ANALYSIS AND OPTIMIZATION FOR PROCESSING GRID-SCALE XML DATASETS

BY

MICHAEL REUBEN HEAD

BS, Harpur College, Binghamton University, 1999
BS, Watson School, Binghamton University, 1999
MA, Brandeis University, 2004

DISSERTATION

Submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2009
Accepted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in Computer Science
in the Graduate School of
Binghamton University
State University of New York
2009

November 30, 2009

Madhusudhan Govindaraju, Department of Computer Science, Binghamton University

Leslie Lander, Department of Computer Science, Binghamton University

Michael Lewis, Department of Computer Science, Binghamton University

Kenneth Chiu, Department of Computer Science, Binghamton University

Fernando Guzman, Department of Mathematics, Binghamton University
Abstract

In the field of Scientific Computing, two trends are clear: the size of data sets in use is growing rapidly and microprocessor performance is improving through increases in parallelism, rather than through clock rate increases. Further, Extensible Markup Language (XML) is increasingly being used to encode large data sets, and SOAP is being used to provide Grid services – uses XML and SOAP were never designed for, and naïve implementations of these standards can lead to performance penalties. As these trends continue, past assumptions about the value of seeking out parallel algorithms should be revisited.

Lexical analysis has traditionally been seen as an inherently serial process. This work seeks to challenge that viewpoint. We start by tracking the performance of state of the art in XML parsers and SOAP toolkits through benchmarks for scientific computing applications. We continue to study the space through an examination of the effects of current workstation- and server-class computer systems’ caching mechanisms on parser performance. Finally, we propose Piximal, an NFA-based parser which uses spare processors to reduce XML parse time. The limits of the Piximal approach to parallel XML parsing are examined.
Dedication

I dedicate this dissertation to my loving wife Shprese Demiri-Head. Without her support through these many years of school, I could not have come as far as I have. Without her determination for me to graduate, I suspect I would still be a student after the turn of the decade.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>vi</td>
<td>vi</td>
</tr>
</tbody>
</table>

## 1 Introduction

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Performance Analysis of XML and SOAP</td>
<td>1</td>
</tr>
<tr>
<td>1.2 On-chip Multiprocessing and XML Processors</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Thesis Contributions</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Thesis Statement</td>
<td>4</td>
</tr>
</tbody>
</table>

## 2 Related Work

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Benchmarks in HPC</td>
<td>5</td>
</tr>
<tr>
<td>2.3 Other XML and SOAP benchmark suites</td>
<td>5</td>
</tr>
<tr>
<td>2.4 Table-driven DFA-based parsing</td>
<td>6</td>
</tr>
<tr>
<td>2.5 Schema specific parsing within a generated scanner</td>
<td>6</td>
</tr>
<tr>
<td>2.6 Parallelized DOM-based parsing: MetaDFA</td>
<td>6</td>
</tr>
</tbody>
</table>

## 3 Benchmarking XML Datasets

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 XML and SOAP Benchmarks</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Design of the Benchmark Suite for XML Processing</td>
<td>8</td>
</tr>
<tr>
<td>3.2.1 Feature probes</td>
<td>8</td>
</tr>
<tr>
<td>3.2.2 Application class benchmarks</td>
<td>10</td>
</tr>
<tr>
<td>3.3 Representative Performance Results</td>
<td>11</td>
</tr>
<tr>
<td>3.4 Recommendations for XML Application Developers</td>
<td>15</td>
</tr>
<tr>
<td>3.5 XML Toolkit Design for Multi-core architectures</td>
<td>19</td>
</tr>
<tr>
<td>3.6 SOAP Benchmark Suite</td>
<td>19</td>
</tr>
<tr>
<td>3.6.1 Serialization, deserialization and round-trip performance</td>
<td>19</td>
</tr>
<tr>
<td>3.6.2 Candidate features for optimizations</td>
<td>21</td>
</tr>
<tr>
<td>3.6.3 Latency</td>
<td>22</td>
</tr>
<tr>
<td>3.6.4 Binary attachments via DIME and MIME</td>
<td>23</td>
</tr>
<tr>
<td>3.6.5 Dynamic vs static code generation</td>
<td>23</td>
</tr>
<tr>
<td>3.7 Performance</td>
<td>23</td>
</tr>
<tr>
<td>3.7.1 Summary of performance results</td>
<td>27</td>
</tr>
<tr>
<td>3.8 Observations on Current Toolkits based on Benchmark Results</td>
<td>29</td>
</tr>
</tbody>
</table>

## 4 Utilizing Spare Cores to Read Ahead

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>36</td>
</tr>
<tr>
<td>4.2 Architectural Changes</td>
<td>36</td>
</tr>
<tr>
<td>4.3 Performance Results</td>
<td>37</td>
</tr>
<tr>
<td>4.3.1 Precached data</td>
<td>38</td>
</tr>
<tr>
<td>4.3.2 Uncached data</td>
<td>38</td>
</tr>
<tr>
<td>4.4 Conclusion</td>
<td>39</td>
</tr>
</tbody>
</table>

## 5 Parallel Processing with PIXIMAL

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>44</td>
</tr>
<tr>
<td>5.2 A hand-coded DFA-based parser: PIXIMAL-DFA</td>
<td>44</td>
</tr>
<tr>
<td>5.3 A parallelized NFA-based parser: PIXIMAL-NFA</td>
<td>44</td>
</tr>
<tr>
<td>5.3.1 Division of work</td>
<td>46</td>
</tr>
<tr>
<td>5.3.2 Contrast with Meta-DFA</td>
<td>46</td>
</tr>
<tr>
<td>5.3.3 Optimization opportunities with PIXIMAL</td>
<td>46</td>
</tr>
<tr>
<td>5.4 PXML: An XML-like Language Optimized for PIXIMAL</td>
<td>47</td>
</tr>
</tbody>
</table>
# 6 Limits of Parallel Processing with PIXIMAL

1 The Limits of PIXIMAL .......................................................... 49
2 Examining Memory Bandwidth and DFA Size ............................ 49
   2.1 Memory bandwidth test .................................................. 49
   2.2 State scalability test ..................................................... 50
3 Bandwidth and Size Test: Results and Discussion .................... 50
   3.1 Memory bandwidth results ............................................. 50
   3.2 State scalability results ................................................ 54
4 Serial NFA Tests ................................................................. 58
   4.1 Experimental environment ............................................. 60
   4.2 Serial NFA results ....................................................... 60
5 Conclusions ................................................................. 63

# 7 Conclusion and Future Work

1 Future Work ................................................................. 72
   1.1 Implement PIXIMAL using a MapReduce-style framework ....... 72
   1.2 Build a non-blocking I/O parser ....................................... 72
   1.3 Zero-copy parsing with multiple network input streams ......... 72
   1.4 Consider patch to an existing lexical analyzer generator ....... 73
2 Conclusion ................................................................. 73

Bibliography ................................................................. 74
List of Figures

3.1 The overhead associated with each parser. We run a tiny XML file through each parser 20 times and measure the parse time. Because the XML file is so small, this effectively measures each parser’s setup and cleanup time. gSOAP’s overhead is the lowest at 110 µs. Xerces-J-DOM’s overhead is twice that of Xerces-J-SAX at 7029 µs. 12

3.2 Performance of C/C++-based parsers on some large grid applications. Files sizes range from 277KBytes (workflow_P1W.xml) to 4.9MBBytes (hapmap_I797SNPs.xml) and are parsed 20 times in succession. All parsers processed the HapMap file in approximately 2s, with the exception of Xerces-C-DOM, which took about 5s. 13

3.3 Scalability of C/C++-based parsers over arrays of doubles in SOAP payloads. Here the parsers are fed XML documents containing SOAP-serialized arrays of doubles. Expat leads the group parsing a document containing 100,000 doubles 20 times in 744ms. Xerces-C-DOM generates a DOM each parse, and performs the same task in 5,965ms. 14

3.4 Scalability of C/C++-based parsers over arrays of integers in SOAP payloads. Similar to Figure 3.3, we test each parser against a set of XML documents containing SOAP-serialized arrays of varying size. In contrast to Figure 3.3, all parsers improve when handling integers versus doubles, though gSOAP and Qt4-SAX both improve more than the others. 14

3.5 C/C++-based parsers using WS-MG notification messages. Again, Expat is the best performing parser, processing the WS-MG notification message 20 times in 1.48ms. Xerces-C-Dom tops the chart at 7.31 ms. 14

3.6 C/C++-based parsers using WSSE security messages. The results are similar to those of Figure 3.5, except that Qt4-SAX performs the worst at 10.2 ms. 15

3.7 Performance of Java-based parsers on some large grid applications. This is the same test as shown in figure 3.2, using Java-based parsers. There is some interesting variability here. XPP3 handles the workflow tests in roughly 10% the time of the other parsers, but is squarely in the middle of the group for the HapMap and Molecule tests. 16

3.8 Scalability of Java-based parsers over arrays of integers in SOAP payloads. The same test as figure 3.4 for parsers written in Java. We see that Piccolo and XPP3 are equivalent here. 16

3.9 Scalability of Java-based parsers over arrays of strings in SOAP payloads. Similar to the other SOAP payload tests, here the elements in the arrays are text strings, as opposed to textual representations of numbers. 17

3.10 Java-based parsing of WS-MG notification messages. The same tests as shown in figures 3.5 and 3.6, applied to Java-based parsers. Here Piccolo and XPP3 again show much better performance than either Xerces-J-DOM or Xerces-J-SAX. XPP3 parses the message in 30% of the time it took Xerces-J-SAX, which has a similar programming model. 17

3.11 TDX parser vs. other C/C++-based parsers decoding a SOAP payload containing an array of strings. TDX combines validation with parsing. It’s table-driven design enables it to perform parsing and validation in less than half the time that it takes expat to parse without validation. 18

3.12 This graph shows the effect on serialization when the size of an integer array is scaled. The performance of AxisJava and XSUL is similar. bSOAP and gSOAP complete the benchmark in 4% and 17% respectively of the time it takes for AxisJava for the largest size (100,000 integers). 24

3.13 Here we compare the serialization performance of four toolkits on arrays of doubles. As with the integer case in Figure 3.12, the Java-based toolkits (AxisJava and XSUL) perform similarly. bSOAP and gSOAP send the largest size (25,000 doubles) arrays in 13% and 30%, respectively, of the time it takes for AxisJava to do the same. 24

3.14 Here we compare the deserialization performance of AxisJava, gSOAP and XSUL. Each toolkit is sent a SOAP payload for double arrays of various sizes, asked to deserialize it and return it’s size to the driver. For arrays of 10,000 doubles, AxisJava takes 5.1 times as long to respond as XSUL, which in turn takes 7.8 times as long to respond as gSOAP. 25
3.15 This graph compares the deserialization performance for strings. Again, XSUL performs significantly better than AxisJava for deserialization. For an array size of 25,000, it takes AxisJava 7.4 times longer to respond than XSUL, and XSUL takes 3.6 times as long to respond as gSOAP.

3.16 This graph compares deserialization performance of Axis, XSUL and .NET for an array of doubles on Windows. XSUL and .NET are comparable, while AxisJava does not scale well for large array sizes. For an array of 10,000 doubles, AxisJava’s deserialization time is 6 times greater than those of XSUL and .NET.

3.17 We compare the end-to-end performance for arrays of Mesh Interface Objects (2 integers and a double). The plots show that for sizes greater than 10,000 objects, XSUL’s performance considerably degrades compared to AxisJava. For 10,000 elements, XSUL and AxisJava respectively take 11.7 and 12.3 times more time to execute the benchmark compared to gSOAP.

3.18 This graph compares the performance for the deserialization benchmark for events. Each event object contains an integer, a double and a string. XSUL’s performance degrades considerably when it deserializes more than 15,000 elements, but for lesser number of events, it outperforms AxisJava. gSOAP is orders of magnitude faster in handling complex types compared to the Java toolkits.

3.19 We compare end-to-end (serialization and deserialization) for .NET, AxisJava, and XSUL on Windows. XSUL’s performance drops for handling more than 10,000 events. For sizes greater than 12,000, AxisJava takes an average of 50% more time than .NET to respond.

3.20 This graph shows the performance results for end-to-end performance of binary data that has been base64 encoded (a feature provided by each of the toolkits). XSUL outperforms AxisJava, while gSOAP is consistently better than XSUL. For an array of size 10,000, XSUL is slower than gSOAP by a factor of 1.6, while AxisJava is slower than XSUL by a factor of 2.3.

3.21 This graphs shows the performance of AxisJava, gSOAP and XSUL for serializing data in Base64 format. AxisJava performs poorly compared to XSUL and gSOAP. XSUL takes 49% more time to complete than gSOAP for arrays of 25,000 elements.

3.22 In this graph we show the end-to-end performance for array of integers for AxisJava, XSUL and gSOAP toolkits. This requires each toolkit to deserialize, then re-serialize the input array. For an array of size 10,000 elements, AxisJava’s time is a factor of 3.4 greater than XSUL’s. For the same size, XSUL’s time is a factor of 6.2 greater than that of gSOAP.

3.23 Similar to Figure 3.22, but using doubles instead of integers, we compare end-to-end performance. For an array of size 10,000 elements, the time taken by AxisJava is a factor of 3.0 more than XSUL. For the same size, AxisJava takes a factor of 3.9 times more than gSOAP.

3.24 We studied the effect of streaming by making AxisJava and gSOAP deserialize a large number of events with and without streaming enabled. AxisJava consistently performs better when streaming is used, taking 22% less time to complete for the 10,000 element array. gSOAP also trims its time with chunking on this test by 22%.

3.25 We studied the effect of streaming by making AxisJava and gSOAP serialize arrays of doubles with and without streaming enabled. Unlike in the case of deserialization of events (see Figure 3.24), streaming does not improve AxisJava’s serialization performance. Streaming helps improve gSOAP’s performance for all sizes up to 25,000 elements, the maximum we tested for this benchmark. By enabling streaming, gSOAP’s average response time for 10,000 elements was reduced by 42%.

3.26 Differential serialization is optimized for use-cases when subsequent sends are similar. In this benchmark, we measured the performance of bSOAP when 0% (bSOAP0), 25% (bSOAP25), 50% (bSOAP50), 75% (bSOAP75), and 100% (bSOAP100) of the values need to be re-serialized, rather than copied, for each message from the previous message. As expected, bSOAP’s optimizations reduce the serialization time as the percentage of values that need to be re-serialized is reduced. For 100,000 doubles, the best case bSOAP0 is faster than the worst case bSOAP100 by a factor of 6.7.

4.1 Rather than seeing any improvement, Readahead exhibits a slowdown on all architectures when the input is cached.
4.2 Both parsers exhibit a slowdown on the uniprocessor machine, Readahead more than Runahead.

4.3 On CMP, the difference between the two parsers is a bit more pronounced, with Runahead providing a minimal speedup and Readahead exhibiting slowdown.

4.4 Runahead really shines here, providing a speedup of nearly 1.4 for the SMP machine – note that the SMP machine has a 15000 RPM Ultra320 SCSI drive while the other two have 7200 RPM SATA drives.

4.5 As in figure 4.4, Runahead provides a very nice speedup on the SMP machine with a very fast hard drive when the file must be read from disk.

4.6 On the dual core machine, Runahead does provide some speedup, around 1.08, while readahead again exhibits (some) slowdown.

5.1 Graph representing the DFA used in PIXIMAL, state 0 is the start state, and state 7 is the final state.

5.2 Symbolic names of each DFA state, for reference when examining figures 6.17 and 6.21.

5.3 Graph representing the DFA used in PIXIMAL-PXML, state 0 is the start state, and 3 is the final state.

5.4 Symbolic names of each DFA state.

6.1 Input read times of the memory bandwidth test for each split_percent and thread count on a standard (2 × uniprocessor). Note the valley when two threads are used with a 50% input split.

6.2 Bandwidth achieved at each split_percent and thread count on the 2 × uniprocessor testbed. In contrast to figure 6.1, the axes are inverted to provide a better view of the space. The maximum bandwidth achieved was 0.94 GiB/s.

6.3 Input read times of the memory bandwidth test on an 2 × dual core configuration (4 total cores). Note that performance increases past the valley seen in figure 6.1.

6.4 Bandwidth achieved at each test point on the 2 × dual core testbed. The axes are inverted from figure 6.3 to better show the data. The maximum bandwidth achieved was 2.28 GiB/s.

6.5 Input read times of the memory bandwidth test on an 2 × quad core configuration (8 total cores). Note that adding threads continues to improve performance, though returns diminish past the 6th thread.

6.6 Bandwidth achieved at each point on the 2 × quad core testbed. The maximum bandwidth achieve in this space was 2.69 GiB/s.

6.7 Speedup comparison on the 2 × dual core (4 total cores) configuration for the memory bandwidth test. split_percent s are selected to maximize the speedup for each number of threads.

6.8 As in figure 6.7, but for an 2 × quad core (8 total cores) configuration. As was visible in figure 6.5, performance gains trail off after 6 threads are given to the test.

6.9 Best speedup achieved for each processor configuration, together with the split_percent that achieves that maximal speedup. Adding more cores continues to improve performance, allowing a greater portion of the input to be divided up. Memory bandwidth between the processor and the mmap(2) ed file is not a strict limiting factor on thread-scalability.

6.10 The lowest input read times for all DFA sizes and thread counts for the 2 × dual core (4 total cores) configuration. For a given DFA size and number of threads, there is a range of possible split_percent s that could be chosen. The split_percent is chosen for each configuration of DFA size and number of threads to minimize the time.

6.11 Input total read times for 2 × quad core (8 total cores) configurations. In contrast to figure 6.10, there is a continued performance increase over the space of parameters tested.

6.12 Speedup for a variety of DFA sizes for 2 × dual core (4 total cores) configurations. This test models a part of the extra work done by the NFAs over the DFA. For each DFA size, the split_percent is chosen which provides the maximal speedup for some number of threads. It is clear that NFAs built from larger sized DFAs incur stuff performance penalties.

6.13 Same as figure 6.12, but for the 2 × quad core (8 total cores) configuration. 6 states appears to be a good DFA size for this particular model of the parallel parser.
6.14 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.

6.15 Similar to figure 6.14, 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.

6.16 The “exploded” view of the data presented in figures 6.14 and 6.15 (arrays of 10,000 integers). All points in the parameter space are presented to give a better view of the space.

6.17 Histogram of DFA state usage when parsing the XML encoded array of 10,000 integers. State 7 represents characters in PCDATA sections (text between tags). Other state names are described in figure 5.2.

6.18 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 6.14, the results here are smoother and exhibit greater overall speedup, even in the worst case.

6.19 Potential PIXIMAL speedup on arrays of 10,000 strings as a function of split point chosen. The contrast with integer arrays (figure 6.15) is more stark here. The results are much more regular, with a clear peak around 26%.

6.20 The “exploded” view of the data presented in figures 6.18 and 6.19 (arrays of 10,000 strings). Compared with the integer data presented in figure 6.16, the result space is much smoother everywhere.

6.21 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.

6.22 Effect of scaling up the size of an integer array SOAP-encoded in XML on potential scalability in PIXIMAL.

6.23 Effect of scaling up the size of a string array SOAP-encoded in XML on potential scalability in PIXIMAL.

6.24 Effect of scaling up the size of a MIOs (mesh interface objects) array SOAP-encoded in XML on potential scalability in PIXIMAL.

6.25 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.
Chapter 1
Introduction

Over the past few years, designers of Web services have closely collaborated with the grid community to propose numerous Extensible Markup Language (XML) based protocol specifications to bridge the platform and programming language gap in heterogeneous wide-area systems. The recently adopted standards such as the Open Grid Services Architecture (OGSA) [31] and Web Services Resource Framework [81] define a set of standard interfaces and behaviors of grid services in terms of Web services based technologies. Some of the other important standards and specifications in the Web services space include Web Services Description Language (WSDL) [18], SOAP (formerly, Simple Object Access Protocol) [37], Business Process Execution Language for Web Services (BPEL4WS) to orchestrate workflows, and WS-Security set of XML specifications. Additionally, many grid applications use well-defined XML schemas for the XML datasets used in various parts of the application.

XML is a ubiquitous tree-oriented data representation language. XML has many important features, including platform and language independence, flexibility, expressiveness, and extensibility. Thus, the combination of these characteristics with the interoperability trait of Web services is an attractive way to compose distributed applications. Additionally, the use of XML based protocols for security, routing, messaging, resource policies, workflows, events, and other tasks, provides an effective platform to build applications over computational grids [11, 30].

1 Performance Analysis of XML and SOAP

Various studies [2, 17, 35] have shown that the use of XML can hinder performance. XML primarily uses UTF-8 as the representation format for data. Sending commonly used data structures via standard implementations of SOAP incurs severe performance overheads, making it difficult for applications to adopt Web services based grid middleware. Due to the widespread adoption of standards in Web services by the grid community, it is critically important to investigate the impact on performance for the kinds of XML datasets used in grid applications. 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 [2–4, 17, 35, 75, 77]. The flexibility and loose coupling of XML-based standards allows senders and receivers of XML data to independently deploy selected optimizations, according to the communication patterns and data structures in use.

The combination of XML and HTTP lends many attractive traits to SOAP, including transparency, expressiveness, platform and language independence, extensibility and robustness. SOAP is a popular choice as the common underlying protocol for interoperability between grid services. These features facilitate the use of SOAP in diverse applications with widely varying characteristics and requirements. Clients and grid Web service endpoints can also add optimizations in their implementations, without making limiting assumptions about the capabilities and configuration of potential receivers of these messages. Though SOAP has many vital features to offer to grid applications, in terms of interoperability, loose-coupling, and compatibility, the specification has not been developed with performance as an important goal.

The convergence of grid and Web services standards has elevated the importance of SOAP, requiring the evaluation of SOAP for data types and communication patterns used by grid applications. It is thus important to have a test framework to determine if a particular SOAP toolkit can meet the performance requirements of an application, or if some other communication protocol should be employed.
Thus, it is important to compare, contrast, and evaluate different XML and SOAP implementations, so that end-users can make informed decisions on which toolkit to use for their particular distributed application. Specifically, the motivations for the design of performance evaluation frameworks for XML and SOAP toolkits are:

- Grid applications place a wide range of requirements on the communication substrate and data formats. These requirements include low latency, high throughput communication, minimal memory footprint for improved caching efficiency, specialized handling of scientific data, and overlap of computation and communication by streaming XML messages via HTTP 1.1 protocol. These disparate requirements have led to a wide range of design and implementation choices. A benchmark suite tailored for grid applications can aid in determining the XML and SOAP toolkit that has the most optimized implementation for the class of grid applications under consideration. A study of the frequently used communication patterns and data structures in grid services and applications will facilitate in the development of grid middleware that is exclusively tailored to meet the needs of grid applications.

- A wide range of implementations of XML Parsers is available [70], including Xerces (DOM and SAX) [82, 83], gSOAP-parser [77], Piccolo [60], Libxml [79], Expat [20], kXML [39], XPP3 [72], VTDXML [85], and Qt4 [73]. Simple and straightforward implementations of XML parsing paradigms can result in a severe impact on performance. A benchmark suite can help library developers identify and isolate the modules in their toolkits that need to be optimized. Ideally, toolkits will be designed to determine the data structures, use-cases, and communication patterns in the application code and have the ability to dynamically switch to the most optimized module for the use-case scenario.

- The reference implementation of the WSRF specification, available from the Globus Alliance website [71], uses the Axis [8] toolkit. The architecture of the reference implementation is modular in nature and facilitates the use of specialized pluggable modules for various aspects of Web services. A benchmark framework will facilitate in the addition of application and feature specific modules to WSRF implementations.

- The current set of grid Web services tools are not tailored to utilize the capabilities for parallelism available in the emerging multi-core architectures. The lessons gained from the execution of the benchmarks will also provide insight into software design of toolkits and the possible changes required in the XML structure itself, to aid in development of parallel processing modules for processing XML that scales gracefully with increase in number of available cores.

With the reasons mentioned above as motivation, this thesis presents common standard XML and SOAP benchmark suites for testing the performance and scalability of different XML toolkits, with a focus on data structures commonly used in grid services and applications. The Web services community currently uses a set of well-known SOAP payloads and interfaces to test the interoperability of various toolkits [84]. Our work complements these efforts in that it aims to provide a standard set of workloads to test the various features and performance characteristics of XML and SOAP implementations, rather than just the interoperability via the SOAP protocol.

These two benchmark suites provide grid middleware and application developers with working examples of XML and SOAP features, and provides a common way of testing and assessing the performance of their specific implementation of these features. Another contribution is the snapshot it provides of the current performance of many popular XML and SOAP implementations. This performance study provides insight into the relative strengths and weaknesses of different implementations under different usage scenarios, and demonstrates the utility of the benchmark suite. Our benchmark framework will benefit both Web services developers and grid application programmers. Web services and grid middleware (library) developers can gain insights into the various factors and design choices that determine the performance of processing XML and SOAP data, thereby improving their ability to build better implementations. Application developers can use the benchmark suite to test and compare the performance of various aspects of different toolkits, and accordingly select the one that best suits their application’s needs.

2 On-chip Multiprocessing and XML Processors

The volume of XML data used by applications has steadily increased over the years in both grid 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 data 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. For example, the MetaData Catalog Service (MCS) [68] runs on top of a Web service that provides functionality to store and retrieve descriptive information (metadata) on millions of data items. Workflows based on the XML specification format have emerged as critical tools to facilitate in the development of complex large-scale scientific applications such as mesoscale meteorology [33]. Another example is the International HapMap project that aims to develop a haplotype of the human genome [38]. 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 [64] 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 data files [26]. Processing of such large-scale XML datasets has emerged as a significant bottleneck in the overall application execution time of distributed applications. It is thus critical to design new XML processing algorithms that can process the emerging large-scale XML data sizes in a scalable manner.

The nature of computing is rapidly changing with the movement of the microprocessor industry towards chip multi-processors (CMPs), commonly referred to as multi-core processors. Web services based applications are expected to be extensively deployed on multi-core processors. Use of the currently available Web services implementation stacks can result in a severe impact on performance of applications when run on CMPs. Our benchmarking work demonstrates that most implementations of Web services do not scale well when the size of the XML data is increased [43, 44]. As memory access and latency can be the choking point in CMP processors, this performance limitation will be exacerbated on multi-core processors because performance gains need to be mainly achieved by adding more parallelism rather than serial processing speed.

In CMPs, the on-chip core-to-core bandwidth and communication latency are orders of magnitude faster than in traditional chip-to-chip multiprocessing systems that are typically used for parallel computing. The recent trends and announcements from major vendors indicate that the number of cores per chip will steadily increase in the near future. It is thus important to design XML processing technologies to leverage the opportunities presented by this trend and gracefully scale with the increase in the number of cores on each processing node.

One of the goals of this thesis is to focus on harnessing the benefits of fine grained parallelism, exploiting SMP programming techniques to process large-scale XML-based application datasets, and designing algorithms that scale well with increase in number of processing cores.

A few fundamental challenges have to be addressed in order to design an efficient XML processing toolkit for multi-core processors. An endpoint typically supports a wide variety of Web services. The services have complex inter-dependencies and as a result a thorough analysis is required to understand the resulting memory access patterns, synchronization between the various executing threads, and automatic detection of independent modules. Another significant factor that can affect the performance is fair and efficient allocation of shared resources such as memory and communication bandwidth among the executing concurrent threads. Additionally, the performance of processing XML data should scale gracefully with the increase in data size and the number processing cores per node.

This thesis presents the design of new techniques for parallelizing parsers for very large XML datasets. Parallel compilation has been studied for many years [5, 21, 36, 50], investigating both compilers that generate parallel code as well as compilers that divide work across multiple processors. Yet, despite this work, little has been applied to related problems in XML parsing of large datasets (with some notable exceptions [54, 61]).

3 Thesis Contributions

- We present the design and implementation of a benchmark suite for XML and SOAP implementations with standard mechanisms to quantify, compare, and evaluate the performance of each toolkit and study the strengths and weaknesses for a wide range of representative use case scenarios.
- We present new techniques to modify the lexical analysis phase for processing large-scale XML datasets to leverage opportunities for parallelism.
- We present an analysis of the scalability that can be achieved with our parallelization approach as the
number of processing threads and size of XML-data is increased.

- We present an analysis on the usage of various states in the processing automaton to provide insights on why the performance varies for differently shaped input data files.

4 Thesis Statement

In this thesis we present a benchmark suite that facilitates the study of the strengths and weaknesses of XML and SOAP toolkits for a wide range of representative use case scenarios. We present a new parallel processing model, applicable to application-based large-scale XML datasets, that achieves parallelism on emerging multi-core CPU architectures.
Chapter 2

Related Work

1 Introduction

This dissertation covers a range of topics related to XML [14], from its use in scientific applications to current parser implementations. There is related work in high performance computing benchmarks, non-scientific XML and SOAP benchmark suits, as well as parallel XML parsers. This chapter surveys the breadth of this space.

2 Benchmarks in HPC

Various benchmarks have been designed to test different features of HPC systems. These benchmarks can be broadly classified into two categories: low level probes and application based benchmarks [19].

Low-Level Probes: Benchmarks in this category are designed as probes to evaluate the performance of a system for fundamental operations. The HPC Challenge Benchmark has been released by the DARPA HPCS program [55]. This benchmark is geared towards evaluating performance boundaries for future petascale computers. The components that the HPCC benchmarks are designed to stress are: LINPACK [63] (CPU floating point performance), STREAM [56] (memory subsystem and streaming performance), GUPS (Giga updates per sec) that stresses the communication fabric and protocol for short messages, and FFT stresses the bisection bandwidth of the system).

Representative Applications Based Benchmarks: these benchmarks capture the requirements of specific class of applications. The NAS Parallel Benchmarks (NPB) [9], which originated from applications in computational fluid dynamics (CFD), are a set of programs designed to compare the performance of parallel supercomputers. These benchmarks consist of three pseudo-applications and five kernels, including GridNPB3 [32], which includes serial and concurrent reference implementations of distributed applications in Fortran and Java. It also has a suite of benchmarks named Rapid Fire that test the capability of a grid infrastructure to manage and execute a large number of short lived processes. The Standard Performance Evaluation Corporation (SPEC) [22] corporation defines several popular benchmarks. These include Java Client/Server benchmark to measure the performance of J2EE application servers, speed of request handling capabilities of an NFS (Network File Server) system, and a suite for evaluation of the performance of parallel and distributed architectures. The ParkBench [45] and SPLASH [80] benchmarks are also well known.

3 Other XML and SOAP benchmark suites

Several general XML benchmarking programs exist [16, 24]. The XML Benchmark [16] tests a number of parsers against arbitrary XML documents, but it does not provide a set of sample input files important for grid applications. The XML Parsing Benchmark [24] tests only two different Java-based parsers, and again is not tailored to the needs of grid application developers.

The XMark project [66] has designed an XML benchmark suite to examine the performance of XML repositories, such as relational databases, for a wide range of queries that are typical of real-world application scenarios. This benchmark effectively compares different implementations of XML databases with queries that test specific primitives of the query processor and storage attributes. Another complementary effort is the
SOAPFix [52] project that studies applicability of SOAP for realistic business computing with data obtained from the Australian Stock Exchange.

To test the interoperability of various SOAP toolkits, the SOAP community uses a set compliant payloads for an “echo” operation of primitives, arrays of primitives, and structs [84]. Our proposed SOAP benchmark suite complements this effort, as it includes some of these payloads to test the performance, along with the compliance to standards. Our SOAP benchmark suite also includes many grid specific feature and application payloads [43]. Our XML benchmark suite focuses specifically on the parsing component that applications embed, rather than the entire SOAP serialization infrastructure provided by SOAP toolkits.

4 Table-driven DFA-based parsing

Many parsers and compilers are built using lexical analysis generators that tokenize input into logical chunks that can be given to a higher level grammar analyzer. These generators generally take as input a collection of regular expressions (patterns) matching each class of token and actions that are executed when those tokens are matched. *flex* is a commonly used scanner generator.

*flex* transforms the collection of regular expressions into a non-deterministic finite automaton (NFA) which recognizes the union of the patterns. This NFA is converted into a DFA and then reduced. The DFA is written out as a table of states of two dimensions, DFA state and character input. The input is scanned by reading each character in input order, looking up the current state and current input in the DFA table to find the next state. Any required action is executed when the state changes.

4.1 Schema specific parsing within a generated scanner

As stated above, TDX used *flex* to parse and validate XML input which is meant to conform to a given schema (a WSDL-defined schema in this case). A tool called *WSDL2TDX* converts the a service-defining WSDL input into an appropriate input file for *flex*. The generated file contains specialized patterns for each type of element described in the schema, so there is a special token for each specialized type of start and end tag.

A user of the this parser can access the stream of tokens generated and receive notifications for each start tag, end tag, and content section, similar to code which uses a SAX parser. A significant difference between SAX-parsing and TDX-parsing is that the SAX events do not encode the element name, it is just stored as a string which must be examined. The TDX parser provides a mechanism, via the integer constant representing the element’s token, to quickly call the appropriate code to handle a particular element type without requiring the programmer to resort to string comparison which can be costly.

TDX performs very well in comparison to other high performance parsers. In addition to scanning and triggering SAX-like events, TDX is able to perform some input validation, due to its basis on a particular schema. It performs both of these functions, validation and parsing, in less time than many parsers are able to scan the input [44, 86].

5 Parallelized DOM-based parsing: MetaDFA

Many parallel compilation ideas have been discussed in the literature years [5, 21, 36, 50], 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.

MetaDFA [62] extends a Parallel XML Parser (PXP [61]) with a parallel scanner. PXP first reads an input document with a simple, low overhead DFA-based scanner to determine the tree structure. Once the underlying tree structure is known, PXP schedules parser tasks concurrently on the partitioned input to build a Document Object Model (DOM) tree from the input in parallel. One result of this early work was similar to that found by the authors of TDX: scanning quickly becomes limiting factor as thread count is increased. Relying on a purely sequential scanning component drastically limits potential speedup as thread and core use increases.

The solution embodied in MetaDFA is to parallelize this scanner using a structure that represents several scanning DFAs that run concurrently. The DFA used in PXP is transformed into a meta-DFA that contains states representing vectors of states in the original DFA. The start state is represented by a vector containing each state in order. Each entry in the vector represents starting the DFA in a different state. Subsequent states
are represented by vectors of states that trace each state in the vector over transitions in the original DFA, with \( \delta \) representing transitions to a dead state. If a trace over a particular entry in the vector leads to a dead state, that indicates the original state must have been the wrong state to start in. The goal is to effectively try to run the original DFA in all states because it is being run on an arbitrary subsequence of the input, and thus its appropriate start state would be unknown.

Each start state represented in the meta-DFA has a context which encapsulates the state and actions that have been performed by that meta-DFA. The input is divided between the several instances of this meta-DFA, and each meta-DFA processes its section of the input independently. The task of this meta-DFA-based preparser is to construct a tree structure which can be given to the PXP post-processor. There is some small amount of work to combine the output of each meta-DFA to properly merge these trees, but this is small compared to the rest of the task.
Chapter 3

Benchmarking XML Datasets

1 XML and SOAP Benchmarks

This chapter describes a set of benchmarks\(^1\) for XML parsers and SOAP servers, tailored to the needs of scientific application developers, that we have designed to study the current state of the art in XML parsing and SOAP serialization and deserialization.

2 Design of the Benchmark Suite for XML Processing

Consistent with trends in HPC, we have divided the benchmark suite for XML processing tools into two categories: feature probes and application-class benchmarks. This section explains the rationale for each benchmark’s design, and describes various optimizations that can be used to improve the performance of a toolkit for the features exercised by the benchmark. The benchmark suite is designed as a set of XML Schema documents along with example conforming documents, and a driver that reads trace data from local files and automates the testing process.

2.1 Feature probes

These probe specific features of XML (and Web service toolkit) implementations such as toolkit overhead, processing of documents as required in serialization and de-serialization in grid communication, management of arrays of various types, exercise of the buffering algorithms, handling of namespaces, scalability when dealing with co-referenced objects (multi-ref feature), and rate of handling typical SOAP messages.

Overhead

The *overhead* of the toolkit quantifies the minimum response time in processing an XML document. This measurement does not include costs associated with *cold start* or *warmup*, such as initialization costs due to loading of the necessary dynamic libraries or Java class files. Measurements are taken after the first few iterations.

The benchmark for this feature is a simple XML document that has a single element with no nested elements or attributes. The measured cost shows the minimum cost associated with memory allocation, de-allocation, and initialization of the parsers internal tables. This cost will be inherent to every use of the XML toolkit, and the results indicate which toolkit is best designed for extremely small XML documents.

Buffering

Since XML toolkits primarily deal with data in ASCII, they make extensive use of string operations, including search for specific sentinel characters, convert binary types to string formats, and incremental run-time allocation of strings. The default implementations of these features can often result in a performance penalty. The parsing and storage of frequently encountered XML constructs can be optimized via *look-aside* buffering schemes. gSOAP reuses the memory allocated for storing attribute name/value pairs to improve performance of parsing XML. This is particularly effective in parsing the *xsi:type* attribute which may be present in every XML element of the SOAP payload. Similarly, XPP3 caches parsed strings and avoids multiple

\(^1\)The SOAP benchmarks were published in [43] and the XML benchmarks were published in [44]
allocations of strings for processing XML input with values that repeat frequently, such as in the case of arrays.

The benchmark for this feature is exercised by XML documents representing SOAP-encoded arrays of various sizes and primitive types. Managing the repeated occurrences of xsi:type for each element of the array tests the buffering algorithm of the XML toolkit. As described in Section 2.2, grid applications typically exchange arrays of various types, and are directly affected by this feature of the toolkit they employ.

Managing namespace-qualified elements

The primary purpose of namespaces is to distinguish between identical names of elements, attributes, and tags that appear in an XML document. The extensive use of namespaces in XML documents makes it critical to evaluate the implementation of this feature. Each namespace is associated with a URI. A specialized attribute xmlns is used by tags to point to a fully qualified name. In a typical XML document, there are usually a few xmlns attributes but a large number of references to these attributes. The standard implementation of namespaces involves the use of a stack to store namespace prefixes and associated URIs. The performance limitation of the stack implementation stems from the repeated comparison operations that are needed in this implementation module. An optimization to manage namespaces is to use one table lookup to determine a corresponding internal namespace prefix of the xmlns attribute. The table should be populated with information obtained from the XML schema of the document being processed. In this scheme, the stack just records the translated prefixes to provide efficient matching of qualified tags. This results in reduction of the amount of storage and number of comparisons of prefixes.

The benchmark consists of XML documents in grid applications with plenty of xmlns bindings, such as those that are generated as a result of applying the canonicalization algorithm [13]. The canonicalization algorithm defines a standard form for an XML document, meant to guaranty bit-wise comparisons for logically equivalent documents. We chose canonicalized forms of example WS-Security standard documents for the benchmark. Another benchmark that tests this feature is the XML representation of nested data structures such as linked lists, wherein several tags and element names are identical. This forces a toolkit to apply its namespace resolution algorithms to correctly resolve all the names according to their namespaces.

Object graphs and co-referenced objects

An important requirement for Web services based grid applications is that data structures and object graphs be consistently stored and manipulated [78]. SOAP-RPC 1.1 encoding provides multi-referencing to serialize (cyclic) object/data graphs, wherein multi-ref ancestors are placed at the end of a message, so that all multi-references are forward pointing. Object copying or pointer back-patching must be used by an XML processor for each forward pointing edge to complete the edge references in the partially instantiated object graph. The SOAP 1.2 RPC encoding format is more natural, and allows both forward and back edges, but no constraints are given to avoid object copying or back-patching. This design is analogous to the use of pointers and references in many programming languages to refer to one instance of an object from multiple locations.

When a streaming parser, such as Simple API for XML (SAX) [83] or XPP [72], is used, a co-referenced object can only be deserialized after the parser has processed the multi-ref objects at the end of the message. Even though the DOM model is simple to use for such cases, it imposes a performance penalty as the entire message has to be stored in memory. Our performance tests show that in the widely used Apache Axis toolkit, every object in the graph is serialized with id and href using an inefficient non-scalable run-time algorithm.

In Java toolkits, if the common approach of using the equals() method is invoked, instead of IdentityHashMap, to compare all objects to check for co-references, decoding an XML document representing a graph can result in an expensive serialization algorithm, hurting the scalability of the application. The gSOAP toolkit uses generated routines to decode the XML document and reconstruct the original data structure graph. The parser takes special care in handling the id and ref attributes to instantiate pointers, using pointer back-patching and object copying when required. When the data structure is reconstructed, temporarily unresolved forward references are kept in a hash table keyed with the id values. When the target objects of the references have been parsed and the data is allocated in memory, the unresolved references are back-patched.

The workloads that we have designed for this benchmark consist of XML representation of a graph of nodes, and an array of strings of various sizes, wherein some of the array elements are identical. A conforming toolkit needs to test for co-references for each node and element. Even though the use of a hash-table is efficient, for large arrays it may result in overflow chains, and the lookup may not always be in constant time.
Processing SOAP messages

SOAP is the most widely used communication protocol in Web services based grid middleware. A SOAP message is formally specified as an XML infoset, which is an abstract description of the contents of the message. XML is the most commonly used on-the-wire representation of the infoset. A wide range of SOAP implementations, developed in various programming languages using different XML parsers, are available today. As a result, it is important to collect and analyze performance statistics for processing of XML messages that are generated as part of on-the-wire format of SOAP communication.

Our benchmark consists of SOAP messages for arrays of different data types and sizes that are commonly used in grid applications. The data types include floats, integers, doubles, strings, base64 encodings, and structs with few primitives. The size of the array for various payloads vary from a few elements to 100,000 elements, as we do not expect the SOAP protocol to be used for larger message sizes.

2.2 Application class benchmarks

The second set of benchmarks in our framework is application-oriented and captures typical XML messages in different classes of grid applications. The analysis of the these applications running on the grid infrastructure based on Web services will provide more insight into what new metrics and core kernel benchmarks need to be added to the suite for a more robust and well designed benchmark suite. Initial set of applications that we have considered include information service components, replica location services, resource management services, security components, and data grid services. In this chapter we present results with example payloa-dimensional data sets, XML messages sent via the SOAP protocol in the MetaData Catalog Service (MCS), application schemas such as HapMap [38] and BioMedical Applications, Mesh Interface Objects (MIOs) used in scientific computing, events stream used in applications such as Linked Environments for Atmospheric Discovery LEAD [27] project.

Workflow documents

Grid workflows have emerged as critical tools to facilitate in the development of complex scientific applications. Workflows allow the integration of legacy code and Web services from various organizations, developed in different languages, into a single distributed applications [33]. There are many scientific workflow systems tailored for use in grid computing applications, many of which use XML based representations to specify the workflows [69].

We have currently added two sets of workflow documents: (1) example workflow documents from the Kepler [51] project for scientific applications. The Kepler project’s goal is to provide an open source scientific workflow system to efficiently execute workflows using emerging Grid-based approaches; (2) example workflow documents currently used in for the LEAD application, which is used for creating an integrated, scalable cyberinfrastructure for mesoscale meteorology research and education. As our benchmark suite will be publicly available for download and use, we expect to add new workflow documents from other frameworks in the near future.

Metadata catalog service

The Metadata Catalog Service (MCS) [68] runs on top of a Web service that provides functionality to store and retrieve descriptive information (metadata) about logical data items. MCS has been developed as part of the Grid Physics Network (GriPhyN) project, with an overall aim of supporting large-scale scientific experiments. MCS is a classical example of a system that uses XML communication between clients and the Grid service, via the SOAP implementation of Axis [8]. The performance study reported in [68] shows that the Web service overhead causes an average performance drop by a factor of 4.8. We used the MCS schema to generate compliant XML documents of various sizes to study the XML toolkit that is most ideally suited to address the performance bottleneck reported by the MCS authors.

Human genome project

The International HapMap project aims to develop a haplotype map of the human genome [38]. The schemas are used to describe the common patterns in human DNA sequence variation. It is expected that grid computing solutions will play a significant role in the human genome project. Our benchmark suite consists of synthetic workloads that are compliant with the schemas for HapMap, to determine the toolkit that performs best for this project.
Mesh interface objects

Mesh interface objects (MIO) structures are of the form (int, int, double), where the two integers represent a mesh coordinate and the double represents a field value. This data structure is often used by scientific components on the grid. MIOs are used in communication between two Partial Differential Equation (PDE) solvers in different domains. An example usage is in a climate model that ties together an atmospheric simulator with an ocean circulation simulator [10]. Another example is a fluid simulation that is coupled with a solids structure code, as is done in some industrial process modeling [49]. Our benchmark framework consists of MIO payloads that test the scalability of the XML parser, as the number of MIOs is varied from a few to 100,000 elements.

Event streams

WS-Notification and WS-Eventing have emerged as the standard XML-based specifications for asynchronous notifications to interested listeners. They define the message exchange formats along with the baseline set of operations required by producers and consumers of events. These event specifications provide a de-coupled communication medium for grid applications. Typical uses of events include monitoring, debugging, and reporting occurrences such as a successful creation of a remote file. Notification services is also an integral part of services described in the WSRF specification [81].

We have defined two types of events. First, a simple event data structure as a struct with three data members: an integer (sequence number), a double (time stamp) and a string to store the event message. This definition provides both simplicity and flexibility. The string can be used to store small values such as a url for GridFTP transfer, or a long string requesting resource properties from a WSRF service. Second, we have included XML documents conforming to the WS-Notification and Eventing schemas to conform to the requirements of many existing and emerging grid applications that are expected to use these specifications.

Our benchmark driver can be configured to choose the size of the elements in the events schema that accurately reflects the needs of events in the application of interest, and accordingly decide the best toolkit to process event streams.

WS-security documents

The WS-Security suite of security specifications address a broad range of issues concerning protection of messages exchanged in a Web services environment. This model brings together formerly incompatible technologies such as Kerberos and public key infrastructure. The broad set of specifications include authentication, authorization, privacy, trust, delegation, integrity, auditing, and confidentiality. The OGSA Security Working Group, whose charge is to address the grid security requirements, has declared that the OGSA security architecture will leverage the Web services security foundations published in the WS-Security specifications [58]. Our benchmark suite consists of example documents from the WS-security specifications. A unique feature of these documents is the large number of namespaces for most of the elements.

Additionally, we have also included sample XML documents used by scientists at the National BioMedical Computation Resource (NBCR), who are building an end-to-end Web services architecture for Bio-Medical applications [53].

3 Representative Performance Results

The Linux test environment consisted of one dual core machine, with an Intel(R) Pentium(R) D CPU 3.00GHz with 256MB PC4200 RAM and a 7200 RPM 80GB SATA-2 drive running the i386 edition of Ubuntu Linux 5.10 (“breezy”) with the 2.6.12 kernel compiled for i686 SMP processors. All C and C++ based parsers were compiled with gcc/g++ version 4.0.2. All Java-based parsers were compiled and run with the Sun Java 5 SDK, version “1.5.0_06.” The C#-based parser is from the implementation from System.Xml in Mono version 1.1.8.3. The version of the other parsers presented are as follows: expat 1.95.8, gsoap 2.7.0d, libxml2 2.6.21, piccolo 1.0.4, xerces-c 2.6.0, xerces-j 1.4.4, and xpp3 1.1.3_6.

Figure 3.1 shows overhead incurred by various toolkits. Among the toolkits we tested, gSOAP-parser has the least overhead of 5.5 μs, and Expat’s overhead at 14 μs is the next best. The Mono-Reader (developed in C#) parser, which is a light-weight pull-model based parser, has the least overhead (33 μs) among non C/C++ parsers. Both Mono-DOM and XPP3 have an overhead of approximately 60 μs. These two have the next lowest overhead among non-C/C++ parsers. Note that the Xerces implementations in both Java and C have relatively high overheads. Libxml, Piccolo, Qt4, perform better than Xerces, but have an overhead more
figure 3.1: The overhead associated with each parser. We run a tiny XML file through each parser 20 times and measure the parse time. Because the XML file is so small, this effectively measures each parser’s setup and cleanup time. gSOAP’s overhead is the lowest at 110 µs. Xerces-J-DOMs overhead is twice that of Xerces-J-SAX at 7029 µs.

than 1 millisecond.

In Figure 3.2, we chose two grid applications (Workflow and HapMap), with different payloads ranging from 277KB to 4.9MB. We found that apart from Xerces-c-DOM, the rest of the parsers were able to execute the benchmark within 2 seconds. Depending on the exact performance needs of the application, one among Expat, gSOAP, and Libxml can be used for C/C++ based middleware for these applications.

Figure 3.3 and Figure 3.4 compare the performance of C/C++ based toolkits for arrays of doubles and integers respectively. The payloads consist of XML documents generated by serialization according to the SOAP protocol. The size of the arrays was varied to 100,000 elements, which we believe is the upper limit for usage via SOAP-based communication. Figure 3.3 shows that the Expat toolkit performs the best (744 ms, for 20 iterations of size 100,000), while the Xerces-C-DOM toolkit is orders of magnitude slower and does not scale well. For the same array sizes, due to conversion to ASCII format, the payload for array of integers is less than that of array of doubles, even though the underlying tree structure is the same. So, the parsers perform better for array of integers, as can be seen in Figure 3.4. In particular, gSOAP and Qt4-SAX show marked improvement.

In Figures 3.5 and 3.6, we present results when toolkits parse typical XML payloads for WS-Messenger [47] and WS-Security documents that are small in size but contain lots of namespace qualifications. The surprising results we see in these two graphs is that Qt4-SAX performs worse than Xerces-DOM for WS-Security documents. Expat, again, is the best toolkit for these kinds of XML messages, slightly outperforming gSOAP.

We present performance of Java-based parsers in Figure 3.7. Interestingly, XPP3 handles the smaller workflow documents better than other parsers, while Piccolo performs best for larger sized documents used in applications for HapMap (hapmap_1797SNs.xml) and biomedical projects (molecule_1kzk_pretty.xml).

Figure 3.8 shows that either of Piccolo or XPP could be used in Java-based grid frameworks for handling XML payloads generated from SOAP representation of integer arrays. The results differ from the case when
**Figure 3.2:** Performance of C/C++-based parsers on some large grid applications. Files sizes range from 277KBytes (workflow_PIW.xml) to 4.9MBytes (hapmap_1797SNPs.xml) and are parsed 20 times in succession. All parsers processed the HapMap file in approximately 2s, with the exception of Xerces-C-DOM, which took about 5s.

**Figure 3.3:** Scalability of C/C++-based parsers over arrays of doubles in SOAP payloads. Here the parsers are fed XML documents containing SOAP-serialized arrays of doubles. Expat leads the group parsing a document containing 100,000 doubles 20 times in 744ms. Xerces-C-DOM generates a DOM each parse, and performs the same task in 5,965ms.
Figure 3.4: Scalability of C/C++-based parsers over arrays of integers in SOAP payloads. Similar to Figure 3.3, we test each parser against a set of XML documents containing SOAP-serialized arrays of varying size. In contrast to Figure 3.3, all parsers improve when handling integers versus doubles, though gSOAP and Qt4-SAX both improve more than the others.

Figure 3.5: C/C++-based parsers using WS-MG notification messages. Again, Expat is the best performing parser, processing the WSMG notification message 20 times in 1.48ms. Xerces-C-Dom tops the chart at 7.31 ms.
Piccolo and XPP handle arrays of complex types (structs) and do not have similar performance, as shown in Figure 3.7. The performance for arrays of strings in Figure 3.9 presents the same conclusions as the case with array of integers in Figure 3.8.

As opposed to the large size application messages used in Figure 3.7, for smaller size XML documents represented by WS-Notification and WS-Security documents in Figure 3.10, Piccolo and XPP3 have comparable performance, while the the Xerces-Java toolkit performs poorly.

TDX combines parsing and validation together, so parsing and validation cannot be separated. TDX scans and tokenizes the XML message in a separate stage, so scanning together with parsing was measured. The other parsers tested combine parsing with scanning. The result is shown in Figure 3.11: TDX scans, parses, and validates in much less time than it takes any other parser to even scan and parse.

4 Recommendations for XML Application Developers

- If low overhead is desired, for example when very small documents need to be processed, then the gSOAP-parser and Expat are the ideal choices for C/C++ frameworks. For C# based toolkits on Linux, the Mono-reader should be used. XPP3 and Mono-DOM also have very low overheads, and should be preferred over Piccolo, Libxml2, and Qt4. The Xerces toolkit, in both the Java and C++ implementations, performs the worst among the toolkits we tested, and should be avoided for applications where overhead is critical.

- Xerces has a modular design and provides a great deal of flexibility for users to add their modules and mappings. As a result it is a popular choice for many applications. So, in C/C++ toolkits, if Xerces has to be used, our results show that the SAX implementation should be used, rather than the DOM model. The DOM model has a prohibitive overhead for arrays of scientific data such as doubles, floats, and integers. If performance and scalability are important, and array sizes beyond 10,000 need to be parsed, then gSOAP-parser and Expat should be employed.

- The buffering algorithms and management of namespaces are exercised extensively for processing array of strings. Among the C/C++ parsers, we note that gSOAP and Expat are comparable. Due to the
Figure 3.7: Performance of Java-based parsers on some large grid applications. This is the same test as shown in figure 3.2, using Java-based parsers. There is some interesting variability here. XPP3 handles the workflow tests in roughly 10% the time of the other parsers, but is squarely in the middle of the group for the HapMap and Molecule tests.

Figure 3.8: Scalability of Java-based parsers over arrays of integers in SOAP payloads. The same test as figure 3.4 for parsers written in Java. We see that Piccolo and XPP3 are equivalent here.
Figure 3.9: Scalability of Java-based parsers over arrays of strings in SOAP payloads. Similar to the other SOAP payload tests, here the elements in the arrays are text strings, as opposed to textual representations of numbers.

Figure 3.10: Java-based parsing of WSMG notification messages. The same tests as shown in figures 3.5 and 3.6, applied to Java-based parsers. Here Piccolo and XPP3 again show much better performance than either Xerces-J-DOM or Xerces-J-SAX. XPP3 parses the message in 30% of the time it took Xerces-J-SAX, which has a similar programming model.
look-aside buffering scheme and optimizations for handling namespaces in gSOAP, it performs well for large array sizes. As with arrays of doubles and integers, Xerces-DOM should not be used for processing large arrays of strings.

- For Java-based frameworks, Piccolo and XPP3 have comparable performance, and out-perform the Xerces-Java implementation. Again, if Xerces-Java has to be used, then for performance, its SAX interface is preferable over its DOM interface.

- XPP3 performs the best among Java toolkits for processing documents with complex types, such as some Kepler based workflow examples, whose sizes are a few hundred KBs. However, once the size exceeds one MB (Biomedical and Genome XML documents), Piccolo outperforms other toolkits.

- The MCS toolkit should use XPP3 or Piccolo to parse XML messages sent between the clients and the MCS server. C/C++ based clients, should use gSOAP or Expat to connect to MCS. These choices, instead of using the currently employed Axis toolkit, will significantly reduce the Web services overhead (factor of 4.8) that was reported in the MCS performance results [68].

- Pluggable modules should be incorporated into the communication medium of the reference WSRF-Java implementation, so that Axis toolkit based processing can be replaced by the efficient libraries of Piccolo or XPP3. These toolkits perform better than the other Java toolkits for WS-Notification documents, arrays of primitives and complex types, and WS-Security documents.

- The model used by the TDX parser has promise for high-performance XML processing needs, as it efficiently combines validation and scanning in one step. However, it is only applicable when the schema for the XML document to be processed is known in advance.
5 XML Toolkit Design for Multi-core architectures

For efficient use of multi-core architectures, it is important for XML toolkits to minimize the cost of synchronization, multi-thread overhead, and use of mutex. With the currently used sequential-access formats of XML documents, if the document is not pre-scanned, the parser threads need to determine their starting points by moving a cursor over the document. The cursor may be controlled by one thread, or cooperatively. However, moving the cursor is a costly sequential operation that must follow XML syntax rules and handle local namespace bindings. Scanning XML is a significant component in the entire parsing process.

Amdahl’s law suggests a high ratio of parsing/decoding time over XML scanning is needed to get reasonable speedups. In our earlier work on designing a table driven parser [87] (whose performance is shown in Figure 3.11), the breakdown in scanning, parsing, and deserialization overhead with TDX parsing is reported and compared to other XML parsers. The analysis shows that scanning can be three times slower than parsing. From Amdahl’s law we see that 14% speedup can be gained with two threads, and 23% with four threads.

An issue with SAX parsing is its inherent event-based processing mode, as a result, parallel threads will not be helpful. It is possible to populate a DOM tree in parallel and gain some speedup, however, the subsequent traversal of the tree by a single thread will be slow. Another approach is to use a read-ahead thread that caches portions of the file ahead of the single-threaded parser.

To make effective use of multi-core architectures, we recommend the following: (1) pre-scanning of the document, to combine parsing with decoding, is essential to decide how to subdivide tasks to the parser threads; (2) random access should be added as a feature in XML documents (e.g. via attributes at the top level element) to aid in avoiding the cost of sequential scanning to determine starting point for each thread; (3) schema developers should specify a set of guidelines for processing instructions, in the XML document itself, to enable high performance processing under multiple threads.

6 SOAP Benchmark Suite

The benchmark suite consists of operations in WSDL files along with bindings for SOAP calls and a driver that reads trace data from local files and automates the testing process. Each SOAP toolkit is made to implement the operations defined in the WSDL documents for the benchmark. This section explains the rationale for each benchmark’s design, and describes various optimizations that can be used to improve the performance of a toolkit for the features exercised by the benchmark.

6.1 Serialization, deserialization and round-trip performance

SOAP has been expressly designed to support interoperability between multiple different implementations. It is important to collect and analyze isolated performance statistics for serialization and deserialization because the toolkits used by clients and servers may differ. With the recent release of WSRF implementations [71], we expect a wide range of SOAP implementations to interact with well known WSRF services.

**Serialization**

SOAP serialization converts in-memory objects into an XML stream that is sent on the wire in UTF-8 format. We list several optimizations that address various stages of serialization.

HTTP 1.0 requires the precise buffer length to be placed in the HTTP header’s Content-Length field. A simple SOAP serialization algorithm allocates and extends the buffer as the data structures are traversed and converted to ASCII. Once this conversion is completed, the buffer length is calculated and added to the HTTP header. This naive approach can invoke multiple expensive memory operations to create a single large memory buffer with the SOAP payload.

In [17] we describe how to avoid reading each character of the XML tags from memory. If the tags are created as literals, the characters comprising the tags are likely to be in the instruction stream as immediate operands.

bSOAP [2] reduces the number of system calls by using vectored send to dispatch multiple buffers with a single call. gSOAP [77], on the other hand, uses a two-iteration algorithm. The first iteration traverses the data structures and calculates the required buffer length. The second iteration generates the HTTP header, fills in the content length, and serializes the SOAP message directly over TCP/IP. This approach avoids keeping the entire buffer in memory.
In earlier work we showed that the conversion of IEEE 754 floating point data to ASCII is complex and can account for 90% of end-to-end communication time [17]. Our analysis with Sun Forte on a Blade 1000 determined a sharp drop in performance when the required precision ranges from 14 digits to 17 digits.

The serialization bottleneck for floating point data is addressed in bSOAP through differential serialization [2]. The idea behind differential serialization is to store copies of previously sent messages within the stub that sends them. When the stub is invoked for future calls, only the data that has been changed needs to be re-serialized into the message. The extent to which this optimization is effective depends on message size, content, structure, and the similarity between consecutive messages. Shifting message contents in memory, stealing space from neighboring fields, and stuffing fields with whitespace can increase the cases when differential serialization can be applied [4]. Performance results demonstrate that differential serialization can improve send-times between 17 and 1000 percent [2–4].

**Serialization Benchmark:** Our benchmark suite measures the serialization performance of various toolkits for arrays of different data types and sizes that are often used in grid applications. The significance of memory management, conversion to ASCII formats, cost of establishing TCP connections, and size of cache varies as the size of the array changes. The benchmark driver generates a request to invoke a method on the SOAP toolkit being tested, which serializes an array of the requested size and type. The driver sends the invocation for several iterations and measures the average performance of the toolkit.

**Deserialization**

Deserialization converts XML streams in wire-format to objects in memory. We discuss several important aspects of the deserialization process.

The widely used paradigms for parsing XML documents include Document Object Model (DOM), Simple API for XML (SAX) and XML Pull Parser (XPP) [72].

The DOM model maps the XML document into a tree representation in memory. This allows the document to be easily traversed and modified. However, for large documents, DOM parsing can be memory intensive. In contrast, SAX parsing never stores the entire XML document in memory. Instead, a callback model emits events for all the document’s elements and tags. For large static documents, SAX is preferable to DOM. SAX is also often used when only a few specific elements of an XML document need to be extracted and processed. Pull parsing, employed by the XPP parser, is specialized for parsing payloads in which elements are processed in succession, and no element needs to be revisited. XPP provides the added feature of building a partial XML Infoset tree in memory in an incremental manner.

DOM, SAX, and XPP require two passes through the XML document; the parser tokenizes the document in the first pass, and the application processes the content in the second. An STL map is typically used to compare each tag retrieved by the parser with the one that is expected. Results in [17] show that a trie data structure, which has $O(1)$ lookups as opposed to $O(\lg n)$ for STL map, can provide significant performance improvement for matching tags that appear repeatedly.

gSOAP uses a performance-aware compiler that generates code for fast XML parsing and processing of native C/C++ types. The algorithms are based on single-pass schema-specific recursive-descent parsers for XML decoding and dual pass encoding of the application’s object graphs in XML.

**Deserialization Benchmark:** Our benchmark consists of SOAP messages for different sizes of frequently used data types (strings, integers and doubles). Payload elements are fully namespace qualified and the driver verifies that the toolkit has appropriately handled all elements. Deserialization benchmarks for complex types are described later. Our tests include toolkits that use different models for parsing XML documents: XSUL uses XPP, gSOAP uses a recursive descent parser, AxisJava can use SAX or DOM, and .NET uses an efficient streaming parser via the XMLReader class. The benchmark driver sends trace data with SOAP payloads of various sizes (and for various types of data) and invokes the method on the SOAP toolkit requiring it to deserialize the request. The driver repeats this test for several iterations for each toolkit and measures the performance.

**End-to-end performance**

The determining factors for end-to-end performance are serialization, deserialization, and available network bandwidth. If the same toolkit is used for both the client and Web service implementation, deserialization could be optimized for the parameters set by the serialization process. Even though the wire protocol is fixed for SOAP (1.1), the names of the tags, placement of whitespaces, and serialization order of data members of an object are not fixed. Apart from SOAP payload specific parameters, the TCP packet size and the size of
each chunk can be fine tuned if the same toolkit is used by a sender-receiver pair.

**End-to-End Performance Benchmark**: This benchmark combines the tests for serialization and deserialization. The driver measures toolkit scalability for all primitive data types, and for arrays of primitives. A different benchmark measures complex data types, discussed later. This benchmark’s example users include those who download WSRF [71] and use the default implementation, based on AxisJava, for both the client and service endpoints.

Results in [67] show that Base64 parameter encoding can significantly reduce overhead, compared to standard XML encoding. To quantify this advantage, for all types and sizes, we have included it in the benchmark suite for serialization, deserialization, and end-to-end performance measurement. The end-to-end benchmark driver sends trace data for serialized SOAP messages (of various types and sizes) to the SOAP toolkit being tested. The toolkit is required to deserialize this message and again serialize it as the return value for the method invocation. The driver repeats this process over several iterations for each data type and size.

### 6.2 Candidate features for optimizations

In this section we isolate the optimizations and related benchmarks for specific SOAP implementation features. The measurement methodology for the benchmark driver is the same as described in the subsection above.

**Streaming vs non-streaming**

As discussed earlier, calculation of HTTP 1.0 content length of the SOAP payload can hinder serialization performance. We showed in [35] that the size of SOAP’s on-the-wire representation is a factor of four to ten times greater than the corresponding binary representation. SOAP processing is affected by both extra memory invocation for calculating the length, and cache misses due to large buffers. HTTP 1.1’s chunking and streaming feature can help address this problem.

HTTP 1.1 explicitly supports overlapping the serialization process with the network transmission of buffers [76]. *Persistent connections* (keep-alive) reuse the same TCP/IP connection for multiple calls between two endpoints. The SOAP payload can be sent in chunks, with each chunk preceded by its length. Small chunk sizes can ensure cache hits but result in many system calls. Large chunk sizes can reduce the number of system calls, but may not always lead to cache hits. Thus, chunk size should be a configurable parameter, set according to the native system characteristics.

**Streaming Benchmark**: The streaming benchmark measures the performance of serialization, deserialization, and end-to-end performance when chunking and streaming is used with a persistent TCP/IP connection, compared to the case when it is turned off. For small messages, the cost of repeatedly establishing network connections can significantly impact performance. This benchmark quantifies the exact performance benefit of using streaming. The driver sends streaming data from trace data for primitives, arrays and complex types. It can be configured to vary the size of each chunk in the SOAP payload.

**Namespaces**

XML namespaces are extensively used in SOAP messages. Namespaces are used to uniquely identify and distinguish between identical names of tags, elements and attributes. Each namespace is associated with a defining namespace name (URI). Each tag has a prefix that points to a fully qualified name via a special attribute named `xmlns`. XML parsers typically use a stack to store namespace prefixes and corresponding URIs. The number of defining `xmlns` namespace bindings in an XML message is typically much smaller than the number of uses of this namespace prefix. As a result, maintaining the stack can result in several comparison operations, and can hurt the overall performance of the deserialization module.

Processing of `xmlns` attributes can be optimized by using just one table lookup to determine a corresponding internal namespace prefix. The table should be populated with prefixes obtained from the XML schemas of the SOAP messages. The namespace stack can simply record the translated prefixes to provide efficient matching of qualified tags and avoid storage and expensive comparison of namespace URIs.

**Namespace benchmark**: Our benchmark consists of SOAP payloads with varying level of nested data structures (linked lists). Several of the tag names and attribute names in each level are identical, forcing
toolkits to correctly resolve them according to the namespace qualifications. The payloads also have a varying number of namespace qualified attributes. The synthetic data for these benchmark is based on the various attributes and nested elements required for emerging security standards for grid Web services, such as WS-Security [48]. The WS-Security standard enhances the SOAP messaging protocol to provide message integrity and confidentiality by associating tokens (included as namespace qualified elements and attributes) with each message.

**Multi-ref**

Each co-referenced object is assigned a unique identifier, represented by an attribute value, when it is serialized the first time. If the same object appears again in a data structure, it can be serialized with a multi-ref accessor, id-ref, that points to the identifier of the original object. Multi-ref is essential to efficiently serialize cyclic data-structures. This design is analogous to the use of pointers and references in many programming languages to refer to one instance of an object from multiple locations. The serialization of co-referenced objects require serialized objects to be stored in a table. Before an object can be serialized, the table needs to be searched to determine if the object has already been serialized, in which case the id-ref attribute must be used. A naive implementation can hurt the scalability of the serialization process. For example, if the equals() method has to be invoked on each object, as is often done in Java, serialization of \( n \) objects will lead to an \( O(n^2) \) serialization algorithm. For toolkits based in Java, we recommend the use of IdentityHashMap, which is optimized for use with with Java references.

**Multi-ref Benchmark**

Our benchmark consists of an array of strings, wherein many of the strings are identical. A multi-ref compliant toolkit must check for co-references for every string. This is usually done via lookups in a hash-table. However, due to hash table’s overflow chains, it may not always perform a lookup in constant time. As the array size increases, the overhead of maintaining the logical coherence of graph structures negatively affects performance.

### 6.3 Latency

We define latency as the overhead incurred by a toolkit in an end-to-end call when no parameters are sent or received. It quantifies the minimum response time of a toolkit. However, the latency measurement does not include costs associated with cold start or warmup, such as initialization costs due to loading of the necessary dynamic libraries or Java class files. Measurements are taken after the first few iterations.

**Latency Benchmark:** The benchmark is void echoVoid() operations to test the overhead imposed by a toolkit. Even though no parameters are sent, the call still traverses all the layers of the serialization and deserialization stack, and effectively measures the overhead that will be inherent to every call.

**Application specific benchmarks**

We describe some benchmarks that are based on well known services and widely used communication patterns used in distributed applications.

**Events**

An event can be broadly defined as a time-stamped message with typed data that is delivered from a source to a set of subscribed listeners. Events provide a de-coupled communication medium for grid applications. Typical uses of events include monitoring, debugging, and reporting occurrences such as a successful creation of a remote file. Event services (also called notification services) should be extensible, language-independent, platform independent and provide ready integration with applications. The SOAP protocol is perfectly suited to meet these requirements. Notification services were recognized as a common port type in the OGSI [74] specification.

**Events Benchmark:** We have defined the event data structure as a complex type with three data members: an integer (sequence number), a double (time stamp) and a string to store the event message. This definition provides both simplicity and flexibility. The string can be used to store small values such as a url for GridFTP transfer, or a long string requesting resource properties from a WSRF service.

Our benchmark driver measures the performance of a toolkit for sending and receiving events ranging from tens to thousands of events. The driver can be configured to choose a string size that accurately reflects the needs of events in the application of interest.

**Mesh interface objects**

Scientific components on the grid frequently exchange mesh interface objects (MIO) structures of the form (int, int, double). The two integers represent a mesh coordinate and the double represents the field value. A
typical use of MIOs is in communication between two partial differential equation (PDE) solvers on different domains. Example applications include a climate model that ties together an atmospheric simulator with an ocean circulation simulator [10] and fluid simulation that is coupled with a solids structure code [49].

**MIO Benchmark:** The performance tests for MIOs record the scalability of a SOAP toolkit as the number of MIOs is varied from ten to 25,000.

**MCS benchmark**
The Metadata Catalog Service (MCS) [68] is a well known grid service that provides a framework for efficiently managing the storage and retrieval of metadata associated with large collections of files generated by data-intensive applications. Clients of MCS interact with the MCS Web service via the Axis [8] SOAP implementation. A scalability study in [68] shows that the Web service overhead causes an average performance drop by a factor of 4.8. We contacted the authors of [68] to obtain the synthetic data (for the names, types and values of the attributes) used for their tests, and used it to define this benchmark. Each attribute is a tuple consisting of a name, type, and value. Our tests can be used to determine which is the best available toolkit for MCS.

**Google web service**
The Web service interface for the Google search engine [34] and Amazon.com [6] portal, are the two most widely used industrial Web services. Though they are not grid services, they are representative of the growing interest in using Web services based protocols. Both Google and Amazon.com sites receive a large volume of requests daily, and serving these requests via SOAP can be expensive. Our benchmark measures the performance of processing the response from the `doGoogleSearch` query of the Google Web service API.

### 6.4 Binary attachments via DIME and MIME

Two important protocols for sending binary large objects (BLOBs) as attachments are Direct Internet Message Encapsulation (DIME) [59] and Multipurpose Internet Mail Extensions (MIME) via SOAP with Attachments (SwA) specification [59]. The motivation for sending binary data as attachments is to avoid the overhead of serialization and deserialization for binary data. Moreover, for digitally signed attachments, it may not be possible to use standard serialization techniques without affecting the integrity of the data. MIME messages are sent as a series of records, with each record separated from the other via a unique marker string. DIME and MIME have many similarities. The important difference is that in DIME, the header for each record contains the exact size of the record. If multiple attachments have been sent, the receiver can directly access a particular record.

Our benchmark sends multiple attachments, one to fifty, varying the size of the attachment from 1KB to 100KB. The aim of the benchmark is to determine the threshold point when it is better to use DIME instead of MIME (or vice versa).

### 6.5 Dynamic vs static code generation

The Java Dynamic Proxy Class [57], introduced in Java 1.3, is an elegant and flexible feature that allows a class to implement a list of interfaces specified at runtime. This design is in stark contrast to the use of classic stubs and skeletons. Stubs and skeletons shield run-time specific details from a user, and are generated by a specialized code generator. The dynamic proxy feature obviates the need for a code generator in Java based SOAP toolkits, and provides a type-safe reflective dispatch of invocation on dynamically created proxies. The dynamic proxy, however, imposes a severe performance penalty. Even though the generation of static stubs and skeletons for every server interface is cumbersome, it offers attractive performance benefits. Our benchmark is designed to illustrate the exact performance penalty of using dynamic proxies. For this test, the toolkit must provide support for switching between the two designs.

### 7 Performance

This section describes the performance of a representative set of five toolkits when run against our benchmarks. We used two sets of drivers. A heavy-weight driver verifies the accuracy of every response from the tested toolkits. A light-weight driver reads trace data from files into a local buffer, then sends it to the different toolkits for several iterations, but the responses are not checked for accuracy. This approach ensures toolkit accuracy, but keeps verification cost out of the reported performance data. The light-weight driver
Figure 3.12: This graph shows the effect on serialization when the size of an integer array is scaled. The performance of AxisJava and XSUL is similar. bSOAP and gSOAP complete the benchmark in 4% and 17% respectively of the time it takes for AxisJava for the largest size (100,000 integers).

Figure 3.13: Here we compare the serialization performance of four toolkits on arrays of doubles. As with the integer case in Figure 3.12, the Java-based toolkits (AxisJava and XSUL) perform similarly. bSOAP and gSOAP send the largest size (25,000 doubles) arrays in 13% and 30%, respectively, of the time it takes for AxisJava to do the same.
Table 3.1: Latency is measured by the time in seconds for the client to receive the full response from an `echoVoid` call. On Linux, XSUL’s overhead is two times greater than gSOAP’s. However, AxisJava’s overhead is 10 times greater than XSUL’s. On Windows, .NET and XSUL have comparable overheads, while AxisJava’s overhead is 33% higher than .NET’s.

<table>
<thead>
<tr>
<th>Toolkit</th>
<th>gSOAP</th>
<th>XSUL</th>
<th>AxisJava</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latency (seconds)</td>
<td>0.00052</td>
<td>0.00107</td>
<td>0.01200</td>
</tr>
</tbody>
</table>

Figure 3.14: Here we compare the deserialization performance of AxisJava, gSOAP and XSUL. Each toolkit is sent a SOAP payload for double arrays of various sizes, asked to deserialize it and return its size to the driver. For arrays of 10,000 doubles, AxisJava takes 5.1 times as long to respond as XSUL, which in turn takes 7.8 times as long to respond as gSOAP.

The versions of the tested toolkits are: gSOAP 2.7e, AxisJava 1.2RC2, .NET 1.1.4322, XSUL 1.99RC2 and bSOAP 0.5alpaha. gSOAP is implemented in C/C++, bSOAP in C++, and XSUL and AxisJava are developed in Java. Some performance variations are due to the inherent efficiency of C and C++ implementations over Java. However, since the SOAP wire-protocol is independent of any programming language and interoperability is an important feature of SOAP, comparing implementations in different languages is useful.

Since the .NET implementation is available only on a Windows platform, we compared its performance with the Java implementations of Axis and XSUL deployed on a Windows box. We also deployed gSOAP, AxisJava, XSUL and bSOAP on Linux machines and tested the performance separately. The results on Windows and Linux need to be viewed separately, as the hardware configurations of the Windows and Linux machines were not identical.

The Linux test environment consisted of two dual processor machines, each configured with 2.0 GHz Pentium 4 Xeon with 1GB DDR Ram and a 15K RPM 18GB Ultra-160 SCSI drive running Debian Linux.
Figure 3.15: This graph compares the deserialization performance for strings. Again, XSUL performs significantly better than AxisJava for deserialization. For an array size of 25,000, it takes AxisJava 7.4 times longer to respond than XSUL, and XSUL takes 3.6 times as long to respond as gSOAP.

Figure 3.16: This graph compares deserialization performance of Axis, XSUL and .NET for an array of doubles on Windows. XSUL and .NET are comparable, while AxisJava does not scale well for large array sizes. For an array of 10,000 doubles, AxisJava’s deserialization time is 6 times greater than those of XSUL and .NET.
Figure 3.17: We compare the end-to-end performance for arrays of MeshInterface Objects (2 integers and a double). The plots show that for sizes greater than 10,000 objects, XSUL’s performance considerably degrades compared to AxisJava. For 10,000 elements, XSUL and AxisJava respectively take 11.7 and 12.3 times more time to execute the benchmark compared to gSOAP.

3.1 (“sarge”) with the 2.4.26 kernel. The machines were connected by Gigabit Ethernet. gSOAP and bSOAP were compiled with gcc/g++ version 3.3.4. XSUL and AxisJava were compiled with Java 1.4.2. Relevant socket options, for both gSOAP and bSOAP, include SO_KEEPALIVE, TCP_NODELAY, SO_SNDBUF = 32768, and SO_RCVBUF = 32768. For fairness, when bSOAP is compared with the other toolkits, it is set to re-serialize all data (BSOAP_100 means bSOAP is serializing 100% of its data).

For the tests on Windows (.NET, XSUL and AxisJava) we used a Dell Dimension 4500 with Intel Pentium 4 2.26GHz processor, 1GB of DDR SDRAM and 80GB Ultra ATA/100 hard drive running Windows XP.

The performance graphs show the measured time in seconds on the y-axis and the size of the application level data on the x-axis. The effective throughput (bandwidth obtained by a given SOAP implementation) can be calculated directly from the data points on each plot.

7.1 Summary of performance results

• **Latency**: Table 3.1 shows the overhead imposed by each toolkit, for both Windows and Linux platforms. gSOAP’s overhead is less than XSUL by a factor of 2. XSUL outperforms AxisJava by a factor of ten. On Windows, AxisJava is also slower than .NET and XSUL. The overhead of gSOAP is lower than that of the Java-based toolkits (XSUL and AxisJava) as it does not incur the cost of using reflection and dynamic proxy classes. gSOAP uses statically generated stubs and skeletons for each remote call, which are known to be faster than dynamically generated code (proxies).

• **Serialization**: Figures 3.12 and 3.13 compare the serialization performance of doubles and integers respectively. XSUL and AxisJava perform similarly for all array sizes. The cost of the toolkit overhead is higher for AxisJava (as can be seen in the results for latency), but it gets amortized with increase in the size of data being sent. The cost of serialization for large array sizes (especially for floating point data) is dominated by the conversion of data from floating point representation to the ASCII format [17]. AxisJava and XSUL use the same conversion routines, and hence have similar performance characteristics for serialization.
Figure 3.18: This graph compares the performance for the deserialization benchmark for events. Each event object contains an integer, a double and a string. XSUL's performance degrades considerably when it deserializes more than 15,000 elements, but for lesser number of events, it outperforms AxisJava. gSOAP is orders of magnitude faster in handling complex types compared to the Java toolkits.

bSOAP is more efficient than gSOAP for all array sizes. For 25,000 elements bSOAP and gSOAP take 4% and 17% respectively, of the time it takes AxisJava to complete the benchmark. gSOAP uses a two-iteration serialization algorithm (described in 6.1), while bSOAP directly allocates buffers during the serialization process.

• Deserialization: We ran the deserialization benchmark for doubles and strings. The tested version of bSOAP did not have support for deserialization, and hence bSOAP is not included in the graphs. Figures 3.14 and 3.15 show that AxisJava does not scale well. For 10,000 elements, AxisJava takes 5 times more time than XSUL, and 35 times more than gSOAP to execute the benchmark. On Windows (see Figure 3.16), XSUL and .NET have similar performance. AxisJava's execution time exceeds .NET by a factor of 6. The choice of an XML parser plays a significant role in deserialization of SOAP payloads. XSUL uses an efficient pull parser (XPP) that has a low memory footprint and is specifically designed to access elements in a SOAP payload. AxisJava uses Xerces, which is modular and flexible but inefficient for large payloads. gSOAP also uses a custom pull parser during the deserialization phase.

• Events and MIOs: XSUL dynamically re-allocates memory as it retrieves new XML nodes while deserializing the XML graphs. This hurts its performance for events and MIOs (Figures 3.17 and 3.18). XSUL's performance exceeds AxisJava for less than 10,000 MIOs and 15,000 events. gSOAP outperforms both AxisJava and XSUL for all sizes of MIOs and events. Figure 3.19 shows a similar pattern for Windows-based toolkits; .NET outperforms AxisJava by 50% for large array sizes.

• Base64 Encoding: Figures 3.20 and 3.21 compare end-to-end and serialization performance respectively for Base64 encoding. Results show that gSOAP and XSUL outperform AxisJava. For end-to-end performance, AxisJava is slower than XSUL by a factor of 2.3, while XSUL is slower than gSOAP by a factor of 1.6. For serialization, XSUL takes 49% more time than gSOAP to complete the benchmark for 25,000 array elements.

• End-to-End Performance: When compared in isolation, XSUL outperforms AxisJava for deserializa-
Figure 3.19: We compare end-to-end (serialization and deserialization) for .NET, AxisJava, and XSUL on Windows. XSUL’s performance drops for handling more than 10,000 events. For sizes greater than 12,000, AxisJava takes an average of 50% more time than .NET to respond.

8 Observations on Current Toolkits based on Benchmark Results

We briefly describe the conclusions that can be drawn from our performance study of the current versions of five toolkits. The benchmarks and the associated drivers facilitate in repeating these tests for newer toolkits and to study the effect of improvements that are added to existing SOAP implementations.

- If low latency is critical, gSOAP is the ideal choice. On Windows, XSUL or .NET have comparable latency. AxisJava is not optimized for low latency requirements on either Windows or Linux.
- MCS [68] should use gSOAP or the .NET environment as these two toolkits scale well with increase in the size of complex data types. These toolkits can minimize the Web service overhead, which was identified as the primary bottleneck in the use of Web services with MCS. AxisJava, which is currently used, can severely hurt the scalability of the system.
- On Java-based Linux environments, XSUL should be used if arrays of primitives need to be sent or received. However, while XSUL can be used to send complex types, its performance does not scale
Figure 3.20: This graph shows the performance results for end-to-end performance of binary data that has been base64 encoded (a feature provided by each of the toolkits). XSUL outperforms AxisJava, while gSOAP is consistently better than XSUL. For an array of size 10,000, XSUL is slower than gSOAP by a factor of 1.6, while AxisJava is slower than XSUL by a factor of 2.3.

- For grid applications that repeatedly exchange data with similar structure, such as exchange of ClasSAds between cluster managers in Flock of Condors [15], bSOAP performs extremely well. It also has impressive performance gains when only a small percentage of data changes during subsequent sends. bSOAP’s caching mechanism is targeted towards such applications.
- bSOAP is comparable to gSOAP for sending arrays of doubles, integers and strings. However, for receiving data or end-to-end communication, gSOAP should be used.
- The use of streaming and chunking greatly improves the performance of gSOAP. With AxisJava, however, it makes minimal difference for serialization in AxisJava, though deserialization is helped. Whenever possible, persistent socket connections should be used for SOAP calls.
- On Windows, .NET is highly optimized for all the benchmarks. Its performance is comparable or better than XSUL for all data types. If available, .NET should be used instead of XSUL or AxisJava.
- The WSRF-Java implementation [71] uses the AxisJava toolkit. Our performance results show that XSUL, .NET or gSOAP based toolkits will be more efficient.

Axis Java is a widely-used toolkit under active development. While it performed poorly compared to the others in our benchmark tests, we expect to see improvements in the future.
Figure 3.21: This graph shows the performance of AxisJava, gSOAP and XSUL for serializing data in Base64 format. AxisJava performs poorly compared to XSUL and gSOAP. XSUL takes 49% more time to complete than gSOAP for arrays of 25,000 elements.
**Figure 3.22:** In this graph we show the end-to-end performance for array of integers for AxisJava, XSUL and gSOAP toolkits. This requires each toolkit to deserialize, then re-serialize the input array. For an array of size 10,000 elements, AxisJava’s time is a factor of 3.4 greater than XSUL’s. For the same size, XSUL’s time is a factor of 6.2 greater than that of gSOAP.
Figure 3.23: Similar to Figure 3.22, but using doubles instead of integers, we compare end-to-end performance. For an array of size 10,000 elements, the time taken by AxisJava is a factor of 3.0 more than XSUL. For the same size, XSUL takes a factor of 3.9 times more than gSOAP.

Figure 3.24: We studied the effect of streaming by making AxisJava and gSOAP deserialize a large number of events with and without streaming enabled. AxisJava consistently performs better when streaming is used, taking 22% less time to complete for the 10,000 element array. gSOAP also trims its time with chunking on this test by 22%.
Figure 3.25: We studied the effect of streaming by making AxisJava and gSOAP serialize arrays of doubles with and without streaming enabled. Unlike in the case of deserialization of events (see Figure 3.24), streaming does not improve AxisJava’s serialization performance. Streaming helps improve gSOAP’s performance for all sizes up to 25,000 elements, the maximum we tested for this benchmark. By enabling streaming, gSOAP’s average response time for 10,000 elements was reduced by 42%.
Figure 3.26: Differential serialization is optimized for use-cases when subsequent sends are similar. In this benchmark, we measured the performance of bSOAP when 0% (bSOAP0), 25% (bSOAP25), 50% (bSOAP50), 75% (bSOAP75), and 100% (bSOAP100) of the values need to be re-serialized, rather than copied, for each message from the previous message. As expected, bSOAP's optimizations reduce the serialization time as the percentage of values that need to be re-serialized is reduced. For 100,000 doubles, the best case bSOAP0 is faster than the worst case bSOAP100 by a factor of 6.7.
Chapter 4
Utilizing Spare Cores to Read Ahead

1 Introduction
This chapter proposes two new parsers, **Runahead** and **Readahead**\(^2\), both based on the Piccolo parser, which was shown to be high performing in chapter 3, section 3.

2 Architectural Changes
In approaching the problem of processing grid-scale XML datasets, we examined a number of parsers. From our benchmarking work [43, 44], we learned about the performance characteristics of a number of XML parsers. Based on the results in Figure 3.7, 3.8, 3.9, we can conclude that XML parsers do not scale well when the size of the document to be processed is very large. Also, among the widely used Java-based XML parsers, Piccolo [60] has the best performance for typical payloads such as SOAP-based serialization of arrays for integers and strings, and example XML documents used in grid applications. We wanted to work with a high performance parser, and also one that uses scanner and parser generators such as Flex [29] and Bison [12]. A table-driven, automata-based parser is necessary to analyze the manipulate the lexical analyzer, and a generator-based implementation affords a more generally applicable solution – improving the performance of code made by a generator may be applicable to more parsers than just the one under investigation. After studying some cumbersome C-based parsers, we decided to switch to Java and use Piccolo [60], which is designed for high performance serial processing, and is implemented using scanner and parser generators [40].

Piccolo’s lexical analyzer is a single-threaded table-driven state machine. The input is fed through the state machine to generate the lexical chunks (words), which are fed into the parser. The parser determines the grammatical structure of the input. The output of the parser is either a sequence of SAX events or DOM tree.

One of the goals of this project in the thesis is to modify the lexical analyzer so that it can scan the input using multiple threads, which run on multiple cores, though still within the same JVM/address space. Recent Java implementations on recent Linux threading implementations effectively schedule runnable threads across multiple cores. This can be observed by, for example, running `top` when a multithreaded application is running and noticing that the %CPU field for the application rises above 100%. The solutions presented below are the result of two separate, but related designs, and an unmodified base case. These are the **Base**, **Runahead**, and **Readahead** parsers. The Base parser is simply the unmodified Piccolo parser, whose design will not be discussed further.

- The **Runahead parser** is a two-threaded design that starts an additional thread, called the `runahead thread`, when the parsing begins, in addition to the main thread where the actual scanning and parsing takes place. These threads do not communicate at all. In fact, the main thread is equivalent to that of the Base parser. The entire purpose of the runahead thread is to attempt to preload the contents of the input file before main thread attempts to read those bits. There is no attempt made (and no need) to ensure that the two threads are aligned or synchronized in any way.

\(^2\)Early versions of these parsers and related analysis have been published in [40].

36
• The Readahead parser is another two-threaded design, however in this case a readahead thread is started. The readahead thread reads blocks of bytes from the disk and writes them into a pipe which is read by the main thread. The main thread is equivalent to the Base parser with one significant difference: the main thread reads from a pipe rather than directly from a file. The only synchronization between the threads occurs when they access the pipe. The goal is to manually load the input into an application-managed data buffer, rather than simply using the operating system’s file cache as the Runahead parser does.

We run the three parsers through three sets of tests. In each test, we use the same large input file. The tests report just the time the parser spends scanning and parsing the input, without the interference of user code.

• The precached data test runs each parser on input that is cached entirely in memory. The measures the performance of the parsers in the absence of disk I/O.

• The uncached data test is run to expose differences between the parsers when disk I/O is required to read the input file.

3 Performance Results

We use a 683MB XML file representing a protein sequence database [64] located on the local hard drive to eliminate network traffic complications. We use the latest available release of Java runtime at the time: 1.6.0-b105, and the version of Piccolo we modified is 1.04. The underlying operating system for these tests is Ubuntu 7.04 beta (Feisty Fawn) [46], which provides a patched Linux 2.6.20 kernel. We had exclusive access to the machine during the test runs to minimize external system effects on our results. Because we are using a Java runtime, which has a progressive just-in-time compiler (JIT), we ensured to “warm up” The JIT to reduce the chance that run-time compilation processes will interfere with our measurements. We use R [65] to analyze and plot the results.

In the past, the Java implementation on Linux suffered from a non-native (“green”) thread implementation. This was, however rectified with the 1.3 release of the language runtime [7]. Linux has also traditionally had a poorly performance thread implementation, but this has been mostly resolved with the 2.6 kernel release [25]. Even though there is constant effort to improve implementation of the Java specification, the Java 1.6 runtime on Linux 2.6 kernels is considered a stable environment in which to perform test multi-threaded algorithms. Periodic inspections of the output of the top program during the test runs confirms that indeed two threads are being scheduled and run on the multi-core system. The %CPU field will rise above 100%, indicating that more than one thread is being scheduled and run in a time slice.

The performance measurements for the dual core (CMP) system are taken on a Dell Precision 390 Workstation with an Intel Core 2 CPU 6600 clocked at 2.40GHz with 3GB of dual-banked 1.9ns PC2-4200 SDRAM. The hard drive is a 160GB SATA drive running at 7200RPM with a 8MB on-drive buffer.

The SMP machine is a IBM eServer cluster node with dual Pentium 4 Xeons, each clocked at 3.2Ghz with 4GB of DDR RAM and a 15000 RPM Ultra-320 SCSI hard drive. This machine is running an older version of Ubuntu which uses the 2.6.15 kernel.

The uniprocessor system is a Dell Precision 370 Workstation with an Intel Pentium 4 CPU clocked at 3.00GHz with 2.5GB of dual-banked 1.9ns PC2-4200 SDRAM. The hard drive is a 160GB SATA drive running at 7200RPM with an 8MB on-drive buffer.

The test code is the same for all tests, though the parameters such as number of times to run the internal loops are varied via commandline options. The tests follow this simple algorithm:

1. Warm up the JIT compiler and parse a given warmup file once. For cached cases, this is the same file that will be timed, for uncached cases, this is a different very large file.
2. For the number of parses specified, run the parser on the test file, timing inside the loop using System.nanoTime() to achieve the highest resolution timer available in Java.
3. The parser uses a SAX event-based interface, which requires callbacks to be implemented. The callbacks functions do not conduct any processing, and allow us to time just the actions of the parser itself.

Each test case is run 5 times (due to time constraints, we prefer to run the tests 20 times to get clearer results). We run two distinct sets of tests: precached data, and uncached data.
3.1 Precached data

In this test, we run the timing loop 50 times per parser, with 20 parses per loop, on the psd7003.xml input, also using it as the warmup file to ensure that it was in the system cache. The results of this test are shown in figures 4.1 – 4.3. The results are normalized by dividing each data point by 20 (the number of parses per loop) and then divided by the average of the Base parser run to get the speedup, which is (more) comparable to the results from the other architectures.

Runahead and Readahead should provide little performance benefit in the precached test because they both work on the principle that accessing the file may be slow, and overlapping the processing of the XML file (in the parser/CPU bound thread) with access to disk (in the runahead/IO bound thread) could provide some overlap. Unsurprisingly the parsers on all the architectures remain fairly close the Base parser, though Readahead notably underperforms in all cases, as seen in figure 4.1. It is interesting to note in figure 4.3 that Runahead actually provides a measurable, if slight speedup, despite the fact that the file is already in the operating system file cache.

3.2 Uncached data

In this test, we run the timing loop 10 times per parser, with 1 parse per loop, on the psd7003.xml input. We use a bit-for-bit copy of the same file to warm up the JIT as well as 7 other copies to flush the input file from the cache as best as possible before each test run. The results are displayed in figures 4.4 – 4.6.

Perhaps the most surprising result here is that Runahead provides a nearly $1.4 \times$ speedup when the data must be read from disk. This is less surprising when considering that the SMP machine actually has the greatest disk bandwidth which leaves the most opportunity for the IO bound thread to “get ahead” of the CPU.

Figure 4.1: Rather than seeing any improvement, Readahead exhibits a slowdown on all architectures when the input is cached
Figure 4.2: Both parsers exhibit a slowdown on the uniprocessor machine, Readahead more than Runahead

bound thread. Interestingly, Readahead (which uses a synchronized pipe buffer between the IO bound thread and the CPU bound thread) provides little to no help. This also indicates that Runahead may be of more use when the parser is in actual use. Since our test here ignores all SAX events. If they were being handled, the Runahead thread might have more chance to proceed further along the input.

4 Conclusion

The goal of this work is not to immediately improve on XML parsing performance, though we found substantial benefit with Runahead on a SMP machine with wide disk bandwidth, but to begin to attack the problem with a performance data-oriented approach. One other useful result here is that, for files of this large size, the overhead of pipe synchronization is constant and reasonable (5%) on machines with more than one CPU. This means that pipes (as implemented by java.io.PipedOutputStream and java.io.PipedInputStream in Java 1.6) could be a feasible communication mechanism for future multithreaded designs. It should be noted that the implementations of those classes in Java 1.5 is not suitable: they were rewritten in 1.6 to provide optimized throughput on large, multi-byte transfers. This caused some performance problems for use early in this project when we were employing only a Java 1.5 runtime.

One reason these parsers may work as well as they do on these machines is that three gigabytes of RAM leaves plenty of room for the file cache. On machines with less memory available, we expect the parsers to perform worse.

It should be noted that the traditional uniprocessor machine exhibited a slowdown for all parsers and conditions in the test. This suggests that existing operating systems, filesystems, and libraries are highly optimized for uniprocessor machines. Trying to “tweak” a little more performance by preloading the file
Figure 4.3: On CMP, the difference between the two parsers is a bit more pronounced, with Runahead providing a minimal speedup and Readahead exhibiting slowdown.

Only generates more interrupts which just slow down the CPU bound parsing thread.

The operating system page cache and buffered I/O routines perform a similar duty to Runahead’s and Readahead’s read thread, but it is notable that the experiments show that Runahead performs favorably compared to what the OS level page cache provides automatically. Modern operating systems do provide specialized, native, runahead-like mechanisms, such as the `readahead(2)` system call, which takes a file descriptor, an offset, and a count as inputs. It blocks until the portion of the file from the offset to the count is read into the page cache. This call is frequently used to improve application and operating system start up time by reading all files required for initialization just before the start up routines are invoked. Services could be envisioned that monitor file access patterns to predictively invoke `readahead(2)`. Indeed, Runahead’s secondary thread could be implemented using calls to this call. It is left to future work to investigate these possibilities.
Figure 4.4: Runahead really shines here, providing a speedup of nearly 1.4 for the SMP machine – note that the SMP machine has a 15000 RPM Ultra320 SCSI drive while the other two have 7200 RPM SATA drives.
Figure 4.5: As in figure 4.4, Runahead provides a very nice speedup on the SMP machine with a very fast hard drive when the file must be read from disk.
Figure 4.6: On the dual core machine, Runahead does provide some speedup, around 1.08, while readahead again exhibits (some) slowdown.
Chapter 5

Parallel Processing with PIXIMAL

1 Introduction

PIXIMAL-DFA is a hand-coded DFA-based table-driven SAX parser. Its input is an XML file and its output is a sequence of SAX events that a developer-user will receive at runtime. It is not meant to be a complete XML parser supporting all possible features of the language, rather, it is a test-bed for the approach outlined below. PIXIMAL-NFA is a parser based on PIXIMAL-DFA that a DFA instances and several NFA instances to make use of additional, unused processing cores. We use PIXIMAL\textsuperscript{3} to refer to the union of PIXIMAL-DFA and PIXIMAL-NFA. The “PIXIMAL approach” encompasses the process of creating a DFA-based parser, constructing an NFA-based parser as described below, and running one automaton on each processing core.

2 A hand-coded DFA-based parser: PIXIMAL-DFA

PIXIMAL-DFA is represented by a DFA, \(P_{DFA}\) and a set of actions that are triggered at a subset of the transitions. Processing proceeds by initializing a state variable to be the start state known by value 0. As each character of input \(c\) is read, state and \(c\) are looked up in the table to find the new state, namely newstate. Another lookup table indexed by both state and newstate is consulted to determine if an action is required for the transition state \(\rightarrow\) newstate. If so, the correct action is performed. Some actions may store an attribute name or value for later, or may trigger a SAX event in the client code (code that uses PIXIMAL via the SAX-style interface).

The goal of PIXIMAL-DFA is not to exercise the memory and I/O subsystems though string copies, but instead to examine the approach under optimal conditions. Rather than reading input using buffered I/O, it uses \texttt{mmap(2)} to load the input into memory. Together with the use of “pascal-style” strings that combine a string length with a \texttt{char*} rather than using NULL-terminated \texttt{char*}, this allows the parser to avoid string copies through the entire parse process, even up through client code. The use of the length value avoids modifying characters, i.e., inserting NULL values, at the end of strings pointing into the \texttt{mmap}ed memory block.

The graph of states and transitions is presented in figure 5.1. The symbolic names for these states are described below in figure 5.2. An implicit Error State with numeric value 11 exists with transitions from all other states for input characters which are not otherwise represented in figure 5.1.

3 A parallelized NFA-based parser: PIXIMAL-NFA

The approach of PIXIMAL-NFA is similar to that of Meta-DFA, described in chapter 2, section 5. The approach is to partition the input file and attempt to apply a DFA-based parser to each element of that partition. This is not possible on its face because a correct start state would not be known for the parsers that start anywhere but at the initial character of the file, so it is necessary to transform the DFA-based parser so that it can be applied.

The DFA above, is transformed into an NFA, \(P_{NFA}\), containing precisely the same state nodes, transitions, and final states as \(P_{DFA}\). One significant change is made: each state node, with the exception of the error

\textsuperscript{3}The description of PIXIMAL has been published in [41] and [42].
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 $P_{DFA}$ 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 and all the NFAs running on the input preceding it are complete. 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 at greater complexity, 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 fixed. When any execution path, $k$, through the NFA leads to the Error state, it 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 $P_{DFA}$ 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 and all the NFAs running on the input preceding it are complete. 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 at greater complexity, 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 fixed. When any execution path, $k$, through the NFA leads to the Error
State, processing on that path terminates and the the action list \( A_S \) is freed.

### 3.1 Division of work

In the current implementation, PIXIMAL-NFA divides work between the *initial DFA* and subsequent NFAs in a straightforward manner. A *split percent* is chosen ahead of time which determines how much of the input file will be given to the initial DFA. The remainder of the input is divided evenly among thread count \(-1\) NFAs, each of which runs in its own thread.

### 3.2 Contrast with Meta-DFA

PIXIMAL-NFA takes an approach similar to Meta-DFA, but there are distinctions. PIXIMAL (-NFA and -DFA) are SAX-style parsers: they generate a sequence of events. Meta-DFA, on the other hand, is a DOM parser, which outputs a tree of objects representing the input file. Different applications have varying needs, and some may be better suited to DOM parsing while others are better suited to SAX parsing. This indicates that it is appropriate to examine both parsing styles. The task of parallelizing a SAX parser may in some sense be more challenging than parallelizing a DOM parser because in general there is less work done by SAX parsers, and a great deal of it is sequential in nature: both reading input from disk as well as outputting the correct stream of SAX events.

There are further implementation differences between the two parsers. PIXIMAL-NFA fully implements a parser using a combination of DFA- and NFA-based parsing threads, whereas Meta-DFA implements just the pre-parsing portion with the Meta-DFA, the rest of the processing is handled by PXP. This allows Meta-DFA to be implemented by a smaller DFA and allows its output to be substantially simpler than PIXIMAL-NFA. This leads to more challenges and room for optimization for PIXIMAL.

Meta-DFA creates a new DFA, whereas PIXIMAL-NFA creates an NFA to fulfill the same need. It will be interesting to study whether the cost of the exponential state explosion of generating such a DFA is less than the cost of tracing multiple execution paths through an NFA. We consider this detailed study to be beyond the scope of this thesis, and plan to discuss it in the future work section of the final thesis. The NFA abstraction may also enable alternate approaches such as dividing the work of an NFA across multiple threads; NFAs inherently allow concurrency, and it may be beneficial to schedule subsets of execution path traces concurrently across multiple cores. This thesis does not address this particular approach, though.

### 3.3 Optimization opportunities with PIXIMAL

The PIXIMAL design allows a wide range of application-specific optimizations to be carried out. Execution of each \( P_{NFA} \) results in consumption of CPU and memory resources. In the worst case scenario, the execution of all possible \( P_{NFA} \) along all possible execution paths. A major optimization target is therefore to reduce the number of \( P_{NFA} \) to be executed using heuristic methods or by elimination of paths due to other considerations. Methods of accomplishing this as well as other optimizations are discussed below.

- **Minimize memory load in NFA.** If memory load is too high, due to the NFAs’ storage of actions and all parameters, it may be an overall performance improvement to reduce the work done by each NFA. Instead of storing all parameters, store just history of state transitions and reference the required actions. The catch-up required will be higher, since the catch-up mechanism will need to consult the input stream to complete the actions, but may lead to an overall performance win because the catch-up should be faster than simply running the original DFA.
- **Scan around the NFA starting point.** The location at which some \( P_{NFA} \) is chosen to start may not be optimal. For example, it may be just after a sentinel character such as \( '<' \), which would reduce the number of possible start states. Many execution paths simply can not start with a left angle bracket, so scanning backwards a little from the starting point could drastically reduce the number of execution paths.
- **Process only the most likely execution paths.** The previous two optimizations are guaranteed to do all work required, but they may end up performing a lot of work. If this guarantee is relaxed, more execution paths can be dropped with potential performance gains. This would involve collecting profiling data on \( P_{DFA} \) on various inputs and determining which states in the DFA are most frequently occupied. The start states in \( P_{NFA} \) could be pruned to avoid working on the least likely paths. In a service engine that repeatedly processes many documents that fall into some pre-determined classes, profiles for each class of document could be collected to create a specialized \( P_{NFA} \).
• **Convert \( P_{\text{NFA}} \) 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 \( P_{\text{NFA}} \) into an equivalent DFA and reduce it using well known 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. In future work, we plan to explore optimal layouts. There’s room for some theoretical work to decide how to optimally layout the state table which could be based around more complex analysis: for instance, examine DFA trace data for transition loops \((i\rightarrow j\rightarrow k\rightarrow i)\) and then arrange the DFA table so that the rows associated with those states are adjacent.

• **Parse the input in chunks to reduce NFA memory load.** Instead of partitioning the entire input into \( N \) parts and processing the entire document in parallel, which leads to memory usage proportional to input size, divide the input into more manageable chunks and process each of these chunks in a sequence of parallel parsing steps. This sets a bound on memory usage for the NFA action history, because each NFA will only process a limited size input. In this case, we’re dividing the input evenly into \( M \) sections of size \( T, S_1, S_2, \ldots, S_M \), and processing each \( S_i \) by partitioning it further into \( N \) parts, \( P_1(S_i), P_2(S_i), \ldots, P_N(S_i) \), and using a DFA to process \( P_1(S_i) \) and NFAs to process the rest. This guarantees that no NFA will need more than (some) \( f(T) \) memory to operate. If all the processing is balanced, there should be little overhead compared to the un-optimized PiXIMAL case.

• **Integrate with 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 queuing them. An XSLT-like 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 along with the NFA, but this would convert the memory load into disk usage.

### 4 PXML: An XML-like Language Optimized for PiXIMAL

As will be seen in chapter 6, it beneficial to reduce the size of \( P_{\text{DFA}} \), the PiXIMAL DFA. Even with the limited portion of the XML specification that PiXIMAL supports, the requisite state machine is still quite large, at 11 states. This leads to many execution paths to track in each \( P_{\text{NFA}} \). As can be seen in figure 5.2, a four states are dedicated to processing attributes. Another state is necessary to handle whitespace in end tags.

In addition to reducing the number of states, it is also beneficial to increase the number of execution paths that lead to contradictions (i.e., transitioning to the Error State) so resources dedicated to processing those paths can be freed. It turns out that whitespace in tags and the fact that \( \langle \) can appear in content sections prevents a number of otherwise incorrect execution paths from reaching a contradiction.

If some changes can be made to XML, PiXIMAL could have improved performance characteristics. The goal of proposing these changes is to either reduce the number of execution paths that must be maintained by either directly reducing the number of states, or by increasing the number of ways to lead to a contradiction. Figure 5.3 represents the DFA that would required to process a PXML document, with 5.4 indicating the logical names of the states in the machine. The sixth Error State is not shown.

• **Disallow attributes.** Attributes lead to several additional states in the DFA and add little to the expressivity of XML. Any attribute can be transformed into an element where the attribute’s name maps to the tag name and the attribute’s value maps to the content section.

• **Disallow whitespace in tags.** Once attributes are removed, allowing non-name text, specifically whitespace characters, inside tags adds little value. It does however prevent a \( P_{\text{NFA}} \) from deciding that it cannot be inside a tag when it reads a space character.

• **Disallow \( \langle \rangle \) in content sections.** Similar to the problem with whitespace in tags, allowing right angle brackets \( \langle \rangle \) in content sections prevents a \( P_{\text{NFA}} \) from determining that it must have been inside a tag when it encounters this character. As will be seen in chapter 6, the fact that the open angle bracket character is disallowed from content sections is very helpful for eliminating execution paths.

We accept that encoding namespace declarations is valuable use of attributes. As a compromise, PXML could be extended to allow just the top level tag in the document to contain namespace declarations as attributes. In this case, the file would be read sequentially until the first tag is read. After all the attributes have
Figure 5.3: Graph representing the DFA used in PXIMAL-PXML, state 0 is the start state, and 3 is the final state.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initial State</td>
</tr>
<tr>
<td>1</td>
<td>Enter Tag State</td>
</tr>
<tr>
<td>2</td>
<td>Start Tag State</td>
</tr>
<tr>
<td>3</td>
<td>Content State</td>
</tr>
<tr>
<td>4</td>
<td>End Tag State</td>
</tr>
</tbody>
</table>

Figure 5.4: Symbolic names of each DFA state

been read and the namespaces setup, the rest of the input file would be divided up and processed in parallel as described above.
Chapter 6
Limits of Parallel Processing with P IXIMAL

1 The Limits of P IXIMAL
In this chapter, we present an examination of the P IXIMAL approach in terms of modern, commercial computing systems. While there should theoretically be possible to obtain a performance improvement by applying NFAs on unused processors, other system-level effects may become bottlenecks. Three tests, memory bandwidth, DFA state size, and serial NFA, are described in detail. The results of running these tests on current server-class hardware are presented.

2 Examining Memory Bandwidth and DFA Size
The XML input for all the test results presented here is SwissProt.xml[28], which encodes a protein sequence database. It is roughly 109 megabytes, contains 2,977,031 elements, 2,189,859 attributes, and has a maximal tree depth of 5. All tests mmap(2) this input file to reduce memory usage and eliminate many string copies. In all cases, input is read from the local disk – not from a network file system. Additionally, the input file is pre-read into the operating system’s disk buffer, so the tests stress the CPU/RAM interface.

We run these tests on a range of differently configured nodes:

- 2× uniprocessor – 1U nodes in a cluster, each of which has two 3.2Ghz Intel Xeon CPUs, 4 gigabytes of RAM and run a 64 bit version of Linux 2.6.15. Results on this class of machines are taken by averaging the timings found on 4 of these nodes. The filesystem in use in the test directory here is reiserfs.
- 2× dual core – 1U nodes in a cluster, each of which has two 2.66Ghz Intel Xeon 5150 CPUs, 8 gigabytes of RAM and run a 64 bit version of Linux 2.6.18. Results on this class of machines are taken by averaging the timings found by running the test on 10 of these nodes. The filesystem in use in the test directory here is xfs.
- 2× quad core – 1U nodes in a cluster, each of which has two 2.33Ghz Intel Xeon E5345 CPUs, 8 gigabytes of RAM and run a 64 bit version of Linux 2.6.18. 10 nodes from this cluster were selected to perform this test and the results presented reflect the mean timings taken. The test directory on these machines is backed by a xfs filesystem.

2.1 Memory bandwidth test
An N-way parallel parser would concurrently be reading using N different threads, so this tests checks 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 P IXIMAL 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

---

4The memory bandwidth and state scalability portions of the chapter have been published in [41] and the portions discussing the serial NFA tests has been accepted for publication in [42].
number_of_threads = 1, split_percent is overridden to be 100% in order to ensure that the entire input is read.

In this test, the DFA is modeled by a thread that reads each byte of memory in its partition of the input and looks up a value in a table indexed by that byte. The NFAs are modeled by threads that do precisely the same task on their given partition of the input.

2.2 State scalability test

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.

3 Bandwidth and Size Test: Results and Discussion

The focus of PIXIMAL is to achieve “scalable parallelism,” wherein processing of large-scale XML data continues to get good performance as the number of cores increases. The performance results are presented for classic 2× uniprocessor (two total cores), 2× dual core (four total cores), and 2× quad core (eight total cores) configurations (all SMP), as these are the processor configurations that are in use today. The trends on quad-core and 8-core nodes are good indicators for how PIXIMAL will scale to a larger number of cores, as they become available.

It is to be noted that the PIXIMAL parallelization approach of providing scalability over multiple cores may be applicable to other table-driven or DFA-based parsers. Also, though we have conducted extensive tests on large-scale XML documents, the approach is tailored to potentially work for a lexical analyzer for any structured application data format.

In the results below, speedup is always calculated using the formula: \( T_1 / T_P \), where \( T_1 \) is the mean time taken when no NFA threads are scheduled and \( T_P \) is the mean time taken at the particular data point – over all test runs where the test parameters and hardware configuration are identical. Mean raw timing values are used to provide an overview of the actual result space, while speedup is used to make certain results comparable across hardware.

3.1 Memory bandwidth results

The memory bandwidth test models the memory work being done by the DFA thread and the NFA threads by sequentially reading the input and passing it through a table. NFA threads do the same amount of work as the DFA and are set to read the sections of input that the actual PIXIMAL NFA would, and we clearly see performance wins by adding extra threads/CPUs to the task of reading the input. This suggests that we’re not causing a serious reduction in cache performance simply by reading from multiple sources of input. It also demonstrates that access to main memory is not a major limiting factor in this approach: each thread is still able to get enough data to do its (small amount of) work. This scales with the number of cores up to around 6-7 cores in the 8-core case.

Figures 6.1 through 6.6 examine the entire parameter space of this test. All split_percent and thread_counts are displayed over the range of values tested. This provides a great deal of visual information about the test space. It is quite clear that adding threads can provide an advantage when there are spare cores to use. It is also instructive to note that once the best time is achieved by for a given split_percent, adding more threads does not detract much from overall performance. This may be due to high performance I/O subsystems and file systems on these cluster nodes. The best performing configuration is the one denoted by the deepest part of the well. The bandwidth (input_size * parse_time) is plotted in blue. The axes are flipped from the time plot because the shape of the curve is flipped, due to the inversion of time in the bandwidth computation.
Figure 6.1: *Input read times of the memory bandwidth test for each split_percent and thread count on a standard (2 × uniprocessor). Note the valley when two threads are used with a 50% input split.*

Figure 6.2: *Bandwidth achieved at each split_percent and thread count on a the 2 × uniprocessor testbed. In contrast to figure 6.1, the axes are inverted to provide a better view of the space. The maximum bandwidth achieved was 0.94 GiB/s.*

51
Figure 6.3: Input read times of the memory bandwidth test on an 2× dual core configuration (4 total cores). Note that performance increases past the valley seen in figure 6.1.

Figure 6.4: Bandwidth achieved at each test point on the 2× dual core testbed. The axes are inverted from figure 6.3 to better show the data. The maximum bandwidth achieved was 2.28 GiB/s.
Figure 6.5: Input read times of the memory bandwidth test on an 2× quad core configuration (8 total cores). Note that adding threads continues to improve performance, though returns diminish past the 6th thread.

Figure 6.6: Bandwidth achieved at each point on the 2× quad core testbed. The maximum bandwidth achieve in this space was 2.69 GiB/s.
Figure 6.7: Speedup comparison on the 2× dual core (4 total cores) configuration for the memory bandwidth test. split_percent s are selected to maximize the speedup for each number of threads.

Figures 6.7 and 6.8 plot the speedup (over the DFA-only case) of the best split_percent for each of varying thread counts on the 2× dual-core and 2× quad-core systems. The split_percent s plotted are chosen because they maximize speedup (compared to all other split_percent s) at the thread_counts plotted. The thread_counts are chosen by the number of CPU cores available on the cluster class. These plots reaffirm the conclusions from the three dimensional plots: (1) shifting the split_percent back affords greater scalability, even up to 6 threads on 8 core machines. They also demonstrate that given an optimal split for a chosen number of threads, performance does not degrade substantially after passing that split_percent. Figure 6.8 additionally demonstrates how returns do eventually diminish as more than 6 threads are used.

Figure 6.9 plots the speedup for the best-case speedup for each CPU configuration, for its performance on the best-case split_percent. This facilitates comparison across configurations. The 4 and 8 CPU cases fare worse than the 2 CPU case at first just because this split_percent is fixed for each configuration in this plot. This approach was chosen to show how performance would be affected by choosing a particular configuration based on the detected machine type. It is clear that even 8 total CPU configurations can potentially provide performance improvements, even with current memory and I/O subsystems.

The structure of these tests imply that conclusions drawn from them apply to most parallel data access algorithms on a single machine. Each thread does the minimal amount of work so that the test is effectively working just the memory subsystem. If another data file type is to be read in parallel by multiple threads or processes each operating sequentially, such an input reader will need to do at least this work. If the workload is dominated by memory access, then these results will apply directly and it is unlikely that attempting to utilize more than 6 threads will yield much benefit on machines that are configured like those tested here.

3.2 State scalability results

The state scalability models the amount of work each combination of DFA/NFA threads would have to do given a certain language/parser complexity. We define DFA complexity by the number of states (N) in the
DFA. An NFA (for this particular test) based on a DFA of complexity $N$ will need to do $O(N)$ times more work than the DFA: for each state in the DFA, the NFA must calculate the next state for each character of input, whereas the DFA must only calculate the next state for one state, namely the distinguished start state.

Figures 6.10 and 6.11 show the best timings for each combination of DFA complexity and thread count. It is interesting to note that for most cases, it is possible to improve overall performance by taking advantage of unused cores, even when the number of threads outnumbers the cores available. There is also a significant contrast between the 2× dual core and 2× quad core configuration: the 2× quad core case shows a stable speedup, which is not indicated in the 2× dual core case. It is also apparent that the best speedup obtainable for a given DFA complexity is when the number of threads matches the number of cores. This is not a clear result of the memory bandwidth test because the extra NFA threads did not add much extra work. Here, each additional NFA does add a substantial amount of work, which increases along with the complexity of the DFA. The final item to note on these perspective graphs is the steep ledge around DFA size 6. This indicates a major crossover point where the complexity of the NFA begins to take over the bulk of the work from the memory subsystem. Greater speedup is more readily obtainable before this point. The ledge above 4 and 8 threads on the respective figures also becomes more pronounced, again indicating that a significant crossover point has been met.

Figures 6.12 and 6.13 are plotted similarly to figures 6.7 and 6.8. That is, given a configuration (number of cores, number of DFA states), a split point is chosen that applies for all threads. The split point, which can be

\[ \text{Figure 6.8: As in figure 6.7, but for an } 2\times \text{ quad core (8 total cores) configuration. As was visible in figure 6.5, performance gains trail off after 6 threads are given to the test.} \]
thought of as the globally optimal split point for a given DFA size is chosen to maximize the speedup across all thread counts shown on the horizontal axis. This is done to show how the parser in a given configuration acts as the number of NFAs used is increased. Because of this, there is a slowdown shown for some cases. In those cases, a better split point could have been chosen that would demonstrated some speedup, specifically, the split point chosen for display for that configuration in figure 6.10 or 6.11. A variety of DFA state sizes are plotted which reinforce a conclusion from above: there is a major changeover when DFA complexity is greater than 6. Examining the globally optimal split_percent for each state suggests why this is so: the split point must shift much farther into the document. For example, in the 2× dual core case, a 6 state DFA yields an optimal split_percent of 36%, slightly more than that of the 4 state DFA case. The 8 state DFA yields an optimal split_percent of 56%: the best performance is achieved here when the DFA thread processes over half of the input. Under such circumstances, it will be impossible to achieve even a 2× speedup over the single threaded DFA case, no matter the number of cores available. The situation is similar on the 2× quad core hardware. At between 6 and 8 states, the optimal split_percent jumps disproportionately, again due to this crossover where read time begins to be dominated by the complexity of the DFA/NFA rather than simply reading the input from memory.

Of special note here is that in figure 6.13, the maximal speedup is greater than that shown in 6.9 (5 vs. 3.5). This is likely due to the fact that the baseline comparison case (the DFA model running over the entire input) is doing substantially more work, so dividing that work out to the NFAs is a significant win, particularly when the number of states is small, because the extra work done by the NFAs is not large. In other words, the NFAs are doing $O(N)$ more work than the DFA, but when $N$ is small, the constants dominate and the
Figure 6.10: The lowest input read times for all DFA sizes and thread counts for the 2× dual core (4 total cores) configuration. For a given DFA size and number of threads, there is a range of possible split_percent values that could be chosen. The split_percent is chosen for each configuration of DFA size and number of threads to minimize the time.

Figure 6.11: Input total read times for 2× quad core (8 total cores) configurations. In contrast to figure 6.10, there is a continued performance increase over the space of parameters tested.
Figure 6.12: Speedup for a variety of DFA sizes for 2× dual core (4 total cores) configurations. This test models a part of the extra work done by the NFAs over the DFA. For each DFA size, the split percent is chosen which provides the maximal speedup for some number of threads. It is clear that NFAs built from larger sized DFAs incur stuff performance penalties.

constants here are small.

4 Serial NFA Tests

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 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
Figure 6.13: Same as figure 6.12, but for the 2× quad core (8 total cores) configuration. 6 states appears to be a good DFA size for this particular model of the parallel parser.

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.

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 [65].

4.2 Serial NFA results

Figure 5.2 presents the symbolic names used in figures 6.17 and 6.21. The DFA we have built has eleven functional states and one (unlabeled) error state.

Figures 6.14, 6.15, and 6.17 present results for a representative input case: a SOAP-encoded array of 10,000 integers. Figure 6.14 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 6.15 presents the same data as figure 6.14 along a different axis, tracking split percent rather than thread 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 6.17 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 5.2. 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 6.18 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 6.14, 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 6.15, figure 6.19 displays the potential speedup when reading an array of 10,000 strings as a function of a pre-determined split percent. The results are much smoother for strings than for integers. Naturally, the high point here is the same as in figure 6.18, 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 $\frac{26}{74} \approx 2.5$ times as much work as the DFA when the work is divided well.

Figure 6.21 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.

Figures 6.22, 6.23, and 6.24 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 6.23 is more uniform and hovers around 3.0-3.2. Even the mean performance shows some speedup across all input sizes.
Figure 6.17: Histogram of DFA state usage when parsing the XML encoded array of 10,000 integers. State 7 represents characters in PCDATA sections (text between tags). Other state names are described in figure 5.2.

Figure 6.25 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.

5 Conclusions

- The PIXIMAL framework allows application programmers to quantify the exact number of threads, processing cores, and split percentage for each core, that should be used for their application data files.
- 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 `mmap(2)` ed files...
is not a strict limiting factor on thread-scalability.

- Memory-bandwidth tests show that when multiple threads process application data, such as structured XML files, the performance scales to 6-7 cores, in the 8-core case. Also, adding threads continues to improve performance up to 6 threads, in the 8-core case.

- The division of work between threads plays a key role in enhancing the performance. Shifting the split_percent back affords greater scalability, up to 6 threads on 8 core machines. Given an optimal split for a chosen number of threads, performance does not degrade substantially after passing that split_percent. PIXIMAL framework can be use to determine the major crossover point where the complexity of the NFA begins to take over the bulk of the work from the memory subsystem.

- It is critical that the complexity of the DFA remain low. Our tests indicated that such a DFA should have no more than 6 states. A lower complexity DFA implies a lower complexity language: a language with less syntax. This implies that if a given grid application dataset is guaranteed to use a restricted set of XML features (instead of attributes, define sub-elements, for example), the data can be efficiently processed in parallel on a multi-core architecture. PIXIMAL framework can be used to guide application developers to design the optimal structure for large-scale data files.

- 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 information 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
Figure 6.19: Potential \textsc{piximal} speedup on arrays of 10,000 strings as a function of split point chosen. The contrast with integer arrays (figure 6.15) is more stark here. The results are much more regular, with a clear peak around 26%.

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 \textsc{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 \textsc{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.

- 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.
Figure 6.21: 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 6.22: Effect of scaling up the size of an integer array SOAP-encoded in XML on potential scalability in PIXIMAL.
Figure 6.23: Effect of scaling up the size of a string array SOAP-encoded in XML on potential scalability in PIXIMAL.
Figure 6.24: Effect of scaling up the size of a MIOs (mesh interface objects) array SOAP-encoded in XML on potential scalability in PIXIMAL.
Figure 6.25: 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.
Chapter 7

Conclusion and Future Work

1 Future Work

Chapter 5, section 3.3 discusses a number of opportunities for optimization in PIXIMAL. These have yet to be investigated and will not be repeated here. There are, however, some larger projects to undertake in the future.

1.1 Implement PIXIMAL using a MapReduce-style framework

The PIXIMAL approach appears to lend itself to distributed processing system that divides large data files across a cluster. In these systems, such as MapReduce [23] and Hadoop [1], large files are separated into shards which are placed on hosts around the cluster. An application built into one of these framework will implement a map function which is applied to each data shard, and a reduce function which processes the result of each map function. Parallelism is achieved by deploying the map to the cluster nodes holding the independent shards of the data file.

Instead of running the parser components (the PDFA and PNFA) in separate threads on processors within a system, 1) build a map function that implements the parser components, 2) Implement a reduce function that performs the inter-component fixup routine, and 3) allow the framework to deploy them to the computer systems that hold the input shards. This would require that the application that is using the parser be built into the map and reduce functions, but could lead to substantial performance gains as the components would run independently.

1.2 Build a non-blocking I/O parser

For a particular on-disk layout of a large file, sequential access may not lead to the optimal read performance. This could be due to fragmentation or disk striping or the effect of external I/O operation. In this case, rather than requesting bytes one after the other, using an event-driven, non-blocking interface to the filesystem would fit well with the PIXIMAL approach. In such a model, the operating system would expose an interface which allows the application to request all bytes of a particular file at once. As the bytes are read from disk, they would be delivered to an event processor that would decide whether to forward the bytes to an existing PDFA, PNFA, or to create a new PNFA.

Given an implementation of PIXIMAL that uses such an operating system feature, existing application code need not be modified, because the parser would still generate the same sequence of SAX events.

1.3 Zero-copy parsing with multiple network input streams

PIXIMAL currently uses mmap to eliminate the need to copy blocks of memory of input on disk. This limits the applicability of PIXIMAL to files in a filesystem. With SOAP, XML-RPC, AJAX and similar technologies, XML is frequently read from a network source. A zero-copy parser which supports reading input from a seekable network stream, such as an HTTP server, could be written. It would open multiple connections and use a zero-copy protocol to read the data. Each connection would be processed by a separate parser component as in the usual PIXIMAL case.
1.4 Consider patch to an existing lexical analyzer generator
The ultimate goal of this work is to be able to provide an option for a tool like flex [29] to generate a parallel lexical analyzer which uses the PXIMAL approach. These tools take as input a set of regular expressions and actions and generate a DFA-based, table-driven lexical analyzer that can process an input file and output a sequence of tokens. TDX [87] is implemented using flex, and PXIMAL-DFA is modeled on the output of this tool.

The size of the DFA generated by flex for even small input files may be too large to achieve significant speedup, without additional optimizations. Such a tool could have analysis features that pinpoint portions of the set of regular expressions that limit scalability. For example, given input for XML, it might point out that ‘>’ in XML should be disallowed from content sections to reduce the number of execution paths a \( P_{NFA} \) must follow.

2 Conclusion
This thesis has dealt with the processing of XML-encoded, scientific data sets on modern workstation and cluster hardware. We have presented a benchmark suite for XML and SOAP toolkits tailored for scientific applications in chapter 3. In chapter 4, we presented analysis of the use of spare cores to populate operating system file buffers just ahead of an XML parser to improve performance. Chapter 5 presented PXIMAL, a parallel XML parser based utilizing an NFA to make use of unused processor cores to improve parser performance. The limits of this approach are discussed in detail in chapter 6.
Bibliography


[72] XSOAP Toolkit. XSOAP (previously known as SoapRMI) is a RMI system based on SOAP that is implemented both in Java and C++, 2004. http://www.extreme.indiana.edu/xgws/xsoap/.


77


