Performance Enhancement with Speculative Execution Based Parallelism for Processing Large-scale XML-based Application Data

Michael R. Head
mike@cs.binghamton.edu

Madhusudhan Govindaraju
mgovinda@cs.binghamton.edu

Grid Computing Research Laboratory
Computer Science Department
Binghamton University
Binghamton, NY, USA

ABSTRACT

We present the design and implementation of a toolkit for processing large-scale XML datasets that utilizes the capabilities for parallelism that are available in the emerging multi-core architectures. Multi-core processors are expected to be widely available in research clusters and scientific desktops, and it is critical to harness the opportunities for parallelism in the middleware, instead of passing on the task to application programmers. An emerging trend is the use of XML as the data format for many distributed/grid applications, with the size of these documents ranging from tens of megabytes to hundreds of megabytes. Our earlier benchmarking results revealed that most of the widely available XML processing toolkits do not scale well for large sized XML data. A significant transformation is necessary in the design of XML processing for distributed applications so that the overall application turn-around time is not negatively affected by XML processing. We discuss XML processing using PiXiMaL, a parallel processing library for large-scale XML datasets. The parallelization approach is to build a DFA-based parser that recognizes a useful subset of the XML specification, and convert the DFA into an NFA that can be applied to an arbitrary subset of the input. Speculative NFAs are scheduled on available cores in a node to effectively utilize the processing capabilities and achieve overall performance gains. We evaluate the efficacy of this approach in terms of potential speedup that can be achieved for representative XML datasets. We also evaluate the effect of two different memory allocation libraries to quantify the memory-bottleneck as different cores access shared data structures.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming;
F.1.2 [**Modes of Computation**]: Parallelism and concurrency

General Terms

Algorithms, Languages, Performance

Keywords

Chip-level multiprocessing, Parallel techniques, XML Datasets

1. INTRODUCTION

XML is now widely used as an application data format. The use of XML as a data-format for distributed applications is due to its support for extensibility, namespace qualification, and data-binding to many programming languages. Scalable processing of XML datasets is an immediate concern as the size of XML data used by applications has steadily increased over the years in both scientific and business applications. For example, recognizing the increasing role of XML in representation and storage of scientific data, XDF, the eXtensible Data Format for Scientific Data, is being developed at GSFC's Astronomical Data Center (ADC), to describe an XML mark-up language for documents containing major classes of scientific data. This effort is expected to define a generic XML representation to accommodate the diverse needs of various scientific applications. The MetaData Catalog Service (MCS) [17] provides access via a Web service interface to store and retrieve descriptive information (metadata) on millions of data items. While the Web service approach of MCS provides interoperability, it also hurts the performance when compared to use of a standard database for storage and retrieval. Scientific applications such as Mesoscale meteorology [6] depend on the orchestration of several workflows, defined in XML format. The international HapMap project aims to develop a *haplotype* of the human genome. The schemas used to describe the common patterns in human DNA sequence variation can have tens of thousands of elements. The XML files in the protein sequence database are close to a gigabyte in size. The eBay Web service specification has a few thousand elements and a few hundred complex type definitions. Communication with eBay via the SOAP protocol requires processing of large XML documents.

The emergence of Chip Multi Processors (CMPs), also called multi-core processors, provides both opportunities and challenges for designing an XML processing toolkit tailored for large-size XML datasets. Compared to classic symmetric multi-processing systems (SMPs) of independent chips, the communication costs of on-chip shared secondary cache in CMPs is considerably less, providing opportunities for performance gains in fine-grained multi-threaded parallel code. CMPs provide special advantages due to locality. The individual cores are more closely connected together than in an SMP system. Multiple cores on the same chip can possibly share various caches, including the translation look-aside buffer (TLB), and the bus. An important design consideration is that off-

HPDC'09, June 11–13, 2009, Munich, Germany.

chip memory access and latency can be the choking point in CMP processors.

Our earlier work on benchmarking XML processing showed that for most XML toolkits scalability is adversely affected as the size of the XML datasets increase [10, 11]. These toolkits are typically designed to process small-sized XML datasets. The recent trends and announcements from major vendors indicate that the number of cores per chip will steadily increase in the near future. The performance limitation of existing XML toolkits will likely be exacerbated on multi-core processors because performance gains need to be mainly achieved by adding more parallelism rather than serial processing speed. Additionally, scalable processing of XML data is now of critical importance in scientific applications where the size of XML can exceed hundreds of megabytes. As a result, our focus is on harnessing the benefits of fine grained parallelism, exploiting SMP programming techniques to process large-scale XML-based application documents, and design of algorithms that scale well with increase in number of processing cores.

Many parallel compilation ideas have been discussed in the literature years [2, 5, 8, 12], studying both compilers that generate parallel code as well as those that divide work across multiple processors. With the popularization of multi-core processors and the disparity between processor and memory speed, we expect that substantial benefits can be uncovered by utilizing more cores during XML document processing. In this paper we use the PIXIMAL toolkit to evaluate the best parallelization strategies for various data structures used in distributed applications. We also compare and contrast the effect of memory allocation libraries on synchronization and management costs as multiple-threads compete for access to the main memory.

The specific contributions of our work include:

- We present a multi-threaded parallelization technique to process large-scale XML data.
- We present a framework that helps evaluate how the size and data-types in an XML document affects the distribution of the data to various threads in a multi-core environment.
- We compare the use of GNU libc 2.7 and Google's thread caching malloc libraries for use in a multi-core environment where use of shared data structures can invoke expensive synchronization algorithms, affecting overall application performance.
- We present the scalability of PIXIMAL in terms of speedup achieved as the size of the XML input data and processing threads are increased.
- We study the usage of various *states* in the processing automaton to provide insights on why the performance varies for differently shaped input data files.

2. RELATED WORK

A wide range of implementations of XML parsers is available, including Xerces (DOM and SAX) [22], gSOAP [20], Piccolo [14], Libxml [21], VTD-XML [23], Qt4 [18], and Expat [4]. XML primarily uses UTF-8 as the representation format for data and various studies have shown that this representation format can hinder the overall application performance. Sending commonly used data structures via standard implementations of XML based protocols, such as SOAP, incurs severe performance overheads, making it difficult for applications to adopt Web services based distributed middleware [11]. Several novel efforts to analyze the bottlenecks and address the performance at various stages of a Web services call stack have been discussed in the literature [1, 3, 7, 20]. These optimizations, which are tailored just for the uni-core case, include: (1) the gSOAP parser [20] uses look-aside buffers to efficiently parse frequently encountered XML constructs; (2) parsing of XML schemas has been improved with the use of schema-specific parsing along with *trie* data structures so that frequently used XML tags are parsed only once [3, 19]; (3) gSOAP uses a performance aware compiler to efficiently parse XML constructs that map to C/C++ types. It uses a single-pass schema-specific recursive-descent parser for XML decoding and dual pass encoding of the application's object graphs in XML [20]; and (4) VTD-XML parser achieves performance improvement via incremental update, hardware acceleration, and native XML indexing.

Recent work by Zhang et al [24] has demonstrated that it is possible to achieve high performance serialized parsing. They have developed a table driven parser that combines the parsing and validating an XML document in a very efficient way. While this technique works well for serial processing, it is not tailored for processing on multi-core nodes, especially for very large document sizes. In our previous work in this area, we focused just on the memory bandwidth in multi-core architectures when multiple threads operate concurrently to read large input files [9].

A related project, MetaDFA [13, 15] toolkit, presents a parallelization approach that chiefly uses a two-stage DOM parser. It conducts pre-parsing to find the tag structure of the input before, or possibly pipelined with, a parallelized DOM builder run on its output (a list of document offsets of start and end tags). Our toolkit, PIXIMAL, however, generates SAX events and thus serves a different class of applications than MetaDFA. Additionally, PIXIMAL conducts parsing work dynamically, and generates as output a sequence of SAX events. This results in larger number of DFA states, and more opportunities for optimizations for different class of application data files.

2.1 Memory Bandwidth and State-Scalability

PIXIMAL can also be used to determine the effective memory bandwidth in reading large-scale application documents, and the effect of the complexity of the XML specification on performance. A thorough description and analysis of the effective memory bandwidth of the PIXIMAL approach is presented in another venue [9]. In this section, we present a summary of research findings on these two topics.

The memory bandwidth and state scalability tests were run on 1U nodes configured with 2× quad core (2.33 Ghz Intel Xeon E5345 CPUs). Each node has 8 gigabytes of RAM and run a 64 bit distribution of Debian 4.0, using Linux kernel 2.6.18. The filesystem in use in the test directory here is xfs.

As an N-way parallel parser would concurrently be reading using N different threads, we conducted tests to check whether the memory subsystem can provide substantial bandwidth when sequentially reading from a very large input. This test has two parameters: $split\_percent$ and *thread count*. The $split\_percent$ is particular to the PIXIMAL approach: it denotes the percent of input that is directed at the DFA thread. The number of threads defines the number of concurrent automata: 1 DFA and $number\_of\_threads - 1$ NFAs. The balance of the input ($input\_size\ (1 - split\_percent/100)$) is divided evenly among the NFA threads. In the case that $number\_of\_threads = 1$, $split\_percent$ is overridden to be 100% in order to ensure that the entire input is read.

The results of these tests demonstrated that there was plenty of memory bandwidth to effectively read the input concurrently from as many as six cores of an eight core machine.

The speculative threads in a parser built using NFAs will have substantially more work than the DFA thread. This test models an aspect of that extra workload – the number of states that the

NFA must initially consider – to examine the affect of language complexity on the efficacy of this approach.

This test has one more parameter than the memory bandwidth test: the size (number of states) of the DFA. Here, the PIXIMAL DFA is modeled as a thread that has a *state_number* which is initialized to 0 and takes values between 0 and dfa_size-1 . The next *state_number* is calculated for each byte of input by looking up the current *state_number* and current byte in a two dimensional array. The NFAs are modeled by threads that start with an array of dfa_size-1 start values, each initialized to a number between 1 and dfa_size-1. An NFA will never start in the state designated by 0, because that is a start state that is only valid before the DFA begins reading. The NFA recalculates each entry of the state array for each byte of input using the same rule as the DFA.

The results of this test showed that the number of states in the DFA is inversely proportional with maximal speedup. Further, the curve between these two variables has a very steep portion around DFA sizes of between 6-8. DFAs with fewer states demonstrated similar performance, with a tight grouping around 4.5 times speedup with 8 threads on an 8-core machine. DFAs with more states had a similar grouping at a much lower speedup around 2. The 6 state DFA performed between the two groupings, with speedup around 3.5 with 8 threads. The more complex the DFA, the more work it can do (i.e., it can recognize a language of greater complexity). It is desirable for the DFA to do as much work as possible because in the table-driven implementation, it has a very low per-byte processing cost. On the other hand, more states in the DFA leads to a greater number of paths through the NFA, which limits the benefit of this parallelization approach. This test aids in quantifying just how much extra work is done by the NFA and how that affects overall performance.

3. PARALLEL XML DATA PROCESSING

A deterministic finite automata (DFA)-based lexical scanner is generally used to *tokenize* the input characters of the file (or string, as in the case of XML) into syntactic tokens that are used later in the parse phase. The DFA based lexical scanner is sometimes hand-coded, and frequently generated by a tool such as `flex`. Every time the scanner recognizes a token, it must perform some action to store the token or pass it to a higher level part of the parser. The various token types and keywords of XML, used in distributed applications, can be defined as regular expressions. A DFA-based scanner can be custom-designed to process the subset of XML specification used in defining large-scale data files in applications. The DFA model for processing is efficient: each character in the input XML document is read only once, minimizing the overhead on a per-character basis.

The DFA approach does not directly lend itself to parallelism. It is required to start at the beginning of the input and process all the characters sequentially. As there is no way to determine in which state the DFA will be in after processing a certain section of the input, it is not possible to simply split the input in two (or more sections) and process the different sections independently. Due to this reason, all the widely used XML parsers are limited to a serialized indivisible scanner. This approach has thus far been acceptable for small files and desktop-style mass storage devices, because the scanner is fast for small input files. Additionally, this approach blends well with desktop mass storage access algorithms that work well reading from a single stream from disk.

3.1 Speculative NFA Execution in Piximal

Our parallelization tool, PIXIMAL, is designed for data-sets of applications running on cluster-class hardware, which are much more amenable to parallelization. In these target application cases, data sets defined in XML can be several hundreds of megabytes. Unlike the desktop case, in such applications mass storage is more likely to be arranged in higher performance configurations (e.g., RAID, NAS, SAN) which can more efficiently feed multiple data streams to concurrent threads. Our parallelization approach can be readily applied to these cases.

The speculative execution approach of PIXIMAL is to divide the input XML document, P, into N substrings, $P_1, P_2, ...P_N$. The processing on substring P_1 is carried out using the standard DFA-based lexical analyzer, as a DFA can only be run at the starting state using the first character of an input string. This DFA instance is termed the “initial DFA.” The other processing units in a multi-core processor are utilized by concurrently executing $N-1$ speculative scanners on the remaining substrings $P_2, P_3, ...P_N$. The processing is speculative as it is not possible to determine the start state for the L_{DFA}, except for P_1. As a result, we have added a transformation module to the PIXIMAL framework that can be applied to create a scanner, which can be applied to any of the substrings.

The DFA above is transformed into an NFA, L_{NFA} containing precisely the same state nodes, transitions, and final states as L_{DFA}. One significant change is made: each state node, with the exception of the error state, is marked as a start state. The parser built around this NFA reads each character of input, traversing along all *execution paths*, one for each state S_i. If a given transition triggers an action (such as triggering a StartElement SAX event in the user code), that action is stored into an *action list* A_{S_i} for that execution path, since it cannot be triggered immediately.

There is a single *correct execution path* which is the path started in state S_k, the state that the L_{DFA} would have been in had it parsed the input up to the beginning of this input substring. S_k will be known when the DFA or NFA running on the input behind it is complete and, if it is an NFA, knows its own correct execution path. Once S_k is known, the actions in action list A_{S_k} can be triggered, after some minor fix-up to merge the parser state from the previous automaton and the first action in this automaton's action list. This is necessary because the NFA may have started in the middle of a token, or more complexly, in the middle of an XML tag, which contains several tokens: a tag name and zero or more attribute name/value pairs. This fix-up is minor and a function to the number of automata used, as opposed to the size of the input, so can be viewed as a $O(1)$ cost once the number of available computing cores is set.

4. SERIAL NFA TESTS AND TESTING ENVIRONMENT

The tests presented here examine the fundamental hypothesis of this work: *the extra work required by using an NFA is offset by dividing processing work across multiple threads.* We run each component (automaton) of the PIXIMAL processor for a given configuration (*split percent* and *thread count*) independently on its element of the input partition, and examine the time each component takes to complete its processing sub-task. We run the test on several classes of homogeneously configured systems and average the results for *equivalent cases*. Equivalent cases here are those that are taken from the same class of computer systems running the same configuration and occur on the same subsequence of input. For each configuration, we calculate the maximum time over all automaton runs. The maximum time here represents the minimal time the complete parser would take to process the full input when running those automata concurrently on independent processors, minus the fixup time which is small. Each component performs all

```
0:  Initial State
1:  Enter Tag State
2:  Start Tag State
3:  Attribute Name State
4:  Attribute EQ State
5:  Attribute Value State
6:  Attribute Interstitial State
7:  Content State
8:  End Tag State
9:  End Tag Rest State
10: End Tag Interstitial State
```

Figure 1: Symbolic names of each DFA state, for reference when examining figures 5 and 9.

the work it must do in a multi-threaded PIXIMAL run, from reading input, to traversing the state table, to storing actions for each live execution path. However, the work is all done sequentially, in a single thread, to isolate each NFA in its own execution environment and obtain the best possible timing in the absence of other processes.

We present these results as *potential speedup*, which is calculated using the usual calculation for speedup by dividing the baseline time by the maximum time found above ($\frac{T_1}{T_N}$). We call these tests the *serial NFA tests*, as they measure the best potential speedup, using measurements taken from the serialized form of the parser.

In addition to "black box" performance tests, we examine the state usage for various inputs. Comparing state usage information is helpful in understanding why the performance varies for differently shaped input.

Some tests presented here use a collection of SOAP request documents, each of these encodes an array of certain type and length to demonstrate performance with respect to varying input size. The documents encode arrays of integers, strings, and "mesh interface objects" (MIOs – a complex type combining two integer values with a floating point value, often used in scientific computing). The array lengths range from 10 elements up to 50,000 elements. This allows us to examine documents ranging in size from a few hundred bytes to tens of megabytes. The integer and MIO arrays simply encode a variety of numbers. The string array encodes strings which are many times longer than the representations of the integers. This tests a hypothesis that if a document has substantially more PCDATA (character data between tags) than tags, then the NFAs' states will quickly collapse upon detection of the open angle bracket (<) which invalidates a large number of potential execution paths.

Another potential bottleneck of the PIXIMAL approach is the requirement that each NFA needs to frequently allocate memory to store actions along all live execution paths. `malloc(3)`, unless it is specially written, may be a hidden synchronization point, in order to protect access to the shared heap resource, that reduces concurrency. In addition to the serial tests described above, we tested PIXIMAL itself, with multiple NFAs running in concurrent threads, in the presence of the default GNU libc 2.7 malloc implementation as well as Google's Thread Caching malloc implementation to quantify the memory bottleneck in multi-core systems.

4.1 Experimental Environment

The serial NFA tests were run on a variety of system architectures, from older SMP machines to newer multi-core systems. Because the tests are serial and do not take advantage of any hardware concurrency, once the results were normalized by calculating the potential speedup, there was little detectable difference. Therefore, the data for the test results presented were collected by running the test on a ten nodes of a cluster of machines with dual-quad-core Intel Xeon E5345 chips clocked at 2.33GHz running Debian etch with Linux kernel 2.6.18. The input is read from local disk, though is expected (by pre-reading the input file before each test) to be in the system cache to eliminate I/O disturbances.

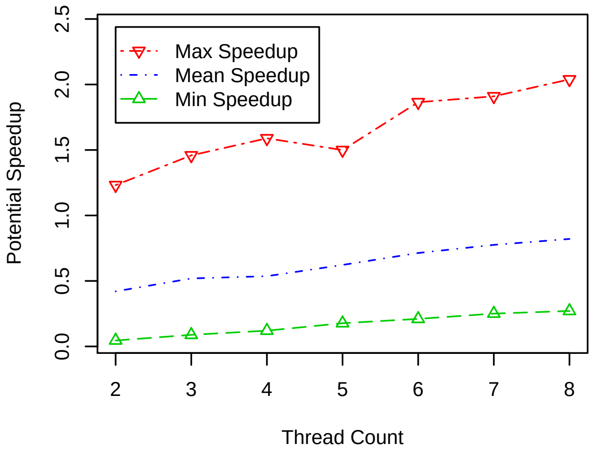

Figure 2: Potential scalability for XML input encoding an array of *10,000 integer* values. The number of threads available is the independent variable here. A slight speedup is possible by adding more threads for this class of input.

The malloc tests were run on a separate machine with a single quad-core Intel Xeon E5320 clocked at 1.86GHz, running Ubuntu 8.04LTS with Linux kernel 2.6.24. The input for this test is also pre-read to avoid noise from the I/O subsystem.

Performance analysis was performed and plots were created using R [16].

5. SERIAL NFA RESULTS

Figure 1 presents the symbolic names used in figures 5 and 9. The DFA we have built has eleven functional states and one (unlabeled) error state.

Figures 2, 3, and 5 present results for a representative input case: a SOAP-encoded array of 10,000 integers. Figure 2 presents potential speedup (the time it takes for a DFA to parse the input divided by the maximal time of each NFA component to parse its subsequence of the input) on this file. The "Max Speedup" line represents the potential speedup from the best possible selection of split percent, of those given in the range of test values, for each thread count. Similarly, the "Min Speedup" represents the speedup associated with the worst possible selection of split percent for each thread count. Of interest here is that there is a potential speedup available in all cases. The best potential speedup achieved on this input over the range of split percents and thread counts tests was 2.04 times the DFA baseline, splitting 34% of the input for the initial DFA and dividing the rest of the input evenly between the remaining 7 NFAs. Using four threads, there is a maximal speedup of 1.59 times the baseline, with 60% of the input being processed by the initial DFA, with the 3 NFAs each processing approximately 13%. It is particularly important to note that many split percent selections will lead to negative performance: input splitting greatly affects the performance. Figure 3 presents the same data as figure 2 along a different axis, tracking split percent rather than thread

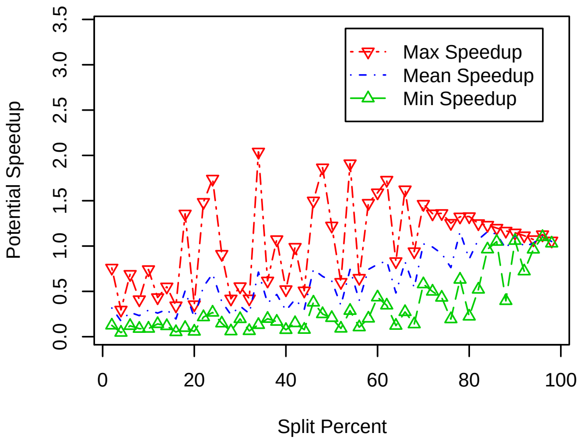

Figure 3: Similar to figure 2, this graph presents the result on speedup of varying the split percent parameter when processing an encoded array of *10,000 integers*. Maximal and minimal speedups for each selection of split percent are shown. The range of values comes from varying the number of threads.

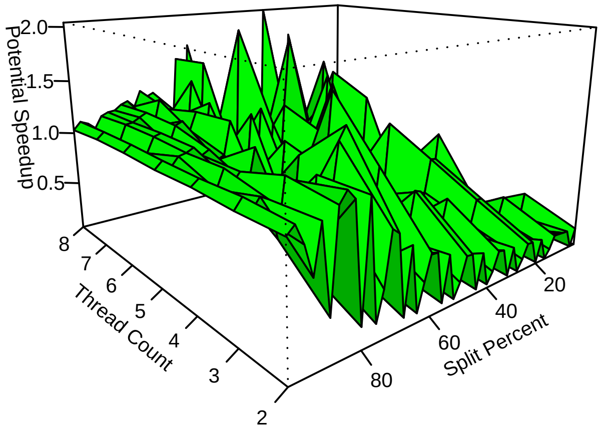

Figure 4: The "exploded" view of the data presented in figures 2 and 3 (arrays of *10,000 integers*). All points in the parameter space are presented to give a better view of the space.

count. Here the shape of the data is quite different. Some of the same high points are present here: the split percent of 34% is naturally still the global maximum, and the speedup at 60% is high here, too. Not all split percents have an associated thread count that provides any speedup. This again indicates that the partitioning of the input is critical to achieve performance gains with this approach.

Figure 5 depicts a histogram of the states used when parsing the encoded array of 10,000 integers. This gives some indication of why potential speedup caps out around 2.0 for this input. Most characters in this input are in PCDATA (content) sections, DFA state 7, which can be discerned by using figure 1. However, there is a significant number of characters which trigger state 1, the enter tag state. These represent open angle brackets in the input, and each one leads to an action (either a Start Element or and End Element SAX event). NFAs must store each one of these actions, so even in the best case, there is a linear relation between the amount of work the NFA must do and the number of times the DFA enters state 1.

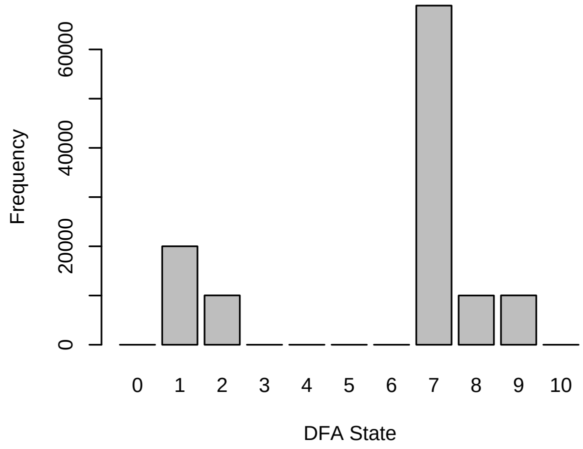

Figure 5: Histogram of DFA state usage when parsing the XML encoded array of *10,000 integers*. State 7 reprensents characters in PCDATA sections (text between tags). Other state names are described in figure 1.

Figure 6 presents the potential speedup achievable on an input SOAP-encoding of 10,000 strings in XML as a function of the number of threads scheduled. Compared with figure 2, the results reach a much higher global maximum and has a much greater rate of increase. The maximal performance achieved is found when processing 26% of the input with the initial DFA and dividing the remainder of the input evenly across 7 NFAs. The potential speedup over the baseline mean of DFA runs on the entire input is 3.17 times. It is also noteworthy that even the mean speedup is greater than 1 for many cases here.

Similarly to figure 3, figure 7 displays the potential speedup when reading an array of 10,000 strings as a function of a predetermined split percent. The results are much smoother for strings than for integers. Naturally, the high point here is the same as in figure 6, 26% with 8 threads, with a clear trend of results sloping up from both sides. This strongly indicates that 26% is nearly the optimal split percent. Further, this indicates that on this input, the NFA is doing roughly $\frac{26}{\frac{74}{7}} \approx 2.5$ times as much work as the DFA when the work is divided well.

Figure 9 indicates why the performance is so much more regular. The distribution of node usage is, by design, substantially different from the integer case. Nearly all characters of input are in content sections. Further, the actual file is frequently punctuated by tags. The input has long content sections and short elements, because it represents an array of lengthy strings. This means that the NFA will, with greater probability, start on a character in a content section and will quickly be able to eliminate most of the incorrect execution paths when the open angle bracket character is read, which will happen in a short amount of time. Thus, it is easy to "luck into" a good division of work due to the structure of the document. In the integer case, where content sections are shorter, there is a greater probability that the NFA will be started at some point in a tag where it is not possible to determine, for example, whether the correct execution path started in a content state or a tag state, because the close angle bracket character may legally appear in content sections. Thus, it does not lead to a contradiction in the way that encountering an open angle bracket does. Performance is similar for the MIO array input because its XML representation more closely matches the representation of integer arrays.

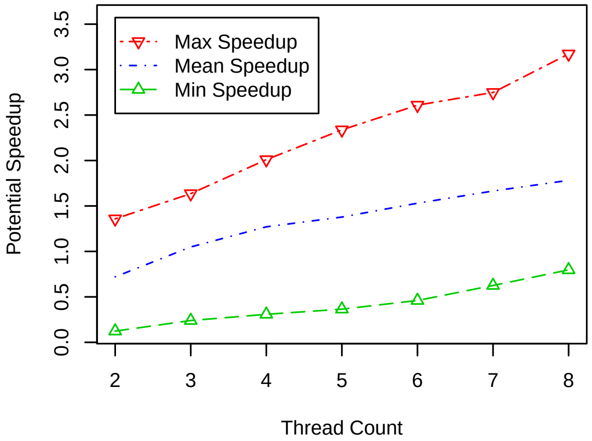

Figure 6: Potential speedup on input of an XML-encoded array of *10,000 lengthy strings* as a function of number of threads simulated. Compared to the integer array examined in figure 2, the results here are smoother and exhibit greater overall speedup, even in the worst case.

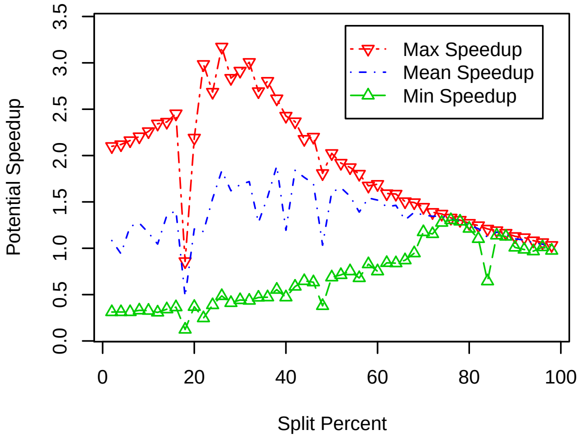

Figure 7: Potential P<span>IXIMAL</span> **speedup on arrays of *10,000 strings* as a function of split point chosen. The contrast with integer arrays (figure 3) is more stark here. The results are much more regular, with a clear peak around 26%.**

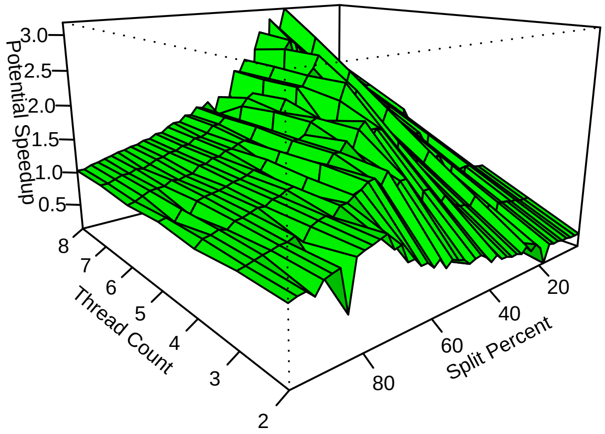

Figure 8: The "exploded" view of the data presented in figures 6 and 7 (arrays of *10,000 strings*). Compared with the integer data presented in figure 4, the result space is much smoother everywhere.

Figures 10, 11, and 12 examine the effect of increasing the input size. Earlier figures examine the shape of performance for particular array sizes, breaking down how the number of threads and divisions of work affects potential speedup. Here the results are aggregate over many runs and many sizes. As mentioned above, MIO and integer arrays see similar results, with the maximal potential speedup trending between 1.5 and 2.0 times the performance of the DFA. Mean and minimal performance for these cases are uniformly low.

Again the string case shows the benefit of its specialized form. Maximal performance in figure 11 is more uniform and hovers around 3.0-3.2. Even the mean performance shows some speedup across all input sizes.

Figure 13 shows the performance difference when running fully concurrent PIXIMAL with and without a specialized malloc implementation on a SOAP-encoded array of 25000 strings. The default malloc here is that which is included with GNU libc 2.7 on Ubuntu 8.04LTS. The results presented are mean timing values over several full PIXIMAL runs. While there is a global performance win for Google's thread caching malloc, this does not translate to increased improved performance as more threads are used, which indicate that heap usage is not a limiting factor here. This is due to the fact that PIXIMAL uses `mmap(2)` to access the input file as a global memory block and utilizes a zero-copy regime to ensure that all strings refer to this single segment of memory to minimize heap usage. The strings used within PIXIMAL are not traditional "C strings" which are NULL-terminated chunks of memory referred to with memory pointers, rather they are "Pascal strings" represented by a data structure containing an integral length and pointer to the start of the string. This allows the list of stored actions to be as small as possible and eliminates almost all memory duplications in the parser. An implementation which incurs a significant amount of memory copies, say several per byte of input, might encounter a greater bottleneck with respect to heap contention, and alternative malloc implementations might ameliorate such problems.

6. CONCLUSIONS

The form of the input XML data affects the overall gains with the PIXIMAL approach. Large data sets with more text content and shorter tags work best, whereas documents that encode more infor-

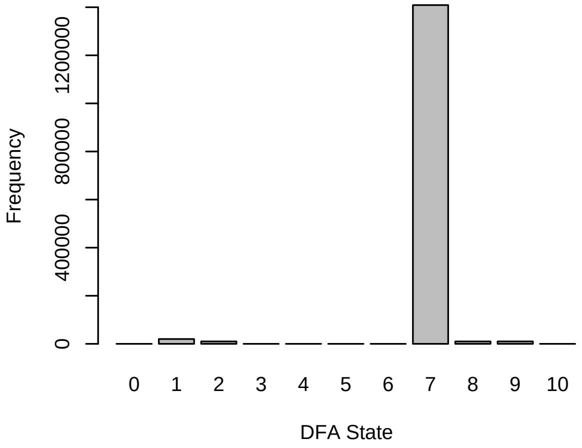

Figure 9: Histogram of state usage for arrays of *10,000 lengthy strings*. The underlying reason for the more regular speedup for arrays of strings over arrays of integers is apparent: most characters in this input are in PCDATA sections (state 7), thus a given NFA is much more likely to start at a character in the input and its execution paths will quickly collapse when a < character is encountered.

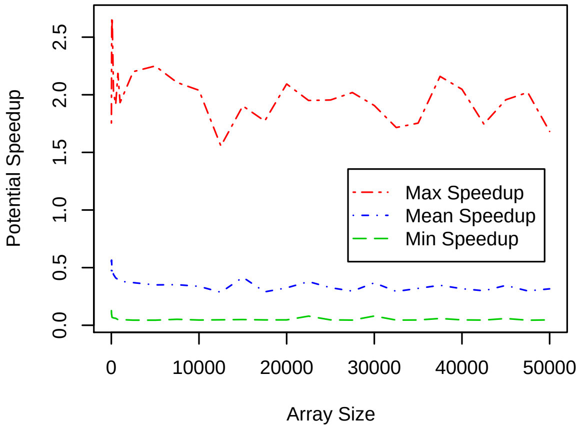

Figure 10: Effect of scaling up the size of an *integer* array SOAP-encoded in XML on potential scalability in PIXIMAL.

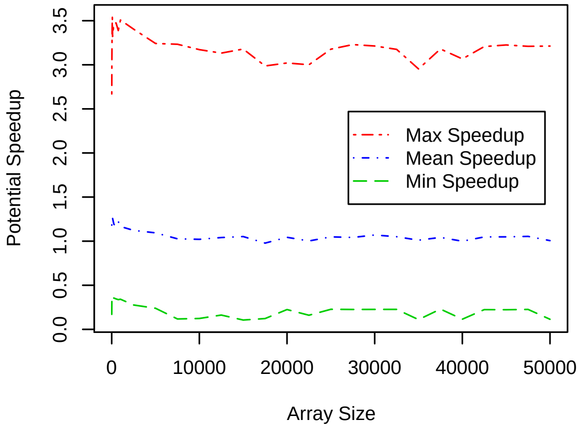

Figure 11: Effect of scaling up the size of a *string* array SOAP-encoded in XML on potential scalability in PIXIMAL.

mation into elements and attributes will prevent the XML processor from providing speedup. This is due to the nature of XML and the number of transitions that lead to a contradiction in the DFA. Nevertheless, even with inputs which lead to suboptimal performance, it should still be possible to split the input to be able to make use of extra processing – speculatively pick a character in a PCDATA text section.

The PIXIMAL approach to reading large-scale structured data files, such as XML documents, effectively uses the available cores on a node. Based on our tests on a variety of CPU configurations, we conclude that even with current memory and I/O subsystems, processing large scale data files can potentially provide performance improvements. Memory bandwidth between the processor and memory mapped files is not a strict limiting factor on thread scalability.

Starting in a content section of an XML data file is beneficial because the '<' character eliminates a large number of execution paths through the NFA. If '>' could be treated similarly by the XML processor, starting in a tag would be less harmful.

If restrictions could be placed on the features of XML specification that are used in the input XML data, even greater performance could be achieved with the PIXIMAL approach because more contradictory transitions could be found. One possible restriction would be to convert all attributes to nested tags and eliminate whitespace and attributes. This would both increase the number of characters of input that would be found in the content state and allow the XML processor to know that whitespace characters must occur in the content state. If this is considered too restrictive for designers of XML datasets, simply requiring that each, for example, millionth character occur in a text section will be useful. This would allow the input to be divided such that the NFA would be known to start processing only in the content state, greatly reducing the amount of work it needs to do. Indeed, it need not be considered an NFA given that supposition, just a DFA that queues parser events.

The PIXIMAL framework can be used to determine the most optimal way to split an XML data input file to obtain the best possible speedup. Performance results for commonly used data structures, such as arrays of integers, strings, and MIOs, indicate that grid application programmers need to carefully choose the split percentage and number of processing threads for the target grid infrastructure nodes. Naïvely dividing the input may lead to a slowdown.

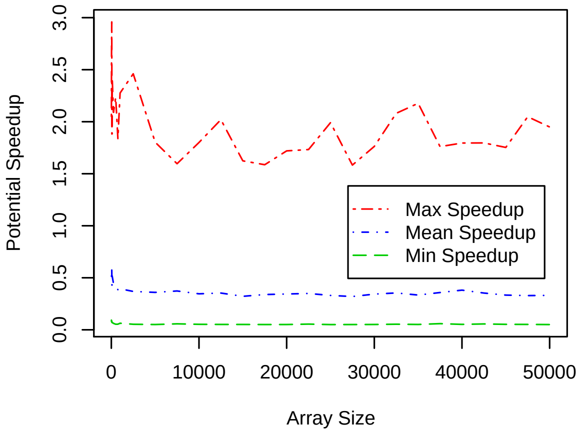

Figure 12: Effect of scaling up the size of a *MIO*s (mesh interface objects) array SOAP-encoded in XML on potential scalability in PIXIMAL.

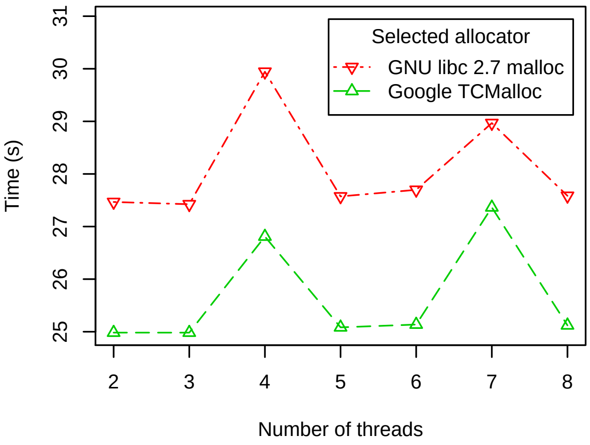

Figure 13: Comparison of overall parse times when running PIXIMAL concurrently with two different `malloc(3)` implementations, GNU libc 2.7 and Google's Thread Caching malloc. The XML input for this case encodes an array of *25,000 strings*.

For arrays of integers commonly used in grid applications, a speedup by a factor of 2 can be obtained. As explained by Amdahl's law, the speedup is limited by the sequential fraction of the program. The sequential aspect for XML data processing includes access to main memory for shared data structures, resolving namespaces that may have dependencies, and updates of data structures to keep track of the automatons that need to be stored and the ones that need to be discarded. The performance results of commonly used data structures in scientific computing, MIOs, is similar to that of array integers as the XML representations are quite similar.

For XML data sets that primarily consist of arrays of strings, a greater overall speedup can be obtained. On an 8-core machine, the best speedup is achieved when the initial DFA thread processes 26% of the input, while the rest of the 7 NFAs speculatively process the rest of the input data. Arrays of strings allow for quicker elimination of incorrect execution paths and hence lead to overall performance gains.

The implementation of memory allocation libraries can make a difference in the overall performance on multi-core architectures. We observed that Google's thread caching malloc performs better than the GNU libc malloc library, widely used in grid applications. However, if zero-copy methods are used, such as `mmap(2)` for reading the input XML data, this performance gain does not improve with increase in the number of threads.

7. FUTURE WORK

In future work we plan to explore pre-fetching and piped implementation techniques that can enhance the performance of PIXIMAL. We will study the effect of operating system-level caching on the processing of large documents that may be read more than one time. We will develop algorithms for optimal layouts of DFA tables in memory to efficiently process frequently occurring transitions. We will build a MapReduce extension of PIXIMAL to process large documents that are stored in a distributed file system. We will further study the scalability of PIXIMAL as processors with multiple cores (greater than 8) are available for research and testing purposes on grid infrastructures.

7.1 Algorithmic Improvement Opportunities

The PIXIMAL design allows a wide range of application-specific optimizations to be carried out. Execution of each L_{NFA} results in consumption of CPU and memory resources. The worst case scenario is the execution of all possible L_{NFA} for any given XML input. The approach to optimizing the parallelization model is therefore to reduce the number of L_{NFA} to be executed using heuristic methods or by elimination of paths due to other considerations. For example, the execution time depends on the starting point for a L_{NFA}. If it starts just after a sentinel character such as '<', it reduces the number of start states that need to be considered. However, given that many execution paths simply can not start with a '<', it may be more efficient to first scan backwards a little from the starting point and potentially reduce the number of execution paths.

In addition to investigating cases such as the above, there are deeper topics to investigate:

Minimize memory load in NFA. Another approach to optimize the memory overhead is to process the input in chunks to reduce NFA memory load. Partitioning the entire input into N parts and processing the entire document in parallel leads to memory usage proportional to input size. To address this memory usage, it is more efficient to divide the input into more manageable chunks, small enough to fit in cache or at least to fit in main memory without requiring a virtual memory swap, and process each of these chunks in a sequence of parallel parsing steps. This requires dividing the input evenly into M sections, each of size T. This sets a bound on memory usage for the NFA action history, because each NFA will only process a limited size input and guarantees that no NFA will need more than a finite $f(T)$ memory to operate. If all the processing is balanced, there should be little overhead compared to the un-optimized PIXIMAL case.

Convert L_{NFA} into a DFA. In this approach, care needs to be taken to ensure that action history for each execution path is maintained properly, but it should be possible to transform the L_{NFA} into an equivalent DFA and reduce it using the standard algorithms to optimize the run time processing overhead.

Optimize DFA Table layout. To maximize the number of cache hits while traversing the DFA table, it is helpful to profile the usage (which states most frequently transition to other states) and rearrange the state layout in memory to keep those states near each other in the table.

Integrate with application level actions to eliminate the action queue. In some applications, such as XML transformation or format conversion, it may be possible to trigger the actions directly from the NFA, rather than queueing them. An XSLT processor could begin writing to multiple files, one for each execution path of each NFA as SAX events are triggered. The application would need to perform its own fix-up routines when the correct execution paths are known, but this would convert the memory load into disk usage.

8. REFERENCES

[1] N. Abu-Ghazaleh and M. J. Lewis. Differential Deserialization for Optimized SOAP Performance. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 21, Washington, DC, USA, 2005. IEEE Computer Society.

[2] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall Professional Technical Reference, 1972.

[3] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the Limits of SOAP Performance for Scientific Computing. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 246, Washington, DC, USA, 2002. IEEE Computer Society.

[4] J. Clark. Expat is an XML parser library written in C. `http://expat.sourceforge.net/`.

[5] J. Cohen, T. Hickey, and J. Katcoff. Upper Bounds for Speedup in Parallel Parsing. *Journal of the ACM*, 29(2):408 – 428 , April 1982.

[6] D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, and A. Slominski. On Building Parallel and Grid Applications: Component Technology and Distributed Services. In *Proceedings of the Second International Workshop on Challenges of Large Applications in Distributed Environments (CLADE 2004)*, pages 44–51, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[7] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and evaluation of RMI protocols for scientific computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomput*, page 61, Washington, DC, USA, 2000. IEEE Computer Society.

[8] T. Gross, A. Sobel, and M. Zolg. Parallel compilation for a parallel machine. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 91–100, New York, NY, USA, 1989. ACM Press.

[9] M. R. Head and M. Govindaraju. Parallel Processing of Large-Scale XML-Based Application Documents on Multi-Core Architectures with PiXiMaL. In *IEEE Fourth International Conference on eScience*, pages 261–268, Indianapolis, IN, December 2008.

[10] M. R. Head, M. Govindaraju, A. Slominski, P. Liu, N. Abu-Ghazaleh, R. van Engelen, K. Chiu, and M. J. Lewis. A Benchmark Suite for SOAP-based Communication in Grid Web Services. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 19, Washington, DC, USA, 2005. IEEE Computer Society.

[11] M. R. Head, M. Govindaraju, R. van Engelen, and W. Zhang. Benchmarking XML Processors for Applications in Grid Web Services. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 121, New York, NY, USA, 2006. ACM Press.

[12] H. P. Katseff. Using data partitioning to implement a parallel assembler. In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 66–76, New York, NY, USA, 1988. ACM Press.

[13] W. Lu, K. Chiu, and Y. Pan. A Parallel Approach to XML Parsing. In *7th IEEE/ACM International Conference on Grid Computing (Grid 2006)*, pages 223–230, 2006.

[14] Y. Oren. Piccolo is a small, extremely fast XML parser for Java, 2006. `http://piccolo.sourceforge.net/`.

[15] Y. Pan, Y. Zhang, K. Chiu, and W. Lu. Parallel XML Parsing Using Meta-DFAs. In *IEEE Third International Conference on eScience and Grid Computing*, pages 237–244, December 2007.

[16] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2007.

[17] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A Metadata Catalog Service for Data Intensive Applications. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 33, Washington, DC, USA, 2003. IEEE Computer Society.

[18] Trolltech. API Documentation for QtXml Module, 2007. `http://doc.trolltech.com/4.2/trolltech.html`.

[19] R. van Engelen. Constructing Finite State Automata for High Performance XML Web Services. In *International Symposium on Web Services*, pages 975–981, 2004.

[20] R. van Engelen. gSOAP: C/C++ Web Services and Clients, 2007. `http://www.cs.fsu.edu/~engelen/soap.html`.

[21] D. Veillard. The XML C parser and toolkit of Gnome, 2006. `http://xmlsoft.org/`.

[22] Xerces-J. Xerces2 Java Parser 2.9.0 Release, 2006. `http://xerces.apache.org/xerces2-j/`.

[23] J. Zhang. Project Homepage of VTD-XML, 2007. `http://vtd-xml.sourceforge.net/`.

[24] W. Zhang and R. A. van Engelen. A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 197–204, Los Alamitos, CA, USA, 2006. IEEE Computer Society.