

# Parallel XML Processing by Work Stealing

Wei Lu  
Indiana University  
150 S. Woodlawn Ave.  
Bloomington, IN 47405  
welu@cs.indiana.edu

Dennis Gannon  
Indiana University  
150 S. Woodlawn Ave.  
Bloomington, IN 47405  
gannon@cs.indiana.edu

## ABSTRACT

A language for semi-structured documents, XML is playing crucial roles in web services, messaging systems, databases, and document processing. However, the processing of XML documents has been regarded as the performance bottleneck in most systems and applications. On the other side, the multicore processor, emerged as a solution for the clock-speed limitation of the modern CPUs, has been growingly prevalent. Leveraging the parallelism provided by the multicore resource to speedup the software execution is becoming the trend of the software development. In this paper, we present a parallel processing model for the XML document. The model is not designed just for a specific XML processing task, instead, it is a general model, by which we are able to explore various parallel XML document processing. The kernel of the model is a stealing-based dynamic load-balancing mechanism, by which multiple threads are able to process the disjointed parts of the XML document in parallel with balanced load distribution. The model also provides a novel mechanism to trace the stealing actions, thus the equivalent sequential result can be gotten by gluing the multiple parallel-running results together. To show the feasibility and effectiveness of our approaches, we present our C# implementation of parallel XML serialization in this paper. Our empirical study shows our parallel XML serialization algorithm can improved the XML serializing performance significantly on a multicore machine.

## Categories and Subject Descriptors

D.1.3 [Concurrent Programming]: Parallel Programming

## General Terms

Performance

## Keywords

XML, Parallel Programming

## 1. INTRODUCTION

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 overhead [4]. These overhead can prevent the acceptance of XML in use cases that may otherwise benefit.

On the hardware front, manufacturers are increasingly utilizing the march of Moore's law to provide multiple cores on a single chip, rather than faster clock speeds. Tomorrow's computers will have more cores rather than exponentially faster clock speeds, and software will increasingly need to rely on parallelism to take advantage of this trend [15].

A XML document essentially represents a tree-structured data model, whose formal name is XML Infoset [18]. The XML document thus can be regarded as the the serialization of this tree model in a depth-first, left-to-right traversing order (i.e., the document order), and the Document Object Model (DOM)[17] is the widely used data structure representing this tree model. Using the DOM, a number of XML document processing have been defined, such as the XML canonicalization, XML signature and XPath query, and they are forming the fundamental building blocks of the service oriented computing.

A number of techniques have been developed to improve the performance of XML processing, ranging from the schema-specific model [5, 10, 16] to the streaming-based model [19, 11]. to the hardware acceleration. Generally speaking, these approaches are designed for the specific XML processing tasks (e.g., XPath query or XML signature verification) and usually assume that the DOM is unavailable. However in the most popular libraries, such as the .NET XML library, the DOM object must be created before other processings. Thus we believe that the parallel XML processing on the DOMs by leveraging the multicore resources will be a more general solution for the performance issue.

In this paper, we investigate a parallel XML processing model on a multicore computer, The model is not designed just for a specific XML processing task, instead, it is a general-purpose model, by which we are able to explore various parallel XML document processing. The model is implemented in C#, The kernel of the model is a stealing-based dynamic load-balancing mechanism, by which multiple threads are able to process the disjointed parts of the XML document in parallel with balanced load distribution. The model also provides a novel mechanism to trace the stealing actions, thus the equivalent sequential result can be gotten by gluing the multiple parallel-running results together. To show the feasibility and effectiveness of our approaches, we present our parallel implementation of the XML DOM serialization.

Operating systems usually provide access to multiple cores via kernel threads (or LWPs). In this paper, we generally assume that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOCP'07, June 26, 2007, Monterey, California, USA.

Copyright 2007 ACM 978-1-59593-717-9/07/0006 ...\$5.00.

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 this model, the stealing-based load balancing mechanism, in detail. We present in Section 4 the algorithm of parallel XML serialization. Its performance result is discussed in Section 5.

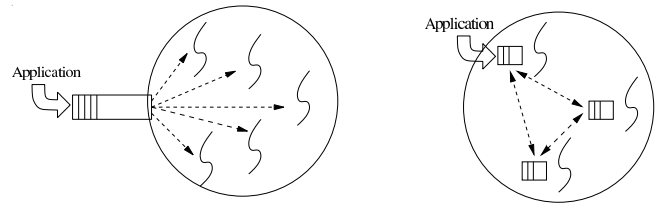
## 2. LOAD BALANCING TECHNIQUES

In term of the parallelization, the tree structure is 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. In some cases, such as the parallel Depth-First-Search of the tree[8], the subtrees rooted at any node can be procced relatively independently with the minimal communication with others. 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 predicted until it is walked, we can't determine the real workload associated with the subtree assigned to the thread. Thus it is very likely that two threads are assigned with two subtrees, which have different processing complexity, and when one thread finishes processing and becomes idle another thread is still busy for working. By the Amdahl's law the imbalanced workload distribution will prevent the parallel algorithm from being scalable and efficient.

There are numbers of approaches to address the load balancing problems. The static load balancing approaches [12, 7], 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 cutoff depth, which usually is a priori knowledge, as well as the shape of the tree structure. Hence a static approach is not a general solution.

Another set of approaches is the dynamic load balancing one, in which the task partitioning and distribution takes place at running time with the help of the dynamic workload information. Stealing based scheme[3] is the one of the approaches that have been widely used in the applications with shared-memory environment[6, 3]. The basic idea of the stealing based scheme is that 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 really on demand and dynamic, thus regardless of the shape of the tree the overall workload tends to be balanced. When all the threads are working busily no any extra cost will need to pay, and when stealing happens other threads except the victim have little impact. The dynamic workload redistribution, however, incurs more sophisticated interaction among the threads. Great care should be taken to the design and the implementation, otherwise the cost of synchronization and communication could easily kill the performance gain by the parallelism.

The ThreadPool class of the .NET framework [13] consists of a global task queue and a number pre-created threads, each of which gets/puts the task from/into the global queue. The number of the threads in the pool is adaptive to the workload, and it could be larger than the number of the available PEs(i.e., Processing Elements). Although ThreadPool is a powerful tool for the general concurrent programming, it is unsuitable for the parallization of those tree based algorithms for the following reasons. First, the global task queue surely is the performance bottleneck, the lock



**Figure 1:** Figure (a) above, illustrates the structure of the ThreadPool. Figure (b) illustrates the structure of the ThreadCrew.

contention by all the thread can dominate the entire performance. Furthermore, every new task generated by the thread will be put back to the global queue and will likely be fetched by other threads, thus the performance will suffer from the poor data locality and the low cache reusing.

## 3. THREAD CREW

To enable the parallel solution to the tree based problems , we designed and the implemented a stealing based load balancing tool, the ThreadCrew class, in C#. Just as ThreadPool, a ThreadCrew also represents a set of threads. However the number of threads in a ThreadCrew is fixed and should always be less than the number of the PEs, and all the threads are supposed to be running on the respective PEs simultaneously. Furthermore, instead of maintaining a global queue as the ThreadPool does, each thread in the ThreadCrew has its own local task queue, which is helpful for minimizing the content and maximizing 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 ThreadPool usually aims to executing multiple unrelated tasks, whereas the ThreadCrew is designed to execute a single complex task which will dynamically generate a number of tightly related sub tasks during the execution. The difference of the two systems is shown in the Fig.1.

Each thread in the ThreadCrew has three running phases:

1. Waiting phase
2. Working phase
3. Stealing phase

, and each thread is running same program, whose pseudo code is listed 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 call-back function
            provided by applications;

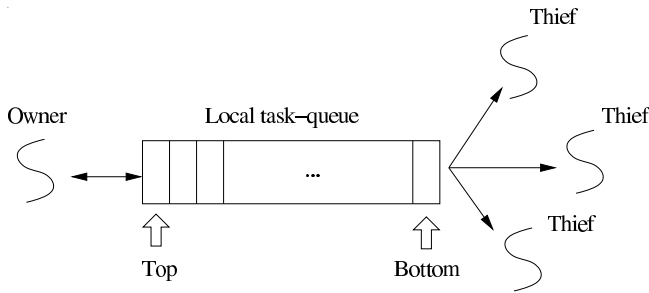
        }

    phase3: //stealing phase

        while (true) {

            if (termination is detected)
                break;

```



**Figure 2: The deque structure as the local task-queue in the ThreadCrew.**

```

pick a victim from the other threads;
try to steal task from the victim's task queue;
if (stealing succeed)
    goto phase2;
}

```

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 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 the working phase, each thread gets the task by popping its local task queue, and then executes the task. If there is a new task generated during the execution, the task will be pushed back into the local task queue. 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 effectiveness of the stealing scheme is based on the assumption that the owner accesses its local task queue much more frequently than thief does, and it is the thief, but not the owner, who should pay the cost of the stealing.

### 3.1 Lock-free Deque

In `ThreadCrew`, each local task queue is owned by its owner thread meanwhile it may be accessed by multiple thieves simultaneously. As a shared resource, the access to the task queue has to be mutually exclusive. The task queue could be designed as a normal stack, in which case both the owner and thieves will compete the top of the stack in order to get the tasks. That means a single lock should be applied to guarantee the mutual exclusion. By this scheme, however, the owner has to acquire the lock every time it pop/push its local task queue even when there is no any thief. The unnecessary contentions poses such a huge performance overhead on the owner that it is absolutely unacceptable for an effective parallel algorithm.

Our solution in `ThreadCrew` is using a deque data structure and having the owner and the thieves access different end of the deque separately. As shown in the Fig.2, the owner thread always accesses the deque as a normal stack and it pops/pushes the task only from the top of the deque, meanwhile the thieves steal the task from the deque by popping at the bottom of the deque. The sep-

aration leads to much less contention on the shared data, thus enabling the fine-grained mutual exclusion mechanism. The stealing scenario also has a nice feature. There is only one owner, though multiple thieves are allowed. By this feature, we can adopt a lock-free stealing-oriented deque structure, which is proposed by Arora, Blumofe and Plaxton [1] and is also known as "ABP-Deque".

ABP-Deque is just designed for the stealing scenario, it assumes that there is only one thread (i.e., the owner) accessing the top of the deque and multiple threads (i.e., the thieves) accessing the bottom of the deque. The ABP-Deque provides three major methods:

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

Internally ABP-Deque uses the Compare-And-Swap atomic operation (i.e., CAS), namely the `Interlocked.CompareExchange()` method in .NET, to solve the mutual exclusion when needed. When a thief tries to steal the task at the bottom of the queue, it always call the CAS operation to make sure the mutual exclusion with the owner and other thieves; whereas for the owner, as long as there are more than one tasks in the queue the owner is safe to access the top of the deque without any CAS operation. Only when there is the last item left in the queue, the owner needs the CAS operation to synchronize with thieves.

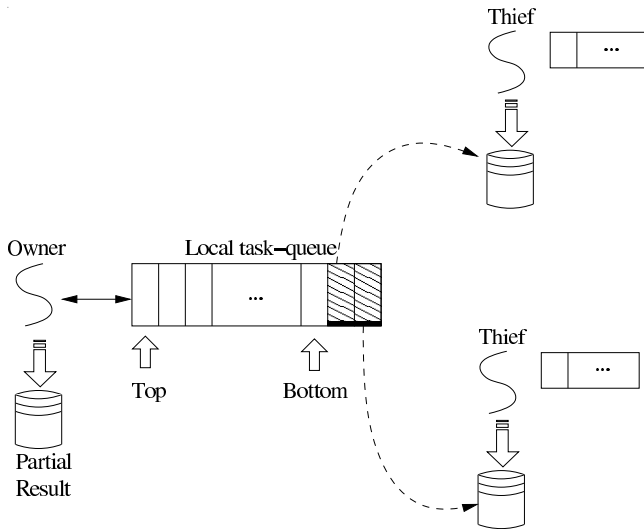
The ABP-Deque follows the thief-pays-the-cost principle very well. And as it is lock-free, those hazards (e.g., dead-lock and lock-out) caused by the lock can be avoided. More attractively, in most time the owner doesn't need to pay any extra performance to access the queue and the overhead is significantly decreased compared the lock based implementation. The thief will take the costly CAS for each stealing, nonetheless, the thief also benefits from stealing-from-bottom policy because the task at the bottom of a stack usually has more work load than the one at the top. For example during the traversal of a tree, the node at the bottom of the stack is the one 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 the issue that when a thread becomes a thief which thread in the crew is selected to steal. Following the work in the [9], `ThreadCrew` provides two common policies. The simplest one is picking the victim randomly. Another easy policy is the global round-robin, in which a global index indicating the current victim is maintained and is shifted to the next one in a round-robin style. Besides, `ThreadCrew` also provide the Pick-The-Richest policy, in which the thread with longest task queue will be selected as the victim. For the multicore system, when number of the cores is relative small we believe the Pick-The-Richest policy should lead to the better result as it incurs less failed stealing; otherwise with the increase of the number of the cores the querying and sorting operations on the task queues will be the bottleneck and the other policies will be more promising.

### 3.3 Termination Detection

The termination of the parallel running of the threads in the crew can be detected when all the threads have being in the stealing phase. To tell that we just need a global counter to indicate how many threads are in the stealing phase. When the system enters this state, that means there is neither any task in the task-queues



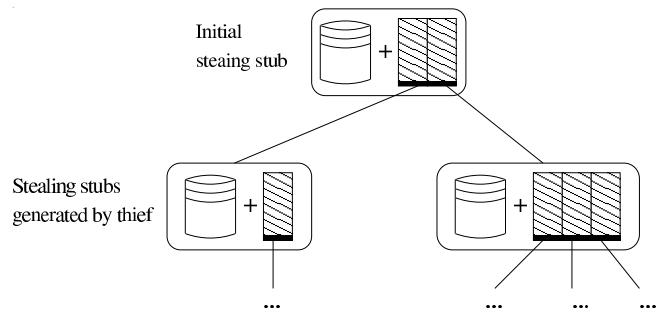
**Figure 3: The tracing of the partial results generated by the thieves.**

nor any new task to be generated. Thus it is safe to say the parallel running beginning from the initial task has completed. Once the termination is detected, all the threads move to the waiting phase and block on the barrier again and wait for the next assignment. The caller (i.e., the application) will be awakened.

### 3.4 Stealing Tracing & Result Gluing

When executing a task, which may be the initial task or one stolen from other threads, the thread may generate the result, such as the query result or traversal output. During the execution, new tasks may 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 any stealing happens during the execution, the generated result by the owner only corresponds 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 should consist of the partial result generated by the owner and those partial results generated by the thieves who stole the tasks during the execution. Note that once a thief begins to execute the stolen task, it may also generate new tasks, which may be stolen by other threads even including the original owner from who the task is stolen. Thus a partial result itself may be composed of multiple partial results in a recursive style.

In order to trace the partial results generated by the thieves, any thief is required to leave a clue, following which we are able to find the corresponding partial result. The clue is called *stealing stub* in ThreadCrew, and a stealing stub consists of a partial result object as well as a sequence of the stealing stubs, called tail. Before a thief is going to steal the victim, it will first generate an empty stealing stub, which initially contains a null partial result object and an empty tail. 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 stealing stub provided by the thief. After getting the task, the thief keeps holding the empty stealing stub as its root stealing stub during the execution of the stolen task. The partial result of the empty stealing stub will be passed to a *task handler* which is provided by the caller and is supposed to incrementally change the value of the partial result.



**Figure 4: The tree of the stealing stubs.**

Once the task queue of a thread becomes empty, all the stealing stubs left in the queue, which represent the clues to the stolen tasks, will be copied into the root stealing stub in order as its tail. As we will see later, the order plays a crucial role to glue the partial result together since this order is just the one in which the corresponding task was pushed into the queue.

As shown in the Fig.4, when the entire execution is terminated, all the stealing stubs form a tree structure rooted from the initial partial result. To glue the partial results together to form the final result, we just need to traverse the tree of the stealing stubs. Application should know how to merge tow consecutive partial results together during the traversing. For example the string concatenation is the one used by the parallel XML serialization.

## 4. PARALLEL XML SERIALIZATION

We pick the XML serialization as the first application of the parallel processing model not only because it is the fundamental traversal problem but also because it requires the most strict order on the generated result.

### 4.1 Basic Idea

Given a DOM tree, we can serialized it into the XML document by traversing the DOM tree in depth-first and left-to-right order, namely the document order. A simple implementation is the stack based traversing algorithm, which keeps popping the tree node from the stack, visiting it, and expending it by pushing all of its children nodes into the stack. The algorithm stops when the stack is empty. The pseudo code of the stack base algorithm is listed below. For simplicity, we just list the code of printing the tag pairs of all the elements in the XML document.

```

void visit(XmlElement root)
{
    push the root into the stack;
    while (stack is not empty) {
        XmlElement node = stack.pop();
        if (node.PreviousSibling != null)
            print the close tag of the previous sibling;
        print the open tag of the node;
        push all its children into the stack
        from right to left;
    }
}

```

With the ThreadCrew class, to parallelize the serialization algorithm we just need to change the above algorithm a little bit to be a

callback function `TaskHandler()` shown below. During the parallel execution, the threads in the `ThreadCrew` will keep calling the `TaskHandler()` function whenever it gets one task from its local queue or steal one from others. The task together with the partial result will be passed to the callback function.

```

void TaskHandler(Task task,
                PartialResult partialResult)
{
    //Note, in this context
    //the partial result is actually a string buffer.
    if (task.PreviousSibling != null)
        print the close tag of the previous sibling
        into the partialResult,

    print the open tag of task.element
    into the partialResult;

    push all children of task.element
    into the stack from right to left;
}

```

A special care should be taken for the outputting the close tag of a non-empty element. Assuming when a thread is serializing the content of a non-empty element, it has no clue if it will process all the content or only part of the content in case of stealing. Thereby always having the thread output the close tag of an element may leave the part the serialization of its content out of its final scope in the serialization. Our solution is assigning the outputting the close tag 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. Therefor if the next task is stolen the thief will be responsible for outputting the close tag, otherwise the owner will output the close tag by itself but at the beginning of the processing of the next task.

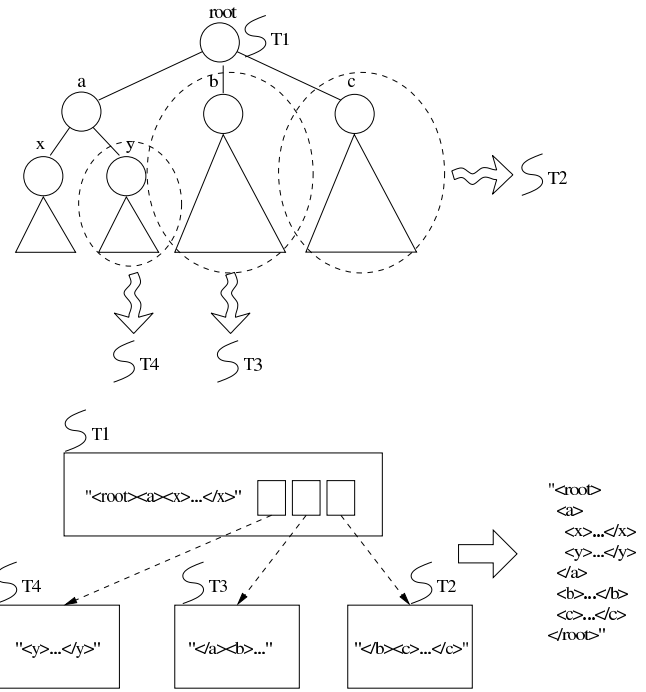
Although from the perspective of the user there is no obvious difference between the parallel algorithm and the sequential algorithm, the internal execution of the parallel algorithm is distinct. In the parallel execution the emptiness of the local task queue doesn't necessary mean that the entire tree 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 sequential 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 tasks from others as to achieve the load balance. Finally the tree nodes passed to the `TaskHandler()` 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 sequential algorithm.

All of the above detail has been hidden by the `ThreadCrew`, and the user are not aware of them. Basically the user only need to provide the `TaskHandler()` function and the method to glue the multiple partial results together.

## 4.2 Result Gluing

As we have already mentioned, the result of the parallel execution is a tree of the stealing stubs, each of which contains a partial result and a tail, namely a sequence of other stealing stubs. In the XML serialization case, the partial result in the stealing stub is a string buffer, representing the serializing result.

Since the thief always steals the node from the bottom of the stack, the serializing result generated by the thief has to be after the serializing result generated by the victim in the document order, as shown in Fig.5a. And because the order of the stealing stubs in the tail is as same as the one in which the corresponding tasks



**Figure 5: How to glue the partial serialization result together in the parallel XML serialization**

were pushed into the stack, for two consecutive stealing stubs in the tail the serializing result contained in the latter has to be after the one contained in the former in the document order. Hence, as illustrated in the Fig.5b, the gluing procedure can simply be Depth-first left-to-right traversing the stealing stub tree and concatenating the serializing results incrementally. The final result will be identical with the result of the sequential algorithm.

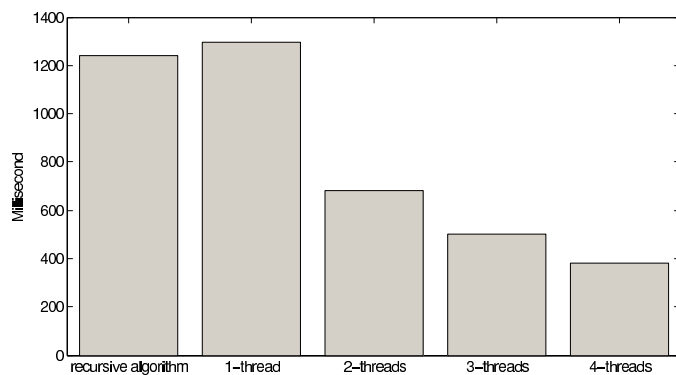
However the constructing the final serialization usually is unnecessary for most applications since it can be obtained by the traversal whenever needed. Also for some applications, such as hashing, in which an incremental processing is allowed, the gluing can definitely be eliminated.

## 4.3 Region-based Task Partitioning

A typical large XML file usually contains one or more large arrays data and its shape tends to be shallow and regular. An extreme example of the XML document, but not unusual in scientific applications, may only contain a single large array, whose items just are simple primitive type data.

In the previous description the algorithm treats the individual node as an independent task, which will either be processed by the owner or be stolen by the thief. It, however, incurs the nontrivial performance overhead, thus be impractical for the normal large XML document. First, the owner needs to explicitly push/pop every node in the array into its local task queue. When the array is large enough, those frequent stack operations will kill the performance. The second impact is the stealing will become very inefficient when processing the array structure. Most likely, the thief spends most the time on stealing a leaf node, finishing the execution quickly and back to stealing again. The worse thing is the tree of stealing-stubs will become very large in term of memory footprint.

Hence to make the parallel model really practical for XML, we need to increase the granularity of the task. One of the solutions is having the task cover a region of nodes in an array. The `TaskHandler()`



**Figure 6: The performance of the parallel XML Serialization algorithm**

function we have presents doesn't need to do much change, except that the `task` argument refers a region and when a node is expanded its children is divided into continuous regions, each of which will be pushed into the task queue. The number of the regions by one partition is same as the number of the threads in the crew so that every thread will have the chance to get one region; when the number of children is less than the number of the threads, the simple dividing-in-half policy is adopted. To give the owner the priority, the first region is always directly assigned to the owner thread without going thought the task queue. As a result the data locality and cache reusing of the owner thread will be improved.

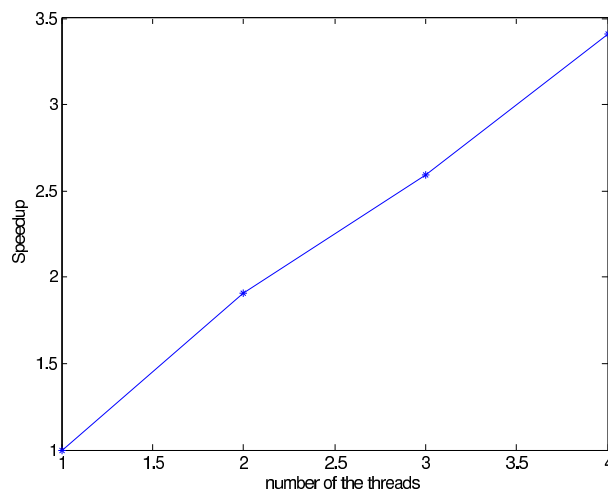
## 5. MEASUREMENT

To show the benefit of our stealing-based parallel XML serialization algorithm, we performed experiments on a multicore machine, which has 2GB memory and two Intel Xeon 5150 processors, each of which has two cores inside. The operating system is Windows XP and the version of the .NET framework is 2.0.

The test XML file contains the molecular information from the Protein Data Bank [14], and its size is about 25MB. The file is selected because we believe it represents the typical structural shape of the large XML documents, particular in scientific applications. It consists of two large array representing the molecule data as well as a couple elements for the molecule attributes; the items of the array are structure type rather than some trivial primitive data types. The schema of the XML file is listed in the Appendix.

The test program first parses and builds the DOM from the XML file sequentially, then serialize the DOM back into the XML format but in parallel. The test is run ten times, the measurement of the first time is discarded and the average value of the rest ones is calculated. The `ThreadCrew` class is configured with the `Pick-The-Richest` as the victim selection policy.

During our initial experiments, we noticed the result was not stable during the ten consecutive tests and we realized that is mainly due to the .NET Garbage collector which was involved more frequently in the parallel algorithm. The default garbage collector in the .NET is a workstation version, which collect the garbage sequentially. The garbage collecting is not suppressible and during the collection all the running thread will be suspended. To minimize the impact of the garbage collector, we configured .NET framework to use the server collector which collect the garbage in parallel on a multicore machine [13]. We implemented a recursive style serialization program as the sequential baseline, and then compared the `ThreadCrew` based parallel serialization program, configured with one or more threads, against the recursive style se-



**Figure 7: The speedup of the parallel XML Serialization algorithm**

rialization program. Figure 6 shows the experiment result, from which we saw the recursive style serialization program is a little bit faster than the parallel serialization program when configured with only one thread. It indicates the lock-free data structure incur ignorable performance overhead when there is not contention.

The parallel configuration speedups the execution the substantially. We calculate the speedup measurement which usually indicates how well a parallel algorithm scales with the number of the PEs. It is calculated by dividing the time of the sequential recursive style program by the time of the parallel serialization program. Figure 7 shows the speedup result of our parallel XML serialization program with one to four threads. We can see our parallel algorithm scales well. And we can predict that if our algorithm is implemented in C or C++, where a user-defined multi-thread memory management (such as `hoard`[2]) is possible, a better scalability can be achieved as the execution will not be limited by the garbage collection.

## 6. CONCLUSION AND FUTURE WORK

In this paper, we present a stealing-based parallel XML processing model. In this model the load balance among the threads is dynamically controlled by the stealing-based mechanism. And the stealing-based mechanism introduces only a little bit performance penalty, and this is mainly contributed by the stealing-from-bottom policy as well as the lock free ABP-deque. We introduce a novel method, stealing stub, to trace the stealing and to merge the parallel result into final sequential result if needed. Finally we show how the stealing-based mechanism, the stealing stub work and result gluing techniques be applied in the parallel XML serialization. Our experiments show that by the parallelism on the multicore machine the XML serialization can have the significant performance improvement. In the near future, we are going to investigate on how to apply our model on more XML-based tasks, such as XPath query and XSLT transformation.

## 7. REFERENCES

- [1] 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.

- [2] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A scalable memory allocator for multithreaded applications. Technical report, University of Texas at Austin, Austin, TX, USA, 2000.
- [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*, pages 207–216, New York, NY, USA, 1995. ACM Press.
- [4] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of soap performance for scientific computing. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 20002 (HPDC'02)*, page 246. IEEE Computer Society, 2002.
- [5] K. Chiu and W. Lu. A compiler-based approach to schema-specific xml parsing. In *The First International Workshop on High Performance XML Processing*, 2004.
- [6] 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.
- [7] M. Furuichi, K. Taki, and N. Ichiyoshi. A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 50–59, New York, NY, USA, 1990. ACM Press.
- [8] 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.
- [9] V. Kumar, A. Y. Grama, and N. R. Vempaty. Scalable load balancing techniques for parallel computers. *J. Parallel Distrib. Comput.*, 22(1):60–79, 1994.
- [10] W. M. Lowe, M. L. Noga, and T. S. Gaul. Foundations of fast communication via xml. *Ann. Softw. Eng.*, 13(1-4), 2002.
- [11] W. Lu, K. Chiu, A. Slominski, , and D. Gannon. A streaming validation model for soap digital signature. In *14th IEEE International Symposium on High Performance Distributed Computing (HPDC-14)*, July 2005.
- [12] 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.
- [13] J. Richter. *CLR via C#*. Microsoft, 2006.
- [14] J. L. Sussman, E. E. Abola, N. O. Manning, and J. Prilusky. The protein data bank: Current status and future challenges.
- [15] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30, 2005.
- [16] R. van Engelen. Constructing finite state automata for high performance xml web services. In *Proceedings of the International Symposium on Web Services (ISWS)*, 2004.
- [17] W3C. Document object model (dom) level 1 specification (second edition). <http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/>, 2000.
- [18] W3C. Xml information set (second edition). <http://www.w3.org/TR/xml-infoset/>, 2003.
- [19] 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.

## APPENDIX

---

```

<xs:element name="MoleculeType">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="moleculeName" type="xs:string" />
      <xs:element name="moleculeRadius" type="xs:double" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="atom">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="fieldName" type="xs:string" />
        <xs:element name="atomNumber" type="xs:int" />
        <xs:element name="atomName" type="xs:string" />
        <xs:element name="elementName" type="xs:string" />
        <xs:element name="residueName" type="xs:string" />
        <xs:element name="residueNumber" type="xs:int" />
        <xs:element name="coordinate">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="x" type="xs:double" />
              <xs:element name="y" type="xs:double" />
              <xs:element name="z" type="xs:double" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="charge" type="xs:double" />
        <xs:element name="radius" type="xs:double" />
        <xs:element name="sysUnique" type="xs:boolean" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:sequence>
</xs:complexType>

```

---