Task Parallelism for TwigStack Algorithm on a Multi-core System

Imam MACHDI†, Toshiyuki AMAGASA††, and Hiroyuki KITAGAWA††

† Graduate School of Systems and Information Engineering
†† Center for Computational Sciences
University of Tsukuba
Tennodai 1–1–1 Tsukuba Ibaraki 305-8573
E-mail: †machdi@kde.cs.tsukuba.ac.jp, ††{amagasa,kitagawa}@cs.tsukuba.acjp

Abstract The advancement of multi-core processor technology has changed the course of computing and enabled us to maximize the computing performance. In this study, we present an approach of task parallelism for the TwigStack algorithm on a multi-core system to find all occurrences of an XML query twig pattern in a large XML database. The TwigStack agorithm comprises two tasks. The first task is to perform query pattern matching against XML data and to generate partial solutions. Meanwhile, the second task is to merge the partial solutions generated by the first task for final solutions. To achieve query performance improvement, the task parallelism employs the pipeline parallelism technique for the two tasks. The experimental results show that the query performance is significantly improved especially for queries having relatively more complex structures and/or higher selectivities.

Key words XML query processing, parallel TwigStack algorithm, task parallelism

1. Introduction

The family of holistic twig joins has appeared as the major important algorithms for processing XML query patterns due to its efficiency and performance advantage. The TwigStack algorithm [3], the initial holistic twig join algorithm, has features of performing simultaneous scan over streams of XML nodes to match their structural relationships holistically, reducing a number of unnecessary intermediate results, and skipping XML nodes that will not contribute to final answers. These features have been inherited intensively by many other algorithms such as Twig2Stack [4], TwigList [14], TwigMix [6], etc.

Recently, multi-core processors have advanced rapidly to alter the course of sequential to parallel computing. They are able to alleviate the overwhelming computation and increase the performance in data-intensive processing systems [2], [5], [7], [13]. Not only allocating partitioned data on different CPU-cores is important to create data parallelism, but allocating tasks for task parallelism may also significantly increase parallelism that eventually attains higher performance.

In this paper, we particularly study task parallelism for the TwigStack algorithm on a multi-core system. It is aimed at executing a single long-running query for intra-query parallelism. The TwigStack algorithm is decomposed into two major tasks, which are associated with the first phase and the second phase of the algorithm. The tasks are parallelized by adopting the pipeline parallelism technique. The first task performs query pattern matching on its own partitioned XML data and generates partial solutions as the outputs. It, then, transfers its outputs periodically to the second task for merging the partial solutions for final solutions such that parallelism is created by overlapping the computation of the second task with of the first task. To achieve optimal parallelism we estimate the size of partial solutions (solution granularity) for each transfer based on the parallel time analysis.

Our main contributions are outlined as follows: (i) We propose task parallelism for TwigStack algorithm using a pipelining technique. (ii) We devise time analysis of performing the task parallelism and estimate the solution granularity for each transfer from the first task to the second task. (iii) We perform some experiments to demonstrate the performance of our proposed technique.

The rest of this paper is organized as follows. Section 2 briefly reviews the related work about parallelization of XML query execution. Section 3 gives some preliminary notions related to TwigStack algorithm. Section 4 describes the XML data partitioning method. The proposed parallel algorithm is explained in Section 5. Section 6 reports experimental results. Finally, we conclude our work in Section 7.

2. Related Work

We briefly review some works of parallel XML query processing that have been extensively studied on multi-core systems.

Liu et al. [8] proposed a parallel structural join algorithm on a multi-core system. They addressed a partitioning technique by creating buckets of XML element lists and allocating them onto CPU-cores for workload balance. Each structural join (parent-child or ancestor-descendant relationship) was evaluated in parallel. As the nature of structural join algorithm gives large intermediate results, the parallel structural join also yields similar characteristics. In relation with structural join algorithm, Mathis et al. [12] proposed a set of locking-aware operators for twig pattern query evaluation to ensure data consistency. The operators were provided for flexible concurrent access on paths of structural relationships to achieve better performance.

Lu and Gannon [9] presented parallel XML processing by work stealing on multi-core processors. It provided a general framework of efficiently coordinating multi-core computation, but it was not designed for a specific parallel XML processing technique. The framework had a dynamic load balancing mechanism where a process computed XML processing tasks and might steal tasks off queues of other processes. To reduce the contention of accessing tasks, tasks in queues were ordered. A process that owned queues computed the task at the top of the queues whereas stealing was done at the bottom.

Bordawekar et al. [2] and Waldvogel et al. [15] proposed XML sub-tree partitioning for parallel XPath query processing. They had a common basic idea that sub-trees at a certain level were partitioned and distributed onto different CPU-cores, while upper nodes were shared among CPU-cores. However, the work of Waldvogel et al. [15] had several different strategies to perform sub-tree partitioning in order to cope with different characteristics of XML tree structures such as level split to partition sub-trees at a certain level for homogeneous structure of XML data, fanout split to partition sub-trees on their roots for sub-trees that are larger than a specified threshold, and semantic split to partition sub-trees on their roots for the root nodes having the same name as their siblings.

Our previous work [10] proposed a parallel TwigStack algorithm based on data parallelism on a multi-core system. The proposed scheme performed XML data partitioning on-the-fly into two levels. The first level of partitioning created buckets of XML streams and aimed at achieving workload balance and avoiding data dependency among buckets. Each bucket was allocated on a CPU-core in the system. The second level created finer partitions in each bucket such that the size of finer partitions was fit in Level-2 cache capacity for the purpose of providing finer parallelism. The entire TwigStack algorithm was executed for each finer partition on a CPU-core in parallel with other CPU-cores.

It can be noticed clearly that none of the works aims at parallel TwigStack algorithm based on task parallelism. In this work, we extend our previous work of data parallelism to task parallelism for the parallel TwigStack algorithm.

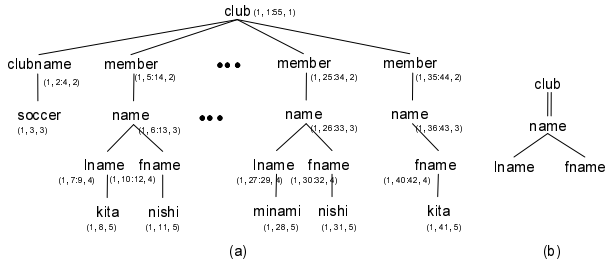

図 1 (a) XML tree representation, (b) a query twig pattern.

3. Preliminaries

In this section, we present briefly some concepts related to the TwigStack algorithm proposed by Bruno et al. [3].

3.1 XML Data and Query Twig Pattern

An XML document can be modeled as a rooted ordered tree that consists of a set of XML nodes including a root, elements, attributes and strings, and a set of edges between two nodes describing a parent-child relationship. The position of every XML node is labeled as 3-tuple (*DocId*, *LeftPos* : *RightPos*, *Level*). As mentioned in the works of [16], [1], [3], this 3-tuple labeling is used as means of describing structural relationship between two XML nodes either a parent-child relationship or an ancestor-descendant relationship. Figure 1 (a) shows XML data represented as a tree model.

A twig query pattern can be modeled as a small rooted tree consisting of query nodes including a root, elements and strings and a set of edges between two query nodes. An edge describes either a parent-child relationship (/) or an ancestor-descendant relationship (//). The term a *query* is simply to represent a query twig pattern in this paper. An example of a query twig pattern is illustrated in Figure 1 (b).

An XML database stores XML nodes of XML documents where each XML node is represented in 3-tuple. The XML database is capable of retrieving a stream of XML nodes by executing a function $\phi(t)$ where t is a given query node name; a stream is a sequence of XML nodes having the same node type and ordered by (*DocId*, *LPos*). For processing a query, associated with each query node there is a stream of XML nodes retrieved from the XML database using the function $\phi(t)$.

3.2 TwigStack Algorithm

The TwigStack algorithm solves the problem: "given a query twig pattern, in an XML database find all occurrences of tree nodes satisfying the specification the query twig pattern". The algorithm basically comprises two phases. The first phase is to perform query pattern matching and to generate root-to-leaf path solutions (partial solutions). The second phase is to merge the partial solutions to give the final answers.

In the first phase, it firstly computes solution extensions that certainly give solutions to individual partial solutions by traversing all input streams simultaneously and matching XML nodes in the streams holistically with the specified query pattern. XML nodes in the solution extensions are stored in their respective stacks that encode root-to-leaf paths for generating the partial solutions. The computational complexity is $O((n+2\beta)\cdot|S|)$ where n is the number of query nodes, β is a query selectivity for estimating the size of partial solutions, and $|S|$ is the size of XML node streams.

In the second phase, the partial solutions are simply merged according to their common root nodes for the final answers that satisfy the query twig pattern. The computational complexity is $O((\beta+\gamma)\cdot|S|)$ where γ is a query selectivity for estimating the size of final solutions.

4. XML Data Partitioning

XML data partitioning is fundamental for creating parallelism. We adopt the stream-based partitioning for XML (SPX) devised in our previous work [11]. The objective of SPX is to partition streams of XML nodes so that each partition contains complete solution nodes and has no dependency on other partitions. Also, the SPX is suitable for on-the-fly execution since it characterizes itself as fast and straightforward computation.

4.1 Basic Notion

Before explaining how to partition streams, we define a range containment property used as the basis of computing partitions. The range containment property for a range applies to two streams (sub-streams). A range of a stream (sub-stream) is indicated by the first XML node *mostL* and the last XML node *mostR* in the stream (sub-stream). We define the range containment property of an ancestor stream (sub-stream) S_a and a descendant stream (sub-stream) S_d if $S_a.mostL < S_d.mostL$ and $S_d.mostR < S_a.mostR$.

A partition is defined as a set of sub-streams associated with query nodes where any of two sub-streams satisfies the range containment property according to the structural relationships of their associated query nodes.

Given XML node streams associated with query nodes, initially the largest stream is subdivided into sub-streams

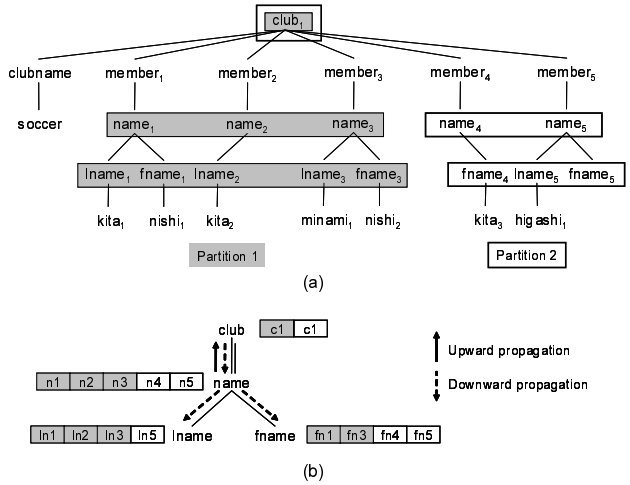

図 2 (a) Overview of partition, (b) Partition propagation.

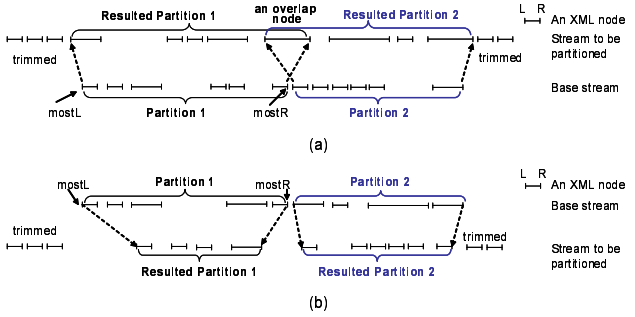

図 3 (a) Upward propagation, (b) Downward propagation.

according to a specified range. For each sub-stream of the largest stream, the algorithm subdivides other streams into their sub-streams through propagation to satisfy the range containment property between two sub-streams. Starting from the initial sub-streams, the propagation goes upward to subdivide its ancestor streams until reaching the root stream; it is called upward propagation. Subsequently, the root stream starts downward propagation to subdivide the rest of the descendant streams. Figure 2 (a) shows an overview of two partitions and Figure 2 (b) illustrates the direction of partition propagation.

4.2 Partition Propagation

Partition propagation comprises the upward propagation and the downward propagation. The upward propagation partitions an ancestor (a parent) stream to satisfy the range containment property according to the given descendant (child) sub-streams as the base stream. As the result of upward propagation, XML nodes belonged to two partitions are duplicated to preserve data independence among partitions. Also, some XML nodes that will not contribute to solutions are trimmed and excluded from partitions. Illustration of the upward propagation is shown in Figure 3 (a).

The downward propagation performs partitioning in similar way, but in the opposite direction of the upward propa-

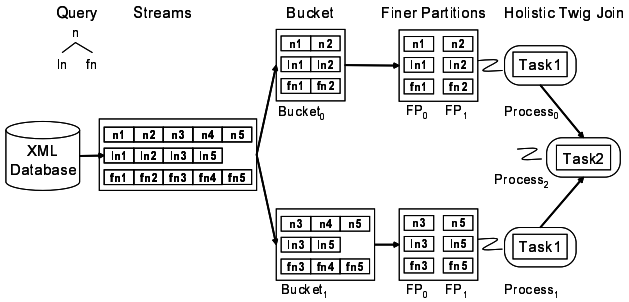

図 4 Overview of task parallelism.

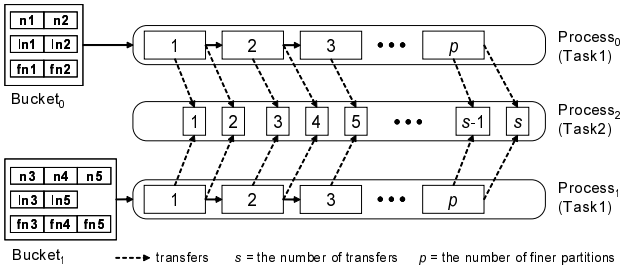

図 5 Execution path of the TwigStack algorithm based on task parallelism.

gation. It partitions a descendant (a child) stream to satisfy the range containment property according to the given ancestor (parent) sub-streams as the base stream. As illustrated in Figure 3 (b), some XML nodes are also excluded from partitions because they will certainly not contribute to final solutions.

5. Parallel TwigStack Algorithm

In this section, we will describe our proposed parallel TwigStack algorithm based on a task parallelism technique. The algorithm of task parallelism is outlined in Algorithm 1.

5.1 Parallel Algorithm Overview

As illustrated in Figure 4, given a query streams of XML data are retrieved from an XML database and partitioned into buckets on-the-fly by the stream-based partitioning for XML as explained in the previous section. The number of buckets created is associated with the number of CPU-cores available in the system. The objective of creating buckets is to balance workloads among CPU-cores and to avoid data dependency that certainly annihilates memory access races. Within each bucket, partitioning is further performed to generate finer partitions for creating finer parallelism and reducing memory access contention.

There are two different tasks to perform task parallelism in the system. Task one is associated with the first phase of the TwigStack algorithm, while task two is associated with the second phase of the algorithm. The number of processes for task one is dependent on the number of buckets created. Task parallelism is conducted by transferring the partial outputs of task one periodically to task two. While task one is performing the first phase of the TwigStack algorithm, task two consumes the partial outputs and merges them for final solutions.

5.2 Task Parallelism

Task parallelism is constructed from two tasks by adopting a pipeline parallelism technique as described in the previous sub-section. The main objective is to hide the computation of task two by overlapping with the computation of task one.

It is important to balance workloads among tasks. The workload is measured according to the computational complexity of task one and task two, which are already specified in Section 3.2. Given a number of tasks one, a certain number of tasks two is determined by the ratio of the computational complexity of task one to of task two.

In the pipeline parallelism, higher parallelism can be achieved by controlling the size of transferred outputs (solution granularity) from task one to task two. The parallelism occurs when task two consumes and merges the partial outputs simultaneously with some tasks one performing query pattern matching and generating partial solutions.

Algorithm 1 Task Parallelism

```
1: nprocs ← Number_CPUCores ()
2: proc_pool ← CreateProcess (nprocs)
3: proc1_pool ← DecomposeTask1 (proc_pool) /*tasks one */
4: proc2_pool ← DecomposeTask2 (proc_pool) /*tasks two*/
5: g ← SolutionGranularity (proc1_pool, proc2_pool, β, streams)
6: num_parts ← NumberFinerPartitions (q_stats, streams)
7: for i = 0 to number(proc1_pool) - 1 do
8:    bucket_i ← CreateBucket (q, streams)
9: end for
10: for all proc_i ∈ proc_pool in parallel do
11:    /* processes of task one */
12:    if proc_i ∈ proc1_pool then
13:       for p = 0 to num_parts - 1 do
14:          partition_{i,p} ← CreateFinerPartition (q, bucket_i)
15:          rtl_sols_i ← rtl_sols_i ⋃ Task1 (q, partition_{i,p})
16:          if |rtl_sols_i| ≥ g then
17:             Synch, Put rtl_sols_i in buffer
18:          end if
19:       end for
20:    end if
21:    /* processes of task two */
22:    if proc_i ∈ proc2_pool then
23:       while ¬ Empty(buffer) AND ¬ End do
24:          Synch, Get rtl_sols in buffer
25:          finalsols ← finalsols ⋃ Task2 (rtl_sols)
26:       end while
27:    end if
28: end for
```

5.3 Time Analysis

As shown in Figure 5, the execution path of the parallel TwigStack algorithm based on task parallelism consists of sequential and parallel parts as follows:

- Time for generating buckets in sequential (T_{bucket}),
- Time for performing task one in parallel (T_{task1}/P),
- Time for performing the remaining task two in sequential (T_{task2}/s),
- Overhead time for transferring partial solutions from tasks one to task two in sequential ($s \cdot P \cdot t_s$), where P is the number of processes of task one, s is the number of transfers, and t_s is the unit time of the transfer overhead.

We can see that the computation of task two is almost completely hidden because most of its computation is performed in parallel with the computation of tasks one.

The execution time can be expressed as follows:

$$T_{taskpar} = T_{bucket} + \frac{T_{task1}}{P} + \frac{T_{task2}}{s} + s \cdot P \cdot t_s \qquad (1)$$

Higher performance of the parallel TwigStack algorithm can be achieved if the parallel portions of computing task one and task two are higher than the sequential portions. A more complex structure of a query will lead to higher parallelism on task one, while a higher query selectivity will lead to higher parallelism on task two.

5.4 Estimation of Solution Granularity

Based on the time analysis in Eq. 1, the optimal execution time of a query is dependent on the number of transfer s from task one to task two. If the transfer occurs more frequently (larger s value), the performance is degraded due to higher transfer overheads; the term $(s \cdot P \cdot t_s)$ tends to increase. On the other hand, if the transfer occurs less frequently (smaller s value), it reduces the benefits of pipeline parallelism; the term of (T_{task2}/s) tends to increase.

To estimate the optimal value of s, the Eq. 1 taken as a function of s is optimized for its first derivative equal to zero with respect to s parameter, while other parameters are fixed. As the result of optimizing the function, the value s is obtained from Eq. 2.

$$s = \sqrt{\frac{T_{task2}}{P \cdot t_s}} \qquad (2)$$

Solution granularity g is the size of root-to-leaf path solutions for each transfer. Whenever a process of task one generating root-to-leaf path solutions reaches the solution size equal to or larger than the specified solution granularity, it transfers the solutions to the process of task two (line 16–18). Assume that XML nodes in the root-to-leaf path solutions are distributed uniformly. The solution granularity g can be directly computed from the estimated size of the entire root-to-leaf path solutions over the number of transfers s and expressed as follows:

$$g = \frac{\beta |S|}{s} \qquad (3)$$

where β is the query selectivity to estimate the total size of root-to-leaf path solutions and $|S|$ is the total stream size (line 5).

6. Experimental Evaluation

We conducted extensive experiments to evaluate the performance of our proposed parallel TwigStack algorithm. This section starts with a description of XML data set and our experimental platform, the estimation of solution granularity, and the parallel performance on multi-cores.

6.1 Experimental Setup

The experiment used a synthetic XML data set generated by XMark generator with the sizes of 1 GB, 2 GB, 3 GB, and 4 GB. We also specified five queries with different structural complexity and selectivity as shown in Table 1.

The platform we used has four-core processors of AMD Opteron-280 and 16 GB memory under Solaris-10 operating system. We used PostgreSQL 8.2 as the XML database to store XML nodes. We tested our implementation using C++ with pthread and MPI.

表 1 Query structure and selectivity.

QID	Query	β	γ
Q1	//item//text	0.97	0.83
Q2	//item[/description//text]/mailbox//date	0.48	0.35
Q3	//category[/name]/desc//text	0.04	0.03
Q4	//open_auction//annotation[/author][/description]/happiness	0.89	0.59
Q5	//person[/name][/address[/city][/province]/country]/profile	0.30	0.19

表 2 Solution granularity for different number of CPU-cores.

QID	Stream Size	Solution Granularity		
		2 Cores	3 Cores	4 Cores
Q1	1,139,840	815	999	1,153
Q2	2,549,040	788	965	1,115
Q3	1,785,930	193	236	273
Q4	1,092,460	674	826	954
Q5	1,172,830	393	482	556

6.2 Estimation of Solution Granularity

Estimating the solution granularity requires some parameters as specified in Eq. 2 and 3. The estimation results for 1 GB XML data are shown in Table 2. As the number of CPU-cores increases, the solution granularity gets larger size or, in other words, the transfer of root-to-leaf path solutions gets less frequent. In this case, the system reduces the transfer overheads to make balance with the computational

表 3 Sequential execution time.

QID	Time (secs)	QID	Time (secs)
Q1	5.18	Q2	11.42
Q3	4.39	Q4	6.12
Q5	4.21		

workloads among processes. On the other hand, if fewer number of CPU-cores is involved in the parallel query processing, the system sets the solution granularity to smaller size and transfers the root-to-leaf solutions more frequently.

6.3 Parallel Performance

In this experiment, we measured the performance of task parallelism against the performance of our previous data parallelism in terms of speedup, efficiency and scalability. In the data parallelism technique, each process handles the computation of task one and task two against its assigned bucket of XML node streams. Before measuring the parallel performance, the sequential execution time of all queries is measured. Note that in this measurement streams of XML nodes are not partitioned at all. Table 3 shows the sequential execution time for 1 GB XML data.

Speedup performance measures the increasing performance on increasing number of CPU-cores for 1GB XML data. Results of the speedup measurements are illustrated in Figure 6. For all queries, as the number of CPU-cores increases, the speedup curves escalate. For task parallelism, at four CPU-cores queries Q2 and Q5 achieve 3.9 as the highest speedup, while query Q3 achieves 2.9 as the lowest speedup. In comparison with data parallelism, all speedup curves of task parallelism attain higher performance; thus, the task parallelism technique is able to improve the parallel performance of the data parallelism technique. In more details, higher speedup improvement is achieved by query Q4 and Q5, while the lowest speedup improvement is achieved by query Q3. Since the task parallelism technique is aimed at improving the performance of task two, which is directly related to the query selectivity, the behavior of query Q3 having the smallest query selectivity shows naturally the smallest speedup performance and improvement. Meanwhile, the behaviors of other queries show better speedup performance and improvement.

We measured the parallel efficiency to indicate the CPU-core utilization, which is computed as the ratio of the gained speedup to the number of CPU-cores utilized. Figure 7 shows the efficiency measurements. In overall, the efficiency of task parallelism is higher than of data parallelism because of higher speedup attainment by the task parallelism. As the number of CPU-cores increases, the efficiency tends to decline. Query Q3 incurs the lowest improvement of efficiency, while others have higher efficiency improvement. If we look

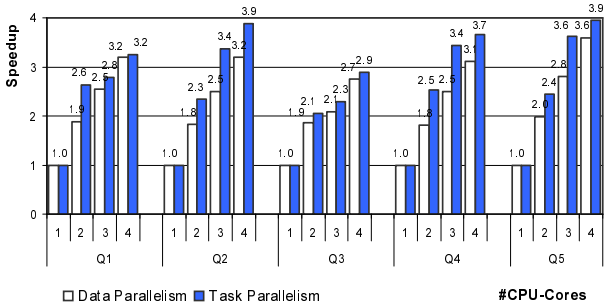

図 6 Speedup performance of data parallelism and task parallelism.

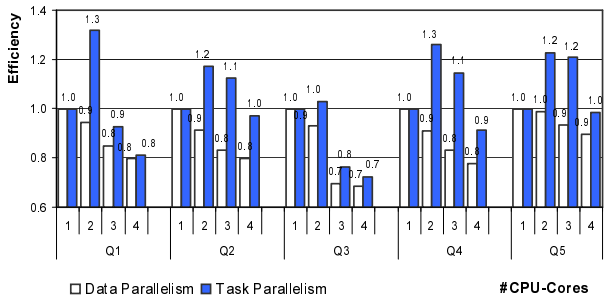

図 7 Efficiency of data parallelism and task parallelism.

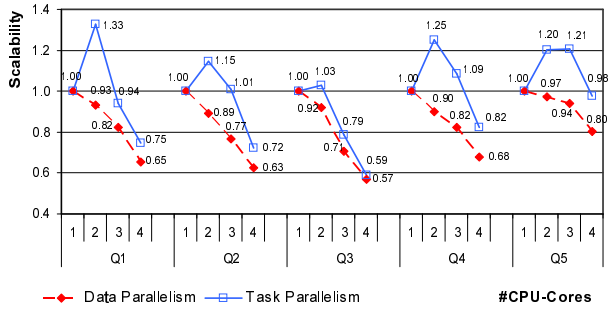

図 8 Scalability of data parallelism and task parallelism.

at more details, some measurements of task parallelism exceed 100% efficiency for two reasons. Firstly, our partitioning method features the reduction of a number of unnecessary XML nodes that certainly do not contribute to final solutions. Meanwhile, as for the speedup measurement, the sequential execution does not take the partitioning method into account. Secondly, the proposed task parallelism is able to hide almost completely the computation of task two.

To measure the effectiveness of our proposed task parallelism technique to utilize a number of CPU-cores with varying sizes of XML data, we measured the scalability performance. We used a scaling function $f(x) = x$, where x is the number of CPU-cores involved and $f(x)$ is the XML data size in GB. Figure 8 shows the scalability performance. In general, as the x value increases, the scalability is more likely to drop. Clearly, the scalability of task parallelism improves the scalability of data parallelism significantly. In more de-

tails, queries Q4 and Q5 are able to maintain the scalability above 0.82 and 0.98, respectively. Query Q3 maintains its scalability above 0.57 as the least scalable. Queries having more complex structures and higher selectivities tend to have higher scalability performance. Some queries at particular number of CPU-cores achieve super-linear scalability whose values are more than 1.0. In this case, the same reasons as mentioned in the efficiency measurement are applied.

7. Conclusions and Future Work

In this paper, we proposed the task parallelism technique that extends our data parallelism technique for parallel TwigStack algorithm on a multi-core system. We devised a pipeline parallelism technique to create parallelism between task one and task two that correspond with the first and the second phases of the TwigStack algorithm, respectively. The technique aims at hiding the computation of task two by overlapping with the computation of task one. In addition, we proposed the time analysis to predict the behavior of queries and an estimation method to derive the solution granularity for optimal query execution. The experimental results confirmed that our proposed task parallelism technique outperformed the data parallelism technique and showed good performance in terms of speedup, efficiency and scalability, particularly for queries having more complex structures and higher selectivities. Moreover, in some cases the performance of task parallelism may achieve super-linear scaleup and very high efficiency.

Some research issues remain for further study. In the future, we will focus our study on the performance improvement of parallelizing not only long-running queries by intra-query parallelism, but also short-running queries by inter-query parallelism. For this purpose, similarity of query patterns and query caching can be explored thoroughly in the context of parallel holistic twig join algorithms.

Acknowledgements

This study has been partially supported by Grant-in-Aid for Young Scientists (B) (#21700093) and Grant-in-Aid for Scientific Research on Priority Areas from MEXT (#21013004).

文　献

[1] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 141–152, 2002.

[2] R. Bordawekar, L. Lim, and O. Shmueli. Parallelization of XPath Queries Using Multi-core Processors: Challenges and Experiences. In *Proceedings of the International Conference on Extending Database and Technology (EDBT'09)*, pages 180–191, 2009.

[3] N. Bruno, N. Koudas, and D. Srivastava. Holistic Twig Joins: Optimal XML Pattern Matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 310–321, 2002.

[4] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig2Stack: Bottom-up Processing of Generalized-tree-pattern Queries over XML Documents. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB '06)*, pages 283–294, 2006.

[5] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: Minimizing Cache Conflicts in Multi-core Processors for Databases. *PVLDB*, 2(1):373–384, 2009.

[6] J. Li and J. Wang. Fast Matching of Twig Patterns. In *Proceedings of the 19th International Conference on Database and Expert Systems Applications (DEXA'08)*, pages 523–536, 2008.

[7] X. Li, H. Wang, T. Liu, and W. Li. Key Elements Tracing Method for Parallel XML Parsing in Multi-core System. In *Proceedings of the International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT'09)*, pages 439–444, 2009.

[8] L. Liu, J. Feng, Q. Qian, and J. Li. Parallel Structural Join Algorithm on Shared-Memory Multi-core Systems. In *Proceedings of the Ninth International Conference on Web-Age Information Management (WAIM'08)*, pages 70–77, 2008.

[9] W. Lu and D. Gannon. Parallel XML Processing by Work Stealing. In *The 2007 Workshop in Service-oriented Computing Performance: Aspects, Issues, and Approaches (SOCP'2007)*, pages 31–38, 2007.

[10] I. Machdi, T. Amagasa, and H. Kitagawa. Executing Parallel TwigStack Algorithm on a Multi-core System. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications and Services (iiWAS'09)*, pages 174–182, 2009.

[11] I. Machdi, T. Amagasa, and H. Kitagawa. XML Data Partitioning Schemes for Parallel Holistic Twig Joins. *International Journal of Web Information Systems*, 5(2):151–194, June 2009.

[12] C. Mathis, T. Harder, and M. Haustein. Locking-Aware Structural Join Operators for XML Query Processing. *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 467–478, 2006.

[13] Y. Pang, W. Hu, L. Sun, and S. Yang. Adaptive Data-driven Parallelization of Multi-view Video Coding on Multi-core Processor. *Science in China Series F: Information Sciences*, 52(2):195–205, 2009.

[14] L. Qin, J. Yu, and B. Ding. TwigList : Make Twig Pattern Matching Fast. In *Proceedings of the 12th International Conference on Database Systems for Advanced Applications (DASFAA 2007)*, pages 850–862, 2007.

[15] M. Waldvogel, M. Kramis, and S. Graf. Distributing XML with Focus on Parallel Evaluation. In *Proceedings of Databases, Information Systems and Peer-to-Peer Computing (DBISP2P'08)*, pages 55–67, 2008.

[16] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On Supporting Containment Queries in Relational Database Management Systems. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of data*, pages 425–436, 2001.