Scientific Research and Essays Vol. 6(18), pp. 3914-3926, 1 September, 2011
Available online at http://www.academicjournals.org/SRE
DOI: 10.5897/SRE11.444
ISSN 1992-2248 ©2011 Academic Journals

Full Length Research Paper

Load balancing parallelizing XML query processing based on shared cache chip multi-processor (CMP)

Wanli Zuo[1]*, Yongheng Chen[1,2], Fengling He[1,2] and Kerui Chen[1,2]

[1]College of Computer Science and Technology, Jilin University, Changchun, 130012, China.
[2]Key Laboratory of Symbolic Computation and Knowledge Engineering of the Ministry of Education, Jilin University, China.

Accepted 27 June, 2011

Chip multi-processor (CMP) could support more than two threads to execute simultaneously, and some executing units are owned by each core. Based on CMP, this paper proposes a novel and complete optimization framework on parallelism for XML database multithreaded query processing that strives for maximum resource utilization. Firstly, a set of algorithms for constructing the parallel sub-query plans and partitioning XML document by parallel sub-query plans are proposed. Furthermore, in order to reduce cache access conflict and address the imbalance of threads' workload, we refine the granularity of partitioned XML document and balance the workload assignment by executive pairs for the unit. Finally, by building the execution plan tree of sub-query plans constructed, the partial solution produced by parallel sub-query plans are merged into final solution. Our theoretical analysis and empirical evaluation show that the proposed algorithm could impressively improve in the performance of XML query processing.

Key words: Chip multi-processor, parallel query processing, XML query optimization.

INTRODUCTION

Extensible markup language (XML) has been emerging as a de facto standard for data representation and exchange among various applications on the internet. An XML document employs a tree-structured model whose nodes are labeled with tags for representing data. Finding all occurrences of a query pattern in XML documents is one of the core operations in XML query processing, both in relational implementations of XML databases, and in native XML databases. Unlike (flat) text documents, XML documents have nested structure (Gou and Chirkova, 2007). Thus the content and the structure of XML data are concerned by XML queries.

Many algorithms have been studied for XPath query processing. Al-Khalifa et al. (2002) proposed a structural joint algorithm, which takes two ordered lists as input, one for ancestors and the other for descendants. To address this problem and process the twigs in XPath without creating large intermediate results, Bruno et al. (2002) proposed a holistic twig join approach for matching XML query twig patterns without creating large intermediate results. Furthermore, many improved holistic twig join algorithms based on Bruno et al. (2002) are proposed, such as TSGeneric (Jiang et al., 2003), TJFast (Lu et al., 2005), iTwigJoin (Chen et al., 2005) and so on.

Meanwhile, on the hardware front, the development trend of processor (Hennessy and Patterson, 2007) is transforming from high-speed single-core to chip multi-processor (CMP), and from instructions parallel to thread level parallel. Tomorrow's computer will have more cores rather than exponentially faster clock speeds, and software designs must be restructured to exploit the new architectures fully.

However, all the above algorithms have a common characteristic: they are proposed for single-core central processing units (CPUs). They cannot take full advantage of multi-core CPUs. To take advantage of multi-cores,

*Corresponding author. E-mail: wanli@jlu.edu.cn.

Abbreviations: CMP, Chip multi-processor; **XML,** extensible markup language; **CPUs,** central processing units; **PL,** predicate node to the linear node; **LL,** linear node to the linear node.

efficient parallel algorithms are desirable for evaluating queries. It therefore presents both opportunities and challenges in the design of XML query processing algorithms. The question for XML database researchers is this: how best can we use this multithreading capability to improve XML database performance in a manner that scales well with machine size?

To tackle this problem and take advantage of multi-cores, there are various methods using XML data partitioning to process parallel XML query processing that have been extensively studied on multi-core systems. Kurita et al. (2007) achieve parallelism by splitting the input query into serial and parallel sub-queries. Bordawekar et al. (2009, 2010) propose three schemes for parallelizing XML queries: Data partitioning, query partitioning, and hybrid (query and data) partitioning. Machdi et al. (2010) present task parallelism for TwigStack algorithm using a pipelining technique by decomposing the TwigStack algorithm into two major tasks. Feng et al. (2010) propose an efficient parallel PathStack algorithm for processing XML twig queries.

These papers as a whole realize parallelism either by rewriting query plan or by decomposing the XML document according to the simply root-to-leaf sub-queries. To the best of our knowledge, none of the above literatures discusses how to integrate the optimal parallel query plan rewriting and the XML document decomposing in accordance with the specific parameters, for example, the capacity of L2_cache and the number of threads, of the CMP system. Motivated by the e-market application case and the aforementioned works above, in this paper we consider and explore efficient multithreaded XML query optimization model based on CMP characteristic.

In this paper, we firstly travel the XPath query plan tree node and calculate the incremental cost of each node by the defined cost model. Using this method we can determine partitioning scheme of every node in the XPath query plan tree. Then the parallel sub-query plans can be constructed in accordance with these partitioning nodes. As a thread must be retained to merge the partial solutions into final solution, the max number of the sub-queries is set to be the number of threads minus one. Secondly, we use the sub-queries to guide the XML document decomposing. When the total size of the partitioned XML documents is more than the capacity of L2_cache, more fine-grained decomposition is considered on these partitioned XML document fragment. When data are seriously uneven among the decomposed XML document according to the parallel query plan, the workload of parallel query plans would be skewwed. Parallel performance will be reduced, because serious discrepancy of the workload allocated to the parallel query plan would lead to load imbalance among threads in CMP. Therefore we further consider developing the thread workload scheduler module by the adjustment of executive pairs. We achieve this by exploiting accommodative load balancing algorithm. Finally, threads are to merge the partial solutions produced by parallel from the parallel sub-query plans for final solutions parallel to the execution strategy tree of sub-query plans constructed.

The main contributions are outlined as follows:

To the best of our knowledge, this is the first study to integrate the optimal parallelizing query plan rewriting and the XML document decomposing in accordance with the specific parameters of the CMP system.

We construct and index the execution strategy tree of sub-query plans to guide the final solution generating.

More importantly, we propose a novel method to optimize the workload-balancing assignment algorithm among threads to fully utilize CMP.

We implement our parallelization algorithm and provide experimental results that validate the effectiveness of our proposed parallelization algorithm.

Related work

In this section, we briefly review some related work of XML query processing that have been extensively studied under the following four categories: XPath query processing, the partitioning approach of large XML document, the parallelizing approach of the XML query plan, and the related parallelizing technology.

Many algorithms have been studied for XPath query processing. Prior works (Al-Khalifa et al., 2002; McHugh and Widom, 1999; Zhang et al., 2001) have typically decomposed the pattern into a set of binary structural relationships between pairs of nodes. The query twig pattern can then be matched by matching each of the binary structural relationships against the XML database, and finally stitch together the results from those basic matches. The main disadvantage of such a decomposition based approach is that intermediate result sizes can become very large, even when the input and the final result sizes are much more manageable. To solve this problem, Bruno et al. (2002) proposed a holistic twig join approach (referred as TwigStack) for matching XML query twig patterns. With a chain of linked stacks to compactly represent partial results of individual query root-to-leaf paths, their approach merges the sorted lists of participating element sets altogether, without creating large intermediate results. They answer the twig query holistically and avoid huge intermediate results. Furthermore, many holistic twig join algorithms based on TwigStack are proposed, such as TSGeneric (Jiang et al., 2003), TJFast (Lu et al., 2005), iTwigJoin (Chen et al., 2005).

There are some suggestions on how to split and distribute large XML documents. Bordawekar et al. (2009) propose three schemes which achieve parallelism via partitioning traversals over the XML documents for parallelizing XML queries. The data partitioning approach executes the same (sub) query on different sections of the same XML document whereas the query partitioning approach executes different (sub) queries on the same XML dataset. Lu and Gannon (2007) present five different algorithms to split a large XML document into a fixed number of XML fragments in order to cope with different characteristics of XML tree structures for parallel XPath query processing.

Bordawekar et al. (2010) research the optimal way of parallelizing an XML query plan by a novel, end-to-end parallelization framework based on a statistics approach that relies both on the query specifics and the data statistics over shared-address space multi-core processors. For a given XPath query, every node in query plan is estimated the relative efficiencies of their different. According to these candidate partitioning points, an optimal parallel XPath processing plan is constructed.

In addition, Lu and Gannon (2007) proposed a parallel processing model for the XML document on a multi-core computer. The dynamic load-balancing mechanism based on stealing is the core of the model, in the light of which the disjoined parts of the XML document can been processed by multiple threads in parallel with balanced load distribution. Li et al. (2006) proposed an even partition based method, which accelerates structure joint dramatically, but requires neither the order of input element sets nor any indices. Moreover, when the distribution of elements of a particular tag/label is skewwed, general partition-based techniques are not cost-efficient, however, this method partitions the two input sets into different buckets evenly and only the structure joint of suit buckets is helpful to the result, therefore it avoids scanning the two input sets many times and is efficient.

It can be noticed clearly that none of the works aims at parallel query processing based on integrating the optimal parallel query plan rewriting and the XML document decomposing is in accordance with the specific parameters of the CMP.

Determine parallelizing strategies of query plan

In this section, a set of algorithms about how to construct parallel sub-query plans and the execution plan tree are introduced.

Cost model

Preliminaries

In order to estimate the cost for the XPath expression, it is sufficient to count the number of single node and node-node pair with predicates in the XML document. The summarized path trees and Markov can been used to count these. We briefly introduce this method (for example, the Markov model) proposed in Aboulnaga et al. (2001).

A path tree summarizes an XML data tree by aggregating every sibling having the same tag into a single node annotated by a count of the number of occurrences in the original XML data tree. Every node in the path tree represents a path starting from the root of the XML document. The root node of the path tree represents the root element of the document. Every path tree node is labeled with the tag name of the elements reachable by the path it represents and with the number of such elements, which we call the frequency of the node.

To estimate the selectivity of a query path expression using a summarized path tree, we try to match the tags in the path expression with tags in the path tree to find all path tree nodes to which the path expression leads. The estimated selectivity is the total frequency of all these nodes. A Markov of order (m-1) is a table storing a set of distinct paths in the XML data up to length (m) along with their corresponding selectivity where m is a parameter ≥ 2. The table provides selectivity estimates for all path expressions of length up to m.

Cardinality

To construct an efficient query plan for executing a query in a database, it is necessary to know the cardinality of the intermediate results. Intermediate result size is an important factor in estimating the cost of a query plan. The cardinality of a step in an XPath expression is the number of nodes in the XML data tree that satisfy the conditions of that step. The information given by Markov table can been used to estimate the cardinality of each step in XPath.

Considering an XPath expression Q = $/v_0[p_0]/v_1[p_1]/\ldots /v_i[p_i] /\ldots /v_n[p_n]$ with predicates or empty predicates. Let q = $/v_0/v_1/../v_i/\ldots /v_n$, where, each v_i is either a tag or the wildcard *. Let q_i denote the sub-expression of q up to step v_i and Q_i denote the sub-expression of Q up to step $v_i[p_i]$. Let p_i denote the predicate of v_i. Then, the cardinality of Q_i is estimated by the multiple of cardinality q_i and the cumulative product of selectivity up to p_i. Equation 1 gives the formula calculating the cardinality of Q_i.

$$Card(Q_i) = \begin{cases} Card(q_i) * \prod_{k=0}^{k=i} sel(p_k) & i \neq 0 \\ 1 \qquad i = 0 \end{cases}$$

$$Card(q) = \prod_{j=1}^{j=i} f(t_j \mid t_{j-1})$$

$$sel(p_i) = min(f(v_{i+0} \mid v_i), 1) \prod_{j=1}^{m} min(f(v_{i+j} \mid v_{i+j-1}), 1) \tag{1}$$

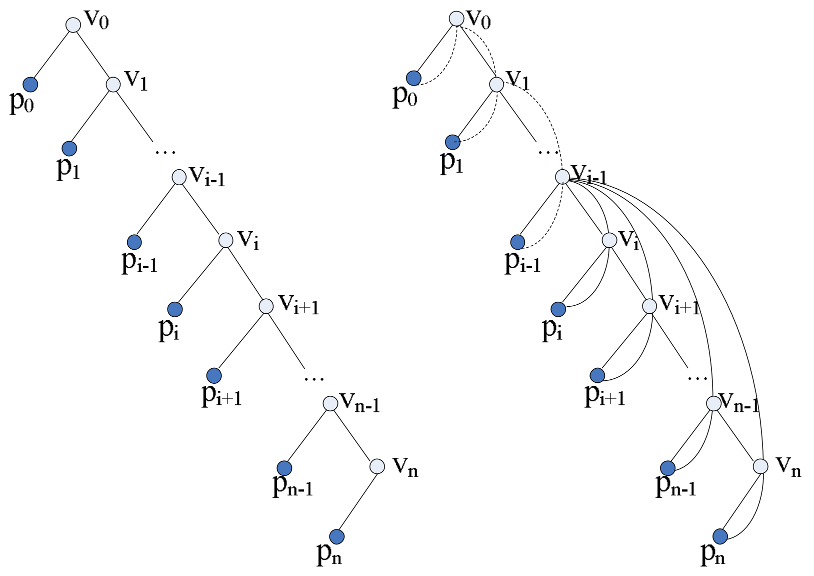


Figure 1. The left part uses a tree to express the XPath query. The right part is used to illustrate the process calculating the sequential cost of v_i.

Cost model

Considering an XPath expression Q = $/v_0[p_0]/v_1[p_1]/\ldots /v_i[p_i] /\ldots /v_n[p_n]$ with predicates or empty predicates, we use a tree in the left part of Figure 1 to express *Q*. This tree includes two types' node, linear node v_i and predicate node p_i. There are two type paths, the predicate node to the linear node and the linear node to the linear node (referred to as *PL* and *LL*), correspondingly.

SCost (v_i) is defined as the sequential cost of traversing the remaining path starting at v_i. The right part in Figure 1 depicts the calculation of *SCost* (v_i). The dotted line is used to calculate the cardinality of Q_{i-1}. The solid lines are used to calculate the average cost of each node in the cardinality of Q_{i-1}. Each solid line includes two type paths. The *LL* path is used to calculate the cardinality from the starting linear node to the ending linear node, and the *PL* path is used to calculate the cost processing the current predicate. In addition, for each *LL* path, we need scan all the children of the ending linear node. The cost of scanning with processing a child is denoted by *Cscan*. So we can calculate the *SCost* (v_i) by the following Equation 2.

$$card(Q_{i-1})[\prod_{j=i}^{n} f(v_j|v_{j-1}) + \sum_{m=i}^{n}\prod_{j=i}^{m} f(v_j|v_{j-1})SCost(P_m)$$
$$+ \sum_{m=i}^{n}\prod_{j=i}^{m} f(v_j|v_{j-1})f(*|v_m)Cscan] \qquad (2)$$

When i = n and pi = empty, SCost(vi+1) = 0 cost(pi) = 0 f(*|vi)Cstep = 1;

When i = n and pi is not empty, SCost (vi+1) = 0.

Determine parallelizing query plan strategy

Incremental Cost

In order to determine the partitioning type of query plan nodes, incremental cost is introduced here. Incremental cost is the max income level selected for partitioning method, data partitioning and query partitioning, relative to sequential processing for one node in query plan. Incremental cost of query partitioning relative to sequential processing is referred to as *ICost_QS*, and data partitioning relative to sequential processing as *ICost_DS* correspondingly. *ICost_QS* and *ICost_DS* are defined as Equation 3.

$$ICost\_QS = \partial Scost(p_i) - \underset{Qn}{\mathrm{OptimalQCost}}[\partial Max\cos t(p_i) + QP\cos t]$$
$$ICost\_DS = Scost(v_i) - \underset{Dn}{\mathrm{OptimalDCost}}[1/Dn * Scost(v_i) + DP\cos t]$$
$$\partial = card(Q_{i-1}) f(v_i|v_{i-1})$$

(3)

In this equation *QPcost* includes the overhead associated with partitioning cost among *Qn* threads and processing the operators cost in predicate. *OptimalQCost* is used to calculate the partitioning number. If the numberof operators in predicate is more than the number of cores minus 1, *Qn* is set up for the number of cores minus 1, otherwise, the number of operators in predicate. *DPcost* includes the overhead associated with partitioning the cardinality of *Qi* among *Dn* threads. As the number of *Dn* increases, *1/Dn*Scost(vi)* decreases, but, the cost of *DPcost* increases. *OptimalDCost* is used to calculate the optimal partitioning number.

Based on the incremental cost equation, we can estimate the partitioning type for every node in query plan using *InCost_Calculating* algorithm. The partitioning plan of every node is encoded by region encoding in the following format (incremental cost, partitioning type, number of partitioning, partitioning node). The variables *S*, *D* and *Q* of partitioning type represent the sequential processing, data partitioning and query partitioning method, correspondingly.

```
 InCost_calculating
Input: XPath query plan.
Output: An incremental cost weighted XPath tree.
Travel this tree from top to down;
For every linear node i=0 to n
  If the predicate of this linear is empty
    ICost_QS=0;
  Else
    Calculate ICost_QS according to Eqn.3;
    Calculate ICost_DS according to Eqn.3;
  If ICost_QS ≤ 0 & ICost_DS ≤ 0
    Add weight (0, S, 0, i) to this linear node;
  If ICost_QS ≤ 0 & ICost_DS ≥ 0
    ICost = ICost_DS;
    Add weight (ICost, D, Dn, i) to this linear node;
  If ICost_QS ≥ 0 & ICost_DS ≤ 0
    ICost = ICost_QS;
    Add weight (ICost, Q, Qn, i) to this linear node;
  If ICost_QS ≥ 0 & ICost_DS ≥ 0
     ICost = ICost_DS;
     Add weight (ICost, D, Dn, i) to this linear node;
     ICost = ICost_QS;
     Add weight (ICost, Q, Qn, i) to this linear node;
```

Determine parallelizing query plan strategy

Given an incremental cost weighted XPath tree called *SETree_MW*, in this sub-section, parallelizing sub-query plans and the execution strategy tree will been constructed.

```
PEP_Constructing
    Input: SETree_MW, the number of parallel sub-query plan ns, the number of cores minus 1 m and
           XML document.//Initial value of ns is 1.
    Output: Sub-query plans and execution strategy tree.
    /*Two-dimensional array IC_ranked is used to store the linear node desceding ICost.*/
    ICost_NoteList=InCost_Calculating;
    For all linear node in ICost_NoteList their ICost more than zero
      IC_ranked=Ranking_Descending (these linear nodes);
      For i=0 to n do
        ns=ns*IC_ranked(i,3);
        If IC_ranked(i,1)>0 and ns<m
          Put IC_ranked(i) into NoteList;
    If NoteList is not empty
      Construct the parallel sub-query plans and execution strategy tree according to NodeList;
End.
```

Firstly, the multi-weighted linear node in *SETree_MW* will be ranked according to the *ICost*. Then a node with the max incremental cost in ranked list will be selected. If the partitioning number is less than the number of cores minus 1 and the max incremental cost is more than zero, the linear node will be stored into a set. Repeating the first step until these conditions is not satisfied. Finally, the selected linear node will be used to construct parallel sub-query plans and the execution plan tree of sub-query plans. *PEP_Constructing* algorithm is introduced to achieve this processing.

The execution plan of sub-query plans is a tree using (*LevelNum, startPos:endPos, LNode*) to label nodes. This tree has two types node. The leaf nodes present the sub-query plans. Other nodes express the buffer node storing the intermediate result. *LevelNum* is the level of a certain element in this tree. *startPos* and *endPos* are calculated by performing a perorder traversal of this tree; *startPos* is the number in sequence assigned to a node when it is first encountered and *endPos* is equal to one plus the *endPos* of the last element visited. Leaf nodes have *startPos* equal to *endPos*. *Node A* is a descendant of node *B* if and only if *startPos(A)* > *startPos(B)* and *endPos(A)* < *endPos(B)*. *LNode* is the element of *NodeList*. The children nodes are incorporated to current node in accordance with *LNode*. The sibling nodes in this tree merge in accordance with the *LNode* of parent node. The *LNode* for leaf nodes is empty.

The example of the implementation of constructing parallel sub-query plans is shown in Figure 2. Figure 2(a) is the multi-weighted tree of XPath *A/B[G operator H]/C[L operator K]/D*. Firstly, we will rank the linear node according to the *ICost*, construct *IC_ranked {(8,Q,2,1)(3,D,2,3)}*. *C* is not selected because the product of partitioning number of *B* and *C* is more than 5. Figures 2(b) and (c) are the results of partitioned Figure 2(a) according to *B (8, Q, 2, 1)*. Figures 2(d) and (e) are the results of partitioned Figure 2(b), Figures 2(f) and (g) are the results of partitioned Figure 2(c) according to *D (3, D, 2, 3)*. Figure 3 shows the execution strategy tree of parallel sub-query plans constructed by Figure 2.

The implementation of parallel query plans

In this section, the parallel sub-query plans and the

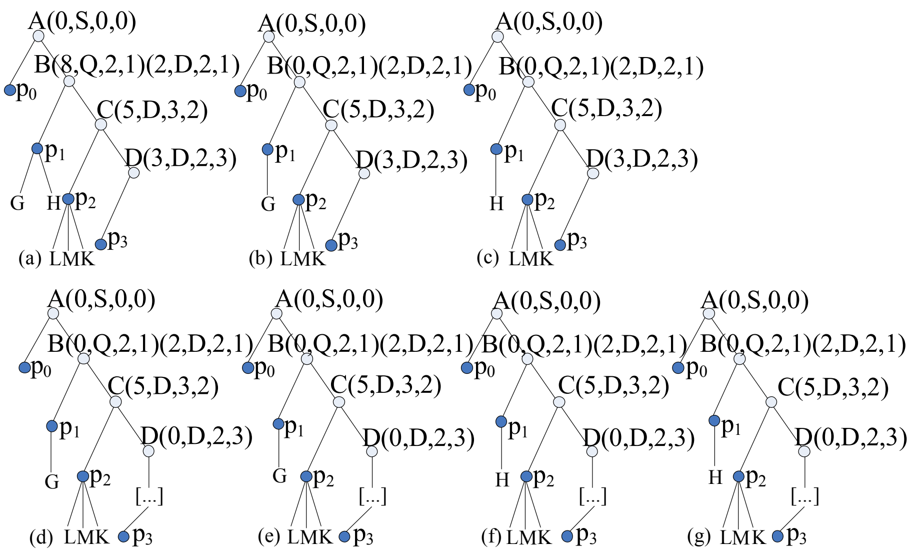


Figure 2. The process of constructing the parallel query plan with six cores.

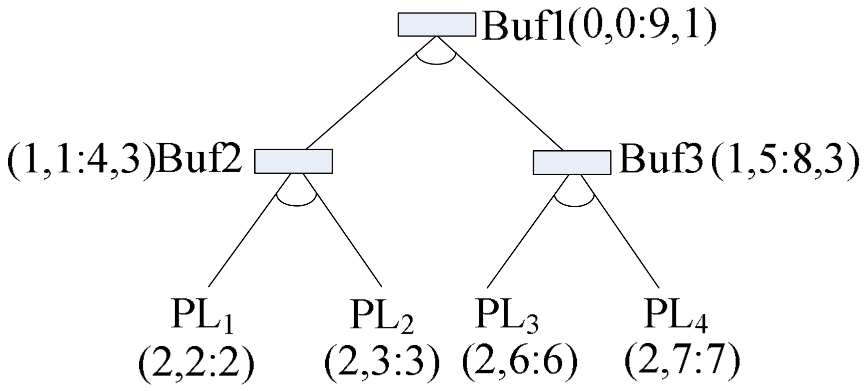


Figure 3. The execution strategy tree of sub-query plans in Figure 2.

partitioning nodes will be employed to split the XML document according to the capacity of L2_cache.

The construction of executive groups

The XML document is encoded by employing region encoding. We assume there is a data stream associated with each XML node. Every element in the data stream is already encoded by region encoding in the following format (level, start: end).

XML document partitioning

Every sub query plan is allocated an executive thread and buffer used to store the partitioned XML data. Every executive thread according the partitioning node in the sub query plan gets a cluster of sub-streams related with this sub query plan. XML data is initially stored to the buffer of every sub query plan.

All executive threads of sub-query plans achieve the following processing parallel. For data partitioning node, the executive thread initially subdivides the stream of this data partitioning node and gets the sub-stream according to a specified range. Furthermore, the executive thread gets other sub-streams of this sub-query plan accordingly through transmission to satisfy the range containment feature between two sub-streams. The allocated buffer stores these sub-streams. For query partitioning node, the executive thread will make the buffer of sub query plan only retain the partitioned predicate node. By this

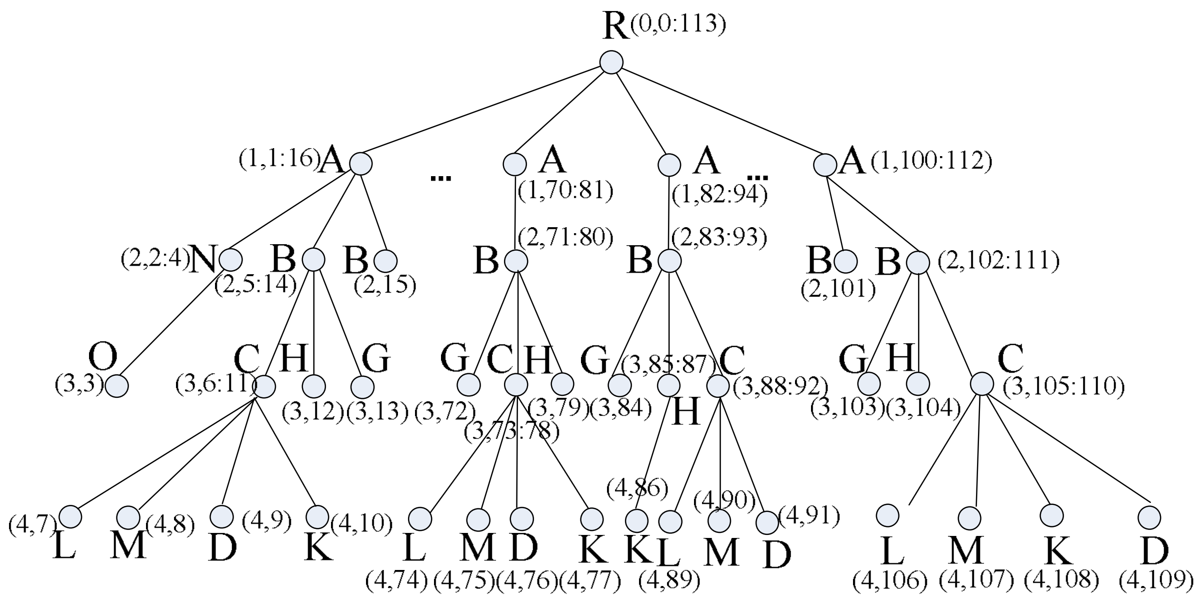


Figure 4. The XML document.

processing the executive groups are constructed in following format.

```
Group1 (p1 , B1)
Group2 (p2 , B2)
......
Groupi (pi , Bi)
```

```
        ExecutionGroup_Constructing
  Input: The selected partitioning nodes, constructed sub-query plans, XML document and Xpath.
  Output: Executive groups
  /* m threads parallel implement sub-query plans, m is the number of sub-query plans.*/
  For i=1 to m
    If not existing data partition node in NoteList
     Buffer[i] = XML data;
    Else
     dn=Get_num(NoteList,D);
     For every data partition node j=0 to dn
       Range=Get_range(partition node, sub-query[i]);
     If j=0 then
        Buffer[i] = Split (XML document, range);
       Else
        Buffer[i] = Split (Buffer[i], range);
    If existing query partition node in NoteList
     qn=Get_num(NoteList,Q);
      For every query partition node j=0 to qn
      Rnodes = Get_node(xpath, sub-query[i]);
      Buffer[i] =Rmove_Nodes(Buffer[i], Rnodes);
    Group_i=Input (Buffer[i], sub-query[i]);
End.
```

ExecutionGroup_Constructing algorithm is introduced to achieve this processing. In this algorithm, *Get_num* function obtains the number of partition node of the specified partitioning type in *NoteList*. *Get_range* gets the specified range of data partition node in sub-query plan. *Get_node* gets the nodes related sub-query$_i$ in xpath query predicate$_i$ apart from the partitioned nodes by query partitioning. *Rmove_Nodes* removes data streams of *Rnodes* in buffer.

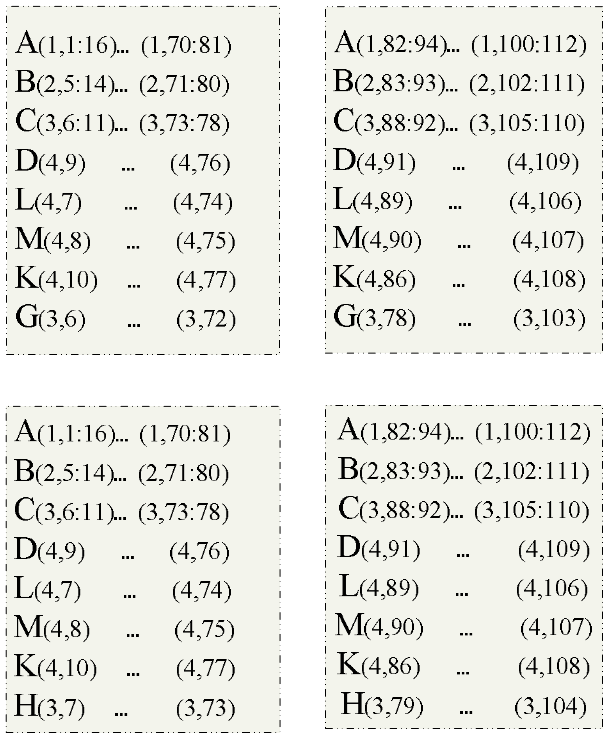
Figure 4 is an XML document. Figure 5 is the partitioned XML document applying the constructed sub-query plans in Figure 2 on this document. The two graphs above Figure 5 are the buffers of sub-query plan (d) and (e) in Figure 2, and the two graphs under Figure 5 are (f) and (g) correspondingly.

The executive thread of the sub-query plan (d) firstly checks if or not exiting data partitioning node by the function. After it found the data partitioning node *D*, this thread by function *Get_range* determines the specified range; that is *D.start* is more than 9 and less than 76. Then this thread gets other sub-streams of this sub-query plan according through transmission to satisfy the range containment feature between two sub-streams. Secondly, this thread finds the query partitioning node *B*. The predicate of *B* in Xpath is *[G operator H]*; the related nodes with this predicate are *G* and *H*. And the partitioned node by query partition *B* in sub query (d) is *G*. So this thread can remove the data stream of *H* by *Rmove_Nodes* in buffer.

Load balancing optimization

Granularity optimization of executive group

The capacity of the L2_cache is *C*. The number of executive groups is *Group.length*. Then average idea size of every partitioned XML document is *C/Group.length* for shared L2_cache. When the total size of the partitioned XML documents is more than *C*, the



```
A(1,1:16)... (1,70:81)      A(1,82:94)... (1,100:112)
B(2,5:14)... (2,71:80)      B(2,83:93)... (2,102:111)
C(3,6:11)... (3,73:78)      C(3,88:92)... (3,105:110)
D(4,9)   ...   (4,76)       D(4,91)   ...   (4,109)
L(4,7)   ...   (4,74)       L(4,89)   ...   (4,106)
M(4,8)   ...   (4,75)       M(4,90)   ...   (4,107)
K(4,10)  ...   (4,77)       K(4,86)   ...   (4,108)
G(3,6)   ...   (3,72)       G(3,78)   ...   (3,103)

A(1,1:16)... (1,70:81)      A(1,82:94)... (1,100:112)
B(2,5:14)... (2,71:80)      B(2,83:93)... (2,102:111)
C(3,6:11)... (3,73:78)      C(3,88:92)... (3,105:110)
D(4,9)   ...   (4,76)       D(4,91)   ...   (4,109)
L(4,7)   ...   (4,74)       L(4,89)   ...   (4,106)
M(4,8)   ...   (4,75)       M(4,90)   ...   (4,107)
K(4,10)  ...   (4,77)       K(4,86)   ...   (4,108)
H(3,7)   ...   (3,73)       H(3,79)   ...   (3,104)
```

Figure 5. The partitioned XML document according to the sub-query plans constructed in Figure 2.

partitioned XML document need to be further refined to fit in the L2_cache according to idea size in order to ensure minimal number of shared L2_cache misses and efficient overall turn-around time. *Granularity_Optimizing* algorithm is introduced to achieve this processing.

```
        Granularity _Optimizing
 Input: the capacity of the L2_cache C, the size of an XML node Ns, executive groups Group
 Ouput: the refined executive groups
 Tsize=Tsize_calculating(C, Group);
 If Tsize > C
  For i=0 to Group.lentgh GL
   If the size of Group[i].Buffer>C/ GL
            Put Group[i] in Queue;
  For every dequeue(Queue) Group in parallel
   Select the largest stream in Group.Buffer to subdivide this Buffer;
   If existing Group.Buffer> C/ GL
        End
```

After the refined granularity optimization of executive group is completed by this algorithm, each executive group may conclude multiple sub-buffers. So every executive group is reconstructed as the following format.

$\text{Group}_i\ ((p_i,\ B_{i1}), (p_i,\ B_{i2})\ldots (p_i,\ B_{ij})$

Workload optimization of executive groups

When data are seriously uneven among the partitioned XML document according to the sub-query plan, the workload of parallel executive threads would be skewed. Parallel performance will be reduced, because serious discrepancy of the workload allocated to the executive groups would lead to load imbalance among cores in CMP.

In this subsection, the allocation scheduler module is optimized by the adjustment of executive pairs among the constructed executive groups. We achieve this by exploiting an accommodative load balancing algorithm. The merit of accommodative load balancing algorithm is that all executive pairs must not be adjusted regarding the degree of data skewed. When the degree of data skewed among executive groups is not high, the necessary boil of the communication and computation will be avoided.

Each step is defined as follows:

Determining laden executive pairs: Each constructed executive group retains only the executive pairs meeting the following condition:

$$\sum_{j=1}^{n} |Executive\ Pair_j| \le \left\lceil \sum_{i=1}^{m} |Bi| / Num \right\rceil$$

$$\sum_{j=1}^{n+1} |Executive\ Pair_j| \ge \left\lceil \sum_{i=1}^{m} |Bi| / Num \right\rceil$$

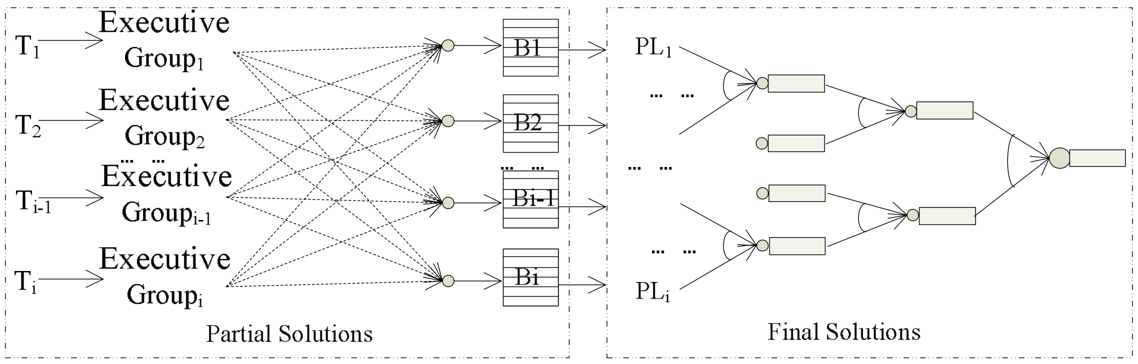


Figure 6. The implementation of executive groups.

$\left\lceil \sum_{i=1}^{m} |Bi| / Num \right\rceil$ is the ideal value of each executive group. Other executive pairs of the group are referred to as laden executive pairs. In this processing, each group tries to retain the large executive pair.

Distributing laden executive pairs: Executive thread of each group reports the size of executive pairs retained and laden executive pairs to the coordinating thread. In accordance with the information of each executive thread's report, the coordination thread firstly uses the automatic matching of least value decline to determine the distribution strategy of laden executive pairs. Then the coordination thread broadcasts the distribution strategy. All laden executive pairs are distributed among threads.

Determining executive set in each group: The executive thread of each group in parallel incorporates the smaller executive pairs to form the executive set that approximately equals to C/Num (referred as B). Num is the number of executive threads. In this phase, the measure factor of balance is introduced to test the executive set correctness. The measure factor of balance is defined as following:

$$F = B - \sum_{i=1}^{m} | EP_i | / \left\lceil (\sum_{i=1}^{m} | EP_i) / B \right\rceil \tag{5}$$

Where, m is the number of executive pairs in one group. EP_i is executive pair and B is the capacity of basic size *C/Num*. $\sum_{i=1}^{m} | EP_i |$ is the sum of executive pairs in one group. If the difference between the basic size and the size of the executive set is no more than the measure factor of balance, this solution is ideal. Every optimized executive group is constructed by three judgments of the measure factor of balance. Firstly, we determine whether or not the continuous executive pairs existing and the difference between the sum of continuous executive pairs and the basic size B is no more than the measure factor of balance. If it is true, we will construct the optimized executive group including the continuous join pairs. Otherwise, this function will find less set of executive pairs from k + 2 (k is the length of the continuous executive pairs). The sum of the continuous executive pairs and less set of executive pairs is calculated, referred to as S. And the difference between the S and the basic size B is judged whether it is no more than the measure factor of balance. If it is true, we will construct the optimized executive group including the continuous executive pairs and the less set of executive pairs found from k + 2. Otherwise, the function deletes the executive pair k and the less set of executive pairs and repeats the search process.

The implement of executive groups

The implement of executive groups comprises two phases. The first phase is parallel performing the executive groups by executive threads (referred *E_threads*). The intermediate result will be classified according to the sub-query plan and stored into different buffers. Meanwhile, the other threads, referred to as *M_threads*, are to merge the partial solutions produced from the first phase for final solutions parallel by the execution strategy tree of sub-query plan. The number of *M_threads* is equal to the number of cores minus the number of executive thread *E_threads* assigned to executive group. This processing is showed in Figure 6.

The different phase is absolute parallel executed. However the phases are parallelized by adopting the incomplete pipeline parallelism technique. If or not the pipeline parallelism technique is adopted according to the partitioning type of leaf nodes' parent node in the execution strategy tree.

When *LNode* is Data Partitioning or Query Partitioning with the decomposed operator is 'or', the *M_threads* of

Table 1. XPath queries tested on dataset.

Dataset	XPath NO	XPath
XMark	XM_1	/site/people/person/name/profile
	XM_2	/site/people/person[address and age]/name/ profile
	XM_3	/site/open_auctions/open_auction[annotation/author and annotation/description and bidder/date]/privacy

the sibling leaf nodes in the execution strategy tree merge buffers classified according to the names of these sibling leaf nodes (the name of sub-query plan) as long as these buffers is not empty. In these situations between phases is pipeline parallelism. Notice that the second situation may transfer the same partial solution to parent node. So the parent nodes need 'distinct' operate.

However, when *LNode* is Query Partitioning and the decomposed operator is 'and', the *M_threads* of the sibling leaf nodes merge buffers classified according to the names of these sibling leaf nodes until all partial solutions have been constructed. So in this situation, the operation between two phases is not pipeline parallelism.

Performance analyses

The goals of out experiments are to reveal that out algorithms significantly improve the performance of XML query processing. All the experiments were performed on a Windows Vista PC with two Intel Xeon Quad Core E540 1.6 GHz CPUs (= 8 cores) and 8 GB of physical memory. Each CPU has two 4 Mbyte L2 caches, each of which is shared by two cores.

Table 1 gives three XPaths queries tested on XMark dataset. The XPath XM1 consists two partitioning nodes/person and/name (referred as dp1 and dp2) without any predicates. The increment cost of them are (12, D, 5, person) and (7, D, 5, name), respectively based on the *PEP_Constructing* algorithms. (12, D, 5, person) is the ideal plan for executing the query XM_1. XM_2 is a predicated query with a conjunction of two path predicates. XM_2 consists of two partitioning nodes/person and its predicate [address and age] (referred as dp1 and dp2). (9, D, 2, person) is firstly selected as data partitioning node, and then (7, Q, 2, person) is selected as query partitioning node. So the combined (9, D, 2, person) and (7, Q, 2, person) are selected as the ideal plan for executing the query XM_2. XM_3 consists of two partitioning node/open_auction and its predicate [annotation/author and annotation/description and bidder/date] (referred as dp1 and dp2). The combined (14, D, 2, open_auction) and (13, Q, 3, open_auction) are selected as the ideal plan for executing the query XM2.

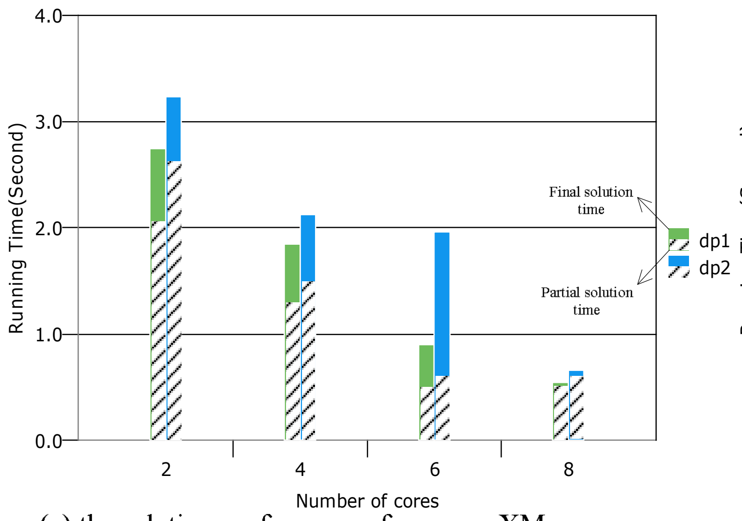
Figure 7 compares the running time of different parallel sub-queries plan according to the selected partitioning node over three XPath queries without considering the optimization of partitioned XML document. The running time consists two parts, partial and final solution times, according to two phases of the implement of executive groups. Figure 7(a) presents relative performance of two data partitioning plans over XM_1. As illustrated in Figure 7(a), the total running time decreases as the number of cores is increased. Because the increment cost reach the max value when the number of cores is more than 5. So after the number of cores is more than 5, the partial solution time cannot be optimized. However, the time of constructing final solution can be reduced with the increasing of the number of cores.

Figures 7(b) and (c) present the performance of different query execution plans for the queries XM2 and XM3. For XM2, *InCost_calculating* algorithm firstly calculates the increment cost of all linear nodes. *PEP_Constructing* algorithm further selected the partitioning node to construct sub-queries plans. The partition node with max increment cost, (9, D, 2, person), is firstly used to sub-queries plans. Then the (7, Q, 2, person) is further selected to construct sub-quries plans. So the original query plan is rewritten into four sub-queries plan. The four sub-queries are then parallel executed to generate partial solutions. Finally, these partial solutions are merged under the guidance of the execution strategy tree of parallel sub-query plans. As illustrated in Figure 7(b), the parallel sub-queries plan constructed by combining (9, D, 2, person) and (7, Q, 2, person) consistently achieves the best performance. Similar to the query XM_1, after the number of cores is more than 4, the partial solution time cannot be optimized. Figure 7(c) shows similar experiment over XM_3 with Figure 7(b).

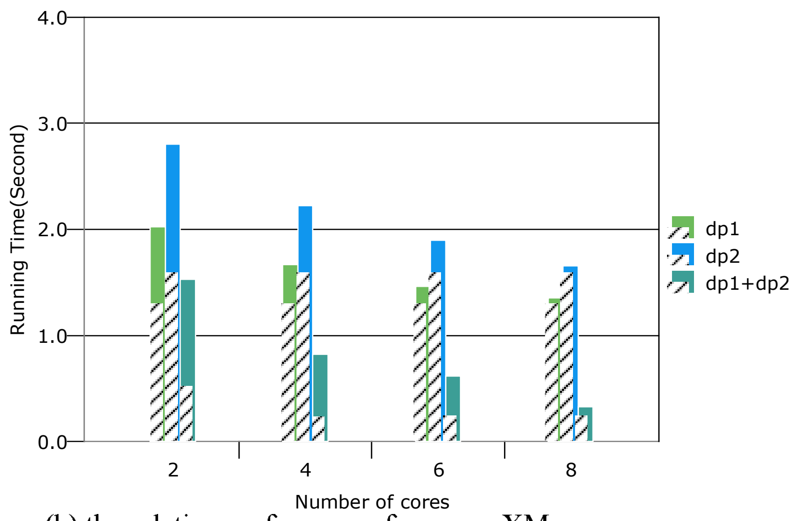
The optimized executive pairs and groups can reduce the time producing the partial solution. Results of the speedup time are illustrated in Figures 8(a), (b) and (c). It is important to note that speed up time decreases with the number of selected partitioning node increasing. This is because the optimization performance of executive pairs and group degrades with the size of partitioned XML data increasing. However, the granularity of partitioned XML data with more partitioning node is less than granularity with less partitioning node. So the speed up time decreases with more partitioning node.

CONCLUSION

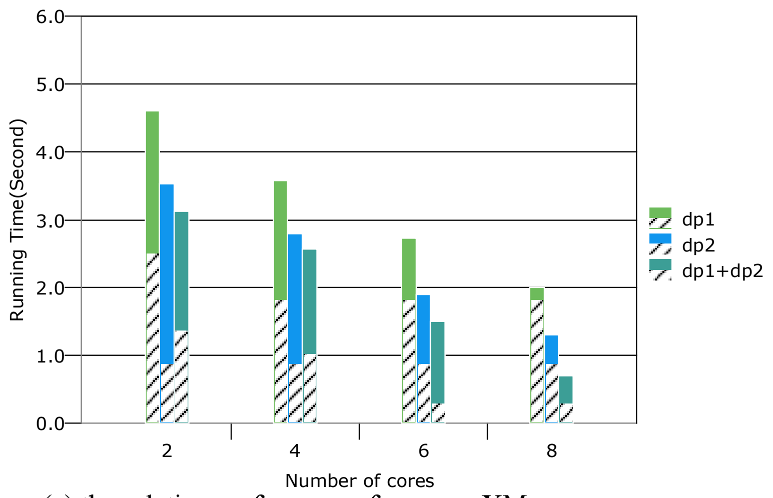
In review, XML query processing is divided into three



(a) the relative performance for query XM_1



(b) the relative performance for query XM_2



(c) the relative performance for query XM_3

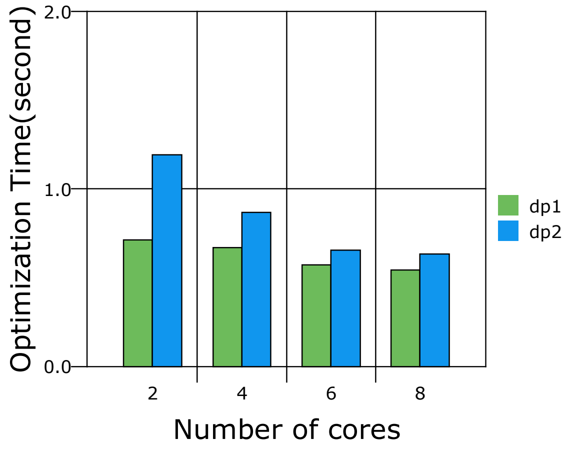
Figure 7. The executive time using different parallel execution plans.

phases in general, building sub-query plans, partitioning XML document and executing in parallel. In the first phase, based on the defined incremental cost model, the sub-query plans and the execution strategy tree are constructed according to the number of thread. In the second phase, these sub-query plans are used to guide the XML document decomposing in parallel. In order to reduce the shared L2_cache access conflict and achieve the workload-balancing assignment among threads, the granularity of executive pair and the workload for all executive groups are further optimized. Finally, the incorporate threads are to merge the partial solutions produced in parallel from the parallel sub-query plans for final solutions by the guidance of the generated execution strategy tree. By implementing our framework and analyzing the experimental results, we have revealed that out algorithms significantly improve the performance of XML query processing.

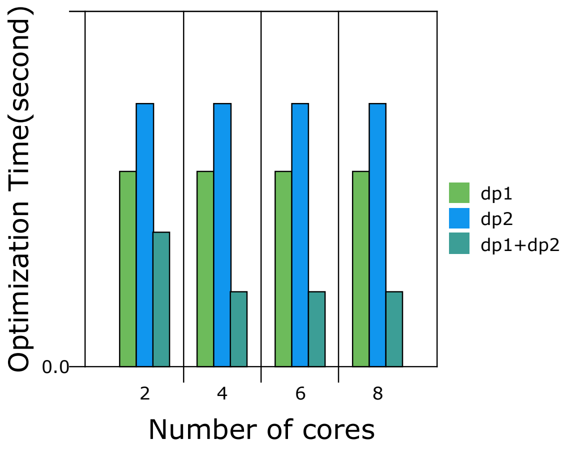
The scale of the experiments performed is far from complete. Future work is still needed in expanding our multithreaded XML query processing to examine performance on other XML dataset.

ACKNOWLEDGEMENT

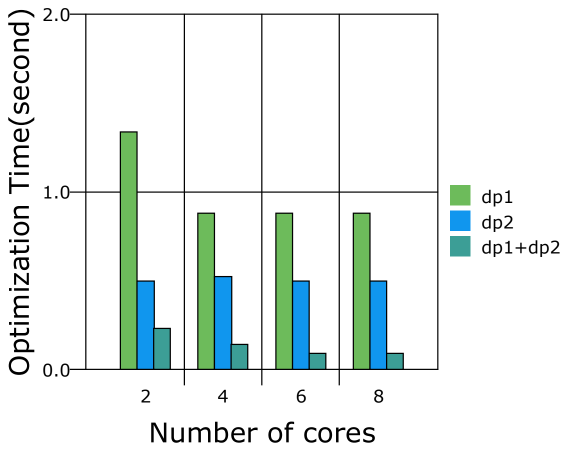
This work is supported by the National Natural Science Foundation of China under grant no. 60973040, no.60903098; the science and technology development program of Jilin Province of China under grant no. 20070533; the specialized research foundation for the doctoral program of higher education of China under



(a) the speed up time for XM_1



(b) the speed up time for XM_2



(c) the speed up time for XM_3

Figure 8. Speed up time with optimized executive pairs and groups.

grant no.200801830021 .

REFERENCES

Aboulnaga A, Alameldeen AR, Naughton JF (2001).Estimating the selectivity of XML path expressions for internet scale applications. In Proceedings of the 27th VLDB Conference, Roma, Italy, pp. 591–600.

Al-Khalifa S, Jagadish HV, Koudas N, Patel JM, Srivastava D, Wu Y (2002). Structural joins: A primitive for efficient XML query pattern matching. In Proceedings of the IEEE International Conference on Data Engineering, California, USA, 19(10): 141-152.

Bordawekar R, Lim L, Kementsietsidis A, Kok BWL (2010).Statistics-based parallelization of XPath queries in shared memory systems. In EDBT, Lausanne, Switzerland, pp.159-170.

Bordawekar R, Lim L, Shmueli O (2009). Parallelization of XPath Queries Using Multi-core Processors: Challenges and Experiences. In EDBT, Saint Petersburg, Russia, pp. 180–191.

Bruno N, Koudas N, Srivastava D (2002). Holistic Twig Joins: Optimal XML Pattern Matching. In Proceedings of the ACM SIGMOD international conference on Management of data, Wisconsin, USA, pp. 310-320.

Chen T, Lu J, Ling TW (2005). On Boosting Holism In XML Twig Pattern Matching Using Structural Indexing Techniques. In SIGMOD, Baltimore, Maryland, USA.

Feng JH, Liu L, Li GL, Li JH, Sun YH (2010). An Efficient Parallel PathStack Algorithm for Processing XML Twig Queries on Multi-core Systems. In DASFAA proceeding, Tsukuba, Japan, 277-291.

Gou G, Chirkova R (2007). Efficiently Querying Large XML Data Repositories: A Survey. IEEE Transactions on Knowledge and Data Engineering, 19(10): 1381-1403.

Hennessy JL, Patterson DA (2007). Computer Architecture, 4th ed.

Jiang HF, Wang W, Lu HJ, Jeffrey XY (2003). Holistic Twig Joins on Indexed XML Documents. In Proceedings of the international conference on Very large data bases, Berlin, Germany, p. 29.

Kurita H, Hatano K, Miyazaki J, Uemura S (2007). Efficient Query Processing for Large XML Data in Distributed Environments. In 21st International Conference on Advanced Networking and Applications, Ontario, Canada, pp. 317–322.

Li GL, Feng JH, Zhang Y, Ta N, Zhou LZ (2006). Exploiting Even Partition to Accelerate Structure Join. In Proceedings of the Seventh International Conference on Web-Age Information Management Workshops, Hong Kong, China.

Lu J, Ling TW, Chan CY Chen T (2005). From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. In VLDB, China, pp. 193-204.

Lu W, Gannon D (2007). Parallel XML Processing by Work Stealing. In SOCP, Monterey, California, USA.

Machdi I, Amagasa T, Kitagawa H (2010).Task Parallelism for Twig Stack Algorithm on a Multi-Core System. Data Engineering and Information Management Forum, Awaji,

McHugh J, Widom J (1999).Query optimization for XML. In Proceedings of VLDB, pp. 315-162.

Zhang C, Naughton J, Dewitt D, Luo Q, Lohman G (2001). On supporting containment queries in relational databse management systems. In Proceedings of ACM SIGMOD, California, USA, 30: 2.