Parallel and Distributed Approach for Processing Large-Scale XML Datasets

Zacharia Fadika [1], Michael R. Head [2], Madhusudhan Govindaraju [3]

Computer Science Department, Binghamton University
P.O. Box 6000, Binghamton, NY 13902-6000, USA
[1] zfadika@cs.binghamton.edu
[2] mike@cs.binghamton.edu
[3] mgovinda@cs.binghamton.edu

***Abstract*—An emerging trend is the use of XML as the data format for many distributed scientific applications, with the size of these documents ranging from tens of megabytes to hundreds of megabytes. Our earlier benchmarking results revealed that most of the widely available XML processing toolkits do not scale well for large sized XML data. A significant transformation is necessary in the design of XML processing for scientific applications so that the overall application turn-around time is not negatively affected. We present both a parallel and distributed approach to analyze how the scalability and performance requirements of large-scale XML-based data processing can be achieved. We have adapted the Hadoop implementation to determine the threshold data sizes and computation work required per node, for a distributed solution to be effective. We also present an analysis of parallelism using our PIXIMAL toolkit for processing large-scale XML datasets that utilizes the capabilities for parallelism that are available in the emerging multi-core architectures. Multi-core processors are expected to be widely available in research clusters and scientific desktops, and it is critical to harness the opportunities for parallelism in the middleware, instead of passing on the task to application programmers. Our parallelization approach for a multi-core node is to employ a DFA-based parser that recognizes a useful subset of the XML specification, and convert the DFA into an NFA that can be applied to an arbitrary subset of the input. Speculative NFAs are scheduled on available cores in a node to effectively utilize the processing capabilities and achieve overall performance gains. We evaluate the efficacy of this approach in terms of potential speedup that can be achieved for representative XML data sets.**

I. Introduction

Scalable processing of XML datasets is an immediate concern as the size of XML data used by applications has steadily increased over the years in both scientific and business applications. For example, recognizing the increasing role of XML in representation and storage of scientific data, XDF, the eXtensible Data Format for Scientific Data, is being developed at GSFC's Astronomical Data Center (ADC), to describe an XML mark-up language for documents containing major classes of scientific data. This effort is expected to define a generic XML representation to accommodate the diverse needs of various scientific applications. The MetaData Catalog Service (MCS) [1] provides access via a Web service interface to store and retrieve descriptive information (metadata) on millions of data items. While the Web service approach of MCS provides interoperability, it also hurts the performance when compared to use of a standard database for storage and retrieval. Scientific applications such as Mesoscale meteorology [2] depend on the orchestration of several workflows, defined in XML format. The international HapMap project 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

Our earlier work on benchmarking XML processing showed that for most XML toolkits scalability is adversely affected as the size of the XML datasets increase [3], [4]. These toolkits are typically designed to process small-sized XML datasets. The recent trends and announcements from major vendors indicate that the number of cores per chip will steadily increase in the near future. The performance limitation of existing XML toolkits will likely be exacerbated on multi-core processors because performance gains need to be mainly achieved by adding more parallelism rather than serial processing speed. Additionally, scalable processing of XML data is now of critical importance in scientific applications where the size of XML can exceed hundreds of megabytes, and processing on a single node may not be a viable option. As a result, our focus is both on micro-level as well as macro-level parallelism. At the micro-level, we harness the benefits of fine grained parallelism, exploiting well-known SMP programming techniques, and design processing modules that scale well with the increase in number of processing cores on a single node. At the macro-level, we apply the distributed processing of large-scale data stored in a cluster, by applying the MapReduce processing paradigm [5]. An advantage of the MapReduce model is that it has relaxed synchronization constraints, which works favorably for large-scale XML data sets, wherein typically namespaces that are once defined at the start of the document are not redefined in in the inner-elements. The simplicity and robustness of the MapReduce model also reduces the burden on application programmers.

The emergence of Chip Multi Processors (CMPs), also called multi-core processors, provides both opportunities and challenges for designing an XML processing toolkit tailored for large-size XML data sets.

Compared to classic symmetric multi-processing systems (SMPs) of independent chips, the communication costs of on-

chip shared secondary cache in CMPs is considerably less, providing opportunities for performance gains in fine-grained multi-threaded parallel code. CMPs provide special advantages due to locality. The individual cores are more closely connected together than in an SMP system. Multiple cores on the same chip can possibly share various caches, including the TLB, and the bus. An important design consideration is that off-chip memory access and latency can be the choking point in CMP processors. Our approach is to speculatively execute XML processing tasks on available cores to leverage the opportunities for fine-grained parallelism available while processing XML datasets on a multi-core node.

In this paper we present a distributed implementation of large-scale XML processing, adapting the splitting module of the Hadoop implementation [6], to determine the threshold values for data size and number of processing nodes. The overall experimental data and analysis can be used to quantify the exact configuration to be used to gain performance enhancements for a given application setting.

II. Related Work in XML Processing

XML primarily uses UTF-8 as the representation format for data and various studies have shown that this representation format can hinder the overall application performance. Sending commonly used data structures via standard implementations of XML based protocols, such as SOAP, incurs severe performance overheads, making it difficult for applications to adopt Web services based distributed middleware [4]. Recent work by Zhang et al [7] 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. 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 [7]–[10]. The toolkits that work efficiently in the uni-core case include gSOAP [11], VTD-XML [12], Libxml [13], Expat [14], Qt4 [15], and Piccolo [16]. The widely used Xerces (DOM and SAX) [17] toolkit, has a rich set of features, but suffers from significant performance bottlenecks.

In our previous work in this area, we focused on state-scalability for the parser and the memory requirements for arrays of primitives when multiple threads operate concurrently to read large input files [18]. One related project by Pan et. al., the MetaDFA [19], [20] toolkit, presents a parallelization approach that 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 states, and more opportunities for optimizations for different class of application data files. Additionally, for distributed filesystems, we have designed the processing as a MapReduce application by distributing work to nodes to execute the processing of a select number of document parts.

III. Micro-Parallelism for XML Data Processing

A deterministic finite automata (DFA)-based lexical scanner is generally used to *tokenize* the input characters of the file (or string, as in the case of XML) into syntactic tokens that are used later in the parse phase. The DFA based lexical scanner is sometimes hand-coded, and frequently generated by a tool such as `flex`. 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 of XML, used in distributed applications, can be defined as regular expressions. A DFA-based scanner can be custom-designed to process the subset of the XML specification used in defining large-scale data files in applications. The DFA model for processing is efficient: each character in the input XML document is read only once, minimizing the overhead on a per-character basis.

The DFA approach does not directly lend itself to parallelism. It is required to start at the beginning of the input and process all the characters sequentially. As there is no way to determine in which state the DFA will be in after processing a certain section of the input, it is not possible to simply split the input in two (or more sections) and process the different sections independently. Due to this reason, all the widely used XML parsers are limited to a serialized indivisible scanner. This approach has thus far been acceptable for small files and desktop-style mass storage devices, because the scanner is fast for small input files. Additionally, this approach blends well with desktop mass storage access algorithms that work well reading from a single stream from disk.

A. Speculative NFA Execution in PIXIMAL

The speculative execution approach of PIXIMAL is to divide the input XML document, P, into N substrings, $P_1, P_2, ...P_N$. The processing on substring P_1 is carried out using the standard DFA-based lexical analyzer, as a DFA can only be run at the starting state using the first character of an input string. This DFA instance is termed the "initial DFA." The other processing units in a multi-core processor are utilized by concurrently executing $N-1$ speculative scanners on the remaining substrings $P_2, P_3, ...P_N$. The processing is speculative as it is not possible to determine the start state for the L_{DFA}, except for P_1. As a result, we have added a transformation module to the PIXIMAL framework that can be applied to create a scanner, which can be applied to any of the substrings.

The DFA above is transformed into an NFA, L_{NFA} containing precisely the same state nodes, transitions, and final states as L_{DFA}. One significant change is made: each state node, with the exception of the error state, is marked as a start state. The parser built around this NFA reads each character of

input, traversing along all *execution paths*, one for each state S_i. If a given transition triggers an action (such as triggering a StartElement SAX event in the user code), that action is stored into an *action list* A_{S_i} for that execution path, since it cannot be triggered immediately.

There is a single *correct execution path* which is the path started in state S_k, the state that the L_{DFA} would have been in had it parsed the input up to the beginning of this input substring. S_k will be known when the DFA or NFA running on the input behind it is complete and, if it is an NFA, knows its own correct execution path. Once S_k is known, the actions in action list A_{S_k} can be triggered, after some minor fix-up to merge the parser state from the previous automaton and the first action in this automaton's action list. This is necessary because the NFA may have started in the middle of a token, or more complexly, in the middle of an XML tag, which contains several tokens: a tag name and zero or more attribute name/value pairs. This fix-up is minor and a function to the number of automata used, as opposed to the size of the input, so can be viewed as a $O(1)$ cost once the number of available computing cores is set.

B. Limitations of Micro-parallelization of XML

Our past work demonstrates that the level of speedup obtainable using micro-parallelization techniques is limited: other system resources, such as memory bandwidth become bottlenecks [21]. We present some new experimental results here to quantify the gains and limitations of the micro-parallelism approach.

1) Systems Used: We run our tests on a selection of four machine classes:

- 1× dual core – One desktop-class machine, which has a single 2.4Ghz Intel Core2 6600 with 2GB of ECC RAM, running Linux 2.6.24. The filesystem in use here is ext3fs.
- 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.

2) Memory Bandwidth: We examined the memory bandwidth limitations of systems in our clusters to determine if the PIXIMAL would be applicable. In these tests, we modeled the work being done to read the input from memory, as PIXIMAL does, directing multiple threads to read from different sections of a 100MB input XML file – a protein sequence database [22]. Figure 1 demonstrates that current systems do provide wide enough access to memory to facilitate the approach, but only to a certain extent. Once the number of cores being utilized goes beyond six, little gain is possible, if the computation requires to parse the input is small in comparison to the time required to access the bytes of the input in memory.

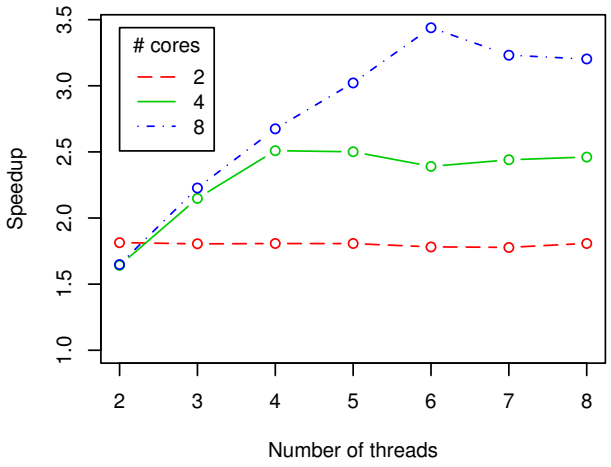

Fig. 1. The limitation on memory bandwidth when attempting to concurrently feed several cores different segments of a large (100MB) file. As the number of cores in a single system increases, memory bandwidth becomes a bottleneck.

3) Maximum Potential Speedup for PIXIMAL*:* We also examined the best-case scenarios for PIXIMAL speedup by running its automata components sequentially instead of concurrently. The time it takes for the longest parser to completely parse its input segment represents a lower bound on the time the concurrent version would take if all components were scheduled to run concurrently. The input is divided up to eight ways and processed by automata sequentially, with each automata allowed as much of any system resource it requests. We discuss some interesting results here.

The input used for the results described is a collection of XML documents containing SOAP-encoded arrays of Mesh Interface Objects (MIOs) which model three dimensional objects as a collection of points with integral X and Y values and a floating point Z value. The size of the array encoded is scaled from ten up to 50,000 elements long. Figure 2 shows the maximum, mean, and minimal potential speedup over that range of inputs.

Figures 3 and 4 investigate one data point in figure 2, namely the maximal value of the independent variable: 50,000 array elements. It is apparent that dividing the work further is not an efficient way to achieve speedup. Note that the low point in 3 when the thread count is 8 is not an indication of the trend, but rather an artifact of the splitting algorithm. If the input is split such that at least one of the NFAs cannot eliminate several dead states as mentioned above, then that automaton will become the processing bottleneck. The portion

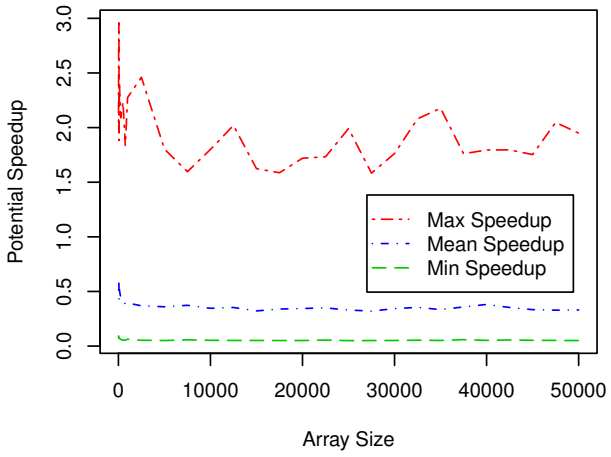

Fig. 2. The maximal, minimal, and mean potential speedup across a range of lengths, from 10 to 50,000, of arrays of *Mesh Interface Objects*.

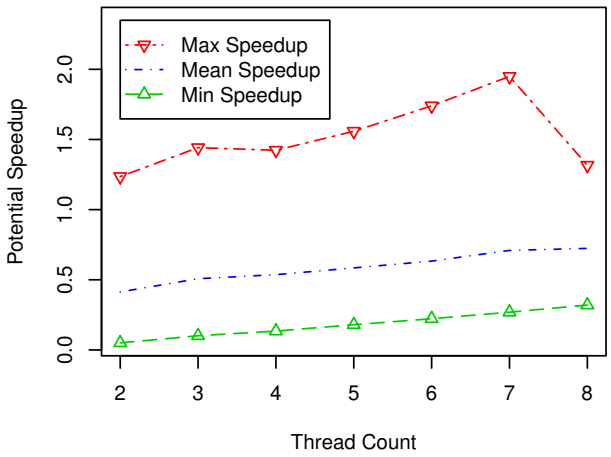

Fig. 3. Potential scalability for XML input encoding an array of *50,000 Mesh Interface Objects*. The number of threads available is the independent variable here. A slight speedup is possible by adding more threads for this class of input.

of the input not given to the DFA is always divided evenly among the NFAs which can lead to this situation.

Figure 4 re-presents the same data from figure 3. The split percent parameter determines how much of the input file is given to the DFA. For each split percent, a variety of thread counts is chosen, from two to eight. It is clear for inputs such as arrays of MIOs, selecting the division point is critical.

These tests represent the *best case* scenario for PIXIMAL, where each thread has uncontended access to all system resources because each component is run by itself. When the threads run concurrently, a number of system level bottlenecks become apparent, further reducing speedup. These limitations become more apparent as the input size increases, because each PIXIMAL thread requires $f(N) \in O(N)$ space to store its sequence of actions.

These results led us to the approach of securing additional resources for each thread by distributing the workload to a cluster of machines using the Hadoop implementation of

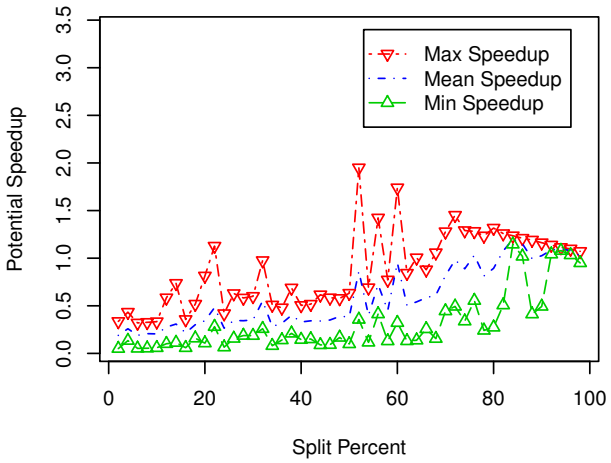

Fig. 4. Similar to figure 3, this graph presents the result on speedup of varying the split percent parameter when processing an encoded array of *50,000 Mesh Interface Objects*. Maximal and minimal speedups for each selection of split percent are shown. The range of values comes from varying the number of threads.

MapReduce.

IV. MACRO-PARALLEL XML PROCESSING WITH HADOOP

We applied the MapReduce model to the parsing of XML data sets, with a widely used web service application, Axis-Java, which is a widely used web services-based application toolkit. For example, it is included in the reference implementation of WSRF available with the Globus Toolkit. AxisJava is primarily a SOAP engine that allows for the creation of SOAP processors used in the parsing of XML dataset items as requests sent from a user or a process. The streaming requests are then generated into an application friendly content, which can subsequently be processed as the user sees fit. For our part, we delved into AxisJava's source code, studied its SOAP decoding module in an effort to integrate it into a Hadoop application. This application then allows us to perform on a given cluster of nodes, the same type of parsing that Axis-Java would normally perform while running on a standalone machine such as a desktop computer. It has been shown that AxisJava tends to perform rather poorly when it comes to parsing and processing arrays of type double [3], [9]. What makes this analysis interesting is that objects of type double are more likely to figure in scientific data samples bound for analysis and processing. This is one of the motivating factors that led us to analyze the problem at hand, and consequently devise a way to integrate the AxisJava parsing mechanism into a computing grid platform; thus allowing researchers and grid application programmers working with costly data types to use this framework for a significant performance improvement.

For our purpose, we adapted Hadoop's *splitter* implementation so that the XML dataset is effectively split on *data* element boundaries, and also, as to allow us to configure at run-time the chunk sizes of *double* objects fed to the nodes.

We devised a *processing barometer*, which allows us to determine the amount of CPU processing taking place on every parsed element. This is particularly useful because prior investigations and experiments performed on Hadoop have proven to show that clusters deprived of meaningful processing on their inputs, will tend to buckle under network and redundancy assurance latencies, enabling them to perform even worse than a standalone node. This is because without meaningful processing on parsed input values, a grid system in general, and here Hadoop in particular will suffer from its own operating cost; such costs being, network latency, initialization time, and redundancy checks (to determine node failure), which undoubtedly as a cumulus of factors will hinder performance. The observation can –and is meant to– serve as a barometer to scientists for evaluating how big a cluster one is to run, when certain expectations on particular types of performance directives are made. Also, and more trivially: how much performance different data sizes will yield, paired with definite amounts of processing on each of data item to be processed. As our experiments have shown, Hadoop performs increasingly differently for different processing work loads. Simply put, increasing how much processing is taking place on each input element, undoubtedly increases the performance of the cluster relative to an individual system, and yields interesting results. This is because in those very conditions, the parallelism of the grid is fully exploited at its utmost potential. We quantify these costs and thresholds to aid application programmers in allocating resources in accordance with their data size and shape.

Our experiments use the same systems described in section III-B1. The *master* system for Hadoop is the $1\times$ dual core desktop system and the *slave* systems are chosen from the collection of 64 $2\times$ dual core cluster systems.

V. Performance Results

Figure 5 shows how a scientist's time can be saved by off-loading processing work to even a 3 node cluster. The single node represents the time it takes to process the input file on a workstation-class (the $1\times$ dual core desktop system). Using our synthetic loading scenario, when the input grows above just about 400 kilobytes, it begins to be valuable to offload to a MapReduce cluster. Below that threshold, not enough processing takes place, the cluster is burdened with redundancy checks and network traffic for just small chunks of input. In this scenario, when computation is not sufficient enough to offset communication latencies due to the number of running computers, a single node, which minimally suffers from the same condition would perform better than a cluster of computers. However, as the input dataset size grows, the cluster can comfortably offset diverse latencies and overhead encountered with abundance of processing, which the single node cannot. This fact is illustrated by the cluster performance curve showing a stable tenure above 400KB, whereas the processing time of the single node skyrockets for increasingly large files.

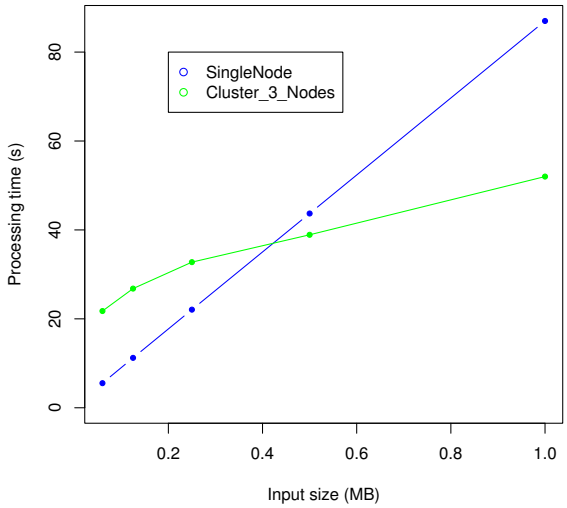

Fig. 5. Shows the performance of a 3 node cluster against a single node computer in processing differently sized arrays of doubles. Up to 400KB of data, the single node performs better than the Hadoop cluster simply because in the 3 node system every machine is competing for one single output file in HDFS, and suffering from latencies, while the input size isn't big enough to offset those factors.

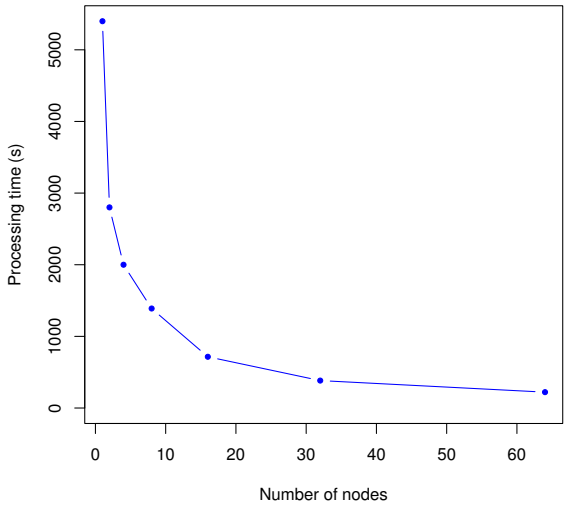

Fig. 6. Performance displayed by the cluster as the number of nodes is gradually increased on a 150 Megabyte dataset.

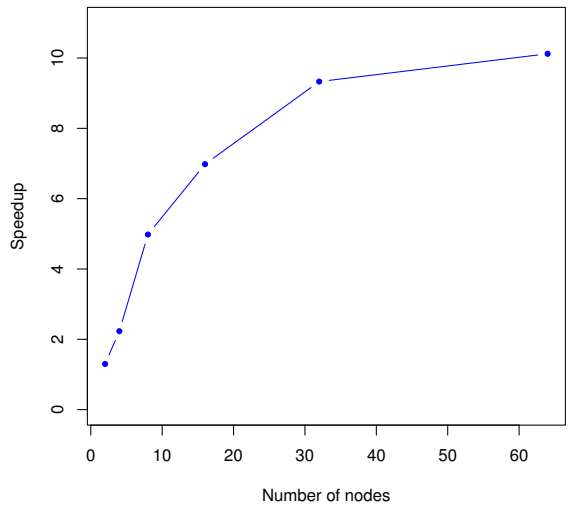

Fig. 7. Speedup achieved by the cluster over a single node computer on a 100 MB dataset. Much like the previous test, the speedup stabilizes over a certain threshold, in this case, over 32 nodes, even as the number of computing nodes is doubled.

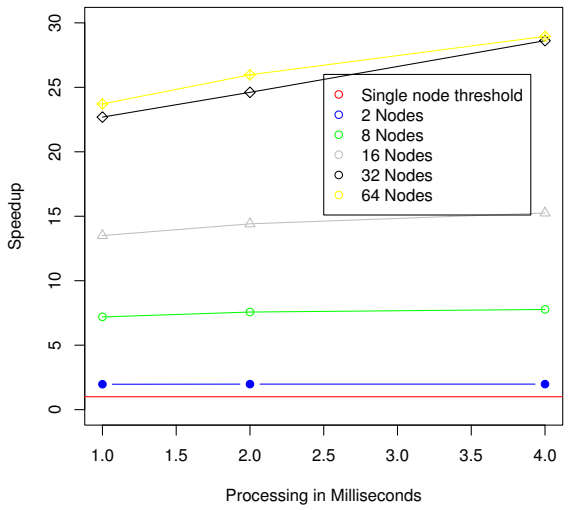

Fig. 8. Speedup achieved by the cluster over a single node computer while varying the amount of processing to be performed on every single input item. This is based on a 100 MB dataset run. The red line shows the performance threshold that a single node system generated. Much like the previous test, the speedup tends to stabilize over a certain threshold, in this case over 32 nodes.

Figure 6 displays the time it takes to process a 150MB input array of doubles as the number of nodes is gradually increased. Performance gains tail off between 16 and 32 node cluster sizes, as communication overhead and the serialized portions of the process dominate the parallelizable portions. The work of shipping data around the cluster and managing the nodes begins to dominate the actual work-load, because the optimum number of cluster machines for the job at hand has been exceeded. The dataset here as in the previous experiment is that of a an array of type double. The file size is 150MB, which roughly corresponds to 12.6 million data items. The graph shows a clear improvement with node increase up to 24 nodes, and then shows a flattening of the performance curve. Beyond 24 nodes, the necessary processing power is at its maximum. The addition of supplementary nodes introduces unnecessary communication overhead, which slowly starts to hamper the performance itself, thus causing a slowing of the curve. This shows that too many nodes can negatively affect performance, just as too few nodes or lack of processing can.

Figure 7 presents data similar to those in figure 6, using speedup (computed as $\frac{T_1}{T_p}$) and a 100MB input file. Performance tails off when the cluster size reaches 32-nodes. In this case, T_1 is computed by running the test against a Hadoop cluster with a single slave system.

Figure 8 presents data representing the performance of the cluster as the required processing time on each one of its individual input items is gradually increased. Using speedup (computed as $\frac{T_1}{T_p}$), a 100MB input file, different processing times on each double parsed from the input file, the individual cluster setups run better with more work to do on each input item. This goes to corroborate the fact that the processing load for a job must trump the different overhead considerations of that very job, for the use of the cluster to be fully efficient. Speedup is computed against a single slave system, and speedup is also shown between increasingly sized clusters. This graph clearly illustrates that as processing is cranked up, regardless of the size the cluster at hand sports, the system performs much better, even outdoing its previous runs. On the same logical path, bigger clusters perform doubly better than smaller ones in the face of processing increases. This trend of improved performance with increased workload plateaus off after 32 nodes for the 100 MB data set.

VI. Conclusions

- As evinced by Amdahl's law, the speedup is limited by the sequential fraction of the program. The sequential aspect for XML data processing includes access to main memory for shared data structures, resolving namespaces that may have dependencies, and updates of data structures to keep track of the automata that need to be stored as well as the ones that need to be discarded. For XML datasets close to 100MB in size, as the number of processing cores in a node is increased beyond 6, memory bandwidth becomes a limiting factor in data processing when dealing with single computing nodes.
- The PIXIMAL framework can be used to determine the most optimal way to split an XML data input file to obtain best possible speedup. Performance results for commonly used data structures, such as MIOs, indicate that application programmers need to carefully choose the

split percentage and number of processing threads for the target computational nodes. Naively dividing the input among the cores may lead to a performance slowdown. Our results quantify the optimal thread count and number of cores to be used for each data size in order to obtain efficient processing. On an 8 core machine, the best speedup of 2.0 is possible when the workload is divided among 6 or 7 threads.

- Micro-parallelization techniques have limited efficacy within a single computing node due to the effect of shared resources such as memory and I/O channels. When these limitations become the bottleneck, application developers should not hesitate to examine existing macro-parallelization techniques like MapReduce as implemented by tools such as Hadoop.
- When using a distributed approach to try to find performance improvements, it is important to consider whether computation or access to I/O is the bottleneck. A MapReduce approach can be indicated in either case, because it facilitates the utilization of many machines' CPU and disk resources. On the other hand, there is a startup cost involved in launching a Hadoop job. There is also a cost associated with insuring that nodes are harmoniously performing the given processing. It is then critical to examine whether these diverse costs are higher than the benefit resulting from the use of the clusters' resources versus a single node in certain cases. It is also critical to recognize before hand, not so much the size of the data to be processed, but rather the intensity of the processing to be taking place.

VII. Future Work

For applications that require processing of terabyte size datasets, mass storage is more likely to be arranged in higher performance configurations such as RAID, NAS, and SAN. These configurations are likely to efficiently feed multiple data streams to concurrent threads. We plan to study the benefits and limitations of our parallelization approach when applied to such cases.

In future work, we plan to also analyze the effect of disk I/O for our micro- and macro-level optimizations schemes. We will quantify the gains that PIXIMAL can achieve with pre-fetching and piped implementation techniques. We will explore the effect of operating system-level caching on the processing of large datasets that are processed repeatedly. We will also develop algorithms for optimal layouts of DFA tables in memory to efficiently process frequently occurring transitions.

We plan to quantify the threshold points for our adapted Hadoop implementation for a wider range of grid application datasets. Apart from size of datasets, we will also identify the threshold points for different network, I/O, memory, and processor configurations that are available in widely used grid infrastructures.

References

[1] 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*. Washington, DC, USA: IEEE Computer Society, 2003, p. 33.

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

[3] 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*. Washington, DC, USA: IEEE Computer Society, 2005, p. 19.

[4] 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*. New York, NY, USA: ACM Press, 2006, p. 121.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[6] Apache Hadoop. [Online]. Available: http://hadoop.apache.org

[7] W. Zhang and R. A. 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)*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 197–204.

[8] 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*. Washington, DC, USA: IEEE Computer Society, 2005, p. 21.

[9] 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*. Washington, DC, USA: IEEE Computer Society, 2002, p. 246.

[10] 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 Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2000, p. 61.

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

[12] J. Zhang, "Project Homepage of VTD-XML," 2007, http://vtd-xml.sourceforge.net/.

[13] D. Veillard, "The XML C parser and toolkit of Gnome," 2006, http://xmlsoft.org/.

[14] J. Clark, "Expat is an XML parser library written in C," http://expat.sourceforge.net/.

[15] Trolltech, "API Documentation for QtXml Module," 2007, http://doc.trolltech.com/4.2/trolltech.html.

[16] Y. Oren, "Piccolo is a small, extremely fast XML parser for Java," 2006, http://piccolo.sourceforge.net/.

[17] Xerces-J, "Xerces2 Java Parser 2.9.0 Release," 2006, http://xerces.apache.org/xerces2-j/.

[18] M. R. Head and M. Govindaraju, "Parallel Processing of Large-Scale XML-Based Application Documents on Multi-Core Architectures with Piximl," in *IEEE Fourth International Conference on eScience*, December 2008, pp. 261–268.

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

[20] Y. Pan, Y. Zhang, K. Chiu, and W. Lu, "Parallel XML Parsing Using Meta-DFAs," in *IEEE Third International Conference on eScience and Grid Computing*, December 2007, pp. 237–244.

[21] M. R. Head and M. Govindaraju, "Performance Enhancement with Speculative Execution based Parallelism for Processing Large-scale XML-based Application Data," in *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2009, pp. 21–30.

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