Design and Implementation of MPICH2 over InfiniBand with RDMA Support*

Jiuxing Liu† Weihang Jiang† Pete Wyckoff‡ Dhabaleswar K Panda†
David Ashton Darius Buntinas William Gropp Brian Toonen

†Computer and Information Science, The Ohio State University, Columbus, OH 43210
{liuj, jiangw, panda}@cis.ohio-state.edu

‡Ohio Supercomputer Center, 1224 Kinnear Road, Columbus, OH 43212
pw@osc.edu

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
{ashton, buntinas, gropp, toonen}@mcs.anl.gov

Abstract

Due to its high performance and open standard, InfiniBand is gaining popularity in the area of high performance computing. In this area, MPI has been the de facto *standard for writing parallel applications. One of the most popular MPI implementations is MPICH. Its successor, MPICH2, features a completely new design which aims to provide more performance and flexibility. To ensure portability, it provides a hierarchical structure based on which porting can be done at different levels.*

In this paper, we present our experiences designing and implementing MPICH2 over InfiniBand. Our study focuses on optimizing the performance of MPI-1 functions in MPICH2. One of our objectives is to exploit Remote Direct Memory Access (RDMA) to achieve high performance. We have based our design on the RDMA Channel interface provided by MPICH2, which encapsulates architecture dependent communication functionalities into a very small set of functions.

Starting with a basic design, we apply different optimizations and also propose a zero-copy based design. We characterize the impact of our optimizations and designs using micro-benchmarks. We have also performed application level evaluation using NAS Parallel Benchmarks. Our optimized MPICH2 implementation achieves 7.6μs latency and 857MB/s bandwidth, which are close to the raw performance of the underlying InfiniBand layer. Our study shows that the RDMA Channel interface in MPICH2 provides a simple yet powerful abstraction, which enables implementations with high performance by exploiting RDMA operations in InfiniBand. To the best of our knowledge, this is the first high performance design and implementation of MPICH2 on InfiniBand using RDMA support.

*This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, a grant from Sandia National Laboratory, a grant from Intel Corporation, and National Science Foundation's grants #CCR-0204429 and #CCR-0311542.

1 Introduction

During the last ten years, the research and industry communities have been proposing and implementing user-level communication systems to address some of the problems associated with the traditional networking protocols. The Virtual Interface Architecture (VIA) [6] was proposed to standardize these efforts. More recently, InfiniBand Architecture [9] has been introduced which combines storage I/O with Inter-Process Communication (IPC).

In addition to send and receive operations, InfiniBand architecture supports Remote Direct Memory Access (RDMA). RDMA operations enable direct access to the address space of a remote process. These operations introduce new opportunities and challenges in designing communication protocols.

In the area of high performance computing, MPI [25] has been the *de facto* standard for writing parallel applications. After the original MPI standard (MPI-1), an enhanced standard (MPI-2) [18] has been introduced which includes features such as dynamic process management, one-sided communication and I/O. MPICH [7] is one of the most popular MPI-1 implementations. Recently, work is under way

to create MPICH2 [1], which aims to support both MPI-1 and MPI-2 standards. It features a completely new design, which provides better performance and flexibility than its predecessor. MPICH2 is also very portable and provides mechanisms which make it easy to re-target it to new communication architectures such as InfiniBand.

In this paper, we present our experiences in designing and implementing MPICH2 over InfiniBand using RDMA operations. Although MPICH2 supports both MPI-1 and MPI-2, our study focuses optimizing the performance of MPI-1 functions. We have based our design on the RDMA Channel interface provided by MPICH2, which encapsulates architecture dependent communication functionalities into a very small set of functions. Despite its simple interface, we have shown that the RDMA Channel does not prevent one from achieving high performance. In our testbed, our MPICH2 implementation achieves 7.6μs latency and 857MB/s peak bandwidth, which are quite close to the raw performance of our InfiniBand platform. We have also evaluated our designs using NAS Parallel Benchmarks [21]. Overall, we have demonstrated that the RDMA Channel interface is a simple yet powerful abstraction which makes it possible to design high performance MPICH2 implementations with less development effort.

In our design, we use RDMA operations exclusively for communication. Our design starts with an emulation of a shared memory based implementation. Then we introduce various optimization techniques to improve its performance. To show the impact of each optimization, we use latency and bandwidth micro-benchmarks to evaluate our design after applying it. We also propose a zero-copy design for large messages. Our results show that with *piggybacking* and *zero-copy* optimizations for large messages, our design achieves very good performance.

The remaining part of the paper is organized as follows: In Section 2, we provide an introduction to InfiniBand and its RDMA operations. Section 3 presents an overview of MPICH2, its implementation structure, and the RDMA Channel interface. We describe our designs and implementations in Sections 4 and 5. In Section 6, we compare our RDMA Channel based design with another design based on a more complicated interface called CH3. Application level performance evaluation is presented in Section 7. In Section 8, we describe related work. In Section 9, we draw conclusions and briefly mention some of the future research directions.

2 InfiniBand Overview

The InfiniBand Architecture (IBA) [9] defines a switched network fabric for interconnecting processing nodes and I/O nodes. It provides a communication and management infrastructure for inter-processor communication and I/O. In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by Channel Adapters (CA). Channel Adapters usually have programmable DMA engines with protection features. There are two kinds of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCAs sit on processing nodes.

The InfiniBand communication stack consists of different layers. The interface presented by Channel adapters to consumers belongs to the transport layer. A queue-based model is used in this interface. A Queue Pair in InfiniBand Architecture consists of two queues: a send queue and a receive queue. The send queue holds instructions to transmit data and the receive queue holds instructions that describe where received data is to be placed. Communication operations are described in Work Queue Requests (WQR), or descriptors, and submitted to the work queue. The completion of WQRs is reported through Completion Queues (CQs). Once a work queue element is finished, a completion queue entry is placed in the associated completion queue. Applications can check the completion queue to see if any work queue request has been finished. InfiniBand also supports different classes of transport services. In this paper, we focus on the Reliable Connection (RC) service.

2.1 RDMA Operations in InfiniBand Architecture

InfiniBand Architecture supports both channel and memory semantics. In channel semantics, send/receive operations are used for communication. To receive a message, the programmer posts a receive descriptor which describes where the message should be put at the receiver side. At the sender side, the programmer initiates the send operation by posting a send descriptor.

In memory semantics, InfiniBand supports Remote Direct Memory Access (RDMA) operations, including RDMA write and RDMA read. RDMA operations are one-sided and do not incur software overhead at the remote side. In these operations, the sender (initiator) starts RDMA by posting RDMA descriptors. A descriptor contains both the local data source addresses (multiple data segments can be specified at the source) and the remote data destination address. At the sender side, the completion of an RDMA operation can be reported through CQs. The operation is transparent to the software layer at the receiver side.

Since RDMA operations enable a process to access the address space of another process directly, they have raised some security concerns. In InfiniBand architecture, a key based mechanism is used. A memory buffer must first be registered before they can be used for communication. Among other things, the registration generates a remote key. This remote key must be presented when the sender initiates an RDMA operation to access the buffer.

Compared with send/receive operations, RDMA operations have several advantages. First, RDMA operations themselves are generally faster than send/receive operations because they are simpler at the hardware level. Second, they do not involve managing and posting descriptors at the receiver side, which would incur additional overheads and reduce the communication performance.

3 MPICH2 Overview

MPICH [7] is developed at Argonne National Laboratory. It is one of the most popular MPI implementations. The original MPICH provides support for the MPI-1 standard. As a successor of MPICH, MPICH2 [1] aims to support not only the MPI-1 standard, but also functionalities such as dynamic process management, one-sided communication and MPI I/O, which are specified in the MPI-2 standard. However, MPICH2 is not merely MPICH with MPI-2 extensions. It is based on a completely new design, aiming to provide more performance, flexibility and portability than the original MPICH. One of the notable features in the implementation of MPICH2 is that it can take advantage of RDMA operations if they are provided by the underlying interconnect. These operations can be used not only to support MPI-2 one-sided communication, but also to implement normal MPI-1 communication. Although MPICH2 is still under development, beta versions are already available for developers. In the current version, all MPI-1 functions have been implemented. MPI-2 functions are not completely supported yet. In this paper, we mainly focus on the MPI-1 part of MPICH2.

3.1 MPICH2 Implementation Structure

One of the objectives in MPICH2 design is portability. To facilitate porting MPICH2 from one platform to another, MPICH2 uses ADI3 (the third generation of the Abstract Device Interface) to provide a portability layer. ADI3 is a full-featured abstract device interface and has many functions, so it is not a trivial task to implement all of them. To reduce the porting effort, MPICH2 introduces the CH3 interface. CH3 is a layer that implements the ADI3 functions, and provides an interface consisting of only a dozen functions. A "channel" implements the CH3 interface. Channels exist for different communication architectures such as TCP sockets, SHMEM, etc. Because there are only a dozen functions associated with each channel interface, it is easier to implement a channel than the ADI3 device.

To take advantage of architectures with globally shared memory or RDMA capabilities and further reduce the porting overhead, MPICH2 introduces the RDMA Channel which implements the CH3 interface. The RDMA Channel interface only contains five functions. We will discuss the details of RDMA Channel Interface in the next subsection.

The hierarchical structure of MPICH2 , as shown in Figure 1, gives much flexibility to implementors. The three interfaces (ADI3, CH3, and RDMA Channel Interface) provide different trade-offs between communication performance and ease of porting.

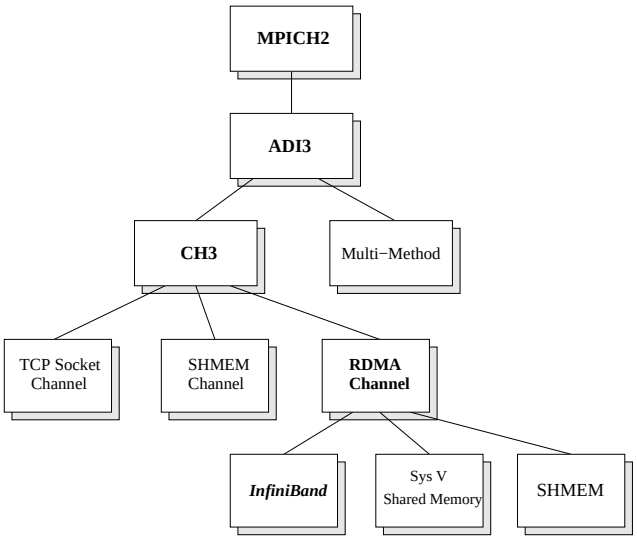

Figure 1. MPICH2 Implementation Structure

3.2 MPICH2 RDMA Channel Interface

MPICH2 RDMA Channel interface is designed specifically for architectures with globally shared memory or RDMA capabilities. It contains five functions, among which only two are central to communication. (Other functions deal with process management, initialization and finalization, respectively.) These two functions are called *put (write)* and *get (read)*.

Both *put* and *get* functions accept a connection structure and a list of buffers as parameters. They return to the caller the number of bytes that have been successfully put or gotten. If the bytes completed is less than the total length of buffers, the caller will retry the same get or put operation later.

Figure 2 illustrates the semantics of *put* and *get*. Logically, a pipe is shared between the sender and the receiver. The *put* operation writes to the pipe and the *get* operation reads from it. The data in the pipe is consumed in FIFO order. Both operations are non-blocking in the sense that they return immediately with the number of bytes completed instead of waiting for the entire operation to finish. We should note that *put* and *get* are different from RDMA write and RDMA read in InfiniBand. While RDMA operations in In-

finiBand are one-sided, *put* and *get* in the RDMA Channel interface are essentially two-sided operations.

Put and *get* operations can be implemented on architectures with globally shared memory in a straightforward manner. Figure 3 shows one example. In this implementation, a shared buffer (organized logically as a ring) is placed in shared memory, together with a head pointer and a tail pointer. The *put* operation copies the user buffer to the shared buffer and adjusts the head pointer. The *get* operation involves reading from the shared buffer and adjusting the tail pointer. In the case of buffer overflow or underflow (detected by comparing head and tail pointers), the operations return immediately and the caller will retry them.

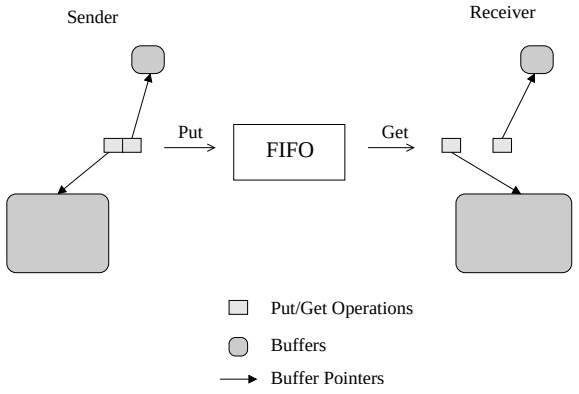

Figure 2. Put and Get Operations

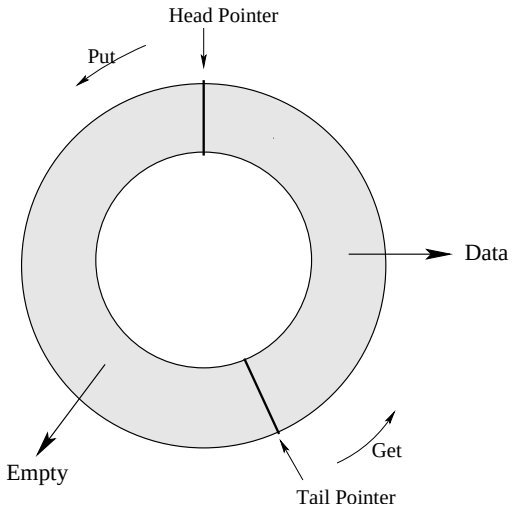

Figure 3. Put and Get Implementation with Globally Shared Memory

Working at the RDMA Channel interface level is better than writing a new CH3 or ADI3 implementation for many reasons:

1. Improvements done at this level can affect all shared-memory like transports such as globally shared-memory, RDMA over IP, Quadrics and Myrinet.
2. Other protocols on InfiniBand need efficient processing, including one-sided communication in MPI-2, DSM systems and parallel file systems. The RDMA Channel interface can potentially be used also for them.
3. Designing proper interfaces to similar systems improves performance and portability in general.

In collaboration, the OSU and ANL teams are also currently working together to design an improved interface which can benefit communication systems in general.

4 Designing and Optimizing MPICH2 over InfiniBand

In this section, we present several different designs of MPICH2 over InfiniBand based on the RDMA Channel interface. We first start with a basic design which resembles the scheme described in Figure 3. Then we apply various optimization techniques to improve its performance. In this section, the designs are evaluated using micro-benchmarks such as latency and bandwidth. We show that by taking advantage of RDMA operations in InfiniBand, we can not only achieve low latency for small messages, but also high bandwidth for large messages using the RDMA Channel interface. In Section 5, we present a zero-copy design. In Section 6, we also present a design based on the CH3 interface and compare it with our designs based on the RDMA Channel interface. In Section 7, we will present performance results using application level benchmarks.

4.1 Experimental Testbed

Our experimental testbed consists of a cluster system with 8 SuperMicro SUPER P4DL6 nodes. Each node has dual Intel Xeon 2.40 GHz processors with a 512K L2 cache and a 400 MHz front side bus. The machines are connected by Mellanox InfiniHost MT23108 DualPort 4X HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The HCA adapters work under the PCI-X 64-bit 133MHz interfaces. We used the Linux Red Hat 7.2 operating system with 2.4.7 kernel. The compilers we used were GNU GCC 2.96 and GNU FORTRAN 0.5.26.

4.2 Basic Design

In Figure 3, we have explained how the RDMA Channel interface can be implemented on shared memory architectures. However, in a cluster connected by InfiniBand, there

is no physically shared memory. In our basic design, we use RDMA write operations provided by InfiniBand to emulate this scheme.

We put the shared memory buffer in the receiver's main memory. This memory is registered and exported to the sender. Therefore, it is accessible to the sender through RDMA operations. To avoid the relatively high cost of registering user buffers for sending every message, we also use a pre-registered buffer at the sender which is the same size as the shared buffer at the receiver. User data is first copied into this buffer and then sent out. Head and tail pointers also need to be shared between the sender and the receiver. Since they are used frequently at both sides, we use replication to prevent polling through the network. For the tail pointer, a master copy is kept at the receiver and a replica is kept at the sender. For the head pointer, a master copy is kept at the sender and a replica is kept at the receiver. For each direction of every connection, the associated "shared" memory buffer, head and tail pointers are registered during initialization and their addresses and remote keys are exchanged.

At the sender, the *put* operation is implemented as follows:

1. Use local copies of head and tail pointers to decide how much empty space is available.
2. Copy user buffer to the pre-registered buffer.
3. Use RDMA write operation to write the data to the buffer at the receiver side.
4. Adjust the head pointer based on the amount of data written.
5. Use another RDMA write to update the remote copy of head pointer.
6. Return the number of bytes written.

At the receiver, the *get* operation is implemented in the following way:

1. Check local copies of head and tail pointers to see if there is new data available.
2. Copy the data from the shared memory buffer to user buffer.
3. Adjust the tail pointer based on the amount of data that has been copied.
4. Use an RDMA write to update the remote copy of tail pointer.
5. Return the number of bytes successfully read.

We should note that copies of head and tail pointers are not always consistent. For example, after a sender adjusts its head pointer, it uses RDMA write to update the remote copy at the receiver. Therefore, the head pointer at the receiver is not up-to-date until the RDMA write finishes. However, this inconsistency does not affect the correctness of the scheme because it merely prevents the receiver from reading new data temporarily. Similar, inconsistency of tail pointer may prevent the sender from writing to the shared buffer. But eventually the pointers will become up-to-date and the sender or the receiver will be able to make progress.

4.2.1 Performance of the Basic Design

We use latency and bandwidth tests to evaluate the performance of our basic design. The latency test is conducted in a ping-pong fashion and the results are derived from round-trip time. In the bandwidth test, a sender keeps sending back-to-back messages to the receiver until it has reached a pre-defined window size W. Then it waits for these messages to finish and send out another W messages. The results are derived from the total test time and the number of bytes sent.

Figures 4 and 5 show the results. Our basic design achieves a latency of 18.6μs for small messages and a bandwidth of 230MB/s for large messages. (Note that unless stated otherwise, the unit MB in this paper is an abbreviation for 10^6 bytes.) However, these numbers are much worse than the raw performance numbers achievable by the underlying InfiniBand layer (5.9μs latency and 870MB/s bandwidth).

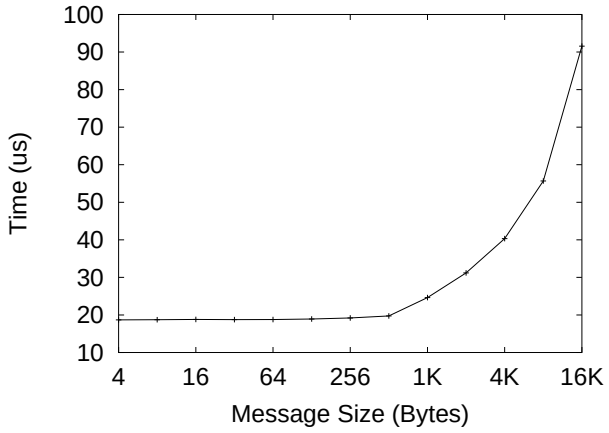

Figure 4. MPI Latency for Basic Design

A careful look at the basic design reveals many inefficiencies. For example, a matching pair of send and receive operations in MPI require three RDMA write operations to take place: one for transfer of data, two for updating head

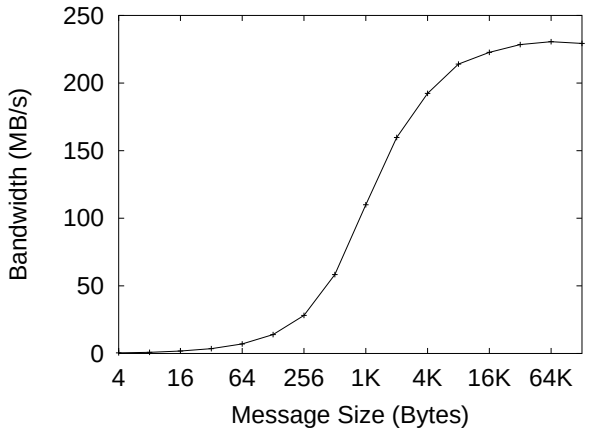

Figure 5. MPI Bandwidth for Basic Design

and tail pointers. This not only increases latency and host overhead, but also generates unnecessary network traffic.

For large messages, the basic scheme leads to two extra memory copies. The first one is from user buffer to the pre-registered buffer at the sender side. The second one is from the shared buffer to user buffer at the receiver side. These memory copies consume resources such as memory bandwidth and CPU cycles. To make matters worse, in the basic design the memory copies and communication operations are serialized. For example, a sender first copies the whole message (or part of the message if it cannot fit itself in the empty space of the pre-registered buffer). Then it initiates RDMA write to transfer the data. This serialization of copying and RDMA write greatly reduces the bandwidth for large messages.

4.3 Optimization with Piggybacking Pointer Updates

Our first optimization targeted to avoid separate head and tail pointer updates whenever possible. The technique we used is piggybacking, which combines pointer updates with data transfer.

At the sender side, we combine data and the new value of head pointer into a single message. To help the receiver detect the arrival of the message, we attach the size with the message and put two flags at the beginning and the end of the message. The receiver detects arrival of the new message by polling on the flags. To avoid possible situations where the buffer content happens to have the same value as the flag, we divide the shared buffer into fixed-size chunks. Each message uses a different chunk. In this way, the situations can be handled by using two polling flags or "bottom fill". Similar techniques have been used in [17, 23].

At the receiver side, instead of using RDMA write to update the remote tail pointer each time data has been read, we delay the updates until the free space in the shared buffer drops below a certain threshold. If there are messages sent from the receiver to the sender, the pointer value is attached with the message and no extra message is used to transfer pointer updates. If there are no messages sent from the receiver to the sender, eventually we will explicitly send the updates by using an extra message. The sender updates its pointer after receiving this message. However, even in this case the traffic can be reduced because several consecutive updates of the tail pointer can be sent using only one message.

The use of piggybacking and delayed pointer updates can greatly improve the performance of small message. From Figure 6 we see that the latency is reduced from 18.6μs to 7.4μs. Figure 7 shows that the optimization also improves bandwidth for small messages.

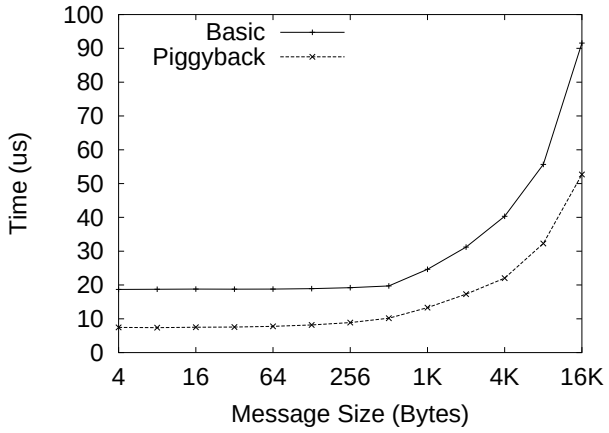

Figure 6. MPI Small Message Latency with Piggybacking

4.4 Optimization with Pipelining of Large Messages

As we have discussed, our basic design suffers from serialization of memory copies and RDMA writes. A better solution is to use pipelining to overlap memory copies with RDMA write operations.

In our piggybacking optimization, we divide the shared memory buffer into small chunks. When sending and receiving large messages, we need to use more than one such chunks. At the sender side, instead of starting RDMA writes after copying all the chunks, we initiate the RDMA transfer immediately after copying each chunk. In this way, the RDMA operation can be overlapped with the copying of the next chunk. Similarly, at the receiver side we start copying from the shared buffer to the user buffer immediately after

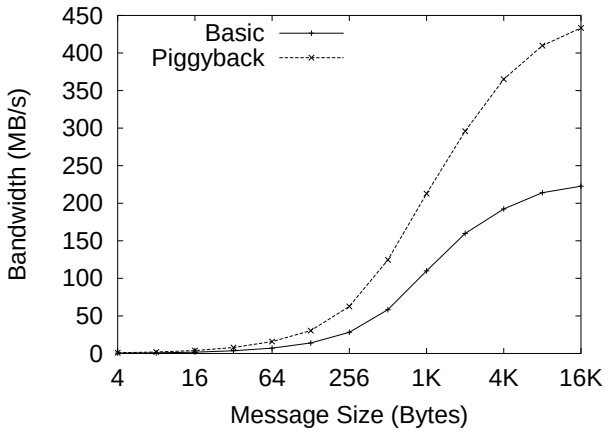

Figure 7. MPI Small Message Bandwidth with Piggybacking

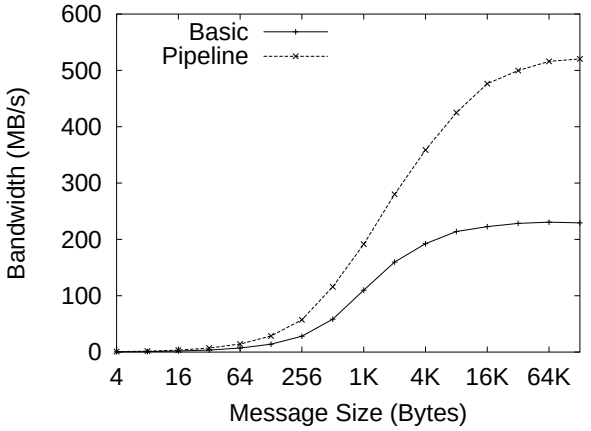

Figure 8. MPI Bandwidth with Pipelining

a chunk is received. In this way, the receive RDMA operations can be overlapped with the copying.

Figure 8 compares the bandwidth of the pipelining scheme with the basic scheme. (Piggybacking is also used in the pipelining scheme.) We can see that pipelining, combined with piggybacking, has greatly improved MPI bandwidth. The peak bandwidth has been increased from 230MB/s to over 500MB/s. However, this result is still not satisfying because InfiniBand is able to deliver bandwidth up to 870MB/s.

To investigate the performance bottleneck, we have conducted memory copy tests in our testbed. We have found out that memory copy bandwidth is less than 800MB/s for large messages. In our MPI bandwidth tests, with RDMA write operations and memory copies both using the memory bus, the bandwidth achievable at the application level is even less. Therefore, it is clear that memory bus becomes a performance bottleneck for large messages due to extra memory copies.

4.4.1 Impact of Chunk Size on Pipelining Performance

In the pipelining optimization, it is very important that we balance each stage of the pipeline so that we can get maximum throughput. One parameter we can change to balance pipeline stages is the chunk size, or how much data we copy each time for a large message. Figure 9 shows MPI bandwidth for different chunk sizes for the pipelining optimization. We can observe that MPI does not give good performance when the chunk size is either too small (1K Bytes) or too large (32K Bytes). MPI performs comparably for chunk sizes of 2K to 16K Bytes. In all remaining tests, we have chosen a chunk size of 16K Bytes.

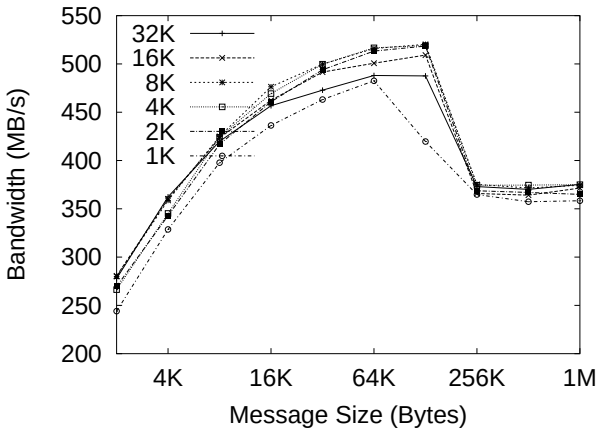

Figure 9. MPI Bandwidth with Pipelining (Different Chunk Sizes)

5 Zero Copy Design

As we have discussed in the previous subsection, it is desirable to avoid memory copies for large messages. In this section, we describe a zero-copy design for large messages based on the RDMA Channel interface.

In our new design, small messages are still transferred using RDMA write, similar to the piggybacking scheme. However, for large messages, RDMA Read, instead of RDMA write, is used for data transfer. The basic idea of our zero-copy design is to let the receiver "pull" the data directly from the sender using RDMA read.

For each connection, shared buffers are still used for transferring small messages. However, the data for large messages is not transferred through the shared buffer. At the sender, when the *put* function is called, we check the user buffer and decide whether to use zero-copy or not based on the buffer size. If zero-copy is not used, the message is sent through the shared buffer as discussed before. If zero-copy is used, the following steps happen:

1. Register the user buffer.
2. Construct a special packet which contains information about the user buffer such as address, size, and remote key.
3. The special packet is sent using RDMA write through the shared buffer.

After these steps, the *put* function returns a value of 0, indicating that the operation is not finished yet. After the packet arrives at the other side, the receiver finds out that it is a special packet by checking its header. When the *get* function is called, the receiver will check the shared buffer and process all the packets in order. If it is a data packet, the data is copied to user buffer. If it is a special packet, the user buffer is registered and an RDMA read operation is issued to fetch the data from the remote side directly to the user buffer. After initiating the RDMA read, the *get* function returns with a value of 0, indicating that the operation is still in progress. When the RDMA read is finished, calling the get function will lead to an acknowledgment packet being sent to the sender. The get function also returns with the number of bytes successfully transferred. Then, at the sender side, the acknowledgment packet is received. Since the sender now knows that the transfer has finished, the user buffer is de-registered and the next call of put function will return with the number of bytes transferred. The zero-copy process is illustrated in Figure 10.

In the current InfiniBand implementation, memory registration and de-registration are expensive operations. To reduce the number of registration and de-registration, we have implemented a registration cache [8]. The basic idea is to delay the de-registration of user buffers and put them into a cache. If the same buffer is reused some time later, its registration information can be fetched directly from the cache instead of going through the expensive registration process. De-registration happens only when there are too many registered user buffers.

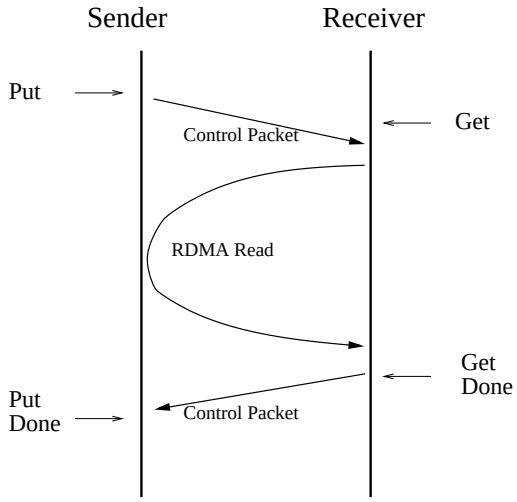

Figure 10. Zero-Copy Design

It should be noted that the effectiveness of registration cache depends on buffer reuse patterns of applications. If applications rarely reuse buffers for communication, registration overhead cannot be avoided most of the time. Fortunately, our previous study with the NAS Parallel Benchmarks [16] has demonstrated that buffer reuse rates are very high in these applications.

We compare the bandwidth of the pipelining design and the zero-copy design in Figure 11. We can observe that zero-copy greatly improves the bandwidth for large messages. We can achieve a peak bandwidth of 857MB/s, which is quite close to the peak bandwidth at the InfiniBand level (870MB/s). We can also see that due to cache effect, bandwidth for large messages drops for pipelining design. Due to the extra overhead in the implementation, the zero-copy design slightly increases the latency for small messages, which is now around 7.6μs.

Our zero-copy implementation uses RDMA read operations, which let the receiver to pull data from the sender. An alternative is to use RDMA write operations and let the sender to "push" data to the receiver. Before the sender can push the data, the receiver has to use special packets to advertise availability of new receive buffers. Therefore, this method can be very efficient if the *get* operations are called *before* the corresponding *put* operations. However, in the current MPICH2 implementation, the layers above the RDMA Channel interface are implemented in such a way that *get* is always called after *put* for large messages. Therefore, we have chosen an RDMA read based implementation

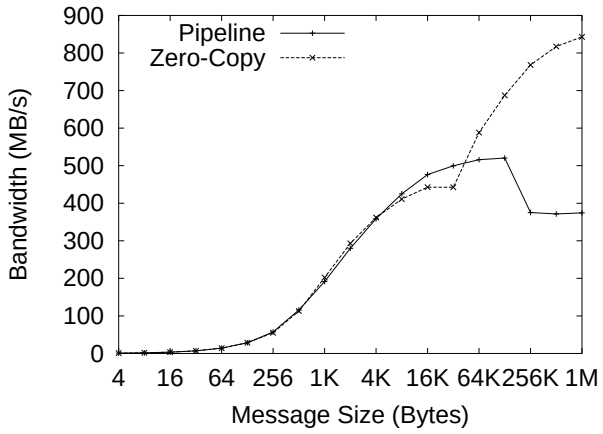

Figure 11. MPI Bandwidth with Zero-Copy and Pipelining

instead of RDMA write.

6 Comparing CH3 and RDMA Channel Interface Designs

The RDMA Channel interface in MPICH2 provides a very simple way to implement MPICH2 in many communication architectures. In the previous section, we have shown that this interface does not prevent one from achieving good performance. Nor does it prevent zero-copy implementation for large messages. Our results have shown that with various optimizations, we can achieve a latency of 7.6μs and a peak bandwidth of 857MB/s.

The CH3 interface is more complicated than the RDMA Channel Interface. Therefore, porting it requires more effort. However, since CH3 provides more flexibility, it is possible to achieve better performance at this level.

To study the impact of different interfaces on MPICH2 performance, we have also done a CH3 level implementation. This implementation uses RDMA write operations for transferring large messages, as shown in Figure 12. Before transferring the message, a handshake happens between the sender and the receiver. User buffer at the receiver is registered and its information is sent to the sender through the handshake. The sender then uses RDMA write to transfer the data. A registration cache is also used in this implementation.

Figures 13 and 14 compare this implementation with our RDMA Channel based zero-copy design using latency and bandwidth micro-benchmarks. We can see that the two implementations perform comparably for small and large messages. However, the CH3 based design outperforms RDMA Channel based design for mid-size messages (32K to 256K bytes) in bandwidth.

Figure 15 shows the bandwidth of RDMA read and RDMA write at the InfiniBand VAPI level. (VAPI is the programming interface for our InfiniBand cards.) With the current VAPI implementation, we can see that RDMA write operations have a clear advantage over RDMA read for mid-size messages. Therefore, the fact that CH3 based design outperforms RDMA Channel based design for mid-size messages is more due to the raw performance difference between RDMA write and RDMA read than the designs themselves.

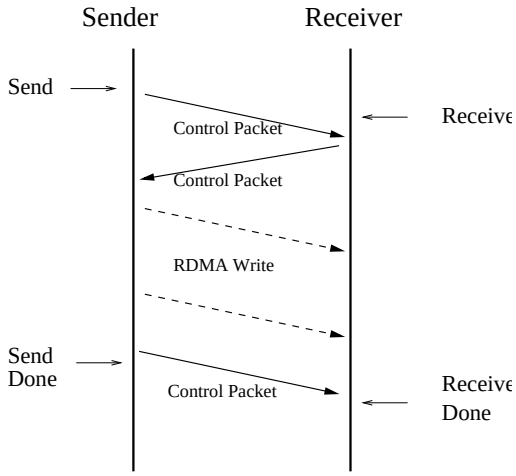

Figure 12. CH3 Zero-Copy with RDMA Write

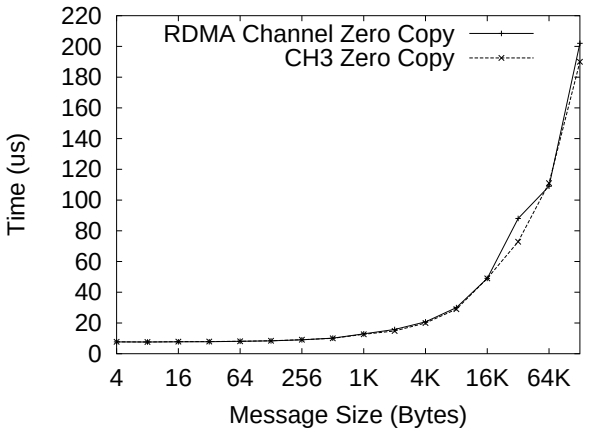

Figure 13. MPI Latency for CH3 Design and RDMA Channel Interface Design

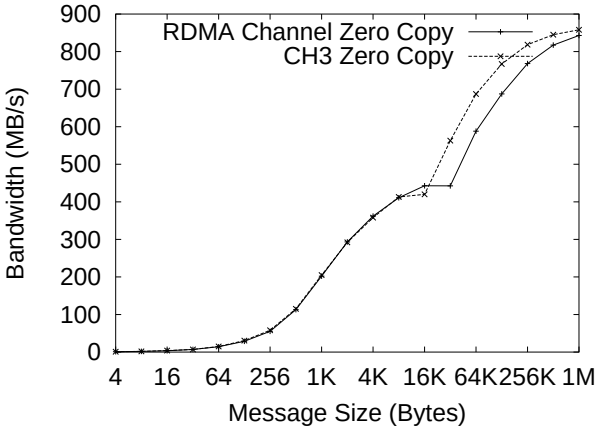

Figure 14. MPI Bandwidth for CH3 Design and RDMA Channel Interface Design

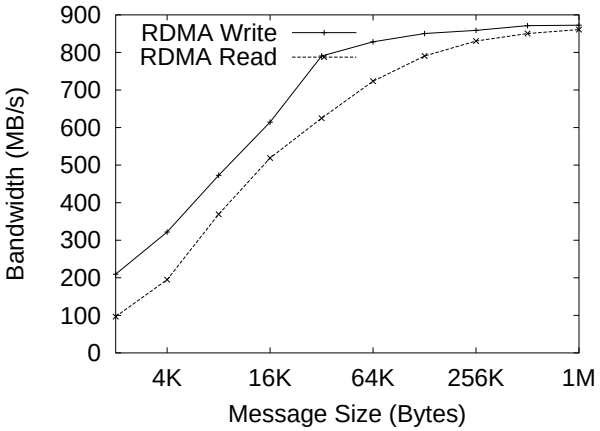

Figure 15. InfiniBand Bandwidth

7 Application Level Evaluation

In this section, we carry out application level evaluation of our MPICH2 designs using NAS Parallel Benchmarks [21]. We run class A benchmarks on 4 nodes and class B benchmarks on 8 nodes. Benchmarks SP and BT require square number of nodes. Therefore, their results are only shown for 4 nodes.

The results are shown in Figures 16 and 17. We have evaluated three designs: RDMA Channel implementation with pipelining for large messages (Pipelining), RDMA Channel implementation with zero-copy for large messages (RDMA Channel) and CH3 implementation with zero-copy (CH3). Although the performance difference of these three designs is not much, we have observed that the pipelining design performs the worst in all cases. The RDMA Channel based zero-copy design performs very close to the the CH3 based zero-copy design. On average, the CH3 based design performs less than 1% better on both 4 nodes and 8 nodes.

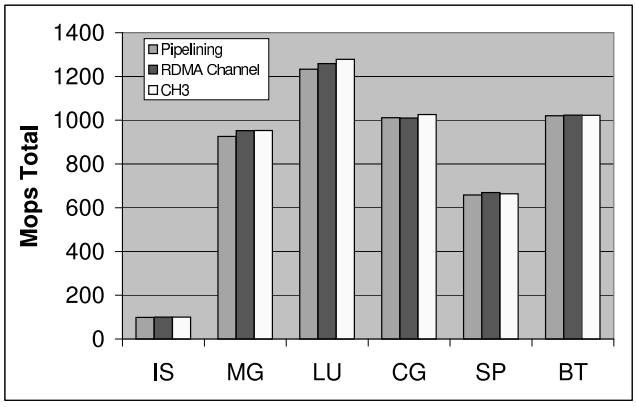

Figure 16. NAS Class A on 4 Nodes

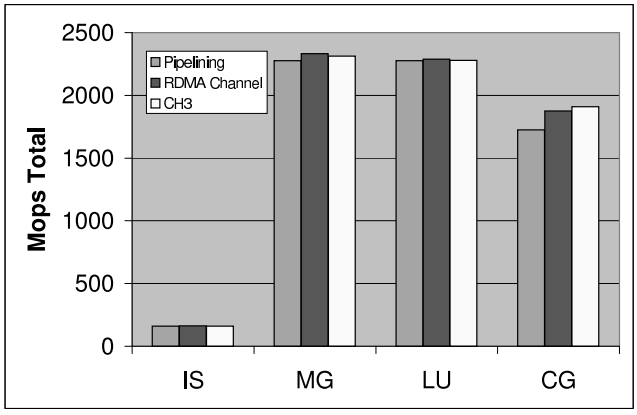

Figure 17. NAS Class B on 8 Nodes

8 Related Work

As the predecessor of MPICH2 and one of the most popular MPI implementations, MPICH supports a similar implementation structure as MPICH2. MPICH provides ADI2 (the second generation of Abstract Device Interface) and Channel interface. Various implementations exist based on these interfaces[3, 19, 22, 14]. Our MVAPICH implementation [17], which exploits RDMA write in InfiniBand, is based on the ADI2 interface.

Since MPICH2 is relatively new, there exists very little work describing its implementations on different architectures. In [2], a CH3 level implementation which is based on TCP/IP is described. Work in [24] presents an implementation MPICH2 over InfiniBand, also using the CH3 interface. However, in our paper, our focus is on the RDMA Channel interface instead of the CH3 interface. MPICH2 is designed to support both MPI-1 and MPI-2 standards. There have been studies about supporting the MPI-2 standard, especially one-sided communication operations [5, 11]. Currently, we have concentrated on supporting MPI-1 functions in MPICH2. We plan to explore the support of MPI-2 functions in the future.

Due to its high bandwidth and low latency, InfiniBand Architecture has been used as the communication subsystem in a number of systems other than MPI, such as distributed shared memory systems and parallel file systems [12, 15].

The RDMA Channel Interface presents a stream based abstraction which is somewhat similar to the traditional socket interface. There have been studies about how to implement user-level socket interface efficiently over high speed interconnects such as Myrinet, VIA and Gigabit Ethernet [20, 13, 4]. Recently, Socket Direct Protocol (SDP) [10] has been proposed which provides a socket interface over InfiniBand. The idea of our zero-copy scheme is similar to the Z-Copy scheme in SDP. However, there are also differences between the RDMA Channel interface and the traditional socket interface. For example, put and get functions in RDMA Channel interface are non-blocking, while functions in the traditional sockets are usually blocking. To support traditional socket interface, one has to make sure the same semantics are maintained. We do not have to deal with this issue for the RDMA Channel interface.

9 Conclusions and Future Work

In this paper, we present a study of using RDMA operations to implement MPICH2 over InfiniBand. Our work takes advantage of the RDMA Channel interface provided by MPICH2.

The RDMA Channel interface provides a very small set of functions to encapsulate the underlying communication layer upon which the whole MPICH2 implementation is built. Consisting of only five functions, the RDMA Channel Interface is easy to implement for different communication architectures. However, the question arises whether this abstraction is powerful enough so that one can still achieve good performance.

Our study has shown that the RDMA Channel interface still provides the implementors much flexibility. With optimizations such as piggybacking, pipelining and zero-copy, MPICH2 is able to deliver good performance to the application layer. For example, one of our designs achieves 7.6μs latency and 857MB/s peak bandwidth, which come quite close to the raw performance of InfiniBand. In our study, we characterize the impact of each optimization by using latency and bandwidth micro-benchmarks. We have also conducted application level evaluation using the NAS Parallel Benchmarks.

So far, our study has been restricted to a fairly small platform which consists of 8 nodes. In the future, we plan to use larger clusters to study various aspects of our designs regarding scalability. Another direction we are currently pursuing is to provide support for MPI-2 functionalities such as one-sided communication using RDMA and atomic operations in InfiniBand. We are also working on how to support efficient collective communication on top of InfiniBand.

References

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

[2] D. Ashton, W. Gropp, R. Thakur, and B. Toonen. The CH3 Design for a Simple Implementation of ADI-3 for MPICH with a TCP-based Implementation. http://www-unix.mcs.anl.gov/mpi/mpich2/docs/tcpadi3.ps.

[3] O. Aumage, G. Mercier, and R. Namyst. MPICH/Madeleine: A True Multi-Protocol MPI for High-Performance Networks. In *Proc. 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*, page 51, San Francisco, Apr. 2001.

[4] P. Balaji, P. Shivam, P. Wyckoff, and D. K. Panda. High Performance User Level Sockets over Gigabit Ethernet. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[5] S. Booth and F. E. Mourao. Single sided MPI implementations for SUN MPI. In *Supercomputing*, 2000.

[6] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, March/April 1998.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[8] H. Tezuka and F. O'Carroll and A. Hori and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Tech-

nique for Zero-copy Communication. In Proceedings of 12th IPPS.

[9] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.

[10] InfiniBand Trade Association. Socket Direct Protocol Specification V1.0, 2002.

[11] J. Traff and H. Ritzdorf and R. Hempel. The implementation of MPI–2 one-sided communication for the NEC SX. In *Proceedings of Supercomputing*, 2000.

[12] J. Wu and P. Wyckoff and D. K. Panda. PVFS over InfiniBand: Design and Performance Evaluation. In *Proceedings of International Conference on Parallel Processing*, 2003.

[13] J.-S. Kim, K. Kim, and S.-I. Jung. SOVIA: A User-level Sockets Layer Over Virtual Interface Architecture. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2001.

[14] Lawrence Berkeley National Laboratory. MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/index.html, August 2001.

[15] L. Liss, Y. Birk, and A. Schuster. Efficient Exploitation of Kernel Access to Infiniband: A Software DSM Example. In *Hot Interconnects 11*, Stanford University, Palo Alto, CA, August 2003.

[16] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *SuperComputing 2003 (SC '03)*, November 2003.

[17] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *17th Annual ACM International Conference on Supercomputing (ICS '03)*, June 2003.

[18] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1996.

[19] Myricom. MPICH-GM. http://www.myri.com/myrinet/-performance/MPICH-GM/index.html.

[20] Myricom. Socket over GM. http://www.myri.com/scs/-index.html.

[21] NASA. NAS Parallel Benchmarks.

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

[23] R. Gupta, P. Balaji, D. K. Panda, and J. Nieplocha. Efficient Collective Operations using Remote Memory Operations on VIA-Based Clusters. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

[24] W. Rehm, R. Grabner, F. Mietke, T. Mehlan, and C. Siebert. Development of an MPICH2-CH3 Device for InfiniBand. http://www.tu-chemnitz.de/informatik/RA/cocgrid-/Infiniband/pmwiki.php/InfiniBandProject/ProjectPage.

[25] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–The Complete Reference. Volume 1 - The MPI-1 Core, 2nd edition.* The MIT Press, 1998.