An MPICH2 Channel Device Implementation over VAPI on InfiniBand

René Grabner, Frank Mietke, Wolfgang Rehm
{regra,mief,rehm}@cs.tu-chemnitz.de

Chemnitz University of Technology
Faculty of Computer Science
Straße der Nationen 62, 09111 Chemnitz, Germany

Abstract—**MPICH2, the successor of one of the most popular open source message passing implementations, aims to fully support the MPI-2 standard. Due to a complete redesign, MPICH2 is also cleaner, more flexible, and faster. The InfiniBand network technology is 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. 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 paper we present our experience that has been gained during the implementation of our MPICH2 Device for InfiniBand. Further, a performance overview is given, as well as ideas for future developments. The device is implemented in terms of the Channel Interface (CH3) and uses both the channel semantics (Send/Receive) and memory semantics (RDMA) provided by Mellanox' Verbs implementation VAPI. With this combined approach a significant performance gain can be achieved. The design decisions discussed may also be of interest beyond the scope of this paper.

Keywords—**Message Passing, MPI2, MPICH2, InfiniBand, RDMA, Cluster Computing**

I. Introduction

High performance computing on low cost commodity-based cluster systems has risen popularity over the last ten years. All began in late 1993 with the first ideas to build such a cost-effective system, which originated the legendary Beowulf-Project [19]. Already its first prototype cluster has revealed a new, so far neglected challenge, namely the performance of the interconnect network. Over the years Moore's law has always been fulfilled as the achievable computing power of modern processors has reached new dimensions. This promoted the elevated need for new technologies, which provide low latencies and high bandwidths as required for a System Area Network (SAN). Many of such mostly proprietary networks have been developed so far, for instance Myrinet, Scalable Coherent Interface (SCI), Quadrics QsNet. Such a variety was almost foreseeable, and additionally several shared memory architectures emerged. Consequentially, there were early efforts to establish a programming standard for distributed systems. At the Supercomputing Conference in November 1993 the draft for the Message Passing Interface (MPI) standard was presented. In parallel William Gropp and Ewing Lusk have developed an intermediate implementation of the forthcoming standard as a kind of proof of concept. This has become the first version of MPICH [7]. Later on a layered design was introduced aiming to easily adopt new network interfaces and different architectures. At least, there is support for each SAN mentioned above implemented by an MPICH software device. Now, MPICH2 is the most recent developement of the MPICH Team at Argonne National Laboratory (ANL) [5] [1] [2].

The InfiniBand Architecture is the latest technology for interconnecting computational nodes and I/O nodes to form a System Area Network. It is an open industry standard [14] that has been developed by several leading IT companies. A successful distribution and practical relevance of this promising architecture depends on the availability of MPI support, which is the most often used programming interface by developers of parallel applications. Due to the early availability of InfiniBand hardware and driver software provided by Mellanox Technologies, Inc. [15], it was possible to start our research activities into InfiniBand enabled MPICH2 implementations.

The rest of the paper is organized as follows: In section II, we present an overview of MPICH2 and InfiniBand. Within section III, we describe challenges and design experiences of our InfiniBand device implementation. First performance results are presented in section IV. In section V related work is presented. In the last section we conclude and give an outlook for future work.

The project is supported by MEGWARE Computer GmbH, Chemnitz, Germany, www.megware.de. 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.

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

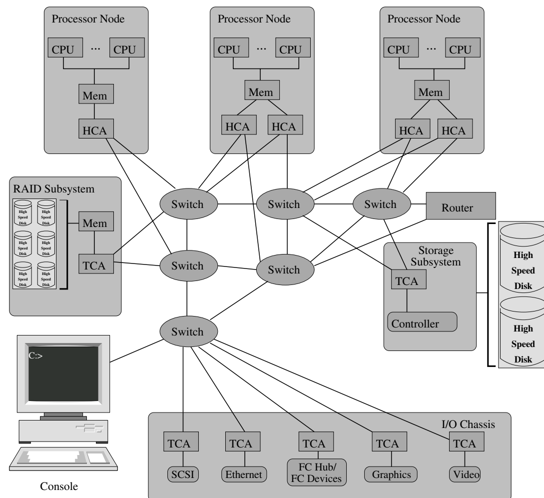

Fig. 1
IBA - System Area Network

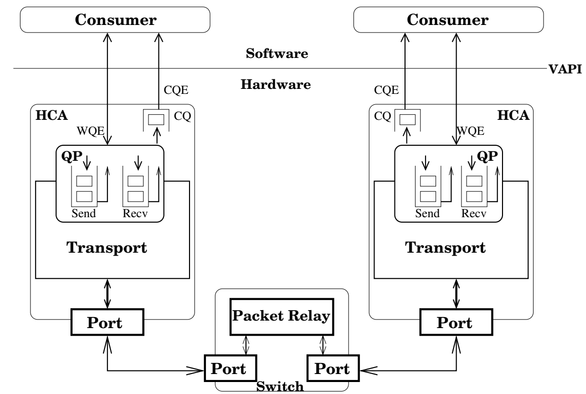

Fig. 2
IBA Communication Stack

II. Fundamentals

A. 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.

A.1 Architecture

InfiniBand is a point-to-point, switched I/O fabric architecture. 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.

A.2 Communication Stack

InfiniBand has been developed with the Virtual Interface Architecture (VIA) [16] 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) queues 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 at least one CQ - Completion Queue 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 holds an instruction, which will be executed by the channel adapter's processing unit. These instructions are divided into two classes, the memory semantic (RDMA, Atomic) and the channel semantic (Send/Recv) operations.

A.3 Verbs API

The InfiniBand Architecture (IBA) [14] 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 a completion status. The intention 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.

B. MPICH2

The second version of MPICH is a complete redesign [3] [8]. It is being developed by the MPICH Team at Argonne National Laboratory, 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 [6], 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 [4]. Systems targeted by this redesign are 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.

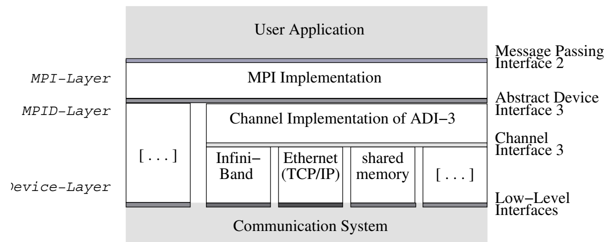

Fig. 3
Layered architecture of MPICH2

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.

III. Prototype

As shown in section II-B, 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.

Not only the MPICH2 implementation itself uses a layered approach, but also the InfiniBand device that is shipped with the MPICH2 package 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 of the InfiniBand device 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 to InfiniBand.

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 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 because it used a discontinued IBM Verbs implementation for InfiniBand at this time. The concepts in the following subsections A-E have been taken over with a few changes.

A. 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 (that could be an MPI envelope, request ids,...) and the data. The packet header is delivered to the destination immediately if possible, see also section III-C. Dependent on the communication protocol the data is or is not transported at once with the packet header.

In the case of eager protocol 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 can be implemented with channel or memory semantics of InfiniBand architecture. For the first prototype we decided to implement channel semantics only.

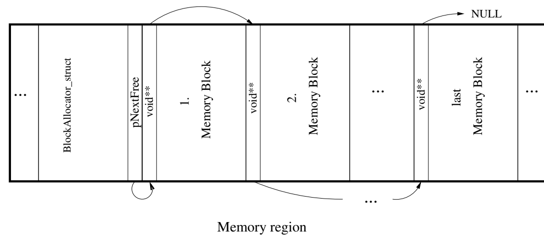

Fig. 4
SEGMENTATION OF A PRE-REGISTERED MEMORY BLOCK

B. 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. After a successful send/receive operation the buffer can be unregistered. Because of the behaviour of the un-/registration process, a kernel involvement is necessary, which 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. The illustration 4 shows such a memory region.

Currently, such a memory region exists for each connection to another process. This is not that scalable and should be improved in the future. In the finalization stage the registered memory regions are deregistered and freed. Unfortunately this concept has a limitation, but this is discussed in chapter III-H.

C. Flow Control

In order to prevent that the remote node's 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 prototype device the memory usage per process can be calculated as follows:
mem(per process) = packet_size(2*flow_control-credit)*
As mentioned in the subsection before this approach is not as scalable. Packet_size has been empirically determined and is 8KB, flow_control-credit has also been empirically determined and is 32. The doubling of flow_control-credit comes from the fact that each process has 32 pre-posted receives for incoming traffic from the other end of the connection and 32 flow_control-credits for send operations. An improvement to the static credit based flow control scheme for better scalability is discussed in [10].

D. 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 coming 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.

E. Progress Engine

The progress engine is responsible for dispatching a 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 an `unex_finished_list`. An element is added to that list if the data had been received even before the read request in the CH3 layer was commited. Afterwards, `ibvapi_wait()` 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.

F. 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 has been 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.

G. 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 [22] 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

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

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 reimplemented to fit better into the InfiniBand Architecture.

H. 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) that have been implemented in our device were chosen for this description.

H.1 RDMA

InfiniBand distinguishes channel semantic- and memory semantic operations, see section II-A. In the prototype device the channel semantic operations have been 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) [14]. The un-/registration issues are discussed in section III-H.2.

The utilization of one sided communication (RDMA) for data is of course not new [24] [26] [25], but it had never been implemented for MPICH2 when we started our work. We discuss here our design decisions and implementation to use RDMA for rendezvous protocol.

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 taken into account. MPICH2 distinguishes the eager and the rendezvous protocol. In the first case the message is directly sent with the header. In the latter case, before a message is sent, a hand-shake protocol is performed. The rendezvous protocol requires that the receiver is ready to get the data. In this case the extra copies can be avoided, when an RDMA mechanism is 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

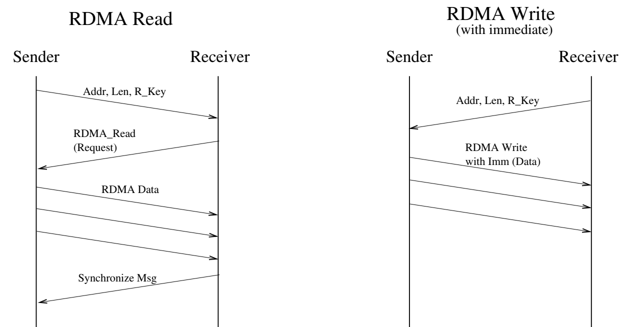

Fig. 5
RDMA-Read vs. RDMA-Write

Two possibilities exist to realize RDMA, these are RDMA-Read and RDMA-Write. The comparison between both is illustrated in figure 5. Since RDMA shall be implemented for rendezvous protocol it has to be integrated in the hand-shake. In CH3-Layer this is only possible after the receiving of a RNDV_SEND message, because then the receiver is ready for receiving data. At this point the memory of the receive buffer has to be registered. In case of RDMA-Read the posting of the read request follows and in case of RDMA-Write the posting of address, length and r_key follows. These should be the starting points in the comparison.

The figure 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 packet sent the immediate data is transfered, which leads to an CQE on the receiver site. After this, both sides know that the application memory is ready for reuse. So RDMA-Write with immediate data is more suitable for our device than RDMA-Read.

RDMA-Write Implementation

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

- rendezvous protocol is used

0-7695-2132-0/04/$17.00 (C) 2004 IEEE

- the data which are transported reside in 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 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. This value is set in the init stage to 32000 bytes. For the receiver follows from the main conditions 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 (equivalent to iov_n) of the receive request was 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 has 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, which in turn calls the function ibvapi_post_readv(). In this function the conditions for the receiver site are checked. When the conditions are fulfilled the receiver site registers its receive buffer. After a successful registration the important values for the sender are stored in a data structure that exists for each connection (stored in the VC - virtual connection structure of MPICH2). These values are the start address, the length and the r_key. They are sent with ibvapi_write_rdma_info() which posts a normal send request. The values are also stored on the sender site.

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 unregistered. Now the send request can be completed by the progress engine. The same happens on the receiver site when the last data packet has been arrived. The completion is signalled on both sites by a CQE.

H.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 unregister again to free claimed resources. But, this would not yield to an improved implementation as we will show later. Both, registration and unregistration are expensive operations. Their costs are functions of the memory region's size to un-/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 that one allocated user buffer will 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 unregister that buffer. Intelligence is necessary to not run into a shortness of resources. Such an intelligent mechanism has been proposed in [23] and is used in our InfiniBand CH3 device. The concept of this mechanism is shortly explained in this subsection.

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

The registration function ibvapi_mem_REGISTER(), implemented by DME, is at first trying to find a memory entry record within the hash-table that corresponds to the passed buffer address and buffer size. The hash-value which was calculated using the base address of the page in which the buffer start address resides, is used as an index to directly access the hash-table. If there is a memory entry record a comparison of its memory region information and the actual one informs about their conformance. The actual base address of the first page must be equal to the one of the memory record, the actual number of pages might be less than or equal to the one stored within the memory record. That is, only a part or the whole previously used buffer is actually being reused. 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 to a minimum. 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, the algorithm tries to evict an unused memory region and actually unregisters it on the HCA. The candidate for the 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 on the HCA. 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 implementation of the DME has been partly derived from the MPICH VIA device implementation MVICH [21].

Both, RDMA and DME, improvements are explained more in detail in [20].

IV. Performance Results

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

- Dual Intel Xeon 2,66 GHz
- SuperMicro motherboard X5DPE-G2
- 1024 MB DDR Memory, PC2100, ECC registered
- 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 ran Redhat Linux 9 with kernel *2.4.23-pre3* from kernel.org. The Mellanox host channel adapters were using firmware version *fw-23108-rel-3_00_0000*, the software development and drivers kit version was *thca-x86-3.0.1*. All compilations were done using the GNU C compiler *gcc-3.2.2*.

The Pallas MPI Benchmark Suites *PingPong* test performs the classical pattern to measure startup and throughput of a single message that is sent between two processes. The code is a loop over `MPI_Recv()` followed by a `MPI_Send()`. The *SendRecv* test is based on `MPI_Sendrecv()`. A chain pattern is used and each process sends messages to its right and receives messages from its left neighbour process.

A. Achievements of RDMA and DME

The improvements that were discussed within the subsections III-H.1 and III-H.2 have been implemented within our MPICH2 Device. In figure 6 we prove their impact on communication performance using Pallas MPI Benchmark. 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. The reason is the domination of registration and unregistration of the user buffers, that is performed before and after each RDMA transfer. With an enabled DME this issue is solved. The expensive registration of the user buffer takes place just once – 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.

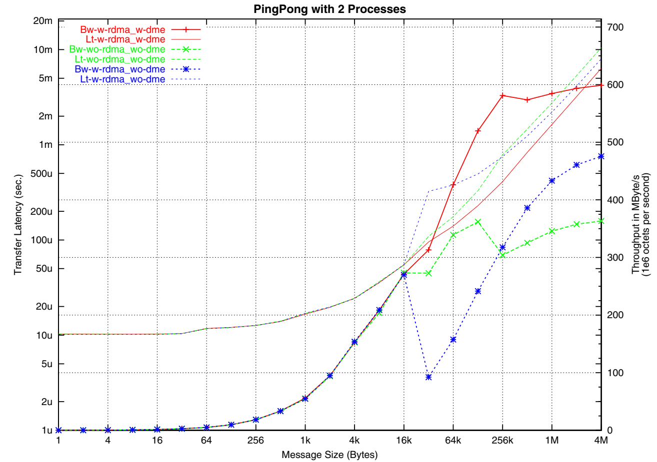

Fig. 6
PingPong with and without RDMA and DME

B. Comparisons

Figure 7 exemplary shows results of Pallas MPI Benchmark (PMB) test cases. In the PMB Barrier test our IB-VAPI device is slower than OSU MVAPICH2 [11]. This is due to the fact that MVAPICH2 uses RDMA also for small message sizes. Following table shows the average timings in microseconds for up to four nodes. Further research will be done in the future to find answers on the scalability when more nodes are involved.

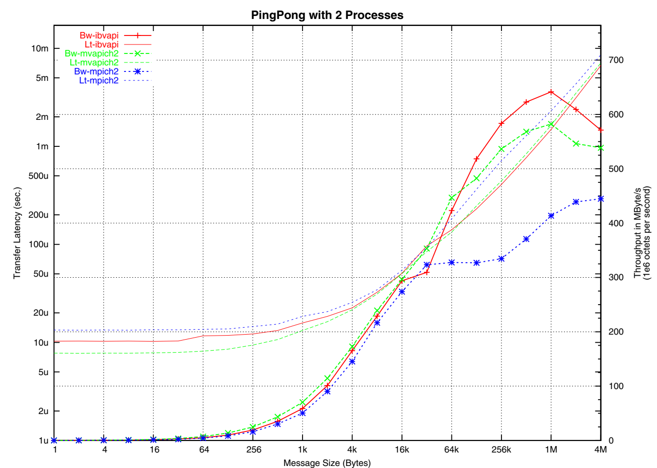

Fig. 7
Comparison: PingPong

# nodes	IBVAPI	MVAPICH2	MPICH2 (ANL)
2	9.63	8.02	13.57
4	18.96	16.47	26.50

V. Related Work

There are a few related research projects that are focusing on MPI in combination with InfiniBand. A very popular and one of the first projects brought up the MVAPICH [9] implementation. It is being developed at the Network-Based Computing Laboratory of the Department of Computer and Information Science at Ohio State University (OSU). The implementation is based on MVICH 1.0 [21] and MPICH1, respectively, and also utilizes the VAPI implementation. In contrast to our device they have implemented MVAPICH using MPICH1-ADI2.

Meanwhile there is also an MPICH2 implementation for InfiniBand released from OSU which uses RDMA [11] [12] [13]. Recent releases of the MPICH2 package also delivers an InfiniBand device with VAPI support.

VMI 2.0, being developed at the National Center for Supercomputing Applications [17], is a messaging respectively middle-ware communication layer, which also supports InfiniBand as one of the its source or sink devices.

VI. Conclusions and Outlook

In this paper, 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 [18]. 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 www.tu-chemnitz.de/informatik/RA/cocgrid/Infiniband.

References

[1] W. Gropp and E. Lusk and N. Doss and A. Skjellum: *A high-performance, portable implementation of the MPI message passing interface standard* Parallel Computing, volume 22, number 6, pages 789–828, Sep. 1996

[2] William D. Gropp and Ewing Lusk: *User's Guide for* `mpich`*, a Portable Implementation of MPI ANL-96/6 Mathematics and Computer Science Division, Argonne National Laboratory 1996*

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

[4] 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.

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

[6] William Gropp et al: *MPICH Abstract Device Interface Version 3.4 Reference Manual* Draft of September 3, 2003

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

[8] Second Version of MPICH: `http://www-unix.mcs.anl.gov/mpi/mpich2/`

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

[10] Jiuxing Liu and Dhabaleswar K. Panda: *Implementing Efficient and Scalable Flow Control Schemes in MPI over InfiniBand* In Int'l Parallel and Distributed Processing Symposium (IPDPS 04)

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

[12] W. Jiang, J. Liu, H. Jin, D.K. Panda, W. Gropp and R. Thakur: *High Performance MPI-2 One-Sided Communication over InfiniBand* In 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 04)

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

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

[15] Mellanox Technologies Inc.: `http://www.mellanox.com`

[16] Virtual Interface Architecture: `http://www.vidf.org`

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

[18] 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`.

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

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

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

[22] Pallas MPI Benchmark Suite: `http://www.pallas.com/e/products/pmb/index.htm`

[23] 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.*

[24] M. Banikazemi, R.K. Govindaraju, R. Blackmore and D.K. Panda: *MPI-LAPI: An Efficient Implementation of MPI for IBM RS/6000 SP Systems*

[25] M. Lauria and A. Chien: *MPI-FM: High Performance MPI on Workstation Clusters* Journal of Parallel and Distributed Computing, pages 4-18, Jan 1997

[26] C. Chang, G. Czajkowski, C. Hawblitzel and T.V. Eicken: *Low Latency Communication on the IBM RISC System/6000 SP* Supercomputing 1996