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

Michael R. Head and Madhusudhan Govindaraju

Grid Computing Research Laboratory
Department of Computer Science
Binghamton University
http://www.cs.binghamton.edu/~{mike,mgovinda}

HPDC 2009
Thursday, June 11, 2009
OUTLINE

1 INTRODUCTION
   - Large XML Data
   - Ubiquity of Multi-processing Capabilities
   - SAX-based parsing

2 PARALLEL XML
   - PIXIMAL: Parallel Approach for Processing XML
   - Serial NFA Tests

3 CONCLUSIONS
   - Final Remarks
1 **INTRODUCTION**
   - Large XML Data
     - Ubiquity of Multi-processing Capabilities
     - SAX-based parsing

2 **PARALLEL XML**
   - PIXIMAL: Parallel Approach for Processing XML
   - Serial NFA Tests

3 **CONCLUSIONS**
   - Final Remarks
XML

- 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
Importance of High Performance XML Processors

- Becoming standard for many scientific datasets
  - HapMap - mapping genes
  - Protein Sequencing
  - NASA astronomical data
  - Many more instances
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
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)
1. **INTRODUCTION**
   - Large XML Data
   - Ubiquity of Multi-processing Capabilities
   - SAX-based parsing

2. **PARALLEL XML**
   - **Piximal**: Parallel Approach for Processing XML
   - Serial NFA Tests

3. **CONCLUSIONS**
   - Final Remarks
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.
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**
   - Large XML Data
   - Ubiquity of Multi-processing Capabilities
   - SAX-based parsing

2. **PARALLEL XML**
   - Piximal: Parallel Approach for Processing XML
   - Serial NFA Tests

3. **CONCLUSIONS**
   - Final Remarks
Sequential processing model

- Program invokes parser with a set of callback functions
- Parser scans input from start to finish
  - `<element attributes...>`
  - `content`
  - `</element>`

- Invokes callbacks in file order
  - `startElement()`
  - `content()`
  - `endElement()`
1 INTRODUCTION
- Large XML Data
- Ubiquity of Multi-processing Capabilities
- SAX-based parsing

2 PARALLEL XML
- PIXIMAL: Parallel Approach for Processing XML
- Serial NFA Tests

3 CONCLUSIONS
- Final Remarks
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**

- **State 0**: Start state, transitions on whitespace, '<', '/'
- **State 1**: Transitions on '>', name start
- **State 2**: Transitions on '>', name char
- **State 3**: Transitions on '>', '='
- **State 4**: Transitions on '>', not '<' or '&'
- **State 5**: Transitions on '>', not '<' or '&'
- **State 6**: Transitions on '>', not '<' or '&'
- **State 7**: Transitions on '>', not '<' or '&'
- **State 8**: Transitions on '>', name start
- **State 9**: Transitions on '>', name char
- **State 10**: Transitions on '>', name char

- **Transition Rules**:
  - Transitions on whitespace:
  - Transitions on '<'
  - Transitions on '/'
  - Transitions on '>
  - Transitions on '='
  - Transitions on not '<' or '&'

- **Final States**:
  - States 0 and 7

This DFA is used in the Piximal-DFA approach for processing XML data.
Piximal-DFA Implementation Details

- `mmap()` of 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
• 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** $\text{Piximal-DFA}$
    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-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.
  - The final \((100 - \text{split\_percent})\)% of the input is divided evenly across the remaining \((\text{number\_of\_threads} - 1)\) partitions.
  - The final partition element gets up to \(\text{number\_of\_threads} - 2\) fewer characters.
OUTLINE

1. INTRODUCTION
   - Large XML Data
   - Ubiquity of Multi-processing Capabilities
   - SAX-based parsing

2. PARALLEL XML
   - PIXIMAL: Parallel Approach for Processing XML
   - Serial NFA Tests

3. CONCLUSIONS
   - Final Remarks
**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
Test Conditions

- Synthetic data
  - Arrays of Integers, Strings, Mesh Interface Objects
  - SOAP encoded
  - Same as previously presented in benchmarks

- Across a cluster (taking mean of results)

- Range of input sizes

- Range of parameters (split_percent, number_of_threads)
**Modest Speedup Scalability for 10,000 Integers**
**Split Percent** Critical for Speedup for 10,000 Integers

![Graph showing the relationship between Split Percent and Potential Speedup](image)

- **Max Speedup**
- **Mean Speedup**
- **Min Speedup**
Inconsistent Speedup Over a Range of Array Lengths

The chart illustrates the potential speedup of a parallel approach for processing XML data over a range of array lengths. The graph shows the max, mean, and min speedup values across different array sizes.

- **Max Speedup**: The highest potential speedup achieved.
- **Mean Speedup**: The average potential speedup across different array sizes.
- **Min Speedup**: The lowest potential speedup achieved.

The x-axis represents the array size, while the y-axis shows the potential speedup. The graph indicates that the speedup is inconsistent and varies significantly over the range of array lengths tested.
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
  \[ ...<i>1234</i><i>1235</i><i>1236</i>... \]
Speedup improves with **Thread_Count** for 10,000 Strings

![Graph showing speedup improvement with thread count](chart.png)
**Split.Percent** LESS CRITICAL FOR 10,000 STRINGS

![Graph showing potential speedup for different split percentages.](image)

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

**Y-axis:** Potential Speedup

**X-axis:** Split Percent
Consistent Speedup Over a Range of Input Sizes
Characters in 10,000 Strings are Mainly in Content
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
1. **Introduction**
   - Large XML Data
   - Ubiquity of Multi-processing Capabilities
   - SAX-based parsing

2. **Parallel XML**
   - **Piximal**: Parallel Approach for Processing XML
   - Serial NFA Tests

3. **Conclusions**
   - Final Remarks
Scientific applications strain existing XML infrastructure

A parallel parsing approach is necessary to achieve increased parser performance as document sizes grow

Restricting XML slightly should provide better performance at a low semantic cost

Piximal’s applicability is dependent on the characteristics of the input file
1. **INTRODUCTION**
   - Large XML Data
   - Ubiquity of Multi-processing Capabilities
   - SAX-based parsing

2. **PARALLEL XML**
   - **Piximal**: Parallel Approach for Processing XML
   - Serial NFA Tests

3. **CONCLUSIONS**
   - Final Remarks
Thank you for your time.
Questions?
The following slides are additional and not part of the presentation.
Limitations

- PThread overhead during concurrent runs
- Restrictions on XML format
  - Namespaces
  - CDATA
  - Unicode
  - Processing Instructions
  - Validation
- Optimal splitting algorithm unknown
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)
**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)
PERSPECTIVE PLOT FOR 10,000 INTEGERS
Perspective Plot for 10,000 Strings

- Thread Count: 2, 3, 4, 5, 6, 7, 8
- Split Percent: 20, 40, 60, 80
- Potential Speedup: 0.5, 1.0, 1.5, 2.0, 2.5, 3.0