TEG: A High-Performance, Scalable, Multi-Network Point-to-Point Communications Methodology

T.S. Woodall[1], R.L. Graham[1], R.H. Castain[1], D.J. Daniel[1], M.W. Sukalski[2], G.E. Fagg[3], E. Gabriel[3], G. Bosilca[3], T. Angskun[3], J.J. Dongarra[3], J.M. Squyres[4], V. Sahay[4], P. Kambadur[4], B. Barrett[4], A. Lumsdaine[4]

[1] Los Alamos National Lab,
[2] Sandia National Laboratories,
[3] University of Tennessee,
[4] Indiana University

Abstract. TEG is a new component-based methodology for point-to-point messaging. Developed as part of the Open MPI project, TEG provides a configurable fault-tolerant capability for high-performance messaging that utilizes multi-network interfaces where available. Initial performance comparisons with other MPI implementations show comparable ping-pong latencies, but with bandwidths up to 30% higher.

1 Introduction

The high-performance computing arena is currently experiencing two trends that significantly impact the design of message-passing systems: (a) a drive towards petascale class computing systems that encompass thousands of processors, possibly spanning large geographic regions; and (b) a surge in the number of small clusters (typically involving 32 or fewer processors) being deployed in a variety of business and scientific computing environments. Developing a message-passing system capable of providing high performance across this broad size range is a challenging task that requires both an ability to efficiently scale with the number of processors, and dealing with fault scenarios that typically arise on petascale class machines.

A new point-to-point communication methodology (code-named "TEG") has been designed to meet these needs. The methodology has been developed as part of the Open MPI project [3] —an ongoing collaborative effort by the Resilient Technologies Team at Los Alamos National Lab, the Open Systems Laboratory at Indiana University, and the Innovative Computing Laboratory at the University of Tennessee to create a new open-source implementation of the Message Passing Interface (MPI) standard for parallel programming on large-scale distributed systems [4,7]. This paper describes the concepts underlying TEG, discusses its implementation, and provides some early benchmark results.

A rather large number of MPI implementations currently exist, including LAM/MPI [1], FT-MPI [2], LA-MPI [5], MPICH [6], MPICH2 [8], the Quadrics version of MPICH [10], Sun's MPI [12], and the Virtual Machine Interface (VMI) 2.0 from NCSA [9]. Each of these implementations provides its own methodology for point-to-point communications, resulting in a wide range of performance characteristics and capabilities:

- LAM/MPI's "Component Architecture" enables the user to select from several different point-to-point methodologies at runtime;
- Sun's and Quadrics' MPIs both will stripe a single message across multiple available similar Network Interface Cards (NICs).
- MPICH2 has a design intended to scale to very large numbers of processors.
- VMI is designed to stripe a single message across heterogeneous NICs, giving it the ability to transparently survive the loss of network resources with out any application modification.
- FT-MPI is designed to allow MPI to recover from processor loss with minimal overhead.
- LA-MPI is designed to stripe a single message across identical NIC's, different messages across different NIC types, recover transparently from loss of network resources, and allows for optional end-to-end data integrity checks.

TEG represents an evolution of the current LA-MPI point-to-point system that incorporates ideas from LAM/MPI and FT-MPI to provide Open MPI with an enhanced set of features that include: (a) fault tolerant message passing in the event of transient network faults—in the form of either dropped or corrupt packets— and NIC failures; (b) concurrent support for multiple network types (e.g. Myrinet, InfiniBand, GigE); (c) single message fragmentation and delivery utilizing multiple NICs, including different NIC types, such as Myrinet and InfiniBand; and (d) heterogeneous platform support within a single job, including different OS types, different addressing modes (32 vs 64 bit mode), and different endianess.

The remainder of this paper describes the design of the TEG point-to-point messaging architecture within Open MPI, the component architecture and frameworks that were established to support it, and the rationale behind various design decisions. Finally, a few results are provided that contrast the performance of TEG with the point-to-point messaging systems found in other production quality implementations. An accompanying paper provides detailed results [13].

2 Design

The Open MPI project's component-based architecture is designed to be independent of any specific hardware and/or software environment, and to make it relatively easy to add support for new environments as they become available in the market. To accomplish this, Open MPI defines an MPI Component Architecture (MCA) framework that provides services separating critical features into individual components, each with clearly delineated functional responsibilities and interfaces. Individual components can then be implemented as plug-in-modules, each representing a particular implementation approach and/or supporting a particular environment. The MCA provides for both static linking and dynamic (runtime) loading of module binaries, thus creating a flexible system that users can configure to meet the needs of both their application and specific computing environment.

Open MPI leverages the MCA framework to map the point-to-point communication system into two distinct components:

1. Point-to-point Management Layer (PML). The PML, the upper layer of the point-to-point communications design, is responsible for accepting messages from the MPI layer, fragmenting and scheduling messages across the available PTL modules, and managing request progression. It is also the layer at which messages are reassembled on the receive side. MPI semantics are implemented at this layer, with the MPI layer providing little functionality beyond optional argument checking. As such, the interface functions exported by the PML to the MPI layer map closely to the MPI-provided point-to-point API.
2. Point-to-point Transport Layer (PTL). At the lower layer, PTL modules handle all aspects of data transport, including making data available to the low level network transport protocols (such as TCP/IP and InfiniBand), receiving data from these transport layers, detection of corrupt and dropped data, ACK/NACK generation, and making decisions as to when a particular data route is no longer available.

At startup, a single instance of the PML component type is selected (either by default or as specified by the user) by the MCA framework and used for the lifetime of the application. All available PTL modules are then loaded and initialized by the MCA framework. The selected PML module discovers and caches the PTLs that are available to reach each destination process. The discovery process allows for the exclusive selection of a single PTL, or provides the capability to select multiple PTLs to reach a given endpoint. For example, on hosts where both the shared-memory and the TCP/IP PTLs are available, the PML may select both PTLs for use, or select the shared-memory PTL exclusively. This allows for a great deal of flexibility in using available system resources in a way that is optimal for a given application.

Another related simplification over prior work includes migrating the scheduling of message fragments into the PML. While LA-MPI supports delivering a message over multiple NICs of the same type, this functionality was implemented at the level corresponding to a PTL, and thus did not extend to multi-network environments. Migrating this into the PML provides the capability to utilize multiple networks of different types to deliver a single large message, resulting in a single level scheduler that can be easily extended or modified to investigate alternate scheduling algorithms.

2.1 PML: TEG

TEG is the first implementation of the Open MPI PML component type. The following sections provide an overview of the TEG component, focusing on aspects of the design that are unique to the Open MPI implementation.

Send Path. TEG utilizes a rendezvous protocol for sending messages longer than a per-PTL threshold. Prior experience has shown that alternative protocols such as eager send can lead to resource exhaustion in large cluster environments and complicate the matching logic when delivering multiple message fragments concurrently over separate paths. The use of a rendezvous protocol—and assigning that functionality into the PML (as opposed to the PTL)—eliminates many of these problems.

To minimize latency incurred by the rendezvous protocol, TEG maintains two lists of PTLs for each destination process: a list of low latency PTLs for sending the first fragment, and a second list of available PTLs for bulk transfer of the remaining fragments. This is done to provide both low latency for short messages as well as high bandwidth for long messages, all in a single implementation.

To initiate a send, the TEG module selects a PTL in round-robin fashion from the low-latency list and makes a down call into the PTL to allocate a send management object, or "send request descriptor". The request descriptor is obtained from the PTL such that, once the request descriptor has been fully initialize for the first time, all subsequent resources required by the PTL to initiate the send can be allocated in a single operation, preferably from a free list. This typically includes both the send request and first fragment descriptors.

Request descriptors are used by the PML to describe the operation and track the overall progression of each request. Fragment descriptors, which describe the portion of data being sent by a specific PTL, specify the offset within the message, the fragment length, and (if the PTL implements a reliable data transfer protocol) a range of sequence numbers that are allocated to the fragment on a per-byte basis. A single sequence number is allocated for a zero length message.

Once the required resources at the PTL level have been allocated for sending a message, Open MPI's data type engine is used to obtain additional information about the data type, including its overall extent, and to prepare the data for transfer. The actual processing is PTL specific, but only the data type engine knows if the data is contiguous or not, and how to split the data to minimize the number of conversions/memory copy operations.

If the overall extent of the message exceeds the PTLs first fragment threshold when the PML initiates the send operation, the PML will limit the first fragment to the threshold and pass a flag to the PTL indicating that an acknowledgment is required. The PML will then defer scheduling the remaining data until an acknowledgment is received from the peer. Since the first fragment must be buffered at the receiver if a match is not made upon receipt, the size of this fragment is a compromise between the memory requirements to buffer the fragment and the desire to hide the latency of sending the acknowledgment when a match is made. For this reason, this threshold may be adjusted at runtime on a per-PTL basis.

The data-type engine can begin processing a message at any arbitrary offset into the data description and can advance by steps of a specified length. If the data is not contiguous, additional buffers are required to do a pack operation—these will be allocated in concert with the PTL layer. Once the buffer is available, the data-type engine will do all necessary operations (conversion or just memory copies), including the computation of the optional CRC if required by the PTL layer. Finally, the PTL initiates sending the data. Note that in the case of contiguous data, the PTL may choose to send the data directly from the users buffer without copying.

Upon receipt of an acknowledgment of the first fragment, the PML schedules the remaining message fragments across the PTLs that are available for bulk transfer. The PML scheduler, operating within the constraints imposed by each PTL on minimum and maximum fragment size, assigns each PTL a fraction of the message (based on the weighting assigned to the PTL during startup) in a round-robin fashion.

As an example, assume that two processors can communicate with each other using an OS-bypass protocol over 4X InfiniBand (IB) and TCP/IP over two Gigabit-Ethernet (GigE) NICs, and that a run-time decision is made to use all of these paths. In this case, the initial fragment will be sent over the low latency IB route, and—once the acknowledgment arrives—the remaining part of the message will be scheduled over some combination of both the IB and GigE NICs.

Receive Path. Matching of received fragments to posted receives in Open MPI is event-based and occurs either on receipt of an envelope corresponding to the first fragment of a message, or as receive requests are posted by the application. On receipt of an envelope requiring a match, the PTL will attempt to match the fragment header against posted receives, potentially prior to receiving the remainder of the data associated with the fragment. If the match is made and resources are available, an acknowledgment can be sent back immediately, thereby allowing for some overlap with receiving the fragment's data. In the case where multiple threads are involved, a mutex is used to ensure that only one thread at a time will try and match a posted receive or incoming fragment.

If the receive is not matched, a receive fragment descriptor is queued in either a per-peer unexpected message or out-of-order message list, depending upon the sequence number associated with the message. As additional receives requests are posted, the PML layer checks the unexpected list for a match, and will make a down call into the PTL that received the fragment when a match is made. This allows the PTL to process the fragment and send an acknowledgment if required.

Since messages may be received out of order over multiple PTLs, an out-of-order fragment list is employed to maintain correct MPI semantics. When a match is made from the unexpected list, the out-of-order list is consulted for additional possible matches based on the next expected sequence number. To minimize the search time when wildcard receives are posted, an additional list is maintained of pointers to lists that contain pending fragments.

When acknowledgments are sent after a match is made, a pointer to the receive descriptor is passed back in the acknowledgment. This pointer is passed back to the destination in the remaining fragments, thereby eliminating the overhead of the matching logic for subsequent fragments.

Request Progression. TEG allows for either asynchronous thread-based progression of requests (used to minimize the CPU cycles used by the library), a polling mode implementation (if threading is not supported or desired), or a hybrid of these. When thread support is enabled, no down calls are made into the PTL to progress outstanding requests. Each PTL is responsible for implementing asynchronous progression. If threading is not enabled, the PML will poll each of the available PTL modules to progress outstanding requests during MPI test/wait and other MPI entry points.

As fragments complete at the PTL layer, an upcall into the PML is required to update the overall status of the request and allow the user thread to complete any pending test/wait operations depending upon completion of the request.

2.2 PTL: TCP/IP

The first implementation of an Open MPI PTL module is based on TCP/IP to provide a general network transport with wide availability. The TCP PTL is also fairly simple, as the TCP/IP stack provides for end-to-end reliability with data validation in main memory by the host CPU, and as such does not require the PTL layer to guarantee reliability itself. It also provides in-order delivery along a single connection.

Initialization. During initialization, the TCP module queries the list of kernel interfaces that are available, and creates a TCP PTL instance for each exported interface. The user may choose to restrict this to a subset of the available interfaces via optional run-time parameters. Interface discovery includes determining the bandwidth available for each interface, and exporting this from the PTL module, so that an accurate weighting can be established for each PTL.

During module initialization, a TCP listen socket is created at each process, and the set of endpoints {address:port} supported by each peer are published to all other peers via the MCA component framework, which utilizes Open MPI's out-of-band (OOB) messaging component to exchange this information, when MPI_COMM_WORLD is created.

Connection Establishment. Connections to peers are deferred until the PML attempts to utilize the PTL to deliver a fragment. When this occurs, the TCP PTL queues the fragment descriptor, and initiates a connection. If the connection completes successfully, any queued fragment descriptors are progressed as the connection becomes available. If the connection fails, an upcall into the PML indicates that the fragment could not be delivered, and should be sent via an alternate PTL. When this occurs, the PTL with the failure will be removed from the list of PTLs used to reach this destination, after several attempts to reconnect fail.

Event Based Progress. The TCP PTL utilizes non-blocking I/O for all socket connect/send/receive operations and an event based model for progressing pending operations. An event dispatching library, libevent [], is utilized to receive callbacks when registered file descriptors become available for send or receive. The library additionally provides an abstraction layer over multiple O/S facilities for I/O multiplexing, and by default enables the most efficient facility available from /dev/epoll (Linux), BSD kernel queues, Real-Time signals, poll(), or select().

When the PML/PTLs are operating in an asynchronous mode, a separate thread is created to dispatch events, which blocks on the appropriate system call until there is activity to dispatch. As the set of file descriptors of interest may change while this dispatch thread is asleep in the kernel, for some facilities (e.g. select/poll) it is necessary to wake up the dispatch thread on changes to the descriptor set. To resolve this issue without polling, a pipe is created and monitored by the event library such that the file descriptor associated w/ the pipe can be signaled where there are changes.

2.3 Results

This section presents a brief set of results, including latency (using a ping-pong test code and measuring half round-trip time of zero byte message) and single NIC bandwidth results using NetPIPE v3.6 [11]. We compare initial performance from Open MPI using the TEG module with data from FT-MPI v1.0.2, LA-MPI v1.5.1, LAM/MPI v7.0.4, and MPICH2 v0.96p2. Experiments were performed on a two processor system based on 2.0GHz Xeon processors sharing 512kB of cache and 2GB of RAM. The system utilized a 64-bit, 100MHz, PCI-X bus with two Intel Pro/1000 NICs (based on the

Super P4Dp6, Intel E7500 chipset), and one Myricom PCI64C NIC running LANai 9.2 on a 66MHz PCI interface. A second PCI bus (64-bit, 133MHz PCI-X) hosted a second, identical Myricom NIC. The processors were running the Red Hat 9.0 Linux operating system based on the 2.4.20-6smp kernel.

Table 1 compares the resulting measured latencies and peak bandwidths. As the table indicates, the initial TCP/IP latency of Open MPI/TEG over Myrinet is comparable to that of the other MPI's. The event based progression, which does not consume CPU cycles while waiting for data, gives slightly better results than the polling mode, but this is within the range of measurement noise. We expect to spend some more effort on optimizing TEG's latency.

Implementation	Myrinet Latency (μs)	Peak Bandwidth (Mbps)
Open MPI/TEG (Polled)	51.5	1855.92
Open MPI/TEG (Threaded)	51.2	1853.00
LAM7	51.5	1326.13
MPICH2	51.5	1372.49
LA-MPI	51.6	1422.40
FT-MPI	51.4	1476.04
TCP	–	1857.95

Table 1. Open MPI/TEG Single NIC latency and peak bandwidth measurements compared to other MPI implementations (non-blocking MPI semantics).

The third column of the table contrasts the peak bandwidth of raw TCP, Open MPI (utilizing the TEG module), and other MPI implementations over a single NIC using Myrinet. With Open MPI/TEG, we measured the bandwidth using both single-threaded polling for message passing progress and a separate thread for asynchronous message passing progress. These yield almost identical performance profiles, peaking out just above 1800 Mb/sec.

The peak bandwidths of Open MPI/TEG are almost identical to TCP, indicating that very little overhead has been added to TCP. However, the other four implementations show noticeable performance degradation for large message sizes, with Open MPI/TEG peaking out at least 30% above them.

3 Summary

TEG is a new MPI point-to-point communications methodology implemented in Open MPI that extends prior efforts to provide a design more suitable for heterogeneous network and OS environments. TEG also addresses fault tolerance for both data and process, issues that are expected to arise in the context of petascale computing. Initial performance data shows ping-pong latencies comparable to existing high performance MPI implementations, but with bandwidths up to 30% higher. Future work will continue to investigate the wide range of runtime capabilities enabled with this new design.

4 Acknowledgments

This work was supported by a grant from the Lilly Endowment, National Science Foundation grants 0116050, EIA-0202048, EIA-9972889, and ANI-0330620, and Department of Energy Contract DE-FG02-02ER25536. Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the United States Department of Energy under contract W-7405-ENG-36. Project support was provided through ASCI/PSE and the Los Alamos Computer Science Institute, and the Center for Information Technology Research (CITR) of the University of Tennessee.

References

1. G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.
2. Graham E. Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Antonin Bukovski, and Jack J. Dongarra. Fault tolerant communication library and applications for high perofrmance. In *Los Alamos Computer Science Institute Symposium*, Santa Fee, NM, October 27-29 2003.
3. E. Garbriel, G.E. Fagg, G. Bosilica, T. Angskun, J. J. Dongarra J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, and T.S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.
4. A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the Message-Passing Interface. In *Euro-Par '96 Parallel Processing*, pages 128–135. Springer Verlag, 1996.
5. R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalksi. A network-failure-tolerant message-passing system for terascale clusters. *International Journal of Parallel Programming*, 31(4), August 2003.
6. 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.
7. Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.
8. Mpich2, argonne. `http://www-unix.mcs.anl.gov/mpi/mpich2/`.
9. S. Pakin and A. Pant. . In *Proceedings of The 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.
10. Quadrics, llc web page. `http://www.quadrics.com/`.
11. Q.O. Snell, A.R. Mikler, and J.L. Gustafson. In *IASTED International Conference on Intelligent Information Management and Systems*, June 1996.
12. Sun, llc web page. `http://www.sun.com/`.
13. T.S. Woodall, R.L. Graham, R.H. Castain, D.J. Daniel, M.W. Sukalsi, G.E. Fagg, E. Garbriel, G. Bosilica, T. Angskun, J. J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine. Open mpi's teg point-to-point communications methodology : Comparison to existing implementations. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.