Approaching a Parallelized XML Parser Optimized for Multi-Core Processors*

Michael R. Head
Grid Computing Research Laboratory
Binghamton University
P.O. Box 6000
Binghamton, NY 13902-6000
mike@cs.binghamton.edu

Madhusudhan Govindaraju
Grid Computing Research Laboratory
Binghamton University
P.O. Box 6000
Binghamton, NY 13902-6000
mgovinda@cs.binghamton.edu

ABSTRACT

Very large scientific datasets are increasingly becoming available in XML formats. At the same time, multi-core processing is increasingly becoming available on desktop- and laptop-class computing machines. Unfortunately, most XML parsers are still using algorithms that are inherently serial, which show little improvement on newer computing hardware. The current XML implementation landscape does not adequately meet the performance requirements of large scale applications. Thus far, applications using Web services (in the grid community, for example) have largely focused on XML protocol standardization and tool building efforts, and not on addressing the performance bottlenecks when dealing with large volumes of XML data. Generic parallel parsing has been studied in depth over the past thirty years. However, as yet, these results have not been applied to the problem of XML parsing. XML documents have some structural properties that make it more amenable to parallelized parsing than general context-free languages. As has been previously shown, XML parsers spend a large percentage of time tokenizing the input in an inherently serial process, typically running a deterministic finite automaton on the input. Our initial approach, described here, separates the process of parsing the XML from the process of reading the input. We take a well-known high performance parser, Piccolo, and apply two different strategies, Runahead and Piped, and examine the timing of the file read time and hence the overall time to parse large scientific XML files. Under the conditions tested here, performance decreases.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Performance

Keywords

XML, Parallelization, Parsing, Multi-Core

*Supported in part by NSF grant CNS-0454298 and DOE grant DE-FG02-08ER25803

SOCP'07, June 26, 2007, Monterey, California, USA.

1. INTRODUCTION

The widespread adoption of Web services in grid middleware and large scale distributed applications is primarily due its rich features including extensibility, flexibility, namespace qualification, data-binding to various languages, and support for wide variety of types. The focus thus far in grid middleware and distributed application design has been on *how* the service-oriented functionality can be achieved using Web services standards. However, performance considerations for Web services is now of critical importance as the volume of XML data used for specification, communication, and data representation has steadily increased over the years in both grid and business applications. For example, the MetaData Catalog Service (MCS) [16] 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 format have emerged as critical tools to facilitate in the development of complex large-scale scientific applications such as mesoscale meteorology [6]. 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 documents [4].

A recent trend in computer architecture is the rapid movement of the microprocessor industry towards chip multiprocessors (CMPs), commonly referred to as multi-core processors and multi-threaded cores. Web services based applications will extensively be 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. In our previous work we showed that most implementations of Web services do not scale well when the size of the XML document that needs to be processed is increased [8, 9], and the performance limitations will be exacerbated on multi-core processors simply because unithreaded processes will see smaller gains as chip-level performance is achieved through more parallelism.

A significant transformation is necessary in the design of Web services toolkits to adapt to the change in hardware technology. There exists a need for programming models

that support concurrency natively with efficient constructs for synchronization among the multiple threads of execution. To address this concern, we have devised new XML processing algorithms that take into account the growing volume of XML data and aim to provide efficient and scalable processing solutions that are tailored to the needs of current and emerging applications.

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 documents should scale gracefully with the increase in document size and the number processing cores per node.

The thrust of our research is the study and development of new techniques for parallelizing parsers for very large XML documents. Parallel compilation has been studied for many years [1, 3, 7, 10], 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 documents (with some notable exceptions [11, 12]). There are a number of reasons for this:

- modern processors are "fast enough" for most parsing tasks,
- XML parsing often represents a small part of the overall execution profile of applications, and
- a large percentage of the work done by parsers is concerns the inherently serial scanning task.

Recent work by Zhang et al. has demonstrated that it is possible to achieve high performance serialized parsing. They have developed a table driven parser that combines the parsing and validating an XML document in a very efficient way [17]. While this technique works well for serial processing, it is not tailored for processing on multi-core nodes, especially for very large document sizes.

The work presented in this paper is based on our hypothesis that the scanning task *can* be parallelized in an efficient way for large XML documents. The motivating factor for our work is the fact that enormous XML files (upwards of 1GB) are becoming prevalent in scientific applications, and new tools need to be developed to address this challenge. Additionally, 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 file processing.

2. ARCHITECTURAL CHANGES

In approaching this problem, we examined a number of parsers. From our previous work [8, 9], we learned about the performance characteristics of a number of XMLparsers. Based on the results in Figure 1, 2, 3, we can conclude that XML parsers do not scale well when the size of the

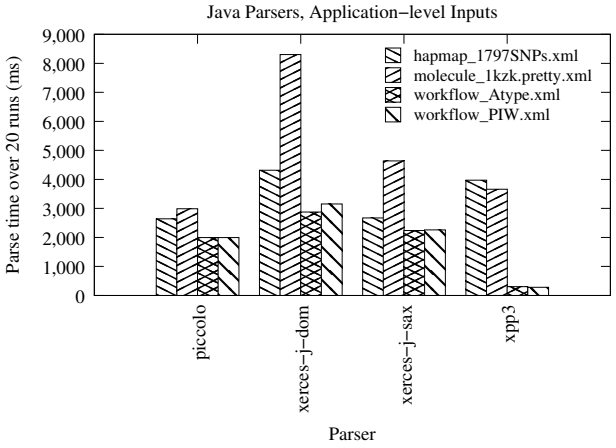

Figure 1: Performance of Java-based parsers on some large grid applications. Files sizes range from 277KBytes (workflow_PIW.xml) to 4.9MBytes (hapmap_1797SNPs.xml) and are parsed 20 times in succession.

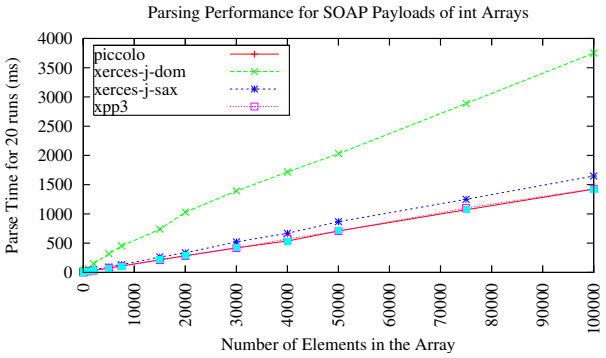

Figure 2: Scalability of Java-based parsers over arrays of integers in SOAP payloads. We see that *Piccolo* and *XPP3* perform the best in this test.

document to be processed is very large. Also, among the widely used Java-based XML parsers, Piccolo [13] 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. For our project, we wanted to work with a high performance parser, and also one that uses scanner and parser generators such as Flex [5] and Bison [2]. 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 [13], which is designed for high performance serial processing, and is implemented using scanner and parser generators.

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

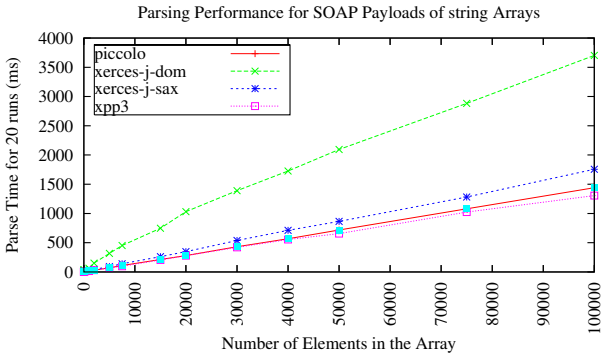

Figure 3: Scalability of Java-based parsers over arrays of strings in SOAP payloads. The elements in the arrays are text strings, as opposed to textual representations of numbers.

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 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 **Piped** 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.
- The **Piped 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.
- The **trending data** set of tests fill in the points in between the two previous tests. The file is parsed an increasing number of times, from 1 to 20, from an uncached source file to expose the trends as the I/O costs of the first run are amortized over successive runs.

3. PERFORMANCE RESULTS

These initial performance measurements were taken on a Dell Precision 390 Workstation with an Intel Core 2 CPU 6600 clocked at 2.40GHz with 1GB of dual-banked 1.9ns PC2-4200 SDRAM. The hard drive is a 160GB SATA running at 7200RPM with a 8MB on-drive cache. We use a 683MB XML file representing a protein sequence database [14] 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 6.10 (Edgy). 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 [15] to analyze and plot the results.

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 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.

For each of the different tests, we loop this code a varying number of times. We run three distinct sets of tests.

3.1 Precached Data

In this test, we run the timing loop 20 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 figure 4.

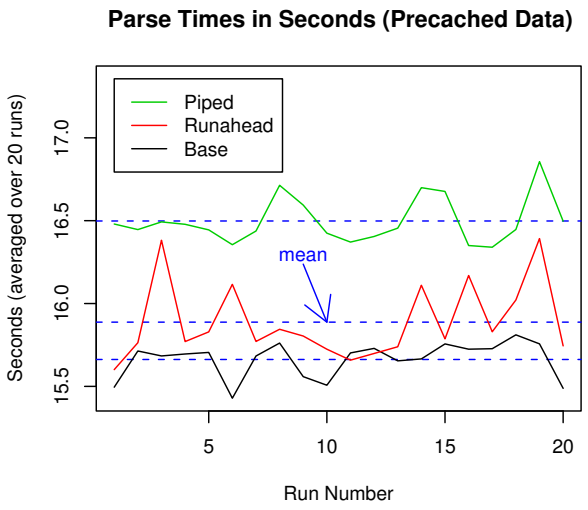

Figure 4: Parse time averaged over 20 runs when the file was precached in file cache. The overhead of the extra thread causes the Runahead case to perform slightly worse than the Base case, while the overhead of thread synchronization causes the Piped case to perform the worst.

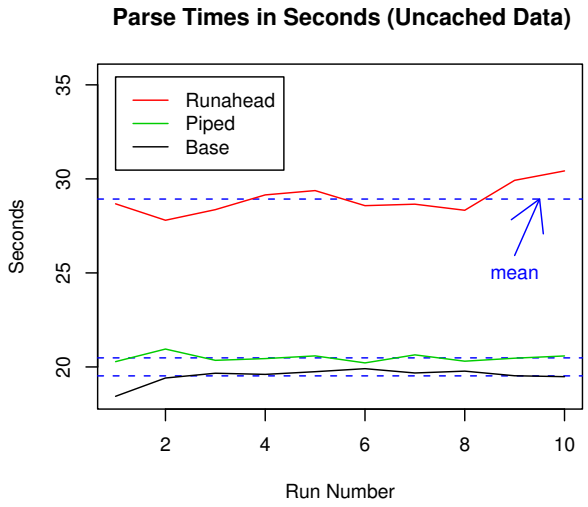

Figure 5: Here we show the performance of each parser when reading a large uncached file. In contrast to figure 4, the Runahead case performs much worse, due to competition for disk I/O between threads.

The results are normalized by dividing each data point by 20 (the number of parses per loop) to get a number that is comparable to the other results. Almost without exception, the Base parser performs better than the Runahead parser, and both always perform better than the Piped parser. The Piped parser's file reader passes the entire contents of the file through a pipe to the actual XML parser. The nature of this constant inter-thread communication adds synchronization overhead. Since the input file should be entirely in cache, there's little benefit in using the pipe because it merely duplicates data already existing in memory.

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-wise copy of the same file to warm up the JIT as well as to clear the cache as best as possible. The results are displayed in figure 5. The first thing we notice is that the Runahead parser is significantly slower this time. One possible explanation is that main thread and the runahead thread are competing for disk I/O, due to the blocking nature of `read()`. Another difference between this and figure 4 is that the data appears smoother in this case, but this is an illusion of scale. The data is just as noisy, but the view is zoomed out by a factor of approximately 10.

3.3 Trending Data

Because of the disparity in the results between the first two data sets, we devised a test to reveal where the crossover point is. In order to do this, we run 20 sets of tests. For each of the 20 tests, the timing loop runs 10 times per parser and varies the number of parses per loop from 1 to 20. We use the same warmup file as in the uncached case to repeat

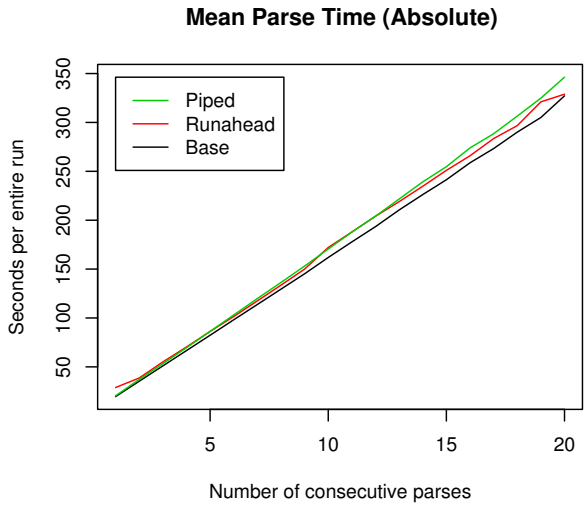

Figure 6: In order to uncover the crossover point indicated by figures 4 and 5, we run the uncached test with an increasing number of parses per run, from 1 to 20. The absolute values are presented here.

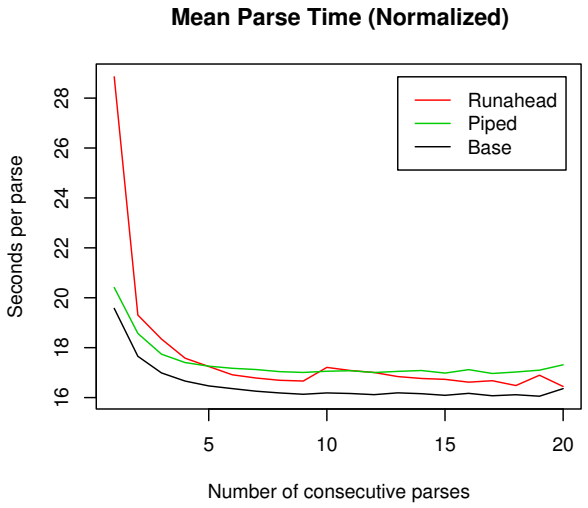

Figure 7: Here the results show in figure 6 are normalized to show the number of seconds each parse takes in a test run, thus when 10 parses are made in a run, the absolute result is divided by 10.

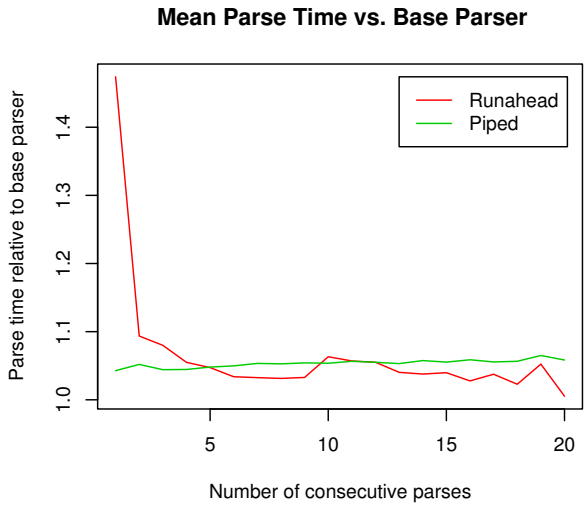

Figure 8: In this case we take the results from 6 and calculate the overhead of the Runahead and Piped parsers relative to the Base parser by dividing the absolute results of the parser by the absolute results of the Base parser. This indicates that the the Piped parser has a roughly constant overhead of 5%.

the circumstances demonstrated in figure 5. As the number of parses increases, the test case comes closer to the cached case. This is because the cost of loading the file into the cache the first time was amortized over the rest of the runs. Once these runs are complete, we take the mean of each run's timings for each parser and plot it a few ways. Figure 6 shows the raw data, matching the number of times the file is parsed to the time the entire timing loop takes in the expected linear fashion. Even so, we can see that the Runahead parser starts out performing worse than the Piped parser and ends up performing better by the end.

We perform two operations on the data. First we normalize the data by dividing the raw value as shown in figure 6 by the number of parses in each run in a manner analogous to the normalization performed for figure 4. The result is presented in figure 7. The crossover at about 5 on the x-axis is clear, and while the data for the Runahead parser is a little noisy around 10, the trend is clear. We show the overhead of the two multi-threaded parsers relative to the Base parser in figure 8. For this case, we take the absolute values from figure 6 and for each point on the x-axis, we divide the time for the parser of interest by the time for the Base parser. We can see that the Piped parser has a constant overhead of about 5%, while the Runahead parser's overhead drops from around 45% to about 4% as the cost of caching the file is amortized.

4. CONCLUSION

The goal of this work is not to immediately improve on XML parsing performance, but to begin to attack the problem with a performance data-oriented approach. One useful result here is that, for files of this large size, the overhead of pipe synchronization is constant and reasonable (5%). This means that pipes (using 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 as they were rewritten for performance in 1.6. This was actually a problem early on when we were using a Java 1.5 runtime.

Another point to highlight here is that the Piped parser has a much smoother performance profile than the Runahead parser. We can conclude that not only is a piped communication mechanism useful, but that the overhead of synchronization actually provides this nice feature. On top of this, while the pipe implementation used here performs well, it may be possible to write an optimized byte array transfer mechanism that outperforms the generic pipes used here. We plan to investigate this in our future work.

One other result is that when parsing large uncached files, attempting to preload the file in a separate thread is not helpful for performance. This result is weaker because of the way our tests are structured. It's possible that the Runahead parser could have a better result if the SAX event handling code written by the user did something other than return immediately. Because of this, we conjecture that there is little chance for the Runahead thread to do much useful preloading work and it ends up competing with the parser thread for bytes of the file from the filesystem.

5. FUTURE WORK

In the immediate future, we plan to test on more systems and add a few more cases of large XML based documents used in real applications. We would like to study how these strategies apply on processors with hyperthreading technology and classic symmetric multi-processors, as well as uniprocessors. It would be useful to investigate how these strategies affect cases when the user code actually runs

during the parsing phase, and see if the performance profile for the Runahead case changes. We plan to add new cases to our current test suite including tests against other pipe implementations and tests involving using cached data with the tests from §3.3. It will also be interesting to integrate these parsers with our comprehensive XML benchmark suite, XMLBench [9], and compare them to all the parsers used for the benchmark evaluation. Similarly, it would also be interesting to apply the Piped and Runahead cases to other high performance parsers in C and C++.

For really enormous files, there is also the possibility of coming up with a parser that parallelizes across address spaces and the network.

6. REFERENCES

[1] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall Professional Technical Reference, 1972.

[2] Bison - GNU parser generator. Bison is a general-purpose parser generator that converts an annotated context-free grammar into an LALR(1) or GLR parser for that grammar, 2006. `http://www.gnu.org/software/bison/`.

[3] J. Cohen, T. Hickey, and J. Katcoff. Upper Bounds for Speedup in Parallel Parsing. *Journal of the ACM*, 29(2):408 – 428 , April 1982.

[4] eBay. eBay Developers Program. `http://developer.ebay.com/developercenter/soap/`.

[5] Flex (The Fast Lexical Analyzer). Flex is a tool for generating scanners, 2006. `http://flex.sourceforge.net/`.

[6] D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, and A. Slominski. On Building Parallel & Grid Applications: Component Technology and Distributed Services. In *CLADE '04: Proceedings of the Second International Workshop on Challenges of Large Applications in Distributed Environments*, page 44, Washington, DC, USA, 2004. IEEE Computer Society.

[7] T. Gross, A. Sobel, and M. Zolg. Parallel compilation for a parallel machine. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 91–100, New York, NY, USA, 1989. ACM Press.

[8] M. R. Head, M. Govindaraju, A. Slominski, P. Liu, N. Abu-Ghazaleh, R. van Engelen, K. Chiu, and M. J. Lewis. A Benchmark Suite for SOAP-based Communication in Grid Web Services. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 19, Washington, DC, USA, 2005. IEEE Computer Society.

[9] M. R. Head, M. Govindaraju, R. van Engelen, and W. Zhang. Benchmarking XML Processors for Applications in Grid Web Services. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 121, New York, NY, USA, 2006. ACM Press.

[10] H. P. Katseff. Using data partitioning to implement a parallel assembler. In *PPEALS '88: Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 66–76, New York, NY, USA, 1988. ACM Press.

[11] W. Lu, K. Chiu, and Y. Pan. A Parallel Approach to XML Parsing. In *7th IEEE/ACM International Conference on Grid Computing (Grid 2006)*, pages 223–230, 2006.

[12] Y. Pan, W. Lu, Y. Zhang, and K. Chiu. A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs. In *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)*, Rio de Janeiro, Brazil, 2007.

[13] Piccolo XML Parser for Java. Piccolo is a small, extremely fast XML parser for Java, 2006. `http://piccolo.sourceforge.net/`.

[14] Protein Sequence Database. Integrated collection of functionally annotated protein sequences. `http://www.cs.washington.edu/research/projects/xmltk/xmldata/data/pir/psd7003.xml`, 2001.

[15] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2007.

[16] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman. A Metadata Catalog Service for Data Intensive Applications. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 33, Washington, DC, USA, 2003. IEEE Computer Society.

[17] W. Zhang and R. van Engelen. A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 197–204, Los Alamitos, CA, USA, 2006. IEEE Computer Society.