Abstract—Since various XML query applications have come to the fore recently, performance optimization becomes the research hotspot. With the popularity of multi-core computing condition, parallelization appears as an important optimization measure. The paper presents a parallel solution to XML query application through the combination of parallel XML parsing and parallel XML query. The XML parsing is based on arbitrary XML data partition and parallel sub-tree construction with the final merging procedure. After XML parsing, the region encodings of XML data are obtained for relation matrix construction in that the XPath evaluation in query procedure is based on relation matrix. The matrix construction procedure and query primitives are parallelized to boost performance. As a whole, our solution makes use of multi-core environment through parallelization of key execution stages in query process. The key execution stages are verified by experiment and the whole effect of the solution is presented.

Keywords- XML query; XML parsing; XPath evaluation; parallelization; multi-core

I. INTRODUCTION

A XML has been widely accepted as a data format standard for information exchange and storage with the rapid development of Internet and web services. Query is the most popular function in XML application. XQuery[1] has become the recommendation standard query language while XPath[2] is the key component of XQuery. Since XML data growing rapidly and query applications expanding fast in many scenarios. Various optimization measures are proposed to tackle the performance problem of query. Some measures focus on query language optimization such as designing optimized algebra like XAT[3], GTP[4] and so on. Other measures work on XPath evaluation, typically, adopting pattern match method to optimize query. Thereby many twig query methods are developed such as TwigStack[5], Twig^Stack[6], TwigList[7] and so on. Recently because of the popularity of multi-core computing condition, parallel optimization begins to attract attentions [8].

XML query application includes XML parsing and XML query procedures. Since XML parsing is CPU intensive procedure especial for big dataset, parallelization in parsing should be considered primarily. XML is semi-structured and XPath evaluation plays the key role in XML query operation, thus schema to facilitate parallelism in XPath evaluation would make for the improvement of XML query performance. Our parallel solution covered the two major procedures:

- XML parsing based on sub-tree construction in parallel. The schema supports arbitrary XML data partition.
- XML query procedure adopts an XPath evaluation approach based on relation matrix match. Thus the XPath evaluation semantics can be well preserved and parallelization opportunities revealed.

The solution focuses on parallelization of the key execution stages which occupy most run time in the whole process in order to ensure the parallelization effects with acceptable cost. The rest of paper is organized as follows. In section II and III, the parallelization methods for both XML parsing and XML query procedures are introduced. In section IV, experiment on verification of key execution procedures and the whole query process effect are presented. In section V conclusions are drawn and the future works are pointed out.

II. PARALLEL XML PARSING

Because XML data is semi-structured and they can’t be fully accessed directly, the extracting of relations and contents of every item from XML data has to rely on parsing. Since XML parsing is a time-consuming task and becomes a key obstacle to a successful XML deployment [9], the optimal design has been a hot topic recently. Much important study on parallel XML parsing has been done by Wei Lu et al. A DOM style parallel parsing method [10] was presented which includes two main process stages named pre-parsing and full-parsing. The purpose of pre-parsing is to generate the skeleton of XML data to guide the following parallel full-parsing tasks. Because pre-parsing stage is often high time-consuming, some optimal designs had been considered. Parallelism methods for skeleton generation, such as Meta-DFA [11] and simultaneous transducers [12], are brought up. However, parallel measures for pre-parsing obviously increase the difficulty of implement.

Our parsing method supports arbitrary segment parsing. That means XML data can be arbitrarily partitioned to segments and parsed independently so that the parsing procedures can be performed in parallel. Because we adopt a light-weighted XML partition method which can avoid the need of skeleton generation, original XML data are scanned only once in general. In consideration of the requirement of the follow-up query procedure, the main purpose of XML parsing is to obtain the region encodings [13] of XML data.

A Parallel Solution to XML Query Application

Rongxin Chen
Computer Engineering College
Jimei University
Xiamen, China
e-mail: ch2002star@163.com

Weibin Chen
Computer Science College
Huaqiao University
Quanzhou, China
e-mail: chenwb@hqu.edu.cn
There are three major process steps in XML parsing as shown in figure 1. In step 1 XML data will be properly partitioned. The parallel tasks of sub-trees construction will proceed in step 2. Then the whole XML tree will be generated in sub-trees mergence in stage 3.

A. Data Partition

The first step in XML parsing is to determine the boundaries of segments for parallel tasks. Considering the instance in figure 2, three segments will be determined according to the arbitrarily given character positions 0, 45, 194 and file end. The solid fold-lines indicate the initial partition positions. After mismatched tag string adjustment, the actual partition positions are 0, 54, 197 and file end as the dotted lines shows.

<table>
<thead>
<tr>
<th>Step1</th>
<th>Step2</th>
<th>Step3</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Diagram" /></td>
<td><img src="image2.png" alt="Diagram" /></td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Figure 1. Parallel XML parsing framework

B. Parallel Sub-trees Construction

In this step, the partitioned XML segments are parsed in parallel to construct sub-trees and relative region encodings are obtained. Tag name stacks in segments are maintained to detect the mismatched tags and root nodes of sub-trees.

Algorithm 1 describes the sub-tree constructing framework. Given an arbitrarily partitioned segment as parameter, it will build all the sub-trees. The main idea is to parse tag name continuously till the end of segment and complete all the sub-trees construction within segment. Line 3 is to get a completed tag name string and its position. Here ParseTagName (line 6) carry out tag name parsing to obtain sub-tree information including region encodings of nodes. Uncompleted end tags in formation will be saved for merging process at the end of construction (line 7).

```
Algorithm 1 GetSubTreeRegionCode(segment)
1: pos ← segment.startPos;
2: while (pos < segment.endPos)
3:   tmp ← GetCompletedTag(segment, pos, segment.endPos);
4:   pos ← tmp.pos;
5:   completedTagString ← tmp.TagString;
6:   ParseTagName(completedTagString, subTrees);
7: ProcessEndUnmatched();
```

A data parallelism framework is described in algorithm 2. The input parameters are partitioned dataset and pre-defined task. Line 1 creates a counter for task synchronization purpose. The counter takes the number of data segments as parameter. Line 2 creates a multi-threaded executing service. Line 5 makes sure the synchronization of all tasks. Finally the executing service is properly shut downed in line 6. The parallel sub-trees construction procedure can be described as ParallelProcess(seg[], GetSubTreeRegionCodeTask). Here seg[] stands for the pre-defined XML data segments and GetSubTreeRegionCodeTask is defined as the wrapped sub-tree construction task which invokes the function as presented in algorithm 1.
C. Sub-trees Mergence

There’re two major works in mergence step, one is node relations adjustment, the other is node content merging and node tag name merging. Since the XML segments parsed independently and only local information is obtain, node relations need further adjustment to reflect the global situation. The adjustment works include: to add the offset of segment to the pointer information in every node; to find parent node of the sub-tree root node, then set the pointer of its first child pointer as this root node’s position; to set the level value of region encodings and to set the real end location of the truncated nodes. Node content merging is realized by combining content mapping tables from different segments into a global table. Tag name merging needs reconstructing the global tag name hash table by importing the tag name information from every segment.

III. Parallel Query

The performance of XPath evaluation greatly affects XML query applications. The parallelization of XML query in our solution is realized through an XPath evaluation method based on node relation matrix. Relation matrix is constructed from XML region encodings, while XPath evaluation is realized by execution of query primitive sequence. The navigation feature of our method tends to comply with XPath semantics well and support the implementation of reversed axis query and branching query expressed in predicates. Furthermore, the general processes in construction of matrix and query on matrix are looping style so that it is easy to be parallelized. The parallel query process includes two main steps: parallel relation matrix construction and parallel query.

A. Parallel relation matrix construction

Let \( R [M \rightarrow N] \) represents the relation calculation from node \( M \) to \( N \); \( RTYE \) stands for relation value, XPath evaluation semantics include the following main relation calculations:

- ancestor/descendant(AD): for \( M/N, R[M \rightarrow N] = RTYE.DE=1 \) means \( N \) is \( M \)'s descendant; on the contrary, \( R[N \rightarrow M] = RTYE.AN=1 \) means \( M \) is \( N \)'s ancestor.
- parent/child(PC): for \( M/N, R[M \rightarrow N] = RTYE.CH=2 \) means \( N \) is \( M \)'s child; on the contrary, \( R[N \rightarrow M] = RTYE.AN=2 \) means \( M \) is \( N \)'s parent.
- preceding-sibling/following-sibling(SB): for \( M \nsim N, R[M \rightarrow N] = RTYE.FS=3 \) means \( N \) is \( M \)'s following-sibling; on the contrary, \( R[N \rightarrow M] = RTYE.FS=3 \)

means \( M \) is \( N \)'s preceding-sibling.

Other relations are not necessary to be stored in matrix and queried by matching because alternate way can be adopted. For example, the preceding/following relation can be judged by just comparing the node IDs during query stage. Thus the matrix construction can be simplified and the matrix space can be greatly saved. Figure 4 show the relation matrix of example in figures 2. In fact, only the numerals which stand for relation values from \( M \) to \( N \) need to be stored. Obviously the relation matrix is sparse matrix.

Algorithm 3 shows the parallel relation matrix construction framework. Line 1 uses \( para\_for \) to apply data parallelism on sequence \( D \) with the in scope code lines from line 2 to 6. Line 3 calls \( CalcRelation \) function to calculate the relation between two nodes. Only non-zero value should be store in matrix.

Algorithm 3 ParallelBuildRelationMatrix()
1: for each nodeID Di from 0 to n
2: for each data dt in dataSet[]
3: rNode = CalcRelation(Di,Dj);
4: if (relation!=0)
5: rNode/Add rNode;
6: \( E \) for dataExp \( Exp \) = \( E \) (ParallelProcess(dataRange,Task))

where dataRange = PackData(dataExp);
Task = PackTask(Exp).

Here \( E \) represents evaluation function; PackData is the auxiliary function to parse data range expression and PackTask is to capsulate expressions in task.

B. Parallel query

XPath expression contains several query steps and every step corresponding to a certain query primitive. The query primitive realize the matching procedure. Since matching procedure is a looping style scan over the relation matrix, it is easy to apply data parallel semantic to such structure. So the query parallelism can be achieved through parallelization of query primitive. Algorithm 3 describes the

<table>
<thead>
<tr>
<th>A1</th>
<th>B1</th>
<th>C1</th>
<th>D1</th>
<th>C2</th>
<th>A2</th>
<th>B2</th>
<th>C2</th>
<th>D2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>D3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
descendant query primitive. Parameter `Seq` is the input node sequence; `nameString` is pending test node name; `isGetFollow` is a boolean value to specify whether to return forward value. Line 3 is to perform name test if name string is not a wildcard. Line 4 uses `para_for` to indicate carrying out data parallel process on input sequence `Seq` with the codes within its scope from line 5 to 13. Line 9 is to check node relation from matrix value and node name by name ID. Note `MNode[]` in line 5 represents relation matrix node list and `RC[]` in line 9 represents XML region encoding list. We have implemented other primitives including filter primitives used for predicate evaluation.

```
Algorithm 3 GetDescendant(Seq, nameString, isGetFollow)
1: nameTest: false;
2: if (nameString = "*") nameTest: true;
3: else nameID: nameIDtable.getNameID(nameString);
4: para_for each M ∈ Seq
5:   mList: MNode[M.nodeID];
6:   for each N ∈ mList
7:     nodeNID: N.getNNodeID();
8:     relation: N.getRelationType();
9:     if (nameTest = true \&\& (RC[nodeNID].nameID = nameID) \&\& (relation = RTYPE.DE) \&\& (relation = RTYPE.CH))
10:    if (isGetFollow = true)
11:       result.add(N);
12:    else
13:       result.add(M);
14:    break;
15: return result;
```

XPath expression is parsed as the series of query primitives. For instance, the query expression for figure 3(b) could be `//A/B[//D]/C`, `//A/B[//D]/C//D` or `//A[B[C]//D]` et al. Take `//A/B[//D]/C`, it can be translated to following primitive invocation sequence:

1. `input1 ← GetDescendant(input0, A.name, true);`
2. `input2 ← GetChild(input1, B.name, true);`
3. `input3 ← GetDescendant(input2, D.name, false);`
4. `result ← GetChild(input3, C.name, true);`
5. `return result;`

IV. EXPERIMENTS

We setup the experiments to verify the key execution stages which occupied most run time and the effect of parallelization under multi-core computing environment. The CPU model of test platform is AMD Athlon II X4 620, with 4 cores, basic frequency is 2.60 Ghz and running in JDK1.6.

We generated the XML documents in size 11.5Mb, 56Mb, 115Mb and 226Mb through the tool provided by the Xmark project [14]. Provided different XML data size, the run time of every stage in query process is recorded and the relative time occupation is calculated. The results are shown in figure 5. DP, SC, SM, MC, QE represents the six main stages – data partition, sub-tree construction, sub-tree merging, matrix construction and query respectively. Obviously, SC and MC stage is the key stages and they have the priority to be parallelized. Because node relations have been stored in matrix, query is low cost through matrix matching rather than relation calculation. So that QE is not a key stage.

As to the effect of our solution, we use the Tree-Bank [15] dataset which is a 82Mb XML data file and run the following six typical queries under 1 to 4 working threads.

Q1. `//S/VP/NP/PP/NN`
Q2. `//S[.//VP]/.//NP//VP//PP[.//IN]/.//NP//VBN`  
Q3. `//EMPTY[.//VP/PP//NNP][.//S[.//PP//JJ]/VBN]/PP/NP//NONE`
Q4. `//SVPP[.//NN][.//NP[.//CD]/VBN]/IN`
Q5. `//NP[.//CD]/*/V`
Q6. `//SVPP[parent::SINV/following-sibling::*]

Figure 6 show the total run time of queries. It’s obvious the parsing performances are promoted with the increase of threads number. As the run times include dataset I/O operation and experiments are performed without load balance optimization, the results haven’t reached optimal status yet. Further optimization will be carried out.

V. CONCLUSION AND FUTURE WORK

The parallelization of XML query application should be carried out in multi-stage. Parallel measures in our solution involve in both XML paring and XML query procedures. We parallelize key execution stages such as sub-tree construction in XML parsing and matrix construction in XML query to ensure the parallelization effect without extra effort. Experiments show our solution can work well under multi-core environment though it hasn’t reached optimal status at present. Since XML query is read only application in general, the XML encodings and relation matrix can be reused for
different queries as long as the XML dataset is not changed. Because query step based on matrix is very efficient, our solution is suitable for such scenarios.

Our future work will focus on optimal implementation to further improve the speedup. For one thing, the prefetching of dataset will be adopted to avoid I/O conflict among parallel tasks. For another, a proper load balance strategy will be combined with the task scheduler.

ACKNOWLEDGMENT

This work is partially supported by the Natural Science Foundation of Fujian Province of China under grant No.2008J04005.

REFERENCES