ANALYSIS AND OPTIMIZATION FOR PROCESSING GRID-SCALE XML DATASETS

Michael R. Head
Ph.D. Candidate

Grid Computing Research Laboratory
Department of Computer Science
Binghamton University
mike@cs.binghamton.edu

Tuesday, May 12, 2009
1. Introduction and Motivation
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions

2. SOAP and XML Benchmarks
   - SOAPBench
   - XMLBench

3. Parallel XML
   - Investigating System Cache Effects
   - Piximal: Parallel Approach for Processing XML

4. Related Work

5. Conclusions and Future Work
<?xml version="1.0" encoding="UTF-8"?>
<ns1:MoleculeType xsd:type="ns1:MoleculeType"
    xmlns:ns1="http://nbcr.sdsc.edu/chemistry/types"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <moleculeName xsi:type="xsd:string"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  1kzk
  </moleculeName>
  <moleculeRadius xsi:type="xsd:double" xsi:nil="true"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
  <atom xsi:type="ns1:AtomType"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <fieldName xsi:type="ns1:FieldNameType">ATOM</fieldName>
    ...  
  </atom>
  <atom xsi:type="ns1:AtomType"
      ...
    
  </atom>
  ...
</ns1:MoleculeType>
1. **Introduction and Motivation**
   - XML and SOAP
     - Ubiquity of Multi-processing Capabilities
     - Contributions

2. **SOAP and XML Benchmarks**
   - SOAPBench
   - XMLBench

3. **Parallel XML**
   - Investigating System Cache Effects
   - Piximal: Parallel Approach for Processing XML

4. **Related Work**

5. **Conclusions and Future Work**
XML Defined

- Text based (usually UTF-8 encoded)
- Tree structured
- Language independent
- Generalized data format
**Motivation from SOAP**

- Generalized RPC mechanism (supports other models, too)
- Broad industrial support
- Web Services on the Grid
  - OGSA: Open Grid Services Architecture
  - WSRF: Web Services Resource Framework
- At bottom, SOAP depends on XML
Becoming standard for many scientific datasets

- HapMap - mapping genes
- Protein Sequencing
- NASA astronomical data
- Many more instances
Explosion of Data

- Enormous increase in data from sensors, satellites, experiments, and simulations
- Use of XML to store these data is also on the rise
- XML is in use in ways it was never really intended (GB and large size files)
Scientific applications place a wide range of requirements on the communication substrate and data formats.

Simple and straightforward implementations can have a severe performance impact.
OUTLINE

1. **Introduction and Motivation**
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions

2. **SOAP and XML Benchmarks**
   - SOAPBench
   - XMLBench

3. **Parallel XML**
   - Investigating System Cache Effects
   - Piximal: Parallel Approach for Processing XML

4. **Related Work**

5. **Conclusions and Future Work**
Prevalence of Parallel Machines

- All new high end and mid range CPUs for desktop- and laptop-class computers have at least two cores.
- The future of AMD and Intel performance lies in increases in the number of cores.
- Despite extant SMP machines, many classes of software applications remain single threaded:
  - Multi-threaded programming considered ‘hard’
**XML and Multi-Core**

- Most string parsing techniques rely on a serial scanning process

- **Challenge**: Existing (singly-threaded) XML parsers are already very efficient (Zhang et al 2006)
1. **Introduction and Motivation**
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions

2. **SOAP and XML Benchmarks**
   - SOAPBench
   - XMLBench

3. **Parallel XML**
   - Investigating System Cache Effects
   - Piximal: Parallel Approach for Processing XML

4. **Related Work**

5. **Conclusions and Future Work**
We present the design and implementation of a comprehensive 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 use case scenarios.

We present an analysis of pre-fetching and piped implementation techniques that aim to offset disk I/O costs while processing large-scale XML datasets on multi-core CPU architectures.
We propose techniques to modify the lexical analysis phase for processing large-scale XML datasets to leverage opportunities for parallelism. (Piximal)

We present an analysis of the scalability that can be achieved with our proposed 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.
Publications

- “A Benchmark Suite for SOAP-based Communication in Grid Web Services,” in *The Proceedings of Supercomputing 2005*
- “Benchmarking XML Processors for Applications in Grid Web Services,” in *The Proceedings of Supercomputing 2006*
- “Approaching a Parallelized XML Parser Optimized for Multi-Core Processors,” in *The Proceedings of SOCP 2007*, workshop held in conjunction with HPDC 2007
- “Parallel Processing of Large-Scale XML-Based Application Documents on Multi-core Architectures with PiXiMaL,” in *The Proceedings e-Science 2008*
- “Performance Enhancement with Speculative Execution Based Parallelism for Processing Large-scale XML-based Application Data,” to appear in *The Proceedings of HPDC 2009*
In this thesis we present a comprehensive benchmark suite that facilitates the study of the strengths and weaknesses of XML and SOAP toolkits for a wide range of use case scenarios.

We propose a parallel processing model for some application-based large-scale XML datasets that can effectively leverage opportunities for parallelism in emerging multi-core CPU architectures.
1 Introduction and Motivation
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions

2 SOAP and XML Benchmarks
   - SOAPBench
   - XMLBench

3 Parallel XML
   - Investigating System Cache Effects
   - Piximal: Parallel Approach for Processing XML

4 Related Work

5 Conclusions and Future Work
SOAP Benchmark Suite

- Defines a set of operations to implement within a SOAP toolkit
- Tests both serialization and deserialization of a variety of data structures over a range of input sizes
  - Simple types: integers, strings, and floats
  - Base64 encoded data
  - Complex types: event streams, mesh interface objects
Outline

1. Introduction and Motivation
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions

2. SOAP and XML Benchmarks
   - SOAPBench
   - XMLBench

3. Parallel XML
   - Investigating System Cache Effects
   - Piximal: Parallel Approach for Processing XML

4. Related Work

5. Conclusions and Future Work
XML Benchmark Suite

1. A chosen set of XML documents
   - Low level probes
   - Application-based benchmarks

2. A driver application for each XML processor
   - Runs the parser on the input, but does not act on the data
     - Eliminates application-level performance differences
     - One for each interface style (SAX/DOM)
OUTLINE

1. **Introduction and Motivation**
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions

2. **SOAP and XML Benchmarks**
   - SOAPBench
   - XMLBench

3. **Parallel XML**
   - Investigating System Cache Effects
   - Piximal: Parallel Approach for Processing XML

4. **Related Work**

5. **Conclusions and Future Work**
**Readahead/Runahead**

- Explore OS level caching effects
- Offload disk input to another thread/core
- Improved the performance of an existing high performance parser by using a separate thread to read the input into cache
1. **Introduction and Motivation**
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions

2. **SOAP and XML Benchmarks**
   - SOAPBench
   - XMLBench

3. **Parallel XML**
   - Investigating System Cache Effects
   - **Piximal**: Parallel Approach for Processing XML

4. **Related Work**

5. **Conclusions and Future Work**
Token-Scanning With a DFA

- DFA-based table-driven scanning is both popular and fast
  - (or at least performance-competitive with other techniques)
- Input is read *sequentially* from start to finish
  - Each character is used to transition over states in a DFA
  - Transition may have associated actions
    - Supports languages that are not "regular"
- Commonly used in high performance XML parsers, such as TDX (C) and Piccolo (Java)
  - Amenable to SAX parsing
  - **Piximal**-DFA uses this approach
DFA Used in Piximal-DFA
**Parallel Scanning With a DFA?**

- DFA-based scanning $\Rightarrow$ sequential operation

- Desire: run multiple, concurrent DFAs throughout the input
  - Generally not possible because the start state would be unknown
**Overcoming Sequentiality With an NFA**

- **Problem:** start state is unknown

- **Solution:** assume every possible state is a start state
  - Construct an NFA from the DFA used in *Piximal*-DFA
  - Such an NFA can be applied on any substring of the input

- **Piximal-NFA** is the parser that does all of this:
  - Partition input into segments
  - Run *Piximal*-DFA on the initial segment
  - Run NFA-based parsers on subsequent partition elements
  - Fix up transitions at partition boundaries and run queued actions
Piximal-NFA’s Parameters

- **split_percent:**
  - The portion of input to be dedicated to the first element of the partition, expressed as a percentage of the total input length

- **number_of_threads:**
  - The number of threads to use on a run
Preliminary Research Questions

- Is there enough memory bandwidth to allow multiple automata to concurrently feed each thread its input?

- Processing each character along several paths through the NFA is costly: how does this work scale with the size of the initial DFA?
  
  (E-science 2008)

- Does the overhead of queuing the NFA actions cost an acceptable amount compared with the cost of DFA-parsing the first partition element?
  
  (HPDC 2009)
Models the work of partitioning the input the way Piximal-NFA does
- File I/O is via `mmap(2)`
- A thread is created for each partition element which accumulates each character
- A variety of `split_percents` and `number_of_thread` are chosen
  - Total time to read a large input a fixed number of times is measured
  - Input file is `SwissProt.xml`, which is 109 MB in size
Run several machines, each from a homogeneous class running 64-bit versions of Linux

- **2× uniprocessor**: 3.2 Ghz Intel Xeon (uniprocessor), 4 GB RAM, Linux kernel 2.6.15, GNU Lib C 2.3.6, GCC 4.0.3
- **2× dual core**: 2.66 Ghz Intel Xeon 5150 (dual core) CPUs, 8 GB RAM, Linux kernel 2.6.18, GNU Lib C 2.3.6, GCC 4.1.2
- **2× quad core**: 2.33 Ghz Intel Xeon E5354 (quad-core) CPUs, 8 GB RAM, Linux kernel 2.6.18, GNU Lib C 2.3.6, GCC 4.1.2

- 4 nodes used from the 2× UP cluster, 10 from each of the other two

- Results for each class are averaged across all runs
Bandwidth is Not a Bottleneck Up to 6 Cores

![Graph showing speedup with varying number of cores](image)

- # cores (split %)
  - 2 (52%)
  - 4 (28%)
  - 8 (12%)

Number of threads: 2, 3, 4, 5, 6, 7, 8

Speedup: 1.0, 1.5, 2.0, 2.5, 3.0, 3.5

- 2 (52%)
- 4 (28%)
- 8 (12%)

The graph illustrates the speedup with the number of threads and the number of cores. It shows that bandwidth is not a bottleneck up to 6 cores.
CONCLUSIONS FROM MEMORY BANDWIDTH TESTS

- Even when doing very little per-character processing, performance gains possible by adding threads
- Returns do diminish rapidly
- More cores lead to smoother results
STATE SCALABILITY TEST

- Models the additional work done by the NFA threads by following multiple execution paths through the table
- Each NFA thread now must remember the state and calculate the next state for each character and for each start state
  - The DFA need only remember and calculate one state per input character
- Does not model the memory used, actions stored, or garbage state elimination
- Goal: to find a balance point for DFA size
  - + increased complexity of the recognized language
  - − more work for the NFA to do, more space required for table
**2× QC – Best Speedup for DFA Sizes**

<table>
<thead>
<tr>
<th>DFA state size (w/split %)</th>
<th>Number of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 states, 12 %</td>
<td>2</td>
</tr>
<tr>
<td>4 states, 16 %</td>
<td>3</td>
</tr>
<tr>
<td>6 states, 20 %</td>
<td>4</td>
</tr>
<tr>
<td>8 states, 36 %</td>
<td>5</td>
</tr>
<tr>
<td>10 states, 40 %</td>
<td>6</td>
</tr>
<tr>
<td>12 states, 40 %</td>
<td>7</td>
</tr>
</tbody>
</table>

- 2× QC – Best Speedup for DFA Sizes
- 2 states, 12 %
- 4 states, 16 %
- 6 states, 20 %
- 8 states, 36 %
- 10 states, 40 %
- 12 states, 40 %
The extra work of pushing characters through the multiple execution paths of the NFA is not in itself a limiting factor.

There is a "sweet spot" for DFA size: around 6-7 states which allows for the greatest language complexity and the best scalability.

This is a crossover point where the O(N) extra NFA work overcomes the O(1) work of simply reading the input.
Serial NFA Tests

- Test hypothesis: the extra work required by using an NFA is offset by dividing processing work across multiple threads
- Run each automaton-parser sequentially and independently
- Divide the work as usual, with a range of split_percents and number_of_threads
- Time each component independently
- Completely parses the input, generating the correct sequence of SAX events
- The maximum time for all components to complete (plus fix up time) represents an upper bound on the time Piximal-NFA would take with components running concurrently
**Differences From Previous Tests**

- Entirely sequential (no concurrency)
- Full XML parsing takes place
- Input file is different
  - “Interop” test from SOAPBench and XMLBench
  - SOAP-encoded arrays of various data types: integers, strings, and MIOs
  - Array size is scaled between 10 and 50,000 elements for each type
Modest Speedup Scalability for 10,000 Integers
**Split Percent** CRITICAL FOR SPEEDUP FOR 10,000 INTEGERS

- **Max Speedup**
- **Mean Speedup**
- **Min Speedup**

![Graph showing speedup vs split percent](image)
Inconsistent Speedup Over a Range of Array Lengths
Characters in 10,000 Integers in a Range of States
Conclusions From Integer Results

- Speedup is possible in this case
- Choice of split point is critical for achieving any speedup at all
- Characters in content sections account for roughly 60% of the input characters
- Input is 117 KB in length
- Consists mainly of
  \[
  \ldots\langle i \rangle 1234 \langle /i \rangle \langle i \rangle 1235 \langle /i \rangle \langle i \rangle 1236 \langle /i \rangle \ldots
  \]
Investigating System Cache Effects

Piximal: Parallel Approach for Processing XML

Memory Bandwidth Test

State Scalability Test

Serial NFA Tests

SPEEDUP IMPROVES WITH $\text{Thread Count}$ FOR 10,000 STRINGS

[Graph showing speedup improvements with thread count]

- Max Speedup
- Mean Speedup
- Min Speedup

Potential Speedup

Thread Count

2 3 4 5 6 7 8

0.0 0.5 1.0 1.5 2.0 2.5 3.0 3.5
**Split Percent** Less Critical for 10,000 Strings

![Graph showing Potential Speedup vs. Split Percent](image)

- **Max Speedup**
- **Mean Speedup**
- **Min Speedup**
Consistent Speedup Over a Range of Input Sizes

![Graph showing consistent speedup over a range of input sizes. The graph plots potential speedup against array size. The x-axis represents array size in increments of 10,000, starting from 0 to 50,000. The y-axis represents speedup in increments of 0.5, starting from 0 to 3.5. There are four lines on the graph:
- Max Speedup: Dashed red line
- Mean Speedup: Dotted blue line
- Min Speedup: Dot-dashed green line
- Potential Speedup: Solid red line

The graph shows that the speedup remains consistent across different array sizes, with the max speedup, mean speedup, and min speedup lines remaining relatively flat and close to each other. The potential speedup line shows a slight decrease as the array size increases, indicating slight decreases in potential speedup with increasing input size.](Image)
CHARACTERS IN 10,000 STRINGS ARE MAINLY IN CONTENT

Frequency

DFA State
Conclusions from String Results

- This sort of input is much more amenable to this approach
  - In maximum potential speedup achieved
  - In number of cases where speedup is > 1
- Split point is much less important here
- Characters in content sections account for roughly 99% of the input characters
- Input is 1.4 MB in size (though similar results are seen in inputs that are 117 KB)
- Consists mainly of ...<i>String content for the array element number 0. This is long to test the hypothesis that longer content sections are better for the NFA.</i>...
Conclusions from Serial NFA Test

- Shape of the input strongly determines the efficacy of the Piximal approach
  - MIO has similar state usage and mix of content and tags as the integer and Piximal has a similar performance profile there
  - Piximal works well on inputs with longer content sections punctuated by short tags
- Starting in a content section helps because the ‘<’ character eliminates a large number of execution paths through the NFA
  - If ‘>’ could be treated similarly by the parser, starting in a tag would be less harmful
PXML: A Better Language for Piximal

Goal: Improve Piximal performance

- Reduce DFA size
- Increase the number of paths that lead to contradictions

Restrict XML (as supported in Piximal) in the following ways:

- **Disallow attributes:** Transform into nested elements
- **Disallow whitespace in tags:** Without attributes, these are completely unnecessary
- **Disallow ‘>’ in content sections:** Unnecessary in any case
- **Ignore distinction between characters that start a name and the rest**
DFA For Piximal-PXML

0

'<'

Whitespace

1

name character

2

name character

3

'>'

4

'/'

name character

'<'

'>'

character data
Related Work in High Performance XML Processing

- Look-aside buffers/String caching (gsoap, XPP)
- Trie data structure with schema-specific parser (Chiu et al 02, Engelen 04)
- One pass table-driven recursive descent parser (Zhang et al 2006)
- Pre-scan and schedule parser (Lu et al 2006)
- Parallelized scanner, scheduled post-parser (Pan et al 2007)
CONCLUSIONS

- Existing XML and SOAP toolkits make limited use of multiple cores
- Scientific applications strain existing XML infrastructure
- Pre-caching mechanisms can improve performance of existing parsers
- A parallel parsing approach is necessary to achieve increased parser performance as document sizes grow
- 5-6 states is a good size for a Piximal DFA
- Restricting XML slightly should provide better performance at a low semantic cost
- Piximal’s applicability is dependent on the characteristics of the input file
LIMITATIONS

- PThread overhead during concurrent runs
- Restrictions on XML format
  - Namespaces
  - CDATA
  - Unicode
  - Processing Instructions
  - Validation
- Optimal splitting algorithm unknown
SUMMARY

1. INTRODUCTION AND MOTIVATION
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions

2. SOAP AND XML BENCHMARKS
   - SOAPBench
   - XMLBench

3. PARALLEL XML
   - Investigating System Cache Effects
   - Piximal: Parallel Approach for Processing XML

4. RELATED WORK

5. CONCLUSIONS AND FUTURE WORK
Thank you for your time.
Questions?
The following slides are additional and not part of the presentation.
RE-RUN BENCHMARKS, NORMALIZE ANALYSIS AND PLOTTING

SOAPBench and XMLBench results should be re-run. Plots should be rebuilt to match the rest of the figures.

- XMLBench is available for researchers to download and use
- SOAPBench is available, but cannot support all the tested SOAP toolkits due to their proprietary nature

ANALYZE A BROADER RANGE OF DATA FROM THE SERIAL NFA TEST

The serial NFA tests show a small portion of the data collected in that test. There is a wealth of information to uncover about the efficacy of this approach in the data.

- Data and analysis is available in our repository and will be posted to a web site shortly
PROPOSED WORK CONTINUED

INVESTIGATE MEMORY ALLOCATION ISSUES

Heap contention is a well known problem for applications with concurrent memory allocations. We plan to investigate the effect of a variety of allocators on Piximal. During Piximal development, we encountered some issues involving the performance of malloc once a thread (even a thread with an empty start_routine) was created. We plan to investigate and report on this in detail.

- Have initial results (HPDC 2009), potential for broader investigation remains
Define characteristics of a restricted subset of XML documents: “PXML”

Based on the above results, we can design a language which works best with Piximal-NFA. Potential targets include eliminating ‘>’ from content sections, removing CDATA sections, disallowing extra whitespace in tags, and perhaps eliminating attributes altogether.

- Briefly described in Chapter 5, Section 4 of the thesis document
- A formal grammar was not considered necessary for the scope of the thesis
Overcoming Sequentiality With an NFA

- Problem: start state is unknown

- Solution: assume every possible state is a start state
  - Construct an NFA from the DFA used in Piximal-DFA
    1. Mark every state as a start state
    2. Remove all the garbage state and all transitions to it
    3. Create an queue for each start state to store actions that should be performed

  - Such an NFA can be applied on any substring of the input

Piximal-NFA is the parser that does all of this:

- Partition input into segments
- Run Piximal-DFA on the initial segment
- Run NFA-based parsers on subsequent partition elements
- Fix up transitions at partition boundaries and run queued actions
Piximal-DFA Implementation Details

- `mmap(2)` s input file to save memory
- Uses `{length, pointer}` string representation
  - Strings (for tag names, attribute values) point into the mapped memory
  - All the way through the SAX-style event interface
- DFA is encoded as two tables
  - Table of ``next'' state numbers indexed by state number and input character
  - Table of boolean ``action required'' indicators indexed by ``current'' state and ``next'' state
    - Action required $\implies$ a function is called to decode and execute the required action
  - DFA table is generated at compile time using a separate generator program
Speedup for the Readahead Parser Relative to Architecture

- CMP
- UP
- SMP

(Input Resides in Filesystem Cache)
**Speedup for the Runahead Parser Relative to Architecture**

- CMP
- SMP
- UP

(Input Resides in Filesystem Cache)
Speedup for the CMP Architecture Relative to Parser Type

Relative Speedup

Run Number

(Input Flushed from Filesystem Cache)
**Benchmark Probes**

- **Overhead test**
  - Minimal XML document
    - (header plus one self-closing element)
- **Buffering**
  - Repeated use of xsi:type attributes
- **Namespace management**
  - Gratuitous use of xmlns attributes
- **SOAP payloads**
  - “Interop” test: arrays of integer, string, double, MIO, event objects
Benchmarks for Selected Applications

- Ptolemy Workflow documents (which Kepler uses)
- Genetic data files
  - (Large) files from the International HapMap Project
- Molecular data
- Mesh interface objects, event streams (WSMG)
- WS-Security documents
OVERHEAD OF EACH Parser

All Parsers, Overhead Test

<table>
<thead>
<tr>
<th>Parser</th>
<th>Parse time over 20 runs (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>expat</td>
<td>0</td>
</tr>
<tr>
<td>gsoap</td>
<td>1</td>
</tr>
<tr>
<td>libxml2−dom</td>
<td>2</td>
</tr>
<tr>
<td>libxml2−sax</td>
<td>3</td>
</tr>
<tr>
<td>mono−dom</td>
<td>4</td>
</tr>
<tr>
<td>mono−reader</td>
<td>5</td>
</tr>
<tr>
<td>piccolo</td>
<td>6</td>
</tr>
<tr>
<td>qt4−sax</td>
<td>7</td>
</tr>
<tr>
<td>xerces−c−dom</td>
<td>81</td>
</tr>
<tr>
<td>xerces−c−sax</td>
<td>59</td>
</tr>
<tr>
<td>xerces−j−dom</td>
<td>48</td>
</tr>
<tr>
<td>xerces−j−sax</td>
<td>38</td>
</tr>
<tr>
<td>xpp3</td>
<td>28</td>
</tr>
</tbody>
</table>
Performance of C and C++-based Parsers

C/C++ Parsers, Application-level Inputs

Parser

- expat
- gsoap
- libxml2-dom
- libxml2-sax
- xerces-c-dom
- xerces-c-sax

Parse time over 20 runs (ms)

- hapmap_1797SNPs.xml
- molecule_1kzk.pretty.xml
- workflow_Atype.xml
- workflow_PIW.xml
C Parser Performance Over SOAP Payloads

Parsing Performance for SOAP Payloads of int Arrays

- expat
- gsoap
- libxml2-dom
- libxml2-sax
- qt4-sax
- xerces-c-dom
- xerces-c-sax

Number of Elements in the Array

Parse Time for 20 runs (ms)
Performance of Java-based Parsers

Java Parsers, Application-level Inputs

Parse time over 20 runs (ms)

- piccolo
- xerces-j-dom
- xerces-j-sax
- xpp3

Files:
- hapmap_1797SNPs.xml
- molecule_1kzk.pretty.xml
- workflow_Atype.xml
- workflow_PIW.xml

Parser

Parse time over 20 runs (ms)
XMLBench Conclusions

- Low overhead \(\implies\) gSOAP and Expat, XPP3
- gSOAP performs well with namespaces due to look-aside buffers
- Piccolo and XPP3 have comparable performance in Java
2× UP Overall Results
2x DC Overall Results
2× QC Overall Results
2× DC Speedup For Best split_percent
2× QC Speedup for Best split_percent
CONCLUSIONS FROM SPEEDUP CROSS SECTIONS

- Reaffirmation that speedup is possible
- Returns diminish for these machines at around 6 threads
- Overall, access to main memory is not an immediate bottleneck

Putting the results from the best *split_percent* for each architecture...
2× UP Overall Raw Results

- Number of DFA states
  - 5
  - 10
  - 15

- Number of threads
  - 5
  - 10
  - 15

- Time (s)
  - 20
  - 25
  - 30
  - 35
  - 40
2x DC Overall Results – Best Times

![Graph showing the relationship between the number of DFA states and the number of threads with time (s) as the z-axis. The graph indicates a trend where increasing the number of threads generally decreases the time (s) for processing, with a peak at 15 threads for states 5, 10, and 15.]
2x QC Overall Results - Best Times

![3D Graph of Time vs Number of DFA states vs Number of threads]

- Time (s)
  - 10
  - 20
  - 30
  - 40

- Number of DFA states
  - 5
  - 10
  - 15

- Number of threads
  - 5
  - 10
  - 15

The graph illustrates the relationship between the number of DFA states, the number of threads, and the time taken to process the data. The results show that increasing the number of threads generally leads to a decrease in processing time, especially when the number of DFA states is higher.
Conclusions from State Scalability Overall Results

- Two major conclusions:
  - The speedup on the 2× quad-core machines appears stable as the number of threads increases.
  - There is a significant steepening when the DFA has 6-7 states.

- Performance reaches its max when the number of threads match the number of processing cores available:
  - Each new thread adds substantial extra work compared with the memory bandwidth test.

- Plotting speedup for certain split_percents.
XML Performance Limitations

- Compared to "legacy" formats
  - Text-based
    - Lacks any "header blocks" (e.g., TCP headers), so must scan every character to tokenize
    - Numeric types take more space and conversion time
  - Lacks indexing
    - Unable to quickly skip over fixed-length records
**Limitations of XML**

- Poor CPU and space efficiency when processing scientific data with mostly numeric data (Chiu et al 2002)
- Features such as nested namespace shortcuts don’t scale well with deep hierarchies
  - May be found in documents aggregating and nesting data from disparate sources
- Character stream oriented (not record oriented): initial parse inherently serial
- Still ultimately useful for sharing data divorced of its application
Introduce two parsers which extend the existing, high performance Piccolo parser (Head et al 2006)

- **Runahead**: opens two file descriptors for the input file
  - Start a thread that repeatedly calls `read()` on one of the file descriptors
  - Pass the other file descriptor to the existing Piccolo parser in the main thread

- **Readahead**: opens one file descriptor for the input file, and one pipe
  - Start a thread that reads from the file descriptor and writes to the pipe
  - Pass the pipe to the existing Piccolo parser in the main thread
Run each parser (Piccolo, Runahead, and Readahead) on a large (GB-scale) XML file
- Specifically, a protein sequence database file, psd7003.xml

No user code is run for any SAX event -- just the parser itself is tested

File cache is cleared between each run running a separate process that reads multiple gigabyte files

Each test is run 50 times for each parser

Hotspot is warmed by running the parser on another input file with identical content before timing begins
Two Environmental Conditions Tested

- **Architectures**
  - **UP:** Classic Uniprocessor P4-based machine (Dell workstation)
  - **SMP:** Classic Symmetrical MultiProcessing P4-based machine (has server-class I/O system) (IBM e-server)
  - **CMP:** Modern Chip MultiProcessing Core 2 Duo-based machine (Dell workstation)

- **System conditions**
  - **Cached:** The input file is read (hence loaded into the system file cache) before timing begins
  - **Uncached:** The input file is not read before timing begins (and flushed between each run)
Data Analysis

- Speedup for both of the proposed parsers is computed to compare across architectures.
- Baseline value is computing by averaging the times for each run of the unmodified Piccolo parser.
- Speedup for each run is computed by dividing the baseline by the time at each test point.
Speedup for the Runahead Parser Relative to Architecture

Run Number
(Input Flushed from Filesystem Cache)
On systems with available memory and an available processing core with fresh inputs, this approach can provide some performance wins.
## Comparison with Expat

<table>
<thead>
<tr>
<th>Input file</th>
<th>Expat</th>
<th>Piximal-dfa</th>
<th>Piximal-nfa</th>
</tr>
</thead>
<tbody>
<tr>
<td>psd-7003</td>
<td>15.51</td>
<td>17.47</td>
<td>14.18</td>
</tr>
</tbody>
</table>

**Table:** Parse time, in seconds per parse, of high performance parsers
Comparison Between GLibC and TCMalloc

Selected allocator
- GNU libc 2.7 malloc
- Google TCMalloc

Number of threads
Time (s)

2 3 4 5 6 7 8
25 26 27 28 29 30 31