RC24659 (W0810-015) October 2, 2008
Computer Science

IBM Research Report

Parallelization of XPath Queries Using Multi-Core Processors: Challenges and Experiences

Rajesh Bordawekar, Lipyeow Lim
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Oded Shmueli
Computer Science Department
Technion
Haifa 32000
Israel

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Parallelization of XPath Queries using Multi-core Processors: Challenges and Experiences

Rajesh Bordawekar [#1], Lipyeow Lim [#2], Oded Shmueli [*3]

[#]*IBM T. J. Watson Research Center, Hawthorne, NY 10532*
[1]bordaw@us.ibm.com, [2]liplim@us.ibm.com

[*]*Computer Science Department, Technion, Haifa 32000, Israel*
[3]oshmu@cs.technion.ac.il

***Abstract*— In this study, we present experiences of parallelizing XPath queries using the Xalan XPath engine on shared-address space multi-core systems. For our evaluation, we consider a scenario where an XPath processor uses multiple threads to concurrently navigate and execute individual XPath queries on a shared XML document. Given the constraints of the XML execution and data models, we propose three strategies for parallelizing individual XPath queries: Data partitioning, Query partitioning, and Hybrid (query and data) partitioning. We experimentally evaluated these strategies on an x86/Linux multi-core system using a set of XPath queries, invoked on a variety of XML documents using the Xalan XPath APIs. Experimental results demonstrate that the proposed parallelization strategies work very effectively in practice; for a majority of XPath queries under evaluation, the execution performance scaled linearly as the number of threads was increased. Results also revealed the pros and cons of the different parallelization strategies for different XPath query patterns.**

I. Introduction

XPath is an expression language used for processing data represented in XML documents [1]. XPath uses a path notation for navigating through the hierarchical structure of an XML document. XPath operates on the XML data model [2] which represents the abstract, logical structure of an XML document as a rooted tree. XPath can be embedded in host languages such as XQuery [3], XSL [4], SQL [5] and it also forms an integral component of various web services interfaces [6]. XPath is also being used for expressing constraints in various XML Schema languages [7].

As XPath is a critical component in many XML-based applications, it is imperative to maximize its performance. While there has been extensive work in optimizing performance of a single XPath query by improving its traversal pattern [8], there have been relatively fewer studies to evaluate how the underlying processor architecture affects XPath performance. This issue has become important due to the wide-spread availability of commodity multi-core processors. Current desktop machines support 2 quad-core processors, i.e., 8 cores, each of which can presumably run 2 hardware threads. Current projections are that by the end of next year, the desktop machines will have processors that can support upto 32 cores each. While most state-of-the art XPath processors, such as the Apache Xalan, can support *concurrent* XPaths (i.e., multiple threads issuing XPath queries simultaneously against the same Xalan instance), each XPath query is still executed serially. While the concurrent XPath execution improves the overall latency, the overall bandwidth can be improved by accelerating individual XPath queries via parallelization. Further, the performance of an individual query can degrade due to additional costs due to locking and increased memory consumption. Parallelization of XPath queries is also essential for parallelizing host languages, such as XSL or XQuery.

In this study, we evaluate opportunities for parallelizing a single XPath query in a shared-address space environment on *commodity* multi-core processors. We assume that the XML document is pre-parsed and can be concurrently accessed by multiple threads. The key aim of this study is understanding the challenges in parallelizing XPath queries using a real production-grade system. Towards this goal, we are emulating a parallel XPath processor by using the latest Xalan XPath processor in a multi-threaded environment. Our parallel XPath processor takes vanilla XPath queries and executes them in parallel on the underlying multi-core processor.

We propose three new schemes for parallelizing XPath queries: (1) Data Partitioning, (2) Query Partitioning, and (3) Hybrid partitioning that combines both the data and query partitioning schemes. These approaches exploit both the read-only nature of XPath processing and the intra-step parallelism within every step of an XPath query. All the three approaches achieve parallelism via partitioning traversals over the XML documents. The data partitioning approach executes the same (sub)query on different sections of the same XML document whereas the query partitioning approach executes different (sub)queries on the same XML dataset. We have implemented these strategies in a multi-threaded driver which first processes the XPath queries, concurrently invokes the Xalan XPath processor via the XPath APIs, and merges or joins the intermediate results to compute the final result set of the query. We evaluated our implementation using a set of complex queries over three large XML documents (XMark, DBLP, and PENN Treebank). Experimental results demonstrate that our proposed parallelization schemes work very well in practice. We observed that in many cases, as the number of threads was increased, we achieved linear speedup. In some cases, we observed an order-of-magnitude speedup by executing the modified query in the parallel manner. We also observed that

parallelization *reduced* the performance of queries with low selectivity or those having structural issues (e.g., low fanout).

Our study makes the following contributions:

- To the best of our knowledge, this is the first study to analyze the parallelization of individual XPath queries. To address this problem, we have proposed three different parallelization strategies.
- We have implemented these strategies using a production-grade XPath processor and evaluated it using realistic and complex XPath queries on large, structurally diverse XML documents. Our experimental evaluation has conclusively demonstrated the effectiveness of our strategies for achieving scalable performance from the parallel execution of the XPath queries.
- Our experiments have identified several key issues that need to be taken into account while implementing a parallel XPath processsor. The experiments have also releaved many interesting research issues that need to be investigated.

The rest of the paper is organized as follows: Section II briefly reviews the related work in parallelization of XML and SQL queries. Section III overviews XPath semantics and parallelization strategies. Section IV discusses parallelization issues in the context of XPath processing and presents three new parallelization strategies. Section V presents results from our experimental evalation and Sec VI presents conclusions and describes future work.

II. Related Work

Parallelization of SQL queries has been extensively studied in the context of both distributed and centralized repositories [9], [10], [11]. Most commercial database systems support parallel query processing using either the shared-nothing or shared-everything architectures. Parallelization has been extremely effective in practice, for both OLTP, OLAP/data warehousing, and web applications. Parallelization of SQL queries differs from the XPath parallelization as follows: (1) The SQL workload supports in-place updates, while XPath processing is read-only, (2) The relational data has a regular 2-dimensional structure that is suitable for partitioning either along rows or columns. The rooted hierarchical structure of XML is not inherently suited for balanced data partitioning., (3) Using hash-partitioning, it is easier to physically distribute relational data across multiple storage nodes while maintaining data affinity. For XML documents, it is very difficult to effectively physically cluster related items, and (4) Unlike relational data, XML can be accessed and stored in many different ways, e.g., in-memory, streaming, relational or native storage. Any XPath parallelization algorithm needs to be tuned to match the XML storage and access characteristics.

Past studies have evaluated XML processing either in distributed or concurrent scenarios. Most existing XML processing engines are thread-safe and allow multiple threads to issue concurrent XPath queries againts an XML document. Distributed XML processing is discussed in [12], [13]. Boolean XML queries expressed in X_BL, a language containing forward axes, labels, text and the Boolean operators and, or and not are treated in [12]. The algorithms are inspired by partial evaluation. In essence, the whole query and all its sub-queries are evaluated in each distributed fragment. Sometimes, data is unknown (at some leaves) as it resides in another fragment and is replaced by Boolean variables. Therefore, the computation at a fragment may result in a Boolean expression in terms of these variables, hence the relationship to partial evaluation. When all fragments complete computing, the final Boolean result may be resolved. The main advantage of the scheme is that computation at various fragments may proceed in parallel and incurs a computational overall cost similar to that of a centralized mechanism. A disadvantage is the usage of various vector data structures, on a per node basis. The work in [13] extends the ideas from Boolean to node returning queries. The idea is to normalize queries, and to treat separately the qualifiers in a query and the selection (main skeleton) part of the query. The various qualifiers are treated using the techniques of [12]. The evaluation of the selection path also uses partial evaluation ideas to "transmit" information between fragments.

The overall scheme of [12], [13] is elegant and theoretically efficient, yet space consuming. The scheme may be of interest in parallel evaluation, for example by introducing fragments and carrying the computation in parallel on these fragments. As it stands, these fragments need be constructed statically. Issues of load balancing and performing the partition optimally have not been addressed and may be of interest. The competitiveness of such a scheme may be hindered by the memory consumption which may turn out to be a bottleneck.

The work of [14] treats distributed query evaluation on semistructured data and is applicable to XML query processing as well. It treats three overlapping querying frameworks. The first is essentially regular expressions. The second is based on an algebra, C, and is aimed at restructuring. An algebraic approach based on query decomposition is provided for solving C queries. Here a query is rewritten into subqueries implied by the distribution. These queries are evaluated at the distributed fragments to produce partial results which are later assembled into a final result. The third is *select-where queries*, declarative queries combining patterns, regular expressions and some restructuring. Here, processing is done in two stages where the first is evaluating a related query that is expressible in C, and hence parallelizable, which produces partial results that are then used to form the final result at the client. The focus is on communication steps.

One may approach the problem of parallelizing XML query processing within the general framework of efficiently programming and coordinating multiprocessor computations, see [15] for a comprehensive treatment. Such an approach appears in [16], [17]. Execution of various XML processing tasks (not including query processing) appears in [16] in the context of multicore systems. The idea is to have a crew of processes each taking tasks out of its own work queue. Once tasks are exhausted, a process may *steal* tasks off queues

of other processes. Tasks are ordered so that processing is done at the top whereas stealing is done at the bottom. This creates less contention. A scheme is presented for constructing the final result. The paper presents the idea of *region-based task partitioning* to increase task granularity. Parallel XML DOM parsing is presented in [18], [17]. The first paper uses a dynamic scheme for load-balancing among cores. The idea in the second paper is to statically load-balance the work among the cores. This latter work is targeted at large shallow files containing arrays and does not scale to many cores (beyond six).

III. Parallelizing XPath: Preliminaries

A. Overview of the XPath Processing Model

An XPath expression consists of a sequence of location steps. Each location step has three components: an axis, a node test, and a predicate. An XPath expression is evaluated with respect to a context node. Given a context node of an abstract XML tree, an XPath expression uses the specified axis to navigate the XML tree. The XPath standard specifies 13 axes for navigation. The node test and the predicate are used to select the nodes specified by the current axis. The node test can select the node depending on its type, e.g., attribute. The predicate can further prune the selection by evaluating various properties (e.g., position) of the navigated nodes. The XPath experession returns a set of unique nodes (or a null node set), ordered in either document or reverse document order.

XPath's execution model is inherently sequential: each location step operates on the node set returned as a result of evaluating the previous location step or the starting context. Therefore, in normal circumstances, execution of location steps can not be reordered. However, XPath provides significant opportunities for parallelism: (1) Accesses to the XML documents are read-only, (2) Execution of a location step can be reordered in any manner as there are no intra-step dependences, and (3) Presence of indexes enables the original XML query to be split into different independent sub-queries that could be executed in parallel, and the final results computed either via merging or performing a union of the resultant node sets.

B. Overview of Parallelization Issues

The key motivation for parallelizing an application is to improve its performance by using multiple processors. The behavior of an application can be characterized by two key attributes: its data and iteration space. The data space captures the structural properties of the program data structures, e.g., dimensions of an array, and scope of each dimension, etc., while the iteration space encodes how these data structures are traversed, e.g., if there are any read-write dependences between iterations. The iteration space determines the portion of the application that can be parallelized.

The most common approach for parallelizing an application is to partition its parallelizable work by distributing the program data among the participating processors. There are several ways of distributing a data set, e.g., partitioning consecutive columns or rows of an array across multiple processors. In a distributed-memory machine, each processor stores its assigned data into its own local memories, whereas on a shared-memory machine, the data is logically partitioned. The data distribution strategy determines the amount of interactions between the participating processors. On a distributed-memory machine, the processors interact via explicit messages, and on a shared-memory system, the inter-processor interaction is executed via shared program variables. For an application, the amount of inter-processor interaction is decided by its data partitioning strategy. Further, data partitioning determines if the workload is equally balanced across the processors. If an application's data partitioning strategy can match its iteration pattern, it can lead to lowered inter-processor communication and better load-balancing. This can result in a scalable parallel application, in particular, for applications whose sequential portion is not dominant.

IV. XPath Parallelization

In this work, we consider the scenario where an XML document has been already parsed and the pre-parsed representations allows applications to traverse the document according to the XML data model.

A. XML Parallelization Issues

For identifying a suitable XPath parallelization strategy, one has to evaluate the following key issues:

- Data Partitioning Strategy: As discussed earlier, the data partitioning strategy is key to achieving scalable performance from a parallel application. For XPath processing, the data space is defined by the abstract tree representation of the XML document and the iteration space is defined by the XPath queries. An XPath query navigates the abstract rooted XML tree using one or more of the XPath axes. To parallelize the XPath query execution, one needs to logically partition the tree as per its traversal pattern. Unlike an array, the rooted tree cannot be easily partitioned into distinct partitions. Therefore, a part of the tree, notably the section near the root node, is usually shared and the descendant subtrees are assigned to different processors. For example, consider the XML document representing the DBLP dataset (Figure 1(A)). Figure 1(B) represents the corresponding abstract XML tree. Figure 1(C) presents a partition of the tree, where the root node and its children are shared by all processors and descendants of every child of the root node are allocated to distinct processors. While the shared section of the XML tree is traversed by only one processor, different processors can concurrently traverse their assigned tree sections. Thus the extent of the shared portion determines the amount of serial work in the application.
- Storage Model: There are several ways of storing an XML document while complying with the XML data model. We assume that the pre-parsed XML document is stored using an in-memory, non-relational representation and

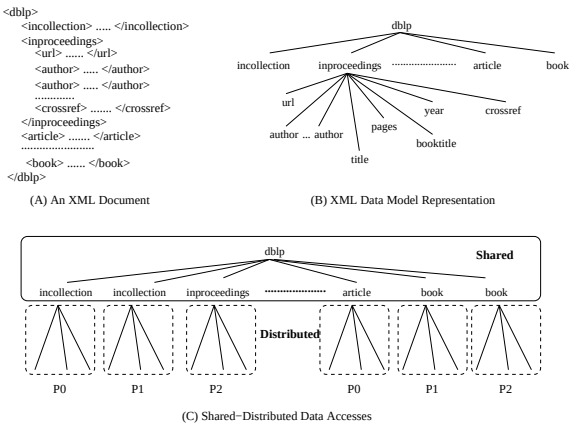

Fig. 1. Partitioning an XML Document

it can be accessed concurrently by multiple application threads in a shared-address space environment. As the data partitioning is implemented on the logical data model, concurrent accesses to distinct subtrees result in accessing different portions of the stored data.

- While deciding on the XPath parallelization strategy, one needs to evaluate the following cost metrics:
  - Execution cost: The execution cost of an XPath operation can be computed in terms of number of traversals of an XML tree and the amount of temporary space generated by the operation. Ideally, in a parallel execution, the total number of traversals by participating processors should match the number of traversals in the corresponding serial execution. Similarly, the space consumption of the parallel implementation should match the consumption of the original serial program.
  - Locking cost: In a parallel implementation, multiple proceossors often use common data structures for managing shared data. Accesses to these data structures is governed by locks. The locking costs increase substantially as the number of participating processors increases. Inefficient implementation of locks can lead to deadlock or livelock scenarios. The parallel algorithm should be designed such that the amount of sharing among the processors is minimized, thus reducing the impact of locks on the overall performance.
  - Merging cost: In a parallel application, temporary results computed by different processors must be merged to compute the final result. Care should be taken that the merging operation does not become performance bottleneck.
- Load balance: Ideal data partitioning results in participating processors performing equal amount of work. In practice, it is often difficult to achieve load balance, in particular, when there is no prior information on the input dataset and its usage pattern. In such cases, simple data partitioning heuristics, e.g., partitioning the input data sets in a round-robin fashion, are often applied to minimize load imbalance.

B. Parallelization Strategies

We now present three strategies for parallelizing an XPath query in a shared-address space environment: (1) Data partitioning, (2) Query partitioning, and (3) Hybrid partitioning. All three approaches exploit the read-only characteristics of XPath processing. These approaches differ in the way the shared XML data is partitioned across multiple processors and how the input query is executed on the partitioned data. As these strategies are defined over the XML data model, they can be adapted to any XML storage format. All the three approaches require some form of query rewriting.

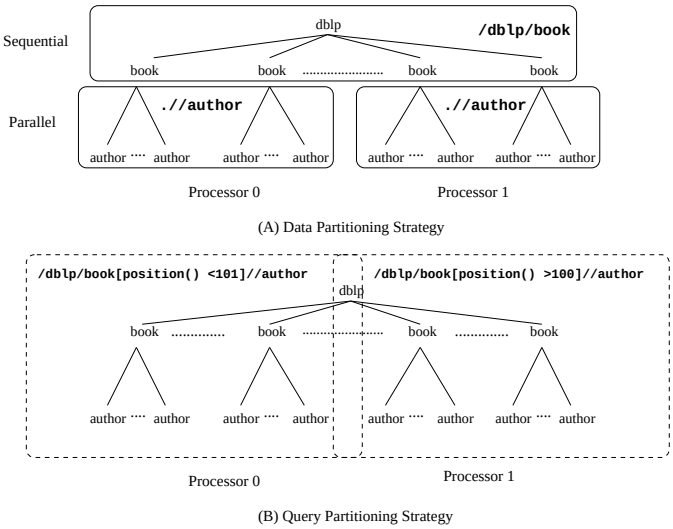

Fig. 2. Parallelization of `/dblp/book//author` via (A) Data partitioning and (B) Query partitioning.

In the data partitioning approach, the input query is split into serial and parallel sub-queries. The serial part of the XPath query is executed by a single processor on the entire document. The resulting node set is then distributed across multiple processors, e.g., using a block-distribution. Each participating processor then uses the locally assigned node set as a set of the context node set and executes the parallel sub-query on every context node. This approach achieves parallelism by executing the same XPath query concurrently on different sections of the XML document. Thus, this approach follows the data parallel style of parallel programming. The scalability of the data partitioning scheme is determined by the serial sub-query; an expensive serial sub-query can degrade the performance of the overall query. Therefore, it is important to effectively the partition the query so that the serial portion performs the least amount of work. Figure 2 (A) illustrates the execution of an XPath query, `/dblp/book//author`, using the data partitioning strategy. This query is split into two sub-queries: `/dblp/book` and `.//author`. The sub-query, `/dblp/book` is executed in a serial manner and the resulting node set of `book` nodes is partitioned across

two processors. Each processor then executes the sub-query, `.//author`, on the set of `book` nodes assigned to it. As a result, each processor navigates a distinct part of the XML tree concurrently. The result of the original query can be then computed by merging the local results from the participating processors.

In the query partitioning approach, the input query is rewritten into a set of sub-queries that can ideally navigate different sections of the XML tree. The number of sub-queries matches the number of participating processors. In many cases, each sub-query is an invocation of the original query using different parameters. Each processor executes its assigned sub-query on the entire XML tree. The final result of the query can be then computed using either the union or merge of the per-processor node sets. Unlike the data partitioning approach, this approach achieves parallelism via exploiting potentially non-overlapping navigational patterns of the sub-queries. In this approach, the overall scalability is determined by the range of the parallel sub-queries. If their traversals do not overlap significantly, the query performance will improve as the number of processors is increased. Figure 2(B) illustrates the execution of the same XPath query, `/dblp/book//author`, using the query partitioning approach. In the illustrated scenario, the original query is rewritten for two processors (assume that there are 200 `book` children of the `dblp` node): the first processor executes, `/dblp/book[position() <101]//author` and the second processor executes `/dblp/book[position() > 100]//author`. The final result, i.e., the set of `author` nodes, can be computed as a merge of the two local result sets. The query partitioning approach can be also applied to XPath queries that contain independent sub-queries (e.g., path predicates that use absolute paths or multi-attribute predicates) or those queries that can use indexes to execute part of the navigation.

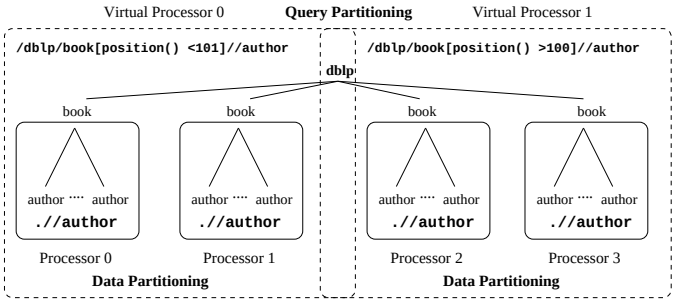

Fig. 3. Parallelization of `/dblp/book/author` via Hybrid partitioning.

The data and query partitioning approaches can be used together to form a hybrid partitioning approach. In this approach, the input XPath query is first partitioned into different sub-queries for a set of *virtual* processors (query partitioning). Each virtual processor is a set of physical processors and it executes its assigned sub-query using the data partitioning approach. Figure 3 demonstrates the hybrid partitioning approach for the XPath query, `/dblp/book//author`, on 4 processors. First, the input query is partitioned into two sub-queries. Each virtual processor then executes part of its query in the serial form (e.g., `/dblp/book[position() <101]` on the virtual processor 0), and the remaining (`.//author`) in the parallel form by partitioning the intermediate result set of the `book` nodes between 2 processors (e.g., on processors 0 and 1 for the group of virtual processor 0).

V. Experimental Evaluation

A. Experimental Setup

We have evaluated the XPath parallelization strategies using the latest version of the Xalan XSLT processor (version 1.10 of the Xalan C++ implementation). The Xalan XSLT processor allowed us to evaluate complex XPath queries using a state-of-art XPath processor. For our evaluation, we developed a pthreads-based multi-threaded driver that simulated a parallel XPath engine. We did not modify Xalan's XPath query processor. The driver first processed the input XPath queries and then concurrently invoked the Xalan XPath APIs using the different parallelization strategies. The input XML document was parsed only once and the parsed representation was shared across multiple threads. Our implementation used hash-based merge-join to merge temporary result sets from different threads. Current, we are not using any shared updatable data structures, therefore, our implementation does not incur any locking costs. While using the data partitioning strategy, we partitioned the input context node set across the threads using the block distribution pattern. For example, given a context nodeset of size 1024 and 4 application threads, every thread gets 256 nodes: thread 0 gets first 256 nodes, thread 1 gets the next 256 nodes, and so on.

Name	Size (MByte)	Elements	Attributes	Depth
XMark.xml	558	8353174	1914186	7
DBLP.xml	442	2164363	0	3
treebank.xml	8.6	2437666	1	7

TABLE I
Characteristics of the XML Documents

We tested our implementation on a 2-node dual-core x86/Linux machine. Each core was a 2-way SMT, hence we were able to run upto 8 user threads. For our experiments, we used three different XML datasets: XMark [19], DBLP [20], and the Penn Treebank [21]. XMark is a synthetic dataset that represents auction data, DBLP is a bibliographical dataset, and the Penn treebank represents linguistic data. Table I presents the structural characteristics of these data sets. Among the three, the treebank dataset is a deep recursive dataset, while the DBLP dataset is shallow and wide.

B. Evaluation of the XPath Queries

Table II presents a set of XPath queries used for the experimental evaluation. In total, we evaluated 12 queries: 6 for the XMark document, and 3 each for the DBLP and treebank documents. Our queries were selected so as to evaluate the

three partitioning strategies under different constraints. The queries include those with long chains of child steps, path predicates, conjunctive and disjunctive predicates, functions such as `last()`, `name()`, and `count()`, different axes (e.g., the "`//`", attribute, parent, following-sibling). We also computed the number of unique absolute paths in each document and used the path statistics to rewrite the original queries. We did not use path indexes for query execution. For each query, we hand-modified the driver code to use either the data or query partitioning. We have applied query partitioning for two cases: (1) when we can partition predicate computations and (2) when we can partition range computations. We have applied hybrid partitioning to cases involving range partitions.

For each query, we present three sets of results (Table III). First, we present the sequential execution time of the original query, the best total time for the split queries in the data partitioning strategy (including the corresponding serial and parallel execution times, and the number of threads used in that parallel execution), and the best time for the query modified according to the query partitioning strategy, along with the number of threads. Second, we present performance of the split queries in the data partitioning strategy as the number of threads is increased from 1 to 8. It is important to note that the performance of the split queries differs from that of the original query.

Therefore, for the data partitioning strategy, we report *absolute* scaleup numbers using the performance of the split queries on a single processor (the serial and parallel queries, one by one, are executed in a sequence by the same processor) as the baseline (i.e., absolute scaleup=(serial time for the split query)/(best total time for the split query)). We also report *relative* scaleup numbers using the performance of the original query as the baseline (i.e., relative scaleup=(serial time for the *original* query)/(best total time for the split query)). Finally, we present scaleup numbers for the query partitioning strategy. In all cases, the queries rewritten using the query partitioning strategy were a version of the original query with different input parameter values. Therefore, we used the sequential execution time of the original unmodified query as the baseline performance number to compute *absolute* scaleup. The bold-faced query keys in Table III represent cases that experience slowdown after parallelization. Note that in many cases, the performance does not improve when the number of threads is increased above 4. This is because we are running our experiments on a 4-core machine.

The XM1 query (Table II) selects a set of nodes with specified names using a disjunctive predicate and the `name()` function. This query can be executed using both the data and query partitioning approaches. In the data partitioning strategy, original query is split into two subqueries: `/site/*` (serial), and `self::*//*[name(.)...]` (parallel). In the query partitioning approach, the original query gets partitioned into three separate queries, each checking for a single predicate. These three queries are then invoked on the entire document and the resultant node set is unioned to obtain the final result. Figure 4 illustrates the performance of data partitioning strategy as the number of threads is increased from 1 to 8. As the Figure illustrates, in all cases, the serial portion is not the bottleneck and the parallel performance improves as the number of threads is increased (i.e., absolute scaleup of 1.93 and relative scaleup of 16.05).

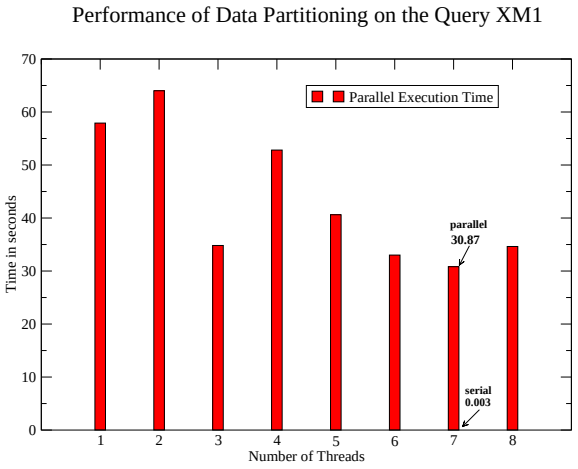

Fig. 4. Data Partitioning of the query XM1. The sub-query `/site/*` is executed serially and the sub-query `self::*//*[name(.)..]` is executed in parallel. The serial execution cost is too small to be represented.

The XM2 query (Table II) returns values of the `id` attributes of all items of the category, `category52`. This query exhibits traversals on the `//`, parent, and attribute axes and a value predicate. This query can be parallelized using both data and query partitioning as follows: In the data partitioning strategy, the original query can be split into 2 ways. In the first approach (XM2(a)), the original query gets partitioned into the serial sub-query, `//incategory`, and a parallel sub-query, `self::*[..]/parent::item/@id`. The second appraoch, (XM2(b)), splits the original query into a serial sub-query, `/site/*` and a parallel sub-query, `.//incategory[..]/parent::item/@id`. In the query partitioning strategy, we re-write the original query into the following two sub-queries: `//item[./incategory/@category=''category52'']`, and `//item[./@id]`. These two queries are executed in parallel by two different threads; the local results are then merged by the driver thread, and the corresponding attributes are returned.

Table III presents the evaluation of these approaches for the query XM2. The performance of two data partitioning approaches varies substantially; the cost of the queries generated in the XM2(a) approach is substantially more than that of the ones generated in XM2(b). Figure 5 presents a more detailed performance evaluation of the two approaches for a varying number of threads. As Figure 5 illustrates, in XM2(a), even though the parallel execution time improves as the number of threads is increased, the overall performance gets affected by the serial sub-query. This sub-query does far more work than the corresponding serial sub-query in XM2(b) (`//incategory` returns 411658 nodes whereas `/site/*`

Document	Key	XPath Query
XMark.xml	XM1	`/site//*[name(.)=''emailaddress'' or name(.)=''annotation'' or name(.)=''description'']`
	XM2	`/site//incategory[./@category=''category52'']/parent::item/@id`
	XM3(A)	`/site//open_auction/bidder[last()]`
	XM3(B)	`/site/open_auctions/open_auction/bidder[last()]`
	XM4	`/site/regions/*/item[./location=''United States'' and ./quantity > 0 and ./payment=''Creditcard'' and ./description and ./name]`
	XM5	`/site/open_auction/open_auctions/bidder/increase`
	XM6	`/site/regions/*[name(.)=''africa'' or name(.)=''asia'']/item/description/parlist/listitem`
DBLP.xml	DB1	`/dblp/article/author`
	DB2	`/dblp//title`
	DB3	`/dblp/book[(count(./following-sibling::book/author) < count(./author))]`
treebank.xml	TB1	`/FILE/EMPTY//NP`
	TB2	`/FILE/EMPTY/S//VP[count(.//NP) > 1]`
	TB3	`/FILE/EMPTY/S/NP/NP//NN`

TABLE II

XPATH QUERIES USED FOR EXPERIMENTAL EVALUATION

Query Key	Original Serial (sec)	Data Partitioning						Query Partitioning		
		Serial (sec)	Parallel (sec)	Total (sec)	# Threads	Absolute Scaleup	Relative Scaleup	Total (sec)	# Threads	Absolute Scaleup
XM1	**494.82**	0.0003	30.82	**30.82**	7	**1.93**	**16.05**	**145.75**	3	**3.39**
XM2(a)	**6.70**	360.53	6.29	**366.82**	7	**1.05**	**0.01**	**12.52**	2	**0.53**
XM2(b)		0.0001	2.87	**2.87**	3	**2.20**	**2.33**	-	-	-
XM3(A)	**10.58**	5.29	0.55	**5.82**	4	**1.22**	**1.82**	-	-	-
XM3(B)	**5.72**	0.03	0.52	**0.55**	4	**3.55**	**10.4**	**1.49**	4	**3.83**
XM4(a)	**1.06**	0.001	0.49	**0.49**	7	**2.13**	**2.16**	-	-	-
XM4(b)		20.67	2.82	**23.49**	7	**1.31**	**0.045**	**28.17**	4	**0.04**
XM5(a)	**192.67**	0.03	0.49	**.52**	8	**3.61**	**370.51**	**7.09**	4	**27.17**
XM5(b)		217.29	1.47	**218.77**	8	**1.02**	**0.88**	-	-	-
XM6	**0.32**	0.000085	0.14	**0.14**	6	**1.14**	**2.28**	**0.088**	4	**3.63**
DB1	**1972.76**	0.29	2.05	**2.34**	4	**3.24**	**842.73**	**202.46**	4	**9.74**
DB2	**2785.58**	0.15	5.15	**5.30**	4	**3.59**	**525.58**	**1042.09**	8	**2.67**
DB3	**299.01**	0.24	77.24	**77.48**	4	**3.59**	**3.87**	**84.78**	6	**3.52**
TB1	**425.64**	0.028	0.70	**0.73**	4	**3.61**	**583.06**	**12.5**	8	**33.99**
TB2(a)	**14.75**	0.028	1.55	**1.57**	4	**3.73**	**9.39**	**2.70**	4	**5.46**
TB2(b)		3.72	1.49	**5.21**	4	**1.73**	**2.83**	-	-	-
TB3	**0.25**	0.028	0.43	**0.46**	8	**3.89**	**0.54**	**0.87**	8	**0.28**

TABLE III

SUMMARY OF PERFORMANCE EVALUATION OF DATA AND QUERY PARTITIONING ON THE XPATH QUERIES

returns only 6 nodes.). Further, the average cost of the parallel sub-query in XM2(a) is more than that in XM2(b). As a result, implementation of the query XM2 using the XM2(a) approach degrades the performance. The XM2(b) approach, on the other hand, results in an absolute scaleup of 2.20 and relative scaleup of 2.33. In the query partitioning approach, the two new queries do more work than the original query, thus leading to reduced performance (i.e., 12.52 seconds vs. 6.7 seconds for the original query).

The queries in XM3(A) and XM3(B) (Table II) are equivalent as there is only an `open_auctions` child of the `site` node, and the `open_auction` nodes are children only of the `open_auction` node. Like the query XM2, the performance of XM3(A) and XM3(B) approaches differs significantly due to the amount of time spent by their serial sub-queries (Figure 6). XM3(A) executes `//open_auction` *serially* on the entire document in 5.29 seconds, where as XM3(B) executes `/site/open_auctions/open_auction` on the same document in 0.03 seconds. Although the parallel sub-

Performance of the Data Partitioning on the Query XM2

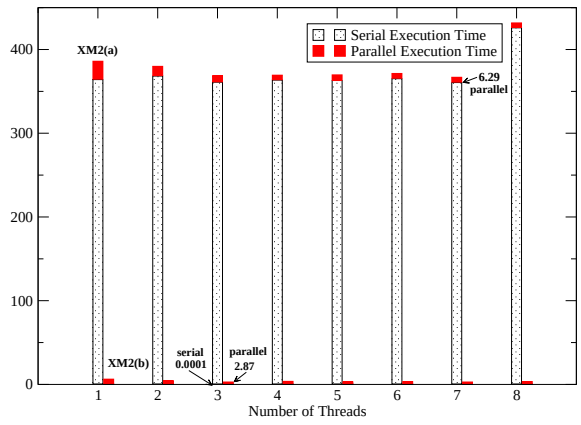

Fig. 5. Data Partitioning of the query XM2. The left bar in every cluster represents the results for the XM2(a) approach, the right bar represents the results for the XML2(b) approach. In XM2(a), the overall performance is affected by the serial execution costs.

# of Threads (Time in seconds)					
	1	2	4	6	8
	Query XM3				
Total	5.72	2.37	1.49	1.85	2.41
Scaleup	1	2.41	3.83	3.09	2.37
	Query XM5				
Total	192.67	49.96	12.83	10.02	7.09
Scaleup	1	3.85	15.01	19.22	27.17
	Query XM6				
Total	0.32	0.15	0.088	0.13	0.13
Scaleup	1	2.13	3.64	2.46	2.28

TABLE IV
PERFORMANCE OF THE QUERIES XM3, XM5, AND XM6 USING QUERY PARTITIONING.

queries in both approaches perform similarly, the XM3(B) approach provides better scaleup than XM3(A), both absolute (3.55 over 1.44) and relative (10.4 over 1.82) (Table III.

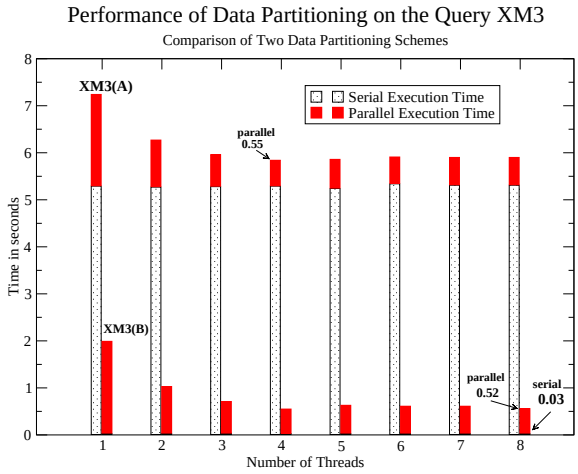

Fig. 6. Data Partitioning of the query XM3 for two different queries: XM3(A), and XM3(B). The serial execution costs of the query XM3(B) are significantly lower than that of XM3(A).

For the query XM3, the query partitioning approach distributes the work by partitioning the range of the `open_auction` nodes. If there are n `open_auction` nodes, each of the p threads would get $\frac{n}{p}$ nodes. To achieve such distribution, the original query gets rewritten to use the `position()` function as follows: `/site/open_auctions/open_auction[position()` $< \frac{n}{p} + 1$`]`, and `/site/open_auctions/open_auction [position() >` $\frac{n}{p}$ `and position()` $< 2 * \frac{n}{p} + 1$`]`. The results of each sub-query are then unioned to compute the final result. Table IV shows the performance of the query partitioning strategy when the number of threads was increased from 2 to 8. When the number of threads was 6 and 8, we used the hybrid approach with 2 and 4 virtual processor groups with 2 and 3 members, respectively. As shown in Table IV, the performance of the queries scales as the number of threads is increased.

The query XM4 (Table II) is an example of an XPath expression with conjuctive predicate. Like the previous examples, XM4 can also be parallelized in two ways using the data partitioning approach, XM4(a), and XM4(b) (Table III). XM4(a) uses `/site/regions/*` as the serial sub-query, whereas XM4(b) uses `/site/regions/*/item` as its serial sub-query. As shown in Figure 7, the execution time of the serial sub-query for XM4(b) is significantly larger than that for XM4(a). The expensive serial sub-query not only reduces the scalability in the data partitioning solutions, it also affects the performance of the parallel query. The XM4(b) approach has to make additional redundant traversals of the `item` nodes as its parallel sub-query is `self::*[./location=..]`. As shown in Figure 7, in all cases, the performance of the parallel sub-query for XM4(b) is much more than that of XM4(a). As a result, the XM4(b) causes the performance to degrade as compared to the serial case (23.49 seconds against 1.06 seconds for the serial execution). In contrast, the XM4(a) approach improves the performance by a factor of 2.

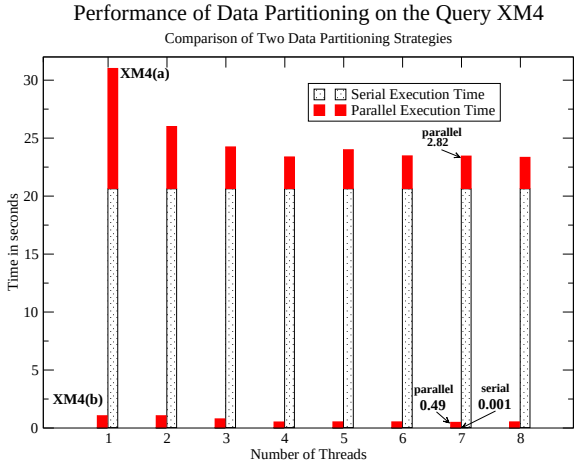

Fig. 7. Data Partitioning of the query XM4 using two different partitioning strategies. The left stacked-bar represents the performance of XM4(a), the right stacked bar represents the performance of XM4(b).

The parallelization of XM4 using the query partitioning approach involves rewriting the original query into four different sub-queries, each with a different predicate on the item nodes. These four queries are then executed in parallel on 4 threads and their results are then merge-joined to compute the final result. As Table III illustrates, this strategy does not perform as well as the XM4(a) approach. In fact, this approach slows down the query execution substantially. This is mainly due to the larger number of nodes traversed by the sub-queries, as each of them operate on the entire document.

The XM5 query (Table II) is an example of a relatively long query of parent-child traversals. Like the previous queries, XM5 can also be executed in two ways using the data partitioning approach. One approach, XM5(b) suffers from a large serial costs as its serial sub-query traverses a far larger number of nodes than the equivalent serial sub-query from XM5(a). Figure 8 presents the detailed performance comparison.

Using query partitioning, the XM5 query is parallelized by the range partitioning approach (like the query XM3). Similar

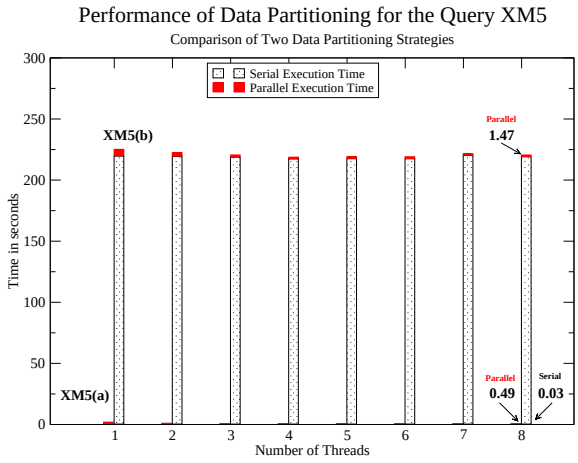

Fig. 8. Data Partitioning of the query XM5 using two different partitioning strategies.XM5(b) suffers from large serial component.

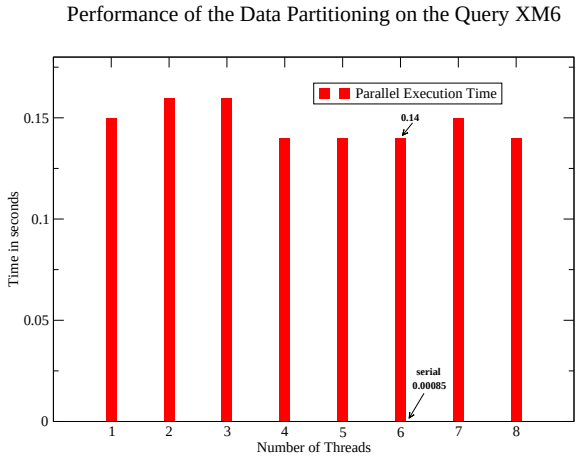

Fig. 9. Data Partitioning of the query XM6.

to query XM3, the query XM5 scales well when parallelized using query partitioning (Table IV).

The query XM6 (Table II) is interesting as it is the only query where query partitioning performs better than data partitioning (scaleup of 3.83 versus scaleup of 2.28). In the query partitioning approach, the original query is rewritten into two queries without predicates, one for `africa`, and one for `asia` (e.g., `/site/regions/africa/item/..`). Further, the traversal on the item nodes is distributed using range partitioning (i.e., we employ hybrid partitioning). Table IV presents the performance of the XM6 query using query partitioning as the number of threads is increased from 2 to 8. In the data partitioning approach, we use `/site/region` as the serial sub-query (Figure 9), partition the resultant `region` node set over a set of processors and then invoke the parallel sub-query on every local node set. However, data partitioning still performs more work than query partitioning as it still needs to evaluates the disjunctive predicate over the children of the `region` nodes.

We now discuss the performance of the XPath queries on the DBLP dataset. The DBLP document is a shallow, but

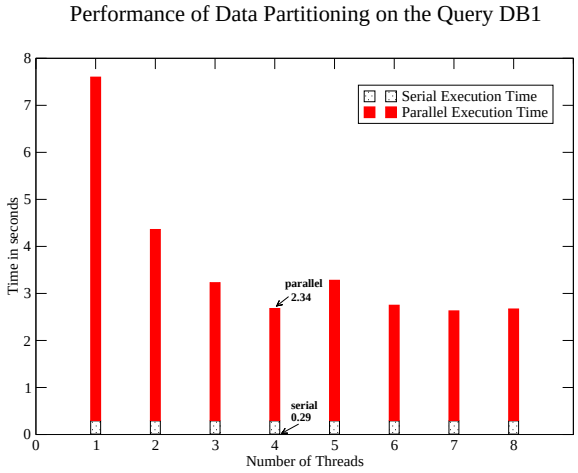

Fig. 10. Data partitioning of the query DB1. The serial execution costs are dominant. The overall performance scales up as the number of threads is increased. The maximum absolute scaleup observed is 3.24 for 4 threads.

very wide document. The first query, DB1 (Table II), finds `author` children of `article` elements. Table V presents the performance of the query when it was parallelized using the query partitioning strategy. The query partitioning strategy used range partitioning on the `article` child. Thus, every thread invoked a modified query with a positional predicate. As shown in Table V, this approach scaled up very well for 2 and 4 threads. For 6, and 8 threads, we were not able to run the query due to memory comsumption issues in the Xalan processor. In the data partitioning strategy, the original query is split into two simple queries with parent-child traversals: `/dblp/article` and `./author`. In this case, the serial component was not dominant and the parallel sub-query scaled very well (absolute scaleup of 3.24 for 4 threads) (Figure 10). The data partitioning implementation did not suffer from high memory consumption, resulting in significant relative scaleup (842.73).

# of Threads (Time in seconds)					
	1	2	4	6	8
	Query DB1				
Total	1972.76	670.94	202.46	-	-
Scaleup	1	2.94	9.74	-	-
	Query DB3				
Total	299.01	164.70	85.15	84.78	84.91
Scaleup	1	1.81	3.51	3.52	3.52

TABLE V
PERFORMANCE OF THE QUERIES DB1, AND DB3 USING QUERY PARTITIONING.

For the DB2 query (Table II), we parallelized the equivalent query, `/dblp/*/title`. In the data partitioning approach, the query was split into `/dblp/*` (serial) and `./title` (parallel) sub-queries. Performance of the modified query was not affected by the serial component, and it scaled very well (absolute scalability of 3.59) (Figure 11). In the query partitioning approach, we rewrote the original query

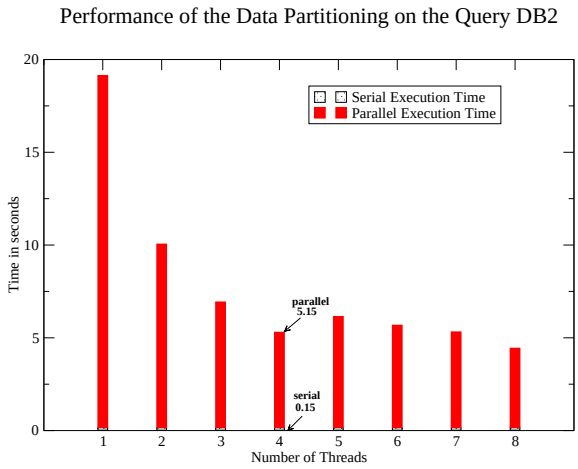

Fig. 11. Data partitioning of the query DB2. The overall performance scales up as the number of threads is increased. The maximum absolute scaleup observed is 3.49 for 4 threads.

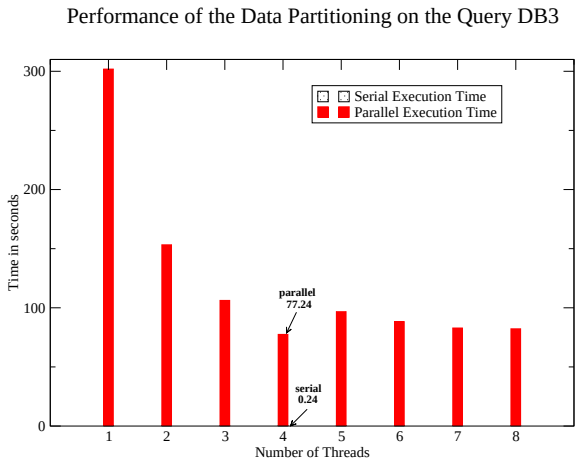

Fig. 12. Data partitioning of the query DB3. The overall performance scales up as the number of threads is increased. The maximum absolute scaleup observed is 3.89 for 4 threads.

into eight sub-queries, each for a child of the `dblp` node. Like in DB1, the query partitioning approach also suffered from excessive memory usage, and we obtained an absolute speedup of only 2.67 over the original query. In contrast, the data partitioning approach improved the performance over the original query by 525.58. The query DB3 (Table II) is an interesting query as it generates significant overlapping accesses from the `following-sibling` axis. The query partitioning strategy used the range-partitioning on the `book` element. The data partitioning strategy split the original query into `/dblp/book` and `self::*[..]` sub-queries. Both approaches scaled well with absolute speedups of 3.87 (for data partitioning) and 3.52 (for query partitioning). However, none of the approaches were able to redundant traversals caused by the `following-sibling` axis. Section V-C outlines a possible optimization to address this problem.

The PENN treebank is a relatively small (8 MB) but highly recursive document. The first query, TB1, finds all instances

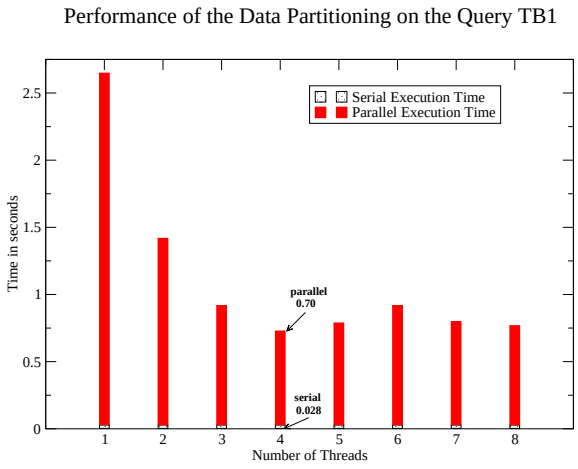

Fig. 13. Data partitioning of the query TB1. The overall performance scales up as the number of threads is increased. The maximum absolute scaleup observed is 3.61 for 4 threads.

of a recursive element `NP`. Both the query and the data partitioning approaches achieve parallelism by partitioning the traversals from the `EMPTY` elements. As both Table VI and Figure 13 illustrates both approaches scale very well, resulting in significant performance improvements (absolute scaleup of 33.99 for the query partitioning and 3.61 for data partitioning.) Due to its small size, queries on the treebank document do not suffer from memory consumption issues. The query TB2 is another example of a case where bad query splitting leads to performance degradation. As illustrated in Figure 14, the TB2(b) approach has a large serial component that reduces its scalability (1.73). In contrast, the TB2(a) approach has a smaller serial component, improving both the absolute (3.73) and relative scaleups (9.39). The final query, TB3, is an example of when not to parallelize. The original query is very efficient and does not traverse a large number of nodes. In this case, although the parallelization strategies are effective (i.e., both scale linearly (Table VI and Figure 15)), the cost of parallelization degrades the overall performance.

	Query TB1				
Total	425.644	114.86	27.97	20.83	12.52
Scaleup	1	3.70	15.21	20.43	33.99
	Query TB2				
Total	14.75	6.26	2.70	2.79	2.98
Scaleup	1	2.35	5.46	5.28	4.94
	Query TB3				
Total	0.25	0.90	0.87	1.34	1.76
Scaleup	1	0.27	0.28	0.18	0.14

TABLE VI
PERFORMANCE OF THE QUERIES TB1, TB2, AND TB3 USING QUERY PARTITIONING.

C. Discussion

As demonstrated in Section V, all three data partitioning approaches are able to scale performance of a majority of XPath queries. These results conclusively illustrate that it

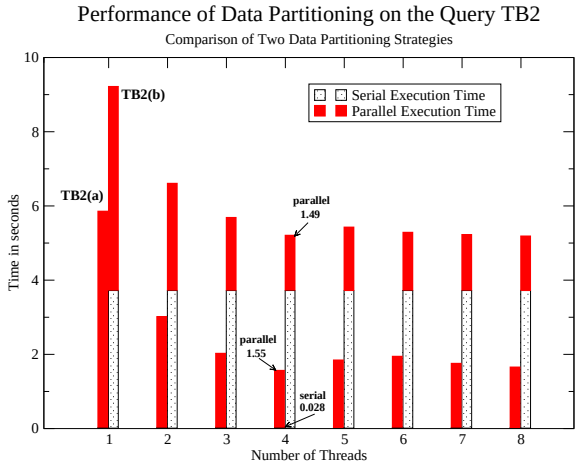

Fig. 14. Data partitioning of the query TB2. The left stacked-bar represents the performance of TB2(a), whereas the right stacked-bar illustrates the TB2(b) performance. The TB2(b) performance gets affected by the large serial costs, reducing its scalability (1.73 for 4 threads). TB2(a) results in a scaleup of 3.73 for 4 threads.

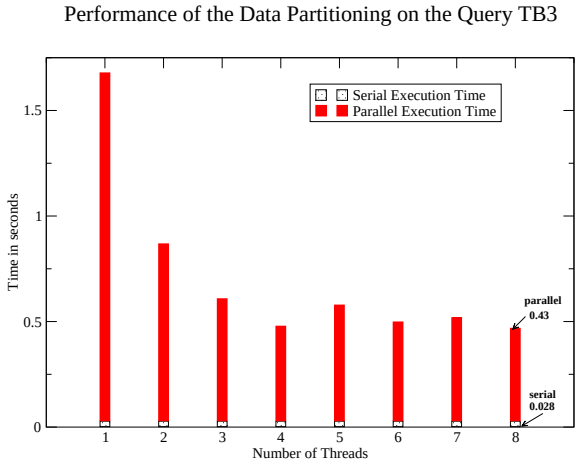

Fig. 15. Data partitioning of the query TB3. Even though the performance scales up (3.89) for 8 threads, the modified query runs slower than the original query.

is possible to accelerate XPath processing effectively using multiple cores of a multi-core processor. However, a number of key issues need to addressed before implementing a parallel XPath processors.

First, given a XPath query, it is not obvious if data or query partitioning would be beneficial. As a rule of thumb, query partitioning is suitable for queries that involve predicates or for those queries that could be rewritten using range partitioning. For queries with predicates, data partitioning often results in traversing the same node twice (once in the serial code and once by the `self::*` step in the parallel code), thus degrading the performance. In the query partitioning strategy, range partitioning needs cardinality information to be effective. Running concurrent queries on the same dataset can also increase the memory usage of the XPath processor, leading to performance degradation and even failure to execute (as we have observed in the query DB1). For a general XPath query, it is not clear how to automatically determine the best partitioning strategy. We believe a comprehensive parallel cost model could help in addressing this issue.

Second, the performance of a query parallelized using the data partitioning strategy depends on how the query was split. In our experiments, we had access to path statistics of the input documents. Without such statistics, it is not clear how one can split a query effectively.

Third, we are currently using a small number of threads for parallelizing the XPath queries. In our experiments, we are using very simple processor allocation to process different sub-queries. However, as the number of available threads increases, the processor allocation becomes very critical to achieve load balance and increase processor utilization. However, an effective processor allocation requires XML document statistics and XPath traversal information. Further, our experiments use simple block distribution to partition the input context nodeset for parallel sub-query execution. It is not yet clear if the block partitioning strategy is the most suitable strategy for the general class of XPath queries.

Fourth, we are currently rewriting the input XPath queries by hand. Auto-parallelization of XPath queries by a compiler is currently an open problem. Such a compiler would need to integrate smoothly with the existing XPath optimizations. Further, the compiler needs to address various semantic issues such as maintaining document order, executing XPath functions in the parallel environment, etc. It is also not clear yet how a parallelizing compiler would use auxiliary information such as XML path statistics and indexes.

Finally, we are using a pre-parsed XML document in an in memory configuration. In reality, XML is processed and stored in different forms, e.g., XML streams, native XML storage or relational storage. Since our parallelization models are defined over the XML data model, they can be adapted to any XML storage layout.

Our experiments have revealed several interesting query optimization problems in XPath parallelization. For example,

- Consider again the query DB3. This query traverses all `book` node siblings and for every `book` node, computes its following `book` siblings. It is immediately obvious that there are lots of overlapping accesses leading to redundant traversals of the `book` nodes. In fact, given a set of `book` siblings, the result of executing the query `following-sibling::book` on the first `book` node generates the answers of executing the same query on all other `book` nodes. We can extend this idea in the parallel domain: First, partition the `book` nodes using the data partitioning strategy in the document order. In every partition, for every first `book` element select the `following-sibling` nodes that lie in its *local* nodeset. Then, store the local results into a shared data structure to generate the final result. This strategy avoids redundant traversals and accelerates the most expensive traversal by executing it in the parallel fashion. This approach, however, consumes significant amount of memory. It is an interesting problem to balance parallelism,

shared resources, and memory consumption.

- Consider the problem of parallelizing the following DBLP query: `(//book)[10]` (i.e., find the tenth book from the document in the global document order). Consider the execution of this query using the data partitioning strategy. Assume that there is a shared list that stores the `book` elements in a sorted order using their document numbering. As soon as a processor stores the tenth book in the shared sorted list, it can inform the other processors and prevent redundant traversals of the XML document.

VI. Conclusions

In this paper, we evaluated the problem of parallelizing XPath processing using commodity multi-core processors. We examine three novel parallelization schemes and evaluated them on a set of realistic documents using a production-grade XPath Processor (Xalan). Our experimental results demonstrate that our proposed strategies work very well in practice. In most cases, we were to be to improve the query performance significantly. Our experiments also releaved various optimization and infrastructure issues that need to be addressed for implementing a scalable parallel XPath processor. We plan to explore these issues in detail in our future work.

References

[1] World Wide Web Consortium, "XML Path Language (XPath) 2.0, W3C Recommendation, 23 January 2007," www.w3.org.

[2] ——, "XQuery 1.0 and XPath 2.0 Data Model (XDM), W3C Recommendation, 23 January 2007," www.w3.org.

[3] ——, "XQuery 1.0: An XML Query Language, W3C Recommendation, 23 January 2007," www.w3.org.

[4] ——, "XSL Transformations (XSLT) 2.0, W3C Recommendation, 23 January 2007," www.w3.org.

[5] "SQL/XML Standard," www.sqlx.org.

[6] "Web Services Activities at W3C," www.w3.org/2002/ws.

[7] "XML Schema Activities at W3C," http://www.w3.org/XML/Schema.

[8] G. Gottlob, C. Koch, and R. Pichler, "Efficient Algorithms for Processing XPath Queries," *ACM Transactions on Database Systems*, vol. 30, no. 2, pp. 444–491, June 2005.

[9] M. F. Khan, R. A. Paul, I. Ahmad, and A. Ghafoor, "Intensive data management in parallel systems: A survey," *Distributed and Parallel Databases*, vol. 7, no. 4, pp. 383–414, October 1999.

[10] D. Kossmann, "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, 2000.

[11] H. Lu, *Query Processing in Parallel Relational Database Systems*, K. L. Tan and B.-C. Ooi, Eds. IEEE Computer Society Press, 1994.

[12] P. Buneman, G. Cong, W. Fan, and A. Kementsietsidis, "Using Partial Evaluation in Distributed Query Evaluation," in *VLDB*, 2006, pp. 211–222.

[13] G. Cong, W. Fan, and A. Kementsietsidis, "Distributed Query Evaluation with Performance Guarantees," in *SIGMOD Conference*, 2007, pp. 509–520.

[14] D. Suciu, "Distributed Query Evaluation on Semistructured Data," *ACM Trans. Database Syst.*, vol. 27, no. 1, pp. 1–62, 2002.

[15] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[16] W. Lu and D. Gannon, "Parallel XML Processing by Work Stealing," in *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, 2007, pp. 31–38.

[17] Y. Pan, W. Lu, Y. Zhang, and K. Chiu, "A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs," in *CCGRID*, 2007, pp. 351–362.

[18] W. Lu, K. Chiu, and Y. Pan, "A Parallel Approach to XML Parsing," in *Grid 2006: The 7th IEEE/ACM International Conference on Grid Computing*, 2006, pp. 28–29.

[19] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, M. J. Carey, I. Manolescu, and R. Busse, "Why and How to Benchmark XML Databases," *ACM SIGMOD Record*, vol. 3, no. 30, September 2001.

[20] "DBLP XML Dataset," http://dblp.uni-trier.de/xml.

[21] "The PENN Treebank Project," http://www.cis.upenn.edu/ treebank.