Speculative \( p \)-DFAs for Parallel XML Parsing

Ying Zhang, Yinfei Pan, Kenneth Chiu
Department of Computer Science, State University of New York - Binghamton
P.O. Box 6000, Binghampton, NY 13902, US
\{yzhang8, ypan3, kchiu\}@cs.binghamton.edu

Abstract

XML has seen wide acceptance in a number of application domains, and contributed to the success of wide-scale grid and scientific computing environments. Performance, however, is still an issue, and limits adoption under some situations where it might otherwise be able to provide significant interoperability, flexibility, and extensibility. As CPUs increasingly have multiple cores, parallel XML parsing can help to address this concern. This paper explores the use of speculation to improve the performance of parallel XML parsing. Building on previous work, we use an initial preparsing stage to build a sketch of the document which we called the skeleton. This skeleton contains enough information so that we can then proceed to do the full parse in parallel using unmodified libxml2. The preparsing itself is parallelized using product machines which we call \( p \)-DFAs. During execution, unlikely possibilities are discarded in favor of more likely ones. Statistics are gathered to decide which possibilities are not likely. The results show good performance and scalability on both a 30 CPU Sun E6500 machine running Solaris and a Linux machine with two Intel Xeon L5320 CPUs for a total of 8 physical cores.

I. Introduction

The wide adoption of XML as the language of choice for large-scale distributed systems has facilitated the development of e-Science and grid computing by providing a flexible, interoperable, extensible lingua franca for describing and representing data. Applications range from financial services [17] to molecular dynamics [13]. Its performance, however, is considered too slow to be useful in many situations, as shown by a number of previous works [6], [15], [19], [24]. That this is a problem is also shown by the fact that there are a large number of efforts to improve the performance of XML. By limiting the scope where XML is used, this also prevents XML from providing benefits where it might otherwise be able to do so. To address these doubts, a number of approaches have been developed. Binary-based encodings of XML can be more efficient [5] [44], differential techniques [1] [38] to cache results from similar messages for reuse can also be helpful, and hardware acceleration to speed XML processing [16] can be used for some applications. Effort also made to optimize SOAP performance [18], [48] to speedup transport.

The growing prevalence of multicore CPUs suggests another approach for improving XML performance: parallelism. For example, a separate core can be used to service each request, including the XML parsing and processing. Under some situations, however, it may be useful to speed up the processing of a single sequence of XML messages. For example, a scientific code that reads in initial conditions from a large XML document may not be able to start executing until all the data has been read in. Or, a high-speed scientific workflow might rely on fast processing of a single stream. In these cases, assigning multiple cores to parse the incoming request may be beneficial, especially if the message is large. Note that our technique does not require a large, multicore server to be useful, since multicore CPUs are found on laptops and desktops, and without parallel parsing, these cores would otherwise be unutilized during XML input.

Current XML parsers are sequential, and thus do not take advantage of multicore CPUs. These current parsers are generally not trivially parallelizable. The difficulty of parallelizing XML is covered in Section II, and in our previous papers [20] [27]. These show how an initial parsing pass can be used to construct an outline of the document, which can then be used to guide a full parallel parse using unmodified libxml2 [42], which provides an API sufficient to parse well-balanced XML fragments. However, the major problem here is that the parsing pass is sequential, and thus becomes a bottleneck when scaling to a large number of cores. This parse could be parallelized by dividing the XML document into sized chunks, and parsing each chunk in parallel with a separate core. (We use the term chunk here to refer to any contiguous sequence of characters.) The preparsing is inherently sequential, however, because the state of the XML parser when reading a given character depends potentially on all preceding characters. Thus, chunks cannot be parsed independently without ambiguity.

Earlier, we used a construct called the meta-DFA to parallelize the parsing pass [31]. We also modified this to be more efficient in [30] by using sets. The previous techniques computed for all possibilities \( a \text{ priori} \), because each core cannot know exactly which possibility will hold before it begins execution. In this paper, we extend our work by examining how speculation can be used to improve performance, using a unified approach we called
a p-DFA. By not computing low-probability possibilities, average-case performance can be improved. We investigate speculative behavior using statistics, and show results on a number of document types. Possibilities that have a low probability are not computed in the initial preparsing pass. The results on a 30-CPU Sun E6500 machine and an 8-core Linux machine with Intel Xeon L5320 CPUs demonstrate good performance and strong scalability.

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.

II. Overview of PXP and Preparsing

A. PXP

In the PXP (Parallel XML Parsing) approach [20], [27], an initial pass is first used to determine the basic structure of an XML document. This structure is then used to divide the XML document into fragments such that the divisions between the fragments occur at well-defined points in the XML grammar. This provides enough context so that each fragment can be parsed starting from an unambiguous state. This simple structure, specifically designed for XML data decomposition, is called the skeleton of the XML document. To distinguish it 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 the basic structure of the XML document is known, the PXP approach divides the document into balanced fragments and then launches multiple threads to fully parse the fragments in parallel using libxml2, using functions provided by libxml2 that allow the parsing of XML fragments. By leveraging libxml2, we are able to provide output that is completely compatible with libxml2, and created using a full-fledged, high-quality parser.

B. Preparsing

1) Skeleton: As described in [20], [27], many of the syntactic units defined by the XML specification [45], such as attributes, namespaces, and even the tag name are not considered in the preparse. Neither does the preparsing need to verify any well-formedness constraints, since these can be verified later during the full parse. The skeleton produced by the parsing thus treats the XML document as simply a sequence of unnamed start- and end-tag pairs. This simplification does not limit the XML accepted by the PXP approach, since all other XML features are handled in the subsequent full parse.

Therefore, the skeleton is stored in memory as an array of items containing information about each XML element. There is one item for each element, and the item contains the start and end position of the element, defined by the first character position of the start-tag and the last character position of the end-tag, respectively. The order of items in the skeleton corresponds to a depth-first ordering of the represented elements.

2) Sequential Preparsing: The most straightforward way to prepare is to use a sequential pass using an automaton with state transitions [20], [27]. Such a sequential pass will limit our speedup per Andahl's Law, so we have investigated parallelizing the preparsing. With the work based on the running of a state machine, preparsing is inherently a pure sequential stage, simply because the true state at any character of the input sequence is undeterministic if the state machine does not run through all its preceding characters. Therefore, though the preparsing is an order of magnitude faster than the full parsing, the effects of Andahl's law [2] limits the scalability of the PXP technique to a modest number of processors.

3) Data Parallel XML Preparing: The work of the XML preparser is to extract the element positions inside a large XML document in depth first order during a fast traversal of the XML character stream. Thus, to parallelize this stage, one approach is data parallelism. Here, the data, in this case the XML document, is divided into some number of chunks, which are then all preparsed in parallel by separate threads, one for each chunk. (Here, we use the term chunk to refer to a sequence of characters with arbitrary boundaries that are to be preparsed, and the term fragment to refer to start-tag/end-tag balanced fragments of XML that have boundaries at well-defined points. The input to data-parallel preparsing is chunks, and the output is fragments.) As the chunks are finished, the results are merged back together to form the complete result. The difficulty with this scheme, however, is that dividing the document into chunks with equal numbers of characters would create chunks beginning at arbitrary locations within the XML document. Thus, all chunks except the first would begin at a character of unknown syntactic role, such as in a tag name, an attribute name or value, element content, etc. The parser would not know the state at which to begin the chunk.

To address this, there exists some ad hoc approaches, such as guessing at the syntactic role. This will be problematic, however, since in most cases the beginning of the chunk is unpredictable. Another way is use multiple processors to prepare the same chunk. Each processor will start at a different syntactic role. Finally, one processor will obtain the correct result. This approach, however, obviously does not utilize processors well.
III. A Model for XML Parsing

We now describe our approach for obtaining data parallelism. This approach is similar to parallel finite automata [35], union automata and product machines, but applies them in a particular way to XML. We first define a model of XML parsing that includes the preparser. We assumed that such kind of XML parsing can be based on a deterministic, finite state automata (DFA) with actions defined on some state transitions. At each state, a number of transitions can be made depending on the next input character. Attached to each transition is an action of some kind. We did not define what the action actually does in the model itself. Rather, each specific XML parser will have a different set of actions, and may have different implementations of those actions. For example, actions may manipulate DOM objects, issue SAX events, etc. In our model, however, actions are abstract.

We modeled an XML parser as a DFA defined by

$$M = (Q, \delta, \Sigma, A, t, q_0)$$

where $Q$ is the set of non-dead normal states, $\Sigma$ is an alphabet of possible input characters, $A$ is a set of actions, $t$ is the transition function, $q_0$ is the start state, and $\delta$ is the dead state. If the machine is triggered in state $\delta$, the input XML document is considered as illegal input to the XML parser. The transition function maps from the current state and input character to the next state and an action:

$$t : Q \times \Sigma \rightarrow (Q \cup \delta) \times A$$

When $M$ executes a particular transition, the corresponding action is also executed. For example, Figure 3 shows a simple DFA with actions $a_0$ through $a_4$. On a $<$ character, it would transition from state 0 to state 1, and execute the action $a_0$.

For convenience, we also define the function $t_{\text{state}}$ which extracts the next state from the transition. Thus $t_{\text{state}}(p, c) = q$ iff $t(p, c) = (q, a)$ where $p \in Q$, $q \in Q \cup \delta$, $c \in \Sigma$, and $a \in A$. Similarly, the function $t_{\text{act}}$ extracts just the action from the transition, so $t_{\text{act}}(p, c) = a$ iff $t(p, c) = (q, a)$.

Following this parsing model, we model the preparser automaton as shown in Figure 2. For clarity, we omit states to handle PI, COMMENT, and CDATA from this diagram. Those add an additional 12 states.

IV. The $p$-DFA

A. Vector-Based $p$-DFA Definition

The fundamental problem with data-parallel pre parsing is that the correct state of the parser at the beginning of each chunk cannot be determined without processing all preceding input characters. Thus the main idea of the $p$-DFA approach was to consider all possibilities, and generate the pre parsing results for all possible initial states as a chunk is pre parsed. Then the results on each chunk will be merged together for a completed skeleton.

The merge is straightforward. Since the start state of the first chunk is just the start state of the sequential pre parse, the correct state at the end of the first chunk is unambiguous. This state is in fact the correct start state of the second chunk, and the correct pre parsing result of the second chunk will be selected. This in turn determines the correct start state of the third chunk, and so on and so forth till the pre parsing result has been completely determined and reconstructed.

We now first describe the vector-based $p$-DFA, because it is conceptually easier to understand. Also, for ease of explanation, the states, transitions and actions of the $p$-DFA are called $p$-states, $p$-transitions, and $p$-actions, respectively. Further details are in [31].

The vector-based $p$-DFA $M'$ is constructed from the original DFA $M$. $M'$ essentially pursues simultaneously all guesses of the true state at the beginning of a chunk, so the start state of $M'$ is simply a vector of all states in $Q$ of $M$. Each state of $M'$ is also a vector of states and is an element from the Cartesian product of $Q \cup \delta$. The transition function of $M'$ runs multiple instances of the original DFA in parallel via inner-$p$-DFAs. An inner-$p$-DFA goes from state $q_i$ to $q_j$ on character $c$ and takes the $a_k$ action only if the original DFA also had $t_{\text{state}}(q_i, c) = q_j$ and $t_{\text{act}}(q_i, c) = a_k$. Figure 4 shows a vector-based $p$-DFA constructed from the simple example DFA in Figure 3. In this figure, we write $a_x \rightarrow i$, which means action $a_x$ should be taken on inner-DFA $i$ during the current $p$-transition. The execution of the vector-based $p$-DFA and its relationship to inner-DFAs is shown in Figure 5.

With this definition, $M'$ is in fact also a DFA—the simultaneity is strictly conceptual—and thus the $M'$ is still executed by a single processor. In other words, the parallelism is due to multiple chunks being pre parsed in parallel, not a single $M'$ execution using multiple cores. Each chunk is pre parsed using a product machine $M'$, but there is no actual parallelism when executing that machine.
In the original DFA, the actions operated on some kind of overall parser result. The p-DFA, however, is pursuing multiple possibilities simultaneously, and thus an instance of the p-DFA must maintain multiple versions of this originally single result. As to the vector-based p-DFA, each original action must now be executed within a separate context, corresponding to each inner-DFA.

With the the vector-based p-DFA constructed from the preparser DFA in Figure 2, the two actions START and END will now be taken on each context as START(i) and END(i), where i is the index of the context. Such actions will now only affect the result of its corresponding context.

B. Actions on the Vector-Based p-DFA

In the original DFA, the actions operated on some kind of overall parser result. The p-DFA, however, is pursuing multiple possibilities simultaneously, and thus an instance of the p-DFA must maintain multiple versions of this originally single result. As to the vector-based p-DFA, each original action must now be executed within a separate context, corresponding to each inner-DFA.

V. Set-Based p-DFA

After applying the vector-based p-DFA generation algorithm on the preparser DFA, the constructed p-DFA has 611 p-states and 41023 lines of source code. Since the automaton will make a state transition on each input character, large code sizes can lead to a preponderance of distant jumps which can impact run time of the automaton. This then will pose a serious impact on the performance.

Referring to Section IV-A, we see that we are using a vector of states to represent each p-state. If the total number of DFA states N = |Q|, the vector size of each p-state is strictly N, and the state set of p-DFA is the Cartesian product (Q ∪ δ)N. This means that there will be (N + 1)N number of possible p-states. Inside the vector of most p-states, however, there are duplicate DFA state numbers as vector elements. Such duplicates can actually be eliminated, since the execution of the inner-DFAs corresponding to these duplicate DFA states is identical. Also, with the vector-based p-DFA definition, the order of state numbers inside the p-state vector is significant, for example, [1,2] and [2,1] will count as two different p-states. However, the execution path of inner-DFAs of these two p-states will be the same, but merely swapped. Thus, they actually can be treated as one p-state. In addition, the dead DFA state δ inside the p-state vector just explicitly indicates that the running of corresponding inner-DFAs are trapped into wrong guesses, and thus the contexts on these inner-DFAs are already dead. This is obviously redundant since we do not need to know explicitly which context is dead.

Therefore, we can use sets instead of vectors to represent p-states, and use the power set instead of the Cartesian product on the states of the original DFA as possible p-states. We name such modified p-DFA the set-based p-DFA M'' [30]. Its start state is simply the set of all states of the DFA, still representing the fact that M'' is starting simultaneously from all states of M. The transition function here will implicitly maintain multiple inner-DFAs. Inner-DFAs may merge together if they transition to the same state after a p-transition. The action on the p-transition here can be represented by a set of pairs that associate state with action like (q, a_i), if there is a transition in M out of state q of the current p-state set on accepting the current input c, and has t_{act}(q, c) = a_i. Details on how transitions are taken during the execution are explained in section V-A. Figure 6 shows the set-based p-DFA that has been constructed from the DFA in Figure 3. Comparing to the vector-based p-DFA constructed in Figure 4, it is much simpler. Further details are in [30].

The number of all possible states in the set-based p-DFA is potentially 2^N, but the vast majority of the possible states in the set-based p-DFA are never actually created. This is analogous to how the well-known subset construction algorithm for converting NFAs to DFAs could result in a state explosion, but in practice rarely does so, and is thus still of great practical use. The same p-DFA is used for all XML documents, since it is a created from the preparser DFA, and not from a document-specific DFA. When we actually do generate the p-DFA, we find that there is no
The action to be taken when transition out of a state should be based on the total size of the XML document, but rather on the DOM case. For streaming XML, the chunk size is not set at runtime, but rather the DOM case. For streaming XML, the chunk size is not set at runtime, but rather the element order is not sensitive in the set. The execution of a inner-DFA thus cannot be explicitly tracked by a fixed index position within the p-state set. As shown in Figure 7, during the execution of the set-based p-DFA, as an inner-DFA follows the state transitions, it will not reside on a fixed position in the p-state set. Thus, several inner-DFAs may merge together on the way and share the subsequent transition path.

Also, since there is no static path index for a inner-DFA, the only way to maintain a context on a inner-DFA is to associate the inner-DFA and its corresponding context with a state number on-the-fly during execution. One state may be associated with multiple inner-DFAs and their contexts if they are currently in this state.

Once a state is associated with multiple inner-DFAs, the transition out of this state in fact indicates that all the associated inner-DFAs should take this transition. So if an action exists on this transition, all the associated inner-DFAs currently in this state should take this action.

Regarding our design, we maintain a context set associated with each state. This can be expressed in Figure 7. Initially, each context set just has one context number which is simply the inner-DFA number. Then, after the set-based p-DFA accepting character C_0, inner-DFA_0, inner-DFA_1, and inner-DFA_2 merge at state q_1, the context set associated with state q_1 will then be changed to \{0, 1, 2\}, and the context sets associated with state q_0 and q_2 will be changed to empty sets. After the set-based p-DFA accepting character C_1, inner-DFA_3 merges with the other inner-DFAs at state q_1, the context set associated with state q_1 will then be changed to \{0, 1, 2, 3\}, and the context set associated with state q_3 will be changed to empty set. After the set-based p-DFA accepting character C_{m-1}, all the inner-DFAs will transition to state q_2, the context set of state q_2 will then be changed to \{0, 1, 2, 3\}, and the context set of state q_1 will be empty.

To determine the actions that need to be taken on the set-based p-DFA, we know from its definition that the action is associated with the original DFA states within the p-state set. However, since a set of contexts is associated with each state, we thus need to apply the action to all the contexts within the context set. Also, since we need to associate contexts with state on the fly, when state changes after a transition, the association also need to be changed to the destination state of the transition. Therefore, we add a CHANGE_STATE(i, j) action to fulfill this requirement. Here i and j means that state is changed from i to j, and this action merely does the contexts association operations.

Our p-DFA concept is not limited to DOM [30], but we have only applied the speculation described next to the DOM case. For streaming XML, the chunk size is not based on the total size of the XML document, but rather on a reasonable buffer size. Furthermore, we cannot leverage libxml2 due to reasons explained in [28] [29].

VI. Speculative Parsing

During our implementation, we found that at the end of most chunks there often exists many live contexts. The true context, however, is just one of them. If we maintain all the contexts, the work of taking actions and generating
skeletons will be incurred on those contexts that will finally turn out to be incorrect contexts. This is somewhat costly. In theory, we cannot leave out any contexts if the coming XML document is unpredictable. This is because if we do not know beforehand what kind of XML document need to be parsed, we do not know the probability at which true state each chunk may start when we divide the coming XML document into equal sized chunks.

However, in some applications, such as scientific computing, e-commerce, and distributed database, the XML documents are similar to each other. In other words, if we prepare them with the preparing DFA, the frequency of occurrence of the states of the preparing DFA will be similar from document to document within a particular application. If, in some applications, we know beforehand this is the case, we can randomly sample documents from the coming XML documents and measure its probability on each preparse state. Then, on pre parsing each chunk, we can leave out some contexts’ work that have low probability of being needed. Once we select the contexts to maintain, we can use their corresponding probabilities to calculate the possibility of successful speculation on each chunk. On the other hand, if we require some minimum success rate on speculation, we also can select contexts to speculate on each chunk accordingly.

More specifically, suppose one application transmits XML documents of some type. At the beginning, we just randomly select some of the coming XML documents and use the preparing DFA to perform a sequential pre parse on it. We measure the frequency of each preparing DFA state while preparing this DFA. We then sort the frequencies from the highest to the lowest and use them to infer probabilities, obtaining

\[ P(i_0) \geq P(i_1) \geq ... \geq P(i_{N-1}) \]

as the probability that at any character, the prep parser is in state \( i_0, i_1, ..., i_{N-1} \). Then, we can start to speculate on contexts for parallel pre parsing on the following documents. We set a probability value \( P_c \), sets minimum desired probability that the true context for all chunks is one of the selected contexts. In other words, \( P_c \) is the desired probability that we do not need to re-preparse any chunk due to speculating on the wrong contexts. We call this the confidence level. Suppose now we need to do parallel preparing on \( M \) chunks (\( M > 1 \)). Except for the first chunk which will always start at Context_0, all the other chunks will have the same probabilities of having the true context in the speculated contexts, so this equation is established:

\[ P_c = P_{\text{least}}^{M-1}. \]

Here, \( P_{\text{least}} \) means that at least this probability will be required for the true context being selected for each chunk, so that our confidence rate is obtained. We then seek to make this inequality hold with the minimum \( k \) value:

\[ P_{\text{least}} \leq P(i_0) + P(i_1) + \ldots + P(i_k). \]

Here, \( k \leq N-1 \). After this step, contexts start at state \( i_0, i_1, ..., i_k \) will be the selected contexts for speculation on each chunk. And such selection will ensure that our the confidence level \( P_c \) is obtained.

In order to clearly see the effects of speculation, we implement the speculation on the set-based \( p \)-DFA since it has much better performance and scales well. Once the selected speculation contexts are known, when starting to prepare a chunk, we just initiate an empty context set to associate with those start states whose corresponding contexts are not selected for speculation. This will make the actions not taken and thus have no cost for these contexts.

VII. Results

The overall architecture is shown in Figure 1. Since our work uses libxml2 to perform the full parse, our output is completely compatible with libxml2, and thus we compare our performance to sequential libxml2. Libxml2 is a popular, full-fledged XML parser. For some idea of how our parser would compare against other parsers, results against other parsers can be extrapolated based on comparisons of libxml2 against other parsers [15]. Note that binary encodings of XML will generally be faster, but such parsers can no longer be used with standard textual XML. We currently do not handle DTDs in the prepare. XML Schema-based validation is handled, however, by the full parse using libxml2’s functionality.

Our experiments were conducted on two different machines. One is a Sun E6500 running Solaris 10 with 30 400 MHz US-II processors. We call it the Solaris machine. The other is a 8-core Linux machine with two Intel Xeon L5320 CPUs at 1.86 GHz. We call it the Linux machine. Each test was run ten times and the first measurement was discarded, so as to measure performance with the file data already cached. The remaining measurements were averaged. The programs were compiled with g++ 4.0 with the option -O3, and with libxml2 2.6.16 library.

Testing with file data already cached was done because our focus here is on the parallelization of the parsing, not on the file I/O. If read from disk, the impact of the parallelization may be less, but it would depend on the exact I/O hardware. A typical RAID system might deliver 150 MB/s, which is significantly faster than what a single core on the Linux machine can parse. High-performance RAID systems or parallel file systems could deliver data at even higher rates.

To show the general applicability of our approach, we tested an XML document from three different sources. The
first XML document is a 34 MB file named 1kzk.xml which contains molecular information. We chose this because it is a typical shape of XML documents in scientific applications. We obtained this XML document from the Protein Data Bank [37]. The second XML document is a 29 MB file named mark.xml which contains online auction data. We chose this because it represents the typical structure of XML documents in e-commerce applications. We generated this XML document through the tool provided by the XMark project [23]. The structure of this document is very complex. The third XML document is a 19 MB file named xbench.xml which contains an order shipment, representing the typical structure of XML-based message exchange. We generated this XML document through the program developed by the XML Benchmark project [46]. These test files and their overall structures can be found at http://www.cs.binghamton.edu/~zhangying/xmlfiles/.

The compositions of these test files are very different. Figure 8 shows each file’s distributions on the frequency of states encountered when using the preparsing DFA in Figure 2 to prepare these documents.

A. Speedup and Analysis

To show the effectiveness of our parallel preparsing approaches, we measure their speedups relative to the sequential version of the parser. Figure 9 and 10 show the parallel preparsing speedups for 1kzk.xml on the Solaris machine and the Linux machine, respectively. The sequential time for Solaris machine is 3.186 seconds, and 0.219 seconds for Linux machine. For xbench.xml, its results are close to the results of 1kzk.xml, so we provide numbers in the text below rather than the graphs.

The line labelled “True contexts only with set-based p-DFA” means that timings on this line are measured by performing the work on the single true context of each chunk. In testing this case, we pre-computed the true context of each chunk. Since for each chunk except the first chunk, the true context is actually unpredictable, this line is thus the upper-bound of the performance of parallel preparsing with the set-based p-DFA. The line labelled “Set-based p-DFA without speculation” means that the results on this line are measured with the work for all contexts being performed when doing preparsing on each chunk. Lines labelled “Set-based p-DFA with 0.6 confidence” and “Set-based p-DFA with 0.9 confidence” mean that we speculate on some contexts of each chunk and the confidence levels are 0.6 and 0.9, respectively. The line labelled “With vector-based p-DFA” means the results of parallel preparsing with the vector-based p-DFA. The drops in the speedup of the 0.6 and 0.9 confidence line in Figure 11, and the speedup of the 0.6 confidence line in Figure 12 are because the speculation failed for those executions, and therefore incurred a large performance penalty.

From these preparsing speedup figures, we see that the set-based p-DFA approach is indeed far better than the vector-based version. For example, without speculation, for mark.xml, on the Solaris machine, the maximum
The maximum preparsing speedup with the set-based DFA is only 2.25. For the Linux machine, the maximum preparsing speedup with the set-based DFA is 1.71. For xbench.xml, the maximum preparsing speedup with 0.6 confidence is 6.22, and the maximum speedup without speculation is 5.44. Such improvement is somewhat less in the other two test documents. However, maintaining multiple contexts does have a cost. When the number of threads increase, the gap between the line with “True contexts only with set-based p-DFA” and the line with “Set-Based p-DFA without speculate” is considerable. This indicates that with an increasing number of threads the cost to maintain multiple contexts on each chunk does incur a considerable amount of overhead. In fact, the reduced gain from the speculation in parallel preparsing lies in that we have to consider more contexts on each chunk so that the expected confidence rate can be guaranteed when there are more number of threads.

We further applied our parallel parser to the full PXP parsing work in [20], [27] and tested the performance of the overall full PXP parsing, to see the effectiveness of our parallel parser on the full parse. The speedup is measured relative to the standalone version of libxml2, rather than relative to our PXP technique (which is built on unmodified libxml2) when running with one thread. We are thus using the standalone performance of libxml2 as representative of the performance that can be expected in a sequential parser. To show the growth trends and the effectiveness clearly assured, here all the tests with the set-based p-DFA do not use speculation. Also, to make the figures clear, we only provide the results on 1kzk.xml and mark.xml. The results on xbench.xml is very close to the 1kzk.xml. Figure 13 shows the speedups on the Solaris machine comparing full PXP parsing with the set-based p-DFA based parallel parser, with the vector-based p-DFA based parallel parser, and with the purely sequential parser. The sequential time for 1kzk.xml is 19.826 seconds, mark.xml is 6.482 seconds. Figure 14 is the results on the Linux machine. The sequential time for 1kzk.xml is 2.084 seconds, mark.xml is 0.693 seconds. As expected, parallel parsing is crucial to maintaining the scalability of PXP and PXP with the sequential parse only scales to a few processors before leveling-off. It is also very noticeable that the set-based p-DFA based parallel parsing is much more effective in boosting the full parse speed than the vector-based version, and thus it is important and even necessary for maintaining strong scalability. We also notice that the full parsing performance on mark.xml is less than the 1kzk.xml. This is because that the structure of mark.xml is much more complex than 1kzk.xml and the full PXP parsing work in [20], [27] does much better on relatively simple structured XML documents. From the preprocessing performance graphs in Figure 9, 10, 11 and 12, this in turn confirms us that our parallel parser is much effective even on very complex XML document structures.

Load balancing between threads during the prepars is achieved by dividing the document into equal-sized chunks. Each core thus gets the same number of bytes. We have found that the number of bytes to be a good predictor of the amount of work required. Load balancing during the full parse is achieved using work described in [27], [20].

VIII. Related Work

There has been a number of related works in parallelism and parsing. We do not claim that our paper makes a general theoretical contribution to that field. Rather, we contribute a specific approach and architecture for speculating with parallelism for XML (and similar languages), drawing on much related work in parallel parsing.
The work in [9] looks at improving the efficiency of parallel finite automata [35]. Parallel finite automata can be used to model concurrency, and can execute multiple automata, but are not in themselves capable of being run on multiple cores. Other work in parallel regular expression examines how to match using bit-level parallelism, and cannot leverage multiple cores [47]. The work in [32] is a hardware design, and cannot take advantage of general-purpose multicomputer systems. Our p-DFA construct is also similar to the work in [3].

The work in [7] proposes the idea of suffix parsing in order to get a different approach to error recovery [33]. If we get error for the input, it assumes that the rest of the input (the input that comes after the error symbol, excluding the error symbol itself) is a suffix (tail) of a sentence of the language. Parsing starts with a parser for the original language, preferably one with the correct prefix property. When an error is detected, it is reported, the error symbol is skipped and a parser derived from the suffix-grammar, a so-called suffix-parser, is started on the rest of the input (which must be a suffix or else there is another error). However, the suffix-grammar may not be suitable for any of the deterministic parsing methods. In fact, the way we constructed the suffix-grammar almost certainly results in an ambiguous grammar [12].

The work in [10] introduces several parallel bottom-up parsing methods, including Simple Precedence, LR(k), SLR(k), and LALR(k). This work allows concurrent reductions to occur. The work in [26] also allows parallel parsing. The above works focus on task parallelism rather than data parallelism, however. The amount of task parallelism available in an XML parser is limited, due to the simplicity of the language compared to typical programming languages such as C, or natural languages.

The work in [11] presents the packrat parser which can recognize any string defined by a top-down parsing language grammar in linear time, and could be suitable for speculative parsing. It is not clear how this would address the problem of the unknown state at the beginning of a chunk, however. The context-free parsing algorithm [8] also runs in linear time on a large class of grammars, and may be suitable for some forms of ambiguity. This ambiguity is often in the grammar itself, or the input, however, and is not the restricted form of indeterminism that we are addressing.

There exist a number of approaches that try to address the performance bottleneck of XML parsing. With the lazy parsing approach [25], a skeleton is also built from the XML document to indicate the basic tree structure, then, based on the user’s access requirements, the required pieces of the XML document will be located through the skeleton and be fully parsed. This approach seeks to do partial parsing of the XML document on demand so that unneeded work can be eliminated. Binary encoding [44] gets better performance and compactness, but they are suitable for use only in controlled environments where interoperability with textual XML is not an issue.

Work in [21] creates a “ThreadCrew” concept which represents a set of threads. Instead of maintaining a global queue as the normal ThreadPool does, each thread in the ThreadCrew has its own local task queue. A thread tries to “steal” the work from other threads in the crew when a thread is out of work. It also provides a mechanism to trace the stealing actions, thus the equivalent sequential result can be obtained by gluing the multiple parallely-produced results together. The work in [21] only covers serialization, but their ParaXML extension also includes parsing [22]. It treats the XML document as a general tree structure and parses the XML using the work stealing mechanism. The ParaXML work uses pre-parsing, but does not address parallelizing the pre-parsing, which is the topic of our paper. The ParaXML work is thus wholly complementary to our work.

Other parallel work such as [36] discusses parallelizing XSLT. It studies four execution models for applying parallelism to XML transformations. The work in [4] studies the application of parallel bit stream technology together with the study of the XML processing performance. Currently, it focuses primarily on the lexical phase of parsing for single cores using SIMD instructions, though they are also considering multicores parallelism for future work. The work in [34] is also another way to increase the XML processing performance. Its approach is to take advantage of the parallelism existing between different XML documents and then structuring XML data accordingly. This approach, however, is focused on processing a certain class of queries, and not on general XML parsing. Also, the work in [39] uses an intermediary-node concept to realize a workload-aware data placement strategy which effectively declusters XML data and thus obtains high intra-query parallelism. It also focuses on queries. The work in [14] uses an NFA approach, but does a full simulation of the NFA, and does not handle namespaces.

Hardware based solutions [16], [41], [40] are also promising, particularly in the commercial sector. The main idea is to offload some aspects of XML parsing to dedicated hardware. These approaches, however, require specialized hardware such as application-specific integrated circuits designed to accelerate XML processing, or field-programmable gate array. While their algorithms offer speedup, they are not suited for implementation on general-purpose CPUs, and cannot take advantage of increasing core counts without modification, in the same manner that our work does. Hardware approaches can thus be used in conjunction with multicore approaches, and are thus complementary to our work.

IX. Conclusion

The PXP technique is promising in that it can parse a single XML document on multiple processors with potentially great speedup. The required pre-parsing stage is naturally sequential, but can be parallelized, as seen in the vector-based and set-based p-DFAs [31], [30]. Parallelization has a cost, however, and involves extra work that will often not be needed. Maintaining multiple possibilities as contexts does incurs costs that will usually be thrown away. One approach to reduce this cost is to avoid doing work for unlikely situations. Building on our
previous work, in this paper we present an approach to analyze gathered statistics to determine likely sequences of states. Unlikely possibilities are discarded in favor of more likely ones. Our results show that this approach can yield significant benefits in a number of situations. Another avenue to explore is to apply the p-DFA approach on other types of XML processing, such as canonicalization [43]. Future work would also extend speculation to our work on streaming XML [29]. Schema-information could also be leveraged to improve parallelism, which we currently do not do.

References


