Communication Libraries for Parallel-Programming-Model Runtime Systems

Andrew Stone
CS653 Mini Research Examination
stonea@cs.colostate.edu

November 9, 2010

Abstract

Due to physical limitations preventing increased CPU clock speeds, several computer scientists have predicted that parallelism will play an important role in future software development [11, 12, 15, 24]. However, programming for parallelism is often challenging. As such, parallel programming models such as the PGAS [8, 21, 25] and DARPA HPCS [17] languages have been developed to alleviate this difficulty. Integral in all of these systems is the need for tasks to communicate in order to transfer data and synchronize. For this reason parallel programming model runtime systems often use communication libraries. In this paper we examine and compare three such libraries: MPI, GASNet, and ARMCI. We compare these libraries in terms of the communication operations they include, and discuss how implementations of these libraries have been modified to work with specific architectures.

1 Introduction

Programming models are often developed in order to address programmability challenges. One challenge that many emerging programming models address is that of expressing and efficiently executing tasks in parallel. However, tasks that execute in parallel often require communication in order to synchronize and transfer data. For this reason, many runtime systems for parallel programming models leverage communication libraries. The choice of runtime system is not arbitrary, however. For a given machine and programming model, certain libraries may be more portable, efficient, or expressible [13].

In this paper we examine and compare three libraries for conducting communication in parallel programs. Specifically, we will examine the communication operations these libraries include, and discuss how implementations within each library vary. The three libraries we discuss are: ARMCI [19], GASNet [7], and MPI [18]. We chose to look at MPI due to its popularity as a parallel programming model [16]. We chose to look at GASNet and ARMCI due their intended use of being integrated into parallel programming model runtime systems. Currently, GASNet is used within Titanium [5], and Chapel [1], as well as the GCC [3] and Cray UPC [9] compilers, and the Cray [10] and Rice [2] CAF compilers. The ARMCI library is used within the Global Arrays shared memory programming toolkit [20].

2 Communication operations

Although ARMCI and GASNet are used in programming model runtime systems, and MPI is itself a programming model, there exist many similarities between the libraries. In particular, many communication operations are shared between them. In this section we discuss the commonly used operations found in communication libraries.

Something that is common among all communication operations is that when realized they involve two or more units-of-execution (UEs). UEs are either threads or processes. In the case of collective communication several UEs may be involved. In the case of point-to-point communication exactly two UEs interact: a sender and a receiver.

Although all point-to-point communication involves a sender and a receiver it is not necessarily the case that both sides will explicitly specify that the communication occurs. Depending on how many UEs specify the need for communication, point-to-point communication operations may be classified as either being one- or two- sided. In two-sided communication the sender explicitly invokes a `send` operation to send data and

the receiver explicitly invokes a `receive` operation to receive data. On the other hand, in one-sided communication an origin process may explicitly invoke a `get` operation to retrieve data from the local memory of a target UE without having the target explicitly specify that the communication should occur. The one-sided `put` operation enables an origin UE to place data into the local memory of a target UE without the need for the target to specify that such an operation should occur.

The MPI-1 standard includes several collective communication routines and several two-sided send and get operations. The MPI-2 standard and the GASNet and ARMCI libraries all include support for one-sided communication operations, however the GASNet and ARMCI libraries have one-sided communication as a stronger focus. In addition to the `put` and `get` operations MPI and ARMCI also include an `accumulate` operation that is used to combining data at a target location with an additional value: for example, a remote increment. MPI allows any of its (commutative and associative) reduction operators to serve as the combining function, for example: `sum`, `multiply`, or `max`. ARMCI, on the other hand, always uses an atomic scale-and-sum operation for its `accumulate` operation. Such an operation takes the form $x = x + \alpha * y$, where α is a scalar, and x and y may be either scalars or vectors [19]. The ARMCI library also includes a one-sided `rmw` (read-modify-write) function that will atomically update a remote integral variable using a specified operator and return the variable's old value.

Both one-sided and two-sided communication operations can be further classified by whether they are blocking or nonblocking. Blocking `send` and `put` calls will not return until the data that they are to-send can be safely overwritten. Note that this does not necessarily imply that the data has been received — a blocking call might return immediately after duplicating the data it is to-send into a network buffer. Regardless of whether a buffering scheme is used on the send-side or not, in the case of blocking `receive` and `get` calls, they will not return until they have completely received the transmitted data.

In non-buffering sends and in all receives, how quickly a blocking call returns is dependent on the amount of data that is involved, as well as the latency and bandwidth of the network. In the case of a buffering send, however, how quickly a blocking operation returns is dependent on the bandwidth and latency of the system's memory. Given that this is the case, a blocking call may impose an undesirable performance penalty. To address this issue parallel programs are often written to latency-hide, which is when a communication call executes in the background while the computation that follows the call proceeds. In order to enable latency hiding, communication libraries often include a set of non-blocking operations. Non-blocking operations are also used to avoid deadlock. For instance, in the absence of non-blocking operations, if two processes were to simultaneously post sends to each other they would endlessly stall. Such a situation could be resolved by having one process post its receive prior to its send, but in some cases it may be clearer or more efficient to use a non-blocking call; this is particularly true in more complicated arrangements where UEs must communicate with multiple other UEs that are identified as "neighbors". Regardless, the ARMCI, GASNet, and MPI libraries all provide blocking and non-blocking variants of their communication operations.

The ARMCI, GASNet, and MPI libraries also provide communication synchronization operations, which are often necessary in order to correctly implement non-blocking communication. Without the semantic guarantees of a blocking call, a program that sends an array A would not be able to safely overwrite any element of this array until the transfer has completed. Similarly, if a program received an array B from a non-blocking receive operation, it would be incorrect to access any element of array B until the transfer has completed. To determine when a transfer has completed MPI includes a `wait` operation. The `wait` function is given a handle for a prior non-blocking send-or-receive call and will block until the operation has completed. MPI also includes a `test` operation that will immediately return the status of a send or receive operation. MPI-2 and the ARMCI and GASNet libraries include analogous `fence` and `poll` operations for one-sided communication.

Communication operations may also be classified in terms of the data they send. The MPI, GASNet, and ARMCI libraries include support for sending contiguous chunks of data. The ARMCI and GASNet libraries also include I/O-vector based put and get operations. I/O vectors enable get and put operations to pull/place data from multiple, equally sized, sources/targets. ARMCI and GASNet also provides variants of their put and get procedures for strided data. Strided operations are a generalized forms of I/O vector operations who's multiple equally-sized buffers are all offset from each other by some constant amount. Strided operations are often useful for sending parts of arrays: for example, a chunk of columns of a two-dimensional array that is stored in row-major order.

3 Implementation Specific Optimizations

There are other similarities and differences between the ARMCI, GASNet, and MPI libraries. All three libraries

are similar in that they are intended to be network and computer-architecture independent. However, the implementations of libraries are often customized for specific systems. IBM, for example, provides an implementation of MPI for its SP system [6].

In [23] Saif and Parashar examine and compare IBM's proprietary SP implementation of MPI [6] against the public domain MPICH [4]. Specifically, the authors compare each implementation's behavior of the non-blocking `MPI_Isend` and `MPI_Irecv` functions. They examine this behavior when varying amounts of messages, message sizes, and system buffer sizes are employed. By varying these parameters the authors are able to identify when send and receive operations of the respective MPI versions introduce synchronous behavior. For instance, in many MPI implementations, when the message size exceeds the network-buffer size the MPI implementations will stall a send until it has received some sort of acknowledgement from the receive side. The authors discover that for small messages (1KB) both the IBM and MPICH versions will not introduce any synchronizing behavior. However, for larger messages (60KB or above) MPICH will introduce synchronizing behavior. With a large message the MPICH version will not return until either 1) the message has been completely sent, 2) the receiving side has posted a `wait` or `test` operation, or 3) a non-deterministic time out period has occurred. The purpose of introducing such a time-out period is to avoid deadlock in the case that two processes simultaneously send large messages to each other.

Saif and Parashar further describe a number of optimizations they believe will help improve the performance of programs that use non-blocking MPI calls. Specifically, they suggest increasing buffer sizes and introducing `test` calls after invoking `MPI_Irecv`. The authors demonstrate that these optimizations can have a significant performance impact by applying them to an example Structured Adaptive Mesh Refinement (SAMR) algorithm within a 3D compressible turbulence application. They measure that such optimizations result in a 27% reduction in communication time.

The reduction in time that Saif and Parashar show comes from modifying an MPI program to better fit the implementation of a library it operates on. In [14] Bonachea et al. take a different approach towards optimizing communication: they tailor an implementation of a library for a specific system. In particular, they port the GASNet library so that it operates with the Cray XT's native communication library: Portals [22].

In [19] Nieplocha and Carpenter briefly discuss optimizations that can have been integrated into the ARMCI library to tailor it to a specific machine. For example, on the IBM SP the ARMCI library will follow an owner-computes rule to perform an accumulate operation. However, on the Cray T3E the requesting processor will lock the remotely accessed memory, copy the variable(s) to accumulate to local memory, perform the accumulation operation(s), and copy the result(s) back to the remote memory.

4 Conclusions

Runtime systems for parallel programming models often use a communication library. In this paper we discussed three such libraries: MPI, GASNet, and ARMCI. All three libraries provide blocking and non-blocking, 1-sided and 2-sided, point-to-point communication routines.

The implementation of libraries can affect performance. For example, in [23] Saif and Parashar examine how non-blocking communication operations can affect performance; in [19] Nieplocha and Carpenter discuss different implementations of the ARMCI library are tailored for shared and distributed memory machines; and in portals Bonachea et al. discuss how they have tailored the GASNet communication library for a Cray XT.

Given this situation different libraries may be more or less applicable than others for different programming models and machines.

References

[1] Chapel programming language homepage. `http://chapel.cs.washington.edu`.

[2] Co-array fortran at rice university. `http://caf.rice.edu/`.

[3] Gcc upc (gcc unified parallel c). `http://www.intrepid.com/upc.html`.

[4] Mpich - a portable mpi implementation. `http://www.mcs.anl.gov/research/projects/mpi/mpich1-old/`.

[5] Titanium project home page. `http://titanium.cs.berkeley.edu`.

[6] Parallel environment (pe) for aix v3r2.0: Operation and use, vol. 1, December 2001.

[7] Gasnet specification, v1.1. Technical Report UCB/CSD-02-1207, CS Division, EECS Department; University of California, Berkeley, 2002.

[8] Upc language specifications, v1.2. Technical Report LBNL-59208, The UPC Consortium, Lawrence Berkeley National Lab, 2005.

[9] Cray inc. cray c and c++ reference manual. *http://docs.cray.com/books/S-2179-70*, (Publication number S-2179-70), 2008.

[10] Cray inc. cray fortran reference manual. *http://docs.cray.com/books/S-3901-70*, (Publication number S-3901-70), 2008.

[11] T. Agerwala and S. Chatterjee. Computer architecture: Challenges and opportunities for the next decade. *IEEE Micro*, 25(3):58–69, 2005.

[12] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[13] D. Bonachea and J. Duell. Problems with using mpi 1.1 and 2.0 as compilation targets for parallel language implementations. In *in 2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC-03)*, pages 91–99, 2003.

[14] D. Bonachea, P. H. Hargrove, M. Welcome, and K. Yelick. Porting gasnet to portals: Partitioned global address space (pgas) language support for the cray xt. In *In Cray Users Group*, 2009.

[15] J. Dongarra, D. Gannon, G. Fox, and K. Kennedy. The impact of multicore on computational science software. *CTWatch Quarterly*, 3(1), February 2007.

[16] W. Gropp. Learning from the success of mpi. In *HiPC '01: Proceedings of the 8th International Conference on High Performance Computing*, pages 81–94, London, UK, 2001. Springer-Verlag.

[17] E. Lusk and K. Yelick. Languages for high-productivity computing: the DARPA HPCS Language Project. *Parallel Processing Letters*, 17(1):89–102, Mar. 2007.

[18] Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[19] J. Nieplocha and B. Carpenter. Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems. In *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 533–546. Springer Berlin / Heidelberg, 1999.

[20] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996. 10.1007/BF00130708.

[21] R. W. Numrich and J. Reid. Co-arrays in the next fortran standard. *SIGPLAN Fortran Forum*, 24:4–17, August 2005.

[22] R. Riesen, R. Brightwell, K. Pedretti, A. B. Maccabe, and T. Hudson. The portals 3.3 message passing interface. Technical Report SAND2006-0420, Sandia National Laboratories, 2006.

[23] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in mpi. In *Euro-Par 2004 Parallel Processing*, volume 3149 of *Lecture Notes in Computer Science*, pages 173–182. Springer Berlin / Heidelberg, 2004.

[24] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[25] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998.