ParaXML: A Parallel XML Processing Model on the Multicore CPUs

Wei Lu, Dennis Gannon

Computer Science Department, Indiana University
{welu,gannon}@cs.indiana.edu

Abstract

XML has emerged as the de facto *standard interoperable data format for the web service, the database and document processing systems. The processing of the XML documents, however, has been recognized as the performance bottleneck in those systems; as a result the demand for high-performance XML processing grows rapidly. On the hardware front, the multicore processor is increasingly becoming available on desktop-computing machines with quad-core shipping now and 16 core system within two or three years. Unfortunately almost all of the present XML processing algorithms are still using serial processing model, thus being unable to take advantage of the multicore resource. We believe a parallel XML processing model should be a cost-effective solution for the XML performance issue in the multicore era. In this paper, we present a general-purpose parallel XML processing model, ParaXML, designed for multicore CPUs. General speaking, ParaXML treats the XML document as the general tree structure and the XML processing task as the extension from the parallel tree traversal algorithm for the classic discrete optmization problems. The XML processing, however, has quite distinct characteristics from the classic discrete optmization problems, thus demanding the special treatments and the fine-grained tuning technologies. ParaXML internally adopts a fine-grained work-stealing scheme to dynamically control the load balance among the parallel-running threads, and a novel approach is also introduced to trace the stealing actions and the running results to facilitate the reducing of those parallel-running results. Besides, ParaXML provides the tuning options, particularly for the large XML documents, to control the trade-off between the parallelism gain and task-partitioning overhead. To show the feasibility and effectiveness of the ParaXML model, we demonstrate our parallel implementations of three fundamental XML processing tasks based on the ParaXML: traversal, serializing and parsing. The empirical study in this paper shows that those parallel implementations substantially improved the performance and scale well on a multicore machine.*

1 Introduction

As manufacturers have encountered difficulties to further exponential increases in clock speeds, they are increasingly utilizing the march of Moore's law to provide multiple cores on a single chip. The multicore processor is rapidly becoming the mainstream on desktop-computing machines with quad-core shipping now and 16 core system within two or three years. Tomorrows computers will have more cores rather than exponentially faster clock speeds. The architectural changes benefit only those software with thread level concurrency in mind and therefore have little value for most existing software. Consequently, the good time when people merely rely on the faster CPU clock to speed up the software execution is over [25], the software must exploit the parallelism and concurrency somehow so that it can take advantage of multicore resource.

By overcoming the problems of syntactic and lexical interoperability, the acceptance of XML as the *lingua franca* for information exchange has freed and energized researchers to focus on the more difficult (and fundamental) issues in large-scale systems. The very characteristics of XML that have led to its success, however, such as its verbose and self-descriptive nature, can incur significant performance penalties. The XML document processing has been identified as the performance bottleneck for the large scale distributed systems and database systems [5, 16]. Here the processing refers not only to the basic XML serialization/parsing, but also to various higher level tasks, such as Schema validation, XPath query and XSLT transformation.

Various techniques have been proposed to improve the performance of XML processing, ranging from the schema-specific solutions [15, 6] to the streaming model [27], to the hardware acceleration [26]. However those approaches are either designed for a specific task, or assuming the existence of some preconditions, such as the streaming context. Moreover they are inherently designed to be serial without

any parallelism in mind, thus unable to take advantage of the multicore CPUs. Hence, we believe a novel XML processing model with the ability of exploiting the parallelism is a more general and cost-effective solution in this multicore era.

Parallelism could be used in a number of ways. One approach would be to use pipelining. In this approach, XML processing could be divided into a number of stages. Each stage would be executed by a different core. This approach may provide speedup, but software pipelining is often hard to scale well, due to synchronization, load-balance and memory access costs. The more promising and scalable one is a data-parallel approach. Here,the XML document would be divided into some number of partitions, and each core would work on the partitions independently. As all the partitions are processed, the parallel-running results are merged. A XML document essentially represents a tree-structured data model, which usually is implemented in the Document Object Model (DOM), and the document self is just the serialization of this tree model in a depth-first, left-to-right traversing order (i.e., the document order). Therefore our idea is if we can adopt those data-parallel based tree traversal formulas [20, 9] on the XML documents we can obtain the parallel XML processing algorithms naturally.

In this paper, we introduce *ParaXML*, a data-parallel XML processing model designed for the multicore system. ParaXML is not just for a specific XML processing task. Instead, it is a general-purpose model severing as the programming paradigm for the potential parallelizations of various XML tasks. ParaXML originates in the parallel tree traversal solutions of the discrete optimization problems [20, 10]; it treats the XML document as the general tree structure and the processing on the XML as the extension of the parallel tree traversal algorithm. Internally, ParaXML adopts a work-stealing engine, implemented in in C#, to dynamically partition the task and control the load balance among the multiple threads, which process disjointed partitions of the XML document in parallel.

The XML processing, however, has quite distinct characteristics from those discrete optimization problems. Unlike the discrete optimization problems, XML processing usually needs a more complicated result handling procedure, such as result accumulation. As a result, ParaXML introduced a novel way to trace the stealing actions and parallel running results so that the final result can be found and reduced. Moreover, compared with the discrete optimization problems the XML processing has the much smaller but more regular problem space, thus being more sensitive to the potential synchronization overhead. Therefore ParaXML provides several fine tuning technologies to control the trade-off between the parallelism and the related overhead.

In order to show the feasibility and effectiveness of the ParaXML model, in this paper we present our parallel implementations of three fundamental XML processing tasks: traversal, serializing and parsing. Each parallel implementation is built upon the ParaXML engine, but with its own processing task and context. The experiments in this paper shows that all the three parallel implementations (i.e., traversal, serializing and parsing) obtain the substantial performance gain and scale well on a multicore machine

Operating systems usually provide access to multiple cores via kernel threads (or LWPs). In this paper, we generally assume that threads are mapped to hardware threads to maximize throughput, using separate cores when possible. We consider further details of scheduling and affinity issues to be outside the scope of this paper.

The rest of the paper is organized as follows. Section 2 describes the background knowledge about the load balancing techniques. Then in the section 3 we present the design and implementation of the kernel of ParaXML, the stealing-based load balancing mechanism, in detail. The ParaXML model is introduced in Section 4. Section 5 describes the parallel implementations of XML traversal, serializing and parsing and their performance experiments result.

2 Load Balancing Techniques

In term of the ease of the parallelization, the tree structure is a double-edged sword. At one side it is fairly easy to partition a tree into several disjoint sub-trees or sub-forests, and each of them can be assigned to the different threads for the parallel processing. On the other side, a tree structure is a awkward one from the perspective of the load balancing. Since the size and the shape of a tree can't be estimated until it is walked through, we can't determine the real workload associated with a subtree or a subforest. It is very likely that two threads are assigned with two subtrees with various processing complexity, and when one finishes processing and becomes idle another thread is still busy for working. By the Amdahl's law [1] the imbalanced workload distribution will prevent the parallel algorithm from being scalable and efficient.

Numbers of approaches have been proposed to address the load balance issue. The static load balancing approaches [21], solve the problems by defining a cutoff depth of the tree, under which each subtree will be treated as the task for the threads. If the cutoff depth is selected properly, there will be enough subtrees and it makes the load imbalanced situation less likely happen. Obviously, the effectiveness of this approach depends on the shape of the tree structure and the cutoff depth, which usually is a priori knowledge.

Instead of planing the load balance before the parallel execution, the dynamic load balancing scheme partitions and distributes the tasks during the running time. Stealing based

scheme is the one of the approaches that have been widely used in the shared-memory environment[8, 3]. By the stealing based scheme every thread works on its own local task queue and whenever it runs out of the task it steals the task from other thread's task queue. The advantage of the stealing based scheme is that the load redistribution is on demand and dynamic, thus regardless of the shape of the tree the overall workload distribution tends to be balanced. Furthermore, the potential performance overhead is inherently low since the stealing, once happened, only interfere the victim thread. The dynamic characteristic, however, incurs more sophisticated interaction across the threads. Great care should be taken to the design and the implementation of the synchronization and communication. otherwise the their cost could easily offset the performance gain brought by the parallelism. Also the effectiveness of the stealing scheme is based on the assumption that the owner thread accesses its local task-queue much more frequently than thief thread does, and it is the thief thread, but not the owner thread, should pay any cost caused by the stealing.

3 ThreadCrew

The `ThreadPool` class of the .NET framework consists of a global task queue and a number pre-created threads, each getting/putting the task from/into the global queue. Although `ThreadPool` is a handy tool for the general concurrent programming, it is inappropriate for numbers of parallel formulas which are highly sensitive to any performance overhead. First of all, obviously the global task queue of the pool is the dominant performance bottleneck due to the intensive lock contention by all threads. Furthermore, as each new task generated by the thread will be put back to the global queue and probably be fetched by other threads, the data locality and the cache hitting ratio are going to be low.

In order to provide higher performance for the parallel tree-based processing, we implement a stealing based load balancing class `ThreadCrew`. Just as the `ThreadPool`, a `ThreadCrew` represents a set of threads. However the number of threads in a `ThreadCrew` is fixed, and all the threads are supposed to be running on the respective cores simultaneously. Instead of maintaining a global queue, each thread in the `ThreadCrew` has its own local task queue, which can minimize the content and maximize the data locality as well. When a thread is out of work from its local task queue, it tries to steal the work from other threads in the crew. The difference between the two classes is depicted in the Fig.1.

The application invokes the `ThreadCrew` by calling

```
ThreadCrew.Execute(Object initTask,
                   TaskHandle taskHandle);
```

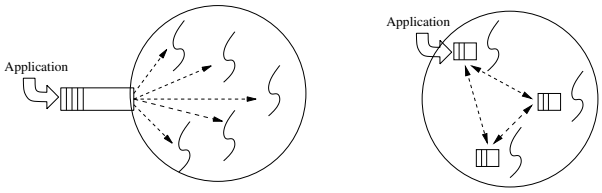

Figure 1. Figure (a) : the structure of the ThreadPool. Figure (b) : the structure of the ThreadCrew.

, where the `initTask` refers to the task the `ThreadCrew` is going to execute at the beginning and the `taskHandle` is a call-back function, which will be invoked by the `ThreadCrew` to process a task. The signature of the call-back function is defined as

```
delegate void TaskHandle(Object task,
                         Object partialResult),
```

where `task` refers the task object passed by the `ThreadCrew` and the `partialResult` argument refers the accumulated result the current thread has generated.

Each thread in the `ThreadCrew` has three running phases: **1) Waiting, 2) Working and 3) Stealing**. At the beginning, all the threads will block-wait on a barrier, which will be opened once the application assigns the initial task to the crew. The initial task is directly pushed into the first thread's task-queue, and the caller will synchronously wait on the barrier until the finishing of the initial task. Once the barrier is opened, all the threads enter the working phase simultaneously. During this phase, each thread gets the task by popping its local queue, then executes the task. If new task is generated, it will be pushed back into the local queue of the thread. The thread keeps running until the local queue is empty. At that moment, the thread becomes a *thief* and enter the stealing phase. Based on the stealing policy the thief picks one thread in the crew as the victim, and try to steal the task from the victim's task-queue. If the stealing succeeded, the thief goes back to the working phase with the stolen task; otherwise the thief keeps stealing until the termination condition is detected. The pseudo code is listed as below:

```
while (true)
{
  phase1: //waiting phase
    block-wait on a barrier;
  phase2: //working phase
    while (local task-queue not empty) {
        get task from the local task-queue;
        pass the task to the TaskHandle callback function;
    }
  phase3: //stealing phase
    while (true) {
        if (termination is detected)
            break;
        pick a victim from the other threads;
        try to steal a task from the victim's task-queue;
        if (stealing succeed)
```

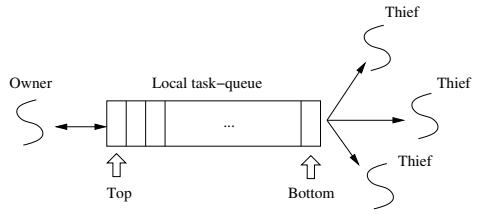

Figure 2. The stealing-oriented deque

```
            goto phase2;
    }
}
```

3.1 Lock-free Deque

In the `ThreadCrew`, those local task-queues are shared by the multiple threads. Each local task-queue is owned by its owner thread, meanwhile it may be accessed by multiple thieves simultaneously. As a shared resource, the local task-queue has to be protected to ensure the mutually exclusion. We could use a single lock to protect the whole task-queue, but this coarse grained synchronization makes the lock be a hot spot which inevitably becomes the system bottleneck. Alternatively, as shown in Fig. 2 we can enables the fine-grained synchronization by using a deque data structure on which the owner thread and the thieves access the different ends separately. Moreover the stealing scenario has one more nice characteristic: there is only one owner for each deque.

Based on that, Arora, Blumofe and Plaxton [2] proposed a lock-free stealing-oriented deque structure known as *ABP-Deque*. In ABP-Deque, the owner thread treats the deque as a normal stack and it always pops/pushes the task from the top of the deque, whereas the multiple thieves steal the task from the deque by popping from the bottom. The ABP-Deque uses the Compare-And-Swap atomic operation (i.e., CAS) instead of the lock to solve the mutual exclusion so that the threads will never be blocked for the contention. Three major methods are provided by ABP-Deque:

- PushTop() : called by the owner to push the data into the top of the stack
- PopTop() : called by the owner to pop the date from the top of the stack
- PopBottom() : called by multiple thieves to steal the data from the bottom

The crux of the APB-Deque is the owner thread doesn't need to perform any CAS operation as long as there is more than one item in the queue, while the thief always needs to call CAS for every stealing. In other word, for most time the owner thread will present the exact performance as the one of the serial execution whereas it is the thief who pays the cost of the contention. Nonetheless the thieves may also benefit from the stealing-from-bottom policy if the task at the bottom of the stack carries more work load than the one at the top. For example for a tree depth-first traversal algorithm, during the traversing the node at the bottom of the stack is the one being at the highest level of the tree, which is more likely to have more children nodes to explore.

3.2 Victim-Selection Policy

The victim-selection policy address which thread in the crew is selected by the thief thread. The simplest policy is picking the victim randomly. Another easy one is the global round-robin, in which the victim thread is picked in the round-robin fashion. Besides, `ThreadCrew` provides one more policy: Pick-The-Richest, by which the thread with longest task queue always be selected as the victim. For the present multicore processor, which has relative small number of cores, we believe the Pick-The-Richest policy should lead to the better result as it incurs less failed stealing. However when the number of cores growing the querying and sorting of the length of all the queues will be prohibited, the other two policies will be more promising.

3.3 Stealing Tracing

When executing a task the thread sometimes will generate the result, such as the query result or traversal output. During the execution, new sub-tasks will be created and pushed into the local task-queue; and they may be stolen by other threads before the owner thread gets them. Consequently, as long as the stealing happens during the execution, the generated result by the owner thread only covers the part of the final result of the original task. We call the result as the *partial result* of the task. In other word, the final result of the current task consists of the partial result generated by the owner thread and the partial results generated by the thieves.

In order to trace those partial results generated by the thieves, we modified the ABP-Deque algorithm. In our modification, after the stealing the thief is required to leave a clue, following which the future partial result can be located. The clue basically is a reference to a *Stealing-stub* object, which consists of a partial result object and a list of the references to other stealing-stubs. Before stealing the victim, the thief first generate an empty stealing stub. As shown in the Fig.3, when the `PopBottom()` methods of the victim's task-queue is invoked by the thief, instead of removing the task object at the bottom from the queue as the original ABP-queue does, the task object in the bottom entry is replaced by the reference to the empty stealing stub provided by the thief.

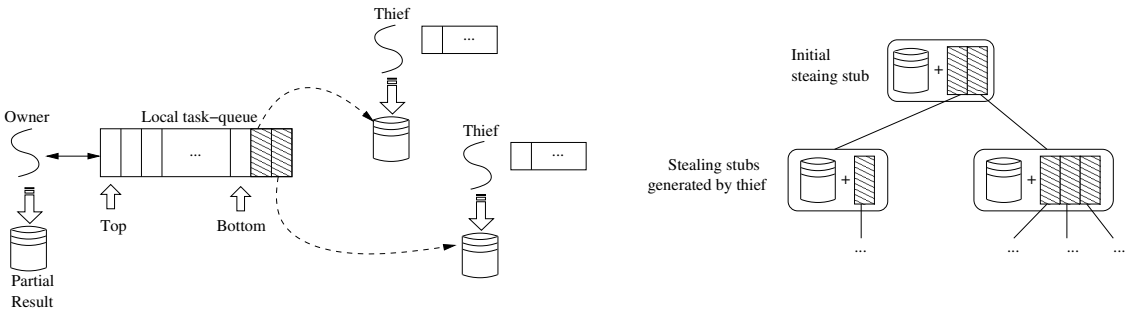

Figure 3. Left: the tracing of the stealing actions, Right: the stealing-stub tree.

After getting the task, the thief holds the empty stealing stub as its root stealing stub during the execution of the stolen task. The partial result of the stealing stub will be passed to the `taskHandle`, which may gradually change its value. Note that once a thief begins to execute the stolen task, it may generate new tasks, which may also be stolen by other threads even including the original owner from whom the task was stolen. Thus the stealing and the replacing will be done in the recursive fashion.

Once the local task-queue becomes empty, the owner thread will copy all the existent stealing stubs, each representing a stealing and referring to the result generated by the thief, from the local queue into the stealing-stub reference list of the root stealing stub. The order among the stealing stubs in the task-queue is preserved. As we will see, this order will play an important role for gluing the partial result together. As shown in the Fig.3, when the entire execution is terminated, all the stealing stubs form a tree structure rooted from the initial partial result.

4 Parallel XML Processing Model

As we mentioned, A XML document essentially represents a tree-structured data model, which usually is implemented in the Document Object Model (DOM), and the document self is just the serialization of this tree model in the document order. Most XML tasks currently are implemented in the serial model and are basically extended from the classical depth-first tree walk algorithm, so our parallel model starts at the parallel XML DOM traversal implementation.

With `ThreadCrew` it is quite easy to implement the parallel DOM traversal by simply invoking `ThreadCrew.Execute()` with the traversal `TaskHandle` shown below.

```
void TaskHandle(XmlElement task, object particalResult)
{
    visits the element;

    expand the element and from right to left push
    all its children into the local task−queue
}
```

The traversal `TaskHandle` will be invoked by the threads in the ThreadCrew whenever an element is going to be visited. Although logically the traversal `TaskHandle` has little difference from the classical serial algorithm, the internal execution of the parallel algorithm is distinct. In the parallel execution the emptiness of the local task queue doesn't necessary mean that the entire tree has been visited since it is most likely that other threads have stolen some nodes from the task queue, however this implication has to be true for a serial algorithm. Also the thread can't just stop the execution when its task queue is empty; other threads may be suffering from the heavy workload at this moment and the thread should try to steal some sub-trees from others to achieve the load balance. Finally the tree nodes fed to the `TaskHandle()` function aren't necessary to be consecutive in the document order since some nodes may be stolen from other threads, whereas the document order is guaranteed for the serial algorithm. Fortunately, the above detail has been hidden by the `ThreadCrew`, and is transparent to the user.

Although the above simple implementation is correct, it has no way to obtain any performance win by the parallelism for the real XML tasks. That is because XML documents have quite distinct characteristics from the classical discrete optimization problems. For the discrete optimization problems usually we assume that the shape of the tree (i.e., the searching space) is arbitrary and even infinite. The structure of most XML documents, however, is much more regular and even can be described in the schema, such as XML-Schema. Particularly, a large XML file usually contains one or more large arrays, presenting a flat structure. For example, the XML documents used in the scientific applications usually only contain a single large array of numbers. Additionally, instead of simply searching for the goal XML tasks usually need a more complicated result handling procedure. In short, in order to achieve the efficient parallelization we need to introduce special tunning mechanisms based on the characteristics of XML documents.

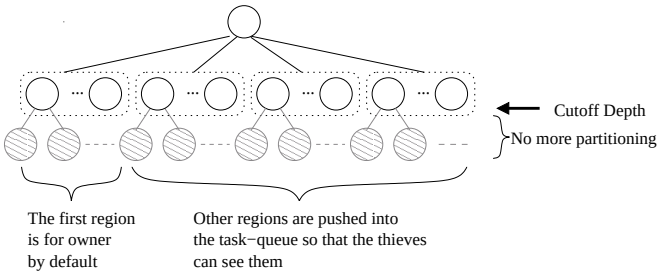

Figure 4. Region based task partitioning and depth based partitioning control

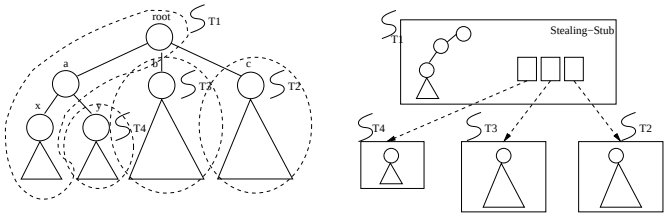

Figure 5. The left part shows a stealing scenario and the implicit partitionning result; The right part shows the corresponding stealing-stub tree.

4.1 Region-based Task Partitioning

Like those parallel formulas for the discrete optimization problem, we can treat the individual element node in the XML as an task, which will either be processed by the owner or be stolen by the thief. This approach, however, incurs the significant performance overhead, thus being impractical, particularly for the large XML document containing large arrays. First of all, the owner needs to explicitly push/pop every element node into its local task-queue, and the frequent stack operations can easily dominate the entire performance. Secondly, the stealing will become extremely inefficient. The thief will spend most the time on stealing a leaf node, finishing the execution quickly and back to stealing again. The worse thing is the stealing-sub tree could become even larger than the DOM tree.

Our solution is increasing the granularity of the task which will cover a region of sibling nodes rather than one single node. As shown in the fig.4 when an element node is expanded, all its children is first divided into continuous regions, each of which will be pushed into the local task-queue. To give the owner thread the priority, the first region is guaranteed to leave to the owner thread without yielding any chance to the thieves. The number of the regions is same as the number of the threads in the crew so that each thread will have the chance to obtain one region; when the number of children is less than the number of the threads, a simple dividing-in-half policy is adopted.

4.2 Depth-based Partitioning Control

While the region-based partitioning is about how the task is partitioned, the technique introduced here is about when the task should be partitioned. As we know, the sole purpose of the task partitioning is for the work sharing. The partitioning itself, however, may incur nontrivial performance overhead. Most DOM implementations implement the children list of a node as a single linked list, thus partitioning entails at least an extra scanning of all children of current node. `XMLDocument` of .NET library even aggravates the problem as it doesn't cache the children number[7], that means to get this number before dividing the children evenly one more scanning is needed. Also if the newly partitioned task is small in term of the size, the stealing will be inefficient.

Due to the flat structure of large XML documents, it is reasonable to estimate the size of the subtree by its depth, namely the deeper the subtree located at, the smaller the subtree might be. Hence to avoid generating tiny tasks, we define a cutoff depth, under which all the subtrees are considered as small sized ones and no more task partitioning will be applied on. The cutoff depth is similar to the tree pruning technology widely used in the discrete optimization solutions. However here the subtrees under the cutoff depth is not really pruned from the tree; instead, they are just invisible from the thieves. To support that, besides the local task queue each thread maintains an internal stack which is only visible to itself. Whenever the thread reaches the cutoff depth during the traversing, it switches its working stack from the local queue to its internal stack and when backtracking to the cutoff depth it switches back to the local queue. When working on the internal stack the thread works exactly same as what a serial implementation does, incurring neither synchronization operations nor stealing actions.

The cutoff depth essentially controls the trade-off between the potential parallelism and the partitioning overhead. It can be set up heuristically based on the schema of the XML document. For example it could be the depth at where the large array appears in the document.

4.3 Result Gluing

The result of the execution of `ThreadCrew` is a stealing-stub tree. The type of partial result in a stealing-stub is defined by the application. For example, the partial result in the parallel XML serializing represents a string fragment, while the partial result for the partial XML parsing is a DOM fragment.

The stealing-stub tree essentially represents the dynamic partitioning layout over the original XML document. As

illustrated in the Fig. 5, each stub in the tree corresponds one partition on the document and its partial result is the processing result over that partition. Recall that the thief always steals from the bottom of the local-queue, it is not hard to observe the below three observations.

Observation 4.1 *The partition represented by each stealing-stub covers the set of XML element nodes which are contiguous in the XML document in term of the document order.*

That is because the partition is formed as the result of traversing of the XML DOM tree in the document order and only the nodes at bottom of the stack could be stolen, thus each nodes in the partition must be contiguous in term of the document order.

Observation 4.2 *For a parent stealing-stub, its corresponding partition precedes all the partitions covered by its child stealing-stubs in the XML document in term of document order.*

That is because any child stealing-stub represents a stolen element node or region, which was originally at the bottom of the owner thread's traversal stack, while the parent stealing-stub represents those element nodes which were pushed into the stack after the stolen node was pushed into.

Observation 4.3 *For any stealing stub, its corresponding partition precedes all the partitions covered by its right siblings in the stealing-stub tree in term of document order.*

Recall that we reserve the order of stealing-stubs when copying them from the stack into the stealing-stub list of the root stealing stub, so the order between the two siblings is same as the order when they are pushed into the stack.

Based the above three observations, we see that when we traverse the stealing-stub tree in the depth-first and left-to-right order we are actually traversing the each partition over the XML document in the exact document order. Thereby to reduce the partial results together to form the final result, we just need to traverse the stealing-stub tree in the depth-first and left-to-right order, and glue all the particle results one by one. The application should have its own method to glue two consecutive partial results together during the traversing.

5 Applications and Experiments

To show the feasibility and effectiveness of the ParaXML processing model, in this section we present our parallel implementations of three fundamental XML processing tasks: traversal, serializing and parsing as well as their performance experiments. We performed the experiments of all the three parallel implementations on a 4-core machine, which has 2GB memory and two Intel Xeon 5150 processors, each having two cores inside. The operating system is Windows XP and the version of the .NET framework is 2.0.

We uses two XML benchmarks, XML benchmark and the XMark, to generate the test files. The XML Benchmark [4], designed for XML parsers comparison generates an XML document with a simple schema, which basically is an array of the structure. The size of the generated documents is set to be 50M Bytes. Conversely, the XML document generated by XMark [23], which examines the performance of queries on the XML repositories, is much more complicated. The file has more than 5 levels of depth and from level 1 to level 4 each contains large arrays. The size of the generated documents is set to be 59M Bytes.

5.1 Parallel Traversal

The parallel traversal is the foundation of the other parallel XML tasks. According to the Amdahl's law, the scalability of a parallel algorithm is determined by the ratio of the serial part of the algorithm. For the parallel traversing, each visiting of a node undertakes the minimal workload, so the scalability of the parallel traversal algorithm demonstrates the guide line for other parallel XML tasks that built upon it. In other word, for one parallel XML task, if it involves more workload in each parallel visiting and those workload are independent then the scalability of the parallel traversal algorithm actually is the low bound of the scalability that parallel XML task can have; however if those workload will causes the contention on resources its scalability can be worse than the parallel traversal algorithm.

Here, we define the parallel traversing task to find all the elements in the document, whose name matches a given simple query criteria. Our intention is to prove the concept and a fully implementation of the parallel XPath query is beyond the scope of this paper. The partial result passed to the task handle is a list, containing the matched element, The task handle scans every element in the region task. If the element matches with the query criteria, it will be put into the partial result (i.e., the list). When the processing on a partition completed, the list contains all the matched elements in the partition; thereby the final result can be gotten by concatenating every list in the stealing-stub tree.

The first experiment is performance on the XML document generated by XMark and is to find the elements which meet the query `//person[@id="person0"]`. We varied the cutoff depth from 1 to 5 to determine its impact on the performance. As shown in Fig.6, we see the cutoff depth substantially affects the performance of the algorithm. When it equals 2 or 3 we get the best performance. Our second test is performed on the XML document generated by XMLBench with the equivalent XPath query: `//shiporder[@orderid="011284"]`. Fig.6 shows

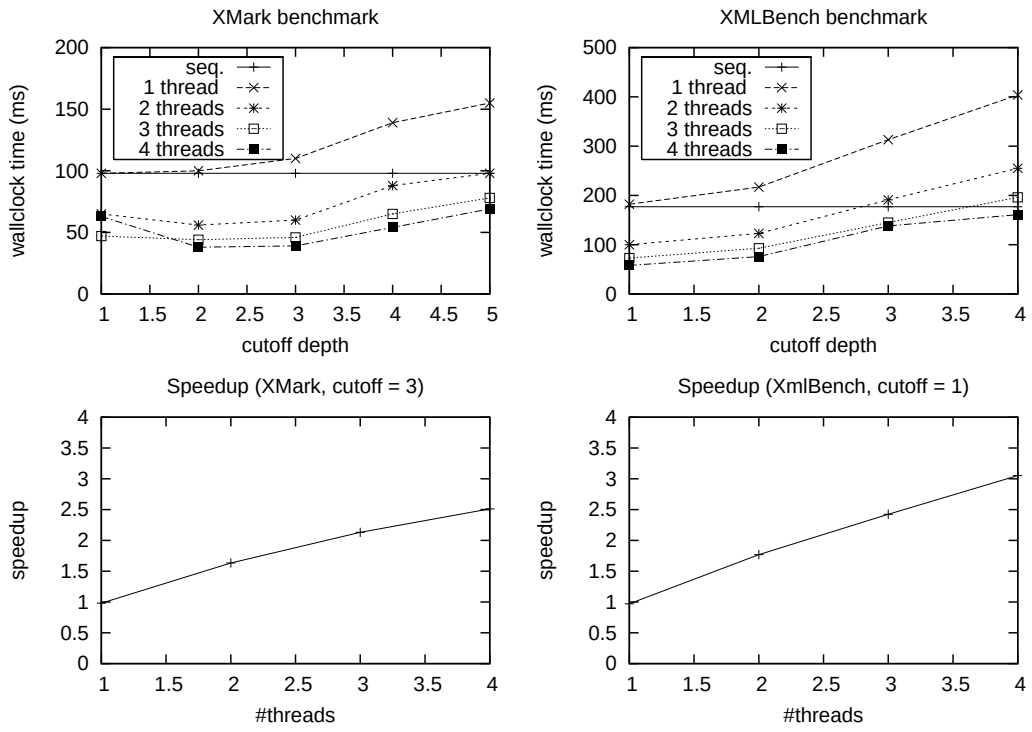

Figure 6. The performance of the parallel traversing

that the best cutoff depth for this document is 1. This is expected since XMLBench file essentially is just one large array under the root element. It should be noted that when the cutoff depth is 1 the dynamical partition formed by the stealing is equivalent as the static linear partition on the large array. We also depict the speedup of the parallel traversal for the two queries in Fig.6, in which the XMLBench document presents a better speedup curve than the XMark documents That is because the structure of XMLBench document is much simpler than the one of XMark document so that the load balance is easier to achieve on the XMLBench document.

In the following experiments in this paper, we will use the optimal value of the cutoff depth for the two documents respectively, namely 1 for the XMLBench documents and 3 for the XMark documents.

5.2 Parallel XML Serialization

Given a DOM tree, we can serialize it into the XML document by simply traversing the DOM tree in the document order and outputting the node content along the path. Therefore the parallel serializing of a DOM tree can be directly extended from the parallel traversal algorithm presented previously with the extra content output operations. Here the partial result in the stealing-stub is a buffer of characters, representing the serializing result on the partition.

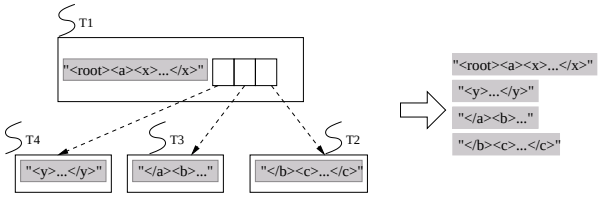

Figure 7. Gluing the serializing results

A special care, however,should be taken for the close tag of a non-empty element. When a thread is serializing the content of an element, it has no clue if it will process all the content or only part of the content in case of stealing. Therefore always having the thread output the close tag of an element may leave the part the serialization out of its lexical scope in the final serialization. Our solution is always assigning the outputting the close tag of the last element of the current region to the next task. Whenever a thread begins executing a task, it first check if there is a close tag needed to be outputted, if so it will first output the close tag into its serialization buffer. In this manner if the next task is stolen the thief will output the close tag, otherwise the owner will output the close tag by itself but at the beginning of the processing of the next task.

As illustrated in the Fig.7, the gluing procedure simply

depth-first traverses the stealing-stub tree and concatenates the consecutive partial results (i.e., the string buffer) together. It should be noticed that the constructing the final serialization is unnecessary for most applications since it can be deferred to the point when the serialization is physically needed.

During the parallel serializing multiple threads may allocate the large string buffers simultaneously. However the allocation by .NET garbage collector is not very scalable [1] particularly for the large objects whose allocation will be done serially [22]. Furthermore, when the memory usage exceeds the budget the GC will suspend all threads to collect the garbage memory. In order to minimize the impact of the GC, we preallocate the string buffer for each thread before the parallel execution, and have each thread write the content into its specific buffer without any interference.

Fig. 8 demonstrates the performance of the parallel serializing algorithm on the two benchmark documents with the optimal cutoff depth value. As expected, the parallel serializing algorithm present the better scalability than parallel traversal algorithm did on both benchmark documents. For example in the 4-core test, the speedup of parallel serializing is 2.8x for the XMark document and 3.2x for the XMLBench document. Clearly, it is mainly due to the extra independent serializing work done by the threads in parallel.

5.2.1 Parallel C14N

An XML document can have multiple various but valid serializations. However this valid variation is problematic for those octet-stream based applications (e.g., signature calculation). To address this issue, XML canonicalization, abbreviated as C14N, defines the canonical form for an XML document, which is guaranteed to be identical if and only if the content of the document is identical. For example, the canonical form requires the superfluous namespace declarations should be removed from each element and all the attributes of one element should be lexicographically ordered by their qualified name. Certainly, those requirements will impose extra nontrivial computation and some research work [24] have identified the canonicalization processing is the performance bottleneck of other higher level XML tasks, such as XML signature which calculates the digest over the canonical form.

In order to parallelize the canonicalization, we simply augment the parallel XML serialization algorithm with the C14N rules enforcement. From Fig. 8 we can see the canonicalization takes more time than the normal serializing processing. More importantly, the extra work by the canonicalization makes the parallel canonicalization presents even better scalability than the normal parallel serializing algorithm.

[1] in .NET only the generation 0 of the Garbage Collector allows the concurrent allocation, but its size usually is very small

5.3 Parallel XML Parsing and DOM Building

The basic idea of the parallel XML parsing is having multiple XML pull parsers parse the disjoint fragments of a XML document and build the subtrees of the final DOM in parallel. The pull parser is chosen because of its ability of the fragment parsing, which means we can designate any the valid document fragment for parsing. The parallel XML parsing, however, has to face a special challenge: from a sequence of characters we can't tell the tree structure directly, thus unable to apply the ParaXML model as previous algorithms did.

5.3.1 Preparsing and Skeleton

Our solution is two-stage processing of the XML document[14], namely a quick sequential scan of the document to identify the structure, followed by a complete parallel parsing. We call the first scan stage the *preparse*, whose sole purpose is to determine the topological structure of the elements in the document. That means most syntactic units defined by the XML specification, such as attributes, namespaces, and even the tag name, can be ignored by the preparse. Neither does the preparse need to verify any well-formedness constraints. In other word the preparsing treats the XML document as simply a sequence of unnamed start- and end-tag pairs. As a result, the generated tree structure, called *skeleton*, is much more light-weight than the DOM. Representing one element in the document each node in the skeleton only contains the lexical scope information of the element in the document. We can view the skeleton as the index of all well-formed fragments in the XML document.

We implement a C# version preparser and compare its performance with the pull parsing (by XmlTestReader class of .NET) and DOM building on the two benchmark documents. The pull parsing is designed to have the minimal workload, namely only scanning the document without any building. From the Fig. 10 we can see the preparsing is about 3x faster than the simple pull parsing and 5x faster than the DOM building.

5.3.2 Stealing based Parallel XML Parsing

With the skeleton, now we are able to apply the ParaXML model to parallelize the XML parsing and DOM building. The pull parser, `XmlTestReader` of .NET, can pull-parse any fragment in the XML document as long as the fragment contains any valid element content. Since now the parallel XML parsing algorithm works on the skeleton, the task here refers a region of skeleton sibling nodes in the

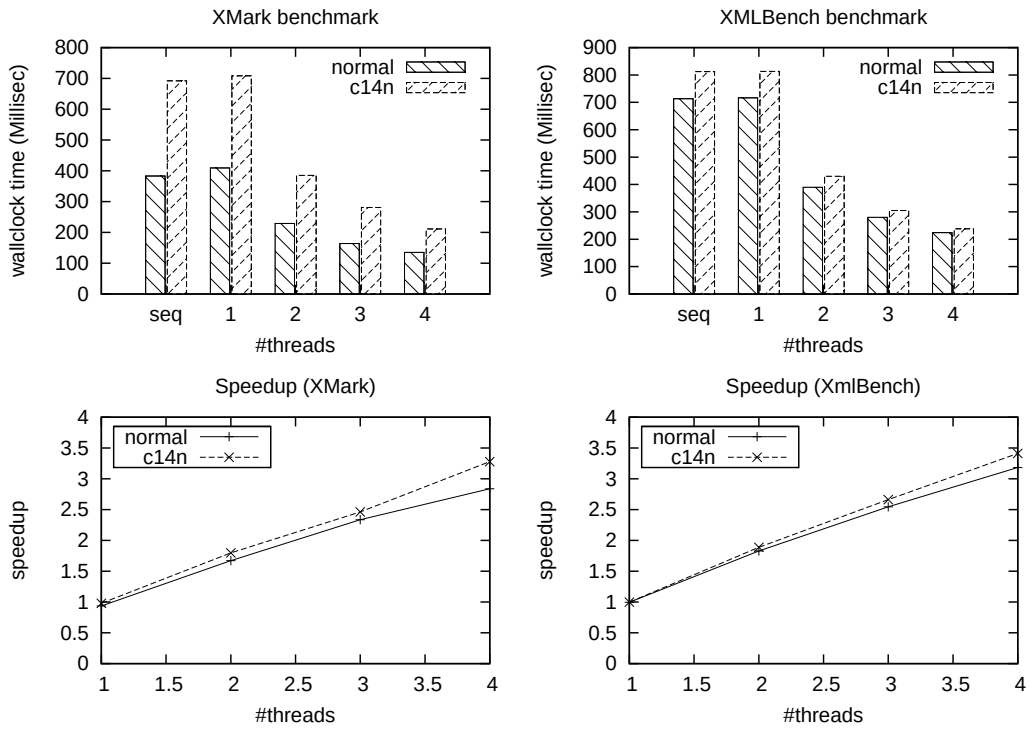

Figure 8. The performance of the parallel serializing & canonicalization algorithms.

skeleton. Given a parsing task, the thread can figure out the lexical offset and size of the corresponding fragment in the document, and then creates a XmlTestReader object for pull-parsing on this fragment.

The thread keeps pull-parsing the fragment and building the DOM fragment as a normal serial parser does. However once the thread pulls out an open tag from the stream it walks the skeleton one step in the depth-first,left-to-right order. The purpose of this walking is to keep track of the current parsing progress of the thread. The walking is realized by popping up the local task-queue and pushing the generated child-tasks bask as the previous algorithms did. As a result, the local queue of the thread actually functions as a "manifest" which reflects all the sub-fragments the thread will parse in the future or the thieves can steal right now. The procedure keeps running until the end of the fragment has been reached or the local task queue is empty. For the latter case, that means one or more sub-fragments in the current fragment has been stolen and processed by other threads.

When the multiple threads in the ThreadCrew pull-parse the disjoint fragments in parallel, the partial result generated by one thread is the forest of the DOM fragments, each corresponding to a skeleton node in the original region task. As shown in the Fig.9, to glue the fragments together to form the final DOM tree we can traverse the stealing-stubs tree

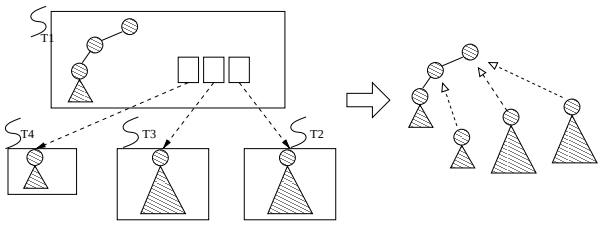

Figure 9. Gluing the DOM fragments generated by the parallel parsing

in the document order and appending the DOM fragments (i.e., the partial result in one stealing-stub) under their parent nodes in the DOM sequentially.

It should be noted that we are using our own simple DOM implementation instead of the XMLDocument class in the .NET library. It is because XMLDocument uses *atomized string*[7], to eliminate the duplicated strings in the DOM. While this optimization saves the memory space, it in deed impedes the efficient parallelism since whenever an element is generated we need to mutual exclusively look up a global name table.

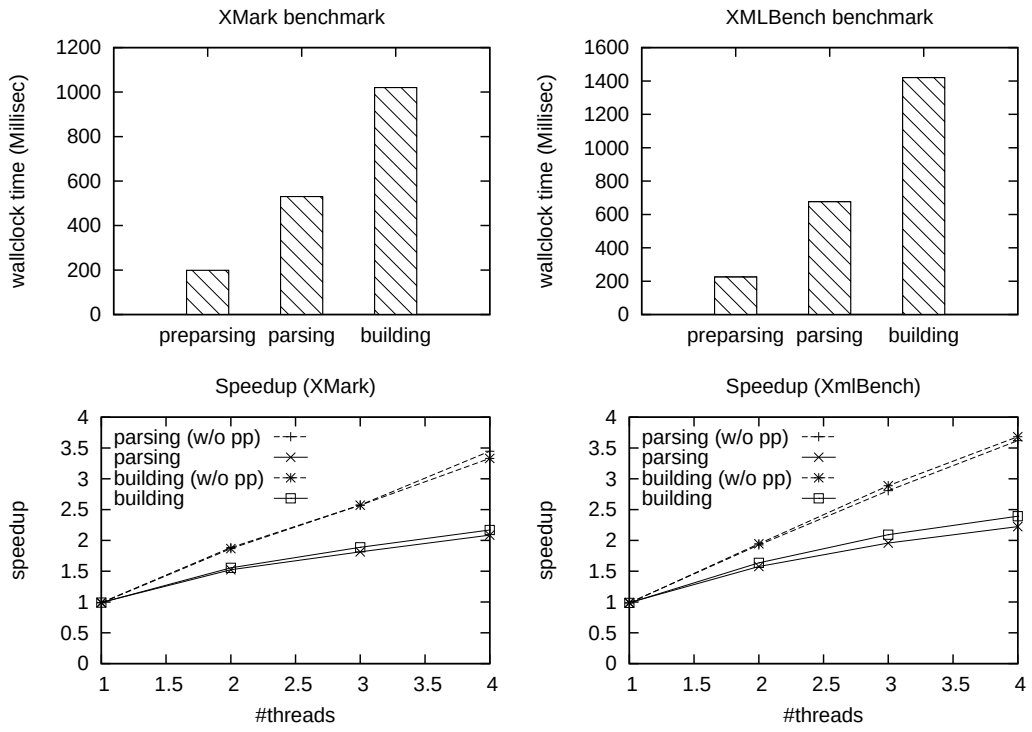

Figure 10. The performance of the parallel parsing & DOM-building algorithms

5.3.3 Measurement

First of all, we configure the parallel XML parsing algorithm to only parse the document without building any DOM structure. In other word, we are measuring the performance of the parallel pull-parsing only. Fig. 10 illustrates the speedup of the parallel pull parsing for the two benchmark documents. The speedup in the 4-core test is roughly 2.1x for the XMark document and 2.25x for the XMLBench document respectively. In order to identify the scalability bottleneck, we exclude the preparsing time from the total time and calculate the speedup again. Now the speedup is significantly improved: 3.4x for the XMark document and 3.7x for the XMLBench document on the 4-core test and both are better than the parallel traversing. It clearly indicates that the scalability bottleneck of the parallel algorithm is at the serial preparsing, the parallel pull-parsing mechanism itself scales well.

Based on the experiment result of parallel parsing, it is reasonable to expect that the parallel DOM building, involving more memory allocation work, would obtain the same or better scalability. However initially our experiment result shew the parallel DOM building doesn't achieve the expected performance, in fact it presents much worse scalability then the parallel parsing. That implies the scalability bottleneck has been shifted. By running the Intel VTune performance analyzer, we observed that during the parallel execution a number of garbage collections were triggered when the large number of DOM nodes were created in memory, thus leading the degradation of the scalability.

To circumvent the GC, again we adopt the thread-specific object pool technology as we did in the parallel serialization experiment. When building the DOM tree, the thread obtain the new node object from its specific pool instead of from the GC. In this manner each thread doesn't interferer with each other at all. Fig. 10 illustrates the performance of the parallel DOM building algorithm with the thread-specific object pool. The result now shows that the speedup of the parallel DOM building is about same as the parallel parsing.

It should be noticed that the preparsing doesn't have to be the serial bottleneck. Pan and Chiu [18] proposed a parallel solution for the preparsing by using a speculative meta-DFA. Alternatively, the skeleton, once generated, can be kept separately as a index object so that the later parallel processing on the same XML document can reuse the skeleton directly without incurring any serial bottleneck.

6 Related Work

Our previous work [14] proposed the two-pass-scanning based parallel XML parsing prototype, called PXP, which is implemented in C++ and relies on the libxml2 for the real

parsing. PXP uses a simple request-response ("begging") scheme rather than the stealing scheme for the load balancing, so in general PXP will incur more overhead and unbalanced distribution than ParaXML; also PXP had a hard time for the result reducing. Based on PXP, Pan et al. [19] introduces a static XML partitioning scheme for the parallel XML parsing. Although the static scheme works well for the XML document with a simple array structure, it is hard to be a general solution for all kinds of XML documents. Also when the document structure becomes more complicated the static partitioning stage will become the serial bottleneck of the entire processing. In fact, for the simple document with the array structure ParaXML model behaves exactly as same as the static partition scheme does, but in the dynamic manner. To solve the serial preparsing bottleneck of the parallel XML parsing, Pan and Chiu. [19] proposed a parallel solution for the preparsing. The XML document is partitioned into multiple equal-sized chunks, each is preparsed by one thread simultaneously. As a chunk may not contain the valid XML content the preparser has to assume all possible contexts and speculatively scan the chunk by using a meta automaton. The parallelized preparsing greatly improved the scalability of the parallel XML parsing. K.Lu et al.[13] proposed a parallel model for the XML query from the database perspective. Instead of focusing on the XML document in the share memory environment, their work studies the data storage strategies and data placement methods in a XML database system for the potential parallel XML data query.

The stealing based load balancing scheme [3] is becoming more popular in the multicore era due to its dynamic and low-overhead characteristics. It has been widely adopted in many areas, such as the garbage collection [8] and the user-level parallel library(e.g., Intel Thread Building Block [12]). ParaXML inherits the lock-free stealing algorithm from ABP-deque [2] with the extension for the result tracing, which is needed by most XML tasks. As mentioned earlier, parallel XML processing can essentially be viewed as the parallel graph search algorithms for the discrete optimization problems [20]. But parallel XML processing doesn't have the searching space as large as the one of the discrete optimization problems, thus it is more sensitive to any overhead and needs finer optimization control. Also parallel XML processing introduces some new issues, such as result reducing, which are not addressed by the general parallel algorithms.

Skeleton concept was firstly proposed in the XML lazy parsing [17]. However, the purpose of lazy parsing and parallel parsing are totally different, so the structure and the use of the skeleton in the both algorithms differs fundamentally.

7 Conclusion

In this paper we present a parallel XML processing model, ParaXML, which is designed for the multicore system. Essentially ParaXml treats the XML processing as the parallel tree traversal problem. By the stealing scheme and the lock-free deque structure, ParaXML succeed in solving the key issue of the parallel program: load balancing with low overhead. Also the stealing tracing mechanisms in ParaXML eases the parallel results reducing. Considering the XML documents usually have a flat and regular structure, ParaXML introduces two optimization technologies, region and depth-based partitioning control, and the experiments shew that they are crucial to achieving the high performance. We also present the parallel implementations of XML traversing, serializing and parsing based on the ParaXML. Our experiments results show those parallel implementations substantially improved the performance and scale well on a multicore machine. However not all software components are ready for the multicore era, for example the garbage collector and the default DOM library in .NET runtime restricts the scalability of our parallel algorithms.

8 Acknowledgment

Our C# implementation of the stealing based ABP Deque algorithm used in this paper is developed based on the pseudo code provided in the book, The Art of Multiprocessor Programming[11], by Maurice Herlihy and Nir Shavit. We wish to thanks Nir Shavit for his important comments on the pseudo code.

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 483–485, 1967.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA '98: Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 119–129, New York, NY, USA, 1998. ACM Press.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 1995. ACM Press.

[4] S. A. Chilingaryan. Xml benchmark project. `http://xmlbench.sourceforge.net/`, 2004.

[5] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of soap performance for scientific computing. In *HPDC '02: Proceedings of the 11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, page 246. IEEE Computer Society, 2002.

[6] K. Chiu and W. Lu. A compiler-based approach to schema-specific xml parsing. In *The First International Workshop on High Performance XML Processing, Satellite workshop of WWW2004 International Conference*, 2004.

[7] D. Esposito. *Applied XML Programming For Microsfot .NET*. Microsoft, 2003.

[8] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, 2001.

[9] A. Y. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11, 1999.

[10] A. Y. Grama and V. Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering*, 11, 1999.

[11] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[12] Intel. Tutorial of intel thread building block, 2006.

[13] K. Lu, Y. Zhu, W. Sun, S. Lin, and J. Fan. Parallel processing xml documents. *Database Engineering and Applications Symposium*, 2002.

[14] W. Lu, K. Chiu, and Y. Pan. A parallel approach to xml parsing. In *The 7th IEEE/ACM International Conference on Grid Computing*, Barcelona, September 2006.

[15] M. Matsa, E. Perkins, A. Heifets, M. G. Kostoulas, D. Silva, N. Mendelsohn, and M. Leger. A high-performance interpretive approach to schema-directed parsing. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, 2007.

[16] M. Nicola and J. John. Xml parsing: a threat to odatabase performance. In *CIKM '03: Proceedings of the twelfth international conference on Information and knowledge management*, 2003.

[17] M. L. Noga, S. Schott, and W. Lowe. Lazy xml processing. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, 2002.

[18] Y. Pan, K. Chiu, Y. Zhang, and W. Lu. Parallel xml parsing using meta-dfas. In *3rd IEEE International Conference on e-Science and Grid Computing*, Bangalore, India, 2007.

[19] Y. Pan, W. Lu, Y. Zhang, and K. Chiu. A static load-balancing scheme for parallel xml parsing on multi-core cpus. In *IEEE International Symposium on Cluster Computing and the Grid*, Rio de Janeiro, 2007.

[20] V. N. Rao and V. Kumar. Parallel depth first search. part i. implementation. *Int. J. Parallel Program.*, 16(6):479–499, 1987.

[21] A. Reinefeld. Scalability of massively parallel depth-first search. In *Parallel Processing of Discrete Optimization Problems*, volume 22 of *DIMACS Series in Discrete Mathem. and Theor. Comp*, pages 305–322, 1995.

[22] J. Richter. *CLR via C#*. Microsoft, 2006.

[23] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001.

[24] S. Shirasuna, A. Slominski, L. Fang, and D. Gannon. Performance comparison of security mechanisms for grid services. In *5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[25] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30, 2005.

[26] J. van Lunteren, J. Bostian, B. Carey, T. Engbersen, and C. Larsson. Xml accelerator engine. In *The First International Workshop on High Performance XML Processing*, 2004.

[27] Y.Diao, P.Fischer, and M.J.Franklin. Yfilter: Efficient and scalable of xml document. In *The 18th International Conference of Data Engineering*, San Jose, 2002.