Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 3, Issue. 8, August 2014, pg.379 – 389

RESEARCH ARTICLE

Schema Based Parallel XML Parser: A Fast XML Parser Designed for Large XML Files

Ravi Varma[1], Dr. G. Venkat Rami Reddy[2]

[1]Computer Networks & Information Security & JNT University, India

[2]Computer Science & Engineering & JNT University, India

[1] ravivarma287@gmail.com; [2] gvr_reddi@yahoo.co.in

Abstract— XML is one of the greatest innovations of the digital world. It has taken the field of Web Technology by storm in the past decade and is becoming an ever-present technology in other fields too. XML with its easy usage has lot of future. But the parsing performance of XML is a big hindrance to its development. Particularly, when dealing with huge XML files, normal XML parsers like DOM, SAX parsers are simply not quick enough. But with the emerging multi-core processors, XML performance can be improved by making the XML parsers run in parallel on different cores of the CPU. There are many parallel XML parser implementations but none of them are designed for extremely large files of sizes greater than 50 Megabytes. This paper presents a new faster XML parallel parser model which improves the parsing performance for these large files.

Keywords— XML, XML Schema, Parallel Programming, DOM parsing, Parallel XML Parsing, Queues

I. INTRODUCTION

The advent of web technology also saw the rise of XML as one of the popular data communication standards. The web services showed the world how important XML can be. After which, slowly but surely, XML has crept into other fields of in the modern world of computing [16] Many data formats for applications such as Microsoft Office, LibreOffice, RSS, Atom, XHTML etc. have used XML as its base. Many software development tools such as Eclipse, Netbeans, etc., along with some operating systems like Android have XML as their configuration tool. Another use of XML is that it can function as database for small applications. And for large scale database applications, it has become an intermediary to link up large systems together.

This long list also includes many of the Grid computing applications such as SETI, CERN's LHC etc. Many of these important projects depend on computing resources spread over the world interconnected by a network (the Internet), to process huge amounts of scientific data collected every second of every day. These Grid computing projects, also known eScience applications, use XML to carry the large data around the world.

XML strength lies in its core principles. It is simple, understandable and general. It is very easy to use and anyone can understand the data presented by just looking at an XML document. Adding to the fact that humans understand XML very well, it is also quite easy to program machines to understand it. XML primary advantage that has led to such adoption rates is that it links up isolated, individual and heterogeneous systems without much ado. [15] All the above reasons have contributed to the continued use of XML that has been introduced almost about two decades ago.

Despite its many profound advantages, XML has one serious disadvantage - its performance. [14] It is a known issue that has haunted XML since its inception. The processing of XML files is one of the slowest in this fast moving world. The processing of XML to be usable by a machine is also called as XML parsing. XML parsing is done using some pre-written API provided by many entities. XML APIs are classified into two types: DOM based API and Event based API. DOM based APIs or parsers construct a DOM (Document Object Model) tree in memory which is tree based representation of data present in XML file. These parsers are very slow in parsing but very good at accessing data quickly once parsed. The event based parsers are very quick in parsing in that they do not waste time by building large data structures. Instead, they produce series of call backs which are handled by the application.

Many benchmark tests have been performed on these API. One such test is done by Tak Lam et al. [5] The mentioned benchmark test proved the slow nature of XML parsing. Add to the fact that the parsing performance of XML increases with the increase of XML file size, the situation has not boded well for XML and some went on to predict the doom of XML. Nevertheless, These API have been improving over the years [10] [11] [12] [13] but are not keeping up with increasing demands for speed. This issue becomes troublesome in applications where the size of an XML file is very large. In some applications such as Grid computing, the average size of a single XML document is in between 150 - 250 Megabytes. And in some databases, the XML file size may be as large as 10 Gigabytes.

The turn of the century provided XML a chance to claim some speed advantage with the arrival multi-core processors. The multi-core processors can process many tasks at once and at the same time provide faster processing of a single task by running its individual components in parallel. These new range of processors are utilized by many programmers to keep increasing the speeds of their programs. [6] Many also felt that XML performance can be improved by utilizing the parallel nature of the new CPUs. This paper describes an approach to parallel XML parsing by using multi-core processors to speed it up. This new approach is called as Schema based Parallel XML Parser.

II. RELATED WORK

Much work has been done in parallelizing XML parsing over multi-core architectures. The important point to be noted here is that almost all the work that has gone into parallelising XML parsing is done in data parallel approach. Despite being difficult, this approach is preferred because the efficiency and speed is quite high when compared with pipelining or task parallelism. The first and most famous work is done by Wei Lu et al [1]. In their work, the XML file is divided into chunks and these chunks are fed to different cores of a CPU and are processed concurrently to get the resultant DOM tree. The chunks are obtained by using a preparser which scans the XML document to identify partition points. A block diagram is shown below in Fig. 1.

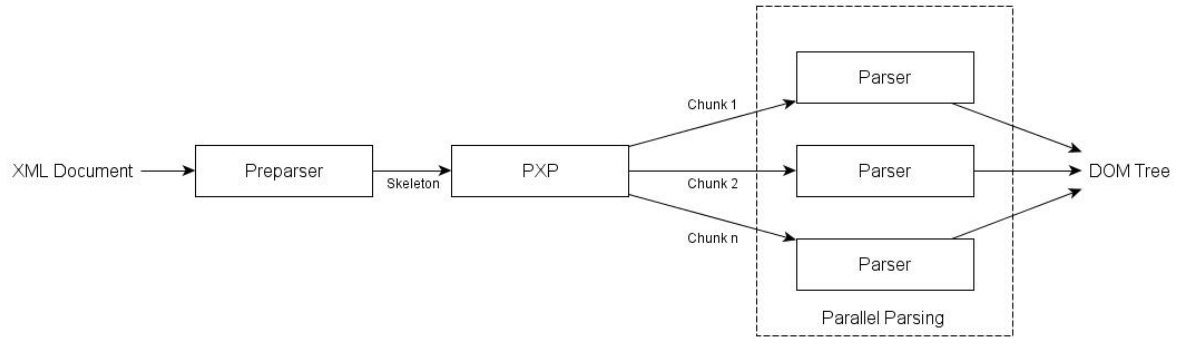


Fig. 1 PXP Parallel XML parser by Wei Lu et al

The preparser is a performance bottleneck in this model as it is serial and has to scan the whole XML document to split apart. To improve this, Pan et al [2] built a Meta-DFA – a DFA based XML preparser to parallelize the slow PXP stage. There are other improvements [17] [18] that have been implemented based on the design of Lu et al [1]. One of them is KEPT (Key Elements Tracing Method) presented by Li et al [3] which introduced task parallelism to data parallelism. Another approach called as ParDOM parallel DOM parser completely removed the PXP stage by using a sort and rearrange algorithm after parsing [4]. The details of the above methods are compiled and written in a survey work by the same authors in their previous work [7].

III. DESIGN OF PROPOSED NEW MODEL

Many parallel XML parsers mentioned in the previous section show performance improvement for files starting from 1 Megabyte to 40 Megabytes but become very slow when it comes to larger files of sizes greater than 50 Megabytes. So for large XML files, a parallel parser called as the Schema based Parallel XML Parser has been designed to support parsing of these very large XML files which are greater than 50 MB in size.

This work concentrates on improving the existing XML parallel parsing performance, particularly at the preparsing stage where a lot of methods have failed to speed it up. The following fig. 2 shows the basic architecture of the new XML parallel parsing model proposed in this paper. The new model like the old model suggested by Lu et al, [1] has three stages. The first is the Preparsing stage; the second is the parallel parsing stage and the final a merging stage. And like all other parallel XML parsers, it is also based on a data parallel approach to speed up parsing.

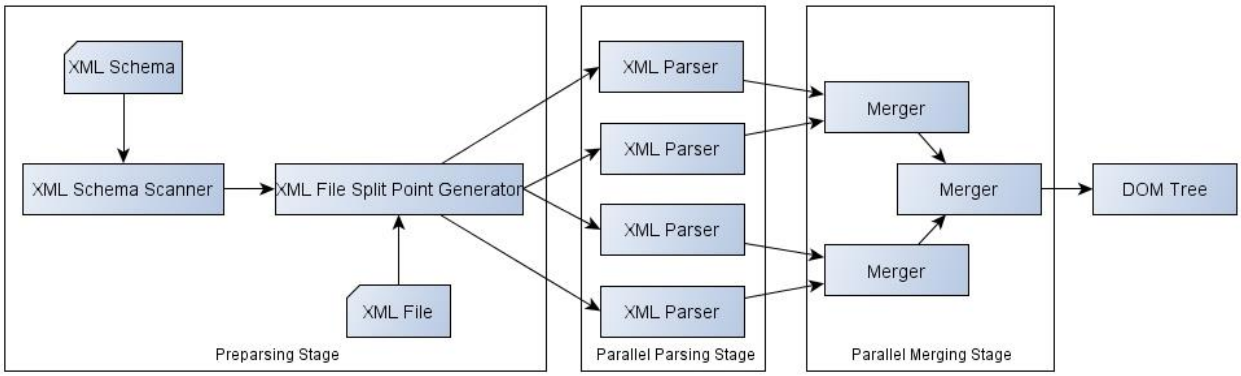


Fig. 2 Schema based Parallel XML Parser model

The job of the first stage, i.e., the preparsing stage, is that to split XML file properly so that the split XML files, also called as Chunks, can be well-formed and readable by the parser. From the second stage, the parallel process begins, where simultaneously many parsers are initiated to parse split XML files concurrently. These parsers give out several DOM objects which are merged in the final phase called as merging phase, which is also run in parallel.

The new Schema based Parallel XML Parser model speeds up the overall process of parsing in two stages. The first one is at the preparsing stage. The PXP approach used to scan the whole XML document to identify break points to split the XML file, the new model will only scan the XML Schema of the XML file to build a list which is used to identify breakpoints in XML file. This process is explained further in the next section. The second place of improvement is the merging phase. Earlier the merging phase used to serial in nature but the new one is parallel improving speed even more. The details are explained in the following sections.

IV. IMPLEMENTATION OF SCHEMA BASED PARALLEL XML PARSER

A. Preparsing stage

The Preparsing stage is the most important stages of this new model as it creates the split points required for proper splitting of XML file. The XML file cannot be split at any arbitrary point; otherwise the parsers in the parallel parsing phase would throw exceptions for non-conformity to well-formedness, premature termination and would be unable to parse those chunks of partial XML files. To eliminate these issues, the XML file has to split at correct points. The correct points would be the ones where the element tags are closed properly. By cutting the XML file at these points, it would ensure proper and safe handling of XML chunks.

To get these properly placed chunks, the PXP approach was to scan the whole XML file. This approach is good enough for smaller XML files which are less than 20 Megabytes in size, but a significant performance bottleneck with large XML files. So to improve speed with higher sizes, the new approach is to scan the Schema and use the schema to split the XML file. The process is shown in the following Fig. 3.

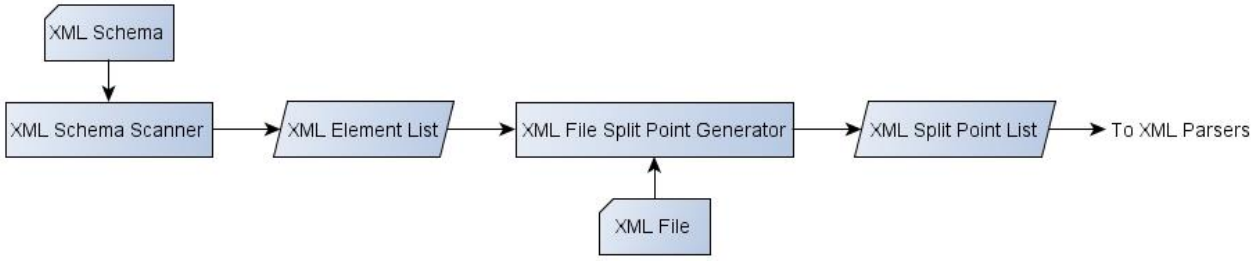


Fig. 3 Preparser Process

Before understanding the XML Schema Scanner algorithm, a point has to be noted, i.e., that this new Parallel Parser model is designed for very large XML files (>50MB). An XML file of a size of 50Megabytes has typically around 100000 – 200000 elements. And it would be reasonable to assume that there are a very large number of Level – 1 elements or nodes in the XML file. Level – 1 elements or nodes are those elements which are direct children of a root node in an XML tree. This is shown in Fig. 4 below where the Level – 1 nodes are shown between the two dotted lines. The advantage of these Level – 1 nodes is twofold. One is that being direct children of root node it is easy to split XML file at the end of their tags and simply insert a root end tag to make well-formed chunk and the second one is it also makes the process of merging XML chunks back together relatively faster. So, the preparser is designed to identify split points at the end of these elements.

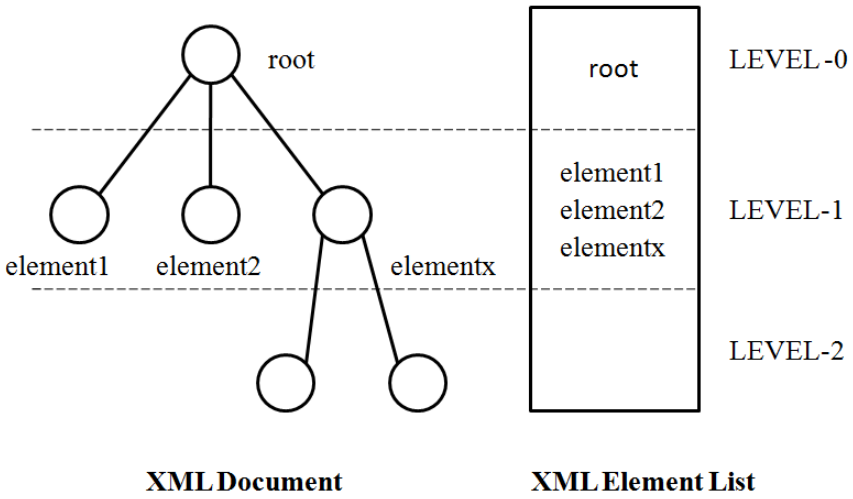


Fig. 4 Sample XML tree indicating Level – 1 nodes

B. XML Schema Scanner

A sample XML Schema is shown in Fig. 5. The XML Schema Scanner reads a Schema file like this and generates an XML Element List to be used by XML File Split Point generator, the next stage in the process. XML schema contains a description of an XML file; the description would be about the structure of the XML file, the XML elements, their names, their type, their constraints, their children, their attributes etc. This means that the XML elements' information can be obtained from the schema itself instead of scanning the whole XML file. This elements' information is stored in the XML Element List. The pseudo code to scan and identify XML elements is shown in Algorithm 1.


```
<?xml version="1.0" encoding="utf-16"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
  <xs:element name="root">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="element1" type="xs:string" />
        <xs:element name="element2" type="xs:string" />
        <xs:element name="elementdeep">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="deep1" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="element3" type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Fig. 5 Sample XML Schema

The XML Schema Scanner takes an XML Schema file as input and gives away a list containing data about Elements in XML file. The algorithm will detect element information by reading the file line by line. It extracts information related to element type and name when it finds *xs:element* tag in the file. This extracted data is stored in *elementlist*. This process is also shown in fig. 4. The namespace *xs* is different from one document to another and so the algorithm will automatically detect it and adopts the new namespace. The XML Element List should contain only Root element and Level–1 elements, so the algorithm uses a variable *level*, an integer which keeps count of the current level the current element is in. The XML Element List, shown in algorithm 1 as *elementlist*, is returned by this procedure.

Algorithm 1 XML Schema Scanner

```
1:  procedure ScanSchema(File xsdfile)
2:      Initialize List elementlist, level = 0
3:      while !EOF of xsdfile do
4:          line ← read line from xsdfile
5:          if line contains "xs:complexType" then
6:              level ← level + 1
7:          end if
8:          if line contains "xs:element" AND level<2 then
9:              elementlist ← element name
10:         end if
11:     end while
12:     return elementlist
13: end procedure
```

C. XML File Split Point Generator

Traditional XML Parallel Parsers have always scanned the whole XML document to generate chunks of main XML file to parse them. This process is a severe headache when it comes to large XML files. To eliminate this problem, the Schema based Parallel XML Parser uses a very different kind of chunk creator. It is called as the XML File Split Point Generator. It takes in the XML Element List produced by the XML Schema Scanner and the XML file as input and generates an array of split points, a series of file pointers, indicating the chunk creation points. The algorithm is shown in Algorithm 2.

Algorithm 2 XML Breakpoint Generator

```
1: procedure GENERATESPLITPOINT(File xmlfile, List elementlist)
2:     chunksize ← (xmlfile file size)/(number of processor cores)
3:     Initialize Array splitpoints[]
4:     while !EOF of xmlfile do
5:         Seek upto chunksize in xmlfile
6:         repeat
7:             line ← read line from xmlfile
8:             if line contains an element in elementlist then
9:                 splitpoints[] ← element ending point in xmlfile
10:            end if
11:        until split point is found
12:    end while
13:    return splitpoints[]
14: end procedure
```

As shown here in algorithm 2, the XML File Split Point Generator also called as XML Breakpoint Generator initially assumes a default chunk size based on the available processor cores. The number of chunks created is equal to the number of CPU cores in the machine. This step is taken so as to ensure maximum efficiency during parallel parsing phase. Then the algorithm seeks directly the end of first chunk in the XML file. This could be any arbitrary point in XML file and the chunk may not be well-formed. This may result in failure of parsing of that chunk. So, the XML Breakpoint Generator automatically reads next few characters so that it can identify the end of an element or node. This is done by continuously comparing the next characters to match the elements in Element List generated by XML Schema Scanner. Whenever a Level-1 element's end tag is detected, the reader stops here and marks the end of the current chunk. The corresponding pointer is written into the array. This process repeats until all the breakpoints are generated. The final array of breakpoints is returned by this procedure.

D. Parallel Parsing stage

The parallel parsing stage is the phase of actual parsing of XML file done. In this stage, a number of parser modules are initialized and run in parallel. These modules are executed concurrently in different cores of a CPU to speed up the overall parsing performance. A block diagram of parallel parsing stage is shown in fig. 6. The figure shows three (say) parallel parsers initialized to run in parallel. These parallel parsers take the XML file as input. But they only parse chunks or parts of the main XML file. The parts are identified in the XML Split Point List. Each parser needs to know only the start and end of its chunk. So, these two points are passed to them. The start of the next chunk is the end of previous chunk. So, a single split point becomes an end indicator of one chunk and also start indicator of the next.

The parser modules use a JDOM parser [8] to build the DOM tree. The DOM tree is a Document object returned by the JDOM parser. A series of DOM trees are given out as output. These partial DOM trees together represent the whole XML file. The algorithm of a parser module is shown in Algorithm 3. The Parser module reads each its chunk into a variable *input*. This is passed to the JDOM parser [8] which parses the input chunk and gives a Document object which is returned by the parser module. The output of the parallel parsing stage is a series or array of Documents (partial DOM trees).

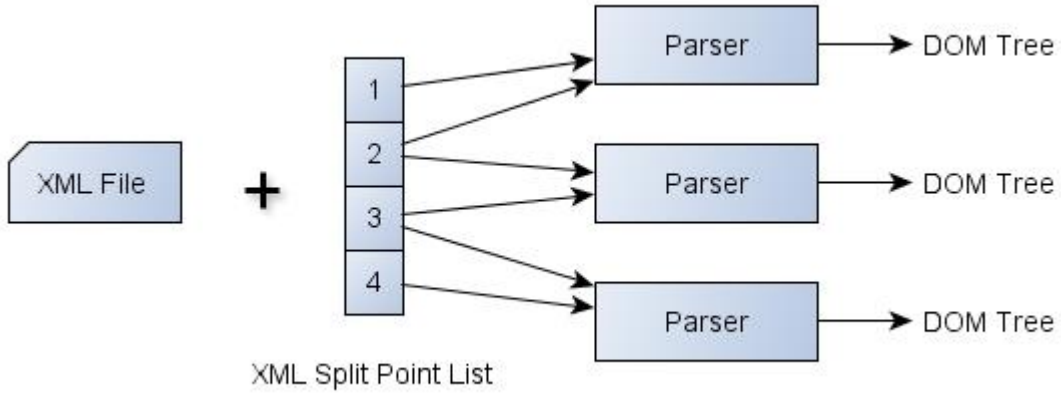


Fig. 6 Parallel Parsing Stage

The Parser modules are initialized simultaneously and are run in parallel. These parsers are started using ExecutorService Framework [9] provided by Java 1.5. Executor Service invokes a thread pool of worker threads [9] where each worker thread runs a parser module. A small code snippet is shown below. An ExecutorService object is created with a pre-assigned number of worker threads. At the same time, a number of Parser modules are initialized and added to a list. These are started by the ExecutorService by using its method `invokeAll()`. Even though, `invokeAll()` might mean to start all parsers at the same time, but in truth, they might not be started all at once. It depends on the number of worker threads that are defined by the ExecutorService object. For example, if there are 4 worker threads and 6 different Parser modules started, the total number of modules running simultaneously is in fact four at any point of time. After finishing execution of any of the Parser modules (whichever task completed first) the new free worker thread picks up a new Parser module to execute. The maximum efficiency is typically obtained when the number of worker threads equals the number of processor cores and the total of available processes is equal to number of worker threads. This is the reason why the number of chunks is defined by the available CPU cores as depicted in algorithm 2 (XML Split Point Generator).

```
ExecutorService executorservice;
List parsers;
parsers.add(new Parser());
.
.
.
executorservice.invokeAll(parsers);
```

Algorithm 3 Parallel Parser Module

1: **procedure** PARSER(File $xmlfile$, Breakpoint $start$, Breakpoint $Stop$)
2: $input \leftarrow$ Read($xmlfile$ from $start$ to $stop$)
3: $document \leftarrow$ Parse($input$) ▷ $document$ is a DOM object
4: **return** $document$
5: **end procedure**

E. Parallel Merging stage

The final stage, i.e., Parallel Merging stage, is setup as optional, in the sense, it is up to the user to decide whether to use the merging stage or not. This is optional because the array of Documents objects produced by Parallel Parsing stage can easily be iterated through and information can be accessed without much trouble as JDOM has sufficient API to provide easy use of a collection of Document objects. For the special cases, the merging stage is included to return a single Document object or a single DOM tree as representation of the whole XML file. So, this final stage in Schema based Parallel XML

Parser model is where all the individual documents that are obtained after the parallel parsing stage are combined together. The process is shown in Fig. 7.

The parallel parsing stage produces a lot of partial Documents, partial because they are representations of chunks of XML file. Though these Documents could still be used on their own individual basis, they are merged for the special range of applications which would require a single DOM Document. These partial Documents are loaded into a Document queue and then passed to a number of Merger modules which are initialized to run in parallel. The first merger module gets the first two Documents by de-queuing the queue ready to be merged and the second merger module gets the next two and the process continues. The combined single Document is queued again at the end of the queue to be ready for further merging process. This process goes on and on until the queue is left with a single Document, then it means that the merging process is done. The remainder document is returned as the Final Document, the whole DOM tree. The algorithmic logic of the Merging stage is shown in Algorithms 4.

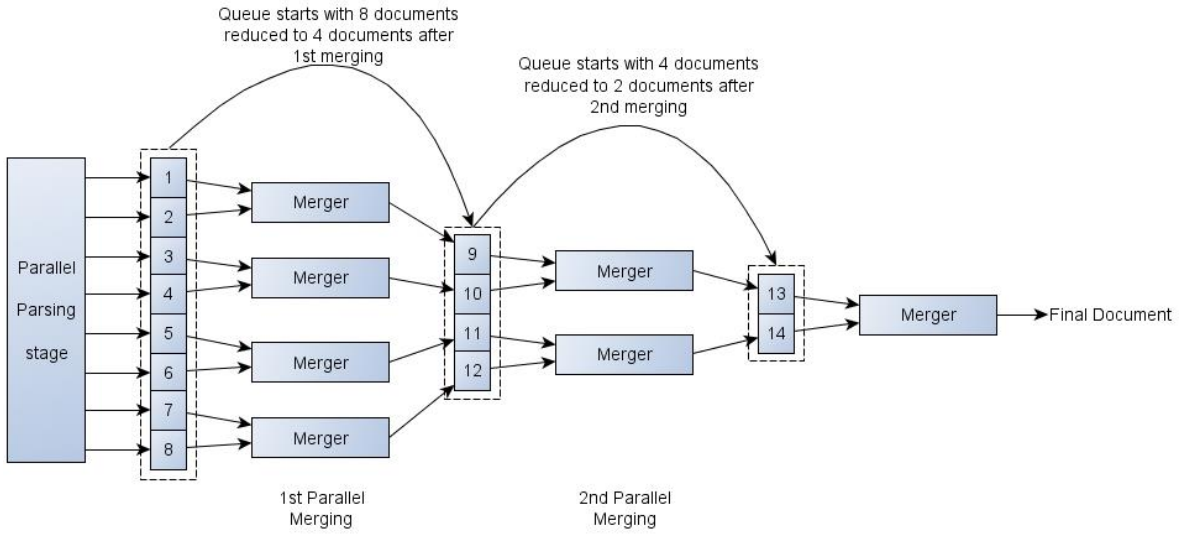


Fig. 7 Parallel Merging stage

Algorithm 4 Merging Stage Algorithm

```
1: procedure MERGE(Queue documentqueue)
2:     while (size of documentqueue) > 1 do
3:         documentqueue ← MergerThread(documentqueue.removeFirstTwo)
4:     end while
5:     return The remainder Document in documentqueue
6: end procedure
```

The Merger module functioning is shown in Algorithm 5. The parallel Merger modules are also initiated and handled by ExecutorService as explained in the previous section. A Merger module takes two Documents as inputs supplied from the Document Queue and then removes content from one Document and adds it to another. These operations are memory intensive and the performance takes a beating. But results have shown that with many parallel Merger modules, the performance is speeded up. They are presented in the next section.

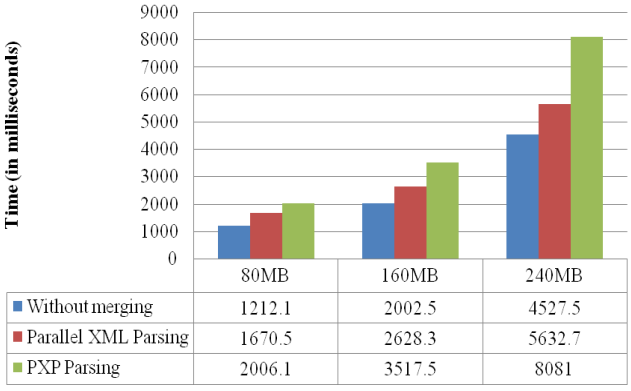
Algorithm 5 Merger Module

```
1: procedure MERGERTHREAD(Document first, Document second)
2:     while second is NOT empty do
3:         first ← second.removeContent
4:     end while
5:     return first
6: end procedure
```

V. EXPERIMENTS

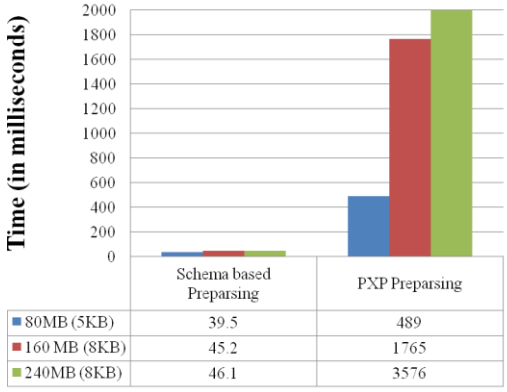
The Schema based Parallel XML Parser is tested on a system with Intel Core - i5 3230M, Quad-core processor over-clocked at 3.2 GHz. It has 8GB of RAM clocked at 1600MHz. The tests are performed for XML files of sizes 80MB, 160MB and 240MB. The performance of the new model is tested against PXP XML parsing on similarly configured system and the results are shown in Fig. 8. The models are tested using JDK 1.8 with preset heap size of 6000MB. This much RAM is required for the Schema based Parallel XML Parser with Merging model. The model without merging doesn't not require as much RAM. But, to get correct results all models are run with 6000MB as default.

The graph shows figures for Schema based Parallel XML Parser without Merging, Schema based Parallel XML Parser with Merging and PXP XML Parsing [1]. The values are obtained by taking average of 10 independent runs. The Schema based Parallel XML Parser without Merging is clearly the fastest with an average execution time of 1.21 seconds for an 80MB file, 2 seconds for 160MB file and 4.5 seconds for 240MB file. It shows an average speed improvement of 42% over PXP XML Parsing. The Schema based Parallel XML Parser with Merging is second fastest with an average execution time of 1.6 seconds, 2.6 seconds and 5.6 seconds for 80MB, 160MB, and 240MB respectively. This produces an average speed improvement of 24% over PXP XML parsing.



	80MB	160MB	240MB
Without merging	1212.1	2002.5	4527.5
Parallel XML Parsing	1670.5	2628.3	5632.7
PXP Parsing	2006.1	3517.5	8081

Fig. 8 Performance comparison between Parallel XML parsers



	Schema based Preparsing	PXP Preparsing
80MB (5KB)	39.5	489
160 MB (8KB)	45.2	1765
240MB (8KB)	46.1	3576

Fig. 9 Preparser Performance Comparison

The next graph in Fig. 9 shows performance improvement in the Preparsing stage from PXP Parallel XML parsing and Schema based Parallel XML Parser. The system used to test these is Intel Core i5 2430M @ 2.4 GHz Processor with 4GB of RAM. The new Schema scanning approach is clearly faster than a PXP approach. This is because the Schema file is independent of the size of XML file and very small in size. The PXP performance decreases with increase in XML size but Schema based Scanner's performance isn't affected by XML size. For a XML file size of 160MB, the PXP approach takes about 1.7 seconds but the Schema based approach takes only 45 milliseconds. These figures prove that the new Schema based Parallel XML Parser is faster than PXP parallel XML parsing.

VI. CONCLUSION AND FUTURE WORK

A new model called Schema based Parallel XML Parser is presented in this paper. The model is designed to be an improvement over other Parallel XML Parsing algorithms, particularly for extremely large XML files of sizes greater than 50MB. It has two improvements over PXP Parallel XML Parser, the very first attempt at parallel XML parsing. Firstly, it has Schema based Preparser which showed a performance improvement that is over 10 times fast. Secondly, it has a parallel Merging phase, a significant improvement of sequential merging

present in PXP approach. These two improvements propelled the Schema based Parallel XML parser to be around 40% faster than PXP approach when tested on files of sizes between 50MB to 250MB.

There is a lot of room for improvement in parallel XML parsing. Newer schemes that can be implemented include using the recursive Fork-Join Framework presented in Java 7 to increase parsing efficiency. Merging stage is improved by concurrent execution of multiple Merger modules on Documents. This can be improved by providing parallelism at Element level inside the Document. A parser with no well-formedness checker could be used to speed up parsing stage.

REFERENCES

[1] W. Lu, K. Chiu, and Y. Pan. "A parallel approach to XML parsing" *The 7th IEEE/ACM International Conference on Grid Computing*, Barcelona, September 2006.

[2] Yinfei Pan, Ying Zhang, Kenneth Chiu, and Wei Lu "Parallel XML Parsing Using Meta-DFAs", *3rd IEEE International Conference on e-Science and Grid Computing*, Bangalore, India, December 10-13, 2007.

[3] Xiaosong Li, Hao Wang, Taoying Liu, Wei Lu, "Key Elements Tracing Method for Parallel XML Parsing in Multi-core System", *International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2009.

[4] Bhavik Shah, Praveen R. Rao, Bongki Moon, and Mohan Rajagopalan, "A Data Parallel Algorithm for XML DOM Parsing", *Database and XML Technologies, volume 5679 of Lecture Notes in Computer Science*, pages 75-90. Springer Berlin / Heidelberg, 2009.

[5] Tak Cheung Lam, Jianxun Jason, DingJyh-Charn Liu, "XML Document Parsing: Operational and Performance Characteristics Computing Practices", *IEEE Computer Society*, 2008.

[6] Herb Sutter, "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software", *Dr. Dobb's Journal*, March 2005.

[7] Ravi Varma, Dr. G. Venkata Rami Reddy, "A Review of Parallel XML Parsing Techniques", *International Journal of Computer Science and Mobile Computing*, April 2014, pages 1149 – 1154.

[8] (2013) JDOM Website [Online]. Available: http://www.jdom.org/

[9] (2014) Java Executor Service Framework Tutorial on Oracle Java Tutorials [Online]. Available: http://docs.oracle.com/javase/tutorial/essential/concurrency/executors.html/

[10] K. Chiu, T. Devadithya, W. Lu, and A. Slominski, "A Binary XML for Scientific Applications", *International Conference on e-Science and Grid Computing*, 2005.

[11] K. Chiu and W. Lu, "A compiler-based approach to schema-specific xml parsing", *The First International Workshop on High Performance XML Processing*, 2004.

[12] W. Zhang and R. van Engelen, "A table-driven streaming xml parsing methodology for high-performance web services", *IEEE International Conference on Web Services (ICWS'06)*, pages 197–204, 2006.

[13] R. van Engelen, "Constructing finite state automata for high performance xml web services", *Proceedings of the International Symposium on Web Services(ISWS)*, 2004.

[14] Jeff Atwood. (2008) XML: The Angle Bracket Tax on Coding Horror. [Online]. Available: http://blog.codinghorror.com/xml-the-angle-bracket-tax/

[15] B. Tommie Usdin. (2014) How and Why Are Companies Using XML? On Mulberry Technologies Inc. [Online]. Available: http://www.mulberrytech.com/papers/ HowAndWhyXML/

[16] (2005) XML, Cover Pages on Oasis [Online]. Available: http://xml.coverpages.org/xmlApplications.html/

[17] Y. Pan, W. Lu, Y. Zhang, K. Chiu, "A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs", *Seventh IEEE International Symposium on Cluster Computing and the Grid(CCGrid'07)*, 2007.

[18] Pan, Y., Zhang, Y., Chiu, K., "Simultaneous transducers for data-parallel XML parsing", *Proc. of Intl. Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–12 (2008).

BIOGRAPHY

Ravi Varma is currently pursuing his M.Tech degree in the Computer Networks and Information Security Department, School of Information Technology, JNT University, Hyderabad, India. He received his Bachelor of Technology degree in 2012 from CMR College of Engineering & Technology, Hyderabad, India. His research interests are Distributed and Parallel computing, Android Programming, Information Security, Computer Networks.

Dr. G. Venkat Rami Reddy is presently Associate Professor in Computer Science and Engineering at School of Information Technology. He is more than 11 years of experience in Teaching, and Software Development. His areas of interests are: image Processing, Computer Networks, Analysis of Algorithms, Data mining, Operating Systems and Web technologies.

