Efficient Wire Formats for High Performance Computing

Fabian Bustamante Greg Eisenhauer Karsten Schwan Patrick Widener*
College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
{fabianb, eisen, schwan, pmw}@cc.gatech.edu

Abstract

High performance computing is being increasingly utilized in non-traditional circumstances where it must interoperate with other applications. For example, online visualization is being used to monitor the progress of applications, and real-world sensors are used as inputs to simulations. Whenever these situations arise, there is a question of what communications infrastructure should be used to link the different components. Traditional HPC-style communications systems such as MPI offer relatively high performance, but are poorly suited for developing these less tightly-coupled cooperating applications. Object-based systems and meta-data formats like XML offer substantial plug-and-play flexibility, but with substantially lower performance. We observe that the flexibility and baseline performance of all these systems is strongly determined by their `wire format', or how they represent data for transmission in a heterogeneous environment. We examine the performance implications of different wire formats and present an alternative with significant advantages in terms of both performance and flexibility.

1. Introduction

High performance computing is being increasingly utilized in non-traditional circumstances. For instance, many high-end simulations must interoperate with other applications to provide environments for human collaboration, or to allow access to visualization engines and remote instruments [15, 16]. In addition, there has been an increasing interest in component architectures[2], with the intent of facilitating the development of complex applications through reusability.

Whenever these situations arise, there is a question of what communications infrastructure should be used to link the different components. This is specially important in the presence of software evolution and/or runtime changes in component couplings. Traditional HPC-style communications systems such as MPI, and client-server communication paradigms such as RPC, offer high performance. Yet, this systems rely on the basic assumption that communicating parties have a priori agreements on the basic contents of the messages being exchanged. Maintaining such agreements has become increasingly onerous for these less tightly-coupled systems.

The flexibility requirements of these new systems has led designers to adopt techniques such as the use of serialized objects as messages (as in Java's RMI [20]) or the use of meta-data representations like XML. However, both of these approaches have marshalling and communications costs that are staggeringly high in comparison to the more traditional approaches.

We observe that the flexibility and baseline performance of all these systems is strongly determined by their `wire format', or how they represent data for transmission in a heterogeneous environment. We examine the performance implications of different wire formats and present an alternative with significant advantages in terms of both performance and flexibility. Essentially, this approach eliminates common wire formats like XDR and instead transmits data in the sender's native format (which we term *Natural Data Representation or NDR*), along with meta-information that identifies these formats. Any necessary data conversion on the receiving side is performed by custom routines created through dynamic code generation (DCG). When the sender and receiver have the same natural data representation, such as in exchanges between homogeneous architectures, this approach allows received data to be used directly from the message buffer, making it feasible for middleware to effectively utilize high performance communication layers like FM [13] or the zero-copy messaging demonstrated by Rosu et al. [17] and Welsh et al. [19]. When conversion between formats is necessary, these DCG conversions are of the same order of efficiency as the compile-time generated

*0-7802-9802-5/2000/$10.00 (c) 2000 IEEE

stub routines used by the fastest systems relying upon a priori agreements[14]. However, because the conversion routines are derived at run-time, our approach offers considerably greater flexibility than other systems.

Our experiments with a variety of realistic applications show that our alternative approach obtains the required flexibility at no cost to performance. On the contrary, the results presented in Section 4 show an improvement of up to 3 orders of magnitude in the sender encoding time, 1 order of magnitude on the receiver side, and a 45 % reduction on roundtrip time, when comparing it to a non-flexible approach like MPI's.

The remainder of this paper is organized as follows. In Section 2 we review related approaches to communication and comment on their performance and flexibility. Section 3 presents our approach, Natural Data Representation, and its implementation in the Portable Binary I/O (PBIO) communication library. In Section 4 we compare the flexibility and performance of PBIO with that of some alternative communications systems. After examining the different costs involved in the communication of binary data on heterogeneous platforms, we proceed to evaluate the relative costs of MPI, XML and CORBA-style communications in exchanging the same sets of messages, and compare them to those of PBIO. Finally, we compare the performance effects of dynamic type discovery and exetension for PBIO and XML-based systems. We present our conclusion and discuss some directions for future work in Section 5.

2. Background and Related Work

In high performance communication packages, the operational norm is for all parties to a communication to have an *a priori* agreement on the format of messages exchanged. Many packages, such as PVM[10] and Nexus[9], support message exchanges in which the communicating applications "pack" and "unpack" messages, building and decoding them field by field. Other packages, such as MPI[8], allow the creation of user-defined datatypes for messages and fields and provide some marshalling and unmarshalling support for them internally. However, MPI does not have any mechanisms for run-time discovery of data types of unknown messages and any variation in message content invalidates communication.

By building its messages manually, an application attains significant flexibility in message contents while ensuring optimized, compiled pack and unpack operations. However, relegating these tasks to the communicating applications means that those applications must have *a priori* agreement on the format of messages. In wide-area high performance computing, the need to simultaneously update all application components in order to change message formats can be a significant impediment to the integration, deployment and evolution of complex systems.

In addition, the semantics of application-side pack/unpack operations generally imply a data copy to or from message buffers, with a significant impact on performance [13, 17]. Packages which perform internal marshalling, such as MPI, could avoid data copies and offer more flexible semantics in matching fields provided by senders and receivers. However, existing packages have failed to capitalize on those opportunities. For example, MPIs type-matching rules require strict *a priori* agreement on the content of messages, and most MPI implementations marshal user-defined datatypes via mechanisms that amount to interpreted versions of field-by-field packing.

The use of object systems technology provides for some amount of plug-and-play interoperability through subclassing and reflection. This is a significant advantage in building loosely coupled systems, but they tend to suffer when it comes to communication efficiency. For example, CORBA-based object systems use IIOP as a wire format. IIOP attempts to reduce marshalling overhead by adopting a "reader-makes-right" approach with respect to byte order (the actual byte order used in a message is specified by a header field). This additional flexibility in the wire format allows CORBA to avoid unnecessary byte-swapping in message exchanges between homogeneous systems but is not sufficient to allow such message exchanges without copying of data at both sender and receiver. At issue here is the contiguity of atomic data elements in structured data representations. In IIOP, XDR and other wire formats, atomic data elements are contiguous, without intervening space or padding between elements. In contrast, the native representations of those structures in the actual applications must contain appropriate padding to ensure that the alignment constraints of the architecture are met. On the sending side, the contiguity of the wire format means that data must be copied into a contiguous buffer for transmission. On the receiving side, the contiguity requirement means that data cannot be referenced directly out of the receive buffer, but must be copied to a different location with appropriate alignment for each element.

While all of the communication systems above rely on some form of a fixed wire format for communication, XML takes a different approach to communication flexibility. Rather than transmitting data in binary form, its wire format is ASCII text, with each record represented in textual form with header and trailer information identifying each field. This allows applications to communicate with no a priori knowledge of each others. However, XML encoding and decoding costs are substantially higher that those of other formats due to the conversion of data from binary to ASCII and vice-versa. In addition, XML has substantially higher network transmission costs because the ASCII-encoded record is larger, often substantially larger, than the binary original (an expansion factor of 6-8 is not unusual).

3. Approach

Our approach to creating efficient wire formats is to use Natural Data Representation (NDR). That is, data is placed `on the wire' in the natural form in which it is maintained by the sender, then decoded by the receiver `into' its desired form. Our current implementation of NDR is termed PBIO (for Portable Binary I/O). The PBIO library provides a record-oriented communication medium. Writers must provide descriptions of the names, types, sizes and positions of the fields in the records they are writing. Readers must provide similar information for the records they wish to read. No translation is done at the writer's end. At the reader's end, the format of the incoming record is compared with the format expected by the program. Correspondence between fields in incoming and expected records is established by field name, with no weight placed on size or ordering in the record. If there are discrepancies in field size or placement, then PBIO's conversion routines perform the appropriate translations. Thus, the reader program can read the binary information produced by the writer program despite potential differences in: (1) byte ordering on the reading and writing architectures; (2) differences in sizes of data types (e.g. long and int); and (3) differences in structure layout by compilers,

The flexibility of the application is additionally enhance by the availability of full format information for the incoming record. This allows the receiving parts to make runtime decisions about the use and processing of incoming messages without previous knowledge. Such required flexibility however, comes at the price of potentially complex format conversions on the receiving end: since the format of incoming records is principally defined by the native formats of the writers and PBIO has no *a priori* knowledge of the native formats used by the program components with which it might communicate, the precise nature of this format conversion must be determined at run-time.

Runtime code generation for format interpretation. To alleviate the increased communication costs associated with interpreted format conversion, NDR-based communications are interpreted with dynamically generated binary code. The resulting customized data conversion routines access and store data elements, convert elements between basic types and call subroutines to convert complex subtypes. Measurements[6] show that the one-time costs of generating binary code coupled with the performance gains by then being able to use compiled code far outweigh the costs of continually interpreting data formats. The analysis in the following section shows that an NDR-based approach to data transmission also results in performance gains through copy reduction, while providing additional type-matching flexibility not present in other component-based distributed programming systems.

4. Evaluation

This section compares the performance and flexibility of our NDR-based approach to data exchange with that of systems like MPI, CORBA, and XML-based ones.

4.1. Analysis of Costs in Heterogeneous Data Exchange

Binary data exchange costs in a heterogeneous environment. Before analyzing the various packages in detail, it is useful to examine the costs in an exchange of binary data in a heterogeneous environment. As a baseline for this discussion, we use the MPICH[12] implementation of MPI. Figure 1 represents a breakdown of the costs of an MPI message round-trip between a x86-based PC and a Sun Sparc connected by 100 Mbps Ethernet.[1]

The time components labeled “Encode” represent the time span between the time the application invokes `MPI_send()` and its eventual call to write data on a socket. The “Decode” component is the time span between the `recv()` call returning and the point at which the data is in a form usable by the application. This delineation allows us to focus on the encode/decode costs involved in binary data exchange. That these costs are significant is clear from the figure, where they typically represent 66% of the total cost of the exchange.

Figure 1 shows the cost breakdown for messages of a selection of sizes (from a real mechanical engineering application), but in practice, message times depend upon many variables. Some of these variables, such as basic operating system characteristics that affect raw end-to-end TCP/IP performance, are beyond the control of the application or the communication middleware. Different encoding strategies in use by the communication middleware may change the number of raw bytes transmitted over the network; much of the time those differences are negligible, but where they are not, they can have a significant impact upon the relative costs of a message exchange.

The next sections will examine the relative costs of PBIO, MPI, CORBA, and an XML-based system in exchanging the same sets of messages.

4.2. Sending Side Cost

Figure 2 shows a comparison of sending-side data encoding times on the Sparc for an XML-based implementation[2],

[1] The Sun machine is an Ultra 30 with a 247 MHz cpu running Solaris 7. The x86 machine is a 450 MHz Pentium II, also running Solaris 7.

[2] A variety of implementations of XML, including both XML generators and parsers, are available. We have used the fastest known to us at this time, Expat [3].

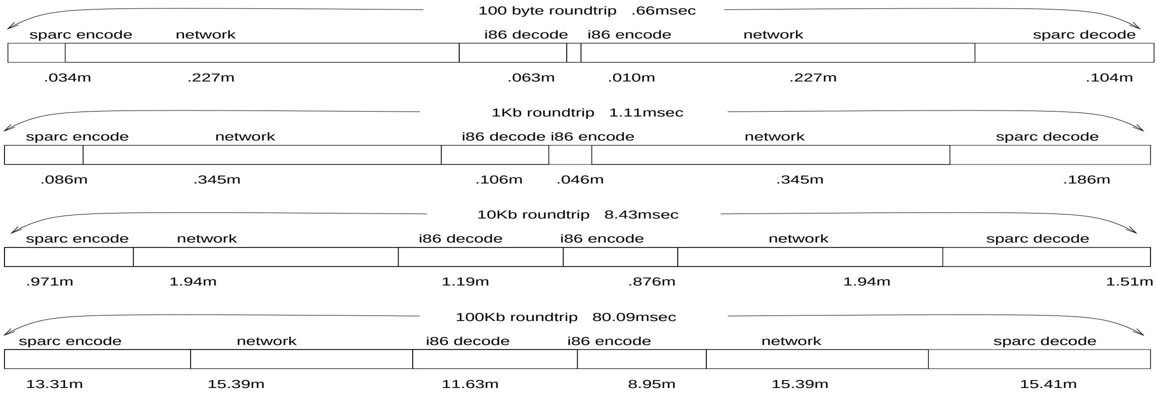


Figure 1. Cost breakdown for message exchange.

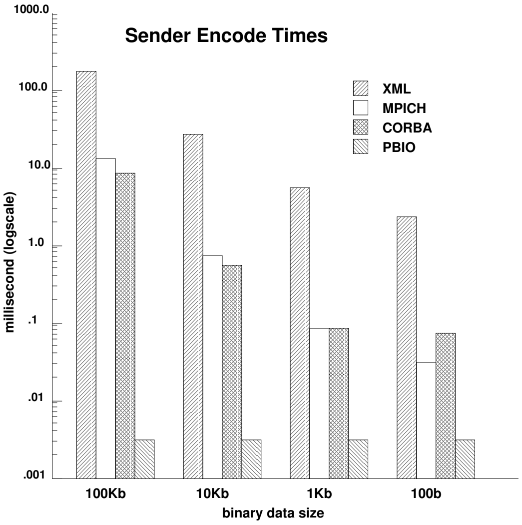


Figure 2. Send-side encoding times.

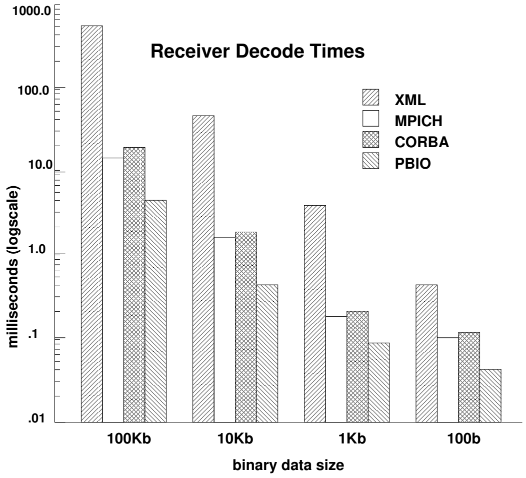


Figure 3. Receive side decode times.

MPICH, CORBA, and PBIO.

XML wire formats are inappropriate. The figure shows dramatic differences in the amount of encoding necessary for the transmission of data (which is assumed to exist in binary format prior to transmission). The XML costs represent the processing necessary to convert the data from binary to string form and to copy the element begin/end blocks into the output string.

Advantages derived from NDR. As is mentioned in Section 3, we transmit data in the native format of the sender. As a result, no copies or data conversions are necessary to prepare simple structure data for transmission. So, while MPICH's costs to prepare for transmission on the Sparc vary from 34μsec for the 100 byte record up to 13 msec for the 100Kb record, PBIO's cost is a flat 3 μsec.

4.3. Receiving Side Costs

Our NDR approach to binary data exchange eliminates sender-side processing by transmitting in the sender's native format and isolating the complexity of managing heterogeneity in the receiver. As a result, the receiver must perform conversion of the various incoming `wire' formats to its `native' format. PBIO matches fields by name, so a conversion may require byte-order changes (byte-swapping), movement of data from one offset to another, or even a change in the basic size of the data type (for example, from a 4-byte integer to an 8-byte integer).

This conversion is another form of the "marshaling problem" that occurs widely in RPC implementations[1] and in network communication. Marshaling can be a significant overhead[4, 18], and tools like the Universal Stub Compiler (USC) [14] attempt to optimize marshaling with compile-time solutions. Although optimization considerations similar to those addressed by USC apply in our case, the dynamic form of the marshaling problem in PBIO, where the layout and even the complete field contents of the incoming record are unknown until run-time, rules out such static solutions. The conversion overhead is nil for some homogeneous data exchanges, but as Figure 1 shows, the overhead is high (66%) for many heterogeneous exchanges.

Generically, receiver-side overhead in communication middleware has several components:

- byte-order conversion,
- data movement costs, *and*
- control costs.

Byte order conversion costs are to some extent unavoidable. If the communicating machines use different byte orders, the translation must be performed somewhere regardless of the capabilities of the communications package.

Data movement costs are harder to quantify. If byte-swapping is necessary, data movement can be performed as part of the process without incurring significant additional costs. Otherwise, clever design of the communication middleware can often avoid copying data. However, packages that define a `wire format' for transmitted data have a harder time being clever in this area. One of the basic difficulties is that the native format for mixed-datatype structures on most architectures has gaps, unused areas between fields, inserted by the compiler to satisfy data alignment requirements. To avoid making assumptions about the alignment requirements of the machines they run on, most packages use wire formats which are fully packed and have no gaps. This mismatch *forces* a data copy operation in situations where a clever communications system might otherwise have avoided it.

Control costs represent the overhead of iterating through the fields in the record and deciding what to do next. Packages that require the application to marshal and unmarshal their own data have the advantage that this process occurs in special-purpose compiler-optimized code, minimizing control costs. However, to keep that code simple and portable, such systems uniformly rely on communicating in a predefined wire format, therefore incurring the data movement costs described in the previous paragraph.

Packages that marshal data themselves typically use an alternative approach to control, where the marshaling process is controlled by what amounts to a table-driven interpreter. This interpreter marshals or unmarshals application-defined data, making data movement and conversion decisions based upon a description of the structure provided by the application and its knowledge of the format of the incoming record. This approach to data conversion gives the package significant flexibility in reacting to changes in the incoming data and was our initial choice when implementing NDR.

XML necessarily takes a different approach to receiver-side decoding. Because the `wire' format is a continuous string, XML is parsed at the receiving end. The Expat XML parser[3] calls handler routines for every data element in the XML stream. That handler can interpret the element name, convert the data value from a string to the appropriate binary type and store it in the appropriate place. This flexibility makes XML extremely robust to changes in the incoming record. The parser we have employed is quite fast, but XML still pays a relatively heavy penalty for requiring string-to-binary conversion on the receiving side.

Comparison of receiver-side costs for XML-based, MPI, CORBA, and PBIO wire formats. Figure 3b shows a comparison of receiver-side processing costs on the Sparc for interpreted converters used by XML, MPICH (via the `MPI_Unpack()`) call, CORBA, and PBIO. XML receiver conversions are clearly expensive, typically between one and two orders of decimal magnitude more costly than our NDR-based converter for this heterogeneous exchange. (On an exchange between homogeneous architectures, PBIO and MPI would have substantially lower costs, while XML's costs would remain unchanged.) Our NDR-based converter is relatively heavily optimized and performs considerably better than MPI, in part because MPICH uses a separate buffer for the unpacked message rather than reusing the receive buffer (as we do). However, NDR's receiver-side conversion costs still contribute roughly 20% of the cost of an end-to-end message exchange. While a portion of this conversion overhead must be the consequence of the raw number of operations involved in performing the data conversion, we believe that a significant fraction of this overhead is due to the fact that the conversion is essentially being performed by an interpreter.

Optimizing receiver-side costs for PBIO. Our decision to transmit data in the sender's native format results in the wire format being unknown to the receiver until run-time. Our solution to the problem is to employ dynamic code generation to create a customized conversion subroutine for every incoming record type. These routines are generated by the receiver on the fly, as soon as the wire format is known. PBIO dynamic code generation is based on Vcode, a fast dynamic code generation package developed at MIT by Dawson Engler[7]. Vcode essentially provides an API for a virtual RISC instruction set. The provided instruction set is relatively generic, so that most Vcode instruction macros generate only one or two native machine instructions. Native machine instructions are generated directly into a memory buffer and can be executed without reference to an external compiler or linker. We have significantly enhanced Vcode and ported it to several new architectures. With the present implementation we can generate code for

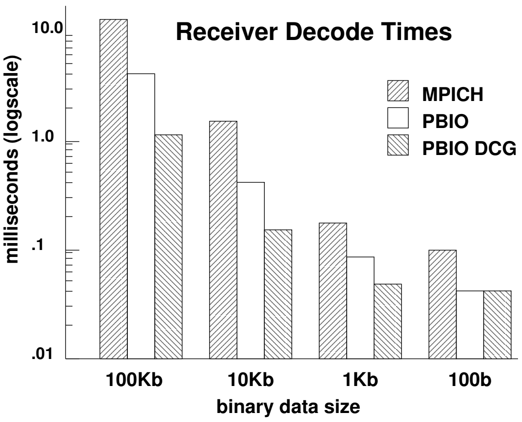


Figure 4. Receiver side costs for interpreted conversions in MPI and PBIO and DCG conversions in PBIO.

Sparc (v8, v9 and v9 64-bit), MIPS (old 32-bit, new 32-bit and 64-bit ABIs), DEC Alpha and Intel x86 architectures [3].

The execution times for these dynamically generated conversion routines are shown in Figure 4. (We have chosen to leave the XML conversion times off of this figure to keep the scale to a manageable size.)

The dynamically generated conversion routine operates significantly faster than the interpreted version. This improvement removes conversion as a major cost in communication, bringing it down to near the level of a copy operation, and it is the key to PBIO's ability to efficiently perform many of its functions.

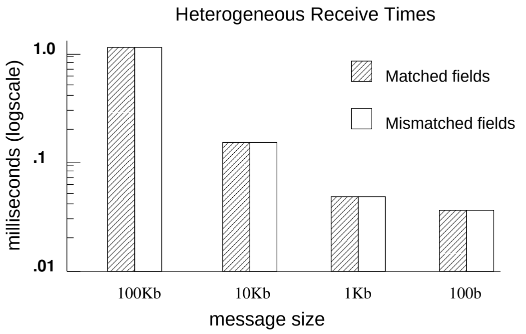
The cost savings achieved for PBIO described in this section are directly reflected in the time required for an end-to-end message exchange. Figure 5 shows a comparison of PBIO and MPICH message exchange times for mixed-field structures of various sizes. The performance differences are substantial, particularly for large message sizes where PBIO can accomplish a round-trip in 45% of the time required by MPICH. The performance gains are due to:

- virtually eliminating the sender-side encoding cost by transmitting in the sender's native format, *and*
- using dynamic code generation to customize a conversion routine on the receiving side (currently not done on the x86 side).

Once again, Figure 5 does not include XML times to keep the figure to a reasonable scale.

[3] More details on the nature of PBIO's dynamic code generation can be found in [5].

Figure 6. Receiver-side decoding costs with and without an unexpected field: Heterogeneous case.



4.4. High Performance and Application Evolution

The principal difference between PBIO and most other messaging middleware is that PBIO messages carry format meta-information, somewhat like an XML-style description of the message content. This meta-information can be a useful tool in building and deploying enterprise-level distributed systems because it (1) allows generic components to operate upon data about which they have no *a priori* knowledge, and (2) allows the evolution and extension of the basic message formats used by an application without requiring simultaneous upgrades to all application components. In other words, PBIO offers limited support for *reflection* and *type extension*. Both of these are valuable features commonly associated with object systems.

PBIO supports reflection by allowing message formats to be inspected before the message is received. Its support of type extension derives from doing field matching between incoming and expected records by name. Because of this, new fields can be added to messages without disruption because application components which don't expect the new fields will simply ignore them.

Most systems that support reflection and type extension in messaging, such as systems using XML as a wire format or marshalling objects as messages, suffer prohibitively poor performance compared to systems such as MPI which have no such support. Therefore, it is interesting to examine the effect of exploiting these features upon PBIO performance. In particular, in the following, we measure the performance effects of type extension by introducing an unexpected field into the incoming message and measuring the change in receiver-side processing.

Figures 6 and 7 present receive-side processing costs for an exchange of data with an unexpected field. These figures show values measured on the Sparc side of heterogeneous and homogeneous exchanges, respectively, using

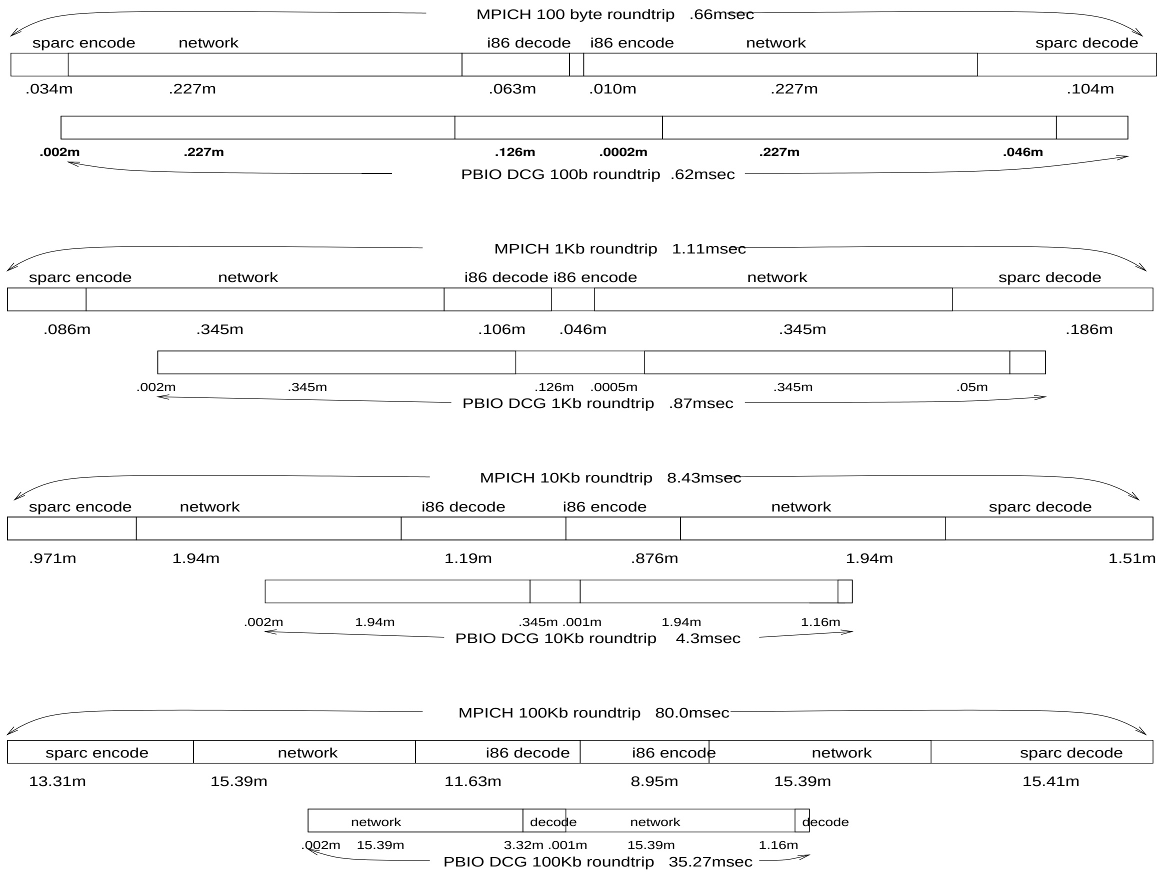
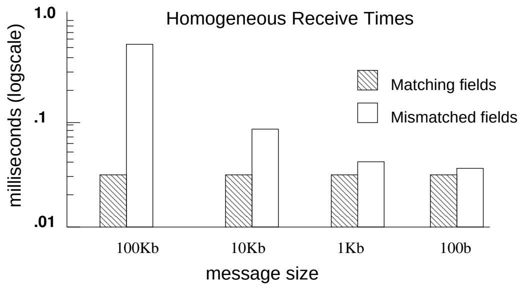


Figure 5. Cost comparison for PBIO and MPICH message exchange.

Figure 7. Receiver-side decoding costs with and without an unexpected field: Homogeneous case.



PBIO's dynamic code generation facilities to create conversion routines. It's clear from Figure 6 that the extra field has not effect upon the receive-side performance. Transmitting would have added slightly to the network transmission time, but otherwise the support of type extension adds no cost to this exchange.

Figure 7 shows the effect of the presence of an unexpected field in the homogeneous case. Here, the overhead is potentially significant because the homogeneous case normally imposes no conversion overhead in PBIO. The presence of the unexpected field creates a layout mismatch between the wire and native record formats and as a result the conversion routine must relocate the fields. As the figure shows, the resulting overhead is non-negligible, but not as high as exists in the heterogeneous case. For smaller record sizes, most of the cost of receiving data is actually caused by the overhead of the kernel `select()` call. The difference between the overheads for matching and extra field cases is roughly comparable to the cost of `memcpy()` operation for the same amount of data.

As noted earlier in Section 4.3, XML is extremely robust with respect to changes in the format of the incoming record. Essentially, XML transparently handles precisely the same types of change in the incoming record as can PBIO. That is, new fields can be added or existing fields reordered without worry that the changes will invalidate existing receivers. Unlike PBIO, XML's behavior does not change substantially when such mismatches are present. Instead, XML's receiver-side decoding costs remain essentially the same as presented in Figure 3. However, those costs are several orders of magnitude higher than PBIO's costs.

For PBIO, the results shown in Figures 6 and 7 are actually based upon a worst-case assumption, where an unexpected field appears before all expected fields in the record, causing field offset mismatches in all expected fields. In general, the overhead imposed by a mismatch varies proportionally with the extent of the mismatch. An evolving application might exploit this feature of PBIO by adding any additional at the end of existing record formats. This would minimize the overhead caused to application components which have not been updated.

5. Conclusions and Future Work

This paper describes and analyzes a basic issue with the performance and flexibility of modern high performance communication infrastructures: the choice of `wire formats' for data transmission. We demonstrate experimentally the performance of different wire formats, addressing the sizes of data transfers (i.e., the compactness of wire formats), the overheads of data copying and conversion at senders and receivers, across heterogeneous machine architectures.

We contribute an efficient and general solution for wire formats for heterogeneous distributed systems, and describe how our approach allows for application flexibility. By using the Natural Data Representation (NDR), copy overheads are avoided at senders and receivers can place data into the forms they desire. We reduce the receiver-side `interpretation' overheads of NDR by runtime binary code generation and code installation at receivers. As a result, the added flexibility comes at no cost and, in fact, the performance of PBIO transmissions exceed that of data transmission performed in modern HPC infrastructures like MPI.

The principal conclusions of this research are (1) that the use of NDR is feasible and desirable and (2) its advantages outweigh its potential costs. Specifically, receivers who have no a priori knowledge of data formats being exchanged can easily `join' ongoing communications. In addition, loosely coupled or `plug-and-play' codes can be composed into efficient, distributed applications whenever desired by end users, without substantial modifications to application components (or recompilation or relinking). In effect, by using NDR and runtime binary code generation, we can create efficient wire formats for high performance components while providing flexibility rivaling that of plug and play systems like Java.

We are continuing to develop the compiler techniques necessary to generate efficient binary code, including the development of selected runtime binary code optimization methods and the development of code generators for additional platforms, most notably the Intel i960 and StrongArm platforms. In related research, our group is developing high performance communication hardware and firmware for cluster machines, so that PBIO-based messaging may be mapped to zero-copy communication interfaces and so that selected message operations may be placed `into' the communication co-processors being used [17, 11].

Acknowledgments

We are grateful to our shepherd Allen D. Malony and the anonymous reviewer for their constructive comments and suggestions that helped to improve the quality of this work. Thanks to Vijaykumar Krishnaswamy and Vernard Martin for reading early versions of this paper. This work was supported in part by NCSA/NSF grant C36-W63, NSF equipment grants CDA-9501637, CDA-9422033, and ECS-9411846, equipement donations from SUN and Intel Corporations, and NSF grant ASC-9720167.

References

[1] G. T. Almes. The impact of language and system on remote procedure call design. In *Sixth International Conference on Distributed Computing Systems, Boston, Mass.*, pages 414–421. IEEE, May 1986.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high performance scientific computing. In *Proceedings of the 8th High Performance Distributed Computing (HPDC'99)*, 1999. http://www.acl.lanl.gov/cca.

[3] J. Clark. expat - xml parser toolkit. http://www.jclark.com/xml/expat.html.

[4] D.D.Clark and D.L.Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Synposium*, pages 200–208, Sept 1990.

[5] G. Eisenhauer and L. K. Daley. Fast heterogenous binary data interchange. In *Proceedings of the Heterogeneous Computing Workshop (HCW2000)*, May 3-5 2000. http://www.cc.gatech.edu/systems/papers/Eisenhauer00FHB.pdf.

[6] G. Eisenhauer, B. Schroeder, and K. Schwan. Dataexchange: High performance communication in distributed laboratories. *Journal of Parallel Computing*, 24(12-13):1713–1733, 1998.

[7] D. R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[8] M. P. I. M. Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, 1995.

[9] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, pages 70–82, 1996.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 Users Guide and Reference manual*. Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 94.

[11] R. Krishnamurthy, K. Schwan, R. West, and M. Rosu. A network coprocessor based approach to scalable media streaming in servers. In *International Conference on Parallel Processing (ICPP 2000)*, Toronto, Canada, August 2000.

[12] A. N. Laboratory. Mpich-a portable implementation of mpi. http://www-unix.mcs.anl.gov/mpi/mpich.

[13] M. Lauria, S. Pakin, and A. A. Chien. Efficient layering for high speed communication: Fast messages 2.x. In *Proceedings of the 7th High Performance Distributed Computing (HPDC7)*, July 1998.

[14] S. W. O'Malley, T. A. Proebsting, and A. B. Montz. Universal stub compiler. In *Proceedings of the SIGCOMM '94 Symposium*, Aug 1994.

[15] C. M. Pancerella, L. A. Rahn, and C. L. Yang. The diesel combustion collaboratory: Combustion researchers collaborating over the internet. In *Proceedings of SC 99*, November 13-19 1999. http://www.sc99.org/proceedings/papers/pancerel.pdf.

[16] B. Parvin, J. Taylor, G. Cong, M. O'Keefe, and M.-H. Barcellos-Hoff. Deepview: A channel for distributed microscopy and informatics. In *Proceedings of SC 99*, November 13-19 1999. http://www.sc99.org/proceedings/papers/parvin.pdf.

[17] M.-C. Rosu, K. Schwan, and R. Fujimoto. Supporting parallel applications on clusters of workstations: The virtual communication machine-based architecture. *Cluster Computing, Special Issue on High Performance Distributed Computing*, 1, January 1998.

[18] M. Schroeder and M. Burrows. Performance of Firefly RPC. In *Twelfth ACM Symposium on Operating Systems, SIGOPS, 23, 5*, pages 83–90. ACM, SIGOPS, Dec. 1989.

[19] M. Welsh, A. Basu, and T. V. Eicken. Incorporating memory management into user-level network interfaces. In *Proceedings of Hot Interconnects V*, pages 27–36, 1997.

[20] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for Java system. In *Proceedings of the USENIX COOTS 1996*, 1996.