# Design and Implementation of Open MPI over QsNet/Elan4

Weikuan Yu    Tim S. Woodall    Dhabaleswar K. Panda    Rich L. Graham

Technical Report
OSU-CISRC-10/04-TR54

# 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, and production-quality implementation of MPI-2 [20] interface based on experiences gained from FT-MPI [8], LA-MPI [10], LAM/MPI [28], and MVAPICH [23] 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[26]. Design challenges related to dynamic process/connection management, utilizing Quadrics RDMA capabilities and supporting asynchronous communication progression are overcome with salient strategies to utilize Quadrics Queued-based Direct Memory Access (QDMA) and Remote Direct Memory Access (RDMA) operations, along with the chained event mechanism. Experimental results indicate that the resulting point-to-point transport layer implementation is able to achieve comparable performance to Quadrics native QDMA operations, from which it is derived. While not taking advantages of Quadrics/Elan4 [26, 2] NIC-based tag matching due to its design requirements, this point-to-point transport layer provides a high performance implementation of MPI-2 [20] compliant message passing over Quadrics/Elan4.*

## 1. Introduction

Parallel computing architecture has recently evolved into systems with thousands of processors. Grid and meta-computing has pushed this trend further into geographically distributed clusters. These frontiers of development have led to a less integrated computing environment with 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, resource provisioning, run-time process control and fault-tolerance.

Open MPI [9] is a recent project initiated not only as a research forum to address these new challenges comprehensively, but also as a development effort to produce an all-new, production-quality MPI-2 [20] implementation. In order to efficiently support a wide range of parallel platforms, besides presenting an MPI [19] interface, Open MPI has designed its communication architecture as two separate abstraction

*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.

layers: a device-neutral message management layer and 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). Currently, PML is co-developed with a PTL implementation on top of TCP/IP [31]. Recent work has demonstrated that PML is able to satisfactorily aggregate bandwidth across multiple network interfaces using this TCP/IP based communication protocol. Although most interconnects have its IP emulation support, network access through TCP/IP incurs significant operating system overhead and also multiple data copies. This prevents Open MPI from taking advantage of the maximum performance advantage of the underlying network. It would be better to take advantage of communication protocols over high-end interconnects to expose their maximum hardware capabilities. However, there will be semantics difference and mismatches from higher layers of Open MPI communication architecture, namely PML, as well as in between TCP/IP and a communication protocol of high-end interconnects. These likely presents challenges to a high performance design of Open MPI communication support over a particular network. Therefore, it is necessary to have an in-depth examination the particular requirements of Open MPI [9, 31] PTL interface and the specific constraints of any new interconnect.

This paper takes on these challenges and proposes a new design of Open MPI [9] point-to-point transport support over Quadrics/Elan4 [26, 2]. First, we start with characterizing communication requirements imposed from Open MPI design objectives, including process initiation, integrating RDMA capabilities of different networks and asynchronous process support. Then we describe the motivation and objectives of the PTL implementation over Quadrics/Elan4. Salient strategies are proposed to overcome these challenges by taking advantages 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 achieve comparable performance to Quadrics native QDMA interface, from which it is derived. Even though this design is not taking advantages of Quadrics/Elan4 [26, 2] NIC-based tag matching due to the design constraints from Open MPI requirements, this point-to-point transport layer provides a high performance implementation of MPI-2 [20] compliant message passing over Quadrics/Elan4, achieving a performance slight lower but comparable to that of MPICH-QsNet$^{II}$ [26].

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 [31] transport layer. Section 3 describes the motivation and objectives of this work. The design of a transport protocol over Quadrics/Elan4 [26, 2] is discussed in section 4. Section 5 provides the implementation. Section 6 provides performance results. Section 7 provides a brief review of related works. Section 8 concludes the paper.

## 2. Overview of Open MPI Communication Architecture

The Open MPI's component-based architecture [29] is designed to provide services separating critical features into individual components, each with its own functional responsibilities 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, 31, 29], 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 [29] 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. 2.1, MPI [19] point-to-point communication is layered directly on top of the PML interface, which provides the generic functionalities of message management, such as handling application requests from MPI [19], fragmenting and scheduling messages across available PTL modules, reassembling messages at the receiver side and monitoring the progress of requests. 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 [33, 17]. At the lower layer, the PTL component is responsible for managing network communication and connection status, delivering packets over a specific network interface, and updating the PML layer about packet progression.

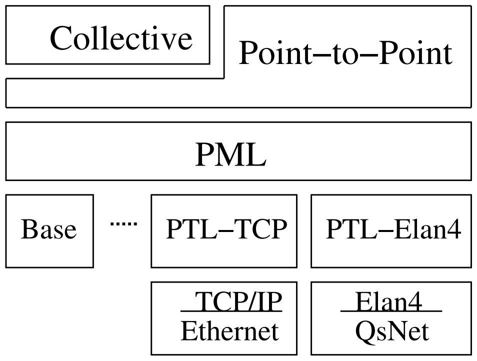


**Fig. 1. Open MPI Communication Architecture**

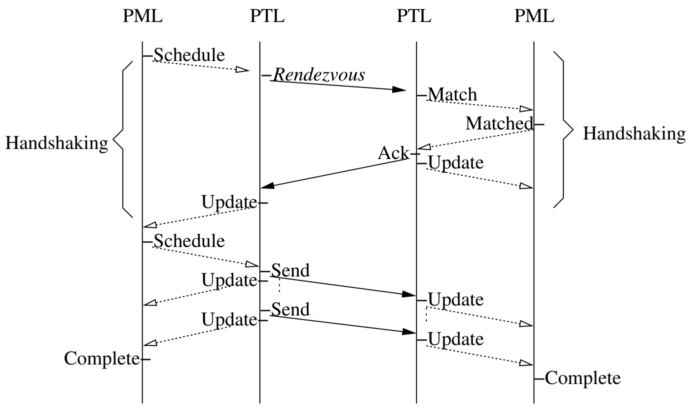


**Fig. 2. Open MPI Point-to-Point Communication Flow Path**

## 2.2. PTL Interface and Communication Flow Path

The PTL layer provides two abstractions: the PTL component and the PTL module [29]. A PTL component encapsulates the functionality of a particular network transport that can be dynamically loaded at run-time; 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 is discussed as follows.

**Joining the Communication Stack** – A PTL component goes through the first two stages, opening and initializing, to join the communication stack. During the opening phase, the PTL component and all its associated dependencies are opened and mapped into the process virtual space. When the dependency and sanity checking completes successfully, this component is entered as one of the available PTL component that could be initialized for communication. Then a process initializes the network device, prepares memory and computing (e.g., additional threads) resources in the form of PTL modules (one per network interface card). These PTL modules are then inserted in the communication stack as available PTL modules. When these procedures successfully completes, 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. 2.1 shows a flow chart of how messages are scheduled across multiple networks. When the PML layer receives a request, it first schedules a *rendezvous* packet to one PTL based on a chosen scheduling heuristic. This *rendezvous* packet is then sent to the receiver side using `ptl_send()` interface. When this packet is detected by one of the PTLs at the receiver side, the receiving PTL request the PML layer to match this packet to the preposted receive requests. If a match is made with a preposted receive request, PML in turn calls `ptl_matched()` interface to return an acknowledgment to the initiating PTL. Any data inlined with the first packet are copied into the application receiver buffer, and the progress of this amount of data is `updated` at the PML layer through ptl_recv_progress(). When the acknowledgment arrives the sender side, the initiating PTL updates the PML layer about the data transmitted inside the first packet through a `ptl_send_progress()` interface. Another scheduling heuristic is then invoked to schedule the rest of the message across available PTLs. The progress of the data transmission are updated accordingly, and this eventually leads to the completion of requests 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. Also in the closing phase, a PTL component makes sure that all the exposed PTL modules are finalized, and that the component-associated memory and threading resources are freed.

## 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, 31, 29]. In order to correctly project the objectives, it is necessary to discuss design requirements for a PTL implementation. Three of Open MPI's main objectives have impacts on the PTL layer, including fault tolerance, concurrent multi-network communication and asynchronous communication progress.

**PTL Requirements for 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/or others to dynamically joining and disjoining the communication stack, checkpoint/restart, and etc. This means that PTL has to handle not only the dynamics status of local network interface, but also the dynamic connections with other PTLs.

**PTL Requirements for Concurrency Communication over Multiple Network:** Open MPI [9] has its messages scheduled across multiple PTLs from the PML layer. However, each network can have dramatically different semantics and memory requirements for the transmission of the same message. In this regard, while the PML layer needs to abstract and encapsulate the difference between different PTLs to integrate them together for the delivery of a single message, each PTL also needs to map the PML function interface onto its existing transmission semantics. The challenges in this respect to the PTL design over Quadrics/Elan4 will be discussed in Section 4.

**PTL Requirements for Dual-Mode Communication Progress:** Open MPI provides two different modes to monitor and progress communication across different network devices: non-blocking polling and thread-based blocking. Non-blocking polling checks the incoming and outgoing traffic of each network device, which can be performed by a MPI process that consists only a single thread. In contrast, additional threads are employed in the thread-based blocking mode to block and wait on the status change 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 [26, 25] 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 card 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 [26]. 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 deferent models: Queue-based model (QDMA) and Remote Directed Message Access (RDMA) model. QDMA allows processes to post messages (up to 2KB) to a remote queue of other processes; 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 [32, 1] for efficient, reliable communication support.

### 3.2. Objectives of PTL Implementation Over Quadrics/Elan4

While Quadrics libraries presents parallel communication over a static pool of processes, Open MPI [9, 31] targets MPI-2 [20] dynamic process management [11] 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 to any existing MPI implementation over Quadrics either because the MPI implementation does not support MPI-2 dynamic process management [5], or because the underlying communication is based on libelan's statically connected processes [26]. In addition, Open MPI targets for concurrent message passing over multiple networks. But the communication/memory semantics can be dramatically different from network to network, e.g., some network are RDMA capable, while others require memory registration before message transmission can take place. Quadrics/Elan4 is RDMA capable and large message ($>$ 2KB) communication has to utilize its RDMA capabilities. A high performance PTL implementation needs to take advantage of these RDMA capabilities. Moreover, the notification of message completion is provided through a different event mechanism, which does not support a poll/select-like [30] mechanism as available in TCP/IP. Thus it is no longer possible for a process to block on any completion of multiple pending messages. This presents another challenge to the support of asynchronous communication progress of multiple pending messages. Taken together, to cope with all the above challenges and 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] communication support over Quadrics [26]. We have proposed strategies to overcome challenges imposed from 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) Asynchronous communication progress.

### 4.1. Communication Initiation and Finalization

As described in Section 3.1, the static pool of processes and static connection between them do not match MPI-2 [20] dynamic process management [11] specifications, in which new processes are allowed to be spawned from and join the existing pool of communicating processes. 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 dynamic process model implies that the default static coupling of Quadrics virtual process ID (VPID) and the rank of a MPI process is no longer possible [26]. This is because VPID is a system related identifier which 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 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 times.

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, we break the complete 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 may not be able to take advantage of the the benefits of hardware broadcast support because it requires the availability of global virtual address space. This does not preclude the possibility for a new global address space to be re-generated from the available address space. This is to be investigated as one of further research topics in Open MPI [9]. Open MPI Run-Time Environment (RTE) can help the newly created 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 running an unpleasant scenario in which a leftover DMA descriptor might regenerate its traffic indefinitely.

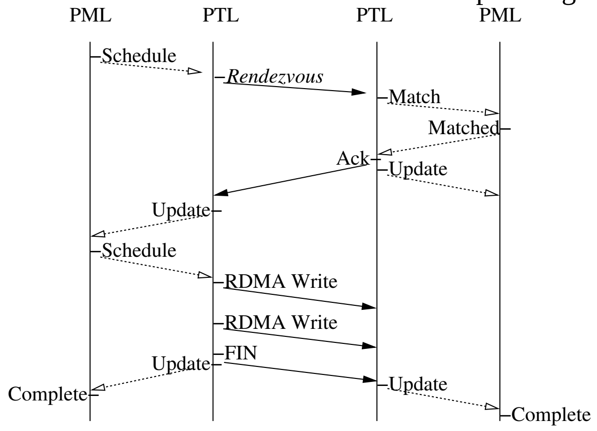


**Fig. 3. Design PTL Interface with RDMA Write Capability**

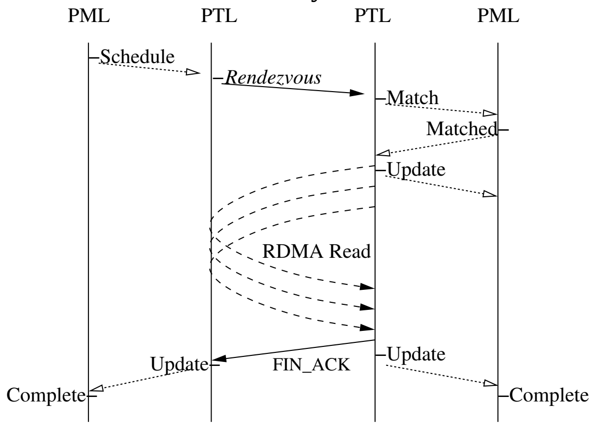


**Fig. 4. Design PTL Interface with RDMA Read Capability**

## 4.2. Integrating Quadrics RDMA Capabilities

In two of the basic Quadrics interprocess communication models, QDMA can only transmit messages up to 2KB. The other model, RDMA read/write, can support transmission of arbitrary messages over Quadrics network. 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. Quadrics RDMA descriptors require the source and destination virtual host memory addresses to be transformed and presented in a different format (E4_Addr) for the network interface card to carry out

RDMA operations. A specially designed Memory Management Unit (MMU) in the Elan4 network interface performs address translation from E4_Addr to physical memory. When needed, the host physical memory can be trapped back by Quadrics system software.

In order to take advantage of these RDMA capabilities, we modify both the memory addressing format and the communication semantics. To support concurrent addressing over Elan4 and other interconnects, a memory descriptor is expanded to include an E4_Addr. This is only a preliminary solution for concurrent message passing over TCP and Quadrics because they both do not require memory pre-registration before communication. Over other interconnects, e.g., InfiniBand [14] and Myrinet [3], the memory range of a message need 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 higher concurrency over different interconnects. Second, we propose two schemes to take advantage of RDMA read and write, respectively. As shown in Fig. 3, in the first scheme, all the send operations after the first *rendezvous* fragment are all replaced with RDMA write operations. At the end of these operations, a control fragment, typed as 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, typed as FIN_ACK, is sent to the sender, both for acknowledging the arrival of the earlier rendezvous fragment and notifying the completion of the whole message. Furthermore, for performance optimization, the transmission of the last control message are be chained to the last RDMA operation using the chained event mechanism. It is automatically triggered when the last RDMA operation is done.

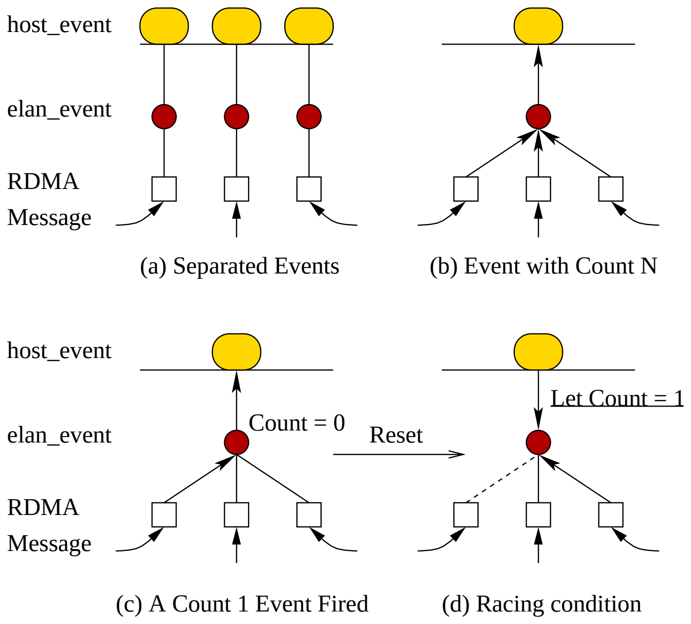


**Fig. 5. Quadrics Chained Event mechanism and Possible Race Condition in Supporting Shared Completion Notification**

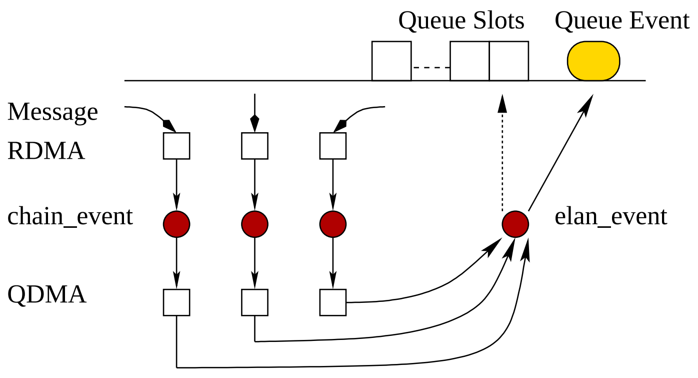


**Fig. 6. Support Shared Completion Notification with QDMA and Chained Event Mechanisms**

### 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, currently available over Solaris and Linux. For the PTL implementation over TCP/IP, because one thread can block and wait on the progress of multiple socket-based file descriptors, it is possible to monitor the progress of all networking traffic with only a single thread per TCP PTL module. However, this is not possible for RDMA descriptors over Quadrics, because the blocking mode of the RDMA descriptor's completion is supported through separated events at different memory locations. This is shown in Fig 5a. A single thread can only block and wait on the host event of a single RDMA descriptor. It is not practically possible 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 at the individual completion of all outstanding 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 progress the communication any further. This is shown in Fig 5c and 5d.

Quadrics QDMA [26] 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 in order to detect multiple outstanding RDMA operations. At the PTL initialization time, a receive queue is pre-created as the shared completion queue, shown on the right side of Fig. 6. When setting up RDMA descriptors, a small message QDMA operation is chained to every RDMA operation. When the RDMA operation completes, the associated chained QDMA will generate a small message to the receive queue. As each of these small messages being posted into one of the queue slots, QDMA will generate an event 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 newly created queue is the same as the pre-created receive queue for the incoming message. Separating the completion notification of local DMA descriptors with the arrival of incoming receive message can make the message handling logic more straightforward, however it requires the additional resources need to maintain the queue. Worse yet, it requires two progressing threads for supporting the asynchronous communication progress. We have provided support to both one queue-based and two-queue-based asynchronous communication progress.

## 5. Implementation

By taking advantages of Quadrics QDMA and RDMA (read and write) [26] operations, and Quadrics chained event mechanism [26], we have implemented our design of the point-to-point transport layer over Quadrics/Elan4, *PTL/Elan4*. Processes are allowed to join the Quadrics Network dynamically and individually by claiming an available context in a system-wide Elan4 capability [26, 4]. Currently, synchronization and connection setup is done collectively during `MPI_Init()` at the run time through the help of other components. To speed up fast transmission of small packets, send buffers (each of 2KB) are preallocated for sending purposes in PTL/Elan4. Accordingly, for receiving short packets, a host-side receive queue is also created with a number of receiver buffers of the same size (often referred to as QSLOTS in Quadrics [26]). Longer packets are delivered either RDMA read or RDMA write capabilities with the help of additional QDMAs for completion notification. Thread-based asynchronous progression is provided by applying additional threads to wait on the pending DMA operations.

## 6. Performance Evaluation

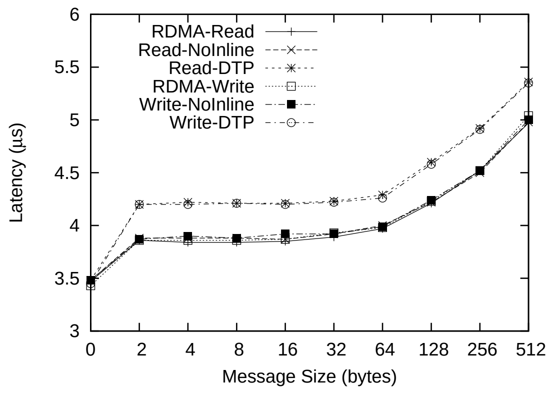
In this section, we describe the performance evaluation of our implementation of the point-to-point transport layer over Quadrics/Elan4. The experiments were conducted on a cluster of eight SuperMicro SUPER X5DL8-GG nodes: each with dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, PCI-X 64-bit 133 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 [26, 2], with a dimension one quaternary fat-tree [7] 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 also provided the overall performance of our design on top of Quadrics/Elan4, comparing to MPICH-QsNet$^{II}$ [26]. 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 are used to warm up the network and nodes in our experiments whenever applicable.

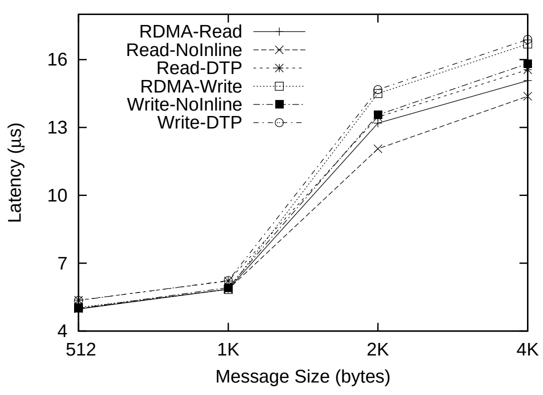
### 6.1. Performance Analysis of Basic RDMA Read and Write

The PML layer schedules the first packet to a PTL module based on the exposed fragment length from that PTL module [9]. In the case of long messages, this packet serves as a *rendezvous* message with an inlined packet of 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 comparing to the networking cost. However, with RDMA capable networks, this strategy would incur an unnecessary memory copying overhead for the first packet while the network interface can directly place the data into destination memory. We have provided an optimization to transmit the *rendezvous* messages without inlined data.

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



(a) Very Small Message



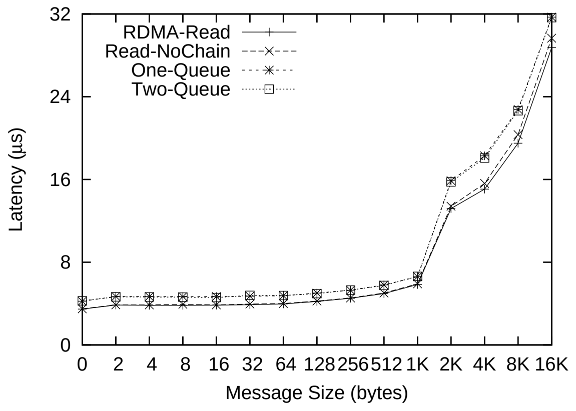
(b) Small Message

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

with each request. For better understanding of the performance strength of PTL/Elan4 module, we have intentionally replaced this copy engine with a generic `memcpy()` call.

In Fig. 7, we have provided 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, this allows us to look at both the small and long message transfer. As shown in Fig. 7(a), the data type component does introduce an overhead about 0.4$\mu$sec compared to the basic support without the datatype component. As also shown in Fig. 7(b), RDMA read is able to delivered 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 both RDMA Read and RDMA write.

## 6.2. Performance Analysis with Chained DMA and Shared Completion Queue



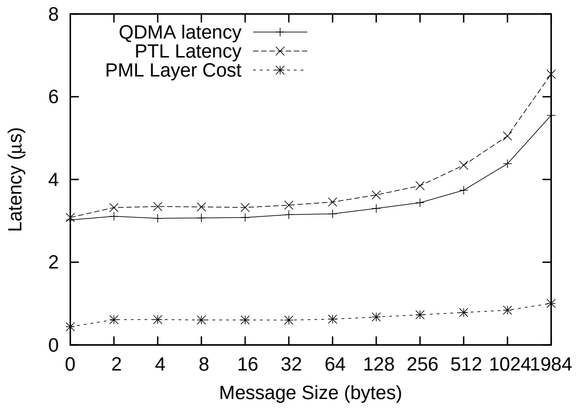


**Fig. 8. Performance Analysis with Chained DMA and Shared Completion Queue**

**Fig. 9. Analysis of Communication Overhead in Different Layers**

In the design of PTL/Elan4, we have utilized Chained DMA mechanism in two scenarios, one to support RDMA read and RDMA write for fast completion notification, the other to support the shared completion notification of RDMA operations with the help of QDMA. Using RDMA read, we have measured the performance with these two strategies As shows in Fig. 8, using the chained DMA for fast completion notification does provide 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 across I/O Bus traffic. PCI-X bus and fast CPU processor (3GHz) used in the experiments also reduces the possible benefits of chained DMA. However, using chained DMA does reduce the host CPU utilization for handling more traffic.

On the other hand, the shared completion queue support does bring performance impacts, with either a combined receive queue, *One-Queue*, or a separate queue (*Two-Queue*) for the completion of local DMA descriptors. This is to be expected because an additional QDMA operation needs to be triggered to the completion queue from a completed DMA operation. Combining shared completion queue with the existing recv queue for incoming remote messages does not provide noticeable performance improvement. This is because using polling-based approach, the cost of checking two eight-byte host-events is about the same as that of checking one. However, the one-queue strategy saves the additional resources needed for another queue and it can also save an additional thread when used to support asynchronous communication progress.

## 6.3. Analysis of Communication Cost in Different Layers

We have performed an analysis of the Open MPI communication stacks. During a ping-pong test, we take the timing from a) when PTL/Elan4 has received a packet from the network and is delivering it to the PML layer for matching, to b) when the next packet is arriving at PTL/Elan4, as the communication time above the PTL layer. An average of that across 5000 iterations is taken as a measurement of the average cost in the PML layer and above, *PML Layer Cost*. Subtracting that from the overall performance is the latency as seen at the PTL/Elan4 layer, *PTL Latency*, which also includes the communication time across the network. This measurement is possible because of the special features of ping-pong micro-benchmarks. A message, like a rotating token, can only be held by one layer at any time. So, on one hand, the PTL layer is not involved in any other work after it handles the packet to PML for matching and before it is called to send a packet, on the other hand, the PML layer is not involved in the communication after it triggers a send operation to PTL and before it receives another packet from PTL. Note although both PTL and PML can detect the completion of local send operations, this cost is not counted into either *PML Layer Cost* or the total time because it is not in the critical path of ping-pong communication.

At the same time, we also measure the performance of the native performance Quadrics QDMA, *QDMA Latency*. Note our implementation is based on Quadrics QDMA model. Also note that the Open MPI communication layer introduces an 64-byte header for matching purpose. A comparison is done with the PTL communication time of a `N`-byte message with the communication time of a `64+N`-byte message using native Quadrics QDMA operations. As shown in Table. 1, the PML layer and above has a communication cost of 0.5$\mu$sec, while PTL/Elan4 delivers the message with a performance comparable to native Quadrics QDMA.

## 6.4. Performance Analysis of Thread-Based Asynchronous Progress

**Table 1. Performance Analysis of Thread-Based Asynchronous Progress (in $\mu$sec)**

| Mesg Length | Basic | Interrupt | One Thread | Two Threads |
|---|---|---|---|---|
| RDMA-Read 4B | 3.87 | 14.70 | 22.76 | 27.50 |
| RDMA-Read 4KB | 15.25 | 27.16 | 32.80 | 47.72 |

The shared completion queue is introduced to support thread-based asynchronous progress. We have performed an analysis of this asynchronous progress support using RDMA read in PTL/Elan4. Fig. 1 shows the performance with four different methods of checking completion. *Basic* progress, *Interrupt*-based progress, *One-Thread*-based asynchronous progress, and *Two-Thread*-based asynchronous progress. Note that the interrupt-based progress is not really a workable strategy under real communication scenarios because the MPI process cannot block within a particular PTL. It is explored here just to find out the cost of interrupt-based communication progress. One-thread and Two-Thread-based progress utilize a combined completion queue or a separated completion queue, respectively. The performance results indicate that one-thread-based asynchronous communication process is more efficient as it reduces the contention on CPU and memory resources. The total threading overhead is around $18us$. About $1us$ due to the chained DMA support as discussed in SectionSubSec:, and $10us$ due to the interrupt. Total around $9us$ is attributed to the threading overhead. Note that, when doing these experiments, we have left both interrupt affinity and processor affinity of the operating system at their default.

## 6.5. Overall Performance of Open MPI over Quadrics/Elan4

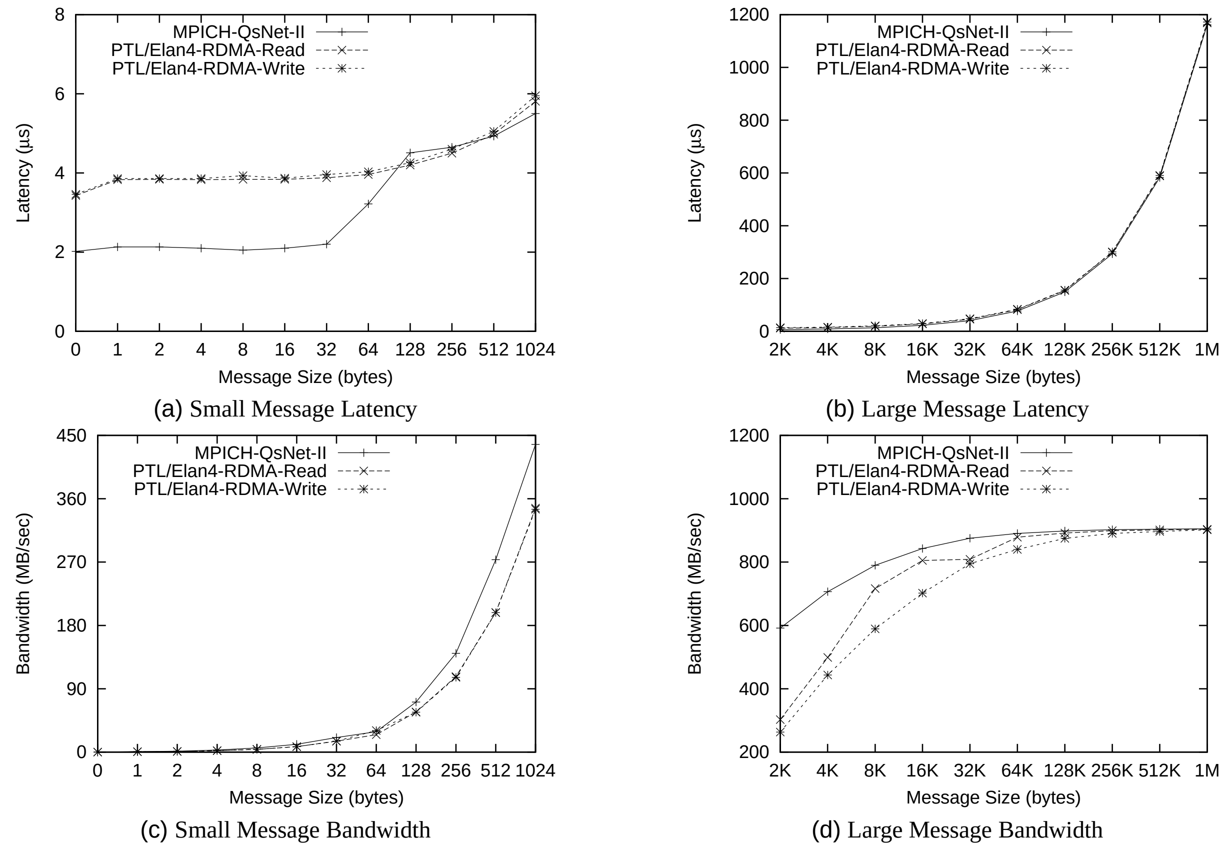


**Fig. 10. Performance Comparisons of Open MPI over Quadrics/Elan4 and MPICH-QsNet$^{II}$**

Fig. 10 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$^{II}$. As shown in Fig. 10, 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, 32-bytes, compared to the 64-bytes in Open MPI, b) MPICH-QsNet$^{II}$ is built on top of Quadrics T-port interface, which does tag matching in the NIC. In terms of bandwidth, our implementation performs particularly worse in the middle range of messages. This is because the Tport support of MPICH-QsNet$^{II}$ does efficient pipelining of messages. Note our implementation starts from different design requirements to co-exist with PTL models of other networks and to be MPI-2 [20] compliant. For example, we are not doing NIC-based tag matching as MPICH-QsNet$^{II}$ does in its underlying Tport [26] interface because currently we intend to have shared request queues for managing traffic from different networks and allow them to be able to crosstalk.

## 7. Related Work

MPI [19] has been the *de facto* messaging passing standard. MPI-2 [20, 13] 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 iterconnects. These include MPICH-GM [22] for Myrinet, MVAPICH [23] and MVAPICH2 [18] for InfiniBand, MPICH-QsNet [26, 2] for Quadrics elan3 and elan4 networks [26], and MPI-Sun [27] for Sun Fire links. Among them, [22, 26, 23] are able to take advantages of RDMA capabilities of their underlying networks. [27] primarily relies on Programmed IO (PIO) for message passing. MPICH-NCSA [24] 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 [28] supports part of MPI-2 interfaces, for example, dynamic process management. MVAPICH2 [18] is an implementation RDMA channel for MPICH2 [21] ADI3 [12] interface. It supports both active and passive one-sided communication [15, 16]. MPICH-QsNet [26] 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 [29, 9, 31] 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.

## 8. Conclusions

In this paper, we have presented the design and implementation of of Open MPI [9, 31] point-to-point tranport layer (PTL) over Quadrics/Elan4 [26, 2]. To match the fault tolerant process management design goals of Open MPI, we have provided a PTL initiation to allow processes to join and disjoin from the Quadircs network communication at any time. Our design has also integrated Quadrics RDMA capabilities into the communication model of Open MPI [9, 31]. Thread-based Asynchronous communication progress are supported with a strategy utilizing Quadrics chained event mechanism and QDMA.

Our evaluation has shown that the implemented point-to-point transport layer achieves comparable performance to Quadrics native QDMA interface, from which it is derived. In addition, this design of point-to-point transport over Quadrics achieves a performance slight lower but comparable to that of MPICH-QsNet$^{II}$ [26], though it does not take advantage of NIC-based tag matching as MPICH-QsNet$^{II}$. Furthermore, this design and implementation provides a high performance, MPI-2 [20] compliant message passing over Quadrics/Elan4.

In future, we intend to study the effectiveness of performance improvement with Open MPI's aggregated communication over network interfaces, including both multi-rail communication over Quadrics [6] and concurrent communication over multiple interconnects. We also intend to study fault tolerant process management, reliability of message delivery over multiple interconnects.

## Acknowledgment

We would like to thank the entire Open MPI team for bringing the Open MPI project avaible as a research platform. In particular, we would like to thank Dr. Jefferey Squyres from Indiana Univeristy and Dr. George Bosilca from The University of Tennessee at Knoxville for their valuable dicussion.

**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] R. T. Aulwes, D. J. Daniel, N. N. Desai, R. L. Graham, L. Risinger, M. W. Sukalski, and M. A. Taylor. Network Fault Tolerance in LA-MPI. In *Proceedings of EuroPVM/MPI '03*, September 2003.

[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] 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.

[4] R. Brightwell, D. Dourfler, and K. D. Underwood. A Comparison of 4X InfiniBand and Quadrics Elan-4 Technology. In *Proceedings of Cluster Computing, '04*, San Diego, California, September 2004.

[5] R. Brightwell and L. Shuler. Design and Implementation of MPI on PUMA Portals. pages 18–25, 1996.

[6] S. Coll, E. Frachtenberg, F. Petrini, A. Hoisie, and L. Gurvits. Using Multirail Networks in High-Performance Clusters. In *IEEE Cluster 2001*, Newport Beach, CA, October 2001.

[7] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach*. The IEEE Computer Society Press, 1997.

[8] 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.

[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] W. Gropp, E. Lusk, D. Ashton, R. Ross, R. Thakur, and B. Toonen. Mpich abstract device interface version 3.4 reference manual draft. Technical report, Argonne National Laboratory, Mathematics and Computer Science Division, May 2003. Draft.

[13] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

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

[15] W. Jiang, J. Liu, H. Jin, D. K. Panda, W. Gropp, and R. Thakur. High Performance MPI-2 One-Sided Communication over InfiniBand. In *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 04)*, Chicago, IL, April 2004.

[16] W. Jiang, J. Liu, H.-W. Jin, D. K. Panda, D. Buntinas, R. Thakur, and W. Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. In *Proceedings of EuroPVM/MPI '04*, Budapest, Hungary, September 2004.

[17] 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.

[18] 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.

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

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

[21] Mpich2, argonne. http://www-unix.mcs.anl.gov/mpi/mpich2/.

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

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

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] J. M. Squyres and A. Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, St. Malo, France, July 2004.

[30] W. R. Stevens. *UNIX Network Programming, Networking APIs: Sockets and XTI*. Prentice–Hall, Inc. , Upper Saddle River, New Jersey 07458, second edition, 1998.

[31] 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.

[32] 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.

[33] 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.