

A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs

Yinfei Pan¹, Wei Lu², Ying Zhang¹, Kenneth Chiu¹

1. Department of Computer Science, State University of New York, Binghamton

2. Computer Science Department, Indiana University
ypan3@binghamton.edu, kchiu@cs.binghamton.edu

Abstract

A number of techniques to improve the parsing performance of XML have been developed. Generally, however, these techniques have limited impact on the construction of a DOM tree, which can be a significant bottleneck. Meanwhile, the trend in hardware technology is toward an increasing number of cores per CPU. As we have shown in previous work, these cores can be used to parse XML in parallel, resulting in significant speedups. In this paper, we introduce a new static partitioning and load-balancing mechanism. By using a static, global approach, we reduce synchronization and load-balancing overhead, thus improving performance over dynamic schemes for a large class of XML documents. Our approach leverages libxml2 without modification, which reduces development effort and shows that our approach is applicable to real-world, production parsers. Our scheme works well with Sun's Niagara class of CMT architectures, and shows that multiple hardware threads can be effectively used for XML parsing.

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, such as semantics, autonomic behaviors, service composition, and service orchestration [22]. 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 [5, 10]. These penalties can prevent the acceptance of XML in use cases that may otherwise benefit.

A number of techniques have been developed to improve the performance of XML, ranging from binary XML [4, 25] to schema-specific parsing [6, 13, 21] to hardware acceleration [1]. Generally speaking, however, these approaches only speed up the actual parsing, and not the DOM con-

struction. They are thus more applicable to SAX-style parsing.

In SAX-style parsing, the parsing results are communicated to the application through a series of callbacks. The callbacks essentially represent a depth-first traversal of the XML Infoset [26] represented by the document. SAX-style parsing has the benefit that the XML Infoset need never be fully represented in memory at any one time. Integrating SAX-style parsing into an application can be awkward, however, due to the need to maintain state between callbacks.

In DOM-style parsing, an in-memory, tree data structure is constructed to represent the XML document. When fully constructed, the data structure is passed to the application, which can then traverse or store the tree. DOM-style parsing can be intuitive and convenient to integrate into applications, but can be very memory intensive, both in the amount of memory used, and in the high overhead of memory management.

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 [20].

In this paper, we investigate parallel XML DOM parsing on a multicore computer, and present a static scheme for load-balancing the parsing load between cores. Our scheme is effective for up to six cores. In our previous work [14], we used a dynamic load-balancing scheme that assigned work to each core on-the-fly as the XML document was being parsed. While effective, we observe that most large XML documents (larger than a few 100K) usually contain one or more large arrays and their structures tend to be shallow (less than 10 deep at the array level). For these documents, static partitioning can be more efficient. Our targeted application area is scientific computing, but we believe our approach is broadly applicable, and works well for what we believe to be the most common subset of large XML docu-

ments. Even if not a proper array, as long as the structure is shallow, our technique will work well.

Even though our technique will not currently scale to large numbers of cores, most machines currently have somewhere from 1-4 cores, and we can scale to that number. Furthermore, the alternative might be simply letting the extra cores go to waste while the application waits for the XML I/O to complete. We have also found that CMT can provide some amount of speedup.

Another advantage of our approach is that we focus on partitioning and integrating sequentially-parsed chunks, and thus can leverage off-the-shelf sequential parsers. We demonstrate this by using the production-quality libxml2 [24] parser without modification, which shows that our work applies to real-world parsers, not just research implementations.

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.

2. Overview

Concurrency could be used in a number of ways to improve XML parsing performance. One approach would be to pipeline the parsing process by dividing it into stages. Each stage would then be executed by a different thread. This approach may provide speedup, but software pipelining is often hard to implement well, due to synchronization and memory access bottlenecks, and to the difficulties of balancing the pipeline stages. More promising is a data-parallel approach. Here, the XML document would be divided into some number of chunks, and each thread would work on the chunks independently. As the chunks are parsed, the results are merged.

To divide the XML document into chunks, we could simply treat it as a sequence of characters, and then divide the document into equal-sized chunks, assigning one chunk to each thread. Any such structure-blind partitioning scheme, however, would require that each thread begin parsing from an arbitrary point in the XML document, which is problematic. Since an XML document is the serialization of a tree-structured data model (called XML Infoset [2, 26]) traversed in left-to-right, depth-first order, such a division will create chunks corresponding to arbitrary parts of the tree, and thus the parsing results will be difficult to merge back into a single tree. Correctly reconstructing namespace scopes and references will be especially challenging.

This thus leads us to the parallel XML parsing (PXP) approach [14]. We first use an initial pass, known as *preparing*, to determine the logical tree structure of an XML doc-

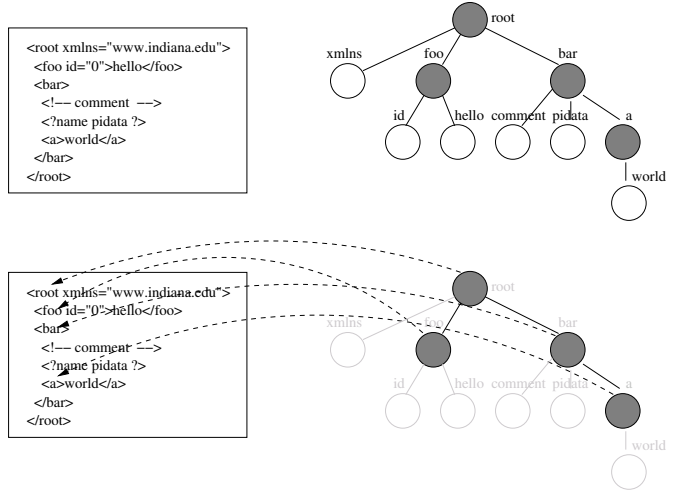


Figure 1: The top diagram shows the XML Infoset model of a simple XML document. The bottom diagram shows the skeleton of the same document.

ument. This structure is then used to divide the XML document such that the divisions between the chunks occur at well-defined points in the XML grammar. This provides enough information so that each chunk can be parsed starting from an unambiguous state.

This seems counterproductive at first glance, since the primary purpose of XML parsing is to build a tree-structured data model (i.e., XML Infoset) from the XML document. However, the structural information needed to locate known grammatical points for chunk boundaries can be significantly smaller and simpler than that ultimately generated by a full XML parser, and does not need to include all the information in the XML Infoset data model. We call this simple tree structure, specifically designed for partitioning XML documents, the *skeleton* of the XML document, as shown in Figure 1. Once the preparing is complete, we use the skeleton to divide the XML document into a set of well-defined chunks (with known grammatical boundaries), in a process we call *task partitioning*. The tasks are then divided into a collection of well-balanced sets, and multiple threads are launched to parse the collection, with one thread assigned to each task set.

Libxml2 provides a number of public functions to facilitate parsing XML fragments, and for gluing DOM trees together. We leverage these functions to do the actual full parsing and reintegration.

When the parsing is complete, we postprocess the DOM tree to remove any temporary nodes that were inserted to isolate concurrent operations from one another. These temporary nodes allow us to use libxml2 without modification. The entire process is shown in Figure 2. Further details on preparing can be found in our previous work [14].

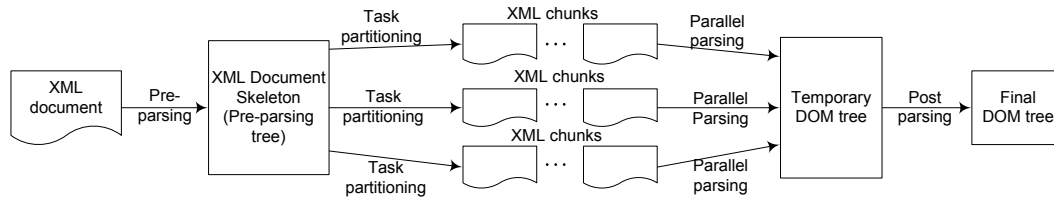


Figure 2: The PXP architecture first uses a preparser to generate a skeleton of the XML document. We next partition the document into chunks, and assign each task to a thread. (This stage is actually sequential, though the diagram has multiple arrows for it.) After the parallel parsing is complete, we postprocess the document to remove temporary nodes.

3. Task Partitioning

Once the skeleton has been computed, we partition the XML document into chunks, with each chunk representing one task. The goal of this stage is to find a set of tasks that can be partitioned into subsets such that each subset will require the same amount of time to parse. In other words, we want the partitioning to be well-balanced. Another goal is to maximize the size of each task, to reduce overhead costs. This problem is a variant of the bin packing problem. Note that these two goals are in opposition. Smaller tasks will make it easier to partition them such that all subsets take the same amount of time.

Following the work in parallel discrete optimization algorithms [9, 8, 12] and parallel depth-first search [17], we consider two kinds of partitioning: static and dynamic. Static partitioning is performed before parallel processing begins, while dynamic partitioning is performed on-the-fly as the parallel processing proceeds based on the run-time load status. Regardless of which scheme is used, the key goal of the task partitioning is to generate a set of optimally balanced workloads.

The conventional wisdom is that dynamic partitioning will lead to better load-balancing because the partitioning and load scheduling are based on run-time knowledge, which will be more accurate. Static partitioning uses *a priori* knowledge to estimate the time required to execute each task. Accurate estimates can be difficult to obtain in some cases, which will lead to poor balancing. Furthermore, static partitioning is usually a sequential preprocessing step, which will limit speedup according to Amdahl’s law[3].

However, the potential benefits of dynamic partitioning are not free. The intensive communication and synchronization between the threads can incur significant performance overhead. In contrast, when static partitioning is completed, all threads can run independently. Hence to achieve a good performance solution, the choice of the partitioning scheme should depend on the complexity of the problem. If the problem corresponds to a flat structure (e.g., an array), static

partitioning is preferred over dynamic partitioning. However, if the problem is a highly irregular, unbalanced tree or graph structure, dynamic schemes become correspondingly more beneficial.

3.1. Subtree Tasks

A graph can be statically partitioned into a set of subgraphs using a number of different techniques [11]. A natural partitioning for XML is as a set of subtrees. We call the set of connected nodes above the subtrees the *top tree*, as shown in Figure 3(b).

Each subtree is considered a task. Our algorithm maintains a list of these tasks. At each iteration, the largest task is removed from the list, and the root of the corresponding subtree is moved to the top tree. Each child of the subtree then forms a new subtree, and is placed onto the list as a new task.

Using the skeleton, our static partitioning generates the set of subtrees starting from the root. We first parse the root node, and initialize the top tree with it. We then add all the immediate subtrees below the root to the task list.

We then proceed recursively in the following fashion. At every iteration, we remove the largest task from the list, parse the root of the corresponding subtree and move it to the top tree. We then add the generated subtrees which were under this node back to the task list. A priority queue is used to efficiently maintain the largest task in the list. The effect is to grow the top tree down as we recursively create subtasks and add new DOM nodes to the bottom of the top tree. Note that the top tree consists of actual DOM nodes, while the skeleton is the abbreviated structure in a concise form.

When a DOM node is added to the bottom of the top tree, the child nodes have not yet been created, but are merely “logical” nodes, which exist only in so far as they are in the skeleton. However, the DOM nodes in the top tree need to be created by parsing the corresponding XML with a full parser, which creates a problem. Because XML includes all descendants of a node within the lexical range of the node, as defined by the start- and end-tags, we cannot use a parser directly on the original text to generate a single, childless

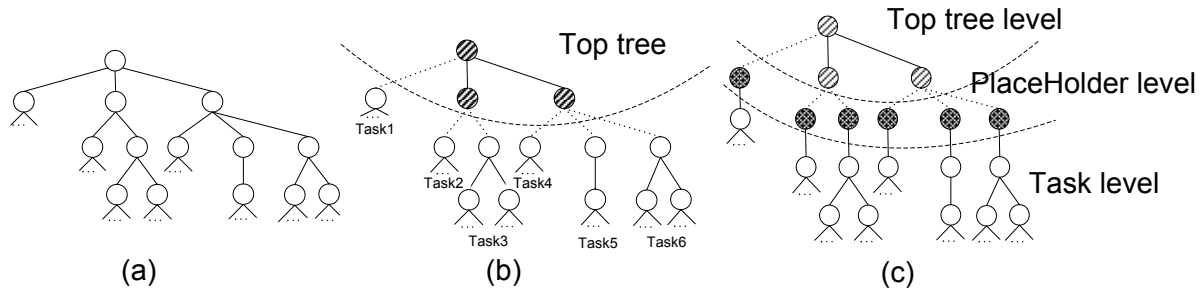


Figure 3: Logical structure of the top tree, tasks' placeholder, and tasks. In (a), we see the complete tree. In (b) we see the top tree and the subtree tasks. In (c), the placeholder level is also shown.

DOM node. Parsing the original text would also generate all the children nodes, leaving no work to be done during the actual parallel parsing stage.

As a workaround to this lexical problem, we create a duplicate, but childless element in a temporary string buffer, and parse this duplicate element to generate a childless DOM node. The duplicate element is created simply by concatenating the start- and end-tags of the original element, and omitting all of the content. For example, if the original element were `<a><c1> . . . </c1><c2/>`, then the temporary buffer would contain `<a>`. This procedure could be expensive if the top tree is large, but in practice we have found that most large documents are relatively shallow, and thus the top tree is small compared to the total size.

The “size” of a task should correspond to how much time it will take to complete. We currently use the number of characters in the corresponding XML chunk as an estimate of this. We have found so far that it works well, but a more sophisticated measure may be needed for some documents.

To prevent too many small subtree tasks from being generated, and thus increasing overhead, we terminate the recursion when the number of tasks is greater than a preset limit, currently chosen to be 100. When such a limit is reached, there will usually be enough tasks such that they can be assigned to threads in a well-balanced manner.

As we proceed to recursively create new subtasks and grow the top tree down, we repeatedly add new DOM nodes to the top tree. Below any node in the top tree, the left-to-right ordering of its children will correspond to the order in which they were added to that node. In a straightforward, but incorrect, implementation, this order would correspond to the order that the corresponding subtrees were removed from the task list, which would not correspond to the actual left-to-right ordering of the elements in the XML document.

Thus, to preserve the left-to-right ordering, we create placeholder children between the bottom of the top tree and the subtrees below it. These placeholders are added to the parent node immediately when it is added to the top tree, and thus the left-to-right ordering is known and can be pre-

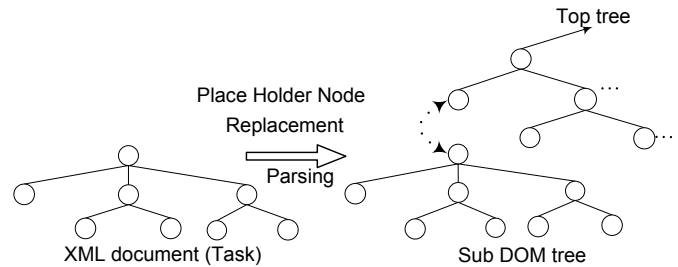


Figure 4: Attaching a subtree to the top tree is done by replacing the placeholder node in-place, after the subtree is parsed.

served at that time. Each new subtree task is created with a pointer to its corresponding placeholder. These placeholders are removed as the top tree grows down, so that only the leaves of the top tree have placeholders. Even these are removed after the parallel parsing is complete, as described below. The entire process is diagrammed in Figure 5.

The placeholder nodes also serve to isolate concurrent operations from one another. Once subtree partitioning is complete, each subtree task will be executed by a separate thread. These threads will complete at different times, however, so if a thread directly adds the root of a subtree to the parent node, this will result in different left-to-right orderings depending on the completion order. The placeholder nodes avoid this problem because we can use libxml2 to directly replace the placeholder nodes with the actual subtree root nodes, in place, as shown in Figure 4. After we finalize the task partitioning, we will see a logical three level structure connecting top tree, placeholders, and tasks together as Figure 3(c) shows. The nodes in the top tree are fully constructed, permanent DOM nodes; the nodes in the placeholder level are temporary DOM nodes used to connect the tasks to the proper left-right position within the permanent DOM nodes; and the nodes below this are logical nodes that currently only exist in the skeleton.

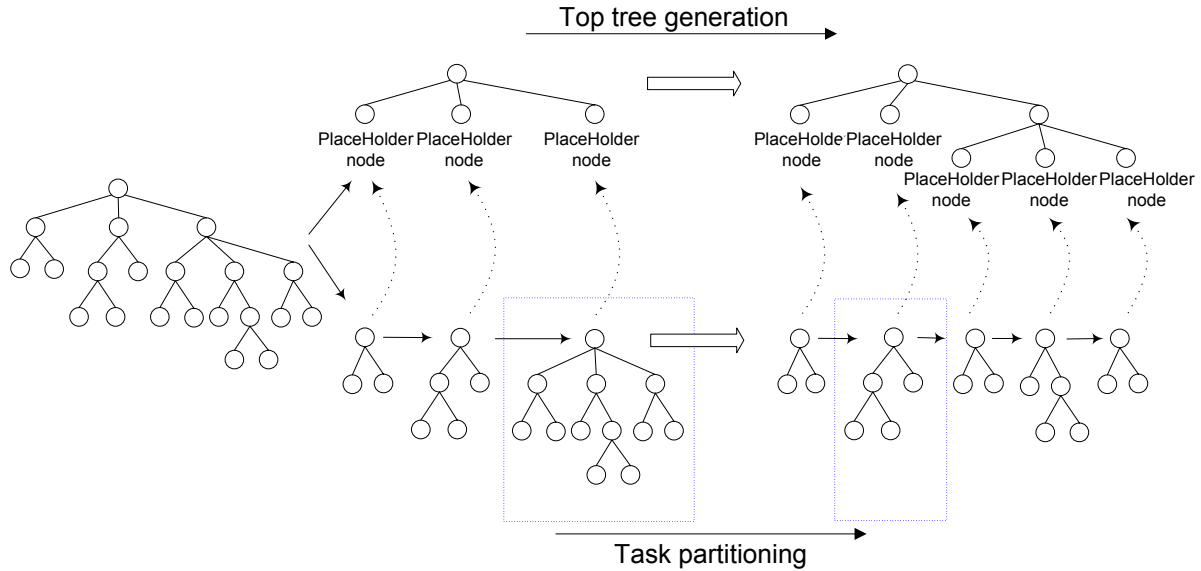


Figure 5: As the parsing proceeds, the top tree is grown downwards. As each subtree is processed, its root node is parsed and moved to the top tree by an in-place replacement of the corresponding placeholder node, and additional placeholder nodes are created. At each iteration, the largest remaining subtree is chosen for processing.

3.2. Array Tasks

As mentioned earlier, we have found that most large XML documents consist of one or more long sequences of child elements. The sequences may have similar, or even identical types of elements. We call such sequences *arrays*.

The subtree-based task partitioning scheme is not suitable for arrays, which may contain a large number of child nodes. Adding all of these to the task list would be prohibitively inefficient. Therefore, we use a simple heuristic to recognize arrays in the skeleton, and handle them differently, which also improves load balancing.

Our heuristic is to treat a node as an array if the number of children is greater than some limit. During the traversal, we check whether or not the number of children is greater than this limit. If so, we treat the children as an array. Our limit is based on the number of threads, and is currently set to 20 times the number of threads.

Once the current node is identified as an array, we divide its elements into equal-sized, continuous ranges. The size is chosen such that the number of ranges in the array is equal to the number of threads. Each range is treated as a task. This differs from the subtree tasks, in that each task is now a forest rather than a subtree. These tasks are added to a FIFO queue as they are generated, which is used during task assignment as described below.

For subtree tasks, we create one node as the placeholder for the entire subtree. When the task finishes, we simply replace the placeholder with the actual DOM node. For array

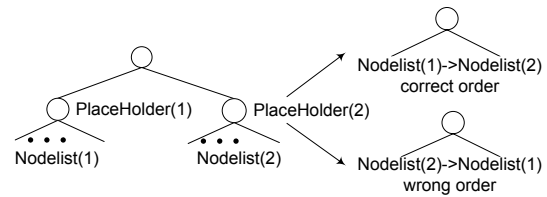


Figure 6: When an array task is completed by a thread, it cannot directly add its completed forest to the corresponding array node, since the tasks may complete in the wrong order.

tasks, we must create a separate placeholder for each range. This is because otherwise, each thread would attempt to add the child nodes in its range concurrently to the parent node, resulting in race conditions. Thus, a range placeholder is created for each array task, to isolate the concurrent operations of each thread from one another.

When a thread finishes a subtree task, it can immediately replace the placeholder node with the actual root node of the subtree. When a thread finishes an array task, however, the result is a forest of subtrees that must be added to the actual array element, which is in the position of the *parent* of the range placeholder (one level higher), rather than in the position of the range placeholder itself. This operation cannot occur in parallel, because otherwise multiple threads would attempt to add child nodes at the same time, as shown in Figure 6.

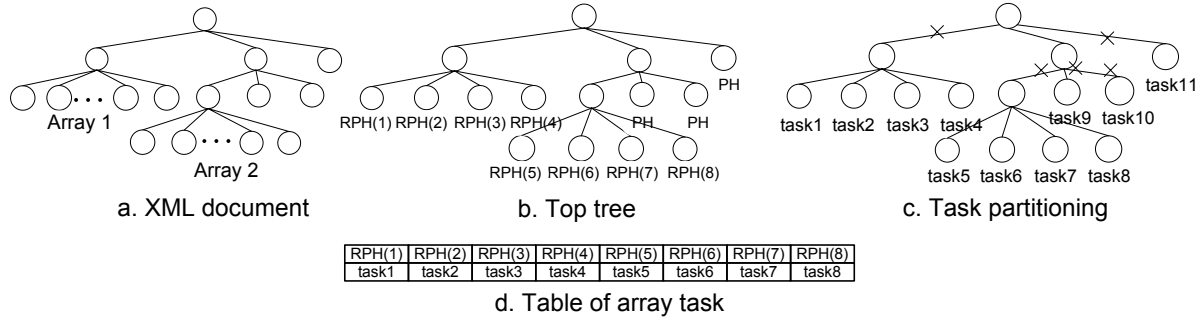


Figure 7: In (a), we see the original tree structure. In (b), range placeholders have been added. In (c), we see how each range corresponds to a task. The table in (d) is used to ensure that the ranges are added to the corresponding array element in the correct order.

Thus, we maintain an additional table for each array to record the proper ordering for the child range placeholders, and traverse this array sequentially after the end of the parallel parsing stage. Figure 7 (a) shows an XML document with two large arrays, its top tree with placeholders and task partitioning are shown in (b) and (c). RPH means range placeholder, PH means a subtree task placeholder.

4. Task Assignment

Once we have partitioned the XML document into tasks, we now must assign them to threads in a balanced manner, which is the goal of the task assignment stage. We have two sets of tasks, the subtree tasks and the array tasks.

We first assign the array tasks in the FIFO queue to threads in a round-robin fashion. This is because for most large XML documents the arrays are the largest part and also the most easily balanced part. As we assign the tasks, we keep track of the current workload of each thread. Because we have carefully chosen the range sizes, each thread will be guaranteed to have exactly one array task per array. Also, the FIFO queue will maintain the order of the ranges on each array, which will be used in post-processing stage. After the array tasks have been assigned, we then assign the subtree tasks to the threads. Each subtree task is assigned to the thread that currently has the least workload.

Note that each thread’s task set is maintained by using a private queue. This eliminates contention during the actual parsing phase.

5. Parallel Parsing

After the task assignment is complete, each thread starts parsing the tasks in its private queue. Each task has a corresponding placeholder node that was created during the top tree generation process. Thus, they inherit the context (DTD, namespaces, etc.) of their parent nodes. Thus,

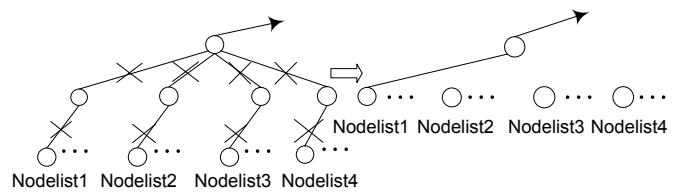


Figure 8: Each array task generates a forest of subtrees, which then must be added back to the associated array node. Intervening links, marked with X, are deleted.

each task is parsed within the context of its corresponding placeholder. In our implementation, we used the libxml2 function, `xmlParseInNodeContext()`. This function can parse any “well-balanced chunk” of an XML document within the context of the given node. A well-balanced chunk is defined as any valid content allowed by the XML grammar. Since the XML fragments generated by our task partition are well-balanced, we can use this function to parse each task.

During the parallel parsing, after each task is completed, its corresponding subtree needs to be inserted into the right place under the top tree. To do that, for each subtree task, we can simply replace its corresponding placeholder with its root node using libxml2 function `xmlReplaceNode()`. However, for array tasks, which produce a forest of subtrees under the range placeholder node, we clearly cannot simply replace the placeholder and as discussed earlier in Figure 6, there may exist race conditions if we remove the placeholders and add back lists of nodes. Therefore, guided by the table of array tasks, shown in Figure 7 (d), we then remove placeholders of array tasks sequentially during the post-parsing stage, and use the libxml2 function, `xmlAddChildList()` to add back node lists in their correct order. This is shown in Figure 8.

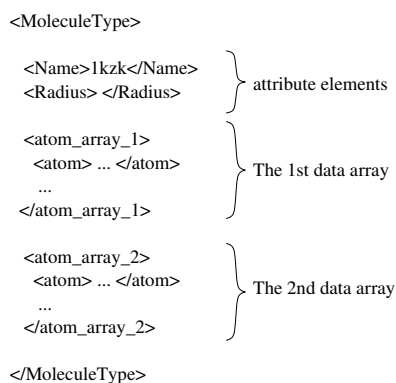


Figure 9: The structure of the test XML documents.

6. Performance Results

Our experiments were run on a Sun Fire T1000 machine, with 6 cores and 24 hardware threads (CMT). We observed that most large XML documents, particularly in scientific applications, are relatively broad rather than deep, also typically contain one or two large arrays and the structure tends to be shallow. Hence we select a large XML file, which contains the molecular information as shown in Figure 9, representing the typical structural shape of XML documents in scientific applications. This was based on XML documents obtained from the Protein Data Bank [19]. It consists of two large array representing the molecule data as well as a couple elements for the molecule attributes. To obtain the different sizes, the molecule data part of the documents were repeated.

Every test is run ten times to get the average time and the measurement of the first time is discarded, so as to measure performance with the file data already cached, rather than being read from disk. The programs are compiled by Sun Workshop 5.2 CC with the option `-O`, and the libxml2 library we are using is 2.6.16.

During our initial experiments, we noticed poor speedup during a number of tests that should have performed well. We attributed this to lock contention in `malloc()`. To avoid this, we wrote a simple, thread-optimized allocator around `malloc()`. This allocator maintains a separate pool of memory for each thread. Thus, as long as the allocation request can be satisfied from this pool, no locks need to be acquired. To fill the pool initially, we simply run the test once, then free all memory, returning it to each pool. Our tests with straight libxml2 use our allocator, since the results of straight libxml2 are better with our allocator than without it.

Our allocator is intended simply to avoid lock contention, since we wanted to focus on the parsing itself. There is significant work on multi-threaded memory allo-

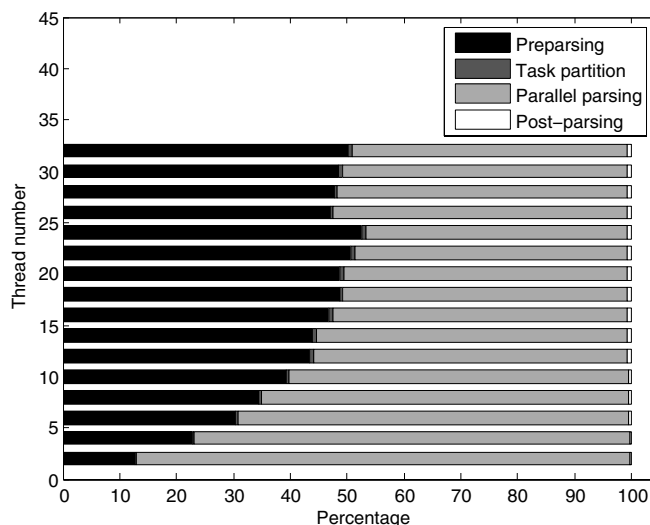


Figure 10: Performance breakdowns.

cation that can be used to provide a more sophisticated solution [15].

6.1. Performance Breakdown

Our implementation performs the stages as described in the paper, and thus we can measure the time of each stage. The sequential stages include preparing, task partitioning, and post-parsing, and the parallel stages are just parallel parsing which is done by all the threads in parallel. Our performance breakdown experiment is done on our test XML document sized to 18M bytes. The result is shown on Figure 10, in which the different gray levels represents the percentage of the running time of each stage. We tested from 2 to 32 threads, but we show only the even numbered results on Figure 10 to make the graph clearer.

The most immediate feature to note is that the preparing is by far the most time consuming part of the sequential stage, and the static partitioning stage is not a significant limit to the parallelism. Thus, if we wish to address the effects of Amdahl's law, we'll need to target the preparing.

As a general trend up to 24 threads, we observed that the percentage on the preparing grows from 13% to 53%. Percentage on task partitioning grows from 0.17% to 0.71%. Percentage on post-processing grows from 0.16% to 0.73%. This means that as the number of threads increase, the sequential stages take in increasing percentage of the total time, and obviously can cause the performance to degrade. Meanwhile, the time cost of the parallel parsing stages drop from 87% to 46%. When the participating threads reach more than 24, we run out of hardware threads, thus requiring the OS to start context switching threads. This causes the percentage of parallel parsing to increase sud-

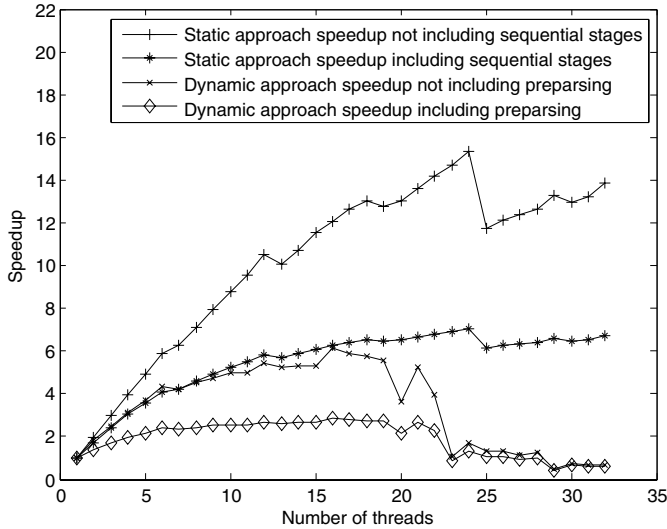


Figure 11: Speedup graph from 2 to 32 threads.

denly from 24 threads' 46% to 26 threads' 52%, and then we have minor reduction at each increased measurement until 32 threads' 48%. As the number of threads increase, the post-processing stages increase from 2 threads' 0.16% to 32 threads' 0.74%.

6.2. Speedup and Efficiency Analysis

To show the benefit of the static approach on these kind of shallow structured XML documents, we compare the static approach with the dynamic approach introduced in our earlier work [14]. Referring to Figure 11, we did four types of measurement on both the static parallel parsing approach and dynamic parallel parsing approach. The tests were conducted on our test XML document sized to 18M bytes, and to better explore the cause of performance limitations on each approach, we have two graph lines: one is the total speedup and the other is the speedup not counting the sequential stages. Speedup in this case is computed against the time obtained by a pure libxml2 parse without any prearsing or other steps only needed for PXP.

For the static approach, the sequential stages include prearsing, task partitioning and assignment, post-processing; and for the dynamic approach, the sequential stages include just prearsing. It appears that every six threads, the speedup will have a distinct drop. This is due to the fact that there are only six cores on the machine. Every six threads, the number of hardware threads per core must increase. To better see the speedup degradation pattern, we generated the efficiency graph as shown in Figure 12. The efficiency is defined as the speedup divided by the number of threads. With the efficiency graph, we can then clearly see the performance degradation.

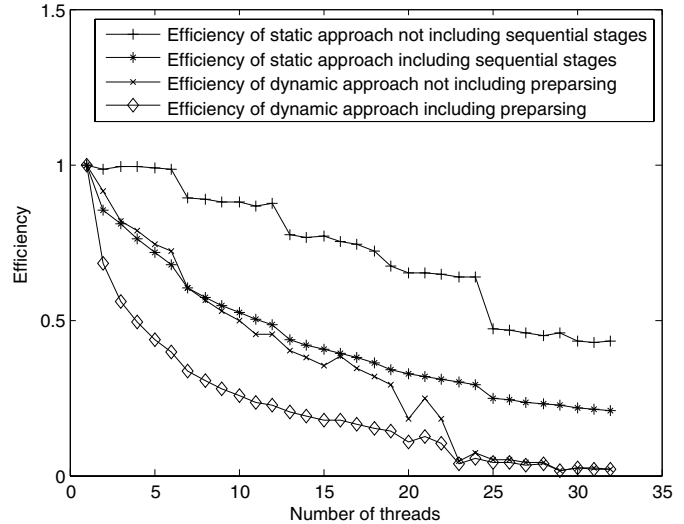


Figure 12: Efficiency graph from 2 to 32 threads.

Since the dynamic load-balancing is obtained by runtime work load stealing, which incurs intensive communication and synchronization cost with an increasing number of threads, it is not scalable to the number of cores, as Figure 11 shows. Without prearsing, it reaches the maximum speedup with 16 threads, which is 6.1. After that point, it drops to the lowest value of 0.47 at 29 threads and continues to below one time speedup with the increase of the threads number. And for efficiency, dynamic approach also dropped to 0.02 in the 32 threads case.

These two figures show that though the dynamic approach has proved to be more flexible and suitable for parallel XML parsing on complex XML documents, for large XML documents with a simple array-like structure, the static approach is more scalable than the dynamic approach. Note although more sophisticated dynamic load-balancing technologies [12] exist to improve the scalability, for those shallow structured documents the static scheme should be hard to challenge.

To better understand the pros and cons between the static approach and the dynamic approach, we also did the tests on complex structured XML. We generated a deep and irregular structured XML document with the size of 19M bytes. The tests results as in Figure 13, however, show that in such case the speedup of static approach is much worse than the dynamic approach. The reason is that for complex XML documents it is harder to statically achieve balanced load. And thus, with the number of threads increased, load imbalance becomes more serious.

In addition, we also note that the near-linear speedup of the parallel parsing when not including the sequential portion means that we can make significant improvements in speed by targeting prearsing, which we have shown by the

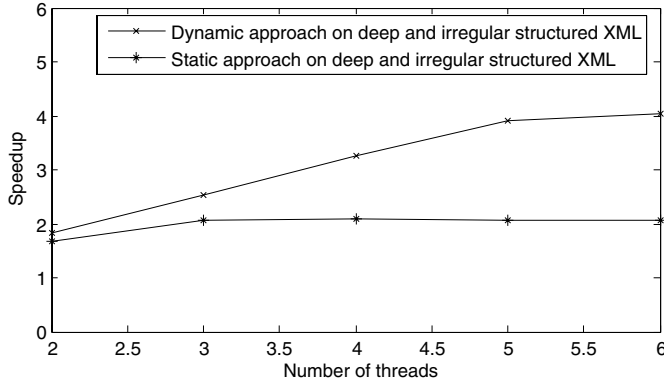


Figure 13: Speedup not including sequential stages on the complex XML document.

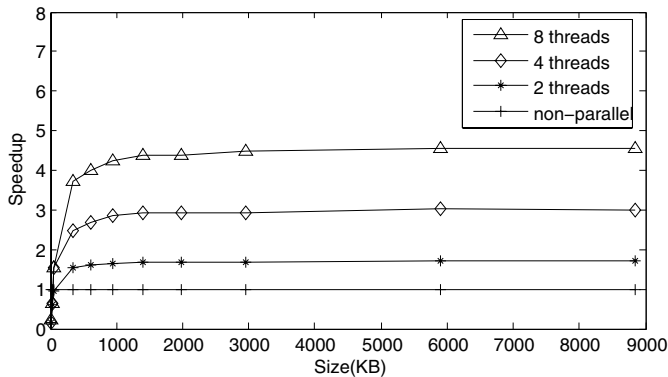


Figure 14: Speedup with various sized simple.xml.

performance breakdown to account for the vast majority of the sequential time.

6.3. Scalability Analysis

To further study the scalability of the parallel XML parsing with the static load-balancing scheme, we vary the size of a different test XML document to see how the speedup varies with different document sizes. This test document contained a simple linear array structure. We see from Figure 14 that our implementation scales well across a range of document sizes. At the smaller sizes, however, our technique begins to show less value, with no appreciable benefit till the file size exceeds 256KB. A test file of 346,293 bytes shows speedups of 1.5, 2.5 and 3.7 for 2, 4, and 8 threads, respectively. Since our target so far has been larger XML documents, we found this to be acceptable, but may seek to address this in future work.

7. Related work

The static partition approach introduced in this paper is similar with the “search-frontier splitting” policy used in [18, 7], which solve the discrete optimization problems by the parallel Depth-first searching. By that policy the algorithm first breadth-first searches the tree until reaching a cutoff depth, called “frontier”, then each sub-tree under the cutoff depth will be treated as a task for the threads. However for XML documents, simply defining a cutoff frontier will be less effective since XML documents should have more regular shape than the searching space of those discrete optimization problems. Moreover by the preparsing, we are able to predict the size of the sub-tree while this kind of prediction will be very hard for the discrete optimization problems. Hence for XML documents our static task partitioning algorithm is more effective to find a load-balanced task set. The work by Reinefeld [18] also shows the static task-partition scheme is more scalable than the dynamic scheme due to the less communication and synchronization cost, which is consistent with our result.

There are a number of approaches trying to address the performance bottleneck of XML parsing. The typical software solutions include lazy parsing [16] and schema-specific parsing [6, 13, 21]. Schema-specific parsing leverages XML schema information, by which the specific parser (automaton) is built to accelerate the XML parsing. For the XML documents conforming to the schema, the schema-specific parsing will run very quickly, whereas for other documents an extra penalty will be paid. Most closely related to our work in this paper is lazy parsing because it also needs a skeleton-like structure of the XML document for the lazy evaluation. That is, first a skeleton is built from the XML document to indicate the basic tree structure, thereafter, based on the user’s access requirements, the corresponding piece of the XML document will be located by looking up the skeleton and be fully parsed. 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 from each other. Hardware based solutions[1, 23] also are promising, particularly in the industrial arena. But to our best knowledge, there is no such work leveraging the data-parallelism model as PXP.

8. Conclusion and Future Work

The advent of multicore machines provides a unique opportunity to improve XML parsing performance. We have shown that a production-quality parser can easily be adapted to parse in parallel using a fast preparsing stage to provide structural information, namely the skeleton of a XML document. Based on the skeleton, load-balancing then becomes the key to a scalable parallel algorithm. As far as

general parallel algorithms are concerned, static partitioning schemes and dynamic schemes have their own advantages and disadvantages. For parallel XML parsing, however we show that the static partitioning scheme introduced in this paper is more scalable and efficient than the dynamic scheme on multicore machines, for what we believe to be common XML documents. This is due to the fact that most large XML documents usually contain array-like structures and their document structures tend to be shallow. Our static load-balancing scheme leverages these characteristics and is designed to quickly locate and partition the array structures. Although it introduces some sequential processing, the highly independent parallelism which can then be obtained minimizes the synchronization cost during the parallel parsing. Furthermore, a significant pragmatic benefit of the static load-balancing scheme is it can directly use off-the-shelf XML parsers without requiring any modification.

The limitation of the static load-balancing scheme is it may not generalize to XML documents with the arbitrary shapes, and may even fail for some deeply-structured XML documents. Hence a hybrid solution, which can provide both static and dynamic load-balancing control, will be interesting. Dynamic partitioning could also be improved by sophisticated work-stealing, scheduling policies, and lock-free structures. Also our experiments show that with greater number of cores the sequential reparsing stage becomes the major limitation to scalability.

References

- [1] Datapower. <http://www.datapower.com/>.
- [2] N. Abu-Ghazaleh and M. J. Lewis. Differential deserialization for optimized soap performance. *SC—05 (Supercomputing): International Conference for High Performance Computing, Networking, and Storage*, Seattle WA, November 2005.
- [3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. pages 483–485, 1967.
- [4] K. Chiu, T. Devadithya, W. Lu, and A. Slominski. A binary xml for scientific applications. In *Proceedings of e-Science 2005*. IEEE, 2005.
- [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*, 2004.
- [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. Grama and V. Kumar. Parallel processing of combinatorial optimization problems. *ORSA Journal of Computing*, 1995.
- [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] M. R. Head, M. Govindaraju, R. van Engelen, and W. Zhang. Grid scheduling and protocols—benchmarking xml processors for applications in grid web services. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 121, New York, NY, USA, 2006. ACM Press.
- [11] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Supercomputing*, 1996.
- [12] 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.
- [13] W. M. Lowe, M. L. Noga, and T. S. Gaul. Foundations of fast communication via xml. *Ann. Softw. Eng.*, 13(1-4), 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. M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2004. ACM Press.
- [16] M. L. Noga, S. Schott, and W. Lowe. Lazy xml processing. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, 2002.
- [17] V. N. Rao and V. Kumar. Parallel depth first search. part i. implementation. *Int. J. Parallel Program.*, 16(6):479–499, 1987.
- [18] 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.
- [19] J. L. Sussman, E. E. Abola, N. O. Manning, and J. Prilusky. The protein data bank: Current status and future challenges.
- [20] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30, 2005.
- [21] R. van Engelen. Constructing finite state automata for high performance xml web services. In *Proceedings of the International Symposium on Web Services (ISWS)*, 2004.
- [22] R. van Engelen and K. Gallivan. The gsoap toolkit for web services and peer-to-peer computing networks. In *the 2nd IEEE International Symposium on Cluster Computing and the Grid*, Berlin, Germany, May 2002.
- [23] 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.
- [24] D. Veillard. Libxml2 project web page. <http://xmlsoft.org/>, 2004.
- [25] W3C. Xml binary characterization properties. <http://www.w3.org/TR/xbc-properties/>.
- [26] W3C. Xml information set (second edition). <http://www.w3.org/TR/xml-infoset/>, 2003.