An Integration Approach for XML Query Parallelization on Multi-thread Systems

RONGXIN CHEN[1], ZONGYUE WANG[1], HUSHENG LIAO[2]
[1]Computer Engineering College, [2]College of Computer Science
[1]Jimei University, [2]Beijing University of Technology
[1]No.185 Yinjiang Rd. Jimei District, Xiamen, [2]No.100 Ping Le Yuan, Chaoyang District, Beijing CHINA
[1]ch2002star@163.com, wangzongyue1979@163.com, [2]liaohs@bjut.edu.cn

Abstract: - The key function parts of an XML query system include XML parsing, XPath and XQuery evaluation. Each part has its specific parallel opportunity and approach. And the efficiency of each part directly affects the overall effect of XML query parallelization. Therefore it is necessary to coordinate each part in a real query application to achieve the best overall parallel performance. In this paper, we propose a novel integration approach for parallelizing XML query. Our integration approach is based on the workflow of XQuery parallelization, where both parallel XML parsing and parallel XPath evaluation are seamlessly integrated. The approach can realize automatic parallelization of XML query and make full use of multi-thread computing resources for parallel processing. Experimental results indicate that our approach can effectively improve the overall performance of XML query application through parallel computing on multi-thread systems.

Key-Words: - XML query, XML parsing, XPath, XQuery, Parallelization, Multi-thread system, Integration approach

1 Introduction

With the development and popularization of Web technology, XML is used extensively as the information exchange and storage standard. XML query processing is the main approach of utilizing XML data. To deal with the rapid growth of XML data and the query requirements, various optimization measures [1] are widely studied to improve the performance of XML query. Recently, the popularity of multi-core environment provides a nice opportunity for parallel computing, and optimization based on multi-thread parallel computing becomes an important way to improve the performance of software [2]. Therefore, how to make full use of multi-thread resources to improve the performance of XML query becomes an important research topic.

Generally an XML query system is based on XQuery [3], and includes some key function parts such as XPath [4] evaluation and XML parsing. XML parsing is necessary in XML applications because XML data are in document form. The parsing of big dataset is time-consuming work, which is liable to be the bottleneck of performance. Therefore parallel processing has significance for improvement of query on multi-thread systems. We have proposed a method called ParaParse [5] to deal with parallel XML parsing. XPath is used to access XML dataset. Navigate style of XPath evaluation can easily fulfill various semantics of XPath, thus it is widely used in implementation of query engines. However, its disadvantage is the relatively low performance comparing with twig [6] style evaluation. We have proposed pM2 [7] to improve the performance of navigate XPath evaluation by parallel processing. Since XQuery, which is a mainstream XML query language, is the core part of the XML query, its performance greatly affects the overall performance of XML applications. Therefore it is critical to parallelize XQuery. XQuery is a functional language which has some advantages in parallel processing [8]. However, the parallel opportunity is hidden in various nested expressions, it is difficult to parallelize XQuery directly. In our previous work [9], we proposed a novel automatic parallelization method for XQuery based on functional intermediate language. Since an XML query system consists of the key parts including XML parsing, XPath and XQuery evaluation. An approach which coordinates the parallelization of each part is necessary to achieve the best overall parallel performance. Unfortunately, no complete integration solution for XML query

parallelization on multi-thread systems is available yet.

In order to automatically parallelize the main function parts of XML query and take full advantage of multi-thread resources to improve the overall performance of the XML application, in this paper, we propose an integration approach for parallelizing XML query based on our earlier works. Our approach takes the workflow of XQuery parallelization as the basis for integration. Then both parallel XML parsing and parallel XPath evaluation are seamlessly integrated. We implement our approach in XML query engine and carry out the experimental evaluation.

The rest of the paper is organized as follows: Section 2 presents a brief review of the related work. Section 3 describes the whole integration workflow. Section 4 describes the basis of integration – XQuery parallelization. Section 5 and 6 describes the integration of parallel XML parsing and parallel XPath evaluation respectively. Section 7 provides case study and the experiments. Section 8 gives the conclusion.

2 Related work

XML query parallel processing includes many parallelization aspects, while XQuery parallelization is the basis. To extend the distributed or parallel ability to ordinary XQuery is an important approach to improve query efficiency by making use of distributed cluster. XQueryD [10] is a lightweight extension to XQuery that allows expressing queries over distributed data sources and supports efficient query shipping. DXQ [11] is another extension of XQuery to support distributed XML query by invoking remote programs and dynamically ship query code to execute at remote servers. Recently, MapReduce processing model becomes a popular parallel framework and widely applied in various distributed environments. A query description language named ChuQL [12] is developed by extending XQuery to support XML parallel processing through MapReduce model. The studies mentioned above provide describing methods and runtime support for distributed or parallel processing of XQuery; however they generally need extending the XQuery language. Moreover, the automatic parallelization issue is not fully discussed especially for multi-thread computing. Li [13] presented a parallelization solution to XQuery through automatic rewriting. However, the study is limited in several specific expression structures and not for the full-fledged XQuery. Miao et al. [14] proposed a query plan decomposition strategy combining data partition technique for parallel XQuery processing. And yet a general parallelization solution to nested FLWOR structures which frequently appear in XQuery is not given. In our earlier work [9], we proposed an automatic parallelization method for XQuery programs based on functional intermediate language. In this method, parallel primitives are arranged in the query plan to perform parallel processing. It provides a basis for integration of XML query parallelization.

As for parallel XML parsing and parallel XPath evaluation, both of them are key parts in parallel XML query, a lot of related work have been done in these areas. Typical work in parallelization of XML parsing includes [5][15]; while typical work in parallelization of XPath evaluation includes [6][7][16]. From the perspective of parallelization for a whole XML query system, the work can only be regarded as a partial work. Effective co-ordination of all parallel parts in XML query system is critical to achieve the best parallel effect. In our previous work [17], we presented an integration parallel solution for XML query application. In the solution, parallel XML parsing and parallel XPath evaluation are combined to improve the performance of XML query application under multi-thread condition. However, the integration solution does not involve the problem of XQuery parallelization.

3 Integration workflow

The automatic parallelization for XQuery is the basis for the integration. Both parallel XML parsing and parallel XPath evaluation are integrated into parallel XQuery process through proper rewriting. Workflow of whole parallelization procedure consists of three major processing stages as illustrated in Fig. 1.

(1) Stage I is pre-processing of parallelization. The primary work in this stage is to translate XQuery into FXQL (Functional XML Query Language) [18], which is a functional intermediate language developed by our work group. In the translating procedure, firstly XQuery source code is translated into XQuery Core [19] code through normalization. Then XQuery Core code is further translated into FXQL code.

(2) Stage II is processing of parallelization. Through dependence analysis and cost estimation, FXQL code is rewritten to pFXQL code and parallel query plan is generated. In the rewriting procedure, XPath relative expressions is rewritten to parallel pM2 primitives, and the XML parsing expression is rewritten to parallel XML parsing primitive.

(3) Stage III is executing of parallel query plan. In essence, the execution of the parallel query plan

is the invoking of primitives contained by the plan. Primitives include various parallel primitives, as well as a variety of non-parallel primitives. The parallel execution of XPath is realized by invoking parallel pM2 primitives. Similarly XML parsing is executed in parallel by invoking parallel XML parsing primitive which is encapsulated with the implementation of ParaParse method. Relation matrix of XML node is required for the execution of pM2 primitives. Obviously the construction of the matrix only needs once for the same XML dataset, and it will be performed after XML parsing.

Since the whole workflow is designed as an automatic processing procedure to support implicit parallelism, application developers only need to write a regular XQuery program, which will be automatically processed in parallel by XML query engine.

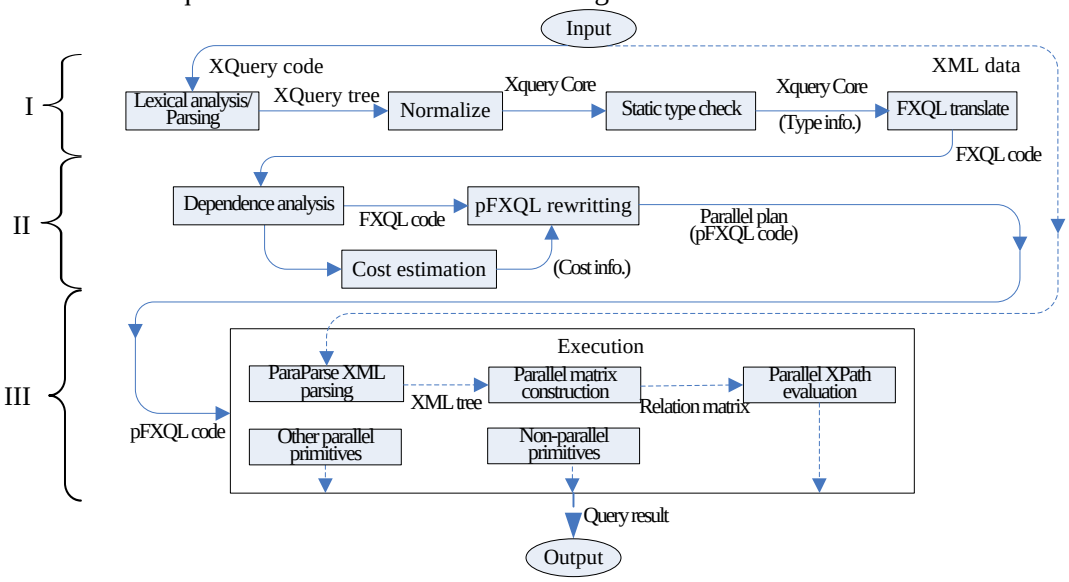

Fig.1: Workflow of integration approach for parallelizing XML query

4 XQuery parallelization

An XQuery program usually contains multiple FLWOR expressions which may be nested. The nested programming style can facilitate describing various flexible query requirements. However, it makes the program structure too complex and difficult to be parallelized. Rewriting becomes an important way for XQuery parallelization. In our earlier work [9], we proposed an automatic method to solve parallelization of XQuery based on functional intermediate language. The basic idea is to design a functional language called FXQL, which is well complied with XQuery, to describe query plans. Furthermore, a functional language called pFXQL, which has parallel semantics, is developed to describe parallel query plans. Task scheduling is carried out by invoking parallel primitives in pFXQL. The design and rewriting of functional intermediate language play the key role in XQuery parallelization; they also provide the basis for integration of XML query parallelization.

4.1 Functional intermediate language

FXQL is a functional intermediate language which is complied with XQuery. It is designed to describe query plans. Syntax of FXQL is listed in Table 1, where *e* is a FXQL expression, *v* is a variable, *f* is a function and *c* is a const. In expression (3), *v=e* is variable binding, indicating that the definition of *v* is expression *e*. *v=f* is function binding, indicating that the definition of *f* is expression *e*. Expression (4) describes function call where *f* is primitives standing for built-in functions or user-defined functions. When *f* is *COND*, it indicates a condition operation; and *f* = *MAP* indicates an iterative operation.

Table 1: Syntax of FXQL

Expression Name	Expression Syntax
(1) Const	*e* ::= *c*
(2) Variable /function	*e* ::= *v* | *f*
(3) Expression with bindings	*e* ::= *e where* (*v* = *e* | *f* = *e*)(*v* = *e*)*(*f* = *e*)*
(4) Function call	*e* ::= *f*(*e**)

There are two kinds of primitives in FXQL: one is iterative primitive, and the other is ordinary primitive. The prototype of iterative primitive is *MAP (D,F,op)*, *op*∈{*foreach*, *foreachat*, *filter*, *filterat*}. It means there is an *op*-type iterative operation on function *F* according to the size of data

sequence which is the evaluating result of expression *D*. The *MAP* primitive comes from *FOR* clause in FLWOR expression and various axis operations. Its role is to control the iterative number of evaluation. While the role of ordinary primitives is to perform various computing.

pFXQL language is the extension of FXQL with parallel semantics to describe parallel query plans. The main syntax structures in pFXQL are the same as in FXQL, while the difference between them is the using of parallel primitives in pFXQL. In the semantic domain of pFXQL, thread information is added to the evaluation environment. The prototype of semantic function is defined as $[\![...]\!]: Exp \to Env \to DV$, where *Exp* is an FXQL expression, *Env* is the evaluating context and *DV* is the denotational value. Here *DV=Val + Def*, *Val* is a variable value and *Def* is a function definition. Instances of context are denoted as $\rho, \rho', \rho'' \in Env = v \mapsto Val + f \mapsto Def + t \mapsto Thread$, where *Thread* is available working thread. The evaluating context is expanded by the binding of variables, functions or threads during evaluation. $\cup$ denotes the expanding operator. Parallel primitives appear only in function call expression in pFXQL as shown in the following semantic equation.

$$
\begin{aligned}
&[\![f(e^*)]\!]_\rho \triangleq \\
&\quad if\ (f(e^*) = PMAP(e1, e2, op))\ then \\
&\qquad foreach(p\ in\ ps)\ r = r \oplus map(p, [\![e2]\!]_{\rho''=\rho' \cup t' \mapsto getThread(1)}, op) \\
&\qquad where\ xs : ps = partition([\![e1]\!]_\rho) \\
&\qquad\quad r = map(xs, [\![e2]\!]_{\rho'=\rho \cup t \mapsto getThread(1)}, op) \\
&\quad else\ if\ (f(e^*) = PIPE(e1, e2))\ then \\
&\qquad [\![e2]\!]_{\rho'=\rho \cup t \mapsto getThread(1) \cup v \mapsto (getone([\![e1]\!]_\rho) | getsome([\![e1]\!]_\rho))} \\
&\quad else\ if\ (f(e^*) = PARA(e))\ then \\
&\qquad [\![e]\!]_{\rho'=\rho \cup t \mapsto getThread(1)} \\
&\quad else\ if\ (f \in prim \wedge f \neq PMAP \wedge f \neq PIPE \wedge f \neq PARA \wedge f \neq \\
&\qquad COND))\ then\ f([\![e^*]\!]_\rho) \\
&\quad else\ if\ (f(e^*) = COND(e1, e2, e3))\ then \\
&\qquad if\ [\![e1]\!]_\rho\ then\ [\![e2]\!]_\rho\ else\ [\![e3]\!]_\rho \\
&\quad else\quad [\![body(f(e^*))]\!]_\rho
\end{aligned}
$$

In above equation, *PMAP* is the primitive for data parallelism, *PIPE* is for pipeline parallelism and *PARA* is for task parallelism. The fourth line from the bottom is corresponding to invoke other primitives including some parallelized specialized primitives. The auxiliary function *map* is used to carry out iterative operation according to operation type *op*, the function *partition* is to divide data sequence, the function *getone* is to fetch an element from sequence and *getsome* is to fetch data block for pipeline processing, the function *getThread(1)* is used to get one worker thread from thread pool, and the function *body* is to get the body of self-defined functions.

4.2 From XQuery to pFXQL

XQuery provides sophisticated syntax and complete language framework to support developing various powerful XML query programs. To simplify the process of XQuery program during automatic parallelization, we use a subset of the XQuery Core [19] which is normalized to eliminate some syntactic sugar. The main syntax of XQuery Core processed in this paper as follows.

e::= *c* | *v* | *for v (at v)? in e return e*
| *let v:=e return e* | *if (e) then e else e*
| *(some|every) in e satisfiers e*
| *e* (*and|or|+|-|*|idiv|=|<|<<|…*) *e*
| *e,e* | *e/e* | *e[e]* | *f(e*)*

The translation function is defined as ***T***[…]: *CExpr→Exp*, where *CExpr* is XQuery Core expression and *Exp* is FXQL expression. The main translation rules are listed as follows.

(R1) ***T***[*c*] = *c*
(R2) ***T***[*v*] = *v*
(R3) ***T***[*for v in e1 where e2 return e3*] = *MAP (MAP(**T**[e1],f1,filter), f2, foreach) where {f1(v) = **T**[e2], f2(v') = **T**[e3]}*
(R4) ***T***[*for v1 at v2 in e1 return e2*] = *MAP(**T**[e1], f, foreachat) where {f(v1,v2) = **T**[e2]}*
(R5) ***T***[*let v=e1 return e2*] = ***T***[*e2*] *where{v = **T**[e1]}*
(R6) ***T***[*if e1 then e2 else e3*] = *COND(**T**[e1], **T**[e2], **T**[e3])*
(R7) ***T***[(*some|every*) *v in e1 satisfies e2*] = (*QUANTIFIEDSOME|QUANTIFIEDEVERY*) *(**T**[e1], f) where {f(v) = **T**[e2]}*
(R8) ***T***[*e1 opt e2*] = *OPT(**T**[e1], **T**[e2])*
(R9) ***T***[*e1,e2*] = *CONCATE(**T**[e1], **T**[e2])*
(R10) ***T***[*e1/e2*] = *MAP(**T**[e1], f, foreach) where{f(v)=AXIS(**T**[e2], v)}*
(R11) ***T***[*e1[e2]*] = *MAP(**T**[e1], f, filter) where{f(v)= CHILD_ONE(**T**[e2], v)}*
(R12) ***T***[*f(e*)*] = *FUN((**T**[e])*)*

It's simple and direct to translate XQuery to FXQL according to the translation rules. Fig.2 shows the translation of an XQuery program which contains FLWOR structure. The XQuery code is firstly translated to FXQL code, and then further rewritten into pFXQL code. Fig.2(b) and Fig.2(c) show the simplified codes. It can be found that the parallel primitive *PMAP* is employed to perform data parallelism in Fig.2(c).

let $a:=3	FLAT(SORTTUPLE(

```
for $b in Dataset
  where $b>$a
  order by $b
return $b+10
```

(a) XQuery code

```
MAP( MAP( Dataset, gq:Func3, filter )
  where { gq:Func3 ( b@3 ) = QUANTIFIEDSOME( b@3, gq:Func2)
   where { gq:Func2 ( v1@4 ) = QUANTIFIEDSOME( a@2, gq:Func1)
    where { gq:Func1 ( v2@5 ) = GT( u1@6, u2@7 ) } } }, gq:Func0,
foreach)
     where { gq:Func0 ( b@3 ) = PLUS_INTEGER( b@3 , 10 )}
     where { a@2 = 3 }, "asc" ) )
```

(b) FXQL code

```
FLAT( SORTTUPLE( v111@112, "asc" ) )
 where {
  v111@111= PMAP( Dataset, gq:Func3, filter )
   where { gq:Func3 ( b@3 ) = QUANTIFIEDSOME( b@3, gq:Func2)
    where { gq:Func2 ( v1@4 ) = QUANTIFIEDSOME( a@2, gq:Func1)
     where { gq:Func1 ( v2@5 ) = GT( u1@6, u2@7 ) }, a@2 = 3 } },
  v111@112= PMAP(v111@111, gq:Func0, foreach )
   where { gq:Func0 ( v111@111 ) = PLUS_INTEGER( v111@111, 10 )} }
```

(c) pFXQL code

Fig.2: Translate from XQuery to pFXQL

5 Integration of parallel XML parsing

In our earlier work [5], we proposed a parallel XML parsing method called ParaParse based on sub-tree construction. It utilizes a light weighted data partition technique and supports parsing arbitrary XML segments in parallel. And after that sub-trees are merged to generate the complete XML tree. During integrating process, the data model should be kept consistent and parallel parsing primitive should be implemented to adapt to pFXQL description.

5.1 Data model

Generally an XML query engine contains various function parts such as XML parsing, XPath evaluation and XQuery query. We allow different parts to use their specific data form for efficient processing. While a unified data model should be complied with in the process of integration. The W3C has defined the XQuery and XPath Data Model (XDM) [20], which defines all permissible values of expressions in XQuery and XPath languages. The value of every expression in the language is closed in the data model. Value types in XDM include sequence and item, while item includes atomic value and node. Every instance of the data model is a sequence, which is an ordered collection of zero or more items. In XDM, a sequence cannot be a member of a sequence, thus it is flat. While in the data model of pFXQL, we utilize generalized list to extend the original XDM. Since generalized list can reserve the grouping and layered information of intermediate results, more flexible evaluation and optimization methods can be applied. Value types in our data model are described as follows,

List ::= ( ) | (*Item*, …, *Item*)
Item ::= *Atom* | *Node* | *List*

Where *List* is a generalized list, *Item* may be an atomic value, an XML node or a generalized list. In order for XML query engines to be able to operate on instances of the data model, a family of *accessor* functions is defined. Some frequently used accessors are listed below, where *dm* is the namespace-prefix of data model.

(1) *dm:children($n)*: Returns the children of the node *$n* as a sequence containing zero or more nodes;
(2) *dm:attributes($n)*: Returns the attributes of the node *$n* as a sequence containing zero or more attribute nodes;
(3) *dm:parent($n)*: Returns the parent of the node *$n* as a sequence containing zero or one nodes;
(4) *dm:node-name($n)*: Returns the name of the node *$n* as a sequence of zero or one *xs:QNames*;
(5) *dm:string-value($n)*: Returns the string value of the node *$n*.

The above accessors are implemented to access to the node information of XML data. Algorithm 1 describes the implementation of *dm:children* accessor by using node information from the parsing result of ParaParse. The function *firstChild* in line 2 is used to get the first child node of the current node. The function *nextSibling* in line 5 is used to get the next sibling node of the current node. All the information can be directly retrieved from the parsing result. The implementation of other accessors is omitted due to space constraints.

Algorithm 1 Implementation of *dm:children*
Node[] GetChildren(Node nd)
Input: node nd
Output: node sequence

```
1: chs←∅; // Node[] chs
2: next←nd.firstChild; //Node next
3: while (next≠∅) do
4:   chs←chs∪next;
5:   next←next.nextSibling;
6: end while
7: return chs
```

5.2 Parallel XML parsing primitive

XML parsing is a relatively independent function within XML query application, the prototype of XML parsing primitive in the FXQL query plan is described as *DOC(XmlDoc)*, where *XmlDoc* is the URL of XML dataset. While the prototype of parallel XML parsing primitive in the pFXQL query plan is described as *PDOC(XmlDoc)*, which corresponds to the parallel implementation of the parsing method ParaParse.

The rewriting method is rather simple. At rewriting stage, the serial XML parsing primitive *DOC* in FXQL query is directly replaced by the corresponding parallel one – *PDOC*. Then in executing stage, XML parsing will automatically execute in parallel with ParaParse method. Taking into account the overhead of parallelization, the parallel effect of parsing a small dataset may be insignificant. Thus an applicable condition is preset to decide whether to perform parallel parsing in executing stage according to the volume of dataset.

6 Integration of parallel XPath evaluation

We proposed a parallel evaluation method called pM2 method [7] for navigate XPath evaluation in our earlier work. The method has two main stages includes parallel relation matrix construction and parallel query. Node relation matrix of XML dataset is firstly constructed according to XML parsing results, and then parallel query is performed by invoking parallelized query primitives. Iterative processes in both matrix construction and query primitives are implemented with data parallelism, so that each stage can take advantage of multi-thread resources. Since the parallel query in pM2 depends on node relation matrix, construction of the matrix should be carried out upon the first XPath evaluation or right after XML parsing. In integrating procedure, parallel pM2 primitives should be designed to wrap the functions of pM2. Then XPath expression is rewritten to a calling sequence of parallel pM2 primitives in pFXQL query plan.

6.1 Parallel pM2 primitives

Parallel pM2 primitives are used to describe the evaluating steps in XPath expression with pM2 method. They are used in form of pFXQL primitives therefore the description of parallel XPath evaluation can be seamlessly integrated into pFXQL query plan. The semantic function of pM2 primitive is defined as ***E*[...]:** *Exp*→Val, where *Exp* denotes pM2 primitives, *val* denotes the evaluation result. The implementation of several frequently used primitives is described in semantic equations as listed below.

(E1) ***E*[** *GET_DESCENDANT(e2, e1)* **]** =
list←GetDescendant(***E*[** *e1* **]**, *e2*.name, true); //NodeCode[] list
result←GetNodeList(list);
return(result);

(E2) ***E*[** *GET_CHILD(e2, e1)* **]** =
list←GetChild(***E*[** *e1* **]**, *e2*.name, true)
result←GetNodeList(list);
return(result);

(E3) ***E*[** *GET_FILTER(e2, e1)* **]** =
list←FilterInput1byInput2(***E*[** *e1* **]**, ***E*[** *e2* **]**)
result←GetNodeList(list);
return(result);

The auxiliary functions appear in the right part of equations are pM2 query primitives from article [7]. Equation E1 to E3 describes the evaluation of the pM2 primitive *GET_DESCENDANT*, *GET_CHILD* and *GET_FILTER* respectively. In equation E1, query primitive *GetDescendant* is used to get the descendants of the input node in parallel, and function *GetNodeList* is utilized to get node sequence which complied with the data model. Evaluating result *list* in the equation is a node sequence which contains XML encoding information. Query primitive *GetChild* is used to get the children of the input node in parallel in equation E2, and query primitive *FilterInput1byInput2* is used to perform predicate evaluation in parallel in equation E3.

6.2 XPath rewriting

In order to describe pM2 evaluation in pFXQL query plan, the XPath primitive in FXQL query plan needs to be rewritten to parallel pM2 primitives. The prototype of XPath primitive is in form of *MAP* function call as shown below.

MAP(e1, f, op)
where{f(v)= AXIS (e2, v)}

Where e1 is the input expression, and the binding function *f* is corresponding to an axis operation. *AXIS*∈ {*DESCENDANT*, *CHILD*, *FOLLOWING_SIBLING*, … }, denotes various types of axis operation. *op*∈{*foreach*, *filter*}, denotes different types of iterative operation. The rewriting function is defined as ***T*[...]**: *Expr*→*Expr*, where *Expr* is a pFXQL expression. Several frequently used rewriting rules are listed as follows.

(R1) ***T*[** *MAP(e1, f, foreach)*
where{f(v)= DESCENDANT (e2, v)} **]** =
*GET_ DESCENDANT (**T**[e2], **T**[e1])*

(R2) ***T*[** *MAP(e1, f, foreach)*
where{f(v)=CHILD(e2, v)} **]** =
*GET_CHILD(**T**[e2], **T**[e1])*

(R3) ***T*[** *MAP(e1, f, filter)*
where{f(v)= CHILD(e2, v)} **]** =
*GET_FILTER(**T**[e2], **T**[e1])*

Rule R1, R2 and R3 are used to rewrite the primitives for getting descendant nodes, for getting node child nodes and for predicate evaluating respectively. After the rewriting process in stage II of the workflow, parallel XPath evaluation is seamlessly integrated into pFXQL query plan.

7 Case study and experiments

7.1 Case study

Case 1 below is a typical XML query program running on XMark benchmark [21]. It comes from article [22] with some modification. There are multiple nested FLWOR structures and several XPath expressions in this case. In this section, we use it to explore the parallelization for sophisticated XML query.

The automatically generated parallel query plan, which is described in pFXQL, is shown in Fig.3. To save space, only partial expressions which contain parallel primitives are shown. In the query plan, there are total 15 extracted query blocks which are in the form of variable binding expression. The query block bound by the variable *auction@1* contains the parallel XML parsing primitive *PDOC*. The query blocks bound by the continuous variables from *v111@111* to *v117@117*, as well as the variables *v119@119* and *v120@120*, contain parallel pM2 primitives. The variable *v118@118* binds a query block which contains a *PMAP* primitive. That means the block will be processed in data parallelism. Each of the query blocks bound by variable *v122@122* and *v123@123* contains a PIPE primitive. The two blocks are arranged in sequence, so that pipeline stages will be constructed to perform pipeline parallelism.

Case 1. A typical XML query.

```
let $auction := doc('xmark.xml') return
let $euro:= for $o in $auction/site/open_auctions/open_auction
for $i in $auction/site/regions/europe/item/@id
   where $o/itemref/@item eq $i
return $o
for $a in $euro
   where ($a/ bidder[1]/increase)*2<= $a/bidder[last()]/increase
return for $p in $auction/site/people/person[profile/@income>5000]
            for $w in $p/watches/watch
                where $a/@id = $w/@open_auction
            return <auction id="{$a/@id}">
                        <increase first="{$a/bidder[1]/increase/text()}"
                            last="{$a/bidder[last()]/increase/text()}"/>
                        <watched_by id="{$p/@id}"/>
                    </auction>
```

Hidden codes in line 11~49, 51~129, 131~279 and 280~506 contains some un-rewritten XPath primitives. That means the query blocks which contain such primitives can be processed in other parallel way instead of using pM2. For instance, the query block corresponding to line 11~49 can be processed in data parallelism by using *PMAP* primitives; the query blocks corresponding to line 131~279 and line 280~506 can be processed in pipeline parallelism.

```
1       FLAT( v123@123)
2       where
3   [-] {  auction@1 = PDOC("xmark.xml"),
4         v111@111 = GET_CHILD(element(site), auction@1),
5         v112@112 = GET_CHILD(element(regions), v111@111),
6         v113@113 = GET_CHILD(element(europe), v112@112),
7         v114@114 = GET_CHILD(element(item), v113@113),
8         v115@115 = GET_CHILD(element(site), auction@1),
9         v116@116 = GET_CHILD(element(open_auctions), v115@115),
10        v117@117 = GET_CHILD(element(open_auction), v116@116),
11        v118@118 = PMAP(v117@117, gq:Func10, foreach)
12  [-]     where { gq:Func10 (o@43) =
13  [+]           FLAT(
49                  )},
50        v119@119 = GET_CHILD(element(site), auction@1),
51        v120@120 = GET_CHILD(element(regions), v119@119),
52        v121@121 = MAP(v120@120, gq:Func19, foreach)
53  [-]     where { gq:Func19 (fs:dot3@20) =
54  [+]           FLAT(
129                 )},
130       euro@2=FLAT(v118@118),
131       v122@122= PIPE(euro@2, gq:Func52, filter)
132 [-]     where { gq:Func52 (a@3) =
133 [+]           QUANTIFIEDSOME(
202 [+]            ) where {
279                 }},
280       v123@123= PIPE(v122@122, gq:Func39, foreach)
281 [-]     where { gq:Func39 (a@3) =
282 [+]           FLAT(
506                 )},
507 }
```

Fig.3: The parallel query plan corresponding to Case 1

7.2 Experimental evaluation

To evaluate the parallelization effect of our approach, we conduct experiments on a multi-core laptop PC with 4Gb RAM. We implement our approach in XQuery engine using Java. The running environment is JRE 1.6 and Windows XP sp3. The typical test cases, which are labeled as W1~W6 and X1~X6 respectively, are selected from W3C XQuery use cases [23] and XMark benchmark cases [21]. The indexes for test cases are listed in Table 2.

Table 2: Index table for test cases

Case	Case in [23]	Case	Case in [21]
W1	1.1.9.5 Q5	X1	Q8
W2	1.1.9.10 Q10	X2	Q9
W3	1.1.9.12 Q12	X3	Q10
W4	1.4.4.3 Q3	X4	Q11
W5	1.4.4.10 Q10	X5	Q12
W6	1.4.4.11 Q11	X6	Q20

The data volume in the original W3C test case is relatively small, so that we generate large-sized XML data sets for testing according to the original data model. In W3C cases, case W1 and W2 are the cases with single FLWOR structure and a single data source, while case W3 and W6 contain multiple nested FLWOR structures. Both W4 and W5 contain multiple data sources. Most of XMark cases contain nested FLWOR structures and multiple long XPath expressions. There is nested FLWOR structure in the return clause of case X1, while there are some complex node construction expressions in the return clause of case X3.

7.2.1 Parallelization effect

We run the tests under four-thread condition and compare the serial execution times with the parallel execution times after automatic parallelization. The comparison result is shown in Fig.4. It can be found that the execution times of most cases are significantly reduced by parallelization. The speedup of case W1, W3, W4, X4 and X5 exceeds 3.0, XPath evaluation and XQuery query occupy a high proportion of total workload in such cases. While the speedup of case X6 is only 1.57, it can be found XML parsing takes the bigger part of total execution time. All the test cases contain FLWOR structures, which may be in multi-layer nested style,

the average speedup of all the cases is 2.61 under four-thread condition. Experimental results show that our approach is suitable for processing the parallelization of FLWOR structures which often appear in an XML query.

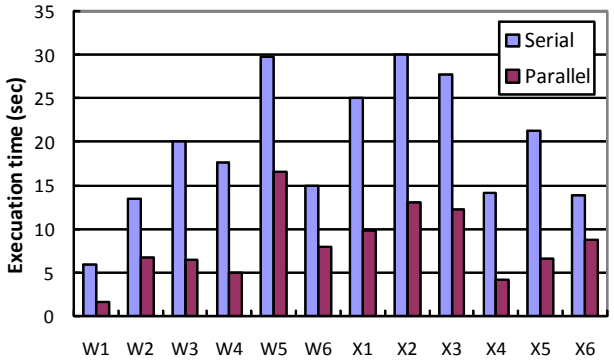

Fig.4: Comparison of execution time in serial and parallel

7.2.2 Contribution rate

We utilize the contribution rate to measure the roles of different function parts in XML query during parallel evaluation. The contribution rate r_{fp} refers to the execution time saved in a certain function part *fp* in the proportion of total time saved by parallelization. The total saving execution time through parallelization is described in formula (1), where *t* denotes execution time, the superior *s* stands for serial processing and *p* for parallel processing, the subscripts *all*, *xml*, *xpath*, *xquery* stand for total parts, XML parsing part, XPath evaluation part and XQuery query part respectively. Formula (2) presents the computing method for the contribution rate of every function part.

$$t^s_{all} - t^p_{all} = (t^s_{xml} - t^p_{xml}) + (t^s_{xpath} - t^p_{xpath}) + (t^s_{xquery} - t^p_{xquery}) \quad (1)$$

$$r_{xml} = \frac{t^s_{xml} - t^p_{xml}}{t^s_{all} - t^p_{all}},\ r_{xpath} = \frac{t^s_{xpath} - t^p_{xpath}}{t^s_{all} - t^p_{all}},$$

$$r_{xquery} = \frac{t^s_{xquery} - t^p_{xquery}}{t^s_{all} - t^p_{all}} \quad (2)$$

The contribution rates are calculated according to execution times. Serial execution times come from the test results of the query plans without any parallelization. The parallel execution times of XML parsing are obtained when parallel primitive *PDOC* is used. The parallel execution times of XPath evaluation are obtained when parallel pM2 primitives are utilized in query plans. As for XQuery query, the parallel execution times are obtained when parallel query primitives are employed in query plans, which are the results of automatic parallelization.

The contribution rates of all the test case are shown in Fig.5. It can be found that in most cases such as case W1 to X5, XQuery query takes up the big part of the total contribution rate. While parallel XML parsing brings the highest contribution rate in case X6. The reason is that the workload of the query part is small in this case, in contrast, XML parsing accounted for a large proportion of total execution time.

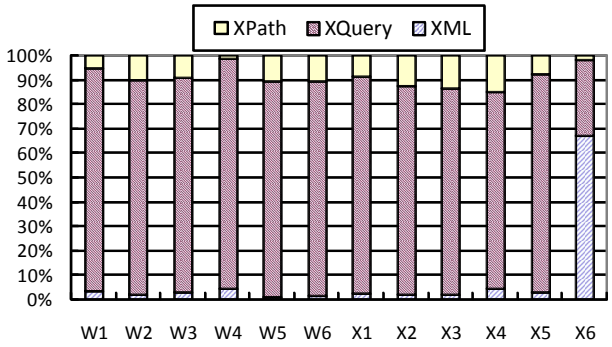

Fig.5: Contribution rates in different cases

In order to further investigate the impact brought by data volume, Case 1 in section 7.1 is selected for testing with different volume of XML dataset. On the 4-thread test platform, the contribution rates are shown in Fig.6. This indicates that with the increasing of XML data volume, the contribution rates of parallel XML parsing gradually decreased, while the contribution rate of parallel XQuery query gradually increased, and contribution rates of parallel XPath evaluation keep stable. The reason is that the workload of parallel XQuery query increases rapidly upon the increasing of data volume, while the XPath evaluation has a relatively small portion of the whole process.

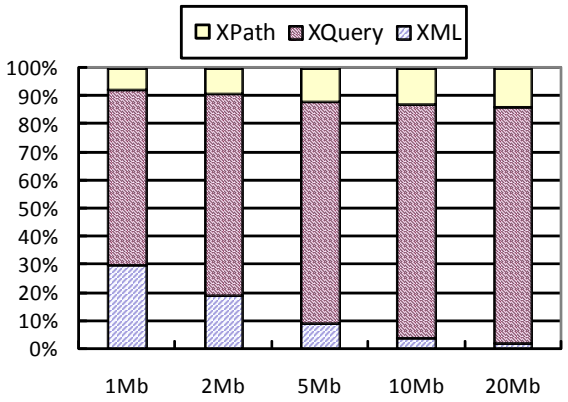

Fig.6: Contribution rates under different data size condition

8 Conclusion

In the XML query system, the three main function parts, which include XML parsing, XPath evaluation and XQuery query, have corresponding parallelizable opportunities and different parallelization methods. The parallelization effect of each part will contribute to the overall effect of parallel XML query. In our approach, the parallelization characteristic of each function part is fully taken into account. XQuery parallelization is chosen as the basis for integration and functional intermediate language is utilized to describe the query plan in a united form. Each function part is integrated through rewriting to make full use of multi-thread resource to improve the performance of XML query. Moreover, the workflow of our approach is an automatic parallelization procedure, therefore implicit parallelism can be achieved to improve ease of use of parallel XML query.

Acknowledgments

This research was supported by the National Natural Science Foundation of China (No.41201462), the Natural Science Foundation of Fujian Province of China (No.2014J01245), the Education Department of Fujian Province Fund for Outstanding Young (No.JA13168) and the Science Foundation of Jimei University (No.ZQ2014003).

References:

[1] X. Meng, Y. Wang, X. Wang, Research on XML Query Optimization, *Journal of Software*, Vol.17, No.10, 2006, pp.2069-2086.

[2] H. Sutter, The free lunch is over: A fundamental turn toward concurrency in software, in: http://www.gotw.ca/publications/concurrency-ddj.htm, 2005.

[3] S. Boag, D. Chamberlin, M.F. Fernández, D. Florescu, J. Robie, J. Siméon, M. Stefanescu, XQuery 1.0: An XML query language, in: http://www.w3.org/TR/xquery/, 2007.

[4] A. Berglund, S. Boag, D. Chamberlin, M.F. Fernandez, M. Kay, J. Robie, J. Siméon, XML path language (XPath) 2.0, in: http://www.w3.org/TR/xpath20/, 2007.

[5] R. Chen, H. Liao, ParaParse: A parallel method for XML parsing, in: *2011 IEEE 3rd International Conference on Communication Software and Networks (ICCSN)*, IEEE, 2011, pp. 81-85.

[6] I. Machdi, T. Amagasa, H. Kitagawa, Parallel holistic twig joins on a multi-core system, *International Journal of Web Information Systems*, Vol.6, No.2, 2010, pp.149-177.

[7] R. Chen, H. Liao, Z. Wang, Parallel XPath Evaluation Based on Node Relation Matrix, *Journal of Computational Information Systems*, Vol.9, No.19, 2013, pp.7583-7592.

[8] K. Hammond, Parallel functional programming: An introduction, in: *International Symposium on Parallel Symbolic Computation*, Citeseer, 1994.

[9] R. Chen, Parallelized XML Query Based on Functional Intermediate Language, *Journal of Chongqing University of Technology(Natural Science)*, No.7, 2011, pp.81-86.

[10] C. Re, J. Brinkley, K. Hinshaw, D. Suciu, Distributed xquery, in: *Workshop on Information Integration on the Web*, Citeseer, 2004, pp. 116–121.

[11] M. Fernández, T. Jim, K. Morton, N. Onose, J. Simeon, DXQ: A distributed XQuery scripting language, in: *Proceedings of the 4th international workshop on XQuery implementation, experience and perspectives*, ACM, 2007, pp. 1-6.

[12] S. Khatchadourian, M.P. Consens, J. Siméon, Having a ChuQL at XML on the Cloud, in: *A. Mendelzon Int'l. Workshop*, 2011.

[13] X. Li, Efficient and parallel evaluation of XQuery, The Ohio State University, 2006.

[14] H. Miao, T. Nie, D. Yue, T. Zhang, J. Liu, Algebra for Parallel XQuery Processing, in: *Web-Age Information Management*, 2012, pp. 1-10.

[15] W. Lu, K. Chiu, Y. Pan, A parallel approach to xml parsing, in: *7th IEEE/ACM International Conference on Grid Computing*, IEEE, 2007, pp. 223-230.

[16] R. Bordawekar, L. Lim, A. Kementsietsidis, B.W.L. Kok, Statistics-based parallelization of XPath queries in shared memory systems, in: *Proceedings of the 13th International Conference on Extending Database Technology (EDBT)*, ACM, 2010, pp. 159-170.

[17] R. Chen, W. Chen, A parallel solution to XML query application, in: *2010 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT)*, IEEE, 2010, pp. 542-546.

[18] X. Zhang, H. Liao, A framework for XQuery system with XML algebra and tree pattern query, *Journal of Frontiers of Computer Science and Technology*, Vol.4, No.11, 2010, pp.996-1004.

[19] D. Draper, P. Fankhauser, M.F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, P.

Wadler, XQuery 1.0 and XPath 2.0 formal semantics, in: http://www.w3.org/TR/xquery-semantics, 2007.

[20] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, N. Walsh, XQuery 1.0 and XPath 2.0 data model, in: http://www.w3.org/TR/xpath-datamodel, 2007.

[21] A. Schmidt, F. Waas, M. Kersten, M.J. Carey, I. Manolescu, R. Busse, XMark: A benchmark for XML data management, in: *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB Endowment, 2002, pp. 974-985.

[22] M. Brantner, C.C. Kanne, G. Moerkotte, Let a single FLWOR bloom, *Lecture Notes in Computer Science*, 2007, pp.46-61.

[23] W.W.W. Consortium, XML Query Use Cases, W3C Working Group Note, 2007.