Parsing XML Using Parallel Traversal of Streaming Trees

Yinfei Pan, Ying Zhang, and Kenneth Chiu

Department of Computer Science, State University of New York,
Binghamton, NY 13902

Abstract. XML has been widely adopted across a wide spectrum of applications. Its parsing efficiency, however, remains a concern, and can be a bottleneck. With the current trend towards multicore CPUs, parallelization to improve performance is increasingly relevant. In many applications, the XML is streamed from the network, and thus the complete XML document is never in memory at any single moment in time. Parallel parsing of such a stream can be equated to parallel depth-first traversal of a streaming tree. Existing research on parallel tree traversal has assumed the entire tree was available in-memory, and thus cannot be directly applied. In this paper we investigate parallel, SAX-style parsing of XML via a parallel, depth-first traversal of the streaming document. We show good scalability up to about 6 cores on a Linux platform.

1 Introduction

XML has become the de facto standard for data transmission in situations requiring high degrees of interoperability, flexibility and extensibility, and exhibiting large degrees of heterogeneity. Its prevalence in web services has contributed to the success of large-scale, distributed systems, but some of the very characteristics of XML that have led to its widespread adoption, such as its textual nature and self-descriptiveness, have also led to performance concerns [5,6]. To address these concerns, a number of approaches have been investigated, such as more efficient encodings of XML to reduce parsing and deserialization costs [4], differential techniques to exploit similarities between messages [17], schema-specific techniques [8,19], table-driven methods [21], and hardware acceleration [7].

On the CPU front, manufacturers are shifting towards using increasing transistor densities to provide multiple cores on a single chip, rather than faster clock speeds. This has led us to investigate the use of parallelization to improve the performance of XML parsing. In previous work, we have explored parallelizing DOM-style parsing [20], where an in-memory DOM tree is generated from an entire document. In many situations, however, SAX-style parsing is preferred [1]. SAX-style parsing is often faster because no in-memory representation of the entire document is constructed. Furthermore, in some situations, the XML itself is streaming, for which an event-based API like SAX is much more suitable. Thus, in this paper, we investigate how SAX parsing can be parallelized. Parallel SAX parsing may seem to be an oxymoron, since SAX is inherently sequential. However, even though the sequence of SAX callbacks (representing the events) are sequential, we show that it is possible to parallelize the parsing computations prior to issuing the callbacks, and only sequentialize just before issuing the callbacks.

P. Sadayappan et al. (Eds.): HiPC 2008, LNCS 5374, pp. 142–156, 2008.

In some applications, application-level processing in the callbacks themselves may dominate the total time. Parallel SAX parsing could alleviate that, however, by leveraging schema information to determine which elements are not ordered (such as those in an `<xsd:all>` group). With applications that are multicore-enabled, callbacks for such elements could be issued concurrently. We do not leverage such information in current work, but see this current work as a necessary precursor to such techniques. Furthermore, we note that XML parsing is also a necessary first step for more complex processing such as XPath or XSLT. Though on a single core, the XML parsing may not dominate this more complex processing, Amdahl's law [2] shows that even relatively short sequential phases can quickly limit speedup as the number of cores increases. Parallel parsing is thus more important than a single core analysis might suggest.

Previous researchers have investigated parallel, depth-first traversal, but with the assumption that the the entire tree was always available [14,15]. Streaming situations, on the other hand, are different because the incoming stream can present itself as a possible source of new work. Furthermore, in a streaming situation, it is important to process work in a depth-first *and left-right* order when possible, so that nodes become no longer needed and can be freed.

A number of previous researchers have also explored parallel XML processing. Our previous work in [9,10,11,12] has explored DOM-style parsing. The work in [13] takes the advantage of the parallelism existing between different XML documents and then structures the XML data accordingly. This approach, however, is focused on processing a certain class of queries, as opposed to XML parsing. The work in [18] 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, however.

2 Background

Parallelism can be broadly categorized into pipelining, task-parallelism, and data-parallelism, depending on whether the execution units are distributed across different sequential stages in the processing of a single data unit, different independent tasks that may be executed in any order, or different, independent pieces of the input data, respectively.

A pipelined approach to XML parsing would divide the parsing into stages, and assign one core to each stage. Such a software pipeline can have satisfactory performance, but if followed strictly and used in isolation, suffers from inflexibility due to the difficulty of balancing the pipeline stages. Any imbalance in the stages results in some cores remaining idle while other cores catch up. Furthermore, adding a core generally requires redesign and re-balancing, which can be a challenge.[1]

[1] To improve cache locality, cores may be assigned to data units rather to stages. So, a single core might execute all stages for a given data unit. This does not resolve the stage balancing issue, however, except when there are no dependencies between data units, in which case we can simply use data parallelism. If there *are* dependencies between data units, then a core C2 processing data unit D2 in stage 1 must still wait for core C1 to complete data unit D1 in stage 2, before C2 can begin stage 2 processing on D2.

Task parallel approaches divide the processing into separate tasks which can then be performed concurrently. The difference between pipelining and task parallelism is that in the former, the output of one stage is fed into the input of the other, while in the latter, the tasks are independent (though the final output will depend on the completion of all tasks). Task parallelism also suffers from inadaptability to a varying number of cores. The code must be specifically written to use a given number of cores, and taking advantage of another core would require rewriting the code such that an additional task could be assigned to the additional core, which may be difficult and time-consuming. Tasks must be carefully balanced so that no cores are left idle while waiting for other tasks to complete.

In a data-parallel approach, the input data is divided into pieces, and each piece is assigned to a different core. All processing for that piece is then performed by the same core. Applied to streaming XML, each core would parse separate pieces of the XML stream independently. As each core finishes, the results would be spliced back together, in parallel. Adding an additional core would only require reading in an additional chunk for the new core to process. The code does not need to be re-implemented, and thus it is possible to dynamically adjust to a variable number of cores on-the-fly.

Under dynamic conditions as typically encountered in software systems (as opposed to hardware systems that do not change unless physically modified), data parallelism is often more flexible when the number of cores may change dynamically, and may have reduced bus bandwidth requirements. The major issue with data parallelism, however, is that the N pieces must be independent of each other. That is, it must be possible to parse a piece of the data without any information from any other piece. Applied to an incoming XML stream, this poses a problem, since XML is inherently sequential, in the sense that to parse a character at position p, all characters at position 1 to $p-1$ may first need to be parsed.

2.1 Hybrid Parallelism

We note that in practice there is no need to adhere strictly to a single style of parallelism in a program. Combining the forms, such as pipelined and data parallelism, to form a hybrid can improve flexibility and load balancing. For example, data dependencies can be handled in the first stage of a pipeline by strictly sequential processing. This processing can annotate the output stream of the first stage to address such ordering dependencies. The second stage could then process chunks of the incoming data stream independently in a data-parallel fashion, using the annotations to resolve any ordering dependencies. For such a technique to be beneficial, it must be possible to address data dependencies using a fast sequential stage, and defer more computationally intensive processing to a data-parallel later stage.

The improved flexibility can be illustrated when we consider adding a core to a simplified example. Consider a balanced, two-stage pipeline, where dependencies prevent a datum from entering a stage until the previous datum has completed that stage. Thus, to add a core, a new stage would need to be created, requiring recoding the software while maintaining stage balance. Now assume that the second stage is data parallel, and that the second stage takes much longer for a given datum than the first stage. That is, a datum cannot enter the first stage until the preceding datum has finished the first stage,

but a datum can begin the second stage as soon as it has finished the first stage, without waiting for previous datum to finish the second stage. In this case, an additional core can simply be added to the data-parallel second stage. Since this stage is slow relative to the first stage, sufficient data units can be in second stage processing to keep the additional core busy.

Stage balancing is thus no longer required, as long as the data parallel stage is slower than all sequential stages. In a pure pipelined parallelism, stage balancing is required because any stage that executes slower will block (or starve) other stages. The core that is assigned to an idle stage cannot be re-assigned to other stages, because all stages are sequential and so cannot utilize another core. If one of the stages is data parallel, however, any idle core can be assigned to the data parallel stage, as long as the data parallel stage has work to do. This will be true as long as it is the slowest stage.

2.2 Amdahl's Law under Hybrid Parallelism

Amdahl's Law states that if the non-parallelizable part of a particular computation takes s seconds, and the parallelizable part takes p seconds for a single core, then the speedup obtainable for N cores is $(s + p)/(s + p/N)$. Amdahl's Law is a particular perspective from which to view parallel performance. It assumes that there is a single, fixed-size problem, and models the computation as consisting of mutually exclusive sequential phases and parallel phases. It then considers the total time to compute the answer when using a varying number of processors, holding the problem size fixed. It shows that the speedup is dramatically limited, even by relatively small amounts of sequential computation. Fundamentally, the problem is because during the sequential phase, all other processors are idle. The more processors are used, the greater the amount of wasted processing power during this idling.

When considering streaming input, however, the input presents as a continuous stream, rather than a fixed-size, discrete problem. Thus, speedup of the throughput can be a more useful metric than speedup of a fixed-size work unit. If there is no pipelining, then Amdahl's law can be applied to throughput directly. The throughput is inversely proportional to the total time to process any unit of data, and that will simply be the sequential time plus the total parallel computation divided by the number of processors. Without pipelining, when the sequential computation is occurring, all other cores are idle, as in the non-streaming case.

If we consider hybrid parallelism under streaming, however, the picture can appear significantly different. Consider a pipeline consisting of an initial sequential stage followed by a data-parallel stage. While one core is performing the sequential computation, all other cores are working in parallel on the second stage. When cores are added, they are assigned to the second stage, where they always have work, as long as the first stage is not the bottleneck. Assuming that the sequential stage can be made fast, often the core assigned to the first stage will have no work to do. Since the second stage is data-parallel, however, this core can simply be time-sliced between the first stage and the second stage to achieve load balancing.

To illustrate this, in Figure 1 we compare the throughput speedup of a two-stage computation under two different scenarios: pipelined, streaming; and non-streaming, non-pipelined. In this manufactured example, we assume that for a given datum, the

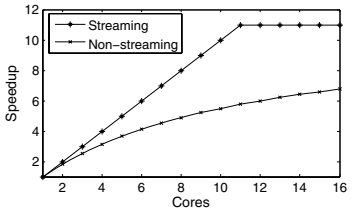

Fig. 1. Throughput speedup of a two-stage computation under streaming, pipelined; and non-streaming, non-pipelined scenarios

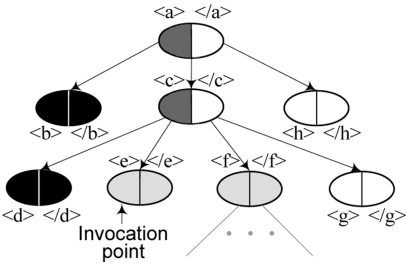

Fig. 2. This shows a conceptual representation of a streaming XML document, in left-to-right order. Elements are shown here as split ovals, representing two nodes. The left half is the start-tag node, and the right half is the end-tag node. Note that the child links actually only point to the start-tag node; end-tag nodes are not actually linked into the tree, but are rather pointed-to by the start-tag node. Nodes in different categories (explained in the text) are shown in different colors.

sequential first stage takes 1 second, and the data-parallelizable second stage takes 10 seconds, for a single core. If a single, discrete problem is presented, and the first stage and the second stage are not pipelined, then the total time per datum is simply $1 + 10/N$, where N is the number of cores. Speedup is then given by $11/(1 + 10/N)$ per a straightforward application of Amdahl's Law. If, however, a stream of data is presented, and we allow the first and second stages to be pipelined, we can assume that one core is time-sliced between the sequential first stage and the data-parallel second stage, and that all other cores are always working on the second stage. If there are 11 cores or fewer, the time taken per datum is then simply $11/N$, since the total computation time for a datum is 11 seconds (for a single core), and no cores are ever idle. The throughput is then given by $N/11$. If there are more than 11 cores, the first stage becomes the bottleneck, and throughput then becomes rate-limited at 1 datum/second by the first stage, which we have assumed cannot be parallelized.

As the graph shows, speedup under hybrid parallelism is limited by the throughput of the slowest sequential stage. Prior to that point, however, the speedup is in fact linear. This observation means that speedup for stream-oriented processing under multicore processing can have significantly different characteristics than the usual non-streaming case, though asymptotically they reach the same limit.

3 Parallel Parsing of Streaming XML

To perform namespace prefix lookup, some kind of data structure is necessary. We could use a stack for this, but a tree is a natural representation that makes the inherent parallelism more explicit, and we chose this model. Thus, we represent start-tags, end-tags, and content as nodes. Only start-tags are actually linked into the tree via parent-child links, since only start-tags need actual namespace prefix lookup. An end-tag is

linked-in via a pointer to it in the corresponding start-tag, as well as by the SAX event list discussed below. Content nodes are linked-in via the SAX event list only.

Although the tree structure is required for the namespace prefix lookups, the nodes are also used to represent and maintain SAX events. To facilitate this usage, we also maintain a linked list of the nodes in SAX order (depth-first). Thus, two types of organization are superimposed on each other within a single data structure. The first organization is that of a tree, and is used for namespace prefix lookup. The second is that of a linked list, and is used for the callback invocations used to send SAX events to the application.

The primary challenge with parallel SAX parsing is that when an XML document is streamed in, only part of the tree is resident in memory at any one moment. This resident part consists of open start-tags, closed start-tags that are still needed for namespace prefix lookup, and nodes that have not yet been sent to the application as SAX callbacks. This rooted, partial tree is called the *main* tree. More precisely, we can divide the nodes of a streaming tree into four categories, as shown in Figure 2. The first category is nodes that have not been received at all yet. Nodes <g>, </g>, </c>, <h>, </h>, and are in this category. The second category is nodes that have been received, but have not yet been invoked. Nodes <e>, </e>, <f>, and </f> are in this category.

Since prefix-to-URI bindings are stored with the nodes, a node needs to stay resident even after it has been sent to the application, as long as it has descendants that have not yet completed namespace prefix lookup. An open start-tag may still have children on the wire, and so must also stay resident. Note that the root start-tag is only closed at the end of the document, and so is always resident. These types of nodes form the third category, and nodes <a> and <c> are examples of this. The last category is nodes that are no longer needed in any way. These nodes represent events that have already been sent to the application via callbacks, and will not be needed for further namespace prefix lookups. These nodes can be deleted from memory. Nodes , , <d>, and </d> are in this category. In summary, category 2 and 3 nodes are resident in memory, while the rest are either still out on the network, or have been deallocated.

3.1 Stages

Our SAX parsing thus uses a five-stage software pipeline with a combination of sequential and data-parallel stages. Stage one (PREPARSE) reads in the input XML stream by chunks, and identifies the state at the beginning of each chunk via a fast, sequential scan (called a preparse). Stage two (BUILD) then takes the data stream from the PREPARSE stage, and builds fragments of the XML tree in a data-parallel, out-of-order fashion. Each chunk from the PREPARSE stage generates a forest of XML fragments. Stage three (MERGE) sequentially merges these forests into the current main tree to allow namespace prefix lookup to occur. Stage four (LOOKUP) then performs lookup processing on nodes in a parallel, depth-first order (though the actual lookup itself goes up the tree). Finally, in stage five (INVOKE), SAX events are sequentially invoked as callbacks to the application.

This pipelining of sequential and parallel stages into a hybrid parallelism provides greater scheduling flexibility as explained in Section 2.1. The fundamental benefit of hybrid parallelism is that it allows cores that might otherwise be idle due to stage

imbalance to be assigned to data-parallel stages, in most cases. Though the PREPARSE, MERGE, and INVOKE stages are still intra-stage sequential, the stages are designed so that as much as computation as possible is moved out of these stages, and into data parallel stages, thus enhancing scalability.

The data unit between the stages changes as the data flows. In stage one, the XML stream is partitioned into chunks, and passed to stage two. In stage two, the chunks are parsed into tree fragments consisting of graph-based nodes. Stage three merges these fragments into the current tree. Stage four can now operate on a tree, and remain mostly oblivious to the fact that it is actually a stream. Stage five operates on a sequence of SAX events.

There is an approximate, but inexact correspondence between the stages and the categories shown in Figure 2 and explained in the previous section. Any node that has been deallocated or not yet received is not in any stage, of course. A node that is in the PREPARSE, BUILD, or LOOKUP stage is in category two, since they have been received, but not yet invoked. A node that has passed the INVOKE stage could be in category three, if it needs to stay resident for children nodes to perform LOOKUP stage; or, it could be deallocated, and thus in category four.

We now describe the stages in more detail.

3.2 Stage One (PREPARSE): Preparse Chunks

To obtain data-parallelism, in the ideal case we could simply divide the incoming XML document into chunks, and then parse each chunk in parallel to build the tree. The problem with this, however, is that we do not know which state to begin parsing each chunk, since the first character is of unknown syntactic role at this point. Thus, a parser that begins parsing at some arbitrary point in an XML stream (without having seen all previous characters) will not know which state in which to begin.

To address this, and still allow parallelism, we use a fast PREPARSE stage to first identify the basic structure of the XML stream. We divide the incoming XML stream into chunks, and then push each chunk through a table-driven DFA that identifies the major syntactic structures within the document. This gives us the state at the end of the chunk, and thus the state at which to begin the next chunk. This initial state is then passed to the next stage, along with the actual chunk data. For example, this DFA will determine if a chunk begins within a start-tag, end-tag, character content, etc. The effect is to "fast-forward" through the XML stream.

Because the preparse process is sequential, this technique works only as long as the PREPARSE stage can be much faster than the actual full parsing. In practice, we have found this to be true. If the preparse itself starts to become a bottleneck, we have shown in previous work how the preparse itself can be parallelized [11].

3.3 Stage Two (BUILD): Build Tree Fragments

The BUILD stage parses chunks in parallel, and builds tree fragments from each chunk, corresponding to the XML fragment in the chunk. The work of the BUILD stage can occur in a data parallel fashion, and so can have multiple cores assigned to it, one for each chunk. It obtains the proper state at which to begin parsing each chunk from the

previous PREPARSE stage. The DFA for this stage, called the bt-DFA (Build Tree Fragment DFA), parses the syntactic units in detail. For example, in the start-tag, the element name will be parsed into namespace prefix and local part. The attribute will be parsed into namespace prefix, attribute name's local part, and attribute value.

The bt-DFA is different from the DFA for the preparse stage, because it more precisely identifies the syntactic units in XML. We thus need to map the preparsing DFA state given by the PREPARSE stage to the state of the more detailed bt-DFA.

In the preparsing DFA, a single state 3 is used to identify any part inside a start-tag that is not inside an attribute value. So, inside an element name is treated the same as inside an attribute name. In the bt-DFA, the start-tag is treated in greater detail, so preparser DFA state 3 may map to a number of different states in the bt-DFA. The preparser DFA, however, simply does not provide enough information to know which bt-DFA state should be used for state 3. To solve this ambiguity, we create a special auxiliary state in the bt-DFA, state 0. Preparse state 3 will map to bt-DFA state 0, but as soon as it receives enough input to distinguish further which syntactic unit it is in, it will go to the corresponding bt-DFA state.

In XML documents, start-tags and end-tags are organized in a paired fashion in depth first order. To assist in building the tree fragments, we maintain a stack of start-tag nodes. When we encounter a start-tag, we add it as a child of the top start-tag node, and then push it on to the stack. When we encounter an end-tag, we close the top start-tag, and pop the stack. Although the tree structure is required for the namespace prefix lookups, the SAX event list must also be maintained, as explained near the beginning of Section 3. Therefore, as the DFA progresses, it also creates the event list. Every time a start-tag, end-tag, or character content SAX event is recognized, its corresponding node is linked to the last such node. This creates the list in SAX event order, which is actually the same as XML document order.

For well-formed XML fragments, the start-tags and end-tags inside will be all paired, thus, the stack at the end of parsing such fragment will be empty. But because the XML fragments are partitioned in an arbitrary manner, there will usually be more start-tags than end-tags, or more end-tags than start-tags. If there are more start-tags than end-tags, there will be start-tags remaining on the stack when this stage finishes a chunk. This stack is preserved for merging in the next stage. The output of this stage is thus a set of tree fragments, and a stack remnant, as shown in Figure 3. The nodes in the tree fragments are linked together in a SAX event list in addition to the tree links.

3.4 Stage Three (MERGE): Merge Tree Fragments

The result of the BUILD stage are disconnected tree fragments. These fragments are not connected to the root, or nodes in previous chunks, and so namespace prefix lookups cannot be done. The purpose of the MERGE stage is thus to merge these fragments into the main, existing tree. The merge is performed using the stack from the tree fragment created from the preceding chunk, and the event list of the successor chunk. The merge occurs by traversing the successor fragment event list and matching certain nodes to the predecessor stack, but in such a manner that only the top edge of the successor fragment is traversed, since only the top edge of the successor fragment needs to be merged. The internal portion of the successor fragment is already complete. The algorithm for

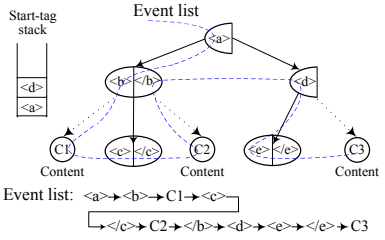

Fig. 3. This diagram shows a tree fragment created by the BUILD stage. Solid-line arrows show actual child links. Dotted line arrows show conceptual child relationships that are not embodied via actual pointers. The dashed line shows how the nodes are linked into an event list for this fragment. The start-tag stack holds unclosed start-tags, to be closed in later fragments.

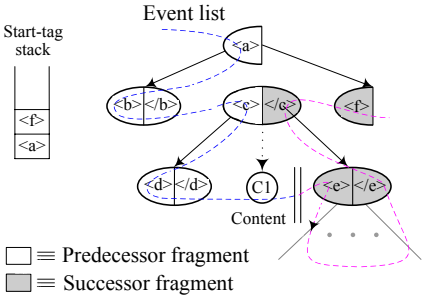

Fig. 4. This diagram shows two tree fragments immediately after they have been merged, including the stack after the merge. Nodes `<e>`, `</e>`, `</c>`, and `<f>` were in the successor fragment, while the other nodes were in the preceding fragment. The double vertical line between nodes `c1` and `<e>` shows the division between the event list of the predecessor tree fragment from the event list of the successor tree fragment. If successor fragment is not the last fragment in the document, some start-tags will be unclosed, such as `<a>` and `<f>` in this example.

merging is given in Algorithm 1. Figure 4 shows two tree fragments after they have been merged.

In addition, the chunking process of the PREPARSE stage creates chunk boundaries which fall at arbitrary character positions. Thus, some XML syntactic units, such as an element name, will be split across two chunks. The MERGE stage also resolves these split units, completing them so that their corresponding nodes can be processed normally in later stages.

3.5 Stage Four (LOOKUP): Namespace Prefix Lookup

The purpose of this stage is to lookup namespace prefixes. The actual lookup is performed by going up the tree to the root. But the order in which nodes undergo lookup processing is top-down. In other words, a child cannot undergo lookup processing unless its parent has already undergone the LOOKUP stage. Thus, lookup processing is ordered using a parallel depth-first traversal of the main tree.

To assist in load-balancing, work queues of nodes that should next be traversed are maintained. These queues essentially contain nodes that are at the “traversal front”. Each thread maintain a work queue of rooted sub-trees, represented by the root node of the sub-tree. As each sub-tree root is processed, each child of this root then becomes the root of another, new sub-tree, and so is added to the thread-local work queue. Because new tree fragments are being merged into the main tree concurrently with this traversal, it is possible that after a node is processed, there are still unmerged, “future” children. These children would not be added to the local work queue, because they were not

Algorithm 1: Merge two tree fragments

```
Data: Tree fragment prev, Tree fragment next
Result: The merged tree fragment into prev
EventNode p ← next.event_list.head;
while p exists AND prev.stack ≠ ∅ do
    if is_StartTag(p) then
        Set p.parent refer to prev.stack.top();
        Link p as a child node of p.parent;
        if p.EndTag exists then
            p ← p.EndTag;
            Set p.parent refer to prev.stack.top();
        else
            break;
    else if is_EndTag(p) then
        StartElement s ← prev.stack.top();
        Set s.EndTag refer to p;
        Set p.parent refer to s.parent;
        Pop prev.stack;
    p ← p.next_event;
Stack next.stack onto prev.stack;
Concatenate prev.event_list with next.event_list;
```

Fig. 5. The algorithm for merging two tree fragments

linked in as children when the parent was processed, and thus would never be processed by the LOOKUP stage.

To address this, when a node is merged into the main tree, a check is performed on the parent node. If the node has already undergone namespace prefix lookup, then the node is instead added to a global, unassigned work queue. When a thread runs out of work, it first checks the unassigned work queue, and if non-empty, it grabs it as its new work.

3.6 Stage Five (INVOKE): Invoke Callbacks

Finally, the actual SAX events must be sent to the application by invoking callbacks. We assume in this paper that the callbacks must be issued sequentially, though it is possible that a multi-core enabled application could handle concurrent callbacks in schema types that are order independent, such as `<all>`. One way of invoking callbacks would be to have a single thread invoke them. This would result in poor cache locality, however, since this thread would likely not belong to the core that last accessed the node currently being invoked.

To improve locality, we utilize a virtual token that is passed along the SAX event chain, called the NEXT token. This token is implemented via a flag. To pass the token from an earlier node to a later node, the flag is set in the later node and cleared in the earlier node. Chained locking is used to prevent race conditions.

When a thread finishes namespace lookup on a node, it checks to see if it is the next node, by virtue of holding the token. If it is, it invokes it, and passes the token to the

next node in the chain, and continues to invoke it also, if it has been completed and is ready to be invoked. The chain continues until encountering a node that is not ready to be invoked. Though multiple threads could be in the LOOKUP stage at the same time, the mechanism used to pass the token down the chain ensures that the INVOKE stage is sequential.

Another way of doing something similar would be have a special global pointer that always points to the next node to be invoked. This would require that all cores continuously read and update the same pointer, however. By using the NEXT token, we avoid excessive operations to the same memory location, avoiding cache-line ping-pong.

3.7 Execution

One way of implementing the stages would be to start a set of threads for every stage, with each sequential stage having just one thread. Parallel stages would have as many threads as there are cores. This architecture relies on the OS to perform all scheduling, however. Furthermore, as work moves along the pipeline, it may jump from one core to another, resulting in poor cache locality.

To address this, another way of implementing the stages is to represent each stage as a set of work units, and is what we use. Each thread then examines each stage, and decides what work to pick up. Once a thread accepts work, it can perform more than one stage on the same data, improving cache locality. In other words, rather than cores exhibiting affinity to stages, cores exhibit affinity to *data*, and so the core and the data move together through the stages.

At the beginning of execution, a thread is started for every core. It then executes a main loop that examines each stage, deciding which stage has work that should be done, and of those, which stage has the highest priority work. It first checks to see whether or not there is any unassigned LOOKUP stage work to perform. If so, it accepts that work. If not, it then checks to see whether or not there is any new data to be read from the network and pushed through the PREPARSE stage. After pushing the data through the PREPARSE stage, the same thread will execute BUILD and MERGE on it. All LOOKUP stage work is initiated through the unassigned work list. INVOKE stage work is initiated internally by the LOOKUP stage itself. If there is no work in the main loop, work stealing is done from a randomly chosen victim. We currently only steal a single sub-tree from a thread at a time, rather than half of the remaining work.

Thus, this is an explicit form of scheduling, which gives us control over what work to do next. Note that if we simply had a set of threads for each stage, the OS may decide to run a thread which is runnable, but may not be best choice for temporal cache locality or other efficiency reasons.

3.8 Memory Management

Memory management is a challenging problem in parallel programs. The complex, dynamic data structures needed for load-balancing and concurrency, complicate determining exactly when an object can be freed. To handle this, we have adopted the commonly used technique of reference counting. In this technique, a count is maintained with every object, indicating the number of pointers that are currently pointing to it. When

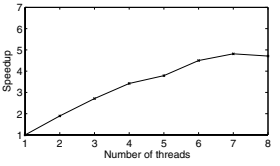

Fig. 6. Speedup of the parallel SAX parser

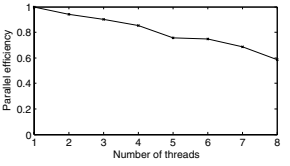

Fig. 7. Efficiency of the parallel SAX parser

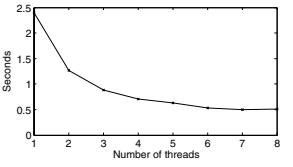

Fig. 8. Total parse time

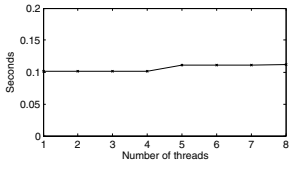

Fig. 9. Time used by the PREPARSE stage over the document

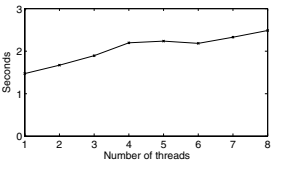

Fig. 10. Time used by the BUILD stage summed over all threads over the document

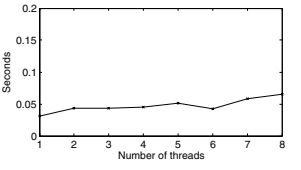

Fig. 11. Time used by the MERGE stage over the document

the count goes to zero, this means that the object is no longer referenced, and can be freed. Atomic increment and decrement operations are used to reduce the cost of locking needed to maintain the counts.

Another issue is thread-optimized memory allocation. This has been the subject of significant research, and since our goal in this paper is not to address this separate topic, we use a simple, free list that we maintain separately for each thread. Thread specific data is used to keep the bookkeeping for each thread separate from the others, thus obviating the need for expensive locking.

4 Performance Results

Our experiments were conducted on a 8-core Linux machine with two Intel Xeon L5320 CPUs at 1.86 GHz. Each test was run five 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 `-O`. The SAX callbacks fulfilled by our hybrid parallel SAX parser are based on the SAX2 [1,3] API, through a C++-style binding. The test file is a file named `1kzk.xml` which contains molecular information representing the typical shape of XML documents. We obtained this XML document from the Protein Data Bank [16]. We varied the number of atoms in the document to create a 34 MB XML document.

The speedup of our parallel SAX parser is shown in Figure 6. The speedup is compared to our parallel SAX parser running with one thread. We obtain reasonable speedup up to about six or seven cores. The corresponding wall clock time is shown in Figure 8. To get a better sense of scalability, we plot the parallel efficiency in Figure 7.

To investigate where time was being spent, we also graphed the time spent in the PREPARSE, BUILD, and MERGE stages, in Figures 9, 10, and 11, respectively.

Measurements for these were done using using the real-time cycle counter based clock available on Intel CPUs. The BUILD time is summed over all threads, and so is greater than the total wall clock time given in Figure 8. The PREPARSE stage is currently sequential, and takes a relatively small amount of the total time. We note that the time spent in this stage does not have the same type of impact that would result from using this directly in an Amdahl's Law calculation, as explained in Section 2.1. This is because this PREPARSE stage is pipelined with other stages. Eventually, however, further improvements in scalability will need to address the PREPARSE stage. In previous work [11], we have shown how to parallelize the preparsing, and so this work would need to be incorporated at some point. The MERGE stage is currently sequential, but could actually be parallelized. That is, it is possible to merge fragment F1 and F2 at the same time that F2 is being merged with F3. We note that the MERGE stage takes a relatively small fraction of the total time, however, so parallelizing it might be of somewhat lower priority.

To investigate how to improve scalability further, we conducted additional timing tests, and manually instrumented the code. Part of the problem is due simply to load balancing. Some thread convoy effects are causing some periods of idle threads, which are exacerbated when the number of cores are high. Better work-stealing, better scheduling, and perhaps introducing feedback though control-theoretic techniques can improve this. Another problem is that large XML documents tend to be broad, and thus have an upper-level node that has a large number of children. This results in a hot-spot on the child list of that node, and also the reference count. This can partially be seen by the fact that even though the total amount of work remains constant, the time taken by the BUILD stage increases as the number of threads increases. Future work will address this by using shadow nodes to turn long children lists into trees of descendants, thereby reducing hot-spots. The shadow nodes will help disperse the hot spots, but will simply be skipped during callbacks.

5 Conclusion

To improve memory usage and performance, streaming XML applications are commonplace. For these, SAX-style parsing is the natural choice due to the stream-orientation of event-based callbacks. However, since XML is inherently sequential, parallelization of SAX-style parsing is challenging. To address this, we have devised a parallel, depth-first traversal technique for streaming trees that is effective in parallel, SAX-style parsing of XML. We have shown how this can be used in a five-stage hybrid pipeline which effectively extracts data parallelism from the parsing process, allowing multiple cores to work concurrently within those stages. The sequential requirements of the SAX parsing will be fulfilled by sequential stages. Since the major computational work are done by the data parallel stages, we are able to achieve good performance gain. The design of this approach allows the flexible utilization of a varying number of cores.

Using this approach, we show scalability up to about 6 or 7 cores. As manufacturers increase the number of cores, our future work will seek to further reduce the sequential stages, and thus exploit future CPUs. Techniques such as lock-free synchronization may also be employed to reduce synchronization costs. We note that when considering the

scalability of processing that has not been parallelized by previous research, the alternative is not to use parallelism at all, which would result in no speedup. For applications that can benefit from faster SAX performance, cores may end up being wasted.

References

1. SAX, http://www.saxproject.org/
2. Amdahl, G.M.: Validity of the single-processor approach to achieving large scale computing capabilities. In: Proceedings of AFIPS Conference, Atlantic City, NJ, vol. 30, pp. 483–485 (1967)
3. Brownell, D.: SAX2. O'Reilly & Associates, Inc., Sebastopol (2002)
4. Chiu, K., Devadithya, T., Lu, W., Slominski, A.: A Binary XML for Scientific Applications. In: International Conference on e-Science and Grid Computing (2005)
5. Chiu, K., Govindaraju, M., Bramley, R.: Investigating the limits of soap performance for scientific computing. In: HPDC 2002 (2002)
6. Head, M.R., Govindaraju, M., van Engelen, R., Zhang, W.: Benchmarking xml processors for applications in grid web services. In: Löwe, W., Südholt, M. (eds.) SC 2006. LNCS, vol. 4089. Springer, Heidelberg (2006)
7. IBM. Datapower, http://www.datapower.com/
8. Kostoulas, M.G., Matsa, M., Mendelsohn, N., Perkins, E., Heifets, A., Mercaldi: Xml screamer: an integrated approach to high performance xml parsing, validation and deserialization. In: WWW 2006: Proceedings of the 15th international conference on World Wide Web, NY, USA (2006)
9. Lu, W., Pan, Y., Chiu, K.: A Parallel Approach to XML Parsing. In: The 7th IEEE/ACM International Conference on Grid Computing (2006)
10. Pan, Y., Lu, W., Zhang, Y., Chiu, K.: A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs. In: 7th IEEE International Symposium on Cluster Computing and the Grid, Brazil (May 2007)
11. Pan, Y., Zhang, Y., Chiu, K.: Simultaneous Transducers for Data-Parallel XML Parsing. In: 22nd IEEE International Parallel and Distributed Processing Symposium, Miami, Florida, USA, April 14-18 (2008)
12. Pan, Y., Zhang, Y., Chiu, K., Lu, W.: Parallel XML Parsing Using Meta-DFAs. In: 3rd IEEE International Conference on e-Science and Grid Computing, India (December 2007)
13. Qadah, G.: Parallel processing of XML databases. In: Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering, May 2005, pp. 1946–1950 (2005)
14. Rao, V.N., Kumar, V.: Parallel depth first search. part i. implementation. Int. J. Parallel Program. 16(6), 479–499 (1987)
15. Reinefeld, A., Schnecke, V.: Work-load balancing in highly parallel depth-first search. In: Proc. 1994 Scalable High-Performance Computing Conf., pp. 773–780. IEEE Computer Society, Los Alamitos (1994)
16. Sussman, J.L., Abola, E.E., Manning, N.O.: The protein data bank: Current status and future challenges (1996)
17. Takase, T., Miyashita, H., Suzumura, T., Tatsubori, M.: An adaptive, fast, and safe xml parser based on byte sequences memorization. In: WWW 2005: Proceedings of the 14th international conference on World Wide Web, pp. 692–701. ACM Press, New York (2005)
18. Tang, N., Wang, G., Yu, J.X., Wong, K.-F., Yu, G.: Win: an efficient data placement strategy for parallel xml databases. In: 11th International Conference on Parallel and Distributed Systems (ICPADS 2005), pp. 349–355 (2005)

19. van Engelen, R.: Constructing finite state automata for high performance xml web services. In: Proceedings of the International Symposium on Web Services (ISWS) (2004)
20. W3C. Document Object Model (DOM), http://www.w3.org/DOM/
21. Zhang, W., van Engelen, R.: A table-driven streaming xml parsing methodology for high-performance web services. In: IEEE International Conference on Web Services (ICWS 2006), pp. 197–204 (2006)