Proposal Defense: Analysis and Optimization for Processing Grid-Scale XML Datasets

Michael R. Head

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

Friday, September 12, 2008

Outline

1. Introduction and Motivation
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions and Thesis Statement
2. Related Work
   - High Performance XML Processing Approaches
3. Work Completed
   - XML and SOAP Benchmarks
   - Investigating System Cache Effects
   - PIXIMAL: Parallel Approach for Processing XML
4. Proposed Work

Outline

1. Introduction and Motivation
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions and Thesis Statement
2. Related Work
   - High Performance XML Processing Approaches
3. Work Completed
   - XML and SOAP Benchmarks
   - Investigating System Cache Effects
   - PIXIMAL: Parallel Approach for Processing XML
4. Proposed Work

BINGHAMTON UNIVERSITY
State University of New York

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

XML Exclusive of SOAP

- General structured data format
- 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)

Benchmark Motivation

- Grid applications place a wide range of requirements on the communication substrate and data formats.
- Simple and straightforward implementations can have a severe performance impact.

XML Performance Limitations

- Compared to "legacy" formats
  - Text-based
    - Lacks any "header blocks" (ex. 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

Outline

BINGHAMTON UNIVERSITY
State University of New York

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”
  - Reinforced in the current curricula and by existing languages and tools

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]

Outline

BINGHAMTON UNIVERSITY
State University of New York

Contributions

- 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 representative 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.

Contributions Continued

- 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.

Thesis Statement

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 representative 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.

Outline

High Performance XML Processing Approaches

- 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]

Outline

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)
3. Published in Proceedings of SC'06 [Head et al 2006]

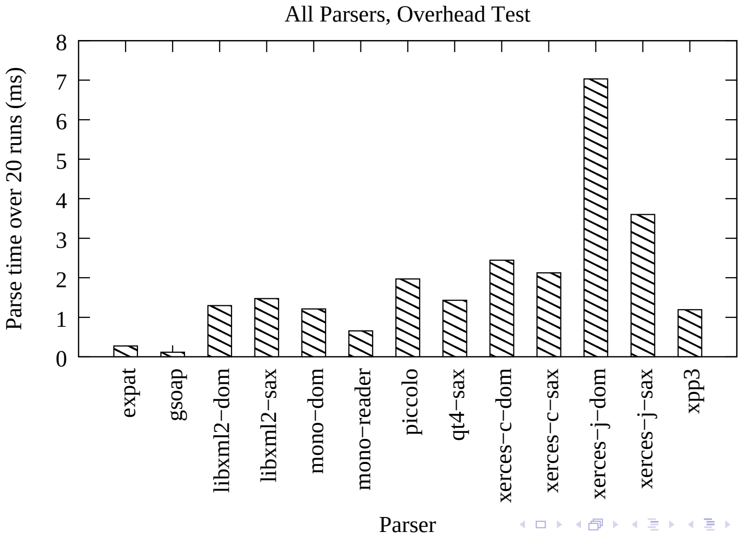
Benchmark Probes

- 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

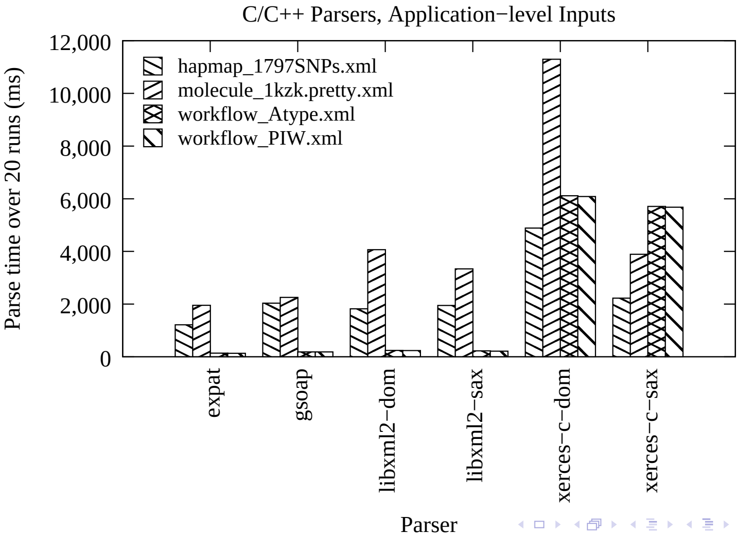
Application Benchmarks

- 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

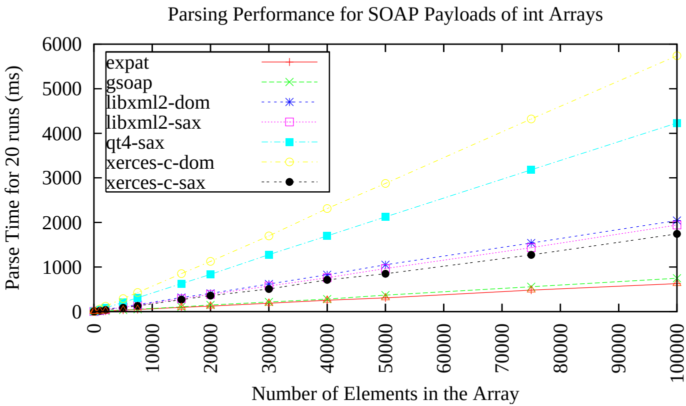
Overhead of Each Parser



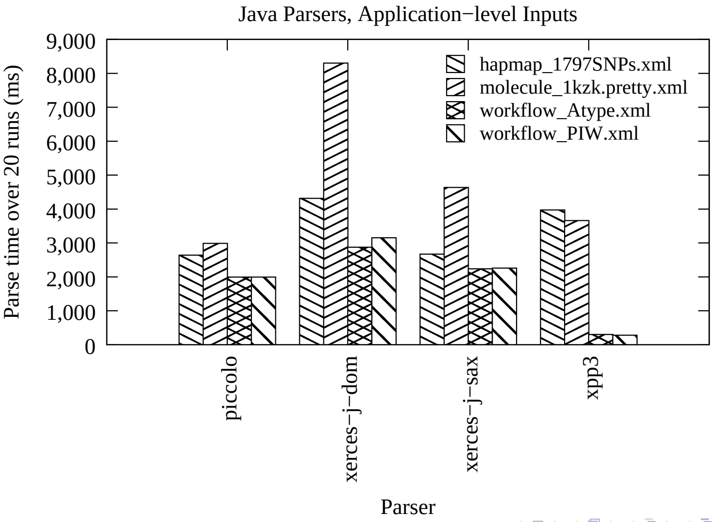
Performance of C and C++-based Parsers



C Parser Performance Over SOAP Payloads



Performance of Java-based Parsers



XMLBench Conclusions

- Low overhead $\Longrightarrow$ gSOAP and Expat, XPP3
- gSOAP performs well with namespaces due to look-aside buffers
- Piccolo and XPP3 have comparable performance in Java

Outline

1. Introduction and Motivation
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions and Thesis Statement
2. Related Work
   - High Performance XML Processing Approaches
3. Work Completed
   - XML and SOAP Benchmarks
   - Investigating System Cache Effects
   - PIXIMAL: Parallel Approach for Processing XML
4. Proposed Work

Readahead/Runahead

- Explore OS level caching effects
- Offload disk input to another thread/core
- Published in SOCP Workshop at HPDC [Head et al 2007]

Reading ahead

- 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

Test run

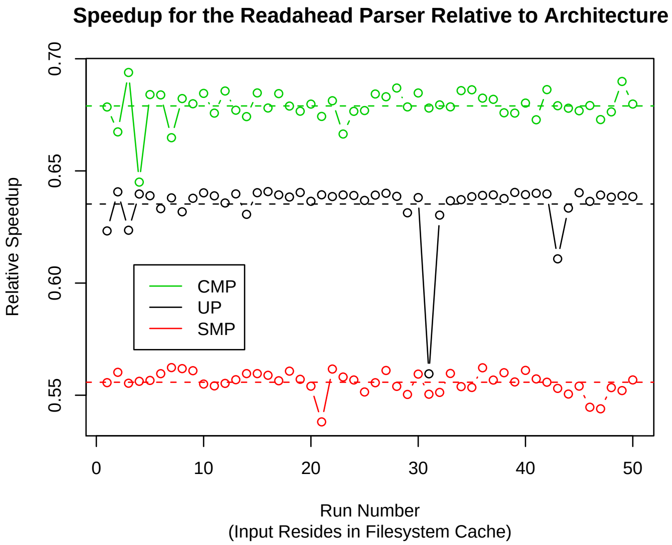
- Run each parser (**Piccolo**, **Runahead**, and **Readahead**) on a large (GB-scale) XML file
  - Specifically, a protein sequence database file, `psd7003.xml`
- No user code is run for any SAX event – just the parser itself is tested
- File cache is cleared between each run running a separate process 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

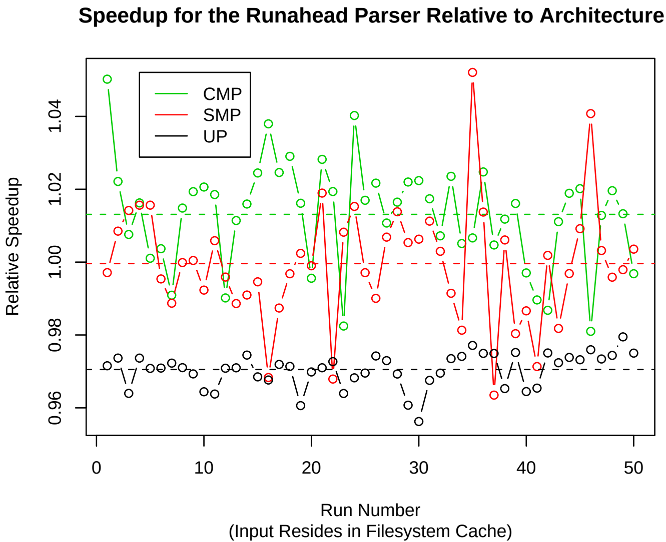
Two Environmental Conditions Tested

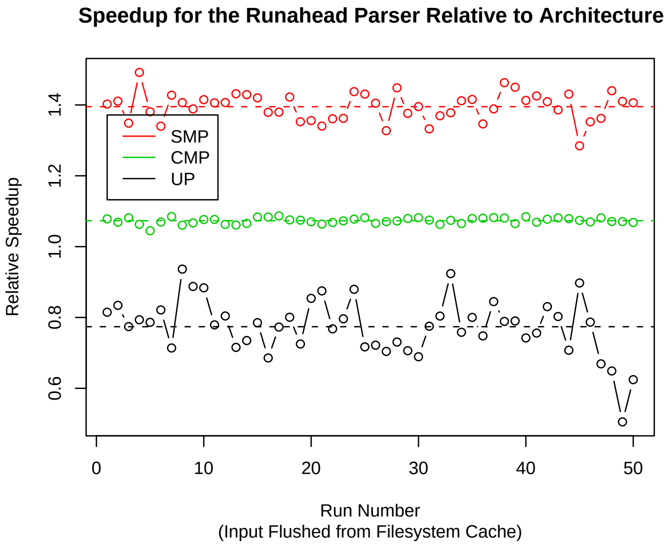
- Architectures
  - **UP:** Classic Uniprocessor P4-based machine (Dell workstation)
  - **SMP:** Classic Symmetrical MultiProcessing P4-based machine (has server-class I/O system) (IBM e-server)
  - **CMP:** Modern Chip MultiProcessing Core 2 Duo-based machine (Dell workstation)
- System conditions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

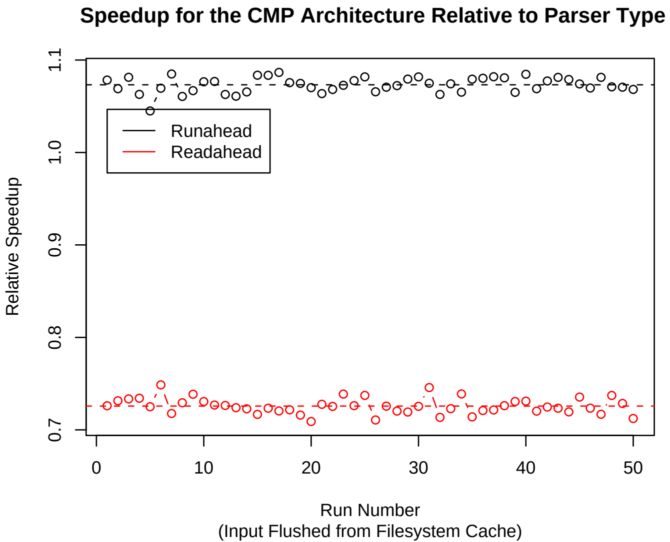
Data Analysis

- 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 Readahead Parser Relative to Architecture
Relative Speedup
0.55
0.60
0.65
0.70
CMP
UP
SMP
0
10
20
30
40
50
Run Number
(Input Resides in Filesystem Cache)


Speedup for the Runahead Parser Relative to Architecture
CMP
SMP
UP
Relative Speedup
0.96
0.98
1.00
1.02
1.04
0
10
20
30
40
50
Run Number
(Input Resides in Filesystem Cache)


Speedup for the Runahead Parser Relative to Architecture
Relative Speedup
0.6
0.8
1.0
1.2
1.4
SMP
CMP
UP
0
10
20
30
40
50
Run Number
(Input Flushed from Filesystem Cache)


Speedup for the CMP Architecture Relative to Parser Type
Relative Speedup
0.7
0.8
0.9
1.0
1.1
Runahead
Readahead
0
10
20
30
40
50
Run Number
(Input Flushed from Filesystem Cache)

Readahead Conclusions

- On systems with available memory and an available processing core with fresh inputs, this approach can provide some performance wins.

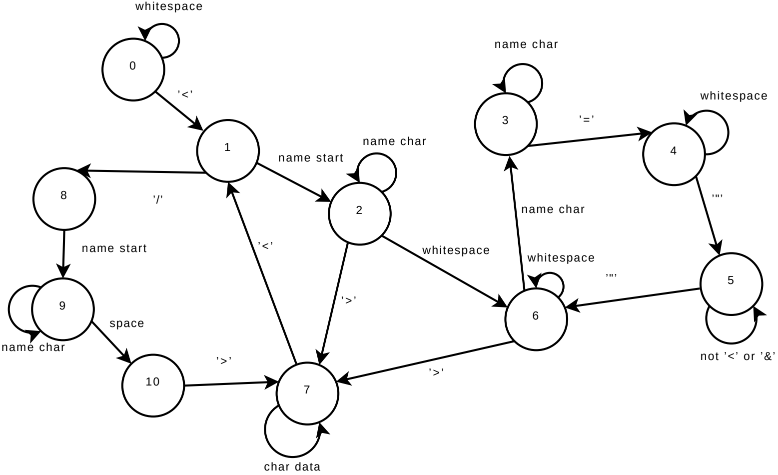
Outline

1. Introduction and Motivation
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
   - Contributions and Thesis Statement
2. Related Work
   - High Performance XML Processing Approaches
3. Work Completed
   - XML and SOAP Benchmarks
   - Investigating System Cache Effects
   - PIXIMAL: Parallel Approach for Processing XML
4. Proposed 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 $\Longrightarrow$ 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 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?
- Does the overhead of queuing the NFA actions cost a reasonable amount compared with the cost of DFA-parsing the first partition element?

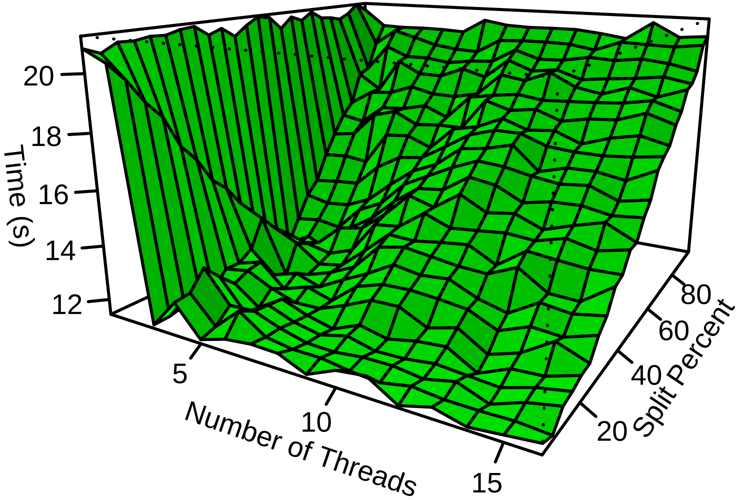
Memory Bandwidth Test

- 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

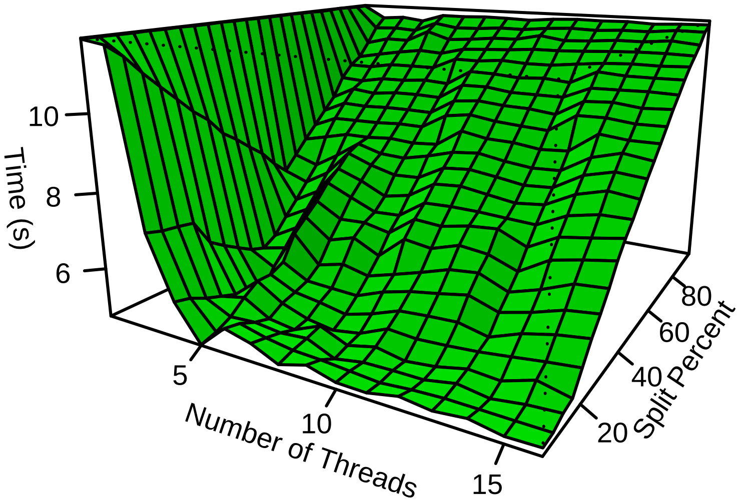
Memory Bandwidth Test – Experimental Setup

- 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

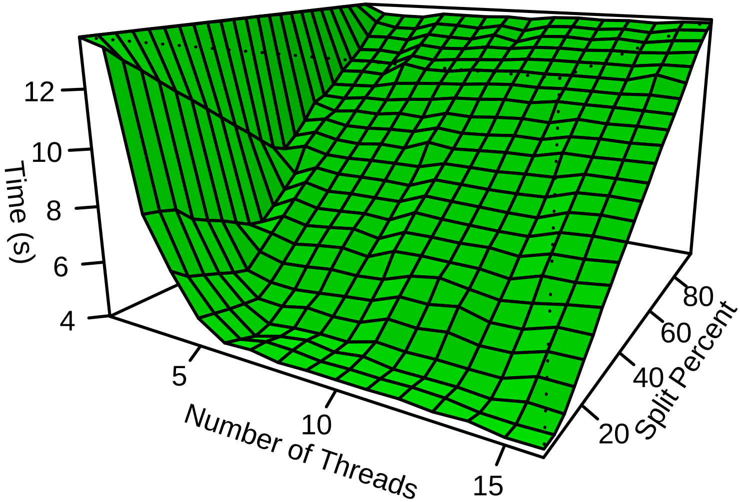
2× UP Overall Results



2× DC Overall Results



2× QC Overall Results

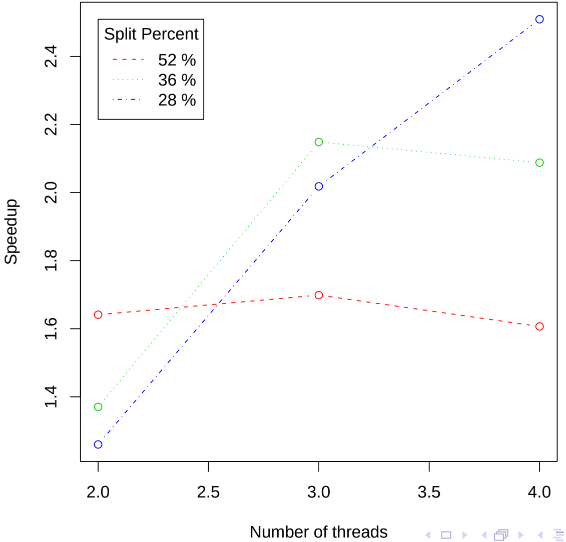


Conclusions From Overall Results

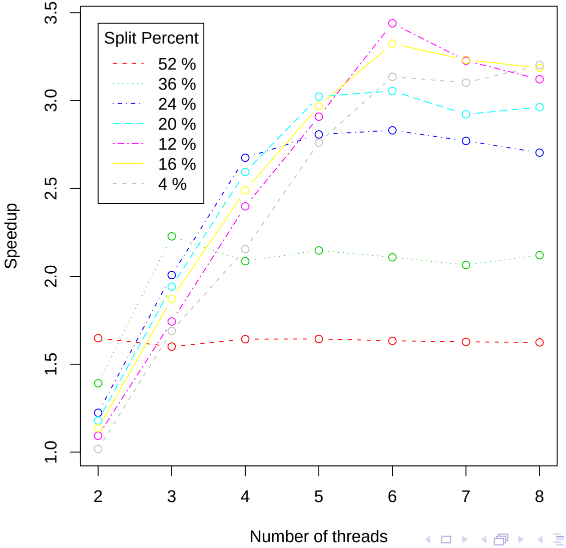
- Even when doing very little per-character processing, performance gains possible by adding threads
- Returns do diminish rapidly
- More cores lead to smoother results
- Adding “too many” threads does not hurt performance in this test

- How much gain in terms of speedup?
  - Calculated by $\frac{T_1}{T_P}$

2× DC Speedup For Best *split_percents*



2× QC Speedup For Best *split_percents*

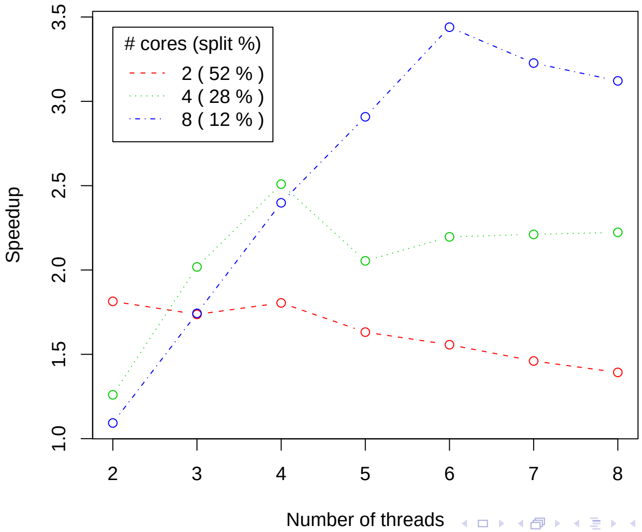


Conclusions From Speedup Cross Sections

- 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_percents* for each architecture...

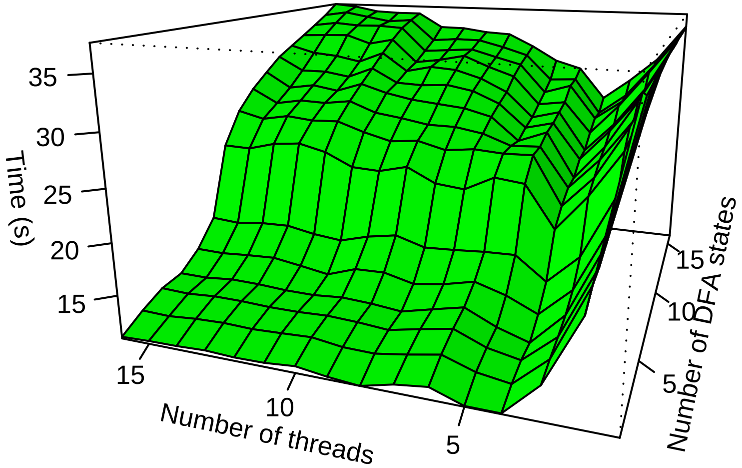
Comparison of Best *split_percent* Per Class



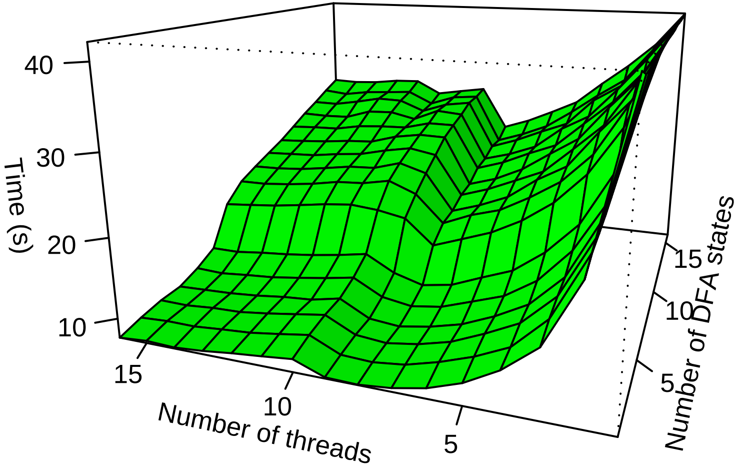
State Scalability Test

- 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

2× DC Overall Results – Best Times



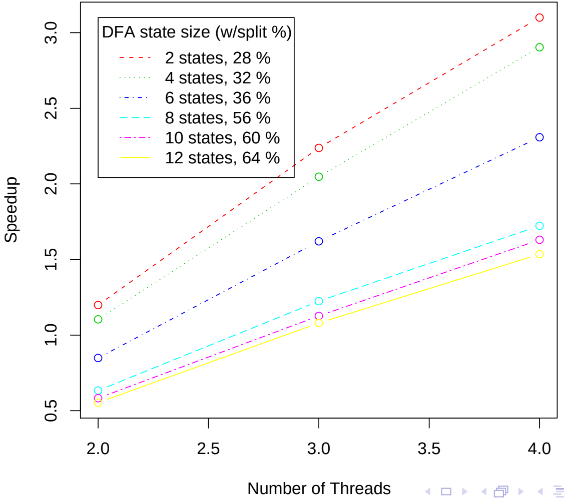
2× QC Overall Results – Best Times



Conclusions From State Scalability Overall Results

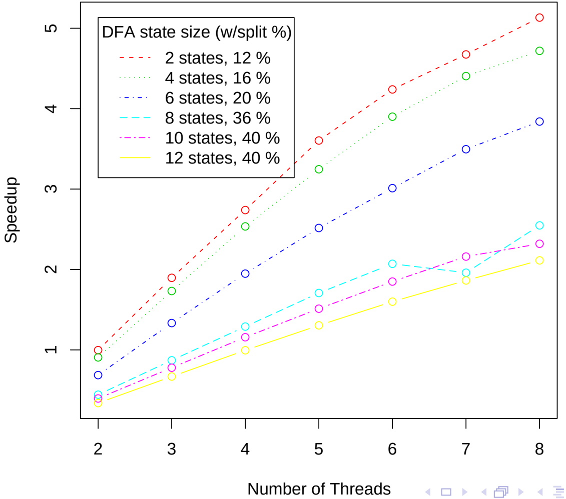
- Two major conclusions:
  - The speedup on the 2× quad-core machines appears stable as the number of threads increases
  - There is a significant steepening when the DFA has 6-7 states
- Performance reaches its max when the number of threads match the number of processing cores available
  - Each new thread adds substantial extra work compared with the memory bandwidth test
- Plotting speedup for certain *split_percents*

2× DC – Best Speedup for DFA Sizes



BINGHAMTON UNIVERSITY
State University of New York

2× QC – Best Speedup for DFA Sizes



Conclusions From State Scalability Test

- 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 the O(1) work of simply reading the input

Serial NFA Tests

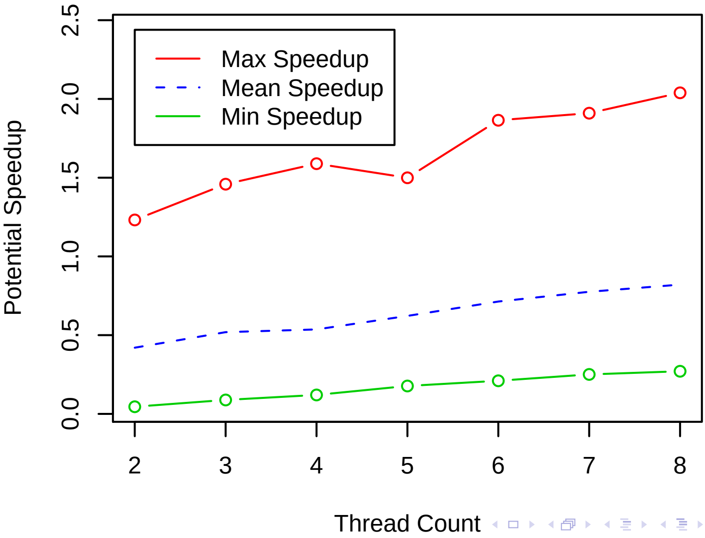
- Test hypothesis: the extra work required by using an NFA is offset by dividing processing work across multiple threads
- Run each automaton-parser sequentially and independently
- Divide the work as usual, with a range of *split_percents* and *number_of_threads*
- Time each component independently
- 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

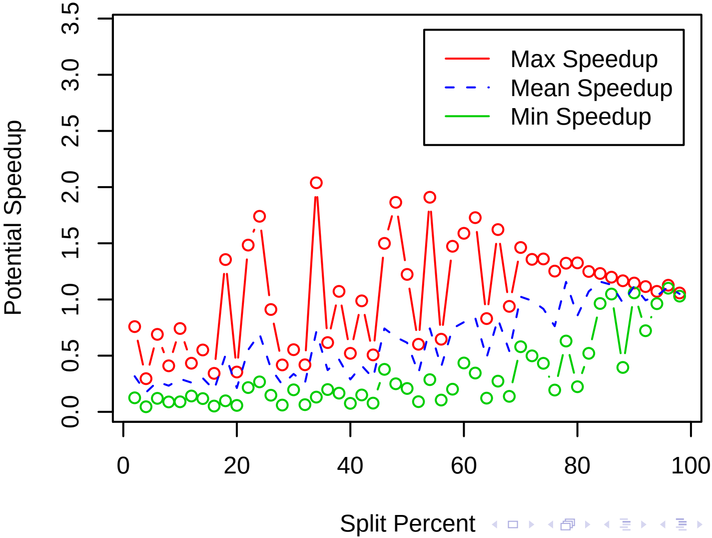
Differences From Previous Tests

- 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

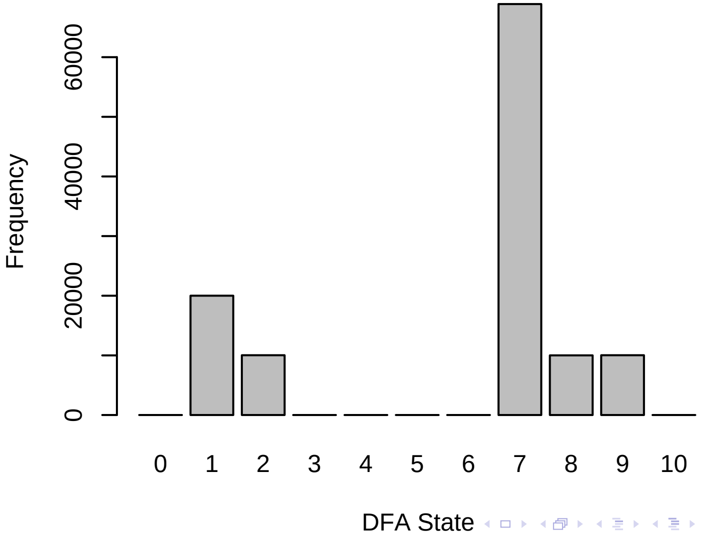
Serial NFA Test: 10,000 Integers By Thread Count



Serial NFA Test: 10,000 Integers By Split Percent



Serial NFA Test: 10,000 Integers State Histogram

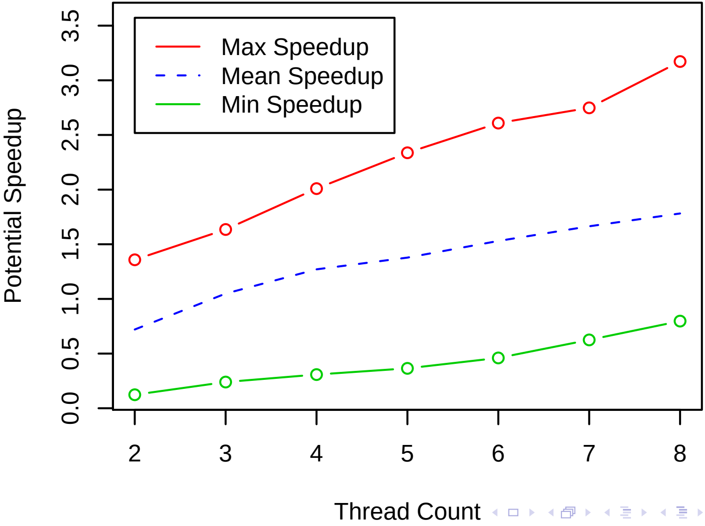


Conclusions From Integer Results

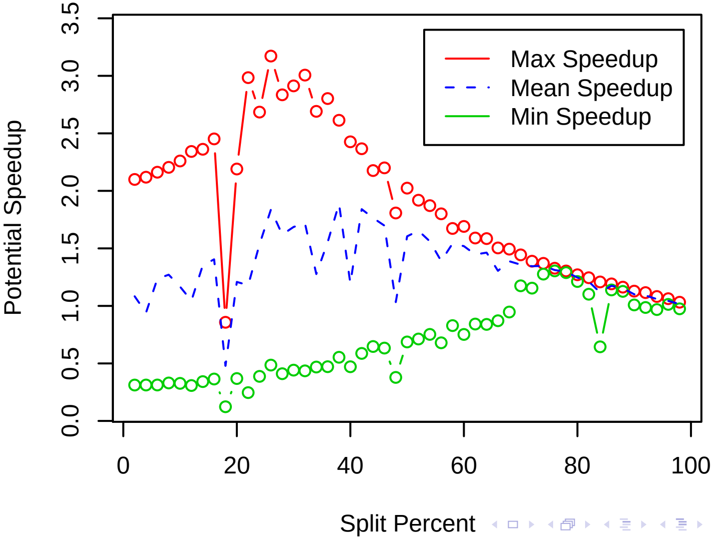
- Speedup is possible in this case
- Choice of split point is critical for achieving any speedup at all
- Characters in content sections account for roughly 60% of the input characters
- Input is 117 KB in length
- Consists mainly of

`...<i>1234</i><i>1235</i><i>1236</i>...`

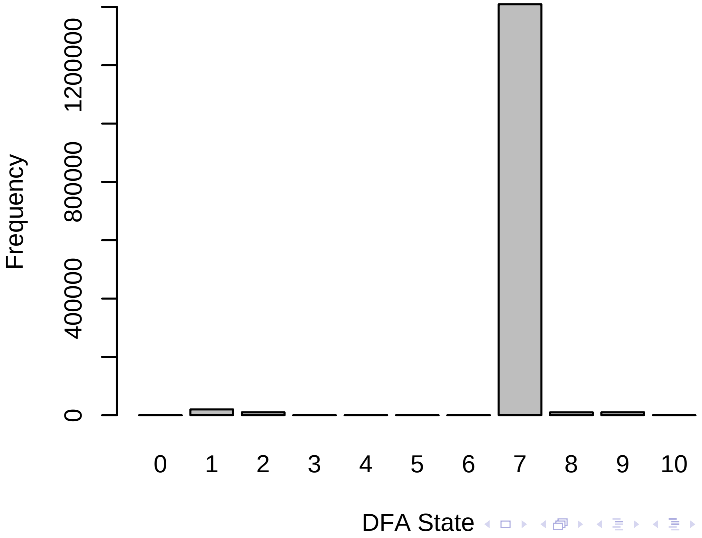
Serial NFA Test: 10,000 Strings By Thread Count



Serial NFA Test: 10,000 Strings By Split Percent



Serial NFA Test: 10,000 Strings State Histogram



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

Outline

BINGHAMTON UNIVERSITY
State University of New York

Proposed Work

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.

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.

Proposed Work Continued

Examine “pthread penalty” associated with glibc

During PIXIMAL development, we encountered some issues involving the 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.

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.

Proposed Work Continued

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.

Thank you for your time.

Questions?

Extra Slides

The following slides are additional and not part of the presentation.

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 tagnames, 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 $\Longrightarrow$ a function is called to decode and execute the required action
  - DFA table is generated at compile time using a separate generator program

BINGHAMTON UNIVERSITY
State University of New York