Automatic Parallelization of XQuery Programs

Husheng Liao
Beijing University of Technology, Beijing, China 100124
Email: liaohs@bjut.edu.cn

Weifeng Shan and Hongyu Gao
Beijing University of Technology, Beijing, China 100124
Email: shwf@163.com, hy_gao@bjut.edu.cn

***Abstract*—XQuery is a functional language with implicit parallelism. It is an important approach to improve the efficiency of XML query by taking full advantage of multi-core environment in the parallel implementation of XQuery language. In this paper, we propose an implementation method for parallelizing XML query represented by XQuery programs automatically. According to the features of its functional language, an XQuery program is divided into a number of tasks that can be executed in parallel. Then, on the basis of the running cost evaluation, three kinds of parallelism are applied to different tasks and they are data parallelism, task parallelism and pipeline parallelism. Under the guidance of a novel scheduling strategy, the execution of the XQuery program is parallelized automatically. The experiments show that this approach improves the efficiency of the execution of XQuery programs and the computing resources of multi-core computer are used efficiently.**

***Index Terms*—XQuery, implicit parallelism, XML, multi-core, task scheduling, task partitioning**

I. Introduction

Extensible Markup Language (XML) has become the standard format for data representation and data exchange on the Internet because of the features of self-description and semi-structured [1]. XQuery is a standard query language from the W3C designed to query XML, and it is also a functional programming language. A number of optimization approaches, such as query algebra, XML tree pattern query algorithm and compile technology, have been proposed to improve the query performance of XQuery.

Now, people begin to try to utilize the computational capacity of multi-core machines to enhance the efficiency of XML query [2]. Reference [3] separated the process of parsing the XML from the process of reading XML files. Each process used a single thread. Reference [4] proposed another parallel XML parsing method, in which an initial preparing phase was used to determine the structure of the XML document, and then a full parallel parse was followed by. Reference [5] encoded XML and indexed it to improve the query performance. Reference [6] researched XPath parallel query solutions on the shared XML documents, including data parallelism and task parallelism strategies. Reference [7] proposed two XML data partitioning strategies to keep workload balance for parallel tree pattern query. They refined an XML partition at various levels of granularities proposed an XML data distribution approach by partitioning XML data on the fly at the stream nodes-based granularity dynamically.

XQuery, as a functional language, is fully implicit parallel programming language. An XQuery program consists of expressions and function calls. The expressions with no data dependency could be evaluated in parallel, and the parallel query has the identical result with serial execution. Over the years, people have been mining the parallelism of functional languages and trying to find an automatic or semi-automatic method to improve the execution efficiency of programs [8-10], while few their works involve the data parallelism and pipeline parallelism in XML data processing. Most parallel query optimizations only consider XML parsing, XPath queries and twig query, and there are no reference has been found to deal with the application of variety of parallel strategies on the whole query task of XQuery programs.

This paper investigates the automatic parallelization of XQuery programs and the main contributions of this paper are as follows:

- The paper firstly proposes an automatic parallelization approach of XQuery language according to the program structure and query characteristics of XML, which provides three parallel strategies: data parallelism, task parallelism and pipeline parallelism.
- The representation method of query plan based on data dependence relationship and task partition strategies based on execution cost model are proposed, and we also implement the algorithms of query plan and task partitioning.
- A task scheduling strategy is realized which considers the dependency relationship and

Manuscript received August 26, 2012; revised XX, 2012; accepted XX, 2011
Corresponding author: Weifeng Shan.

execution cost of different tasks. Based on the task scheduling, a parallel XQuery engine has been implemented.

The paper is organized as follows. Section II introduces the automatic parallelism strategies of XQuery language. Section III shows the query plan of XQuery and task partitioning approach. Section IV presents the running cost model. In section V we present the task scheduling strategy. The implementation of a parallel XQuery engine is introduced in section VI. Evaluation and experiment results are discussed in Section VII.

II. Automatic Parallelization of XQuery

XQuery is a standard XML query language, while it doesn't provide parallel control technology such as multi-thread and synchronization control. To improve efficiency of XQuery in multi-core environment, parallel implementation of XQuery is one solution. As a functional language, it makes itself possible to mine its implicit parallelism.

Consider the XQuery program in Fig. 1, we could find three kind of parallel strategies as follows, which can be applied to improve the efficiency of the program.

- Task parallelism. FLWOR expressions are the core expression of XQuery, some of them can be executed concurrently. In Fig. 1, the evaluations of top two *let* clauses(line 1, 2) do not rely on each other, so they can be viewed as two independent tasks and can be executed in parallel.
- Pipeline parallelism. The data model of XQuery is the data sequence, which is the sequence of XML nodes in most cases. It is obvious that two expressions in an XQuery program may have the relationship of producer and consumer. As shown in Fig. 1, the *book* elements generated from the *let* clause at line 1 are dealt as the input of the *for* clauses at line 3. It isn't necessary to wait until all *book* elements are obtained before the execution of the *for* clause. We may take the pipeline approach to make them work in parallel.
- Data Parallelism. In FLWOR expressions, the same operation is applied to every XML nodes of their input sequence from the *in* clause. They can be divided into several sub sequences and process them in parallel. As shown in Fig. 1, both FLWOR expressions in line 2-4 and line 5-8 are suitable for this approach.

```
1.  let $buy := doc("books.xml")/book
2.  let $book := for $b in doc("bookstore.xml")/book
3.                   where $b/num >30
4.                   return $b
5.  for $b1 in $buy
6.  for $b2 in $book
7.               where $b1/title = $b2/title
8.               return $b1/title, $b2/price
```

Figure 1. An example of XQuery program.

To find out every task which can be executed in parallel, XQuery programs should be translated as a kind of middle representation, which is suitable to express the query plan for parallel execution.

A. Query Plan of XQuery Program

We use data flow diagram to represent the query plan of XQuery program. A data flow diagram is a directed acyclic graph:

Graph= (V, R)

V is the set of nodes, and each node is an independent query task. *R* is the set of directed arc between nodes ($V \times V$). *R* indicates the dependence relationship between two query tasks, and the arc's direction points out the direction of data flow. A task is a node of data flow diagram.

Task = (*expr*, *type*, *arcIn*, *arcOut*, *pipePred*, *pipeSucc*, *subGraph*)

where *expr* is an expression of the query task and *type* indicates that this task whether supports data parallelism and pipeline parallelism. *arcIn* and *arcOut* are the collection of directed arcs, and point out the precursor and the successor tasks of the current task, respectively. *pipePred* and *pipeSuc* indicate precursor and successor tasks of the current pipeline task. *subGraph* shows sub diagrams of the current task.

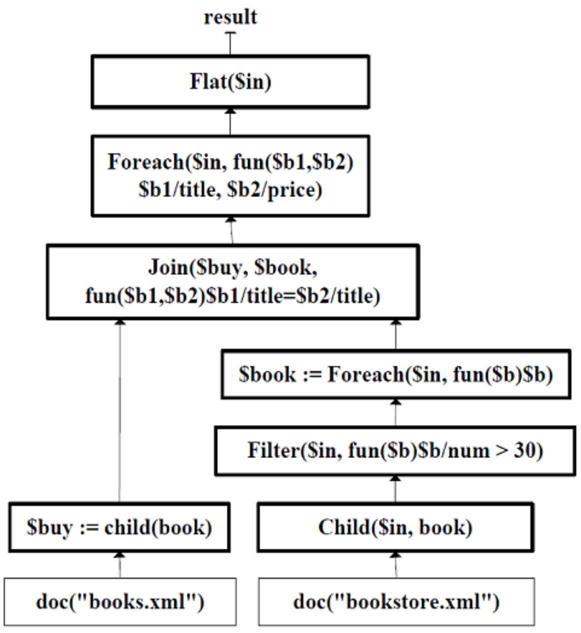
result
Flat($in)
Foreach($in, fun($b1,$b2) $b1/title, $b2/price)
Join($buy, $book, fun($b1,$b2)$b1/title=$b2/title)
$book := Foreach($in, fun($b)$b)
Filter($in, fun($b)$b/num > 30)
$buy := child(book)
Child($in, book)
doc("books.xml")
doc("bookstore.xml")

Figure 2. Data flow diagram of Fig.1.

According to the above definition, the XQuery program of Fig. 1 can be converted into the data flow diagram as shown in Fig. 2. Each node in Fig. 2 is an independent computing task, and the arcs between tasks indicate their data dependence relationship. A task can be executed as long as all tasks it depends on have been completed. Execution order of tasks may be decided by the topological sorting algorithm. In the process of generating data flow diagram, we can distinguish which kind of parallelism can be performed on the task based on a static analysis of its expression. As shown in Fig. 2, it also presents a query plan for the parallel execution of the XQuery program. Some of them may be executed concurrently, and some of them must be executed one by one.

B. Automatic Parallelization Method

In order to use the above three parallel strategies in the implementation of XQuery language, this paper presents a novel automatic parallelization method as shown in Fig. 3. In this approach, XQuery programs are translated into FXQL language, which is a simple middle language and its core syntax is as follows.

$e \rightarrow$ *const | id | if e then e else e | id(e*)*
| fun(id)e | e where id = e*

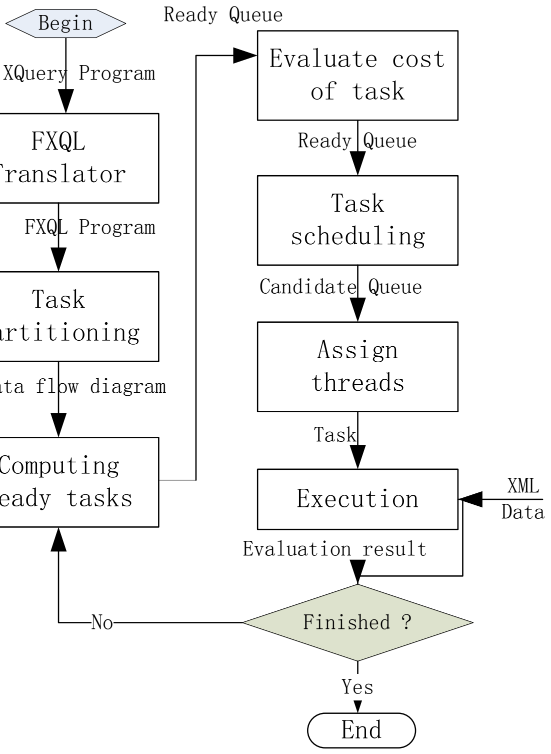

Figure 3. The process of automatic parallelization of XQuery programs.

In FXQL programs, the FLWOR expression of XQuery is converted into the calls of simple functions such as *Join*, *Foreach* and *Filter*, which are the algebraic operators of an XML query algebra. Some logic optimizations can be performed based on the XML query algebra, which will be introduced in other papers. Let's take the following XQuery as an example:

```
for $book in doc("bib.xml")//book
        return $book/title
```

After translated , the FXQL program is as follows:

```
Flat(
        Foreach(
                DescentOrSelf( $var,"book" ),
                fun($book) Child( $book,"title" ) )
        where
                $var = doc("bib.xml")     )
```

where the built-in function *doc* gets the root element of XML file *bib.xml*. Function *DescentOrSelf* get all descendant elements with *book* tag. *fun* represents an anonymous function, and in its body, *Child* function gets all child elements of *book* nodes labeled by *title*. Function *Foreach* applies the anonymous function on each *book* node which is the descendant node of the document node of XML file "*bib.xml*". The node list obtained from the *Foreach*'s application is converted into a node sequence by the *Flat* function.

And then, by the task partitioning, FXQL programs will be split into many tasks. All these tasks form a data flow diagram as shown in Fig. 2, where each task is marked with different parallel strategies. We select the tasks, which is ready to run, into a queue via a topological sort algorithm, and then evaluate the execution cost of each task. Finally, a group of tasks, chose by task scheduling algorithm on the basis of cost of tasks, task character and current available resource, are assigned a number of threads and executed. And the cost evaluation and task scheduling should be done repeatedly whenever some tasks are completed.

III. Task Partitioning

Although each expression of XQuery programs is an independent task, their execution costs vary considerably. In order to get the better parallel efficiency, we shouldn't assign threads to the tasks which execution cost is too low. It is obvious that operations on XML sequence nodes, such as filtration, projection, connection and axis operations, is time consuming. However, arithmetic and relational operations are not time consuming. Our principle of task partitioning is as follows.

- If the expression is a functional call and the called function is a primitive function of query algebra operator, it will be treated as a task.
- If the expression is a functional call and the called function is a user-defined function and the types of its arguments contain an XML node sequence, it will be viewed as a task.
- Each tree pattern query is viewed as an independent task.

FLWOR expression, conditional expression and user-defined function are the logical control mechanisms of XQuery language. As described above, FLWOR expression has been converted into the composition of number of query primitive function's calls. Condition expression is dealt as a special task, and each branch of it is regarded as a subtask. Each user-defined function call is treated as an independent task.

The different parallel strategies may be applied to different tasks. *Filter*, *Foreach*, *Flat* and *Join* functions support data parallelism and pipeline parallelism. Tree pattern query, which usually starts with getting XML sequence nodes and ends with count aggregation functions or node construction function, may take pipeline parallelism strategy. As shown in Fig. 4, *XQParti* algorithm of task partitioning generates the query planning diagrams by analyzing the main expression and user-defined function of FXQL programs. The input data of *XQParti* is a FXQL expression *e*, which is a main query expression or the body of user-defined functions. It is assumed that the current task has been created in advance. The algorithm will be used to identify which part of the expression should be included in the current task and the precursory tasks. *XQParti*'s output data is a three triple *(r, t, g)*. Where, *r* is the subexpression of expression *e* which will be included in the current task. *t* is the list of precursory tasks and *g* is the subgraph of the current task when there is an If branch in the expression.

In the algotithm, *newTask* function is used to create a new task, and *newGraph* function creates a new data flow graph, and ++ means list connection. The environment *w* is used to store the binding of every local variable with its task. The computation of every local variable in an FXQL program should be treated as an independent task since its result may be shared by several tasks.

Algorithm 1. The core algorithm of task partitioning

```
      XQParti: Expr$\rightarrow$Env$\rightarrow$(Expr, Task*, Graph*)
input:  e $\in$ Expr, w $\in$ Env :Id$\rightarrow$Task
output: ( r, t, g ) where r $\in$ Expr, t $\in$ Task*, g $\in$ Graph*
XQParti[ id ]w      =        ( id, w(Id), null )
XQParti[ const ]w =          ( const, null, null )
XQParti[ id($e_1$, ..., $e_n$) ]w =
      if id is query algebraic operator then
          ( x, {t}, null )
      else if id is an user-defined function     then
           if no sequence in id's parameters    then
                ( [ id($r_1$, ..., $r_n$) ], $t_1$++...++$t_n$, $g_1$++...++$g_n$ )
          else
                ( x, {t}, null )
      else           // other primitive functions
          ( [ id($r_1$, ..., $r_n$) ], $t_1$++...++$t_n$, $g_1$++...++$g_n$ )
      where
           ($r_i$, $t_i$, $g_i$) = XQParti[ $a_i$ ]w; for i = 1,..., n
          x$\leftarrow$new variable ;           // create a new task
           t = newTask(x, [id($r_1$, ..., $r_n$)], $t_1$++...++$t_n$, $g_1$++...++$g_n$ );
XQParti[ if $e_1$ then $e_2$ else $e_3$]w =
      ( [ if $r_1$ then $r_2$ else $r_3$ ], $t_1$, $g_1$++$q_2$++$q_3$) )
      where
          ( $r_2$, $t_2$, $g_2$ ) = XQParti[$e_2$]w;
          ( $r_3$, $t_3$, $g_3$ ) = XQParti[$e_3$]w;
          $q_2$ = newGraph( $r_2$, $t_2$, $g_2$ ); // create a new sub graph
          $q_3$ = newGraph( $r_3$, $t_3$, $g_3$ ); // create a new sub graph
          ( $r_1$, $t_1$, $g_1$ ) = XQParti[$e_1$]w
XQParti[ $e_0$ where $x_1$ = $e_1$ ]w =
       if $t_1$ = $g_1$ = empty list           then
          ( [ $r_0$ where $x_1$=$e_1$ ], t0, g0 )
       else
          ( $r_0$, $t_0$, $g_0$ ) = XQParti[ $e_0$ ]$w_0$
      where
          t = newTask( $x_1$, $r_1$, $t_1$, $g_1$ ); // create a new task
          $w_0$ $\leftarrow$ w ++ ( $x_1$ $\rightarrow$ t );
          ( $r_1$, $t_1$, $g_1$ ) = XQParti[ $e_1$ ]w;
XQParti[ other ]w =          ( other, null, null )
```

Figure 4. The core algorithm of task partitioning

In the analyzing of function call *id(e_1,…,e_n)*, the algorithm is applied to its each actual parameter, *n* triples *(r_i,t_i,g_i)* are returned. The expression will be rewritten into *id(r_1,…,r_n)* if no task is created for the call.

If *id* is the name of a query algebraic operator such as *Filter* and *Foreach*, a *newTask* function will be called for creating a new task for the call. And a new variable is also created for rewriting the call in the current expression. All new tasks will be the precursory tasks of the current task.

For the call of user-defined functions, they are processed in the same way as query algebraic operator, if one of its parameter has the type of sequence. Otherwise, no task is created.

In the analyzing of conditional expression, each branch generates a new DFG by calling *newGraph* function. And it is also rewritten in terms of the result expressions of recursive application of the algorithm to each branch and condition expression.

For *where* expressions with the definition of local variable expression e_0 where $x_1=e_1$, a new task will be created for the local variable, and it is treated as the precursory task of current task too. Then variable is bound to the task and expanded to the environment *w*.

As shown in Fig. 4, task partitioning is completed in one parse on the expression, so the algorithm takes time *O(n)*, where *n* is the size of the expression.

Since the algorithm is used before the execution of FXQL program, it is a kind of static analysis. During the analysis, every expression in a FXQL program is rewritten into the expression in a task. The current task for the main query expression and its precursory tasks obtained from the analysis form the query plan for parallel execution of the FXQL program in the form of data flow diagram.

IV. Execution Cost Model

In order to reasonably assign the computing resources to each task, the amount of data should be taken into account. Since XML data come from the Internet, XQuery programs have to handle large data size. For the parallel execution of XQuery programs, the calculating amount of each task is determined by the data size of its input data. Therefore, to decide which task should acquire computation resources, the execution cost of a task has to be evaluated based on the data size of its input at running time.

The paper puts forward a dynamic execution cost model for the purpose. In XQuery execution, each task will be assigned threads according to the cost evaluated by the cost model. In our cost model, cost calculating is based on the information of input data sequence of tasks. The cost of a task is mainly decided by the length of input sequence *sqln* and the count of its descendant nodes *elsz*. The computation rule of cost is shown in Fig. 5. In the algorithm, the values of all expression are sequences of XML nodes. Thus, Len[*e*]*w* means the sequence length of the value of the expression *e*, and Size(*e*) is the number of descendant nodes of the result sequence. An environment *w* is used to provide the context information. Every local variable is bound to its sequence size and descendant number *id*→(*sqln*, *elsz*) in the context environment. In the initialization of the environment, *sqln* and *elsz* of each variable should be computed with the value of the variable, which come from the execution of the corresponding precursory task.

Based on the first evaluation rule for the cost of an expression Cost[e]w, the cost of an expression is the sum of the costs of its sub-expressions in most cases, as shown in the Fig. 5. For example, the cost of a function call is the sum of the costs of every parameters and the cost of the body of the function's definition.

In the algorithm, *body(id)* represents the body of the definition of the function *id*, *var(id,i)* means the *i*'th argument of the function *id*.

For the query algebraic operators, the cost of these primitive function's calls is the product of the length of the input sequence and the cost of the body of their iterate function. The former is computed by the rule Len[e_1]*w*, and the latter will be evaluated under the new context including the binding of their loop control variable. Since the lengths of the member of the input sequence are different, it is assumed to be the square root of Size(e_1), which is the number of the descendants of the result nodes gotten from the source expression e_1.

Duo to the same reason, we also assume the cost factors *kid*, which means the computation strength, and *hid*, which means the number of result nodes, for each axis operations and built-in functions, respectively.

The evaluation rule for result length of an expression Len[*e*] describes how to get the length of the XML node sequence which is the value of an expression *e*. Since the value of an expression is a node sequence, Size(*e*) can be computed from the XML tree.

As shown in Fig. 5, generally, the time complexity of both Cost[*e*]*w* and Len[*e*]*w* is linear with the size of task expressions. In the implementation of the algorithm, we also consider the recursive function and guarantee the termination of the cost computation.

V. Task Scheduling

As discussed in Section III, an XQuery program has been divided into many computation tasks. It is easy to get a group of tasks that are ready for being executed via the topological sort algorithm. Our task scheduling method is that we put all tasks that are ready into a ready queue and then compute their costs. Finally, some of them are moved from the ready queue to a candidate queue and assigned number of threads to execute. Whenever there is a task is over, ready queue will be updated and the above task scheduling process is repeated.

Our task scheduling algorithm has the following features: *(1)* Assigning threads based on the cost of tasks, *(2)* Pipeline parallelism is superior to other approaches, and *(3)* Allocating more threads for data parallelism tasks.

As shown in Fig. 6, there are three inputs for the task scheduling: *seqReady* and *num*. *seqReady* is a task queue of task which includes the tasks in ready condition and *num* is the number of available threads. Variables *seqCondi*, *seqReady* and *num* are used as the outputs of the algorithm. *seqCondi* is a task queue containing the tasks to which threads have been allocated. When the task scheduling is completed, the allocated threads will execute the tasks in the queue *seqCondi*. And the tasks with no thread remain in the queue *seqReady*. The number of the remaining threads is returned by the variable *num*.

Procedure of the task scheduling algorithm can be divided into two phases. In first phase, the task with the highest cost in the ready queue *seqReady* will be selected and put into the candidate queue *seqCondi*. If the task supports pipeline parallelism, all tasks in the pipeline will be moved to the candidate queue *seqCondi* as much as possible. The process will be repeated until no task exists in the ready queue or all threads have been allocated.

In second phase, the threads that are occupied by tasks with smaller running cost are reallocated to those with higher running cost if the latter support data parallelism. The task supporting data parallelism should have more than one thread. This kind of reallocation must assure that each moved task doesn't support pipeline parallelism and its running cost is more than double the average cost of the tasks in the candidate queue *seqCondi*. In this way, every thread which has been allocated to the tasks in the candidate queue will be used to perform their subtask with similar running cost.

```
Algorithm 2：The main rule of cost evaluation model
(1) Evaluation rule for the cost of an expression.
        Cost: Exp → CEnv → Number
        where   CEnv=Id→(Number, Number)
Cost[ id ]w       = 1
Cost[ const ]w    = 1
Cost[ if $e_1$ then $e_2$ else $e_3$ ]w =
        Cost[$e_1$]w + max( Cost[$e_2$]w+Cost[$e_3$]w)
Cost[ $e_0$ where id=$e_1$ ]w =
        Cost[$e_1$]w + Cost[$e_0$]w'
        where   w' = w ++ <id→(Len[$e_1$]w, Size($e_1$))>
Cost[ id($e_1$, …, $e_n$) ]w =
    if  id is a query algebraic operator
          and $e_2$ is an iteration function    then
        Cost[$e_1$]w + Len[$e_1$]w * Cost[body($e_2$)]w'
        where
            w' = w++ <var($e_2$)→(sqr(x), sqr(x))>
            x = Size($e_1$)
    else if  id is user-defined function        then
        Cost[body(id)]w' + Cost[$e_1$]w+…+Cost[$e_n$]w
        where
            w'=w ++ <var(id,1)→(Len[$e_1$]w,Size($e_1$))> ++ …
                  ++ <var(id,n)→(Len[$e_n$]w,Size($e_n$))>
    else if id is axis operation        then
        Cost[$e_1$]w + $k_{id}$*Len[$e_1$]w
    else
        kid + Cost[e1]w + … + Cost[en]w
 (2) Evaluation rule for result length of an expression:
        Len: Exp → CEnv → Number
Len[ id ]w = 1st(w(id))
Len[ const ]w = 1
Len[ if $e_1$ then $e_2$ else $e_3$]w =
        max(Len[$e_2$]w, Len[$e_3$]w)
Len[$e_0$ where id = $e_1$]w =
        Len[$e_0$]w'
        where w' = w + id→(Len[$e_1$]w, Size($e_1$))
Len[id($e_1$,..,$e_n$)]w          =
    if  id is a query algebraic operator    then
        Len[$e_1$]w
    else if  id is axis opertion   then
        Len[$e_1$]w * $h_{id}$
    else if  id is user-defined function        then
        Len[body(id)]w'
        where
            w'= w++<var(id,1)→(Len[$e_1$]w),Size($e_1$))>
                ++…++<var(id,n)→(Len[$e_n$]w,Size($e_n$))>
    else   $h_{id}$
```

Figure 5. The core algorithm of cost evaluation model

VI. Implementation

Based on the task scheduling approach, we have implemented a parallel XQuery engine in Java. In the implementation, we use the Fig. 4 to perform task partitioning on the FXQL programs which are translated from XQuery program. Before the task partitioning, some optimizations are applied to FXQL program by program rewritten. The execution process of XQuery program in the parallel engine can be described by Fig. 7. The running cost model and task scheduling are used in the algorithm.

In the algorithm, the function *Translate* is used to convert an XQuery program into a FXQL program, function *Optimization* rewritten the FXQL program for better performance. It also uses the funcition *createDFG* to create a data flow diagram from the tasks which is generated by the task partitioning TQParti presented in section III. Variable *seq* means the ready queue and *seq'* is the candidate queue.

The core of the algorithm is the while loop. The ready task is gotten from the DFG *flow* by the function *getReady*. Through the task scheduling TSchedul presented in section V, the selected tasks are moved to the *seq'*, other tasks remain in *seq*. Then the tasks in the candidate queue *seq'* are executed in parallel. Whenever any thread end, more tasks are selected from the DFG. The thread will be allocated to them in the same way. Until no thread is active and no task exists in the ready queue, the execution is terminated.

An execution environment *env* is used for storing the result values of every task by variable binding. The result of the query expression is bound to the special variable *result* which is used to return the query result in the last statement.

```
Algorithm 3：The Core Algorithm of Task Scheduling
TSchedul：TSequence → Integer
                    → TSequence×TSequence×Integer
Input：   seqReady : TSeqence //ready queue
          num      : Integer   //thread number
Output：  seqCondi:  TSeqence //candidate queue
          seqReady: TSeqence  //ready queue
          num      : Integer   //number of left threads
Begin
while num > 0 and seqReady isn't empty do
  task ← the highest cost task of seqReady
  allocate a thread to task(num--);
  move task to seqCondi queue;
  if task supporting pipeline then
        len ← the length of task pipeline - 1;
        if num > len then
          allocate len threads for other tasks;
          move to seqCondi queue;
          num ← num – len;
end of while;
foreach task in seqCondi
  if task supporting data parallelization   then
     last ← the minimum cost task of seqCondi
     while last exist do
        if last dost not supporting pipline or its pipline is not in seqCondi and cost(task)> 2×aveCost(seqCondi) then
            if num=0 then
               move last from seqCondi to seqReady
               num++；
            increase a new thread for task(num--)；
            last← next minimum cost task of seqCondi
     end of while;
end of foreach;
End
```

Figure 6. Task scheduling algorithm

VII. Experiments

We have tested the performance of the implementation of the parallel XQuery query engine. All experiments run on HP Z600 workstation, which has two Xeon E5504 CPUs (four cores per CPU) and 4G RAM. Test cases are from W3C (http://www.w3.org/TR/2007/NOTE-XQuery-use-cases-20070323/).

As shown in Fig. 8, they are performance tests. We choose seven XQuery test cases and run them on different size XML files with different number of threads.

Experiments show that the execution efficiency of XQuery programs has been improved to varying degrees in most case. The performance of these is much better when the number of threads is equal to or more than four. Since these test programs have different structures and different size, the implicit parallelism in these programs vary considerably.

When the size of XML files increases to 104Mb, the serial execution of the some programs (Q4, Q5 and Q6) fails with the error of JVM memory overflow. But their parallel executions perform their regular function and the running speeds is increased as the more thread is used.

On the other hand, since the overhead of the task scheduling and the evaluation of running cost, the performances of some test cases with two threads are not significantly improved. It should be improved in future works.

As shown in Fig. 9, we also compared the execution time under different parallelism strategies. Five kinds of parallelism strategies are used in the experiment, including serial execution (S), task parallel execution (TP), data and task parallel execution (DTP), pipeline and task parallel execution (PTP) and task, data and pipeline parallel execution (PDTP). Only one or two strategies are used for the test programs with different number of threads.

Experiments show that the data parallelism obviously increases the performance of XQuery programs. Pipeline parallelism is helpful in some case and helpless for others, while it is useful to avoid memory overflow as shown in Fig. 9(c). Task parallelism brings the performance improvement of XQuery programs a little, since no complex structure is used in these tests. More complex programs should be tested in future works.

```
Algorithm 4:  parallel execution of XQuery program
   ParallExcute:  Expr → Num → Value
Input:    query∈ Expr        // query expression
          num∈Num            // number of thread
Output: value∈Value          // instance of XDM
Variable:  exp∈Exp           // FXQL expression
          flow∈Graph         // Data Flow Graph
          seq,seq'∈TSequence        // task queue
          env∈Env=Var→Value
Begin
exp = Translate( query );     // from XQuery to FXQL
exp = Optimization( exp );  // optimization
(r, t, g) = XQParti( exp );     // task partitioning
flow = createDFG(r, t, g);   // generate DFG
seq = getReady(flow,  nil);
env = initEnv(flow, "result");
while  seq isn't empty and no thread is active  do
      (seq', seq, num) = TSchedul(seq, num);
      Start threads in each task in seq';
      if  any thread end   then
            store its result in env;
            num = num + 1;
            seq = getReady(flow, seq)
end of while
value = env("result");
```

Figure 7. Algorithm of parallel execution of XQuery parogram

VII. Conclusions

In order to improve the performance of XQuery programs, this paper presents an automatic parallelization method based on its function language features. We divide the XQuery program into many computable tasks, and develop a new scheduling strategy, which considers the running cost of each task and the parallelism strategy it can support. Different task may use different parallel strategy, including task parallelism, pipeline parallelism and data parallelism. Experimental results show that this parallel approach may use computing resource of multi-core environment efficiently, and improve the execution efficiency of XQuery programs. As future works, more static analysis and dynamic analysis should be taken into account for the improvement of the task assignment, task scheduling and cost evaluation.

ACKNOWLEDGMENT

This work was supported by a grant from Beijing Natural Science Foundation (4122011).

REFERENCES

[1] Viet Hung Nguyen and Tran Khanh Dang . “A Novel Solution to Query Assurance Verification for Dynamic Outsourced XML Databases”. Journal of Software. 2008, vol. 34 (4), pp. 9-16.

[2] Ying Liu, Fuxiang Gao and Shiyuan Wang. “Parallel Implementation of Xvid Decoder on Multi-Core”. Journal of Computers. 2012, vol. 7(7), pp. 1639-1646.

[3] H. Michael and G. Madhusudhan. “Approaching a Parallelized XML Parser Optimized for Multi-Core Processors,” 16th Int. Symp. High Performance Distributed Computing., New York, 2007, pp. 12-25.

[4] W. Lu, K. Chiu and Y. Pan. “A parallel approach to XML parsing,” Proc. of the 7th IEEE/ACM Int. Conf. on Grid Computing, 2006, pp. 223-230.

[5] K. Lu, Y. Zhu and W. Sun. “Parallel processing XML documents,” Proc. Int. Database Engineering and Applications, 2002, pp. 96-105.

[6] B. Rajesh and L. Lipyeow. “Parallelization of XPath queries using multi-core processors: Challenges and experiences,” 12th Int. Conf. on Extending Database Technology: Advances in Database Technology, 2009, pp.180-191.

[7] M. Imam, A. Toshiyuki and K. Hiroyuki. “XML data partitioning strategies to improve parallelism in parallel holistic twig joins,” 3rd Int. Conf. on Ubiquitous Information Management and Communication, 2009, pp.471-480.

[8] H. W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, and et al. “Comparing parallel functionallanguages: Programming and performance,” Higher order and symbol computation, 2003, vol. 16(3), pp.203–251.

[9] T. Harris and S. Singh. “Feedback directed implicit parallelism,” Proceedings of the 12th ACM SIGPLAN Int. Conf. on Functional programming, 2007, pp. 251-264.

[10] S. Marlow, Si. Peyton Jones and S. Singh. “Runtime support for multicore Haskell,” ACM SIGPLAN Notices, 2009, vo. 44(9), pp. 65-78.

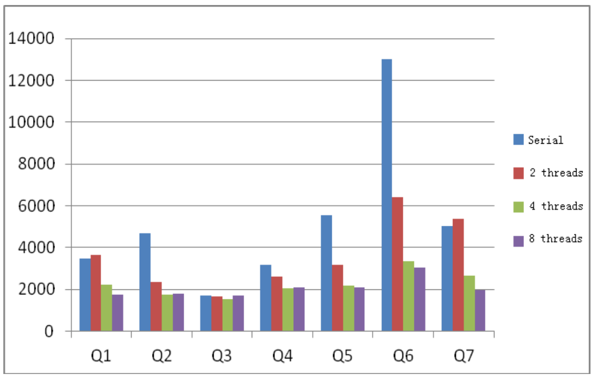

Figure 8 (a). Performance Test Result on 32MB XML

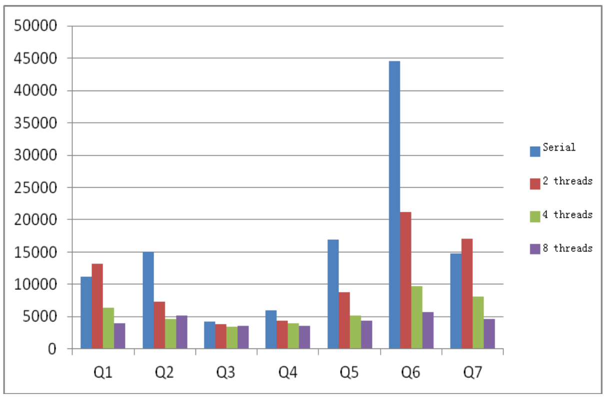

Figure 8 (b). Performance Test Result on 64MB XML

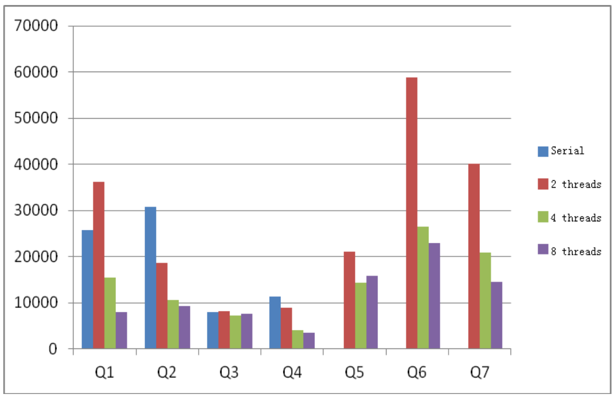

Figure 8 (c). Performance Test Result on 104MB XML

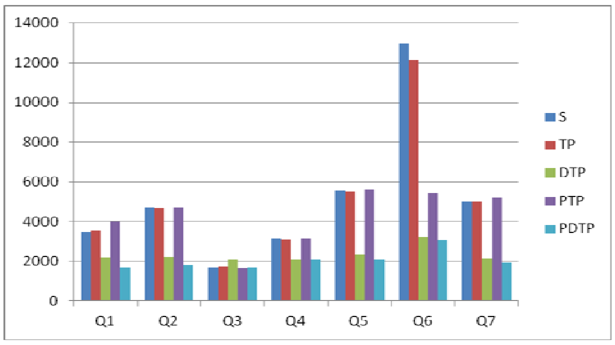

Figure 9(a). Execution time on 32MB XML File

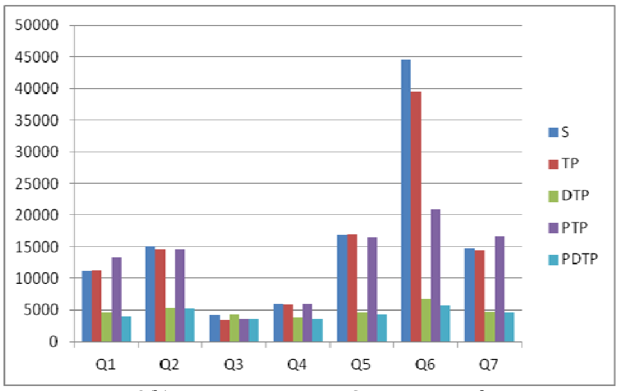

Figure 9(b). Execution time on 64MB XML File

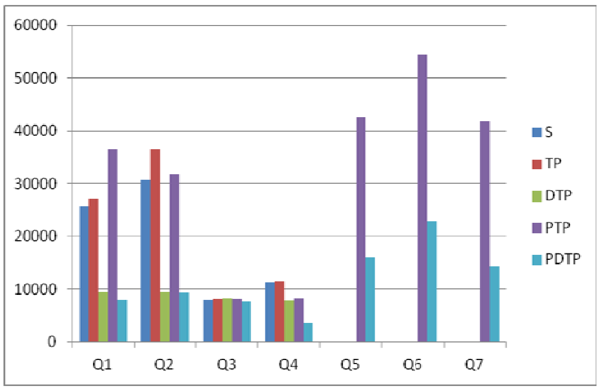

Figure 9(c). Execution time on 104MB XML File

Husheng Liao received his M.S degree from TsingHua University in 1981. He is currently a professor of Beijing University of Technology. His main research interests include complier, program languages and XML.

Weifeng Shan is a PhD student at the Beijing University of Technology with research interests in XML and parallel computing. He has received his Master’s degree in Software Engineering from the YunNan University, China.

HongYu Gao was born in Beijing, China in 1968, and received his Bachelors and Master degree in computer science from Beijing University of Technology, China in 1992 and 1998 respectively. He is an Associate Professor in College of Computer Science, Beijing University of Technology.