Exploiting Transparent Remote Memory Access
for Non-Contiguous- and One-Sided-Communication

Joachim Worringen, Andreas Gäer, Frank Reker and Thomas Bemmerl

Abstract

The availability of an implementation of the Message Passing Interface (MPI) is essential for each interconnect designed for communication in HPC clusters. Using the open-source implementation of MPI, MPICH, and creating a communication device for it based upon the low-level communication libraries of the interconnect, this goal can be achieved quiet easily. However, optimizing the resulting MPI implementation to make maximum benefit of the characteristics of the interconnect is a more complex task. This paper presents two of the most recent optimizations in SCI-MPICH, an MPICH variant for the SCI interconnect, which make use of the global shared memory provided by this interconnect: efficient communication with non-contiguous MPI datatypes and one-sided communication according to the MPI-2 standard. We show that the transparent low-latency communication characteristics of SCI provides these techniques an excellent platform.

Keywords: MPI-2, SCI, remote memory access, non-contiguous datatypes, one-sided communication

1. Introduction

The dominant programming model for parallel scientific and technical computing is message passing, using the *Message Passing Interface* (MPI) standard [1][2]. Therefore, every HPC cluster platform, of which the interconnect is a crucial component, needs to offer an implementation of MPI for it. This implementation should make optimal use of the characteristics of the interconnect. This goal can usually not be achieved when using a legacy protocol like TCP/IP which includes a very high software overhead, but requires the adaption of the MPI implementation to the interconnect-specific low-level protocols.

Examples for such potential optimizations are utilizing hardware-supported broadcast messages (like the Quadrics interconnect [3][4]) or moving parts of the MPI protocols onto the interconnect adapter if it is equipped with a general-purpose processing unit (as found on the Myrinet adapter boards [5], and also on Quadrics). These are options for „smart" interconnect adapters. However, for a „dumb", but nevertheless efficient interconnect like shared memory, other options arise. The *Scalable Coherent Interface* (SCI [6]) provides a global shared memory space between all nodes of a cluster. We have exploited this shared memory space,

which can be used for read and write accesses with very low latency and high bandwidth write-access, to create an optimized MPI implementation named SCI-MPICH [7], part of a cross-platform MPI implementation named MP-MPICH [8]. In this paper, we present two new techniques used in SCI-MPICH which make use of the special characteristics of SCI and set it apart from other MPI implementations that we have evaluated.

The first technique is an optimization for sending non-contiguous datatypes in MPI. We have implemented an efficient algorithm which allows to omit process-local intermediate copy operations on the data by exploiting low-latency communication via SCI. The local copy operations are usually required to transform the disjoint data blocks of a non-contiguous datatype into a contiguous block of bytes in memory (and vice versa) to transmit them via the network. The other technique is the way that the MPI-2 one-sided operations are implemented in SCI-MPICH, which strives to give the best performance possible for each setup of a one-sided communication operation.

The next section gives an overview on the interconnect-related issues when implementing MPI for SCI-connected cluster. Section III presents the *direct_pack_ff* algorithm in SCI-MPICH which allows for efficient communication with non-contiguous datatype. After this, we present the implementation of one-sided communication in SCI-MPICH. We set our results in relation to related work done in this field in section V and summarize our findings in section VI.

2. MPI via SCI

Communication between nodes in an SCI-connected cluster can be performed via load and store operations issued by the CPU, directed at remote memory segments which are transparently mapped into the process' address space. This means that an MPI implementation on top of SCI is basically a shared-memory MPI (in fact, the initial version of SCI-MPICH was based on a native shared-memory device). The performance charts in figure 1 show the memory performance of SCI remote memory for a typical current-generation cluster node[1]. It shows the latency and bandwidth of PIO transfers[2] (transparent remote memory access by the CPU) and DMA transfers (performed by a DMA engine on the PCI-SCI adapter). However, there are certain differences

---

1. For this work, Dual Pentium-III 800MHz, ServerWorks ServerSet III LE based motherboard with 64bit/66MHz PCI, Dolphin D330 PCI-SCI Adapter are used.
2. The bandwidth reduction for PIO-transfers beyond 128kiB is caused by the limited local memory bandwidth and doesn't show up for chipsets with higher memory performance like the HE variant of the ServerSet III chipset.
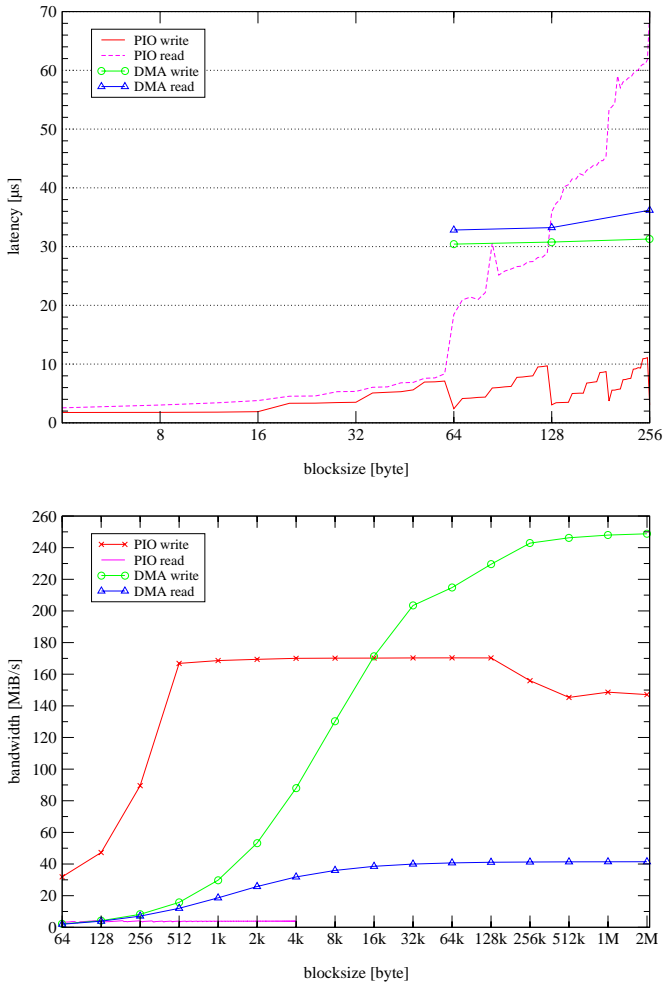
Figure 1. Raw SCI communication performance (intra-node communication) *top:* **small data latency** *bottom:* **bandwidth**

between intra-node and SCI shared memory next to the access latency which make a number of adaptions necessary to achieve good performance:

- The performance of remote reads is only a fraction of the write performance due to the fact that the CPU stalls until requested data is available, while writing to remote memory is performed in a write-and-forget fashion. However, remote-reads of small data units still have a relatively low latency.
- Achieving maximum performance for remote writes requires accessing strictly sequential, contiguous, ascending addresses to make best use of the *stream buffers* on the PCI-SCI adapters. These buffers gather consecutive transactions and can in turn generate bigger SCI transactions which have a higher efficiency. Also, for small accesses, aligning data access granularity to SCI transaction sizes delivers higher bandwidth.
- The fact that data is written out by the CPU does not incur that it has arrived at the receiver in this moment of time. It may still be buffered in the network. Store-barriers are required to ensure complete delivery of all data written at a certain moment of time.
- Due to retried transfers after a transmission error, it can not be

guaranteed that data appears at the receiver in the same order as it was written on the sender. Again, a store-barrier ensures that the data has arrived completely the moment the calling process returns from the barrier.

- In a cluster environment with physically distributed nodes, the SCI interconnect is based on cable connections. Thus, although a shared address space is provided, SCI is still a network in which single nodes may fail or physical connections may be disturbed (i.e. by plugging a cable). This makes a connection monitoring and transfer checking necessary, which is not required for intra-node shared memory communication.

All these issues make an efficient implementation of MPI on top of SCI more complex than an MPI for intra-node shared memory, although the communication principles and basic architectures are very similar [7]. On the other hand, SCI offers more than intra-node shared memory, i.e. DMA transfers and most of all, easier scalability and thus cost-efficiency for the kind of applications covered in this paper.

Throughout the paper, all SCI-related performance measurements are performed on a cluster of 8 of such nodes connected via a single SCI ringlet.

3. Communication of Non-contiguous Data

In typical MPI applications, the data that the processes do exchange is part of the global data structures representing the problem domain to be solved. In many cases, i.e. if the global data structure is a multi-dimensional array, the data to be exchanged is not a single block of memory (*contiguous data*), but is made up of multiple blocks of contiguous data, separated by gaps (*non-contiguous data*). A typical example of such an application are ocean models in which the decomposition of the simulation volume is done along the two horizontal dimensions [9]. For the exchange of the boundary data, this leads to strided or even double-strided data (see figure 2).
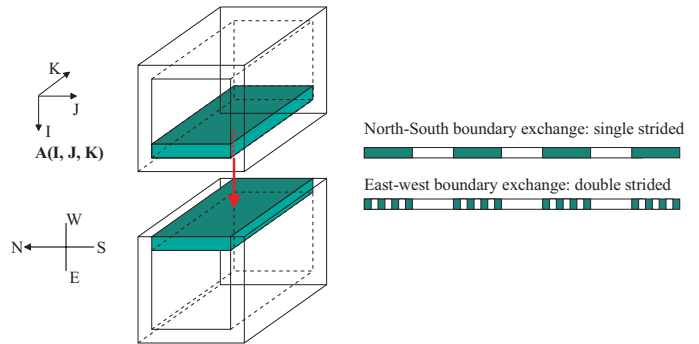
Figure 2. Decomposition of on ocean model and resulting non-contiguous data for boundary exchanges (from [10])

Different ways exist to transfer such non-contiguous data via messages in MPI:

1. Send one message for each contiguous block of data.
2. Copy all contiguous data blocks into a separate buffer to build one single block of contiguous data to be send with one message (this operation is called *packing)*. The receiver needs to *unpack* the data again.

3. Define an *MPI datatype* which represents the non-contiguous data and send a single message using this datatype.

Technique 1 reduces the communication performance because the message startup costs occur for each message. Technique 2 requires at least one additional copy operation and thus reduces the communication performance, too.

With technique 3, the decision on how to transmit the data is made by the MPI library which thus can choose the optimal way. We will show how MPI datatypes can be constructed and explain the generic solution to send non-contiguous data defined by an MPI datatype. We will then present our new technique to perform this task and evaluate the effects it has on the communication performance.

3.1 MPI Datatypes

All data specifications in MPI are based on datatypes. MPI provides basic datatypes which are essentially the datatypes which exist in the C and Fortran programming languages. To represent more complex data structures (like *structs* in C) or to group various data elements into one datatype to simplify communication, MPI supports user-defined datatypes. These datatypes can be defined using a variety of MPI API functions. All these functions basically do the same thing; they are designed to facilitate the mapping of common data arrangements in an application to a new MPI datatype. Such new MPI datatypes may in turn be used to create other datatypes. Figure 3 gives an example for a non-contiguous datatype which is constructed as a vector of a structure, which in turn is made up from an `integer`, an array of `char` and two gaps. Each type is specified by combination of the parameters *blocklen*, *count*, *extend* and *stride* (see [1]). Before a datatype can be used for communication, it needs to be committed, which informs the MPI library that this datatype will be used for communication. It is at this moment that the library may generate an optimized representation of the datatype. The internal representation of a datatype is up to each MPI implementation. Usually, a tree-based representation is choosen. Figure 3 also illustrates the internal tree-like datatype representation of MPICH [11].
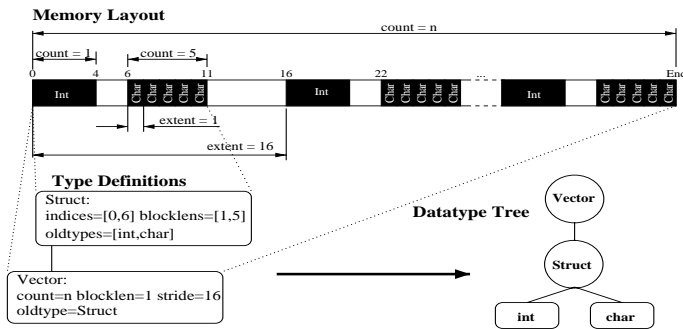

Figure 3. Datatype representation in memory and internal tree structure

3.2 Non-Contiguous Data Transmission

Because many communication interfaces do only offer the transmission of contiguous data blocks, defined by a memory address and the length, a generic technique to transmit non-contiguous data is to pack data before transmitting it, and unpacking it when it has arrived. Depending on the type of communication interface and the MPI implementation, this introduces one or two additional copy operations.
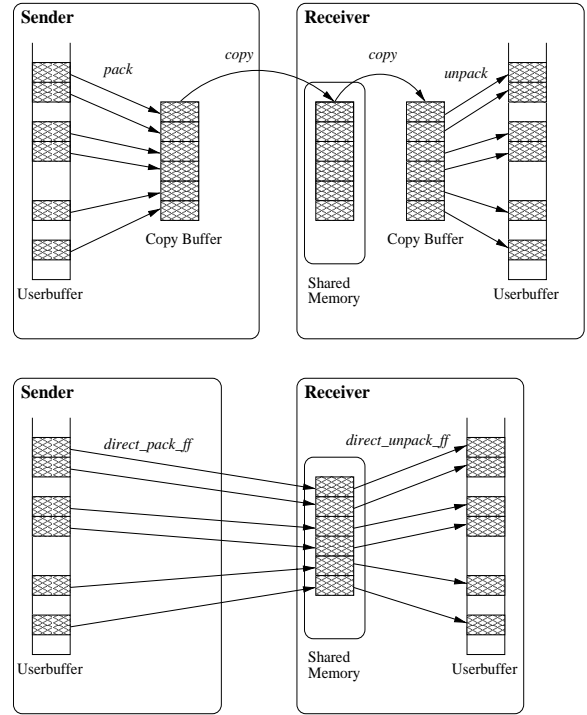

Figure 4. Transmission of non-contiguous data. *Top:* generic technique (i.e. generic MPICH). *Bottom:* direct_pack_ff technique (SCI-MPICH)

A communication interface that can transmit data blocks of arbitrary length (stream-oriented interfaces like sockets) can transmit each MPI message with a single invocation of a transfer operation. In this case, only one additional copy is needed for transmitting non-contiguous data. However, because these interfaces need to buffer internally, they are generally less efficient on high-performance interconnects. Non-buffering interfaces (like shared-memory, which is always limited in size) need to split large MPI message in parts which are transferred separately. For such an interface, two additional copy operations are required (figure 4, *top*), which may however overlap.

3.3 direct_pack_ff-Algorithm

To eliminate these superfluous copy operations, we implemented a packing algorithm that can be used to pack the non-contiguous distributed data directly into the SCI shared memory as shown in figure 4, *bottom*. This requires that the discrete pieces of the non-contiguous datatype are transferred individually into the receive buffer. With SCI, these kinds of transfers can be performed via transparent remote writes by the origin CPU with a reasonable bandwidth (see figure 1) even for small block sizes, provided they are written into a consecutive manner (see chapter II, point 2). This condition means that directly writing the data into the user buffer, even it would be exported to shared memory, would not be efficient for small block sizes.

The algorithm is derived from the 'flattening on the fly'-technique presented in [12] and thus called *direct_pack_ff*. One main requirement this algorithm needs to fulfill is the ability to pack only parts of the data starting at an arbitrary point in the structure and having no constraints about the length of the data to pack.

When examining the derived datatype structure, one may notice that a derived datatype can be interpreted as a tree with basic contiguous datatypes as the leaves. The path from the root to a specific leaf describes the repeat pattern of this basic datatype in the user-buffer. This pattern is defined by two informations on each level of the datatype tree: the *replication count* and the *extent of the data* (including a stride between items). The total size for each level is also used to speed up some operations, although it is not essential for the algorithm. With this information, it is now possible to build up a stack for each *basic block* (contiguous entities which can be transferred in one copy operation), describing the arrangement of the data in a very compact way, without losing information which can be used for the optimization of the copy process. For each of the different MPI type constructors (vector, hvector, indexed, hindexed, struct and contiguous), there is a special way to place the information on the stack.

3.3.1 Building the stack

A suitable data structure to store and access the type construction information is a list of stacks, because it can be traversed in a non-recursive manner (for a detailed description of this concept see [12]). These stacks are build up when committing the datatype, so it is not exactly 'on the fly'. But as the memory consumption of the stacks is very low, it can be tolerated for an even faster packing operation. When committing a new datatype, a list is added to the internal structure. This list represents the datatype as it holds an item for each leaf. This item consists of the contiguous size of the leaf, the position where the first block is to be found in the user buffer and the stack describing the repeat pattern.
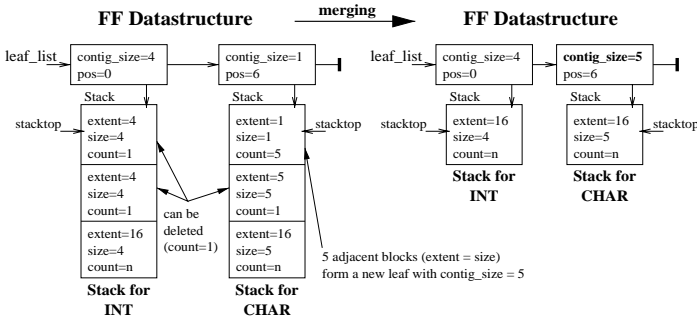

Figure 5. Internal datatype representation (of the type given in figure 3) with *direct_pack_ff*-Algorithm

After the stack for a leaf is built, it will be *merged*: it often is possible to build up larger blocks of adjacent basic blocks. Additionally, stack items with a replication count of 1 which are not the only element in the stack can be deleted as they don't represent any effective replication. Figure 5 shows the application of this algorithm to the vector-of-structs datatype given in figure 3.

3.3.2 Sending non-contiguous data

When performing a send operation with a non-contiguous

datatype, the `direct_pack_ff` function is used and the list of basic blocks is scanned. For each basic block, the corresponding stack is used to pack the data directly into SCI shared memory. There is some additional functionality for the handling of split blocks, which is not described here in detail. The top function `direct_pack_ff` is shown in figure 6.

```
/* find initial position for partial sends */
leaf = find_position(byte_offset);

/* copy the rest of a split block */
copy_split_block();

/* traverse the list of leaves */
while (sufficient space in target buffer) {
  copy_leaf_basic (leaf);
  leaf = leaf->next;
}
```

Figure 6. Top-level loop of the direct_pack_ff algorithm.

The function `find_position` is used to resume after a part of a large message block was already sent up to position `byte_offset`. It also uses the ff-stack information of the datatype and completes in a maximum time of $O(N) + O(D)$, where N is the number of basic blocks and D the maximum depth of the datatype tree.

The *direct_pack_ff*-algorithm itself is implemented in `copy_leaf_basic`. This function copies the data for one leaf in the datatype tree by evaluating the repeat pattern stored in the corresponding stack. On the receiving side, the same function is used just by swapping the direction of the copy operation. This new packing algorithm has two main advantages over the old recursive implementation. First, it replaces the time consuming repeated recursive transversal of the datatype tree by two nested loops with only simple stack (array) operations. Second, it is able to copy partial blocks of arbitrary length within the data structure. Now it is possible to pack directly into the remote shared memory, avoiding two copies into local buffers and thus speeding up communication between nodes.

It must be noted that the memory accesses of the *direct_pack_ff* algorithm are no longer performed with strictly increasing addresses for datatypes with differently sized basic blocks. To avoid cacheline thrashing in these cases, the amount of data copied in one handshake cycle of the *rendez-vous protocol* (see [7]) should be kept below the size of the 2nd level cache. This can easily be assured by setting the protocol parameters accordingly. On the other hand, the *direct_pack_ff* algorithm avoids cache pollution as it does not perform any local copy operations.

3.4 Performance Evaluation

For every communication channel with a non-zero startup latency, transfers of smaller blocks are less efficient than transfers of big blocks. To evaluate the influence of the blocksize, we would need to do tests with increasing blocksizes. The complexity of the datatype should have little influence on the performance of our optimization, since the algorithm is generic. However, we wanted to verify this, too. Therefore, we designed a micro-benchmark *noncontig* which transmits a simple single-strided vector datatype. This blocksize of this vector is increased from 8 byte (a single

`double` element) up to 128 kiB throughout the test The stride between two blocks is twice the blocksize, creating an equal-sized sequence of data and gaps in memory. The *generic* and the *direct_pack_ff* transfers are compared, with the bandwidth of an equivalent *contiguous* data transfer as reference. Each transfer transmits the same amount of data, which is 256kiB for this case.
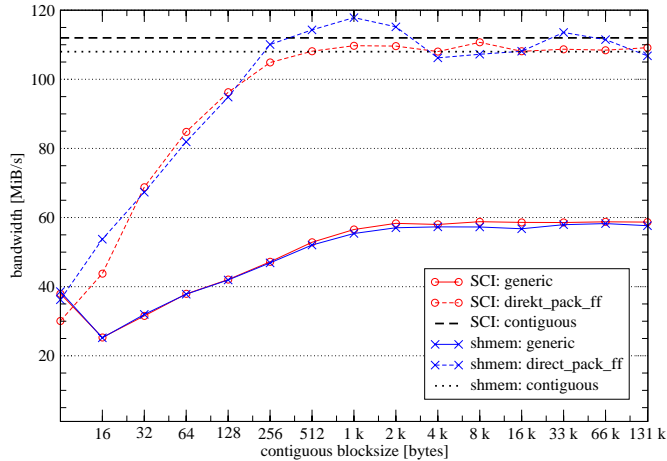
Figure 7. Performance of non-contiguous data transfers in SCI-MPICH (generic vs. *direct_pack_ff*) for inter- and intra-node communication, using SCI and shared memory

The results given in figure 7 show that the bandwidth for non-contiguous transfer using the *direct_pack_ff* transport technique approximates the bandwidth for contiguous transfers, and already reaches 90% of it for blocksizes of 128 byte. It delivers already twice the bandwidth of the generic algorithm for a blocksize of 16 bytes and above. Only for the case of 8 byte-blocksizes, the generic technique proves to be faster for inter-node communication, due to the relatively high latency of remote memory accesses with 8 byte granularity[1].

Interestingly, the performance of the non-contiguous transfer with *direct_pack_ff* via shared memory can surpass the bandwidth of the equivalent transfer of contiguous data. We have observed this not only on the Pentium-III platform used for the testing above, but also for a Sun UltraSparc II. The block sizes for which non-contiguous transfer is faster than contiguous transfer are different on these two platforms, but the effect is fully reproducible. We suspect that due to the different access pattern of the stack-storage used by the *direct_pack_ff* technique, the cache utilization can improve for certain blocksizes. This assumption is based on the fact that this effect does not occur for blocksizes bigger than the 1st or 2nd level caches. A detailed survey is beyond the scope of this paper, considering the marginal effect this behavior has in practice.

4. One-Sided Communication

The MPI-1 standard does only define two-sided communication, in which every communication is made up from a receive opera-

---

1. This can be controlled by specifying a minimal block size for the direct_pack_ff algorithm; we have set this to zero for this experiment to do a full comparison.

tion which needs a matching send operation to succeed. However, several application areas with irregularly distributed data (e.g. sparse matrices) or which require dynamic load balancing with strongly varying task sizes (e.g. in computational chemistry) are hard to implement with this model of communication: to enable arbitrary access to local data by remote processes with two-sided communication, all processes need to repeatedly perform global computation or poll explicitly for incoming requests. *Remote memory access* (RMA) would make these accesses much easier. The MPI-2 standard has defined RMA as *one-sided communication* in which all parameters for a communication are supplied by one process only.

4.1 MPI-2 One-Sided Communication

The concept of one-sided communication as defined in the MPI-2 standard is based on *windows*. A window defines a contiguous memory area of each process in a group which is made accessible to all other processes of the group. The process may have allocated this memory area in a random way; however, MPI-2 defines a special memory allocation function to let the MPI library allocate memory with special attributes to increase communication performance. Once a window has been created, the data within (located at the *target process)* may be accessed by any *origin process* via three functions:

- `MPI_Put`: move data from the origin to the target (write access)
- `MPI_Get`: move data from the target to the origin (read access)
- `MPI_Accumulate`: move data from the origin to the target and combine it with the existing data at the specified location.

Different synchronization and consistency schemes can be used to coordinate accesses of multiple origins towards one target and to specify the point in time in which the transaction are visible to all other processes. These relaxed consistency schemes allow implementations to optimize transfers by delaying and possibly gathering multiple requests up to the synchronization point.

4.2 Issues with Single-Sided Communication on SCI

Due to its nature as a memory-coupling interconnect, SCI is well suited for efficient single-sided communication. However, some restrictions need to be taken into account when implementing MPI-2 single-sided communication on top of SCI.

First, with SCI as it can be used in commodity clusters by PCI-SCI adapters, only parts of the address spaces can be shared between processes for direct remote access. Usually, the memory for these parts (the *shared regions*) must have been allocated via the SCI kernel driver. For this purpose, the MPI-2 memory allocation function `MPI_Alloc_mem()` has been implemented to allocate memory from such shared regions. Recent developments for the SCI driver will make it possible to dynamically use arbitrary user-allocated memory regions for remote access via SCI [13]. In any case, a complete implementation needs to provide remote access for arbitrary memory regions, shared or private. On the creation of a window for single-sided communication (MPI-2 call `MPI_Win_create`), SCI-MPICH remembers which parts of the global window are SCI shared memory and thus can be accessed directly. Accessing data from such areas can be done transparently by the CPU, potentially followed by a *load* or *store barrier* to

ensure the completion of all ongoing transactions (which means that the data has arrived at the destination). To access the remaining parts, internal control messages in conjunction with a remote interrupt are used to invoke a remote handler on a process to accept or deliver data using the standard transfer protocols (so-called *emulation* because the direct access is emulated).

Secondly, the bandwidth for remote read access via SCI is much lower than for remote write (see figure 1). This means that direct reading will only be effective up to a certain amount of data, from which on a so-called *remote-put*, in which the target process writes the data into the origin process' address space, will be faster. Such a *remote-put* is also triggered by the target process calling a remote handler at the origin process as described above.

The required mutual exclusion for passive and active target synchronization (see below) is performed via shared memory locks and barriers, using techniques described in [14]. These techniques provide a very low latency for scenarios with little contention. For large-scale clusters and contention, other distributed locking mechanisms based on control messages will probably perform better. Generally, access patterns with lock contention (multiple processes competing continuously to write into one process' window) should be avoided for performance reasons.

4.3 Performance Evaluation

The evaluation of the performance of MPI-2 one-sided communication is difficult because neither recognized application benchmarks nor micro-benchmarks do exist. Also, for micro-benchmarks, many different acceptable performance metrics do exist due to the various consistency modes in MPI-2 one-sided communication. Accesses to windows need to be synchronized in MPI-2 for two reasons:

- Avoid race conditions between concurrent accesses of different processes to the same window.
- Let the MPI library know when accesses start and when they are due to have completed to allow for optimizations like gathering multiple small accesses into a single large access.

To satisfy the requirements of different data access scenarios, MPI-2 defines three synchronization techniques, based on two different models, which the application designer can choose from:

- *active target*: both processes (source and target of the transfer operation) are explicitly involved in the synchronization. This can be achieved either by using a barrier-like operation `MPI_Win_fence` or by defining exposure (at the target process via `MPI_win_post` and `MPI_Win_wait`) and access epochs (at the origin process via `MPI_Win_start` and `MPI_Win_complete`).
- *passive target*: the origin process acquires a lock (via `MPI_Win_Lock`), guaranteeing exclusive access for a window located at the target process. The target process does not take any action.

We have designed a micro-benchmark called *sparse* (see figure 8) which measures fine-grained read and write accesses to remote memory as they are likely to occur in sparse matrix operations. With a fixed access size and a stride bigger than this size (to create non-contiguous accesses), each process iterates through another process' part of the global window using `MPI_Put` or `MPI_Get`

calls. All processes synchronize after having posted all communication calls by calling `MPI_Win_fence`.

```
MPI_Win_create (..., winsize, ...)
for (increasing values of access_cnt) {
  offset = 0
  stride = access_cnt + sizeof(datatype)
  flush_cache()
  time = MPI_Wtime()
  while (offset + access_size < winsize) {
    MPI_Get/Put (.., partner, offset, access_cnt, ..)
    offset = offset + stride
  }
  MPI_Win_fence(...)
  tíme = MPI_Wtime() - time
}
```

Figure 8. Pseudo-code for micro-benchmark *sparse*

Other synchronization techniques could also be used, but due to the collective nature of the operation, active target synchronization is appropriate, and the fence-synchronization causes the least overhead[1]. Depending on the implementation, the required communication operations for each communication call may be delayed until this synchronization point. The pseudo-code in figure 8 illustrates this benchmark. For our experiments, we used a stride of 2, which means after each data element, a gap of the same size follows which is not accessed.

The results of this benchmark, expressed as latency for each communication call and overall bandwidth for all accesses, are shown in figure 9 for two processes located on distinct nodes. Both data directions, get and put, are performed with the communication window located in *shared* SCI memory (direct remote access possible) or in *private* process memory (which means that access is performed via message exchange).

The results for the bandwidth with direct access to remote SCI memory show a relatively low bandwidth for small access sizes, somewhat below the raw SCI performance as given in figure 1. We suspected the strided access to be the reason for the problems, which proved to be true after we evaluated the performance of strided remote write access by another (low-level) benchmark which performed remote writes with various access and stride sizes. The numbers we got bthrough this test show a strong dependency of the effective bandwidth from the stride of the accesses, varying between 5and 28 MiB/s for 8 byte access size, or 7 and 162 MiB/s for 256 byte access size. The values for strides which deliver the maximum performance are multiples of 32, which is the size of the write-combine buffer in the CPU. The reason for the sensitivity of the performance to the stride is the use of *write-combining* for accesses directed towards the remote SCI addresses, imported via the PCI bus. The write-combining buffer of the Pentium-III CPU is 32 bytes big, and does not perform well for mis-aligned accesses. Disabling the write-combining avoids the performance drops, but lowers the overall bandwidth about 50%.

The latency for `MPI_Get` from shared memory is increasing rapidly, which is also due to the strided access pattern. The spike for accessing 3 elements (24 bytes) is reproducible and could not yet

---

1. For the given benchmark, the performance for exposure/access epoch style synchronization is nearly identical for the SCI platform.
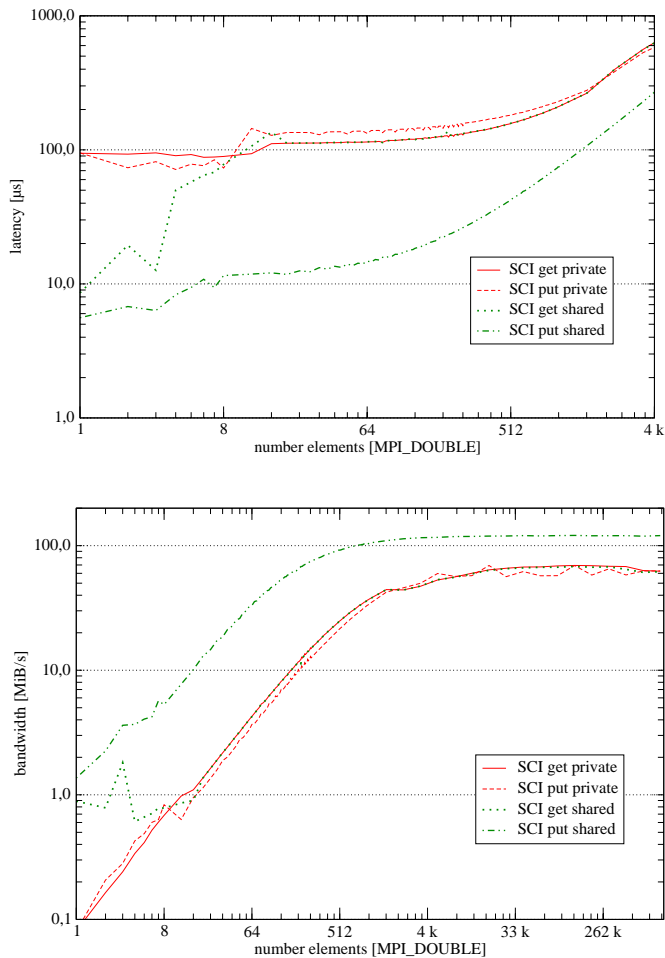
Figure 9. Performance of MPI_Get() and MPI_Put() in SCI-MPICH (*sparse* micro-benchmark, strided access)
top: latency bottom: bandwidth

be specified. The latencies for accessing remote private memory are high due to the required signalling of the remote process and the message exchange involved. Signalling could be omitted for a polling remote thread, but this is would only increase benchmark numbers; real applications would probably suffer even higher latencies.

The bandwidth numbers for accessing remote private memory and reading remote shared memory become very similar for bigger access sizes as they are all performed via message exchange.

5. Related Work & Performance Comparison

A large number of MPI implementations does exist, freely available open-source implementations as well as undisclosed vendor implementations. The internal architecture of some open-source implementations is documented to a certain degree, while the internal architecture of most vendor implementations is not publicly documented in most cases.

5.1 Non-contiguous Datatype Communication

The handling of non-contiguous datatypes is rarely documented at all. [12] describes an optimized packing algorithm, especially well-suited for the NEC SX-5 parallel vector computer with hard-

ware support for strided memory transfers. An library-internal optimization for the handling of non-contiguous vector types is presented in [21]. However, direct packing including communication (to avoid local packing) is not performed in either case.

[22] mentions the possibility of optimizing non-contiguous datatype communication with hardware support, but no results of such an optimization have been published.

A quite complete, but slightly outdated overview of performance for non-contiguous datatype communication is given in [9]. It shows that no MPI implementation on any interconnect could deliver reasonable performance for the non-contiguous case, compared with the contiguous case. Our performance evaluation shows that this has not changed very much.

Also for other platforms, like NEC SX-5 or IBM SP2, the results published in [24] show a significantly reduced performance for non-contiguous datatypes opposed to the contiguous equivalent.

5.2 One-Sided Communication

Much more work has been done regarding the implementation of one-sided communication because it is an obvious characteristics of an MPI implementation to support one-sided communication or not. Many vendor MPI implementations (like SGI, Cray, Hitachi, HP, IBM) support one-sided communication using the custom interconnect of the respective machine, but very little information on implementation and performance is publicly available.

The MPI implementation for the NEC SX-5 features one-sided communication as described in [16]. Like an SCI-connected cluster, the SX-5 knows global shared memory and process local memory. For large data blocks, the one-sided communication via global shared memory is considerably faster then via process local memory, especially when communication is performed with more than one partner in a single synchronization phase. In [15], the SX-5 implementation of one-sided communication has been ported to a SMP-cluster with VIA [25] communication mechanisms, which only features write-access to remote memory. Due to the required explicit synchronization, one-sided communication in both of these implementations has considerably higher latencies than the equivalent two-sided send-recv construct.

[17] and [18] describe the implementation of one-sided communication for generic interconnects (including TCP/IP and shared memory), SMP shared-memory and a combination of both. The results show that only for the specialized shared-memory implementation, one-sided communication can achieve lower latencies than the equivalent two-sided communication. It must be noted that the authors used ping-pong communication without synchronization to evaluate the performance, which is not at all a typical communication pattern of one-sided communication.

In [19], the implementation of one-sided communication for Myrinet-connected SMP nodes is presented. This is not MPI-2, but a custom API *ARMCI* which can operate next to MPI (or serve to implement MPI-2 on top of it). The Myrinet interconnect is used via the GM API [20]. As with Myrinet, remote memory access does always need to be performed via the DMA engine of the PCI adapter, the latency is considerably higher then with SCI. The authors have put a lot of effort into the implementation to increase the performance, especially for non-contiguous data.

Machine	Interconnect	MPI	OSC	ID
Cray T3E-1200	custom	Cray	yes	C
Sun Fire 6800	Gigabit Ethernet	Sun HPC 3.1	no[a]	F-G
24-way SMP, 750 MHz	shared memory	64 bit PCI	yes	F-s
PentiumIII Dual SMP	SCI	MP-MPICH	yes	M-S
800 MHz, 64 bit PCI	shared memory	1.2.1 beta	yes	M-s
Pentium III Xeon	fast ethernet	LAM 6.5.4	yes	X-f
Quad SMP, 550 MHz	shared memory		yes[b]	X-s
Pentium II Dual SMP	Myrinet 1280	SCore 2.4.1	no	S-M
400 MHz, 32 bit PCI	shared memory		no	S-s

    a. Myrinet installed, but not yet available (does also not support one-sided communcation with Sun MPI)

    b. only MPI_Get(), MPI_Put() deadlocked

Table 1. Cluster platforms for evaluation of MPI performance

However, the peak bandwidth is not reached until the size of the blocks to transfer is bigger than 700 kiB - a lot of performance degradation is due to the slow registering of DMA memory by the GM driver. If registering is omitted, two additional memory copy operations are required to transfer data from and to pinned memory for DMA. Adapting this library to the SCI interconnect, as it is planned by the authors, will allow interesting comparisons.

5.3 Performance Comparison

The lack of hard information on the communication characteristics for non-contiguous datatypes and one-sided communication made it necessary to do evaluate them ourselves. We had access to a number of MPI platforms, including an MPP and clusters of 24-way, 4-way and 2-way SMPs, connected via Myrinet, SCI, Gigabit Ethernet or Fast Ethernet (see table 1). The overview table lists the hardware platform, the interconnect used for message passing and the MPI implementation (and whether it supports one-sided communication). Each configuration is given an ID for easier reference in the following figures.

We performed the non-contiguous benchmark for the simple vector type on all of these configurations. .

The results are shown in figure 10 and show that obviously none of the tested MPI implementations has a consistent technique to optimize non-contiguous data transfers. The Cray T3E reaches an efficiency of about 1 for blocksizes between 8 and 32 kiB, but has a very low efficiency for very small (< 4 kiB) and big (> 32 kiB) blocksizes. Sun MPI for shared memory shows a very constant efficiency, which jumps from 0.5 to 1 for blocksizes of 16k and above, which indicates that a simple optimization has been implemented. However, no information is available concerning this feature [23]. All other implementations seem to use the generic pack-and-send technique for intra- and inter-node communication as well.

The configurations supporting one-sided communication also ran the *sparse* micro-benchmark with the results given in figure 11.

Obviously, Sun MPI delivers very good performance for shared
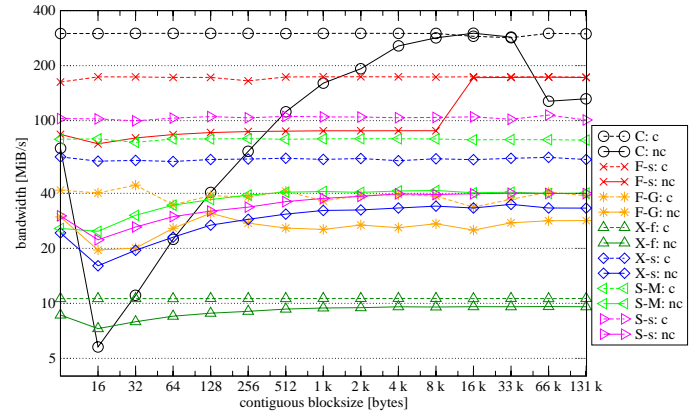
Figure 10. Non-contiguous datatype communication: bandwidth for strided vector *(nc)* and equivalent contiguous vector *(c)*

memory communication, while it does not yet support one-sided communication for inter-node communication, neither via Ethernet nor via Myrinet. Cray T3E also shows good performance, which is in the same range as the performance of SCI-MPICH for SCI remote shared memory. LAM-MPI is one of the few freely available MPI implementations to offer one-sided communication. As expected, it has very high latencies and gives a maximum of 10 MiB bandwidth via fast ethernet. Surprisingly, the performance of the shared memory implementation is a little bit lower than SCI-MPICH via SCI. One-sided communication using a VIA-interface as presented in [15] shows significantly higher latencies: for 1024 bytes, it's about a factor 3 (compared with one-sided communication via messages on SCI) up to a factor of 15 (compared with direct SCI put) slower than using the presented solution via SCI.
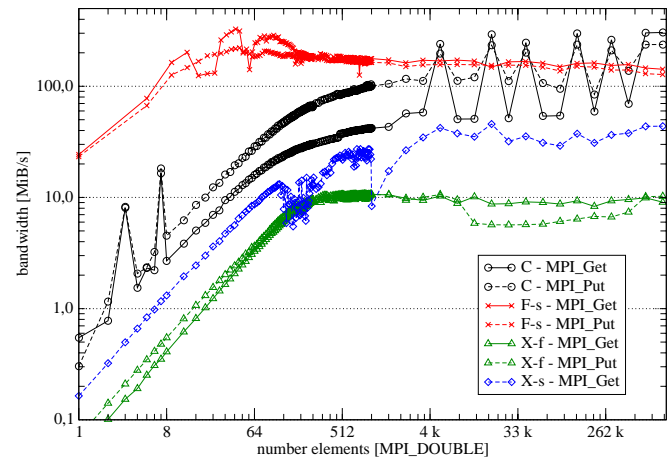
Figure 11. Performance for single-sided communication in *sparse* micro-benchmark

We have compared the scaling behavior for one-sided communication on all of the platforms with hardware-supported (shared memory, SCI or Cray interconnect) one-sided communication as illustrated in figure 12. As expected, the shared-memory platforms exhibit a much higher bandwidth for fine-grained accesses. How-
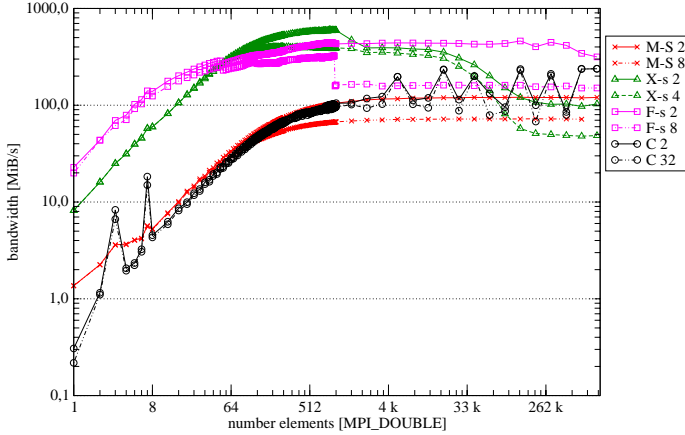
Figure 12. Scaling of one-sided strided communication (*sparse* benchmark, `MPI_Put()`) on platforms with hardware-support for this operation. On the Xeon platform, SCI-MPICH is used for intra-node shared memory communication. Bandwidth shown is the mininun of the per-process maximum bandwidths achieved.

ever, platforms with an inferior memory system design like the 4-way Xeon SMP scale very badly for coarse-grained accesses and deliver a bandwidth below the SCI-connected system. The high-performance (and high-cost) shared-memory designs of the Sun Fire system scale better, but even it's bandwidth declines notable for more than 6 active processes. On the other hand, the distributed memory architectures with hardware-supported remote memory access scale significantly better. The Cray T3E keeps its uneven, but regular bandwidth characteristics constant for up to 32 processes.

The SCI interconnect has a considerably higher bandwidth for single-element accesses and a constant peak bandwidth of 120 MiB/s for up to 5 nodes. For more than 5 nodes, the single SCI ringlet used in this tests does not supply sufficient bandwidth: using the default link frequency of 166MHz, the ring bandwidth is at 633 MiB/s. and the peak bandwidth declines accordingly down to 71.8 MiB/s for 8 nodes. However, as SCI is based on independent point-to-point connections (*segments)*, the effective bandwidth for such a communication scenario depends on the number

Active Nodes	1 transf. / segment		8 transfers / segment			
	p. node	acc.	p. node	acc.	load	eff.
4	122.94	491.8	120.70	482.8	76.3 %	76.3 %
5	120.69	603.5	115.80	579.0	95.3%	91.5%
6	120.88	725.3	97.75	586.5	114,4%	92.7%
7	120.66	844.6	79.3	555.1	133,5%	87.7%
8	120.83	966.6	62.78	502.2	152.5%	79.3%

Table 2. Scalibility for different segment utilization levels (the *p. node* and *acc.* columns show the bandwidth in MiB/s). The *eff.* column gives the effiency of the utilization of the available bandwitdth for the worst case setup (link saturation).

of concurrent data transfers running over each segment (*segment utilization)*. The topology of the cluster used for these experiments is a single ring of 8 nodes. For the experiment above, we have chosen an average scenario in which each segment is utilized by 4 transfers. The maximal utilization of a segment with this topology is 8 transfers, while the minimal utilization is of course a single transfer from one node to the next one. The results for these two extremes are given in table 2. They show that for the minimal segment utilization, the bandwidth per node remains constant at 120 MiB/s, although the overall traffic on the ring rises due to flow control packets sent back for each data packet received by a node. With a segment utilization of 8, the per node bandwidth declines as the ring becomes saturated. To check for saturation effects, we calculated the relative ring load (output traffic of nodes relative to nominal ring bandwidth) and the ring efficiency (ratio of accumulated bandwidth to nominal ring bandwidth). The efficiency 79.3% for a load of 152.5% indicates that little congestion is present.

However, it is obvious hat the link bandwidth needs to be increased for ringlets with more than 6 nodes. We did do so by increasing the link frequency up to 200 MHz (by software), resulting in a nominal link bandwidth of 762 MiB/s. The measured bandwidth for the worst case scenario (8 sending nodes with segment utilization of 8) increased linearly with the ring bandwidth indicating that this is the next step to go for increased scalability.

It shows that for this application, the SCI interconnect offers similar performance for commodity clusters as the custom Cray interconnect does deliver in the T3E. With the increased link frequency, a limit of 8 nodes per ringlet seems reasonable, which gives a 512 nodes system when using 3D-torus topology.

6. Conclusions & Outlook

Our results show that global shared memory as implemented by SCI offers possibilities for optimization of cluster communication as demonstrated for the MPI implementation SCI-MPICH. Such techniques are not easily possible with cluster interconnects not supporting transparent load and store operations. However, it once again became clear that SCI as implemented for clusters (via the PCI bus) is *not* shared memory, but I/O-access which is very sensitive concerning *how* data is passed onto the bus and from there onto the PCI-SCI adapter. Proper alignment is very important, which sometimes may severely limit the efficiency of transparent remote accesses, if not intercepted by software. The performance numbers achieved are already satisfying, although some aspects as the MPI_Put() on shared memory leave room for improvement. An interesting side effect is that all of the work presented for the SCI interconnect can equally be applied to intra-node shared memory communication thanks to the abstraction of the SMI library (*Shared Memory Interface* [26]), as shown for the performance measurements.

It will be interesting to evaluate the possibilities of non-contiguous data transfers with DMA-based interconnects. This can be done with the DMA-engine of the PCI-SCI adapters, but also with „smart" interconnect adapters like Myrinet. Also, true single-sided communication (without a software handler at the target process) should be possible with such interconnects. It is also included in

the VIA specifications, but since it is optional, it is rarely implemented. It probably is the higher complexity involved in such a solution which has hindered implementations so far.

Generally, if synchronization is considered, one-sided communication does usually not provide lower latencies if compared directly with two-sided communication using micro-benchmarks. However, all these numbers do not, and can not, include the potential synchronization delay included when using two-sided communication in a global exchange phase. Because the communication models are so different (one-sided vs. two-sided and also the different one-sided synchronization schemes), ping-pong-like comparisons are not really meaningful, but can give an upper limit of performance. Only comparing the performance and algorithmic complexity of applications solving a given problem with one- or two-sided communication will allow to decide for one or the other technique. Such benchmarks are not yet commonly available; but we expect SCI to support these communication models well.

The complete source of the software packages discussed in this paper, including the benchmarks used, is freely available at [8].

7. Acknowledgements

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]   Quadrics Supercomputer World Ltd: *QsNet High Performance Interconnect.* Manufactures product information. Retrieved October 2001.
URL: http://www.quadrics.com/web/support/flyers/QsNet.pdf

[4]   Fabrizio Petrini, Adolfy Hoisie, Wu-chun Feng and Richard Graham: *Performance Evaluation of the Quadrics Interconnection Network.* Workshop on Communication Architecture for Clusters (CAC '01), in conjunction with Int'l Parallel and Distributed Processing Symposium (IPDPS '01), San Francisco, April 2001

[5]   Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su: *Myrinet - A Gigabit-per-Second Local-Area Network.* IEEE Micro vol.15 (1), February 1995.

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

[7]   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

[8]   J. Worringen, K. Scholtyssik: *MP-MPICH: Multi-Platform MPICH.*
URL: http://www.lfbs.rwth-aachen.de/users/joachim/MP-MPICH

[9]   Mike Ashworth: *A report on further progress in the development of codes for the CS2.* In Deliverable D.4.1.b F. Carbonnell (Eds), GPMIMD2 ESPRIT Project, EU DGIII, Brussels, (1996)

URL: http://www.cse.clrc.ac.uk/PublicationAbstract/943

[10]   Mike Ashworth: *OCCOMM Benchmark Code.* Presentation at the Workshop on Scalable Parallel Computing on Cray Systems (Berlin) and the 8th RAPS Workshop (Offenbach), November 1995
URL: http://www.dl.ac.uk/TCSC/CompEng/OCCOMM/slides.html

[11]   W. Gropp, E. Luskk, N. Doss and A. Skjellum: *A high-performance, portable implementation of the{MPI} message passing interface standard.* Parallel Computing, vol. 22 (6), pp 789-828, September 1996.

[12]   Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, Falk Zimmermann. *Flattening on the Fly: efficient handling of MPI derived datatypes.* In Recent Advances in Parallel Virtual Machine and Message Passing Interface. 6th European PVM/MPI Users' Group Meeting, volume 1697 of Lecture Notes in Computer Science, pages 109-116, 1999.

[13]   Friedrich Seifert, Joachim Worringen, and Wolfgang Rehm: *Using Arbitrary Memory Regions for SCI communication.* In Proc. SCI Europe Conference 2001, pp. 59-64, Trinity College, Dublin, October 2001.

[14]   Martin Schulz: *Efficient Coherency and Synchronization Management in SCI based DSM systems.* In Proc. of SCI Europe Conference 2000, pp. 31-36. Helt in conjunction with EuroPar 2000 Conference, Munich, September 2000.

[15]   Maciej Golebiewski, Jesper Larsson Träff. *MPI-2 One-sided Communications on a Giganet SMP Cluster.* In Recent Advances in Parallel Virtual Machine and Message Passing Interface. 8th European PVM/MPI Users' Group Meeting, volume 2131 of Lecture Notes in Computer Science, pages 16-23,2001 (to appear).

[16]   Jesper Larsson Träff, Hubert Ritzdorf, Rolf Hempel. *The Implementation of MPI-2 One-Sided Communication for the NEC SX-5.* In Proc. Supercomputing 2000 Conference, Dallas, USA, November 2000.

[17]   Stephen Booth, Elson Mourao: *Single sided MPI implementations for SUN MPI.* In Proc. Supercomputing 2000 Conference, Dallas, USA, November 2000.

[18]   Elson Mourao, Stephen Booth: *Single Sided Communications in Multi-Protocol MPI.* J.Dongarra et al. (Eds.): EuroPVM/MPI 2000, LNCS 1908, pp. 176-183, Springer-Verlag Berlin Heidelberg 2000.

[19]   Jarek Nieplocha, Jialin Ju, and Edoardo Apra: *One-sided Communication on the Myrinet-based SMP Clusters using the GM Message-Passing Library.* Workshop on Communication Architecture for Clusters (CAC '01), in conjunction with Int'l Parallel and Distributed Processing Symposium (IPDPS '01), San Francisco, April 2001

[20]   Myricon Inc.: *The GM API.* Online documentation of the GM API specificatin. Last update October 30th, 2000.
URL: http://www.myri.com/scs/GM/doc/gm_toc.html

[21]   William Gropp, Ewing Lusk, and Deborah Swider: *Improving the Performance of MPI Derived Datatypes.* Message Passing Interface Developer's and User's Conference (MPIDC '99), Atlanta, USA, March 9-12,999.

[22]   David Sitsky, David Walsh, and Chris Johnson: *An efficient implementation of the message passing interface (MPI) on the Fujitsu AP1000.* In Mitsuo Ishii (Eds), Proceedings of the Third Parallel Computing Workshop, Kawasaki, Japan, November 1994..
URL: http://cap.anu.edu.au/cap/projects/mpi/mpi.html

[23]   Sun Microsystems Inc.: *Sun HPC ClusterTools" 3.1 Performance Guide.* Rev. A,Palo Alto, CA, March 2000

[24]   Ralf Reussner, Jesper Larsson Träff, Gunnar Hunzelmann. *A Benchmark for MPI Derived Datatypes.* In Recent Advances in Parallel Virtual Machine and Message Passing Interface. 7th European PVM/MPI Users' Group Meeting, volume 1908 of Lecture Notes in Computer Science, pages 10-17, 2000.

[25]   Compaq, Intel and Microsoft Corporations: *The Virtual Interface Specification.* Version 1.0. Dec 16, 1997.
URL: http://www.viarch.org

[26]   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
URL: http://www.lfbs.rwth-aachen.de/users/joachim/SMI