# XML Parsing on Multicore Systems using Pipeline Parallelism

Dan Lin
School of Computing Science, Simon Fraser University, Vancouver, Canada
lindanl@sfu.ca

## Abstract

*XML, a widely used format, plays an important role in today's web services and databases. But the performance and energy efficiency of XML parsing is often considered a problem. As multicore architectures become the mainstream, parallelizing the XML parser offers an attractive way to achieve better performance and reduce energy consumption. However, XML parsing, well known as a sequential process, poses tough problems for parallel programming. Although data parallelism has been applied to XML parsers in some previous works, the overhead is fairly high. This paper presents a pipeline parallelizing method for Parabix, a high performance XML parser based on parallel bitstream technique. Its unique structure allows us to take advantage of the inherent efficiency of pipeline parallelism. Experimental results show more than a 2X speedup compared with the sequential version. It is also argued that this method of pipeline parallelism could be easily applied to other applications based on parallel bitstream techniques.*

***Keywords*** parallel bitstream, pipeline parallelism

## 1. Introduction

Extensible Markup Language(XML) is widely used for web applications and services. Parsing of XML may be a significant factor affecting the response time and even the throughput. XML also plays an important role in databases. Enterprises keep large amounts of business critical data permanently in XML format. However, those databases may not provide the same high performance characteristics as relational data processing because processing of XML requires parsing of XML documents which can be very CPU-intensive [9].

CPU-intensive applications are able to benefit from multicore systems by dividing the work to run on different cores. The growing prevalence of multicore systems motivates the investigation of parallelizing XML parsers. However, parallelizing programs is challenging because the gains from parallel execution can be overshadowed by the cost of communication and synchronization. Parallelizing XML parsing software is further complicated because it is sequential in nature.

There are three basic parallelizing methods. Task parallelism refers to parallel executions where the output of each task never reaches the input of the other. Task parallelizing XML parsers can be easily achieved by processing different files on different cores at the same time. Unfortunately, the files used with either web services or databases are often large requiring fast parsing on a single file.

Data parallelism refers to the partitioning of data into several chunks which are each processed in parallel. Data parallelism is well-suited to multicore systems when there are no dependencies between each partition. Otherwise, it can introduce excessive communication overheads and potential hazards. The nature of XML files makes them hard to partition nicely for data parallelism. Several approaches have been used to address this problem. A preparsing phase has been proposed to help partition the XML document [8]. The goal of this preparsing is to determine the tree structure of the XML document so that it can be used to guide the full parsing in the next phase. Other methods such as overlapping and speculation have also been used for data parallel processing [7].

Pipeline parallelism is appropriate when the computation can be divided into multiple stages such that data naturally flows from one stage to another and synchronization is needed only at stage interface. Compared to data parallelism, it offers reduced latency, reduced buffering and less communication [6]. But it is often hard to implement well, due to problems of synchronization and balancing the load between stages. However, previous work presented a high performance XML parser, Parabix, using parallel bitstream techniques [2, 3]. Its unique structure makes it easy to separate the parsing into several passes. This multipass structure makes it suitable to apply pipeline parallelism on multicore systems.

This paper presents a pipelining strategy for Parabix based on the observation of data dependencies and analysis of the characteristics of each pass. The resulting performance and energy consumption is compared with sequential Parabix and other sequential XML parsers that have been widely used. The generalization of this pipelining strategy to other applications based on parallel bitstream techniques is described.

## 2. Background

### 2.1 XML

Extensible Markup Language (XML) is a simple text format derived from SGML, officially adopted by W3C as a standard in 1998 [1]. It is essentially a set of rules for encoding documents or data in a human readable form. For example, a typical XML file could be:

```
<?xml version="1.0"?>
<shipping>
  <shipTo  country="Canada">
    <name> Alice </name>
    <address> XXX </address>
  </shipTo>
  ...
</shipping>
```

**Figure 1.** Simple XML Document

### 2.2 XML Well-Formedness Checking

The XML specification defines an XML document as a text which is well-formed. That is, it satisfies a list of syntax rules provided in the specification. If a document are correctly formed and conform to all the well-formedness rules, then it is considered as well formed.

XML well-formedness checking application is evaluated for each XML parsing technology. The decision to perform XML well-formedness checking for this study is based on the following rational. First, each XML parser must provide well-formedness checking functionality. Secondly, this functionality meets the minimum requirement of an XML document being readable by computers and avoids any additional costs due to non-parsing related computation.

### 2.3 Traditional XML Parsers

Traditional XML parsers such as Expat and Xerces process XML one byte at a time. They parse a source document serially, from the first to the last byte of the source file. Each character of the source text is examined in turn to distinguish between the XML-specific markup, such as an opening angle bracket '<', and the content held between markups. This method creates a lot of branches in the program. In fact, Xerces can execute as many as 13 branches per byte it processed.

Both Expat and Xerces-C are C/C++ based and open-source. Expat was originally released in 1998; it is currently used in Mozilla Firefox and Open Office [4]. Xerces-C was released in 1999 and is the foundation of the Apache XML project [5].

### 2.4 Parabix: SIMD-based XML Parser

Parabix is a SIMD-based parallel bitstream XML parser. It first transposes the input byte stream into eight parallel bitstreams called basis bitstreams. Then it calculates all the markup item streams based on those basis bitstreams using SIMD bitwise logic instructions. A markup item streams is simply a sequence of 0s and 1s, where there is one such bit in the stream for each markup item in a source data stream. As shown in Figure 2, $M_0$ is left angle stream shifted by 1. Given all the markup item streams, parsing becomes a series of parallel bit scans. For example, in Figure 2, the first step of parsing the start tag is to scan through the tag names using $M_0$. On a little endian byte order machine, parallel bit scan is implemented using addition and logic operations [2]. $M_1$ gives the result of scanning.

| | |
|---|---|
| source data | `<t1>abc</t1><tag2/>` |
| $Name = [0\text{-}9a\text{-}z]$ | `.11.111..11..1111..` |
| $M_0 = [<] >> 1$ | `.1......1....1.....` |
| $M_1 = scanThru(M_0, Name)$ | `...1....1........1.` |

**Figure 2.** Partial Start Tag Parsing

Figure 3 shows the results of the overall performance for XML well-formedness checking evaluated as CPU cycles per input bytes. Parabix is 2 to 7 times faster than traditional parsers.

Parabix exploits the data parallelism via SIMD instructions. The goal of this study is to further improve the performance using chip-multiprocessors.

## 3. Parallelizing Strategy

### 3.1 Pipelining Parallelism

The typical approach to parallelizing software (data parallelism) requires nearly independent data, which is a difficult task for dividing

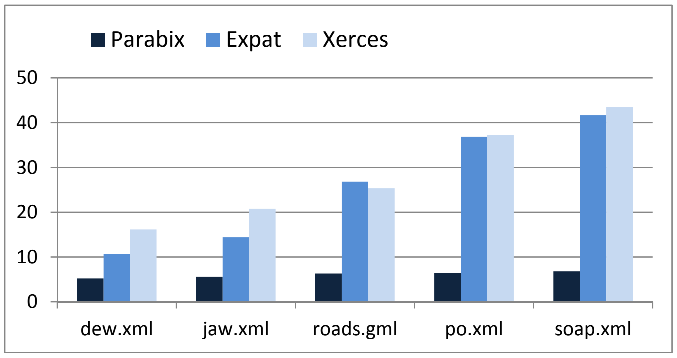

**Figure 3.** Processing Time (y-axis: CPU cycles per byte)

XML data. A simple division determined by the segment size can easily make most of the segments illegal according to the parsing rules while the data as a whole is legal. For example, Figure 1 is a legal XML file, but Figure 4, which contains a incomplete start tag will not be considered as a legal piece.

```
country="Canada">
  <name> Alice </name>
  <address> XXX </address>
</shipTo>
```

**Figure 4.** XML Document Segment

Therefore, instead of dividing the data into segments and assigning different data segments to different cores, we divide the process into several stages and let each core work with one single stage. As shown in Figure 5, data dependencies exist between the same segment processed in different stages. Take the first segment S1 as an example, S1-B depends on S1-A, S1-C depends on S1-B and S1-D denpends on S1-C. There are also dependencies between different segments processed in the same stages. For example, S2-A depends on S1-A, S3-A depends on S2-A and so on so forth. However, there is no dependency between different segments processed in different stages, such as S1-A and S2-B. Thus, using a pipeline strategy shown in Figure 6 ensures that at any time slice, there is no dependency between each thread.

### 3.2 Work Division

Ideally, no thread will stall when all of them process a given segment with the same speed, that is, for example, when T1 finishes with S6, T2 should complete with S5, T3 should complete with S4 and T4 should complete with S3. However, it is difficult to evenly divide up the work. As shown in Figure 7, Parabix can be separated into eleven passes. The time consumed by each pass could be different depending on the characteristics of the test file.

On a quad core machine, we divide the eleven passes into four stages based on the work distribution calculated by analyzing the sequential Parabix (Figure 7). StageA contains pass one to pass three, which are fill_buffer, s2p and classify_bytes. StageB contains pass four to pass seven, which are validate_u8, gen_scope, parse_CtCDPI and parse_ref. StageC contains pass eight to pass ten, which are parse_tag, validate_name and gen_check. StageD contains the last pass, postprocessing. Table 1 shows the time consumed on each stage. Overhead will be introduced when each stage is running on a different core due to the resource contention and data migration, which will be discussed in the next section. Therefore, the processing time of each stage on a separate thread might be more than what is shown in the table.

| | | Data Structures | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | srcbuf | basis_bits | u8 | lex | scope | ctCDPI | ref | tag | xml_names | check_streams |
| StageA | fill_buffer | write | | | | | | | | | |
| | s2p | read | write | | | | | | | | |
| | classify_bytes | | read | | write | | | | | | |
| StageB | validate_u8 | | read | write | | | | | | | |
| | gen_scope | | | | read | write | | | | | |
| | parse_CtCDPI | | | | read | read | write | | | | write |
| | parse_ref | | | | read | read | read | write | | | |
| StageC | parse_tag | | | | read | read | read | | write | | |
| | validate_name | | | read | read | | read | read | read | write | write |
| | gen_check | | | read | read | read | read | | read | read | write |
| StageD | postprocessing | read | | | read | | read | read | | | read |

**Table 2.** Relationship between Each Pass and Data Structures

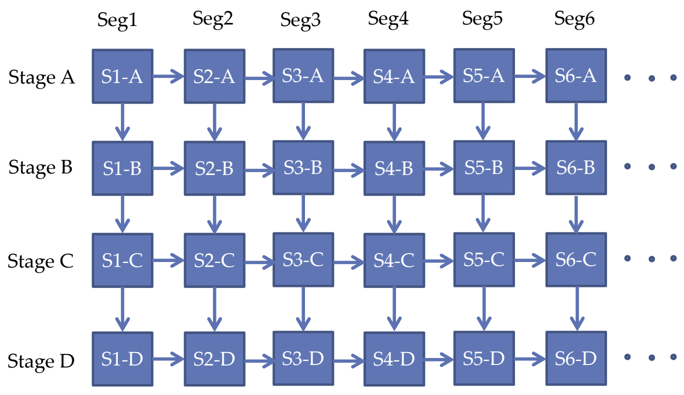

**Figure 5.** Data Dependency

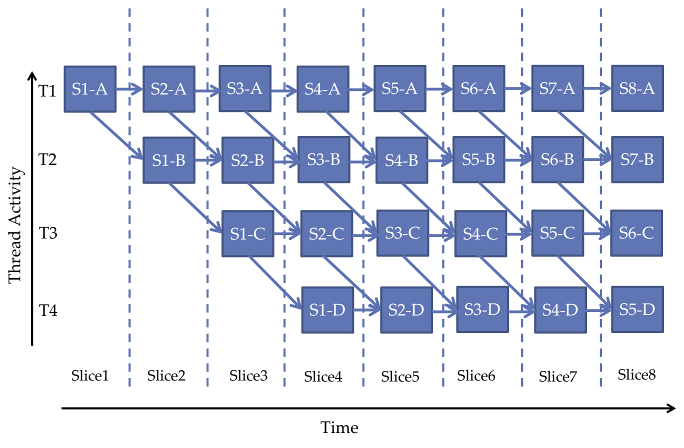

**Figure 6.** Pipelining Strategy

| | dew | jaw | roads | po | soap |
|---|---|---|---|---|---|
| stageA | 1.981 | 1.973 | 1.967 | 1.97 | 1.97 |
| stageB | 1.161 | 1.403 | 0.895 | 1.724 | 0.877 |
| stageC | 1.582 | 1.601 | 2.383 | 2.141 | 2.451 |
| stageD | 0.855 | 0.928 | 1.28 | 1.704 | 1.818 |

**Table 1.** Time Consumed on Each Stage under Sequential Processing (CPU cycles per byte)

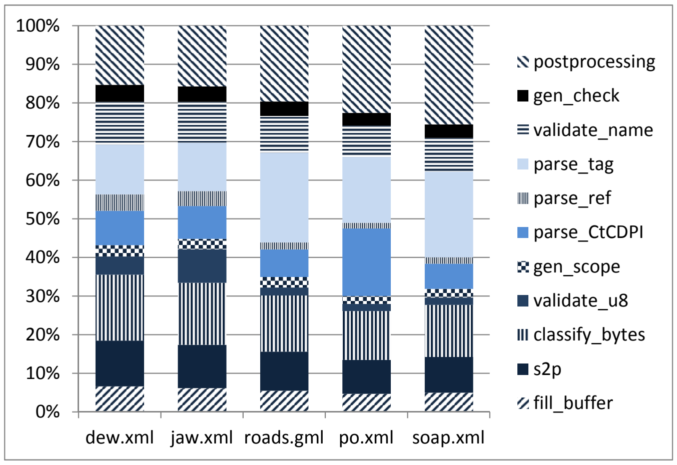

**Figure 7.** Work Distribution Between Each Pass

### 3.3 Circular Queue

The interface between stages is implemented using a circular array, where each entry consists of all ten data structures for one segment as listed in Table 2. Each thread keeps an index of the array ($I_N$), which is compared with the index ($I_{N-1}$) kept by its previous thread before processing the segment. If $I_N$ is smaller than $I_{N-1}$, thread N can start processing segment $I_N$, otherwise the thread keeps reading $I_{N-1}$ until $I_{N-1}$ is larger than $I_N$. The time consumed by continuously loading the value of $I_{N-1}$ and comparing it with $I_N$ will be later referred as stall time. When a thread finishes processing the segment, it increases the index by one.

This algorithm has three basic sources of overhead. The first one is data migration. As shown in Table 2, many of the data structures are needed by more than one stage and hence processed by more than one core. Therefore, those data structures have to be loaded several times into different caches. The second overhead comes from maintaining the cache coherency of those data structures. Table 2 shows the relationship between each pass and data structures. Fortunately, all of the data structures except check_streams are written only once when they are brought into the cache the first time. Further optimization could be done by compressing some of the data structures as well as restructuring check_streams such that it is written only by pass gen_check. The third overhead is caused by the control data (index). Since $I_N$ is shared between thread N and thread N+1, each time thread N modifying $I_N$ will cause cache invalidation and when thread N+1 tries to read it, it generates a cache miss.

## 4. Evaluation

### 4.1 Test Data and Platform

Distinguishing between "document-oriented" XML and "data-oriented" XML is a popular way to describe the two basic classes of XML documents. Data-oriented XML is used as an interchange format. Document-oriented XML is used to impose structure on information that rarely fits neatly into a relational database–particularly information intended for publishing. Data-oriented XML are characterized by a higher markup density. Markup density is defined as the ratio of the total markup contained within an XML file to the total XML document size. This metric may have substantial influence on the performance of XML parsing. As such we choose workloads with distinguishable markup densities. Table 3 shows the document characteristics of the XML instances selected for this performance study.

All the experiments are run on a quad core machine with Linux kernel 2.6.35. Table 4 gives the hardware description of the machine selected.

| | |
|---|---|
| Processor | Intel Sandybridge i5-2300 (2.80GHz) |
| L1 Cache | 4 X 32KB I-Cache 2 X 32KB D-Cache |
| L2 Cache | 4 X 256KB |
| L3 Cache | 6-MB |
| Front Side Bus | 1333 MHz |
| Memory | 6GB DDDR |
| Max TDP | 95W |

**Table 4.** Machine

### 4.2 Parameters

#### 4.2.1 Segment Size

Increasing the segment size reduces the synchronization overhead. As shown in Figure 8, the processing time drops dramatically as the segment size goes from 128 bytes to 2KB. However, the performance cease to improve, actually slightly degrades when segment size is larger than 16KB because of cache contention. In real applications, we would like to use as little memory as possible without hurting much of the performance. The rest of the experiments are run using segment size 16KB.

#### 4.2.2 Circular Array Size

When the circular array size $C$ is smaller than the number of threads, only $C$ threads will be able to do useful work at the same time. As shown in Figure 9, when there are only two entries, the performance is even worse than the sequential Parabix. When the circular array size is larger than number of threads, increasing the size of the circular array allows threads to be pushed deeper into the queue. Then if one thread runs faster than the following thread, it has more time to process before it must stop and wait. To some extent, this can even out the variations of processing time. However, a much larger array size won't help because allocating a larger memory area can degrade the performance and the processing time is depending on the slowest stage in the pipeline not the fast one. Therefore, the rest of the experiments are run using only 6 entries.

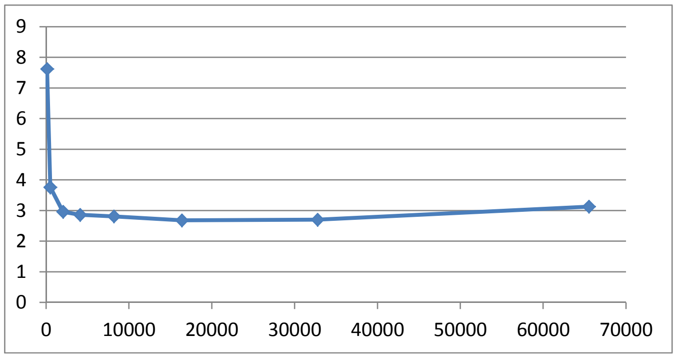

**Figure 8.** Processing Time with Different Segment Size (x axis: byte, y axis: CPU cycles per byte)

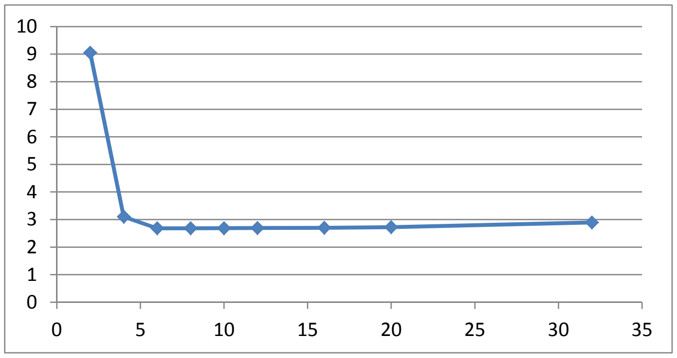

**Figure 9.** Processing Time with Different Circular Array Size (x axis: number of entries, y axis: CPU cycles per byte)

### 4.3 Load Balance

Figure 10 shows the work time and stall time of each thread with different test files. As discussed in the previous section, the work loads are not evenly divided. Therefore, the threads that process faster have to wait for predecessors to finish and thus incur a certain amount of stall time. The overhead introduced by data migration and resource contention is 27% to 37%, calculated as $(overall\_worktime - sequential\_time)/sequential\_time$. The overhead introduced by synchronization is 15% to 50%, calculated as $overall\_stalltime/sequential\_time$.

### 4.4 Performance

Figure 11 demonstrates the XML well-formedness checking performance of the parallelized Parabix in comparison with the sequential version. The parallelized Parabix is more than 2 times faster on the quad core machine. With the sequential Parabix, the performance decreases as markup density of the test files increases. However, the high density files are better balanced and consume less stall time. Thus, it turns out that the processing time of each of the test files is about the same at 2.7 cycles per byte.

| File Name | dew.xml | jaw.xml | roads.gml | po.xml | soap.xml |
|---|---|---|---|---|---|
| File Type | document | document | data | data | data |
| File Size (kB) | 66240 | 7343 | 11584 | 76450 | 2717 |
| Markup Density | 0.07 | 0.13 | 0.57 | 0.76 | 0.87 |

**Table 3.** XML Document Characteristics

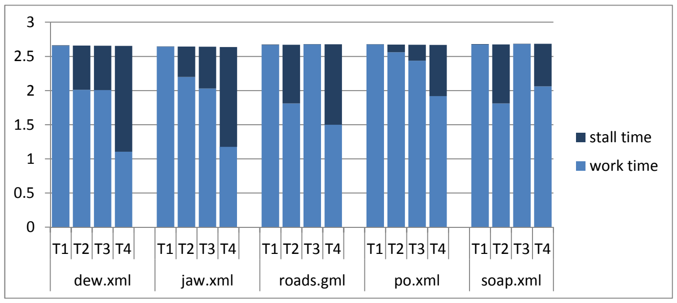

**Figure 10.** Processing Time of Each Thread (y axis: CPU cycles per byte)

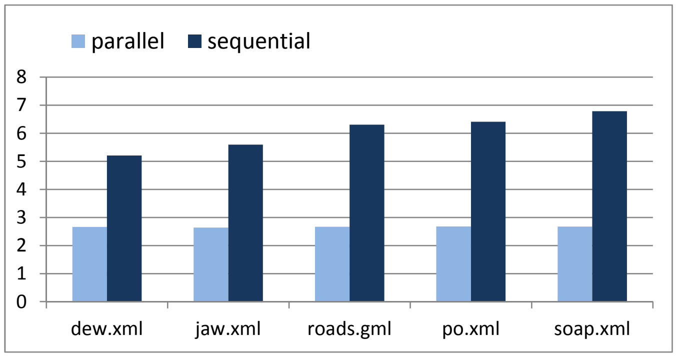

**Figure 11.** Processing Time (y axis: CPU cycles per byte)

## 4.5 Power and Energy

Figure 12 shows the average power consumed by the parallelized Parabix in comparison with the sequential version. By running four threads and using all the cores at the same time, the power consumption of the parallelized Parabix is much higher than the sequential version. However, the energy consumption is about the same, because the parallelized Parabix needs less processing time. In fact, as shown in Figure 13, parsing soap.xml using parallelized Parabix consumes less energy than using sequential Parabix.

## 4.6 Performance vs. Energy

Figure 14 shows the performance and energy consumption of sequential and parallelized Parabix as well as two other XML parsers, Expat and Xerces. Parabix consumes 25% of the energy of Xerces and Expat but with much better performance. Although the parallelized Parabix consumes slightly higher average energy than the sequential Parabix, it is more than 2 times faster.

## 5. Conclusion and Future Work

This paper studied and analyzed the structure of Parabix and presents a pipeline strategy for XML parsing. Performance and energy consumption are evaluated on a quad core machine and

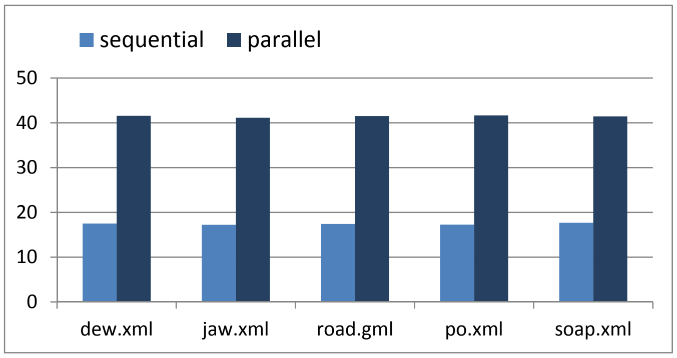

**Figure 12.** Average Power (watts)

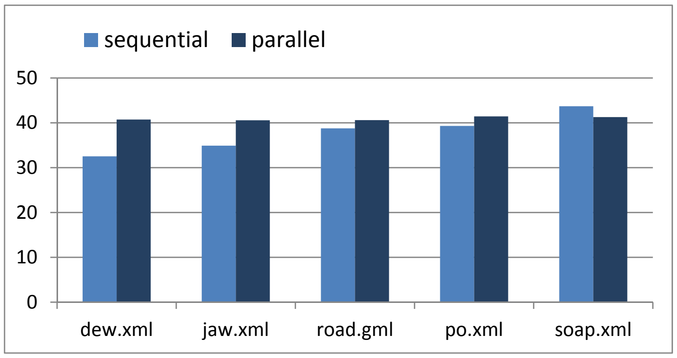

**Figure 13.** Energy Consumption (nJ per byte)

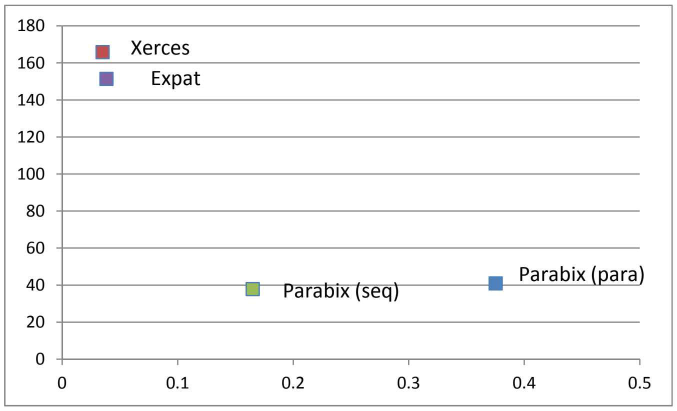

**Figure 14.** Energy vs. Performance (x axis: bytes per cycle, y axis: nJ per byte)

compared with the sequential Parabix as well as two other XML parsers. The parallelized Parabix provides a 2X speedup over the sequential version by using about the same amount of energy. It also shows a substantially better performance and less energy consumption compared with Xerces and Expat.

The presented parallelizing strategy can be applied to other sequential applications that share the same data dependency structure as described for Parabix, especially for applications based on parallel bitstream technologies, where the bitstream processing can be easily divided into different stages.

There are many possible optimizations for further research. For example, the frequency of each core can be dynamically changed to reduce stall time and save energy. A new thread can be created and assigned to help thread one on the first stage since the first stage is the bottleneck of the overall performance as shown in Figure 10. However, only the first three passes (first stage) of Parabix are data independent and hence can be processed by multiple threads at the same time. Therefore, this method might not work for a different work division.

Other future research includes porting the parallelized Parabix to different architectures (e.g. more cores, NUMA, asymmetric machine) and dynamically assigning the workload by sampling the performance at run time instead of predefining work division based on analysis of the sequential Parabix.

## Acknowledgments

## References

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Franois Yergeau. Extensible markup language (XML) 1.0 (fifth edition). W3C Recommendation, 2008.

[2] Robert D. Cameron, Ehsan Amiri, Kenneth S. Herdy, Dan Lin, Thomas C. Shermer, and Fred P. Popowich. Parallel parsing with bitstream addition: An XML case study. Technical Report TR 2010-11, Simon Fraser University, School of Computing Sciece, October 2010.

[3] Robert D. Cameron, Kenneth S. Herdy, and Dan Lin. High performance XML parsing using parallel bit stream technology. In *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*, pages 222–235, New York, NY, USA, 2008. ACM.

[4] James Clark. The Expat XML Parser. http://expat.sourceforge.net/.

[5] Apache Software Foundation. Xerces C++ Parser. http://xerces.apache.org/xerces-c/.

[6] Michael Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, October 2006.

[7] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. Parallelizing the web browser. *In Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[8] Wei Lu, Yinfei Pan, , and Kenneth Chiu. A parallel approach to xml parsing. *The 7th IEEE/ACM International Conference on Grid Computing*, 2006.

[9] Matthias Nicola and Jasmi John. Xml parsing: a threat to database performance. In *Proceedings of the twelfth international conference on Information and knowledge management*, CIKM '03, pages 175–178, New York, NY, USA, 2003. ACM.