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

Michael R. Head[†] Madhusudhan Govindaraju[‡]
Grid Computing Research Laboratory, SUNY Binghamton, NY
{[†]mike, [‡]mgovinda}@cs.binghamton.edu

Abstract

Very large scientific datasets are becoming increasingly available in XML formats. Our earlier benchmarking results show that parsing XML is a time consuming process when compared with binary formats optimized for large scale documents. This performance bottleneck will get exacerbated as size of XML data increases in e-science applications. Our focus in this paper is on addressing this performance bottleneck. In recent times, the microprocessor industry has made rapid strides towards Chip Multi Processors (CMPs). The widely available XML parsers have been unable to take advantage of the opportunities presented by CMPs, instead, passing the task of parallelization to the application programmer. The paradigms used thusfar to process large size XML documents on uni-processors are not applicable for CMPs. We present the design, implementation, and performance analysis of PiXiMaL, a parallel processing library for large-scale XML-data files. In particular, we discuss an effective scheme to parallelize the tokenization process to achieve an overall performance increase when parsing large-scale XML documents that are increasingly in use today. Our approach is to build a DFA-based parser that recognizes a useful subset of the XML specification and converts the DFA into an NFA which can be applied on any subset of the input.

1 Introduction

The widespread adoption of Web services in e-science 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 e-science 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 e-science and business applications. For example, the MetaData Catalog Service (MCS) [12] 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 specification format have emerged as critical tools to facilitate in the development of complex large-scale scientific applications such as mesoscale meteorology [5]. Another example is the International HapMap project that aims to develop a *haplotype* of the human genome. The schemas used to describe the common patterns in human DNA sequence variation can have tens of thousands of elements. The XML files in the protein sequence database are close to a gigabyte in size. Processing of such large-scale XML documents has emerged as a significant bottleneck in the overall application execution time of e-science applications. It is thus critical to design new XML processing algorithms that can process the emerging large-scale XML data sizes in a scalable manner.

The nature of computing is rapidly changing with the movement of the microprocessor industry towards chip multi-processors (CMPs), commonly referred to as multi-core processors. Web services based applications are expected to be extensively 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. Our previous benchmarking work demonstrates that most implementations of Web services do not scale well when the size of the XML document, which needs to be processed, is increased [7, 8]. As memory access and latency can be the choking point in CMP processors, this performance limitation will be exacerbated on multi-core processors because performance gains need to be mainly achieved by adding more parallelism rather than serial processing speed.

In CMPs the on-chip core-to-core bandwidth and communication latency are orders of magnitude faster than in traditional chip-to-chip multiprocessing systems that are typically used for parallel computing. The recent trends and announcements from major vendors indicate that the number of cores per chip will steadily increase in the near future.

It is thus important to design XML-based e-science middleware technologies to leverage the opportunities presented by this trend and gracefully scale with the increase in the number of cores on each processing node.

It has been shown that the current XML processing landscape does not apply parallelization techniques to XML processing [7, 8]. While this was acceptable for small-sized documents, it has emerged as a significant bottleneck in e-science applications where the size of XML data can sometimes exceed size of available system memory. As a result, our focus is on harnessing the benefits of fine grained parallelism, exploiting SMP programming techniques to process large-scale XML-based application documents, and designing algorithms that scale well with increase in number of processing cores.

Parallel compilation has been studied for many years [2, 9], 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. This can be attributed to two reasons: (1) modern processors are "fast enough" for most parsing tasks for small-sized documents; (2) it is challenging to automatically identify the parts that need to be serially scanned and those that can be parallelized.

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 100MB) 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 XML document processing.

The specific contributions of our work include:

- We propose techniques to modify the lexical analysis phase for processing large-scale XML datasets to leverage opportunities for parallelism.
- We contrast the effect of our techniques on various processor configurations. We also find the practical limits for this particular parallelization approach, testing on a variety of systems, under a variety of conditions.
- We study efficient use of multiple threads to read from a large applications data file to achieve gains in overall throughput.
- We study how the complexity of large-scale XML data affects possibilities for parallel processing on multiple cores.
- We model the work done by our parser with tests to find the practical limits of our parallelization approach.

The reference implementation of the WSRF specification, available from the Globus Alliance website is based on the toolkit. The architecture of the reference implementation is modular in nature and facilitates the use of specialized pluggable modules for various various aspects of Web services implementation stack. The work presented in this paper can be incorporated as an optimization module in the WSRF implementations for the XML processing phase.

2 Related Work

A wide range of implementations of XML parsers is available, including Xerces (DOM and SAX), gSOAP[13], Piccolo, Libxml, and XPP3. Simple and straight forward implementations of XML parsing paradigms and directly using the available XML processing toolkits result in a severe impact on performance of current and emerging e-science applications [1, 3, 8], To address the performance limitation, several novel efforts to analyze the bottlenecks and address the performance at various stages of a Web services call stack have been discussed in the literature [1, 3, 6, 13]. These optimizations are however applicable only in the uni-core case.

Recent work by Zhang et al [14] 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. While this technique works well for serial processing, it is not tailored for processing on multi-core nodes, especially for very large document sizes. One related project, MetaDFA [10, 11] toolkit, presents a parallelization approach that chiefly uses a two-stage DOM parser. It conducts pre-parsing to find the tag structure of the input before, or possibly pipelined with, a parallelized DOM builder run on its output (a list of document offsets of start and end tags). Our toolkit, PIXIMAL , however, generates SAX events and thus serves a different class of applications than MetaDFA. Additionally, PIXIMAL conducts parsing work dynamically, and generates as output a sequence of SAX events. This results in larger number of DFA (Deterministic Finite Automata) states, and more scope for optimizations for different class of application data files. For distributed filesystems, PIXIMAL is designed to work as a MapReduce application: it can distribute work to nodes that have the data and executing speculative NFAs (Nondeterministic Finite Automata) on portions of the input file that don't include the beginning.

3 Multi-Core Case: Use of DFAs in Parsing

In general, all parsers and compilers begin by *tokenizing* the input characters of the file into syntactic tokens that are used later in the parse phase. This process is generally (and efficiently) implemented with a deterministic finite automata (DFA)-based lexical scanner – whether hand coded or generated by a tool such `flex`. The scanner's job is to match different sections of the input to different parts of the language. For example, in a given XML parser, the scanner might recognize a '<' and insert a constant representing the `LEFT_ANGLE` token, and a ' ' character as a `WHITESPACE` token. In addition to recognizing keywords composed of more complicated sequences of characters ('`id`' or '</'), the tokenizer also demarcates the remaining input characters that are not contained in keywords and reliably group them into words. Every time the scanner recognizes a token, it must perform some action to store the token or pass it to a higher level part of the parser. The various token types and keywords can be defined as regular expressions, which can be readily (both theoretically and practically) converted into a DFA-based scanner. The DFA representation is chosen because of its efficiency: each character in the input need be read only once, and there is very little overhead on a per-character basis.

Unfortunately, DFAs are inherently serial in nature. It must start at the beginning of the input and proceed character-by-character until the end. It is not possible to simply split the input in two and process the first and second halves independently; there is no way to know in which state to start the DFA operating on the second half of the input.

In this way, at bottom, all the standard XML parsers are limited to a serialized indivisible scanner. For small files and desktop-style mass storage devices, this is acceptable, even desirable, because the scanner is fast – especially so for small input files – and because desktop mass storage access algorithms work well reading from a single stream from disk. If multiple threads concurrently access different disk blocks, many more read head movements will be required and overall performance will suffer as the otherwise independent threads will be waiting more often for data. Indeed, desktop users (whose applications are generally I/O bound) may see increasingly smaller performance gains as more and more cores are packed onto chips.

However, data-sets of e-science applications running on cluster-class hardware, are much more amenable to parallelization. In these circumstances, (XML) input files can be very large, larger than main memory, and mass storage is more likely to be arranged in higher performance configurations (e.g., RAID, NAS, SAN) which can more efficiently feed multiple data streams to concurrent threads. Our research results can be readily applied to these cases.

3.1 Speculative Execution with an NFA

We partition the input XML document (stored in string format as P) into N substrings, $P_1, P_2, ...P_N$, run the usual DFA-based lexical analyzer (L_{DFA}) on substring P_1 concurrently with $N-1$ speculative scanners on substrings $P_2, P_3, ...P_N$. As mentioned in §3, a DFA-based scanner can only be run on an input by starting at the initial state of the DFA at the initial character in the input. The problem is that it is not possible to know which state to start the L_{DFA} in for any substring other than P_1. To address this issue, we have designed a transformation that can be applied to create a scanner that can be applied to any of the substrings.

A DFA requires that only one state be active at a given time, whereas an NFA can be active in any number of states. For the first character in the i^{th} partition, P_i, there is some state of L_{DFA}, S_i, which L_{DFA} would be in after traversing P up to P_i. If S_i were known ahead of time, we could simply start L_{DFA} in S_i at the beginning of P_i and continue processing. Unfortunately, it is not possible to know this ahead of time. However, if L_{DFA} is transformed into an NFA, L_{NFA}, where *every* state is a start state, it can be applied to any P_k and perform some scanning work. In this way, L_{NFA} begins processing along multiple *execution paths* through L_{DFA}. Further, there is one *correct execution path* that started with S_i, which remains unknown until all the lexical analyzers running on P up to P_i complete.

Along each execution path e, there is a corresponding traversal through L_{DFA}, which would trigger some sequence of actions, $A_{i,e}$, as the DFA recognizes various tokens. Because each L_{NFA} runs without the knowledge of which state it should have been in (i.e., S_i), it cannot reliably perform the actions of $A_{i,e}$, because the path recognizing the state might not be on the correct execution path. So for each execution path e, a NFA starting at some point i must store all $A_{i,e}$ until S_i is determined. Unfortunately, most of the stored actions for L_{NFA} must be discarded, which implies that some work will be wasted.

4 Tests and Testing Environment

The XML input for all the test results presented here is `SwissProt.xml`[4] which encodes a protein sequence database. It is roughly 109 megabytes, contains 2,977,031 elements, 2,189,859 attributes, and has a maximal tree depth of 5. All tests `mmap(2)` this input file to reduce memory usage and eliminate many string copies. In all cases, input is read from the local disk – not from a network file system. Additionally, the input file is pre-read into the operating system's disk buffer, so the tests stress the CPU/RAM interface. The tests are designed to model the PIXIMAL approach described in §3.

We run these tests on a range of differently configured nodes:

- 2× uniprocessor – 1U nodes in a cluster, each of which has two 3.2Ghz Intel Xeon CPUs, 4 gigabytes of RAM and run a 64 bit version of Linux 2.6.15. Results on this class of machines are taken by averaging the timings found on 4 of these nodes. The filesystem in use in the test directory here is reiserfs.
- 2× dual core – 1U nodes in a cluster, each of which has two 2.66Ghz Intel Xeon 5150 CPUs, 8 gigabytes of RAM and run a 64 bit version of Linux 2.6.18. Results on this class of machines are taken by averaging the

timings found by running the test on 10 of these nodes. The filesystem in use in the test directory here is xfs.

- 2× quad core – 1U nodes in a cluster, each of which has two 2.33Ghz Intel Xeon E5345 CPUs, 8 gigabytes of RAM and run a 64 bit version of Linux 2.6.18. 10 nodes from this cluster were selected to perform this test and the results presented reflect the mean timings taken. The test directory on these machines is backed by a xfs filesystem.

4.1 Memory Bandwidth Test

An N-way parallel parser would concurrently be reading using N different threads, so this tests checks whether the memory subsystem can provide substantial bandwidth when sequentially reading from a very large input.

This test has two parameters: *split_percent* and *thread count*. The *split_percent* is particular to the PIXIMAL approach: it denotes the percent of input that is directed at the DFA thread. The number of threads defines the number of concurrent automata: 1 DFA and $number\_of\_threads - 1$ NFAs. The balance of the input $(input\_size * (1 - split\_percent/100))$ is divided evenly among the NFA threads. In the case that $number\_of\_threads = 1$, *split_percent* is overridden to be 100% in order to ensure that the entire input is read.

In this test, the DFA is modeled by a thread that reads each byte of memory in its partition of the input and looks up a value in a table indexed by that byte. The NFAs are modeled by threads that do precisely the same task on their given partition of the input.

4.2 State Scalability Test

The speculative threads in a parser built using NFAs will have substantially more work than the DFA thread. This test models an aspect of that extra workload – the number of states that the NFA must initially consider – to examine the affect of language complexity on the efficacy of this approach.

This test has one more parameter than the memory bandwidth test: the size (number of states) of the DFA. Here, the PIXIMAL DFA is modeled as a thread that has a *state_number* which is initialized to 0 and takes values between 0 and $dfa\_size - 1$. The next *state_number* is calculated for each byte of input by looking up the current *state_number* and current byte in a two dimensional array. The NFAs are modeled by threads that start with an array of $dfa\_size - 1$ start values, each initialized to a number between 1 and $dfa\_size - 1$. An NFA will never start in the state designated by 0, because that is a start state that is only valid before the DFA begins reading. The NFA recalculates each entry of the state array for each byte of input using the same rule as the DFA.

5 Results and Discussion

The focus of PIXIMAL is to achieve “scalable parallelism”, wherein processing of large-scale XML data continues to get good performance as the number of cores increases. The performance results are presented for classic 2× uniprocessor (two total cores), 2× dual core (four total cores), and 2× quad core (eight total cores) configurations (all SMP), as these are the processor configurations that are in use today. The trends on quad-core and 8-core nodes are good indicators for how PIXIMAL will scale to a larger number of cores, as they become available.

It is to be noted that the PIXIMAL parallelization approach of providing scalability over multiple cores may be applicable to other table-driven or DFA-based parsers. Also, though we have conducted extensive tests on large-scale XML documents, the approach is tailored to potentially work for a lexical analyzer for any structured application data format.

In the results below, *speedup* is always calculated using the formula: T_1/T_P, where T_1 is the mean time taken when no NFA threads are scheduled and T_P is the mean time taken at the particular data point – over all test runs where the test parameters and hardware configuration are identical. Mean raw timing values are used to provide an overview of the actual result space, while *speedup* is used to make certain results comparable across hardware.

5.1 Memory Bandwidth Results

The *memory bandwidth test* models the memory work being done by the DFA thread and the NFA threads by sequentially reading the input and passing it through a table. NFA threads do the same amount of work as the DFA and are set to read the sections of input that the actual PIXIMAL NFA would, and we clearly see performance wins by adding extra threads/CPUs to the task of reading the input. This suggests that we’re not causing a serious reduction in cache performance simply by reading from multiple sources of input. It also demonstrates that access to main memory is not a major limiting factor in this approach: each thread is still able to get enough data to do its (small amount of) work. This scales with the number of cores up to around 6-7 cores in the 8-core case.

Figure 1 examines the entire parameter space of this test on the 2×-quad core configuration. All *split_percent*s and *thread_count*s are displayed over the range of values tested. This provides a great deal of visual information about the test space. It is quite clear that adding threads can provide an advantage when there are spare cores to use. It is also instructive to note that once the best time is achieved by for a given *split_percent*, adding more threads does not detract much from overall performance. This may be due to high performance I/O subsystems and file systems on these cluster nodes. The best performing configuration is the one denoted by the deepest part of the “well.”

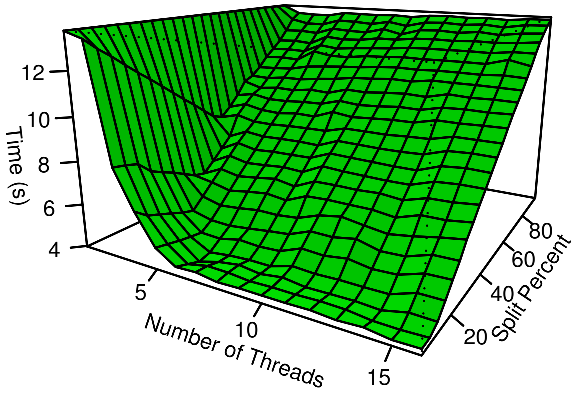

Figure 1. Input read times of the *memory bandwidth* test on an 2× quad core configuration (8 total cores). Note that adding threads continues to improve performance, though returns diminish past the 6th thread.

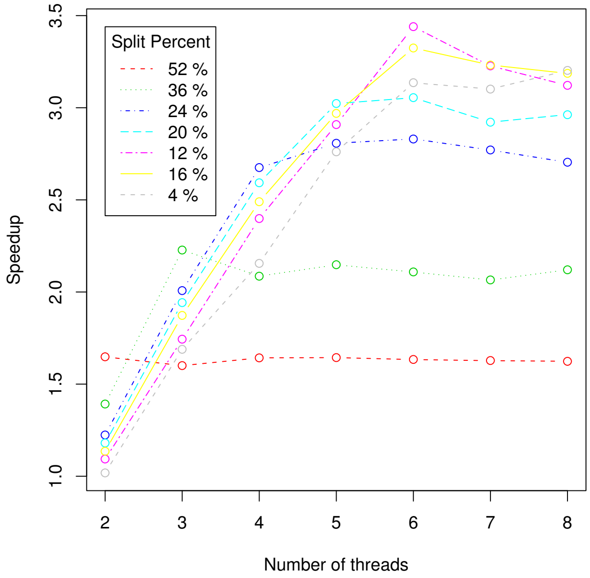

Figure 2. Speedup comparison on the 2× quad core (8 total cores) configuration for the *memory bandwidth* test. *split_percent*s are selected to maximize the speedup for each number of threads. Performance gains trail off after 6 threads are given to the test

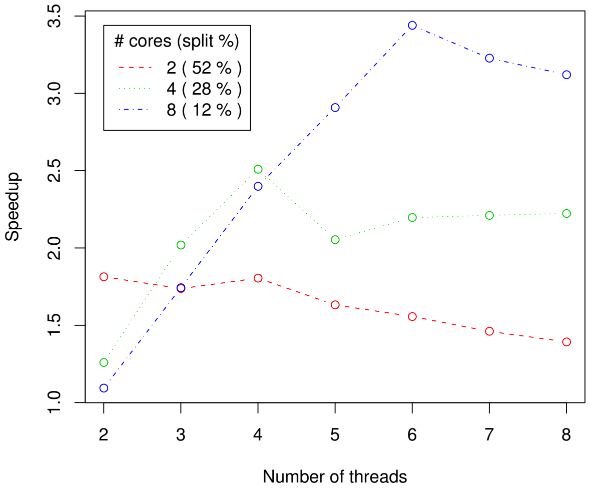

Figure 3. Best speedup achieved for each processor configuration, together with the *split_percent* that achieves that maximal speedup. Adding more cores continues to improve performance, allowing a greater portion of the input to be divided up. Memory bandwidth between the processor and the `mmap(2)`ed file is not a strict limiting factor on thread-scalability.

Figure 2 plots the speedup (over the DFA-only case) of the best *split_percent* for each of varying thread counts on the 2× quad-core systems. The *split_percent*s plotted are chosen because they maximize speedup (compared to all other *split_percent*s) at the *thread_count*s plotted. This plot reaffirms the conclusions from the three dimensional plot: shifting the *split_percent* back affords greater scalability, even up to 6 threads on 8 core machines. They also demonstrate that given an optimal split for a chosen number of threads, performance does not degrade substantially after passing that *split_percent*. Figure 2 additionally demonstrates how returns do eventually diminish as more than 6 threads are used.

Figure 3 plots the speedup for the best-case speedup for each CPU configuration, for its performance on the best-case *split_percent*. This facilitates comparison across configurations. The 4 and 8 CPU cases fare worse than the 2 CPU case at first just because this *split_percent* is fixed for each configuration in this plot. This approach was chosen to show how performance would be affected by choosing a particular configuration based on the detected machine type. It is clear that even 8 total CPU configurations can potentially provide performance improvements, even with current memory and I/O subsystems.

The structure of these tests imply that conclusions drawn

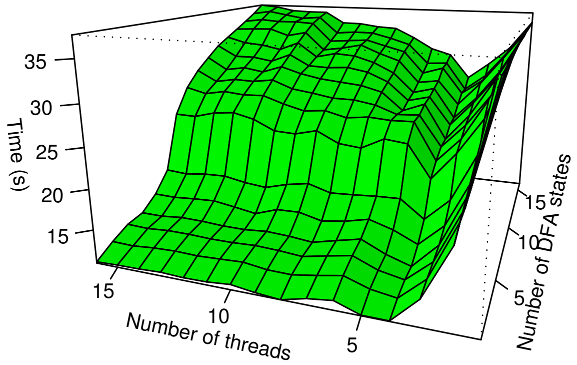

Figure 4. The lowest input read times for all DFA sizes and thread counts for the 2× dual core (4 total cores) configuration. For a given DFA size and number of threads, there is a range of possible *split_percent*s that could be chosen. The *split_percent* is chosen for each configuration of DFA size and number of threads to minimize the time.

from them apply to most parallel data access algorithms on a single machine. Each thread does the minimal amount of work so that the test is effectively working just the memory subsystem. If another data file type is to be read in parallel by multiple threads or processes each operating sequentially, such an input reader will need to do at least this work. If the workload is dominated by memory access, then these results will apply directly and it is unlikely that attempting to utilize more than 6 threads will yield much benefit on machines that are configured like those tested here.

5.2 State Scalability Results

The state scalability models the amount of work each combination of DFA/NFA threads would have to do given a certain language/parser complexity. We define *DFA complexity* by the number of states (N) in the DFA. An NFA (for this particular test) based on a DFA of complexity N will need to do $O(N)$ times more work than the DFA: for each state in the DFA, the NFA must calculate the *next state* for each character of input, whereas the DFA must only calculate the next state for one state, namely the distinguished *start state*.

Figures 4 and 5 show the best timings for each combination of DFA complexity and thread count. It is interesting to note that for most cases, it is possible to improve overall performance by taking advantage of unused cores, even when the number of threads outnumbers the cores available. There is also a significant contrast between the 2× dual core and 2× quad core configuration: the 2× quad core case shows a stable speedup, which is not indicated in the 2× dual core case. It is also apparent that the best speedup obtainable for a given DFA complexity is when the number of threads matches the number of cores. This is not a clear result of the memory bandwidth test because the extra NFA threads did not add much extra work. Here, each additional NFA does add a substantial amount of work, which increases along with the complexity of the DFA. The final item to note on these perspective graphs is the steep ledge around DFA size 6. This indicates a major crossover point where the complexity of the NFA begins to take over the bulk of the work from the memory subsystem. Greater speedup is more readily obtainable before this point. The ledge above 4 and 8 threads on the respective figures also becomes more pronounced, again indicating that a significant crossover point has been met.

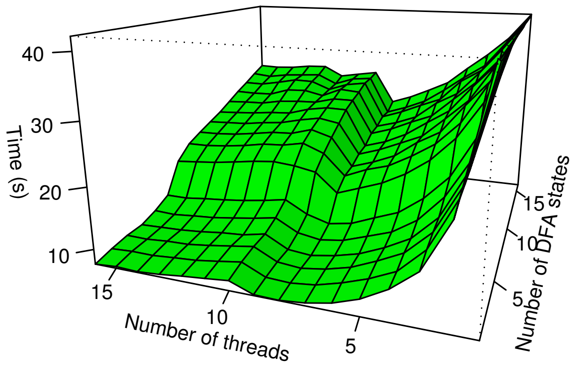

Figure 5. Input total read times for 2× quad core (8 total cores) configurations. In contrast to figure 4, there is a continued performance increase over the space of parameters tested.

Figures 6 and 7 are plotted similarly to figure 2. That is, given a configuration (number of cores, number of DFA states), a split point is chosen that applies for all threads. The split point, which can be thought of as the *globally optimal split point* for a given DFA size is chosen to maximize the speedup across all thread counts shown on the horizontal axis. This is done to show how the parser in a given configuration acts as the number of NFAs used is increased. Because of this, there is a slowdown shown for some cases. In those cases, a better split point could have been chosen that would demonstrated some speedup, specifically, the split point chosen for display for that configuration in figure 4 or 5. A variety of DFA state sizes are plotted which reinforce a conclusion from above: there is a major changeover when DFA complexity is greater than 6. Examining the globally optimal *split_percent* for each state suggests why this is so: the split point must shift much farther into the document. For example, in the 2× dual core case, a 6 state DFA yields

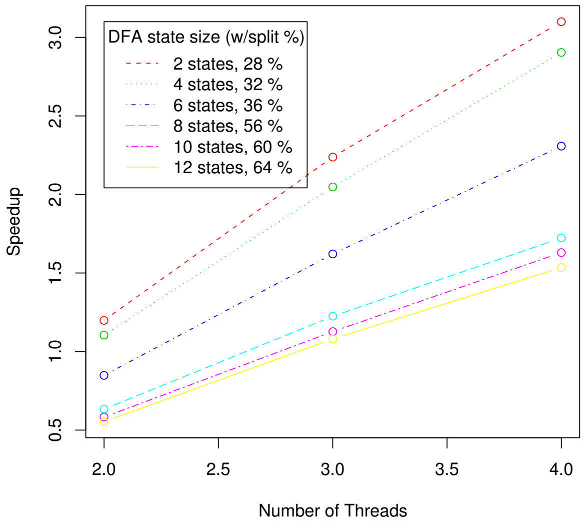

Figure 6. Speedup for a variety of DFA sizes for 2× dual core (4 total cores) configurations. This test models a part of the extra work done by the NFAs over the DFA. For each DFA size, the *split_percent* is chosen which provides the maximal speedup for some number of threads. It is clear that NFAs built from larger sized DFAs incur stuff performance penalties.

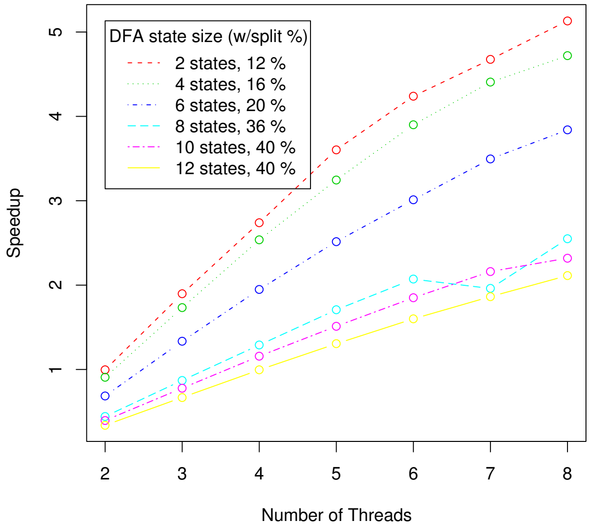

Figure 7. Same as figure 6, but for the 2× quad core (8 total cores) configuration. 6 states appears to be a good DFA size for this particular model of the parallel parser.

an optimal *split_percent* of 36%, slightly more than that of the 4 state DFA case. The 8 state DFA yields an optimal *split_percent* of 56%: the best performance is achieved here when the DFA thread processes over half of the input. Under such circumstances, it will be impossible to achieve even a 2× speedup over the single threaded DFA case, no matter the number of cores available. The situation is similar on the 2× quad core hardware. At between 6 and 8 states, the optimal *split_percent* jumps disproportionately, again due to this crossover where read time begins to be dominated by the complexity of the DFA/NFA rather than simply reading the input from memory.

Of special note here is that in figure 7, the maximal speedup is greater than that shown in 3 (5 vs. 3.5). This is likely due to the fact that the baseline comparison case (the DFA model running over the entire input) is doing substantially more work, so dividing that work out to the NFAs is a significant win, particularly when the number of states is small, because the extra work done by the NFAs is not large. In other words, the NFAs are doing $O(N)$ more work than the DFA, but when N is small, the constants dominate and the constants here are small.

5.3 Conclusions

The PiXiMaL framework allows application programmers to quantify the exact number of threads, processing cores, and split percentage for each core, that should be used for their application data files.

The PiXiMaL approach to reading large-scale structured data files, such as XML documents, effectively uses the available cores on a node. Based on our tests on a variety of CPU configurations, we conclude that even with current memory and I/O subsystems, processing large-scale data files can potentially provide performance improvements. Memory bandwidth between the processor and `mmap(2)`ed files is not a strict limiting factor on thread-scalability.

Memory-bandwidth tests show that when multiple threads process application data, such as structured XML files, the performance scales to 6-7 cores, in the 8-core case. Also, adding threads continues to improve performance upto 6 threads, in the 8-core case.

The division of work between threads plays a key role in enhancing the performance. Shifting the *split_percent* back affords greater scalability, up to 6 threads on 8 core machines. Given an optimal split for a chosen number of threads, performance does not degrade substantially after passing that *split_percent*. PiXiMaL framework can be use to determine the major crossover point where the complexity of the NFA begins to take over the bulk of the work from the memory subsystem.

It is critical that the complexity of the DFA remain low. Our tests indicated that such a DFA should have no more than 6 states. A lower complexity DFA implies a lower complexity language: a language with less syntax. This implies that if a given e-science application file is guaranteed to use a restricted set of XML features (instead of attributes, define sub-elements, for example), the data can be efficiently processed in parallel on a multi-core architecture. PIXIMAL framework can be used to guide application developers to design the optimial structure for large-scale data files.

6 Future Work

In future work we plan to thoroughly study pre-fetching and piped implementation techniques that can enhance the performance of PIXIMAL . We will study the effect of operating system-level caching on the parsing process of large documents that may be read more than one time. We will develop algorithms for optimal layouts of DFA tables in memory to efficiently process frequently occurring transitions. We will also accurately model and obtain performance data for the additional work done by NFAs in PIXIMAL including detecting and queuing SAX events that need to be played back when its initial state is finally calculated. We will build a MapReduce extension of PIXIMAL to process large documents that are stored in a distributed file system. We will further study the scalability of PIXIMAL as processors with multiple cores (greater than 8) are available for research and testing purposes.

References

[1] N. Abu-Ghazaleh and M. J. Lewis. Differential Deserialization for Optimized SOAP Performance. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 21, Washington, DC, USA, 2005. IEEE Computer Society.

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

[3] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the Limits of SOAP Performance for Scientific Computing. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 246, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Expert Protein Analysis System. SwissProt curated protein sequence database. `http://www.cs.washington.edu/research/xmldatasets/www/repository.html`.

[5] 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.

[6] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and evaluation of RMI protocols for scientific computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomput*, page 61, Washington, DC, USA, 2000. IEEE Computer Society.

[7] 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.

[8] 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.

[9] 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.

[10] Y. Pan, W. Lu, Y. Zhang, and K. Chiu. A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 351–362, Washington, DC, USA, 2007. IEEE Computer Society.

[11] Y. Pan, Y. Zhang, K. Chiu, and W. Lu. Parallel XML Parsing Using Meta-DFAs. In *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 237–244, December 2007.

[12] 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.

[13] R. van Engelen. gSOAP: C/C++ Web Services and Clients, 2007. `http://www.cs.fsu.edu/~engelen/soap.html`.

[14] 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.