

BXSA for Fast Processing of Scientific Data

Tharaka Devadithya¹, Zongde Liu², Nayef Abu-Ghazaleh², Wei Lu¹, Kenneth Chiu², Stephane Ethier³
Randall Bramley¹

¹Computer Science Department, Indiana University

²Department of Computer Science, State University of New York (SUNY) at Binghamton

³Princeton Plasma Physics Laboratory, Princeton, New Jersey

Keywords: binary xml, xml, scientific computing, web services, high performance computing

Abstract

XML has become the de facto standard for semi-structured data across a variety of domains. XML is generally considered to be slow for scientific data and therefore only used for control information. However, this approach prevents scientific data from being “first class members” in XML, especially in a web services framework. It also puts a burden on application developers as they have learn two type systems. XML, however, can be represented via more efficient encodings, often known as “binary XML”, which efficiently handles large data in XML format. In this paper we extend our previous work on Binary XML for Scientific Applications (BXSA) (1) applying BXSA to the Gyrokinetic Toroidal Code fusion application, and showing that performance is better than HDF5 in our test cases; (2) demonstrating an interoperable Java implementation that is faster than Xerces and Sun’s Fast Infoset on common document types; and (3) showing that BXSA is also applicable to business data in addition to scientific data by evaluating its performance on a variety of XML documents against libxml2 and expat. These results show that BXSA is suitable for both business and scientific data.

1 INTRODUCTION

XML is a flexible, hierarchical and self describing data format used to represent tree-based structures. Due to its wide adaptability, XML is becoming the standard for communication, archiving and saving configurations.

Web services have become the dominant framework for building large scale Service Oriented Architectures (SOA). They have garnered much attention from both industry and academia, leading to wide-spread activity in research, specification, and implementation. Though not strictly required by web services standards, XML is also the de facto standard wire format for web services.

However, in many cases XML’s processing cost prevents XML being used to represent large data sets, especially scientific data. Thus, the conventional wisdom is that web services are for *control*, while *data* must be stored, managed, and transmitted using binary, non-XML-based formats. To maintain this separation between control and data, distributed applications such as LEAD [4] typically handle data in a non-web-services-based, ancillary framework consisting of binary

data formats, various mechanisms from transmitting files in these formats, and additional libraries and APIs for using these formats. The result is essentially two systems that parallel each other: a web services infrastructure for the control of applications, and a bespoke infrastructure for large data. Even though, using WSDL, a service can technically be bound to a non-XML protocol, such a binding would still require two systems, one for handling the usual binding to XML-based standards, and another for handling the non-XML-based bindings.

This approach has several disadvantages. Developers must bear the burden of using and maintaining a separate code infrastructure for the data along with the additional mental cost of learning two different sets of concepts, terminology, and software. Binary data is usually sent as an attachment of some kind or as a URL referring to the data, which is then used to retrieve the data. SOAP Message Transmission Optimization Mechanism (MTOM) [21] standard promotes yet another technique.

As a solution, we have developed a Binary XML for Scientific Applications (BXSA). Our focus on scientific computing, however, does not imply that we support two separate standards, one for business computing and one for scientific computing. In fact, our experiments, described later in this paper, show that BXSA is fast for both scientific business computing. The active participation of the scientific computing community can ensure that the resulting standard will satisfy the requirements of scientific computing as well as business computing.

In this paper, we extend our previous work [5][11] with a number of contributions. We modify the Gyrokinetic Toroidal Code (GTC) fusion simulation code to use BXSA, thus showing BXSA’s applicability to real scientific applications, and present the results from multiple parallel runs. We show that BXSA is faster than HDF5, the main format used in the GTC application. We have also implemented a Java version, interoperable with the C++ implementation, and show that the Java version is faster than DOM4J [1], Sun’s Fast Infoset, and Xerces for our test cases. Finally, we show with additional tests that BXSA is faster than textual XML even for XML that is not dominated by arrays of doubles. These results show that BXSA is suitable for a wide variety of applications, not just for scientific computing.

In Section 2, we consider the motivation for BXSA. Section 3 describes the notion of binary XML with related work in Section 4. Section 5 gives the data model used in BXSA.

Sections 6 and 7 describes the Java implementation and BXSA's application to GTC fusion code, respectively. Section 8 gives the performance results.

2 MOTIVATION

XML's poor performance, prevents it from being used to transmit and store large data, especially scientific data. This poor performance arises from a number of causes:

1. XML is verbose. Both the start tag and the end tag contain the element name, which improves readability but is redundant. Also, the ASCII representation of numerical data is usually longer than the equivalent two's complement or IEEE 754 representation.
2. The syntax of XML encourages, or even requires, inefficient parsing. For example, processing an element usually requires multiple passes over the same information. The parser first scans the data to find the end tag. The information is then copied into some kind of data structure to pass to the application. Finally, the application must make another pass over the data as it executes. Namespace processing is another area of XML parsing that requires backtracking or multiple passes over the input stream.
3. Most significantly for scientific applications, the computational costs of converting floating-point numbers from ASCII to a native machine representation such as IEEE 754 is unacceptable for many scientific applications [6].

The relative importance of these three sources of inefficiencies varies according to the context. For business data, parsing complexity might dominate. For scientific data, the ASCII-to-machine conversions are the predominant bottleneck. But even if these bottlenecks were removed, the verbosity of XML would become an issue for high-performance, high-bandwidth applications. These applications demand the very highest bandwidth that can currently be provided over WANs, and excessive verbosity would be unacceptable.

3 BINARY XML

W3C antipated the need for an abstract XML data model and has defined a specification for XML Information Set (*infoset*) [20]. The XML infoset data model closely matches the information contained within a stand-alone XML document.

Since data models are abstractions, a single data model can have multiple serializations. Applications can then be written to an API representing the data model, and not the actual encoding of the data model. The encoding can then be changed without requiring the application to be rewritten.

Being able to serialize large amount of binary data as XML would obviate the need to handle data in a separate framework. The data would be just another element in the XML (or binary SOAP message). Most of the features available within XML, such as XML Schema, are now available in this serialization for binary data. Furthermore, the API is independent of the underlying serialization,

We do not see binary XML as a replacement for data formats such as netCDF [13] or HDF5 [7], but as a lower layer on top of which these formats can be built on. This layering can already be seen with textual XML where specialized XML languages such as MathML [18] and CML [12] are based on. It might, however, be usefull to clone the existing APIs of these data formats (e.g., netCDF API), in order to preserve compatibility with existing applications.

It can be argued that switching to binary XML from textual XML will affect interoperability. We do not, however, advocate switching to binary XML from textual XML, but rather switching (when appropriate) to binary XML from specialized data formats. Arguably, we are thus increasing the overall level of standardization and interoperability, rather than lowering it. Thus, we are addressing applications where the decision is not between binary XML and textual XML, but rather large data applications where overriding performance concerns leave textual XML completely out of consideration.

Though our initial driver for BXSA was scientific computing, we have also optimized it for business computing. We do not believe that the requirements of scientific computing and business computing are fundamentally in conflict.

4 RELATED WORK IN BINARY XML

We discussed some of the proposed formats for Binary XML is our previous work [5]. Most of those formats did not focus on large arrays of doubles. Here we briefly describe some additional formats.

Fast Infoset. The Fast Infoset [17] is an effort by Sun to develop a fast serialization for XML. Unfortunately, since it is being submitted as an ISO specification, it is currently not publicly available for review and comment. It is based on ASN.1 [8], which is commonly used in the telecommunications industry. Tests of the only publicly available implementation of Fast Infoset currently show that it is generally slower than BXSA (Section 8).

WBXML. WBXML [19] is a standard from W3C that focused on a compressed XML for wireless devices. It did not use any machine representation for numbers, and is thus not suitable for scientific computing, but it was one of the early binary XML efforts.

XBIS. XBIS [16] is an encoding format that emphasizes full transcodability to and from textual XML. However, this transcodability comes at the cost of maintaining numerical data as text. The resulting performance costs make XBIS unsuitable for scientific data.

XMill. XMill [9] focuses on the compression of textual XML, and in fact uses standard text compression techniques. These techniques are usually computationally intensive, and are thus generally not suitable for scientific computing. Also numerical data is presented as text rather than in native format.

5 EXTENDED XDM AND BXSA

XML is designed as a textual format for highly structured data. Several models have been defined to represent the log-

ical structure of the XML document such as the XML Information Set (infoset) [20], the post-schema validation infoset, and the XQuery and XPath Data Model (XDM) [22]. Those data models essentially serve as an abstraction layer for the information which is stored in the XML documents. The actual application layer is thus protected from the lower-level serialization details. All fundamental building blocks of web service architectures, such as XPath, XSLT, SOAP and so on, rely on this abstraction layer to describe the data they are processing.

We selected XDM rather than the more well-known XML infoset as the core XML data model in our implementation, because beyond the information contained in the XML infoset, XDM also supports typed atomic values and XML Schema type information in the data model itself. This type information is crucial for coding efficiency. For instance, the typed atomic value allows our API to represent numbers in their native machine form, rather than as a character string as required in an XML infoset; therefore obviating the expensive conversion between machine form and ASCII. Based on our experience, this small improvement is essential to the high performance encoding of scientific data in XML.

However XDM is still awkward for handling large arrays of numbers, which dominate large data applications. XDM has no inherent notion of arrays, and thus XDM requires that each element of the array be a separate node. This is prohibitively costly for large arrays; therefore, we extended XDM, which we call *bXDM*, by adding the concept of an array as a single node.

bXDM includes the seven node types defined in XDM (Document, Element, Attribute, Namespace, PI, Text, and Comment), but also further refines the Element node into two subtypes: *LeafElement* and *ArrayElement*. The *LeafElement* represents elements of a primitive, atomic data type, such as integers and floating-point numbers. So the atomic data value of the *LeafElement* can be saved in the machine native format without any overhead. The *ArrayElement* is designed to represent an array of identically-typed, atomic elements as a single node in the model. Internally the data value of the *ArrayElement* is stored as a packed array of primitive types. This allows the *ArrayElement* to efficiently represent arrays and matrices as well as raw octet streams.

5.1 Current Version

The initial version of BXSA was based on the XML infoset, but the current version was modified to be based on *bXDM*. Other than that, the data types and frame formats are the same as described in [5].

The *bXDM* model allows our API to represent numbers in their native, machine form, rather than as character strings.

6 JAVA IMPLEMENTATION

Java BXSA adopts XDM as its in-memory data structure but provides DOM interfaces for manipulation. As in the C++ version, we extended the Element node of XDM to add *ArrayElement* and *LeafElement*. *LeafElement* and *ArrayElement*

represent data values in Java's internal format and thus eliminate data conversion from character strings to numeric values. The framework of Java BXSA is extensible; user-defined data types can be easily added to the framework.

The in-memory XDM data structure can be serialized to the BXSA format or to textual XML. The binary format is interoperable with its C++ version and Java BXSA supports serialization to either big endian or little endian. As the BXSA frame structure is self-contained, a frame contains all necessary information for the parsing of data values, namespaces, and attributes.

Java BXSA consumes much less memory to construct in-memory data structure than Xerces DOM and DOM4J. The memory used by Java BXSA is 75% of DOM4J and 50% of Xerces DOM. The parsing performance of Java BXSA outperforms DOM4J and Xerces DOM parsing. It also outperforms Xerces SAX parsing, even though SAX parsing does not maintain a DOM tree. In our test, DOM4J and Xerces DOM did not convert the data values from their character representation to the numeric values. However, Java BXSA maintains numeric representation of data values in all cases, so computation on data values do not need any data conversion. Thus, in a realistic application scenario, BXSA will further outperform the others due to this savings.

Java BXSA can record the size information of each frame. Based on the size information, complex strategies can be employed for binary XML data processing, such as on-demand, lazy parsing. These characteristics will make Java BXSA the ideal choice for data intensive and computation intensive applications.

We also note that Java BXSA is fully interoperable with C++ BXSA.

7 APPLICATION TO THE GYROKINETIC TOROIDAL CODE (GTC)

In order to measure the performance of BXSA when used in a real-life scientific application, we integrated BXSA with the Gyrokinetic Toroidal Code (GTC) [10], a simulation application from the fusion energy research community.

GTC is a massively parallel particle-in-cell (PIC) code used for studying microturbulence in magnetically confined fusion plasmas. Microturbulence is believed to be the main mechanism by which the hot ionized gas in the core of the fusion device loses energy to the much cooler edge plasma, thereby making it difficult to maintain the reaction. Since energy losses translate directly to higher costs in the operation of fusion devices, this research is of utmost importance for understanding the current machines and for the design of future ones. GTC is the flagship code of the DOE SciDAC Center for Gyrokinetic Particle Simulation of Turbulent Transport in Burning Plasmas. It has been extensively benchmarked on most of the HPC platforms currently available, including the 5,120-processor Earth Simulator in Japan [14][15], the 5,212-processor CRAY XT3 at the Oak Ridge National Laboratory, and the 40,960-processor Blue Gene/L at IBM Watson. GTC has been shown to run efficiently on all platforms and to scale

to more than 16,000 processors [3].

As a PIC code, GTC has both grid and particle data. The kinetic equation for the plasma is solved by following the particles in time. However, the particles interact with each other via a grid on which the electromagnetic field produced by the particles is solved. This reduces the very expensive N^2 binary interaction calculation to a much more efficient order N calculation. It also removes some undesirable small-scale features in the field that can potentially mask the important physics at late times. The current implementation of GTC uses parallel HDF5 to output the 3D grid data and tracked particle data. Most production simulations write out only the grid data, however. The number of particles in a simulation can vary from tens of million up to a few tens of billion, and the amount of data written to disk quickly becomes a challenge as the number of particles being tracked is increased to several million and with a frequency of outputs of only a few time steps. Most of the diagnostics involving the particles are done within the simulation so particle tracking is mainly for visualization purposes at this point. In this study, we thus focus on the output of grid data and compare the performance of BXSA and HDF5. Also, since the performance of parallel I/O depends strongly on the performance of the underlying parallel filesystem and that a fully parallel BXSA has yet to be implemented, we limit our comparison to writing the data to its own independent file. Each one of these files can be a few hundred kBytes to about 10 Mbytes depending on the size of the fusion device being simulated. These files are generated at regular step intervals during the run.

8 PERFORMANCE RESULTS

We tested three different XML documents, which we believe cover a range of use cases. We tested an array of doubles (`array`), a molecule described in XML obtained from the Protein Databank (`1kzk`) that contains number of elements with atomic types, and a document that heavily uses namespaces and attributes (`ns_att`). Samples of the documents are available at <http://www.cs.binghamton.edu/~kchiu/bxsa-tests/>.

We were more interested in the encoding time rather than the communication time. This was because the communication time depends mainly on the size of the encoded data and BXSA encoding results in the most compact form compared to textual XML or base64 encoding of other binary formats. Therefore, the communication with BXSA format was always going to be faster.

Since it can be argued that we gain performance by avoiding the ASCII-to-double conversion, we tested against other parsers both with and without converting numbers from their ASCII representations to their native representations. Since the application use cases we are interested in will treat numbers numerically, we believe that this is a reasonable comparison.

The tests were conducted on a dual AMD Opteron systems with 16GB of memory, running Gentoo Linux. The XML documents were read from a local file system. Timings are

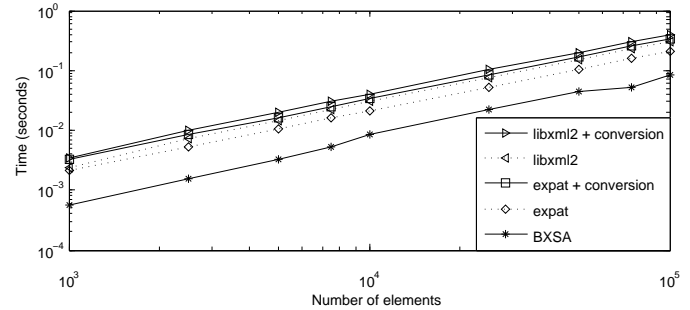


Figure 1: [C++ array] Deserializing an array of doubles.

from a warm file cache.

8.1 C++

We compared the C++ version’s performance to libxml2 and expat. For libxml2, we used its DOM interface. Code was compiled using gcc 3.4 with `-O3` optimization flags. The results are shown in Figures 1 (`array`), 2 (`1kzk`), and 3 (`ns_att`). In each, the line with “conversion” includes a numeric conversion from ASCII. Any application that needed the numeric values of the numbers will need to perform this conversion.

As can be seen from the graphs, BXSA outperforms libxml2 and expat, even when libxml2 and expat do not perform any data conversions. Furthermore, expat is not building a DOM-like tree, and, therefore, is doing significantly less work. Compared against libxml2, which is building a DOM-like tree, BXSA is faster by several factors.

For arrays of doubles, BXSA is almost an order of magnitude faster. This shows the significant performance advantage of the extended XDM data model over XML Infoset for scientific data. For the `1kzk` document, BXSA is still several times faster. This is mainly due to the use of `LeafElement` for the elements with atomic data types as well as the counters within a BXSA frame, which allow for a more efficient memory management while constructing a DOM tree. The `ns_att` document stresses attributes and namespaces. Here, the advantages of BXSA are somewhat less, but it still outperforms the others due to the use of namespace indexes rather than textual namespace prefixes. Overall, these results show that BXSA yields significant performance gains compared to textual XML for a variety of XML documents.

8.2 Java

we compared Java BXSA’s performance to Xerces DOM and SAX, Fast Infoset DOM and SAX, and DOM4J (with included parser). The Fast Infoset implementation was from the open source, Fast Infoset Project [2]. We used JDK1.4, and BEA Weblogic JRockit JVM, with options `-Xms512m -Xmx512m`. The results are shown in Figures 4 (`array`) and 5 (`1kzk`).

As can be seen from the graphs, Java BXSA significantly outperforms Fast Infoset when used for arrays of doubles.

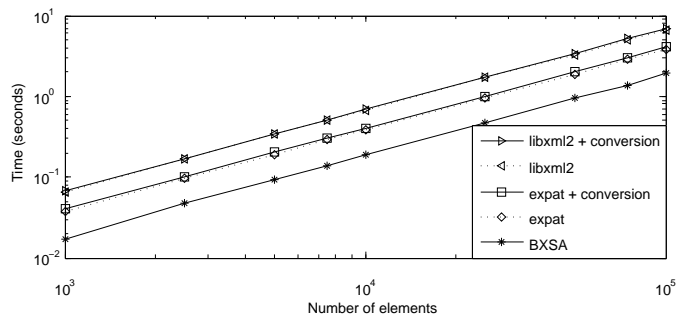


Figure 2: [C++ 1kzk] Deserializing an XML document representing a molecule from the Protein Databank.

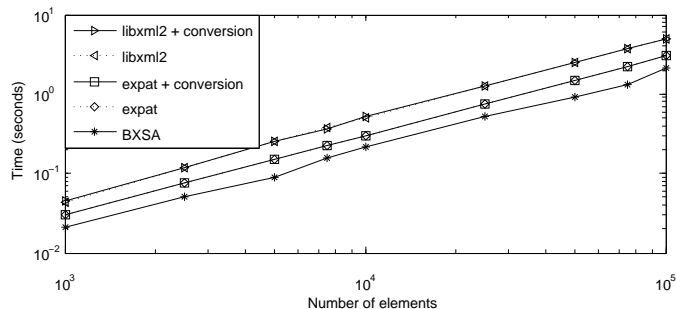


Figure 3: [C++ ns_att] Deserializing a test document consisting of many attributes and namespaces.

This again shows the benefit of using bXDM, which represents arrays in a single node. To illustrate the cost of converting ASCII to double, for this test we added this cost to the Fast Infotet parsing with SAX interface. From the diagram we can see Fast Infotet SAX with the additional type conversion performs significantly worse in this test. Note that Fast Infotet does include additional extensions to represent doubles in a native format, but these were not yet available at the time of testing.

For the 1kzk document, BXSA's performance was matched only by Fast Infotet in SAX mode. Since this mode does not build a tree, while BXSA does, it is not a direct comparison. When we use Fast Infotet DOM, which does build a tree, then BXSA performs somewhat better. DOM4J has lower memory usage than Xerces but also lower parsing performance. We believe that with further optimizations targeted toward structurally complex XML, Java BXSA can be further improved.

8.3 GTC Performance Comparisons

We measured how BXSA performs with data output from the GTC Fusion simulations, in relation to HDF5. The GTC code is written in Fortran and is modularized in such a way that one output routine can be easily replaced with another. Currently the GTC application has three output routines: serial HDF, parallel HDF, and native Fortran formats. We have developed another routine that outputs simulation results in

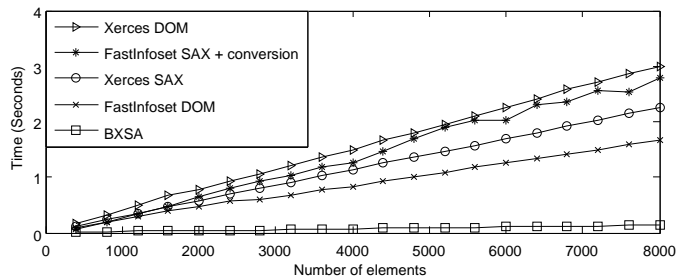


Figure 4: [Java array] Deserializing an array of doubles.

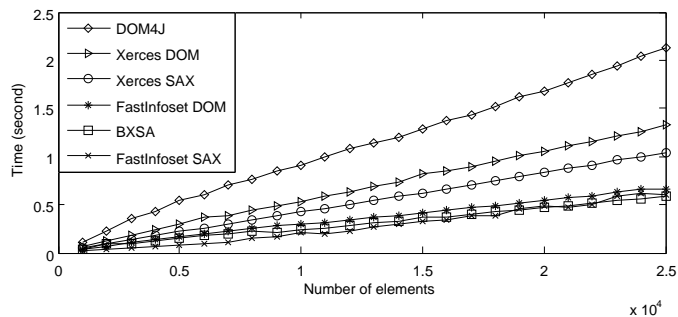


Figure 5: [Java 1kzk] Deserializing an XML document representing a molecule from the Protein Databank.

BXSA format.

As this was a parallel application, the tests were conducted on a cluster that contains 128 Dual AMD Opteron nodes with 4 GB of memory running Redhat Enterprise Linux. We used 64 nodes for our tests, giving us 128 CPUs. The PGI 6.0 Fortran compiler was used to compile the GTC code along with the BXSA output routine, while the BXSA libraries were compiled with gcc 3.4.4.

The program outputs data files at regular step intervals during the simulation. We varied the frequency of output by allowing the output to occur after every 1, 2, 4, 8, and 16 steps, respectively. Several files are being output by different processes during a call to the output routines. The majority of output data consists large arrays of floating point arrays. However, there are a few small arrays and scalar values too. The data files were written to the local disk of the cluster.

Figure 6 shows the output times for BXSA and HDF for different frequencies of output. The results indicate that BXSA is approximately four times faster than HDF for these outputs.

Our previous studies [5] indicated BXSA was comparable to HDF5 for large arrays. However, the fusion application also writes a number of smaller arrays, which BXSA handles in an efficient manner. According to the documentation, the `h5dwrite()` function in HDF5 writes a dataset from application memory to a file, irrespective of the size of the dataset. In contrast, BXSA maintains a buffer for serializing output data and it will write the buffer content to a file only when it is full. This ensures that smaller arrays are not written out

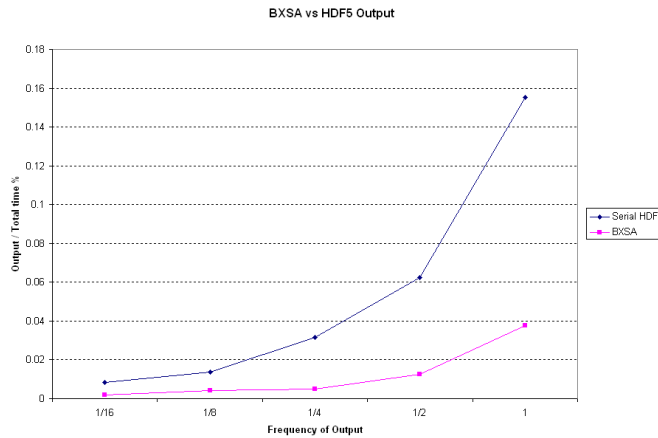


Figure 6: Performance of HDF5 versus BXSA for the output of the GTC fusion application. The x -axis is the frequency of output in time steps. The y -axis is the percentage of time spent doing output relative to the total time.

directly to the file with each call to serialize the array. This minimizes the number of system calls and therefore results in improved performance for BXSA.

9 CONCLUSION

We have presented BXSA, a Binary XML for scientific applications. In this paper, we have extended our previous work in a number of respects. We have applied BXSA to the Gyrokinetic Toroidal Code fusion application, and shown that performance is better than HDF5 in our test cases. We have demonstrated a Java implementation interoperable with the C++ implementation, and shown that it is faster than Xerces and Sun's Fast Infoset on common XML document types. Also, we tested BXSA against a wider variety of XML document types, which demonstrates that BXSA has overall performance that benefits business data as well as scientific data.

While most other binary XML formats are not designed to handle large arrays of numbers, BXSA efficiently handles them, mainly due to its bXDM model.

Our results challenge the prevailing practice in distributed scientific applications that data should be handled separately from control. This separation of data from control is often based on the conclusion that XML is simply too inefficient for scientific data, which typically consists of large arrays of floating-point numbers.

We have demonstrated that large amounts of scientific data can be sent more efficiently using BXSA than even standard binary formats such as HDF5. Handling data and control information with a single type system relieves the burden of application developers to learn two type systems.

References

- [1] dom4j: the flexible XML framework for Java. <http://www.dom4j.org>.
- [2] Fast infoset project. <https://fi.dev.java.net/>.

- [3] Scidac Review: The GTC code scales new heights. <http://www.scidacreview.org/0601/html/news4.html>.
- [4] Linked Environments for Atmospheric Discovery project web page. <http://lead.ou.edu/>, 2005.
- [5] K. Chiu, T. Devadithya, W. Lu, and A. Slominski. A Binary XML for Scientific Applications. In *International Conference on e-Science and Grid Computing*, 2005.
- [6] 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 HPDC-11 20002 (HPDC'02)*, page 246. IEEE Computer Society, 2002.
- [7] NCSA - HDF5. <http://hdf.ncsa.uiuc.edu/HDF5/>, 2004.
- [8] ITU-T. Abstract syntax notation one (asn.1). <http://asn1.elibel.tm.fr/en/standards/index.htm#asn1>, 2002. ITU-T Rec. X.680 (2002) | ISO/IEC 8824-1:2002.
- [9] H. Liefke and D. Suciu. Xmill: an efficient compressor for xml data. In *Proceedings of SIGMOD*, number 153-164, 2000.
- [10] Z. Lin, T. S. Hahm, W. W. Lee, W. M. Tang, and R. B. White. Turbulent transport reduction by zonal flows: Massively parallel simulations. *Science*, 281:1835–1837, Sep 1998.
- [11] W. Lu, K. Chiu, and D. Gannon. Building generic soap framework over binary xml for scientific applications. In *The 15th IEEE International Symposium on High Performance Distributed Computing (HPDC-15)*, 2006.
- [12] P. Murray-Rust and H. S. Rzepa. Chemical markup language. <http://www.xml-cml.org/>.
- [13] Unidata - NetCDF. <http://my.unidata.ucar.edu/content/software/netcdf/index.html>, 2005.
- [14] L. Oliker, A. Canning, J. Carter, J. Shalf, and S. Ethier. Scientific computations on modern parallel vector systems. In *SC04: High performance computing, networking, and storage conference*, Nov 2004.
- [15] L. Oliker et al. Leading Computational Methods on Scalar and Vector HEC Platforms. In *SC05*, Nov 2005.
- [16] D. M. Sosnoski. XBIS XML Information Set Encoding. <http://xbis.sourceforge.net/>, 2004.
- [17] Sun. Fast infoset. <http://asn1.elibel.tm.fr/xml/finf.htm>.
- [18] W3C. Mathematical markup language. <http://www.w3.org/Math/>.
- [19] W3C. Wap binary xml content format. <http://www.w3.org/TR/wbxml/>.
- [20] W3C. Xml information set (second edition). <http://www.w3.org/TR/xml-infoset/>, 2003.
- [21] W3C. SOAP message transmission optimization mechanism. <http://www.w3.org/TR/soap12-mtom/>, 2005.
- [22] W3C. Xquery 1.0 and xpath 2.0 data model (xdm). <http://www.w3.org/TR/xpath-datamodel/>, 2005.