A Parallel Approach to XML Parsing

Wei Lu #1, Kenneth Chiu *2, Yinfei Pan *3

#*Computer Science Department, Indiana University*
150 S. Woodlawn Ave. Bloomington, IN 47405, US
[1]welu@cs.indiana.edu

**Department of Computer Science, State University of New York -Binghamton*
P.O. Box 6000, Binghamton, NY 13902, US
[2]kchiu@cs.binghamton.edu [3]ypan3@cs.binghamton.edu

Abstract— **A language for semi-structured documents, XML has emerged as the core of the web services architecture, and is playing crucial roles in messaging systems, databases, and document processing. However, the processing of XML documents has a reputation for poor performance, and a number of optimizations have been developed to address this performance problem from different perspectives, none of which have been entirely satisfactory. In this paper, we present a seemingly quixotic, but novel approach: parallel XML parsing. Parallel XML parsing leverages the growing prevalence of multicore architectures in all sectors of the computer market, and yields significant performance improvements. This paper presents our design and implementation of parallel XML parsing. Our design consists of an initial preparsing phase to determine the structure of the XML document, followed by a full, parallel parse. The results of the preparsing phase are used to help partition the XML document for data parallel processing. Our parallel parsing phase is a modification of the libxml2 [1] XML parser, which shows that our approach applies to real-world, production quality parsers. Our empirical study shows our parallel XML parsing algorithm can improved the XML parsing performance significantly and scales well.**

I. Introduction

XML's emergence as the *de facto* standard for encoding tree-oriented, semi-structured data has brought significant interoperability and standardization benefits to grid computing. Performance, however, is still a lingering concern for some applications of XML. A number of approaches have been used to address these performance concerns, ranging from binary XML to schema-specific parsing to hardware acceleration.

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. Tomorrow's computers will have more cores rather than exponentially faster clock speeds, and software will increasingly have to rely on parallelism to take advantage of this trend [2].

In this paper, we investigate the seemingly quixotic idea of parsing XML in parallel on a shared memory computer, and develop an approach that scales reasonably well to four cores.

Concurrency could be used in a number of ways to improve XML parsing performance. One approach would be to use pipelining. In this approach, XML parsing could be divided into a number of stages. Each stage would be executed by a different thread. This approach may provide speedup, but software pipelining is often hard to implement well, due to synchronization, load-balance and memory access costs.

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. This requires that each thread begin parsing from an arbitrary point in the XML document, however, which is problematic. Since an XML document is the serialization of a tree-structured data model (called XML Infoset [3]) 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 also be challenging. Furthermore, most chunks will begin in the middle of some string whose grammatical role is unknown. It could be a tag name, an attribute name, an attribute value, element content, etc. This could be resolved by extensive backtracking and communication, but that would incur overhead that may negate the advantages of parallel parsing. Apparently, instead of the equal-sized physical decomposition, the ability of decomposing the XML document based on its logical structure is the key toward the efficient parallel XML parsing.

The results of parsing XML can vary from a DOM-style, data structure representing the XML document, to a sequence of events manifest as callbacks, as in SAX-style parsing. Our parallel approach in this paper focuses on DOM-style parsing, where a tree data structure is created in memory that represents the document. Our targeted application area is scientific computing, but we believe our approach is broadly applicable. Our implementation is based on the production quality libxml2 [1] parser, which shows that our work applies to real-world parsers, not just research implementations.

Current programming models for multicore architectures provide access to multiple cores via threads. Thus, in the rest of the paper, we use the term thread rather than core. To avoid scheduling issues that are outside the scope of this paper, we assume that each thread is executing on a separate core.

The rest of the paper is organized as follows. Section II

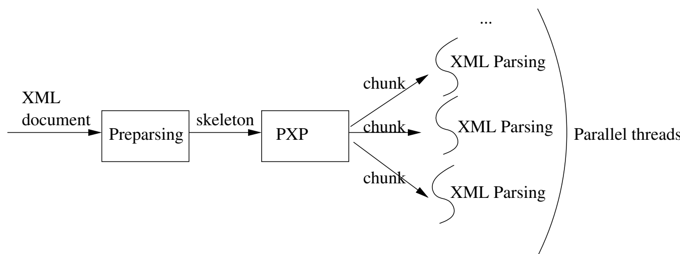

Fig. 1. The PXP architecture first uses a preparser to generate a skeleton of the XML document. This is then used to guide the partitioning of the document into chunks, which are then parsed in parallel.

describe the general architecture of our approach, PXP. Then in the section III and IV we present the algorithm design and implementation details. We present in Section V performance results. Related work is discussed in Section VI.

II. PXP

Any kind of parsing is based on some kind of machine abstraction. The problems of an arbitrary division scheme arise from a lack of information about the state of the parsing machine at the beginning of each chunk. Without this state, the machine does not know how to start parsing the chunk. Unfortunately, the full state of the parser after the Nth character cannot be provided without first considering each of the preceding $N-1$ characters.

This thus leads us to the PXP (Parallel XML Parsing) approach presented in this paper. We first use an initial pass to determine the logical tree structure of an XML document. This structure is then used to divide the XML document such the divisions between the chunks occur at well-defined points in the XML grammar. This provides enough context 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 tree structure needed to guide the parallel parsing can be significantly smaller and simpler than that ultimately generated by a normal 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 XML data decomposition, the *skeleton* of the XML document.

To distinguish from the actual XML parsing, the procedure to parse and generate the skeleton from the XML document is called *preparsing*. Once the preparsing is complete and we know the logical tree structure of the XML document, we are able to divide the document into balanced chunks and then launch multiple threads to parse the chunks in parallel. Consequently, this parallelism can significantly improve performance. Our overall architecture is shown in Figure 1.

For simplicity and performance, PXP currently maps the entire document into memory with the `mmap()` system call. Nothing precludes our general approach from working on streamed documents, or documents too large to fit into memory, but the design and implementation would be significantly more complex.

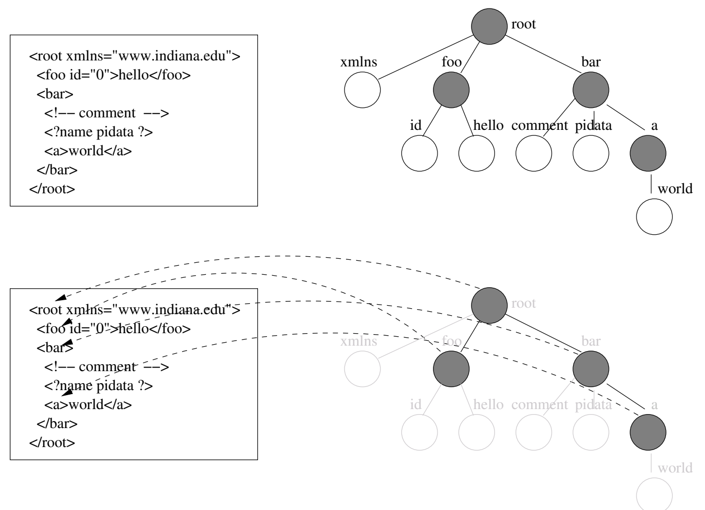

Fig. 2. The top diagram shows the XML Infoset model of a simple XML document. The bottom diagram shows the skeleton of the same document.

III. Preparsing

The goal of preparsing is to determine the tree structure of the XML document so that it can be used to guide the data-parallel, full parsing.

A. Skeleton

Conceptually the XML Infoset represents the tree structure of the XML document. However since only internal nodes (i.e., the element item) determine the topology of the tree, which is meaningful for XML data decomposition, those leaf nodes in the XML Infoset, such as attribute information items, comment information items, and even character information items, can be ignored by the skeleton. Further the element tag names are also ignored by the skeleton since they don't affect the topology of the tree at all. So as shown in the Figure 2, the skeleton essentially is a tree of unnamed nodes, isomorphic to the original XML document, and constructed from all start-tag/end-tag pairs. To facilitate the XML data decomposition, Our skeleton records the location of the start tag and end tag of each element, the parent-child relationships, and the number of children of every element.

B. Implementation

Well-formed XML is not a regular language [4], and it cannot be parsed by a finite-state automaton, but rather requires at least a push-down automaton. So even determining the fundamental structure of the XML document, just for preparsing, requires executing a push-down automaton. However since preparsing is an additional processing step for parallel parsing, it is an additional overhead not normally incurred during XML parsing. Furthermore, since it is sequential, it fundamentally limits the parallel parsing performance. Hence, a fundamental premise of our work is that preparsing can build the skeleton at minimal cost.

According to the XML specification [5] a non-validating [1] XML parser must determine whether or not a XML document is well-formed. A XML document is considered well-formed if it satisfies both requirements below:

1) It conforms to the syntax production rules defined in the XML specification.
2) It meets all the well-formedness constraints given in the specification.

However, since preparsing will be followed by a full-fledged XML parsing stage, the preparsing itself can ignore many errors. That is, for a well-formed XML document, the preparser must generate the correct result, but for a ill-formed XML document, the preparser does not need to detect any errors. Thus, our preparser only detects weak conformance to the XML specification, and hence is simpler to implement and optimize.

As the skeleton only contains the location of the element nodes in the XML document, preparsing only needs to consider the element tag pairs, and can ignore other syntactic units and production rules for such as comments, character data, and attributes. Consequently, the preparsing has a much simpler set of production rules compared to standard XML. For example the production rule of the start tag in XML 1.0 is defined as:

```
STag      ::= '<' Name (S Attribute)* S? '>'
Attribute ::= Name Eq AttValue
Name      ::= (Letter | '_' | ':') (NameChar)*
AttValue  ::= '"' ([^<&"] | Reference)* '"'
            | "'" ([^<&'] | Reference)* "'"
```

Because preparsing can ignore `Attribute` and `AttValue`, and even the entire `Name` production rule, the syntax could seemingly be simplified to just:

```
STag      ::= '<' ([^>])* '>'
```

However the above simplified production rule is incorrect due to ambiguity, because `AttValue` allows the > character by its production rule, which, if it appears, will cause the preparser to misidentify the location of the actual right angle bracket of the tag. Therefore, the correct rules are:

```
STag      ::= '<'([^'"])* AttValue* '>'
AttValue  ::= '"' ([^'"])* '"' | "'" ([^'"])* "'"
```

With same concern of possible ambiguity, the `PI`, `Comment`, and `CDATA` elements should be preserved in the preparsing production rules set because they are allowed to contained any string, including the < character, which would otherwise cause the preparser to misidentify the location of the end tag. The rest of production rules of standard XML are ignored by the preparsing.

The simplified preparsing syntax results in a much simpler parsing automaton (Figure 3), which only requires six major states, than the one needed by complete XML parsing. Predictably, the preparsing automaton runs much faster than the general XML parsing automaton.

[1] DTD and validating XML parsing are not supported by our current system for simplicity. Also DTD is being replaced by the XML Schema validation, which is usually a separate process after the XML parsing.

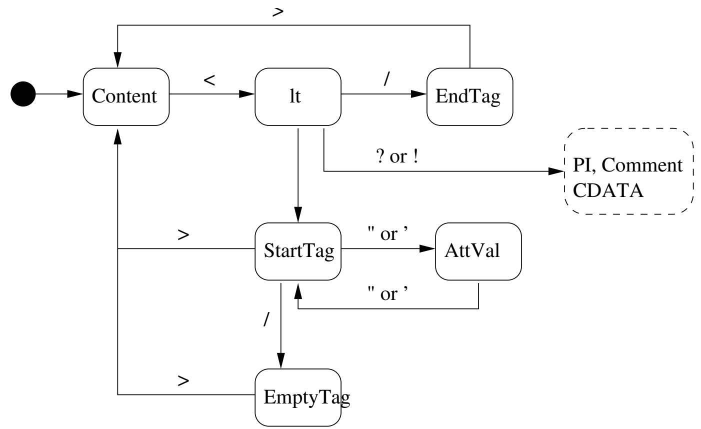

Fig. 3. This automaton accepts the syntax needed by preparsing. (To emphasize the major states, we omit the states for the PI, Comment, and CDATA productions by enclosing them in the dashed line box.)

In addition to the simplified syntax preparsing also benefits from omitting other well-formedness constraints. Usually in order to check the well-formedness constraints, a general XML parser will perform a number of additional comparisons, transformations, sorting, and buffering, all of which can result in significant performance bottlenecks. For instance, the fundamental well-formedness constraint is that the name in the end-tag of an element must match the name in the start-tag. To check this constraint, the general XML parser might push the start tag name onto a stack whenever a start tag is encountered, and pop the stack to match the name of the end tag. The preparser, however, treats the XML document as a sequence of unnamed open and close tag pairs. Therefore, it can merely increment the top pointer of the stack for any start tag, and decrement for any end tag. Finally, if the top pointer points to the bottom of the stack, the preparser considers the XML document to be correct without an expensive string comparison.

Another well-formedness constraint example is that the attribute name must not appear more than once in the same start-tag. To verify that, a full XML parser must perform an expensive uniqueness test, which is not required for preparsing.

Finally, preparsing obviously does not need to resolve namespace prefixes, since it completely ignores the tag name. However, a full XML parser supporting namespaces, requires expensive lookup operations to resolve namespace prefixes.

The only constraint the preparsing requires is that the open tag has to be paired with a close tag. A simple stack is adopted for this checking, and the skeleton nodes are generated as the result of pushing and popping of the stack.

Another important source of performance advantages of preparsing compared to full parsing is that the skeleton is much lighter-weight than the DOM structure. Thus, preparsing is able to generate the skeleton substantially faster than full XML parsing is able to generate the DOM. When compared to SAX, the preparser benefits from avoiding callbacks.

IV. Parallel Parsing

During the parallel parsing phase, we use the structural information in the skeleton to divide the document into chunks, each of which contains a forest of subtrees of the XML document. Each chunk is parsed by a thread. For any data parallel technique to be effective, load-balancing must be used to prevent idle threads. Ideally, we could divide the document into chunks such that there is one chunk for each thread and such that each chunk takes exactly the same amount of time to parse. Depend on when and how the partitioning is performed, we have two strategies: *static partitioning* and *dynamic partitioning*.

A. Static Partitioning

Naturally, we can statically partition a tree into several equally-sized subparts by using a graph partitioning tool (e.g., Metis [6]), which can divide the graph/tree into N equally-sized parts. The advantage of static partitioning is it can generate a very well-balanced load for every thread, thus leading to good parallelism.

However since the static partitioning occurs before the actual XML parsing, it knows little about the parsing context (e.g., namespace declarations). In other words, cuts made by the static partitioning will create following problems:

1) The characters of the XML document corresponding to the subgraph may no longer be contiguous. Metis will create connected subgraphs, but a connected subgraph of the logical tree structure does not necessarily correspond to a contiguous sequence of characters in the XML document. In order to parse the resulting characters, we must either reconstruct a contiguous sequence by memory copying, or modify the XML parser to handle non-contiguous character sequences, which may be challenging.
2) The namespace scope may be split between subgraphs, which means a namespace prefix may be used in one subgraph, but defined in another. These inter-chunk references will create strong memory and synchronization dependencies between threads, which will degrade performance.

The static partitioning strategy also suffers because the static partitioning algorithm must be executed sequentially before the parallel parsing, thus the performance gained by the parallelism will very easily be offset by the cost of the static partitioning algorithm, which usually is not trivial.

However for XML documents representing an array structure, such as

```
<data>
    <item>....</item>
    ...
    <item>....</item>
</data>
```

which are responsible for the bulk of most large XML documents, static partitioning is able to provide the best parallelism. That is because a linear array can easily be divided into equal-sized ranges (i.e., subgraphs) without an expensive graph-partitioning step. The division is based on the left to right order, so every range is contiguous in the XML document.

We have developed the *static PXP* algorithm, a simple static partitioning and parallel parsing algorithm capable of parsing XML document with array structures. This serves to provide a baseline against which we can compare more realistic techniques. Conveniently, we are able to leverage a function from libxml2 [1], which is a widely-used and efficient XML parsing library written in C, to perform the parsing.

```
xmlParseInNodeContext(xmlNodePtr node,
        const char * data,
        int datalen,
        int options,
        xmlNodePtr * lst)
```

This function can parse a "well-balanced chunk" of an XML document within the context (DTD, namespaces, etc.) of the given node. A well-balanced chunk is defined as any valid content allowed by the XML grammar. Since the regions generated by our static partitioning are well-balanced, we can use the above function to parse each chunk. Obviously any element range generated by static array partitioning is a well-balanced chunk. Then the static PXP algorithm consists of the following steps:

1) Construct a faked XML document in memory containing just an empty root element. by copying the open/close tag pair of the root element from the original XML document, Since we assume that the size of the root element is much smaller than the whole document, the cost of any memory operations used by this step are acceptable.
2) Call the libxml2 function `xmlParseMemory()` to parse the faked XML document, thus obtaining the root XML node. This node contains the namespace declarations required by its children, and will be treated as the context for the following parse of the ranges of the array.
3) The number of elements in each chunk is calculated by simply dividing the total number of elements in the array, which was calculated during the preparsing stage, by the number of available threads, so that every thread has a balanced work load. The start positions and data length of the chunk can be inferred from the location information of its first element and the last element.
4) Create a thread to parse each chunk in parallel. Each thread invokes `xmlParseInNodeContext()` to parse and build the DOM structure.
5) Finally the parsed results of each thread will be spliced back under the root node.

In summary, the static partitioning strategy is not really practical for XML documents with irregular tree structures, due to strong dependencies between the different processing steps. However for those XML documents containing an array, it provides an upper bound on the performance gain of parallel parsing, and is useful for evaluation of other parallel parsing

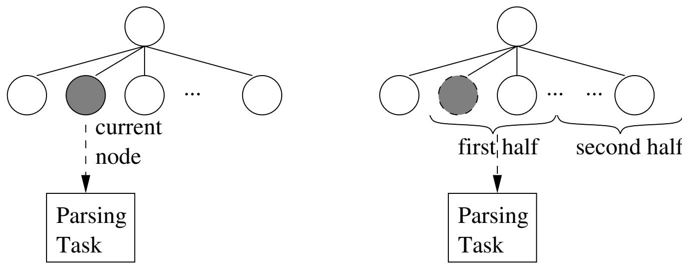

Fig. 4. The left diagram illustrates the general node splitting strategy. Each node becomes a subtask. The right diagram illustrates the split-in-half strategy. The nodes of the current parsing task are split in half, with the first half given to the requesting task, while the current task finishes the second half.

approaches as the guideline.

B. Dynamic Partitioning

In contrast with static partitioning, the dynamic partitioning strategy partitions the XML document and generates the subtasks during the actual XML parsing. After the preparser generates the skeleton, the tree structure is traversed in parallel to complete the parsing. Whenever a node is visited by a thread its corresponding serialization (start tag) will be parsed and the related DOM node will be built.

The parallel tree traversal is equivalent to a complete, parallel depth-first search (DFS) (in which the desired node is not found), which partitions the tree dynamically and searches for a specific goal in parallel using multiple threads.

After Rao [7], dynamic partitioning consists of two phases:

- Task partitioning
- Subtask distribution

Task partitioning refers to how a thread splits its current task into subtasks when another thread needs work. A common strategy is *node splitting* [8], in which each of the n nodes spawned by a node in a tree are themselves given away as individual subtasks. However for parallel XML parsing, node splitting may generate too many small tasks since most of nodes represents a single leaf element in the XML document, thus increasing the communications cost.

Since XML is a depth-first, left-to-right serialization of a tree, a sequence of sibling element nodes in the skeleton corresponds to a contiguous chunk of the XML document. Therefore, if each parsing task covers a sequence of sibling element nodes, this will maximize the size of each workload, with little communication cost. In dynamic partitioning, we adopt a simple but effective policy of splitting the workload in half, as shown in Figure 4. That is, the running thread splits the unparsed siblings of the current element node into two halves in the left-to-right order, whenever the partitioning is requested.

Subtask distribution refers to how and when subtasks are distributed for the donator thread to the requester thread. If work splitting is performed only when an idle processor requests for work, it is called *requester initiated subtask distribution*. In contrast if the generation of subtasks is independent of the work requests from idle processors the scheme is referred to as a *donator initiated subtask distribution* scheme. For parallel XML parsing, we desire that the parsing thread will parse as much XML data as possible without any interruption, unless other threads are idle and asking for tasks, so as to achieve a better performance. Also, any thread can be the donator or the requester. We adopt the requester initiated subtask distribution as the partition strategy in the PXP.

To implement parallel parsing with dynamic partitioning, we again use libxml2. Since dynamic partitioning requires the parser do the task partitioning and subtask generation during the parsing, however, we cannot simply apply the libxml2 `xmlParseInNodeContext()` function as in the static partitioning scheme. Instead, we need to change the `xmlParseInNodeContext()`[2] source code to integrate the dynamic partitioning and generation logic into the original parsing code. The modified algorithm is called *dynamic PXP* and its basic steps are:

1) Create multiple threads, and assign the root node of skeleton as the initial parsing task to the first thread. Other threads are idle.
2) When a thread is idle, it posts its request on an request queue, and waits for the request be filled by some donator thread.
3) Every thread, once it begins parsing, parses normally as libxml2 does, except when an open tag is being parsed. At that time, it checks the request queue for threads that need work. If such a requester thread exists, the thread splits the current workload (i.e., the unparsed sibling nodes) into two regions. The first half is donated to the requester thread, and the thread resumes parsing at the beginning location of the second half region. Since every skeleton node records the number of its children elements, as well as its location information, it is easy to figure out the begin location and data length of the subtask. Also to avoid excessively small tasks, the user can set a threshold to prevent task partitioning if the remaining work is less than the threshold.
4) Once the requester thread obtains the parsing task, it begins the parsing at the beginning location of the donated subtask. Due to the dynamic nature, the donator is able to pass its current parsing context (e.g., the namespace declarations) to the requester as the requester's initial parsing context, which will in turn makes a clone of the parsing context for itself before parsing to avoid the synchronization cost. Also the donator will create a dummy node as the "placeholder" for the parsing task, the subtrees generated by the requester will be inserted under the placeholder and once the parsing task is completed, the placeholder will be spliced within the entire DOM tree.
5) This process continues until all threads are idle.

In summary, dynamic partitioning load-balances during the parsing, and it can be applied to any irregular tree structure

[2] In fact the actual modified function is xmlParseContent(), which is invoked by xmlParseInNodeContext() to parse the XML content.

without the need of the extra partitioning algorithm. However, the dynamic nature incurs a synchronization and communication cost among the threads, which is not needed by the static partitioning scheme.

V. MEASUREMENT

We first performed experiments to measure the performance of the preparsing, and then performed experiments to measure the performance improvement and the scalability of the parallel XML parsing (static and dynamic partition) algorithm over the different XML documents. The experiments are running on a Linux 2.6.9 machine which has two 2 dual-core AMD Opteron processors and 4GB of RAM. Every test is run five 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 g++ 3.4.5 with the option -O3. 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 allocator is intended simply to avoid lock contention. A production allocator would use other techniques to reduce lock contention. One possibility is to simply use a two-stage technique, where large chunks of memory are obtained from a global pool, and then managed individually for each thread in a thread-local pool.

A. Preparsing Performance Measurement

Preparsing generates the skeleton which is necessary for PXP. However, this is an additional step compared to normal XML parsing, which, unfortunately, also needs to be performed sequentially before the actual parallel parsing. Thus, to help determine whether or not this cost is acceptable, and understand the overall PXP performance, we measured preparsing time and also compared it to full libxml2 parsing.

Since preparsing linearly traverses the XML document without backtracking or other bookkeeping, the time complexity is linear in the size of the document, and independent of the structural complexity of the document. We thus designed the preparsing test to maximize the performance of a full sequential parser, and used a simple array of elements which varied in size. The test document is shown in the Appendix.

First, we varied elements in the array to increase document size. Then for the comparison, we measured the costs of two widely-used parsing methods: building DOM with libxml2, and parsing with the SAX implementation in libxml2. In addition, for the libxml2 SAX implementation, we used empty callback routines. Thus, libxml2 SAX is expected to be extremely fast. The results are shown in Figure 5.

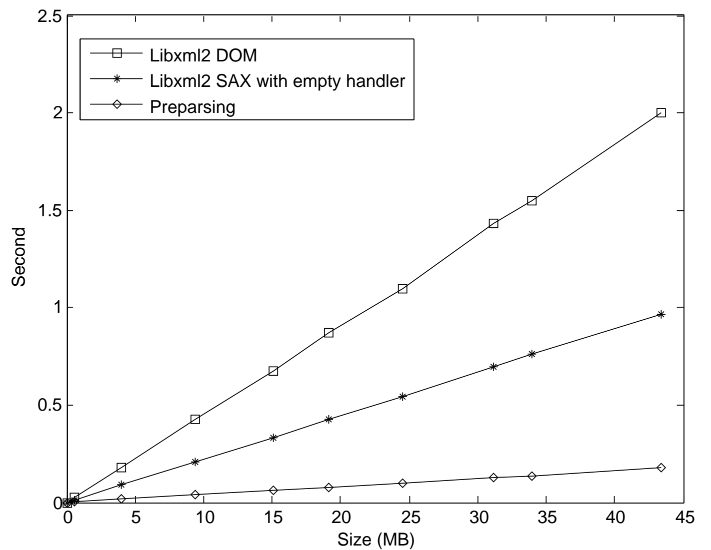

Fig. 5. Performance comparison of preparsing.

According to Figure 5, we see that preparsing is nearly 12 times faster than sequential parsing with libxml2 to build DOM. Even for libxml2 SAX parsing, preparsing is over 6 times faster. Even though the preparser builds a tree, the tree is simple and does not require expensive memory management.

These results show that even the preparsing does not occupy much time, and the time left for actual parallel parsing is enough to result in significant speedup.

B. Parallel XML Parsing Performance Measurement

Speedup measures how well a parallel algorithm scales, and is important for evaluating the efficiency of parallel algorithms. It is calculated by dividing the sequential time by the parallel time. For our experiments, the sequential time refers to the time needed by libxml2 `xmlParseInNodeContext()` to parse the whole XML document. To be consistent, static PXP, dynamic PXP, and the sequential program are all configured to use the thread-optimized memory allocator. Each program is run five times and the timing result of the first time is discarded to warm the cache.

We first measure the upper bound of the speedup that the PXP algorithms could achieve. To do that, we select a big XML document used in the previous preparsing experiment as the parsing test document. The array in the XML document has around 50,000 elements and every element includes up to 28 attributes and the size of the file is 35 MB. Since the test document just contains a simple array structure, we are able to apply both static PXP and dynamic PXP algorithms on it. Figure 6 shows how the static/dynamic PXP algorithms scales with the number of threads when parsing this test document. The diagonal dashed line shows the theoretical ideal speedup. From the graph we can see that when the threads number is one or two the speedups of the PXP is sublinear, but if we subtract the preparsing time from the total time the speedups of static PXP is close to linear. This indicates the preparsing

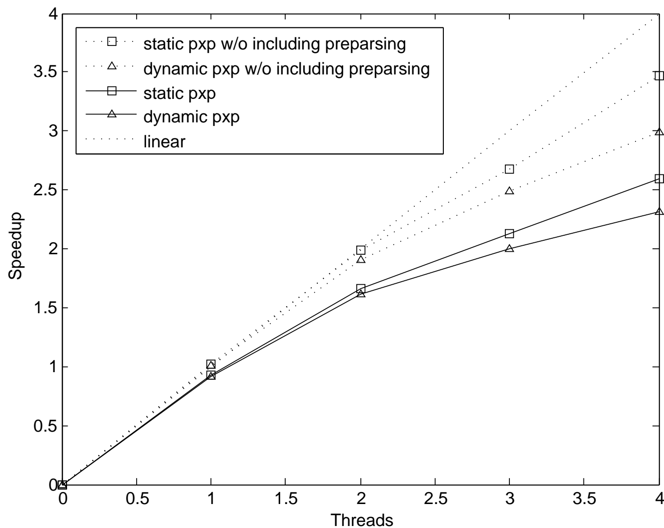

Fig. 6. This graph shows the upper bound of the speedup of the PXP algorithms for up to four threads, when used to parse a big XML document which only contains an array structure.

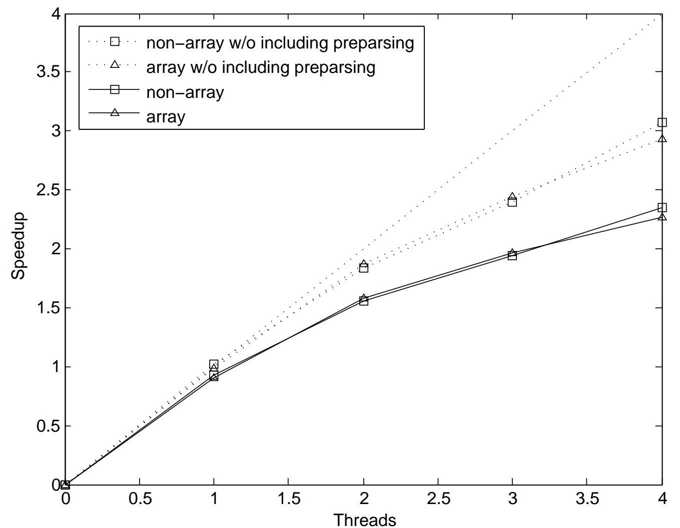

Fig. 7. This graph shows the speedup of the **dynamic PXP** for up to four threads, when used to parse two same-size XML documents, one with irregular tree shape and one with regular array shape.

dominates the overhead, and the static PXP presents the upper bound of the parallel performance.

The speedups of dynamic PXP are slightly lower than the ones of the static PXP, which indicates the cost of communication and synchronization starts to be a factor, but is relatively minor. When the threads number is increased the speedup of the PXP (dynamic or static) become less, that is because when the work load of every thread decreases, the overhead of the preparsing becomes more significant than before. Also the dynamic PXP obtains less speedup than the static PXP due to the increasing communication cost. Furthermore, even the speedup of the static PXP omitting the preparsing cost starts to drop away from the theoretical limit. We speculate that shared memory or cache conflicts are playing a role here.

Unlike the static PXP, dynamic PXP is able to parse the XML documents with any tree shape. So to further study the performance improvement of dynamic PXP, we modified the previous XML document with big array structure to be irregular tree shape, which consists of a five top-level elements under the root, each with a randomly chosen number of children. Each of these children is an element from the array of the first test, and so the total number of these child elements in the modified document is same as the one of the original document.

We compare the dynamic PXP on this modified XML document against the dynamic PXP on the original array XML document. This comparison can show how the dynamic PXP scales for the XML documents with irregular shape or regular shape. From the results shown in Figure 7 we can see there is little difference between two XML documents, which imply that dynamic PXP (and our task partitioning of dividing the remaining work in half) is able to effectively handle the large XML file with irregular shape.

These tests did not actually further parse the element contents. In a typed parsing scenario, where schema or other information can be used to interpret the element content, we would obtain even better scalability. For example, if we are parsing a large array of doubles including the ASCII-to-double conversion, each thread has an increased workload relative to the preparsing stage and other overheads, and thus speedup would be improved.

VI. Related work

As mentioned earlier, parallel XML parsing can essentially be viewed as a particular application of the graph partitioning [6] and parallel graph search algorithms [7]. But the document parsing and DOM building introduces some new issues, such as preparsing, namespace reference, and so on, which are not addressed by those general parallel algorithms.

There are a number of approaches trying to address the performance bottleneck of XML parsing. The typical software solutions include the pull-based parsing [9], lazy parsing [10] and schema-specific parsing [11], [12], [13]. Pull-based XML parsing is driven by the user, and thus provides flexible performance by allowing the user to build only the parts of the data model that are actually needed by the application. 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 the extra penalty will be paid. Most closely related to our work in this paper is lazy parsing because it also need a skeleton-similar structure of the XML document for the lazy evaluation. That is firstly 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 the 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[14], [15] also are promising, particularly in the industrial arena. But by our best knowledge, there is no such work leveraging the data-parallelism model as PXP.

VII. Conclusion and Future Work

In this paper, we have described our approach to parallel XML parsing, and shown that it performs well for up to four cores. An efficient parallel XML parsing scheme needs an effective data decomposition method, which implies a better understanding of the tree structure of the XML document. Preparsing is designed to extract the minimal tree structure (i.e., skeleton) from the XML document as quickly as possible. The key to the high performance of the preparsing is its highly simplified syntax as well as the obviation of full well-formedness constraints checking. Aided by the skeleton, the algorithm can partition the XML document into chunks and parse them in parallel. Depending upon when the document is partitioned, we have the static PXP and dynamic PXP algorithms. The former is only for the XML documents with array structures and can give the best case benefit of parallelism, while the latter is appliable to any structures, but with some communication and synchronization cost. Our experiments shows the preparsing is much faster than full XML parsing (either SAX or DOM), and based on it the parallel parsing algorithms can speedup the parsing and DOM building significantly and scales well. Since the preparsing becomes the bottleneck as the number of threads increase, our future work will investigate the feasibility of the parallelism between the preparsing and real parsing. Also new approaches for very large XML documents will be studied under the shared memory model.

Acknowledgment

We would like to thank professor Randall Bramley for his insightful suggestion and help on the graph partition and Metis. We also thank for Zongde Liu and Srinath Perera for the useful comment and discussion.

References

[1] D. Veillard, "Libxml2 project web page," http://xmlsoft.org/, 2004.
[2] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, 2005.
[3] W3C, "Xml information set (second edition)," http://www.w3.org/TR/xml-infoset/, 2003.
[4] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2000.
[5] W3C, "Extensible Markup Language (XML) 1.0 (Third Edition)," http://www.w3.org/TR/2004/REC-xml-20040204/, 2004.
[6] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Supercomputing*, 1996.
[7] V. N. Rao and V. Kumar, "Parallel depth first search. part i. implementation," *Int. J. Parallel Program.*, vol. 16, no. 6, pp. 479–499, 1987.
[8] V. Kumar and V. N. Rao, "Parallel depth first search. part ii. analysis," *Int. J. Parallel Program.*, vol. 16, no. 6, pp. 501–519, 1987.
[9] A. Slominski, "Xml pull paring," http://http://www.xmlpull.org/, 2004.
[10] M. L. Noga, S. Schott, and W. Lowe, "Lazy xml processing," in *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, 2002.
[11] 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.
[12] W. M. Lowe, M. L. Noga, and T. S. Gaul, "Foundations of fast communication via xml," *Ann. Softw. Eng.*, vol. 13, no. 1-4, 2002.
[13] R. van Engelen, "Constructing finite state automata for high performance xml web services," in *Proceedings of the International Symposium on Web Services(ISWS)*, 2004.
[14] 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.
[15] "Datapower," http://www.datapower.com/.

Appendix

Structure of the XML document ns_att_test.xml

```
<xml xmlns:rs='urn:schemas-microsoft-com:rowset'
     xmlns:z='#RowsetSchema'
     xmlns:tb0='table0' xmlns:tb1='table1'
     xmlns:tb2='table2' xmlns:tb3='table3'>
     <z:row tb1:PRODUCT=... tb0:CCIDATE=...
            tb0:CLASS=... tb2:ADNUMBER=...
            tb0:PRODUCTIONCATEGORYID_FK=...
            tb3:ADVERTISERACCOUNT=...
            tb1:YPOSITION=... tb2:CHEIGHT=...
            tb2:CWIDTH=... tb2:MHEIGHT=...
            tb2:MWIDTH=... tb2:BHEIGHT=...
            tb2:BWIDTH=... tb3:SALESPERSONNUMBER=...
            tb3:SALESPERSONNAME=...
            tb1:PAGENAME=... tb1:PAGENUMBER=...
            tb2:BOOKEDCOLOURINFO=... tb1:EDITION=...
            tb1:MOUNTINGCOMMENT=... tb1:TSNLSALESSYSTEM=...
            tb1:TSNLCLASSID_FK=... tb1:TSNLSUBCLASS=...
            tb1:TSNLACTUALDEPTH=... tb1:XPOSITION=...
            tb0:TSNLCEESRECORDTYPEID_FK=...
            tb0:PRODUCTZONE=... ROWID=.../>
     <z:row ... />
     <z:row ... />
     ...
</xml>
```