A Data Parallel Algorithm for XML DOM Parsing

Bhavik Shah[1], Praveen R. Rao[1], and Bongki Moon[2] and Mohan Rajagopalan[3]

[1] University of Missouri-Kansas City
{BhavikShah,raopr}@umkc.edu
[2] University of Arizona
bkmoon@cs.arizona.edu
[3] Intel Research Labs
mohan.rajagopalan@intel.com

Abstract. The extensible markup language XML has become the de facto standard for information representation and interchange on the Internet. XML parsing is a core operation performed on an XML document for it to be accessed and manipulated. This operation is known to cause performance bottlenecks in applications and systems that process large volumes of XML data. We believe that parallelism is a natural way to boost performance. Leveraging multicore processors can offer a cost-effective solution, because future multicore processors will support hundreds of cores, and will offer a high degree of parallelism in hardware. We propose a *data parallel* algorithm called *ParDOM* for XML DOM parsing, that builds an in-memory tree structure for an XML document. *ParDOM* has two phases. In the first phase, an XML document is partitioned into chunks and parsed in parallel. In the second phase, partial DOM node tree structures created during the first phase, are linked together (in parallel) to build a complete DOM node tree. *ParDOM* offers fine-grained parallelism by adopting a flexible chunking scheme – each chunk can contain an arbitrary number of start and end XML tags that are not necessarily matched. *ParDOM* can be conveniently implemented using a data parallel programming model that supports `map` and `sort` operations. Through empirical evaluation, we show that *ParDOM* yields better scalability than PXP [23] – a recently proposed parallel DOM parsing algorithm – on commodity multicore processors. Furthermore, *ParDOM* can process a wide-variety of XML datasets with complex structures which PXP fails to parse.

1 Introduction

The extensible markup language XML has become the de facto standard for information representation and exchange on the Internet. Recent years have witnessed a multitude of applications and systems that use XML such as web services and service oriented architectures (SOAs) [16], grid computing, RSS feeds, ecommerce sites, and most recently the Office Open XML document standard (OOXML). Parsing is a core operation performed before an XML document can be navigated, queried, or manipulated. Though XML is simple to read and process by software, XML parsing is often reported to cause performance bottlenecks for real-world applications [22, 32]. For example, in a SOA using web services technology, services are discovered, described, and invoked using XML messages [10]. These messages can reach up to several megabytes in size, and thus parsing can cause severe scalability problems.

Recently, high performance XML parsing has become a topic of considerable interest (*e.g.*, XML Screamer [19], schema-specific parser [11], PXP [23, 24], Parabix [7]). XMLScreamer and schema-specific parser leverage schema information for optimizing tasks such as scanning, parsing, validation, and deserialization. On the other hand, PXP and Parabix exploit parallel hardware to achieve high XML parsing performance. Our work in this paper also exploits parallel hardware to achieve high parsing performance.

With the emergence of large-scale throughput oriented multicore processors [26][15], we believe parallelism is a natural way to boost the performance of XML parsing. Leveraging multicore processors can offer a cost-effective way to overcome the scalability problems, given that future multicore processors will support hundreds of cores, and thus, offer a high degree of parallelism in hardware. A data parallel programming model offer numerous benefits for future multicore processors such as expressive power, determinism, and portability [12]. For instance, traditional thread-based approaches suffer from non-deterministic behavior and make programming difficult and error prone. On the contrary, a program written in a data parallel language (*e.g.*, Ct [13]) has deterministic behavior whether running on one core or hundred cores. This eliminates data races and improves programmer productivity. Thus, there has been a surge of interest to develop data parallel models for forward scaling on future multicore processors [8, 12].

With these factors in mind, we propose a *data parallel* XML parsing algorithm called *ParDOM*. In this paper, we focus on **XML DOM (Document Object Model) parsing** [30], because it is easy to use by a programmer and provides full navigation support to an application, and it is widely supported in open-source and commercial tools (*e.g.*, SAXON [18], Xerces [3], Intel Software Suite [1], MSXML [2]). Further, DOM parsing poses a fundamental challenge of parallel tree construction. Since DOM parsing requires documents to fit in main memory, we only consider XML documents that are of several megabytes in size.

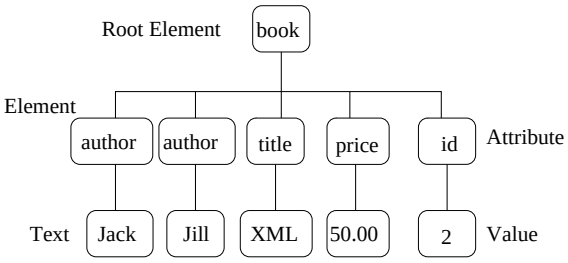
ParDOM is a two-phase algorithm. In the first phase, an XML document is partitioned into chunks and are parsed in parallel. In the second phase, partial DOM node tree structures created during the first phase, are linked together (in parallel) to build a complete DOM node tree in memory. Our algorithm offers fine-grained parallelism by adopting a flexible chunking scheme. Unlike a previous parallel algorithm called PXP [23, 24], wherein chunks represent subtrees of a DOM tree, *ParDOM* creates chunks that can contain an arbitrary number of start and end XML tags that are not necessarily matched. *ParDOM* can be conveniently implemented using a data parallel programming model that supports `map` and `sort` operators. Through empirical evaluation, we show that *ParDOM* yields better scalability than PXP on commodity multicore processors. Furthermore, *ParDOM* can process a wide-variety of XML datasets with complex structures which PXP fails to parse.

2 Background & Motivation

2.1 XML Documents and Parsing Techniques

An XML document contains elements that are represented by start and end element tags. Each element can contain other elements and values. An element can have a list of (attribute, value) pairs associated with it. An XML document can be modeled as an

ordered labeled tree. A well-formed XML document follows the XML syntax rules. For example, each element has a start tag and a matching end tag.



(b) DOM node tree representation

Fig. 1. Example

For an XML document to be accessed and manipulated, it should first be parsed. Many XML parsing models have been developed that trade off between the ease of use, APIs exposed to applications, memory footprint, parsing speed, and support for XPath [5].

Among these, DOM parsing and SAX parsing are widely supported. Document Object Model (DOM) [30] parsing builds an in-memory tree representation of an XML document by storing its elements, attributes, and values along with their relationships. (Other DOM node types have been defined by W3C [30]. We restrict ourselves to the most common ones: Element, Attribute, Text/Value.) A DOM node tree aids easy navigation of XML documents and supports XPath [5]. A DOM tree for a document is shown in Figure 1. The order of siblings in the tree follows the order in which their elements appear in the document (a.k.a. document order).

SAX parsing [21] is an event based parsing approach. It is light-weight, fast, and requires a smaller memory footprint than DOM parsing. However, an application is responsible for maintaining an internal representation of a document if required. Newer parsing models such as StAX [6] and VTD-XML were developed to improve over DOM and SAX. The Binary XML standard [14], though not a parsing model, was proposed to reduce the verbosity of XML documents and the cost of parsing. However, human-readability is lost.

2.2 Prior Work on Parallel XML Parsing

Recently, Pan *et al.* proposed a parallel XML DOM parsing algorithm called PXP for multicore processors [23]. This approach first constructs a skeleton of a document in memory. Using the skeleton, the algorithm identifies chunks of the document that can be parsed in parallel. (Each chunk denotes a subtree of the final DOM tree.) This task requires recursively traversing the skeleton until enough chunks are created. After the chunks are created, they are parsed in parallel to create the DOM tree. Subsequently, Pan *et al.* proposed an improved algorithm to parallelize the skeleton construction [24].

However, these algorithms have the following shortcomings that motivate our research. First, the skeleton requires extra memory that is proportional to the number of node in the DOM tree. Further, the partitioning scheme based on subtrees can cause load imbalance on processing cores for XML documents with irregular or deep tree structures (*e.g.*, TREEBANK with parts-of-speech tagging [29]). This scheme severely limits the granularity of parallelism that can be achieved, and thus cannot scale with increasing core count.

Wu *et al.* proposed a parallel approach XML parsing and schema validation [31]. Although their chunking scheme during parsing is similar to that of *ParDOM*, the partial DOM trees for each chunk are linked sequentially during post-processing. Parabix [7], though not a parallel DOM parsing algorithm, exploits parallel hardware for speeding up parsing by scanning the document faster. Rather than reading a byte-at-a-time from an XML document, Parabix fetches and processes many bytes in parallel.

2.3 Prior Work on Data Parallel Programming Models

The emergence of multicore processors demands new solutions for expressing parallelism in software to fully exploit their capabilities [4]. There has been a keen interest in developing parallel programming models for this purpose. Intel's Ct [13] supports a data parallel programming model and aims on forward scaling for future multicore processors. Data Parallel Haskell is another effort to exploit the power of multicores [8].

In recent years, programming models to support large-scale distributed computing on commodity machines have been developed. The MapReduce paradigm and associated implementation was introduced by Google for performing data intensive computations that can be distributed across thousands of machines [9]. Hadoop (`http://hadoop.apache.org`) and Disco (`http://discoproject.org`) are two different open source implementations of MapReduce. Phoenix [25] is a shared memory MapReduce implementation. Recently, a distributed execution engine called Dyrad [17] was proposed for coarse-grained data parallel applications.

3 Our Proposed Approach

We begin with a description of a serial algorithm for building a DOM tree. We present a scenario to motivate the design of our parallel algorithm *ParDOM*. We focus on XML documents whose DOM trees can fit in main memory. (For very large XML documents, other parsing models (*e.g.*, SAX [21]) should be used.) For ease of exposition, we focus on elements, attributes, and text/values in XML documents. Although a text can appear anywhere within the start and end tag of an element, we shall first assume that it is strictly enclosed by start and end element tags, *e.g.*, `<author>Jack</author>`. Later in Section 4.4, we will discuss how to handle the case `<author>US<first> Jack</first>English</author>`. Here `US` and `English` are text associated with `author` according to the XML syntax.

3.1 A Serial Approach

A DOM tree can be built by extracting tokens (*e.g.*, start and end tags) from a document by reading it from the beginning. A stack S is maintained and is initially empty. This stack essentially stores the information of all the ancestors (in the DOM tree) of the current element being processed in the document. When a start element tag say `<e>` is read, a DOM node d_e is created for element `e` and any (attribute,value) pair that is associated with the element is parsed and stored, by creating the necessary DOM nodes. If S is not empty, then this implies that d_e's parent node has already been created. Node

d_e is linked as the rightmost child of its parent by consulting the top of stack S. (The order of siblings follows the order in which the elements appear in the document.) The pair (d_e, e) is pushed onto the stack S. If e encloses text, then a DOM node for the text is also created and linked as a "text" child of d_e. When an end element tag say </e> is read, e is checked with the top of stack S. If the element names do not match, then the parsing is aborted as the document is not well-formed. Otherwise, the top of S is popped and the parsing continues. After the last character of the document is processed, if S is empty, then the entire DOM tree has been constructed. Otherwise, the document is not well-formed.

3.2 A Parallel Approach

Given an XML document, any data parallel algorithm would perform the following tasks: (a) construct partial DOM structures on chunks of the XML document, and (b) link the partial DOM structures. Suppose n processor cores are available, each core can be assigned a set of chunks. Each core then processes one chunk at-a-time and establish parent-child links as needed.

Example 1. Figure 2 shows three chunks 0, 8, and 20 whose partial DOM trees have been constructed. Suppose elements Y and Z are child elements of X. The parent-child links between them have been created as shown.

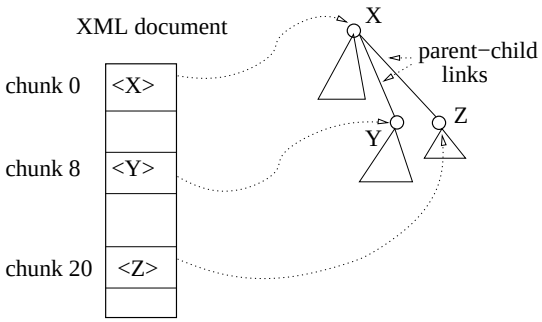


Fig. 2. Partial DOM construction & linking process

Motivating Scenario: If the linking tasks were to be done concurrently with the partial DOM construction tasks, then *synchronization* is necessary to ensure that parent-child links are updated correctly without race conditions. (Note that according the XML standard, there is an ordering among siblings based on their relative positions in the input document.) It is also possible that a parent DOM node has not been created yet, while its child DOM node (present in a subsequent chunk) has already been created. As a result, an attempt to create a link to the parent would have to wait. Mutexes can be used for the purpose of synchronization. But can synchronization primitives be avoided altogether? We believe this is possible, if we design a two-phase parallel algorithm. In the first phase, partial DOM structures are created in parallel over all the chunks. Once all the chunks have been processed, in the second phase, for each parent node, with at least one child in a different chunk, all its child nodes appearing in subsequent chunks are grouped together. Each group is processed by a single task, and all the missing parent-child links are created. Such tasks can be executed in parallel.

Challenges in ParDOM*:* Two challenges arise in the design of our two-phase parallel algorithm. First, to obtain fine-grained parallelism, each chunk should be created using a criteria independent of the underlying tree structure of a document. Second, the partial DOM structure (created for a chunk) must be located and linked correctly in the final DOM tree.

To address the first challenge, *ParDOM* adopts a flexible chunking scheme – each chunk contains an arbitrary number of start and end tags that are not necessarily matched. The required chunk size can be specified in many ways such as (a) the number of bytes per chunk, (b) the number of XML tags per chunk, or (c) the number of start tags per chunk. (We ensure that a start tag, end tag, or text is not split across different chunks.)

Example 2. Consider an XML document in Figure 3. It is partitioned into three chunks where the i^{th} chunk ($i \geq 0$) starts from the $(3 * i + 1)^{th}$ start element tag.

To address the second challenge, *ParDOM* uses a simple numbering scheme for XML elements and a stack P that stores the element numbers and names. Numbering schemes were proposed in the past for indexing and querying XML data (*e.g.*, Extended-preorder [20], Dewey [27]). Essentially, each element is assigned a unique id. Relationships between elements (*e.g.*, parent-child, ancestor-descendant, sibling) in an XML document tree can be inferred from their ids. *ParDOM* uses *preorder numbering*, where each element's id is the preorder number of its node in the XML document tree. The ids can be computed on-the-fly while extracting tokens from a document. Starting with a counter value of 0, each time a start element tag is seen, the counter is incremented and its value is the preorder number of the element. The root element is thus assigned the preorder number 1. In Figure 3, elements `book`, `last`, and `title` are assigned preorder numbers 1, 4, and 7, respectively.

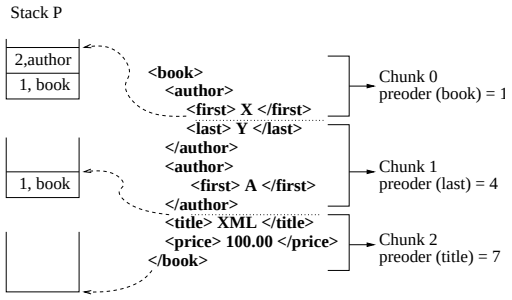


Fig. 3. Three chunks and the state of stack P

While preorder numbers can be used to determine the ordering among siblings (by sorting their ids), they cannot determine parent-child or ancestor-descendant relationships between elements. The parent-child relationship between elements is inferred using the stack P that is maintained similarly to stack S described in Section 3.1. Each entry in P is a pair *(id,element)*. Suppose the serial algorithm is applied to an input document. When a new chunk is read, the top of stack P, if P is not empty, denotes the element in some previous chunk whose end tag has not yet been encountered. In addition, exactly one entry in P denotes the parent of the first start element tag that appears in the current chunk (except for chunk 0).

Example 3. In Figure 3, the ids of the first elements in each chunk are shown. After chunk 0 is processed, the state of stack P is shown. The top element `author` in P

denotes the parent of `last` that appears in chunk 1. Similarly, the state of P is shown after processing chunks 1 and 2.

When a chunk is parsed independently, if the state of stack P is known just after processing the previous chunk, then the parent of every element in the chunk can be determined. Thus the partial DOM structure constructed for the chunk can be correctly linked to the final DOM tree. At first glance, it may seem that each chunk should be parsed serially for correctness. However, this is not the case – only stack P should be correctly initialized, and this can be done without actually constructing partial DOM trees for a chunk.

One approach is to first read the entire document, compute preorder numbers (or ids) of elements and update the stack P appropriately. At each chunk boundary, the stack P is copied and stored. We call this copy of P *a chunk boundary stack*. Once all *chunk boundary stacks* are created, the chunks can be parsed in parallel. Note that to link the partial DOM structures into the final DOM tree, the references to DOM nodes of elements whose end tags were not present in the chunk should be maintained.

4 Implementing *ParDOM*

ParDOM can be conveniently implemented in a data parallel programming model that supports `map` and `sort` operators. Given a sequence of items, a `map` operation applies a function f to each item in the sequence. Parallelism can be exploited for both the `map` and `sort` operators. For subsequent discussions, we will use the term "*a map task*" to refer to a map operator being applied to a single item in a sequence.

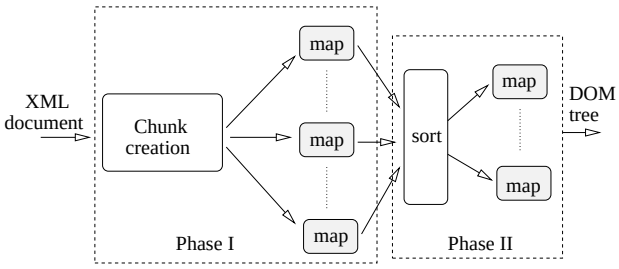


Fig. 4. Sequence of tasks in *ParDOM*

Figure 4 shows the overall sequence of tasks performed by *ParDOM*. Phase I begins with chunk creation that includes establishing chunk boundaries, assigning preorder numbers to elements, and creating chunk boundary stacks. Then the map tasks are run in parallel on all the chunks – each map task constructs partial DOM trees on its chunk. Note that as soon as the boundary of a chunk is established and its chunk boundary stack is constructed, a map task can be executed on that chunk. A map task also outputs information regarding those elements whose parents appear in some preceding chunks along with their parent ids. Once all the map tasks complete, in Phase II, the information output by the map tasks are grouped according to the parent node ids, using a sort operation. For each parent id, its group is processed by exactly one map task. A map task creates missing parent-child links between a parent DOM node and all its child DOM nodes in the group. It also ensures that siblings are in document order. Since *ex-*

Algorithm 1: Chunk creation

Global: int $nodeId \leftarrow 0$; int $chunkId \leftarrow 0$; intArray[] $firstNodeId$; stack P; stackArray[] P_c;

```
    procedure ChunkCreate(dataIn, size)
 1: begin ← dataIn;
 2: end ← begin + size + δ; /* avoid splitting XML tags and going beyond EOF */
 3: foreach (e, type) ∈ [begin, end] do
 4:     switch type do
 5:         case START:
 6:             nodeId++; /* Next preorder number */
 7:             if first START tag in chunk then
 8:                 P_c[chunkId] ← P; /* Copy stack P */
 9:                 firstNodeId[chunkId] ← nodeId;
                end
10:             P.push(nodeId, e);
11:             break;
12:         case END: P.pop(); break;
13:         otherwise do nothing;
        end
    end
14: chunkId++;
15: dataIn ← end + 1;
```

actly one map task creates the missing parent-child links, no locks are needed. Next, we describe the algorithmic details of each phase in *ParDOM*.

4.1 Phase I - Chunk Creation

The steps performed during chunk creation are shown in Algorithm 1. Each invocation of $ChunkCreate()$ identifies the boundaries of a single chunk, computes preorder numbers for the elements in it, and constructs its chunk boundary stack. The global variables are used for preorder numbering of elements and for storing chunk boundary stacks. The input arguments are $dataIn$, that points to the beginning of the current chunk, and a suggested chunk size. Lines 1-2 set up the chunk boundaries, where δ is chosen to ensure that a start tag, end tag, or text is not split across two chunks, and that the last chunk does not span beyond the end-of-file. Line 3 simply denotes tokenization of the chunk based on start and end XML tags. (The attributes and text/values are not needed at this stage and are ignored.) As the document is processed, stack P is copied and stored when the first start tag is encountered in a chunk (Line 8). Thus, a chunk boundary stack $P_c[chunkId]$ is created. (This differs slightly from our earlier discussion where P would have been copied at the beginning of a chunk.) In addition, the preorder number assigned to this element is stored (Line 9) so that during the execution of map tasks in Phase I, the element ids can be regenerated correctly. Finally, on Line 15, $dataIn$ is initialized to the beginning of the next chunk. The next invocation of $ChunkCreate()$ uses $dataIn$ as its input. Whether a document is well-formed or not can be checked during chunk creation.

4.2 Phase I - Partial DOM construction

Once Algorithm 1 completes on a chunk, a map task processes that chunk to create partial DOM trees. Algorithm 2 describes the steps involved. A local stack T, initially empty, is used to store an element's id and a reference to its DOM node. It is updated similar to stack P.

When a start of an element e is encountered, a DOM node is created, and the (attribute,value) pairs are processed and stored (Line 6). If T is empty, then e's parent is in some previous chunk. The parent of e is known from the top entry of the chunk boundary stack. A key-value pair is output where the key denotes the parent of e and the value is a reference to the DOM node for e (Lines 9-10). If T is not empty, then e's parent is the top entry of T. The DOM node for e is added as the rightmost child of its parent (Line 11).

When an end of an element e is encountered, stack T is checked. If T is empty, then e's start tag was present in some previous chunk. (Note that T cannot be empty at this point for chunk 0 if the document is well-formed.) The chunk boundary stack is updated if a start tag was already encountered while processing this chunk (Line 17). When a text is encountered, it is associated with its element using stack T (Line 21).

Finally, we pop all entries in T (Lines 24-28). These correspond to elements whose end tags were absent in the current chunk, and thus may have child elements in subsequent chunks. To link an element's DOM node correctly to a child node, a reference to it should be available in Phase II. To achieve this, a key-value pair is output where the key is the element's id and value is a special DOM node that contains the reference to its actual DOM node (Line 27). This is done to distinguish this special node from other DOM node references corresponding to child nodes output in Line 10.

Example 4. The partial DOM tree structures are shown in Figure 5 for the chunks in Figure 3. The key-value pairs are output for chunk 1 and chunk 2. The key-value pairs output in Line 27 are not shown.

4.3 Phase II - Linking Partial DOM Trees

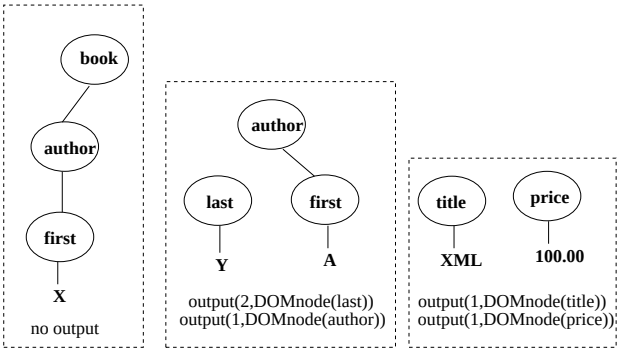


Fig. 5. Partial DOM construction in Phase I

The linking process is straightforward. The key-value pairs output in Phase I are sorted by the key *i.e.*, parent id. (The value component denotes a reference to a DOM node.) For each group of key-value pairs with the same key, a map task creates parent-child links between DOM nodes, and ensures that the child DOM nodes

Algorithm 2: Map task for Phase I in *ParDOM*

```
    procedure MapPhaseI(begin, end, chunkId)
 1: stack T; /* Each entry contains a DOM node ptr and node id */
 2: nodeId ← firstNodeId[chunkId];
 3: foreach (e, type) ∈ [begin, end] do
 4:     switch type do
 5:         case START:
 6:             create DOM node for element e including its attributes, and also store
                nodeId
 7:             let d_e denote a reference to e's DOM node
 8:             if T is empty then
 9:                 (parentId, tag) ← P_c[chunkId].top()
10:                 Output(parentId, d_e) /* Like emitIntermediate() of MapReduce */
                else
11:                 add d_e as the right most child of DOM node referenced by T.top()
                end
12:             T.push(d_e, nodeId);
13:             nodeId++; break;
14:         case END:
15:             if T is EMPTY then
16:                 if a START tag was seen in chunk then
17:                     P_c[chunkId].pop();
                    end
                else
18:                 T.pop();
                end
19:             break;
20:         case TEXT:
21:             store text as child of DOM node referenced by T.top();
22:             break;
23:         otherwise do nothing;
        end
24:     while T is not EMPTY do
25:         (nodeId, d_e) ← T.top()
26:         create a special node d_* containing the reference d_e
27:         Output(nodeId, d_*) /* Like emitIntermediate() of MapReduce */
28:         T.pop()
        end
    end
```

are in document order. Each DOM node stores its node id and can be ordered by sorting on the node id. In the interest of space, the algorithm is not outlined here.

Example 5. The partial DOM structures in Figure 5 are linked during phase II. The DOM nodes for `author`, `title`, and `price` are linked as child nodes of `book` (with id 1) after sorting them based on their node ids. The DOM node for `last` is linked to `author` (with id 2).

4.4 Extensions and Memory Requirement

To support text that are not strictly enclosed within a start and end tag the following modifications are needed. If the element containing the text appears in the same chunk, then it is linked to the text node. Otherwise, Algorithm 2 should be modified to output

a key-value pair (similar to Line 10) when a text appears as the first item. The parent is known from the chunk boundary stack. In Phase II, this text will be linked to its element DOM node.

In *ParDOM*, the additional memory required to store chunk boundary stacks depends on the number of chunks and the maximum depth of the document tree. On the contrary, PXP [23] consumes additional memory that is linear in the number of tree nodes for skeleton construction.

5 Experimental Results

We compared *ParDOM* with PXP [23] – a data parallel DOM parsing algorithm. We obtained a Linux binary for PXP from the authors. All experiments were conducted on a machine running Fedora 8 with a Intel Core 2 Quad processor (2.40GHz). The machine had 2GB RAM and 500GB disk space.

5.1 Using MapReduce to Implement *ParDOM*

We implemented *ParDOM* using Phoenix [25], which is a shared memory MapReduce implementation written in C. The code was compiled using the GNU gcc compiler version 4.0.2. The MapReduce model provides a convenient way for expressing the two phases of *ParDOM*. This model has two phases, namely, the Map phase and the Reduce phase. The input data is split, and each partition is provided to a Map task. Each Map task can generate a set of key-value pairs. The intermediate key-value pairs are merged and automatically grouped based on their key. In the Reduce phase, each intermediate key along with all the associated values is processed by a Reduce task. A MapReduce program written in Phoenix allows a user-defined split(), map(), and reduce() procedures. In our MapReduce implementation of *ParDOM*, split() implemented Algorithm 1, map() implemented Algorithm 2, and reduce() implemented the steps described in Section 4.3.

5.2 *ParDOM* vs PXP

ParDOM and PXP were evaluated on a variety of XML datasets with different structural characteristics and sizes.[4] These datasets were obtained from University of Washington [29]. Figure 6 shows the characteristics of each dataset in terms of its size, number of elements and attributes, and maximum tree depth. DBLP contains computer science bibliographic information. SWISSPROT is a curated protein sequence database. TREEBANK captures linguistic structure of a Wall Street Journal article using parts-of-speech tagging. It has deep, irregular structure. LINEITEM contains data from the TPC-H Benchmark [28].

PXP requires scanning the input document during a preparsing phase for constructing a skeleton of the document. A skeleton is a light-weight representation of the document's structure and does not involve the creation of DOM tree nodes. Then the document is partitioned into tasks (denoted by subtrees) using the skeleton, and these tasks

[4] These datasets are different from those used by the authors of PXP [23].

are run in parallel to create partial DOM trees. Preparsing and task partitioning are performed sequentially. Finally, PXP requires a postprocessing phase to remove some temporary DOM nodes.

ParDOM also requires scanning the input document during chunk creation (Algorithm 1). However, a careful implementation in Phoenix allows us to interleave the chunk creation phase with the Map tasks in Phase I. Note that once a chunk boundary stack is computed for a chunk, it is ready to be processed by a Map task.

Dataset	Size	Max depth	# of elements	# of attributes
DBLP	127MB	6	3332130	404276
SWISSPROT	109MB	5	2977031	2189859
TREEBANK	82MB	36	2437666	1
LINEITEM	30MB	3	1022976	1

Fig. 6. XML datasets and their characteristics

Measurements & Results For each dataset, we ran *ParDOM* and PXP on 2, 3, and 4 cores. For *ParDOM*, chunks were created by specifying *bytes per chunk*, and each chunk was extended to contain the nearest end tag of an element. The PXP code provided to us could not process XML documents beyond a certain size and crashed during preparsing. Therefore, we created smaller datasets of size 8MB, 16MB, and 32MB from our original datasets. We measured the wall-clock time and computed the average over three runs. Each dataset was read once before parsing so that it is cached in the file system buffer to avoid I/O while parsing.

To compute speedup, we ran a serial parsing algorithm (Section 3.1) and PXP on one core. Let us call them as T_s and T_{PXP}, respectively. *ParDOM*'s speedup was measured by computing the ratio of T_s with its parallel parsing time. (The parallel parsing time included the cost of chunk creation.) PXP's speedup was measured by computing the ratio of T_{PXP} with its parallel parsing time. (The parallel parsing time included the cost of preparsing.)

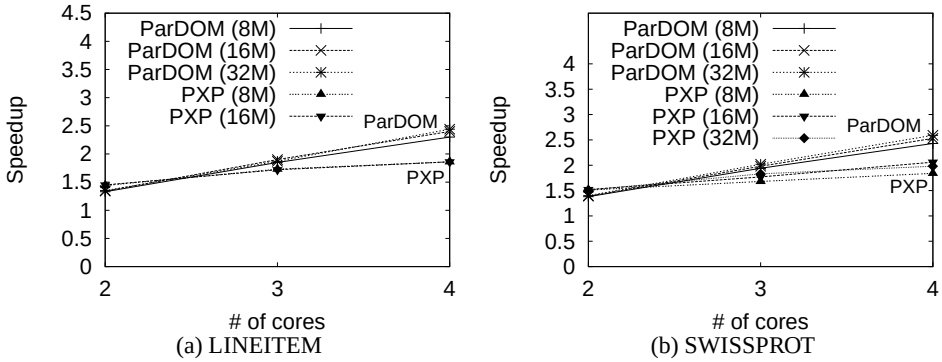


Fig. 7. Speedup measurements

Speedup: Figure 7(a) and 7(b) show the speedup of *ParDOM* and PXP for LINEITEM and SWISSPROT, respectively. The chunk size of 256KB was selected for *ParDOM*, beyond which the parallel parsing time did not improve significantly. Clearly, *ParDOM* had better speedup than PXP at 4 cores for both LINEITEM and SWISSPROT. *ParDOM* achieved a speedup of around 2.5 with 4 processing cores. (Note that PXP crashed for 32MB of LINEITEM dataset during preparsing phase, and hence is not shown in the plot.) Interestingly, PXP failed to parse TREEBANK and DBLP even for 8MB dataset sizes and crashed. The crash occurred in the preparsing phase. In these datasets, the fanout at nodes other than the root were not large. Further, TREEBANK had deep tree structures. This clearly demonstrates the superiority of *ParDOM* over PXP for parallel DOM parsing as it can process a variety of tree structures and document sizes.

Figure 8(a) shows the speedup for *ParDOM* on all the four datasets, each of size 64MB. We achieved the best speedup of 2.61. We observed similar trends in the speedup for *ParDOM* when the original datasets in Figure 6 were used.

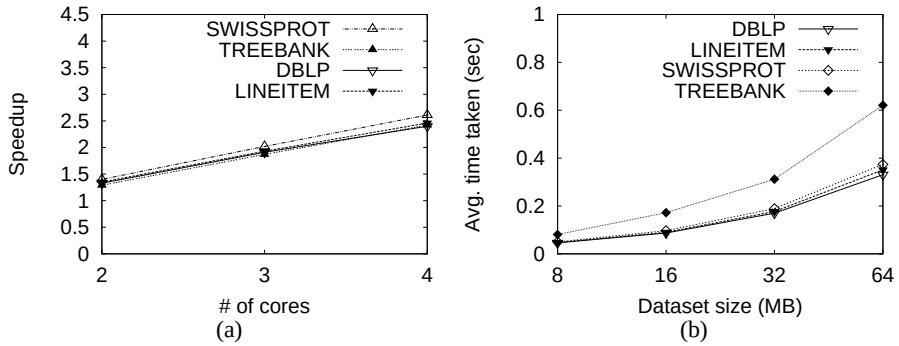


Fig. 8. (a) Speedup of *ParDOM* (64MB). (b) Data scalability

Data Scalability: To measure how *ParDOM* scales with increase in dataset size, we measured the average parsing time (over 3 runs) for datasets size of 8MB, 16MB, 32MB, and 64MB. The results for 4 cores is plotted in Figure 8(b). For instance, *ParDOM* required 0.312 secs and 0.621 secs to process 32MB and 64MB of TREEBANK, respectively.

For *ParDOM*, we measured the effectiveness of our simple chunking scheme on the distribution of load among the Map tasks in Phase I. We used the original datasets in Figure 6. A Map task that processed more elements created more DOM nodes. Figure 9 shows the mean and standard deviation of the number of elements processed per Map task excluding the last Map task that can have a smaller chunk size. We observed that for TREEBANK and LINEITEM the load was well-balanced among Map tasks as compared to DBLP and SWISSPROT. This is evident from the smaller σ values. DBLP and SWISSPROT datasets contained text of varied lengths that resulted in higher σ values. Thus chunking based solely on *bytes per chunk* may not be ideal for such datasets.

We also measured the load during Phase II of *ParDOM*, by considering the number of child nodes that were linked per task, excluding the root node. (The root node of each dataset had very large fanout.) The total, mean, and standard deviation for the

number of child links created are shown in Figure 9. Note that more tasks were required for TREEBANK as compared to the other datasets because an average of 1.5 child nodes were linked per task. SWISSPROT had larger fanout among nodes as compared to DBLP and this is reflected in the total number of child nodes that were linked in Phase II.

Dataset	# of elements per Map task Phase I		Total # of parent−child links created in Phase II		
	Mean	σ	Total	Mean	σ
DBLP	24338.6	1545.3	1670	5.3	5.6
SWISSPROT	22789.8	725.0	4155	9.2	16.4
TREEBANK	23323.3	274.6	1622	1.5	0.9
LINEITEM	29041.9	33.1	425	5.4	4.8

Fig. 9. Load measurement

Load Balancing: Finally, we measured how much time was spent in the Map and Reduce phases in our *ParDOM* implementation. We used the original datasets for this experiment. We observed that in all cases the Reduce phase consumed less than 8% of the total time.

6 Conclusions

ParDOM is a data parallel XML DOM parsing algorithm that can leverage multicore processors for high performance XML parsing. *ParDOM* offers fine-grained parallelism by using a flexible chunking scheme that is oblivious to the structure of the XML document. *ParDOM* can be conveniently implemented in a data parallel language that supports `map` and `sort` operations. Our empirical results show that *ParDOM* provides better scalability than PXP [23] on commodity multicore processors. Further, it can process a wide variety of datasets as compared to PXP.

Acknowledgments We thank the authors of PXP for their code and the anonymous reviewers for their insightful comments.

References

1. Intel XML Software Suite Performance Paper. http://intel.com/software/xmlsoftwaresuite.
2. Microsoft XML Core Services (MSXML). http://msdn.microsoft.com/en-us/xml/.
3. Xerces-C++ XML Parser. http://xerces.apache.org/xerces-c/.
4. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
5. A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon. XML path language (XPath) 2.0 W3C working draft 16. Technical Report WD-xpath20-20020816, World Wide Web Consortium, Aug. 2002.
6. L. Cable and T. Chow. JSR 173: Streaming API for XML, 2007. http://jcp.org/en/jsr/detail?id=173.
7. R. D. Cameron, K. S. Herdy, and D. Lin. High performance XML parsing using parallel bit stream technology. In *CASCON '08: Proc. of the 2008 conference of the center for advanced studies on collaborative research*, pages 222–235, New York, NY, 2008.

8. M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *Proc. of the 2007 Workshop on Declarative Aspects of Multicore Programming*, pages 10–18, Nice, France, Jan. 2007.
9. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the OSDI '04*, San Francisco, CA, Dec. 2004.
10. R. A. V. Engelen. A framework for service-oriented computing with C and C++ Web service components. *ACM Transactions on Internet Technology*, 8(3):1–25, 2008.
11. Z. Gao, Y. Pan, Y. Zhang, and K. Chiu. A high performance schema-specific xml parser. *IEEE Intl. Conf. on e-Science and Grid Computing*, pages 245–252, Dec. 2007.
12. A. Ghuloum, T. Smith, G. Wu, X. Zhou, J. Fang, P. Guo, B. So, M. Rajagopalan, Y. Chen, and B. Chen. Future-proof data parallel algorithms and software on intel multi-core architecture. *Intel Technology Journal*, 11(4):333–348, 2007.
13. A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-scale Architectures, 2007. Intel White Paper 2007.
14. O. Goldman and D. Lenkov. XML Binary Characterization. Technical report, World Wide Web Consortium, Mar. 2005.
15. G. Grohoski. Niagara 2: A highly threaded server-on-a-chip. In *18th Hot Chips Symposium*, Aug. 2006.
16. M. Huhns and M. P. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1):75–81, Jan. 2005.
17. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.
18. M. Kay. SAXON: The XSLT and XQuery Processor. http://saxon.sourceforge.net.
19. M. G. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, and M. Mercaldi. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. In *Proc. of the 15th International Conference on World Wide Web*, pages 93–102, New York, NY, 2006.
20. Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proc. of the 27th VLDB Conference*, pages 361–370, Rome, Italy, Sept. 2001.
21. D. Megginson. Simple API for XML. http://sax.sourceforge.net/.
22. M. Nicola and J. John. XML parsing: a threat to database performance. In *Proc. of the 12th International Conference on Information and Knowledge Management*, pages 175–178, 2003.
23. Y. Pan, W. Lu, Y. Zhang, and K. Chiu. A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs. In *Proc. of the 7th International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 351–362, Washington D.C, May 2007.
24. Y. Pan, Y. Zhang, and K. Chiu. Simultaneous transducers for data-parallel XML parsing. In *Proc. of Intl. Symposium on Parallel and Distributed Processing*, pages 1–12, Apr. 2008.
25. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA)*, Phoenix, AZ, Feb. 2007.
26. L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.
27. I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. of the 2002 ACM-SIGMOD Conference*, pages 204–215, June 2002.
28. TPC. TPC-H. http://www.tpc.org/tpch/, 2002.
29. UW XML Repository. http://www.cs.washington.edu/research/xmldatasets, 2001.
30. W3C. The document object model, 1998. http://www.w3.org/DOM.
31. Y. Wu, Q. Zhang, Z. Yu, and J. Li. A Hybrid Parallel Processing for XML Parsing and Schema Validation. In *Proceedings of Balisage Markup Conference*, 2008.
32. J. Zhang and K. Lovette. XimpleWare W3C Position Paper. In *W3C Workshop on Binary Interchange of XML Information Item Sets*, 2003.