Efficient Parallel Compression and Decompression for Large XML Files

Mohammad Ali and Minhaj Ahmad Khan
Department of Computer Science, Bahauddin Zakariya University, Pakistan

Abstract: *eXtensible Markup Language (XML) is gaining popularity and is being used widely on internet for storing and exchanging data. Large XML files when transferred on network create bottleneck and also degrade the query performance. Therefore, efficient mechanisms of compression and decompression are applied to XML files. In this paper, an algorithm for performing XML compression and decompression is suggested. The suggested approach reads an XML file, removes tags, divides the XML file into different parts and then compresses each different part on a separate core for achieving efficiency. We compare performance results of the proposed algorithm with parallel compression and decompression of XML files using GZIP. The performance results show that the suggested algorithm performs 24%, 53% and 72% better than the parallel GZIP compression and decompression on Intel Xeon, Intel core i7 and Intel core i3 based architectures respectively.*

Keywords: *XML, distributed computing, XML compression, GZIP, performance.*

Received May 26, 2014; accepted January 27, 2015; published online August 22, 2015

1. Introduction

eXtensible Markup Language (XML) [4] started almost one decade ago and was initially used by very few people. After some time it started to gain popularity, and nowadays it is being used everywhere on internet for storing and exchanging data [5, 6, 12, 23]. XML has now become a common standard for exchanging data of heterogeneous systems. Its deployment exists in small as well as large groups and organizations e.g., in banks, sports, chemical, business reporting and healthcare etc.

With the increase in data of an organization, the size of the XML files also increases. Since, XML files are transferred on a network to provide compatibility among several formats, the data traffic on the network also increases. Consequently, a bottleneck is created thereby degrading the performance of the network as well as associated queries. A natural solution is to compress and decompress XML files in order to be transferred on the network. There are different compression mechanisms such as GZIP [7], XMILL [11], XGRIND [21] etc., however, these mechanisms are not very efficient when working on text oriented XML files.

In this paper, we suggest an algorithm for performing efficient XML compression and decompression. The suggested approach reads an XML file, removes tags and divides the XML file into different parts and then compresses each different part on separate core for achieving efficiency. We compare performance results of the proposed algorithm with parallel compression and decompression of XML files using GZIP.

The remaining part of this paper is organized as follows: Section 2 describes XML compression. The optimized algorithm is suggested in section 3. The experimental setup and results are given in section 4, and last section provides conclusion and future work.

2. XML Compression

Compression [17] is a process of reducing large amount of data in to small size. There are different categories of compression mechanisms including lossless and lossy compression [3], symmetric and asymmetric compression [2], statistical and pattern model based compression [10] and adaptive and non-adaptive [2] compression. By using compression, the storage space is saved and data transfer rate improves as well. In this regard, Run-Length Encoding (RLE), Lempel-Ziv-Welch (LZW), fractal ART, JBIG, Discrete Cosine Transform (DCT) and CCITT (variation of Huffman encoding) are widely used schemes of compression.

We know that in XML files same tags and attributes are repeated again and again. Each data item has starting and as well as ending tags. Because of this, the size of file may increase many times. If this file is transferred over network, performance will degrade. Large XML file may create bottleneck on network and may also degrade query performance thereby requiring compression of XML documents.

Figure 1 shows XML document that contains metadata with a total size of 177bytes. If the metadata is excluded, the file size becomes 29bytes. It means that original XML file size was almost six times larger than that without metadata. Furthermore, an XML file may contain lengthy tags and attributes that are repeated many times. Consequently, the XML file size increases, but the efficiency of transfer/processing decreases. Therefore, the XML files must be compressed.

```
<?xml version="1.0"  encoding="UTF-8"?>
<Address>
<HouseNo> 744 </HouseNo>
<StreetNo> 4 </StreetNo>
<Colony> Noor-ul-Islam </ Colony>
<City name="Multan" />
</Address>
```

Figure 1. XML file.

The following algorithms are widely used while compressing data.

2.1. LZ77

LZ77 [24] is a general purpose algorithm and is used in different compression tools such as GZIP. LZ77 parses the input string and finds same pattern. It checks each symbol one by one and when the same symbol is repeated, it finds out the longest duplicate substring.

2.2. Huffman Encoding

In Huffman encoding [9, 15] scheme, the symbol with higher frequency is assigned short code and the symbol with lower frequency is assigned a longer code. An important thing is that, longer codes are constructed in such a way that shorter codes do not use as prefixes. In this coding, a Huffman tree is constructed, which is a binary tree. In binary tree, left branch represents 0 and the right branch represents 1.

2.3. Arithmetic Encoding

As in Huffman encoding, different number of bits are used for different symbols. But arithmetic coding uses a different approach [8, 15, 22]. It stores the output in a single floating point number greater than or equal to 0 and less than 1. This scheme is much flexible than Huffman encoding.

2.4. XML Compression Tools

The following existing tools are used for compressing XML files.

2.4.1. GZIP

GZIP [7] is one of the most common and general purpose tools used for compression and decompression. It was developed by Gailly and Adler [7]. It finds out the similar substrings in an XML file and replaces them to reduce the size of the file. It is good enough for those documents in which same substring is used again and again. It is normally based on DEFLATE algorithm, where DEFLATE uses LZ77 and Huffman encoding. Gzip has following advantages [18]:

- It does not concentrate on document structure.
- It is a general purpose tool.
- Its compression rate may be up to 50%.

The main disadvantage of GZIP is that it does not support semantic compression because it is a general compression algorithm.

2.4.2. XMILL

XMILL [11] is an efficient lossless compressor for compressing and XdeMILL is a decompress or for decompressing XML documents. Comparatively its compression rate is twice than GZIP with a small overhead. For compression only XML file is used, that is, DTD is not required. The most important reason for popularity of XMILL is that it is extensible.

2.4.3. XGRIND

The limitation of XMILL is that it does not support querying on compressed data. For query processing complete decompression is required which is not possible for limited resource devices such as handheld devices. XGRIND [21] supports querying process on compressed data. Consequently, the disks seek time decreases and the efficiency improves.

3. Proposed Framework

It is well known that there are many limitations of serial/sequential computation. It is not possible to get too much efficiency with a single processor, and therefore, multiple processing unit/cores are required. For achieving efficiency during XML file compression, this paper proposes an optimized algorithm which makes use of multiple processing units/cores. This algorithm reads an XML file, removes tags and then divides extracted data into different number of parts and sends each part on different processor for compression. For parallelism, we used MPJ Express [14, 19, 20], which is a library implemented by the mpiJava1.2 API [13, 16] and supports Single Program Multiple Data (SPMD) based model. SPMD is a subset of Multiple Instruction Multiple Data (MIMD) [1].

3.1. Optimized Algorithm for Compression

Algorithm 1 shows the steps required for compression of XML files.

Algorithm 1: Optimized algorithm for compression.

1. *Read XML file.*
2. *Remove tags.*
3. *Store extracted data in String form.*
4. *Divide String into n parts,* $\{S_1, S_2, \ldots, S_n\}$
5. *// The code below executes for each processor* $P_1, \ldots, P_n$
6. *If "Processor is* P_1 *[MASTER]" then*
7. *Send substrings towards SLAVE Processors.*
8. *Else // Processors* P_i $i€\{2, 3, \ldots, n\}$
9. *Receive substring* S_i.
10. *Compress substring by GZIP*
11. *Store each Compressed substring in a separate file* F_i, $i€\{1, 2, 3, \ldots, n\text{-}1\}$.
12. *End If*

The algorithm reads an XML file (step 1). In step 2, it removes the tags, because tags are repeated again and again and increase the file size. After this the extracted data (without tags) is stored in string form in step 3. In step 4, the string is divided into *n* parts where *n* is the number of processors and is set by the programmer.

Subsequently parallel execution starts. *If* part is executed only by "Master" processor which is represented by P_1. Moreover, the substrings are sent towards "Slave" processors for parallel execution. In contrast, the *Else* part is executed by all the slaves simultaneously. Each slave receives substring S_i from the Master processor and compresses it. After compression, each slave stores the compressed string in a separate file F_i.

3.2. Optimized Algorithm for Decompression

Algorithm 2 shows the steps required for decompression.

Algorithm 2: Optimized algorithm for decompression.

1. *Read DTD file*
2. *Extract tags from DTD file.*
3. *Read compressed files* F_i $i \in \{1, 2, 3, \ldots, n\text{-}1\}$.
4. *Uncompress* F_i *to string* S_i $i \in \{1, 2, 3, \ldots, n\text{-}1\}$.
5. $S_n \leftarrow S_1 + S_2 \ldots\ldots + S_{n-1}$
6. *Merge* S_n *with extracted tags to generate original XML file.*

Initially, the algorithm reads DTD file in step 1. In step 2, the algorithm extracts tags from DTD, because compressed files are without tags. In steps 3 and 4, compressed files are read and uncompressed to a string form. Subsequently the strings are concatenated to form a single string. In the last step, the algorithm generates the original XML file.

4. Experimentation: Setup and Results

This section presents the configuration used for experimentation and the results obtained after executing the optimized algorithm in comparison with the standard parallel execution of GZIP. We have used different files of sizes 500KB, 1MB, 1.5MB, 2MB, 2.5MB, 3MB, 4MB and 5MB on three architectures A1, A2 and A3 as given in Table 1.

Table 1. Architectures and their configurations used for experimentation.

Arichtecture (A1)	Arichtecture (A2)	Arichtecture (A3)
Intel Xeon® X5560, 64 bit, 12GB RAM	Intel Core i7- q720, 64-bit, 6GB RAM	Intel Core i3, 2.53 GHz, 64 bit, 2GB RAM

In the rest of the paper, we use the term "optimized algorithm" to refer to the proposed algorithm and the term "standard algorithm" to refer to the GZIP based parallel compression and decompression. The execution speed for all the results is measured in seconds.

4.1. Performance Results for File Size=500KB

The performance results for a file of size 500KB are given in Figure 2. As shown in the figure, the optimized code performs better than the standard code on all the three architectures A1, A2 and A3. There is an improvement of 25%, 43% and 65% in performance of the optimized code in comparison with the standard code. Overall, there is an average improvement of 44% for the file size 500KB.

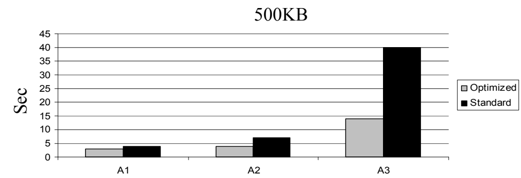

Figure 2. Performance results of optimized algorithm on 500kb.

4.2. Performance Results for File Size=1MB

The performance results for a file of size 1MB are given in Figure 3. As shown in the figure, the optimized code performs better than the standard code on all the three architectures A1, A2 and A3. There is an improvement of 20%, 59% and 83% in performance of the optimized code in comparison with the standard code. Overall, there is an average improvement of 54% for the file size 1MB.

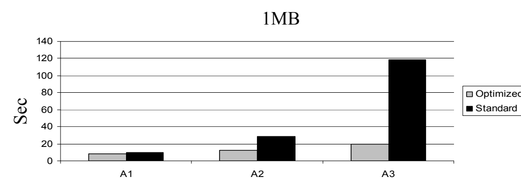

Figure 3. Performance results of optimized algorithm on 1MB.

4.3. Performance Results for File Size=1.5MB

The performance results for a file of size 1.5MB are given in Figure 4. As shown in the figure, the The optimized code performs better than the standard code on all the three architectures A1, A2 and A3. There is an improvement of 10%, 60% and 64% in performance of the optimized code in comparison with the standard code. Overall, there is an average improvement of 45% for the file size 1.5MB.

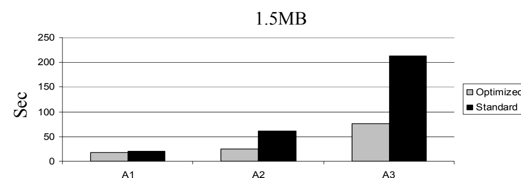

Figure 4. Performance results of optimized algorithm on 1.5MB.

4.4. Performance Results for File Size=2MB

The performance results for a file of size 2MB are given in Figure 5. As shown in the figure, the optimized code performs better than the standard code on all the three architectures A1, A2 and A3. There is an improvement of 17%, 54% and 73% in performance of the optimized code in comparison with the standard code. Overall, there is an average improvement of 36% for the file size 2MB.

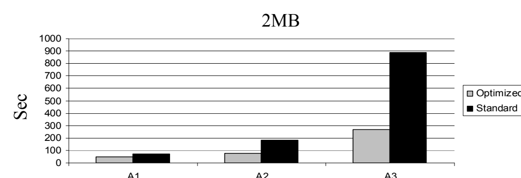

Figure 5. Performance results of optimized algorithm on 2MB.

4.5. Performance Results for File Size=2.5MB

The performance results for a file of size 2.5MB are given in Figure 6. As shown in the figure, the optimized code performs better than the standard code on all the three architectures A1, A2 and A3. There is an improvement of 32%, 58% and 70% in performance of the optimized code in comparison with the standard code. Overall, there is an average improvement of 53% for the file size 2.5MB.

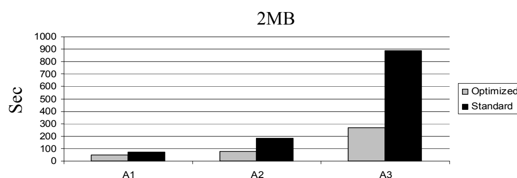

Figure 6. Performance results of optimized algorithm on 2.5MB.

4.6. Performance Results for File Size=3MB

The performance results for a file of size 3MB are given in Figure 7. As shown in the figure, the optimized code performs better than the standard code on all the three architectures A1, A2 and A3. There is an improvement of 30%, 57% and 75% in performance of the optimized code in comparison with the standard code. Overall, there is an average improvement of 54% for the file size 3MB.

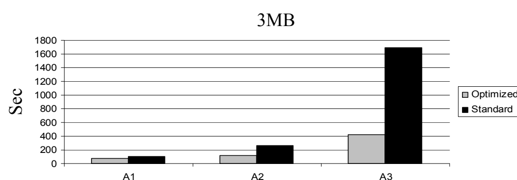

Figure 7. Performance results of optimized algorithm on 3MB.

4.7. Performance Results for File Size=4MB

The performance results for a file of size 4MB are given in Figure 8. As shown in the figure, the optimized code performs better than the standard code on all the three architectures A1, A2 and A3. There is an improvement of 32%, 57% and 76% in performance of the optimized code in comparison with the standard code. Overall, there is an average improvement of 55% for the file size 4MB.

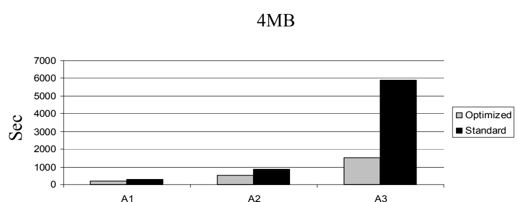

Figure 8. Performance results of optimized algorithm on 4MB.

4.8. Performance Results for File Size=5MB

The performance results for a file of size 5MB are given in Figure 9. As shown in the figure, the optimized code performs better than the standard code on all the three architectures. There is an improvement of 30%, 39% and 74% in performance of the optimized code in comparison with the standard code. Overall, there is an average improvement of 35% for the file size 5MB.

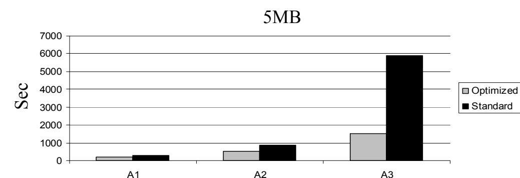

Figure 9. Performance results of optimized algorithm on 5MB.

4.9. Performance Results on Architecture 1

Figure 10 shows the performance results of both the optimized and the standard versions on architecture 1. As shown in the figure, there is small improvement for small file sizes. The improvement obtained by optimized version increases with the increases in the file size. Overall, there is an average improvement of 24% on architecture 1.

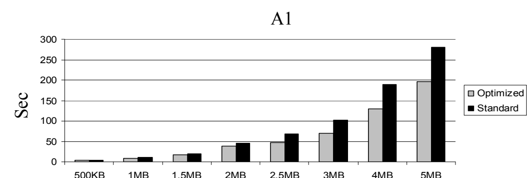

Figure 10. Performance results of optimized algorithm on A1.

4.10. Performance Results on Architecture 2

Figure 11 shows the performance results of both the optimized and the standard versions on architecture 2. As shown in the figure, there is small improvement for small file sizes. The improvement obtained by optimized version increases with the increases in the file size. Overall, there is an average improvement of 53% on Architecture 2.

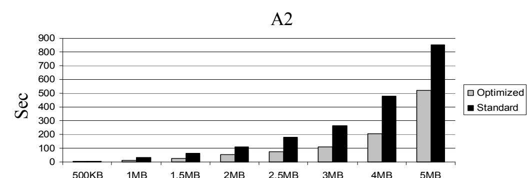

Figure 11. Performance results of optimized algorithm on A2.

4.11. Performance Results on Architecture 3

Figure 12 shows the performance results of both the optimized and the standard versions on architecture 3. As shown in the figure, there is small improvement for small file sizes. The improvement obtained by optimized version increases with the increases in the file size. Overall, there is an average improvement of 72% on architecture 3.

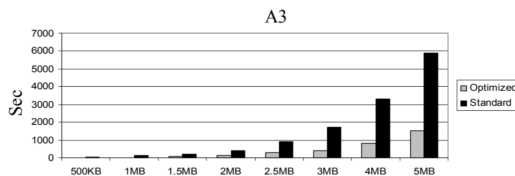

Figure 12. Performance results of optimized algorithm on A3.

5. Conclusions

This paper suggests an algorithm for performing XML compression and decompression. The suggested approach reads an XML file, removes tags, divides the XML file into different parts and then compresses each different part on separate core for achieving efficiency. We compare the performance results of the optimized algorithm with parallel compression and decompression of XML files using GZIP. We have uses files of sizes 500KB, 1MB, 1.5MB, 2MB, 2.5MB, 3MB, 4MB and 5MB on the Intel Xeon, Intel core i7 and Intel core i3 based architectures. The performance results show that, on average, the suggested algorithm performs 24%, 53% and 72% better than the parallel GZIP compression and decompression on Intel Xeon, Intel core i7 and Intel core i3 based architectures respectively. Moreover, for files of sizes 500KB, 1MB, 1.5MB, 2MB, 2.5MB, 3MB, 4MB and 5MB, the optimized algorithm is able to achieve an average improvement of 44%, 54%, 45%, 36%, 53%, 54%, 55% and 35% respectively.

In future work, we shall target the XML compression on heterogeneous systems simultaneously using Remote Method Invocation (RMI).

Acknowledgement

This paper is based on the research work conducted during MS Thesis at Bahauddin Zakariya University, Multan, Pakistan.

References

[1] Barney B., "Introduction to Parallel Computing," available at: https://computing.llnl.gov/tutorials/parallel_comp/, last visited 2013.

[2] Data Compression., available at: www.file format.info/mirror/egff/ch09_01.htm, last visited 2014.

[3] Data Compression Algorithms, "Compression," available at: www.ccs.neu.edu/home/jn122/oldsite/cshonor/jeff.html, last visited 2014.

[4] Eckstein R. and Casabianca M., *XML Pocket Reference*, O'Reilly and Associates, 2001.

[5] Extensible Markup Language (XML) 1.0., available at: http://www.w3.org/TR/REC-xml, last visited 2014.

[6] Fawcett J., Quin L., and Ayers D., *Beginning XML*, John Wiley and Sons, 2012.

[7] Gailly J., and Adler M., "GZIP," available at: http://www.gzip.org, last visited 2014.

[8] Howard P. and Vitter J., "Analysis of Arithmetic Coding for Data Compression," *in Proceedings of the IEEE Data Compression Conference*, Snowbird, pp. 3-12, 1991.

[9] Huffman D., "A Method for Construction of Minimum-Redundancy Codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098-1101, 1952.

[10] Kuri A. and Galaviz J., "Pattern-Based Data Compression," *in Proceedings of the 3rd Mexican International Conference on Artificial Intelligence*, Mexico, pp. 1-10, 2004.

[11] Liefke H. and Suciu D., "XMill: An Efficient Compressor for XML Data," *in Proceedings of ACM SIGMOD International Conference on Management of Data*, pp. 153-164 2000.

[12] Lv T. and Yan P., "A Framework of Summarizing XML Documents with Schemas," *the International Arab Journal of Information Technology*, vol. 10, no. 1, pp. 18-27, 2013.

[13] MPI: A Message-Passing Interface Standard Version 3.0., available at: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, last visited 2013.

[14] MPJ Express., available at: http://www.mpj express.org, last visited 2013.

[15] Nelson M., "Arithmetic Coding and Statistical Modeling," available at: http://www.drdobbs.com/parallel/arithmetic-coding-and-statistical-modeli/184408491, last visited 1991.

[16] Pacheco P., *Parallel Programming with MPI*, Morgan Kauffman Publishers, 1997.

[17] Salomon D., *Data Compression*, *the Complete Reference*, Springer, 1997.

[18] Severson E. and Fife L., "XML Compression: Optimizing Performance of XML Applications," *Flatirons Solutions*, 2003.

[19] Shafi A., Carpenter B., and Baker M., "Nested parallelism for Multi-Core HPC Systems using Java," *Journal of Parallel and Distributed Computing*, vol. 69, no. 6, pp. 532-545, 2009.

[20] Shafi A., Carpenter B., Baker M., and Hussain A., "A Comparative Study of Java and C Performance in Two Large Scale Parallel Applications," *Concurrency and Computation: Practice and Experience*, vol. 21, no. 15, pp. 1882-1906, 2009.

[21] Tolani P. and Haritsa J., "XGrind: A Query-Friendly XML Compressor," *in Proceedings of the 18th International Conference on Data Engineering* (ICDE), San Jose, pp. 225-234, 2002.

[22] Witten I., Neal R., and Cleary J., "Arithmetic Coding for Data Compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520-540, 1987.

[23] XML Tutorial., available at: www.w3schools. com/xml/default.asp, last visited 2013.

[24] Ziv J. and Lempel A., "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337-343, 1997.

Mohammad Ali has completed his MS degree in Computer Science from Bahauddin Zakariya University, Multan. Currently, his research interests include algorithms and networks.

Minhaj Ahmad Khan completed his MS and PhD degree from University of Versailles, France. He is currently working as Assistant Professor at Bahauddin Zakariya University, Multan. His research interests include code optimization and high performance computing.