ARGONNE NATIONAL LABORATORY

ANL/MCS-TM-284

Understanding the Requirements Imposed by Programming Model Middleware on a Common Communication Subsystem

prepared by
Mathematics and Computer Science Division
Argonne National Laboratory

ANL/MCS-TM-284

Understanding the Requirements Imposed by Programming Model Middleware on a Common Communication Subsystem

by
D. Buntinas and W. Gropp
Mathematics and Computer Science Division, Argonne National Laboratory

July 2005

This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract W-31-109-Eng-38.

Contents

Understanding the Requirements Imposed by Programming Model Middleware on a Common Communication Subsystem

by

Darius Buntinas and William Gropp

Abstract

In high-performance parallel computing, most programming-model middleware libraries and runtime systems use a communication subsystem to abstract the lower-level network layer. The functionality required of a communication subsystem depends largely on the programming model implemented by the middleware. In order to maximize performance, middleware libraries and runtime systems typically implement their own communication subsystems that are specially tuned for the middleware, rather than use an existing communication subsystem. This situation leads to duplicated effort and prevents different middleware libraries from being used by the same application in hybrid programming models. In this paper we describe features required by various middleware libraries as well as some desirable features that would make it easier to port a middleware library to the communication subsystem and allow the middleware to make use of high-performance features provided by some networking layers. We show that none of the communication subsystems that we evaluate support all of the features.

1 Introduction

In high-performance parallel computing, most programming-model middleware libraries and runtime systems use a communication subsystem to abstract the lower-level network layer, providing portability to different architectures and interconnects and simplifying implementation. The functionality required of a communication subsystem depends largely on the particular programming model implemented by the middleware. For example, a middleware library for the message-passing model would require operations that optimize data transfer of objects located anywhere in a process's address space, whereas a middleware runtime system for a global address space language would require optimized transfer of smaller data objects located in a special memory region allocated at initialization.

Because of these differences, most middleware libraries and runtime systems implement their own communication subsystems, rather than use one from another middleware library. But, despite the differences in requirements, the communication subsystems have many common features, such as bootstrapping and remote memory access (RMA) operations. Implementing and maintaining a different communication subsystem for each middleware system lead to duplicated effort. Furthermore, a common communication subsystem would be needed for hybrid programming models, where, for example, a program would use both MPI and UPC operations. A common communication subsystem would provide the best

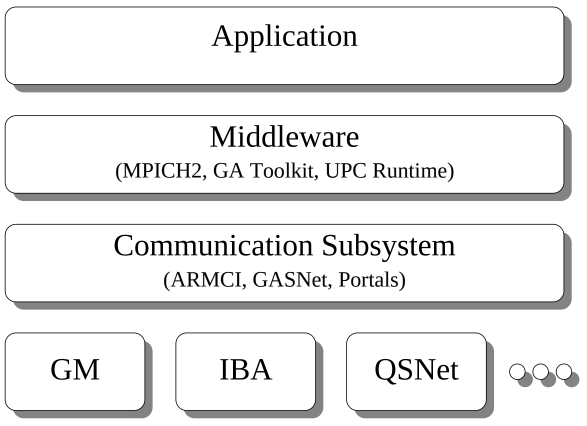

Figure 1: Software layers of current high-performance computing systems

performance to both programming models and avoid possible deadlocks when the different middleware libraries block for messages.

In this paper we describe features required by various middleware libraries as well as some desirable features that would make it easier to port a middleware library to the communication subsystem and allow the middleware to make use of high-performance features provided by some networking layers. We evaluate whether existing communication subsystems support these features efficiently. We show that none of the existing communication subsystems that we evaluated support all of the features. Furthermore, we found no conflicting requirements, a result that indicates that a communication subsystem can be designed to efficiently support all of the programming models examined in this paper. In [7] we present our design of such a common communication subsystem.

The rest of this paper is organized as follows. In Section 2 we introduce the communication subsystems that we will evaluate. In Section 3 we identify the critical design issues necessary to support the various programming models. We conclude and present future work in Section 4.

2 Typical Communication Subsystems

In this section, we first describe the software layers of a typical high-performance parallel computing system, then describe several communication subsystems used by different programming models.

Figure 1 shows the software layers of a typical high-performance parallel computing system. The application, at the top layer, is written by using a particular programming model, for example, message passing, remote memory, or a global address space (GAS) language. The middleware, in the next layer, is the implementation of the programming model and defines the programming interface, for example, MPICH2 [1, 11], Global Arrays (GA) toolkit [18], or the Berkeley UPC runtime system [14]. The middleware often implements a standard API, such as MPI-2 [10], GA [16], or UPC [8].

In order to provide portability to different interconnects, middleware typically is implemented over a communication subsystem, in the third layer, which abstracts the low-level interconnect API. Examples of communication subsystems are CH3 for MPICH2, ARMCI [17] for the Global Arrays toolkit, and GASNet [5] for the Berkeley UPC runtime. At the lowest layer is the interconnect library, which is usually provided by the interconnect vendor, such as GM/Myrinet [15, 3] and InfiniBand (IBA) [12].

We now describe the communication subsystems that we examine in this paper. These are ARMCI [17], GASNet [5], LAPI [13], Portals [6], and MPI-2 [10].

ARMCI – The Aggregate Remote Memory Copy Interface (ARMCI) library [17] is the communication subsystem for Global Arrays.

GASNet – GASNet [5] is designed to support parallel global address space (GAS) languages and is the communication subsystem for the Berkeley UPC runtime system.

LAPI – IBM's Low-level Application Programming Interface (LAPI) [13] is a proprietary message-passing API that provides RMA and active message operations.

Portals – Portals is the communication subsystem for the CPlant project at Sandia National Laboratories [20]. The main focus of the design of Portals [6] was to support MPI.

MPI-2 – MPI-2 [10] is an extension of the original Message Passing Interface (MPI) standard, providing one-sided remote-memory operations.

While MPI-2 can be considered a programming model middleware, we are including it as a communication subsystem because it has many features that would make it an attractive candidate for a common communication subsystem. In fact, for this reason, some communication subsystems, such as GASNet, have been implemented on top of an MPI implementation.

3 Design Features for Communication Subsystems

In this section, we identify important features in designing communication subsystems to support different programming models. We also discuss how each communication subsystem described above either supports or fails to support the features. We concentrate on the features required by MPI-2, Global Arrays, and UPC.

It is not our intention to decide whether one communication subsystem is better than another, as each is very well suited for the particular purpose for which it was written. Rather, we want to demonstrate that none of these communication subsystems is able to efficiently support all of the programming models.

We make certain assumptions about the architecture. First, we assume that the interconnect is reliable. This is a reasonable assumption as most modern networks provide reliable message delivery. Second, we assume that the system is cache-coherent. Although the MPI-2 RMA model was designed to support non-cache-coherent systems, we won't consider those in this paper.

We divide this section into two parts: required features and desired features. Communication subsystems that lack a required feature cannot effectively support a particular programming model. Desired features are features that, when implementing a programming model on top of the communication subsystem, make the implementation simpler or more efficient.

3.1 Required Features

3.1.1 Remote Memory Access Operations.

RMA operations allow a process to transfer data between its local memory and the memory of another remote process without involving that remote process. These operations are especially important for global address space and remote-memory programming models, as well as for message-passing applications that have irregular communication patterns.

Important RMA operations are *Put*, *Get*, and *Accumulate*. In a Put operation data is transferred from the initiating process's memory to the target process's memory. Data is transferred in the opposite direction in a Get operation. An Accumulate operation is similar to a Put operation, except an arithmetic operation is performed on the incoming data and the data is stored at the target buffer.

It is important for RMA operations to be nonblocking to allow for better overlap of communication and computation. A *fence* operation is also needed to ensure that RMA operations have completed at a particular remote process. Also useful is a global fence operation, which would be collective and ensure that all RMA operations have completed at every process.

All of the communication subsystems we examined support RMA operations; however, some impose restrictions on the memory that can be used for those operations. We identify four classes of RMA memory from most restrictive to least restrictive.

1. RMA memory defined collectively at initialization time

2. RMA memory defined collectively at any time during execution

3. RMA memory defined noncollectively at any time during execution

4. All of process memory is RMA memory

ARMCI supports the second class of RMA memory. Memory that is accessible by RMA operations must be allocated by using a collective memory allocation function.

GASNet supports the third class of RMA memory, when compiled with the `GASNET_SEGMENT_FAST` and `GASNET_SEGMENT_LARGE` flags. These flags allow for optimized implementations of GASNet but restrict the RMA memory to that which individual processes can allocate from a pool of memory set aside at initialization. The size of this pool is passed as a parameter at initialization. If GASNet is compiled with the `GASNET_SEGMENT_EVERYTHING` flag, it would support the fourth class. But this flexibility may have a performance penalty in some implementations.

MPI-2 also supports the third class of RMA memory. MPI-2 RMA operations are either *active target* or *passive target* operations. In active target operations, the target process is actively involved in performing the RMA operation; in passive target operations, the target process is not. The MPI-2 standard allows the implementation to require that memory used for passive target operations be allocated with a special memory allocation function.

A group of MPI processes will collectively create a *window* that identifies the memory region at each process that can be accessed by MPI-2 RMA operations. The individual memory regions do not have to be the same size and can even have zero length. This memory may be statically defined or dynamically allocated, but the allocation is not a collective operation; in other words, the processes don't all have to allocate the memory at the same time.

LAPI and Portals support the fourth class of RMA memory. RMA operations can be performed by using any process memory.

We next discuss requirements specific to supporting MPI-2 RMA features, GAS language and remote-memory models, and large two-sided messages in MPI.

3.1.2 MPI-2 RMA Features.

In order to support MPI-2 RMA operations, the communication subsystem needs to support at least the third class of RMA memory for passive-mode and the fourth class for efficient active-mode operations. This requirement means that ARMCI RMA operations cannot support MPI-2 RMA operations, because in ARMCI, RMA memory must be collectively allocated.

GASNet RMA operations cannot support MPI-2 active-mode RMA operations when compiled with `GASNET_SEGMENT_FAST` or `GASNET_SEGMENT_LARGE` flags. Only when GASNet is compiled with the `GASNET_SEGMENT_EVERYTHING` flag would it be able to support MPI-2 active-mode RMA operations. But, as mentioned before, there may be a performance penalty in some implementations.

Because both Portals and LAPI support the fourth class of RMA memory, they can support both active- and passive-mode RMA operations efficiently.

3.1.3 GAS Language and Remote-Memory Model Support.

GAS language and remote-memory model runtime systems require the ability to make concurrent conflicting RMA accesses to the same memory region. Similarly, they require the ability to make local load/store accesses to a memory region concurrently with RMA accesses. The MPI-2 standard makes such concurrent accesses erroneous. Such restrictions were added to the MPI-2 standard because MPI-2 is implemented as a library and so cannot guard against accesses of which it is unaware. However, these restrictions make MPI-2 unable to support GAS language and remote-memory models. These issues are discussed in detail in [4].

Another important feature for very fast interconnects or for shared-memory implementations is the ability to allow the source of a Put or the target of a Get operation to be a register, rather than a memory location. This avoids having to copy the value through memory.

3.1.4 Efficient Transfer of Large MPI Two-Sided Messages.

In MPI, two-sided message operations involve the sender calling `MPI_Send()` and the receiver calling `MPI_Recv()`, which matches the `MPI_Send()` from the sender. The sender specifies the source buffer, and the receiver specifies the destination buffer. In a typical MPI implementation, small messages are sent eagerly by the sender and buffered at the receiver until a matching receive is called and the destination buffer is known. This method requires that the data be copied several times, thus making it impractical for large messages.

Communication subsystems supporting MPI typically transfer data for large messages by using RMA operations.

However, RMA operations cannot be used directly because the sender doesn't know the destination address at the receiver and because the receiver doesn't know the source address at the sender. Hence, rendezvous protocol is used to send either the destination buffer information to the sender or the source buffer information to the receiver. Once the send and receive calls are matched, one side can use RMA operations to transfer the data directly between the two buffers.

Because the source and destination buffers can be located anywhere in a process's address space, in order to be able to use RMA operations to support the efficient transfer of large two-sided messages, the communication subsystem must support the fourth class of RMA memory. Only LAPI and Portals are able to support this efficiently. As noted previously, GASNet can support this class only when compiled with the `GASNET_SEGMENT_EVERYTHING` flag, which may impose a performance penalty.

Other methods can be used for transferring large two-sided messages, which most likely involve a rendezvous-like protocol internal to the communication subsystem. An example is the active message interface of LAPI. When an active message is received, a handler is called at the receiver. The handler determines the destination address and passes it to LAPI, which transfers the data directly into the buffer.

3.2 Desired Features

3.2.1 Active Messages.

Using active messages [21], the sender specifies a function to be executed at the receiver when the message is received. This handler can perform whatever processing is necessary on the message data, such as message-matching operations in MPI or an accumulate operation in Global Arrays.

Before active messages can be used, the active message handlers must be *registered*. The registration process sets up the mechanism through which the receiving process identifies which handler to execute when a message arrives. In order to allow multiple upper-layer libraries to use the same communication subsystem at the same time, each library needs to be able to register its own handlers without interfering with the other libraries.

Active messages are provided in GASNet and LAPI. GASNet requires that all handlers be registered at the same time. This requirement means that only a single library can use GASNet at a time.

3.2.2 Symmetric Allocation of Shared-Memory Regions.

In GAS languages and remote-memory model runtime systems where shared objects are allocated collectively, it would be beneficial to allow symmetrically allocated memory regions. In a symmetrically allocated region, the base addresses for the regions at each process are the same, allowing the upper layer to optimize remote pointer translation.

3.2.3 In-Order Message Delivery.

In-order message delivery is required for message-passing programming models. However, for those programming models that don't require in-order messages, a message-ordering mechanism can add a performance penalty. In fact, in some cases performance can even be improved by reordering and coalescing messages. A common communication subsystem would need to provide a way to order messages when required, but allow them to be reordered otherwise.

In MPI-2, two-sided messages are ordered, but RMA operations are not guaranteed to be ordered. LAPI, GASNet, and ARMCI do not guarantee message order. While a fence operation can be used to force ordering of messages, it would not be efficient to perform a fence after every message. Portals messages are all ordered.

3.2.4 Noncontiguous Data.

Modern interconnects, such as IBA [12] and Quadrics [19], support transferring noncontiguous data. In order to take advantage of this functionality, the communication subsystem itself must support noncontiguous data.

There are several ways that the upper layer can describe the data layout to the communication subsystem. The best method depends on how the data is actually laid out. The most general way is to use an I/O vector (IOV), which is an array of offsets and lengths, each describing the location of a piece of the data. However, the size of the description itself can grow with the length of the data and can even exceed the size of the data itself for sparse data. For specifying data that is distributed in same-sized blocks spaced evenly apart, a *strided* description can be more efficient. In a strided description the block length is specified along with the number of blocks and the distance between them. This description is more compact, but less general. *Blockindexed* describes data that is in fixed-sized blocks but not necessarily evenly distributed. A block size and an array of offsets define blockindexed data layout. For data with uniform block size, blockindexed is a more compact representation than IOV and is more general than strided. These descriptions can also be nested to describe more complex or multidimensional data distributions.

LAPI, ARMCI, and MPI-2 support transfer of noncontiguous data. In LAPI, noncontiguous data can be specified only by using the I/O vector format. ARMCI supports both IOV and strided formats. MPI-2 supports MPI *datatypes*, which allow the application to describe the data layout recursively from variations of I/O vector, strided, and blockindexed formats. GASNet and Portals support only contiguous data transfer.

3.3 Summary of Design Issues

Table 1 summarizes the features described above and indicates whether each is supported by the communication subsystems. Other issues such as supporting dynamic processes, collective communication, thread safety, and heterogeneous system support, also are important, but space limitations preclude discussion in detail. We can see from Table 1 that none of the communication subsystems we studied supports all of the features necessary for message-passing, remote-memory, and GAS language programming models. The table

Table 1: Feature summary of the communication subsystems.

	RMA operations	MPI-2 active-mode RMA	MPI-2 passive-mode RMA	GAS language support	Transfer of large MPI messages	Active messages	In-order message delivery	Noncontiguous data*	Portability
ARMCI	●			●				V, S	●
GASNet	●		●	●		●			●
LAPI	●	●	●	●	●	●		V	
Portals	●	●	●	●	●		●		●
MPI-2	●	●	●		●		●	V, S, B	●

* V = I/O vector; S = strided; B = blockindexed

also shows a column for portability. While portability is a main goal for ARMCI, GASNet, Portals, and MPI-2, LAPI is available only on IBM systems.

While the lack of some of the features we described does not necessarily mean that the middleware cannot be implemented over a particular communication subsystem, the implementations would be less efficient. In fact, MPI has been implemented over LAPI [2], UPC has been implemented over MPI [14], and MPI-2 has been implemented over GASNet [1], but these implementations are not as efficient as they could be if all of the features had been supported by the communication subsystem.

4 Discussion and Future Work

A common communication subsystem can reduce the duplicated effort to support communication subsystems for individual programming models. In addition, the development time for new middleware libraries can be reduced by building the library on top of the common communication subsystem and allowing it to take advantage of the communication subsystem's highly tuned features. In this paper we have demonstrated that no existing communication subsystem has all of the features we described. Furthermore, we have shown that there are no mutually exclusive requirements, indicating that a common communication subsystem can be implemented. We are, in fact, currently working on implementing a prototype of a common communication subsystem and have described our design in [7].

In this paper we have considered support only for programming models. High-performance parallel I/O libraries have different communication subsystem requirements from those of do programming-model libraries [9]. We intend to examine what additional features a communication subsystem would need in order to support parallel I/O libraries.

Acknowledgments

We thank Rusty Lusk, Rajeev Thakur, Rob Ross, Brian Toonen, and Guillaume Mercier for their valuable comments and suggestions.

References

[1] Argonne National Laboratory. MPICH2. http://www.mcs.anl.gov/mpi/mpich2.

[2] M. Banikazemi, R. K. Govindaraju, R. Blackmore, and D. K. Panda. Implementing efficient MPI on LAPI for IBM RS/6000 SP systems: Experiences and performance evaluation. In *Proceedings of the 13th International Parallel Processing Symposium*, pages 183–190, April 1999.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. Seizovic, and W. Su. Myrinet - A gigabit per second local area network. In *IEEE Micro*, pages 29–36, February 1995.

[4] D. Bonachea and J. Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing, SHPSEC-PACT03*, September 2003.

[5] Dan Bonachea. GASNet specification, v1.1. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.

[6] Ron Brightwell, Rolf Riesen, Bill Lawry, and A. B. Maccabe. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC)*, April 2002.

[7] Darius Buntinas and William Gropp. Designing a common communication subsystem. In *Proceedings of the 12th European Parallel Virtual Machine and Message Passing Interface Conference (Euro PVM MPI)*, 2005.

[8] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. Introduction to UPC and language specification. Technical Report CCS-TR-99-157, Center for Computing Sciences, IDA, Bowie, MD, 1999.

[9] P. H. Carns, W. B. Ligon III, R. B. Ross, and P. Wyckoff. BMI: A network abstraction layer for parallel I/O. In *Proceedings of the Workshop on Communication Architecture for Clusters (CAC05) in conjunction with the International Parallel and Distributed Processing Symposium (IPDPS05)*, April 2005.

[10] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. http://www.mpi-forum.org/docs/mpi-20.ps, Jul 1997.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[12] InfiniBand Trade Association. *InfiniBand Architecture Specification.* http://www.infinibandta.com.

[13] International Business Machines. *RSCT for AIX 5L LAPI Programming Guide*, second edition, September 2004. SA22-7936-01.

[14] Lawrence Berkeley National Laboratory and University of California, Berkeley. Berkeley UPC runtime. http://upc.lbl.gov.

[15] Myricom. The GM-2 message passing system – The reference guide to the GM-2 API. http://www.myri.com/scs/GM-2/doc/refman.pdf.

[16] J. Nieplocha, R. J. Harrison, and R. L. Littlefield. Global Arrays: A portable shared memory programming model for distributed memory computers. In *Supercomputing 94*, 1994.

[17] Jarek Nieplocha and Bryan Carpenter. ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems. *3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) of International Parallel Processing Symposium IPPS/SPDP '99*, April 1999.

[18] Pacific Northwest National Laboratory. Global Arrays toolkit. http://www.emsl.pnl.gov/docs/global/ga.html.

[19] Quadrics Supercomputers World Ltd. QsNet high performance interconnect. http://www.quadrics.com/website/pdf/qsnet.pdf.

[20] Rolf Riesen, Ron Brightwell, Lee Ann Fisk, Tramm Hudson, Jim Otto, and Arthur B. Maccabe. Cplant. In *Proceedings of the Second Extreme Linux Workshop at the 1999 USENIX Annual Technical Conference*, June 1999.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 256–266, June 1992.