View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AIS Electronic Library (AISeL)



# Low-latency XPath Query Evaluation on Multi-Core Processors

Ben Karsin
University of Hawai'i at Manoa
karsin@hawaii.edu

Henri Casanova
University of Hawai'i at Manoa
henric@hawaii.edu

Lipyeow Lim
University of Hawai'i at Manoa
lipyeow@hawaii.edu

## Abstract

*XML and the XPath querying language have become ubiquitous data and querying standards used in many industrial settings and across the World-Wide Web. The high latency of XPath queries over large XML databases remains a problem for many applications. While this latency could be reduced by parallel execution, issues such as work partitioning, memory contention, and load imbalance may diminish the benefits of parallelization. We propose three parallel XPath query engines: Static Work Partitioning, Work Queue, and Producer-Consumer-Hybrid. All three engines attempt to solve the issue of load imbalance while minimizing sequential execution time and overhead. We analyze their performance on sets of synthetic and real-world datasets. Results obtained on two multi-core platforms show that while load-balancing is easily achieved for most synthetic datasets, real-world datasets prove more challenging. Nevertheless, our Producer-Consumer-Hybrid query engine achieves good results across the board (speedup up to 6.31 on an 8-core platform).*

## 1. Introduction

The increasing number of processing cores on modern commodity multi-core systems represents an opportunity for reducing the latency of XPath query processing. Most state-of-the-art XPath processing libraries, such as Apache Xalan [18], leverage multi-core architectures through the concurrent execution of multiple XPath queries: multiple threads each evaluating a different XPath query simultaneously. Although the overall throughput is increased, there is no improvement in query latency because each query is executed sequentially. Concurrent evaluation of multiple XPath queries can actually *increase* individual query latencies due to sharing of hardware resources (caches, memory bus). Previous work on parallel evaluation of XPath queries on multi-core architectures have used a static partitioning of the query evaluation task and a fixed assignment of partitions to cores in order to achieve significant speedups on well-behaved datasets and queries [4, 3]. However, such static approaches are unlikely to work well when the partitions are unbalanced in the amount of work they contain.

In this work we study the parallelization of XPath query evaluation with the goal of reducing the latency of a single XPath query over an in-memory XML Document Object Model (DOM) tree on multi-core architectures. This work focuses on single-query latency because, for web-based applications, this latency can severely degrade performance for end-users. Parallelization is achieved by partitioning the query evaluation into *work units*, i.e., XML tree nodes that can be evaluated independently

We investigate three strategies: (i) Static Work Partition (SWP), (ii) Work Queue (WQ), and (iii) Producer-Consumer-Hybrid (PCH). SWP is a straightforward approach used in previous work [4, 3] in which work units are assigned to threads statically before the onset of concurrent computation. Because it uses static work partitioning, SWP can lead to poor load balance if query processing is more computationally intensive for some sub-trees than some others. With WQ, work units are placed in a thread-safe shared workqueue. Threads retrieve work units from the workqueue and evaluate them dynamically, which improves load balance when compared to SWP but comes with additional overhead for ensuring thread-safety of the workqueue. Both SWP and WQ correspond to well-known parallelization strategies that have been used in countless parallelization contexts. Another such well-known strategy is the producer consumer approach in which threads can either produce work units into or consume work units from a shared workqueue. We generalize the producer consumer approach and develop the *PCH* strategy in which can also be a "hybrid": it can play the role of both consumer and producer. Given $n$ threads, PCH allows any combination of $p$ producers, $c$ consumers, and $h$ hybrids, where $p+c+h=n$, thus widening the design space compared to the standard producer consumer approach. This paper presents the PCH strategy and an empirical study of all three SWP, WQ, and PCH strategies, making the following contributions:

- We implement a custom sequential query engine and

URI: http://hdl.handle.net/10125/41916
ISBN: 978-0-9981331-0-2
CC-BY-NC-ND

HICSS

an accompanying performance model that serve as bases for developing parallel query engines and assessing their performance.
- We implement query engines that parallelize the execution of a single XPath query on multi-core platforms using the SWP, WQ, and PCH strategies. While SWP and WQ are standard approaches, PCH is a novel generalization of the producer consumer model.
- We evaluate our query engines with synthetic benchmarks, and a real-world XML dataset (DBLP).
- We find that all of our parallel query engines can achieve some speedup, with PCH achieving near-linear speedup for most synthetic datasets and queries, and significant speedup for most queries on real-world and benchmark datasets. The good performance of PCH is explained by the use of hybrid threads that can be used to improve upon the traditional producer consumer approach.
- We find that some queries over the DBLP dataset are particularly difficult to parallelize, and we identify the root causes of poor parallel performance. Some of these causes can be addressed by modifying our parallelization approach. Others are inherent to querying XML DOM trees and would require modifying the document order to achieve good parallel speedup.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes our baseline sequential XPath query engine and a simple performance model. Section 4 details our three parallel XPath query engines. Their performance is evaluated on synthetic and real-world datasets in Section 5. Finally, Section 6 summarizes our results and highlights future work directions.

## 2. Related Work

The field of XML processing is prolific, with many studies focusing on both analytic and empirical aspects. Several areas have been studied extensively and are related but often orthogonal to our work. The problem of XML cardinality and selectivity estimation is one such area [2, 15, 5, 17, 11, 12]. Incorporating estimation techniques in our methods could yield performance improvements, but it is beyond the scope of this paper. Another area of XML processing that has been extensively studied is that of embedded XML processing and hybrid XML/relational systems. This area has been a hotbed in recent years (see several broad surveys [7, 8, 14, 1]). The recent focus has been on comparing and integrating native XML query processing with embedded XML in relational systems. The work in [7] compares these two approaches (and integration), applied to the problem of twig pattern matching. Our work could also be applied to pattern matching, but in this paper we only attempt to leverage multi-core hardware for native XML query processing. The work in [8] focuses on XED (embedded databases) versus NED (native databases) and relevant query optimization techniques. Moro et al. [14] provide a broader overview of the entire field, with discussions on many topics including query processing, views, and schema evolution. Schwentick [1] provides a more analytic study, focusing on automata capable of processing XML trees efficiently. While some of the ideas pertaining to tree processing are relevant for our work, this paper provides a more empirical study of parallel XPath query execution.

Several authors have studied multi-query processing on XML databases in recent years [16, 19, 6]. These studies aim to improve query throughput on XML databases by leveraging parallel hardware architectures. Wang et al. [16] utilize a range of methods to increase throughput when evaluating multiple queries over a compressed XML dataset. The work in [19] incorporates large numbers of queries into a single NFA (Non-deterministic Finite Automata), which is then fragmented to enable parallelization. Results show significant throughput improvement. Our work differs because we focus on reducing the latency of a single query. Cong et al. [6] aim to increase query throughput as well as reduce query latency. The proposed approach consists in decomposing XPath queries into several smaller sub-queries for parallel execution and then merge query results. The authors describe parallelization methods for XPath queries on both cluster and multi-core platforms. They obtain significant speedup using all methods, though there are concerns about redundant processing of query nodes. Query decomposition results in several smaller sub-queries, causing the possibility of duplicate processing on portions of each sub-query. Additionally, the process of merging results of sub-queries can be computationally costly, especially if there are many query matches. Our approach avoids these problems by implementing intra-query parallelism through distribution of XML document tree nodes to processor cores. The work in [13] improves the performance of XML serialization through parallel processing of XML DOM structures. The authors use *work-stealing* to balance work among threads, with region-based query partitioning to improve scalability. While their approach successfully achieves parallel speedup, it is limited to XML serialization. Furthermore, results are only provided for a single 4-core platform and for one relatively small (25MB) dataset. Our work aims at parallelization of general XPath queries on XML DOM tree structures, with results presented for 4- and 8-core platforms on a range of synthetic and real-world datasets (up to 1GB in

size).

Some of the work in this paper builds on [3] and [4]. The work in [3] gives an overview of the problem of parallelization of XPath queries and presents preliminary experimental results obtained with the Xalan query engine. The work in [4] compares XPath parallelization results using three static work partitioning techniques (data, query, and hybrid partitioning), with arguably limited success. In this work we extend the work in [3, 4] by developing and evaluating parallel query engines that utilize more sophisticated and dynamic work partitioning approaches to achieve drastically more efficient parallelization.

| | Xalan (sec.) | | | Custom (sec.) | | |
|---|---|---|---|---|---|---|
| Tag / Context | 2 | 4 | 6 | 2 | 4 | 6 |
| 1 | 54.25 | 0.16 | 28.77 | 0.26 | 0.17 | 0.17 |
| 2 | 65.54 | 0.16 | 26.76 | 0.28 | 0.19 | 0.17 |
| 4 | 65.41 | 0.16 | 26.95 | 0.30 | 0.20 | 0.18 |
| 8 | 64.90 | 0.29 | 29.19 | 0.35 | 0.23 | 0.20 |
| 16 | 55.02 | 0.36 | 28.01 | 0.44 | 0.29 | 0.25 |
| 32 | 66.05 | 0.25 | 28.71 | 0.58 | 0.39 | 0.33 |
| 64 | 57.99 | 0.28 | 27.02 | 0.85 | 0.57 | 0.49 |
| 128 | 66.45 | 0.27 | 29.49 | 1.34 | 0.89 | 0.77 |
| 256 | 59.66 | 0.36 | 28.28 | 2.37 | 1.58 | 1.35 |
| 512 | 57.25 | 0.32 | 27.04 | 4.23 | 2.82 | 2.42 |
| 1024 | 54.97 | 0.31 | 27.77 | 7.61 | 5.07 | 4.35 |

**Table 1: Custom vs. Xalan parallel runtimes using an all-match query on a 20 level deep binary tree with 4 threads on Greenwolf. Varying context depth has a large and unpredictable impact on Xalan performance, while tag length does not.**

# 3. Sequential XPath Query Engine

## 3.1. Custom Query Engine

Some XPath query engines, such as Apache Xalan, utilize additional indexing data structures to support more efficient navigation of the XML DOM tree for certain types of traversals. While this provides improved performance for some queries, it creates unpredictability when analyzing the overall performance of the query engines especially in a parallel context. Our initial experiments with the parallelization of Xalan produced inconsistent results. Table 1 shows parallel query evaluation times in seconds, averaged over 100 trials using a binary XML tree and a query resulting in every leaf node being a match node. These results are obtained on Greenwolf, the 4-core platform described in Table 2. As explained in more detail in later sections, a simple approach for parallelizing an XPath query is to first use a sequential phase that traverses the XML tree down to some predefined depth, which we call the context depth. This traversal leads to $N$ sub-match nodes at that depth, and these nodes can then be evaluated in parallel by $P$ threads on $P$ cores, each thread evaluating $N/P$ nodes. The context depth is thus a key driver of parallel query evaluation time. Another such driver is the tag length, since longer tags imply more time consuming tag comparisons. The left-hand side of Table 1 show results obtained with Xalan as the basis for the simple parallelization described above. We see that with a context depth of 2 or 6 the query time is very large above 50 sec or 25 sec, respectively, even though it is below 1 sec for a context depth of 4. While one may expect that there would be an optimal context depth, such drastic differences are difficult to explain. Furthermore, the query time is not necessarily monotonically increasing with the tag length. Other results not presented here show that the query times are not always reproducible. We conclude that Xalan likely utilizes indexing data structures that have complex effects on the query time. As a result, Xalan is not a viable target for this work.

Since our goal is to study the parallelization of XPath queries on an in-memory DOM tree, and given the results obtained with Xalan, we opt to develop our own sequential query engine to form the basis for our parallel query engines. Our custom query engine recursively traverses the XML document tree, comparing the XML tag of each traversed node with the XPath query string of the corresponding depth. If the comparison succeeds, the nodes' children are evaluated recursively (recall that such a node is called a sub-match node). When a node is found with a tag matching the final string of the XPath query at the query depth, it is saved as a match node. The right-hand side of Table 1 shows results obtained for the same parallel query engine as above but using our custom sequential query engine as a basis instead of Xalan. The results show stable trends, with increasing query times as the tag length increase, and comparable results across different context depths.

## 3.2. Performance Model

Given an XML tree and an XPath query, our goal is to derive a closed form formula to estimate the performance of our custom sequential query engine. We achieve this by averaging attributes of individual nodes over the entire XML tree and measuring single-node computation times on the given hardware. The result is a performance estimate based on the XML tree, XPath query, and hardware environment. However, due to limited space, we provide only a brief overview of our sequential performance model in this paper. We direct interested readers to [9] for a full derivation and analysis. In short, the performance model relies on the following

parameters:

- Tree Height ($H$): The maximum depth of the XML tree;
- Branch Factor ($B$): The number of children per non-leaf node;
- Tag Length ($L$): The number of characters per query tag;
- Query Depth ($D \leq H$): The maximum depth of the query; and
- Selectivity ($S$): The number of (sub-)match children per non-leaf node.
- $\tau$: The time to compare a single character of a node tag to a single character of a query tag, so that comparing two tags of $L$ characters takes at most $\tau \times L$ seconds; and
- $\alpha$: The time to perform all processing of a single node besides tag comparison (e.g., pointer chasing).

Using these parameters and counting the *average* number of nodes evaluated, we obtain the following formula for our sequential performance model:

$$T_{seq} \quad = \quad [\frac{B(S^{D-1}-1)}{S-1}+1] \times (\tau \times L + \alpha).$$

To enable accurate performance estimation, we execute several benchmarks to measure $\alpha$ and $\tau$ given a particular hardware platform. More specifically, we consider two platforms, Greenwolf and DiRT, described in Table 2 and used in dedicated mode for all experiments. By measuring query execution time on each environment for a range of synthetic data sets, we empirically measure $\tau$ and $\alpha$. To conserve space, we omit the details and direct the reader to [9] for full details. With empirical values for $\tau$ and $\alpha$, we are able to estimate the execution time of any query using the parameters described above.

| Env. Name | Cores | Proc. Type |
|---|---|---|
| Greenwolf | 4 | Intel Core i7 2.67 GHz |
| DiRT | 8 | Intel Xeon 2.40 GHz |

**Table 2: Overview of the 2 hardware environments used for experimentation.**

## 3.3. Performance Model Validation

In this section we validate both our query engine implementation and our instantiated performance model by comparing actual query processing times to analytical estimates. We perform experiments for a set of synthetic datasets (XML trees and queries), i.e., for various values of the $D$, $B$, $L$, and $S$ parameters. We define a notation for our synthetic datasets based on these parameters. Each dataset comprises an XML tree and a query for that tree, and the query depth ($D$) is equal to the tree depth ($H$). We use a D$w$ B$x$ L$y$ S$z$ notation to describe the query/tree depth, the branching factor, the tag length, and the selectivity for a given dataset. For example, D20 B2 L16 S1 denotes a 20-level deep binary tree with 16-character tags and a 20-level deep query that matches 1 child at each tree node. All generated trees are complete, and (sub-)match children are selected among a non-leaf node's children using a uniform probability distribution. Each run of a query engine for an experimental scenario is repeated 100 times. The average is reported and error bars are displayed on all graphs (often they are so small that they cannot be seen).

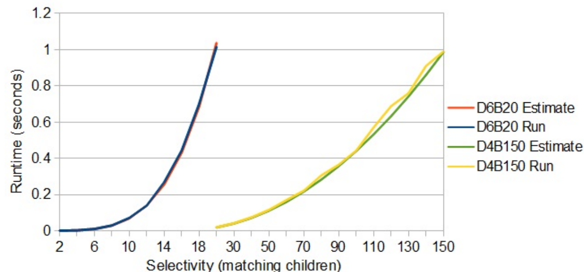



**Figure 1: Average query runtime vs. $S$ (Selectivity) on Greenwolf and DiRT. Results shown for D6 B20 L1 and D4 B150 L1 synthetic experiments. All initial results indicate that our custom query engine performs closely in-line with our performance estimates.**

Figure 1 shows the results of one set of experiments and the corresponding performance estimates on DiRT. Due to lack of space, we omit further results of the other experiments, though we find that estimates are relatively accurate (relative error rates under %10) with respect to actual query evaluation times and they exhibit the same trends. For this particular experiment, we obtained an average error of %3.5. We conclude that our custom query engine performs as expected, that our performance model can be used to estimate query evaluation times, and that our query engine can serve as a viable baseline for developing parallel query engines.

# 4. Parallel XPath Query Engines

In this section, we introduce our three parallel XPath query engines: Static Work Partitioning (SWP), Work Queue (WQ), and Producer-Consumer-Hybrid (PCH). SWP and WQ are based on the same two-phase approached described below, but they differ in how they

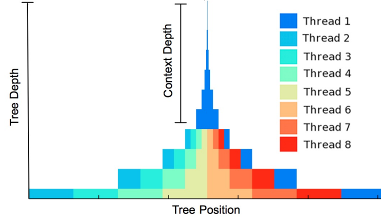



**Figure 2: Example allocation of work distributed among 8 threads for each level of a 1024-node XML tree using SWP.**

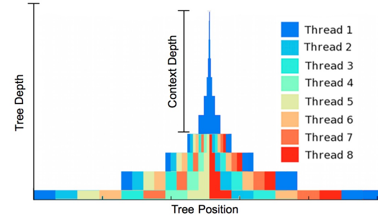



**Figure 3: Example allocation of work distributed among 8 threads for each level of a 1024-node XML tree using WQ.**

implement the second phase leading to various trade-offs between load-balance and overhead. All three parallel engines use the custom sequential query engine described in Section 3.1 as a basis.

A simple approach for parallelizing our sequential query engine is to use two phases:

- Sequential Phase – Execution of the query to depth $C$, the *context depth*, generating a set of sub-match nodes. These nodes, which we call the *context nodes*, can be evaluated independently.
- Parallel Phase – Parallel evaluation of the context nodes by $N$ threads on $N$ cores.

## 4.1. Static Work Partitioning (SWP)

SWP uses a static work distribution. After the sequential phase, the context nodes are partitioned into $N$ "blocks" as evenly as possible, and each block is assigned to a thread for evaluation. Figure 2 shows an example work distribution across eight cores using SWP on a small 1024-node synthetic dataset (D11 B2 L1 S2, see Section 3.3 for details on our synthetic datasets). This figure is generated from an actual query execution on an 8-core platform during which each node was labelled by the index of the thread that evaluated it. These nodes are displayed in the figure based on their location in the document tree, and the top $C = 6$ levels correspond to the context nodes.

SWP is designed to minimize runtime overhead, and it can achieve good load-balancing, as seen for instance in Figure 2. However, several factors can degrade its performance. First, the sequential context computation limits overall parallel speedup (Amdahl's law). Second, if the context nodes are not evenly divisible by the number of processor cores, some cores will inevitably have more work. Third, if some context nodes are more computationally intensive than others (e.g., they have more children, they have more sub-match children), load balance will be poor and so will be the parallel speedup. For these reasons, we view SWP as a baseline approach that is simple, low-overhead (no shared state among threads and thus no synchronization necessary), but that is not expected to achieve high speedup in practice.

## 4.2. Work Queue (WQ)

WQ employs a simple workqueue to attempt to avoid load imbalance and improve parallel performance. The goal is to distribute work among threads more evenly by dynamically assigning work to idle threads. Once the context depth $C$ is reached through sequential execution, the context nodes are divided into $W$ *work units* and placed in a *shared*, i.e., thread-safe, workqueue data structure. The $N$ threads then begin parallel execution, each reading a work unit from the queue, processing it, and returning to the workqueue for more work. Once the workqueue is empty and all threads finish, the query is complete. The use of a shared thread-safe workqueue increases overhead. Figure 3 is similar to Figure 2 and shows an example work allocation across 8 threads using WQ with $W = 16$ for the same small synthetic dataset.

Parameters $W$ and $C$ determine the size and the number of the work units read from the workqueue. Through empirical testing, we determine that good values of $C$ and $W$ can be easily found. We find that we can achieve good performance with small values of $C$ and $W$, but we omit the details due to limited space and direct interested readers to [9]. For all further experiments with SWP and WQ, we use $C = 3$ and $W = 16$.

## 4.3. Producer-Consumer-Hybrid (PCH)

PCH extends the well-known producer-consumer model to evaluate XPath queries in parallel using a novel parallel execution model. Unlike SWP and WQ, PCH does not require a sequential phase to compute context nodes. Instead, parallel execution can begin as early as possible (i.e., once the root node has been processed). By Amdahl's law we know that reducing the amount of sequential computation even by a small amount can lead to large improvements in speedup. We define three types

of threads that share a thread-safe workqueue of work units and can be mixed to achieve various trade-offs between load imbalance and overhead. An overview of these thread types and their corresponding activities is:

- Producer – given one tree node, evaluates all children of this node and writes those children that are sub-match nodes to the shared workqueue;
- Consumer – reads a tree node from the shared workqueue, recursively evaluates all its children nodes and add match nodes to the query results, repeats; and
- Hybrid – reads a tree node from the shared workqueue, recursively evaluates all its children nodes, recursing only up to $R_{depth}$ times, writes sub-match nodes back to the shared workqueue and adds match nodes to the query results, repeats.

The next three sections give details on the algorithms used by each type of thread above as well as relevant implementation details.

**Producer Threads.** Producer threads are designed to recurse only once given a tree node and write sub-match nodes to the workqueue so that these nodes can be processed by Consumer or Hybrid threads. Consequently, producer threads never read from the workqueue. One difficulty with this approach is that it is very likely that a producer threads would become idle early in the query evaluation process. To address this problem, producer threads do not write back to the workqueue all of the sub-match nodes they have identified. Instead, they keep a small number of sub-match nodes for themselves. We use $N_{keep}$ to denote this number of sub-match nodes. These kept nodes are used to continue execution once all nodes in the set of nodes initially assigned to the producer have been evaluated. We find that any small value (between 2 and 10) for $N_{keep}$ results in good performance. Due to limited space, we omit the details of this experimentation and direct interested readers to [9]. We utilize $N_{keep} = 5$ for further experimentation.

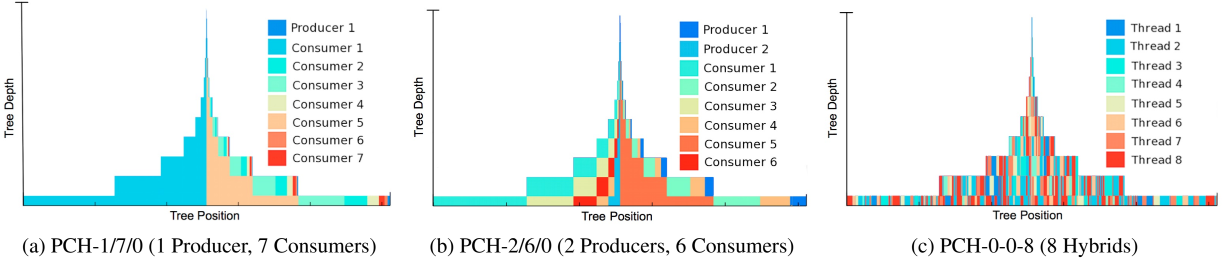
Our implementation assumes assumes that a "list of nodes" abstract data type is available, with usual *removeFirst*, *removeFirstN*, and *append* operations to remove and return the first node from a list, remove the first $N$ nodes of a list and return them as a list, and append a node or a list to the end of a list. All operations on the shared workqueue are enforced to be atomic by using a single lock. All our implementations use a spinlock so as to reduce locking overhead.

**Consumer Threads.** Like producers, consumer threads have limited interaction with the shared workqueue. While producers never *read* from the workqueue, consumers threads never *write* to the queue. They perform a standard tree traversal for whatever node(s) they are given. Consequently, load-balancing is achieved solely by the use of producer and/or hybrid threads. For instance, if a consumer thread happens to be given the root node of the document tree, it will evaluate the entire query itself and all other threads will remain idle. To prevent this worst-case scenario from occurring, our implementation insures that all producer and hybrid threads receive work, i.e., nodes to evaluate, before any consumer thread reads from the workqueue.

**Hybrid Threads.** Hybrid threads incorporate features from both consumer, i.e., *reading* from the shared queue and *recursing*, and producers, i.e., *writing* sub-match nodes to the shared queue. The $R_{depth}$ parameter controls the number of times hybrid threads will recurse before writing to the shared queue. If $R_{depth} = 0$, hybrids are producers, although they read when they run out of work to do. If $R_{depth} = D$, hybrids are consumers and never write back to the workqueue. Through experimentation we find that performance is not greatly impacted by $R_{depth}$, as long as it is close to $D/2$. Due to limited space, we omit the details of these experiments and utilize $D/2$ for further experimentation with PCH.

**PCH Thread Mix.** Due to differences between each thread type and their potentially complex interactions, PCH corresponds to a *class* of query engines. We use the PCH-$p$/$c$/$h$ notation to denote an instance of PCH with $p$ producer threads, $c$ consumer threads, and $h$ hybrid threads (e.g., PCH-3/4/1 would be three producers, four consumers, and one hybrid). A challenge with PCH is determining the best mix of producer, consumer, and hybrid threads. As might be expected, the best mix depends on the dataset, the query, and the hardware.

Figure 4 shows how work is distributed by three different PCH instances on a small 1024-node synthetic dataset on an 8-core platform. Not surprisingly, the numbers of producer, consumer, and hybrid threads greatly affects work distribution. In Figure 4a we see that load-balancing is poor since only 1 producer thread (in this case configured with $N_{keep} = 1$) is available. The first node placed on the work queue requires a large amount of of work, causing the first consumer thread to be assigned much more work than the other consumers. Adding a second producer thread, shown in Figure 4b, greatly improves load-balancing by further partitioning the work. By adding the second producer thread, we increase the maximum possible speedup from 2 to 4. Figure 4c is the most extreme case of work partitioning, with all 8 threads as hybrids ($R_{depth} = 0$ in this example). We see that the entire tree is segmented and threads all have seemingly random partitions. While such an ex-




(a) PCH-1/7/0 (1 Producer, 7 Consumers) (b) PCH-2/6/0 (2 Producers, 6 Consumers) (c) PCH-0-0-8 (8 Hybrids)

**Figure 4: Example allocation of work to 8 threads for each level of a 1024-node XML tree when using PCH on an 8-core platform (results obtained by executing a query in which all leaf nodes are match nodes).**

ecution provides good load balance, it comes at the price of high overhead due to constant thread interactions with the shared workqueue.

# 5. Parallel Query Engine Evaluation

## 5.1. Performance Bounds

In Section 3.2 we presented a performance model that accurately estimates the average execution time of a given query on a given hardware platform. It uses a series of data-, query-, and hardware-specific parameters and was seen to accurately estimate the performance of our sequential query engine on a series of synthetic experiments. Using our sequential performance model as a basis, we can define a simple estimate of a lower bound on parallel query execution time, $T_{bound}$, as:

$$T_{bound} = \frac{T_{seq}}{P}$$

where $T_{seq}$ is the sequential performance estimate from Section 3.2, and $P$ is the number of threads/cores. This lower bound ignores all parallelization overhead and assumes a perfect parallel speedup of $P$ when using $P$ cores. $T_{bounds}$ is based on $T_{seq}$, and $T_{seq}$ is only an estimate of sequential query evaluation time. Therefore, $T_{bound}$ is not a lower bound in the theoretical sense of the term. Nevertheless, our results show that $T_{bound}$ is meaningful to assess the absolute performance of our parallel query engines.

## 5.2. Validation Experiments

To initially measure the performance of our parallel query engines, we perform a range of synthetic experiments varying each of our query- and data-specific parameters (defined in Section 3.3). We measure the speedup of each parallel query engine and compare results with our parallel lower-bounds. Note that the SWP and WQ engines cannot achieve perfect parallel

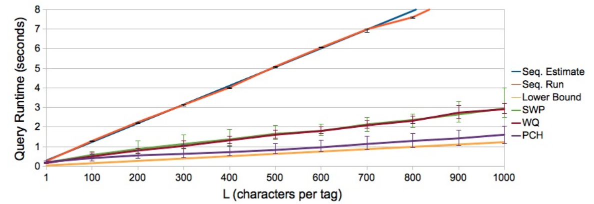



**Figure 5: Average query runtime on DiRT vs. $L$ (tag length). Results shown for the D4 B100 S100 synthetic experiments. All query engines achieve good parallel speedup, with the best parallel performance obtained using PCH (speedup of 5.92 using PCH-2/1/5).**

speedup, as they have a sequential computation phase. The PCH query engine does not use a sequential phase, and thus could conceivably achieve linear speedup.

Figure 5 shows one among many sets of (similar) results for a particular synthetic dataset and query (D4 B100 S100) for varying tag length ($L$) on DiRT. As expected all curves grow roughly linearly as $L$ increases. The results of sequential execution indicate that our query engine exhibits predictable results, with an average deviation from the sequential query time estimate of 3.1% for this experiment. Our three parallel query engines achieve good parallel speedup, with a maximum value for SWP, WQ, and PCH of 4.10, 4.15, and 5.86, respectively, out of an absolute maximum of 8. We use PCH-2/1/5 for these experiments, which we found to perform well for synthetic datasets. Similar results are obtained for other experiments on both Greenwolf and DiRT. In all of these experiments PCH achieves significantly better performance than SWP and WQ, close to the lower bound estimate.

## 5.3. Query engine parameters

Each of our parallel query engines has a distinct set of parameters that define its behavior. In this section we present results from experiments used to measure the

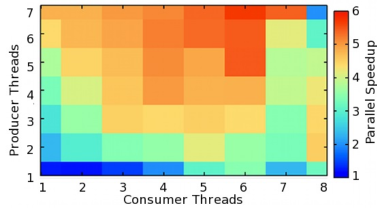



**Figure 6: Speedup achieved by PCH for every $p/c/h$ combination for D13 B4 synthetic test on DiRT.**

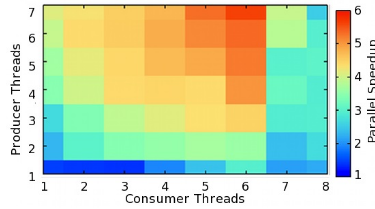



**Figure 7: Speedup achieved by PCH for every $p/c/h$ combination for D24 B2 synthetic test on DiRT.**

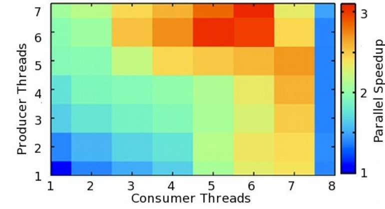



**Figure 8: Speedup achieved by PCH for $Q2_{dblp}$ on the DBLP dataset using varying $p/c/h$ combinations on DiRT.**

impact of these parameters, making it possible to determine good parameter values for each query engine for a given dataset/query on a given hardware platform. Due to limited space, we omit the details of several of these parameters and direct interested readers to [9] for full details.

We evaluate the performance of different PCH-$p/c/h$ instances via a series of experiments with synthetic datasets. In each experiment we measure average query execution time using every possible $p/c/h$ combination, from 1/1/0 to 0/0/$N$, where $N$ is the number of threads. We count a hybrid thread as both a producer and consumer, so that PCH-$N/N$/0 is actually PCH-0/0/$N$ (i.e., all threads are both producers and consumers). Figures 6 and 7 show the parallel speedup achieved for each P/C/H combination on DiRT for two different synthetic experiments. Each experiment has similar tag length and selectivity and corresponds to a document tree with a similar number of nodes but with different depth and width (D13 B4 and D24 B23). In this figure, all data points above the anti-diagonal correspond to cases in which at least one hybrid thread is used (with the upper-right corner corresponding to an all-hybrid query engine).

We see from Figures 6 and 7 that significant parallel speedup is achieved with a range of combinations. The best parameter configuration is PCH-2/1/5 for both of these tests. The maximum speedup achieved in Figures 6 and 7 are 5.69 and 5.62, respectively. In all of our synthetic experiments, PCH-2/1/5 is never more than $\%10$ slower than the fastest configuration and performs better than other configurations in most cases. We see similar results with the 4-core Greenwolf environment, with a maximum speedup using PCH-1/0/3 of 3.06 and 3.13, respectively.

The results, overall, indicate that the load-balancing benefits of hybrid threads lead to performance improvements when compared to a standard producer-consumer approach. However, the all-hybrid configuration exhibits poorer performance, indicating that over-utilizing hybrid threads makes the shared workqueue a bottleneck. Note that these experiments are for complete and balanced trees and queries. We expect that the best PCH-$p/c/h$ combination may differ for real-world datasets.

## 5.4. Experiments on real-world dataset

The DBLP [10] dataset is a 900MB XML file with nearly 25 million lines. The structure of the XML file, when parsed and loaded into memory, results in a very shallow ($H = 2$) tree with widely varying branch factor. The root node (depth 0) has approximately $B = 2.7$ million children, yet at the subsequent level the branch factor is much lower (ranging from $B = 5$ to $B = 100$). The chosen queries we use for DBLP execute in a reasonable time on our multi-core platforms (between 1 and 100 seconds), have various node selection patterns, and thus lead to a range of performance behaviors. See the top part of Table 3 for full details on each query.

As in Section 5, we measure the performance for all possible $p/c/h$ combinations for each query to determine the best combination. Figure 8 shows the parallel speedup achieved for $Q2_{dblp}$ for each $p/c/h$ combination on DiRT. The results indicate that, like previous synthetic experiments, the best configuration for queries over the DBLP dataset as well is also PCH-2/1/5. However, the maximum speedup achieved on queries over DBLP is 3.08, which is much less than previous synthetic experiments. We see similarly poor parallel performance for all queries we execute on the DBLP dataset.

The results of executing six diverse queries over DBLP are shown in Table 4 and indicate that PCH outperforms SWP and WQ in all cases. However, we find that while some parallel speedup is achieved, it is much lower than that seen for synthetic experiments. Furthermore, it is inconsistent across queries. On both platforms, speedup between 1.92 and 3.11 is achieved for queries $Q2_{dblp}$, $Q5_{dblp}$, and $Q6_{dblp}$ (results shown in boldface). However, queries $Q1_{dblp}$, $Q3_{dblp}$, and $Q4_{dblp}$ show low parallel speedup (even a slowdown for

| | Query ID | Query String | Depth | Match Nodes |
|---|---|---|---|---|
| DBLP | $Q1_{dblp}$ | /dblp/incollection/cite | 3 | 736 |
| | $Q2_{dblp}$ | /dblp/article/author | 3 | 1782468 |
| | $Q3_{dblp}$ | /dblp/mastersthesis/* | 3 | 50 |
| | $Q4_{dblp}$ | /dblp/book/cite | 3 | 3319 |
| | $Q5_{dblp}$ | /dblp/inproceedings/pages | 3 | 949501 |
| | $Q6_{dblp}$ | /dblp/www/title | 3 | 1008156 |
| XMark | $Q1_{xmark}$ | /site/open_auctions/open_auction/ bidder/increase | 5 | 59486 × scaling |
| | $Q2_{xmark}$ | /site/people/person/name | 4 | 25500 × scaling |
| | $Q3_{xmark}$ | /site/people/person/profile/interest | 5 | 37688 × scaling |
| | $Q4_{xmark}$ | /site/closed_auctions/closed_auction/ annotation/description/parlist/listitem/text | 8 | 6799 × scaling |

**Table 3: Queries on real-world data sets. For XMark each query is run on several differently scaled datasets, each returning a different number of query matches.**

| Runtime in seconds and (speedup) on DiRT | | | | |
|---|---|---|---|---|
| Query | Seq. | SWP | WQ | PCH |
| $Q1_{dblp}$ | 1.94 (1) | 1.64 (1.18) | 1.82 (1.06) | 1.21 (1.60) |
| $Q2_{dblp}$ | 3.94 (1) | 3.05 (1.29) | 2.85 (1.38) | **1.28 (3.08)** |
| $Q3_{dblp}$ | 1.22 (1) | 1.36 (0.90) | 1.40 (0.87) | 1.22 (1.00) |
| $Q4_{dblp}$ | 1.20 (1) | 1.31 (0.92) | 1.35 (0.89) | 1.24 (0.97) |
| $Q5_{dblp}$ | 5.12 (1) | 5.58 (0.92) | 5.64 (0.91) | **1.78 (2.87)** |
| $Q6_{dblp}$ | 2.73 (1) | 1.80 (1.52) | 1.79 (1.53) | **1.42 (1.92)** |
| Runtime in seconds and (speedup) on Greenwolf | | | | |
| Query | Seq. | SWP | WQ | PCH |
| $Q1_{dblp}$ | 1.37 (1) | 1.29 (1.06) | 1.23 (1.11) | 1.20 (1.14) |
| $Q2_{dblp}$ | 5.17 (1) | 2.67 (1.94) | 2.61 (1.98) | **1.66 (3.11)** |
| $Q3_{dblp}$ | 1.31 (1) | 1.37 (0.96) | 1.41 (0.93) | 1.27 (1.03) |
| $Q4_{dblp}$ | 1.25 (1) | 1.22 (1.02) | 1.35 (0.93) | 1.11 (1.13) |
| $Q5_{dblp}$ | 6.66 (1) | 2.64 (2.52) | 2.58 (2.58) | **2.26 (2.95)** |
| $Q6_{dblp}$ | 2.65 (1) | 1.75 (1.51) | 1.76 (1.50) | **1.19 (2.23)** |

**Table 4: Average runtimes of six queries over the DBLP data set on the DiRT and Greenwolf platforms. On three of the queries, speedup is significant (shown in boldface), though on the others there is very little speedup or even slowdown. Details about each query are given in Table 3.**

$Q4_{dblp}$). We obtain similar parallel speedup on Greenwolf (4 cores) and on DiRT (8 cores), showing that parallel efficiency is low.

By examining the shape of the tree, we can determine if load imbalance is the cause of the poor parallel speedup. DBLP is a very large, shallow tree with widely varying numbers of children per node. Table 5 outlines some details about the dataset and our six queries. We see that the root node has a very large number of children, as expected. For queries $Q1_{dblp}$, $Q3_{dblp}$, and $Q4_{dblp}$ (those that show poor speedup), a very small number of those children match the query. We postulate that this, combined with the small query depth, causes load imbalance for all of our parallel query engines. Figure 9 further illustrates the cause of load imbalance.

| Query | Children of Root | % Children Match Query | Complete Query Matches |
|---|---|---|---|
| $Q1_{dblp}$ | 2767177 | 0.60% | 738 |
| $Q2_{dblp}$ | 2767177 | 25.7% | 1782468 |
| $Q3_{dblp}$ | 2767177 | <0.01% | 50 |
| $Q4_{dblp}$ | 2767177 | 0.32% | 3319 |
| $Q5_{dblp}$ | 2767177 | 36.2% | 949501 |
| $Q6_{dblp}$ | 2767177 | 36.4% | 1008156 |

**Table 5: Details about the DBLP dataset and queries. The root node has an enormous number of children (over 2.7 million). The three queries that lead to poor parallel performance ($Q1_{dblp}$, $Q3_{dblp}$, and $Q4_{dblp}$) have a very small number of sub-match nodes. The high initial branch factor and the low number of sub-matches contribute to load imbalance.**

The large initial branch factor and shallow query depth combine to cause load imbalance that cannot be avoided by our query engines. Since all of the 2.7 million nodes at depth 1 are children of the root node, they must all be processed by a single thread (the thread to which the root node is assigned, say thread 1). While processing these nodes, thread 1 writes query sub-matches to the shared workqueue. Other threads read these nodes from the workqueue and process their children. As Figure 9 shows, however, the small number of matches and shallow total depth enables these nodes to be processed quickly, causing other threads to remain mostly idle while thread 1 works. This bottleneck at a single thread is also the reason for the comparable parallel performance between Greenwolf and DiRT (more cores do not help since they remain idle anyway).

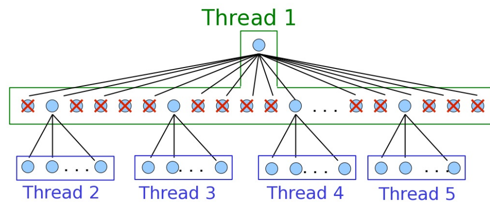



**Figure 9: An illustration of what a query over the DBLP may look like and how work would be distributed among threads by a parallel query engine (nodes assigned to a thread are grouped in boxes). In this example thread 1 has a much larger workload than the other threads, thus degrading parallel performance.**

The three queries on which we do see significant parallel speedup ($Q2_{dblp}$, $Q5_{dblp}$, and $Q6_{dblp}$) have a much higher selectivity, giving the other threads more work while thread 1 evaluates all 2.7 million nodes.

As expected, idiosyncrasies of real-world datasets cause parallel performance degradation. However, despite the extreme shape of the DBLP dataset (2.7 million children of the root and a maximum depth of 3), PCH does achieve some parallel speedup on three of the queries, thereby outperforming SWP and WQ.

## 6. Conclusion

In this paper we have evaluated the performance of three different parallelization approaches that achieve different trade-offs between overhead and load imbalance. The first two approaches, SWP and WQ, correspond to well-known parallel computing techniques, while the third, PCH, is a generalization of the producer-consumer paradigm. We have evaluated implementations of parallel XPath query engines that use these three approaches on a range of synthetic and real-world datasets. Our results indicate that the addition of Hybrid threads to the well-known Producer-Consumer paradigm provides a significant improvement, though an all-Hybrid configuration is not ideal. Through experimentation, we found that the PCH-2/1/5 configuration lead to best overall performance on our 8-core environment (and PCH-1/0/3 using 4 cores). Using this configuration, the PCH query engine consistently achieves good parallel speedup and out-performs our other parallel query engines. However, PCH fails to achieve acceptable parallel performance on some queries of the DBLP dataset. We have identified the cause of this behavior, which is attributed to idiosyncratic features of certain queries on that dataset. Overall, we expect PCH to lead to good results in the vast majority of practical scenarios.

## References


[1] Automata for XML: A survey. *Journal of Computer and System Sciences*, 73(3):289–315, 2007.

[2] A. Aboulnaga, A. Alameldeen, and J. Naughton. Estimating the selectivity of XML path expressions for internet scale applications. In *VLDB*, pages 591–600, 2001.

[3] R. Bordawekar, L. Lim, A. Kementsietsidis, and B. Kok. Statistics-based parallelization of XPath queries in shared memory systems. In *EDBT*, pages 159–170, 2010.

[4] R. Bordawekar, L. Lim, and O. Shmueli. Parallelization of path queries using multi-core processors: Challenges and experiences. In *EDBT*, pages 180–191, 2009.

[5] Z. Chen, H. V. Jagadish, F. Korn, and N. Koudas. Counting twig matches in a tree. In *Internation Conference on Data Engineering (ICDE)*, pages 595–604, 2001.

[6] G. Cong, W. Fan, A. Kementsietsidis, J. Li, and X. Liu. Partial evaluation for distributed XPath query processing and beyond. In *Transactions on Database Systems*, 2011.

[7] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE TKDE*, 19(10):1381–1403, 2007.

[8] S. Haw and C. Lee. Data storage practices and query processing in XML databases: A survey. *Knowledge-Based Systems*, 24:1317–1340, 2011.

[9] B. Karsin. Parallel XPath Query Evaluation on Multi-core Processors. Master's thesis, University of Hawai'i at Manoa, 2012.

[10] M. Ley. Digital bibliography and library project, 2011.

[11] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Parr. XPathLearner: An on-line self-tuning markov histogram for XML path selectivity estimation. In *VLDB*, 2002.

[12] L. Lim, M. Wang, and J. S. Vitter. CXHist : An on-line classification-based histogram for XML string selectivity estimation. In *VLDB*, pages 1187–1198, 2005.

[13] W. Lu and D. Gannon. Parallel XML processing by work stealing. In *SOCP*, pages 31–38, 2007.

[14] M. M. Moro, V. Braganholo, C. F. Dorneles, D. Duarte, R. Galante, and R. S. Mello. XML: Some papers in a haystack. *SIGMOD Record*, 38(2):29–34, 2009.

[15] J. Teubner, T. Grust, S. Maneth, and S. Sakr. Dependable cardinality forecasts for XQuery. *VLDB Endow.*, 1(1):463–477, 2008.

[16] X. Wang, A. Zhou, J. He, W. Ng, and P. Hung. Multi-query evaluation over compressed XML data in DaaS. *Lecture Notes in Business Information Processing*, 74(3):185–208, 2011.

[17] Y. Wu, J. Patel, and H. Jagadish. Estimating answer sizes for XML queries. In *EDBT*, pages 590–608, 2002.

[18] Xalan. `http://xml.apache.org`.

[19] Y. Zhang, Y. Pan, and K. Chiu. A parallel xpath engine based on concurrent NFA execution. In *ICPDS*, pages 314–321, 2010.