Parallel Processing of Large-Scale XML-Based Application Documents on Multi-core Architectures with PiXiMaL

Michael R. Head
Madhusudhan Govindaraju

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

December 7-12, 2008

Outline

1. Introduction and Motivation
   - XML and SOAP
   - Ubiquity of Multi-processing Capabilities
2. Related Work
   - High Performance XML Processing Approaches
3. Work Completed
   - PIXIMAL: Parallel Approach for Processing XML

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)

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]

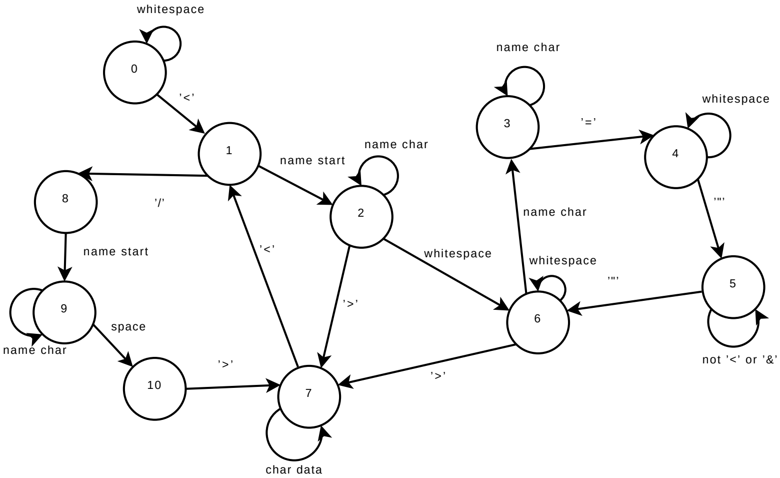
High Performance XML Processing Approaches

- 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]

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?

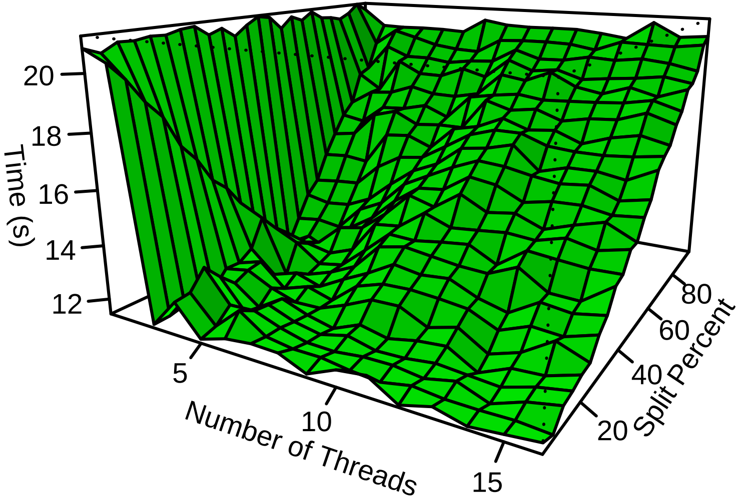
Memory Bandwidth Test

- 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

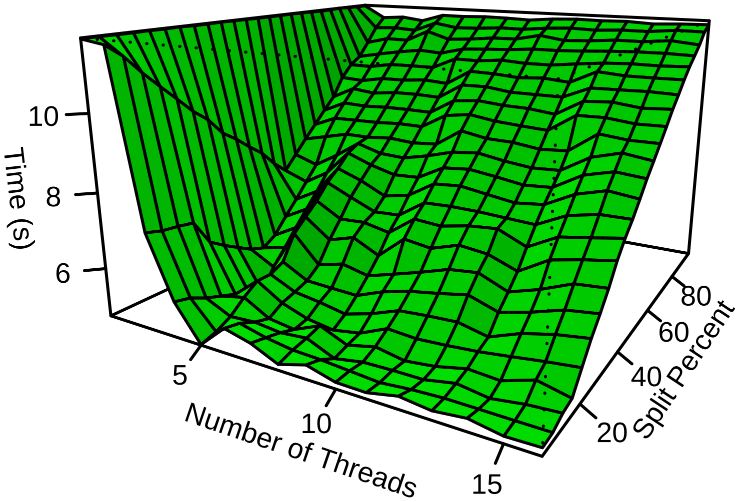
Memory Bandwidth Test – Experimental Setup

- 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

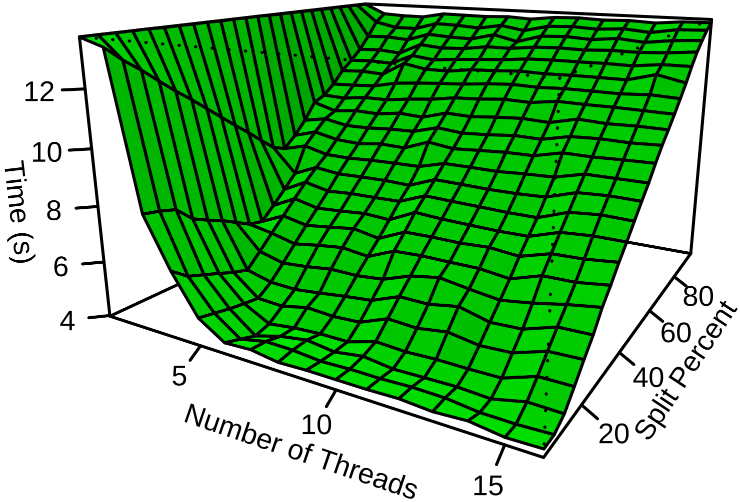
2× UP Overall Results



2× DC Overall Results



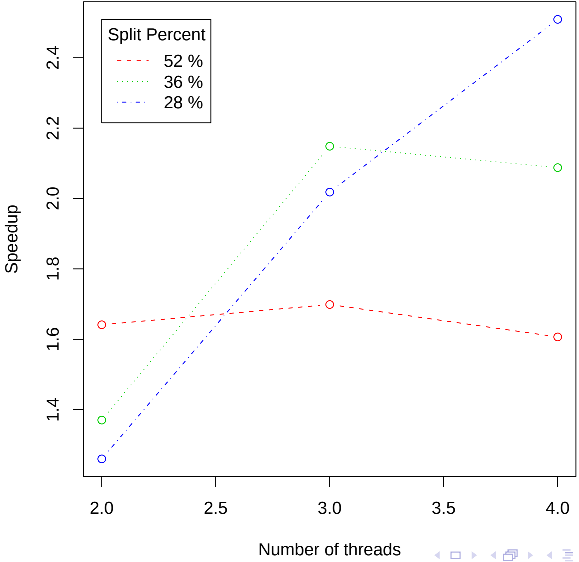
2× QC Overall Results



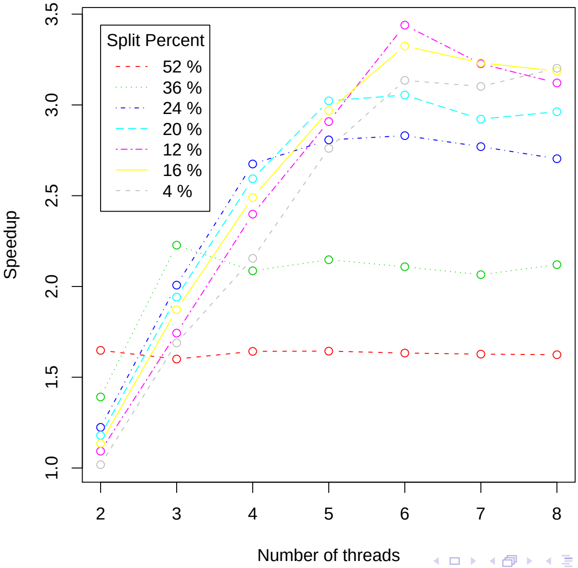
Conclusions From Overall Results

- 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*



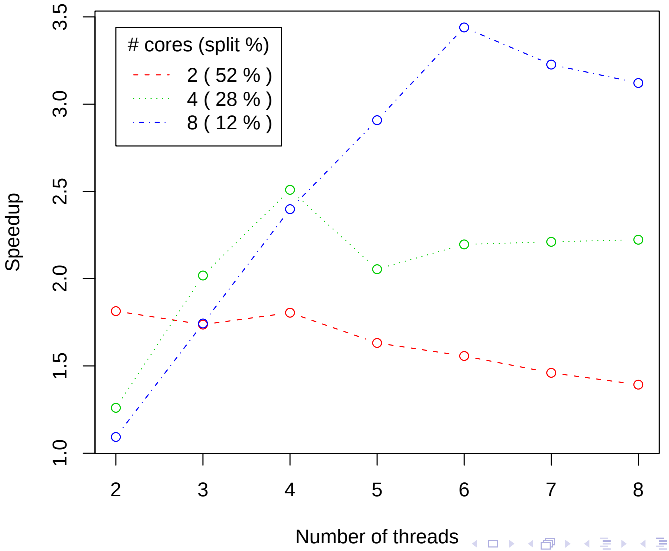
2× QC Speedup For Best *split_percents*



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...

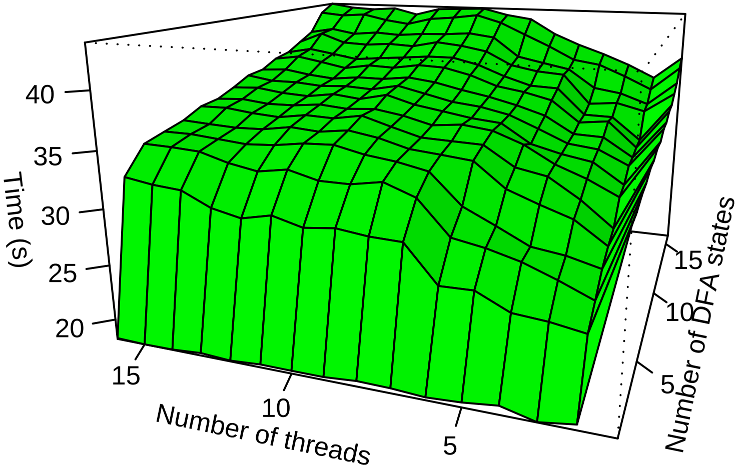
Comparison of Best *split_percent* Per Class



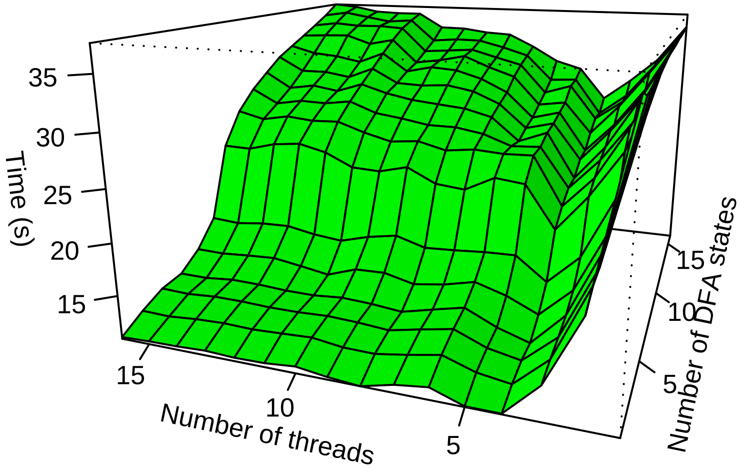
State Scalability Test

- 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

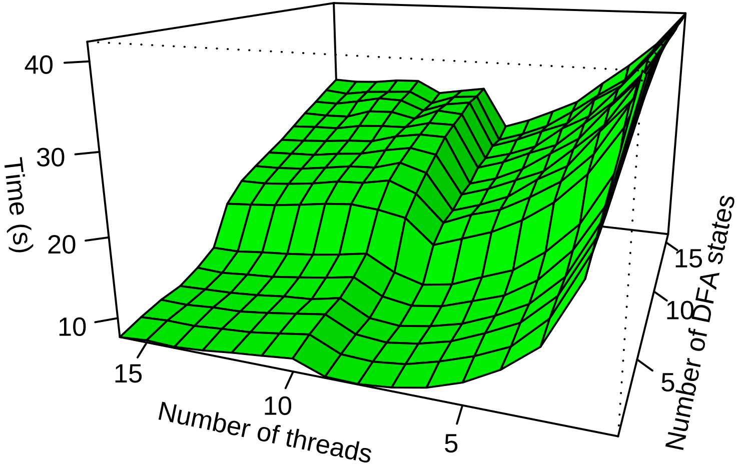
2× UP Overall Raw Results



$2\times$ DC Overall Results – Best Times



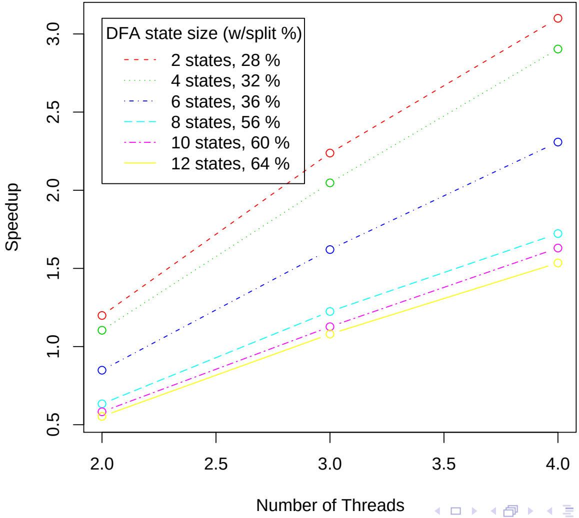
2× QC Overall Results – Best Times



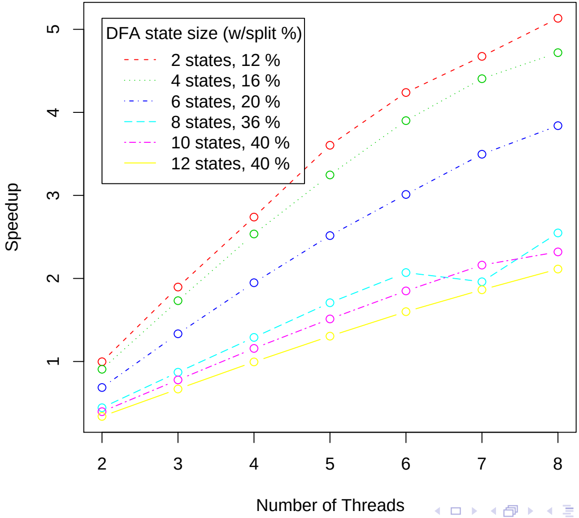
Conclusions From State Scalability Overall Results

- Two major conclusions:
  - The speedup on the $2\times$ 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*

$2\times$ DC – Best Speedup for DFA Sizes



2× QC – Best Speedup for DFA Sizes



BINGHAMTON UNIVERSITY
State University of New York

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

Thank you for your time.

Questions?

Extra Slides

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