Investigating the Limits of SOAP Performance for Scientific Computing

Kenneth Chiu, Madhusudhan Govindaraju, Randall Bramley
Department of Computer Science
Bloomington, Indiana University
chiuk@cs.indiana.edu, mgovinda@cs.indiana.edu, bramley@cs.indiana.edu

Abstract

The growing synergy between Web Services and Grid-based technologies [7] will potentially enable profound, dynamic interactions between scientific applications dispersed in geographic, institutional, and conceptual space. Such deep interoperability requires the simplicity, robustness, and extensibility for which SOAP [4, 3] was conceived, thus making it a natural lingua franca. Concomitant with these advantages, however, is a degree of inefficiency that may limit the applicability of SOAP to some situations. In this paper, we investigate the limitations of SOAP for high-performance scientific computing. We analyze the processing of SOAP messages, and identify the issues of each stage. We present a high-performance SOAP implementation and a schema-specific parser based on the results of our investigation. After our SOAP optimizations are implemented, the most significant bottleneck is ASCII/double conversion. Instead of handling this using extensions to SOAP, we recommend a multiprotocol approach that uses SOAP to negotiate faster binary protocols between messaging participants.

1 Introduction

The growing synergy between Web Services and Grid computing potentially can make practical distributed computing based on independently written software components for scientific and engineering computing. This potential depends in part on interoperability that is semantically deep but syntactically shallow. That is, the system must be loosely-coupled, but when coupling does occur, it must be at a deeper level than that typified by applications such as SETI@Home. The SOAP protocol was conceived expressly to support such interoperability in-the-large. Designed using principles learned from HTTP and HTML, it facilitates interdependent interactions between otherwise independent entities. SOAP is also the standard binding for the the emerging Web Services Description Language [6]. SOAP easily achieves high interoperability by requiring that its messages be in the Extensible Markup Language (XML) [17], which has been gaining acceptance as a canonical data representation. HTTP is a ubiquitous network protocol used extensively over the Internet. SOAP does not mandate an underlying transport protocol, but HTTP has emerged as the most widely used one for SOAP. Since SOAP can combine the strengths of XML and HTTP, it is an attractive candidate for Grid communication.

A common trade-off in computing is between the needs of universality and high performance. The qualities of SOAP that make it universally usable tend to work against high performance communications. In particular, XML specifies a primarily ASCII format. A characteristic that distinguishes scientific and engineering components is the need to frequently exchange large arrays of floating point numbers, with full accuracy to assure reproducibility. Scientific computing also demands the full range of capabilities that industrial computing does: reliable transfer in distributed heterogeneous environments, parallel programs sending large, complex and rapidly changing data objects or self-contained modules sending events to steer other modules, and complex run-time systems designed for heterogeneous environments with dynamically varying loads, multiple communication protocols, and differing Quality of Service (QoS) requirements.

A Web Services approach to scientific components would allow users to combine the strengths of different systems by developing applications that have components from several of them. Since these systems can be connected via more than one protocol, a common denominator protocol is needed to negotiate more specialized protocols. This common denominator protocol must provide reliability, robustness, readability, ease of use, seamless integration with existing computational code and interoperability. In earlier work [9] we showed that SOAP can be used to meet this criteria.

This paper examines the limits of SOAP performance for scientific computing and describes the design of a SOAP implementation suitable for systems with stringent memory and bandwidth requirements. A thorough analysis of all performance issues shows that major improvements result from using schema-specific parser mechanisms for arrays, trie data structure for matching tags, efficient memory management, persistent connections as allowed by HTTP 1.1, and chunking (streaming) of messages to avoid the overhead of fully serializing objects to determine their content length before transmission begins. Surprisingly, after these optimizations the most critical bottleneck is conversion to and from floating point numbers and their ASCII representation in XML. This accounts for over 90% of the end-to-end message time when using a fully compliant SOAP over HTTP protocol.

Two basic approaches can then be taken to resolve the tension between SOAP's universal lowest common denominator capabilities, and the need for high performance communications: (1) extend the SOAP protocol to provide for binary representation of floats, or (2) use SOAP as an initial mechanism that can then negotiate other (faster) shared protocols.

Our recommendations are to follow the second approach, unless and until the community can persuade the W3C organization to extend SOAP to include binary protocols. In this way, scientific components can remain fully interoperable with SOAP-compliant commercial and industrial components, without giving up high performance capabilities for components which can support them. In the event that two components do not have a shared faster protocol, SOAP over HTTP can still be used as a "fail-over" tool that assures communications can succeed.

2 Sending SOAP

SOAP messages are encoded using XML, which requires that all self-described data be sent as ASCII strings. The description takes the form of start and end tags which often constitute half or more of the message's bytes.

2.1 Stages

The sending of a SOAP message can be divided into several stages. These stages may not have a one-to-one mapping with functions in a code implementation, but still provide context useful for discussion and analysis.

1. Traverse data structures representing message.
2. Convert machine representation of data to ASCII.
3. Write ASCII to buffer.
4. Initiate network transmission.

2.1.1 Stage 1

A SOAP message begins as some kind of data structure in a program. Stage 1 traverses this structure to impart a corresponding structure to the SOAP XML. This traversal is not a significant part of the serialization, because each leaf element can can be traversed with simple operations such as member offset calculations, array indexing, or pointer following.

2.1.2 Stage 2

The strings or numbers that comprise the actual data are usually in machine representation, and are converted in Stage 2 to the ASCII form required by XML. For strings already in ASCII, conversion is simple and fast. Strings in UNICODE may require some processing to convert to UTF-8/16. The majority of characters used in scientific computing fall within the ASCII range, and therefore require minimal processing to convert to UTF-8/16.

Integers are usually in two's-complement representation. Conversion to ASCII involves a binary-to-decimal conversion. Floating-point numbers are usually in IEEE-754 representation. Conversion to ASCII also requires a binary-to-decimal conversion, but the floating-point conversion is considerably more complex than the integer conversion.

2.1.3 Stage 3

The ASCII form of the data is stored, along with the appropriate XML tags, to a memory buffer in Stage 3. Exactly how the ASCII is stored can affect the number of memory operations required. For example, if an integer element named 'el' is serialized with

```
sprintf(buf,"<integer>%d</integer>",el);
```

each character of the start tag must first be read from memory, then written to the buffer. However, if the start tag is created with a sequence of statements such as

```
*buf++ = '<';
*buf++ = 'i';
*buf++ = 'n';
```

the characters comprising the start tag may likely already be in the instruction stream as immediate operands.

2.1.4 Stage 4

Finally, in Stage 4 the operating system transmits the contents of the memory buffer. This requires a system call, which is relatively expensive, so the buffer should be flushed sparingly. Using a buffer that is too large to fit in cache, however, may increase cache misses.

When using HTTP 1.0 [10], the length of the body must be specified in a `Content-Length` header field. Because this value cannot be determined until the SOAP message is serialized, the straightforward implementation would (1) use two separate buffers, one for the header and one for the body, and (2) serialize the complete SOAP message before completing the header. This can require two system calls, and consume much memory if the message is large.

The first issue can be resolved by either back-patching or vectored sends. If we insert spaces for the content length during the initial header generation, we can later replace the spaces with the actual content length once the SOAP message has been processed. Alternatively, we can use a vectored send (available on both UNIX and Win32 machines) that concatenates multiple memory buffers in one send call.

The second issue can be be resolved by either using HTTP 1.1 (discussed below), or using a two-step serialization. In the first step the length of the body is calculated without actually storing to the memory buffer. The header can then be completed and the body serialized into the memory buffer. Because the memory buffer does not need to hold the entire body at one time, memory usage is reduced. The actual transmission is most commonly over TCP/IP. Since TCP/IP requires one packet exchange before transmission can begin, establishing a separate connection for each message adds a round-trip delay to each message.

In addition to the delay, creating a connection per message consumes operating system resources. The TCP/IP protocol requires that one end of a closed connection remain in `TIME_WAIT` state for twice the maximum segment lifetime. This period can be as long as four minutes. During this period, a certain amount of memory must be maintained, and the port cannot be reused for the same remote host and port[13].

2.2 Design

Previous work has shown that the memory usage of SOAP can be prodigious. A typical SOAP message may be 4-10 times the size of its corresponding machine representation. This can be particularly significant for large arrays, which are common objects in scientific computing.. Besides being careful not to create extra copies, we chose to address the memory concern by using HTTP 1.1 [11] instead of HTTP 1.0.

HTTP 1.1 supports a form of streaming called chunked encoding. The body is sent in chunks, with each chunk preceded by its size. The content length is no longer necessary, because a receiver can determine the end of the body by processing the chunks. The elimination of the content length means that the sender does not need to buffer the whole message before transmission, which also allows overlap between network transmission and serialization.

Most operating systems copy the user-space memory buffer to a kernel-space buffer during the send system call, and immediately return. The actual transmission occurs in the background. By calling send multiple times for a large message, overlap between the transmission of the previous buffer by the kernel and the preparation of the next buffer by the program will occur. Of course, calling send too many times may cause the overhead of system calls to dominate.

HTTP 1.1 also supports persistent connections, which remain open for multiple messages. This reduces the overhead of creating a new connection for every message.

3 Receiving SOAP

3.1 Stages

Though in some sense receiving a SOAP message is the inverse of sending a SOAP message, the issues are somewhat different. Conceptually, we divide the receiving process into four stages:

1. Read from network into memory buffer.
2. Parse XML.
3. Handle elements.
4. Convert ASCII to machine representation.

3.1.1 Stage 1

The SOAP message is read from the operating system into a memory buffer in Stage 1. This requires a system call, so reading as much data as possible will minimize the number of system calls. If the amount of data is larger than will fit into cache, however, the number of cache misses will increase.

3.1.2 Stage 2

In this stage, the XML is parsed to identify its syntactic constructs. Comments are stripped, start tags located, etc. This parsing normally involves a state machine of some kind. Possible choices are coding the transitions in a switch-statement, or using a table-driven approach.

There are currently two popular paradigms for processing XML, the Document Object Model (DOM) [16] and the Simple API for XML (SAX) [5]. DOM builds a complete object representation of the XML document in memory. This can be memory intensive for large documents, and entails making at least two passes through the data. During the first pass, the object model is constructed. Only after the document is completely parsed can the application interpret the data in another pass.

SAX operates at one level lower. Rather than actually constructing a model in memory, it informs the application of elements through callbacks. This also requires at least two passes through the data. The first pass is performed inside the SAX implementation, and is required to identify tags and element content. The second pass is performed by the application after the XML constructs are pushed to the application through callbacks.

Pull parsing, as exemplified by the XML Pull Parser [1], is an efficient paradigm similar to SAX in that it does not build a complete object model in memory. It differs in that the tags and content are returned directly to the application from calls to the parser, rather than indirectly in the form of callbacks.

A number of dual-mode models also exist. Progressive DOM, for example, switches between a SAX model and a DOM model depending on the needs of the application at that point in the XML processing.

3.1.3 Stage 3

Once the tags are identified, the content of each tag is interpreted. For the parsing paradigms described above, the parser presents the tag and the content as simple text. So the actual interpretation of an element is completely delegated to the application. This means that even if the application uses an efficient data structure like a red-black tree to match actions to tags, it still must examine the tag after the parser has already made one pass through it.

3.1.4 Stage 4

Ultimately the ASCII text must be converted to the machine representation. For numerical data, this will involve a decimal-to-binary conversion. For string data, it may involve a UTF-8/16 to wide string conversion.

3.2 Design

Our design for receiving SOAP messages rests on two interdependent principles. The first is that we read from the network only when we run out of data, and once we have issued a read, we process until we do run out of data.

This means that the stages need to be fully pipelined. The second principle is that we never examine data more than once. Thus, stages 1-4 should all be performed with one pass.

The fundamental reason the popular parsing paradigms require two passes is that they present a data-centric interface to the application. Thus, for example, the parser must first make one pass to syntactically identify the end of the content. The application then makes another pass to interpret the content. Likewise, the parser first makes one pass to demark the end of a start tag. The application must then examine the tag again to decide how to handle it.

To avoid this we interface to the application through a streamed, push-pull model. Like SAX, we make callbacks to the application. Unlike SAX, however, the parser has already matched the tag to a specific callback. The application therefore does not need to examine the tag again, and in fact the tag may not have ever existed as a complete string anywhere in memory.

One candidate for tag matching is perfect hashing. Perfect hashing, however, may generate a valid hash code for an item not in the hash table. Thus, the item must be reexamined after the hash code has been computed, which requires two passes over the tag. We therefore elected to use *tries*, which unambiguously determine whether or not a key is valid. A trie is essentially a table-driven deterministic finite automaton for a fixed set of strings. As the parser encounters each character of a tag, it simultaneously feeds it into the trie. Upon reaching the end of the tag, the trie has already matched the tag to a specific handler.

When the handler is called, the content of an element is not presented as data, but as a function that the application can call for the next character. The function parses the XML on-the-fly as the application requests each character of the content. Thus, the parser does not need to make an initial pass through the XML to identify the end of the character data. Attributes are handled similarly.[1]

3.3 Schema-Specific Parsers

For a scientific application to interpret the data in a SOAP message, it must have some idea of its structure. This structure may be known in ad hoc manner, or it may be formalized through an XML Schema.

In either case, performance can be improved by using this information to generate a schema-specific parser. For example, rather than representing the tag-matching tries as tables, they can be represented directly in the program itself as code. One can even imagine a parser-generator that directly generates machine-code (or Java byte-code) from a schema.

For our tests, we implemented a limited form of a schema-specific parser. The SOAP encoding rules specify an array form that is easier to parse than general XML. A parser written just for this form can be faster than a general XML parser. By using the schema to direct the parsing, we can use the array parser whenever we are parsing an array.

[1]Our XML parser does not currently support the full XML specification. For example, CDATA sections are not supported. Namespaces are parsed, but we do not yet have an API for tag handlers within namespaces.

4 Performance Measurements

As a sample of the kind of interacting scientific components which distributed computing needs to address, we first examine a “mesh interface object.” This is an array of objects of the the form (int, int, double), with the first two entries representing a mesh coordinate and the double representing a field value. This is the kind of information that might be used for communicating between two partial differential equation (PDE) solvers on different domains. One example of this is a climate model that ties together an atmospheric simulator with an ocean circulation simulator [2]. Another example is a fluid simulation that is coupled with a solids structure code, as is done in some industrial process modeling [8]. We also examine performance for large arrays of IEEE 754 standard doubles [15] (64-bit floats) which routinely occur in scientific computing and typically dominate the total number of bytes sent between procedures and functions. We varied the size of the double arrays from 10 to 1,000,000. All Solaris machines mentioned below were running Solaris 8, with code that was compiled with the Workshop 5.3 C++ compiler. All Linux machines were running the Linux 2.4 kernel with Redhat 7.1. The code was compiled with g++ 3.0.3. All machines were connected via 100 Mbps Ethernet. All performance profiles were produced with the Sun Forte 6 Update 2 performance tools.

4.1 Deserialization

Some of our performance enhancements affected deserialization only. Since our deserialization was no slower than our serialization, the deserialization was isolated to reveal the effects. A dummy sender repeatedly transmitted a precomposed SOAP message to the SOAP receiver.

4.1.1 Trie

Processing SOAP messages involves repeated matching of XML tags. The STL `map` is implemented as a balanced binary tree, for which lookups are $O(\lg n)$, compared to the trie for which lookups are $O(1)$. The trie also has a lower constant because the matching is done as the parser scans the tag. The binary tree, on the other hand, needs to make repeated comparisons of the tag against keys stored at the nodes.

Two experiments tested the effect of tries. The first test used an array of mesh interface objects (see Figure 1). This shows that tries do improve performance significantly, but the improvement was much larger on the Linux platforms. To understand why, we deleted the double from the object and tested just transmitting the “mesh coordinate” object (see Figures 2).

Architecture	*STL Map*	*Trie*	*Improvement*
Linux	73,000	96,000	31.5%
Solaris	22,000	27,000	22.7%

Figure 1. Objects/second throughput for mesh interface object on Linux.

Architecture	*STL Map*	*Trie*	*Improvement*
Linux	123,000	185,000	51.2%
Solaris	59,000	89,000	50.8%

Figure 2. Objects/second throughput for an array of objects containing two integers.

Now the improvement from using tries on the two platforms is larger and almost identical, showing the effectiveness of this technique. However, the tests show that processing of doubles has a significant performance impact. This, together with the importance of arrays of doubles in scientific computing, motivated us to more closely examine the performance of SOAP arrays of doubles.

4.1.2 Array Parsing

We use knowledge of the SOAP message schema to enable an array-specific parser when parsing SOAP arrays. Figures 3 and 4 show that the array parser significantly improves performance, from about 170000 to 225000 doubles/second on Linux. Again, however, the performance gain was significantly lower on Solaris machines, suggesting that some other feature of handling doubles is at play. Section 4.3 shows that this is from the conversion between ASCI and binary representations, but first we examine reducing the network costs via persistent connections and streaming.

4.2 End-To-End

The end-to-end results include the cost of serialization, deserialization and communication over the network for sending a message between two nodes.

4.2.1 Persistent Connections and Streaming

Figures 5 and 6 show that persistent connections improve performance for messages that have less than 100,000 doubles. Because the cost of establishing a connection increases with latency, the benefit of persistent connections will be especially pronounced for a high-latency, high-bandwidth network. However, for larger messages, the cost of establishing socket connections is amortized over many doubles.

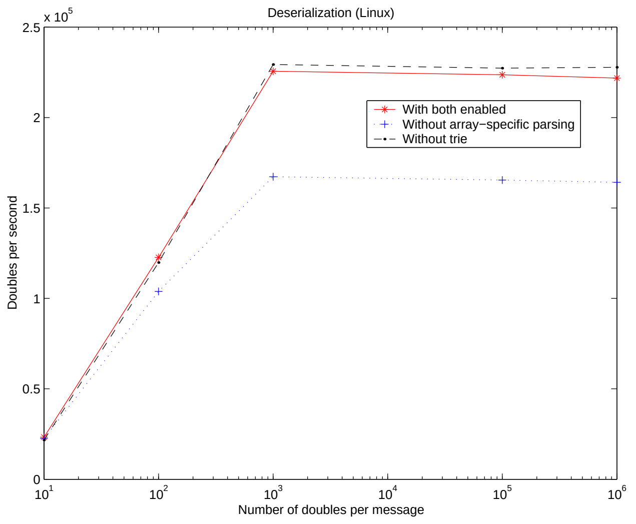

Figure 3. Effect of array-specific parsing and trie data structure on deserialization of array of doubles on Linux. Both sender and receiver were 600 MHz Pentium IIIs.

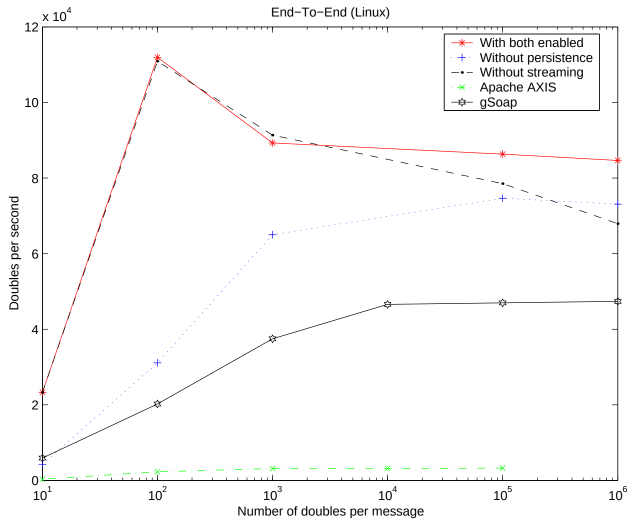

Figure 5. Effect of persistent connections and streaming for array of doubles on a Linux machine. The sender was a Pentium III (Coppermine) with 256 KB of on-die Level 2 cache. The receiver was a Pentium III (Katmai) with 512 KB of off-die Level 2 cache. The peak for arrays of 100 doubles is likely due to cache effects.

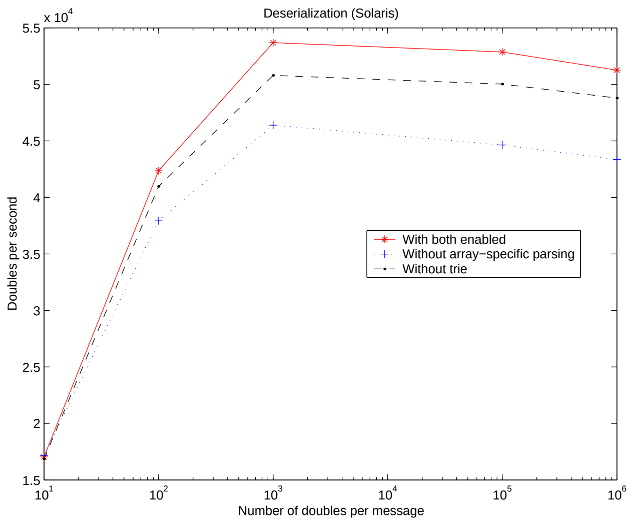

Figure 4. Effect of array-specific parsing and trie data structure on deserialization of array of doubles on Solaris. The sender and receiver were both Sun Blade 100 machines. Each machine had a 500 MHz UltraSPARC-IIe.

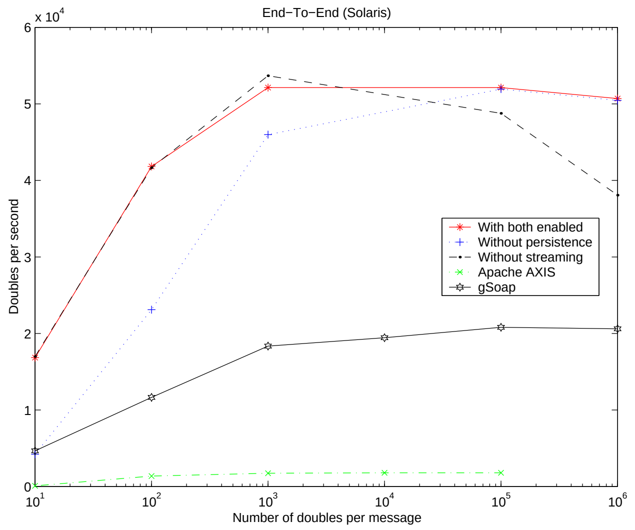

Figure 6. Effect of persistent connections and streaming for an array of doubles on Solaris machines. The sender and receiver were both Sun Blade 100 machines. Each machine had a 500 MHz UltraSPARC-IIe.

Streaming has larger impact on the performance for large messages, because it allows overlap between communication and deserialization that would otherwise not be possible. This overlap is not significant for small messages since the communication time is short. The test shows, however, that the overhead is low for using streaming even for small messages.

Analysis with the Sun Forte 6 tools on a Blade 1000 showed that for messages of 1,000,000 doubles, streaming (via chunking) reduced time stalled on L2 cache misses from .147 to .009 seconds, but this is insignificant compared to the total execution time of 35 and 26 seconds, respectively. The large peak for messages with 100 doubles in the Linux chart is likely a cache effect that manifests in the presence of 16 KB of L1 cache and full-speed L2 cache. The sender in this case was a Pentium III with 256 KB of full-speed L2 cache, while the receiver was a Pentium III with 512 KB of half-speed L2 cache. When the roles were reversed, the peak was not as pronounced. Furthermore, when we used a Pentium 4, there was no peak at all. The Pentium 4 has 256 KB of full-speed L2 cache, but only 8 KB of L1 cache. For Solaris, the peak is not present at all. This is because other factors mask the cache effect. Analysis with the Sun Forte 6 tools on a Blade 1000 showed that the function that consumed the most CPU cycles for message sizes from 10 to 1000 (a Solaris internal function named `multiply_base_2_vector()`) was actually incurring no cache misses.

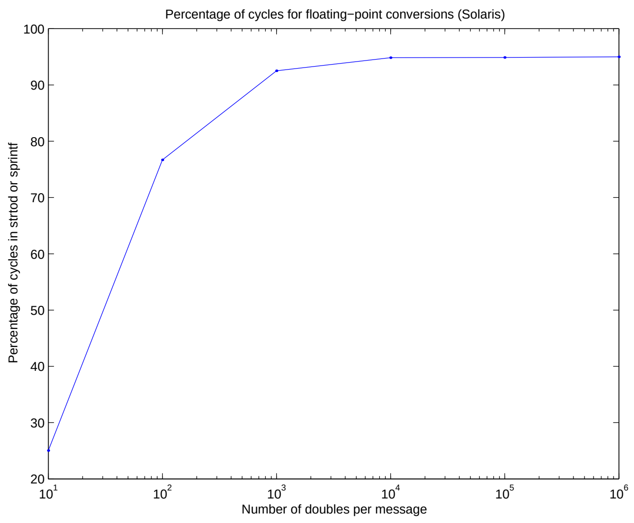

Figure 7. Percentage of CPU cycles spent performing double-to-ASCII or ASCII-to-double conversions as a function of message size. The test was conducted on a Sun Blade 1000 with 2x750 MHz UltraSPARC-IIIs.

4.3 ASCII/Double Conversion

Linux and Solaris platforms differ greatly in their performance improvements from using tries and array-specific parsing involving doubles. We examined this in detail, and found that for a high-performance fully compliant SOAP over HTTP implementation, the two most costly operations are the conversion of ASCII to double and vice versa. Figure 7 shows the fraction of the total CPU cycles that is spent in the functions `sprintf()` and `strtod()`, as a function of the message size. This graph shows that any further improvements to SOAP message processing will have little effect on the efficiency of sending arrays of doubles, unless the conversion of doubles is specifically addressed. In other words, with a high-performance SOAP implementation, an upper bound on the performance of arrays of doubles can be obtained simply by measuring the performance of `sprintf()` and `strtod()`.

If the full 18 digits of double precision are not required, some performance enhancement, especially on the Solaris platform, can be obtained by reducing the digits of precision. Figure 8 and 9 show the performance of the respective conversions as a function of the number of digits of precision. The Solaris platform exhibits sharp drops in performance between 14 and 17 digits. Profiling showed that somewhere in this range the library calls switch to costly high-precision integer operations. Note that the IRIX platform performs remarkably well. The IRIX machine is significantly slower than the other two, yet it performs almost as well as the 600 MHz Pentium III.

4.4 Comparison with Other SOAP Implementations

To corroborate our results, we also compared our performance against gSOAP [12] and Apache Extensible Interaction System (AXIS) [14]. The gSOAP system provides a language binding for deploying SOAP in applications using C/C++ code. Since the gSOAP system only supports request-response messaging, we estimated the double/second rate by using the total time it took to send and receive arrays of doubles. gSOAP's performance is lower than ours, as shown in in Figures 5 and 6. The tests for gSOAP were run without the HTTP keep-alive option and that may have slightly degraded the performance.

We also used a Java based implementation of the AXIS system. It performs poorly when compared to gSOAP and our implementation. However, it is still in its early stages of development, and our results may help later in enhancing its performance.

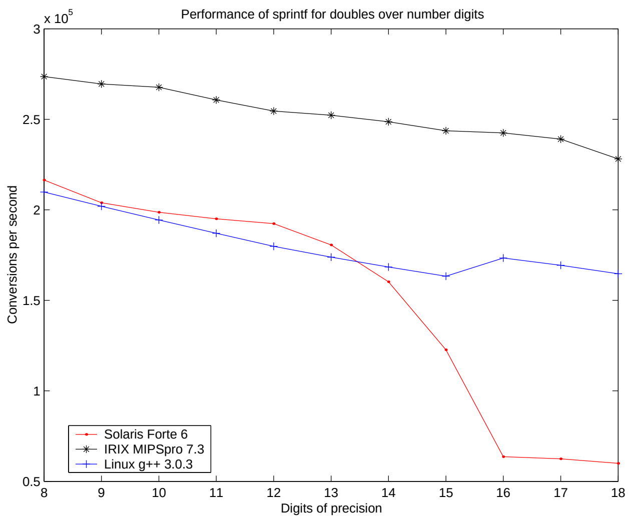

Figure 8. Performance of double to ASCII conversion over digits of precision. The Solaris machine was a Sun Blade 100 with a 500 Mhz UltraSPARC-IIe. The IRIX machine was an SGI Octane with a 195 Mhz R10000. The Linux machine was a 600 MHz Pentium III (Katmai).

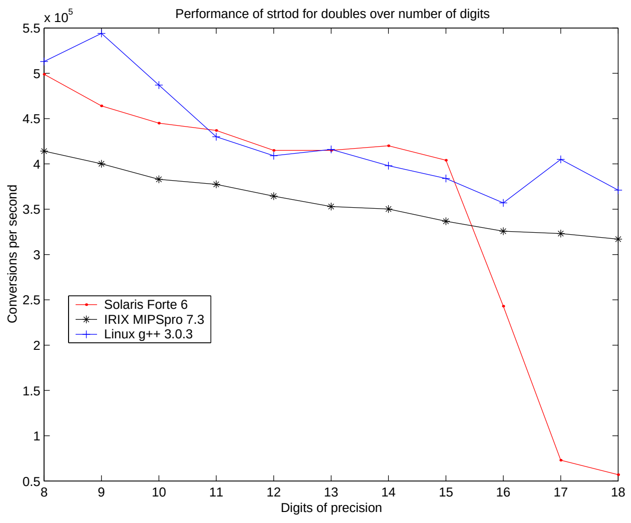

Figure 9. Performance of ASCII to double conversion over digits of precision. The Solaris machine was a Sun Blade 100 with a 500 Mhz UltraSPARC-IIe. The IRIX machine was an SGI Octane with a 195 Mhz R10000. The Linux machine was a 600 MHz Pentium III (Katmai).

5 Summary

By dividing the process of sending and receiving a SOAP call into various stages, we analyzed efficient ways to handle each phase. We have introduced several techniques that significantly improve SOAP performance:

1. The use of tries to more efficiently handle the processing of tags, which can account for half or more of the SOAP message.
2. An XML parser that is specialized for SOAP arrays and greatly improves the performance of deserialization routines.
3. The use of persistent connections to eliminate time spent in establishing and tearing down multiple TCP//IP connections.
4. HTTP 1.1 mechanisms for chunking, a form of streaming.
5. XML parsing techniques which minimize the number of passes over the data.

We measured and analyzed the performance of SOAP messaging with these techniques, starting out with a “mesh interface object” consisting of an array of the form (int, int, double). Differences in the performance improvements the techniques provided on two different platforms led to a more detailed examination of the processing of arrays of doubles; such arrays are common in scientific computing and their efficient handling is crucial. After our performance optimizations, over 90% of the CPU cycles were spent performing double-to-ASCII conversions for arrays of 1000 or more doubles. Further improvements must come from improving the numerical algorithms for these conversions, rather than from improving the parsing of the SOAP message *per se*. Our work also shows that users can readily estimate the communications performance for their applications by measuring the performance of their functions that convert strings to/from binary representations.

6 Recommendations

This work shows that for SOAP to support high performance distributed scientific computing, the SOAP standard will need to be modified or extended. One modification would be to allow a user-specified “tolerance” for doubles and floats. Our results show that on the IRIX platform the conversion from binary to ASCII is very efficient, while that used by Solaris and Linux are significantly slower. A major reason for their large performance drop near full precision (17 digits) is because the IEEE standard specifies rounding modes, causing the conversion functions to use multi-precision libraries. The disadvantage of allowing users to specify lower tolerances is that it departs from strict adherence to the IEEE 754 numerical standard. However, programming languages rarely provide mechanisms for users to access features like setting rounding modes, and full adherence would require the SOAP messages to also include that information. Furthermore, high performance computing users are already accustomed to variations in results for parallel programs resulting from, e.g., different summation orders for dotproducts.

Another possibility is to use binary encoding of arrays in SOAP. For scientific computations, this would provide the single largest performance increase. Note that the peak transmission rate on Linux is approximately 10^5 doubles/second, and with string conversion taking 90% of the time this implies a transmission rate close to the maximum possible on 100 Mb/sec Ethernet. So for array-dominated scientific objects, such a binary encoding (combined with the other optimizations we have introduced) would achieve network-limited rates.

However, this means the protocol would no longer comply with the SOAP standard. Even if all scientific component writers agreed to such an extension, it implies that components would not be interoperable with those being developed by industry, e.g. Web Services-compliant instrument components which scientific simulations might need to access for input data.

Our work indicates that to maximize both interoperability and performance, components need to be able to use multiple communication protocols. A fully compliant SOAP standard would be used initially to negotiate what other protocols a remote component "understands." Those other protocols might include SOAP enhanced with binary representations of arrays, a pure binary protocol, or even parallel protocols for components consisting of parallel processes. If the remote component cannot use any of those, standard SOAP can still be used for communications.

References

[1] Aleksander Slominski. XML Pull Parser, visited 04-15-02. http://www.extreme.indiana.edu/xgws.

[2] Climate Research Committee (E. J. Barron, D. S. Battisti, B. A. Boville, K. Bryan, G. F. Carrier, R. D. Cess, R. E. Davis, M. Ghil, M. M. Hall, T. R. Karl, J. T. Kiehl, D. G. Martinson, C. L. Parkinson, B. Saltzman, R. P. Turco). Global ocean-atmosphere- land system (goals) for predicting seasonal-to-interannual climate. National Academy Press, Washington, D.C., 1994.

[3] D. Box et al. Simple Object Access Protocol. Technical report, IETF, 1999. http://www.ietf.org/internet-drafts/draft-box-http-soap-01.txt.

[4] D. Box et al. Simple Object Access Protocol 1.1. Technical report, W3C, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[5] D. Megginson et al. SAX 2.0: The Simple API for XML, visited 07-01-00. www.megginson.com/SAX/.

[6] Erik Christensen, Francisco Curbera, Greg Meredith and Sanjiva Weerawarana. Web Services Description Lanaguage, visited 03-01-02. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[7] Ian Foster and Carl Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.

[8] F. Illinca, J.-F. Hetu, and R. Bramley. Simulation of 3-d mold-filling and solidification processes on distributed memory parallel architectures. Proceedings of International Mechanical Engineering Congress & Exposition.

[9] Madhusudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, Dennis Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Proceedings of SuperComputing 2000, Dallas TX, 2000*, November 2000.

[10] Network Working Group . Hypertext Transfer Protocol 1.0, visited 03-04-02. http://www.ietf.org/rfc/rfc1941.txt.

[11] Network Working Group. Hypertext Transfer Protocol 1.1, visited 03-04-02. http://www.ietf.org/rfc/rfc2616.txt.

[12] Robert A. van Engelen and Kyle A. Gallivan. The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. In *Proceedings of IEEE CC Grid Conference*, 2002.

[13] T. Faber, J. Touch, W. Yue. The TIME-WAIT State in TCP and its Effect on Busy Servers. In *Proceedings of IEEE INFOCOM*, March 1999.

[14] The Apache XML Project. Apache AXIS, visited 05-01-02. http://xml.apache.org/axis.

[15] William J. Cody, Jr. and Jerome T. Coonen and David M. Gay and K. Hanson and David Hough and W. Kahan and R. Karpinski and John F. Palmer and F. N. Ris and D. Stevenson. A proposed radix- and word-length-independent standard for floating-point arithmetic. *IEEE MICRO*, 4(4):86–100, Aug. 1984.

[16] World Wide Web consortium. Document object model, visited 7-15-99. http://www.w3c.org/DOM.

[17] World Wide Web Consortium. XML, visited 7-20-99. http://www.xml.org.