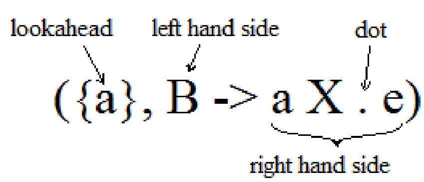

Figure 1. An RL(1) item.

Figure 1 shows an item for the production B $\rightarrow$ a X e, where the terminal 'e' has already been processed, and 'a' is a lookahead terminal that might precede X.

LALR(1) and LARL(1), usually abbreviated as LALR or LARL, where LA stands for "lookahead", were designed to retain the RL(1) and LR(1) grammars' flexibility, but without having to create bulky automata with an unnecessarily large number of states. In LALR and LARL grammars, the states that contain identical productions that differ only in lookaheads are merged. All the transitions of the merged states are retained – if a grammar holds to the LALR properties, those transitions would lead to the states that would also be merged. An LARL parser can be represented by a deterministic pushdown automaton. This imposes some limits on which languages can be parsed in a LALR or LARL way – meaning, only the languages with a context-free property. For example, natural languages cannot be parsed that way, but many of their unambiguous parts can be [12].

A context-free grammar consists of a set of terminals, a set of nonterminals (including a start symbol), and a set of productions. Terminals are the literal characters that can appear in the inputs to or outputs from the production rules of a formal grammar and that cannot be broken down into "smaller" units. In the practical applications, terminals are the tokens of which the input word, that is to be parsed by the grammar. Nonterminals are the symbols used to represent terminals and other nonterminals in the grammar productions. The latter are the body of the grammar, consisting of the left hand side, which is always a single nonterminal, and the right hand side, which can consist of one to any finite number of terminals, nonterminals, or both, or a symbol that represents null word. A word is null (also known as empty) if its length is zero. Productions link terminals to nonterminals, and nonterminals to each other and the start symbol [11].

The context-free language this research refers to is the Unified Modeling Language (UML). It is a general-purpose modeling language in the field of object-oriented software engineering. The standard was managed and created by the employees of the Rational Software Corporation: Ivar Jacobson, Grady Booch, and James Rumbaugh. UML combines techniques from data modeling, business modeling, object modeling, and component modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies [10]. UML parsing has possible uses in reverse software engineering.

Sequential parsing, no matter the type or method, has always left room for improving its efficiency. Its time complexity was $O(n+|G|)$ in the very best case, with n being a number of tokens in the word, and $|G|$ being the number of symbols in the grammar, or its size. With the introduction of multi-core processors, there have been many attempts to introduce parallelism to the process of parsing. The problem here is in the splitting of an input word into chunks to parse them in parallel. If done arbitrarily, the finite state machine would fail to parse most of those chunks - it always starts the parsing process in state 0, while the appropriate state for properly parsing most of the chunks would be different.

One recent static parallel parser for XML was described by *Wei Lu* et al. in their work "*A Parallel Approach to XML Parsing*". Their parser solves the splitting problem by pre-parsing the document to create its "skeleton" structure, which divides it into sets of tags corresponding to the certain subsets of the grammar. When starting the parallel parsing process, the finite state machine starts with the state corresponding to a particular grammar subset. Compared to the sequential parsing, this approach involves a considerable amount of overhead because of the preprocessing, but still offers a significant increase in parsing speed [5].

Another related approach to the parallel parsing of XML documents was described by *Yu Wu, Qi Zhang, Zhiqiang Yu,* and *Jianhui Li* in their work *"A Hybrid Parallel Processing for XML Parsing and Schema Validation"*. Their approach avoids any pre-processing overhead by introducing a notion of speculative parsing. A speculative parser treats each chunk as a separate document, and processes it starting with the first opening tag. All the segments of the chunks that fall out of the XML document structure, like an unresolved opening tag, are grouped by type and put onto respective queues, and resolved after the parallel phase of processing is complete. This approach lessens the amount of overhead by limiting it to the less resource-consuming post-processing [6].

Another type of a parallel parser was described in "Parallel Parsing-based Reverse Engineering" [1]. The author presented a bidirectional parallel parser, which, using an RL and an LR parsers equivalent to each other, parsed the input word in parallel starting simultaneously in the beginning of the input, and at the end. The workability of creating both an LR and an RL parser from one grammar that holds to the LR or RL property, and the complete equivalence of them to each other was proved in [2]. The two parsers stop processing the word nondeterministically, by halting when all of the input words have been consumed. This is also described in detail in [1]. This approach avoids the word-splitting problem entirely, but does not make use of more than two processors.

Motivation. This paper explores the effectiveness of combining bidirectional parallel parsing with splitting a document into statically defined sets of tokens and processing them in parallel. The potential increase in efficiency of parsing after using multiple bidirectional parsers in parallel is investigated. A Java implementation of different parsing algorithms is created to evaluate the performance of parallel bidirectional parsing.

Structure of the paper. Section 2 explains the notations and grammar used. Section 3 provides the description of the implementation and the way it operates, and gives the examples of execution. Section 4 gives the experimental results. Conclusion and Future Work end this paper.

2. Preliminaries and Notations

We denoted an empty string as 'lambda', strings that contain lowercase letters as terminals (except for 'Smth'), and the all-caps strings as nonterminals. The following UML grammar is used in the running example of our implementation:

```
1.      CLASS = ( object Class ATTR_ATTRIBUTES ATTR_OPERATIONS Smth)
2.      Smth = ATTR_NAME ATTR_QUID ATTR_DOCUMENTATION ATTR_ABSTRACT
3.      ATTR_ATTRIBUTES = attributes ( list Attributes ATTRIBUTES )
4.      ATTRIBUTES = ATTRIBUTE ATTRIBUTES
5.      ATTRIBUTE = ( object ClassAttribute ATTR_NAME ATTR_QUID
   ATTR_DOCUMENTATION ATTR_EXPORT_CONTROL ATTR_TYPE ATTR_INITV ATTR_STATIC
   ATTR_DERIVED )
6.      ATTR_TYPE = type stringType
```

```
7.       ATTR_INITV = initv stringInitv
8.       ATTR_STATIC = static BOOLEAN
9.       ATTR_DERIVED = derived BOOLEAN
10.      ATTR_OPERATIONS = operations ( list Operations OPERATIONS )
11.      OPERATIONS = OPERATION OPERATIONS
12.      OPERATION = ( object Operation ATTR_PRECONDITION
   ATTR_POSTCONDITION ATTR_SEMANTICS ATTR_NAME ATTR_QUID
   ATTR_DOCUMENTATION ATTR_RESULT ATTR_EXPORT_CONTROL ATTR_EXCEPTION
   ATTR_CONCURENCY )
13.      ATTR_PRECONDITION = pre_condition string
14.      ATTR_POSTCONDITION = post_condition string
15.      ATTR_SEMANTICS = semantics string
16.      ATTR_RESULT = result stringResult
17.      ATTR_EXCEPTION = exception stringException
18.      ATTR_CONCURENCY = concurency STRING_CONCURENCY
19.      ATTR_NAME = name string
20.      ATTR_QUID = quid string
21.      ATTR_EXPORT_CONTROL = exportControl STRING_CONTROL
22.      ATTR_ABSTRACT = abstract BOOLEAN
23.      ATTR_DOCUMENTATION = documentation stringDoc
24.      STRING_CONTROL = "Public"
25.      STRING_CONTROL = "Protected"
26.      STRING_CONTROL = "Private"
27.      STRING_CONCURENCY = "Sequential"
28.      BOOLEAN = TRUE
29.      BOOLEAN = FALSE
30.      ATTR_ABSTRACT = lambda
31.      ATTR_TYPE = lambda
32.      ATTR_INITV = lambda
33.      ATTRIBUTES = lambda
34.      ATTR_STATIC = lambda
35.      ATTR_DERIVED = lambda
36.      OPERATIONS = lambda
37.      ATTR_RESULT = lambda
38.      ATTR_EXCEPTION = lambda
39.      ATTR_CONCURENCY = lambda
40.      ATTR_EXPORT_CONTROL = lambda
41.      ATTR_DOCUMENTATION = lambda
```

3. The implementation of the parallel bidirectional parsing algorithm

This particular implementation of the algorithm uses a shortened version of the UML grammar, but other grammars should also work with it, if they are context-free and hold up to the LALR(1) properties. To parse the input word in parallel, the grammar is divided into sub-grammars, each of which has one of the starting production’s right hand side nonterminals as its own start symbol. Separate finite automata is constructed for each sub-grammar to accept the viable prefixes for LR(1) and RL(1) items.

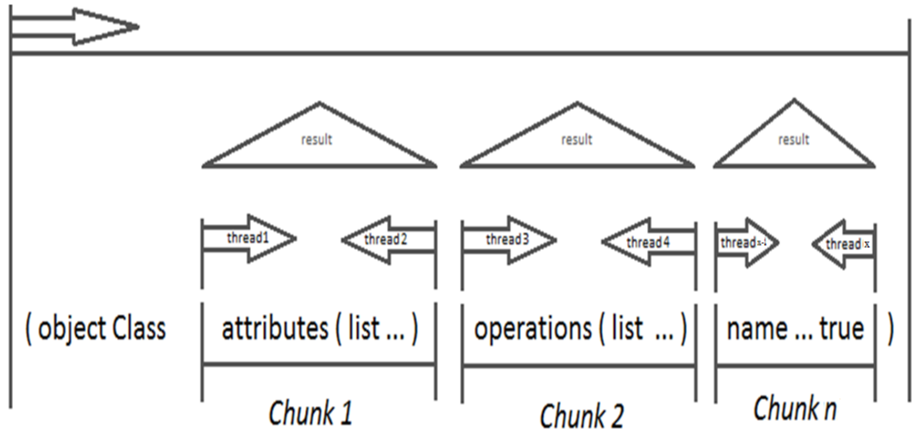

Figure 2. Parallel bidirectional parsing

To read the input word, an ad-hoc lexical analyzer is used. It reads the input file sequentially, token by token; if a certain trigger token is read, the analyzer reacts by creating a separate input stack, which will contain a chunk of text that acts as an input word for one of the sub-grammars.

Then, three parsers are created for each of the grammars: an RL, an LR, and a special-case LR for post-processing. The latter should be able to not only parse input that contains nonterminals, but also construct a valid final derivation of the input word.

A bidirectional parser framework object contains the three parsers mentioned above, and starts them as needed. The parallel bidirectional parser framework creates as many of the mentioned bidirectional parser objects as there are chunks of text. As many bidirectional parsers as possible are started simultaneously. The number of objects started depends on the number of processors available to the Java virtual machine. As the bidirectional parser uses two threads at most, the parallel bidirectional framework starts one bidirectional parser for every two processors available. As all the bidirectional parsers finish processing their part of the input, the results are recombined and processed sequentially, left-to-right.

Figure 3 demonstrates the most important classes in the implementation. The `Main` class creates instances of both `ParallelBidirectional` and `Sequential` classes, and feeds them the input words. The `ParallelBidirectional` class creates an appropriate number of `Chomper` instances, each of which has two instances of the `SimpleParse` class, and an instance of the `ActionIII` class. The latter is passed its parent `Chomper` as an argument, to keep track of the execution process. The `SimpleParse` class contains the right number of `Grammar` and `Automaton` instances. Both `ParallelBidirectional` and `Sequential` classes also have an instance of the `SimpleLex` class. The `Sequential` class contains only one `SimpleParse` instance. The automaton needs to be passed an instance of `Grammar` as an argument, so it always has one. Classes not mentioned in the diagram are `PseudoStack`, `Production`, `TableRule`, and `dCoordinate`. A `PseudoStack` object inherits all `Stack` methods and parameters, with only one change: a ‘mirrored’ boolean is introduced, and based on that boolean, either 0^{th} element or the end of the stack are considered its top. Production class contains all necessary variables to describe a grammar production – left hand side, right hand side, lookahead, etc., - so as some helper methods irrelevant to parsing. The `TableRule` class is a representation of one entry in an action table. The `dCoordinate` class represents a state transition in the viable prefix automaton.

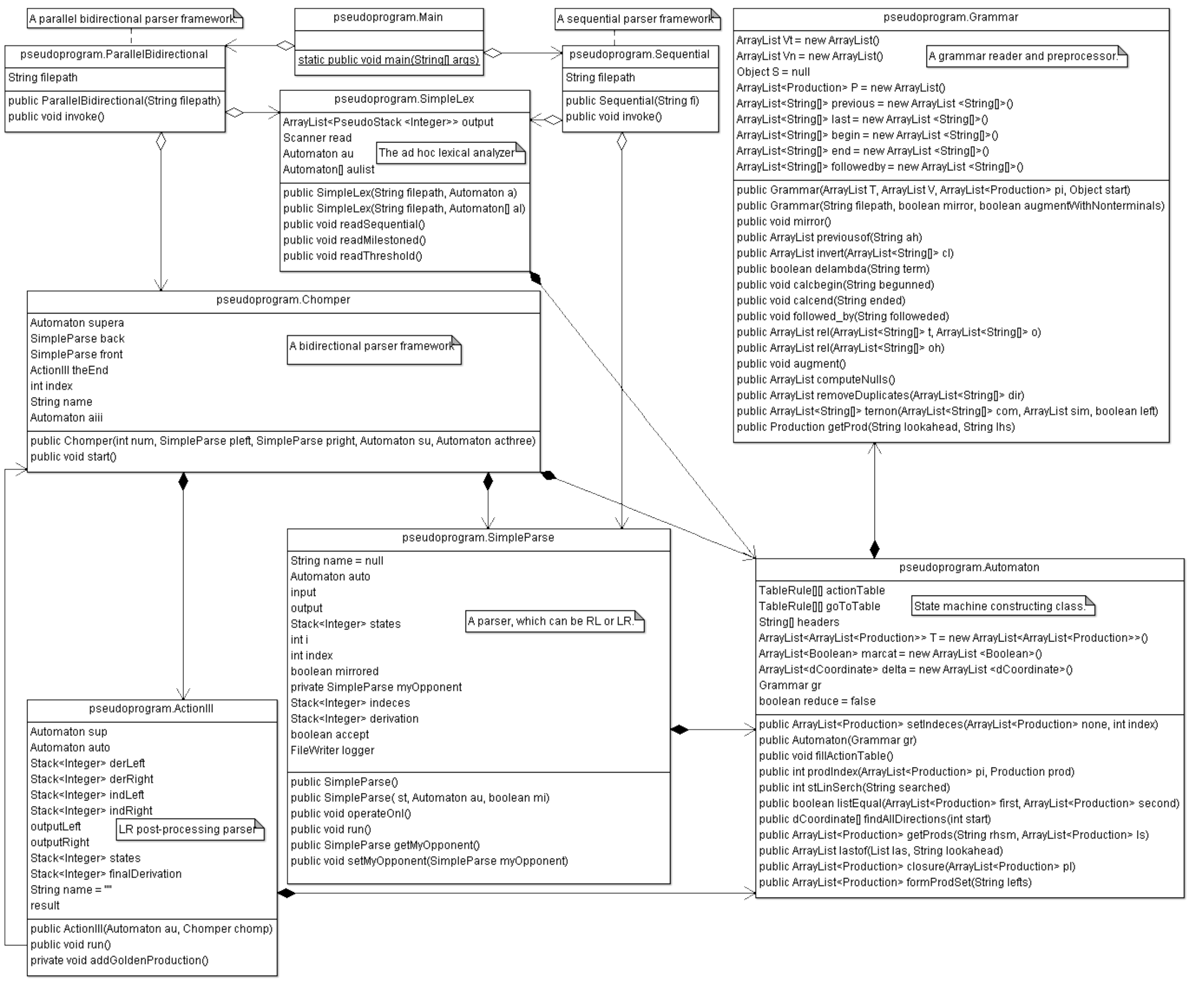

Figure 3. The class diagram of our implementation

As a short, but illustrative, example, let us consider a parallel bidirectional parsing of the following word:

```
( object Class
attributes ( list Attributes ) operations ( list Operations )
name string quid string documentation stringDoc  abstract TRUE )
```

It is split initially into three chunks: `attributes ( list Attributes )`, `operations ( list Operations )`, and everything between the tokens `name` and `TRUE`. Ideally, all three chunks would be parsed simultaneously, using two threads for each. Here is the execution table for the LR parser:

Symbol	State	Action	Input	Output	States
attributes	0	shift 2	) Attributes list (	attributes	0 2
(	2	shift 3	) Attributes list	attributes (	0 2 3
list	3	shift 4	) Attributes	attributes (list	0 2 3 4

The RL parser would at the same time parse the word from right to left:

Symbol	State	Action	Input	Output	States
)	0	shift 2	Attributes list (attributes	)	0 2
Attributes	2	reduce 0 ATTRIBUTES, go to 3	Attributes list (attributes	ATTRIBUTES)	0 2 3
Attributes	3	shift 4	list (attributes	Attributes ATTRIBUTES)	0 2 3 4

Both parsers would stop when all the input have been processed, as described in [1]. The post-processing phase would use the LR parser's states, and both parsers' output stacks, left one's as its own output, and right one as input. The execution trace is as follows:

Symbol	State	Action	Input	Output	States
Attributes	4	shift 5	ATTRIBUTES)	attributes (list Attributes	0 2 3 4 5
ATTRIBUTES	5	shift-nonterminal 6	)	attributes (list Attributes ATTRIBUTES	0 2 3 4 5 6
)	6	shift 9	null	attributes (list Attributes ATTRIBUTES)	0 2 3 4 5 6 9
null	9	reduce 6 ATTR_ATTRIBUTES, go to 1(accept)	null	ATTR_ATTRIBUTES	0 1

The Operations chunk would be parsed in the very same way. The LR execution trace is:

Symbol	State	Action	Input	Output	States
operations	0	shift 2	) Operations list (	operations	0 2
(	2	shift 3	) Operations list	operations (	0 2 3
list	3	shift 4	) Operations	operations (list	0 2 3 4

Operations	4	shift 5	)	operations (list Operations	0 2 3 4 5

The RL parser would execute in the following way:

Symbol	State	Action	Input	Output	States
)	0	shift 2	Operations list (operations	)	0 2

This time, the LR thread has done most of the processing. The LR post-processing phase is executed as follows:

Symbol	State	Action	Input	Output	States
)	5	reduce 0 OPERATIONS, go to 6	)	operations (list Operations OPERATIONS	0 2 3 4 5 6
)	6	shift 9	null	operations (list Operations OPERATIONS)	0 2 3 4 5 6 9
null	9	reduce 6 ATTR_OPERATIONS, go to 1(accept)	null	ATTR_OPERATIONS	0 1

The last chunk's execution is no different. Here is its LR thread:

Symbol	State	Action	Input	Output	States
name	0	shift 3	TRUE abstract stringDoc documentation string quid string	name	0 3
string	3	shift 6	TRUE abstract stringDoc documentation string quid	name string	0 3 6
quid	6	reduce 2 ATTR_NAME, go to 2	TRUE abstract stringDoc documentation string quid	ATTR_NAME	0 2
quid	2	shift 5	TRUE abstract stringDoc documentation string	ATTR_NAME quid	0 2 5
string	5	shift 9	TRUE abstract stringDoc documentation	ATTR_NAME quid string	0 2 5 9
documentation	9	reduce 2 ATTR_QUID, go to 4	TRUE abstract stringDoc documentation	ATTR_NAME ATTR_QUID	0 2 4
documentation	4	shift 8	TRUE abstract	ATTR_NAME ATTR_QUID	0 2 4 8

			stringDoc	documentation	
stringDoc	8	shift 12	TRUE abstract	ATTR_NAME ATTR_QUID documentation stringDoc	0 2 4 8 12
abstract	12	reduce 2 ATTR_DOCUMENTATION, go to 7	TRUE	ATTR_NAME ATTR_QUID ATTR_DOCUMENTATION	0 2 4 7
abstract	7	shift 11	TRUE	ATTR_NAME ATTR_QUID ATTR_DOCUMENTATION abstract	0 2 4 7 11

The RL parser executed much slower, again:

Symbol	**State**	**Action**	**Input**	**Output**	**States**
TRUE	0	shift 4	abstract stringDoc documentation string quid string name	TRUE	0 4

The post-processing is then as follows:

Symbol	**State**	**Action**	**Input**	**Output**	**States**
TRUE	11	shift 14	null	ATTR_NAME ATTR_QUID ATTR_DOCUMENTATION abstract TRUE	0 2 4 7 11 14
null	14	reduce 1 BOOLEAN, go to 13	null	ATTR_NAME ATTR_QUID ATTR_DOCUMENTATION abstract BOOLEAN	0 2 4 7 11 13
null	13	reduce 2 ATTR_ABSTRACT, go to 10	null	ATTR_NAME ATTR_QUID ATTR_DOCUMENTATION ATTR_ABSTRACT	0 2 4 7 10
null	0	reduce 4 Smth, go to 1(accept)		Smth	0 1

The results of all three parsing passes are recombined, the few nonterminals that have not been included into the chunks of text are added to it, and the result is parsed one last time. The input for the final parse would look exactly like the starting production rule of the grammar, and its parsing would be comprised of one reduce operation. As mentioned above, the time complexity of the sequential parsing has an order of O(n), where n is the length of the input word. The bidirectional parser's time complexity, as described in [2], is the same in the worst case, and O(n/2+x), where x is the length of the post-processed sub-word, which is the remainder of the input that needs to be parsed after each of the bidirectional parsers finish processing their pieces of input. Using parallelism, we further improve the time complexity to O(n/k+x), where k is the number of processors.

4. Experimental results

The implementation was tested on several processors with varying number of available threads. Each run included the parsing of different-length words. The time measurement results are the following:

For Intel Core 2 Duo E8400, dual-core, with core clock 3.0 GHz, Table 1 and Figure 4 shows the execution times expressed in nanoseconds (ns) for various words:

Word length	Parallel Bidirectional	Sequential
506	25252123 ns	2654669 ns
1014	5110424 ns	4787401 ns
1958	33894243 ns	8860047 ns
11518	67451818 ns	88791835 ns
34870	398498048 ns	549453461 ns
79558	1350515189 ns	2439088921 ns
139414	3984107440 ns	6978141087 ns
218950	9043524881 ns	16930300920 ns
437878	35076111342 ns	67219607845 ns
557590	56221657049 ns	108247496322 ns

Table 1. Comparison between sequential and parallel implementations on a dual-core processor.

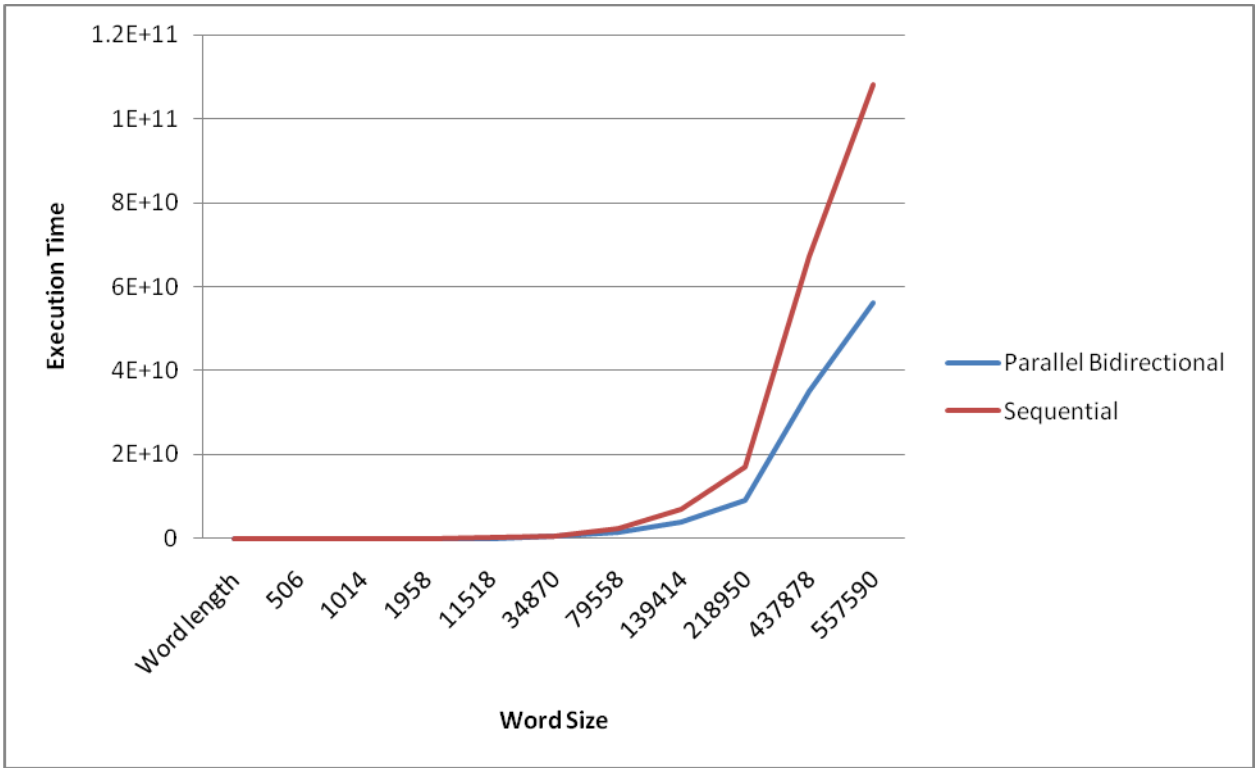

Figure 4. Execution time comparison #1

Analyzing Table 1 and Figure 4, we observe that the dual-core architecture does not imply a significant improvement in the performance of a parallel implementation versus the sequential one, until the length of the input increases dramatically. We believe this is due to the overhead needed in the communication between the threads responsible for parsing individual chunks of the input word.

For Intel Core 2 Quad Q9550, core clock 2.83 GHz, the data is as follows:

Word length	Parallel Bidirectional	Sequential
506	5864090 ns	4189464 ns
1014	24166500 ns	25377723 ns
1958	10943041 ns	10056091 ns
11518	79229527 ns	105060509 ns
34870	390534631 ns	615374024 ns
79558	1428639078 ns	2721375107 ns
139414	4021738981 ns	7371264836 ns
218950	9392302734 ns	17810047685 ns
437878	36580530061 ns	70750707513 ns
557590	59502111399 ns	1.15471E+11 ns

Table 2. Comparison between sequential and parallel implementations on a quad-core processor

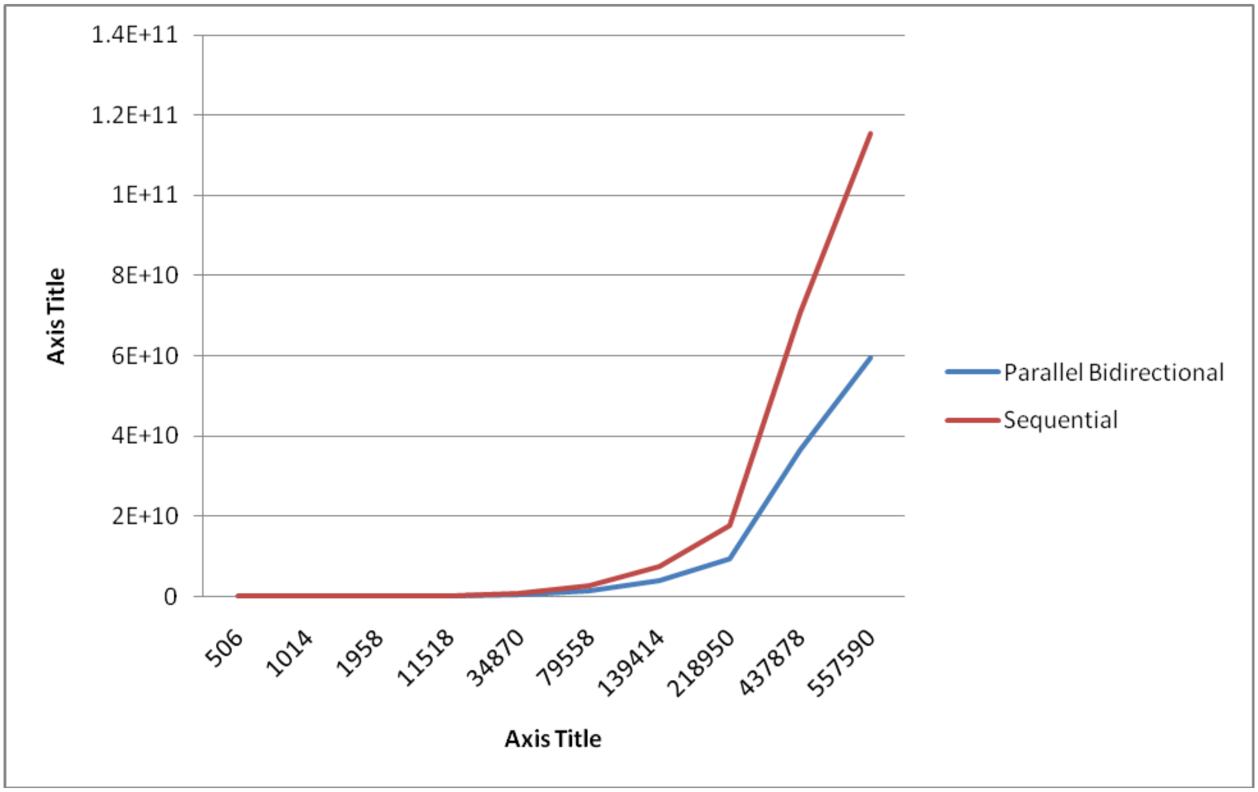

Figure 5. Execution time comparison #2

As we can see in Table 2 and Figure 5, the sequential algorithm has an advantage over the parallel bidirectional only for the smaller input words, and the advantage is only slight. Being able to run four threads simultaneously reduces the impact of the overhead significantly.

And, finally, for Intel Core i7 920, quad-core with eight available threads and core clock of 2.67 GHz, the data is shown in the following figures:

Word length	Parallel Bidirectional	Sequential
506	11703265 ns	13509113 ns
1014	5005921 ns	4486783 ns
1958	9158259 ns	9223535 ns
11518	77737110 ns	111370893 ns
34870	458384083 ns	776490257 ns
79558	1987139041 ns	3654058494 ns
139414	5836634648 ns	10881424143 ns
218950	13948330771 ns	26525701950 ns
437878	54400616131 ns	1.04487E+11 ns
557590	88217821877 ns	1.68993E+11 ns

Table 3. Comparison between sequential and parallel implementations on a quad-core, eight-thread processor

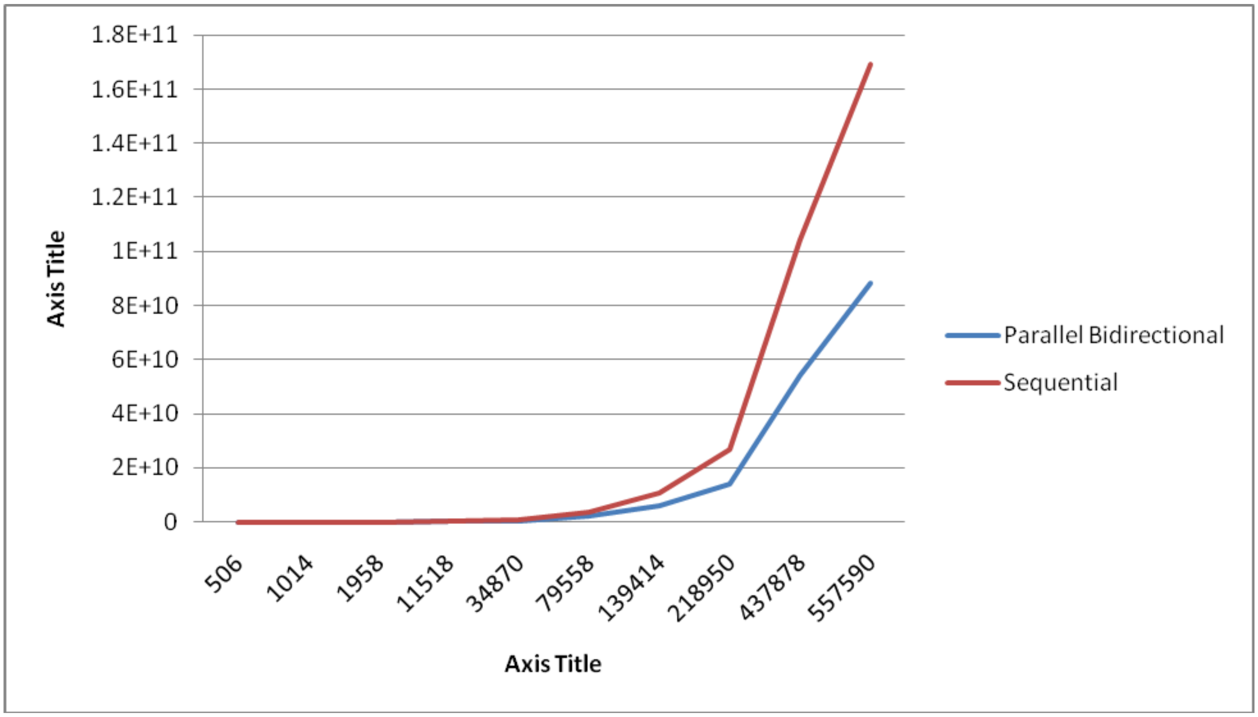

Figure 6. Execution time comparison #3

From Table 3 and Figure 6, we can see that the increase in the amount of parallelism leads to an even further improvement in performance. The execution times are comparable for the shorter input words, and significantly better for the longer ones.

5. Conclusions and Future Work

In this paper, we have described an implementation of a parallel parsing algorithm based on the bidirectional parsing algorithm [1], which certainly has a lot of potential. Though the described implementation was programmed in a more of a proof-of-concept than practical way, the experimental data yields reassuring results. As it can be seen from the graphs and tables from Section 4, the parallel bidirectional algorithm offers a significant, close to 50% speed improvement compared to sequential parsing for the longer words. In the future, the implementation can be further refined and brought up to the level of widely used state-of-the-art parsers like CUP [8] and ANTLR [7]. Adapting it to other languages should present no difficulty. Another direction to investigate is a less-static, size-based input word splitting, that would improve the parsing efficiency by distributing the load more evenly between threads.

6. Acknowledgements

This project was funded by National Science Foundation, grant no. 0851912, "*REU SITE: Engagement of Undergraduates in Theory, Algorithm & Applications of Science and Engineering in Information Technology*". We would like to thank Dr. S. Kami Makki for all of his help on our way. We also thank Quentin Mayo and Angel Vazquez for their ongoing support.

7. Bibliography

[1] Stefan Andrei: "Parallel Parsing-based Reverse Engineering". *Proceedings of the First 2009 World Congress on Computer Science and Information Engineering* (CSIE 2009), IEEE Computer Society, pp. 503-507, Los Angeles, USA, March 31-April 2, 2009.

[2] Stefan Andrei: "*Bidirectional Parsing*" (PhD Dissertation), Hamburg, February 2000, [available online at http://www.sub.uni-hamburg.de/disse/134/inhalt.html], 150 pages.

[3] Donald E. Knuth: "On the Translation of Languages from Left to Right", *Information and Control*, Vol. **8**, pg. 607-639, 1965.

[4] D. J. Rosenkrantz, R.E. Stearns: "Properties of deterministic top down grammars", *Proceedings of the first annual ACM symposium on Theory of Computing*, ACM New York, NY, USA, 1969

[5] Wei Lu, Kenneth Chiu, and Yinfei Pan: "A Parallel Approach to XML Parsing". *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, IEEE Computer Society Washington, DC, USA, 2006

[6] Yu Wu, Qi Zhang, Zhiqiang Yu, and Jianhui Li: "A Hybrid Parallel Processing for XML Parsing and Schema Validation". *Balisage: The Markup Conference*, August 12 - 15, 2008.

[7] T. J. Parr, R.W. Quong: "ANTLR: A Predicated LL(k) Parser Generator", *Software—Practice and Experience*, Vol. **25**(7), 789–810 (July 1995).

[8] S. Hudson, F. Flannery, and S. Ananian: "CUP – a LALR Parser for Java". *Technical Report*, Technical University of Munich, [available online at http://www2.cs.tum.edu/projects/cup], 2006.

[9] Tak Cheung Lam, Jianxun Jason Ding, Jyh-Charn Liu. XML Document Parsing: Operational and Performance Characteristics. *IEEE Computer*, pp. 30-37, 2008.

[10] Grady Booch, James Rumbaugh, and Ivar Jacobson. Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Technology Series). *Addison-Wesley Professional*, 2005.

[11] Chomsky, Noam. "Three models for the description of language". *IEEE Transactions on Information Theory*, Vol. **2** (3), September 1956.

[12] Shieber, Stuart (1985). "Evidence against the context-freeness of natural language". *Linguistics and Philosophy* **8** (3): 333–343.