The Journal of Supercomputing (2022) 78:5420–5449
https://doi.org/10.1007/s11227-021-04074-y

Parallel XPath query based on cost optimization

Rongxin Chen[1,2] · Zhijin Wang[1] · Hang Su[3] · Shutong Xie[1] · Zongyue Wang[1]

Accepted: 4 September 2021 / Published online: 24 September 2021

Abstract
The performance of XPath query is the key factor to the capacity of XML processing. It is an important way to improve the performance of XPath by making full use of multi-threaded computing resources for parallel processing. However, in the process of XPath parallelization, load imbalance and thread inefficiency often lead to the decline of parallel performance. In this paper, we propose a cost optimization-based parallel XPath query method named coPXQ. This method improves the parallel processing effect of navigational XPath query through a series of optimization measures. The main measures include as follows: first, by optimizing the storage of XML node relation index, both storage and access efficiency of the index are improved. Secondly, load balancing is realized by a new cost estimation method according to the number of XML node relations to optimize parallel relation index creation and parallel primitive execution. Thirdly, the strategy of determining the number of worker threads based on parallel effectiveness estimation is utilized to ensure the effective use of threads in query. Compared with the existing typical methods, the experimental results show that our method can obtain better parallel performance.

Keywords XPath query · Relation index · Cost estimation · Load balancing · Parallel effectiveness

1 Introduction

As a powerful semi-structured data [1] description tool, XML is suitable for describing complex web information and system integration information. As a general data exchange and storage standard, XML has been widely used in various fields, and the requirement for the performance of XML processing is increasing. XPath [2] query is the key part of XML data processing, and its performance directly affects

✉ Rongxin Chen
ch2002star@163.com

Extended author information available on the last page of the article

the processing ability of XML. The common implementation methods of XPath query include twig [3] method and navigational [4] method. Twig uses path pattern matching to find structures that satisfy the conditions of XML node relationship and has high query efficiency for branch paths described by predicates. However, twig is not suitable for some operations, such as reverse axis and sibling axis; thus, it is difficult to support complete XPath query semantics. The navigational process is to evaluate each query step one by one according to the syntax of XPath and navigate through the hierarchical structure of XML document tree until the final result is obtained. The navigational method is simple and direct, and easy to implement the rich semantics of XPath; thus, it is widely applied to the implementation of many popular XML engines [5–7]. However, due to the iterative processing of the query step in navigational method, the query efficiency is often low, and optimization is essential to improve performance.

Recently, with the popularity of multi-threaded computing environment, making full use of multi-threaded parallel computing to improve performance has become a common optimization approach. Parallelization related work in XML processing, including XML parsing parallelization [8, 9], XPath query parallelization [10–13] and XQuery parallelization [14, 15]. The implementation technology of XPath parallelization includes automata method [16, 17], pipeline [18, 19], task parallelism [20], and data parallelism [11, 12]. Kim et al. [16] presented a multi-query parallel method for multi-core computing, which transforms multiple XPath expressions into FSA-based query indexes, and then process XML streams in parallel. Jiang et al. [17] proposed an XML grammar-aware parallelization method for XPath which leverages the grammar of the semi-structured data to guide the design of parallelization. Karsin et al. [18] gave a low latency pipelined XPath parallel query scheme. Only simple XPath without predicate is tested in their work. Chen et al. [19] proposed a pipelined XPath evaluation method based on cost optimization, which can support more complete XPath semantics. Huang et al. [20] proposed a task parallelism method for XPath query. The method determines the parallelizable parts by constructing task dependency graph and dynamically divides query tasks to balance the load. Shnaiderman et al. [13] proposed XPath parallelization methods based on twig pattern. Their basic idea is to perform twig evaluation on different subtrees of XML document tree in parallel. In recent years, the ability of GPU in parallel computing has attracted extensive attention. There have been some XPath parallelization work for GPU processing. For example, Moussalli et al. [21] used GPU to accelerate the XPath evaluation in multiple query scenarios, which improves the query throughput. Kim et al. [22] presented an XML path filtering method for large collection of XPath queries, which uses matrix-based XML stream to process queries on GPU [21] and [22] only show simple path query cases, but still lack support for more complex operations such as predicates. GPU can provide a large number of threads for concurrent processing. However, XPath evaluation process is not a large-scale numerical calculation suitable for GPU, and it processes complex semi-structured XML data. GPU still has some limitations in processing XPath evaluation.

In the implementation of efficient parallel processing, data parallelism is a simple and efficient method, which usually uses barrier mechanism to synchronize [23]. Load balancing can reduce the synchronization overhead and effectively

improve the parallelization effect. Therefore, it is an important factor to be considered in the process of parallel processing. There have been lots of researches on load balancing in parallel processing for general computing [24, 25]. The load balancing for XML parallelization needs to be combined with the specific scenarios of XML processing. For instance, Pan et al. [9] proposed a method of load balancing by static task partition in the process of XML parsing; Zuo et al. [26] gives a method of load balancing through query cost estimation in the process of XML query to guide the selection of parallel query plan. Subramaniam et al. [27] presented an XML query method which can adapt to distributed environment and support the optimal processing of distributed load. As the input of XPath processing is XML semi-structured data, XML encoding [28] is necessary to obtain appropriate access mechanism, and the index is used to improve the access efficiency [29]. Widemann et al. [30] applied the node relation information of XML to the type checking of XPath query and XSLT transformation, which improved the accuracy and flexibility of the type checking. The pM2 method proposed in [12] transforms XPath query into XML node relation search by introducing node relation matrix. Through the parallel construction of relation matrix and data parallel execution of query primitives, the query performance is improved in multicore computing environment. However, there are still some problems, such as the high storage cost of relation matrix; the load imbalance is prone to occur when the relation matrix is created and query primitives are executed in parallel; the query primitives are often inefficient in parallelization. These problems greatly affect the parallel effect.

In order to improve the parallel effect of navigational XPath query, in this paper, we propose a cost optimization-based parallel XPath query method named coPXQ. This method improves the parallel effect through a series of optimization measures. Through the comparison test and analysis with the existing methods, it shows that our method has better parallel performance. The main contributions of this paper include as follows:

1. An optimized XML node relation index storage scheme is proposed, which has better storage efficiency and access efficiency, and can support efficient query.
2. A cost estimation method based on the number of XML node relations is proposed to realize the load balancing in both parallel index creation and parallel query primitive execution.
3. A method of determining the number of worker threads based on parallel effectiveness estimation is proposed to ensure the effective utilization of threads when query primitives are executed in parallel.

The remainder of the paper is structured as follows. Section 2 introduces the related work, focusing on the parallelization of navigational XPath evaluation and the related cost estimation issues. Section 3 describes the method of coPXQ in detail, including optimized relation index storage, parallel index creation and parallel query primitives. Section 4 presents comparative experiments and evaluation

of relevant indicators. The last section draws the conclusion and proposes future work.

2 Related work

This paper focuses on the parallelization of navigational XPath query. Typical studies in this area include the work done by Bordawekar et al. [11, 31]. Later, Sato et al. [32] carried out XPath parallelization research on the new XML query platform BaseX [5] on the basis of Bordawekar et al.'s work. In the previous work [12], we propose an efficient XPath evaluation method named pM2 based on XML node relation matrix, which adopts data parallelism in processing. In addition, we recently proposed a pipeline-based XPath evaluation method called PXQ [19] to support efficient query of XML data streams in parallel environment.

Bordawekar et al. [11] proposed three strategies for parallelizing XPath queries: data partitioning, query partitioning and hybrid partitioning. The parallel query plan was rewritten manually and tested on various test platforms. On the basis of this work, a method of automatic parallelization of XPath queries using cost model to guide parallel query plan optimization is proposed in [31]. It uses XML statistics to estimate the relative efficiencies of different alternative plans and find an optimal parallel XPath processing plan. This method is suitable for automatic parallelization of navigational evaluation. However, an accurate cost model is needed to implement. The experimental results in [11] and [31] show that the parallel effects of different query plans are quite different, and the selection of query plans is cost sensitive. In addition, the cost model based on statistics used in the method is closely related to the specific situation of XML data and XPath expression, so it needs extra overheads to obtain statistical information. Sato et al. [32] used Bordawekar's idea of data partition to parallelize XPath queries, and combined with the advantages of BaseX engine, experiments were carried out on both the server side and the client side, respectively, but no automatic parallelization approach was provided.

In [12], we proposed a parallel evaluation method pM2 for XPath based on XML node relation matrix. The pM2 method consists of two data parallel execution phases: the construction of relation matrix and the evaluation of query primitives. The query function is carried out by relation matrix search, and parallel query primitives are selected for each query step to maximize the parallelizable opportunities. The pM2 method performs data partition according to the number of XML nodes. In the execution of query primitives, the length of input sequence is used to estimate the cost of computation. However, this cost estimation method is difficult to reflect the real load condition; therefore, it can easily lead to load imbalance.

In the process of query parallelization, it is usually necessary to combine the cost model to guide the generation of query plan. Hartmann et al. presented a cost estimation method for XML query in [33]. By estimating the sizes of intermediate results, the method is used to guide the vertical fragmentation of XML data in distributed environment and improve the parallel processing performance.

Georgiadis et al. [34] designed the physical operators of XPath and the corresponding cost model, which provides a basis for the optimization of XPath evaluation. Hidaka et al. [35] proposed a relative cost model for XQuery, which is used to optimize the XML query plan. The cost is calculated recursively according to XQuery expression, involving the estimation of data sizes and probability of selection. The relative cost is estimated at the level of logical query plan, and the overhead related to parallel scheduling is not reflected. In [31], Bordawekar et al. [19] used a statistical cost model to guide query and data partition in XPath parallelization. The model considered the cardinality of the XPath query step and the selection rate with predicates. However, this model mainly considers the computational cost and also does not consider the cost of parallel scheduling. In PXQ, during the construction of pipeline, the query primitive sequence is divided and merged according to the estimated cost, so as to form a pipeline with relatively balanced load in each stage.

Since load balancing is an important factor to ensure the parallel performance, it is usually concerned about in parallel processing [24, 26]. Consider that if all available threads are used for parallel processing when the computational cost is low, the parallel efficiency may be reduced [36] due to the coordination overhead of threads. Such parallel effectiveness problem should also be considered in the process of automatic parallelization. The coPXQ method in this paper obtains the optimized parallel effectiveness by setting the estimated number of worker threads. In PXQ method, by merging low-cost query steps, the granularity of pipeline stage is improved, which essentially limits the number of threads, so as to improve the efficiency of pipeline. The main features of related works are summarized in Table 1.

3 Proposed method

3.1 The framework of coPXQ

The proposed method consists of three stages: XML parsing and statistics, index creation and primitive evaluation, as shown in Fig. 1. On the one hand, according to the input XML document, XML parsing and index creation are carried out to provide the required data for query. On the other hand, the XPath query expression is rewritten as a parallelized sequence of query primitives for query processing. The first stage is preprocessing, which is in the process of XML document parsing. The statistical information needed for cost estimation is collected in this stage. The second stage is the creation of relation indexes. According to the estimated cost of index creation, the input region encoding data is partitioned to achieve load balance and improve the efficiency of parallel creation of index data. In the third stage, parallel primitive evaluation is carried out. According to the estimated cost of primitive query, on the one hand, the parallel effectiveness is estimated to determine the number of worker threads; on the other hand, the partition of input data is guided to achieve load balancing during query. The cost model plays a key role in the whole process of parallel optimization. According to the characteristics of relation index, the cost model proposed in this paper is based on XML statistical information,

Table 1 Comparison of main related works

	Evaluation method	Parallel method	Cost estimation	Parallel effectiveness
PXQ [19]	Navigational	Pipeline	Cost is estimated according to the number of XML nodes and node relationships to guide the construction of pipeline stage	Merge low-cost query steps, and each pipeline stage is allocated a worker thread
PTS [13]	Twig	Data parallelism	Cost is estimated according to the size of XML subtree, and provide enough subtree data for each thread	Not available
pNav [11, 31]	Navigational	Data parallelism	Calculate cardinality and selectivity according to the statistical information of XML nodes, and guide the data and query partitioning	Not available
pM2 [12]	Navigational	Data parallelism	Roughly partition workload according to the number of XML nodes	Not available
coPXQ (This work)	Navigational	Data parallelism	Accurately partition workload according to the number of XML node relations	For the parallel processing of each query step, the number of worker threads required is estimated

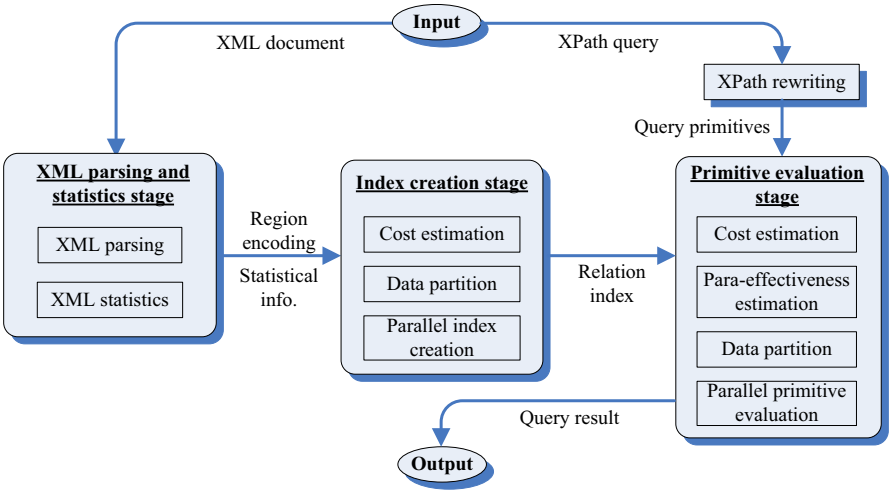


Fig. 1 Processing framework of coPXQ

Table 2 Symbols and corresponding meanings

Symbol	Meaning
$\mathcal{E}_u$	The region encoding of an XML node u. u is represented by the node ID value
$\mathcal{I}_u$	The relation index of a node u, see Definition 1
$\mathcal{P}_k$	The block boundary of a block k, see Definition 2
$N_u^{\text{DS\_CH}}$	The total number of descendants (denoted as 'DS' type) and children (denoted as 'CH' type) in the subtree with node u as its root
N_u^{AT}	The number of attributes (denoted as 'AT' type) of a node u
N_u^{TAT}	The number of non-direct attributes (denoted as 'TAT' type) of a node u, that is, the number of attribute nodes of u's descendants
$\mathcal{R}_u$	The number of XML node relations of a node u
$G_{\text{p/s}}$	Parallel effectiveness of a primitive, see Definition 3
C	The estimated cost, which is associated with execution time, comparable, and dimensionless. Its specific semantics are distinguished according to the subscript and the superscript

which can reasonably estimate the computational cost and provide the possibility for effective load balancing and optimizing the number of threads.

The symbols and their corresponding meanings in this paper are listed in Table 2.

3.2 Optimized node relation index storage

The region encoding of XML nodes is represented by 6-tuple in this paper. For example, the region encoding of node u is <*ID*, *nodeType*, *tagName*, *begin*, *end*, *level*>, where *ID* is the node ID. Since the node ID is unique, node u can be represented by the ID value. *nodeType* is XML node type. We consider the most

frequently used XML node types: element and attribute, so *nodeType*∈{ *ELEMENT*, *ATTRIBUTE*}. *tagName* is the label name of node; *begin* is the start position of the node in document; *end* is the end position of node; *level* is the hierarchical value of node.

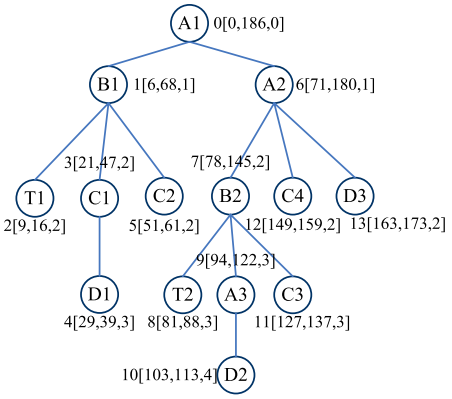
There are various relation between any two XML nodes u and v. According to the query semantics of XPath [2], these relations include parent (denoted as 'PA'), child (denoted as 'CH'), ancestor (denoted as 'AN'), descendant (denoted as 'DS'), attribute (denoted as 'AT'), preceding-sibling (denoted as 'PS'), following-sibling (denoted as 'FS'), preceding (denoted as 'PP'), following (denoted as 'FF'), etc. The relation between two nodes is bi-directional, and the two directions are semantically relative. Therefore, the relation type of one direction can be deduced from the one of the other direction. When discussing the relation between two nodes u and v, this paper refers to the relation from node u to node v. We specify that the node u is prior to node v in the document order, that is, the ID value of u is less than that of v, so that only one-direction relation need to be generated. The relation type values in this paper are limited to DS, CH, AT, TAT and NN, which represent descendant, children, attributes, indirect attribute relation and unrelated relation, respectively. Among them, the TAT and NN type are used for auxiliary purpose. In this paper, coPXQ utilizes the relation indexing infrastructure, which provides an efficient way to deal with navigational evaluation.

Definition 1 (*Relation index*) Refers to a storage structure that records the effective relation between XML nodes. An index is represented by a tuple as $< u, v, r_{u\to v} >$, which indicates the unique relation type of node u and node v is r, $r\in${DS, CH, AT}. The relation of a node u refers to the relation index set of node u and all its subsequent nodes v in the document order that have effective relation (DS, CH or AT). In order to save the storage space and facilitate access, the node ID is used to represent the node, and the relation index is simplified as $< id_v, r_{u\to v} >$, then the relation index $\mathcal{I}_u$ for node u is a tuple set of all nodes v which are relative with node u. $\mathcal{I}_u$ is described as $\mathcal{I}_u = \bigcup_j \{< id_{vj}, r_{u\to vj} >\}$. For an entire XML document with N nodes, the index space is expressed as $\mathcal{I} = \bigcup_{i\in N} \mathcal{I}_{ui} = \bigcup_{i\in N} \{\bigcup_{j\in N} \{< id_{vj}, r_{ui\to vj} >\}\}$. For a given node list $\mathcal{E}$, the index space is expressed as $\mathcal{I}^{\mathcal{E}} = \bigcup_{ui\in\mathcal{E}} \mathcal{I}_{ui} = \bigcup_{ui\in\mathcal{E}} \{\bigcup_j \{< id_{vj}, r_{ui\to vj} >\}\}$.

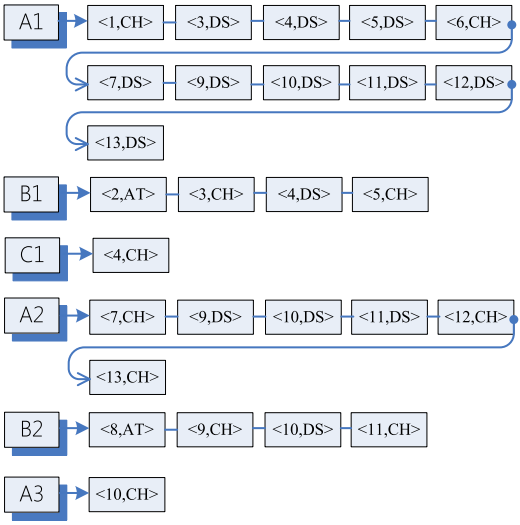
Compared with pM2, the index of coPXQ only stores {DS, CH, AT} types; thus, the storage space and index scan time can be saved effectively. Queries involving other relation are processed by query rewriting transformation. The XML document in Fig. 2a has 14 nodes, including 12 element nodes and 2 attribute nodes. The document tree with simplified region encoding is shown in Fig. 2b. The tag name of each node is marked in the circle, and different nodes with the same tag name are distinguished by numbers. The string beside the circle is a simplified region encoding, which contains the node ID value, the document start position of the node, the document end position of the node and the hierarchical information of DOM. For example, the code "5 [51,61,2]" of node C2 indicates that the ID value of the node is 5. The position starts from the 51st byte to the end of the 61st byte in the XML document and is at the level 2 of the XML document tree. The corresponding node

```
<A>
    <B T="t01">
        <C>
            <D>d01</D>
        </C>
        <C>c01</C>
    </B>
    <A>
        <B T="t02">
            <A>
            <D>d02</D>
            </A>
            <C>c02</C>
        </B>
        <C>c03</C>
        <D>d03</D>
    </A>
</A>
```

(a) XML document



(b) XML tree



(c) Relation index

Fig. 2 XML encoding and relation index

relation index structure is shown in Fig. 2c. The nodes with index are only limited to non-leaf nodes, and the index item is <*node ID*, *relation type*>.

Proposition 1 *Suppose that the root node of a subtree of an XML document tree is u, then the number of relations corresponding to u is the sum of the number of DS or CH type nodes in the subtree and AT type nodes in the root node u. We have the following equation.*

$$\mathcal{R}_u = N_u^{DS\_CH} + N_u^{AT} \tag{1}$$

Proof Let a node u be the root node of the subtree. If u has no descendant node, then $\mathcal{R}_u=0$. When u has 1 descendant node, which must be a child or attribute node of u, then $\mathcal{R}_u=1$. When there are more than one descendant node under u, there are two ways to connect to u, either to the u itself or to a descendant of u. When connected to u itself, if it is an element node, a child node is added, and $N_u^{\text{DS\_CH}}$ increases by 1; if it is an attribute node, N_u^{AT} increases by 1. All of these increase $\mathcal{R}_u$ by 1. For the node connected to the descendant node of u, if it is an element node, a DS type node is added to u, that is, $N_u^{\text{DS\_CH}}$ increases by 1, thus $\mathcal{R}_u$ increases by 1; if it is an attribute node, no nodes of any type are added to u, while $N_u^{\text{DS\_CH}}$ and N_u^{AT} are unchanged.

In the process of XML parsing, the statistical information of each XML node u, including $N_u^{\text{DS\_CH}}$, N_u^{AT} and N_u^{TAT}, will be collected. The process of parsing and obtaining statistical information is described by Algorithm 1. XML document is parsed in sequential style. When parsing a start tag, a new node u is created with a new ID in document order, and the current *level* value is recorded (line4–5). When parsing an end tag, statistics is made and the current node u information is updated (line6–11).

Algorithm 1. GetXMLStats($\mathcal{D}$)
Input: $\mathcal{D}$ - XML document data.
Output: XML statistics.
1: id←0, level←0; // Record the node ID and the level value of the current node.
2: **while**(!EOF($\mathcal{D}$))
3: p←Parsing($\mathcal{D}$);
4: **if**(p is StartElement) // When parsing start tag.
5: u←id, $\mathcal{E}_u$ ←CreateNewNode(p), $\mathcal{E} \leftarrow \mathcal{E} \cup \{\mathcal{E}_u\}$, id←id+1, level←level+1;
6: **if**(p is EndElement) // When parsing end tag
7: **foreach** node id k∈(u,id)
8: **if**($\mathcal{E}_k$.nodeType=ELEMENT) $N_u^{DS\_CH} \leftarrow N_u^{DS\_CH}+1$;
9: **if**($\mathcal{E}_k$.nodeType=ATTRIBUTE ∧ ($\mathcal{E}_k$.level-1=level)) $N_u^{AT} \leftarrow N_u^{AT}+1$;
10: **if**($\mathcal{E}_k$.nodeType=ATTRIBUTE ∧ ($\mathcal{E}_k$.level-1≠level)) $N_u^{TAT} \leftarrow N_u^{TAT}+1$;
11: $\mathcal{R}_u \leftarrow N_u^{DS\_CH}+N_u^{AT}$, level←level-1;
12: **return** $\mathcal{R}_u$, N_u^{TAT}

For the XML case in Fig. 2a, the XML statistics obtained are the number of relations $\{\mathcal{R}_0=11; \mathcal{R}_1=4; \mathcal{R}_3=1; \mathcal{R}_6=6; \mathcal{R}_7=4; \mathcal{R}_9=1\}$ and non-direct attributes of each node $\{N_0^{\text{TAT}}=2; N_6^{\text{TAT}}=1\}$, where the numerical subscript is the node ID value. The number of relations reflects the storage complexity of relation index. Figure 3

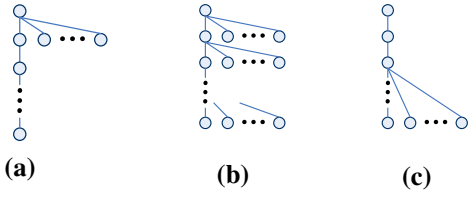


Fig. 3 Distribution of nodes in the XML tree: **a** the case with the minimum number of relations; **b** the general case; **c** the case with the maximum number of relations

shows the distribution of nodes in the XML tree. It can be found that the lower level the nodes are distributed, the more times they are included by other nodes, so the total number of relations is also increased. For an XML document with N nodes, generally, the tree depth of XML is fixed to H. Figure 3c indicates that in addition to the H nodes required to form the depth H of the tree, the maximum number of total relations can be obtained if a node is added to the position of H-1 level. The number of all relations is $\mathcal{R}=H(H-1)/2+(N-H)*(H-1)$. In Fig. 3a, the minimum number of relations is obtained. Except for the H nodes required to form the tree depth H, all the other nodes are the children of the root node, and its relation number is $\mathcal{R}=H(H-1)/2+(N-H)$. Figure 3b shows a situation in between. Therefore, the maximum storage complexity of relation index in coPXQ is measured by the relation number in Fig. 3c. H is regarded as a constant, except for the case of very small XML files, generally $H<<N$, so there is $O(\mathcal{R})=O(N)$.

3.3 Load-balanced parallel index creation

3.3.1 Node relation calculation

Calculating the relation between XML nodes is the basic step of creating index. Considering the directivity and semantic correspondence of the relations between XML nodes, by specifying the order of nodes to be calculated, we can avoid repetitive calculation and obtain one-direction relation. Algorithm 2 describes the process of calculating the relation between the two nodes.

Algorithm 2: GetRelation($\mathcal{E}_u$, $\mathcal{E}_v$)

Input: Input the two nodes u and v to be calculated, and specify $\mathcal{E}_u$.id<$\mathcal{E}_v$.id.

Output: Relation type value $r_{u\to v}$.

1: if(($\mathcal{E}_u$.begin<$\mathcal{E}_v$.begin)∧($\mathcal{E}_v$.begin<$\mathcal{E}_u$.end)∧($\mathcal{E}_u$.level=$\mathcal{E}_v$.level-1)∧($\mathcal{E}_v$.nodeType=ELEMENT)) $r_{u\to v}$←CH;

2: else if(($\mathcal{E}_u$.begin<$\mathcal{E}_v$.begin)∧($\mathcal{E}_v$.begin <$\mathcal{E}_u$.end)∧($\mathcal{E}_u$.level ≠$\mathcal{E}_v$.level-1)∧($\mathcal{E}_v$.nodeType=ELEMENT)) $r_{u\to v}$←DS;

3: else if(($\mathcal{E}_u$.begin<$\mathcal{E}_v$.begin)∧($\mathcal{E}_v$.begin <$\mathcal{E}_u$.end)∧($\mathcal{E}_u$.level=$\mathcal{E}_v$.level-1)∧($\mathcal{E}_v$.nodeType=ATTRIBUTE)) $r_{u\to v}$←AT;

4: else if(($\mathcal{E}_u$.begin<$\mathcal{E}_v$.begin)∧($\mathcal{E}_v$.begin<$\mathcal{E}_u$.end)∧($\mathcal{E}_u$.level≠$\mathcal{E}_v$.level-1)∧($\mathcal{E}_v$.nodeType=ATTRIBUTE)) $r_{u\to v}$←TAT;

5: else $r_{u\to v}$←NN;

6: return $r_{u\to v}$;

Since only one relation is stored for two nodes, while the descendant relation contains child relation semantically, the constraint $\mathcal{E}_u$.level≠$\mathcal{E}_v$.level-1 is added to line 2 in Algorithm 2. For the case of query descendant node, the child node condition is included in the query primitive design. The purpose of line 4 is to identify the non-direct attribute nodes of node u for optimization. The process of creating relation index is to call Algorithm 2 to calculate and store the relation value between any two nodes.

Proposition 2 *Consider all possible relation values {DS, CH, AT, TAT, NN}, in the process of creating the relation index of node u, when calculating the relationship between u and its subsequent nodes v in document order, if the first relation value is NN, the relation value with subsequent nodes in document order must be NN.*

Proof Suppose the first document order successor node with the relation value of NN with node u is *v0*, if there is a node *v1*, which is the successor node of *v0*, and the relation value with u is not NN, then the discussion is as follows. According to the document order, there is "$\mathcal{E}_u$.begin < $\mathcal{E}_{v0}$.begin < $\mathcal{E}_{v1}$.begin" (Condition 1). According to Algorithm 2, the condition that the relation value is not NN is "$\neg((\mathcal{E}_u$.begin < $\mathcal{E}_v$.begin) $\bigwedge$ ($\mathcal{E}_v$.begin < $\mathcal{E}_u$.end))", that is, equivalent condition "($\mathcal{E}_u$.begin > $\mathcal{E}_v$.begin) $\bigvee$ ($\mathcal{E}_v$.begin > $\mathcal{E}_u$.end)". For u and *v0*, the condition "($\mathcal{E}_u$.begin > $\mathcal{E}_{v0}$.begin) $\bigvee$ ($\mathcal{E}_{v0}$.begin > $\mathcal{E}_u$.end)" (Condition 2) must be satisfied. According to the Condition 1, it can be deduced that the left part of Condition 2 is not true, so "$\mathcal{E}_{v0}$.begin > $\mathcal{E}_u$.end" (Condition 3) must be satisfied. For u and *v1*, according to Algorithm 2, it can be deduced that "($\mathcal{E}_u$.begin < $\mathcal{E}_{v1}$.begin) $\bigwedge$ ($\mathcal{E}_{v1}$.begin < $\mathcal{E}_u$.end)" (Condition 4) should be satisfied. Combined with Condition 3 and the right part of Condition 4, it can be deduced that "$\mathcal{E}_{v1}$.begin < $\mathcal{E}_{v0}$.begin" (Condition 5) is satisfied. According to the contradiction between Condition 5 and Condition 1, the relation value between u and *v1* can only be NN.

According to Proposition 2, when a NN relation is obtained, the coPXQ method does not need to continue to calculate the relation value with the subsequent nodes. Through the optimized design, the index construction can skip a lot of calculation with the result value of NN, thus greatly saving time. Obviously, for non-NN relation calculation, all relation values except TAT type need to be stored in relation index. This also relates the computational cost of index creation to the number of relations. In the process of index creation, Algorithm 2 needs to be called $\mathcal{R}_u + N_u^{\mathrm{TAT}}$ times for each node u. In addition, it needs to calculate until the result of the first NN relationship is obtained; thus, an additional call is required. Since the calculation cost here is only used to guide data partition, the calculation cost of calling Algorithm 2 has been normalized to a measure with a value of 1. Therefore, the number of calls to Algorithm 2 can be used to estimate the total computational cost of index creation. The estimation formula is

$$C_{\mathrm{index}} = \sum_{u=0}^{N-1} \left(\mathcal{R}_u + N_u^{\mathrm{TAT}} + 1\right) \approx \sum_{u=0}^{N-1} \left(\mathcal{R}_u + N_u^{\mathrm{TAT}}\right) \tag{2}$$

Corollary 1 *If the node ID value of node u is k, then the node ID values of node v corresponding to all non-NN type relation of u are sequential sequences starting from k + 1.*

Proof According to Proposition 2, when the first NN type relationship appears, the calculation of relation will be terminated. Therefore, the results of the previous calculation are all non-NN type. In the process of calculating the relation for node u,

the subsequent nodes v are calculated with node u one by one according to the document order, so the non-NN type relation result is a continuous sequence according to the node ID sequence. Since the first node is next to u, its node ID value is $k + 1$.

Corollary 2 *The node relation storage $\mathcal{I}_u$ corresponding to node u is a continuous sequence in the order of node ID.*

Proof According to Corollary 1, during getting the relation index of node u, the node v of non-NN type relation obtained is a continuous sequence according to the node ID order. These non-NN type relations are added into the relation index $\mathcal{I}_u$ one by one, so the index information of nodes v in $\mathcal{I}_u$ must be a continuous sequence in the order of node ID.

3.3.2 Parallel index creation

In the process of index creation, we need to get the relation value between every two nodes, so the cost of index creation is reflected in the number of calls of Algorithm 2. In the process of parallel index creation, firstly, the data to be processed should be partitioned into blocks, and then multi-threads are used to process the blocks in data parallelism. Data partition is the key factor affecting load balancing. Obviously, the number of relations contained by each node is generally quite different. From the perspective of subtree, the node with lower level contains more nodes, so it generally has more relations; while the number of relations owned by leaf node is 0. As discussed in Sect. 3.3.1, the cost of index creation is related to the number of relations. Therefore, this paper proposes a load balancing method based on relation number. The idea is to partition the input data into data blocks according to the cost estimation, while the cost estimation is based on the relation number rather than the number of nodes. Suppose the load is evenly partitioned into m blocks, according to Eq. (2), the computational cost of each block is

$$C_m = C_{\text{index}}/m = \sum_{u=0}^{N-1} \left(\mathcal{R}_u + N_u^{\text{TAT}}\right)/m \tag{3}$$

The advantage of this method is that the load can be partitioned in fine granularity according to the number of node relations, which is conducive to load balancing, and the estimated computational cost can better reflect the real load.

Definition 2 (*Block boundary*) Refers to the partition location information of a given space. It contains two components, i and j, which record the position of node u and node v, respectively. If the boundary of block k is represented by $\mathcal{P}_k$, then the two components are $\mathcal{P}_k$.i and $\mathcal{P}_k$.j. In the index creation stage, the given space refers to the region encoding information of the entire XML document; in the query primitive evaluation stage, the given space refers to the index information of the input nodes.

The whole process of parallel index creation includes two stages: block boundary extraction and data parallel processing, as described in Algorithm 3. Line 1–5

are used to estimate the computational cost and then calculate the block boundary. In line 6–29, according to the block boundary, the threads are allocated to create indexes for each block in data parallelism. The XML data stored in the form of region encoding are partitioned, and the boundary information of partition is recorded by $\mathcal{P}_k$. The value of $\mathcal{P}_k$.i records the position of node u. Because the ID of all nodes is continuous in the whole XML document, the node ID value of node u is used as the boundary value. The value of $\mathcal{P}_k$.j records the position of node v. From Proposition 2, the nodes corresponding to node u are equivalent to all nodes in the subtree with node u as the root. According to Corollary 1, the node IDs of nodes v which have non-NN type relation are continuous sequence starting from node ID + 1 of node u, obviously, the offset of node v's node ID value can be reflected by the relation number. Line 5 is used to calculate the node ID value of node v, which is taken as the boundary value. Line 12–16 are used to process the first node of the block; line 17–22 are used to process the intermediate nodes of the block; and line 23–26 are used to process the last node of the block. Line 16 and 22 are used to realize optimization processing according to Proposition 2. The index creation in coPXQ only needs to calculate the relation value of non-NN type, and the number of non-NN types is the number of index relations. Since the spatial complexity of relation index is O(N) (see Sect. 3.2, N denotes the number of XML nodes), the time complexity of index creation can achieve the effect of O(N).

Algorithm 3. ParaCreateRIndex($\mathcal{E}$, $\mathcal{R}$, N^{TAT} ,*m*)

Input: $\mathcal{E}$ - XML node list; $\mathcal{R}$ - the relation number list of each node; N^{TAT} - the number of non-direct attribute nodes of each node; *m* - the number of blocks.
Output: The relation index of the entire XML data.

1: Calculate C_m according to Equation (3), k←0, Rc←0; // k is the ID of the block and is used for counting, k∈[0,m-1]; Rc is the cumulative count of relations.
2: **foreach** node id u in $\mathcal{R}_u$
3: Rc←Rc + $\mathcal{R}_u + N_u^{TAT}$;
4: **while**(Rc>= C_m) // Conditions for partitioning new blocks.
5: $\mathcal{P}_k$ .i←u, $\mathcal{P}_k$ .j←u+ $\mathcal{R}_u + N_u^{TAT}$ -(Rc- C_m), k←k+1, Rc←Rc- C_m ;
6: $\mathcal{I} \leftarrow \emptyset$;
7: ***Exec*** ← Executor(); //*Exec* is the thread pool manager.
8: ***Latch*** ← CountDownLatch(m); //*Latch* is the synchronous barrier counter.
9: **foreach** thread t∈{T|0≤T≤m-1} //t is the thead id.
10: $\mathcal{I}^t \leftarrow \emptyset$; // $\mathcal{I}^t$ is the partial result under thread t.
11: $\mathcal{I}^t$ ←***Exec***.execute(***Task***(){ // Start of task body.
12: **foreach** node id j∈{J| $\mathcal{P}_i$ .j+1≤J≤N-1}
13: r←GetRelation($\mathcal{E}_{\mathcal{P}_i.i}, \mathcal{E}_j$); // Call Algorithm 2 to get the relation value.
14: **if**(r =DS ∨ r=CH ∨ r=AT)
15: $\mathcal{I}^t_{\mathcal{P}_i.i} \leftarrow \mathcal{I}^t_{\mathcal{P}_i.i} \cup \{<j,r>\}$;
16: **if**(r=NN) **break**;
17: **foreach** node id i∈{I| $\mathcal{P}_i$ .i <I< $\mathcal{P}_{i+1}$ .i-1}
18: **foreach** node id j∈{J|i+1≤J≤N-1}
19: r←GetRelation($\mathcal{E}_i, \mathcal{E}_j$);
20: **if**(r=DS ∨ r=CH ∨ r=AT)
21: $\mathcal{I}^t_i \leftarrow \mathcal{I}^t_i \cup \{<j,r>\}$;
22: **if**(r=NN) **break**;
23: **foreach** node id j∈{J| $\mathcal{P}_{i+1}$ .i-1≤J≤ $\mathcal{P}_{i+1}$ .j-1}
24: r←GetRelation($\mathcal{E}_{\mathcal{P}_{i+1}.i-1}, \mathcal{E}_j$);
25: **if**(r=DS ∨ r=CH ∨r=AT)
26: $\mathcal{I}^t_{\mathcal{P}_{i+1}.i-1} \leftarrow \mathcal{I}^t_{\mathcal{P}_{i+1}.i-1} \cup \{<j,r>\}$;
27: }) // End of task body.
28: ***Latch***.await();
29: $\mathcal{I} \leftarrow \overrightarrow{\bigcup} \mathcal{I}^t$; // Add the partial result in order.
30: **return** $\mathcal{I}$

3.4 Effectiveness-based parallel query primitives

3.4.1 XPath query rewriting

Query primitives represent the basic steps of XPath query, and various powerful XPath expressions are rewritten into execution sequences composed of multiple query primitives. In coPXQ, the primitives have been parallelized, which support the execution in data parallelism; while the query process is to return the

query result through looking up the relation index. coPXQ includes two types of primitives: non-filter primitives and filter primitives. Non-filter primitives are the implementation of axis operations corresponding to XPath, such as the primitive *ParaGetDescendant* for descendant axis, *ParaGetChild* for child axis, etc. Filter primitives are the implementation of predicate operations in XPath, including the basic filter primitive *ParaFilterInput1byInput2*, and several variants, such as filter primitive with AND condition, filter primitive with OR condition, and filter primitive with NOT condition. The XPath query expression needs to be translated into a query step composed of multiple parallel query primitives. The translation function is defined as $T[PExp]_{\mathcal{E}} = Exp$, where *PExp* is an XPath expression, *Exp* is an expression organized by parallel query primitives, and $\mathcal{E}$ is the input node sequence in current context. The main translation rules are as follows, where *e* in the rules denotes XPath expression.

- (R1) $T[//e]_{\mathcal{E}0} = T[e_{\text{tail}}]_{\mathcal{E}1}$ where $\mathcal{E}1 \leftarrow ParaGetDescendant(\mathcal{E}0, e_{\text{head}}, \ldots)$
- (R2) $T[/e]_{\mathcal{E}0} = T[e_{\text{tail}}]_{\mathcal{E}1}$ where $\mathcal{E}1 \leftarrow ParaGetChild\ (\mathcal{E}0, e_{\text{head}}, \ldots)$
- (R3) $T[[e]]_{\mathcal{E}0} = ParaFilterInput1byInput2(\mathcal{E}0, \mathcal{E}1)$ where $\mathcal{E}1 \leftarrow T[e]_{\mathcal{E}0}$
- (R4) $T[e1\ and\ e2]_{\mathcal{E}0} = ParaFilterInput1byInput2\_AND(\mathcal{E}0, \mathcal{E}1, \mathcal{E}2, \ldots)$ where $\{\mathcal{E}1 \leftarrow T[e1]_{\mathcal{E}0}, \mathcal{E}2 \leftarrow T[e2]_{\mathcal{E}0}\}$
- (R5) $T[e1\ or\ e2]_{\mathcal{E}0} = ParaFilterInput1byInput2\_OR(\mathcal{E}0, \mathcal{E}1, \mathcal{E}2, \ldots)$ where $\{\mathcal{E}1 \leftarrow T[e1]_{\mathcal{E}0}, \mathcal{E}2 \leftarrow T[e2]_{\mathcal{E}0}\}$
- (R6) $T[not(e)]_{\mathcal{E}0} = ParaFilterInput1byInput2\_NOT(\mathcal{E}0, \mathcal{E}1)$ where $\mathcal{E}1 \leftarrow T[e]_{\mathcal{E}0}$
- In rules (R1) and (R2), e_{head} denotes the beginning of expression *e*, corresponding to a tag name; e_{tail} denotes the remaining part of expression *e* after e_{head} is removed. For example, the XPath expression "*//A[/B or //C]*" is translated into the following result through $T[//A[/B\ or\ //C]]_{\mathcal{E}0}$,

$\mathcal{E}4 \leftarrow ParaFilterInput1byInput2\_OR(\mathcal{E}1, \mathcal{E}2, \mathcal{E}3, \ldots)$
where{ $\mathcal{E}2 \leftarrow ParaGetChild(\mathcal{E}1, \text{B}, \ldots)$, $\mathcal{E}3 \leftarrow ParaGetDescendant(\mathcal{E}1, \text{C}, \ldots)$}
where $\mathcal{E}1 \leftarrow ParaGetDescendant(\mathcal{E}0, \text{A}, \ldots)$

3.4.2 Parallel effectiveness estimation

In XPath query, the workload of some query steps may be very small. If this lightweight load is partitioned according to the number of available threads and all available threads are used for parallel processing, the overall parallel performance may be degraded due to the overhead of thread coordination. Therefore, when parallelizing query primitives, we need to consider not only load balancing but also parallel effectiveness. coPXQ introduces parallel effectiveness calculation as the basis for determining the number of worker threads. For each query step, the preferred number of worker threads is obtained through parallel effectiveness calculation, and then, the data are partitioned according to the number of threads for parallel processing. Since the parallel effectiveness of current query primitive is calculated based on the execution results of the previous query primitive, it is essentially a dynamic scheduling strategy.

Definition 3 (*Parallel effectiveness*) It is used to reflect the parallelization effect of query primitives under the constraint of the number of worker threads. Its formula is $G_{p/s} = C_{parallel}/C_{serial}$, where $C_{parallel}$ is the estimated parallel cost of the query primitive and C_{serial} is the estimated serial computational cost of the query primitive.

Considering the impact of the real parallel environment on the computational cost, the unit time of execution is normalized to facilitate the comparison and processing. The following four parameters are introduced: C_{inital}-thread initialization cost. The thread pool manages the working threads and allocates the threads once for each data block in each query step. $C_{barrier}$-the average communication cost of synchronization in block computing. Using barrier approach, each block has similar overhead. C_{check_f}-the cost of a node filter condition check in a filter primitive. C_{check_nf}-the cost of a node relation check in a non-filter primitive. Actually, the four parameters used in the following equations only need to know their relative values.

Under the condition of T parallel working threads, the estimated cost of parallel computing for a query primitive is as follows:

$$C_{parallel} = C_{inital} + \max_{t\in[0,T-1]}(C^t_{part}) + \sum_{t=0}^{T-1} C^t_{barrier}, \tag{4}$$

where C^t_{part} denotes the overhead of block calculation under T threads. Considering that the load balancing processing has been combined, there is $\max_{t\in[0,T-1]}(C^t_{part}) \approx C_{serial}/T$, where C_{serial} is the serial computational cost of the query primitive. According to Definition 3, the parallel effectiveness estimation of query primitive can be simplified as follows:

$$G_{p/s} = C_{parallel}/C_{serial} \approx 1/T + (C_{inital} + T \times C_{barrier})/C_{serial} \tag{5}$$

If $G_{p/s} \geq 1$, it means that serial processing is more reasonable, that is, $T=1$. The number of worker threads required is calculated by solving the following optimization problem:

$$\min(G_{p/s}) \quad \text{s.t.} \quad G_{p/s} < 1, \quad 1 < T \leq T_{available} \tag{6}$$

In other words, under the constraint of the number of available threads $T_{available}$, $G_{p/s}$ should be minimized to calculate the required number of worker threads T. To explain the problem intuitively, Eq. (5) is simplified to obtain the formula of G value as follows:

$$G = 1/T + T \times (C_{barrier}/C_{serial}) \tag{7}$$

The cc value is used to represent the ratio of barrier cost to serial computing cost, i.e., $cc = C_{barrier}/C_{serial}$. Figure 4 shows the calculating curve of G value under different cc values. It can be intuitively found in the figure that the number of threads corresponding to the lowest point of G value in each curve is the required number of worker threads. When the cc values are 0.01, 0.02, 0.1, 0.3 and 0.5, respectively, the corresponding number of threads is 10, 7, 3, 2 and 1.

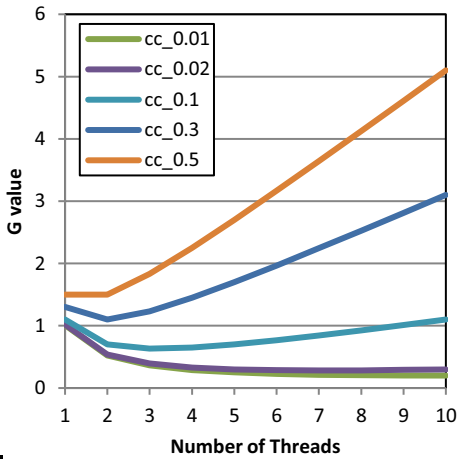


Fig. 4 Example of parallel effectiveness estimation

Algorithm 4. ParaGetDescendant($\mathcal{E}^{in}$,*tagName*, $\mathcal{E}$, $\mathcal{R}$, $\mathcal{I}$)

Input: $\mathcal{E}^{in}$ - input XML node list; *tagName* - tag name; $\mathcal{E}$ - XML node list; $\mathcal{R}$ - the relation number list of each node; $\mathcal{I}$ - the relation index of each node.
Output: $\mathcal{E}^{out}$ - the result node list.

1: Calculate T_{calc} according to Equation (6), k←0, Rc←0; // k is the ID of the block and is used for counting, k∈[0, T_{calc} -1]; Rc is the cumulative count of relations.
2: $C_t \leftarrow C_{non\_filter\_s} / T_{calc}$; // C_t is the load of each thread.
3: **foreach** node u in $\mathcal{E}^{in}$
4: Rc←Rc + $\mathcal{R}_u$;
5: **while**(Rc>=C_t) // Conditions for partitioning new blocks.
6: $\mathcal{P}_k$ .i← p_u , $\mathcal{P}_k$ .j← $\mathcal{R}_u$ -(Rc- C_t), k←k+1, Rc←Rc- C_t ; // p_u denotes the position of node u in list $\mathcal{E}^{in}$.
7: $\mathcal{E}^{out} \leftarrow \emptyset$;
8: ***Exec***←Executor(); //*Exec* is the thread pool manager.
9: ***Latch***←CountDownLatch(T_{calc});//*Latch* is the synchronous barrier counter.
10: **foreach** thread t∈{T | 0≤T≤T_{calc} -1} //t is the thread id.
11: $\mathcal{E}^{out\_t} \leftarrow \emptyset$; // $\mathcal{E}^{out\_t}$ is the partial result under thread t.
12: $\mathcal{E}^{out\_t}$ ←***Exec***.execute(***Task***(){ // Start of task body.
13: **if**(*tagName*="*") nameTest←true; // nameTest is true when tag name checking is not required
14: **else** nameTest←false;
15: **foreach** index s∈{ $\mathcal{I}_u$ |u= $\mathcal{E}_p^{in}$ .id ∧ *p*= $\mathcal{P}_t$ .i ∧ $\mathcal{I}_u$ .id≥ $\mathcal{P}_t$ .j}
16: **if**((nameTest=true ∨ $\mathcal{E}_{s.id}$ .tagName=tagName)∧(s.r=DE ∨ s.r=CH))
17: $\mathcal{E}^{out\_t} \leftarrow \mathcal{E}^{out\_t} \cup \{\mathcal{E}_{s.id}\}$;
18: **foreach** node u∈{ $\mathcal{E}_M^{in}$ | $\mathcal{P}_t$ .i<M< $\mathcal{P}_{t+1}$ .i-1}
19: **foreach** index s∈{ $\mathcal{I}_u$ }
20: **if**((nameTest=true ∨ $\mathcal{E}_{s.id}$ .tagName=tagName)∧(s.r=DE ∨ s.r=CH))
21: $\mathcal{E}^{out\_t} \leftarrow \mathcal{E}^{out\_t} \cup \{\mathcal{E}_{s.id}\}$;
22: **foreach** index s∈{ $\mathcal{I}_u$ |u= $\mathcal{E}_p^{in}$ .id ∧ *p*= $\mathcal{P}_{t+1}$ .i-1 ∧ $\mathcal{I}_u$ .id≤ $\mathcal{P}_{t+1}$ .j-1}
23: **if**((nameTest=true ∨ $\mathcal{E}_{s.id}$ .tagName=tagName)∧(s.r=DE ∨ s.r=CH))
24: $\mathcal{E}^{out\_t} \leftarrow \mathcal{E}^{out\_t} \cup \{\mathcal{E}_{s.id}\}$;
25: }) // End of task body.
26: ***Latch***.await();
27: $\mathcal{E}^{out} \leftarrow \vec{\bigcup} \mathcal{E}^{out\_t}$; // Add the partial result in order.
28: **return** $\mathcal{E}^{out}$;

3.4.3 Non-filter primitives

There are two aspects that affect the performance of parallel query primitives: one is the load balancing, and the data blocks for data parallelism need to be well balanced to avoid too much synchronization waiting; the other is the parallel effectiveness. When the computational cost is relatively small, more worker threads mean more synchronization overhead, which results in low efficiency. In the process of parallel primitives, the number of worker threads required is obtained through the parallel effectiveness calculation, then the input data are partitioned into blocks according to the number of threads to realize the load balancing, and finally the blocks is processed in data parallelism. During the evaluation of non-filter primitives, the relation between nodes is checked according to the relation index, and the computational cost is related to the number of relations per node. The computational cost of each node is estimated as $C_u = \mathcal{R}_u \times C_{\text{check\_nf}}$. The serial computational cost of each primitive is the sum of all input nodes, so the cost estimation equation is as follows, where input is the sequence of input XML nodes.

$$C_{\text{non\_filter}} = \sum_{u \in \text{input}} C_u \tag{8}$$

Since only the relative cost values of blocks is needed to be known when data are partitioned in the same primitive, $C_{\text{check\_nf}}$ can be ignored and the cost value is directly represented by the number of node relations. So Eq. (8) is simplified as follows:

$$C_{\text{non\_filter\_s}} = \sum_{u \in \text{input}} \mathcal{R}_u \tag{9}$$

Getting descendant nodes is a typical case of non-filter primitives. The process is shown in Algorithm 4. Similar to Algorithm 3, Algorithm 4 also consists of two stages. Firstly, the data are partitioned into blocks according to cost estimation (line 1–6), and then the blocks are processed in data parallelism to get the result (line 7–27). Line 1 uses Eq. (6) to calculate the number of required worker threads T_{calc}, where C_{serial} is the $C_{\text{non\_filter}}$ in Eq. (8). Line 2 uses Eq. (9) to estimate the computational cost, and partition the total relations of all input nodes as the load of each thread. Line 5–6 are used to calculate the block boundary, $\mathcal{P}_k$ is used to record the boundary information of block k, and its i component records the position of the node u in the input node sequence $\mathcal{E}^{in}$, and its j component records the position of the node u in the index $\mathcal{I}_u$. According to Corollary 2, the node relation storage is a continuous sequence in the order of node ID; therefore, the corresponding node ID of nodes on the boundary can be deduced when partitioning by the cumulative number of relations. Similar to Algorithm 3, the nodes in different positions of the block need to be processed differently. Line 15–17 are used to process the first node of the block; line 18–21 are used to process the intermediate nodes of the block; and line 22–24 are used to process the last node of the block.

3.4.4 Filter primitives

The parallelization algorithm of filter primitives has the similar process as filter primitives, which includes two stages: partitioning input data into blocks according to cost estimation and parallel processing of blocks. In the stage of data partition, the number of worker threads is obtained by parallel effectiveness calculation, and then each block boundary is calculated according to the number of threads. In the evaluation stage, all the nodes in the second input node sequence that can meet the conditions are filtered out according to all the relation indexes of the first input node sequence. The second sequence is sorted by the node ID, and the binary search method is used for fast positioning, so the scanning complexity is $\log_2(N_{\text{input2}})$, where N_{input2} is the number of input nodes. In this way, the computational cost of each input node is

$$C_u = \mathcal{R}_u \times \log_2(N_{\text{input2}}) \times C_{\text{check\_f}}, \tag{10}$$

where $\mathcal{R}_u$ is the relation number of a node u in the first sequence. For the sequence *input1*, the serial computational cost of filter primitives is $C_{\text{filter}} = \sum_{u \in \text{input1}} C_u$. Similar to non-filter primitives, $C_{\text{check\_f}}$ can be ignored when estimating the cost for data partitioning, thus C_{filter} is simplified to

$$C_{\text{filter\_s}} = \sum_{u \in \text{input1}} \mathcal{R}_u \times \log_2(N_{\text{input2}}). \tag{11}$$

The function prototype of basic parallel filter primitive is *ParaFilterInput1byInput2*($\mathcal{E}^{\text{in1}}$,$\mathcal{E}^{\text{in2}}$,$\mathcal{R}$,$\mathcal{I}$), where the input parameters include as follows: $\mathcal{E}^{\text{in1}}$-the first input node sequence, $\mathcal{E}^{\text{in2}}$-the second input node sequence, $\mathcal{R}$-the relation number list of each node; $\mathcal{I}$-the relation index of each node; the return result is node sequence $\mathcal{E}^{\text{out}}$. The description of the algorithm is omitted.

4 Experiments

4.1 Experimental settings

We use cases from different test platforms to conduct experiments. The test data of the first part are from the Treebank project [37], which provides an XML document of about 82 MB. The document is a deep recursive XML data set. The maximum depth of the document tree is 36, and the average depth is 7.8. Four typical test cases covering common query semantics of XPath are used in Table 3. Among them, T1 is a simple path query; T2 is a query with juxtaposed predicates and an axis operation with wildcards; T3 is a query with nested predicates, and the predicate contains more than one query step; T4 has a logical expression inside the predicate.

The test data of the second part are from XMark [38], which is a general test platform, and can generate XML documents of any size. For comparison, we use

the tools provided by XMark to generate an XML document about 82 MB in size to Treebank for testing the four typical XPath queries in Table 3. X1 is a simple path query; X2 query has a predicate; X3 has a logical expression inside the predicate; X4 query contains nested predicates with an attribute axis operation.

The evaluation indicators are the execution time of each method under different query cases and different thread conditions, and the speedups [39] are calculated according to the following formula:

$$S_{\mathrm{p}} = \frac{T_{\mathrm{s}}}{T_{\mathrm{p}}}, \tag{12}$$

where T_{s} represents the serial execution time under the condition of single thread; T_{p} represents the parallel execution time when the number of threads is p. The experiment is carried out on a PC equipped with AMD fx-8320 eight core CPU (3.5 GHz) and 8 GB physical memory. The software environment is JDK1.8 and Windows 7 (SP1) operating system.

4.2 Comparative experiments

We use our method to compare with the classical parallelization method of navigational XPath query [11, 32] named pNav, the pM2 method proposed in [12], as well as the PXQ method proposed in [19]. In addition, in order to expand the scope of investigation, we also choose a typical parallel twig evaluation method named PTS [13] for comparison. For the convenience of comparison, all the methods involved in the comparison use the region encoding of XML parsing results. Considering the differences of the implementation of each method, the execution time test is limited to the specific evaluation steps of each method and does not include the common time-consuming part of XML parsing. For pNav, pM2, PXQ and coPXQ methods, both index creation time and query execution time are included. In particular, the execution time of coPXQ includes the time required for obtaining XML statistics. The execution time of PTS method includes the overhead of specific operations,

Table 3 XPath query cases

Case	XPath expression	Number of results
T1	//S//NP/PP/NN	15
T2	//S[./VBP][.//NP/VP]//PP[.//IN]/*//VBN	174
T3	//EMPTY[.//VP/PP//NNP][.//S[.//PP//JJ]//VBN]//PP/ NP//_ NONE_	1589
T4	//EMPTY[./_PERIOD_ or./S[./VBP]//TO]	37,320
X1	//open_auctions/open_auction//time	42,915
X2	//regions/asia/item[./payment]//name	1440
X3	//people/person[.//emailaddress and.//creditcard]	9179
X4	//categories[./category[./name]/@id]//description	720

such as the time of constructing pattern tree and label stream, and the time of subtree partition in the process of parallelization.

When pNav method is tested, three partition strategies [31] are integrated and the best parallel plan is manually selected for testing. Because the overhead of automatic cost estimation and selection of parallel plan is not included, the low bound of pNav execution time is obtained. For example, the data partition strategy is applied to case T2, the results obtained after the serial execution of "//S" are partitioned, and the remaining queries are performed in data parallelism. The query partition strategy is applied to case T4, the sub-query "//EMPTY./_PERIOD_" and "//EMPTY/S[./VBP]//TO" are evaluated in parallel, and then the predicate operation of OR is performed. For case X1, first execute "//open_auctions/open_auction" serially, then the result is partitioned into blocks, and then "//time" is processed in parallel on each block. Case X3 uses a hybrid partition strategy. First, "//people/person" is executed serially, and then the result is partitioned into blocks. The sub-query "//emailaddress" and "//creditcard" is executed, respectively, on each block in parallel. Finally, the results are merged.

The idea of PTS method is to evaluate each XML subtree in parallel using twig method. According to the basic idea of PTS, the XML tree is partitioned and load balanced manually, and the final tree is constructed for each partition. Then, twig pattern evaluation is performed in parallel on each final tree, and the results are merged finally. Since the operation of automatic partition is omitted, the execution time of low bound is obtained. Since the twig method cannot directly support predicates with logical operations, case T4 is decomposed into sub-queries like pNav; while case X3 is rewritten as "//people/person[.//emailaddress] [.//creditcard]".

PXQ method uses pipeline to process XML data stream. At present, it does not support OR logic operation, so it cannot be used to test case T4.

The results of comparative experiments on Treebank are shown in Figs. 5 and 6. Figure 5 shows the execution time obtained by the test; Fig. 6 shows the speedup obtained by calculation. As shown in Fig. 5, in the case of single thread, PTS has the most execution time, while the execution time of the other methods is relatively close. The reason is that except PTS, the navigational methods utilize relation indexing facility. Due to the optimized index, coPXQ can reduce the scanning of index during query, so it is more efficient; however, due to the extra cost of acquiring XML statistics, the time-saving effect is offset to some extent. Under the condition of multithreading, the execution time of coPXQ is less than that of the other methods, and with the increase of threads, the advantage of coPXQ is more obvious. For example, for the more complicated case T3, coPXQ is 38%, 13%, 10% and 7% faster than PTS, pNav, pM2 and PXQ on 2 threads; 43%, 18%, 14% and 13% faster on 4 threads; and 44%, 19%, 18% and 15% faster on 8 threads, respectively. In terms of speedup, the speedup of coPXQ is 3%, 11%, 10% and 8% higher than PTS, pNav, pM2 and PXQ on 2 threads; 6%, 16%, 14% and 14% on 4 threads and 7%, 17%, 17% and 16% on 8 threads, respectively. From Fig. 6, it is found that in each query, with the increase of threads, the speedup of coPXQ increases, while pNav and pM2 sometimes decrease when there are more threads. For example, in case T2, the speedup of pM2 on 8 threads is lower than that on 4 threads; similarly, in case T4, the speedup of pNav on 8 threads is lower than that on 4 threads. The reason is

that the two methods do not consider parallel effectiveness. The increase of worker threads brings more synchronization overhead, which offsets the time-saving effect obtained by parallelism.

The experimental results on XMark platform are shown in Figs. 7 and 8. As can be seen from Fig. 7, under the condition of multithreading, the execution time of coPXQ is less than that of the other methods, which is similar to Fig. 5, which shows that the method has the speed advantage in parallel environment. In terms of speedup, the speedup of coPXQ increases with the increase of threads. However, the speedup on 8-threads is not significantly higher than that on 4 threads. The reason is that the query cost of XMark is relatively low; thus, the parallel effect of query primitives is not significant.

In general, the efficiency of PTS is relatively low, mainly because the optimization measures of relation index are not suitable for twig algorithm. Among the navigational evaluation methods, pNav is relatively inefficient, because input data in pNav are partitioned according to the number of XML nodes, which is easy to cause load imbalance. Moreover, the partitioned data are often processed

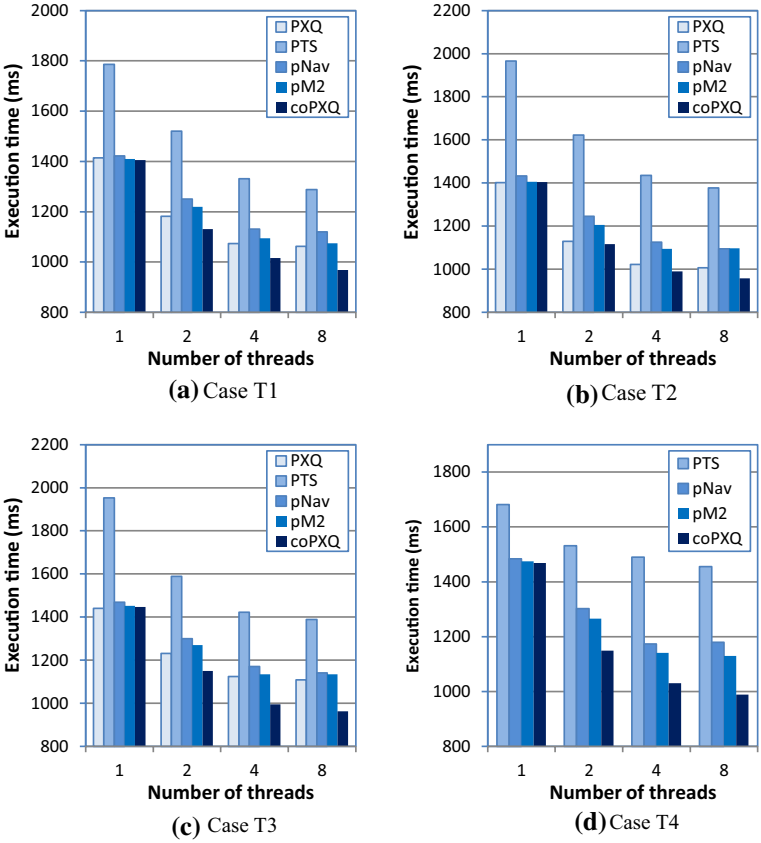


Fig. 5 Execution time comparison on Treebank

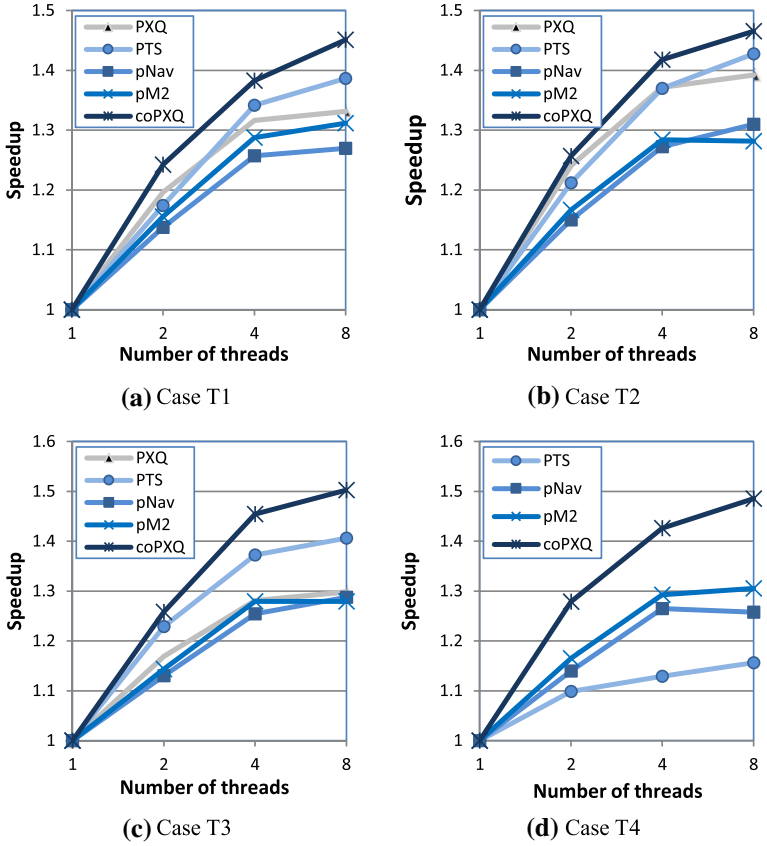


Fig. 6 Speedup comparison on Treebank

by multiple query steps. If the input is unbalanced, the subsequent load imbalance will be more obvious. pM2 is similar to pNav in that the data are partitioned according to the number of XML nodes. The difference is that pM2 can perform parallel processing for each query step, in addition to having more opportunities for parallelization, the possibility of load imbalance caused by small step calculation is lower than that of pNav. The cost estimation of coPXQ method is based on the relation number, which makes the estimation result more accurate. The load balancing method based on the estimation can better overcome the imbalance of load; moreover, the number of worker threads is obtained by parallel effectiveness estimation, which further avoids the inefficient parallelism. The efficiency of PXQ is close to that of coPXQ. However, since coPXQ adopts data parallelism and optimizes index processing, the performance of coPXQ is superior on the whole. In addition, PXQ does not support OR logic operation, and the construction of pipeline phase is complicated under complex query conditions, which is easy to cause performance degradation.

4.3 Evaluation of cost estimation and parallel effectiveness

4.3.1 Evaluation of cost estimation

In order to explore the improvement effect of load balancing brought by the cost estimation method based on relation number proposed in this paper, we carry out an experimental comparison with the estimation method based on node number. We first obtain the processing time of each block in the parallel index creation process of Treebank and XMark data sets, and then complete the imbalance calculation. The results are shown in Table 4.

The experimental condition is to partition the data set into 8 data blocks for parallel processing under the condition of 8 worker threads. The processing time of data block reflects the computational cost. The imbalance σ_{p} [40] in Table 4 is calculated according to the following formula:

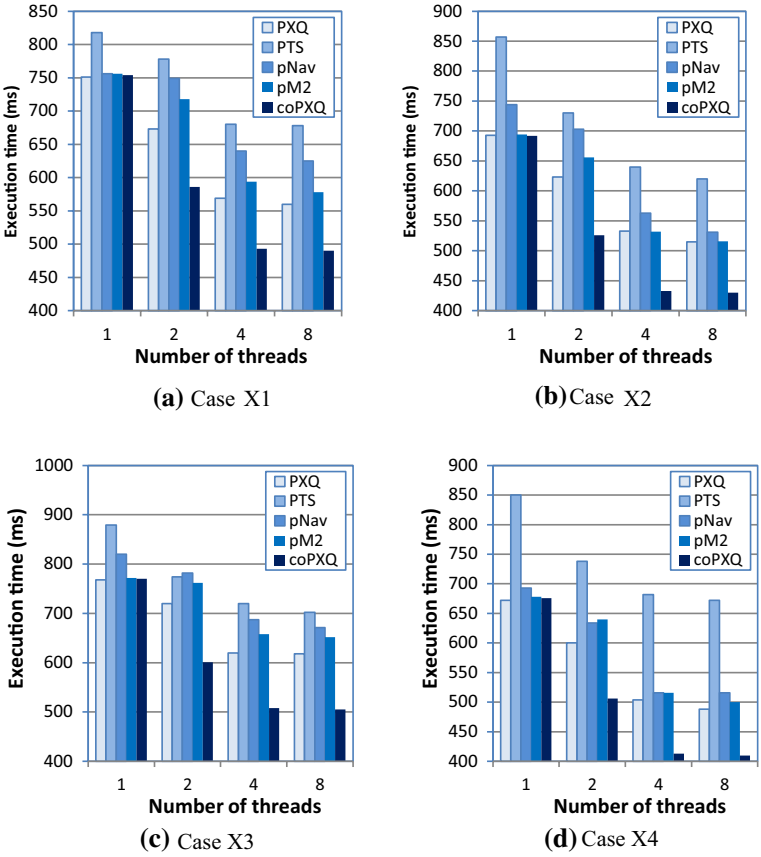


Fig. 7 Execution time comparison on XMark

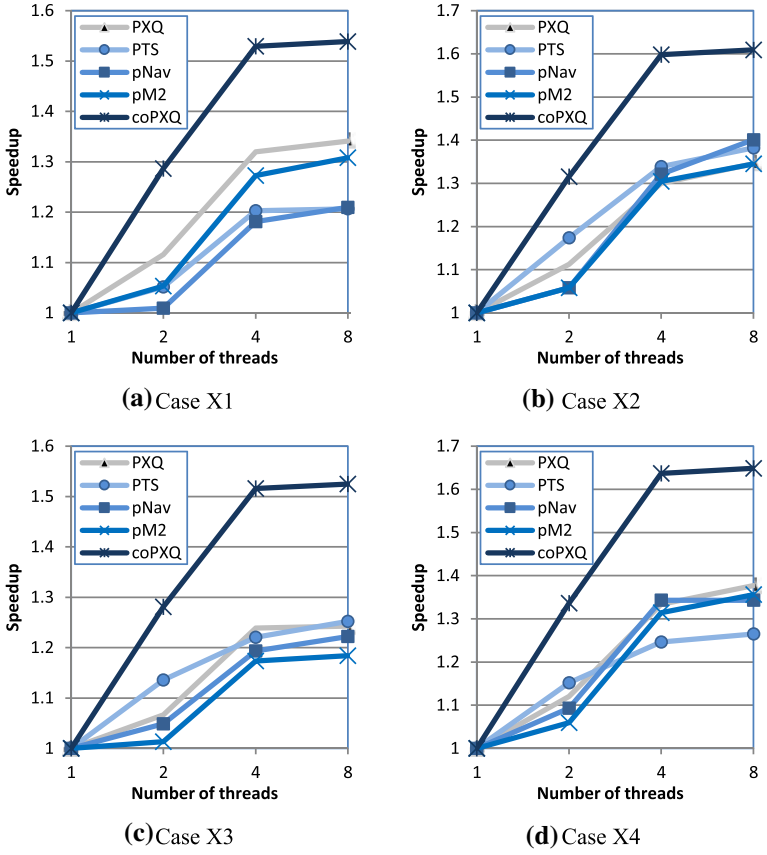


Fig. 8 Speedup comparison on XMark

$$\sigma_{\mathrm{P}} = \frac{1}{T_{\mathrm{avg}}} \sqrt{\frac{1}{P} \sum_{i=1}^{P} (T_i - T_{\mathrm{avg}})^2} \tag{13}$$

The average processing time of each block is $T_{\mathrm{avg}} = (\sum_{i=1}^{P} T_i)/P$, where P denotes the number of blocks, and T_i denotes the processing time of each block. The smaller the imbalance value, the more balanced the load. Comparing the imbalance values, it can be seen that the cost estimation method based on relation number can obtain better balance effect. In addition, the effect of the new cost estimation method is also reflected in the improvement of the efficiency of index parallel creation. For Treebank data, load balancing is guided by the old cost estimation method and the new cost estimation method, respectively. As a result, the total processing time of parallel index creation is 720 ms and 608 ms, respectively. For XMark data, it is 422 ms and 340 ms, respectively.

4.3.2 Evaluation of parallel effectiveness

In the parallel process of coPXQ query primitive, a reasonable number of worker threads is configured to ensure the parallel effectiveness. In order to investigate the effect of introducing parallel effectiveness estimation, we set two experimental conditions to obtain the query execution time comparison of each case in Table 3 before and after parallel effectiveness processing. The purpose of selecting query execution time for comparison is to eliminate other influencing factors and focus on the effect of parallel effectiveness processing. Condition 1: parallel effectiveness estimation is not introduced, and each query step is processed in parallel according to all available threads. Condition 2: each query step is processed in parallel according to the number of threads obtained from the parallel effectiveness estimation. We set the number of available worker threads in the test environment to 8. The comparison of results under the two conditions is shown in Fig. 9. It can be found that the query speed has been improved after parallel effectiveness processing.

Now give some examples to illustrate the results of parallel effectiveness processing. For case T2, there are 11 query steps. The number of threads required for the first four query steps obtained by calculation is 4, 8, 4 and 2, respectively. The number of threads required for the other query steps is 1 due to the low cost. For case X2, there are 6 query steps. The number of threads required for the first two query steps is 6 and 4, respectively, while that in the other steps is 1.

5 Conclusion and future work

Parallel XPath query technology for multi-core computing provides powerful support for high-performance XML data processing. However, due to the semi-structured characteristics of XML data and the complexity of XPath query, it is often

Table 4 Imbalance under different cost estimation methods

Block Id	TreeBank		XMark	
	Nodes number-based (ms)	Relations number-based (ms)	Nodes number-based (ms)	Relations number-based (ms)
1	675	593	401	316
2	468	546	218	301
3	593	468	234	203
4	593	531	186	265
5	639	515	265	218
6	561	452	202	211
7	561	484	202	206
8	531	515	234	220
σ_P	0.103	0.083	0.264	0.174

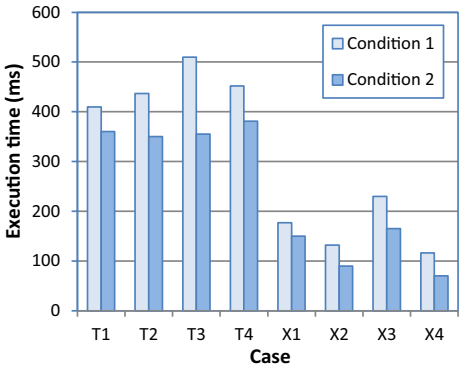


Fig. 9 Comparison of query time before and after parallel effectiveness processing

difficult to obtain desired parallelization effect. The existing problems include load imbalance and thread inefficiency, which limit the performance of parallelization. The coPXQ method proposed in this paper is a parallel navigational XPath query method based on cost optimization. In order to avoid load imbalance caused by inaccurate cost estimation, coPXQ utilizes a new cost estimation method based on relation number to balance the load and optimize the parallel creation of relation index and the parallel execution of primitives. In order to avoid the situation that the synchronization overhead exceeds the parallel benefit caused by the abuse of threads, coPXQ ensures the effective use of threads in query through the strategy of determining the number of worker threads based on parallel effectiveness estimation. In addition, coPXQ optimizes the storage of relation index to further improve the overall performance of XPath query. Compared with the existing typical methods, the test results show that coPXQ can achieve better parallel performance.

The future work is to explore the complete XPath semantic support, such as reverse axis, sibling axis and complex predicate evaluation, optimize the design of various query primitives, and improve the navigational evaluation performance for multi-core computing environment. Furthermore, we will consider the combination with XQuery [41] and integrate parallel XPath evaluation into general XML query application through path extraction and automatic parallelization technology.

Acknowledgements This research was supported by the Natural Science Foundation of Fujian Province of China (2018J01538, 2020J01697), the Science Foundation of Jimei University (ZQ2014003), and Open Fund of Digital Fujian Big Data Modeling and Intelligent Computing Institute.

References

1. Buneman P (1997) Semistructured data. In: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems. ACM, pp 117–121
2. Robie J, Dyck M, Spiegel J (2017) XML path language (XPath). https://www.w3.org/TR/xpath/
3. Bruno N, Koudas N, Srivastava D (2002) Holistic twig joins: optimal XML pattern matching. In: the 2002 ACM SIGMOD International Conference on Management of Data, Wisconsin, USA, 2002. ACM, pp 310–321

4. Cate BT, Marx M (2007) Navigational XPath: calculus and algebra. ACM SIGMOD Rec 36(2):19–26
5. Grün C, Worteler L, Kircher L, Shadura R (2018) BaseX: the XML framework https://basex.org/
6. Meier W (2019) EXist-db Project https://github.com/exist-db/exist
7. Franc X (2019) Qizxopen http://www.axyana.com/qizxopen
8. Shah B, Rao P, Moon B, Rajagopalan M (2009) A data parallel algorithm for XML DOM parsing. In: Database and XML technologies, pp 75–90
9. Pan Y, Lu W, Zhang Y, Chili K (2007) A static load-balancing scheme for parallel XML parsing on multicore CPUs. In: Seventh IEEE international symposium on cluster computing and the grid (CCGRID 2007). IEEE, pp 351–362
10. Machdi I, Amagasa T, Kitagawa H (2010) Parallel holistic twig joins on a multi-core system. Int J Web Inf Syst 6(2):149–177
11. Bordawekar R, Lim L, Shmueli O (2009) Parallelization of XPath queries using multi-core processors. In: International Conference on Extending Database Technology: Advances in Database Technology (EDBT2009), pp 180–191
12. Chen R, Liao H, Wang Z (2013) Parallel XPath evaluation based on node relation matrix. J Comput Inf Syst 9(19):7583–7592
13. Shnaiderman L, Shmueli O (2015) Multi-core processing of XML twig patterns. iEEE Trans Knowl Data Eng 27(4):1057–1070
14. Chen R, Liao H, Wang Z, Su H (2016) Automatic parallelization of XQuery programs on multi-core systems. J Supercomput 72(4):1517–1548
15. Miao H, Nie T, Yue D, Zhang T, Liu J (2012) Algebra for parallel XQuery processing. Web Age Inf Manag 2012:1–10
16. Kim SH, Lee KH, Lee YJ (2016) Multi-query processing of XML data streams on multicore. J Supercomput 73(6):1–30
17. Jiang L, Zhao Z (2017) Grammar-aware parallelization for scalable XPath querying. In: the 22nd ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP '17),2017. ACM, pp 371–383
18. Karsin B, Casanova H, Lim L (2017) Low-latency XPath query evaluation on multi-core processors. In: Hawaii International Conference on System Sciences, 2017, pp 6222–6231
19. Chen R, Wang Z, Hong Y (2021) Hong Y (2021) Pipelined XPath query based on cost optimization. Sci Program 19:1–16
20. Huang X, Si X, Yuan X, Wang C (2014) A dynamic load-balancing scheme for XPath queries parallelization in shared memory multi-core systems. J Comput 9:6
21. Moussalli R, Halstead R, Salloum M, Najjar WA, Tsotras VJ (2011) Efficient XML path filtering using GPUs. In: International workshop on accelerating data management systems using modern processor and storage architectures (ADMS 2011), Seattle, WA, USA
22. Kim S, Lee Y, Lee JJ (2015) Matrix-based XML stream processing using a GPU. In: IEEE international congress on big data
23. Sampson J, Gonzalez R (2006) Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In: The 39th annual IEEE/ACM international symposium on microarchitecture, Orlando, USA, 2006. pp 235–246
24. Willebeek-Lemair MH, Reeves AP (1993) Strategies for dynamic load balancing on highly parallel computers. IEEE Trans Parallel Distrib Syst 4(9):979–993
25. Weissman JB (2002) Predicting the cost and benefit of adapting data parallel applications in clusters. J Parallel Distrib Comput 62(8):1248–1271
26. Zuo W, Chen Y, He F, Chen K (2011) Load balancing parallelizing XML query processing based on shared cache chip multi-processor (CMP). Sci Res Essays 6(18):3914–3926
27. Subramaniam S, Haw SC, Soon LK (2021) Improved centralized XML query processing using distributed query workload. IEEE Access 9:29127–29142
28. Zhang C, Naughton J, DeWitt D, Luo Q, Lohman G (2001) On supporting containment queries in relational database management systems. In: ACM SIGMOD record, 2001, vol 2. ACM, pp 425–436
29. Sestakova E, Janousek J (2018) Automata approach to XML data indexing. Information 9(1):12
30. Widemann BT, Lepper M (2019) Simple and effective relation-based approaches to XPath and XSLT type checking. Technical Report, Bad Honnef (2015)

31. Bordawekar R, Lim L, Kementsietsidis A (2010) Statistics-based parallelization of XPath queries in shared memory. In: The 13th International Conference on Extending Database Technology (EDBT), 2010. ACM
32. Sato S, Hao W, Matsuzaki K (2018) Parallelization of XPath queries using modern XQuery processors. In: New Trends in Databases and Information Systems. ADBIS 2018
33. Hartmann S, Ma H, Schewe KD (2007) Cost-based vertical fragmentation for XML. In: al. KCCe (ed) APWeb/WAIM 2007. Springer, Berlin, Heidelberg, pp 12–24
34. Georgiadis H, Charalambides M, Vassalos V (2010) Efficient physical operators for cost-based XPath execution. In: Paper presented at the EDBT 2010
35. Hidaka S, Kato H, Yoshikawa M (2007) A relative cost model for XQuery. In: Proceedings of the 2007 ACM symposium on Applied computing, 2007. ACM, pp 1332–1333
36. Herlihy M, Shavit N (2008) The art of multiprocessor programming. Morgan Kaufmann, New York
37. University of Pennsylvania Treebank Project (2002) http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/data/reebank/treebank_e.xml
38. Schmidt A, Waas F, Kersten M, Carey MJ, Manolescu I, Busse R (2002) XMark: a benchmark for XML data management. In: Proceedings of the 28th International Conference on Very Large Data Bases, 2002. VLDB Endowment, pp 974–985
39. Wilkinson B, Allen M (2005) Parallel programming: techniques and applications using networked workstations and parallel computers. 2nd edn, Pearson Education
40. Linford JC, Hermanns M-A, Geimer M, Boehme D, Wolf F (2008) Detecting load imbalance in massively parallel applications. Technical Report FZJ-JSC-IB-2008–09. Forschungszentrum Julich
41. Robie J, Dyck M, Spiegel J (2017) XQuery 3.1: an XML query language. https://www.w3.org/TR/xquery

Authors and Affiliations

Rongxin Chen[1,2] · Zhijin Wang[1] · Hang Su[3] · Shutong Xie[1] · Zongyue Wang[1]

Zhijin Wang
zhijin@jmu.edu.cn

Hang Su
suhang@bjut.edu.cn

Shutong Xie
15186307@qq.com

Zongyue Wang
wangzongyue@jmu.edu.cn

[1] Computer Engineering College, Jimei University, Xiamen, China

[2] Digital Fujian Big Data Modeling and Intelligent Computing Institute, Xiamen, China

[3] College of Computer Science, Beijing University of Technology, Beijing, China