Parallel Structural Join Algorithm on Shared-memory Multi-core Systems

Le Liu, Jianhua Feng, Guoliang Li
Department of Computer Science and Technology
Tsinghua University
Beijing, China
{le-liu02@mails., fengjh@, liguoliang@}tsinghua.edu.cn

Qian Qian, Jianhui Li
Intel Corporation
Shanghai, China
{qian.qian, jian.hui.li}@intel.com

***Abstract*—The leap from single-core to multi-core has permanently altered the course of computing, enabling increased productivity, powerful energy-efficient performance, and leading-edge advanced computing experiences. Although traditional single-thread XPath query evaluation algorithms can run properly on multi-core CPUs, they cannot take full use of the computing resources of multi-core CPUs. To take advantage of multi-core, efficient parallel algorithms are fairly desirable to evaluate XPath in parallel. In this paper, we present, PSJ, an efficient Parallel Structural Join algorithm for evaluating XPath. PSJ can skip many ancestor or descendant elements by evenly and efficiently partitioning the input element lists into some buckets. PSJ obtains high performance by evaluating XPath step in each bucket in parallel. It is very efficient to partition the input lists and is effective to evaluate XPath step in buckets, and therefore PSJ achieves a high speedup ratio. We have implemented our proposed algorithm and the experimental results show that PSJ algorithm achieves high performance and outperforms the existing state-of-the-art methods significantly.**

***Index Terms*—even partition, parallel, speedup ratio, structural join**

I. INTRODUCTION

As XML becoming *de facto* standard of data presentation and exchange over the Internet, masses of data pour in the form of XML document. How to store and query XML documents is a hot topic for database researchers. Many XML query languages, such as XPath[1], XQuery[2], XML-QL[3], have been studied. One of the key techniques is to use a path expression to express and search particular structure patterns. An XML document can be labeled with some numbering schemes [4]. By incorporating labels numbering, we can quickly determine the parent-child or ancestor-descendant relationship of element nodes and attribute nodes without traversing the original XML document.

Many algorithms have been studied for XPath query processing. S. AI-Khalifa et al. [5] proposed a structural join algorithm, which takes two ordered lists as input, one for ancestors and the other for descendants. To process the twigs in XPath and avoid large intermediate results, many holistic twig join algorithms are proposed, such as TwigStack [6] , TSGeneric [7], TJFast [8], iTwigJoin [9] and so on.

In addition, the leap from single-core to multi-core has permanently altered the course of computing; enabling increased productivity, powerful energy-efficient performance, and leading-edge advanced computing experiences. All the above algorithms have a common characteristic: they are proposed for single-core CPU. Although they can run properly on multi-core CPUs, they can't take fully use of the computing resources of multi-core CPUs. To take advantage of multi-core, efficient parallel algorithms are very desirable to evaluate XPath. In this paper, we present, PSJ, an efficient parallel algorithm for structural join on shared-memory multi-core systems.

We consider XPath query processing from two aspects: one is data partition, and the other is task partition. We only consider data partition in this paper. So when we consider parallel algorithm, we first need a method for partitioning XML elements into even parts. Guoliang Li et al. [10] proposed an even partition based method, which partitions the input XML element lists into buckets evenly and may skip many ancestor or descendant elements. Here we will borrow the idea of even partition from [10], and adapt it to be fit for our need. We use region based numbering scheme instead of BBTC [11], as region encoding is simple and useful, while BBTC is strong but complicated. Accordingly, we can adopt the rules of partition for region encoding. When we evenly partition XML data into some buckets, we can evaluate XPath step (Parent-Child or Ancestor-Descendant relationship) in each bucket in parallel. This is the idea of our PSJ algorithm proposed in this paper. And the experimental results prove that PSJ has a good speedup ratio, and even when in single-thread running state, it outperforms the standard structural join algorithm significantly.

Our main contributions are summarized as follows:

1) We adapt even partition approach from [10] on our problem and make it keeps the original excellence of skipping ancestor or descendant nodes and partition elements into buckets faster than the original.

2) We propose the algorithm PSJ, which evaluates XPath in each bucket in parallel and gets a very good speedup ratio. Optimize the algorithm PSJ. In the result, even when in single-thread running state, PSJ still outperforms the standard structural join algorithm.

The rest of the paper is organized as follows. Section II gives some previous work on XML query processing in parallel. We give the preliminary of PSJ algorithm in Section III. Section IV presents the parallel structural join algorithm PSJ and analyzes the complexity of PSJ. In section V, we give experimental results of PSJ. And we conclude in section VI and acknowledge in section VII.

II. Related work

Many studies have been proposed for XML processing in parallel. In the context of semi-structured and XML databases, structural join was essential to XML query processing as XML queries usually imposed certain structural relationships.

For binary structural join, Zhang et al. [19] proposed a multi-predicate merge join (MPMGJN) algorithm based on <start,end,level> labeling of XML elements. Li et al. [20] proposed EE/EA Join, which decomposed the structure join into element-element join and element-attribute join. Stack-tree-Desc/Anc was proposed in [5], which was the first stack-based algorithm. [21], [22], [23] are index-based approaches. They examined the indices of B^+-tree, R-tree and XR-tree to improve the efficiency of XML queries processing. The later work by Wu et al. [24] studied the problem of binary join order selection for complex queries on a cost model, which took into consideration factors such as selectivity and intermediate results size. Although structure join is more efficient than the navigation based methods, it will involve huge intermediate results.

To address this problem, holistic twig join is proposed. Bruno et al. [6] proposed a holistic twig join algorithm, namely TwigStack, to avoid producing a large intermediate results. With a chain of linked stacks to compactly represent partial results of individual query root-to-leaf paths, TwigStack merged the sorted lists of participating element sets altogether, without creating large intermediate results. TwigStack had been proved to be optimal in terms of input and output sizes for twigs with only A-D (Ancestor-Descendant) edges.

Jiang et al. [7] studied the problem of holistic twig joins on all/partly indexed XML documents. Their proposed algorithms used indices to efficiently skip the elements that do not contribute to final answers, but their method cannot reduce the size of intermediate results. Choi et al. [25] proved that the optimality evaluation of twig patterns with arbitrarily mixed A-D and P-C (Parent-Child) edges was not feasible. Lu et al. [26] proposed the algorithm TwigStackList, which was better than any of previous work in term of the size of intermediate results for matching XML twig patterns with both P-C and A-D edges. Chen et al. [27] proposed an algorithm iTwigJoin, which was still based on region encoding, but worked with different data partition strategies (e.g. Tag+Level and Prefix Path Streaming). Tag+Level streaming can be optimal for both A-D and P-C only twig patterns whereas PPS streaming could be optimal for A-D only, P-C only and one branch node only twig patterns assuming there was no repetitive tag in the twig patterns. Lu et al. [8] proposed a novel algorithm, TJFast, on extended Dewey that only used leaf nodes' streams and saved I/O consumption.

More recently, Mathis et al. [28] proposed a set of new locking-aware operators for twig pattern query evaluation to ensure data consistency. Chen et al. [29] presented Twig2Stack algorithm to avoid huge intermediate results. However, Twig^{2}Stack reduced the intermediate results at the expense of a huge memory requirement and it was restricted by the fan-out of the XML documents. Our prior work TJEssential [30] proposed a root-to-leaf combining with leaf-to-root way to improve the performance of XML query processing.

In addition, Wei Lu et al. [12] proposed a parallel approach for XML parsing, which is the first to use an initial pass to determine the logical tree structure of an XML document and then divide the document between the chunks occur at well-defined points in the XML grammar. Wei Lu et al. [13] proposed the concept of work stealing. If a thread is idle, it will choose a busy thread, and steal a half of work from the busy thread. Xiaogang Li [14] distributed XML data into several different machines according to common path prefix of XQuery queries, evaluated on each machine, and finally combined the distributed results. Distributed evaluation is not what we need, but we focus on parallel evaluation on memory-shared and multi-core systems.

III. Preliminaries

Even partition approach [10] divides AList (the input list of ancestor elements) and DList (the input list of descendant list) into different buckets, $AList_i$ (the *i-th* bucket of AList) and $DList_i$ (the *i-th* bucket of DList) respectively, and only structure joins of suited buckets are useful to the join results. It makes sure $AList \bullet DList = \bigcup_{i=1}^{n_b} (AList_i \bullet DList_i)$, where $AList_i$ and $DList_i$ denote the element sets of AList and DList respectively in i-th bucket after partition, and n_b denotes the number of buckets. In other words, only $AList_i \cdot DList_i$ is helpful to the final result and $AList_i \cdot DList_j$ $(i \neq j)$ is not useful. In the algorithm PRIAM proposed in [10], first partition DList into different buckets $DList_i$, and the size of each bucket (except the last one) is constant, denoted with b_s. And then partition AList into the buckets $AList_i$ accordingly to DList. $\forall n_a \in AList$, $n_a \in AList_i$ as long as n_a maybe has one or more descendants in $DList_i$. In most cases, this partition approach can assure that all elements, except the root, in different buckets don't have the ancestor-descendant or parent-child relationship, that is, $AList_i \cdot DList_j = \emptyset$ $(i \neq j)$. Even if $AList_i \cdot DList_j \neq \emptyset$, the partitioning conditions assures $AList_i \cdot DList_j \subseteq AList_j \cdot DList_j$.

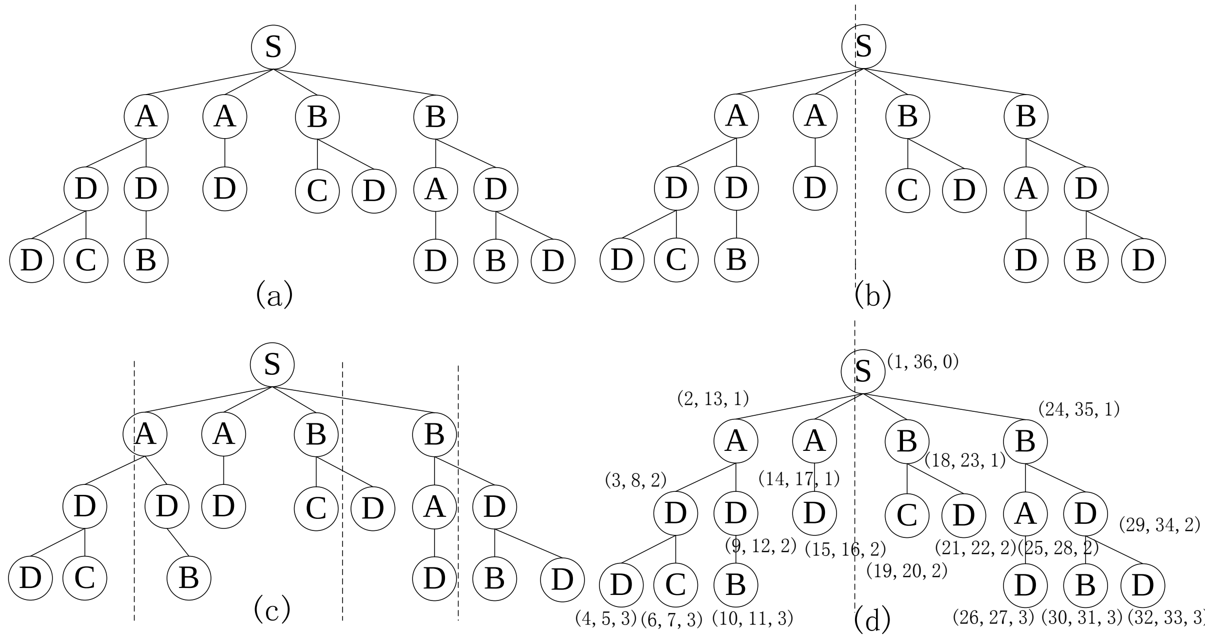


Figure 1 An XML tree and corresponding partition results

The partitioned buckets satisfy the following conditions:

1) $DList = \bigcup_{i=1}^{n_b} DList_i$ and $DList_i \cap DList_j = \varnothing (i \neq j)$

2) $\forall i, 1 \le i < n_b = \lceil |DList| / b_s \rceil, |DList_i| = b_s$ and $|DList_{n_b}| = |DList| - b_s * (n_b - 1)$

3) $\bigcup_{n=1}^{n_b} AList_i \subseteq AList$

4) $AList \bullet DList = \bigcup_{i=1}^{n_b} AList_i \bullet DList_i$

For example, suppose AList={all the elements whose local name is *A* in Fig. 1(a)} and DList={all the elements whose local name is *D* in Fig. 1(a)}. Then AList and DList can be partitioned into different buckets as shown in Fig. 1(b), 1(c). In Fig. 1(b), each bucket contains four *D* elements, and in Fig. 1(c) each bucket contains two *D* elements. In Fig. 1(c) the first *A* element will be put into both $AList_1$ and $AList_2$. Although $AList_1 \cdot DList_2 = \{D_3\}$, $AList_2 \cdot DList_2 = \{D_3, D_4\}$, that is, $AList_1 \cdot DList_2 \subseteq AList_2 \cdot DList_2$, so we can safely ignore the operation $AList_1 \cdot DList_2$. The same occurs with $AList_2$ and $DList_1$.

PRIAM uses the BBTC encoding [11]. The approach of coding an XML document using BBTC is: the code of root node is 1, and code of the leftmost child is its parent's code multiplied by 2; orders of other children except the leftmost child are their preceding siblings' code multiplied by 2 plus 1. From the approach we can know if an XML document is of large size, its BBTC code will impropriate much storage. So in this paper we will use region encoding, as Fig. 1(d). Correspondingly, we will rework the partition rules, which also make the partition much faster. And in section IV we will give our rules to determine size of each bucket particularly for parallel structural join.

IV. PSJ: A PARALLEL STRUCTURAL JOIN ALGORITHM

In this section we will first give our even partition approach, and then describe algorithm PSJ in detail. We will also analyze the algorithm complexity of PSJ.

A. Even Partition

We employ region encoding <*start, end, level*> to encode XML documents. With region encoding, we can determine quickly the relationship between two nodes, such as parent-child or ancestor-descendant relationship. And as the size of XML document increases, the space cost by region encoding increases linearly. Suppose AList and DList denote the ancestor list and the descendant list respectively, and they are in document order. We will partition AList and DList into different buckets $bucket_i$ and $bucket_i$ which contain both $AList_i$ and $DList_i$.

Now we introduce two Rules to partition DList and AList into different buckets.

Rule 1: Partition DList

for $i = 0 \ldots (n_b-1)$
  $bucket_i.dstartpos = i * b_s$
  if $i < b_s - 1$
    $bucket_i.dendpos = (i+1) * b_s - 1;$
  else
    $bucket_i.dendpos = |DList| - 1$
end for

Rule 2: Partition AList

for $i = 0 \ldots (n_b-1)$
  $bucket_i.astartpos = \min \{p | a_p.end > bucket_i.minstart, 0 \le p < |AList|, a_p \in AList\}$
  $bucket_i.aendpos = \max \{p | a_p.start < bucket_i.maxend, 0 \le p < |AList|, a_p \in AList\}$
end for

$bucket_i.dstartpos$ means the start position of descendant elements in DList contained by *i-th* bucket.

$bucket_i.dendpos$ means the end position of descendant elements in DList contained by *i-th* bucket.

$bucket_i.astartpos$ means the start position of ancestor elements in DList contained by *i-th* bucket.

$bucket_i.aendpos$ means the start position of ancestor elements in DList contained by *i-th* bucket.

$bucket_i.minstart = d_k.start,\ d_k.start \le d_j.start,\ \forall j \in [bucket_i.dstartpos, bucket_i.dendpos]$. In fact, as elements in DList are in document order, $bucket_i.minstart$ equals the $bucket_i.dstartpos$-*th* element's *start* value in DList.

$bucket_i.maxend = d_k.start,\ d_k.end \ge d_j.end,\ \forall j \in [bucket_i.dstartpos, bucket_i.dendpos]$. In fact,we can use DList's $bucket_i.dendpos$-*th* element's *end* value as the value of $bucket_i.maxend$.

b_s denotes the number of descendant elements each bucket contains.

n_b denotes the number of buckets.

Rule 1 means that DList is partitioned into n_b buckets, each one (except the last one) contains b_s descendant elements and the last one contains $(|DList| - (n_b-1)*b_s)$ elements. The elements from $bucket_i.dstartpos$-*th* to $bucket_i.dendpos$-*th* of DList belong to $bucket_i$.

Rule 2 means that if more than one ancestor elements have descendants in $bucket_i$, there is a start position and end position in AList. The elements between *astartpos* and *aendpos* are contained by $bucket_i$. If $aendpos < astartpos$, there are not ancestor elements in $bucket_i$, the structural join result in $bucket_i$ will be empty.

For example, the XML document in Fig. 1(d) will be partitioned below:

$bucket_0$: dstartpos=0, dendpos=3, astartpos=0, aendpos=1, minstart=3, maxend=16

$bucket_1$: dstartpos=4, dendpos=7, astartpos=2, aendpos=2, minstart=21, maxend=34

B. Work Balance

The purpose of even partition is to evaluate XPath in parallel. Thus we should consider work balance when partitioning.

We can see that Rule 1 in Section IV (*A*) makes size of each bucket is the same, except the last one. In most cases, Rule 1 will assure the work load between each bucket balanced.

Only balance between buckets is not enough, we should also consider the balance between threads in parallel. In other words, the number of buckets assigned to each thread should be the same. To solve this problem, we determine n_b and b_s as below:

```
n_b=8
b_s=|DList|/ n_b
while b_s >20000
  n_b = n_b *2
  b_s = |DList|/ n_b
end while
while b_s <3500
  if b_s == 1
    break
  else
    n_b = n_b /2
  end if
  b_s = |DList|/ n_b
end while
```

From the pseudo code above, we partition DList into n_b buckets, and n_b is power of 2, such as 2, 4, 8, 16 and so on. As multi-core CPUs inside computers commonly have 2, 4 or 8 cores, and then each thread on a core will get the same number of buckets. And b_s is between 3500 and 20000. If $|DList|<3500$, there is only one bucket. The numbers 3500 and 20000 are experiential values for structural join in this paper. If b_s is too large, it will make against work balance; if b_s is too small, it will make against the exertion of parallel predominance, because parallel scheduling needs extra cost.

C. The Parallel Structural Join Algorithm PSJ

In this section, we describe the parallel structural join algorithm PSJ, which is the key part of this paper.

Fig. 2 describes the PSJ algorithm in detail. The algorithm PSJ first determines the size of each bucket and the number of buckets in line 1, then partitions DList into n_b buckets according to Rule 1 in line 2; in fact it gets the values of $bucket_i.minstart$ and $bucket_i.maxend$, $bucket_i.dstartpos$ and $bucket_i.dendpos$ and doesn't cost any I/O. In line 3, PSJ partitions AList according to Rule 2; in fact, it only gets the values of $bucket_i.astartpos$ and $bucket_i.aendpos$. After partitioning, each bucket holds $AList_i$ and $DList_i$, and then we evaluate ancestor-descendant relationship in each bucket in parallel in line 4. We use the openmp technology [15] to implement parallel execution. Openmp uses thread pool technology, and it can execute "*for* loop" perfectly in parallel.

The function *SJ_Stack_Tree_Desc* is borrowed from the algorithm Stack-Tree-Desc structural join in [5], but we make a little modification. In a word, the algorithm here is more detailed. In line 6 we change the primary conditions *(the input lists are not empty or the stack is not empty)* to *($DList_i$ is not empty and ($AList_i$ is not empty or the stack is not empty))*. And in line 16 we add a condition *(a.end>d.start)* to push the ancestor element *a* into the stack. Because only element whose *end* value is larger than current descendant's *start* value, it may be an ancestor of current descendant. This skips directly ancestor elements which do not have a descendant and avoid these elements entering the stack.

Consider the XML document tree in Fig. 1(d). Suppose we want to evaluate the XPath *//A//D* on a machine with a two-core CPU. And suppose the tree is partitioned as in Section IV(A). Then we get two buckets, and create two threads. Each thread evaluates one bucket. In theory the time it costs is a half of that in serialization execution. In actual environments, it is impossible, because creating and scheduling multi-threads will cost extra time. We will discuss the factors which affect the performance of execution in parallel in next section *D*.

```
PSJ (AList, DList, ResultList)
Input:
    AList: an ancestor list in document order
    DList: a descendant list in document order
    ResultList: a list for holding results
Output:
    ResultList={(a,d)|a∈AList, d∈DList, and a is an ancestor of d}

Begin
1. Determine the value of n_b and b_s according to Section IV(B)
2. Partition DList according to the Rule 1
3. Partition AList according to the Rule 2
4. #pragma omp parallel for
   // the line above is the syntax of openmp [15]
   // execute "for loop" in parallel
   for i=0 to n_b-1
      SJ_Stack_Tree_Desc(bucket_i,AList, DList,
                         tempRListi)
      ResultList=ResultList ∪ tempRListi
   end for
5. return ResultList

SJ_Stack_Tree_Desc (bucket_i, AList, DList, tempRListi) [5]
Input:
    bucket_i: the i-th bucket
    AList: the ancestor list
    DList: the descendant list
    tempRListi: the list for holding results
Output:
    tempRListi: the structural join result in the i-th bucket

Begin
6. a=AList_i.firstNode, d=DList_i.firstNode
7. while(DList_i is not empty and (AList_i is not empty or the
   stack is not empty))
8.   if (the stack is not empty)
9.      tempend = stack.top()
10.  else
11.       tempend.start=INT_MAX, tempend.end=INT_MAX
12.  end if
13.  if(a.start>tempend.end && d.start>tempend.end)
14.       stack.pop()
15.  else if(a.start<d.start)
16.      if(a.end>d.start) stack.push(a)
17.      if(AList_i is not empty)  a=a->nextNode
18.      else a.start=INT_MAX, a.end=INT_MAX
19.  else
20.       for(a1=stack.bottom; a1!=NULL;a1=a1->up)
21.            append (a1, d) to tempRListi
22.       d=d->nextnode
23.  end if
24.  end while
25.  return tempRListi
```

Figure 2 PSJ: A Parallel Structural Join Algorithm

D.Analysis of PSJ

In this section we discuss the factors which affect the performance of execution in parallel.

In Section *IV(B) Work Balance* we have discussed is a factor which affects the performance of execution in parallel. That's the work balance. To achieve a good speedup ratio, we should assure the work load of each thread is approximately the same, and the complete same is best.

Besides work balance, we should also reduce communications among threads and accessing shared memory. In the algorithm PSJ, there is almost none communication between threads. And only AList and DList, which store XML encoding data, are shared data among threads. It looks like nothing can be done to improve the performance. However, But it's not true.

Let's check the function *SJ_Stack_Tree_Desc* in Fig. 2. We find that PSJ will write data into main memory in line 16 and 21. If every time writing data into memory it needs applying a new main memory area for storing data, many main memory accessing conflicts may occur, which will involve much more time than serially accessing. Our solution is applying a large enough memory area in advanced. We can initialize the stack in line 16 with a capacity of 100, as almost all XML documents have a depth less than 100 (The largest depth of Treebank dataset is 36). And we can initialize the *tempRListi* in line 21 with a capacity of size of DList, as the size of results is not large than that of DList. With this solution, although simple, the performance of PSJ improves a lot. Even when with single-thread, PSJ outperforms the standard structural join algorithm. The experimental results in Section V will prove this.

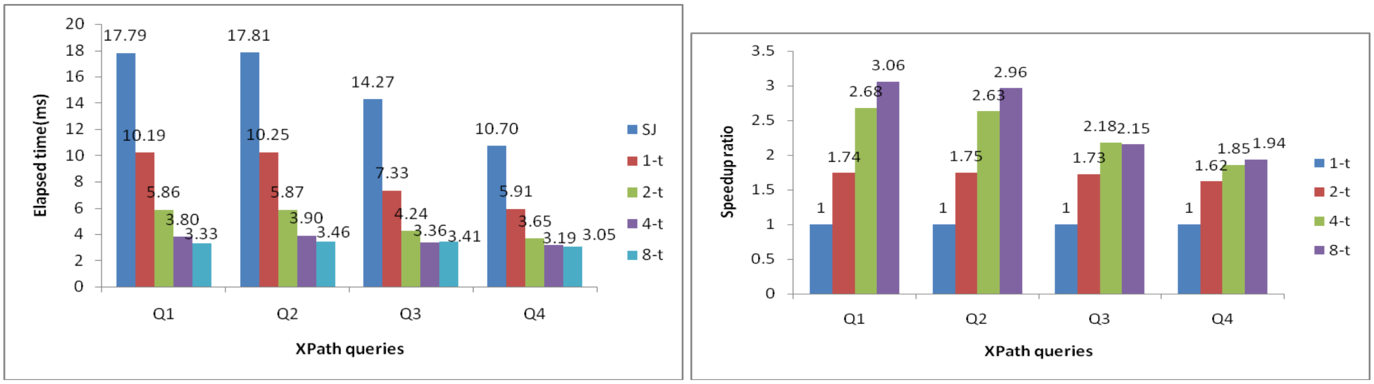
Now we analyze the algorithm complexity of PSJ.

In Fig. 2, it costs the complexity of O(log(|DList|/(8*b_s))) in line 1 to determine the value of n_b and b_s. As b_s is between 3500 and 20000, even size of |DList| equals 1 million (Treebank has 435689 NPs), *log (|DList|/(8*b_s))* is less than 10; it's so small, thus we just ignore it. Also we ignore the complexity of partitioning DList in line 2.

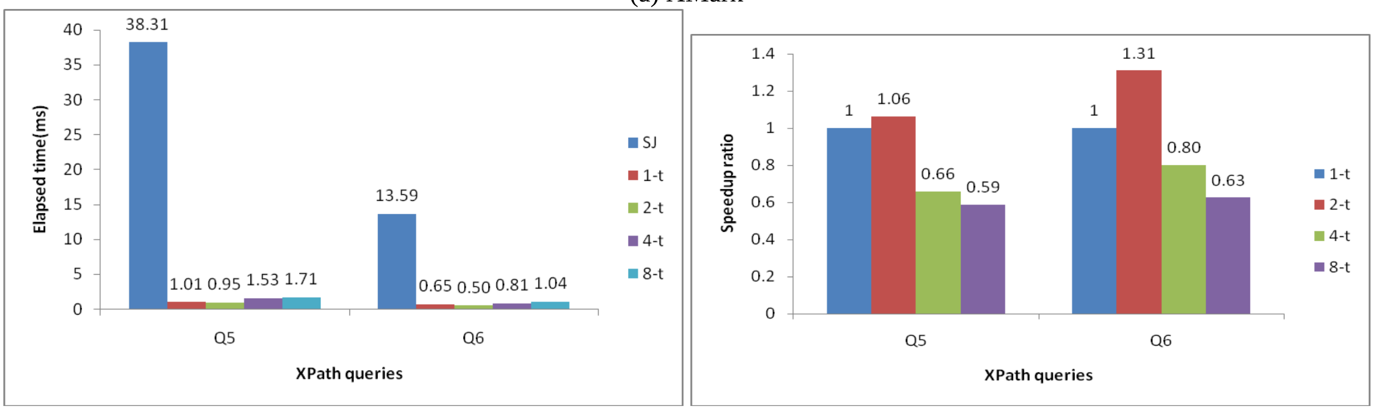
In line 3 when partitioning AList, if the XML document does not contain nesting homonymy elements, we can use binary search to find values of $bucket_i$.aendpos and $bucket_i$.astartpos. When finding the values of $bucket_i$.aendpos, the search end position is the end of AList; while $bucket_i$.astartpos with the end $bucket_i$.aendpos. Both start positions are $bucket_{i-1}$.aendpos. So the average total complexity is

$$\log\frac{|AList|}{n_b}+\log\frac{2*|AList|}{n_b}+\cdots+\log\frac{n_b*|AList|}{n_b}+n_b\log\frac{2*|AList|}{n_b}$$
$$=\log n_b!(\frac{|AList|}{n_b})^{n_b}+n_b\log\frac{2*|AList|}{n_b}$$
$$<2n_b\log|AList|$$

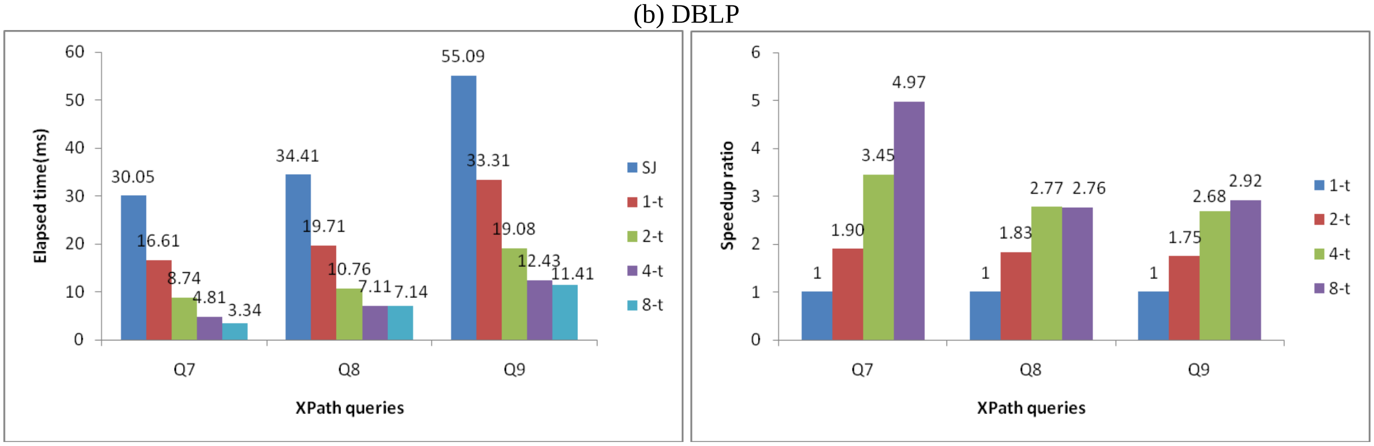
If the XML document contains nesting homonymy elements, we can only use binary search to find value of $bucket_i$.aendpos,



(a) XMark



(b) DBLP



(c) Treebank

Figure 3 Elapsed time and speedup ratio with different datasets

SJ: the standard Stack_Tree_Desc structural join algorithm;
n-t: the PSJ algorithm with n threads, n=1,2,4,8

because the elements' *end* values are not in document order. To find the value of $bucket_i$.astartpos, we can use linear search, but it's not efficient. Instead, we can extend region encoding to the form <*start*, *end*, *level*, *sibpos*>, in which *sibpos* denotes the position of current node's first right sibling's position in AList. For example, the first *A* element in Fig. 1(d) can be encoded with <2, 13, 1, 1>. With this extended region encoding method, we can find value of *$bucket_i$.astartpos* much more efficiently, than linear search. And the existence of *sibpos* can help skip more ancestor element in line 16.

From [5], we know that the function *SJ_Stack_Tree_Desc* in line 4 costs |AList|+|DList|+|ResutList| in serial mode, and ideally (|AList|+|DList|+|ResutList|)/n in parallel, where *n* is the number of threads. As n_b (the number of buckets) is small in general, the cost of partitioning is much less than that of structural join in each bucket. Accordingly our algorithm can get a good speedup ratio. Our experimental results in Section V will prove this.

V. EXPERIMENTAL ANALYSIS

We conducted a set of extensive experiments to study the performance of PSJ in this section. We tested totally 9 XPath queries on three different datasets, which are XMark[16], DBLP[17], and Treebank[18]. All the experiments are carried out on a computer with Intel Xeon (8-core) CPU, 4G main memory and Windows Server 2003 operating system, and we used C++ for programming and Visual Studio 2005 as the compiler. And we used OpenMP [15] for parallel programming, which is contained in VS 2005 and whose version is 2.0.

The dataset DBLP is a set of bibliography files, and the size of the raw text files is around 420M. We generated the XMark data with scale factor = 4 and the raw text file is about 454MB. And the size of Treebank dataset is about 82MB. Table I lists the XPath queries tested on the three datasets. There are four XPath queries for XMark, two for DBLP and three for Treebank. And Table II lists the number of element nodes used in the queries in Table I. As the three dataset are all of large sizes, the numbers of element nodes are large, except the node *regions* for XMark. We choose large datasets because for conveniently analyzing performance of the algorithm PSJ.

Table I XPath queries tested on datasets

Dataset	XPath Number	XPath
XMark	Q1	description//emph
	Q2	description//keyword
	Q3	item//mailbox//from
	Q4	regions//item//mail
DBLP	Q5	book//author
	Q6	incollection//title
Treebank	Q7	NP//NN
	Q8	NP//VP//PP//DT
	Q9	S//NP//NN

Table II Number of nodes in datasets

Tag Name	Number	Tag Name	Number
description	178,000	NN	186,597
emph	280,290	VP	154,298
keyword	281,234	PP	135,771
item	87,000	DT	115,863
mailbox	87,000	S	153,270
from	83,527	book	1,248
regions	1	author	2,410,223
mail	83527	incollection	2,610
NP	435,689	title	987,075

Fig. 3 shows the experimental results on different datasets.

First, the algorithm PSJ on 1-t (PSJ with 1 thread, the same below) is better than SJ on all the three datasets. On XMark, 1-t outperforms SJ by as much as nearly 95% for Q3 in Fig. 3(a). And on DBLP, 1-t amazedly outperforms SJ by 36.93 times for Q5 and 19.91 times for Q6 in Fig. 3(b). And on Treebank, 1-t outperforms SJ by 81% for Q7 in Fig. 3(c). There are two reasons: that outperforms SJ. The first is that the even partitioning as described in Section IV (A) skips many ancestor elements, and the second is applying a large enough memory area in advance when calling the function *SJ_Stack_Tree_Desc* in line 4 in Fig. 2, which will avoid frequently applying memory space and save much time.

Let's analyze Q5 in detail. DBLP has 1,248 elements with tag name *book*, and 2,410,223 elements with tag name *author*. Then *author* list is partitioned into 128 different buckets. The size of last bucket is 18940, and the size of other buckets is 18829. We find that the first bucket contains 1,239 *book* elements; the 74-th bucket contains 1; the 84-th bucket contains 7, and the last bucket contains 1; other buckets contain none *book* elements. So PSJ only needs to evaluate four buckets, i.e. 4/128 of *author* list, and the evaluation time cost in the 74-th, 84-th and last bucket is less than average evaluation time for bucket because there are only few *book* elements in these buckets, but SJ needs to evaluate the entire author list. Thus it's absolutely possible that PSJ with 1 thread outperforms SJ by 36.93 times for Q5, which is larger than 128/4=32 times. Similarly for Q6, DBLP has 2,610 elements with tag name *incollection*, and 987,075 with *title*. The title list is partitioned into 64 buckets. The size of last bucket is 15,426, and the size of others is 15,423. The first bucket contains 2526 *title* elements, the 35-th contains 16, the 40-th bucket contains 52, and the 56-th bucket contains 17. Then PSJ only need to evaluate 4 buckets. And because the 35-th, 40-th and 56-th buckets contain very few *incollection* elements, the evaluation time cost in these buckets is less than average evaluation time for bucket. Thus it's absolutely possible that 1-t outperforms SJ by 19.91 times for Q6, which is larger than 64/4=16 times.

Second, let's consider the speedup ratio of PSJ. The definition of speedup ration of n-t is below:

speedup_ratio = time_cost_by_1-t/time_cost_by_n-t

where *time_cost_by_1-t* denotes the elapsed time cost by 1-t and *time_cost_by_n-t* denotes the elapsed time cost by n-t.

In order to scale the efficiency of parallel, we define Relative Parallel Efficiency (RPE),

RPE=speedup_ratio/n

where *speedup_ratio* is the speedup ratio of PSJ with n threads, and *n* is the number of threads.

For Q1, the speedup ratio is 1.74 for 2-t, 2.68 for 4-t and 3.06 for 8-t. The corresponding RPE is 87%, 67% and 38.25%. We can see that with the number of used threads increases, the value of RPE decreases sharply. The same case occurs with all the other XPath queries. For Q5, the speedup ratio values of 2-t, 4-t and 8-t are 1.06, 0.66 and 0.59 respectively. As the number of used threads increases, the value of speedup ratio doesn't increase but decreases. There are two main reasons for this. Firstly, as the amount of computing is not enough, we can't take full advantage of all threads, and starting and scheduling more threads cost more time. For Q6, there are only two valid buckets after partitioning, if we use two threads to evaluate it, each thread can be assigned to one valid bucket; and if we use four threads, at least two threads can't be assigned to valid buckets, but starting and scheduling these threads need extra more time. So 4-t and 8-t for Q6 even perform worse than 2-t. The same case occurs with Q5. Secondly, when using more threads, we must spend more time in keeping correctly accessing memory, specially writing into memory. In the function *SJ_Stack_Tree_Desc* there are many operations of writing into memory in line 21 in Fig. 2, and for 4-thread or 8-thread, the program will cost much extra time to schedule all threads accessing the shared memory.

From Fig. 3 we can know that algorithm PSJ obtains a good speedup ratio. All the speedup ratio values of 2-t on XMark and Treebank are beyond 1.62, for Q7 the value reaches 1.90. The average ratio for Q1~Q4, Q7~Q9 reaches 1.76, and RPE reaches 88%. And the speedup ratio value of 4-t for Q7 reaches

3.45, 8-t 4.97, and the corresponding RPE reaches 86.3% and 62.1% respectively.

To summarize, the experimental results in this section show that PSJ is an efficient parallel algorithm which outperforms traditional structural join algorithms. .

VI. CONCLUSION

As multi-core CPUs become more and more popular, parallel algorithms for XPath and XQuery processing become more and more stringent and important. We have had a good attempt n this paper in the aspect of parallel processing. We proposed the algorithm PSJ to address this problem, which is an efficient algorithm for shared-memory multi-core systems. PSJ first partitions AList and DList into different buckets, and then evaluates structural join in each bucket in parallel. We also present some optimization techniques. PSJ outperforms the standard structural join even with only one thread. And PSJ obtains a good speedup ratio. Our experiments have shown that PSJ is a good parallel algorithm for evaluating XPath.

For future works, we want to devise much stronger algorithms to evaluate complex XPath and XQuery queries in parallel. We want to implement twig join algorithm in parallel.

VII. ACKNOWLEDGEMENT

This work is partly supported by the Intel Semiconductor (US) Ltd., the National Natural Science Foundation of China under Grant No.60573094, the National High Technology Development 863 Program of China under Grant No. 2006AA01A101, and the National Grand Fundamental Research 973 Program of China under Grant No.2006CB303103.

REFERENCES

[1] Clark Jamex et al. XML path language (XPath). http://www.w3.org/TR/xpath
[2] Scott Boag et al. XQuery: An XML Query Language. http://www.w3.org/TR/xquery/
[3] Alin Deutsch, Mary Fernandez, Daniela Florescu et al. A query language for XML. WWW, 1999
[4] D. Florescu, D. Kossman. Sorting and Querying XML Data using an RDBMS. IEEE Data Engineering, 1999.
[5] S. AI-Khalifa, H.V. Jagadish, et al. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In Proceedings of the 18th International Conference on Data Engineering. San Jose, California, USA, 2002. 141~152
[6] Nicolas Bruno and Nick Koudas and Divesh Srivastava. Holistic Twig Joins:Optimal XML Pattern Matching. SIGMOD 2002, P310-321
[7] Haifeng Jiang at al. Holistic Twig Joins on Indexed XML Documents. VLDB 2003
[8] Jiaheng Lu and Tok Wang Ling and Chee-Yong Chan and Ting Chen. From Region Encoding To Extended Dewey: On Efficient Processing of XML Twig Pattern Matching. VLDB 2005, P193-204
[9] T. Chen and J. Lu, and T.W. Ling. On Boosting Holism In XML Twig Pattern Matching Using Structural Indexing Techniques. SIGMOD 2005, P455-466
[10] Guoliang Li, Jianhua Feng, et al. Exploiting Even Partition to Accelerate Structure Join. IEEE XWICT 2006.
[11] J.H. Feng, Guoliang Li, Lizhu Zhou et al. BBTC: A New Update-supporting Coding Scheme for XML Documents. WAIM, 2005
[12] Wei Lu, Kenneth Chiu, Yinfei Pan. A Parallel Approach to XML Parsing. 2006 7th IEEE/ACM International Conference on Grid Computing, p223-230
[13] Wei Lu, Dennis Gannon. Parallel XML Processing by Work Stealing. SOCP 2007
[14] Xiaogang Li. Efficient and Parallel Evaluation of XQuery. Doctor Dissertation. The Ohio State University, 2006
[15] OpenMP: Simple, Portable, Scalable SMP Programming. http://www.openmp.org/drupal/
[16] http://www.xml-benchmark.org
[17] http://dblp.uni-trier.de/xml/
[18] G. Miklau. UW XML Repository. http://www.cs.washington.edu/research/xmldatasets
[19] C. Zhang and J. F. Naughton and D. J. DeWitt and Q. Luo and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. SIGMOD 2001, P425-436
[20] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. VLDB 2001, p361-370
[21] S.Y. Chien and Z. Vagena and D. Zhang and V. J. Tsotras, and C. Zaniolo. Efficient Structural Joins on Indexed XML Documents. VLDB 2002, P263-274
[22] T. Grust. Accelerating XPath Location Steps. SIGMOD 2002, p109-120
[23] H.F. Jiang and H.J. Lu and Beng Chin Ooi and Wei Wang. XR-Tree:Indexing XML Data for Efficient Structural Joins. ICDE 2003
[24] Y. Wu and J. Patel, and H. Jagadish. Structural join order selection for XML query optimization. ICDE 2003, P443-454
[25] B. Choi and M. Mahoui and D. Wood. On the Optimality of Holistic Algorithms for Twig Queries. DEXA 2003, P28-37
[26] Jiaheng Lu and Ting Chen and Tok Wang Ling. Efficient Processing of XML Twig Patterns with Parent Child Edges: A Look-ahead Approach. CIKM 2004, p533-542.
[27] T. Chen and J. Lu, and T.W. Ling. On Boosting Holism In XML Twig Pattern Matching Using Structural Indexing Techniques. SIGMOD 2005, P455-466
[28] Christian Mathis and Theo Harder and Michael Peter Haustein. Locking-aware structural join operators for XML query processing. SIGMOD 2006, P467-478
[29] Songting Chen and Hua-Gang Li and Junichi Tatemura and Wang-Pin Hsiung and Divyakant Agrawal and K. Selcuk Candan. Twig^{2}Stack: Bottom-up Processing of Generalized Tree Pattern Queries over XML Documents. VLDB 2006
[30] Guoliang Li, Jianhua Feng, Lizhu Zhou. Efficient Holistic Twig Joins in Leaf-to-Root Combining with Root-to-Leaf Way. DASFAA 2007, Bangkok, Thailand.