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. 4, April 2014, pg.1149 – 1154

REVIEW ARTICLE

A Review of XML Parallel Parsing Techniques

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

[1]Computer Networks & Information Security, JNT University, Hyderabad, India
[2]Computer Science & Engineering, JNT University, Hyderabad, India
[1] ravivarma287@gmail.com; [2] gvr_reddi@yahoo.co.in

Abstract— XML is a popular markup language which is used because of its simplicity, generality, and usability. As a result, it is widely used in modern computing. It has also become the backbone of Web Services. However, when it comes to parsing an XML file, there is a serious issue of speed and efficiency. There have been many methods developed to speed up parsing. Modern techniques of speeding up XML parsing include utilizing the multiple cores of present day processors by following a parallel approach. This review describes a set of parallel techniques applied to DOM parser and SAX parser to improve parsing speed. It also inspects the differences among the various techniques developed and highlights the improvements made from one to another.

Keywords— XML; XML parsing; DOM; Parallel programming; Finite Automata

I. INTRODUCTION

XML is a phenomenon in web services. It has dominated the web industry ever since it was introduced, so much so that it is being called the *lingua franca* of web services. It has also crept into many other forms of modern computing such as Grid Computing where majority of eScience applications use XML as a communication or data interchange standard. Its success is attributed to its advantages – simplicity, usability and more than anything else – its generality which helps anyone, man or machine, understand easily. Despite its many advantages, it has its own drawbacks, with the biggest one being the performance.

One of the serious issues with XML is the time utilized to parse an XML file. XML has been dogged with this problem since its inception. Parsing an XML file involves analysing the file to extract and build a structure which makes the XML data usable and available to applications and programs. An XML parser is a program that does the job of parsing an XML document. There are two major types of XML parsers – event based (SAX) parser and DOM based parser. DOM (Document Object Model) parser as the name suggests builds a tree structure containing all the XML file data. Whereas SAX (Simple API for XML) produces a series of events or callbacks at the start and end of an XML tag and lets the user handle them.

Event based parsers are very quick in parsing, take a lot less memory and are less CPU intensive than DOM parsers but a DOM parser creates a tree structure which helps extremely fast random access of data. Tak Lam *et al*. [8] tested various parsers on a single machine where SAX parser was able to parse any 20 Megabytes of data per second whereas DOM managed only 5-10 Megabytes per second depending on file size. There is a type of parser which tries to take advantages of SAX and DOM parser and combine into one. It is called StAX (Streaming API for XML) or Pull parser. It provides faster parsing as well as random access.[9]

There have been many improvements [13, 14, 15, 16] to these parsers but none of them are really quick enough to meet the fast demands of modern day computing. So, the next turn to improve parsing has led to utilization of power of present day processors. Processor development took a completely approach at the start of

the millennium when instead of increase in clock frequency, the development turned to multi-core processors, i.e., two or more processors in one chip.[12] This turn helped in development of multi-threaded and parallel computing. XML parsing has also jumped on this bandwagon and many developments have taken place in trying to create parallel XML parsers. This paper outlines these parallel parsing techniques and the advances made in this area that have been developed over the last decade.

II. LITERATURE SURVEY

Parallelism can be obtained in three ways – pipelining, task parallelism and data-division. Pipelining involves cutting a sequential process into various stages and running these stages on different cores of a CPU. This is often hard as there are significant problems with synchronization, memory access etc. A task parallelism approach involves dividing the process into different tasks and executing them concurrently. The difference between pipelining and task parallelism is that in pipelining the output of current stage is taken by the next stage, but in task parallelism, the tasks are executed in parallel where one task is not dependent on the execution of the other. Task parallelism suffers from inflexibility and low adaptability. On the other hand, a data-division approach involves cutting the input data into many parts and these parts are handled by different but simultaneously executing threads running on different cores of a CPU.

In parallel XML parsing by data-parallel approach, the XML file is cut into many parts or chunks and these chunks are handled by a different threads running simultaneously. This approach is preferred for XML parsing and almost all the work related to parallel parsing is done based on the same. This review paper explores various parallel programming methods developed to improve construction of DOM tree of XML document and also those which provide SAX callbacks for SAX events.

III. PARALLEL DOM PARSING

All the research that so far has been done to parallelize DOM parsing has focused on data parallel approach. Despite being the hardest way, there has been significant improvement to performance of DOM parsing. This is particularly preferred because DOM parsing is used in areas where fast random access of XML data is important also the whole XML document is available at a single place.

A. Parallel XML Parser (PXP)

Y Wei Lu *et al* [1] first proposed a data based approach to parallel parsing of XML utilizing multi-core processors which they called as PXP (Parallel XML Parser). In their ground breaking work, the XML document was divided into chunks or parts and they are fed to different threads running on different cores of a multi-core CPU. These threads run simultaneously creating a state of concurrency in parsing an XML document. The thread outputs are then joined together after parsing. A block diagram of the system is shown below in fig.1.

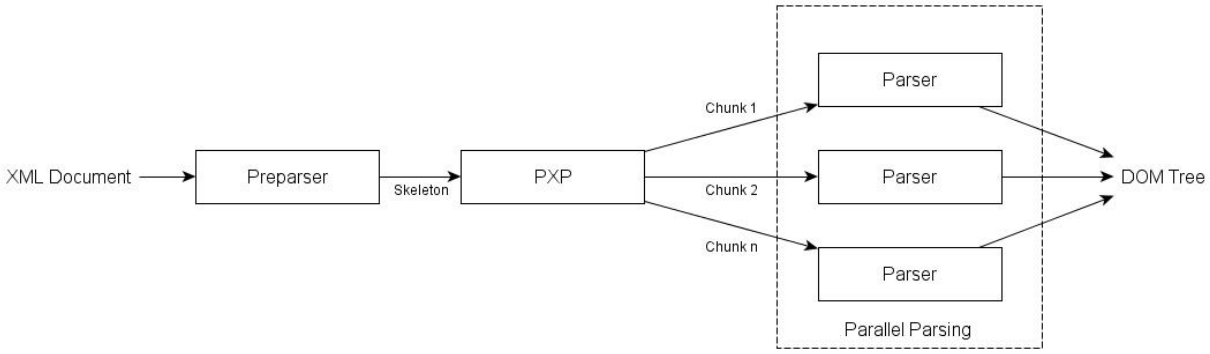

Fig. 1 Parallel XML Parser (PXP) model presented by Wei Lu *et al*

In the figure, there are three stages in parallel parsing. The first is the Preparsing stage which takes an XML document as input and produces a skeleton of the XML file. The skeleton is given to PXP stage (the second stage) which is responsible for division of the XML document into chunks. The chunks are given to third and final stage which is the parallel parsing phase where concurrently executing threads parse the XML chunks and give the output DOM tree structures which are joined together.

1) Preparsing and PXP:

Even though XML is, at core, a sequence of characters, it is actually a serialized tree structure. So when dividing it into chunks, one cannot simply assume to be able to divide the XML document at any arbitrary point. This is because the current chunk is dependent on data present in chunks which precede it. To understand where to divide the file, preparsing stage is necessary to know the XML logical structure also called as the Skeleton of

the XML file. The skeleton is generated by a fast parse but limited processing compared to normal parsing of XML file. It looks counter-productive at first, but according to Wei Lu *et al*, the time lost to preparse an XML file is gained during parallel parsing phase.

The next stage, PXP's job is to divide the XML document into parts or chunks based on the skeleton generated. The PXP divides the XML in such a way that parsing of any chunk of XML document can start without depending on the state of its preceding chunks. As the PXP creates new chunks, it makes them available to threads which parse them to generate a DOM tree.

Two different methods of partitioning have been proposed by Lu *et al* – Static partitioning and dynamic partitioning. In static partitioning, the splitting of XML document is done by graph partitioning tools which are readily available. These tools provide simple yet balanced sub-trees which are then parsed. But using these tools would mean significant performance reduction and therefore, static partitioning is used for XML documents which have very small tree depths and relatively large number of homogenous nodes attached to a single parent node called as arrays. This ensures performance improvement in parsing an XML document.

The second method proposed is the dynamic partitioning model which is applied to large complex XML documents with deep tree structures where partitioning is done on the fly, i.e., XML file is split into chunks and as the chunks are created they are thrown to waiting threads ready to parse them. This scheme contains two stages: Task Partitioning and Sub-task Distribution. As chunks are created during the task partitioning stage, these are assigned various cores in sub-task distribution phase. Pan *et al* [2] in continuation of their previous work [1], also described a combination of static and dynamic partitioning methods for deep tree structured XML documents with large number of arrays where the load balancing scheme is provided which ensures ideal distribution of load between arrays and single nodes at the sub-task distribution stage.

2) Parallel Parsing:

In this phase, the chunks obtained from PXP are handed over to threads waiting to parse and all the threads are independent from one another and, therefore, they are executed simultaneously creating parallelism. Each thread has a DOM parser of its own which generates the partial XML DOM tree. Lu *et al* used a libxml2 [11] parser which provides good support to parse fragmented XML files. After generating the partial DOM trees, they are merged together to form the full DOM tree structure of the XML document. This is done quite easily as the XML document is divided into proper chunks at correct points creating independent chunks. Another important factor to be considered during parallel programming is load balancing. This is taken care by the PXP as it divides the XML document as discussed above.

3) Issues with Preparsing and Improvements

As mentioned earlier, preparsing stage is a sequential stage similar to normal XML parsing but it is relatively faster because it has very few operations. Initially, this was considered as acceptable additional burden considering the parsing speed of the successive parallel stage. But since then, there has been much work put in to make this stage even faster by creating parallelism. One such work is presented by Pan *et al* [3], in which the preparsing stage is parallelized by a new concept called Meta-DFA. Pan *et al* introduced this new concept generated from DFA (Deterministic Finite Automaton). In their work, Pan *et al* considered an XML parser as a DFA or a Deterministic Finite State Machine (also known as Finite State Machine) which generates a specific output for a specific input, i.e., the next parser output is considered as a resultant of previous element or state. But XML is not a regular language to build a DFA. But the authors assumed and generated a DFA which has a special state called a dead state when the DFA runs into ambiguous or non-existent state. This new DFA is called the Meta-DFA and is constructed from original DFA. It runs multiple instances of the original DFA in the form of sub-DFAs while evaluating and running all possible states and executions that are possible from the original DFA. It then checks for dead states and eliminates them as non-possibilities, i.e., chunk creation is not possible at these locations. The remaining states can run the original DFA unambiguously and at the end, the chunk is cut and fed to a different thread for parsing. This method showed significant speed up of preparsing stage.

Pan *et al* continued their work to improve the preparsing stage and implemented another algorithm based on simultaneous finite transducers (SFT) [4](also known as finite state transducers) instead of DFA based Meta-DFA. Unlike a DFA, a simultaneous finite transducer has both input and output tapes although they are not used in the same machine. And just like Meta-DFAs, SFTs runs and maintains multiple results of preparsing stage and accepts only the unambiguous and continuous states. This mechanism looks quite similar to that of Meta-DFA, but the results show that a simultaneous finite transducer based preparser is much more efficient and has less number of states to traverse through to provide appropriate chunk creation.

B. KEPT Architecture DOM Parser

Key Elements Tracing Method, proposed by Xiaosong Li *et al*, [5] is a slight extension as well as improvement of parallel XML parser (PXP) that mentioned earlier in section 2. Li *et al* presented a hybrid

parallel parser which exhibited both task parallelism and data parallelism approach. The architecture of the system is presented below in figure 2.

KEPT parser is an extension of the PXP approach and the only difference is an improved preparser stage. The preparsing stage is split into two parts. One part is executed in serial and another part is executed in parallel. Explanation to this double stage preparser is provided below.

KEPT expands to Key Elements Tracing and as the name suggests, the algorithm traces and extracts key elements from an input XML document. The key elements are defined as the parent nodes and nodes with dependencies, i.e., nodes whose data may be important to another node in another part of the same XML document. These key elements are to be parsed first and then their child and dependent nodes are parsed. The advantage of this method is that non-dependent nodes and elements which are generally stand-alone are parsed in parallel, therefore speeding up the process.

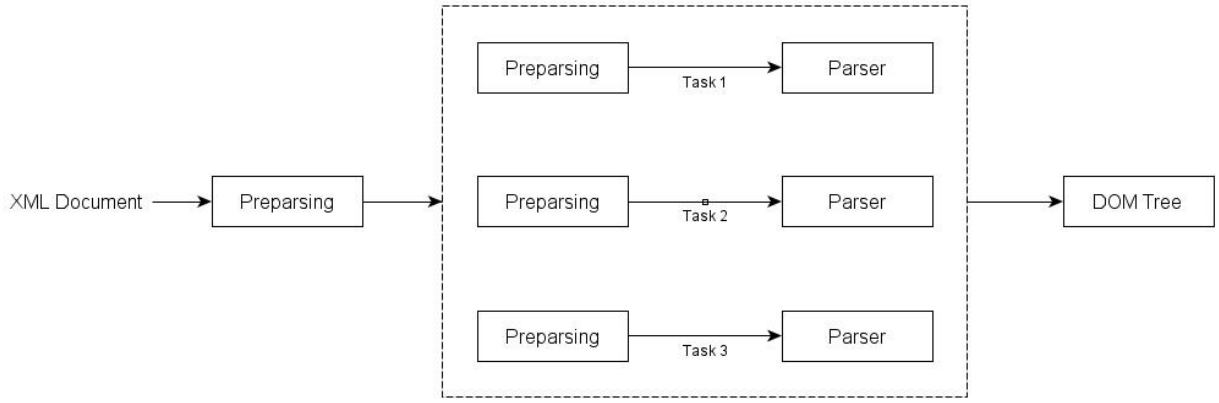

Fig. 2 KEPT Architecture by Li *et al*

The first preparsing part's function is to identify and extract those key elements from the XML document. The second preparsing part's function is to group the key elements and their dependencies into a single task to be executed in a single core of the processor and the other independent chunks which are independent are run in parallel in the other cores of the same processor. So, by leveraging on the unique opportunity presented by XML document formation, the KEPT architecture was found to speed up the preparsing stage in large XML documents.

C. ParDOM parallel DOM parser

Bhavik Shah *et al* [6] presented a different approach to parallel DOM parsing to that of PXP approach of Lu *et al*. They called it the ParDOM parser. It is also a data parallel approach algorithm and has two stages. The logic of the algorithm is shown in figure 3.

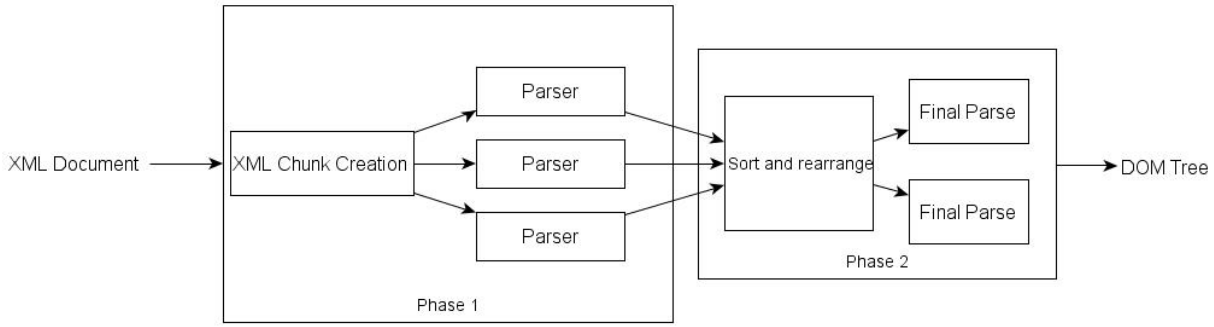

Fig. 3 ParDOM parser by Shah *et al*

The ParDOM parser contains two phases. In phase 1, the XML Document is split into chunks and these chunks are parsed in parallel. In phase 2, the partial DOM trees which result from phase 1 are sorted and rearranged in correct order to generate a final DOM tree.

The ParDOM approach is quite different from that of PXP in that it does not have a preparsing stage before parallel parsing to determine the skeleton (logical structure) of the XML document in order to create well-balanced chunks. Here, the XML document is divided at arbitrary points with equal number of start and end tags per chunk. Since ParDOM doesn't produce a skeleton to divide to divide XML at proper points, the start tag of an element may not have its end tag in the same chunk. That means the chunks are created much more quickly

in the XML document by which ample time is saved from preparsing. The chunks are then parsed in parallel by simultaneously executing DOM parsing threads to create temporary partial DOM trees.

The partial DOM trees are then sorted and merged back together in phase 2. To properly identify and merge partial trees together at the correct positions, a numbering scheme is used to uniquely identify each and every node and a stack object is used to mark a parent-child relationships as well as sibling relationships. The numbering scheme is a preorder numbering devised by Li and Moon [17]. By using this scheme, the partial DOM trees are rearranged in proper order and they are knit back together in a final parallel parse where the parent-child relationship is re-established and the final DOM tree is built to be used. Tests on this approach have shown significant performance improvement over PXP when it comes to complex tree structures and large XML files.

IV. PARALLEL SAX PARSING

A large number of applications, particularly in the area of Distributed computing, use streaming XML, i.e., a continuously streamed XML file from network. The problem with them is that a complete XML document is never in the memory of the system. It is impossible to use a parallel DOM parser in such applications. SAX (Serial API for XML) parsing is quite suitable in such situations. Even though SAX parsing is quicker than DOM parsing, it is felt that it could be made even faster by parallelism. Parallelising SAX parsing seems absurd as it is supposed to be serial. But there have been attempts to make a parallel SAX parser. But the successful ones are limited and very little work could be found about them. The following idea is one such successful attempt.

Zhang *et al* [7] presented a parallel SAX parsing model by utilising both pipelined and data parallelism to make a hybrid parallel parser. They devised a mechanism which incorporates 4 stages that are pipelined and the input XML streaming file is parsed in chunks. The model is presented in the figure 4 below.

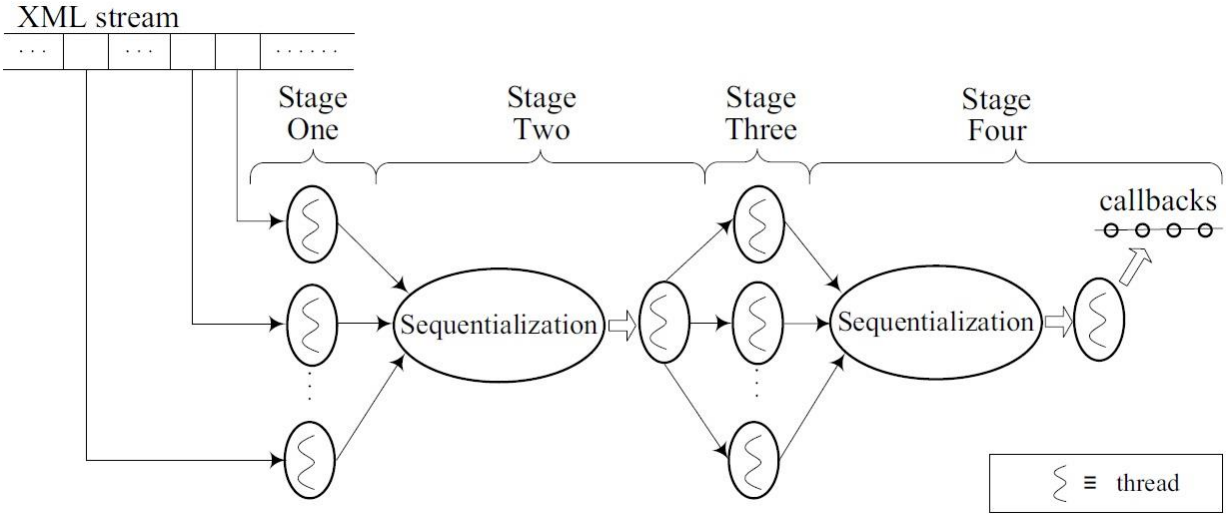

Fig. 4 Hybrid Parallel SAX parser by Zhang *et al*

The first stage is called as Preliminary Parsing and it takes in the XML stream as chunks. It performs a data-parallel preliminary parse to generate the structure of the XML stream. The chunks are independently processed without any ordering, resulting in multiple ambiguous interpretations of the XML file. To remove this ambiguity, a sequential second stage is used which is called as Chunk Resolution. This stage connects the different chunks by identifying the end state of starting chunk and matches it to the starting state of second chunk. It repeats this procedure again and again to produce one continuous stream.

The third stage called as Namespace processing takes in data stream from second stage and identifies namespace prefixes. It also performs intra-chunk namespace lookup. It does this process in a random, out of order, data-parallel approach. This stage produces structures which identify and mark start tags, end tags and content items inside a chunk. These structures are used later to generate SAX callbacks. The last stage is called as Callbacks, is again a sequential process and performs the remaining inter-chunk namespace references where an element starts in one chunk and ends in another. After this operation, the fourth stage invokes the actual callbacks for SAX events. The callbacks are issued at the last because they have to be strictly sequential and no

ambiguity should arise when generating the events. The hybrid SAX parser showed significant performance improvement over SAX parsing. Results proved it is 5 times faster than normal SAX parser.

V. CONCLUSIONS

XML parsing is a serious bottleneck which chokes the performance of many web services and grid computing applications. The modern day multi-core processors provided a fantastic opportunity to improve efficiency of XML by following the parallel approach to speed up parsing. This paper reviewed the various parallel mechanisms designed to improve XML parsing performance. Despite being difficult, data parallel approach was very much preferred rather than task parallelism or pipelining as the speed up is quite higher than the other two. A lot of work concentrated on DOM parser to improve it and the major successful methods were reviewed here. Very little work could be found about SAX parser as it is already a very fast parser. Nevertheless, the little work that was found has been reviewed and presented here.

There is tremendous scope for improvement in XML parallel parsing particularly SAX parsing because of the little work that has been done. Present SAX parsing assumed only 4-stage pipelining but with the number of cores increasing, number of pipelined stages could be increased. To improve parallel DOM parser, a combination of task parallelism, pipelining and data-parallel approach can be used to further improve parsing speed. Other developments include work stealing based parallel approach, better finite state machines to preparse XML files etc.

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] 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.

[3] 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.

[4] 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).

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

[6] 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.

[7] Yinfei Pan, Ying Zhang, Kenneth Chiu, "Hybrid Parallelism for XML SAX Parsing", *IEEE International Conference on Web Services*, 2008.

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

[9] Elliotte Rusty Harold. "An Introduction to StAX". http://www.xml.com/pub/a/2003/09/17/stax.html

[10] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. "Introduction to Automata Theory, Languages, and Computation" (2 ed.). Addison Wesley, 2001

[11] D. Veillard, "Libxml2 project web page," http://xmlsoft.org/, 2004.

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

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

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

[15] 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.

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

[17] Li, Q., Moon, B., "Indexing and querying XML data for regular path expressions", *Proc. of the 27th VLDB Conference*, Rome, Italy, September 2001, pp. 361–370

