

SHIBA – Shared Memory Support for InfiniBandTM MPICH2 Device *

Frank Mietke, Marco Steiger, Torsten Mehlan, Torsten Höfler, Wolfgang Rehm
{mief,marst,tome,htor,rehm}@informatik.tu-chemnitz.de

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

Abstract

With the Top500 list from June 2004, cluster systems exceeded not only the 50% threshold in number of systems the first time, but also in total performance. Due to the good price-performance ratio of cluster systems, these rates will further increase in the next lists. Multiprocessor nodes – typically 2 or 4 way systems – can improve that price-performance ratio in clusters.

This paper presents our MPICH2 device called SHIBA (Shared-Memory + InfiniBand) which is the logical successor of our InfiniBand-only device to follow the above mentioned trend. We aim to provide a fast connection for intra-node communication and therewith to disburden the PCI connection to the InfiniBand Host Channel Adapter (HCA) for inter-node communication.

1 Introduction

The development of CPUs has dramatically changed and the Gigahertz-race for faster processors is over. More promising technologies like dual- and multi-core CPUs are in range. So, the multi-processor approach will be as normal in consumer PCs as it is in highend cluster nodes today. Due to the increasing number of clusters comprising commercial off the shelf components and the appearance of multi-core CPUs, our InfiniBand only device [9] for MPICH2 should be extended to support shared memory for intra-node communication.

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 [13] that has been developed by several leading IT companies. The current version of the standard specifies full-duplex bandwidths from 2.5Gbit/s up to 120Gbit/s.

This paper presents our SHIBA [19] device for MPICH2 [11, 12]. It uses shared memory for intra-node communication and InfiniBand for inter-node communication. We aim to disburden the PCI connection of the InfiniBand HCA and thus to reduce the communication performance loss in multiprocessor nodes sharing the same HCA. Figure 1 depicts our SHIBA device inside the MPICH2 structure.

The rest of the paper is organized as follows. In section 2 we describe the basics of our SHIBA device. Some improvements for faster reading of messages from the receive buffer are discussed in section 3. Performance results from the NAS application benchmark are presented in section 4. In section 5 related work is presented and finally 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.

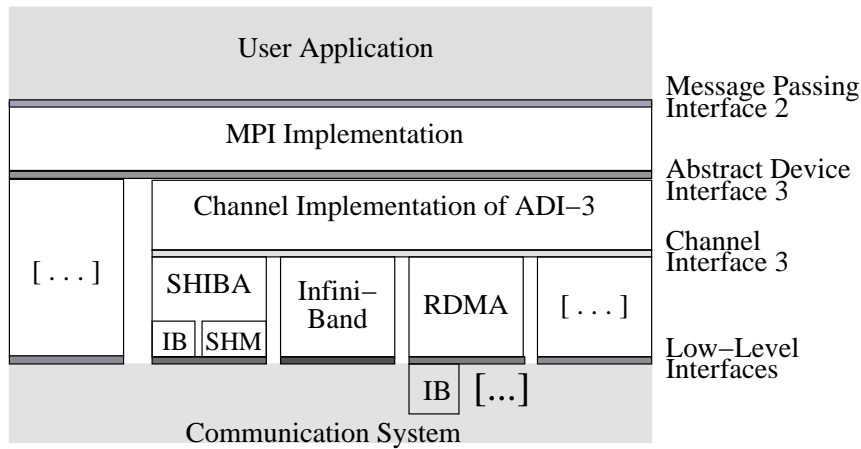


Figure 1: MPICH2 structure

2 Basics

As starting point we used our MPICH2-CH3 device for InfiniBand that was presented in [10]. That work bases on the Mellanox InfiniBand Verbs API, utilizes RDMA for rendezvous protocol and normal Send/Recv semantics for short eager messages. MPICH2, the successor of one of the most popular open source message passing implementations, aims to fully support the MPI-2 standard [5, 6]. Due to a complete redesign, MPICH2 is also cleaner, more flexible, and faster. Some ideas and concepts were taken from the Socket Shared Memory (SSM) device of MPICH2 to accelerate the development. SysV [1] and POSIX [2] shared memory semantics were integrated into the SHIBA device.

To get the Shared Memory support working the following steps have to be done:

- recognize the processes on the same host
- establish shared memory connections between the process pairs
- realize communication over Shared Memory (writing, reading, progress)

2.1 Process-to-Host

The first step before establishing a shared memory connection is the creation of a message queue and the recognition of the processes that are working on the same host. The message queue can be implemented either as a single message queue (SysV) or as a message queue for each process (POSIX) and can be used to pass information like address, size and the shared memory id to the other processes.

To detect processes on the same host you can imagine several possibilities. The most obvious solution would be the usage of the Key-Value-Space (KVS), a distributed "database" in the multi-purpose daemon (mpd). The mpd is a process management daemon and is part of the default job management of MPICH2. At least one mpd runs on each host, all of them are connected by a ring to exchange management information. But this method is too oversized for the detection because it is only necessary on the appropriate host.

We utilize another shared memory segment in our implementation, the advantage is that the shared memory segment is only available on each host. Due to the shared memory semantics only the first calling process creates the segment. The other processes get the identifier of the created shared memory. Its size is a multiple of the number of processes depending on the information that should be provided. As depicted in figure 2 every process on the host writes its rank and further information at the proper position in the shared memory block. In the SHIBA implementation only the process identifier is provided as further information, because the

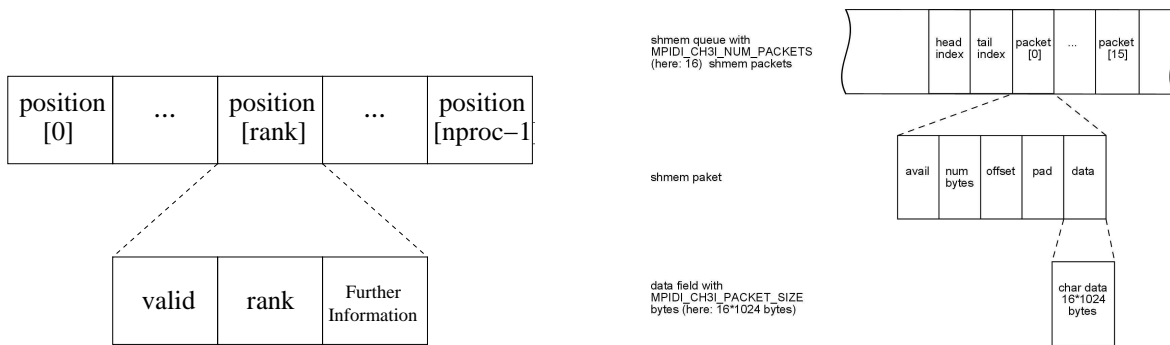


Figure 2: Structure of the process-to-host shared memory and memory scheme of the shmем queue

process id is sufficient to get the message queue id. The advantage is that this mechanism is as fast as possible and the memory consumed in this stage can be freed immediately after the init phase. Another advantage of this mechanism is that it is independent of the underlying network (it doesn't use hostname like in other implementations) and can therefore be applied to every SMP cluster.

2.2 Post-Connect

After we know which processes are on the same host, the next step is to establish shared memory connections between those processes. To manage the communication between two processes, MPICH2 uses the Virtual Connection structure. The Virtual Connection manages only the one-way connection from one process to another but not vice versa. The assignment whether a process pair is connected through shared memory or InfiniBand is made in the init stage.

There are two possible time ranges to make a connection. The first one is in the init stage but has the disadvantage that much memory is wasted if the processes do not communicate during the program run. We preferred the post-connect mechanism that is the second possibility. In the post-connect mechanism the one-way connection is established when the first communication between the processes occurs. This mechanism creates the shared memory block for writing data to the other process in the pair. The shared memory block consists of a head index, tail index and several packets (see figure 2). The head index points to the first unread packet and the tail index points to the first free available packet for sending, whereby the head index is only changed by the receiver and the tail index by the sender.

To complete the connection establishment the other process has to attach to the created shared memory. Therefore the message queue from step one is needed to provide the other process with appropriate information, namely the starting address, the size and the id of the shared memory as well as rank and process id.

2.3 Communication

After creating and attaching the shared memory the one-way connection is established. The shared memory block is seen as the writing shmем queue for the one process and the reading shmем queue for the other. Due to the relation between sender and receiver, flow control is implicitly given. If the tail index points to the same packet as the head index, no further packets can be sent and the appropriate sending process is delayed.

After communication has occurred, the next step is the progress engine. The progress engine is responsible for dispatching a send/receive request and for triggering outstanding send requests in the send queue or reloading a receive request. InfiniBand and shared memory communication should be handled in a fair manner. Therefore we decided to take the Adaptive-Progress from the MPICH2 socket-shared-memory device and to implement a second mechanism we call Toggle-Progress. There are two progress variants a blocking and a nonblocking. Only the blocking one is interesting and is considered in this paper because the nonblocking version checks

each connection once and returns. Shared memory, InfiniBand as well as the message queue have to be checked. A comparison between both progress mechanisms is given in section 4.

2.3.1 Adaptive-Progress

This progress mechanism is a weighted progress type. After entering the progress function, shared memory and InfiniBand are queried once. If there is no work for the progress engine the query is repeated in the same manner. After a certain number of repetitions the message queues are also checked for a shared memory post-connect request.

Has a packet arrived to be processed in the progress engine and the appropriate MPI-Request is not completed yet, then the Adaptive-Progress recalculates the new number of repetitions for each communication channel. If only one packet has arrived, the channel which received this packet gets a higher number of repetitions in the next loop of the progress engine than the other channel. And, if packets have arrived on both channels, each gets 50% of the time for the next loop. This procedure is repeated until the MPI-Request has been completed.

2.3.2 Toggle-Progress

This progress approach is unweighted and uses two flags, the active-progress-flag and the take-progress-flag to make a decision. Their values can either be set to `SHM_ACTIVE` and `IBVAPI_ACTIVE` or vice versa. The active-progress-flag specifies the transfer type that has done some work in the last progress loop. And the take-progress-flag specifies the transfer type which is taken in the next loop. Thereby the message-queue-progress belongs to the shared memory transfer. Thus, the progress mechanism toggles the sequence of checking either InfiniBand at first and then shared memory or vice versa.

2.3.3 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 (e.g. MPI envelope and request ids) and the data. Dependent on the communication protocol the data is or is not transported at once with the packet header.

Due to the different communication characteristics of shared memory and InfiniBand, we have to handle the switch from eager to rendezvous protocol in an adequate manner. Therefore we differentiate the protocol switching point between InfiniBand and shared memory. Because shared memory is much faster the switching point is set to 256 Kilobyte to hold the performance while the one for InfiniBand remains at 8 Kilobyte.

3 Improved Reading Process

The SHIBA device uses another approach for the reading process as it is in the SSM device of MPICH2. The following describes the performance bottleneck that we see in the SSM device.

The sent data is read and handled in the `shm_read_progress`. This progress part reads only the shm packet on the head of the reading shm queues in the `shm_reading_list` (This list stores all post-connected Virtual Connections). If a CH3 (Channel Interface 3) header is in a head shm packet, then this packet is handled with the appropriate packet handle-function. Whereas this function calls the `MPIDI_CH3_iRead()` function to get the remaining data from the head shm packet. If the MPI-Request cannot be completed with this shm packet, then the remaining shm packets of the shm queue are not read immediately afterwards. That is, after the progress engine has read one shm packet, the next entry in the `shm_reading_list` is processed by the reading progress. The further shm data packets are read only within the next progress sequence. So, the reading progress starts again with the first entry in the `shm_reading_list`. If the sending process fills the shm queue completely then the receiving process delays the writing of more data by reading with this "slow" mechanism. Therefore a one-shm-packet-window can arise, i.e. the reader reads and releases one shm packet and the writer fills this packet again and waits for the next available

shmem packet.

Three possible solutions are presented in the following subsections.

3.1 Contiguous-Read

The Contiguous-Read mechanism tries to read all remaining shmem packets of an MPI-Request in the reading shmem queue of the actual virtual connection. But this is done if there is only real data without headers in the remaining shmem packets. Whether there is more data to read is known after handling the header. Therefore the Contiguous-Read of the remaining data is done when the virtual connection is checked the next time. This read procedure is done until the reading shmem queue becomes empty or the MPI-Request is completed.

The advantage of this method is that `shm_read_progress` can read all data of a MPI-Request nearly immediately and thus, the completion of the receive request can be faster. The disadvantage is that the remaining data is only read when the Virtual Connection is checked the next time.

3.2 Progress-Post-Read

To further improve the Contiguous-Read mechanism, the reading of the remaining data is moved up. Now the contiguous read is directly performed after handling the header packet instead of waiting for the next check of the Virtual Connection. The Progress-Post-Read is only activated if the message size of the MPI-Request is larger than the shared memory queue to not further hinder the writing process. In our case, the size of the shared memory queue is set to 256KB. If the MPI-Request cannot be fully completed, the Contiguous-Read mechanism finishes its work the next time the Virtual Connection is checked.

3.3 Immediate-Read

This mechanism is similar to Progress-Post-Read but is executed during handling the header packet of the MPI-Request instead afterwards.

It is application dependent whether these improvements make sense. If not, the original SSM device mechanism can be used as well.

4 Benchmarks

We have conducted some benchmarks with the Pallas-Suite [8] as well as the NAS benchmark [3]. For these testings we used MVAPICH2, the MPICH2 InfiniBand device from Argonne National Laboratories as well as our two InfiniBand devices for MPICH2. As test environments we have chosen the Mozart Xeon cluster from Universität Stuttgart and an Opteron cluster from Universität Bayreuth to see the differences between a shared bus multiprocessor platform (Xeon) and NUMA based multiprocessor platform (Opteron). We used 32 nodes (64 processors) for the NAS benchmarks and two or three nodes for the Pallas benchmark.

4.1 Cluster-Configuration

4.1.1 Mozart Xeon Cluster

This cluster comprises 64 nodes with dual Intel Xeon 3,066GHz FSB533 and 4GB RAM on a Supermicro X5DPA-GG mainboard. All nodes are connected with the InfiniBand switching fabric through a PCI-X 4X Host Channel Adapters. The operating system is RedHat Linux 9.

4.1.2 Opteron Cluster

This cluster comprises 32 nodes with Dual AMD Opteron 246 and 3,5GB RAM on a TYAN Thunder K8S Pro S2882 mainboard. All nodes are connected with the InfiniBand switching fabric through a PCI-X 4X Host Channel Adapters. The operating System is SuSe Enterprise Server 8.1.

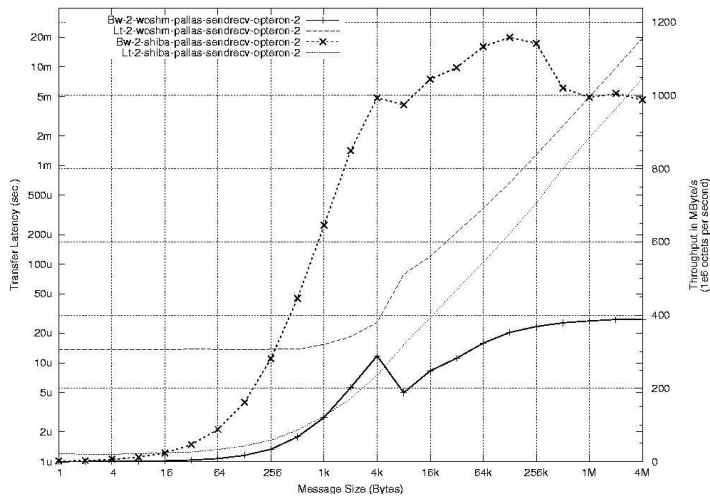


Figure 3: MPI_Sendrecv comparison with shared memory support switched on/off (Opteron)

4.2 Pallas Benchmarks

The following pictures show the performance results gained with Pallas Benchmark Suite. A comparison of a SendRecv operation between two nodes, with InfiniBand and shared memory respectively for the Opteron architecture, is given in diagram 3. The protocol switches from eager to rendezvous for both communication methods at the same packet size of 8 Kilobyte. One can see in diagram 5 that a better performance is achieved with a different handling of the protocol change for shared memory. Diagram 4 shows the same information as diagram 3 but for the Xeon architecture. What you can see is that the Xeon behaves worse than the Opteron. The reason for the better performance of the Opteron architecture lies in the way how the processor communicates with the memory. In a multiprocessor Opteron system each processor has its own memory controller on die. In a Xeon SMP system all processors are connected through one bus to one memory controller which is on the mainboard. With a packet size of 512 Kilobyte the shared memory communication has the same performance (400MByte/s) than InfiniBand for Xeon while the Opteron sustains 1GByte/s.

The bad performance of the memcpy operation is clearly visible in diagram 3 and 4. It is only a fourth of the theoretical memory bandwidth. Despite this the bandwidth and latency numbers of the shared memory communication are much better than the one of PCI-X InfiniBand except for packets bigger than 512 Kilobyte on the Xeon architecture. This situation changes a little bit with the usage of PCI-Express or Hypertransport InfiniBand adapters because they achieve twice as much of the bandwidth of PCI-X adapters and have better latency numbers. In this case it would be better to switch back to InfiniBand for intra-node communication on Xeons if the packets are bigger than 256 Kilobytes.

A comparison of the different progress engines is given in diagram 6. For bigger packet sizes the toggle progress shows a better performance than the adaptive progress. This is clear due to the overhead of the calculations within the adaptive progress engine in combination with the Pallas approach of stressing the network.

4.3 NAS Benchmark

Figure 7 shows the results of the LU benchmark (class B) comparing the four MPICH2 InfiniBand implementations mentioned above on both clusters. As you can see on the diagrams (both diagrams have a different scaling), the Opteron cluster achieves more than 50% higher performance rates than the Xeon cluster. The benefits of the NUMA architecture characterizing the multiprocessor Opteron systems can be seen as one reason for the impact on the performance numbers. The single bus architecture of the Xeon SMPs leads to much more memory contention and therefore achieves worse results.

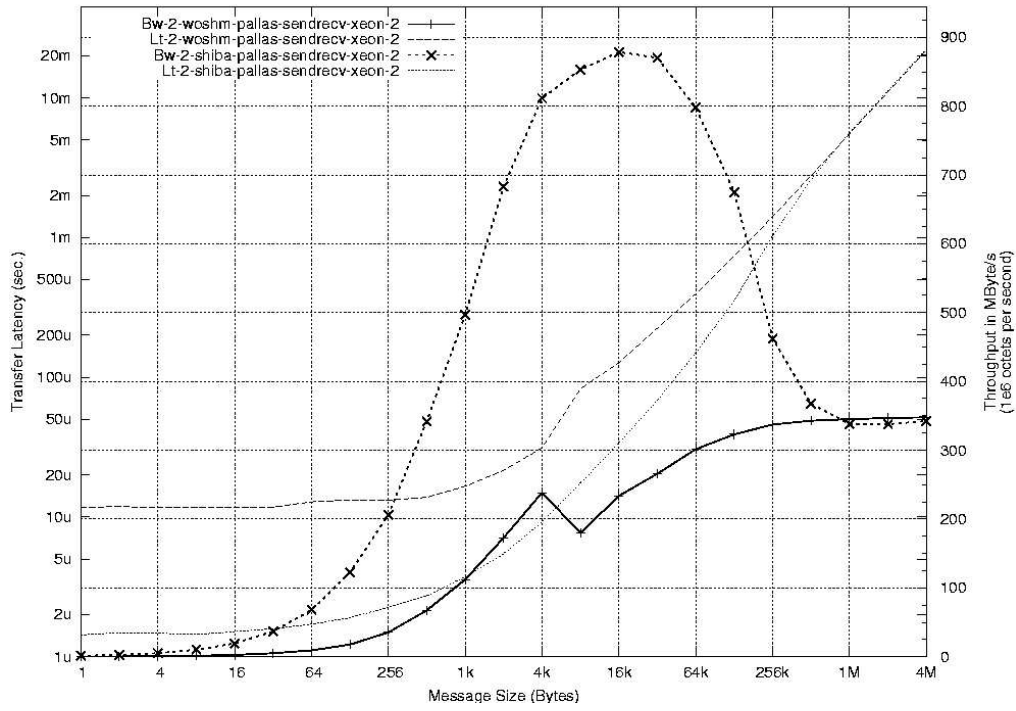


Figure 4: MPI_Sendrecv comparison with shared memory support switched on/off (Xeon)

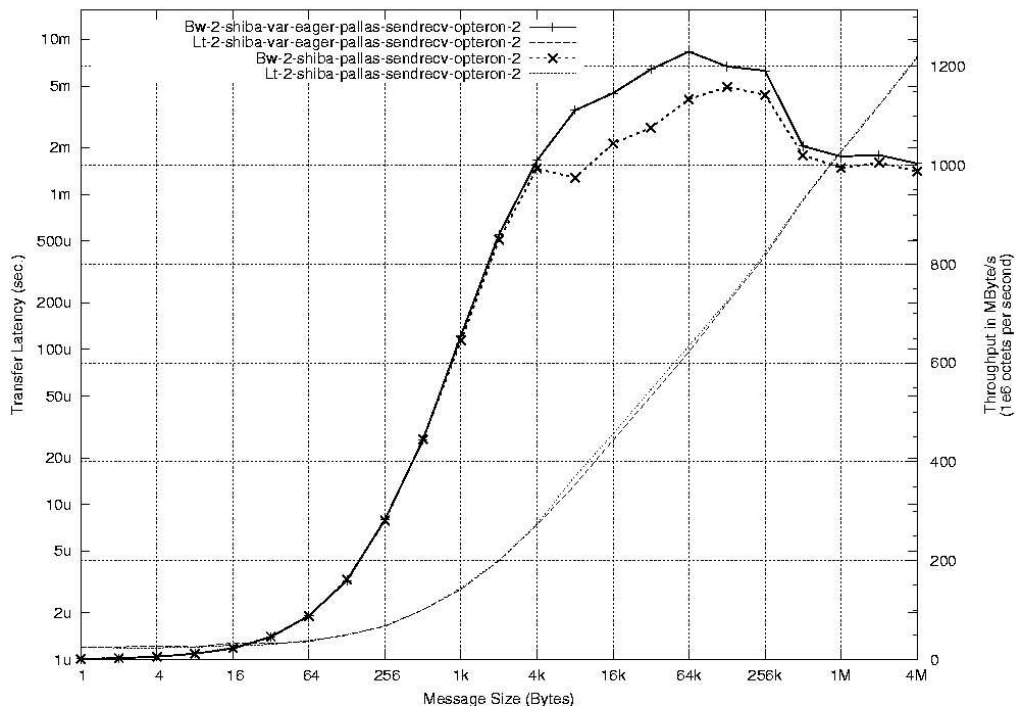


Figure 5: MPI_Sendrecv comparison with constant/variable Eager-Rndv-Switch (Opteron)

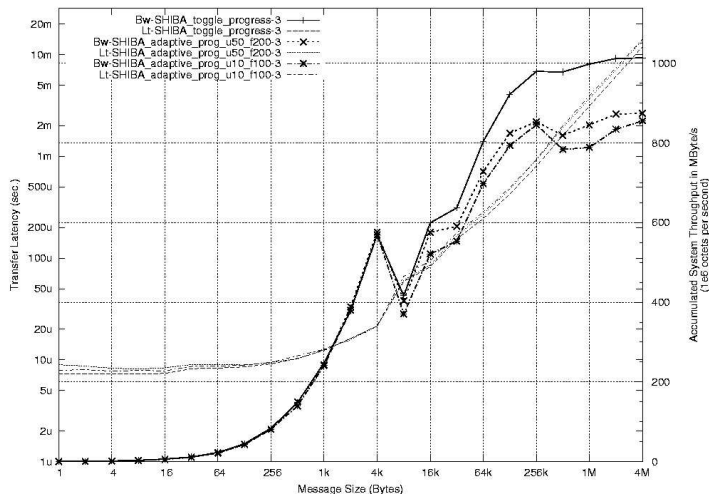


Figure 6: MPI_Bcast comparison with Toggle- and Adaptive-Progress (Opteron)

While there are almost no effects on the Xeon cluster using the shared memory approach, a little performance gain can be achieved on the Opteron architecture. This can be explained with the communication pattern of the LU benchmark and the Pallas results. The LU Benchmark mostly does send/rcv operations with many small messages and some bigger messages with packet sizes of more than 256 Kilobytes. As you can see in the Pallas results of the Xeon architecture, for packet sizes bigger than 512 Kilobytes the shared memory communication has nearly the same performance than Infi niBand while the Opteron is much better for big messages. A further detail is the different behaviour of the other MPI implementations on both hardware architectures. It shows that a deeper investigation is necessary to explore the reasons why the MPI implementations behave differently on several hardware architectures.

5 Related Work

There are a few related research projects that are focusing on MPI in combination with Infi niBand (and shared memory). A very popular and one of the first projects brought up the MVAPICH [17] implementation. It is currently under development 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 [15] and MPICH1, respectively, and also utilizes the VAPI implementation in combination with shared memory support. In contrast to our device they have implemented MVAPICH using MPICH1-ADI2.

There is also an MPICH2 implementation for Infi niBand released from OSU [16] which was implemented using the RDMA channel of MPICH2. Recent releases of the MPICH2 package also deliver an Infi niBand device with VAPI support. Currently, these MPICH2 devices do not support the combination of shared memory and Infi niBand. Another interesting approach for shared memory communication was proposed by the same research group using a special kernel module that achieves outstanding performance results [4, 14].

VMI 2.0 [18], being developed at the National Center for Supercomputing Applications, is a messaging respectively middle-ware communication layer, which also supports Infi niBand as one of its source or sink devices. Currently, there is no shared memory device available but it is conceptual possible that Infi niBand and shared memory can work together.

Open-MPI [7] which combines technologies and resources from several other projects (FT-MPI, LA-MPI, LAM/MPI and PACX-MPI) is a quite new MPI library. Its modular design is promising and it supports heterogeneous architectures so that the combination of Infi niBand and shared memory is possible.

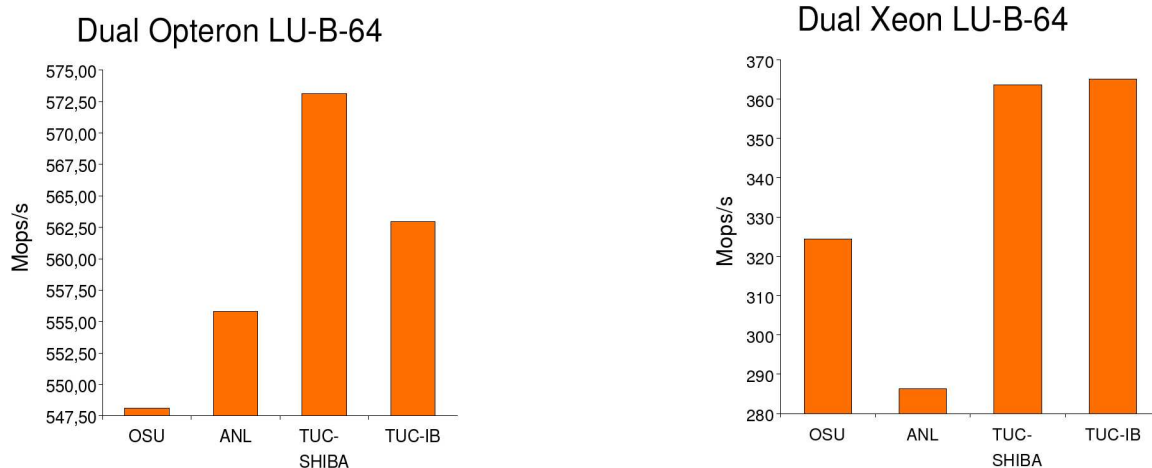


Figure 7: NAS LU benchmark class B on 32 Dual AMD Opterons and 32 Dual Intel Xeons

6 Conclusion and Outlook

We have presented our recent work on integrating shared memory support into our InfiniBand CH3-device for MPICH2 in this paper. By improving the intra-node communication in this way, we also disburden the PCI connection of the InfiniBand adapter for the benefit of the inter-node communication. We have shown two approaches for handling the progress of communication, the Toggle progress and the Adaptive progress and described three different reading schemes for our SHIBA device. We have further shown that the performance of the SHIBA device significantly depends on the architecture of multiprocessor nodes.

Nevertheless, the big problem of wasting memory still remains because the usage of InfiniBand as well as shared memory needs pre-registered memory for the communication. So, one possible direction for further research is to reduce the amount of memory needed for communication and the necessary number of memory copy operations.

Due to the weak expressiveness of the NAS benchmark we will conduct more benchmarks with real applications. Therefore we have also to consider bigger SMPs with four, eight or more CPU cores to evaluate the relationship between the node count, the SMP size and the application's communication patterns.

Another direction is the improvement of collective MPI operations for efficiently combining shared memory and InfiniBand resources.

Acknowledgements

We would like to thank Peter Burger and Markus Brenk from Universität Stuttgart as well as Simon Melzner from Universität Bayreuth. They made it possible that we got access to their InfiniBand clusters to conduct the benchmarks.

References

- [1] *System V Interface Definition Fourth Edition*. The SCO Group, 1995.
<http://www.caldera.com/developers/devspecs/>.
- [2] *IEEE Std 1003.1, 2004 Edition*. The Open Group Base Specifications Issue 6, 2004.
<http://www.unix.org>.

- [3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrisnan, and S. K. Weeratunga. The nas parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.
- [4] L. Chai, S. Sur, H. W. Jin, and D. K. Panda. Analysis of Design Considerations for Optimizing Multi-Channel MPI over InfiniBand. In *Proceedings of the 19th Int'l Parallel and Distributed Processing Symposium, IPDPS*, 2005.
- [5] M. P. I. Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, TN, July 1997.
- [6] M. P. I. Forum. MPI-2 Journal of development. Technical report, University of Tennessee, Knoxville, TN, July 1997.
- [7] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *In Proceedings, 11th European PVM/MPI Users' Group Meeting*, September 2004. Budapest, Hungary.
- [8] P. GmbH. Pallas MPI Benchmarks - PMB, Part MPI-1. Technical report, Pallas GmbH, 2000.
- [9] R. Grabner and F. Mietke. MPICH2-Device for InfiniBand. Master's thesis, Chemnitz University of Technology, 2003.
- [10] R. Grabner, F. Mietke, and W. Rehm. Implementing an MPICH-2 Channel Device over VAPI on InfiniBand. In *Proceedings of the 18th Int'l Parallel and Distributed Processing Symposium, IPDPS*, 2004.
- [11] W. Gropp, D. Ashton, E. Lusk, R. Ross, and B. Toonen. MPICH2 Design Document. Technical report, Argonne National Laboratory, 2002. Draft.
- [12] W. Gropp and B. Toonen. The CH3 Design for a Simple Implementation of ADI-3 for MPICH with a TCP-based Implementation. Technical report, Argonne National Laboratory, 2002.
- [13] InfiniBand Trade Association. *InfiniBand Architecture Specification 1.2*, 2004.
- [14] H. W. Jin, S. Sur, L. Chai, and D. K. Panda. Design and Performance Evaluation of LiMIC (Linux Kernel Module for MPI Intra-node Communication) on InfiniBand Cluster. Technical Report OSU-CISRC-10/04-TR58, Ohio State University, 2004.
- [15] L. B. N. Laboratory. MVICH: MPI for Virtual Interface Architecture, August 2001. <http://www.nersc.gov/research/FTG/mvich/>.
- [16] 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 *In Int'l Parallel and Distributed Processing Symposium (IPDPS 04)*, April 2004.
- [17] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *In the Proceedings of 17th Annual ACM International Conference on Supercomputing*, June 2003.
- [18] S. Pakin and A. Pant. VMI 2.0: A Dynamically Reconfigurable Messaging Layer for Availability, Usability, and Management. Technical report, National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Jan 2002.
- [19] M. Steiger. Shared Memory Support for InfiniBand MPICH2-Device. Master's thesis, Chemnitz University of Technology, 2004.