Efficient Asynchronous Message Passing via SCI with Zero-Copying

Joachim Worringen[1], Friedrich Seifert[2] and Thomas Bemmerl[1]

Abstract. **Passing messages between processes, as it is done when creating parallel applications based on MPI, does always involve copying data from the address space of the sending process to the address space in the receiving process. The fastest, and commonly most efficient way for this is a direct copy operation between these two locations without any intermediate copies. In this paper, we present different techniques, implemented in SCI-MPICH, to achieve such a behavior for MPI on SCI-connected clusters. These techniques use both, CPU and DMA driven data transfers, and require support through MPI API functions or are transparent for the MPI application by making use of new advanced techniques in the SCI driver software. We describe the implementation details in SCI-MPICH and the underlying SMI (Shared Memory Interface) library and evaluate the performance achieved in different communication setups.**

Keywords: **MPI, MPICH, SCI, zero-copy, overlapping of computation and communication**

I Introduction

The development of fast interconnects for the PCI I/O bus helped to diminish the bottleneck in communication performance which the standard 100Mbit network in commodity clusters presents for many applications when scaling beyond a small number of nodes. This was essential to establish compute clusters built of commodity components as a serious alternative to custom-designed solutions by the established HPC industry. However, next to supplying an interconnect with giga-bit link-level bandwidth, additional efforts are required to efficiently use this increased communication bandwidth from a user-level application. The most commonly used programming model for technical and scientific parallel applications is message-passing, with MPI [1][2] as the de-facto API. Therefore, a high-performance cluster computing platform for technical or scientific applications needs to deliver the performance of the interconnect through MPI to achieve high application performance.

A. The Need for Zero-Copy Protocols

The most commonly used communication network in commodity clusters is *Fast Ethernet* with a link-level bandwidth of 100 MBit/s, equivalent to a peak bandwidth of 11.92 MiByte/s[1]. Using the TCP/IP protocol, typical current-generation cluster nodes come close to this value for inter-node transfer. But when looking at TCP/IP transfers via Gbit-Ethernet (1 GBit/s) or even the local loop-back device, the bandwidth of the TCP/IP protocol suffers from numerous copy-operations performed on the to-be-transmitted data on it's way from the source to the destination buffer in user-space [3]. The impact of *n* additional copy-stages in the data-path (resulting in a *n-way* copy protocol), each with a distinct bandwidth B_i, on the resulting effective peak bandwidth B_{eff} achieved for a data transmission via a channel with a peak bandwidth of B_{peak} can be expressed in a simple formula:

$$B_{eff} = \frac{1}{\frac{1}{B_{peak}} + \sum_{i=1}^{n} \frac{1}{B_i}} \quad (1)$$

The efficiency ε can be defined as

$$\varepsilon = \frac{B_{eff}}{B_{peak}} \quad (2)$$

The graph in figure 1 illustrates this efficiency for the case of a communication setup where each inter-node message is copied twice (once on the sender- and once on the receiver-node), only once or without any intermediate copies. For the two cases C1 and C2, a copy bandwidth of 200 MiByte/s is assumed, which is a good average value for the memory bandwidth of current commodity nodes. The case C3 illustrates the scenario of locking memory at a rate of 4 GB/s for zero-copy data transfer (see following chapters and [4]). The interconnect bandwidth B_{peak} for the inter-node data movement is given on the x-axis from 0 to 300 MiB/s.

This graph demonstrates why zero-copy transfers are becoming more and more crucial for efficient utilization of high-speed interconnects: for B_{peak} = 10 MiB/s, even a two-copy protocol achieves an efficiency of more than 90%. This efficiency drops to less than 40% for B_{peak} = 250 MiB/s, which is a typical bandwidth for high-performance interconnects like SCI (see figure 5).

B. Related Work

Different *thin* communication protocols and interfaces have been specified and implemented [5][6] which move the performance-critical path of the communication completely into user space, avoiding costly context-switches and copy operations to

1 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 .

2 Lehrstuhl für Rechnerarchitektur, TU Chemnitz
Straße der Nationen 162, D-09107 Chemnitz, Germany
e-mail: friedrich.seifert@informatik.tu-chemnitz.de
WWW: http://www.tu-chemnitz.de/informatik/RA .

1. To avoid confusion, we are using the standard base-ten (M) and base-two (Mi) abbreviations for the respective *Mega* (*Kilo*, *Giga*) prefixes.

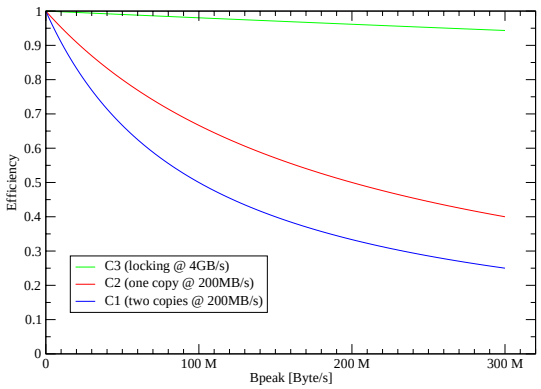

Fig. 1. Efficiency of n-way copy communication for variable interconnect bandwidth

move data from the user-buffer to kernel-buffers. The *Virtual Interface Architecture* (VIA) [7] describes a common interface to such thin communication architectures. Naturally, such communication protocols serve as a basis for MPI implementations. This can be done by adding a new communication device to the MPICH distribution [8] as it was done in [9][10]. Others have designed new MPI implementations which make use of the VI architecture [11][12]. Although many of these MPI implementations claim to support zero-copy transfers, only very few do supply detailed information on the implementation and effects on performance [13][14], but do not compare the zero-copy protocol performance to other possible protocols. [15] does make detailed performance comparisons between different protocols and interconnects which show the good suitability of SCI for zero-copying. However, these are measurements for VIA, not for MPI.

The other known implementation of MPI for SCI-connected clusters, ScaMPI [16], does not use DMA or zero-copy techniques. [17][18] describe the implementation of MPI on the proprietary *Memory Channel*. However, current performance numbers [19] indicate that little improvement has been made in the last years.

C. Zero-Copy with SCI

Naturally, SCI [20] with its transparent remote memory access is the leanest inter-node communication protocol that can be thought of, representing pure zero-copy. However, the implementation of SCI via the I/O bus implies several restrictions which so far disallowed for a general zero-copy communication between arbitrary user-allocated memory buffers. It is only recently that work has been done to be able to expose arbitrary regions of a process' address space for direct remote access via the SCI interconnect [4] by *registering* a user-allocated memory buffer with the SCI driver. In this paper, we will show how this new technique has been integrated in SCI-MPICH and demonstrate the impact on MPI communication performance.

D. MPI Performance Metrics

The characterization of the performance of an MPI platform, consisting of node and interconnect hardware and the MPI library, and even more the comparison of the performance of different MPI platforms is a very delicate topic. Ultimately, such a comparison of characteristics is only possible by comparing the effective performance of a given application solving a specific problem. However, this approach is naturally limited by the uncountable number of possible application and problems. It leads to a situation where each MPI platform is presented with its optimal application-problem combination; a projection to the effective performance of other application-problem combinations is not possible without analyzing the communication characteristics of both application-problem combinations.

The other extreme are micro-benchmarks which give numbers for the performance characteristics for a synthetic, highly limited workload. The most frequently presented micro-benchmark performance numbers are bandwidth and latency for ping-pong communication between two processes although these numbers are not necessarily related to high application performance [21]. However, these numbers help to evaluate the efficiency of an MPI implementation when they are compared to the raw performance number of the underlying interconnect. For this reason, this paper will also present such numbers.

There are numerous other characteristics for the performance of MPI implementations, like performance of collective operations or the possibility of overlapping communication and computation. The latter is not supported by many MPI implementations (because it is not a required characteristic), Thus, many MPI applications are not designed to exploit potential performance benefits from this approach although such a potential does exist [22]. Therefore, we will emphasize on the support of asynchronous communication with zero-copy protocols that is offered by SCI-MPICH.

Finally, application-kernel benchmarks represent a compromise of application-problem evaluation and micro-benchmarks. The high acceptance of the NPB benchmarks [23] demonstrates the relevance of this approach. This paper evaluates the performance impact of zero-copying for one application kernel of the NPB benchmark suite.

E. Organization of the Paper

The next chapter will explain how the SMI library supports memory registration. Chapter III illustrates how the designer of an MPI application can optimize the application for communication performance with respect to zero-copy protocols, and describes the implementation of the *true zero-copy* protocols in SCI-MPICH. In chapter IV, the performance of the existing one- and two-copy protocols will be compared with the expected performance of the zero-copy protocol variants.

II Managing Shared Memory with SMI

The *Shared Memory Interface (SMI)* [24] is a user-level library to support shared memory programming on SMPs and especially on SCI-connected clusters of SMPs, representing a NUMA-System. SCI functionality is accessed via the SISCI API which is much more low-level. The SMI API consists of 70+ functions and has some intended similarity with the MPI API.

A. Registering and Sharing Memory

Establishing shared memory areas between processes (called

regions in SMI) is done via the `SMI_Create_shreg()` function. This function can be called individually or collectively, depending on the *type* of the region to be established. To register an already allocated buffer with the SCI driver and export it, a region of type `LOCAL` with the attribute `REGISTERED` needs to be established, passing the address of the user-allocated buffer. If this region should not be visible to external nodes, but should only serve as a source for DMA transfers, the additional attribute `PRIVATE` needs to be provided. `SMI_Create_shreg()` calls the appropriate SISCI functions.

Another process which wants to access a memory region of type `LOCAL` exported by another process needs to create a region of type `REMOTE` or `RDMA`. A `REMOTE` region allows full transparent PIO and DMA access, while an `RDMA` region can only be accessed by DMA operations because it is not mapped into the address space of the remote process. This simplifies the connection to the remote segment as described in [4] and is sufficient for asynchronous zero-copy data transfers in most cases (see II.B).

The required information to establish the SCI shared memory region or DMA target region are the SCI segment id and the PCI-SCI adapter id of the remote segment. This information is distributed between all processes in the case of a collective establishment of a new region. If the region is established in a non-collective manner as it is done with `RDMA` and `REMOTE` regions, it needs to be transmitted separately from the exporter of the region to the importer either by SMI-supported communication (`SMI_Send()` / `SMI_Recv()` or shared memory) or by external communication means like MPI.

B. Alignment

Shared memory regions need to be placed and sized with page-size granularity. In contrast, user-allocated buffers are usually not aligned this way, but start at arbitrary addresses. This is not relevant to the process registering its user-buffer as the address of the buffer remains the same. It only means that eventually some of the memory before and after the user memory will be registered, too. Potential multiple locking of different user-buffers located on the same page is handled by the LMM (*locked memory manager* [24], kernel module to lock/pin down arbitrary memory regions). If a remote process wants to access the registered user-buffer and maps the related shared memory region into its address space, the starting address of this region is the beginning of the remote user-buffer - the SCI driver includes the offset when connecting to a remote segment.

For DMA data transfer operations, certain alignment restrictions concerning addresses, offsets and sizes need to be met (which is the case for usual basic data types). In case that a misalignment occurs, the SMI library can try to eliminate this using PIO operations which transfer the misaligned data using the CPU (which has no alignment boundaries). This requires that the remote region is mapped into the local address space.

C. Detection of Memory Type

Because SMI keeps track of all local and remote shared memory regions established by the user, it can determine if a given memory address range is made up of non-shared memory, local SCI shared memory, local non-SCI shared memory or remote SCI shared memory and if the memory was allocated by the user or by the SCI driver. This avoids duplicate registration of memory, and allows higher software levels (like SCI-MPICH) to choose an appropriate way of accessing the memory behind this address range (to choose the right protocol, for SCI-MPICH).

D. Data Transfer

Although the standard `memcpy()` function can be applied on shared memory regions, SMI also provides optimized memory copy functions: `SMI_Memcpy()` for synchronous copy operations, and `SMI_Imemcpy()` for asynchronous, DMA-based copy operations on shared memory regions. `SMI_Memtest()` and `SMI_Memwait()` are used to test or wait for the completion of an asynchronous memory transfer. `SMI_Memcpy()` / `SMI_Imemcpy()` accept additional attributes which describe the type of the source and destination memory: local non-shared memory, local SCI-registered memory or remote SCI shared memory. These attributes can also be determined by the SMI library itself as described above. However, it is more efficient to pass the information (which is often known to the application) down to the SMI library using the described attributes.

All memcpy-style functions for data transfer require a mapping of source and target memory into the local address space. However, mapping remote memory comes with high overhead, and is not required if block-transfers of data are to be performed via DMA. The specification of a remote memory location as a target or source for a DMA transfer does only consist of the SCI segment number, the SCI node number and possibly an offset relative to the start of the segment. To support such low-overhead DMA transfers, the SMI library offers the region type `RDMA` which does not provide an address, but is only to be used with `SMI_Put()` and `SMI_Get()` functions which write to respectively read from a remote memory location using DMA. Additionally, `SMI_Iput()` and `SMI_Iget()` functions can be used for asynchronous transfers.

III Zero-Copy Support in SCI-MPICH

Message Passing is the dominant, because most scalable and portable programming model in high-performance computing today [24]. The MPI standard made this programming model extremely portable. This also means that an efficient MPI implementation is crucial for any high-performance platform to succeed. At the Lehrstuhl für Betriebssysteme, we have developed an open-source implementation of MPI for SCI-connected clusters named SCI-MPICH [27][28] which we will use to evaluate the benefits of zero-copy protocols on the performance of low-level message-passing and on application performance.

SCI-MPICH uses three different message transfer protocols, depending on the size of the payload to be transferred: *short* (0 up to 128 Byte), *eager* (129 Byte up to 32 kiB) and *rendez-vous* (more than 32 kiB)[1]. By default, all protocols perform PIO-based data transfer. The eager and rendez-vous protocol do also support DMA-based message transfers on demand of the user.

1. The limits for short and eager protocols are freely adjustable by the user; we give the default settings.

However, the size of messages transferred via the eager protocol is usually too small[1] to make registering of the communication buffers and DMA transfer efficient: copying 32kiB to remote memory does only take 195 s,while registering the send buffer and transferring the data via DMA would take at least 400 s. Also, zero-copy transfers are generally not possible with the eager protocol due to the principle of this protocol of delivering messages unannounced. Therefore, we concentrate on implementing a zero-copy variant of the rendez-vous protocol, which promises the best performance gain: registering and thus eventually locking a memory region in the kernel is at least 5 times faster than copying it once (comparing the rate the LMM achieves for registering [24] and typical `memcpy()` rates).

The existing synchronous and asynchronous implementation of the rendez-vous protocol is described in [27][28]. Asynchronous message transfers are especially valuable in combination with DMA for CPU-less message transfers in the background, supported i.e. by `MPI_Isend()` / `MPI_Wait()`.

A. Synchronous PIO-Transfers

CPU-driven PIO-transfers are not well suited for asynchronous transfers as a second CPU is required to make it efficient. Therefore, we only consider synchronous PIO transfers for which a zero-copy implementation removes the receiver-side copy operation from the rendez-vous memory pool into the user-allocated receive buffer by registering and exporting this buffer as a SCI shared memory segment. This segment will be imported by the sender and filled up with the data from the sender buffer. The related synchronous zero-copy PIO rendez-vous protocol is illustrated in figure 2.

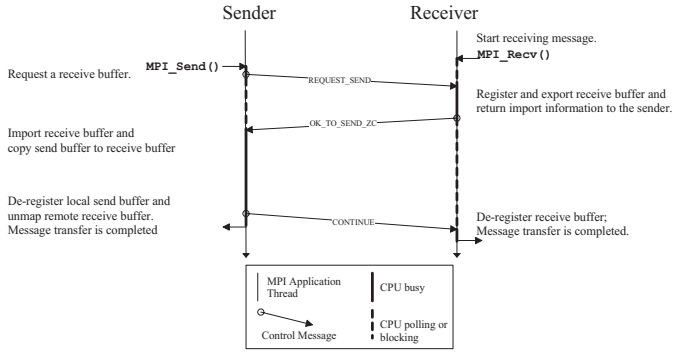

Fig. 2. PIO-based zero-copy rendez-vous protocol

Compared to the original protocol, this variant uses a new type of control message: OK_TO_SEND_ZC. While the OK_TO_SEND message contained an offset and size of a buffer in the rendez-vous memory pool of the receiver, the OK_TO_SEND_ZC message informs the sender on the SCI segment id, SCI adapter id and offset by which the sender can access the exported receive buffer. When the receiver reads a CONTINUE control message for a completed message transfers, it checks if this was a zero-copy transfer and de-registers the buffer with the SCI driver.

B. DMA-Transfers

Zero-copy DMA transfers require the same communication protocol as zero-copy PIO transfers plus additional local steps: the user-allocated send buffer needs to be registered with the SCI driver in order to be used as a DMA source. This local registration can overlap with the registration of the receive buffer as illustrated in figure 3.

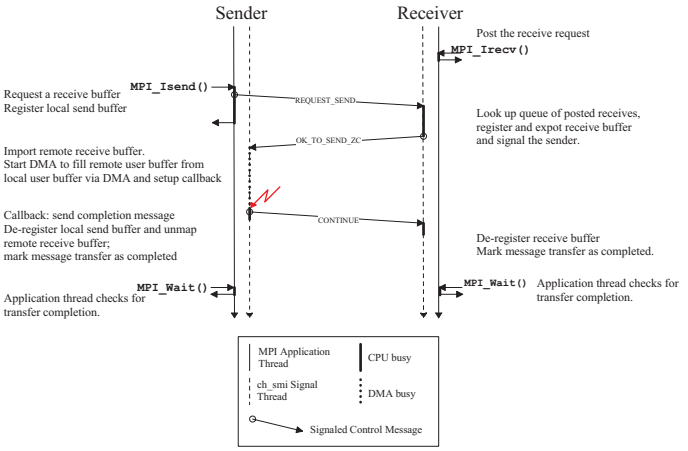

Fig. 3. DMA-based zero-copy rendez-vous protocol

When both, local memory registration and remote memory registration and export are complete, the DMA transfer can start. On completion, the CONT message indicates the completeness of the transfer to the receive process, while the local process de-registers the send buffer.

C. MPI Support for Memory Allocation

The overhead that occurs with registering user-allocated memory for SCI zero copying reduces the effective bandwidth. We have introduced techniques into SCI-MPICH based on the MPI interface which the user can use to reduce this overhead.

C.1 Persistent Communication

MPI supports a mode of communication with fixed send or receive buffers which is called *persistent communication.* A request for persistent communication is initialized once with `MPI_Send_init()` or `MPI_Recv_init()` respectively and bind the list of communication arguments for an asynchronous (non-blocking) send or receive operation to the request. Subsequently, this request may be activated as often as required by passing its handle to `MPI_Start()`. This will cause the associated communication operation to start. If the request is no longer needed, it can be deleted with `MPI_Request_free()`.

Persistent communication supports efficient zero-copying because the user specifies buffers that will frequently used for communication. Registering these buffers with SCI once the request is created allows zero-copy operation for all subsequent operations without the overhead of repeated registering and de-registering. Because persistent communication operations are always asynchronous, they will be performed via DMA.

However, the setup of a persistent operation is defined as a

1. On the described test platform, eager messages deliver less bandwidth than rendez-vous messages for message sizes of above 38kiB due to the memcpy()-pipelining in the rendez-vous protocol.

strictly local operation. This means that the matching remote buffer can not be determined by SCI-MPICH. Thus, this buffer can not be imported during the setup of the operation, but only after the first communication has actually taken place.

C.2 Memory Allocation via MPI

In the context of single-sided operations, the MPI-2 Standard defines `MPI_Alloc_mem()` and `MPI_Alloc_free()` functions to allocate and free memory of an implementation and platform dependent type which allows the MPI library together with the communication subsystem to perform better in case these buffers are used for communication. SCI-MPICH has implemented these functions to work with local SCI segments. These buffers, if used for communication operations, do not need to be registered, but can be used immediately as a source or a target for PIO or DMA zero-copy style operations.

For allocation memory via MPI, two thresholds are defined: the minimum size of a memory request to be served from SCI segments, and the minimum size of a request for which a separate SCI segment will be created. Transfers below the first threshold would not benefit from zero-copying because they would be transferred using the eager protocol. Allocations below the second threshold will be served from a single SCI segment, serving as a „buffer pool“. Allocations above this threshold will be served by creating a distinct local SCI segment. If the available SCI resources are insufficient, a standard memory allocation of non-shared memory will be performed.

By the means of an MPI Info handle, a memory request to `MPI_Alloc_mem()` can be further specified by the attached attributes (and their possible key values). Two attributes are recognized:

- `must_be_shared` (no key value)*:* The request shall fail if it can not be served from a regular SCI segment.
- `must_be_aligned` (with key value): The starting address returned for this request must be aligned according to the supplied key value. A key value of zero indicates that the MPI library should align the memory according to its own requirements (SCI-MPICH will return page-aligned memory in this case).

C.3 Replacement of `malloc()` and `free()`

Using the functions described above may require modifications of the source code. A transparent support can be achieved by replacing the `malloc()` and `free()` functions of the C-library with functions that try to allocate physically contiguous memory. For MPI applications, this can easily be achieved with definitions in the `mpi.h` include file which redirect calls to `malloc()` and `free()` to `MPI_Alloc_mem()` and `MPI_Free_mem()`, respectively. This has the advantage that applications only need to be recompiled, not edited. On the other hand, this approach may be sub-optimal because the limited resource of physically contiguous memory may be wasted for user memory that will never serve as a MPI send or receive buffer. A more flexible approach, in combination with the technique described in the next chapter, will be to use normal memory allocation and let these buffers be registered for SCI access afterwards (if necessary). But also for this case, a redirection through the MPI allocation functions makes sense to allocate page-aligned memory.

D. *Caching and Lazy-release of SCI Segments*

As illustrated above, the performance of zero-copy protocols benefits from leaving out intermediate copies, but suffers from the overhead of the registration of local memory and the import and mapping of remote memory. The techniques described in chapter III.C may help, but do not have effect for the general case and are not transparent to the application.

Generally, for each zero-copy send operation, two additional latencies t_{import} will $t_{un\text{-}import}$ occur at the sender process as it needs to import and un-import the receive buffer. The duration of these latencies depends on the type of the remote SCI segment (regular or user-allocated) and whether the buffer is mapped into the address space. If the receive buffer is mapped into the local address space, the import latency scales with the size of this buffer (see figure 4). This effect does also occur if the remote SCI segment is a registered user-buffer [4]. The overhead for registering a local segment or connecting to a remote regular SCI segment is significantly lower. Nevertheless, the non-negligible duration of these operations has a negative influence on the effective bandwidth.

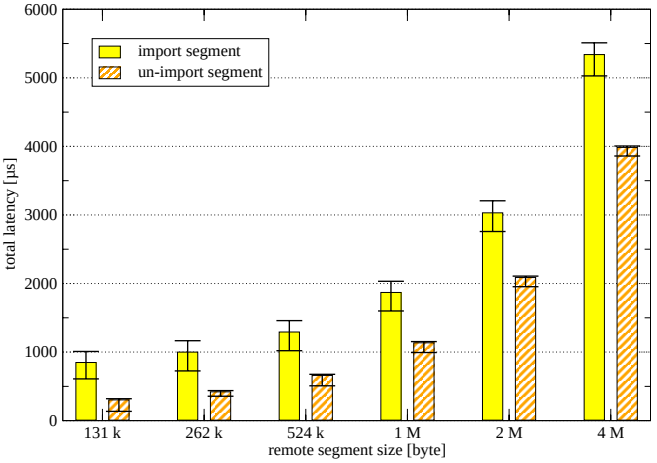

Fig. 4. Latency for importing and un-importing remote memory when it is mapped into the local address space

Because communication is often performed using the same memory buffers (but not necessarily by means of persistent communication) more than once throughout the execution of an application, it is desirable that a registration or an import of such buffers needs to be performed only once. This behavior could be achieved by simply not de-registering or disconnecting a memory buffer after the communication has completed. However, since the SCI resources are limited (local SCI memory size, shared memory limits, ATT entries on the PCI-SCI-Adapter, SISCI descriptors and others), this technique may well lead to the situation where a requested registration or connection can not be performed, possibly leading to performance reduction or even communication deadlock (if the request is related to a communication operation which can not be performed by other means).

For this reason, we introduced a software layer into SCI-MPICH which caches the requests for operations on local and remote SCI segments (abstracted as *SMI shared regions* from SCI-MPICH). This layer provides the required services for local

and remote SCI segment operations via related `acquire()` and `release()` functions. Internally, a *cache entry* is assigned to each SCI segment which contains all relevant information, including access counters for a displacement strategy and an *in-use* counter. Thus, a release request is not directly translated into a SISCI operation, but does only decrement the *in-use* counter of the related cache entry which has been increased by a preceding *acquire()* request. An *acquire()* request related to a SCI segment which has been used before and is still in the cache (or even is currently in use) can be efficiently satisfied by returning a reference to the related SCI segment.

The cache displacement strategy comes into operation when an *acquire()* request can not be served from the cache, and the related SISCI operations fail due to resource shortage. The cache scheduler than tries to free resources by deallocating SCI segments with an in-use counter of 0 until the request can be satisfied (or no more SCI segments can be deallocated). To ensure consistency between the processes concerning the withdrawal of exported segments, the segment event callback mechanism offered by SISCI is used. This way a remote process is informed if a segment to which he is connected is withdrawn, and can synchronize its local segment resources accordingly.

Different scheduling strategies for the cache displacement can be employed, like *least-recently-used* (LRU), *least-frequently-used* (LFU), b*est-fit* (based on the SCI segment size) or *random*. Also, a multi-level strategy may be used. Currently, we have implemented the LRU strategy. Another strategy, named *immediate*, which does not cache at all, but immediately deallocates each SCI segment if its *in-use* counter becomes 0, is used to evaluate the performance impact of the lazy-release technique.

IV Performance Evaluation

Before evaluating the results of the benchmarks for different protocol implementations, it is necessary to determine the key performance characteristics of the evaluation platform. This platform is a cluster of 8 dual-SMP-nodes running Linux 2.4.4[1]. Each node hosts 2 Pentium-III CPUs (800MHz, 8kiB 1st-level cache, 256kiB 2nd-level cache) and 512MB RAM (in 2 256MB PC133-DIMMs) on a Supermicro 370DLE mainboard which is equipped with a Serverworks ServerSetIII LE chipset. The interconnect is made of one Dolphin ICS PCI-SCI adapter running in a 64-bit 66MHz PCI slot per node. All 8 PCI-SCI adapters are connect in a single ringlet. Figure 5 shows the relevant bandwidth values for the different kinds of memory-to-memory transfers which are possible on this platform:

- *memcpy() local-local:* copying between two local non-shared memory regions using memcpy()
- *memcpy() SCI-local:* copying between a local SCI-shared and a local non-shared memory region using memcpy()
- *PIO write SCI:* copying from a local non-shared to a remote shared memory region using an optimized copy function
- *DMA write SCI:* copying from a local non-shared to a remote regular SCI region using the DMA engine of the PCI-SCI adapter.

1. The benchmarks have been run with non-SMP-Kernels due to a SCI driver or kernel problem with DMA transfers between registered user-allocated buffers.

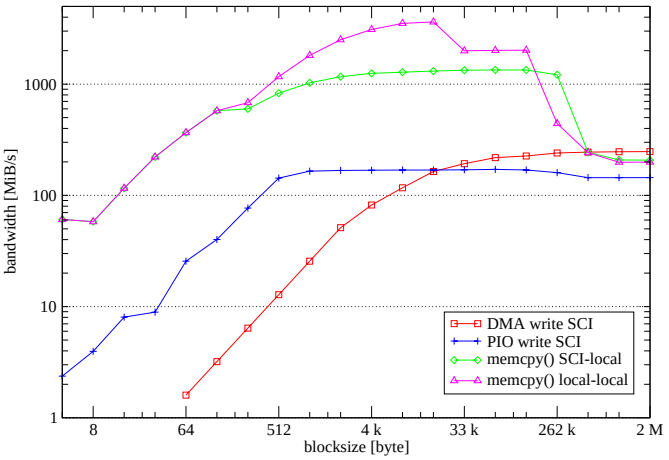

Fig. 5. Memory transfer bandwidth values on evaluation platform

transfer type	peak	at	sustained
local-local	**3624**	**16kiB**	**199**
local-SCI	**1343**	**64kiB**	**207**
SCI-PIO	**171.2**	**64kiB**	**144.3**
SCI-DMA	**247.6**	**inf**	**247.6**

Tab. 1. Key memory transfer performance values

Measuring memory bandwidth can be done in a variety of ways. For this evaluation, we performed a big number of `memcpy()`-style operations on a pair of allocated buffers. This leads to hot caches, but is similar to typical MPI communication scenarios where the message buffers have recently been touched and may thus reside in the cache. All non-DMA transfer techniques show a distinctive dependency on the sizes of the installed 1st- and 2nd-level-caches, as indicated in table 1 and degrade drastically for buffers sized more than twice as big as the 2nd level cache. This leads to DMA being the fastest way of copying more than 512kiB of data between two remote or local memory locations.

A. Rendez-Vous Protocol Variants

Regarding the different data transfer modes (PIO and DMA), the synchronous and asynchronous variants and the possible types of source (local) and target (remote) buffers with respect to their accessibility via SCI (not accessible via SCI, regular SCI segment, or user-allocated registered SCI segment), a number of variants for the rendez-vous protocol in SCI-MPICH is available and might implicitly (determined by the SCI-MPICH library based on resource characteristics and availability) or explicitly (determined by the user via the configuration of SCI-MPICH and the use of non-blocking MPI communication calls) be used during the execution of an MPI application. Below, we describe these variants that we will evaluate later on.

A.1 Synchronous PIO-based 1-way Copying (*s-PIO-1*)

This protocol requires two copy operations:

- *Lns-Rs:* from the local send buffer to the remote memory pool for incoming messages (performed by the sender via local-

nonshared to remote-shared memcpy operation)

- *Ls-Lns:* from the local memory pool for incoming messages to the local receive buffer posted by the application (performed by the receiver via local-shared to local-nonshared memcpy operation)

However, these two operations are pipelined very efficiently, and thus the effective bandwidth is nearly as high as the lower bandwidth of *Lns-Rs* and *Ls-Lns* (which is usually *Lns-Rs*).

A.2 Asynchronous DMA-based 2-way Copying (*a-DMA-2*)

Without the possibility of registering user-allocated buffers, the DMA protocol requires three memory transfer operations:

- *Lns-Ls:* from the local send buffer to the local SCI shared memory buffer which will be used as DMA source (performed by the sender via memcpy operation)
- *Ls-Rs*: from the local DMA source to the remote DMA target buffer (via DMA-write by the sender)
- *Ls-Lns:* from the local memory pool to the local receive buffer.

A.3 Asynchronous DMA-base 1-way Copying (*a-DMA-1*)

By registering the send buffer with the SCI driver (or using a send buffer allocated from an SCI memory area, as in this case), the memory transfer operation *Lns-Ls* (A.2) can be omitted. Instead, DMA directly uses the send buffer as it's source. The destination buffer, however, is not imported, which means that the bandwidth consists of the consecutive bandwidths *Ls-Rs* (DMA) and *Ls-Lns* (PIO).

A.4 Asynchronous DMA-based 0-Copying (*a-DMA-0*)

The DMA-based zero-copy protocol safes even two copy operations when compared with the protocol as described in A.2. The two required locking operations for user-buffer registering can fully overlap as illustrated in figure 3 when the small latency of the initial control message is neglected. However, the import and un-import latency of the remote segment can not be hidden (in case they do actually occur).

B. *Ping-pong Bandwidth*

To validate the protocol implementations, we measured ping-pong bandwidth values between two processes on different nodes using the `mpptest` [29] benchmark. We compare the existing rendez-vous message transfer protocols in SCI-MPICH which make no use of zero-copying techniques with the new zero-copy protocols.

We modified the benchmark to be able specify the desired type of memory allocation. We tested with communication buffers allocated via `MPI_Alloc_mem()`, which means that the communication buffers are persistently mapped into the SCI address space, or via `malloc()`, in which case the buffers need to be registered to serve as DMA source or target regions.

If the communication buffers were allocated via `MPI_Alloc_mem()`, SCI-MPICH can always use the a-DMA-0 protocol without registering the buffers because they are allocated from *regular* SCI segments (*a-DMA-0-regular*). The variant with `malloc()`-allocated buffers is labeled *a-DMA-0-user*.

The resulting performance of all the evaluated protocol variants is depicted in figure 6.

To evaluate the effect of the SCI segment caching, we have

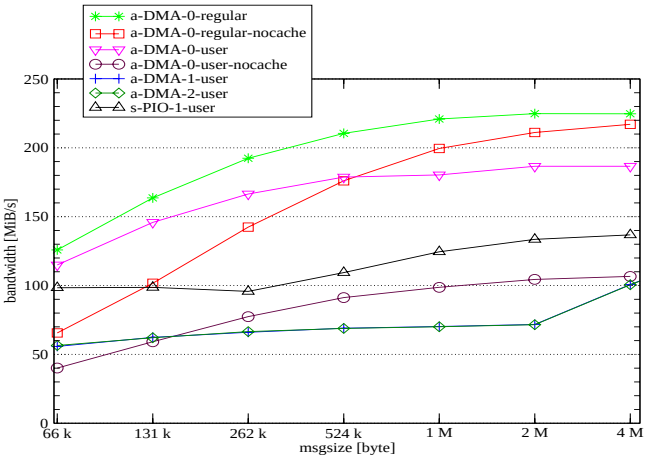

Fig. 6. Ping-pong bandwidth for different rendez-vous protocol variants

performed another series of ping-pong benchmarks in which we varied the number of messages exchanged in one run. The results are given in table 2 for different message sizes.

reps	segment caching	segment type	128kiB	512kiB	2MiB
1	no	regular	**78.44**	**154.80**	**203.48**
		user	**39.77**	**57.59**	**64.32**
	yes	regular	**81.83**	**158.27**	**205.65**
		user	**43.13**	**59.87**	**64.89**
10	no	regular	**97.21**	**172.43**	**210.28**
		user	**55.81**	**85.92**	**98.38**
	yes	regular	**145.66**	**204.05**	**222.79**
		user	**118.50**	**149.10**	**158.22**
100	no	regular	**100.45**	**175.01**	**211.08**
		user	**57.58**	**89.66**	**103.73**
	yes	regular	**159.41**	**210.37**	**224.94**
		user	**144.20**	**177.07**	**185.17**

Tab. 2. Effect of SCI segment caching on *a-DMA-0* transfers on ping-pong bandwidth (MiB/s)

C. *Overlapping Communication and Computation*

The effect of overlapping communication with computation when using non-blocking MPI communication operations with SCI-MPICH has already been described in [28]. We will compare the results of this implementation with our current technique to illustrate the performance that zero-copy via DMA delivers for asynchronous communication.

Figure 7 shows the pseudo code for a basic benchmark *overlap* to simulate overlapping of communication and computation. For all synchronous protocols, the effective MPI communication for rendez-vous messages (messages bigger than 32kiB) will take place in `MPI_Wait()`, sequentializing communication and

```
latency = MPI_Wtime()
if (sender)
   MPI_Isend(msg, msgsize)
   while (elapsed_time < spinning_duration)
      spin (with multiple threads)
   MPI_Wait()
else
   MPI_Recv()
latency = MPI_Wtime() - latency
```

Fig. 7. Pseudo Code for *overlap* benchmark

computation. The asynchronous protocols do transfer the data without any MPI library activities of the application thread, allowing for overlapping of computation and communication. The *spinning* can be performed by a selectable number of threads, and in two different ways:

- *FIXED*: spinning on a single variable for a fixed period of time. This keeps the CPU busy, but incurs no memory accesses.
- *DAXPY*: performing a specified number of DAXPY-type operations ($y[j] = A \cdot x[j] + y[j]$) on vectors of doubles with the length of the message which is transferred. This will also stress the memory subsystem.

We have performed this benchmark for the s-PIO-1, a-DMA-0-user and a-DMA-0-regular variants of the rendez-vous protocol on two single-CPU nodes with one spinning thread, transferring messages of different sizes with *FIXED* and *DAXPY* spinning. We also ran the benchmarks without transferring any messages, which shows the pure busy period which the CPU is spinning (labeled *perfect hiding* in the plots).

The results are depicted in figure 8 and indicate that the overlapping does become nearly perfect if the duration of the computation period passes the break-even point. The break-even point, which is the required length of the computation period to make a zero-copy DMA protocol more efficient than synchronous PIO, is the difference in the latency of these two protocols. For messages from 64kiB length up, the transfer with a-DMA-0-regular is always more efficient than s-PIO-1. For a-DMA-0-user, the break-even point is reached with about 30k DAXPY operations, which is equivalent to about 200µs on this platform.

The overlapping efficiency of the overlapping is derived from the duration of the computation period only (l_{busy}), the shortest latency of a message transmission only (l_{msg}) and the duration of the combined communication and computation operation ($l_{overlap}$). Thus, the efficiency $\varepsilon_{overlap}$ of a given l_{busy} can be expressed as

$$\varepsilon_{overlap}(l_{busy}) = 1 - \frac{l_{overlap} - l_{busy}}{l_{msg}} \quad (3)$$

The efficiency for the three test setups of figure 8 for the *saturated* case (this means where $l_{msg} < l_{busy}$ for all protocol variants) is given in table 3. The efficiency for the synchronous protocol is always close to 0, indicating that no overlapping takes place. For the asynchronous, DMA based protocols, the efficiency is considerably higher, and getting closer to 1 the longer the message to be transferred is. It is also visible that the

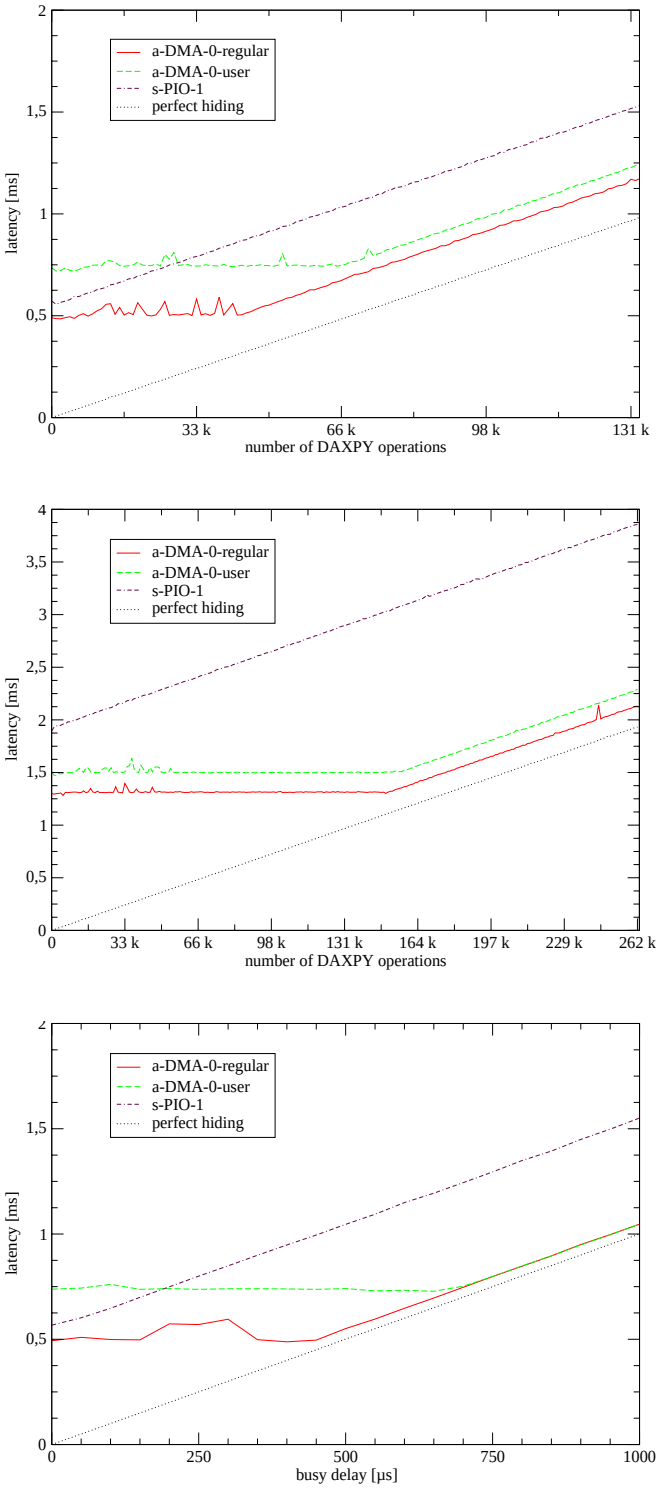

Fig. 8. Overlapping of communication and computation (single thread on a single-CPU system) with different protocols:
top: DAXPY spinning with 64kiB message
middle: DAXPY spinning with 256kiB message
bottom: FIXED spinning with 64kiB message

stress on the memory bus using DAXPY spinning does increase the startup latency for DMA, too, if the efficiency for the 64kiB cases are compared. The bandwidth does not necessarily decrease, as is visible by the increasing efficiency for 256kiB message transfers with DAXPY spinning.

message size	busy type	l_{busy}	protocol variant	l_{msg} [ms]	$l_{overlap}$ [ms]	$\varepsilon_{overlap}$
64 kiB	DAXPY	**128ki**	**a-DMA-0-regular**	**0.490**	**1.170**	**0.581**
		equiv.	**a-DMA-0-user**	**0.735**	**1.227**	**0.643**
		0.965 ms	**s-PIO-1**	**0.572**	**1.512**	**0.043**
256 kiB	DAXPY	**256ki**	**a-DMA-0-regular**	**1.300**	**2.132**	**0.845**
		equiv.	**a-DMA-0-user**	**1.506**	**2.289**	**0.762**
		1.931 ms	**s-PIO-1**	**1.895**	**3.856**	**- 0.015**
64 kiB	FIXED	**1 ms**	**a-DMA-0-regular**	**0.493**	**1.047**	**0.904**
			a-DMA-0-user	**0.738**	**1.047**	**0.936**
			s-PIO-1	**0.567**	**1.551**	**0.028**

Tab. 3. Overlap efficiency for different protocol variants, message sizes and spinning types.

D. Application Performance

The IS benchmark of the NAS Parallel Benchmark Suite [23] is an implementation of a parallel bucket-sort with integer numbers as keys. Its communication is dominated by exchange of large messages using `MPI_Alltoallv()`, which in turn uses asynchronous communication via `MPI_Isend()` and `MPI_Irecv()`. The IS benchmark can be run with different data set sizes (*classes)* of which we took the classes W and A. The vector size given in table 4 is the size of the data blocks which are exchanged in the `MPI_Alltoallv()` operation, of which different message sizes result in dependency of the number of processes used. The duration of a single operation and percentage of the accumulated times on the total execution time (for s-PIO-1 protocol variant) is also given. All numbers are based on the two cases of running the benchmark with 4 processes.

Class	vector size [MiB]	procs	msg size [kiB]	Alltoallv duration [ms]	% of total time
W	1	4	256	**16.363**	**34.6**
A	8	4	2048	**123.921**	**36.2**

Tab. 4. Communication characteristics of IS benchmark classes

Problems with the alignment of the messages which could not be resolved in time hindered us to perform the complete benchmark. We therefore can only give the duration of equivalent `MPI_Alltoallv()` operations performed with the a-DMA-0 protocol (on both, regular and user SCI segments for communication) and deduce the impact on the benchmark performance related to these values. These results are show in table 5 and indicate a potential performance improvement of 20%.

V Summary & Outlook

Our results show that zero-copy DMA can improve performance of MPI on SCI-connected clusters, delivering a high peak bandwidth with very little CPU load. Also, the CPU caches are not touched by DMA as opposed to PIO, but the effects of this are subject to further studies.

equivalent IS class	procs	regular [ms]	speedup	user [ms]	speedup
W	4	**7.578**	**1.22**	**9.617**	**1.16**
A	4	**52.415**	**1.26**	**63.957**	**1.21**

Tab. 5. `MPI_Alltoallv()` performance with a-DMA-0-regular protocol and predicted effect on IS performance

The most obvious problem of zero-copying with SCI, the high latency for registering, importing and un-importing SCI shared memory regions could be solved with two presented techniques:

- *remote DMA* into non-mapped remote SCI segments eliminates the need for costly mapping operations in most cases
- *caching SCI segments* that were established for a zero-copy operation

Both techniques generally increase the effective bandwidth, sometimes more than doubling it. This will improve the performance of communication-bound applications which transfer messages of 256kiB and more, as it has been deduced for the IS benchmark. The effects of other cache replacement strategies than the implemented *least recently used* will be studied in the near future.

Next to the improved bandwidth, the overlapping of computation and communication is more efficient than ever on this platform: fast data transfer do only cost very little CPU cycles and come practically for free if the overlapping is perfect. Application programmers need to become aware of this feature to consider it in the design of parallel algorithms.

A number of other possible performance parameters, like the use of persistent communication operations, better aligned local memory for registering could not be evaluated in the scope of this paper. Also, the robustness of the techniques need to be increased to run any application without modifications. The good performance of the DMA transfers make it desirable to use them for synchronous transfers, too.

REFERENCES

[1] Message Passing Interface Forum: *MPI: A message-passing interface standard.* International Journal of Supercomputing Applications, 8(3/4), 1994.
URL: http://www.mpi-forum.org/docs/docs.html

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

[3] Hong Ong and Paul A. Farrell: *Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network*. In Proc. of 4th Annual Linux Showcase & Conference, USENIX 2000, Atlanta, USA, October 2000.

[4] Friedrich Seifert, Joachim Worringen and Wolfgang Rehm: *Using Arbitrary Memory Regions for SCI Communication.* In Proc. of SCI-Europe 2001 conference. Trinity College, Dublin, October 2001.

[5] Th. v. Eicken, A. Basu V. Buch, W. Vogels: *U-Net: A User-Level Network Interface for Parallel and Distributed Computing.* In Proc. 15th ACM Symposium on Operating System Principles. Copper Mountain, Colorado, December 3-6, 1995.

[6] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos N. Damianakis, Kai Li: *Shrimp Project Update: Myrinet Communication.* IEEE Micro vol. 28 (1), pp. 50-52, January/February 1998

[7] Compaq, Intel and Microsoft Corporations. *The Virtual Interface Specification. Version 1.0.* Dec 16, 1997.

[8] 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

[9] Giuseppe Ciaccio and Gio anni Chiola: *GAMMA and MPI/GAMMA on Gigabit Ethernet*. J. Dongarra et al. (Eds.): EuroPVM/MPI 2000, LNCS 1908, pp. 129 136, 2000. c Springer-Verlag Berlin Heidelberg 2000

[10] Patrick Geoffray, Loic Prylli, Bernard Tourancheau: *BIP-SMP: High Perormance Message Passing over a Cluster of Commodity SMPs.* In Proceedings of Supercomputing '99.

[11] Rossen Dimitrov and Anthony Skelljum: *Efficient MPI for Virtual Interface (VI) Architecture.* In Proc. of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications. Las Vegas, Nevada, USA, June 1999, Vol.6, pp: 3094-3100.

[12] Sven Schindler, Wolfgang Rehm, Carsten Dinkelmann: *An optimized MPI library for VIA/SCI cards* In Proc. of the Asia-Pacific International Symposium on Cluster Computing (APSCC'2000), held in conjunction with the Fourth International Conference/Exhibition on High Performance Computing in Asia-Pacific Region (HPCAsia2000), May 14-17, 2000, Beijing, China. Volume II, pp. 895-903.

[13] Francis O'Carroll, Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa, *The Design and Implementation of Zero Copy MPI Using Commodity Hardware with a High Performance Network*, ACM SIGARCH ICS'98, pp. 243--250, July 1998.

[14] Toshiyuki Takahashi, Francis O'Carroll, Hiroshi Tezuka, Atsushi Hori, Shinji Sumimoto, Hiroshi Harada, Yutaka Ishikawa, and Peter H. Beckman. *Implementation and Evaluation of MPI on an SMP Cluster.* In Proc. Parallel and Distributed Processing IPPS '99 / SPDP Workshops, LNCS 1586, Springer-Verlag, April 1999.
URL: http://pdswww.rwcp.or.jp/db/paper-E/1999/1999.html

[15] Karim Ghouas, Knut Omang, Hakon Bugge: *VIA over SCI - Consequences of a Zero Copy Implementation, and Comparison with VIA over Myrinet.*In Proc. Workshop on Communication Architecture for Clusters 2001, in conjunction with Int'l Parallel and Distributed Processing Symposium (IPDPS '01), San Francisco, April 2001.
URL: http://www.ifi.uio.no/~knuto/Publications/

[16] Scali AS: *Scali MPI - ScaMPI.* URL: http://www.scali.com

[17] Richard Gillett, Richard Kaufmann: *Using the Memory Channel Network.* IEEE Micro vol. 17 (1), January/February 1997.

[18] James V. Lawron, John J. Brosnan, Morgan P. Doyle, Seosamh D.Ó Riordáin, Timothy G. Reddin: *Building a High-performance Message-passing System for MEMORY CHANNEL Clusters.* Digital Technical Journal, vol. 8(2), October 1996

[19] Compaq HPTC Info Center: Compaq MPI. Last update from June 2001
URL: http://www.compaq.com/hpc/software/dmpi.html

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

[21] Jenwei Hsieh, Tau Leng, Victor Mashayekhi and Reza Rooholamini: *Architectural and Performance Evaluation of GigaNet and Myrinet Interconnects on Clusters of Small-Scale SMP Servers.* In Proceedings of Supercomputing 2000.

[22] Rossen Petkov Dimitrov: *Overlapping of Communication and Computation and Early Binding: Fundamental Mechanisms for Improving Parallel Performance on Clusters of Workstations.* Ph.D. thesis, Mississippi State University, U.S.A., May 2001

[23] D.H. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo and M. Yarrow: *The NAS Parallel Benchmarks 2.0*, NASA Technical Report NAS-95-020, NASA Ames Research Center, December 1995
http://www.nas.nasa.gov/Software/NPB

[24] Friedrich Seifert, Wolfgang Rehm: *Proposing a Mechanism for Reliably Locking VIA Communication Memory in Linux*. In Proc. 1st IEEE International Conference on Cluster Computing CLUSTER2000, Nov 28 - Dec 1, 2000, Chemnitz, Germany.
URL: http://www.tu-chemnitz.de/informatik/RA/papers/p97/papers.html

[25] M.Dormans, K.Scholtyssik, Th.Bemmerl: *A Shared Memory Programming Interface for SCI Clusters*, In *SCI: Scalable Coherent Interface*, Edited by H.Hellwagner and A.Reinefeld, LNCS 1734, Springer, 1999

[26] TOP500 Supercomputer Sites - Architecture of the fastest machines is distributed memory. URL: http://www.top500.org

[27] Joachim Worringen, Th. Bemmerl: *MPICH for SCI-connected clusters.* In Proc. SCI Europe '99, helt in conjunction with EuroPar '99, pp. 3-11, Toulouse, France, September 1999
URL: http://www.lfbs.rwth-aachen.de/users/joachim/publications (paper),
http://www.lfbs.rwth-aachen.de/users/joachim/SCI-MPICH (software)

[28] Joachim Worringen: *SCI-MPICH - The Second Generation.* Proc. SCI Europe 2000 (conference stream of EuroPar 2000), pp. 10-20, Munich, Germany, August 2000
URL: http://www.lfbs.rwth-aachen.de/users/joachim/publications

[29] William Gropp and Ewing Lusk: *Reproducible Measurements of MPI Performance Characteristics*., Proc. PVMMPI'99.
URL: http://www-unix.mcs.anl.gov/~gropp/papers.html (paper),
http://www-unix.mcs.anl.gov/mpi/mpptest (software)