Computer Systems Science and Engineering

Tech Science Press

DOI: 10.32604/csse.2023.043010

ARTICLE

Performance Enhancement of XML Parsing Using Regression and Parallelism

Muhammad Ali and Minhaj Ahmad Khan*

Department of Computer Science, Bahauddin Zakariya University, Multan, 60000, Pakistan

*Corresponding Author: Minhaj Ahmad Khan. Email: mik@bzu.edu.pk

Received: 19 June 2023 Accepted: 27 October 2023 Published: 19 March 2024

ABSTRACT

The Extensible Markup Language (XML) files, widely used for storing and exchanging information on the web require efficient parsing mechanisms to improve the performance of the applications. With the existing Document Object Model (DOM) based parsing, the performance degrades due to sequential processing and large memory requirements, thereby requiring an efficient XML parser to mitigate these issues. In this paper, we propose a Parallel XML Tree Generator (PXTG) algorithm for accelerating the parsing of XML files and a Regression-based XML Parsing Framework (RXPF) that analyzes and predicts performance through profiling, regression, and code generation for efficient parsing. The PXTG algorithm is based on dividing the XML file into n parts and producing n trees in parallel. The profiling phase of the RXPF framework produces a dataset by measuring the performance of various parsing models including StAX, SAX, DOM, JDOM, and PXTG on different cores by using multiple file sizes. The regression phase produces the prediction model, based on which the final code for efficient parsing of XML files is produced through the code generation phase. The RXPF framework has shown a significant improvement in performance varying from 9.54% to 32.34% over other existing models used for parsing XML files.

KEYWORDS

Regression; parallel parsing; multi-cores; XML

1 Introduction

With its simple textual structure, the eXtensible Markup Language (XML) has become a universal standard for storing data and information transmission between different formats for internet applications. XML is used as a markup language to express data in a hierarchical structure that is parsed for processing by a web application. Enterprises permanently store large amounts of critical business data in XML format. Many native XML database systems have evolved, and almost all prime relational database developers include XML features in their products. Every XML document requires parsing, which uses a lot of CPU time, thereby requiring the development of efficient techniques for XML parsing.

In general, the effectiveness of the XML parsers is determined by the performance of various XML operations. Several software, hardware, and hybrid methods are used to accelerate XML parsing [1,2]. The software-based approaches incorporate techniques such as schema-specific parsing [3],

This work is licensed under a Creative Commons Attribution 4.0 International License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

lazy parsing [4], and pull-based parsing [5]. The hardware-based approaches, in contrast, employ specialized hardware accelerators such as FPGA custom circuits [6] and LTMAdder [7]. Similarly, the hybrid XML parsing approaches [8] use both hardware (such as FPGA) and software parsing techniques.

The Document Object Model (DOM) parsers are widely used for reading and validating XML documents. A DOM parser checks an XML file after reading it against an XML schema. If the XML file is valid, the DOM parser will return a Document object that is represented in memory for random access to an element. The analysis of DOM parsing proves that storing the full tree structure in memory by DOM is inefficient. On the other hand, Simple API for XML (SAX) parsers employ a top-down approach to access XML documents without providing any random access. Similarly, the Streaming API for XML (StAX) extracts information through a pull-based technique, while JDOM is a Java-based library for parsing XML files using a low memory footprint. The JDOM parsers are approximately as fast as SAX but are not suited for large XML documents. Furthermore, many formats including binary XML [9] have been proposed in the literature for improving XML parsing performance.

Parsing and data processing using multicores has become a popular way to accelerate XML parsing. To fully exploit the processing capability of multicore processors, there is a need for programming models that may naturally enable concurrency with effective synchronization mechanisms. To resolve this issue, numerous techniques have been developed to accelerate parsing XML documents using parallel processing, including pre-parsing [10], direct parallel [11], and indexing [12] techniques. However, the performance is still limited for large XML files. The limitations of current parsing techniques create a gap for a more efficient algorithm.

In this paper, we propose a Parallel XML Tree Generator (PXTG) algorithm that aims at the efficient parsing of XML files through the parallel generation of XML trees. This article also suggests the Regression-based XML Parsing Framework (RXPF) that analyzes and predicts performance through profiling and regression, and subsequently, generates final code for efficient XML parsing. The RXPF framework analyzes the performance of various XML parsing algorithms including the PXTG algorithm, corresponding to multiple XML file sizes. The performance prediction through the RXPF framework is used to invoke the best algorithm for parsing that in turn improves the parsing time of XML files.

The remaining paper is structured as follows. Related work is described in Section 2, in terms of parallel XML parsing and regression modeling. The proposed PXTG algorithm used to accelerate XML parsing and proposed the RXPF framework used to analyze and predict performance through profiling, regression, and code generation are described in Section 3. The experimental setup and results obtained by executing the proposed framework and the existing well-known algorithms are described in Section 4. The final section of this document discusses conclusion and future work.

2 Related Work

Serial and parallel techniques have been widely developed and implemented in the existing practices to improve XML parsing. Zhang presented an innovative non-extracted XML parsing methodology known as VTD-XML [13,14]. VTD-XML produced a one-dimensional array by examining the XML document tree that could minimize the memory needs. A four-phase parallel approach for capturing, annotating, and visualizing parallel structures in XML documents was presented by Linghua et al. [15]. While their algorithm was built to visualize similarities as part of plagiarism detection tasks, they believed that a well-planned and integrative solution that separates addressing

match sites and inserting highlight marks can help various applications. Lam et al. [1] compared four distinct XML parsing models DOM, SAX, StAX, and Virtual Token Descriptor (VTD) to determine which applications are best suited for each model. Many researchers used DOM [16] and SAX [17] based techniques to accelerate XML parsing. The DOM-based model defines a variety of interfaces to update and access the contents and structures of XML documents. It returns a tree structure containing all the elements after parsing an XML file. Node, Element, Attr, Text, and Document are the common interfaces used by DOM for manipulating the document and its structure. In contrast to the DOM-based parser, the SAX-based parser is event-based and does not create a fully parsed tree in memory. SAX offers a streaming interface during parsing. This implies that the SAX parser begins scanning the document from beginning to end node while ensuring well-formedness. It invokes the relevant event handler to process each token so that it appears after recognizing tokens. It is a simple process and takes up less memory to process large trees which is a significant advantage. The primary drawback of the SAX parser is that it does not provide random access due to its forward-only operation.

Lu et al. [18] proposed a variety of parallel parsing techniques, with two primary processes known as pre-parsing and full-parsing. Pre-parsing is done to know the structure of the XML document that serves as a guide for the parallel while full-parsing processes that come afterward. The pre-parsing stage is time-consuming because of serial processing giving rise to taking consideration more ideal designs. You et al. [11] suggested an XML parsing optimization method that does not require a pre-parsing phase for parallel parsing. This method applies data parallelism by partitioning an XML document for performing parallel processing over several threads by finding special symbols as delimiters. Similarly, other researchers have focused on parallel parsing to achieve more efficiency in XML parsing [19–21].

Zhang et al. [22] proposed a Map-Reduce approach to handle the Smallest Lowest Common Ancestor (SLCA) based keyword search for continuous XML documents. Their approach processes several XML documents in the Hadoop environment using parallel algorithms. Their design is based on distributed SLCA, in which each net node determines SLCA separately, and just a small amount of data is transferred. Gang et al. contributed to the effect of memory change during XML document parsing. Pan et al. [23] proposed a mechanism called meta-DFA to parallelize the pre-parsing process. Hence, the meta-DFA executes many instances of preparers concurrently. The meta-DFA enables each chunk of XML file to be pre-parsed in parallel. Following the pre-parsing, libxml2 is used to conduct the parallel full parsing. Another approach proposed by Pan et al. [24] to generate a parallel skeleton is simultaneous transducers. It is perceived that these parallel techniques made implementation more difficult and obstructed system optimization. Bessai-Mechmache et al. [25] described a genetic algorithm in their proposal that maximizes the similarity between a group of XML components and the user query. A crowdsourcing-based XML keyword-searching strategy was proposed by Amini et al. [26]. In addition to explaining the basic architecture and details of various sections of the proposed approach, the effectiveness of the new technique is demonstrated through some tests.

Pan et al. [27] proposed another parallel parsing approach based on the SAX model. There are several stages in the parsing process and each stage is connected by a pipeline. This technique uses a streaming method that requires a partition boundaries determination method to be approximately equivalent to skeleton generation. Similarly, Shah et al. [28] suggested another parallel approach to parsing XML documents called ParDOM to increase scalability to parallel XML parsing by Lu. This method uses two phases to create an in-memory tree structure. The first phase is used to divide XML documents into chunks and each chunk parses concurrently. The second phase involves linking all

partial DOM trees together to create the complete DOM tree in parallel. ParDOM provides fine-grained parallelism by offering a flexible scheme for chunking in which each chunk can have an arbitrary number of starting and ending tags. Jianliang et al. [29] offered another parallel parsing technique that utilizes partition on XML document called parallel speculative Dom-based XML parser (PSDXP) which is based on Field Programmable Gate Array (FPGA). They implement their technique on two and four threads by using the Xilinx Virtex-5 board.

Ahmed et al. [7] suggested another efficient XML parsing approach that involves reading multiple bytes and processing various tags simultaneously. The method takes advantage of specialized hardware for single-pass tree generation for enhancing the speed of XML parsing. Hammad et al. [30] discussed XML indexing by applying a cluster-based framework. They have designed a parallel model of storing and querying data on the distributed parallel environment for large XML files. Even though, parallel XML parsing has been the subject of several researchers, the efficiency of parsing large XML remains limited. Therefore, this paper proposes an algorithm has aims at the optimized performance for large XML files in parallel and also proposes a framework that incorporates regression to analyze and predict performance. Regression has been the subject of a lot of recent research for prediction in a variety of disciplines such as health [31], sports [32], stock market [33,34], and weather forecasts [35].

3 Proposed Methodology

This paper proposes the PXTG algorithm for accelerating XML parsing and the RXPF framework to analyze, predict the performance, and generate code for efficient XML parsing, as elaborated below.

3.1 Parallel XML Tree Generator (PXTG)

The PXTG algorithm aimed at efficient parsing takes as input an XML file and produces n subtrees. Before the creation of parallel subtrees, the proposed approach traverses the input file in n parts, serially. The traversal is based on the number of characters in the file. The XML contents are patched (replaced/merged) to bring in a proper format for XML traversal. To ensure the hierarchy of the XML file and the well-formedness for traversal of each part, the XML contents are adjusted with duplicate entries of the required nodes and edges, as shown in Figs. 1 and 2. Subsequently, the well-formed chunks are sent to individual cores for parallel creation of pre-parsed subtrees. The PXTG ends with transformations $T_s\ \forall s = 1, 2, \ldots, n$ that take pre-parsed subtrees and produce corresponding well-formed subtrees τ_i, using the following Eq. (1):

$$T_s:\ (\tau_i, \tau_{i+1})\ \ \tau_i,\ \ \forall i = 1, 2, 3, \ldots, n-1 \tag{1}$$

For the PXTG algorithm, let the XML file be represented by S in string form and the number of chunks is represented by $\acute{C}$ (obtained from user input). The number of characters in the XML file is denoted by Ω. For parsing, multi-processors have been used that are represented by P. Chunk length is represented by weight that depends on $\acute{C}$.

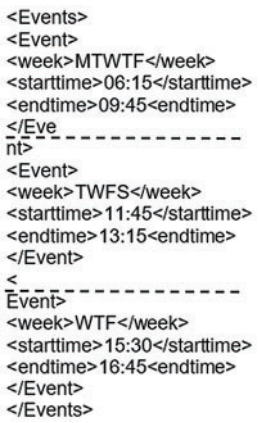

Figure 1: Chunks division of XML file

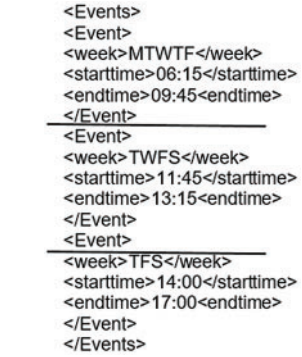

Figure 2: Patching of XML chunks

The PXTG algorithm uses Lexer( ) and SubTreeCreator( ) modules. The Lexer( ) module is used to create tokens and SubTreeCreator( ) module creates n subtrees. The proposed PXTG algorithm is given below:

Algorithm 1: PXTG (S, $\acute{C}$, P, DTD)

Let $\acute{\boldsymbol{C}}$ be the Number of Chunks
Let $\boldsymbol{P}$ be the Processors
Let Ω be the Number of Characters in the XML file
Let $\boldsymbol{S}$ represent the XML file
Let the ***weight*** be the offset
Let ***Partition*[]** and ***Imbalanceddata*[]** be the arrays to store chunks of XML file
Let ***Tokens***[] be the array of tokens generated by Lexer( )

1. $\boldsymbol{weight} \leftarrow \Omega / \acute{\boldsymbol{C}}$
2. **FOR** $\boldsymbol{j} = 1$ to $\acute{\boldsymbol{C}}$-1
   ***Partition*[*j*]** ← chunk of $\boldsymbol{S}$ from $\boldsymbol{weight} * (\boldsymbol{j} - 1) + 1$ to $\boldsymbol{weight} * \boldsymbol{j}$
   END FOR
3. ***Partition*[*k*]** ← chunk of $\boldsymbol{S}$ from $\boldsymbol{weight} * \acute{\boldsymbol{C}} + 1$ to Ω, where $\boldsymbol{k} = \acute{\boldsymbol{C}}$

(Continued)

Algorithm 1 (continued)

4. ***Imbalanceddata***[***m***] = chunk of ***Partition***[***m*** + 1] from (***weight*** ∗ ***m***) + 1 to carriage return, ∀***m*** = 1, 2, . . . , n − 1
5. ***Partition***[***m***] ∪ ***Imbalanceddata***[***m***], ∀***m*** = 1, 2, . . . , n − 1
6. ***Partition***[***m***] ***Partition***[***m***] − ***Imbalanceddata***[***m*** − 1], ∀***m*** = 2, . . . , n − 1
7. **FOR** ***i*** = 1 to ***P***
   P $_i$ ← ***Partition***[***i***]
   Tokens[***i***] ← **Lexer**(***Partition***[***i***])
   SubTreeCreator (***Tokens***[***i***], **DTD**)
   END FOR
8. Compute T_s to create well-formed subtrees using Eq. (1) ∀s = 1, 2, . . . , n

The PXTG (Algorithm 1) starts with the declaration of essential variables $\acute{C}$, P, Ω, S, weight, Partition[ ], Imbalanceddata[ ] and Tokens[]. $\acute{C}$ is used to represent the number of chunks for XML files. P deals with the number of processors used for the parallel creation of well-formed subtrees. The function of Ω is to store the number of characters in an XML file. S is used to represent XML files. Step 1 creates an offset based on the number of chunks received by the PXTG algorithm that is represented by *weight*. As far as step 2 is concerned, it divides the XML file into n-1 parts and stores the chunks in a *Partition*[ ] array, where each part is equal in terms of the number of characters. Step 3 is applied to store the last chunk in the *Partition*[ ] array, where the last chunk has more characters. Step 4, properly handles the imbalanced data of each chunk except the first chunk to accurately store and place them. In step 5, each element of Imbalanceddata[ ] is embedded in the *Partition*[ ] array to get the well-formed chunks. Step 6 is used to maintain the hierarchy by removing imbalanced data from each *Partition*[ ]. Step 7 consisting of three major phases creates pre-parsed subtrees in parallel. The first phase allocates each XML chunk to a separate processor for parallel parsing, in the second phase each processor sends the XML chunk to the Lexer( ) module that returns tokens while the last phase applies the SubTreeCreator( ) module to create parsed subtrees. Step 8 which is the last step of the PXTG performs well-formedness to produce a well-formed subtree by computing Eq. (1).

Algorithm 2: SubTreeCreator (Tokens[], DTD)

1. **FOR** ***t*** = 1 to Length of ***Tokens***[]
2. **IF** Processor ID is 1
   IF ***Tokens***[***t***] is ROOTNODE
   Create ***Tokens***[***t***] as RootNode
   ELSE IF ***Tokens***[***t***] is STARTTAG
   Create ***Tokens***[***t***] as ParentNode
   ELSE IF ***Tokens***[***t***] is DATANODE
   Create ***Tokens***[***t***] as ChildNode
   ELSE IF ***Tokens***[***t***] is ENDNODE
   Create ***Tokens***[***t***] as ParentNode
   END IF
3. **ELSE** //Processor ID is Not 1
   Create RootNode from **DTD**
   Create All Start & Ending Tag from **DTD**
   IF ***Tokens***[***t***] is DATANODE
   Create ***Tokens***[***t***] as ChildNode

(Continued)

Algorithm 2 (continued)

```
            END IF
        ENDIF
    END FOR
```

The Lexer( ) module of the PXTG receives a chunk of XML file on each processor to produce a set of tokens such as start tag, ending tag, data tags, etc., and then stores the tokens in an array for further processing. This is followed by a call to the SubTreeCreator( ) module (Algorithm 2) that takes array of tokens and a DTD file as input to create subtrees. The loop in steps 1 to 3 works for each token to create elements of a subtree. For the first processor, the tokens are generated corresponding to the root node, start tags, data nodes, and end nodes. For the rest of the processors, the root node, start nodes, and end nodes are retrieved from DTD, and the child nodes are created, as mentioned in step 3. The architecture of the proposed algorithm is described in Fig. 3.

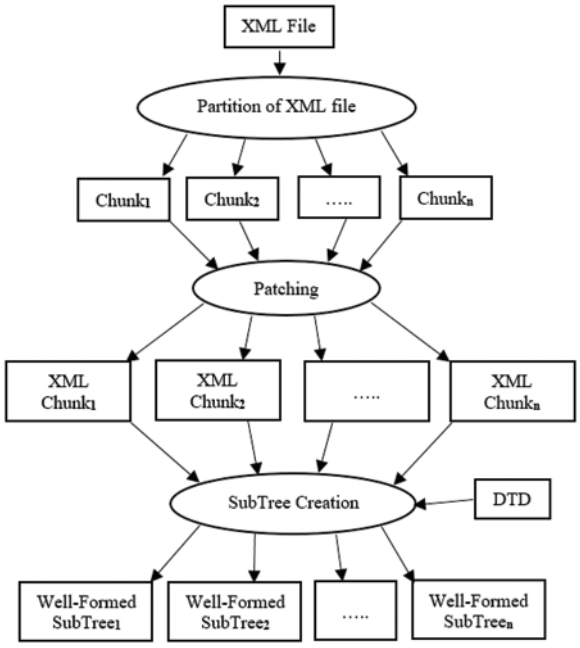

Figure 3: Architecture of proposed algorithm (PXTG)

As shown in Fig. 3, the PXTG algorithm divides the XML file into n chunks after computing the offset. The contents of the chunks are replaced and merged through patching to make them traversable. The functionality of the last phase is to create subtrees while ensuring hierarchy and the well-formedness of the XML file by adjusting entries of the required nodes and edges.

3.2 Regression-Based XML Parsing Framework (RXPF)

The proposed framework RXPF analyzes and predicts the performance of different parsing algorithms through profiling, regression, and then code generation. The framework initially performs the profiling phase and produces the dataset by calculating the parsing time of existing StAX, SAX, DOM, JDOM, PXTG_1, PXTG_4, PXTG_8, PXTG_12, and PXTG_16 algorithms. PXTG_1 treats XML files as a single chunk while PXTG_4, PXTG_8, PXTG_12, and PXTG_16 divide XML files

into 4, 8, 12, and 16 chunks, respectively. The profiling phase is completed by calculating the parsing time of these algorithms by using different XML file sizes. The next phase of the RXPF framework is a regression which creates models of algorithms for prediction. The functionality of the last phase is to generate code for invoking the best algorithm in terms of execution time. The architecture of the RXPF framework is given in Fig. 4.

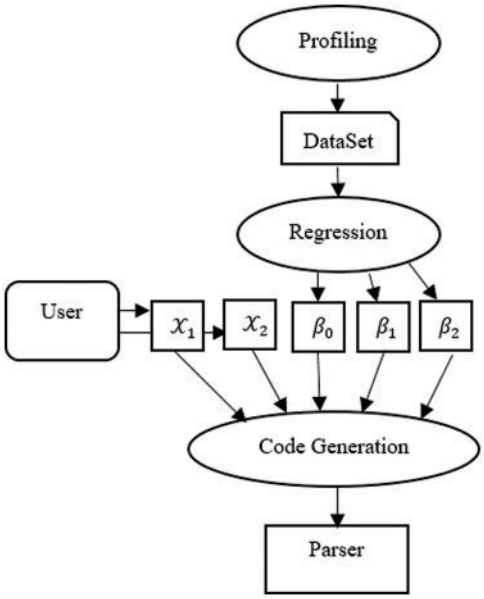

Figure 4: Architecture of proposed framework RXPF

Fig. 4 shows three major components that constitute the RXPF framework. Profiling is the first component of this framework is applied to create a dataset by measuring the processing time of different algorithms. The second component (regression) performs computations based on the linear regression technique to analyze the performance of profiling by producing an array of intercepts that is represented by $\beta_0[]$ and arrays of regression coefficients $\beta_1[]$ and $\beta_2[]$. Code generation, the third and final component, uses values generated by regression and generates the parsing code that invokes the most efficient algorithm corresponding to $\mathcal{X}_1$ (size) and $\mathcal{X}_2$ (cores) values, obtained from users.

3.2.1 Profiling

The profiling is performed by measuring the performance of different algorithms including StAX, SAX, DOM, and JDOM, and variants of the proposed algorithm PXTG_1, PXTG_4, PXTG_8, PXTG_12, and PXTG_16 as shown in Fig. 5. PXTG_1 treats XML file as single chunk while PXTG_4, PXTG_8, PXTG_12, and PXTG_16 divide XML into 4, 8, 12, and 16 chunks.

All algorithms including the variants of the proposed PXTG algorithm are tested on a variety of XML file sizes 10, 15, 20, 25, 50, and 75 MB by using 1, 2, 4, 6, 8, 10, 12, 14, and 16 cores. The proposed algorithm PXTG can divide XML files into several (n) parts for achieving parallelism and accelerating the parsing by using threads. The function of the profiling phase is to produce the dataset for the regression phase. The instances of a dataset are based on file size, number of cores, and parsing time (seconds). The results of profiling for utilized algorithms on 25, 50, and 75 MB file sizes are shown in Figs. 6–8.

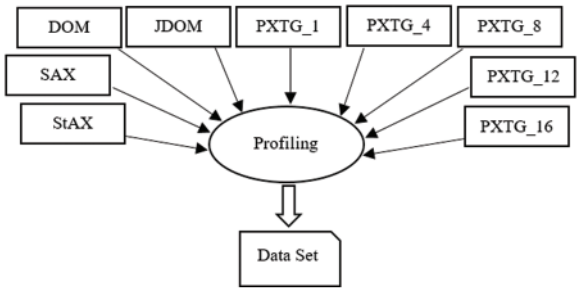

Figure 5: Profiling

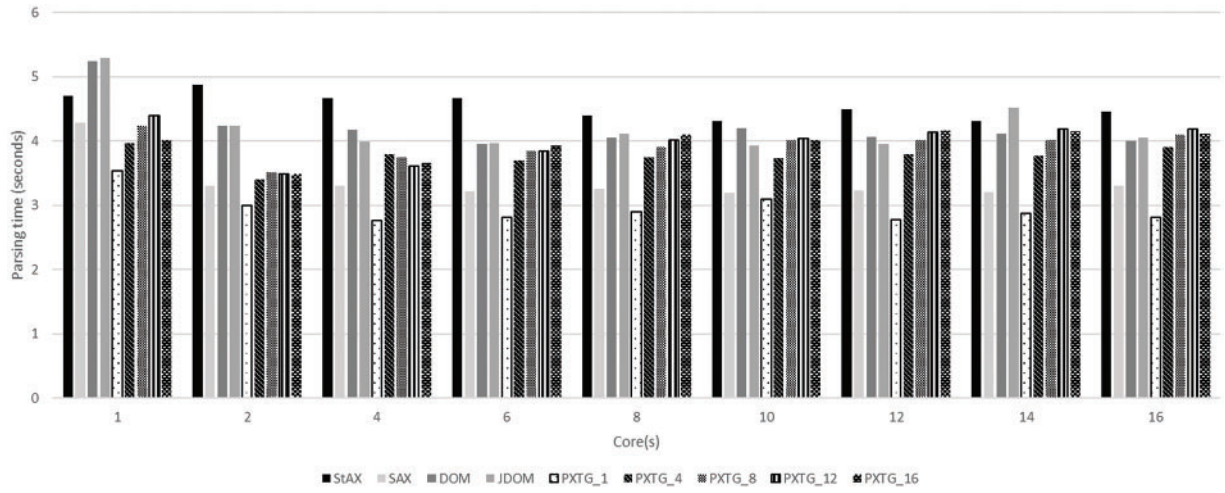

Figure 6: Profiling of algorithms for 25 MB file

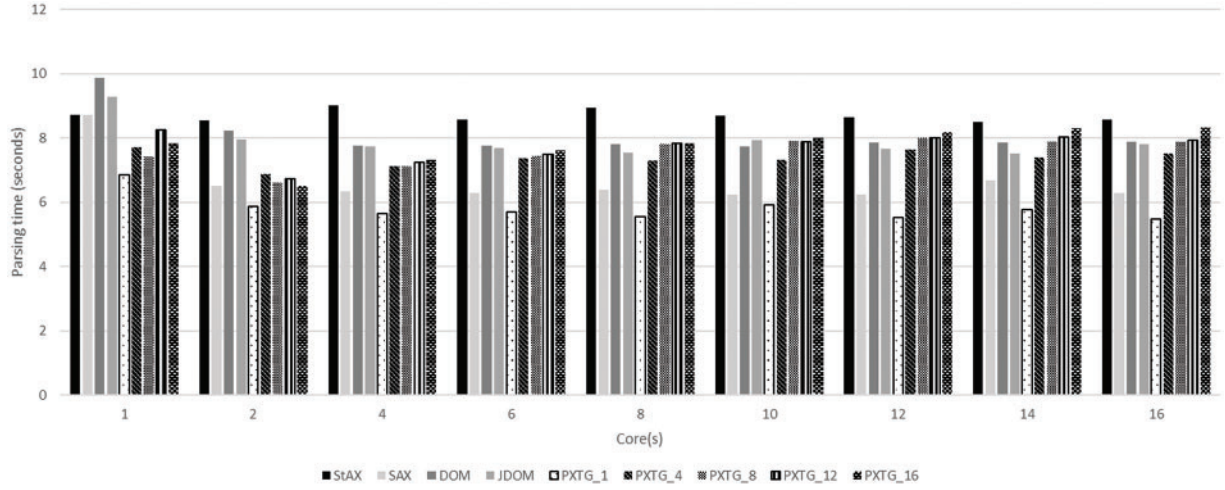

Figure 7: Profiling of algorithms for 50 MB file

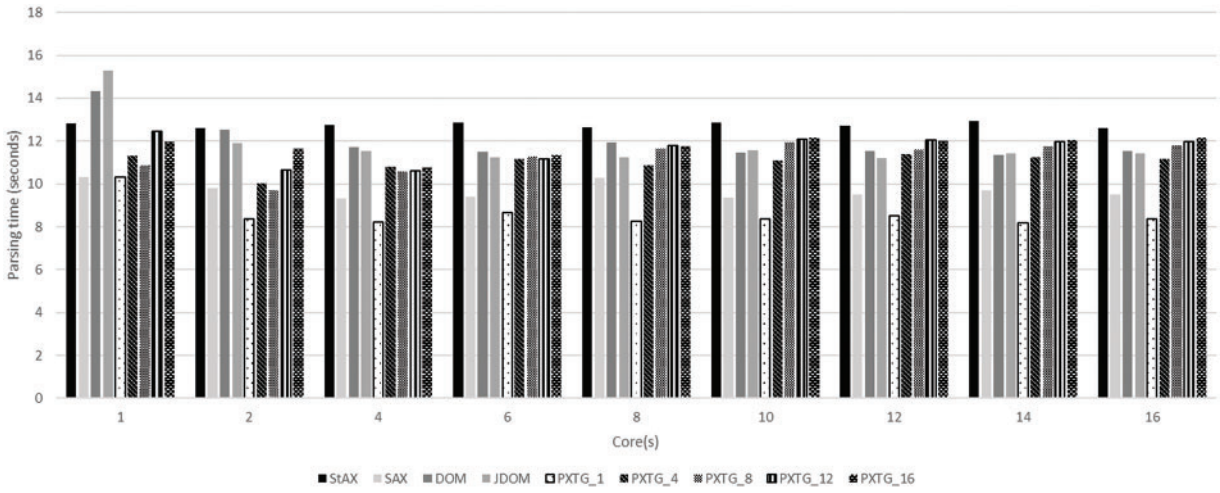

Figure 8: Profiling of algorithms for 75 MB file

3.2.2 Regression

The RXPF framework progresses towards regression, which is a statistical technique to describe the relationship among variables [36,37]. Normally the linear regression technique analyses the relationship between independent and dependent variables, as given below:

$$\hat{\mathcal{Y}} = \beta_0 + \beta_1 * \mathcal{X}_1 + \beta_2 * \mathcal{X}_2 + \ldots + \beta_n * \mathcal{X}_n + \epsilon \tag{2}$$

where $\hat{\mathcal{Y}}$ represents the predicted value, also known as a dependent variable. β_0 describes intercept, which is the predicted value of dependent variables, if all $\mathcal{X}_i$'s are 0. $\mathcal{X}_i$, $\forall \mathrm{i} = 1, 2, \ldots, \mathrm{n}$ represent a list of all independent variables. β_i, $\forall \mathrm{i} = 1, 2, \ldots, \mathrm{n}$ describes the regression coefficient, whereas ϵ describes the predicted error. The regression phase applies multiple regression on a dataset to create a regression model for each algorithm by using Eq. (3). Each model produces an intercept and regression coefficients of dependent variables are reported in Table 1.

$$\hat{\mathcal{Y}} = \beta_0 + \beta_1 * (size) + \beta_2 * (cores) \tag{3}$$

Table 1: Regression models obtained for StAX, SAX, DOM, JDOM, and PXTG

Model	β_0	β_1	β_2	R^2
StAX	0.405	0.167	−0.015	0.998
SAX	0.581	0.126	−0.036	0.987
DOM	0.612	0.156	−0.043	0.990
JDOM	0.649	0.154	−0.043	0.984
PXTG_1	0.415	0.112	−0.029	0.991
PXTG_4	0.107	0.145	0.005	0.997
PXTG_8	0.095	0.147	0.017	0.990
PXTG_12	0.120	0.152	0.011	0.992
PXTG_16	0.068	0.153	0.015	0.993

$\boldsymbol{\beta_1}$ describes the rate of change in $\hat{\mathcal{Y}}$ (performance, i.e., execution time in seconds) by a unit change in $\mathcal{X}_1$ (size), when the effect of another variable $\mathcal{X}_2$ (cores) is kept constant. $\boldsymbol{\beta_2}$ describes the rate of change in $\hat{\mathcal{Y}}$ by a unit change in $\mathcal{X}_2$ (cores), when the effect of another variable $\mathcal{X}_1$ (size) is kept constant, while the performance of the model is measured by R^2. The value of R-square (almost 0.99) shows that the execution time of parsers depends significantly on parameters, as computed by our model R-square is stated by Eq. (4), where $\mathcal{Y}$ represents actual value and $\overline{\mathcal{Y}}$ represents mean of $\mathcal{Y}$.

$$R^2 = 1 - \frac{\sum(\mathcal{Y} - \hat{\mathcal{Y}})^2}{\sum(\mathcal{Y} - \overline{\mathcal{Y}})^2} \tag{4}$$

3.2.3 Code Generation

The RXPF framework uses the values generated by regression and generates the parsing code that invokes the most efficient algorithm for specified XML file size and the number of cores by calling the ParserTemplate( ) module (Algorithm 3), as given below:

Algorithm 3: ParserTemplate ($\mathcal{X}_1, \mathcal{X}_2$)

$\boldsymbol{\beta_0}$[] an array of intercepts
$\boldsymbol{\beta_1}$[] and $\boldsymbol{\beta_2}$[] are the arrays of regression coefficients
1. Computing $\hat{\mathcal{Y}}[\boldsymbol{i}]$ **by** Eq. (2) $\forall i = 0, 1, 2, \ldots, 8$
2. Find an *index* of the model having minimum $\hat{\mathcal{Y}}$ value
3. **IF** *index* is 0
   CALL StAX()
   ELSE IF *index* is 1
   CALL SAX()
   ELSE IF *index* is 2
   CALL DOM()
   ELSE IF *index* is 3
   CALL JDOM()
   ELSE IF *index* is 4
   CALL PXTG_1()
   ELSE IF *index* is 5
   CALL PXTG_4()
   ELSE IF *index* is 6
   CALL PXTG_8()
   ELSE IF *index* is 7
   CALL PXTG_12()
   ELSE IF *index* is 8
   CALL PXTG_16()
   END IF

ParserTemplate( ) (Algorithm 3) receives an XML file size represented by $\mathcal{X}_1$ and the number of cores represented by $\mathcal{X}_2$. Step 1 computes the list of dependent variables of all algorithms by computing Eq. (3). Step 2 returns the index of the predicted algorithm after computing the minimum value from $\hat{\mathcal{Y}}[i]$. This is followed by calling the code of the best algorithm for parsing XML against $\mathcal{X}_1$ and $\mathcal{X}_2$.

4 Experimentation: Setup and Results

Our experimentation has been performed on the proposed framework RXPF. For training, the RXPF framework uses XML files of size 10, 15, 20, 25, 50, and 75 MB. For each XML file, the profiling has been performed by applying 1, 2, 4, 6, 8, 10, 12, 14, and 16 cores. The profiling phase generates a dataset containing file size, number of cores, and parsing time (seconds). The performance results generated after profiling[1] are given in Figs. 6–8. The RXPF framework uses this dataset to create models by applying multiple regression technique. The testing process applies these models on different XML files of sizes 1, 5, 100, and 125 MB while executing code on 4, 8, and 16 cores. Finally, the parser code is generated by the RXPF framework. The architecture used for evaluating the algorithms is given in Table 2.

Table 2: Architecture used by the RXPF framework

Processor	Intel core i7, 2.90 GHz, 64-bit
RAM	16 GB
OS	Ubuntu 20.04.6

The performance results for the file of 1 MB size are given in Fig. 9. For this XML file, the RXPF framework takes 0.24, 0.26, and 0.24 s for parsing, while using 4, 8, and 16 cores, respectively. As Fig. 9 shows, the code generated by the RXPF framework for 4 cores produces speedup of 27.27%, 11.11%, 20%, 27.27%, 4%, and 7.69% over StAX, SAX, DOM, and JDOM, and variants of the proposed algorithm PXTG_1 and PXTG_12, respectively.

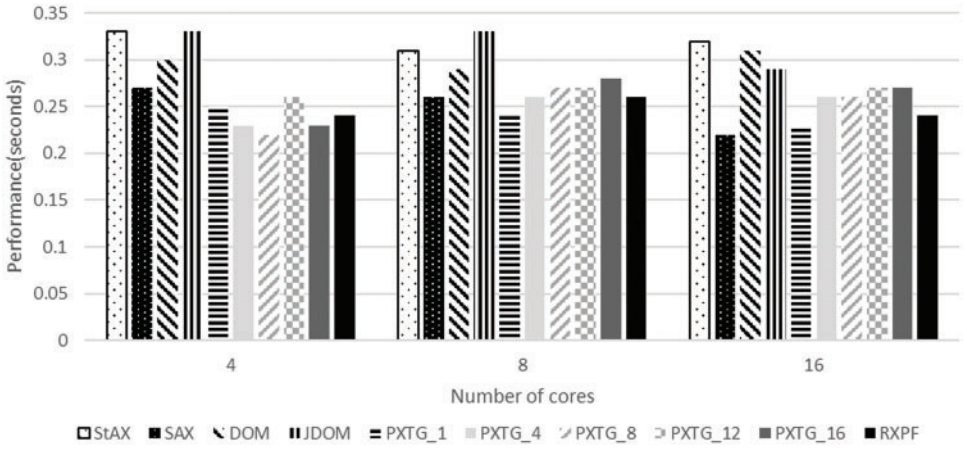

Figure 9: Parsing time for the evaluated approaches using a 1 MB XML file

The code generated by RXPF framework for 8-cores produces speed up of 16.12%, 10.34%, 21.21%, 3.70%, 3.70%, and 7.14% over StAX, DOM, and JDOM, and variants of the proposed algorithm PXTG_8, PXTG_12, and PXTG_16, respectively. The code generated by RXPF framework for 16-cores produces speedup of 25%, 22.58%, 17.24%, 7.69%, 7.69%, 11.11%, and 11.11% over StAX, DOM, and JDOM, and variants of the proposed algorithm PXTG_1, PXTG_4, PXTG_8, PXTG_12, and PXTG_16, respectively. This performance of the RXPF framework is based on the selection of

[1] For profiling, the *taskset* command has been used.

PXTG_8, PXTG_1, and SAX parser selected by the framework as the most efficient algorithm, for 4, 8, and 16, respectively.

The performance results for file a of 5 MB size are given in Fig. 10. For this XML file, the RXPF framework takes 0.73, 0.74, and 0.68 s for parsing while using 4, 8, and 16 cores, respectively. As the Fig. 10 shows, the code generated by the RXPF framework produces average speedup of 29.48%, 14.02%, 25.80%, 25.81%, 12.12%, 16.12%, 19.42%, and 18.70% over StAX, SAX, DOM, and JDOM, and variants of the proposed algorithm PXTG_1, PXTG_4, PXTG_8, PXTG_12 and PXTG_16, respectively.

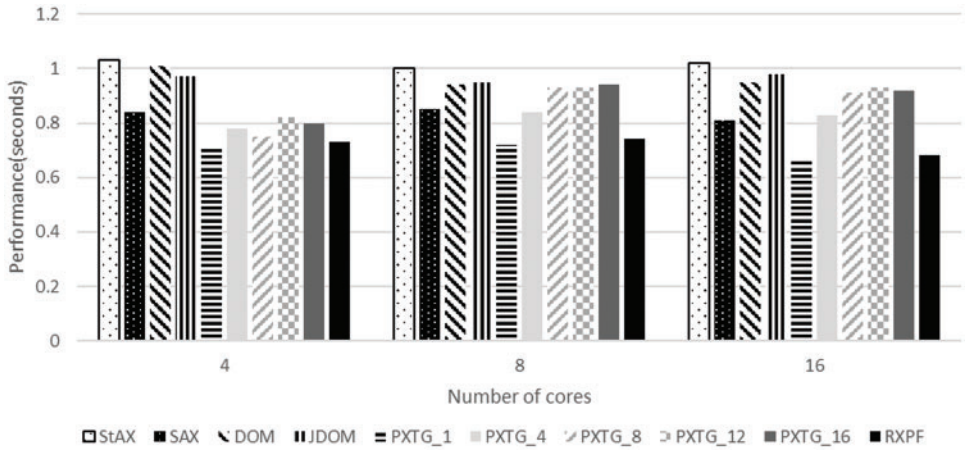

Figure 10: Parsing time for the evaluated approaches using a 5 MB XML file

The performance results for a file of 100 MB size are given in Fig. 11. For this XML file, the RXPF framework takes 11.33, 11.92, and 11.32 s for parsing, while using 4, 8, and 16 cores, respectively. As the Fig. 11 shows, the code generated by the RXPF framework produces average speedup of 32.32%, 11.71%, 25.20%, 21.60%, 22.57%, 24.08%, 25.21%, and 25.83% over StAX, SAX, DOM, and JDOM, and variants of the proposed algorithm PXTG_1, PXTG_4, PXTG_8, PXTG_12, and PXTG_16, respectively.

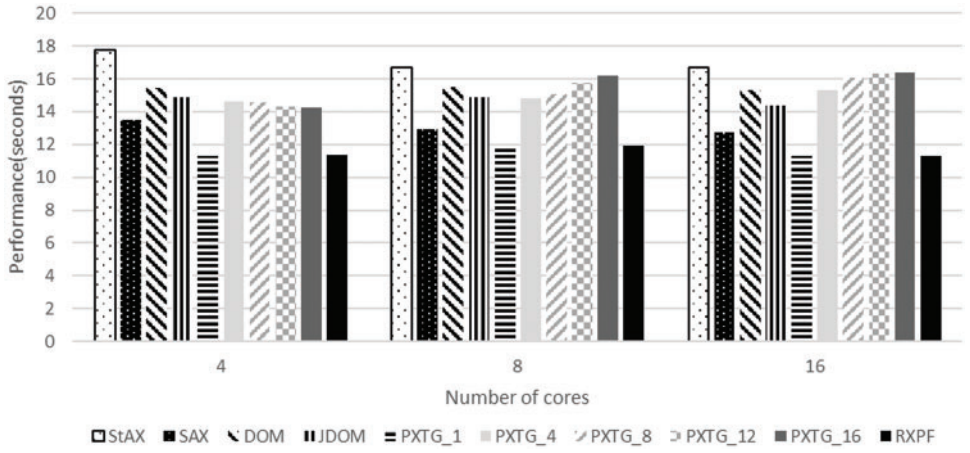

Figure 11: Parsing time for the evaluated approaches using a 100 MB XML file

The performance results for a file of 125 MB size are given in Fig. 12. For this XML file, the RXPF framework takes 13.63, 14.84, and 14.57 s for parsing, while using 4, 8, and 16 cores, respectively. As the Fig. 12. shows, the code generated by the RXPF framework produces average speedup of 32.55%, 7.55%, 26.06%, 22.10%, 22.75%, 23.16%, 24.57%, and 24.96% over StAX, SAX, DOM, and JDOM, and variants of the proposed algorithm PXTG_1, PXTG_4, PXTG_8, PXTG_12, and PXTG_16, respectively.

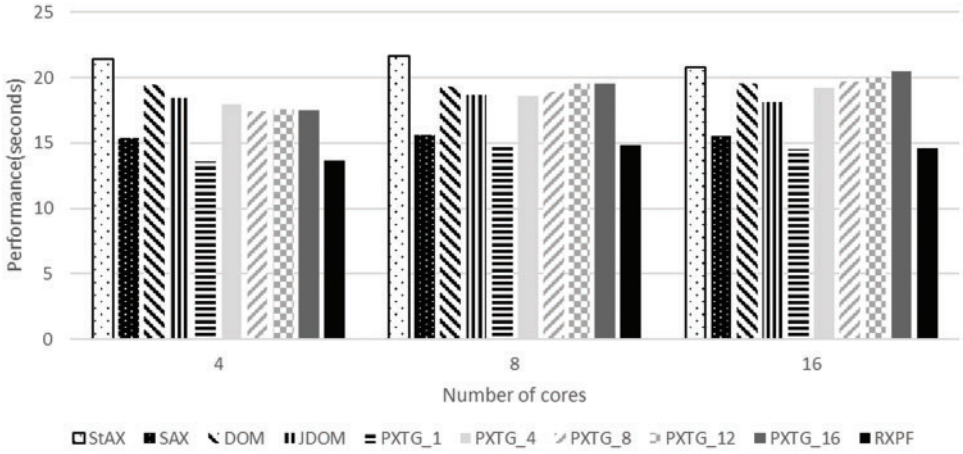

Figure 12: Parsing time for the evaluated approaches using a 125 MB XML file

The average execution time of the RXPF framework over the StAX, SAX, DOM, and JDOM algorithms is given in Fig. 13. As Fig. 13 shows, the results obtained for parsing XML files show that the code generated by the RXPF framework outperforms other algorithms, by producing average improvements of 32.34%, 9.54%, 25.62%, and 22% over the StAX, SAX, DOM and, JDOM algorithms, respectively.

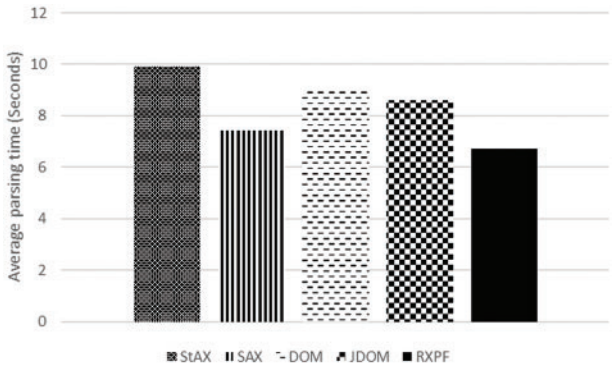

Figure 13: Average execution time for all configurations of file sizes and cores

5 Conclusion and Future Work

This paper proposes a parallel algorithm PXTG to accelerate the parsing of XML files and a framework RXPF that incorporates regression. The PXTG algorithm divides an XML file and

produces multiple well-formed trees in parallel. The RXPF framework analyzes and predicts the performance of different XML parsing algorithms through profiling and regression, and then performs code generation for efficient parsing. The framework produces the dataset for regression through profiling of StAX, SAX, DOM, JDOM, and the proposed PXTG algorithm, while using multiple configurations of XML file sizes and processing cores. The framework generates the parsing code that invokes the best parsing algorithm based on the regression model. To evaluate the performance of the code generated using the RXPF framework, we have performed experimentation for different file sizes and processing cores. The results obtained for parsing XML files show that the code generated by the RXPF framework outperforms other algorithms, by producing average improvements of 32.34%, 9.54%, 25.62%, and 22% over the StAX, SAX, DOM, and JDOM algorithms, respectively. Our future research will enhance XML parsing performance by further exploring and applying neural network techniques for performance prediction. Moreover, currently, our approach is limited to shared memory architectures. In the future, we intend to target distributed memory architecture for parallel parsing.

Acknowledgement: Not applicable.

Funding Statement: The authors received no funding for this study.

Author Contributions: Mr. Muhammad Ali has performed implementation, experimentation, literature survey and writing of the paper draft. The idea and concept of regression and parallelism for XML in this article have been modeled by Minhaj Ahmad Khan, as well as the drafting and critical revision of the article.

Availability of Data and Materials: The data and materials used for experimentation can be accessed via GitHub repository https://github.com/muhammadalimaqbool/RXPF.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] T. C. Lam, J. J. Ding and J. C. Liu, "XML document parsing: Operational and performance characteristics," *Computer*, vol. 41, no. 9, pp. 30–37, 2008.

[2] W. Gang, X. Cheng, L. Ying and C. Ying, "Analyzing XML parser memory characteristics: Experiments towards improving web services performance," in *IEEE Int. Conf. on Web Services (ICWS)*, Chicago, Illinois, USA, pp. 681–688, 2006.

[3] W. M. Löwe, M. L. Noga and T. S. Gaul, "Foundations of fast communication via XML," *Annals of Software Engineering*, vol. 13, no. 1–4, pp. 357–379, 2002.

[4] M. L. Noga, S. Schott and W. Löwe, "Lazy XML processing," in *Proc. of the 2002 ACM Symp. on Document Engineering*, McLean, Virginia, USA, pp. 88–94, 2002.

[5] I. Kaplan, "Processing XML with the XML pull parser," 2006. [Online]. Available: http://bearcave.com/software/java/xml/xmlpull.html (accessed on 01/06/2023).

[6] Z. Dai, N. Ni and J. Zhu, "A 1 cycle-per-byte XML parsing accelerator," in *Proc. of the 18th Annual ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, Monterey, California, USA, pp. 199–208, 2010.

[7] I. Ahmad, S. Patil and S. R. Sarangi, "HPXA: A highly parallel XML parser," in *IEEE Design, Automation & Test in Europe Conf. & Exhibition (DATE)*, Dresden, Germany, pp. 249–252, 2018.

[8] Z. Pan, X. Jiang, J. Wu and X. Li, "Hybrid XML parser based on software and hardware co-design," in *2019 IEEE 27th Annual Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, San Diego, California, USA, pp. 325, 2019.

[9] J. Schneider, T. Kamiya, D. Peintner and R. Kyusakov, "Efficient XML interchange (EXI) format 1.0," Second ed., 2014. [Online]. Available: https://www.w3.org/TR/exi/ (accessed on 26/05/2023).
[10] W. Lu, K. Chiu and Y. Pan, "A parallel approach to XML parsing," in *2006 7th IEEE/ACM Int. Conf. on Grid Computing*, Barcelona, Spain, pp. 223–230, 2006.
[11] C. H. You and S. D. Wang, "A data parallel approach to XML parsing and query," in *2011 IEEE 13th Int. Conf. on High Performance Computing and Communications (HPCC)*, Banff, Alberta, Canada, pp. 520–527, 2011.
[12] K. Song, H. Lu and X. Qin, "An efficient parallel approach of parsing and indexing for large-scale XML datasets," in *2016 IEEE 22nd Int. Conf. on Parallel and Distributed Systems (ICPADS)*, Wuhan, China, pp. 184–191, 2016.
[13] XimpleWare, "VTD-XML: The future of XML of processing," 2015. [Online]. Available: https://vtd-xml.sourceforge.io/ (accessed on 04/06/2023).
[14] J. Zhang, "Simplify XML processing with VTD-XML," 2006. [Online]. Available: https://www.infoworld.com/article/2071745/simplify-xml-processing-with-vtd-xml.html (accessed on 04/06/2023).
[15] M. Beck, M. Schubotz, V. Stange, N. Meuschke and B. Gipp, "Recognize, annotate, and visualize parallel content structures in XML documents," in *2021 ACM/IEEE Joint Conf. on Digital Libraries (JCDL)*, Champaign, IL, USA, pp. 258–261, 2021.
[16] J. Robie, "What is the document object model?," 1998. [Online]. Available: https://www.w3.org/TR/WD-DOM/introduction.html (accessed on 27/04/2023).
[17] D. Megginson, "Simple API for XML," 2004. [Online]. Available: http://www.saxproject.org (accessed on 28/05/2023).
[18] Y. Pan, W. Lu, Y. Zhang and K. Chiu, "A static load-balancing scheme for parallel XML parsing on multicore CPUs," in *Seventh IEEE Int. Symp. on Cluster Computing and the Grid (CCGrid)*, Rio De Janeiro, Brazil, pp. 351–362, 2007.
[19] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.
[20] Y. Zhang, "A parallel XML parsing algorithm based on NEM-XML," in *IEEE 8th Annual Int. Conf. on Network and Information Systems for Computers (ICNISC)*, Hangzhou, China, pp. 437–439, 2022.
[21] X. Li, H. Wang, T. Liu and W. Li, "Key elements tracing method for parallel XML parsing in multi-core system," in *IEEE Int. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, Higashi-Hiroshima, Japan, pp. 439–444, 2009.
[22] C. Zhang, Q. Ma, X. Wang and A. Zhou, "Distributed SLCA-based XML keyword search by map-reduce," in *DASFAA 2010: Database Systems for Advanced Applications,* Tsukuba, Japan, pp. 386–397, 2010.
[23] Y. Pan, Y. Zhang, K. Chiu and W. Lu, "Parallel xml parsing using meta-dfas," in *Third IEEE Int. Conf. on e-Science and Grid Computing (e-Science)*, Bangalore, India, pp. 237–244, 2007.
[24] Y. Pan, Y. Zhang and K. Chiu, "Simultaneous transducers for data-parallel XML parsing," in *IEEE Int. Symp. on Parallel and Distributed Processing*, Miami, Florida, pp. 1–12, 2008.
[25] F. Z. Bessai-Mechmache, K. Hammouche and Z. Alimazighi, "A genetic algorithm-based XML information retrieval model," in *2020 21st Int. Arab Conf. on Information Technology (ACIT)*, Giza, Egypt, pp. 1–5, 2020.
[26] L. M. Amini and M. Keyvanpour, "A crowdsourcing-based approach for efficient XML keyword search," in *2019 5th Int. Conf. on Web Research (ICWR)*, Tehran, Iran, pp. 16–21, 2019.
[27] Y. Pan, Y. Zhang and K. Chiu, "Hybrid parallelism for XML SAX parsing," in *IEEE Int. Conf. on Web Services (ICWS)*, Beijing, China, pp. 505–512, 2008.
[28] B. Shah, P. R. Rao, B. Moon and M. Rajagopalan, "A data parallel algorithm for XML DOM parsing," in *Springer Database and XML Technologies: 6th Int. XML Database Symp. (XSym)*, Lyon, France, pp. 75–90, 2009.
[29] M. Jianliang, S. Zhang, T. Hu, M. Wu and T. Chen, "Parallel speculative Dom-based XML parser," in *IEEE 14th Int. Conf. on High Performance Computing and Communication & 2012 IEEE 9th Int. Conf. on Embedded Software and Systems (HPCC-ICESS)*, Liverpool, UK, pp. 33–40, 2012.

[30] R. Hammad and M. Banikhalaf, "A parallel approach for managing XML-based electronic medical records," in *IEEE/ACS 15th Int. Conf. on Computer Systems and Applications (AICCSA)*, Aqaba, Jordan, pp. 1–5, 2018.
[31] E. Brilliandy, H. Lucky, A. Hartanto, D. Suhartono and M. Nurzaki, "Using regression to predict number of tourism in Indonesia based of global COVID-19 cases," in *IEEE 3rd Int. Conf. on Artificial Intelligence and Data Sciences (AiDAS)*, Ipoh, Malaysia, pp. 310–315, 2022.
[32] G. Bhatambarekar and S. Rai, "Batsmen performance prediction using regression models," in *IEEE Int. Conf. on Artificial Intelligence and Data Engineering (AIDE)*, Karkala, India, pp. 80–85, 2022.
[33] A. Sharma, D. Bhuriya and U. Singh, "Survey of stock market prediction using machine learning approach," in *IEEE Int. Conf. of Electronics, Communication and Aerospace Technology (ICECA)*, Coimbatore, India, pp. 506–509, 2017.
[34] D. Bhuriya, G. Kaushal, A. Sharma and U. Singh, "Stock market predication using a linear regression," in *IEEE Int. Conf. of Electronics, Communication and Aerospace Technology (ICECA)*, Coimbatore, India, pp. 510–513, 2017.
[35] R. Sharapov, "Using linear regression for weather prediction," in *IEEE Wave Electronics and its Application in Information and Telecommunication Systems (WECONF)*, Petersburg, Russia, pp. 1–4, 2022.
[36] A. Ali and S. Wasimi, *Data Mining: Methods and Techniques*. South Melbourne, Vic.: CQUniversity, Thomson Learning Australia, 2007.
[37] L. H. Li, W. J. Cai, D. Y. Ming, X. Zheng, Y. He *et al.,* "Research on the algorithm model of predicting line loss diagnosis based on computer multiple linear regression," in *IEEE 5th Int. Conf. on Information Systems and Computer Aided Education (ICISCAE)*, Dalian, China, pp. 800–803, 2022.