MPICH for SCI-connected Clusters

Joachim Worringen, Thomas Bemmerl

Abstract— **MPICH is the most commonly used, freely available implementation of the MPI-1 standard including parts of the MPI-2 standard. It is available for nearly every Unix-based system and can use a variety of communication facilities through its low-level *Abstract Device Interface* (ADI-2). However, no adaption to the *Scalable Coherent Interface* (SCI) existed so far. This paper presents the design and implementation of such an adaption consisting of an ADI-2 device for the current MPICH distribution. The performance of this device is compared to other ADI-2 devices of MPICH usable on Intel x86 based clusters and also with a commercial MPI implementation for SCI-connected clusters.**

Keywords— **message passing, cluster, SCI, MPI, MPICH, ADI-2**

I. Introduction

Since the presentation of the first standard [1] in 1994, the *Message Passing Interface* (MPI) has become one of the most commonly used API for parallel computing due to its availability on nearly every parallel computer. Contrariwise, this leads to the necessity to offer MPI for a parallel computer to make it a useful tool for researchers outside the field of computational science.

A. MPICH Implementation

The freely available Open-Source implementation MPICH [2] was very important for this development. MPICH is the most commonly used, freely distributed implementation of the MPI-1 standard (including parts of the MPI-2 standard [3]) which is also used as a base for commercially distributed MPI implementations (PateNT MPI by Genias [4] using a port of the p4 library and others, see below). It is publicly available for nearly every Unix-based system and can utilize a variety of communication facilities. The interface through which the actual communication facility is accessed is defined as the *Abstract Device Interface* (ADI-2 [5], [6]). The ADI-2 interface defines a set of point-to-point send and receive operations which are required by the upper layers of MPICH. This precisely defined interface lead to the availability of MPI on a wide range of platforms, from TCP/IP connected workstations up to supercomputers like the Cray T3D/T3E series.

B. Message-passing on Clusters

In the last few years, a new class of parallel platforms, commonly referred to as *clusters*, has arised. Following the classification given in [7], we refer to homogenous, high-performance group clusters built from PCs, Workstations or SMPs running any operating system and communicating via a *system area network* (SAN) like *MEMORY CHANNEL* [8], *Myrinet* [9] or the *Scalable Coherent Interface (*SCI [10]). Due to the importance of the availability of standard APIs on a parallel computer, several efforts have been made to offer MPI or PVM [11] on this platform:

- Digital has adopted MPICH to utilize the MEMORY CHANNEL SAN to create the commercially distributed *Digital MPI* [12].
- CSAG at the UCSD have implemented PVM and MPI (based on MPICH) [13] on top of their transport layer *Fast Messages* [14] which uses Myrinet for inter-node communication.
- Scali AS has created an MPI implementation on top of SCI named *ScaMPI* [15] which they distribute commercially. It is used for comparison in the performance chapter of this paper.
- *SCIPVM* [16] implements PVM on top of SCI offering the flexibility of PVM. It does, however, not exploit the full performance potential of the SCI interconnect.
- In the scope of the SISCI project [17], a *common messaging layer* (CML [18]) has been developed to serve as a basis for PVM and MPI implementations [19]. However, no results for an MPI implementation based on CML have been published so far.

C. Motivation

In spite of all these efforts, the publicly available MPICH distribution can still only be utilized with TCP/IP for inter-node communication. To use a more sophisticated cluster interconnect like SCI, only a commercially developed MPI implementation can be used (ScaMPI by Scali) which must be purchased and does not come with source code. Support of SCI-connected clusters by MPICH would help to make this platform more affordable and thus more commonly used. Therefore, we developed an ADI-2 device for SCI-adapters.

D. Organization of the Paper

The next chapter informs about the key characteristics of the SCI-connected cluster on which this research was conducted. Chapter III. presents details of the implementation of the device itself, while chapter IV. gives a theoretical performance calculation, an overview over the performance of the current implementation and a comparison to other solutions. The final chapter summarizes the results and gives options for further development.

Joachim Worringen and Thomas Bemmerl are with the Lehrstuhl für Betriebssysteme, RWTH Aachen, Kopernikusstr. 16, D-52056 Aachen, Germany.
E-mail: contact@lfbs.rwth-aachen.de, WWW: http://www.lfbs.rwth-aachen.de .

II. SCI Cluster Platform

A. Hardware

The development of the presented work took place on a cluster of SMPs. The SMP nodes are dual Intel PentiumII (450MHz) boards with 256 MB memory and the Intel BX-chipset. The SCI interconnect is realized with PCI-SCI (32 bit, 33 MHz PCI bus) adapters from Dolphin Interconnect Solutions equipped with the LC2 version of the SCI link chip and revision D of the PSB. The Dolphin drivers were configured to enable speculative reads for a higher remote read performance. Additionally, they are slightly modified to map SCI memory segments to predefined addresses. The nodes are also connected via a switched 100 MBit full-duplex Ethernet using 3COM NICs. The key performance values latency and bandwidth for remote memory access via SCI on this platform are shown in figure 7.

B. Software

To ease the development of efficient parallel programs using the shared-memory model provided by SCI, a complex library has been developed on top of the vendor supplied driver and programming API titled the *Shared Memory Interface* [20] (SMI). It currently supports Unix (Solaris and Linux) on Intel and Sparc platforms as well as Windows NT on Intel platforms and offers a C and Fortran 77 binding. From the many services offered by the SMI library, the implementation of the MPICH device relies just on a small selection:

- Initial configuration of the processes on the cluster, delivery of topology information and finalization of the environment
- Allocation of globally shared memory regions with different physical distributions to account for the NUMA (non-uniform memory access) performance characteristic.
- Dynamic memory allocation within globally shared memory regions.
- Synchronization services (barriers).

Because of the availability of the SMI library on Windows NT, Solaris and Linux, all these three platforms have been used for development, and in fact SCI-MPICH is available on all of these platforms. With the SMI library, SCI-MPICH has been ported to Windows NT in a very short time [21]. The primary development platform, however, is Solaris x86 which also was used for the benchmarks presented in this paper.

III. Implementation of SCI-MPICH

The basic layered design of MPICH is shown in figure 1. Located on top is the MPI API, with the profiling interface below. The MPIR layer is responsible for the transformation of the complex MPI functions into point-to-point communications. These communications are performed by the MPID layer (the ADI-2 device) situated below. An adaption of MPICH to a specific platform is limited to the MPID layer, all the layers above remain untouched.

A. ADI-2 Device ch_smi

The implementation of the SCI-specific ADI-2 device *ch_smi*[1] (figure 2) is based on the SMI library which in turn uses the SISCI API [22] and the IRM driver of the Dolphin PCI-SCI

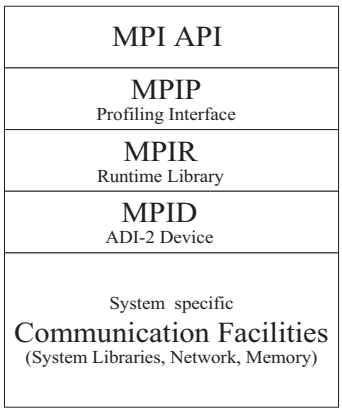


Fig. 1. Layered design of generic MPICH

adapter. However, three phases have to be distinguished regarding these software layers:

- *Initialization*: Only during the initialization (and the finalization) of the processes which form the MPI application, the SMI library and the SISCI API and the IRM driver are required to establish globally shared SCI memory segments mapped into the process's address space.
- *Services*: During the execution of the MPI application, the ch_smi device uses services offered by the SMI library. The most commonly used services are dynamic memory management of shared memory and barrier synchronization.
- *Communication*: Most of the time, the ch_smi device handles point-to-point communication. For this purpose, no underlying software layers have to be utilized, but only direct accesses to the user address space of the process are required. This mode of operation (which is a typical characteristic of SCI) allows for exceptional low communication latencies.

The development of the ch_smi device is based on the *ch_shmem* (shared memory) device which is part of the MPICH distribution. The ch_shmem device is designed for use on multiprocessor SMP systems featuring an UMA architecture[2]. In a first attempt of running MPICH on our SCI-Cluster, the function calls of ch_shmem to allocate shared memory were simply translated to the corresponding functions of the SMI library,

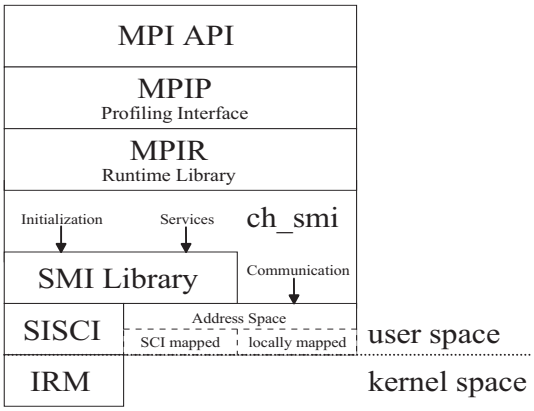


Fig. 2. Layered design of the SCI-MPICH implemenation

1. All ADI-2 devices are named beginning with *ch_* (for channel) followed by the name of the underlying communication facility.

while all protocols and data structures remained the same. This adaption was done in one hour and provided a correctly working ADI-2 device. However, with the strong NUMA characteristic of the SCI-cluster, the performance was very low due to the UMA (uniform memory access)-oriented design of ch_shmem.

Obviously, a complete redesign of the protocols and the according data-structures was necessary to achieve performance values in the proximity of the raw memory transfers over the SCI network. A couple of key characteristics of the SCI network had to be considered:

- Remote write accesses achieve a bandwidth which is ten times higher than the maximum bandwidth of remote read accesses
- The reason for the better write performance is the more efficient use of the stream buffers on the SCI boards by the PCI-bridge. The PCI-bridge supports write gathering to write as much data in one PCI transaction as possible. However, it triggers a PCI transaction for each read operation of the CPU. To take full advantage of the stream buffers for write operations, remote memory accesses must be scheduled in a certain consecutive way to cause as few SCI transactions as possible.
- The use of the stream buffers has the dangerous side effect of creating inconsistent memory states between different nodes. This must be taken into account and, if necessary, has to be avoided.

B. Message transfer protocols

Depending on the length of the message, MPICH chooses between three different protocols (named *short*, *eager* and *rendez-vous*) to transfer a message from one process to another. This allows to find an optimized trade-off between performance and resource usage.

B.1 Short

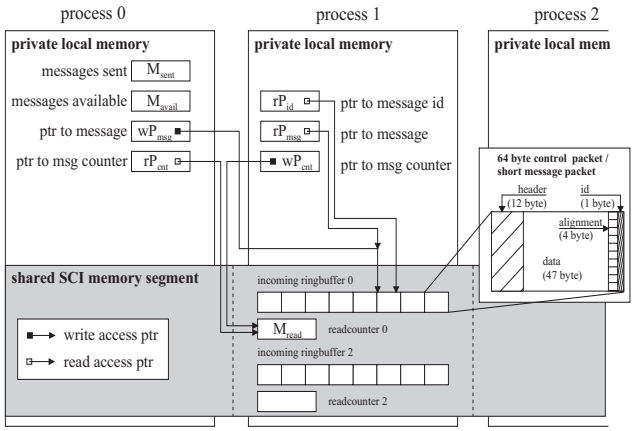
The short protocol is suitable for messages which are small enough to fit into a control packet. The gross size of control packets in SCI-MPICH is 64 byte. This size was chosen because this amount of data makes optimal use of the stream buffers, requires no explicit flushing of the stream buffers and can be transferred in one single SCI transaction. This results in a very low latency (see figure 5) of about 4µs for a remote write of 64 byte. Furthermore, this single transaction won't be split during transmission via the SCI interconnect. This means that the data contained in the packet arrives completely in the order that it was sent. These 64 byte contain a header of 12 byte, a maximum payload of 47 byte followed by an alignment buffer of 4 byte and a 1 byte packet identifier.

The protocol which is used for the transfer of control packets aka short messages is based on a separate ring buffer for each directed sender-receiver pair. The data structures and pseudo code for the basic send and receive operations are given in figure 2 for the case of sending short messages from process 0 to process 1. The critical point, the synchronization, is done using the last byte of the packet. As the control packet is transferred in a single SCI transaction, this byte is guaranteed to be the last byte of the packet to be written in the remote memory. It contains the message identifier, on which the receiver is polling. This identifier is calculated the same way by both, sender and receiver, via a modulo operation. It is ensured that one single slot of a ring buffer never uses the same message identifier for two consecutive messages.

2. The ch_shmem device can optionally be compiled for use on a Convex cache-coherent-NUMA-machine using vendor specific functions. The data-structures and protocols remain nearly identical.

B.2 Eager

The eager protocol lets the sender transmit a message without the receiver requesting it. For this purpose, each process keeps a number of buffers in local shared memory (memory within an SCI shared segment which is physically located on the local node) towards each other process. To manage the eager buffers of another process, a ring buffer of pointers pointing to the according buffers on the receiving process is stored in local shared memory. The transmission of an eager message consists of copying the message data to an available remote buffer and indicating the new message by a control packet. The maximum message size for the eager protocol is typical in the range of a few kB, but can be adjusted to the amount of memory available. Again, the data structures and the basic send and receive algo-




```
SendShortMessage:
    // calculate next message id
    msgid := (msgid + 1) modulo ID_WRAP
    // get a free message slot
    while (M_avail = 0) {
        // update counter via remote read
        M_avail := M_sent - *rP_cnt }
    copy message from local_memory to wP_msg
    *(wP_msg + 63) := msgid
    // local accounting
    increment wP_msg and M_sent
    decrement M_avail

ReceiveShortMessage:
    // poll id field of next message to arrive
    // until it matches the expected id
    while (*wP_id != msgid) {}
    // new message has arrived
    copy message from rP_msg to local_memory
    increment *wP_cnt
    // calculate id of next message
```

Fig. 3. Data structures and pseudo code for the short protocol

rithm are given in a pseudo code notation in figure 3.

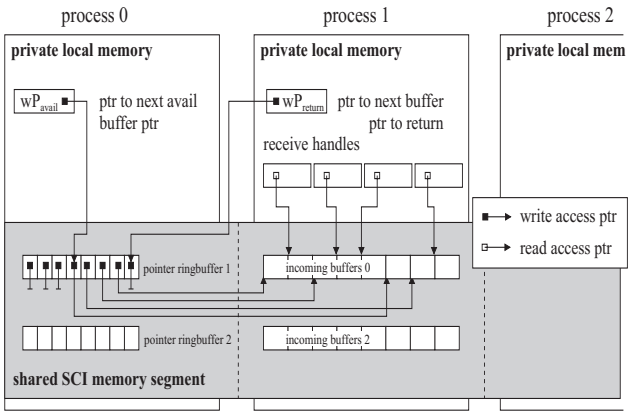
B.3 Rendez-vous

The eager protocol relies on statically allocated resources and thus is not suited to transfer messages which are longer than the size of the incoming buffers on the receiving process. The rendez-vous protocol can transfer arbitrary sized messages by using dynamically managed resources. However, this ability requires a more complex protocol. It is based on handshakes to transmit the address of the transfer buffer and for synchronization in case that the transfer buffer is smaller than the message size.

To increase the effective bandwidth, a write-read-interleave enables the receiver to start reading from the transfer buffer before the sender has filled it completely. The data integrity is ensured by unacknowledged BLOCK_READY control packets sent by the sender after a certain amount of the transfer buffer has been filled. The diagram in figure 5 gives an example for a case where the transfer buffer can not hold the complete message (multi-part transfer) and each part is transferred with a certain interleave.

IV. Performance

The raw transfer performance via the SCI network is depicted in figure 7. Based on these numbers, we can calculate the upper bound of the message passing performance for the different protocols. These numbers are compared with measurements of our SCI-MPICH implementation. Application performance, however, depends on a lot more and complex performance properties. We give a perspective of what can be expected from SCI-MPICH.




```
SendEagerMessage:
    // get a free eager buffer
    while ( *wP_avail = NULL) {
        // wait for update from remote process
        process_other_messages }
    copy message from local_memory to *wP_avail
    // indicate the receipt of a new eager message
    // containing the value of *wP_avail
    send control_packet
    // local accounting
    increment wP_avail

ReceiveEagerMessage:
    // control packet for eager message has
    // arrived, containing a rP_msg
    copy message from rP_msg to local_memory
```

Fig. 4. Data structures and pseudo code for the eager protocol

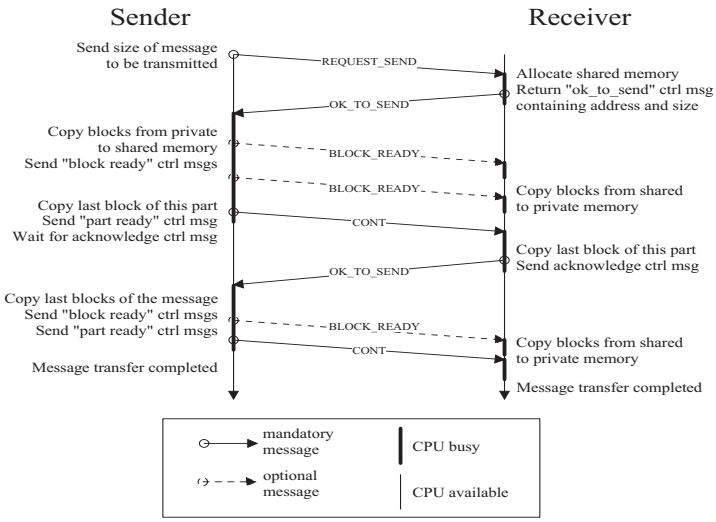


Fig. 5. Definition of the rendez-vous protocol

A. Upper Bound of Message-Passing Performance

Based on the raw performance data for local and remote memory transfers and the protocol specifications, it is possible to calculate an upper bound of the message passing performance that can be obtained on such a system. The relevant parameters for message-passing via shared memory are given in table 1.

Parameter	Description	peak Value
$B_{lr}(N)$	bandwidth of sequential *reads from local memory* for blocks of N bytes	142.9 MB/s
$B_{cr}(N)$	bandwidth for *copying a block of* N *bytes* from *private local memory* to *shared remote memory*	76.3 MB/s
$B_{cl}(N)$	bandwidth for *copying a block of* N *bytes* from *shared local memory* to *private local memory*	334.0 MB/s
$L_{rw}(N)$	minimal latency of a *remote write operation*	2.7 µs
$L_{rr}(N)$	minimal latency of a *remote read* operation	4.4 µs
$L_{lr}(N)$	minimal latency of a *local read* operation	31 ns

Tab. 1. Parameters for message-passing performance

The fact that *reading* a block (in the form of an assignment like `a = b[i]`) results in a lower bandwidth than *copying* a block using some `memcpy()` function is due to the different assembler code used for these operations. For the three protocols short, eager and rendez-vous, this results in the following calculations:

A.1 Short

The amount of data that has to be transferred to send a short message is constant and equals the size of a control packet with $S_{packet} = S_{streambuffer} = 64byte$. This includes the header with a size of $S_{header} = 12byte$. The resulting latency for a short message with a payload of *N* bytes follows as

$$(1) \qquad L_{short}(N) = \frac{S_{packet}}{B_{cr}(S_{packet})} + L_{lr}(4) + \frac{S_{header}}{B_{lr}(S_{header})} + \frac{N}{B_{cl}(N)}$$

In this formula, the first addend describes the remote write of the complete packet by the sending process, the second addend the minimal time for the receiving process to detect the arrival of the new message, the third addend gives the time required to analyze the header and read the data while the last addend describes the store of the message data in the local receive buffer provided by the application. The latency for a control packet can be determined using the same formula, but the value of N that has to be used depends on the exact type of the control packet. The effective bandwidth for short messages can easily be calculated as

$$(2) \qquad B_{short}(N) = \frac{N}{L_{short}(N)}$$

Due to some system-bus-to-PCI-bus host-bridge peculiarities which might result in a SCI packet loss, a verification has to be done by the sending process by reading an error counter on the PCI-SCI adapter board. If an error occurred, the complete packet is retransmitted because in this case, the complete packet did not arrive. Fortunately, this error checking has only a small impact on the latency since it is done by the sending process while the receiving process already reads the new message.

A.2 Eager

The transmission of an eager message includes copying the message data and sending a SEND_ADDRESS control packet of length $C = 16byte$ to indicate the new message. The receiving process has to copy the message data into the receive buffer and return the pointer to the sending process. This gives a latency of

$$(3) \qquad L_{eager}(N) = \frac{N}{B_{cr}(N)} + L_{short}(C - S_{header}) + \frac{N}{B_{cl}(N)} + L_{rw}(4)$$

Again, the bandwidth is calculated as

$$(4) \qquad B_{eager}(N) = \frac{N}{L_{eager}(N)}$$

A.3 Rendez-vous

The rendez-vous protocol is the most complex protocol to describe as the number of control packets that have to be transmitted for one rendez-vous message depends on three parameters:

- the size of the rendezvous-message N
- the size of the dynamically allocated transfer buffer in shared memory S_{buf}
- the chosen block-size for the write-read interleaving S_{block}

This results in $N_{part} = \lceil N/S_{buf} \rceil$ parts in which the message has to be transferred, with a total number of $N_{block} = \lfloor N/S_{block} \rfloor - N_{part}$ blocks that can be transferred interleaved, leaving a rest of $N_{rest} = N - S_{block} \cdot N_{block}$ bytes. All together, it gives us a minimal latency for a rendez-vous message as described in formula 5 if we can assume that $B_{cr}(N) < B_{cl}(N)$ (so that $B_{cr}(N)$ determines the effective bandwidth for interleaved copying). The numerous addends of formula 5 have the following meaning related to the protocol definition:

- The mandatory control packets at the beginning and end of each message and between multiple parts (if the message does not entirely fit into the allocated buffer). These control packets have to transfer 16 bytes of infomation in addition to the standard header of each control packet.
- Remote writing of the interleaved blocks and the optional control packets between the blocks. These packets are optional because if no send packet is available, the transmission of such a packet can safely be omitted.
- The local copy of the last block of each part from shared to private memory. The local copy operations for all other blocks do not influence the latency under the condition $B_{cr}(N) < B_{cl}(N)$ given above.
- Remote writing of the part of the message which does not make up a full block.

Once more, the bandwidth is calculated as

$$(6) \qquad B_{rndv}(N) = \frac{N}{L_{rndv}(N)}$$

B. SCI-MPICH Performance

The foundation of all performance observations are the transfer rates that can be achieved via the SCI interconnect which are shown in figure 7. Optimal performance can be achieved using 64 bit transfers from the processor to the PCI bus which we implemented using the FPU while the transfers which are done by the CPU are 32 bit transfers.

For the measurements of latency and bandwidth, we use a simple Ping-Pong benchmark between two MPI processes (see figure 6 for the basic algorithm). Each process executes blocking send and receive operations to wait for an incoming message (`MPI_Recv()`) and immediately responds (`MPI_Send()`) once it has arrived. The resulting round-trip times are then halved to give the effective latency, from which we derive the bandwidth. Using a high-resolution, low-latency timer based on the Intel

$$(5) \qquad L_{rndv}(N) = (1 + 2 \cdot N_{part}) \cdot L_{short}(16) + N_{block} \cdot \left(\frac{S_{block}}{B_{cr}(S_{block})} + L_{rw}(S_{packet}) \right) + N_{part} \cdot \frac{S_{block}}{B_{cl}(S_{block})} + \frac{N_{rest}}{B_{cr} N_{rest}}$$

```
Process 0:
    MPI_Barrier()
    MPI_Wtime(start)
    for (number_loops)
        MPI_Send()
        MPI_Recv()
    MPI_Wtime(end)
    latency = (end - start)/(2 * number_loops)

Processs 1:
    MPI_Barrier()

    for (number_loops)
        MPI_Recv()
        MPI_Send()
```

Fig. 6. Ping-Pong benchmark between 2 MPI processes

x86 `rdtsc` instruction, we could not only measure the latency of a great number of Ping-Pong cycles, but also create histograms to show the distribution of the latencies .

B.1 Benchmarks

From the measurements of the raw SCI transfer rates and of the local memory transfer rates, we gain the parameters for the model described above (see table 1). We then compare the theoretical values created by the model (which are the upper limit of performance that can be achieved) to the values measured on our cluster. The results are shown in figure 8. The short protocol was used for messages smaller than 48 bytes. For the eager and rendez-vous protocol, we chose overlapping message size ranges to find the intersection between both bandwidth curves. This intersection represents the optimal upper limit for the size of messages transferred via the eager protocol.

Another comparison was done between SCI-MPICH and other communication devices of MPICH for Intel based SMPs (*ch_shmem* with locks and *ch_lfshmem* without locks, both using shared memory on a single node), clusters (*ch_wsock* for TCP/IP communication between nodes) and also for the other existing MPI implementation on top of SCI for Intel x86 machines, ScaMPI. Of course, the simple measurement of the latency and bandwidth for blocking send and receive operations between two processes is not to be considered a complete metric for the performance of an MPI implementation. It rather is thought to give an impression of the performance of the core functionality.

The system on which we conducted the ScaMPI measurements is a 16-node Siemens *hpcLine* running Linux, which consists of nearly identical hardware (PentiumII CPUs on Intel BX boards, 32 bit Dolphin PCI-SCI adapters with the same chip revisions) differing only in the CPU clock speed of 400MHz compared to 450MHz of our cluster (on which all other benchmarks were run). Furthermore, it has to be noted that we used the factory default settings of all ScaMPI tuning parameters [23] for our measurements, only the size and number of the eager

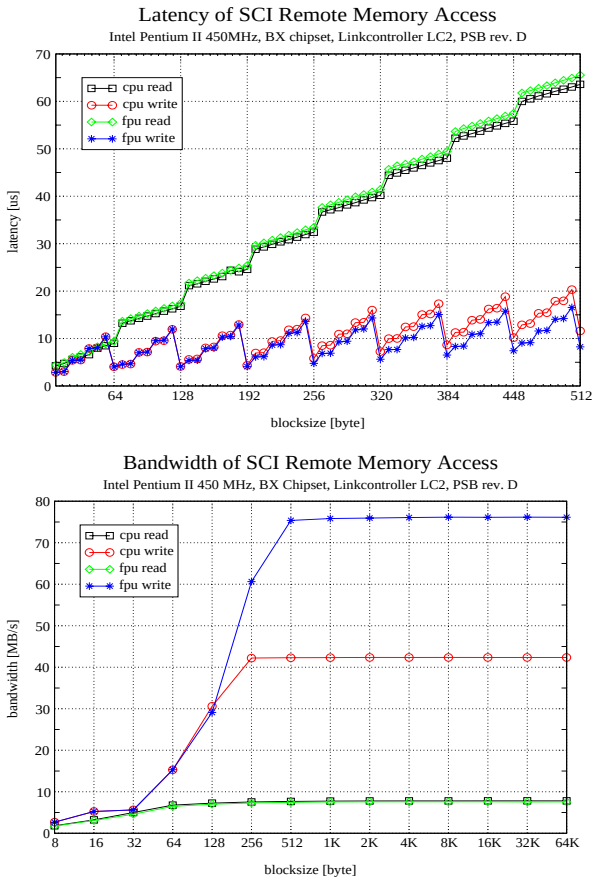


Fig. 7. Bandwidth and latency of remote memory access via SCI using CPU (32 bit) and FPU (64 bit) transfers

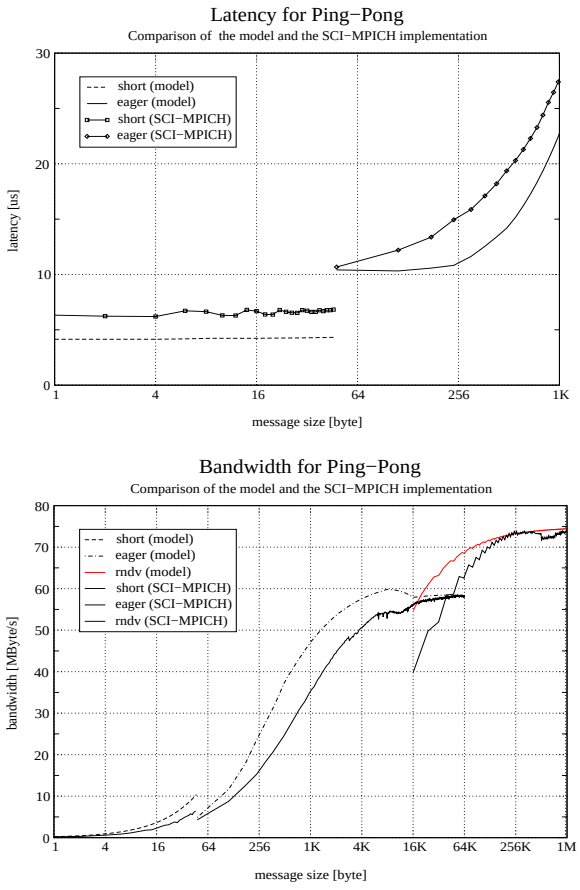


Fig. 8. Modeled and measured Ping-Pong bandwidth and latency (roundtrip/2) between 2 MPI processes

buffers was set equal to the settings used by SCI-MPICH.

The *ch_wsock* device for MPICH is a device for our own NT port of MPICH to Windows NT which uses Windows sockets for communication. As it performs better than the *ch_p4* device-under NT, Solaris and Linux, we chose to use it as a basis for the comparison of MPI via SCI and via the usual fast ethernet.

The results of this comparison are depicted in figure 9. The latency for small messages via the ch_wsock device is not displayed as its minimum latency is about 150 µs.

B.2 Discussion

The benchmark results show that in terms of latency, SCI-MPICH is able to compete with the SMP communication devices shmem and lfshmem. The difference of approximately 2.5µs for SCI-MPICH short messages against the latency from the model mainly represents the internal overhead of the MPIR layer. This overhead can not be reduced without changing the general MPICH code which is not desired.

For messages send via the eager protocol, SCI-MPICH's bandwidth trend resembles the curve of the model and comes very close to the theoretical maximum for message sizes above 16kB. The bandwidth as described by the model reaches its peak value for message sizes of 8kB, then approximates to a value little below the peak value. This behavior is due to the peak of B_{cl} which is located at a block size of 8kB. However, the effective bandwidth is dominated by B_{cr} and the internal overhead of the MPIR layer (which is not described by the model). This results in a less distinct peak at 8kB message size and an approximation of the effective bandwidth to the model bandwidth for larger message sizes.

	min. latency	**max. bandwidth**
SCI-MPICH	6.6 µs	73.8 MB/s
MPICH lfshmem	1.6 µs	141.9 MB/s
MPICH shmem	4.6 µs	110.4 MB/s
ScaMPI	16.2 µs	65.7 MB/s
MPICH wsock	146.7 µs	10.2 MB/s

Tab. 2. Ping-Pong minimal latency (round-trip/2) and maximum bandwidth between 2 MPI processes

The maximum bandwidth for messages transferred via the eager protocol being situated well below the raw SCI transfer rates is due to the miss of an interleaved copy mechanism: for the eager protocol, the receiver is not notified on the arrival of the new message until it is transferred completely. This leads to the simple addition of the time required for the remote copy operation from the sender to the receiver the and local copy operation of the receiver from shared to private memory.

The rendez-vous protocol delivers a performance which is close to the values predicted by the model for message sizes beyond 128kB. The important technique is the interleaving of write and read operations by the sender and the receiver. Without this overlapping, the bandwidth for large messages decreases as the devices shmem and lfshmem for SMPs show. SCI-MPICH's bandwidth for large messages comes close to the raw peak bandwidth of the SCI interconnect. The only drawback in performance are the partial transfers for messages which do not fit entirely in the memory pool for rendez-vous messages.

The effective bandwidth for rendez-vous messages below 128kB could be improved by using a modified protocol which uses implicit synchronization for the write-read interleaving. Instead of sending control packets for each block, it is possible to only send such a packet for the first block and use special synchronization marks inside the message buffer for the rest of the transfer. This would reduce the synchronization overhead and lead to higher point-to-point bandwidth for small rendez-vous messages. However, it would also lead to a polling behavior of the receiving process for the usual case of $B_{cr} < B_{cl}$ resulting in a waste of CPU cycles: the CPU-available phases (see figure 5) between each transferred block would disappear. This would lead to a lower accumulated bandwidth for other communication types than unidirectional point-to-point sends.

The minimal latency that ScaMPI achieves is significantly higher than SCI-MPICH's, and also the bandwidth for messages transferred via either protocol is lower. It seems that ScaMPI uses at least two SCI transactions to transfer a control packet [24]. We tried to use an equal buffer layout of ScaMPI and SCI-MPICH using some of the startup-parameters that ScaMPI offers. It is possible that ScaMPI delivers results similar to SCI-

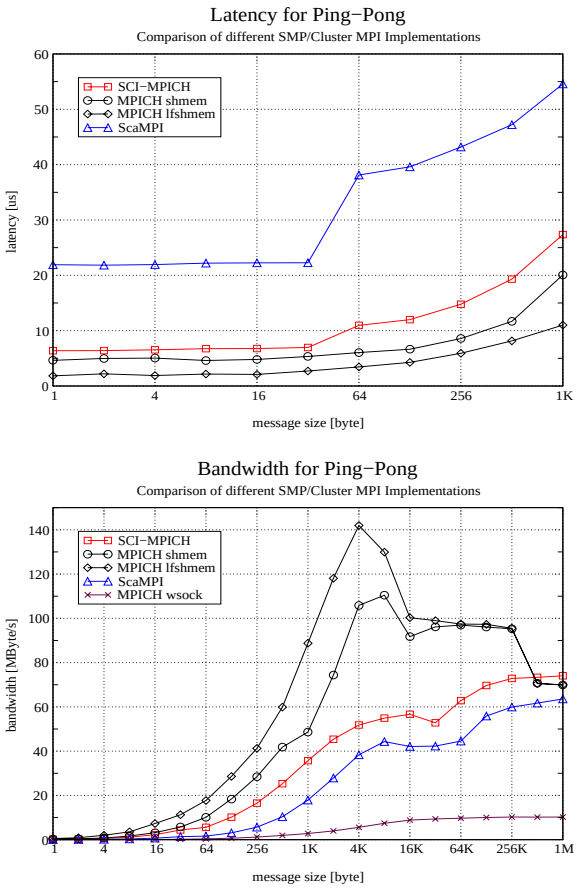


Fig. 9. Ping-Pong bandwidth and latency (roundtrip/2) between 2 MPI processes

MPICH's with an optimized memory configuration, but surpassing SCI-MPICH will be difficult regarding the overhead of only 2.5µs for control packets and the efficiency (effective bandwidth related to raw SCI transfer rate) of 96% for large eager and rendez-vous messages which SCI-MPICH achieves.

The shared-memory SMP devices shmem and lfshmem do, of course, deliver lower latencies and a considerably higher bandwidth for messages which fit well into the memory caches. However, the performance gap is less significant than might be expected. First tests with the application benchmark Spark 98 [25] even show that it may result in higher performance to run SCI-MPICH processes on distinct nodes than using an equivalent SMP setup. This effect is due to the simple design of the memory interface in low-cost Intel based SMP machines which gives only half of the bandwidth to each CPU if they are competing for memory access. With SCI-MPICH, the communication between processes imposes less stress to the memory bus. This effect is also responsible for the bandwidth of large rendezvous messages being higher for SCI-MPICH than for the SMP devices, even if these do also use an interleave technique.

The results of the wsock device for MPICH are very close to the raw TCP/IP performance on the 100Mbit ethernet. However, the resulting performance is well below the performance of SCI-MPICH in terms of bandwidth and even more in terms of latency. The introduction of Gigabit ethernet networks will improve the bandwidth, but the inclusion of the operating system into the message transmission will avoid latencies as low as SCI based solutions achieve today.

V. Summary & Future Work

The presented MPI implementation offers full MPI-1 functionality based on the widespread and reliable MPICH distribution which is extendable with a variety of important tools for tracing or parallel debugging. What makes this implementation special for cost-effective cluster-solutions are the extremely low latencies of small messages and the high maximum bandwidth. The free availability of the source code may help to establish SCI connected clusters as a high-performance, solid yet affordable platform for technical and scientific computing next to the popular ethernet connected clusters. These perform nearly an order of magnitude worse when it comes to inter-node communication.

However, while the software is running stable in our configuration of 6 nodes, it is still in an early stage of development. It has not yet been tested on larger configurations, and the performance that it delivers to real applications has to be evaluated in depth. It offers a lot of room for improvement next to the two key parameters bandwidth and latency which are now near to the theoretical maximum. The most important issues that we have in mind are briefly discussed below.

- Driver Issues (SMP and Caching)
  The driver for the Dolphin SCI-PCI adapter which were available for the development on Linux and Solaris x86 do neither support multiple processes on one node nor remote interrupts. The resulting limitations are automatically solved with fully functional drivers, as internal tests with the Windows NT version of SCI-MPICH and with new beta-release drivers for Linux and Solaris x86 show.
- Multithreading
  Tests with the Spark98 benchmark indicate that memory-intensive MPI applications perform better if each process is run on a dedicated node than using multiple process on a SMP node. This is due to the simple design of the memory interface on the dual CPU main boards. To make efficient use of the second CPU, a second thread could be used for message transfers. In conjunction with remote interrupts, this would provide fully asynchronous sending and receiving.
- DMA
  Although, on the hardware which is used for this development, DMA transfers via SCI are generally slower than transfers by the CPU, they allow for true asynchronous transfers. If computation and communication can be overlapped, it might give more performance even with lower transfer bandwidth.
- Dynamic configuration
  The configuration of the protocols and the according data structures is currently set up on the application startup. To change these settings, the application needs to be restarted with a different configuration description. Dynamic configuration on runtime would allow to adjust settings (i.e. the size and the number of the transfer buffers for the eager protocol) during the execution of the application, potentially in an adaptive manner by analyzing the communication pattern to optimize the setup for this specific application.
- I/O
  Parallel applications often suffer from the bottleneck of slow file I/O. MPICH offers parallel I/O by the implementation of MPI-IO, ROMIO. We are currently developing support of ROMIO for parallel file access via SCI.
- Collective Operations
  Especially for larger configurations, the performance of collective operations is important for the overall performance of an application. The standard MPICH routines for collective operations should be replaced with special SCI shared memory functions. This is already done for `MPI_Barrier()`.
- Cluster Manager
  The startup of a SCI-MPICH application under Unix is currently performed via a script and remote shell invocations without any queuing, scheduling and protection (the same is true for the startup under NT which is performed with the tool *NTRexec* [21]). If one process should crash, the others continue running. These and other problems are currently addressed with the development of a Java-based cluster management software and new startup and shutdown techniques for SCI-MPICH.

REFERENCES

[1] Message Passing Interface Forum: *MPI: A message-passing interface standard.* International Journal of Supercomputing Applications, 8(3/4), 1994

[2] W. Gropp, E. Lusk, N. Doss and A. Skjellum: *A high-performance, portable implementation of the MPI message passing interface standard.* Parallel Computing, 22:789-828, September 1996

[3] Message Passing Interface Forum: *MPI-2: Extensions to the Message-Passing Interface.* http://www.mpi-forum.org/docs/docs.html, July 1997

[4] Genias Software GmbH: *Parallel Tools Environment on NT.* http://www.genias.de/products/patent

[5] E. Lusk and W. Gropp: *The implementation of the second generation MPICH ADI*. MPICH working note (draft) from http://www.mcs.anl.gov/mpi/mpich/workingnote/adi2impl/note.html, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997

[6] E. Lusk and W. Gropp: C*reating a new MPICH device using the channel interface.* Technical Report ANL/MCS-TM-213, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1995

[7] R. Buyya: *High Performance Cluster Computing: Architectures and Systems, Volume 1.* Prentice-Hall, 1999.

[8] R. Gillet, *MEMORY CHANNEL Network for PCI: An Optimized Cluster Interconnect.* IEEE Micro (February 1996):12-18

[9] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, W. Su: *Myrinet - a gigabit-per-second local-area network.* IEEE Micro, 15(1):29-26, February 1995

[10] IEEE: ANSI/IEEE Std. 1596-1992, *Scalable Coherent Interface (SCI)*. 1992

[11] V. S. Sunderam: *PVM: A Framework for Parallel Distributed Computing*. Concurrency: Practice and Experience, Vol. 2 No. 4, pp.315--339, December 1990. available from http://www.netlib.org/ncwn/pvmsystem.ps

[12] J. Lawton, J. Brosnan, M. Doyle, S. Ó Riordáin, T. Reddin: *Building a High-performance Message-passing System for MEMORY CHANNEL Clusters.* Digital Technical Journal, Vol. 8 No. 2, 1996

[13] M. Lauria, A. Chien: *MPI-FM: High Performance MPI on Workstation Clusters.* Journal of Parallel and Distributed Computing, Vol. 40, No. 1, pp. 4-18, January 1997

[14] M. Lauria, S. Pakin, A. Chien: *Efficient Layering for High Speed Communication: the MPI over Fast Messages (FM) Experience.* Dept. of Computer Science and Engineering, University of California, San Diego, accepted for publication on Cluster Computing, 1999

[15] Scali AS: *Scali MPI - ScaMPI.* http://www.scali.com/html/scali.html

[16] I. Zoraja, H. Hellwagner, V. Sunderam: *SCIPVM: Parallel Distributed Computing on SCI Workstation Clusters*. To appear in: Concurrency: Practice and Experience.

[17] ESPRIT HPCN Project EP23174: *Standard Software Infrastructure for SCI based parallel systems - SISCI.*

[18] M. Eberl, H. Hellwagner, Bjarne G. Herland: *A Common Messaging Layer for MPI and PVM over SCI*. Proc. HPCN Europe 1998, Amsterdam, The Netherlands, April 21-23, 1998, LNCS 1401, Springer Verlag 1998

[19] M. Eberl, W. Karl, M. Leberecht, M. Schulz: *Eine Software-Infrastruktur für Nachrichtenaustausch und gemeinsamen Speicher auf SCI-basierten PC-Clustern*. 2nd Workshop Cluster Computing, Karlsruhe, March 1999

[20] M. Dormanns, W. Sprangers, H. Ertl, T. Bemmerl: *A Programming Interface for NUMA Shared-Memory Clusters.* Proc. High Performance Computing and Networking (HPCN), pp. 608-707, LNCS 1225, Springer, 1997

[21] J. Worringen, K. Scholtyssik: *MP-MPICH: Multi-Platform MPICH.* http://www.lfbs.rwth-aachen.de/~joachim/MP-MPICH.html

[22] F. Giacomini, T. Amundsen, A. Bogaerts, R. Hauser, B.D. Johnsen, H.Kohmann, R. Nordström, P. Werner: *Low-level SCI software functional specification*, Esprit Project 23174, CERN, Geneva and Dolphin Interconnect Solutions AS, Oslo, Version 2.1.1. March 1999

[23] Scali AS: *ScaMPI User's Guide*, Scali AS, Oslo, Version 1.7.0, 1999

[24] Scali AS: *ScaMPI - Design and Implementation.* Scali AS, Oslo, Version 1.7.0, 1999

[25] O'Hallaron, D.R.: *Sparse Matrix Kernels for Shared Memory and Message Passing Systems.* Technical Report CMU-CS-97-178, School of