Optimization of Message Passing Libraries – Two Examples

Frank Mietke, Rene Grabner and Torsten Mehlan
University of Technology Chemnitz

December 1st, 2003

Abstract

The computation of large problems in scientifical and industrial fields needs for efficient computer systems. A rising number parallel computers are used to deliver necessary computation resources. The physical restrictions of circuit integration limit the speed of single processor solutions. The use of consumer components is a cost–effective way to built a parallel computer. The deployment of a high–speed network enables most parallel applications to run fast. Optimized communication libraries are important to enable the use of high–speed networks with parallel applications. This field is subject to active research and development.

1 An optimized Library on Top of SISCI

Parallel computations need for information exchange to compute meaningful results. The quality of the network may have a significant impact on the execution speed of some applications. The Scalable Coherent Interface (SCI) was designed to satisfy demands for high bandwidth and low latency. Applications running on clusters of workstations may benefit from SCI interconnections. As of today a single SCI link can transmit up to 320 MByte per second [15]. Also network latancy is much lower than expected from conventional network links. The latency period for transmitting a 4 byte message is considered to be about 1.4 μs. Compared with conventional ethernet communication over TCP/IP these values are realy impressive. The small delay of network transmissions is considered to be the highlight of SCI technology. Even the upcoming Infiniband technology has a latency larger than 4 μs [16]. Depending on the behaviour of the applications the use of SCI networks can lead to a significant speedup.

1.1 General Overview

Many cluster computers are equipped with a SCI network. The native interface to the SCI services is known as Software Infrastructure for SCI (SISCI). This specification defines a programming interface that follows the Distributed Shared Memory (DSM) paradigm. The programmer operates on memory areas that are shared by several processes. The participating processes may run on different nodes located anywhere inside the SCI cluster. In contrast to the SISCI interface the message passing paradigm exposes inter–process communication. The data transfer has to be handled by executing dedicated send functions and receive functions. This paradigm is widely used. Many distributed applications rely on the presence of message passing libraries. To enable the use of this kind of applications with SCI the VIA2SISCI library was developed. The aim was to implement a message passing library that uses SISCI functions to provide fast data transfer. The Virtual Interface Architecture (VIA) contains the definition of a message passing interface.

This interface is provided to the consumer of VIA2SISCI. To sum it up the VIA2SISCI library maps the semantics of VIA to the services of SISCI.

This report describes the architecture of VIA2SISCI. The discussion presents the differences between VIA and SISCI that had to be handled. The presented concepts may be interesting beyond the scope of this report. The discussion is organized as follows: At first the goals of the development process are presented. Next the report gives some fundamental knowledge about VIA and SISCI. After the reader became familiar with the underlying concepts the VIA interface is mapped to the SISCI services. The most important differences of both interface specifications are identified. Each subsection deals with special problems and solutions. This discussion leads to the current design of VIA2SISCI.

1.2 Definition of aims

The VIA2SISCI library is expected to serve as small middleware between the SISCI library and higher level communication facilities. Within this environment the perfomance requirements play an important role. Thus the VIA2SISCI library has to be highly optimized. The minimization of performance impacts is a priority aim. Extensibility and interoperability have to keep behind this aim. Nearly the raw SCI performance has to be delivered to the VIA2SISCI consumer.

VIA2SISCI implements the interface of the VI Provider Library (VIPL) as defined by [17]. This interface is flexible and easy to understand. The VI Provider Library was defined to serve as interface to communication facilities that are compliant to VIA [17]. The VI Provider Library contains pure message passing semantics. Nevertheless it is also possible to issue Remote Direct Memory Access (RDMA) requests. One should notice that RDMA in terms of VIA is not similar to DSM in terms of SISCI. On the whole the VI Provider Library follows the message passing paradigm.

The actual data transfer has to use SISCI services. Data transmission as well as event signalling has to run over SCI shared memory. Thus the high SCI performance is applied to the time critical tasks. Some operations don't need to be accomplished with full SCI speed. In these cases it is acceptable to use conventional TCP/IP connections. Nevertheless all payload data is transmitted using SCI connections. Only management tasks can run over TCP channels.

The VI Provider Library is adjusted to the capabilities of the VIA framework. In contrast to the assumptions of VIA the SCI hardware doesn't implement some necessary features. This restriction introduces several problems. During the VIA2SISCI development these problems had to be handled. The aim of the VIA2SISCI middleware is the complete support of all functions of the VIPL without affecting performance.

1.3 Fundamentals

This section deals with the participating interfaces and technologies. The basic concepts of VIA and SCI are introduced. This becomes necessary since the corresponding interfaces are designed to expose the native hardware capabilities. Understanding of some properties of the hardware helps to learn more about the corresponding interfaces. Furthermore the behaviour and properties of VIPL and SISCI are presented. This discussion gives important knowledge for understandig subsequent paragraphs.

1.3.1 VIA and the VI Provider Library

The VI Architecture was developed to serve as framework for high–speed networks. The specification describes several capabilities, logical entities and work flows. The actual layout of hardware and low–level software is not specified. The VIA specification has a lot of room for different implementations. In fact there is no real hardware available that fits to the VIA specification. Nevertheless many ideas of VIA were picked up by the Infiniband Architecture (IBA) [18]. The IBA interface is slightly different from the original VI Provider Library. However, the well-known M-VIA implementation [19] uses the VIPL to provide fast access to ethernet hardware.

The VIA framework assumes that the network hardware is able to directly access the user memory. Using this feature the network transfer can go on without operating system invocation. Moreover additional copies of the payload data are avoided. This capability is the basis for fast network transactions. To accomplish this task the network adapter has to maintain a memory translation table. This can not be done automatically. The consumer has to register the memory areas serving as communication buffers. Thus the network hardware is enabled to update the memory translation table and to lock the participating memory pages. As a result the memory pages stay persistent at their location. Thus the network adapter can access the memory safely.

If the memory wouldn't be locked down the operating system would be able to to change the location of the memory pages. The network adapter wouldn't be able to catch the corresponding event. Thus the memory translation table would become messy. The system could enter an inconsistent state. From this it follows that memory registration is necessary to achieve a proper behaviour.

Here a further problem emerges. The operating system loses total control about memory access. The network adapter performs access to the user space without operating system invocation. Thus the network hardware is expected to enforce protection of memory. The user process is enabled to restrict access to each registered memory area. To avoid performance impacts the network hardware has to support memory protection. Unauthorized access is denied.

The actual data transfer goes behind program execution. The user issues a request to a work queue. The corresponding library call is expected to return immeadetly. While program execution goes on the network adapter performs the data transfer. At any time the user process may decide to examine the status of the data transfer. In this way the process is notified about transfer completion and errors.

The VI Provider Library is designed with respect to the capabilities of the VIA hardware. The functions of the VIPL are suitable to issue data transfer requests, to receive notifications and to manage communication buffers. The user creates several Virtual Interfaces (VI). Each VI comprises two work queues. The send queue is used to issue transfers. In contrast the receive queue is used to handle incoming data. The items of the work queues are known as descriptor. The descriptor is a well–known data structure that describes data locations and operations. The user has to format a descriptor prior to posting it to the appropriate queue.

The VIPL provides connection oriented network access. Thus each VI has to be connected to exactly one VI to be able to perform network operations. Once a connection is established it remains persistent until disconnection is requested or an error occures. The data transfer takes place between the connected VIs. The VI connection setup is considered to be slow. Thus the user is required to create multiple VIs in order to prepare necessary communication channels.

There are four different types of descriptors. Normally the send descriptor and receive descriptor are used together. The send descriptor contains a gather list which is processed by the network hardware. The receiver has to provide a receive descriptor containing a scatter list. Alternatively one may use RDMA descriptors. Either RDMA read or RDMA write can be performed. This kind of data transfer doesn't need for a receive descriptor. Both the source buffer and the destination buffer is specified within the RDMA descriptor.

1.3.2 SCI and SISCI

As like VIA the SCI network architecture aims at support of high–speed communication. To avoid context switches and data copies the main memory is directly accessed. Unfortunately some SCI cards are unable to maintain a memory translation table. Thus the SCI memory is allocated before the operating system boots. The SCI hardware gains full control of the allocated memory.

The SCI hardware shares memory across different compute nodes. Once a user process publishes SCI memory other nodes are enabled to access the remote memory area. The SCI hardware cares about necessary network transactions, error conditions and memory access. However, there is no possibility to protect shared memory areas from unauthorized access. Once the memory is published each process running inside the SCI network is able to read or write the memory.

The SISCI interface provides direct access to remote memory. There is no full transparency provided since remote memory has to be treated by special functions. SISCI allows to create and to connect remote memory areas. Once the memory is available any access to it will start a network transmission. Conventional read/write access as well as copying blocks of memory with standard operating system functions and SISCI related functions is sufficient to transmit data.

Each public memory area used with SISCI has to be requested from the library. The specific memory area has to be prepared and finally it has to be set available. Once the memory is published in this way any other node is free to establish a connection. For remote nodes it is not sufficient to connect to the memory. They also have to map the remote memory area into the virtual address space. After this work is done the memory can be used as like as local memory would be used.

Among the support of Distributed Shared Memory the SISCI interface provides a signalling mechanism. Remote interrupts can be created to receive events from remote nodes. The management of remote interrupts is straight forward. Each node may create an arbitrary number of interrupts. Remote nodes may connect to any interrupt. When an event has to be indicated the node uses an SCI function to trigger the interrupt at the origin node. This mechanism may have blocking or non–blocking behaviour.

1.4 Implementing VIA semantics on top of SISCI

The support of fast message passing requires the use of SCI shared memory. A major part of the VIPL services have to be mapped to the SISCI interface. This section deals with the problems and solutions that emerged during the development process. The discussion separates several concerns. The problems of mapping the behaviour of VIPL are mentioned. Finally reasonable solutions are presented.

1.4.1 Connection Management

The VIA framework provides connection oriented communication. Each VI has to be connected to a remote VI. Only if an operational connection exists the corresponding VI starts processing descriptors. The VIA2SISCI middleware has to implement the connection setup. At the same time the procedure can be used to create necessary SCI resources. The asynchronous behaviour of VIA connections has to be supported.

The VIPL provides two different kinds of connection setup. The client–server paradigm is used to fit into asymmetric applications. The server process has to issue a connection request prior to the client requests. If a client request is detected the server receives a notification. Each client can either be accepted or rejected. The client request returns until the server responses or a timeout terminates the connection attempt. The appropriate functions are blocking. The functions return only if the remote end sends a response.

Against that the peer–to–peer connection paradigm is symmetric. Both processes are equal. The first request is buffered until a matching request arrives. It doesn't matter which process issues the first request. In contrast to the client–server model the peer–to–peer functions are asynchronous. The function returns immeadetly. The process may poll for the state of the connection request.

The connection setup doesn't need to be fast. The VIA specification states that the user has to expect slow connection setup. Thus VIA2SISCI is allowed to bypass SCI communication. The connection setup runs via TCP/IP. For most clusters the TCP/IP protocols use ethernet. The speed of connection setup is reduced to normal network speed in such cases. Each SCI cluster is expected to be equipped with a conventional ethernet.

There are several advantages of using ethernet for connection setup. At first this protocols are available on all machines. Especially clusters are equipped with TCP/IP connection for management purposes. Second the technology is well known and works stable inside all environments. Finally TCP/IP provides reliable data transfer. VIA2SISCI is not needed to care about network errors during connection setup.

VIA2SISCI implements a connection protocol. This protocol is used to handle both peer–to–peer requests as well as client–server requests. VIA2SISCI uses the connection setup to create some SCI resources. These resources are mainly used for management purposes. Both VIs are enabled to exchange information. For instance the receive descriptor buffer is created during connection setup.

1.4.2 Memory Management

This section addresses the memory management. Both SCI and VIA rely on special treading of communication memory. Any interference into the memory management of the kernel is slow. Both communication facilities are designed to avoid frequent invocation of the memory management functions. The challange is to provide the semantics of memory registration as specified by VIA [17] while preserving a good performance.

There are only four memory management functions specified by VIA [17]. The relevant tasks of memory registration and deregistration are provided by two functions. Before the memory can be used for communication a registration has to take place. During registration of memory the mapping between virtual addresses and physical addresses is stored inside the network card. Moreover the relevant pages have to be locked down into physical memory. The VIA2SISCI middleware doesn't care about details of memory managememt. The memory registration has to be expressed using SISCI functions. Due to the leading role of memory in the SISCI programming paradigm the memory administration scheme is more complex. Nevertheless most of the tasks can be accomplished during the initial memory registration.

The main problem comes from the lack of address translation capabilities of SCI. The SISCI interface denies any request for registering user space memory. One have to request the communication memory directly from the SCI subsystem. Allocation and registration is done by the SISCI library and the appropriate kernel modules. In contrast the VIPL functions don't support memory allocation. The user submitts a pointer to a memory location that was allocated prior to the request.

There are three possible solutions. The first one consists of the creation of a small kernel module. This module would be responsible for modifying the page table of the kernel. Thus the addresses of the user allocated memory can be remapped in order to reside in the memory area used by the SCI hardware. This solution would be the only one offering full transparency to user applications without affecting performance. Regardless of this major advantage the solution was dropped for several reasons. Consider the layout of VIA2SISCI. This middleware is built to run between the VI Provider Library and the SISCI interface. Providing a kernel module as essential part of VIA2SISCI would be messy. Furthermore manipulating the memory management of the GNU/Linux kernel is considered to be error–prone. The risk of causing the system to run instable doesn't justify the advantage. Moreover the portability would suffer. To sum it up it is not suitable to hack the kernel in order to hide the lacks of SCI hardware.

The second solution provides full compliance to the VIA specification. No kernel support becomes necessary. Unfortunately the performance will suffer. This solution is explained in the following. Since the user passes the pointer to a memory area VIA2SISCI is enabled to access the payload data. The description of the user buffer is saved. Moreover VIA2SISCI creates a SCI segment of the same length. Each time the user specifies this buffer within a data transfer the corresponding SCI segment serves as communication buffer. The user doesn't have to be aware of this fact. Obviously the user places the payload data into the registered memory area and posts a descriptor. During descriptor processing the properties of the memory are checked. The VIA2SISCI library copies the payload data into the local SCI segment. Finally the data transfer to the remote SCI segment is started. The receiver has to copy data back from the SCI buffer to the user memory. This becomes necessary since the user expects data to be present in the registered memory area.

The last suggestion behaves different from the other solutions. The semantics of the VIPL functions is changed. The user has to handle the differences. There are two new functions introduced by this solution. These functions are responsible for memory registration and memory deregistration. The original functions are deprecated. Together with the new functions a new behaviour is introduced. The user specifies the address of a pointer variable in order to receive the location of the communication buffer. Allocation and Deallocation is done by VIA2SISCI. Except this major difference the meaning of the parameters remains unchanged.

The second solution provides full transparency at the price of a performance loss. The payload data is copied two times. Since both VIA and SCI were developed to avoid data copies the initial aim would be missed. Instead the second solution doesn't suffer from additional data copies. However, many users don't like to modify the source code of their applications. But this becomes necessary while using solution three. To satisfy most requirements both the second solution and the third one are implemented. Thus the user can weigh the advantages and disadvatages. To start the applications immeadetly the second solution is suitable. Instead for high–performance requirements the user should call the new functions of solution three.

1.4.3 Starting Data Transfers

As mentioned in section 1.3 the most significant difference between VIPL and SISCI is the way of issuing data transfers. The VIPL interface provides a function that issues a data transfer. Thus network transfers are explicitely requested. The actual kind of data transfer is determined by the descriptor. The send descriptor and the receive descriptor provide pure message passing semantics. The data source is described by a gather list of the send descriptor. The data destination is given by the scatter list of the receive descriptor. The receive descriptor has to be posted by the receiver prior to the actual data transfer. The size of the receive buffers has to be larger or equal to the size of the send buffers. If these rules are violated an error is returned by the descriptors.

The RDMA descriptors behave different. There are either RDMA read or RDMA write operations available. Per default no receive descriptor is consumed. The remote memory location is accessed directly. The data is written to the local buffers in case of an RDMA read. The RDMA write may consume a receive descriptor if requested. This property can be used for synchronization purposes. As like send descriptors the RDMA descriptors report the state of the data transfer via the status field of the descriptor.

The first thing we have to keep in mind is the fact that data transfers are associated with at least one descriptor. The second thing we should remember is the main role of the VI in managing data transfers. There is never the possibility to read or write directly to the memory. Even an RDMA transfer is issued via a descriptor.

The SISCI interface provides shared memory. Each process is able to read and write memory areas that are shared by several processes. Data written into these SCI segments is visible to all participating processes. Thus the data transfer is hidden by the memory access semantics. Nevertheless SCI provides also some functions that explicitely start data transfers. There are three groups of data transfer functions. Either memory copy requests or block transfers or DMA transfers can be issued. Using the appropriate functions the user gains control about the mechanisms used to transmit data.

The VIA2SISCI middleware has to implement the meanings that are suitable to handle descriptors. Descriptor processing has to be mapped to the SISCI functions providing data transfer services. The scatter/gather semantics of VIA descriptors have to be translated in order to fit into the SISCI scheme. Among this issue there will be a need for address translation.

Consider the presence of a send descriptor and a receive descriptor. Thus a gather list and a scatter list is available. Obviously there must be a loop processing the elements of both lists in an iterative way. In other words the data compound is split into several contiguous data blocks. These blocks are transfered using the appropriate SISCI functions. VIA2SISCI has to start a new transfer for each block of data. Only DMA transfers are started at once after specifying all data blocks.

The presence of both send descriptor and receive descriptor is a prerequisite for beeing able to issue data transfers. The sending node has to give a precise description of the data transfer. Thus the receive descriptor has to be accessible at the sending node. This task may be achieved by placing the descriptors into SCI memory. The VIA framework requires each descriptor to reside within registered memory. Under the condition that registered memory is equivalent with SCI memory the sending node can read the descriptor directly. Alternatively receive descriptors have to be written to a buffer at the sending node.

The assumption that registered memory resides within SCI memory is not always true. Thus a suitable solution has to write the receive descriptor into a buffer. If this data transfer would occur each time a send descriptor is posted the delay would reduce performance. Thus each VI implements attempts to transmit several receive descriptors at a time. Although there is no guarantee this mechanism may help to avoid the overhead in most cases. The limiting factors are the size of the scatter lists, the number of available receive descriptors and the size of the buffer. Per default the VIA2SISCI library allows for 32 receive descriptors to be submitted.

1.4.4 Asynchronous Events

The VIA2SISCI middleware has to implement fast event management facilities. To provide good performance some events has to be delivered as fast as possible. Consider the case of a callback registration. The associated process relies on the invocation of the callback. Thus an event has to be signalled. Any delay that would be introduced by this event would delay the entire data transmission.

The VI Provider Library provides different types of event indication. Events are associated with descriptors. Once a descriptor is posted the state of the data transfer is accessible via the state of the descriptor. The user is allowed to poll for the completion of data transfers. Alternatively one may block until the data transfer completes. Finally a callback can be registered that is invoked on descriptor completion. These events has to be managed by VIA2SISCI in order to satisfy the expectations of the VIPL consumer.

The SISCI interface doesn't provide a native association between data transfers and events. Normally the user doesn't know about data transfers since memory access hides the distributed behaviour of the system. Nevertheless the data transfer functions can be used to determine the state of data transfers. In the simplest case a synchronous memory copy is requested. Thus the sending node is able to determine successful completion of data transfers.

The remote node has to be notified about data transfer completion. Thus the sender has to signal an appropriate event. The SISCI interface provides remote interrupts to signal events across nodes. A process has to create an interrupt locally. Remote processes may connect to this interrupt. An event can be triggered at any time. The owner of the interrupt may catch the event via a callback. Alternatively one may wait for the occurence of an event.

During the development of VIA2SISCI the remote interrupt mechanism caused a lot of problems. Thus the decision was made to avoid the use of remote interrupts. The VIA2SISCI middleware supports events by using SCI shared memory. An event manager is responsible for catching events. This module calls the appropriate handler functions as soon as an event is detected. The trigger of an event has to directly interact with the remote event manager.

1.4.5 Address Translation

The VIPL interface and the SISCI interface use different address modes. Since the VIA hardware is able to resolve virtual addresses the user specifys only virtual addresses within descriptors. Against that the SISCI interface is unable to resolve virtual addresses. This is always true for receive buffers residing at remote nodes. Thus the user specifies the identifier of the appropriate SCI segment together with an offset. The VIA2SISCI library has to take care for proper translation of virtual addresses.

To determine the virtual addresses of the receive buffers the sending node has to access the receive descriptor. Since the appropriate descriptor is considered to be present the information can be easily accessed. This was discussed in section 1.4.3. To be able to calculate the offsets the VIA2SISCI middleware needs access to further information. The start address of the SCI segment has to be provided. Since SCI segments has to be connceted to enable remote access the necessary information can be fetched during the connection process.

Once the addresses of the remote SCI segment is known the offset can be calculated. Optionally the VIA2SISCI library has to handle shadow SCI segments. In section 1.4.2 the memory registration was discussed. When shadow SCI memory is used the VIA2SISCI library has to copy data prior to the transfer. The receiver has to copy the data to the user buffer. This has to be done with respect to the original addresses. Since this operation is local the necessary information is available at the node.

2 An MPICH2-CH3 InfiniBand – Device

MPICH2 is the next step in implementing the MPI-Standard. It aims to fully support the MPI-2 standard as well as the MPI-1 standard. Due to a complete redesign, MPICH2 is also cleaner, more flexible, and faster. The InfiniBand network technology is based on an open industry standard and provides high bandwidth and low latency, as well as reliability, availability, serviceability (RAS) features. It is currently spreading its influence on the market of cost-effective cluster computing. The first big cluster with InfiniBand is the $5.2 million "Big Mac" at Virginia Tech. It reaches 10.28 TFlops and is now the number 3 of the top 500 Supercomputer list. We expect for the near future that upcoming requirements in many cluster environments can only be satisfied by the functionality of

MPICH2 and the performance of InfiniBand. Hence, there is the need for an effective support of the InfiniBand interconnect technology by MPICH2.

In this section we present our experience while implementing, an performance overview, as well as problems and ideas for future developments of our MPICH2 Device for InfiniBand[1]

2.1 InfiniBand

InfiniBand is a play on the words "infinite bandwidth". It is an advanced interconnect technology for interconnecting processor nodes and I/O nodes to form a SAN. This section provides an short overview about the InfiniBand architecture, communication stack and verbs API.

2.1.1 Architecture

InfiniBand is a point-to-point, switched I/O fabric architecture. Point-to-point means that only two devices have a communication link and only these devices have full and exclusive access to this communication path. To build a fabric, switches for interconnecting multiple points come into play. More switches can be added to increase the aggregated bandwidth of the fabric. By adding multiple paths between devices, switches also provide a greater level of redundancy. In figure 1 an example of such a SAN is given.

In the figure 1 all relevant components are shown. First there is the HCA (Host Channel Adapter) that resides in a host processor node and connects the node with the IBA fabric. Further there is the TCA (Target Channel Adapter) that reside in an I/O node. It functions as the interface to an I/O device, e.g. an Storage Area Network. Routers handle outbound, or inter-subnet traffic, passing packets across the fabric from an end node within the router's subnet to another subnet, based on the packets' global routing headers. And last but not least there is the Subnet Manager that is responsible for configuring the local subnet and ensuring its continued operation. Configuration responsibilities include managing switch and router setups and reconfiguring the subnet if a link goes down or a new one is added.

2.1.2 Communication Stack

InfiniBand was developed with the VIA architecture in mind. It off loads traffic control from the software client by the usage of execution queues. This means that a consumer (consumer = process or thread) queue up a set of instructions that the hardware executes. These queues, called WQ - Work Queues, are always created in pairs. The QP - Queue Pair is composed of one WQ for send operations and one WQ for receive operations. Each consumer may have its own set of work queues, each pair of work queues is independent from the others. Each consumer creates one CQ - Completion Queue or more and associates each send and receive queue to a particular CQ. It is not necessary that both work queues of a QP use the same CQ. The communication stack is illustrated in figure 2.

Each WQE - Work Queue Element, pronounced "wookie", holds an instruction, that is when the channel adapter executes the WQE it executes this instruction. These instructions are divided into two classes, the memory semantic (RDMA, Atomic) and the channel semantic (Send/Recv) operations.

[1]The research is part of the project 7455/1180 and funded within the framework for technology promotion by means of the European Fund for Regional Development (EFRE) 2000-2006 as well as by means of the SMWA Saxony ministries.

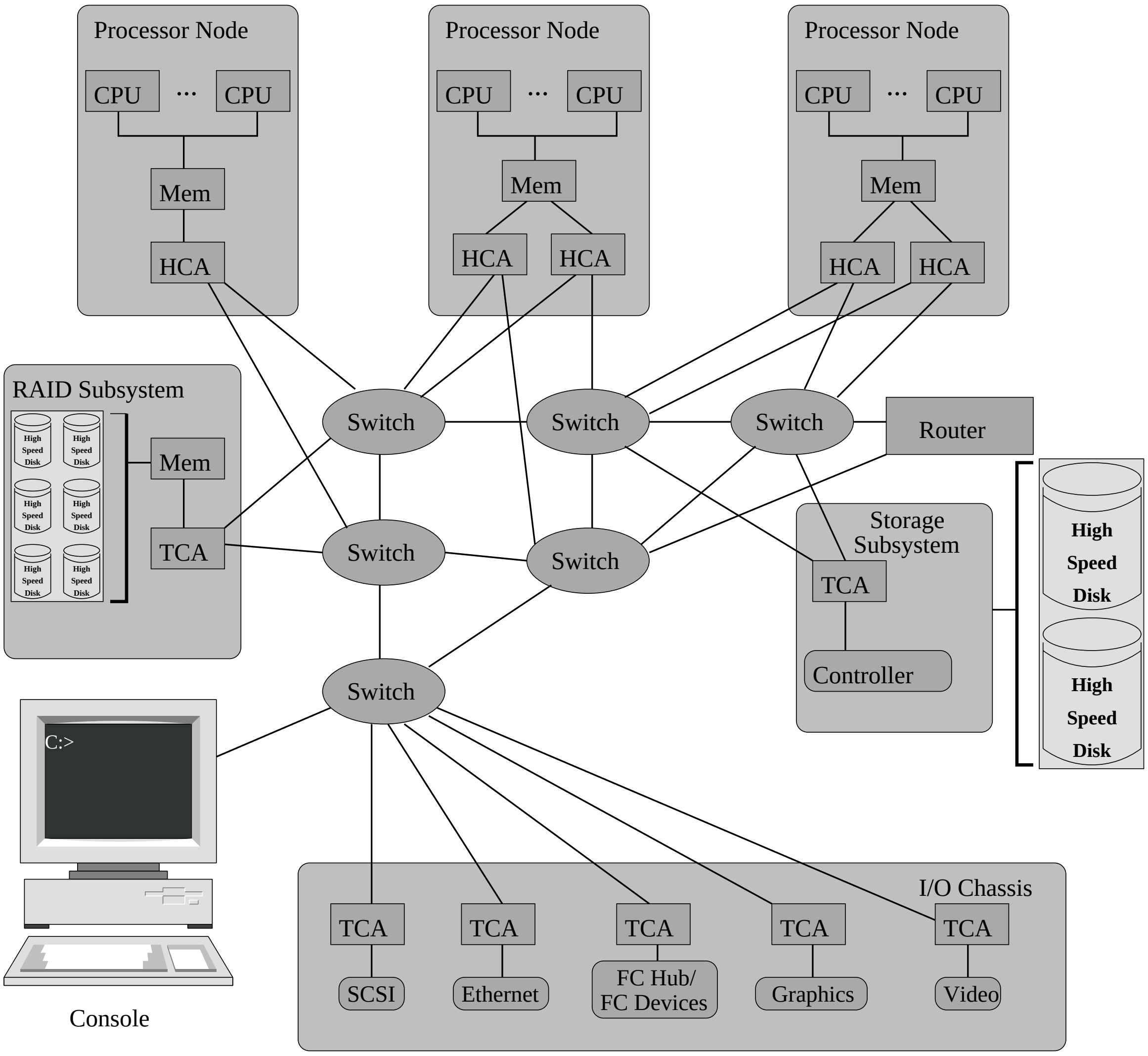

Figure 1: IBA - System Area Network

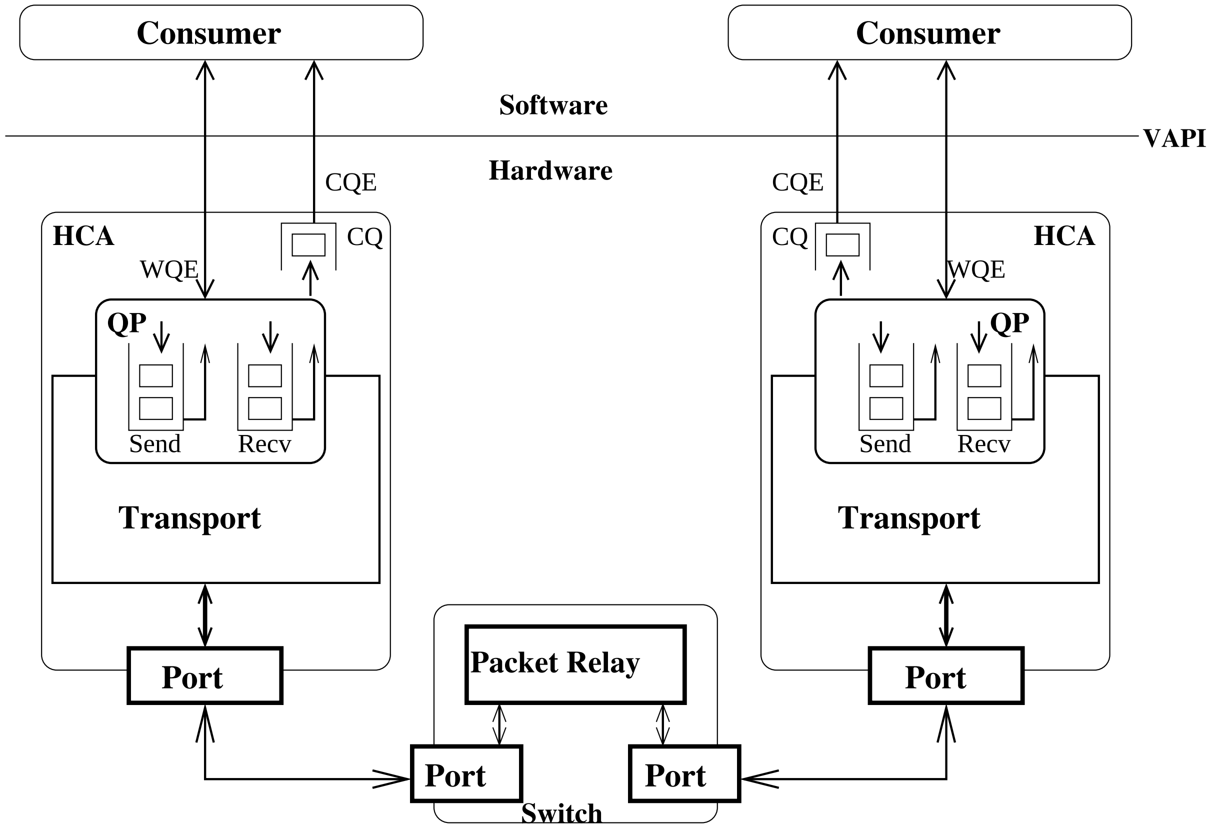

Figure 2: IBA Communication Stack

2.1.3 Verbs API

IBA describes the service interface between a host channel adapter and the consumer by a set of semantics called Verbs. Verbs describe operations that take place between an HCA and its consumers based on a particular queuing model for submitting work requests to the channel adapter and returning completion status. The intent of Verbs is not to specify an API, but rather to describe the interface sufficiently permitting, for example the operating system vendors, to define appropriate APIs that take advantage of the architecture. Verbs describe the parameters necessary for configuring and managing the channel adapter, allocating (creating and destroying) QPs, configuring QP operation, posting work requests to the QP, getting completion status from the CQ. One implementation of the Verbs is the VAPI from Mellanox. This API is utilized within the MPICH2 Device for InfiniBand.

2.2 MPICH2

The second version of MPICH is a complete redesign [1]. It is being developed by the MPICH Team at Argonne National Laboratory and Mississippi State University, and is both a research project into MPI implementations and a software development project. Goals of the current MPICH2 development are to fully implement the MPI-2 standard, to scale up to a large number of processes, high performance, thread safety and modular design.

A new and full featured Abstract Device Interface (ADI-3) has been proposed, which provides routines to support MPI-1 as well as MPI-2. The intermediate Channel Layer implements the ADI-3 interface and in turn provides the still hardware independant but quite compact Channel Interface CH3 [2]. Systems targeted by this redesign are above all clusters connected with conventional networks (such as Ethernet) or networks that provide support for remote memory access (such as InfiniBand), large scale SMP systems and experimental systems used for research activities. The layered approach of MPICH2 is shown in figure 3.

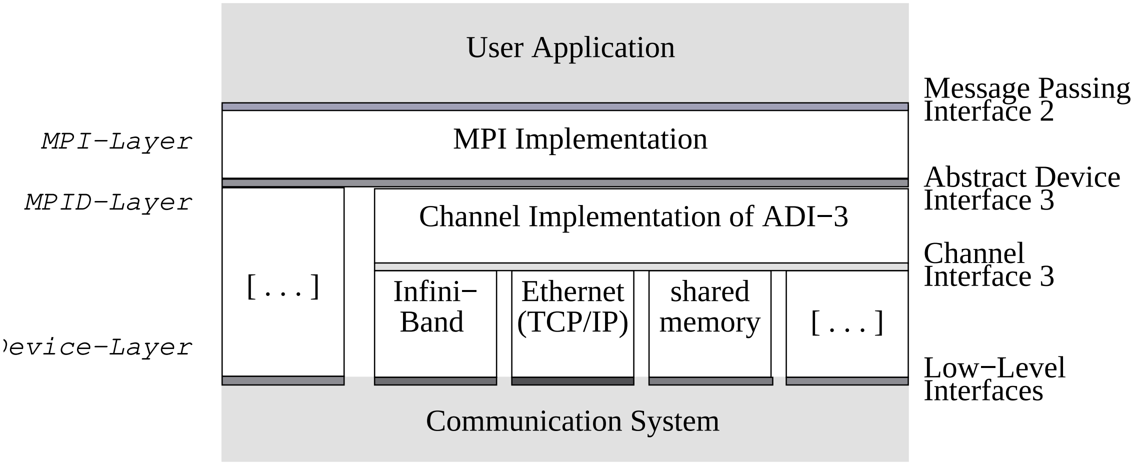

Figure 3: Layered architecture of MPICH2

The three major enhancements in MPI-2, that will also be implemented by MPICH2, match major advantages of the InfiniBand architecture. One sided communication primitives such as *put*, *get*, and *accumulate*, also known as remote memory operations, can be implemented in terms of InfiniBand RDMA operations. Dynamic process management matches the high flexibility and scalability of InfiniBand, which enables the usage of systems with thousands of computing nodes. Moreover, InfiniBand's high bandwidth and low latency plays perfectly with the parallel I/O functionality of MPI-2. Due to the ever-increasing requirements for high performance communication and high volume storage of today's and future's parallel applications, programmers will be forced to use these advanced features to be able to solve their problems in an efficient way.

2.3 Prototype

As shown in section 2.2, the CH3 is the lowest interface within MPICH2. It is quite small, which results in a relatively small number of interface functions. A device that implements the CH3 interface is responsible to transfer the data of passed buffers in a point-to-point manner to another process, and on the remote side to receive the data and give it back to the CH3 layer. What is special about that, is all communication has to be performed in a non-blocking way. This means, for instance, a process that wants to receive data into a buffer cannot block and wait without doing anything else until the data for this very buffer have been received. If all processes of the distributed application were behaving in this way, it would easily lead to deadlock situations among them. Furthermore, it would not be possible to implement non-blocking operations, as requested within the MPI standard, at the higher layers.
Not only the MPICH2 implementation itself uses a layered approach, but also the InfiniBand device consists of two layers. These are coupled very tightly for performance reasons, but can be distinguished. A major reason for that approach can be seen in the requirement for non-blocking communication operations. The higher level implements all of the functions of the channel interface, and additionally several auxiliary functions, both either as C functions or as preprocessor macros. They can be grouped as follows:

- *Send/Receive*
- *Request handling and queue management*
- *Progress*
- *Init/Final*

The higher layer's comprehensiveness is currently 24 functions to implement the CH3 interface and an additional number of 16 auxiliary functions and macros.
Although none of the functions of the higher level does directly access the Verbs interface, it cannot be seen as one more hardware independent implementation. There are a lot of details, especially within the data structures, that are very special.

The lower layer of the device provides basic functionality for the layer above it. It does not need any knowledge about the higher level request objects. The functions can also be classified into several groups, which also gives an overview of this layer's major tasks:

- *Buffer management for send/recv*
- *Flow control for controlled communication*
- *Posting of work requests*
- *Progress for polling the CQ*
- *Init/Final:*
- *Interface and Auxiliary to access internal data*

This layer directly invokes the routines of the Verbs implementation, so it is very dependent on the hardware and on the Verbs programming interface. So, our first step in realizing the prototype was to migrate the existing InfiniBand device of MPICH2 to the Mellanox VAPI.

In this section not only the prototype is presented, it is also discussed what extensions would bring an performance gain, where are bottlenecks or scalability problems. This should show whereof we work and think about.

2.3.1 Communication Protocols

MPICH2 usually uses two internal communication protocols, eager and rendezvous. The information that is transported is divided into two parts: the packet header, which contains the information about the message (This could be an MPI envelope, request ids...) and the data. The packet header is delivered to the destination immediately if possible, see also section 2.3.3. Dependent on the communication protocol the data is or is not transported at once with the packet header.
In the eager case the data is immediately transfered with the header and without waiting for the receiver. This protocol is used for small packet sizes. The rendezvous protocol initiates a handshake before sending all the data. So, in this communication protocol the sender waits until the receiver requests the data.
Both protocols fits perfectly in the channel (eager) and memory semantics (rendezvous) of InfiniBand architecture. For the first prototype we decided to implement channel semantics only.
The interesting thing in communication protocols is the rendezvous case, because in this case a handshake is performed first. So, we have to think about which information is important to deliver during the handshake. In the later discussed RDMA implementation we use exclusively the rendezvous protocol but the RDMA itself makes a kind of handshake after the rendezvous. Because only when the RNDV_SEND header has arrived the

receiver initiates an appropriate read operation. We think this behaviour is improveable if we extend some of the header information so that the second handshake in the RDMA implementation is fully integrated in the first one.

2.3.2 Pre-registered Memory

A problem that exists in conjunction with InfiniBand is the very expensive un-/registration process. To send data from or receive into a buffer, it is necessary to register the buffer before this. After successful send/receive the buffer can be deregistered. Because of the behaviour of the un-/registration process, an kernel involvement is necessary. This should be avoided. To achieve this the concept of pre-registered memory was integrated.

In the initialization stage a memory region is allocated and registered for each connection. In the finalization stage the registered memory regions are deregistered and freed. Unfortunately this concept has some limitations.
The first one is a scalability problem. If you pre-register memory (e.g. 1MB) for each connection and you have a 512-Node cluster than you need 512MB pre-registered memory. So, this approach must be reimplemented in this way, that the pre-registered memory exists only for the process and not for each connection. And it must be as scalable as possible so that it is not limited to a specific node number.
The second problem is that for large messages the approach of pre-registered memory is not convenient, because there is a trade-off between short and large messages and also the network performance. As stated in [10] the memory is divided into small blocks, so that not so much memory is wasted for small packets and the network performance is at maximum. But for large messages the process have to copy into so many small blocks that it is unprofitable. One solution for the second limitation is discussed in chapter 2.4.

2.3.3 Flow Control

In order to prevent that the remote nodes memory is exhausted, in every communication system a flow control mechanism should be implemented. Such a mechanism is, of course, also implemented in the delivered device. This mechanism is a static credit based flow control scheme. This concept leads to a predictable memory usage per queue pair connection. This is necessary to prevent memory exhaustion.
In the same manner as the pre-registered memory approach is reimplemented to achieve better scalability the flow control scheme have to be adjusted. For better scalability the static scheme should be changed to a dynamic one.

2.3.4 Unexpected Data

Since the receiver expects a header at first - in order to decide what is the next action - the also delivered data must be stored in a buffer. This data that comes along with the header is called Unexpected Data. It is also possible that the receiver expects nothing, thus the header and the data must be stored. The unexpected data queue is organized as FIFO. To achieve the behaviour of the header first concept, the first read of a packet header is constituted in the connection setup. So, all communication nodes wait at the beginning on a packet header. After completion of a receive request, a new read of a packet header is immediately started.
This must be also adjusted for better scalability in the same fashion as the pre-registered memory and the flow control.

2.3.5 Progress Engine

The progress engine is responsible for dispatching an send/receive request and to trigger outstanding send requests in the send queue or reload a receive request. For this the `MPIDI_CH3_Progress()` function uses the function `ibvapi_wait()`, which checks a `unex_finished_list` for completed receives, that have got the data only from the unexpected data buffer, and polls the completion queue of the InfiniBand HCA.

One thing to note is following. While the progress engine is active and waiting, it does a hot polling on the completion queue. This means an eager iteration over the VAPI completion queue poll function is performed until there is a new entry. This consumes all of the available CPU time. Advantage of this concept is that the progress engine gets to know about a newly completed work request as soon as possible, which in turn results in very low overall latencies for communication functions. A disadvantage is, of course, a lot of CPU performance is wasted instead of being used for meaningful operations. Currently, MPICH2 is monolithic and single threaded, and if the user application does not use more than one thread, a waste of CPU cycles has no negative impact and can be tolerated.

2.3.6 Debugging

Our debugging concept is similar to the one that exists in MPICH2. We notice the function entry and exit, and we save some important variable values. Further the debugging was extended by the feature of profiling, so that we can measure the time a function takes. The result of a debugging run is a complete work flow of the device structured as a tree. With this mechanism, we have found some inconsistencies and failures in the original InfiniBand device. Among them there was a problem during the connection setup. The debug messages revealed for instance that only under certain circumstances a connection was established. All these problems have been corrected by means of the debugging facility.

2.3.7 Performance Evaluation

A comparison between the prototype and the achievable performance on the raw VAPI level should show how fast the CH3 device implementation is. For this we have taken the PingPong test from the Pallas test suite for the prototype and perf_main from Mellanox for the raw VAPI measurements.
The performance results showed that the maximum bandwidth, which is achieved at a message size of four Megabytes, was 429 Megabytes per second, the latency for small packets was about 10 microseconds. These values were not very good, but we expected them. In the next subsection it is shown that some concepts must be reimplemted to fit better in the InfiniBand architecture.

2.4 Improvements

This chapter describes improvements and extensions to the prototype, problems during their implementation, and presents performance results of the improved version(s). The two main improvements (RDMA, DME) were chosen for this description.

2.4.1 RDMA

InfiniBand distinguishes channel semantic- and memory semantic operations, see section 2.1. In the prototype device the channel semantic operations were utilized only. With regard to RDMA as a form of memory semantic operations, the channel semantic operations have to make at least one extra buffer copy. The prototype needed two extra copies, because of the very expensive un-/registration (It is so expensive because the operating system kernel is involved and interaction between the HCA and the host is needed) [6]. The un-/registration issues are discussed in section 2.4.2.

To avoid the expensive un-/registration the prototype uses a pre-registered memory area which is divided into small blocks. These blocks can be used to hold the received data or the data for the transfer to the destination. It is important to keep in mind that the HCA can only transfer from and receive into registered memory regions.

There is also another aspect that must be take into account. MPICH2 distinguishes the eager and the rendezvous protocol. In the first the message is directly sent with the header and in the latter case, before a message is sent, a hand-shake protocol is performed. This shows that in the eager protocol case the receiver need not be ready (The InfiniBand consumer must be ready), this fits best with channel semantics. The rendezvous protocol instead requires that the receiver is ready to get the data. In this case the extra copies could be avoided, when an RDMA mechanism will be implemented. But this leads to the problem that the application buffers must be registered for the use by InfiniBand.

The Solution: RDMA-Write
RDMA has several advantages in opposite to normal Send/Recv operation like:

- it is an one-sided operation that can read from or write into a remote application buffer
- it avoids the fragmentation of the application buffer, so that it fits in the small blocks of the pre-registered memory
- zero-copy is possible

But it has also some disadvantages (restrictions) like:

- expensive un-/register of the application buffer is necessary
- the destination address, length and the r_key must be known in advance

Two possibilities exists to realize RDMA, these are RDMA-Read and RDMA-Write. The comparison between both is illustrated in figure 4. It shows that in the RDMA-Read case an additional synchronize message is necessary to complete the operation, because the sender site does not know if the memory is reusable or not (The DMA-Engine transfers the data from the sender to the receiver, the remote process is not involved). In the RDMA-Write case a special flag is used, so that no additional synchronize message must be sent. This flag is the immediate data flag. If the RDMA-Write operation has been succesfully completed, the sender gets an completion queue entry. And in the last sent packet the immediate data is transfered, this leads to an CQE on the receiver site. After this, now both knows that the application memory is ready for reuse. So RDMA-Write with immediate data is more suitable than RDMA-Read.

For a first implementation of this mechanism this is sufficient. But there is another problem with "RDMA-Write with immediate". The advantage we see is that we get on sender and receiver side an appropriate CQE. But it is possible that this CQE in big

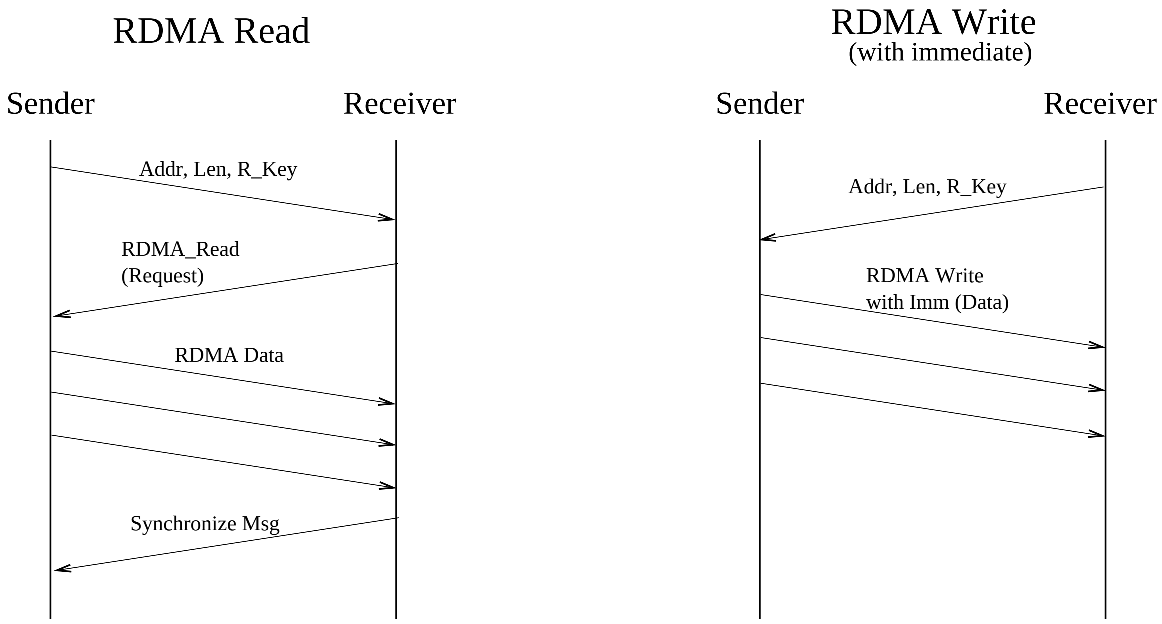

Figure 4: RDMA-Read vs. RDMA-Write

cluster systems is quite at the end of the queue. To achieve a better performance it must be possible to accelerate the recognition of a completed RDMA transfer. In [13] a so-called "RDMA polling set" have been introduced but the test results are only from an eight node cluster. And in this case it seems that all nodes are in the polling set. There are no practical tests on bigger systems published. But the approach is interesting and shows that RDMA completion with other means can improve the performance.

RDMA-Write Implementation

First of all, RDMA is used only in combination with the rendezvous protocol of MPICH2. The main conditions that must be fulfilled are:

- rendezvous protocol is used
- the data which should be transported is a contiguous memory block
- the memory block is of certain size

For the sender this means that the function `MPIDI_CH3_iSendv()` is called with the argument `iov_n` set to 2, that is a header and a contiguous data block are to be sent. Further the type field of the header has the value `MPIDI_CH3_PKT_RNDV_SEND` and the length of the contiguous data block is greater or equal than `IBVAPI_RDMA_MIN_PCK_SIZE`. The last value is set in the init stage to 32000 bytes. For the receiver follows from it that the function `MPIDI_CH3_iRead()` is called with a receive request that would like to receive one contiguous data block. This means that the parameter `iov_count` (this is equivalent to `iov_n`) of the receive request is set to 1. The length of this data block should be greater or equal than `IBVAPI_RDMA_MIN_PCK_SIZE`.

If the prerequisites have been fulfilled, RDMA can take place. On the sender site the RDMA operation begins. When the conditions in `MPIDI_CH3_iSendv()` are fulfilled the RNDV_SEND header is sent with the function `ibvapi_write_rdma_hdr()`, which posts a normal send request to the InfiniBand work queue and stores the send request structure for later usage. If the header have been successfully sent (a CQE have been enqueued in the CQ) the sender registers the contiguous data block for the HCA.

The receiver receives the header and processes it. Now the function `MPIDI_CH3_iRead()` is called and calls on his part the function `ibvapi_post_readv()`. In this function the conditions for the receiver site are checked. When the conditions are fulfilled the receiver site register its receive buffer. After successful registration the important values for the sender are stored. These are the start address, the length and the r_key. These values are sent with `ibvapi_write_rdma_info()` which posts a normal send request.

After the receiving of these values on the sender site, the RDMA operation is initiated with `ibvapi_write_rdma_data()` which posts a work request to the WQ with the instruction to perform an RDMA-Write with immediate.

When the RDMA operation has been successfully completed on the sender site, the memory is deregistered. Now it can be completed by the progress engine. The same happens on the receiver site when the last data packet have been received. The completion is signalled on both sites by a CQE.

This implementation permits that one site can be initiator and destination of an RDMA operation at the same time. For this two preposted receives are reserved for the receiving of the header and the RDMA info packet. For the transfer of these packets, two memory blocks from the preregistered memory region are reserved also. This shows that an RDMA operation can take place at each time. It must not payed attention on send queues. To handle a further RDMA-Write with immediate at a time an RDMA-Queue is introduced, so that only one RDMA-Write operation per connection is active. After completion of the active RDMA-Write, the queue is checked if further RDMA-Writes are pending. The queue is ordered as FIFO.

As stated above the RDMA operation consists of two handshake protocols. The first is the rendezvous handshake and the second is the transmission of the RDMA relevant parameters (r_key, address, length). The merger of both handshakes could be a first improvement. Therefore the RDMA parameters must migrate from the connection structure to the request structure. If this is changed then the device is also not limited to only two RDMAs at a time. Another advantage is that the minimum packet size for the rendezvous protocol could be decreased and smaller messages can profit from RDMA.

Another aspect to take into account is the fact that RDMA operations have better performance properties than normal Send/Recv operations. As shown in [13] an implementation of the eager protocol in terms of RDMA-Write can speed up the transfer performance of small and control messages (e.g. header, acknowledgements).

2.4.2 Dynamic Memory Engine - DME

RDMA transfers have several advantages over the message passing mechanisms. A major one is the possibility to implement a zero-copy protocol for both, sender and receiver. That can be accomplished by registering the user buffer on the HCA and using it directly as send respectively receive buffer. The most obvious and simplest solution is to register the buffers on both sides, perform the RDMA transfer, and finally deregister again to free claimed resources. But this would not yield to an improved implementation as we will show later. Both, registration and deregistration are expensive operations. Their costs are functions of the memory region's size to de-/register. The allocated memory region needs to be pinned within the physical memory, which invokes a jump into the operating system kernel. Further, the HCA has to set up tables for translations from virtual addresses used by the application and physical addresses used by the DMA engine.

In most distributed applications the probability of one allocated user buffer to be used for communication several times is very high. Hence, costs for re-registration could in many cases be saved if there was an intelligent mechanism that previously did actually not deregister that buffer. Intelligence is necessary to not run into a shortness of resources.

Basic data structures for the DME implementation are a hash-table with has chains, memory entry records that describe one registered memory region, and additionally a few lists are involed.

The registration function implemented by DME `ibvapi_mem_REGISTER()` firstly tries to find an memory entry record within the hash-table that corresponds to the passed buffer address and buffer size. The hash-value is used to directly access the hash-table. If there is an memory entry record a comparison of its memory region information and the actual one informs about their conformance. The actual page number must be equal to the one of the memory record, the actual number of pages might be less than the one stored within the memory record; that is only a part of a previously used buffer is actually being used. So, on conformance the registration function increments the memory region record's reference counter, removes the record from the unused list if it is an element, and immediately terminates. The costs for a buffer re-registration was reduced. On non-conformance the next entry within the hash-chain, if it exists, is examined. If no matching memory entry record can be found, the function searches for a new and empty record. That can be obtained from the free list, if there is at least one element. Otherwise, all of the records allocated during initialization have already been used. Instead of allocating more of them, it is tried to evict an unused memory region and actually deregister it on the HCA. The candidate for eviction can be found on the tail of the unused list, if the list is not empty. As mentioned above, entries are added on the head of the unused list, so that implements a simple LRU strategy. The reclaimed memory entry record can now be used for the new memory region that will actually be registered. If there was no entry in the unused list, which should only happen in very few cases, the registration function indicates that by returning a `NULL`-pointer. So the calling function can react appropriate.

The deregistration function of DME `ibvapi_mem_DEREGISTER()` is very lightweight. It decrements the memory region record's reference counter. If it reaches zero the record will be added to the unused list.

The concept of the DME implemented within the MPICH2 Device for InfiniBand is based on the ideas of [12], and the implementation is partly derived from the MPICH VIA device implementation MVICH [11].

This approach of lazy unregistering increases the performance clearly. But it has also a disadvantage. If you have two memory regions and you register the first one that is two times larger than the second one and the second memory region begins for instance at the third page of the first memory region and fits completely in the first one then you have to register the second memory block again. This problem is an inherently limitation of the hash algorithm, because the beginning address of the first memory region results in another hash value than the second one. A new algorithm that considers this problem further increases the performance on the one hand and save resources on the other hand.

Both, RDMA and DME, improvements are explained more in detail in [10].

2.5 Performance Results

In this section we present several performance results. Within the first subsection we refer to the improvements mentioned, and afterwards we compare the the current version of the device to other implementations.

The hardware basis for all of the tests was a six node cluster at the our research lab with following components:

- Dual Intel Xeon 2,4 GHz
- SuperMicro motherboard X5DMS-6GM
- 2048 MB DDR Memory, PC2100, with ECC
- Mellanox InfiniHost MT23108 Cougar, 10Gb/s, 4x, dual port, PCI-X 133 64bit

The systems were connected by an InfiniScale MTEK43132 eight 4x Port InfiniBand switch. The nodes were running Red Hat Linux 7.3, kernel *2.4.23-pre3* from kernel.org. The Mellanox host channel adapters were using firmware version *fw-23108-rel-2_00_0000-rc14-build-001*, the software development and drivers kit version was *thca-x86-1.01-build-001*. All compilations were done using the GNU C compiler *gcc-3.3.1*.

2.5.1 Achievements of RDMA and DME

The improvements of using Remote Direct Memory Access and an intelligent memory caching engine, that were discussed within the subsections 2.4.1 and 2.4.2 have been implemented within our MPICH2 Device. In figures 5 and 6 we prove their impact on communication performance using Pallas MPI Benchmark. One pair of graphs, labeled *wo-rdma_wo-dme* show the behaviour of the prototype, the other pairs of graphs have enabled RDMA support and disabled DME (*w-rdma_wo-dme*), and both improvements enabled (*w-rdma_w-dme*), respectively. Up to a message size that is smaller than the abovementioned minimum RDMA packet size, the improvements do not influence the performance and the implementations behave identically. For larger messages the RDMA enabled implementation performs very bad, and loses about two third of the bandwidth in case of PingPong and even more in case of SendReceive. The reason is the domination of registration and deregistration of the user buffer, that is performed before and after each RDMA transfer. With an enabled DME this issue is solved. The expensive registration of the user buffer only takes place one time – before the first transfer. If the buffer is reused later, which is the case in many MPI applications, the registration is skipped as the memory region is still being registered.

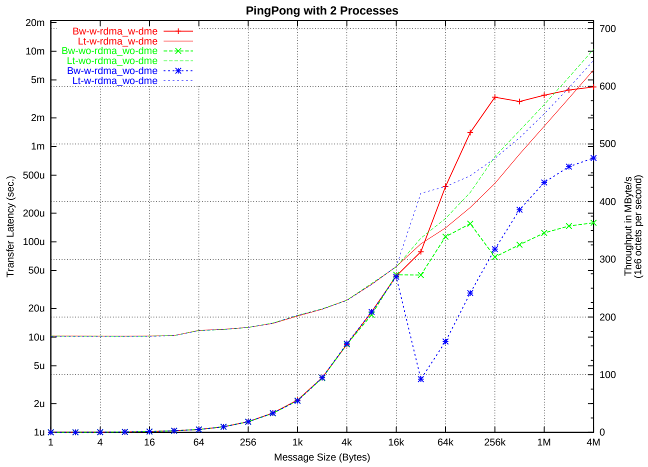

Figure 5: PingPong with and without RDMA and DME

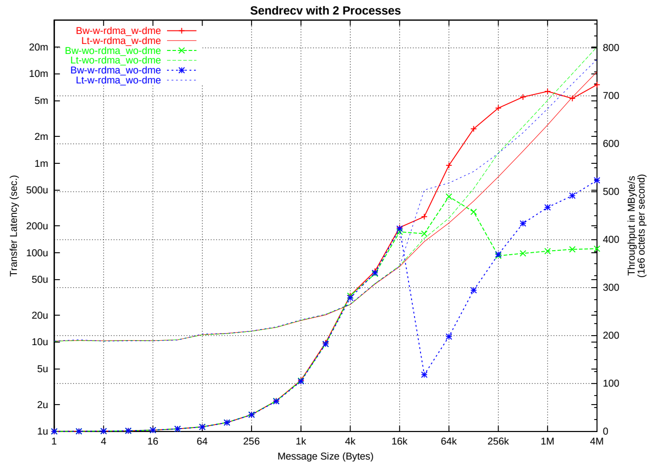

Figure 6: SendReceive with and without RDMA and DME

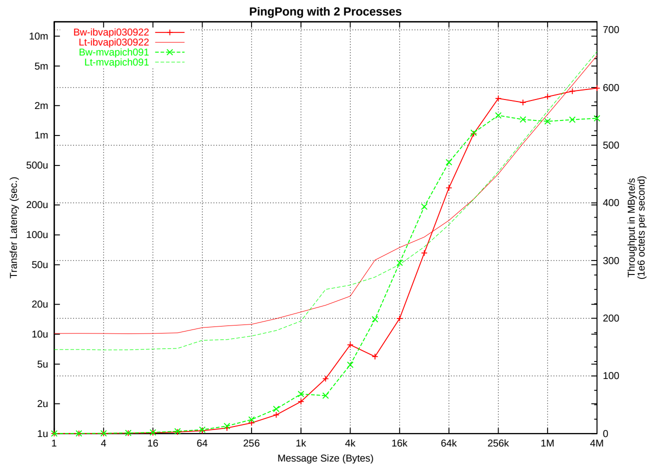

Figure 7: Pallas MPI Benchmark: PingPong

2.6 Comparisons

Figures 7 to 9 exemplary show results of Pallas MPI Benchmark (PMB) test cases. In the PMB Barrier test the MPICH2 InfiniBand device is faster than OSU MVAPICH. Following table shows the average the timings in microseconds for up to six nodes. Further research will be done in the future to find answers on the scalability when more nodes are involved.

# nodes	MPICH2 IB	MVAPICH
2	10.37	11.09
4	22.33	23.96
6	30.82	37.77

2.7 Related Work

There are a few related projects that are focusing on MPI in combination with InfiniBand. A very popular and one of the first projects brought up the MVAPICH [5] implementation. It is being developed at the Network-Based Computing Laboratory of the Department of Computer and Information Science at Ohio State University. The implementation is based on MVICH 1.0 [11] and MPICH1, respectively, and also utilizes the VAPI implementation.

VMI 2.0, being developed at the National Center for Supercomputing Applications [7], is a messaging or middle-ware communication layer, which also supports InfiniBand as one

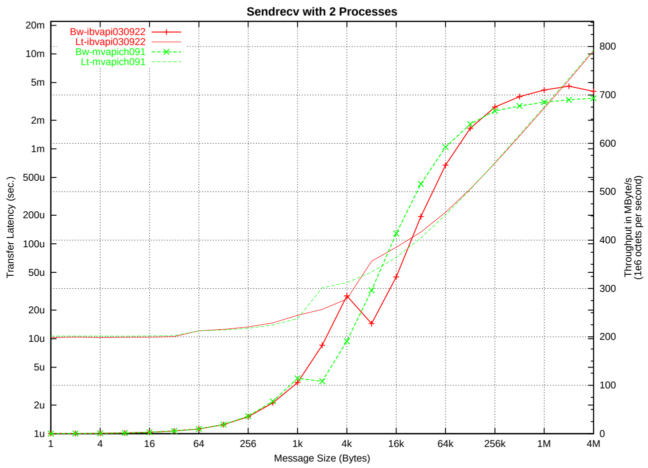

Figure 8: Pallas MPI Benchmark: SendReceive

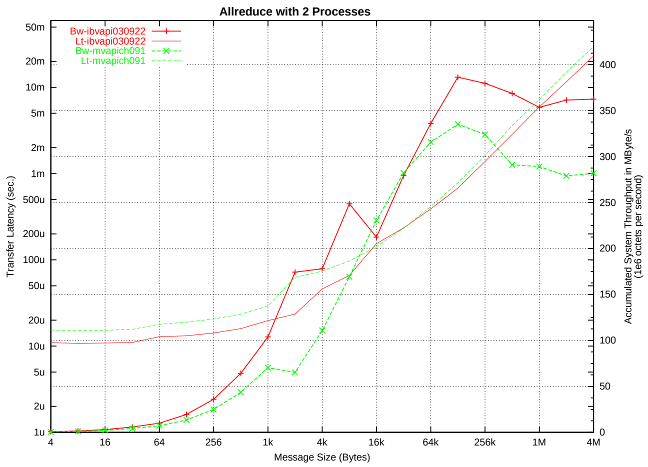

Figure 9: Pallas MPI Benchmark: Allreduce

of the its source or sink devices. As there is an enhanced version of MPICH1 that provides support for VMI as communication medium implemented as a channel device named *ch_vmi*, it provides another possibility to run MPI application on InfiniBand network technology.

2.8 Conclusions and Outlook

In this section, we presented an short overview of InfiniBand and MPICH2 and pointed out the advantages. Further, we described the current state and the concepts of our MPICH2-CH3 device implementation for InfiniBand. The dual approach of implementing the CH3 point-to-point semantic in terms of channel semantic (Send/Receive) and memory semantic (RDMA) operations has shown an positive impact on communication performance. Compared to the results of the raw VAPI performance, our preliminary implementation of the InfiniBand device has potentials for further improvements.

In the future, profiling mechanisms, experiences from testing several benchmarks and parallel applications, and further research into conceptual approaches will lead to improved implementations of the MPICH2-CH3 device for InfiniBand. As soon as MPICH2 provides full support for all MPI2 functionality our device will be adopted. Also, we work on the migration to the open source implementation of the Verbs interface that is currently being developed [8]. Furthermore, scalability issues will be taken into account as more nodes equipped with InfiniBand are available at our research lab. As one part of the project, we also focus on effective support for shared memory systems (SMPs) interconnected by InfiniBand. We will also take advantage of InfiniBand Multicast to implement collective operations on CH3 level.

Latest information about the project described within this paper can be obtained from our web page at
`http://www.tu-chemnitz.de/informatik/RA/cocgrid/Infiniband.`

References

[1] David Ashton, William Gropp, Ewing Lusk, Rob Ross, and Brian Toonen: *MPICH2 Design Document* Draft, October 14 2002.

[2] William Gropp, and Brian Toonen: *The CH3 Design for a Simple Implementation of ADI-3 for MPICH with a TCP-based Implementation* September 5 2002.

[3] MPICH - A Portable Implementation of MPI: http://www-unix.mcs.anl.gov/mpi/mpich/

[4] William Gropp, and Ewing Lusk: *A Test Implementation of the MPI Draft Message-Passing Standard* December 1992.

[5] MVAPICH: MPI for InfiniBand on VAPI Layer, Ohio State University: http://nowlab.cis.ohio-state.edu/projects/mpi-iba/

[6] InfiniBand Trade Association: *InfiniBand Architecture Specification.* Release 1.1, November 6 2002. See also: http://www.infinibandta.org.

[7] Virtual Machine Interface 2.0: http://vmi.ncsa.uiuc.edu/

[8] Linux InfiniBand Sourceforge Project *The project is focused on promoting, enabling and delivering the software components needed to support an InfiniBand fabric for the Linux operating system*: http://infiniband.sourceforge.net.

[9] Donald Becker, and Phil Merkey *Beowulf History*: http://www.beowulf.org/beowulf/history.html.

[10] René Grabner, Frank Mietke: *MPICH2-Device for InfiniBand*, Diploma Thesis, Chemnitz University of Technology, July 22 2003.

[11] MVICH: MPI for Virtual Interface Architecture, Berkeley Lab: http://www.nersc.gov/research/FTG/mvich/.

[12] H. Tezuka and F. O'Carroll and A. Hori and Y. Ishikawa: *Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication*, 12th Int. Parallel Processing Symposium, Orlando, FL, March 1998.

[13] J.Liu, J. Wu, S.P. Kini, P. Wyckoff, D.K. Panda: *High Performance RDMA-Based MPI Implementation over InfiniBand*, In the Proceedings of 17th Annual ACM International Conference on Supercomputing. San Francisco Bay Area. June, 2003.

[14] R. Grabner, F. Mietke, W. Rehm: *An MPICH2 Channel Device Implementation over VAPI on InfiniBand*, submitted to the International Parallel and Distributed Processing Symposium (IPDPS '04).

[15] Dolphin Interconnect Solutions: *Benchmarks* http://www.dolphinics.com/products/benchmarks.html

[16] Mellanox Technologies: *Infiniband HPC Clustering Solutions*, www.mellanox.com/technology/shared/ Mellanox_HPCC_June_03.pdf

[17] Intel, Microsoft, Compaq: *VI Architecture Specification*, http://www.viarch.org

[18] InfiniBand Trade Association: *Infiniband Architecture Specification*, http://www.infinibandta.org/specs

[19] National Energy Research Scientific Computing Center: *M-VIA: A High Performance Modular VIA for Linux*, http://www.nersc.gov/research/FTG/via