УДК 004.021
DOI: 10.20535/SRIT.2308-8893.2018.4.05

A PARALLEL SEARCH ALGORITHM FOR FORMAL GRAMMAR DATA TYPES

ANASTASIIA PRODAN

Abstract. In this paper, we developed a concurrent generic heuristic algorithm for parallel parsing and searching in structured text datasets. The main objective of the algorithm was to increase an efficiency of central processing unit dependent operations when parsing large-scale datasets by using a parallel approach. The developed algorithm uses heuristics to find requested data without needing to process the whole file and without syntax tree building. It can be applied to any data formats. An increase in efficiency was discovered when input-output operations take significantly less time than the process of searching, the file is loaded into random access memory or when an efficient non-sequential access to file is possible. We also developed a prototype implementation of the algorithm for use in performance comparisons. The prototype supports searching in large-scale XML datasets using a subset of XPath expressions to specify search request. Our experimental results show that the developed algorithm is faster than classical algorithms, when all the requirements are met and the desired data is located closer to the beginning of the dataset. In worst cases, our algorithm gives nearly the same results as the others, but consumes more memory.

Keywords: grammar, search, parallelism, concurrency, heuristics.

INTRODUCTION

Nowadays there are a lot of digital data representation formats, many of them are broadly used in almost all fields of human's interest. Recent achievements of the information technology have changed the meaning of the information in business and everyday life. The efficiency of the data storing methods and the speed of data search have become a valuable advantage.

Most commonly used search methods are divided into three common parts:

1) parsing,
2) decoding,
3) searching.

Parsing refers to lexical and syntax analysis. Input for this stage is raw text data, and abstract syntax tree is the output. For context-free LL(1) grammars, lexical analysis can be done using the state machine. Tokenized text is then processed by one of the forward recursive parsing algorithms.

Decoding refers to semantic analysis. Input for this stage is abstract syntax tree, and the output depends on file format and decoding engine. For XML-based data types document object model has to be built. For JSON and other data types the output of decoding stage is not standardized and depends on the current application. Data is represented in graph or tree structure.

Search stage can be started only after all the previous stages are completed. On the search stage, we traverse the inner structure of the decoded data and return part of the data, if it matches all the search criteria.

To increase speed and efficiency of the search, indexing is used. Indexing allows search algorithm to go directly to the searched data, skipping first two stages. The problem is that full index is not always available. The process of indexing requires much time, memory and storage space, so it is redundant when we need to process the file only once, or when we do not have enough storage space to store full index. Partial index can only speed up some simple queries, but is useless for complex ones, so search system has to fall back to the first algorithm, that is less efficient.

ALGORITHM

For the cases, where full indexing is not possible or not necessary, we developed our concurrent heuristic search algorithm. The inputs for this algorithm are search query and raw text data, and the output is the found data.

Classical way of increasing calculations speed is to run them in parallel. Using high parallel approach, we can process large files faster than sequentially, but we should be able to read file non-sequentially. The general algorithm scheme is shown at fig. 1.

Main steps of the algorithm are listed below:

1. Split file into n fixed-size buffers.
2. Run k parser threads, where $k \le n$. Each parser thread process a buffer sequentially, from the beginning.
3. When thread has completed the processing of the buffer, it consumes next buffer from the remaining queue.
4. If all the search criteria are satisfied by one of the threads and all the previous buffers, data is found. Return the data.
5. When the queue is empty, return failure.

The following data structures are required for this algorithm:

1. Currently used buffers.
2. Buffer queue.
3. Results list.

Currently used buffers store raw text data that is being processed by the parser thread. Size of the buffer is fixed. To calculate optimal size of the buffer, we have to consider limitations of maximum available memory and minimal processing unit size in formal grammar representation.

Buffer queue stores pointers to data that is not yet available for reading, and has to be loaded into one of the buffers yet.

Results list is a simple list that contains data structures of the special type - result. Every time a worker thread finishes processing of a buffer, it has to put a result into the results list, so decoded information will be available for all the other threads. The structure of a result will be described above.

The main process of lexical and syntax analysis is executed in parser threads. Each parser thread executes the following set of operations:

- lexical analysis;
- structural analysis;
- data search;
- result updating.

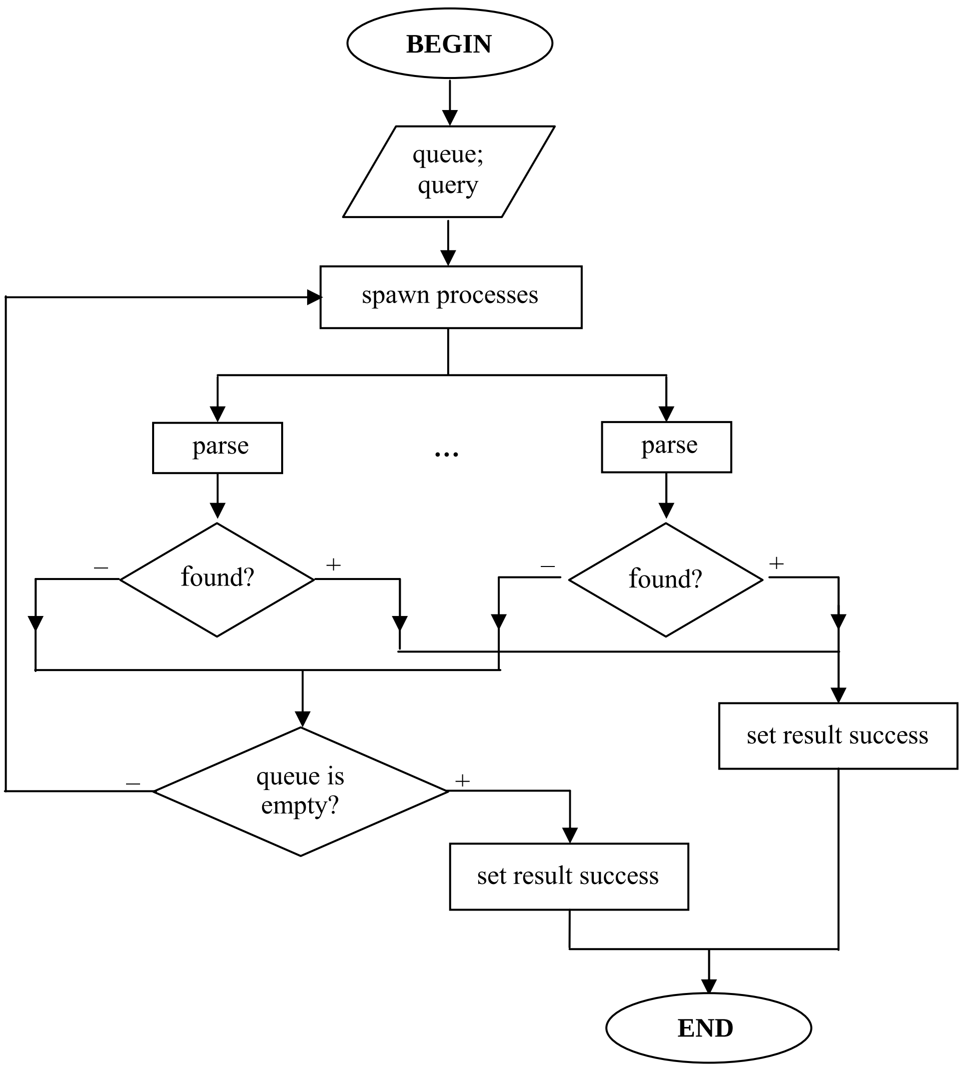


Fig. 1. General search algorithm structure

Parsing is based on a state machine, that reads input symbols one by one and changing its state. We use simple pushdown automata for parsing, because it allows passing the context from one buffer to the next one and allows speculative parsing. Parsing process is shown at fig. 2.

There are two possible ways for a thread to start processing a buffer. If previous buffer was already processed or if it is the first buffer from the beginning of the dataset, we can use the information from previous buffer to determine, what was the previous state and at what state we are beginning.

If we do not know about previous buffer, we still can process the current buffer. We create multiple state machines, one for every possible state. On each step every state machine receives the next character as its input data. For each state machine, if it receives incorrect data and enters error state, it is destroyed. For simple context-free grammars, this process allows to distinguish only one or two possible states without knowing about results of parsing previous buffers.

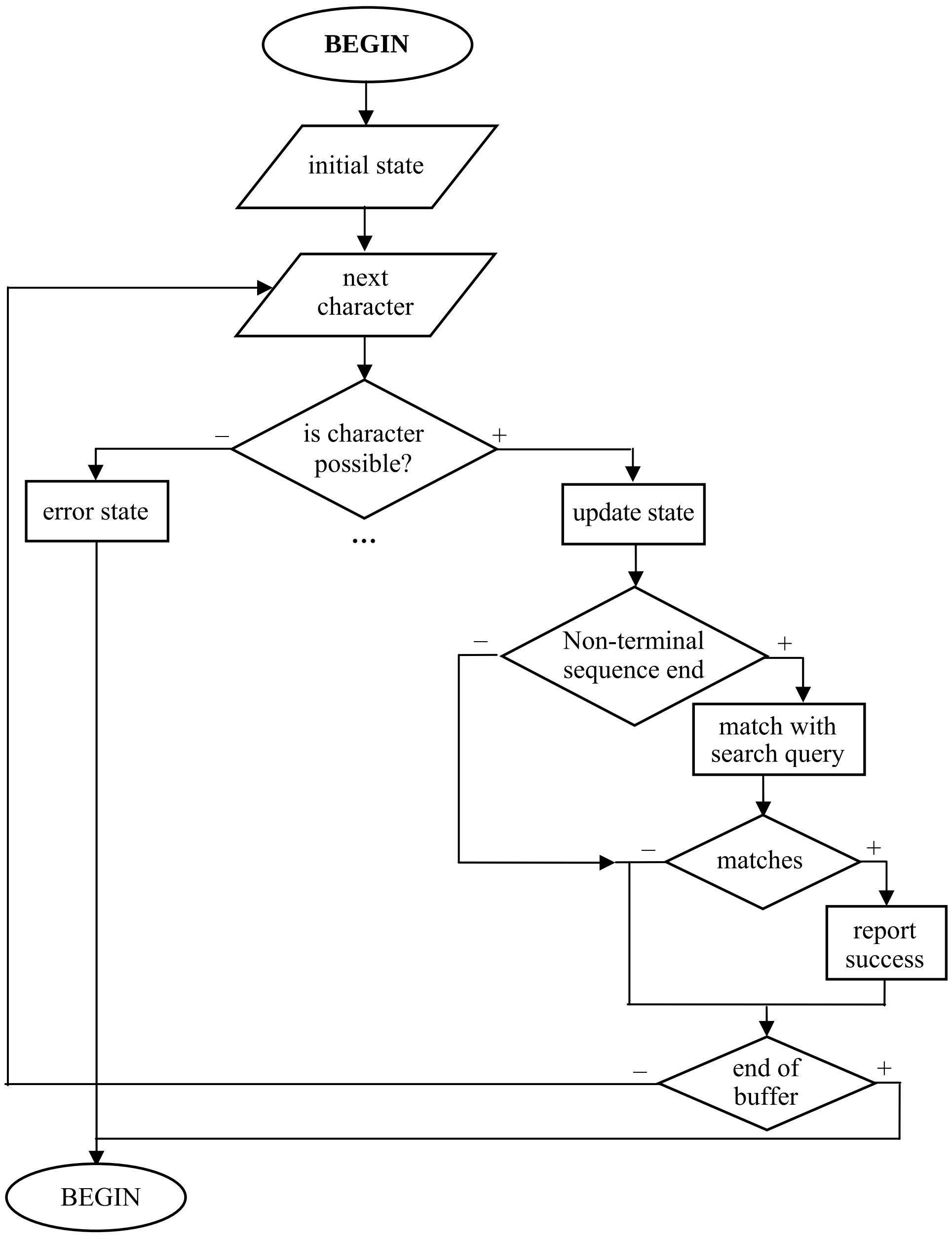


Fig. 2. Parsing process

Search process is executed every time a token is parsed. All the non-terminal character sequences are being tested against the search query. For the first token, if it is not a terminal character, not complete match is also allowed, if it matches the end of a search query. The same applies to the last token, if it matches the beginning of a search query. Match shows that it is possible for this buffer to contain searched data.

After all of the buffer content is processed, and after the result is pushed to the result list, thread has to check if search was successfully completed. This step is only taken if a thread has a full match or an incomplete match at the beginning. The thread has to wait for all the preceding buffers to be processed. If all the search criteria are satisfied by the previous buffers, the thread stops all the other parser threads and returns successful search result. A detailed scheme is shown at fig. 3.

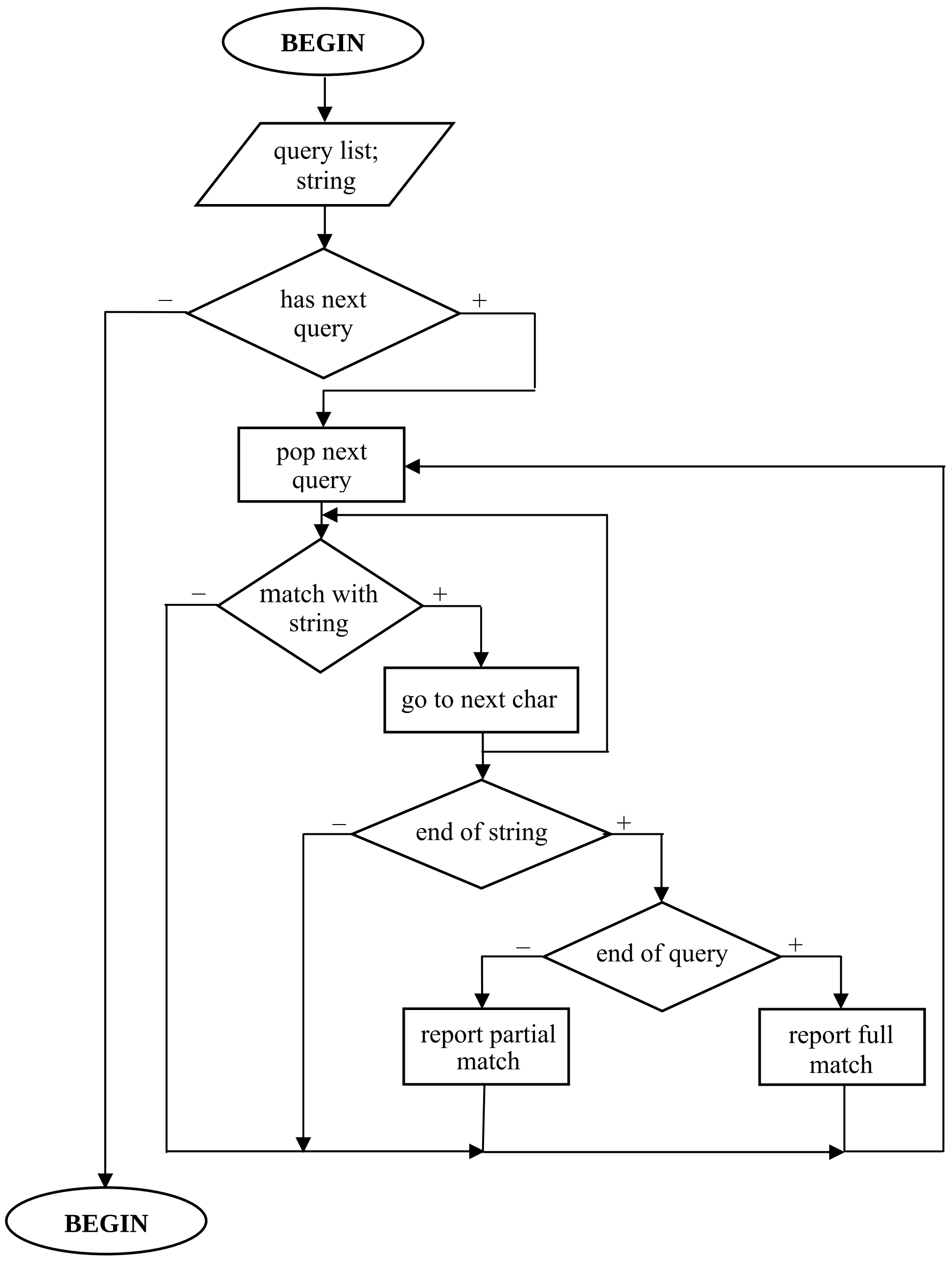


Fig. 3. Full text search process

Result is a data structure, that is able to store all the state machines, that survived (have never entered the error state) during the parsing operation of a buffer. Stack of every state machine for storing uncompleted non-terminal character sequences also should be stored inside result data structure, so it can be used when the next buffer is being processed or after the next buffer has been processed. This way of sharing the state between partitions, processed in parallel, allows us to keep the hierarchy.

Also result data structure has to store the results of search query matching. This information can be represented as a list of discriminated unions (with element count equals to search criteria count) with four possible states:

- no match;
- full match;
- partial match (beginning);
- partial match (ending).

For partial match cases, part of the string that is matched should also be included. For the ending match, we need to keep only index of the match string, because the string itself is already stored in the stack of the state machine.

IMPLEMENTATION FOR XML

To make a research and get the experimental results, we developed an algorithm implementation for the XML language. To define a minimal subset of XML, which can be used for algorithm testing, we need to select element types that are supported in our XML grammar subset [1]. Supported elements are shown in the Table 1.

Table 1. Supported elements in XML implementation

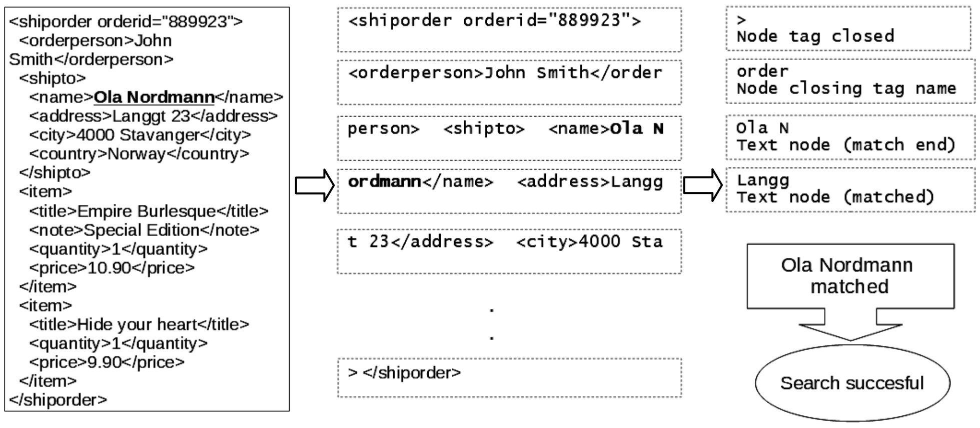
N	Element type	Example
1	Node	<node>
2	Text node	Text node
3	Attribute	attribute=”value”
4	Closing node	</node>

At first, we define a finite state machine for the XML grammar, considering our imitations to keep it simple enough for testing purposes. Then, we define a search query. In many applications, XPath query language [2] is used to define the data to be found. To keep the example implementation simple, only one axis and only one search method of XPath will be used – forward traverse with full text search. The developed algorithm can only be efficient on forward axis, because it processes file in forward direction, from the beginning to the end.

In the example (shown in fig. 4) we use only one search query, a text node with full text matching criteria. The following figure shows the process of search for a text “Ola Nordmann” inside the sample XML dataset. The input dataset is a regular XML document with typical hierarchical structure, where data is represented as text nodes. Text that satisfies the search criteria is located at the beginning of the file, so the algorithm does not need to process the entire file to the end.

After the first processing stage, XML document is split into 15 buffers, containing 30 characters each. It is still just raw XML document, but every buffer is processed independently in parallel by some worker threads, beginning from the first buffer.

After the second execution stage, data is represented as “result” data structures. Each structure contains the stack for a state machine (upper text field) and its state, described in lower text field. Also it contains a match flag. Full text matching with text node started in the third buffer, and completed in the fourth buffer, so there is no need to continue the search process.



```
<shiporder orderid="889923">
  <orderperson>John
Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
  <item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
  <item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
</shiporder>
```

<shiporder orderid="889923">
<orderperson>John Smith</order
person> <shipto> <name>Ola N
ordmann</name> <address>Langg
t 23</address> <city>4000 Sta
> </shiporder>
>
Node tag closed
order
Node closing tag name
Ola N
Text node (match end)
Langg
Text node (matched)
Ola Nordmann matched
Search succesful

Fig. 4. XML processing

Only four buffers were processed, ant it is enough to find the data. If we use four threads, because four cores (physical or virtual) is a popular solution for desktop processors, we can achieve the result of completed search in just one run. It could be almost four times faster than if it is done sequentially. If we are using classic non-parallel parsing algorithm, we need to process all the dataset, all 15 buffers, so developed algorithm can be almost 15 times faster in this particular case in theory.

EXPERIMENTAL RESULTS

For the testing purposes, classical search algorithm from the default .NET platform XML library will be used to compare efficiency of the algorithms on different sizes of the dataset.

Result of the experiment can be seen in the Table 2. Graphically experimental results are shown at the fig. 5. As we can see, the developed parallel search algorithm is faster on the larger datasets, because it works in parallel and it does not need to read the entire file to the end, if the searched node is found.

If the dataset is growing linearly, time spend for the search process is growing linearly as well for both algorithms.

Table 2. Results of the experiment

N.	File size, Mb	Time for the developed algorithm, ms	Time for the library algorithm, ms
1	100	137	108
2	500	242	319
3	1000	398	480
4	2000	514	827
5	5000	843	1746

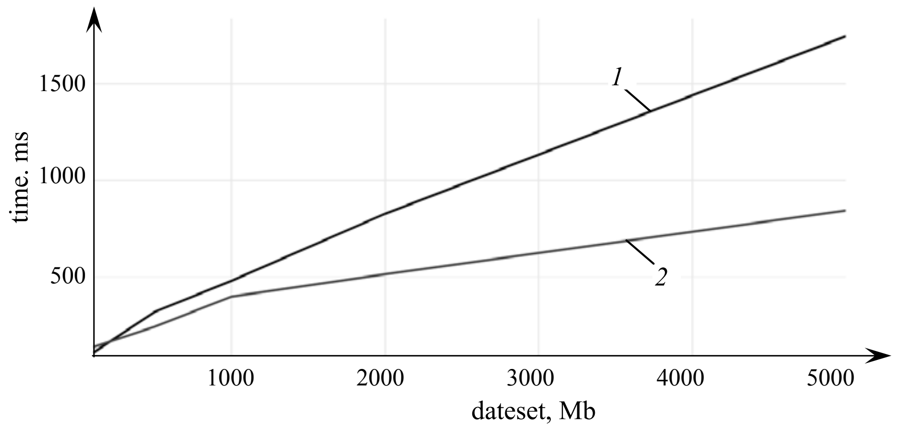


Fig. 5. Experimental results: *1* — library; *2* — developed

We generated a sample XML dataset that includes all of the supported element types. Each name or value field length is between 2 and 20 characters. Element that is being searched is always present in the dataset. Location of the element is randomly generated on each test run.

Classical algorithm can achieve better results on small datasets, because it uses multiple optimizations and more efficient parsing algorithms, than our sample implementation. Also, our algorithm requires a complex initialization stage, with multiple data structures allocation and multiple threads initialization.

For growing datasets, developed algorithm requires less time to find matching data, but there are several conditions that should be met. Also the speed of the developed algorithm depends on searchable element's position. In the worst case possible for the algorithm, when the searched element is not present in the dataset and the entire file should be processed, our algorithm still works faster because of its highly parallel nature, but the difference is less significant.

RELATED WORK

Many methods of parallel text data processing have been presented. Many of them are applicable only to XML processing, especially for XML parsing. Parallel approach is well known in finite automata based parsing methods [3] speculative parsing methods [4]. Most of parallel approaches can be used for context-independent LL(1) formal grammars. XML grammar is a subset of LL(1) grammar, but with some unique differences. Parallel XML processing includes parsing, syntax tree building and XML graph tree building [5]. Modern XML processing methods use special optimization techniques, applicable only to XML format [6].

Parallel depth-first search is widely used in highly efficient searching systems [7]. In this paper, we developed a search algorithm, based on parallel search methods and concurrent formal grammars processing methods for efficient search in any grammar that differ from the existing methods by using the combination of parallel parsing and parallel search in one run.

CONCLUSION

We presented a concurrent algorithm for search in text documents, represented using formal context-independent grammars. The developed algorithm was tested on subset of XML grammar using XPath as query language grammar. Experimental results show that good speeding up was achieved for large-scale datasets.

Speed up is only possible under some constraints. If the speed of read operations is limited by the speed of the disk, and parallel reading is not possible, there is no reason of using highly parallel approach. The developed algorithm will only slow down the process, because of thread management and context switching. Also it is recommended limiting thread count to be less or equal to the physical processing units, to make use of real parallel execution and reduce number of CPU cache misses. Optimal buffer size depends on multiple factors, from CPU cache size and random access memory available to the data representation format, formal grammar and most common text node sizes. To get the best results from using the developed algorithm, it is recommended to configure these parameters manually for each application to meet its requirements.

In comparison to the commonly used search methods, this concurrent heuristic search method demonstrates higher efficiency in terms of execution time, but uses more memory and utilizes more system resources.

To improve the performance characteristics of the algorithm other parsing methods can be used. Dynamic buffer sizes, used alongside with special data splitting algorithm, capable of splitting the dataset by terminal characters, would be great improvement to the developed search algorithm.

REFERENCES

1. *Extensible* Markup Language (XML) 1.0 (Third Edition). — Available at: http://www.w3.org/TR/2004/REC-xml-20040204/. — 2004.
2. *Clark J.* XML Path Language (XPath) Version 1.0. / J. Clark, S. DeRose. — Available at: https://www.w3.org/TR/1999/REC-xpath-19991116/. — 1999.
3. *Chang J.H.* Parallel Parsing on a One-Way Array of Finite-State Machines / J.H. Chang, O.H. Ibarra, M.A. Palis. — 1987. — P. 64–75.
4. *Veillard D.* Libxml2 project web page / D. Veillard. — Available at: http://xmlsoft.org/. — 2004.
5. *Chiu K.* A compiler-based approach to schema-specific xml parsing / K. Chiu, W. Lu. — Available at: https://www.researchgate.net/publication/228586122_A_compiler-based_approach_to_schema-specific_XML_parsing. — 2004.
6. *Noga M.L.* Lazy xml processin / M.L. Noga, S. Schott, W. Lowe. — 2002. — P. 4–7.
7. *Rao V.N.* Parallel depth first search. part 1. Implementation / V.N. Rao and V. Kumar. — 1987. — P. 15–21.

Received 01.08.2018

From the Editorial Board: the article corresponds completely to submitted manuscript.