A Dynamic Load-balancing Scheme for XPath Queries Parallelization in Shared Memory Multi-core Systems

Xiaocheng Huang
Department of Computer Science, Nankai University, Tianjin 300071, China
Email: huangxiaocheng@dbis.nankai.edu.cn

1Xujie Si, 2Xiaojie Yuan and 2Chao Wang+
1, Department of Electrical Engineering and Computer Science, Vanderbilt University, USA
Email: sixujie@gmail.com
2, Department of Computer Science, Nankai University, Tianjin 300071, China
Email: {yuanxiaojie, wangchao}@dbis.nankai.edu.cn

***Abstract*—Due to the rapid popularity of multi-core processors systems, the parallelization of XPath queries in shared memory multi-core systems has been studied gradually. Existing work developed some parallelization methods based on cost estimation and static mapping, which could be seen as a logical optimization of parallel query plan. However, static mapping may result in load imbalance that hurts the overall performance, especially when nodes in XML are not evenly distributed. In this paper, we solve the problem from another view using parallelizing techniques. We use dynamic mapping to improve XPath query performance, which can achieve better load balance no matter what XML document is queried. Compared with static mapping, dynamic mapping is a more general method. We first design a parallel XPath query algebra called PXQA (ParallelXPath Query Algebra) to explain the parallel query plan. And second, using PXQA we extract the task-dependence graph to define which operations can be executed in parallel and help analyze the overheads of dynamic mapping. At last, we discuss how to do the data partition based on dynamic mapping in accordance with the runtime situations adaptively. Experimental results show that the adaptive runtime XPath queries parallelization achieves a good performance in shared memory multi-core systems.**

***Index Terms*—XPath, Query Parallelization, Shared Memory**

I. INTRODUCTION

XML's emergence as the *de facto* standard for encoding tree-oriented, semi-structured data has brought significant interoperability and standardization benefits. Meanwhile, XPath [15] as an expression language based on XML attracts many researchers. When XML document becomes large and recursive, time cost in XPath querying becomes unbearable. How to accelerate the query speed of XPath becomes a hot topic. Due to the rapid popularity of multi-core processors systems, it is being studied to execute XPath in parallel.

One viable option is to modify existing query algorithms from single-core to multi-cores. However, these methods cannot be applied widely. As a solution, [2] proposes data and query partition which are independent on certain query algorithms. Despite these, there still are some problems to be solved. Existing work focus on optimizing the plan logically, such as partition point chosen. In this paper, we'll optimize the plan physically using the parallelizing techniques.

In this study, we develop a dynamic load-balancing scheme to accomplish better performance. Towards the goal, we first design a parallel query algebra called PXQA. PXQA is based on set theory and takes great advantage of the properties of XPath. It defines a data model, four operators and three rules to generate parallel plan. What's more, according to PXQA, task-dependency graph is extracted. Task-dependency graph plays an important role in task mapping when designing parallel algorithms. Using task-dependency graph, we analyze the overheads of parallelization and make it clear why dynamic mapping leads to load balance. At last, we build dense index among tasks in work pool, which aims to reduce the task mapping overhead caused by exclusive lock and increase the concurrency. Experimental results reveal that, compared with static mapping, dynamic mapping decreases query time greatly when load distribution is uneven in the XML tree and does not hurt performance when load distribution is balanced.

The core contributions of this paper are summarized as follows:

- We design a parallel algebra called PXQA to address the parallel plan. Further, task-dependency graph can be extracted for dynamic mapping.
- Compared with static mapping, we propose a general mapping method adapted to any XML document or any XPath query. We analyze the disadvantages of static

+Corresponding author: Chao Wang

doi:10.4304/jcp.9.6.1436-1445

mapping and show how dynamic mapping leads to load balance.

• We do our experiments in a realistic dataset XStu. Experimental results demonstrate that the dynamic load-balancing scheme works well and has a good scalability.

II. Related Work

There are many studies in how to optimize the performance of a single XPath by improving its traversal pattern or structural join method, such as [12] [16] [17] [19]. However recently, some researchers explore other methods using parallelization, since multi-core systems have become commodity hardwares. It is possible to execute XPath in parallel.

Due to the wide-spread availability of commodity multi-core processors, XML parsing and XPath queries in shared memory multi-core system are being studied these years. Ref. [6] investigates the seemingly quixotic idea of parsing XML in parallel on a shared memory multi-core computer. It preparses XML document to determine the logical tree structure of the document and then use the logical tree to divide the document into chunks. As an improvement, [7] presents a stealing-based parallel XML processing model, using which the load balance among the threads is dynamically controlled. Compared with [7], [11] gives a static load-balancing scheme for parallel XML parsing on multi-core CPUs. It uses a static, global approach to reduce synchronization and load-balancing overhead to improve performance.

Besides XML parsing, XML query in shared-memory multi-core system has been studied meanwhile. There are many studies about parallelization of XML query algorithm. These studies aim to change traditional single-thread XPath query evaluation algorithms to multi-threads. Ref. [5] is based on structural join algorithm. It partitions elements into buckets and then evaluates XPath step (Parent-Child or Ancestor-Descendant relationship) of every buckets. Ref. [3] is based on twig queries. It proposes an efficient parallel PathStack algorithm for processing XML twig queries. In addition, [8] [9][10] aim to execute holistic twig joins in parallel. Ref. [8] proposes a grid metadata model for XML that gives a conceptual view to partition XML data, specifically for holistic twig joins processing. Ref. [9] [10] try to improve workload balance on both static data distribution and dynamic data distribution.

Meanwhile, Parallelization of XPath queries using multi-core processors has been studied. It's quite different from the previous research. The evaluations are based on the scenario where an XPath processor uses multiple threads to concurrently navigate and execute individual XPath queries on a shared XML document. Ref. [2] raises the problem and lists some parallelization issues, such as storage model, cost estimation and load balance. The most important contribution is that it proposes three strategies for parallelizing individual XPath queries: Data partitioning, Query partitioning, and Hybrid (query and data) partitioning. Ref. [1] proposes a parallelization algorithm that uses the statistics together with several heuristics to find and select parallelization points in an XPath query. It describes the statistics-based model used to estimate the running time of different parallel execution strategies. It also shows that query partitioning is involved in semantic of XPath and hardly get better performance than data partitioning.

III. Parallel XPath Query Algebra (PXQA)

In order to explain the parallelization of XPath query, we design a parallel XPath Query Algebra, PXQA. In this paper, PXQA mainly focuses on the presentation of XPath query process. What's more important, the task-dependency graph extracted from PXQA plays an essential role in parallelization analysis. Due to space constraint, more content about PXQA optimization is not included and will occur in future work.

A. Data Model, Operators and Plan Generating

An XML document on which XPath operates is a tree (Figure 3). An abstract data set called the XML Information Set (Infoset) is recommended by W3C in [14]. An XML document's information set consists of a number of information items. Information items include the document information item, element information item, attribute information item, etc. PXQA is based on XML Infoset.

There are three operators in PXQA, i.e.

ψ: query processing. $\psi_{xpath}(infoset)$ returns the query result of $xpath$ on the context nodes $infoset$. The query result is an infoset too.

σ: data partition. σ means choosing part of the infoset.

op: the common operations on sets, such as union $(\cup)$, intersection $(\cap)$.

Note that, during the partition, the document order of infoset may be messed up. So we need to reform document order of results. In the following statement, we will use infoset to represent XML document and intermediate results. Using $\psi, \sigma, op, infoset$, all parallel query plans generated are based on the following three rules where the rule 1 focuses on the infoset and the rule 2 and 3 focus on the XPath.

1. If $infoset = infoset1\ op\ infoset2$ then

$$\psi_{xpath}(infoset) = \psi_{xpath}(infoset1)\ op\ \psi_{xpath}(infoset2)$$

2. An XPath query can be rewritten as

$$\psi_{xpath}(infoset) = \psi_{xpath1}\left(\psi_{xpath2}(infoset)\right)$$

The result of $\psi_{xpath2}(infoset)$ is an infoset, which is the context nodes of $xpath1$.

3. From [15], productions of OrExpr and AndExpr of XPath are:

$$OrExpr ::= AndExpr("or" AndExpr)^*$$

AndExpr ::=

ComparisonExpr("*and* "*ComparisonExpr*)*

The instances of *OrExpr* and *AndExpr* can be rewritten as

$$\psi_{xpath}(infoset) = \psi_{xpath1}(infoset)\ op\ \psi_{xpath2}(infoset)$$

To illustrate how to generate a plan, we list two examples using data partition and (or) query partition.

Data partition proposed by [2] means executing the same (sub)query on different sections of the same XML document, for which we use σ. An instance using data partition is:

Example 1: If we want to execute the XPath */FILE/EMPTY//NP* on a XML document, for example, *treebank.xml* , using two processes while partitioning the data on EMPTY half, the query plan is

$$\begin{aligned}&\psi_{/FILE/EMPTY//NP}(treebank.xml)\\ &=\psi_{//NP}(\psi_{/FILE/EMPTY}(treebank.xml))\ //\ rule2\\ &=\psi_{//NP}(\sigma_{part}(\psi_{/FILE/EMPTY}(treebank.xml)))\bigcup\\ &\quad\sigma_{part}(\psi_{/FILE/EMPTY}(treebank.xml)))\ //\ data\ partition\\ &=\psi_{//NP}(\sigma_{part}(\psi_{/FILE/EMPTY}(treebank.xml)))\bigcup\\ &\quad\psi_{//NP}(\sigma_{part}(\psi_{/FILE/EMPTY}(treebank.xml)))\ //\ rule1\end{aligned}$$

Proposed by [1], *EMPTY* here is called **Partition Point**.

Another instance, using query partition, which is also proposed in [2], meaning executing different (sub)queries on the same XML document, is:

Example 2: If we want to execute XPath */FILE/EMPTY* [*count*(.//*NP*)>1 *and count*(.//*VP*)>1] while partitioning the query into two subqueries */FILE/EMPTY*[*count*(.//*NP*)>1]and */FILE/EMPTY*[*count*(.//*VP*)>1], the query plan is:

$$\begin{aligned}&\psi_{/FILE/EMPTY[count(.//NP)>1\ and\ count(.//VP)>1]}(treebank.xml)\\ &=\psi_{/FILE/EMPTY[count(.//NP)>1]}(treebank.xml)\cap\\ &\quad\psi_{/FILE/EMPTY[count(.//VP)>1]}(treebank.xml)\ //\ rule3\end{aligned}$$

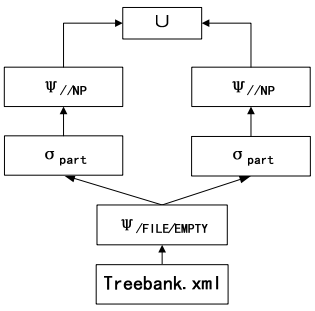


Figure 1 Task-dependency Graph of Example 1

Any plan should be deduced in terms of the three operators and the three rules above. As a counterexample, we do not provide extra operators. If we want to partition the query */FILE/EMPTY//NP* into two single queries: *query*1: */FILE/EMPTY* and *query*2: *//NP*, and then choose the nodes from query2 who are descendent of nodes in query1, this kind of plan is not accepted by PXQA, because PXQA does not provide the operator of choosing descendent nodes.

B. Plan Choosing

Given a XPath query, there are many candidate plans. e.g. Given a long XPath */FILE/EMPTY/S// VP*[*count*(.//*NP*)>1], the candidate plans could be:

candidate plan1:

$$\begin{aligned}&\psi_{/FILE/EMPTY/S//VP[count(.//NP)>1]}(treebank.xml)\\ &=\psi_{//VP[count(.//NP)>1]}(\sigma_{part}(\psi_{/FILE/EMPTY/S}(treebank.xml)))\\ &\bigcup\psi_{//VP[count(.//NP)>1]}(\sigma_{part}(\psi_{/FILE/EMPTY/S}(treebank.xml)))\end{aligned}$$

candidate plan2:

$$\begin{aligned}&\psi_{/FILE/EMPTY/S//VP[count(.//NP)>1]}(treebank.xml)\\ &=\psi_{/S//VP[count(.//NP)>1]}(\sigma_{part}(\psi_{/FILE/EMPTY}(treebank.xml)))\\ &\bigcup\psi_{/S//VP[count(.//NP)>1]}(\sigma_{part}(\psi_{/FILE/EMPTY}(treebank.xml)))\end{aligned}$$

Note that, the main difference of the two plans is the different partition points. Diverse query plans lead to diverse performances. It has been studied to decide the best partition point in [1]. [1] proposed a cost-estimation model based on query specifics and data statistics. We will apply the algorithm of [1] to optimize the plan. Nevertheless, given the best partition point, the query plan cannot be optimized logically any more. So we optimize the query plan from physical perspective by executing it in parallel.

IV. Parallel Execution of Plan

A. Task-dependency Graph Extraction

Given a plan, first we extract its task-dependency graph. A task-dependency graph is a directed acyclic graph in which nodes represent tasks and directed edges indicate the dependencies amongst tasks. A task-dependency graph determines which tasks can run in parallel and which must be executed serially, i.e. to distinguish independent and dependent tasks. This extraction phase is simple. Task-dependency graphs of ***Example*1** and ***Example*2** can be quickly extracted as Figure 1 and Figure 2. The data stream flowing across the operators is infoset.

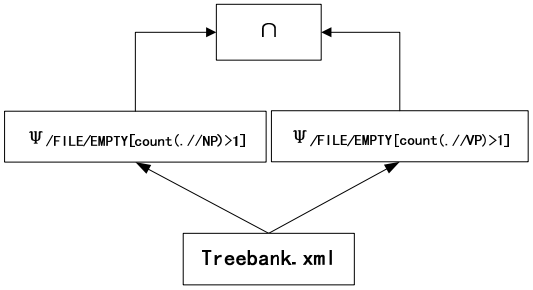


Figure 2 Task-dependency Graph of Example 2

We can see from Figure 1 that $\psi_{//NP}$ depends on $\psi_{/FILE/EMPTY}$ and different $\psi_{//NP}$ are independent of

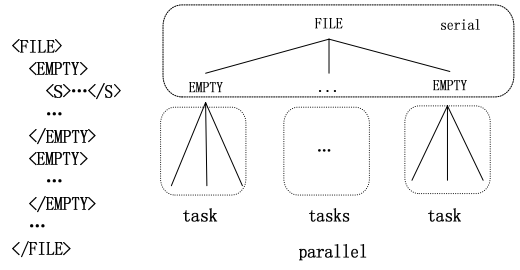
each other. It's similar from Figure 2 that $\psi_{/FILE/EMPTY[count(.//NP)>1]}$ is independent of $\psi_{/FILE/EMPTY[count(.//VP)]>1}$ and $\cap$ depends on both $\psi_{/FILE/EMPTY[count(.//NP)>1]}$ and $\psi_{/FILE/EMPTY[count(.//VP)]>1}$ This kind of dependency and independency is helpful for the analysis of load-balancing design below. The experiments in [2] show that query partition is harder to get good performance than data partition. So in this paper we focus on data partition.

B. Overhead Analysis of Parallelization

Formerly, [1] constructs a cost model to evaluate the expense considering the data distribution or the complexity of XPath query. The model recommends the partition method (data or query or hybrid partition) and partition point. Then using static mapping, distributes tasks evenly among processes, which may subsequently cause load imbalance. It is a bottleneck to obtain the optimal performance under the given best partition point. So in this paper, we try to accomplish load balance to accelerate XPath query.

To achieve the shortest execution time, the overheads of executing the tasks in parallel must be minimized. For a given task decomposition, there are two key sources of overhead. One is inter-process interaction. The other is processes idle. Processes idle is caused by a variety of reasons. Considering executing XPath query in share-memory system, there are some problems to take into account. As is known, in the shared-memory system, for consistency, thread interactions are in form of synchronization between concurrent tasks. Talking about parallel XPath query using data partition, the overhead of interactions is so small to be ignored. That's because that in parallel phase, the execution of several ψ are independent from each other.

In the contrast, the cost of processes being idle plays an important role on query performance. Uneven load distribution may cause some processes to finish much later than others. It reduces the efficiency of CPUs and results in a bad performance. It is especially true when executing XPath query in parallel in share-memory systems. On one hand, task decomposition typically is coarse-grained. Consider **Example 1**, the tasks to be executed in parallel are thousands of sub-trees rooted by EMPTY, shown as Figure 3. Each sub-tree rooted by EMPTY is a task. However, the structures of all these sub-trees may vary seriously. Some have a large quantity of descendant nodes, while the others do not. Consequently, even task distribution leads to uneven load distribution, which will impact overall performance significantly. On the other hand, it is hard to construct a general model to estimate the cost of tasks that is adapted to any query. A model may fit for some queries and datasets, but not for others. Load-imbalance always occurs.



a. treebank.xml b. Data Access Model

Figure 3 Data Access Model of Example 1

As mentioned, even task distribution may cause uneven load distribution when using static mapping. So we try to accomplish load balance by dynamic mapping. The core point is to schedule tasks at runtime.

C. Dynamic Mapping

We use the work pool [4] model. The model is characterized by a dynamic mapping of tasks onto processes in which any task may potentially be performed by any process. In our scenario, the total tasks are known as a priori, i.e. that there is no task produced dynamically during executing parallel tasks. So the work pool shrinks monotonously as processes fetch tasks to execute continuously.

Initially, all processes are idle and all tasks in work pool are in unplayed state. Then, idle process applies a certain amount of tasks from work pool and then executes the tasks. After the process accomplishes the tasks, it becomes idle again. So it repeats the procedure until the work pool becomes empty. This way of self-scheduling turns out to be very useful in parallel XPath query in share-memory system. In our design, every time process applies the same amount of additional tasks, so-called chunk. Here, chunk is the granularity of dynamic mapping. Proper granularity selection is an important factor to the performance of dynamic mapping. Too coarse granularity may cause load imbalance like static mapping. Too fine granularity will cause heavy extra interaction overhead.

The biggest overhead of dynamic mapping is caused by task application and assignment, which is exclusive lock overhead in work pool model. To minimize this overhead, we build a dense index on all sub-trees as shown in Figure 4. There is an index pool containing the indices of all tasks in the work pool. An exclusive lock is used for index pool to guarantee consistency when concurrent task applications are issued by multiple processes. Therefore, task assignment and execution are completely independent. That is, when some processes fetch tasks from index pool serially, other processes execute their tasks simultaneously on the work pool. This

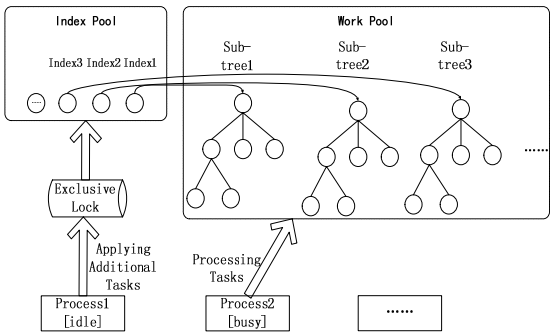


Figure 4 Processing of Dynamic Mapping

model increases concurrency obviously. In addition, our experimental evaluation show that even interaction overhead of extreme fine granularity is too light to impact performance perceptibly.

D. Performance Analysis

In this subsection, we will show that our dynamic scheme will get much better load balance compared with static mapping, especially when the problem size is large. Analytically, we pay much attention to the parallel phase after data partitioning, which is the core difference between dynamic and static mapping. As shown in Subsection 4.1, ψ in different processes are independent of each other, so we first assume there is no extra overheads in parallel phase.

We denote by n the number of tasks in work pool, and g the mapping granularity. So there are $\lceil n/g \rceil$ chunks to be assigned to threads.

LEMMA 1. For two threads i and j, let p_1, p_2 ... p_{k_i} and q_1, q_2 ... q_{k_j} be two chunk sequences which are assigned to the two threads and run on them sequently, respectively. Let t_p be the execution time of task p, t_g be the largest execution time among those of $\lceil n/g \rceil$ chunks and T_i denote the running time of thread i. Using the scheduling strategy described in the last subsection,

$$\left|T_j - T_i\right| \le t_g$$

PROOF. First, we prove $T_i \ge T_j - t_{q_{k_j}}$ by contradiction. WLOG, suppose that $T_i < T_j - t_{q_{k_j}}$. After the p_{k_i} finishes, thread i will no longer get scheduled by any chunk. It must be true that at the very moment when p_{k_i} finishes, there is no unfinished chunk. Obviously, $T_i < T_j - t_{q_{k_j}}$ implies that at least one chunk, q_{k_j}, remains unprocessed after p_{k_i} finishes, which is a contradiction. So there is $T_i \ge T_j - t_{q_{k_j}}$. As the same, $T_j \ge T_i - t_{p_{k_i}}$. Since $t_{p_{k_i}} \le t_g \; and \; t_{q_{k_j}} \le t_g$, $\left|T_j - T_i\right| \le t_g$.

With Lemma 1 we know that, for any two processes, the difference of their final running time cannot be larger than the longest execution time of a single chunk. So, with appropriate g, the load is well balanced. What's more, $\left|T_j - T_i\right| \le t_g$ is irrelevant of n, i.e. when n becomes large, the absolute load difference does not rise.

Now consider the overheads involved in dynamic scheduling. As mentioned, we use index pool to reduce the cost of lock. There are $\lceil n/g \rceil$ times exclusive lock acquirements and releases in total. Assuming the cost of lock and chunk assignment of every chunk with mapping granularity g is $C_{lg} + C_{ag}$, so overhead of our algorithm is estimated as $\lceil n/g \rceil * \left(C_{lg} + C_{ag}\right)$. The overhead will rise as n becomes large. Actually, we observe in our experiments that $\left(C_{lg} + C_{ag}\right)$ is too small compared with the computation time. The overhead is acceptable.

There are a few of performance metrics for parallel algorithms. Like serial algorithms, the execution time is an important metric. The parallel runtime T_p is the time that elapses from the moment a parallel computation starts to the moment the last processing element finishes [4]. Suppose that each thread gets at least one task chunk, we estimate the worst case of the parallel time:

$$T_p = n * t_c / p + t_g + \left(\lceil n/g \rceil - p + 1\right) * \left(C_{lg} + C_{ag}\right)$$

where t_c denotes the maximum execution time of a single task.

The speedup S and the efficiency E are metrics specific for parallel algorithms. The speedup is defined as the ratio of sequential time T_s to parallel time T_p to measure how much performance gain is achieved by parallelizing a sequential algorithm [4]. The efficiency measures how efficient the processing elements are employed that is defined as the ratio of speedup to the number of processing elements p [4].

Another important metric is isoefficiency function for scalability [4]. If a parallel algorithm can keep the efficiency fixed by increasing both problem size W and the number of processing elements p simultaneously, it is scalable. Isoefficiency function quantifies scalability by describing how faster W must be increased with respect to p to keep the efficiency fixed. Slower growing isoefficiency function means higher scalability. Following the method described in [4], we get the isoefficiency function of our algorithm $W = KT_o = K\left(t_g + \left(\lceil n/g \rceil - p + 1\right) * \left(C_{lg} + C_{ag}\right)\right)$ where T_o denotes the total overhead, and K is a constant. Since W is actually the amount of computation of the problem, we can express it using the sequential time, $W = T_s = nt_c$. Solve the above isoefficiency function, we get

$$W = O(p)$$

which implies that our algorithm is perfectly scalable.

V. EXPERIMENTS

A Experimental Design

We have performed extensive experiments on several types of XPath queries over many XML datasets. In this section, we describe our experiments and present a representative subset.

Prototype Implementation. To afford the XPath queries, we choose Xerces-C and XALAN DOM APIs [13], since DOM APIs have been proved to be a very useful standard for XML query. We use the XALAN interfaces as the

operator ψ, and we implement the other operators in PXQA. To afford the parallel computing, we use pthread to create more than one threads to compute in parallel. The experiments are performed on TYAN FT72-B7015, a 4U server with two Intel Xeon 5520 quad-core processors. It supports up to eight physical threads. The server has 12GB of memory running Federo 12.

Experimental Schemes. We design our experiments from the following four aspects. First, we show that the mapping granularity affects the performance of dynamic mapping. We will compare coarse-grained and fine-grained scenarios to determine an appropriate granularity. Second, we present load balance and faster query speed accomplished by dynamic mapping when the dataset is not even-distributed. In addition, third, we show that dynamic mapping does not hurt the good performance achieved by static mapping when dataset is already even-distributed. At last, we validate the scalability of our dynamic scheme.

Datasets and Queries. We experiment with several typical datasets, such as DBLP, Treebank. These datasets are all even-distributed, so the results are similar. Though Treebank is not very large, it is deeply recursive. The queries in Treebank are more complex. We just show the results of Treebank on behalf of these typical datasets. What's more, we experiment with a realistic dataset, XStu, from a college. XStu is a set of realistic data, which collects the records of students' personal information and course details. XStu is wide and deeply recursive. Due to space constraint, we show part structure of XStu in Figure 5. The original XStu is 663 MB from Table 2. But as said above, to measure the scalability of dynamic mapping, we need datsets of different size. So we extract part of XStu to form XStu-2 and XStu-3, which are much smaller than XStu, see also Table 2. XStu collects information of 35778 students, while XStu-2 collects 1517 students and XStu-3 collects 12274 students. These three datasets are uneven-distributed and will lead to load-imbalance, which we will show next.

The characteristics of the datasets are shown in Table 2. In addition to the real XStu data, we also use XPath expressions extracted from a real-world query workload provided by the college. Table 2 lists the representative XPath queries over the two XML datasets. We have tested many queries, the results are similar. We show only two representative queries. Query TB is from [2], and query XS is a real query.

Comparing Arguments. Given a data partition point and threads, we test our dynamic mapping algorithm with the plain static mapping algorithm as the control group. Besides the common performance metrics for parallel algorithms, the parallel time, speedup, isoefficiency function (described in Subsection 4.4), we also use a dedicated metric for this problem, the load gap among the threads, denoted by d.

TABLE 1
QUERIES AND PARTITION POINTS

Dataset		Key	Query	$T_s(s)$	Partition Point	Number of tasks
TreeBank		TB	/FILE/EMPTY/S//VP[count(.//NP)>1]	29	EMPTY	52851
XStu-2		XS2	/totalstudents/s//sc[./FinalExamScore/text()>=60]	6.39	s	12274
XStu-3		XS3	/totalstudents/s//sc[./FinalExamScore/text()>=60]	130.57	s	1517
XStu		XS	/totalstudents/s//sc[./FinalExamScore/text()>=60]	371.6	s	35778

TABLE 2
CHARACTERISTICS OF TREEBANK AND XSTU

	Treebank	XStu-2	XStu-3	XStu
Size(MB)	84	86.9	334	663
Number of Elements	9 Million	2 Million	34 Million	68 Million
Max Depth	36	7	7	7
Avg Depth	7.87	7	7	7

To evaluate the load distribution among threads, the running time of all threads are collected to indicate the load of threads. If load in a thread is higher, the running time of the thread is longer. We define d as:

$$d = \left(Max\left(T_1, T_2 ... T_p\right) - Min\left(T_1, T_2 ... T_p\right)\right) / Min\left(T_1, T_2 ... T_p\right)$$

$$T_i : the\ running\ time\ of\ the\ i^{th}\ process$$

The higher d is, the more imbalanced load is.

TABLE 3
EFFICIENCY OF DIFFERENT PARAMETERS

	p=2	p=4	p=6	p=8
XS2	4.38	4.32	3.83	2.66
XS3	22.43	22.2	22.19	13.37
XS	32.25	32.02	27.67	19.2

B Evaluation1: Mapping Granularity

The query XS, based on XStu, returns all passed course selections. In the data partitioning strategy, original query is split into two subqueries: /*totalstudents*/*s* (*serial*), and .//*sc*[./*FinalExamScore*/*text*()>=60] (*parallel*).

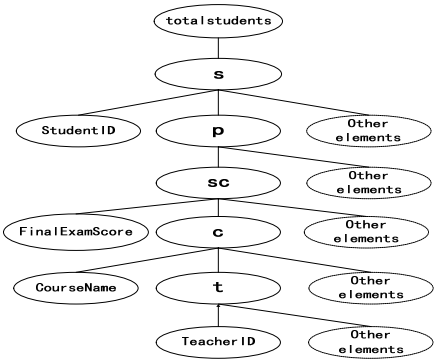


Figure 5 Part Structure of XStu

Using four threads, the plan is:

$$\psi_{/totalstudents/s//sc[./FinalExamScore/text()>=60]}(XStu.xml)$$

$$=\psi_{//sc[./FinalExamScore/text()>=60]}\left(\sigma_{part}\left(\psi_{/totalstudents/s}(XStu.xml)\right)\right)$$

$$\cup\psi_{//sc[./FinalExamScore/text()>=60]}\left(\sigma_{part}\left(\psi_{/totalstudents/s}(XStu.xml)\right)\right)$$

$$\cup\psi_{//sc[./FinalExamScore/text()>=60]}\left(\sigma_{part}\left(\psi_{/totalstudents/s}(XStu.xml)\right)\right)$$

$$\cup\psi_{//sc[./FinalExamScore/text()>=60]}\left(\sigma_{part}\left(\psi_{/totalstudents/s}(XStu.xml)\right)\right)$$

At the beginning, $\psi_{/totalstudents/s}(XStu.xml)$ is executed serially. After serial phase there are 35778 unprocessed tasks (sub-trees rooted by s) in work pool. And next, we experiment dynamic mapping with different mapping granularity. We test from fine mapping granularity 1 to coarse mapping granularity 10000, shown in Figure 6. The curve is flat when mapping granularity is in [1, 100], which implies that tasks applying and assignment overhead is negligible. Index pool mechanism is the most important reason for this. Then, the curve rises slowly when mapping granularity is between 100 and 2000 and rises sharply when mapping granularity is beyond 2000. It is reasonable, for the mapping granularity is too coarse. In the next statement, we will always experiment with mapping granularity 100.

C. Evaluation2: Comparison between Dynamic Mapping and Static Mapping

Table 1 presents a set of XPath queries used for the experimental evaluation. The two queries include those with long chains of child steps, path predicates, functions such as text() and count() and different axes. The two instances TB and XS stand for the two aspect of dynamic mapping. For one thing, when the data is well-distributed, an instance using static mapping causes load balance, like TB, by coincidence. Then we will show dynamic mapping will not reduce the good performance. The extra overhead of dynamic mapping can be made up. For another thing, when the data is uneven-distributed, an instance using static mapping leads to load-imbalance such as XS. Consequently, the execution time of XPath query is long. Yet, we will show dynamic mapping will gain a good load distribution and accelerate the query speed.

The query TB (Table 1), based on Treebank, returns all elements labeled VP, who have more than one descendants labeled NP. In the data partitioning strategy,

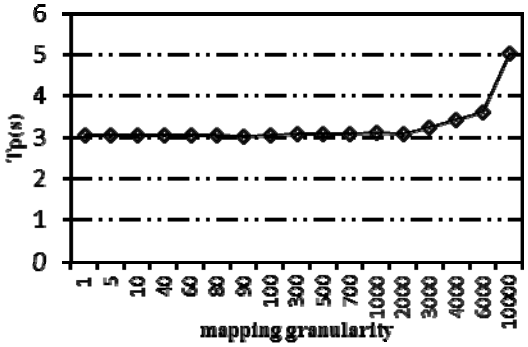


Figure 6 Mapping Granularity

EMPTY as partition point, is chosen according to [1]. Original query is split into two subqueries: /*FILE*/*EMPTY*(*serial*), and ./*S*//*VP*[*count*(.//*NP*)>1](*parallel*). Using four threads, the plan is:

$$\psi_{/FILE/EMPTY/S//VP[count(.//NP)>1]}(treebank.xml)$$

$$=\psi_{/S//VP[count(.//NP)>1]}\left(\sigma_{part}\left(\psi_{/FILE/EMPTY}(treebank.xml)\right)\right)$$

$$\cup\psi_{/S//VP[count(.//NP)>1]}\left(\sigma_{part}\left(\psi_{/FILE/EMPTY}(treebank.xml)\right)\right)$$

$$\cup\psi_{/S//VP[count(.//NP)>1]}\left(\sigma_{part}\left(\psi_{/FILE/EMPTY}(treebank.xml)\right)\right)$$

$$\cup\psi_{/S//VP[count(.//NP)>1]}\left(\sigma_{part}\left(\psi_{/FILE/EMPTY}(treebank.xml)\right)\right)$$

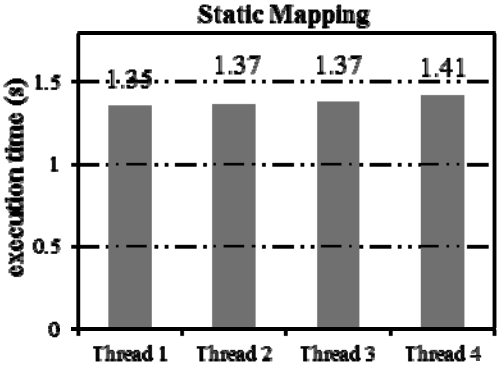


Figure 7 Load Balance Distribution of TB with 4 threads (Static)

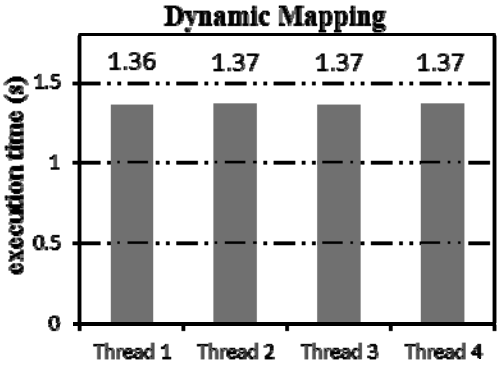


Figure 8 Load Balance Distribution of TB with 4 threads (Dynamic)

At the beginning, $\psi_{/FILE/EMPTY}(treebank.xml)$ is executed serially. As shown in Table 1, after serial phase there are 52851 unprocessed tasks (sub-trees rooted by EMPTY) in work pool. And then, we experiment two kinds of mapping methods in parallel phase, static mapping and dynamic mapping. Figure 7 shows the

running time of four threads about TB using static mapping in [1]. Using static mapping, three threads get $\lfloor 52851/4 \rfloor = 13212$ tasks and the forth gets 13215 tasks. The running time of four threads are 1.355s, 1.365s, 1.374s, 1.414s respectively. The load of four threads are more or less the same with d = 4.35%. So, using static mapping, TB does not cause load imbalance with $T_p = 1.414s$.

Correspondingly, Figure 8 presents the performance of TB using dynamic mapping. In this case, the running time of four threads are 1.362s, 1.367s, 1.365s and 1.373s, with d = 0.8%, when mapping granularity is 100. Compared with static mapping, dynamic mapping has smaller d, which means dynamic mapping gains a little better load balance. But the imperfect thing is that the execution time is not much better than static mapping. Figure 11 presents the execution time (T_p) of dynamic mapping with different mapping granularity. The first point stands for static mapping and other points stand for dynamic mapping with certain mapping granularity. It's seen that dynamic mapping does not always gain better performance than static mapping. That's because static mapping has already gained good load balance and dynamic mapping costs some time in overheads, such as lock cost. However, it is acceptable. In a word, dynamic mapping does not hurt the performance when load distribution is even.

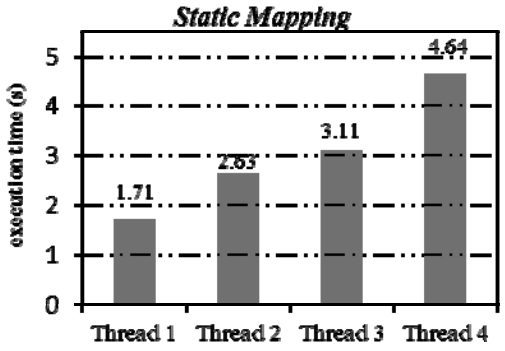


Figure 9 Load Balance Distribution of XS with 4 threads (Static)

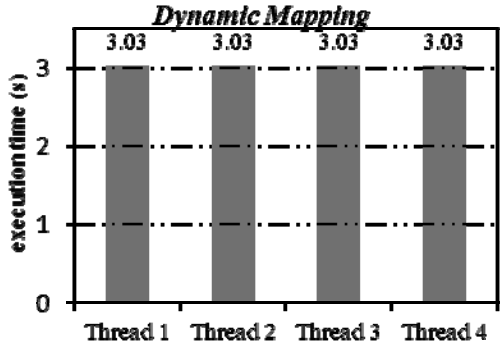


Figure 10 Load Balance Distribution of XS with 4 threads (Dynamic)

The execution of XS is similar with TB. $\Psi_{/totalstudents/s}(XStu.xml)$ is executed serially at first. After serial phase there are 35778 unprocessed tasks in work pool. And next, we experiment static mapping and dynamic mapping.

Figure 9 shows the running time of XS using four different threads in static mapping. It is clear that loads among threads are quite different from others, as 1.711s, 2.629s, 3.107s, 4.645s. d reaches 171.48%. There is a serious load-imbalance. It is not surprising of this load imbalance if looking inside the structure of XStu. As mentioned above, XStu is about the students' personal information and selected courses. Naturally, the elements labeled s are ordered by StudentID. Smaller StudentID means the student having higher grade, which indicates he chooses more courses. As a result, the thread allocated too many s with higher StudentID will last long and vice versa. Load-imbalance takes place in this case. In contrast, Figure 10 gains load balance clearly with d = 0.17% when using dynamic mapping with mapping granularity 100. As a result, the total execution time decreases from 4.657s to 3.048s. It accelerates the query speed by 34.5%. It is a great achievement.

D. Evaluation3: Scalability Evaluation

Our dynamic scheme for load balance is designed and tested for smaller datasets on fewer processing elements. However, the real datasets we want to solve may be much larger, and the computers may contain larger number of processing elements, such as cluster system. So in this evaluation, we investigate the scalability of our scheme.

As mentioned in Subsection 4.4, there are two factors to be taken into account when evaluating scalability. One is problem size. As shown in Table 2, we extract part of XStu to form XStu-2 and XStu-3, which have different problem size from each other. XStu-2 is 86.9 MB and XStu-3 is 334 MB while XStu is 663 MB. We experiment the same query *./totalstudents/s//sc*[*./FinalExamScore/text*()>=60] with partition points on XStu, XStu-2 and XStu-3. To distinguish, XS, XS2,XS3 are in use. It is only true for part of parallel algorithms that the efficiency increases with increasing problem size and fixed number of processing elements. The other factor is the number of processing elements (threads). We experiment dynamic mapping of XS, XS2, XS3 using 2, 4, 6, 8 threads respectively. All parallel algorithms will show efficiency degradation with increasing number of processing elements and fixed problem size. If a parallel algorithm can keep constant efficiency by increasing both system size and problem size, we say it is scalable. When mapping granularity is 100, the efficiency is shown in

Table 3.

Note that our dynamic mapping algorithm achieves super-linear speedup, that is, an efficiency greater than 1. This is because XALAN algorithm is cache sensitive. Multi-thread means multiplying cache size, so improves performance dramatically.

As shown in

Table 3, with fixed number of threads, the efficiency increases remarkably as problem size increases. This implies that our algorithm can make better use of the processing elements as the problem size increases. On the other hand, with the fixed problem size, the efficiency remains the same when p increases from 2 to 4, and

decreases slowly when p increases to 6 and 8. We can see that, the efficiency remains fixed (even increases significantly) as both the problem size and the number of processing elements increases. This result shows the good scalability of our algorithm.

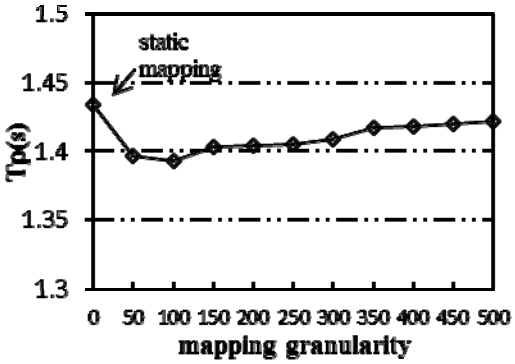


Figure 11 Comparison of Static and Dynamic Mappings on TB

VI Conclusions and Future Work

Motivated by the emergence of multi-core processors in commodity systems, this paper presents that dynamic mapping improve performance by up to 34.5% when static mapping can not distribute load evenly among threads. Meanwhile, it shows the good scalability of the dynamic mapping algorithm agreeing well with our theoretical analysis. This means that the algorithm can efficiently deal with large datasets on large systems. In this paper, we implement dynamic mapping algorithm in the multi-core shared memory system. However, our algorithm is platform-independent. It can easily be ported to Message Passing platforms such as clusters. The work pool (index pool) is implemented as a main process rather than a global object. Task assignment consistency is implemented by message passing between processes instead of exclusive lock. Certainly, the overhead of message passing is higher than shared object access. An important future work is to minimize this overhead.

It would be very interesting to consider the scenario where the dataset is too large to fit in the main memory. Besides, the optimization of PXQA and granularity chosen automatically need to be studied more. In addition, GPU is becoming the most promising platform for high performance computing. Design and implementation of our algorithm on GPU is also planned.

Acknowledgment

This work is partly supported by the grants from the National High-Tech Research and Development Plan of China (No.2013AA013204); the Doctoral Fund of Ministry of Education of China (No.20120031120038), and Natural Science Foundation of Tianjin, China (No.13JCQNJC00100 and No.3ZCZDGX02200).

References

[1] Rajesh Bordawekar, Lipyeow Lim, Anastasios Kementsietsidis, and Bryant Wei-Lun Kok. “Statistics-based Parallelization of XPath Queries in Shared Memory Systems.” In EDBT, pages 159–170, New York, NY, USA, 2010.

[2] Rajesh Bordawekar, Lipyeow Lim, and Oded Shmueli.“Parallelization of XPpath Queries Using Multi-core Processors: Challenges and Experiences”, in EDBT, pages 180–191, New York, NY, USA, 2009.

[3] Jianhua Feng, Le Liu, Guoliang Li, Jianhui Li, and Yuanhao Sun. “An Efficient Parallel Pathstack Algorithm for Processing XML Twig Queries on Multi-core Systems.” Database Systems for Advanced Applications, volume 5981 of Lecture Notes in Computer Science, pages 277–291, 2010.

[4] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. Introduction to Parallel Computing (2nd Edition). Addison Wesley, 2 edition, 2003.

[5] Le Liu, Jianhua Feng, Guoliang Li, Qian Qian, and Jianhui Li. “Parallel Structural Join Algorithm on Shared-memory Multi-core Systems”, in WAIM, pages 70–77, Washington, DC, USA, 2008.

[6] Wei Lu, Kenneth Chiu, and Yinfei Pan. “A Parallel Approach to XML Parsing”, in GRID, pages 223–230, Washington, DC, USA, 2006.

[7] Wei Lu and Dennis Gannon. “Parallel XML Processing by Work Stealing”, in SOCP, pages 31–38, New York, NY, USA, 2007.

[8] Imam Machdi, Toshiyuki Amagasa, and Hiroyuki Kitagawa. “GMX: An XML Data Partitioning Scheme for Holistic Twig Joins”, in iiWAS, pages 137–146, 2008.

[9] Imam Machdi, Toshiyuki Amagasa, and Hiroyuki Kitagawa. “XML Data Partitioning Schemes for Parallel Holistic Twig Joins”, IJWIS, 5(2):151–194, 2009.

[10] Imam Machdi, Toshiyuki Amagasa, and Hiroyuki Kitagawa. “XML Data Partitioning Strategies to Improve Parallelism in Parallel Holistic Twig Joins”, in ICUIMC, pages 471–480, 2009.

[11] Yinfei Pan, Wei Lu, Ying Zhang, and Kenneth Chiu. “A Static Load-balancing Scheme for Parallel XML Parsing on Multicore CPUs”, in Cluster Computing and the Grid, IEEE International Symposium on, 0:351–362, 2007.

[12] Jens Teubner, Torsten Grust, Sebastian Maneth, and Sherif Sakr. “Dependable Cardinality Forecasts for XQuery”, in PVLDB, 1(1):463–477, 2008.

[13] http://www.w3.org/DOM/.

[14] http://www.w3.org/TR/xml-infoset/.

[15] http://www.w3.org/TR/xpath20/.

[16] Xiaojie Yuan, Xin Lian, Ya Wang, Xiangyu Hu, Haiwei Zhang, "DPIX: A Dynamic Path Index for XML Data in Relational Database", JDCTA, Vol. 7, No. 5, pp. 434 ~ 442, 2013

[17] LEAGUE, C., ENG, K..“Schema-Based Compression of XML Data with Relax NG” in Journal of Computers, North America, 2, dec. 2007

[18] Husheng Liao, Weifeng Shan, Hongyu Gao, “Automatic Parallelization of XQuery Programs” in Journal of Software, North America, 2013

[19] LI, R., LUO, J., YANG, D., HU, H., CHEN, L. “A Scalable XSLT Processing Framework based on MapReduce” in Journal of Computers, North America, 8, sep. 2013.

Xiaocheng Huang is a Ph.D. candidate in Computer Science of Nankai University. Her research interests are broadly in the areas of data and information management, including XML query/update, and keyword search/query on scientific workflows (context-free graph grammars).

Xujie Si is a Ph.D. candidate in Computer Science, Vanderbilt University.

Xiaojie Yuan is a professor and dean of College of Computer and Control Engineering in Nanakai University. Her research focuses on data management, including database and information retrieval.

Chao Wang is a Ph.D. candidate in Computer Science of Nankai University.