Design and Implementation of Open MPI over Quadrics/Elan4[*]

Weikuan Yu[†] Tim S. Woodall[‡] Rich L. Graham[‡] Dhabaleswar K. Panda[†]

Network-Based Computing Lab[†]
Dept. of Computer Sci. & Engineering
The Ohio State University
{*yuw,panda*}*@cse.ohio-state.edu*

Advanced Computing Laboratory[‡]
Computer & Computation Sci. Division
Los Alamos National Laboratory
{*twoodall,rlgraham*}*@lanl.gov*

Abstract

Open MPI *is a project recently initiated to provide a fault-tolerant, multi-network capable implementation of MPI-2 [16], based on experiences gained from FT-MPI [7], LA-MPI [10], LAM/MPI [23], and MVAPICH [18] projects. Its initial communication architecture is layered on top of TCP/IP. In this paper, we have designed and implemented Open MPI point-to-point layer on top of a high-end interconnect, Quadrics/Elan4 [21]. The restriction of Quadrics static process model has been overcome to accommodate the requirement of MPI-2 dynamic process management. Quadrics Queued-based Direct Memory Access (QDMA) and Remote Direct Memory Access (RDMA) mechanisms have been integrated to form a low-overhead, high-performance transport layer. Lightweight asynchronous progress is made possible with a combination of Quadrics chained event and QDMA mechanisms. Experimental results indicate that the resulting point-to-point transport layer is able to achieve comparable performance to Quadrics native QDMA operations, from which it is derived. Our implementation provides an MPI-2 compliant message passing library over Quadrics/Elan4 with a performance comparable to MPICH-Quadrics.*

[*]This research is supported by a grant from Los Alamos National Laboratory. 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.

1. Introduction

Parallel computing architecture has recently evolved into large scale systems with tens of thousands of processors [1] or geographically distributed clusters [8]. These emerging computing environment leads to dramatically different challenges and requirements which include not only the traditional crave for low latency and high bandwidth but also the need for fault-tolerant message passing, scalable I/O support, and fault-tolerant process control. Open MPI [9] is a recent project initiated not only as a research forum to address these new challenges, but also as a development effort to produce a new MPI-2 [16] implementation.

To support a wide range of parallel platforms, Open MPI has designed its communication architecture as two separate abstraction layers: a device-neutral message management layer and a network-specific transport layer. The latter is referred to as point-to-point transport layer (PTL) and the former as point-to-point management layer (PML). Work in [24] has demonstrated that PML is able to satisfactorily aggregate bandwidth across multiple network interfaces. The TCP/IP based network communication incurs significant operating system overhead and also multiple data copies [6]. It would be better to take advantage of user-level communication protocols [3] over high-end interconnects to expose their maximum hardware capabilities. However, there are semantics differences and mismatches between the upper layers of Open MPI communication architecture and the lower level communication protocols of high-end interconnects. It is necessary to have an in-depth examination of the particular requirements of Open MPI [9, 24] PTL interface and the specific constraints of any new interconnect.

In this paper, we take on the challenges to provide a new design of Open MPI [9] point-to-point transport layer over Quadrics/Elan4 [21, 2]. First, we start with characterizing communication requirements imposed by Open MPI design objectives, including process initiation, integrating RDMA capabilities of different networks and asynchronous communication process. Then we describe the motivation and objectives of the PTL implementation over Quadrics/Elan4. Salient strategies are proposed to overcome these challenges by taking advantage of Quadrics Queued-based Direct Memory Access (QDMA) and Remote Direct Memory Access (RDMA) operations, as well as its chained event mechanism. Experimental results indicate that the implemented point-to-point transport layer achieves comparable performance to Quadrics native QDMA interface, from which it is derived. This point-to-point transport layer provides a high performance implementation of MPI-2 [16] compliant message passing over Quadrics/Elan4, achieving a performance slightly lower but comparable to that of MPICH-QsNet^{II} [21].

The rest of the paper is presented as follows. In the next section, we describe in detail the communication architecture of Open MPI [9] and its requirements to the point-to-point [24] transport layer. Section 3 provides the motivation and objectives of this work. The design of a transport protocol over Quadrics/Elan4 [21, 2] is discussed in section 4. Section 5 provides performance results. Section 6 provides a brief review of related works. Section 7 concludes the paper.

2. Overview of Open MPI Communication Architecture

Open MPI's component-based architecture [9] is designed to provide services separating critical features into individual components, each with its own functionalities and interfaces. In this section, we provide a brief overview of the components relevant to Open MPI communication architecture. For the convenience of discussion, we present the layering of the related components based on the communication flow path. This can be slightly different from the layering presented in other literatures [9, 24], where the emphasis is given to how the components are related from the perspective of software engineering.

2.1. Open MPI Communication Stack

The basic Open MPI [9] communication architecture is mapped onto two distinct components: Point-to-point Management Layer (PML) and Point-to-point Transport Layer (PTL). As shown in Fig. 2, Open MPI [15] point-to-point communication is layered directly on top of the PML interface, which provides generic functionalities of message management, such as handling application requests, fragmenting, scheduling and reassembling messages, as well as monitoring the progresses. Currently, collective communication is provided as a separated component on top of point-to-point communication. Further research will exploit the benefits of hardware-based collective support [26, 13]. At the lower layer, the PTL component is responsible for managing connection and communication status, delivering packets over a specific network interface, and updating the PML layer about packet progression.

2.2. PTL Interface and Communication Flow Path

The PTL layer provides two abstractions: the PTL component and the PTL module [9]. A PTL component encapsulates functionalities of a particular network; a PTL module represents an "instance" of a communication endpoint, typically one per network interface card. In order to join and disjoin from the pool of available PTLs, a PTL has to go through five major stages of actions: opening, initializing, communicating, finalizing and closing. These stages of PTL utilization are discussed below:

Joining the Communication Stack – A PTL component goes through the first two stages: opening and initializing, to join the communication stack. A PTL component is to be opened as a shared library. When it is loaded successfully, a process initializes the network interfaces, prepares memory and computing (e.g., additional threads) resources, and fills in the correct fields of PTL modules (one per network interface). These PTL modules are then inserted in the communication stack as available PTL modules. When these procedures successfully complete, the activation of this PTL component is triggered through a component control function.

Inside the Communication Stack – PML schedules messages across a new network when the PTL component has activated its PTL modules. Fig. 2 shows a diagram of how messages are scheduled across multiple networks. When the PML layer receives a request, it schedules the first packet to one PTL based on a chosen scheduling heuristic. For large messages, this packet serves as a *rendezvous* packet to the receiver. When it is received by one of the PTLs, the receiving PTL asks the PML layer to match this packet to the pre-posted receive requests. If a match is made with a pre-posted receive request, PML calls `ptl_matched()` to

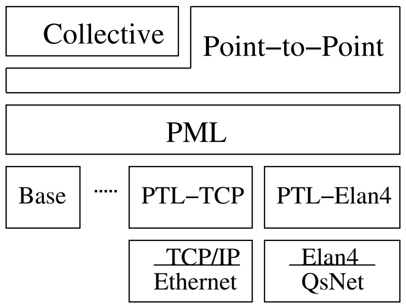


Fig. 1. The Communication Architecture of Open MPI

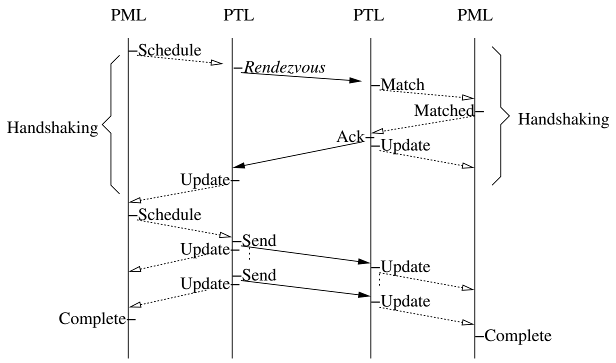


Fig. 2. Open MPI Point-to-Point Communication Flow Path

receive this message. An acknowledgment is returned to the initiating PTL if this is a *rendezvous* packet. Any data inlined with the first packet are copied into the application receive buffer, and the progress of this amount of data is `update`d at the PML layer through `ptl_recv_progress()`. When the acknowledgment arrives at the sender side, the initiating PTL updates the PML layer about the amount of data transmitted through `ptl_send_progress()`. Another scheduling heuristic is then invoked to schedule the rest of the message across available PTLs. The progress of the data transmission is updated accordingly, and this eventually leads to the completion on both sides.

Disjoining from the Communication Stack – There are also two stages to disjoin a PTL from the communication stack: finalizing and closing. During the finalizing stage, a PTL first finalizes its pending communication with other peer processes, then releases all the associated memory and computing resources. The open share library is closed after all the exposed PTL modules are finalized.

3. Objectives

Open MPI [9] has its first PTL implementation on top of the TCP/IP. Many of the strength and advantages have been described in the earlier literatures [9, 24]. In order to correctly project the objectives, it is necessary to discuss design requirements for a PTL implementation. The design of Open MPI transport layer needs to meet the requirements from three of Open MPI's main objectives, including fault tolerance, multi-network concurrent communication and dual-mode progress.

Fault Tolerance: Open MPI [9] targets at both process fault tolerance and end-to-end reliable message delivery [10]. While the latter requires PTL to be able to keep track of the progressing of individual message/packet, the former requires PTL to be prepared for itself and others to dynamically joining and disjoining the communication stack, checkpoint/restart, and etc. Each PTL has to handle not only the dynamics status of local network interface, but also dynamic connections with other PTLs.

Multi-network Concurrent Communication: Open MPI [9] scheduled messages across multiple PTLs. However, different networks have different communication semantics and requirements. In this regard, while the PML layer needs to abstract and encapsulate the difference between different PTLs, each PTL also needs to map the PML function interface on its existing transmission semantics. Section 4 provides a discussion of related design challenges over Quadrics/Elan4.

Dual-Mode Progress: Open MPI provides two different modes to monitor and progress communication across multiple network interfaces: non-blocking polling and thread-based blocking. Non-blocking polling checks the incoming and outgoing traffic of each network device in a polling manner, which can be performed by a MPI process that consists of only a single thread. In contrast, in the thread-based blocking mode additional threads are employed to block and wait on the status updates of pending messages. A PTL component needs to support thread-based blocking mode with minimum amount of memory resources and number of threads.

3.1. Overview of Quadrics/Elan4

Quadrics network [21, 20] has recently released its second generation network, QsNet^{II} [2]. This new release provides very low latency, high bandwidth communication with its two building blocks: a programmable Elan-4 network interface and the Elite-4 switch, which are interconnected in a fat-tree topology. Quadrics provides its libraries: `libelan` and `libelan4`, on top of its Elan4 network [21]. Within these default Quadrics programming libraries, a parallel job first acquires a job-wise capability. Then each process is allocated a virtual process ID (VPID), together they form a static pool of processes, i.e., the process membership and connections among them cannot change. Interprocess communication is supported by two different models: Queue-based Directed Message Access (QDMA) and Remote Directed Message Access (RDMA). QDMA allows processes to post messages (up to 2KB) to a remote queue of another process; RDMA enables processes to write messages directly into remote memory exposed by other processes. `libelan` also provides a very useful *chained event* mechanism, which allows one operation to be triggered upon the completion of another. This can be utilized to support fast and asynchronous progress of two back-to-back operations. Similar mechanisms over Quadrics/Elan3 have been utilized in [25].

3.2. Objectives of PTL Implementation Over Quadrics/Elan4

While Quadrics libraries present parallel communication over a static pool of processes, Open MPI [9, 24] targets MPI-2 [16] dynamic process management and process checkpoint/restart. The PTL implementation over Quadrics needs to support dynamic joining of PTL modules over Quadrics network. To the best of the authors' knowledge, this is not available in any existing MPI implementation over Quadrics either because the MPI implementation does not support MPI-2 dynamic process management or because the underlying communication is based on libelan's model of statically connected processes [21]. In addition, Open MPI aims for concurrent message passing over multiple networks. The communication/memory semantics can be different between networks. For example, Quadrics/Elan4 is RDMA capable while TCP/IP-based communication is not. The way RDMA works is quite different from TCP/IP sockets. To provide a high performance implementation over Quadrics/Elan4, this work has the following objectives:

1. Supporting dynamic joining of PTL modules over Quadrics
2. Integrating Quadrics RDMA capabilities into the point-to-point transport layer
3. Providing asynchronous communication progress while minimizing the performance impacts over Quadrics

4. Design of Open MPI Communication Support over Quadrics

In this section, we describe the design of Open MPI [9] transport layer over Quadrics [21]. We have proposed strategies to overcome challenges imposed by Open MPI requirements. The rest of the section describes our strategies in these aspects: (a) communication initiation and finalization, (b) integrating RDMA capabilities and (c) communication progress.

4.1. Communication Initiation and Finalization

As described in Section 3.1, Quadrics static model of processes and static connection between them do not match MPI-2 [16] dynamic process management [11] specifications. Open MPI further requires processes to be able to checkpoint/restart and migrate to a remote node on-demand or in case of faults. This model implies that the default static coupling of Quadrics virtual process ID (VPID) and the rank of a MPI process is no longer possible [21]. This is because VPID is a system identifier related to the hardware capability and the context on a specific node, while the process rank is a feature of a MPI communicator/universe that cannot change even if processes migrate. In addition, the initial global shared virtual address space over Quadrics is no longer possible because it is not guaranteed that processes are synchronized in their memory allocation when processes initiate the network and join the parallel communication at arbitrary time.

We propose to handle these challenges with the following strategies. First, we decouple the static coupling of MPI rank and Quadrics VPID in a process, leaving MPI rank for the identification of MPI processes and VPID for Quadrics network addressing. Second, for each new job, we create a Quadrics hardware capability that can provide more VPIDs than initially needed. This creates a pool of free VPIDs to be claimed by the processes spawned later. Third, we eliminate the dependence on global virtual address space for communication. For the processes that initially join parallel communication synchronously, a global virtual address space is made available. Processes that join (or rejoin) later will not be able to utilize this global address space. As a result of this, these processes cannot take advantage of the the benefits of hardware broadcast support. However, it does not preclude the possibility to regenerate a new global address space from the available address space. This possibility is to be investigated as one of further research topics in Open MPI [9]. Open MPI Run-Time Environment (RTE) can help new processes to establish connections with the existing processes. An existing connection can go through its finalization stage only when the involving processes has completed all the pending messages synchronously. This is to avoid an unpleasant scenario in which a leftover DMA descriptor might regenerate its traffic indefinitely.

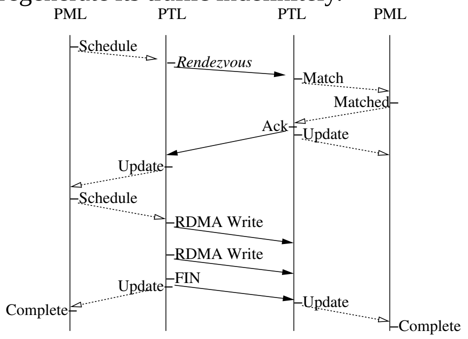


Fig. 3. Design PTL Interface with RDMA Write

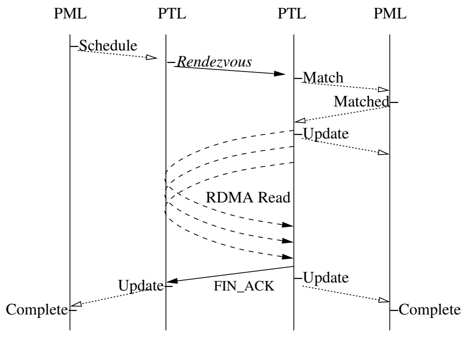


Fig. 4. Design PTL Interface with RDMA Read

4.2. Integrating Quadrics RDMA Capabilities

Quadrics QDMA communication model can only transmit messages up to 2KB. Another model, RDMA read/write, can transmit arbitrary messages. Additional support needs to be provided for integrating Quadrics RDMA capabilities into Open MPI communication architecture. There is another constraint over Quadrics to use these capabilities. For the network interface card to carry out RDMA operations, the source and destination memory addresses need to be presented in a different format (E4_Addr), which will be translated into physical memory address by a specially designed Memory Management Unit (MMU) in the Elan4 network interface.

We first modify both the memory addressing format and the communication semantics take advantage of these RDMA capabilities. The memory descriptor format is expanded to include an E4_Addr for addressing over Elan4 and other interconnects. This is only a preliminary solution for concurrent message passing over TCP and Quadrics communication protocol because both of them do not require memory registration before communication. Over other interconnects, e.g., InfiniBand [12] and Myrinet [4], the memory range of a message needs to be registered with the network interface before the communication can take place. As a part of the further research, we are experimenting with a more informative memory descriptor to support network concurrency. Second, we developed two schemes to take advantage of RDMA read and write, respectively. As shown in Fig. 3, in the first scheme, all send operations after the first *rendezvous* fragment are all replaced with RDMA write operations. At the end of these operations, a control fragment with a type FIN is sent to the receiver for the completion notification of the full message. In the second scheme, shown in Fig. 4, when the *rendezvous* packet arrives at the receiver, instead of returning an acknowledgment to the sender, the receiver initiates RDMA read operations to get the data. When these RDMA read operations complete, a different control message with a type FIN_ACK is sent to the sender, acknowledging the arrival of the earlier rendezvous fragment and notifying the message completion. For performance optimization, the transmission of the last control message can be chained to the last RDMA operation using the chained event mechanism. When the RDMA operation is done, the control message is automatically triggered without host CPU detecting the completion of RDMA operation and firing it off.

4.3. Asynchronous Communication Progress

One of Open MPI's requirements to the transport layer is asynchronous communication progress, in which it employs additional threads to monitor and progress pending messages. For the PTL implementation over TCP/IP, because one thread can block and wait on the progress of multiple sockets, it is possible to monitor the progress of all networking traffic with only a single thread per TCP PTL. However, over Quadrics, the blocking mode of the RDMA descriptor's completion is supported through separated events at different memory locations, as shown in Fig 5a. A single thread

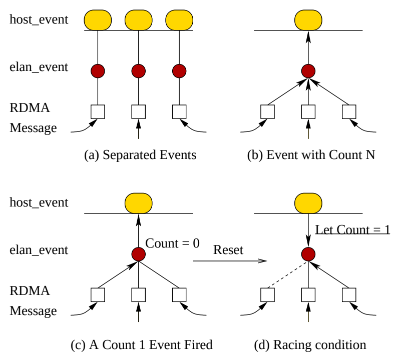


Fig. 5. Quadrics Chained Event mechanism and Possible Race Condition in Supporting Shared Completion Notification

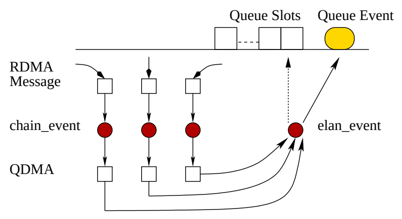


Fig. 6. Support Shared Completion Notification with QDMA and Chained Event Mechanisms

can only block and wait on the host event of a single RDMA descriptor. It is practically inhibitive to have one thread to block on each of all outstanding DMA descriptors. Quadrics provides an event mechanism that can be utilized to detect combined completion notification of multiple outstanding RDMA operations. As shown in Fig 5b, one event can be created with a count to wait on the completion of multiple outstanding RDMA operation. This count is decremented by 1 when a RDMA descriptor completes. In the end, an interrupt will be generated to the host process that is blocked on this event. This mechanism requires a predefined count. That many RDMA descriptors have to be completed before an interrupt can be triggered. With a count bigger than 1, it cannot wake up a blocking process upon the completion of individual RDMA operations. With a count of 1, the completion of the first one or the first few RDMA operations can be detected. But there is no available mechanism over Quadrics to atomically reset the event count back to 1 and block the process again for other RDMA operations to complete. This is because at the same time the other outstanding RDMA operations are potentially modifying the same event count when their messages are completed from the network, resulting in a race condition. The progressing thread may fail to detect the completion of some RDMA descriptors and not progress the communication any further. This is shown in the Figures 5c and 5d.

Quadrics QDMA [21] allows a process to check incoming QDMA messages posted by any process into its receive queue. We propose to take advantages of both QDMA shared completion queue and the chained DMA mechanism to detect multiple outstanding RDMA operations. During the PTL initialization, a receive queue is pre-created as the shared completion queue for multiple RDMA operations, shown on the right side of Fig. 6. When setting up RDMA descriptors, a QDMA operation is chained to every RDMA operation. When a RDMA operation completes, its associated chained QDMA will generate a small message to the receive queue. For every message being posted into the queue slots, an event is generated to the host side for notification. Thus with this shared completion queue, a single thread can be introduced to block and wait on the host event for the completion of many RDMA operations. This strategy of a shared completion queue is shown in Fig. 6. In terms of functionality, the new queue is the same as the pre-created receive queue for the incoming message. These two receive queues (*Two-Queue*) can be effectively combined as a single queue (*One-Queue*) to support asynchronous progress. Needless to say that using one-queue-based approach leads to a slightly more complex message handling logic, but it reduces the need for additional resources and enables the use of only one thread for asynchronous communication progress.

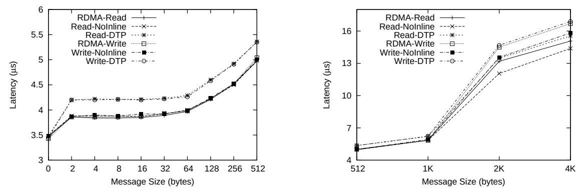
5. Performance Evaluation

We have implemented the proposed design of Open MPI transport layer over Quadrics/Elan4. Implementation details can be found in [27]. In this section, we describe the performance evaluation of our implementation. The experiments were conducted on a cluster of eight SuperMicro SUPER X5DL8-GG nodes: each with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, PCI-X 64-bit 133 MHz bus, 533MHz Front Side Bus (FSB) and a total of 1GB PC2100 DDR-SDRAM physical memory. All eight nodes are connected to a QsNet^{II} network [21, 2], with a QS-8A switch and eight Elan4 QM-500 cards.

We have first performed experiments to evaluate all of our design strategies. Then we have studied the layering overhead of Open MPI communication stacks and evaluated its overall performance compared to MPICH-QsNet^{II} [21]. Since the strategies are specific for the point-to-point message transport over Quadrics only, in all of our experiments, we have activated only the PTLs over Quadrics/Elan4 unless otherwise specified. The first 100 iterations in each test are used to warm up the network and nodes whenever applicable.

5.1. Performance Analysis of Basic RDMA Read and Write

The PML layer schedules the first packet to a PTL module based on the exposed fragment length. In the case of large size messages, this packet is composed of a *rendezvous* header with some inlined data. This strategy is beneficial to the PTL design over TCP protocol, because the cost to initiate send/receive operations through the operating system is rather high compared to the networking cost. However, with RDMA capable networks, this strategy would incur an unnecessary memory copying overhead for data in the first packet. We have provided an optimization to transmit the *rendezvous* messages without data inlined in the first packet. Note that Open MPI provides a datatype component to perform efficient packing and unpacking of sophisticated datatypes. However, it introduces some overhead because a complex copy engine is initiated with each request. For better understanding of the performance strength of the Elan4 PTL, we have intentionally replaced this copy engine with a generic `memcpy()` call.



(a) Very Small Message (b) Small Message

Fig. 7. Performance Analysis of Basic RDMA Read and Write

Fig. 7 shows both the performance of RDMA read and write with or without utilizing this datatype component, and the performance with or without inlined data. This evaluation focuses on messages up to 4KB. With the threshold of sending rendezvous messages being 1984 bytes, it allows us to look at message communication in both eager and rendezvous modes. As shown in Fig. 7(a), the data type component does introduce an overhead about 0.4μsec compared to the basic support without the datatype component. As also shown in Fig. 7(b), RDMA read is able to deliver better performance compared to RDMA write. This is to be expected because the RDMA read-based scheme essentially saves a control packet compared to RDMA write-based scheme. When using the optimization to transmit the *rendezvous* packet without inlined data, the performance is improved for all message sizes with either RDMA Read or RDMA write.

5.2. Performance Analysis with Chained DMA and Shared Completion Queue

The design of PTL/Elan4 has utilized chained DMA mechanism in two scenarios: one to notify the completion of RDMA read and RDMA write, the other to direct the completion notification of multiple RDMA operations to a shared completion queue. We have measured their performance using RDMA read as an example case. As shown in Fig. 8, using the chained DMA for fast completion notification provides only marginal improvements for the transmission of long messages. The benefits is small because the total communication time for messages $\geq 2KB$ is rather high comparing the possible benefits, i.e., automatically triggering the next DMA without I/O bus traffic. PCI-X bus and fast CPU processor (3GHz) used in the experiments also reduce the possible benefits of chained DMA. On the other hand, the shared completion queue introduces around 1μsec performance cost. This is to be expected because of the time to fire an additional QDMA operation. In addition, using either a combined receive queue (*One-Queue*) or two separated completion queues (*Two-Queue*) provides about the same performance. This is because the cost of checking the heads of two queues or one queue is about the same with the polling-based progress.

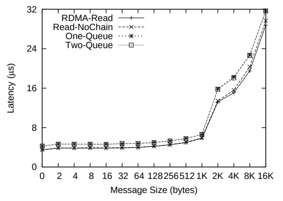


Fig. 8. Performance Analysis with Chained DMA and Shared Completion Queue

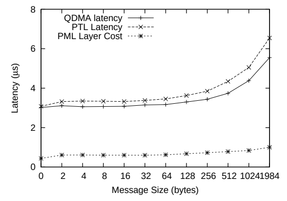


Fig. 9. Analysis of Communication Overhead in Different Layers

5.3. Analysis of Communication Time in Different Layers

We have performed an analysis of the Open MPI communication stacks. Fig. 10 shows a diagram of communication time in different layers. In a ping-pong latency test, we take the timing from a) the time when PTL/Elan4 has received a packet from the network and is delivering it to the PML layer for matching, to b) the time PTL/Elan4 receives another packet delivery request from PML layer, as the communication time above the PTL layer. This includes the communication time in the PML layer and above. We refer to it as *PML layer cost*, shown in Fig. 10 as L1. The latency at the PTL layer, *PTL Latency*, is derived by subtracting that from the overall ping-pong latency. This is shown in Fig. 10 as L2. Strictly speaking, L1 also includes the time that the PML layer notifies the PTL layer about the successful matching of a packet. We include this in the PML latency for the convenience of analysis. Because the short message transmission is based on top of Quadrics QDMA model, we also compare L2 to the performance of the native performance of QDMA, *QDMA Latency*. Note that Open MPI communication layer uses a 64-byte header for matching purpose. To achieve a fair comparison, the PTL communication time of a `N`-byte message is compared against with the communication time of a `64+N`-byte QDMA operations. As shown in Fig. 9, the PML layer and above has a communication cost of 0.5μsec, while PTL/Elan4 delivers the message with a performance comparable to QDMA.

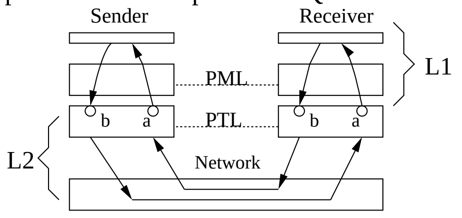


Fig. 10. Communication Time in Different Layers

5.4. Performance Analysis of Thread-Based Asynchronous Progress

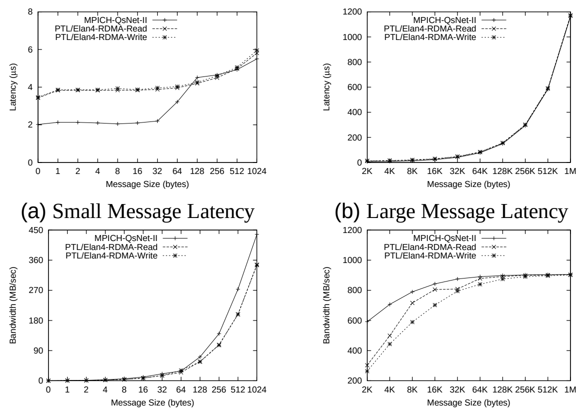
The shared completion queue is introduced to support thread-based asynchronous communication progress. We have analyzed the performance of the asynchronous progress using RDMA read in the Elan4 PTL. Table 1 shows the performance with various types of progress rules: (a) *Basic* polling-based, (b) *Interrupt*-based, (c) *One-Thread*-based, and (d) *Two-Thread*-based. Note that the interrupt-based progress is not really a workable strategy in real-time communication because the progress of MPI communication cannot block within a single PTL. It is evaluated here to find out the cost of interrupt that is used to wake up a thread. Cases (c) and (d) utilize a combined completion queue or separated completion queues, respectively. The performance results indicate that one-thread-based asynchronous progress is more efficient. The overall cost of thread-based communication progress using one-thread is around 18.9μsec, comparing its latency 22.76μsec to the 3.87μsec latency of Case (a). An interrupt costs about 10.8μsec, comparing the latency of Case (b) to that of Case (a). About 1μsec is due to the additional chained DMA, as discussed in Section 5.3. Thus around 7μsec can be attributed to the threading cost. Note that, when doing these experiments, we have left both interrupt affinity and processor affinity of the operating system at its default.

Table 1. Performance Analysis of Thread-Based Asynchronous Progress (in μsec)

Size	Basic	Interrupt	One Thread	Two Threads
4B	3.87	14.70	22.76	27.50
4KB	15.25	27.16	32.80	47.72

5.5. Overall Performance of Open MPI over Quadrics/Elan4

Fig. 11 shows the overall latency and bandwidth performance of Open MPI over Quadrics/Elan4 with the best options as described above, such as using chained DMA for completion notification, using polling-based progress without shared completion queue, and using *rendezvous* packets without inlined data. The comparison is made to the default MPI implementation MPICH-QsNet^{II}. Our implementation has a latency performance comparable to that of MPICH-QsNet^{II}, except in the range of small messages. This is due to the following reasons: (a) MPICH-QsNet^{II} transmits a shorter-header, which is 32 bytes compared to 64 bytes in Open MPI, (b) MPICH-QsNet^{II} is built on top of Quadrics T-port interface. Tport does a fast tag matching in the NIC. For very short messages, host CPU does not touch the message and its header in the critical communication path, therefore no data touching overhead [5, 6]. In terms of bandwidth, our implementation performs particularly worse in the middle range of messages. This is because Tport does efficient pipelining of messages. There is a dip of bandwidth for 2KB messages. This is because the threshold between small and rendezvous messages is 1984 bytes, the size of maximum QDMA messages minus 64 bytes of the header. Note our implementation starts from different design requirements to co-exist with PTL models of other networks and to be MPI-2 [16] compliant. For example, we are not doing NIC-based tag matching as MPICH-Quadrics^{II} does in its underlying Tport [21] interface to support MPI-1 [15] interface. Currently we intend to have shared request queues for managing traffic from different networks and allow them to be able to crosstalk. The performance of our current implementation may not be as good as, but still comes close to that of MPICH-QsNet^{II} [2].



(a) Small Message Latency (b) Large Message Latency

(c) Small Message Bandwidth (d) Large Message Bandwidth

Fig. 11. Performance Comparisons of Open MPI over Quadrics/Elan4 and MPICH-QsNet^{II}

6. Related Work

MPI [15] has been the *de facto* messaging passing standard. MPI-2 [16] extends MPI-1 with one-sided communication, dynamic process management, parallel I/O and language bindings.

Numerous implementations have been provided over different networks, including high-end RDMA capable interconnects. These include MPICH-GM [17] for Myrinet, MVAPICH [18] and MVAPICH2 [14] for InfiniBand, MPICH-QsNet [21, 2] for Quadrics elan3 and elan4 networks [21], and MPI-Sun [22] for Sun Fire links. Among them, [17, 21, 18, 14] are able to take advantage of RDMA capabilities of their underlying networks. [22] primarily relies on Programmed IO (PIO) for message passing. MPICH-NCSA [19] and LA-MPI [10] support message passing over multiple networks. However, a single message cannot be scheduled across multiple networks. Different semantics in addressing remote memory over different networks are also not addressed. LAM/MPI [23] supports part of MPI-2 interfaces, for example, dynamic process management. MVAPICH2 [14] is an MPI-2 implementation over InfiniBand. It supports both active and passive one-sided communication. MPICH-QsNet [21] and LA-MPI [10] provide MPI implementation over Quadrics network, but they do not support dynamic process management or process checkpoint/restart. Change of the membership and connections among MPI processes usually aborts the parallel job. Open MPI [9, 24] is initiated as a new MPI-2 implementation that support fault tolerant and concurrent message passing over multiple networks. This work provides a design and implementation of high performance communication of Open MPI over Quadrics/Elan4.

7. Conclusions

In this paper, we have presented the design and implementation of Open MPI [9, 24] point-to-point transport layer (PTL) over Quadrics/Elan4 [21, 2]. To match the fault tolerant design goals of Open MPI, we have designed the transport layer to allow dynamic processes over Quadrics. Our design has also integrated Quadrics RDMA capabilities into the communication model of Open MPI [9, 24]. Thread-based asynchronous communication progress is supported with a strategy using both Quadrics chained event mechanism and QDMA. Our evaluation has shown that the point-to-point transport layer implementation achieves a latency comparable to Quadrics native QDMA for small messages. Its overall performance is slightly lower but comparable to that of MPICH-QsNet^{II} [21]. Our implementation provides a MPI-2 [16] compliant message passing over Quadrics/Elan4.

In future, we intend to study the effectiveness of performance improvement with Open MPI's aggregated communication over multiple Quadrics network interfaces and across different interconnects. We also intend to study fault tolerant process management and reliable message delivery over multiple interconnects.

Acknowledgment

We would like to thank the entire Open MPI team for bringing the Open MPI project available as a research platform. In particular, we would like to thank Dr. Jeffrey Squyres from Indiana University and Dr. George Bosilca from The University of Tennessee at Knoxville for valuable discussion with them.

Additional Information – Additional information related to Open MPI and this research can be found on the following website: http://www.open-mpi.org/.

References

[1] TOP 500 Supercomputers. http://www.top500.org/.

[2] J. Beecroft, D. Addison, F. Petrini, and M. McLaren. QsNet-II: An Interconnect for Supercomputing Applications. In *the Proceedings of Hot Chips '03*, Stanford, CA, August 2003.

[3] R. Bhoedjang, T. Rhl, and H. Bal. User-Level Network Interface Protocols. *IEEE, computer*, pages 53–60, November 1998.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.

[5] H.-K. J. Chu. Zero-Copy TCP in Solaris. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996.

[6] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwn. An Analysis of TCP Processing Overhead. *IEEE Communications Magazine*, 27(6), July 1989.

[7] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, A. Bukovski, and J. 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.

[8] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[9] E. Garbriel, G. Fagg, G. Bosilica, T. Angskun, J. J. D. J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

[10] 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.

[11] W. Gropp and E. Lusk. Dynamic process management in an MPI setting. In *Proceedings of Seventh IEEE Symposium on Parallel and Distributed Processing*, pages 530–533, October 1995.

[12] Infiniband Trade Association. http://www.infinibandta.org.

[13] Jiuxing Liu and Amith Mamidala and Dhabaleswar K. Panda. Fast and Scalable Collective Operations using Remote Memory Operations on VIA-Based Clusters. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2004.

[14] J. Liu, W. Jiang, P. Wyckoff, D. K. Panda, D. Ashton, D. Buntinas, W. Gropp, and B. Toonen. Design and Implementation of MPICH2 over InfiniBand with RDMA Support. In *Proceedings of Int'l Parallel and Distributed Processing Symposium (IPDPS '04)*, April 2004.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[16] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[17] Myricom. Myrinet Software and Customer Support. http://www.myri.com/scs/GM/doc/, 2003.

[18] Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand on VAPI Layer. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html.

[19] S. Pakin and A. Pant. VMI 2.0: A Dynamically Reconfigurable Messaging Layer for Availability, Usability, and Management. In *Proceedings of The 8th International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.

[20] F. Petrini, W.-C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[21] Quadrics Supercomputers World, Ltd. Quadrics Documentation Collection. http://www.quadrics.com/onlinedocs/Linux/html/index.html.

[22] S. J. Sistare and C. J. Jackson. Ultra-High Performance Communication with MPI and the Sun Fire Link Interconnect. In *Proceedings of the Supercomputing*, 2002.

[23] J. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, Venice, Italy, September / October 2003. Springer-Verlag.

[24] T. Woodall, R. Graham, R. Castain, D. Daniel, M. Sukalsi, G. Fagg, E. Garbriel, G. Bosilica, T. Angskun, J. J. Dongarra, J. 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.

[25] W. Yu, D. Buntinas, R. L. Graham, and D. K. Panda. Efficient and Scalable Barrier over Quadrics and Myrinet with a New NIC-Based Collective Message Passing Protocol. In *Workshop on Communication Architecture for Clusters, in Conjunction with International Parallel and Distributed Processing Symposium '04*, April 2004.

[26] W. Yu, S. Sur, D. K. Panda, R. T. Aulwes, and R. L. Graham. High Performance Broadcast Support in LA-MPI over Quadrics. In *Los Alamos Computer Science Institute Symposium*, October 2003.

[27] W. Yu, T. S. Woodall, R. L. Graham, and D. Panda. Design and Implementation of Open MPI over Quadrics/Elan4. Technical Report OSU-CISRC-10/04-TR54, Columbus, OH 43210, 2004.