Pipelining and Overlapping for MPI Collective Operations

Joachim Worringen
C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany *
worringen@ccrl-nece.de
http://www.ccrl-nece.de

Abstract

Collective operations are an important aspect of the currently most important message-passing programming model MPI (Message Passing Interface). Many MPI applications make heavy use of collective operations. Collective operations involve the active participation of a known group of processes and are usually implemented on top of MPI point-to-point message passing. Many optimizations of the used communication algorithms have been developed, but the vast majority of those optimizations is still based on plain MPI point-to-point message passing. While this has the advantage of portability, it often does not allow for full exploitation of the underlying interconnection network. In this paper, we present a low-level, pipeline-based optimization of one-to-many and many-to-one collective operations for the SCI (Scalable Coherent Interface) interconnection network. The optimizations increase the performance of some operations by a factor of four if compared with the generic, tree-based algorithms.

Keywords: collective operations, pipelining, overlapping, MPI, SCI

1 Introduction

The currently most important API definition for parallel programming using the message-passing paradigm is the *Message Passing Interface* (MPI [10, 11]). MPI offers numerous variants of the basic point-to-point message passing functions `MPI_Send` and `MPI_Recv`. Additionally, it contains a wide range of so-called *collective operations* (CO). They are designed to perform *one-to-many* (1:N), *many-to-one* (N:1) or *many-to-many* (N:N) data distributions. Typical examples of such operations are `MPI_Bcast`, which is a 1:N multicast, and `MPI_Reduce`, which performs a N:1 data gathering with simultaneous combination of each process' data vector into one result vector. An example for an N:N operation is `MPI_Allreduce` which can be described as `MPI_Reduce` followed by `MPI_Bcast`, using the result of the previous reduction.

A CO for a specified "communicator" (group) of P processes can not complete for all P processes unless all of them have invoked CO. This inherent characteristic of all collective operations allows for optimizations which are not possible for point-to-point communication. The active participation of all processes involved in the CO makes it possible to coordinate the data transfers between the processes to make optimal use of the underlying communication system. This mostly concerns the interconnect, but also other communication resources like shared memory on nodes running more than one process.

In this paper, we present optimizations for the COs mentioned above for SCI-MPICH [22]. SCI-MPICH is an MPI implementation for the SCI interconnect [7] which is utilized via the SISCI API [19, 2]. The optimizations constitute a data-transfer protocols which is based on the pipelining principle. Additionally, they make use of concurrent intra- and inter-node data movements and computation which leads to a speedup of more than 4 compared to the generic tree-based algorithms. An overview of related work in this area is given in Chapter 2. The basic aspects of intra-node communication using SCI are shown in Chapter 3. Chapter 4 explains our optimization approach and derives analytical models from the communication operations. The results of these models are compared with some experimental results in Chapter 5.

2 Related Work

During the last decade, a much work has been performed on collective communication in general, and on implementations of COs in MPI. Mitra et.al. [12] give basic analytical models of the most common collective operations, with different algorithms for short and long vectors and consid-

*This work was performed at the Chair for Operating Systems, Technical University of Aachen, `http://www.lfbs.rwth-aachen.de`

eration of mesh topologies. Their algorithms contain a variant of pipelining for long vectors. For short vectors, they propose an interleaving of communication and computation (for reduce operations). Their implementation of those algorithms for the Intel Paragon achieved considerable performance increases for `MPI_Bcast` and `MPI_Allreduce` if compared with existing approaches. Despite this early work, when Luecke [9] evaluated the performance of collective operations on SGI and IBM system four years later, he found that on each system certain COs did not perform as well as a reasonable generic algorithm (while others performed better). Likewise, the popular open-source MPI implementation MPICH has just recently integrated a range of existing generic algorithms for COs to replace the existing ones and could achieve significant performance increases on two different platforms [20]. An optimized algorithm for `MPI_Reduce` and `MPI_Allreduce`, implemented by Rabenseifner [16], also delivered an increased performance when integrated into an MPI library for the SCI interconnect [5].

These achievements show that even generic algorithms, using message-based point-to-point communication, can increase performance if they are carefully adjusted to the characteristics of an interconnect. However, an even higher performance can be achieved if special capabilities of an interconnect are exploited by means which are not accessible through message-based point-to-point communication. Fleischmann [3] did so by using direct shared-memory communication on the Convex, a cc-NUMA SMP system, considering the different performance levels of local and remote memory. For clusters with message-based interconnects, it is necessary to perform low-level accesses to the network adapter like Bhoedjand et.al. [1] did for Myrinet and Petrini et.al. [15] for Quadrics. Both implementations have limitations, though: for Myrinet, a custom firmware (*Myrinet Control Program*) is required, which many user hesitate to use. For Quadrics, the applicability of the low-level COs depends on the placement of the processes in the network. It is not known if Petrini's work is applied to the MPI implementation for Quadrics.

Oral and George [13] evaluated different communication topologies for multicast operations in a two-dimensional SCI torus. They achieve the best results for with multiple sequential trees along one dimension of the torus. However, they do only consider broadcast operations (no reduction operations), operate on a lower level than MPI, and do not use DMA transfers. The completion latency achieved for a broadcast of 512KB across 8 nodes is about twice as high as the performance of the approach presented in this paper. The hardware configuration of the test systems is identical to the setup used in this paper, except for slightly faster CPUs.

Sanders [17] theoretically describes a communication algorithm termed "fractional tree" for broadcast and reduction operations. It is an hybrid of a linear pipeline and a binary tree and thus is suited to alleviate the scaling problem of the linear pipelining which will show up with the performance modelling in chapter 5.3.

3 SCI Communication

An SCI interconnect [7] between a number of nodes is based on PCI-SCI adapter boards (PSA [8]) which communicate via a switched fabric of point-to-point connections. This way, the fabric can have virtually any topology. Typical topologies are *star* (using a central switch) and *k-ary n-cubes* (typically two- or three-dimensional tori).

The PSA boards contain a PCI-to-PCI bridge and translate accesses to certain parts of the node-local PCI address space into accesses to the global SCI address space of all nodes. Likewise, a PSA also translates accesses to the global SCI address space (which come in as packets via the switched fabric) back to the local PCI address space if the local PCI-SCI address mapping indicates this. Furthermore, packets are routed through the PSAs on the fabric on their way from the source to the destination node.

Using a current-generation SCI interconnect, communication between processes on different nodes can be performed in two different ways:

PIO By mapping remote memory segments into the local process address space, it is possible to write or read from remote memory the same way as it is done from local memory. This means, the CPU can perform arbitrary load and store operations as the mapping is fully transparent. However, the latency for the accesses to remote memory is higher than for local memory. Next to this, different consistency semantics apply for remote memory due to additional buffering on the PCI-SCI adapters and the lack of cache-coherence for remote memory. This requires explicit memory synchronization like flushing local write buffers (*flush*), reloading local read buffers (*load barrier*) or waiting for completion of outstanding write operations (*store barrier*) to ensure certain memory states.

DMA The PSA has an integrated DMA engine which allows for data transfers with very little CPU activity. Once the description of the desired transfer is loaded into the DMA engine, all data transfers are executed independently from CPU activity. For DMA into remote memory, the remote memory segment does not need to be mapped into the address space, but only needs to be "connected" as the transfers are based on physical (not virtual) addresses. Both the source and the target buffer need to be allocated via the SCI driver, or must have been *registered* for SCI usage [1].

All benchmarks in this paper were performed on a cluster of 8 identical nodes. Each node has two PentiumIII CPUs, 512MB of RAM and a ServerWorks ServerSet III-LE chipset, which offers a $64\,\mathrm{Bit}$, $66\,\mathrm{MHz}$ PCI bus.

Benchmarks for this platform show that the minimal latency of PIO transfers of $1.5\,\mu s$ (for a $4\,\mathrm{Byte}$ write) is much lower than for DMA which starts at about $30\,\mu s$ (for a $64\,\mathrm{Byte}$ write). Likewise, the bandwidth for PIO reaches 90% of peak bandwidth for blocksizes less than $512\,\mathrm{Byte}$. With DMA, a blocksize of $128\,\mathrm{KB}$ is required for 90% of peak bandwidth. However, the peak bandwidth of PIO transfers ($170\,\mathrm{MB}/s$) is lower than for DMA transfers ($250\,\mathrm{MB}/s$). Additionally, the bandwidth of PIO transfers depends on the implementation of the interface between CPU bus and PCI bus (the "chipset") and also on the memory access performance. The latter leads to a performance decrease for blocksizes beyond $128\,\mathrm{KB}$ (50% of the CPU cache size) on our platform. On more recent platforms, a non-decreasing peak PIO bandwidth of $260\,\mathrm{MB}/s$ has been observed. The DMA bandwidth, in contrast, is independent from the mentioned interface, CPU performance and any cache effects.

4 Employing Efficient Pipelining

Pipelining is a well known technique to reduce the processing time T of N_t tasks which have to pass a number N_s of sequential stages with identical processing latencies l_s [2]. Naive sequential processing would lead to

$$T_{seq} = l_s \cdot N_s \cdot N_t$$

with only 1 active stage at a time. Pipelined processing results in

$$T_{pipe} = l_s \cdot (N_s + N_t) \quad (1)$$

for total processing time, with more than one (up to all) stage being active except for the very first and last processing step. In (1), N_s identifies the stages which fill up the pipeline, while N_t stages are processed in parallel. For pipelining a single task (like broadcasting a given amount of data), it is necessary to split it into sub-tasks which can be processed independently. In this case, the impact of l_s increases as it does not occur only $1 \cdot N_s$ times, but $N_t \cdot N_s$ times. As l_s also contains a certain amount of overhead (the communication startup latency), this may result in reduced performance.

Therefore, when COs are based on `MPI_Send` and `MPI_Recv`, using pipelining is usually less efficient than using other communication topologies. For maximal performance of collective operations, it is crucial to achieve high concurrency of all stages. With all stages doing the same work independently from each other, this can also be achieved by using other processing schedules, especially tree-oriented topologies. A binary tree has the advantage of reaching maximal communication parallelism in $O(ld(N))$ instead of $O(N)$ steps. Additionally, the total number of communication steps is lower than for pipelining. The bandwidth of a 1:N- or N:1-style CO for a vector of size D_v can be defined as $B_{CO} = \frac{D_v}{T_{CO}}$, with T_{CO} being the time difference between the first call and the last exit of the collective function by any process involved.

This shows that the applicability of pipelining for the implementation of COs is limited. However, with the low-latency communication characteristics of SCI it is possible to define communication protocols in which l_s contains very little overhead. Together with concurrent intra- and inter-node communication, efficient pipelining is possible as we will show below.

4.1 Generic Principle

To pipeline a single CO, the vector on which this operation is to be performed needs to be split into N_t parts. Each of these parts should be transfered with minimal overhead to ensure efficiency of the pipeline. This requirement also applies to the flow control needed to avoid data corruption. With SCI, the most efficient data transfer between $P = 2$ processes on remote nodes is to use a *ring buffer* of D_{ring} bytes of SCI shared memory at the receiving process. The sending process writes data into the ring buffer, D_b bytes at a time. The receiving process reads the data with the same granularity. The flow control is realized via two additional locations in shared memory which are updated by the processes according to the position in the ring buffer up to which they have written or read data. This technique has the potential disadvantage that only one transfer can be handled at a time. For COs this is not a problem as MPI does not allow concurrent COs.

For a pipelined transfer with $P > 2$ processes, each process p_j which has to receive data allocates an inbound ring buffer in its local SCI shared memory and tells process p_{j-1} how to access this buffer. Once it has received such information from process p_{j+1} (which is the next stage in the pipeline) for the outbound buffer, it polls the inbound buffer for data to arrive. Once this data arrives, it processes it locally as required (i.e. copying it into the local receive buffer that is provided by the user) and writes another block of data into the outbound buffer. Using PIO transfers, these two operations are serialized. Only when using DMA for outbound data transfers, it is possible to overlap local processing of the data and the outbound data transfer. The next

[1] The *registering* functionality is not yet included in the standard SCI driver, but only in a development branch.

[2] If the phases have different processing latencies, the maximal latency will dominate except for the filling of the pipeline.

chapters will explain how this is realized for the different COs.

The following parameters are required for the analytical models presented in the next chapters:

P	number of processes which take part in the CO
D_v	size of vector to be communicated
D_b	size of data block to be transfered without flow-control
$l_{PIO}(d)$	latency of a PIO write operation for d bytes into remote memory
$l_{DMA}(d)$	latency of a DMA write operation for d bytes into remote memory
$l_{cpy}(d)$	latency of a copy operation for d bytes in local memory
$l_{cmb}(d)$	latency of a combine computation for d bytes
l_{sb}	latency of a store barrier

Each 1:N- or N:1-CO has a root process. For 1:N, it is the process which owns the data for all processes before the CO is performed. Likewise, with N:1, it is the process which receives data from all processes, including itself. For simplicity, we assume that the root process always has rank 0. A simple rank transformation is required to satisfy this condition for arbitrary root processes.

4.2 Pipelined Broadcast Operation

From the generic principle, we derived the *pcast*-protocol which is explained in Figure 1. It illustrates an `MPI_Bcast` for $P = 3$ with process p_0 being the root process sending its data to p_1 and p_2. They have allocated ring buffers of size D_{ring}, divided into $N_b = D_{ring}/D_b$ blocks. Likewise, the vector will be transfered through the pipeline in $N_t = D_v/D_b$ pieces. All pipelined protocols presented in this paper use not only flow control within the single pipeline, but also between different subsequent calls to distinct CO's. This situation can occur as one process at the start of the pipeline may already be done with its transmissions while other processes at the end are still busy.

We can see that p_0 only needs to send its data into the current incoming block of the ring buffer of p_1 using PIO transfers (action 1). The reason why PIO and not DMA is used for this transfer is that the user-supplied send buffer would need to be *pinned* to use it as a DMA source buffer. This functionality is not yet implemented in the standard SCI driver, although experimental work has shown that it is possible to do this with little overhead [18, 23], and the protocol is prepared to make use of this functionality once it is available. p_1 in turn has to transfer the data from the forward block (which was the incoming block in the previous step) into the receive buffer (action 2) using PIO transfers, too. Concurrently, it forwards the same data to p_2, using

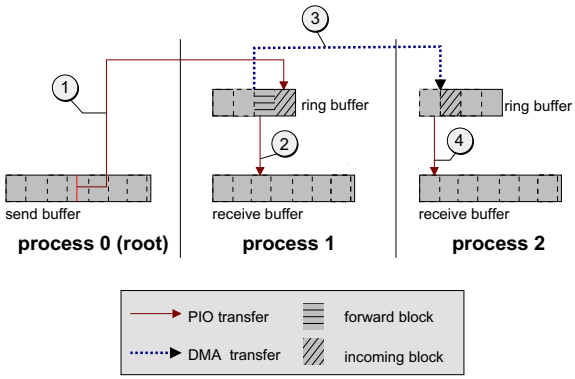

Figure 1. Data flow for the *pcast* protocol using concurrent PIO and DMA transfers

a DMA transfer (action 3). This is the same task for all p_k, $0 < k < P - 1$. The last process in the pipeline, p_2 in this example, only needs to copy the incoming data into the user buffer (action 4).

For the *pcast* protocol, T_{bcast} can be calculated according to (1):

$$T_{bcast}(D_v) = (P + \frac{D_v}{D_b}) \cdot (l_{DMA}(D_b) + l_{PIO}(4) + l_{sb}) \quad (2)$$

4.3 Pipelined Reduce Operations

Similar to the *pcast* protocol, the *rpipe* protocol was defined and implemented to perform pipelined reduction operations. The data flow of the reduce pipeline is more complex (relative to the broadcast pipeline) as the data is not only forwarded, but also modified by each process. This modification (the combine operation) requires that at each process p_j, $0 < j < P - 1$, has not only one, but two active blocks in its local ring buffer:

combine block: The data contained in the *combine block* is currently combined with the matching block of data of the local vector, stored in the send buffer.

forward block: The block that is the *combine block* in step s of the pipeline becomes the *forward block* in step $s + 1$ and is forwarded to process p_{j+1}.

Figure 2 shows the data flow of the *rpipe* protocol for $P = 3$ processes. The pipeline starts at process p_1 which writes the data via PIO into the current incoming block of the ring buffer of p_{P-1} (action 1). At the same time, p_2 combines the related data of its local send vector with the data in the combine block (action 2), and forwards data from the forward block to p_0 (action 3). At the end of the pipeline, p_0 combines the data from the combine block and its local send

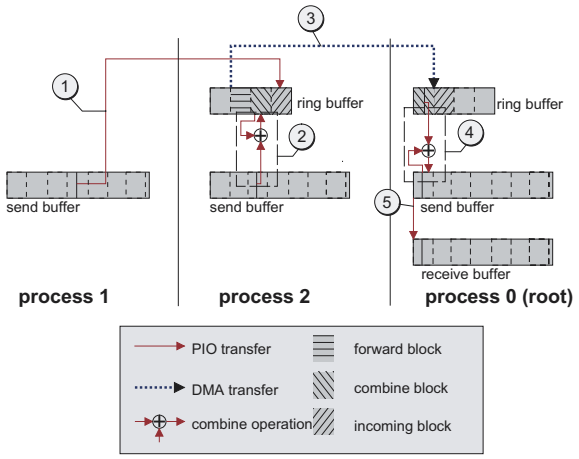

Figure 2. Data flow for the *rpipe* protocol using concurrent PIO and DMA transfers

vector (action 4) and stores the result in the receive buffer (action 5).

The transfer time T_{reduce} of the *rpipe* protocol can be determined as

$$\begin{aligned} T_{reduce}(D_v) &= l_{PIO}(D_b) + l_{cmb}(D_b) + l_{cpy}(D_b) \\ &+ (P-2)\cdot(l_{cmb}(D_b) + l_{DMA}(D_b) + l_{PIO}(4) + l_{rw}) \\ &+ (\frac{D_v}{D_b} - 1)\cdot(\max(l_{cmb}(D_b), l_{DMA}(D_b)) \\ &+ l_{PIO}(4) + l_{rw}) \end{aligned} \quad (3)$$

The first two lines in (3) describe the fill time of the pipeline (at p_1, p_0 and the $P-2$ other processes). This time is longer than for the *pcast* protocol as there are nearly twice as many stages. Additionally, these stages perform two different operations. The remaining lines relate to the overlapped processing of the data blocks once the pipeline is filled: only the maximum of the two times (transfer via DMA or combine operation by the CPU) is relevant.

Closely related to `MPI_Reduce` is `MPI_Scan`, a non-exclusive prefix reduction. In contrast to a plain reduction, every process p_j will have the combined vectors of all processes p_k, $k \leq j$. This operation is also performed via the *rpipe* protocol. The only difference is that each combine block is not only forwarded to the next process, but is also copied into the local receive buffer. Therefore, T_{scan} is identical to T_{reduce} except for the addition of $l_{cpy}(D_b)$ to the factor of copy operation latencies for the full pipeline operation. This results in an execution time T_{scan} of

$$\begin{aligned} T_{scan}(D_v) &= l_{PIO}(D_b) + l_{cmb}(D_b) + l_{cpy}(D_b) \\ &+ (P-2)\cdot(l_{cmb}(D_b) + l_{DMA}(D_b) + l_{PIO}(4) + l_{rw}) \\ &+ (\frac{D_v}{D_b} - 1)\cdot(\max(l_{cmb}(D_b), l_{DMA}(D_b) + l_{cpy}(D_b)) \\ &+ l_{PIO}(4) + l_{rw}) \end{aligned} \quad (4)$$

4.4 Pipelined Global Reduce Operation

A global reduce operation (`MPI_Allreduce`) can be performed by calling `MPI_Reduce` followed by `MPI_Bcast`. In case this is done with the pipelined implementations, two pipeline fill operations will occur. In contrast, an algorithm as implemented by Rabenseifner runs "continuously" but without overlapping of combine and communication operations.

It is possible to define a single-pipeline protocol for `MPI_Allreduce`by running the *pcast* pipeline directly after the *rpipe* pipeline. This would remove one of the currently two pipeline fill times. However, such a protocol would either require to pass the data two times through each node (at the same time) or to buffer the complete vector at the root. For the first variant, the single DMA engine on the PSA will be a bottleneck. The second variant seems more worthwhile, but has not yet been implemented.

5 Performance Evaluation

Firstly, we evaluate the implementation of these protocols and compare the results with the generic algorithms. We will then evaluate certain characteristics of the pipeline protocols by applying our models.

5.1 Experimental Results

We have measured the optimized collective operations by running the Pallas MPI Benchmark[3] [14] on the test cluster described above. The topology of the SCI interconnect used in this cluster is a single ring. However, the topology is not relevant for the performance of the presented pipelined data transfer protocols as each node does only communicate with its direct neighbor. This means that for each inter-node communication, a different, independent SCI link segment is used.

For each type of collective operation, we compare the pipelined version with the generic algorithm found in MPICH [4][4]. The generic algorithms use PIO-based point-to-point communication and tree-oriented communication topologies like binary or binomial trees which give a scaling property of $O(log(N))$. The results are depicted in Figure 3 and 4. The charts show the effective bandwidth per process B for different vector sizes D_v and process counts P. Per default, the pipelined protocols are used for $D_v \geq 32KB$; we also show the results for shorter vectors

[3] We modified the benchmark to allow a more fine-grained measurement concerning the number of processes and message sizes for which the tests are performed.

[4] SCI-MPICH is based on MPICH, version 1.2.0.

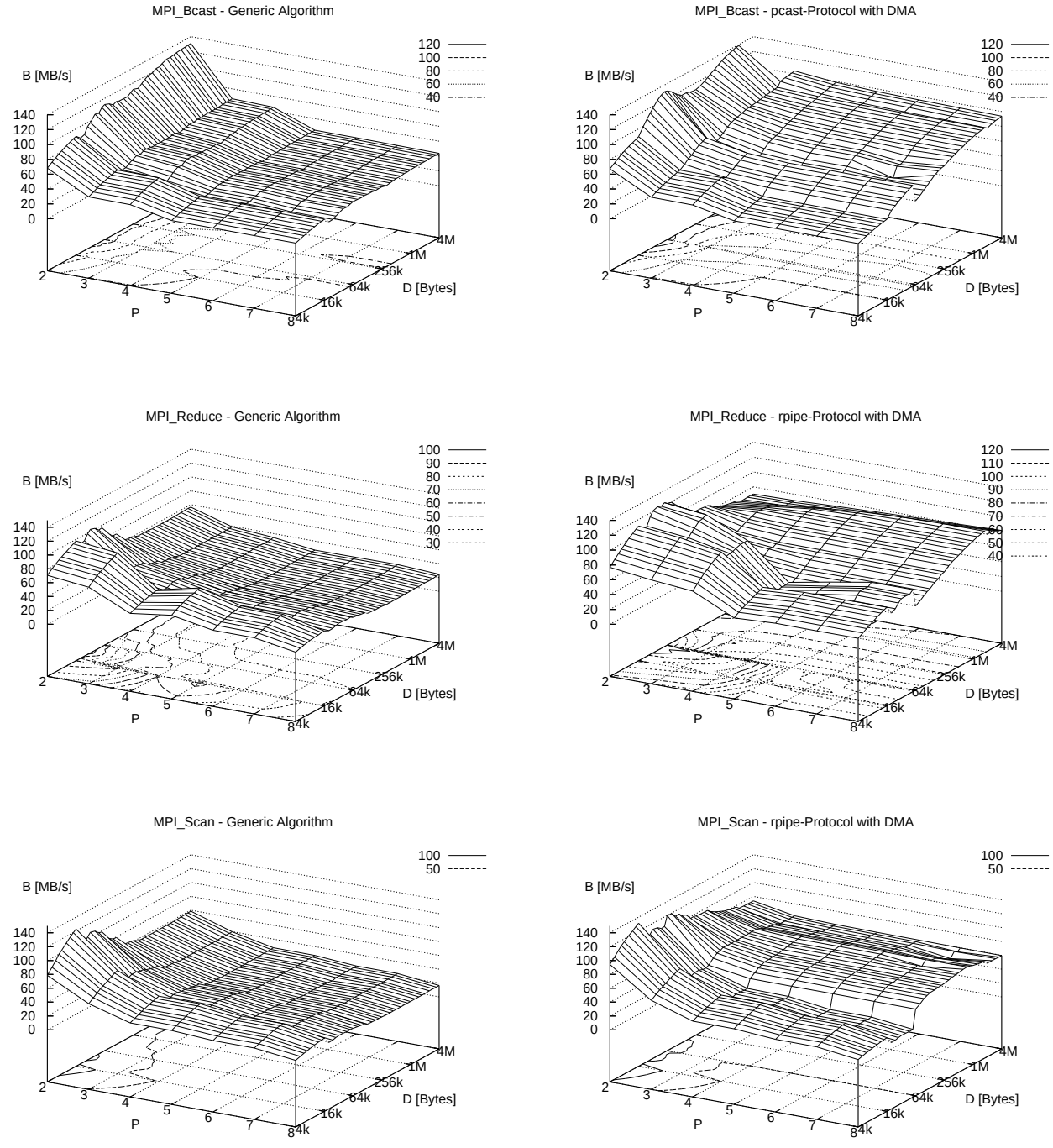

Figure 3. Experimental performance comparison between generic (left) and pipelined (right) collective operations: `MPI_Bcast`, `MPI_Reduce` **and** `MPI_Scan` **(from top to bottom)**

(using the generic algorithm) to see if this threshold is valid for all COs.

The results for `MPI_Bcast` show that the performance of the generic algorithm decreases with every new level of the binary tree used as communication topology (steps at $P = 3$ and $P = 5$). For $P = 8$ and $D_v = 4\,\mathrm{MB}$, a value of $B_{P=8} = 43,7\,\mathrm{MB/s}$ is achieved. Using the *pcast* protocol, the corresponding performance decreases only slightly for increasing values from $B_{P=3} = 103,9\,\mathrm{MB/s}$ down to $B_{P=3} = 93,7\,\mathrm{MB/s}$. The performance of the *pcast* protocol is higher than for the generic algorithm for all tests performed, even for small values of P and D_v. Depending on the vector length, the *rpipe* protocol is observed to be between 20% and more than 100% faster.

We observe the highest bandwidth values for $P = 2$ for both protocol variants. For the generic algorithm, $B_{P=2} \simeq 2 \cdot B_{P=3}$ applies because for $P = 3$, two instead of just one serialized transfers of the complete message have to be performed. In contrast, the *pcast* protocol performs a pipelined transfer (in this case, between just two processes) and by this achieves a bandwidth which is about 10% higher than for the generic algorithm. The significant performance decrease of about 25% for the transition from $P = 2$ to $P = 3$ shows that the bottleneck of the *pcast* protocol is not the first or last process in the pipeline, but the processes which need to forward the data. This applies if a value of D_b is chosen which has a lower transfer bandwidth for DMA than for PIO. Increasing D_b might reduce this effect, but leads to longer pipeline fill delays. This will be further evaluated by the results of the models presented in Chapter 5.3.

Finally, the performance of *pcast* protocol increases with the vector length D_v. This can be expected from pipeline processing: the impact of the P pipeline processing phases for the fill time decreases relatively to the number of the end-to-end data throughput stages D_v/D_b (see (2), first factor).

The bandwidth `MPI_Reduce` for the generic algorithm is about 50% of the bandwidth for the generic `MPI_Bcast`, but the run of the curve looks similar. This shows that the amount of time for sending the vector is about equal to the time needed to combine the two vectors. For the *rpipe* protocol, the run of the curve looks different. For $D_v = 4\,\mathrm{MB}$, the bandwidth is reduced by about 15%. However, for $D_v < 2\,\mathrm{MB}$, `MPI_Reduce` achieves even slightly higher performance than `MPI_Bcast`, with a maximum at $D_v = 1.8\,\mathrm{MB}$. This shows that the overlapping of computation and communication works well. The reason for the performance decrease for long vectors requires further evaluation, as it does not match with the general pipelining characteristics and there is no obvious reason for the communication or computation performance to decrease for $D_v > 2\,\mathrm{MB}$. The maximal performance ratio between the *rpipe* protocol and the generic algorithm is 4 (observed for

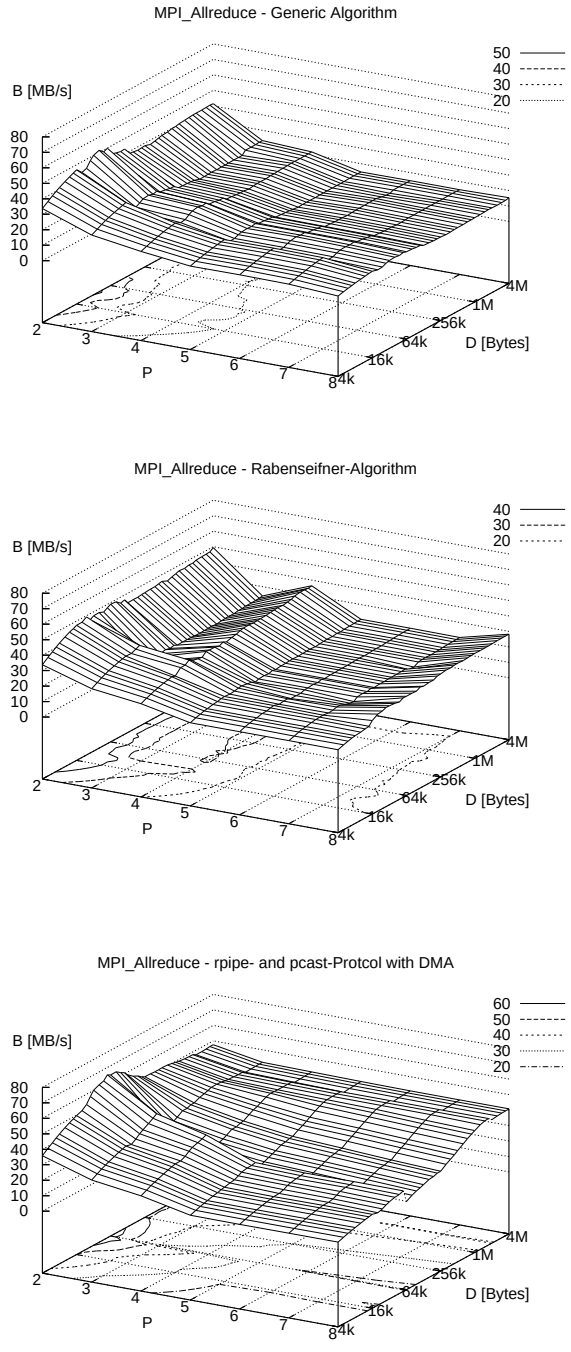

Figure 4. Experimental performance comparison of different `MPI_Allreduce` implementations: generic algorithm, Rabenseifner algorithm and *rpipe*-protocol (from top to bottom)

$P = 8$ and $D_v = 1.8\,\mathrm{MB}$). The average ratio is about 3.

The results for `MPI_Scan` are similar to the results for `MPI_Reduce` for both protocols. However, due to the additional local copy operation, both protocols suffer a 25 to 33% performance decrease relative to `MPI_Reduce`. The peak performance ratio exceeds 4 for $P = 8$ and $D_v = 640\,\mathrm{KB}$: the *rpipe* protocol achieves $70.5\,\mathrm{MB/s}$ while only $15.7\,\mathrm{MB/s}$ can be delivered by the generic algorithm.

The last CO that was improved via pipelining is `MPI_Allreduce` which uses both protocols, *rpipe* followed by *pcast*. In Figure 4, we do not only show the results for the generic algorithm and the pipelined protocols, but also include the results for the improved generic algorithm as proposed by Rabenseifner. The numbers for both, the generic algorithm and the pipelined protocols, reflect the serialized execution of `MPI_Reduce` and `MPI_Bcast` which results in

$$B_{allreduce} = \frac{1}{\frac{1}{B_{reduce}} + \frac{1}{B_{bcast}}}.$$

The results for the Rabenseifner implementation look differently. As it uses a binary exchange pattern to create a higher parallelism for the combine operation, it works best for process counts which are an exponent of 2. For other process counts, the performance decreases significantly as extra communication steps are required. Compared with the generic algorithm using a tree-topology, this leads to about a 100% performance increase for $P = 2^n$, but to only slightly better performance for other values of P. For $P = 2^n$, the pipelined protocols deliver a performance up to 33% higher than Rabenseifner for $D_v > 256\,\mathrm{MB}$. Below this threshold, Rabenseifner's implementation is about 20% faster. For other values of P, however, the pipelined protocols are always faster than Rabenseifner and deliver a performance which is up to twice as high.

5.2 Comparison with ScaMPI

Next to the comparison with the generic algorithms, it is worthwhile to compare the new pipelined protocols of SCI-MPICH with ScaMPI's performance for the same operations. ScaMPI [6]is a commercial MPI implementation for the SCI interconnect. We could run a direct comparison on a Cluster of Pentium 4 systems with Intel i860 chipset. This experiment took place in March 2002, using the most current version of ScaMPI at this time. Due to space limitations, we can only give the key performance values shown in Table 1 (the complete comparison is available at http://www.mp-mpich.de). Although, the test platform has a low DMA performance of only $150\,\mathrm{MB/s}$, it shows that SCI-MPICH performs better than ScaMPI with respect to the duration of the operations for this test. The relatively small performance advantage for `MPI_Allreduce` is related to ScaMPI's use of the Rabenseifner algorithm [REF]. However, it has been shown that this algorithm does only perform well for process numbers 2^n.

	ScaMPI	SCI-MPICH
`MPI_Bcast`	77,4	53,4
`MPI_Reduce`	96,2	44,8
`MPI_Allreduce`	98,1	86,0

Table 1. Comparison of the performance of ScaMPI and SCI-MPICH (duration of the complete operation in ms for 8 processes and 4MB vector length)

5.3 Results from Modeling

The modeling of the pipeline protocols opens a wide range of possible explorations for performance effects of varied runtime parameters and validation of the implementation. For this paper, we confine ourself to the following questions as they can not easily be answered by experiments on the available hardware:

- How relevant is the pipeline block size D_b for the effective performance? We vary D_b over P for different vector sizes D_v. This will show us if a single value for D_b is sufficient, or how it may be chosen dynamically.
- How does the performance develop for increasing values of P? We simulate this for different vector sizes D_v. We also compare the performance of the pipelined protocol with the generic algorithm. This will show us if switching points between these protocols should be established.
- Which influence does the flow control have on the achieved performance? This will give us a hint if other flow control techniques can be used efficiently.

We have performed the simulation for the *pcast* protocol as a first approach of evaluation of the characteristics of pipelined transfers, using the more simple model. Figure 5 shows three charts with the results which we will use to answer the three questions above.

The top chart shows the bandwidth per process B_{pcast} for $D_v = 1\,\mathrm{MB}$ over D_b for different process counts P. We can see that the choice of D_b has a significant impact on B_{pcast}. The optimal value of D_b for the evaluated process counts varies between $20\,\mathrm{KB}$ for $P = 8$ and $4\,\mathrm{KB}$ for $P = 256$. Additionally, it shows that the range of D_b, in which it delivers nearly optimal performance, decreases

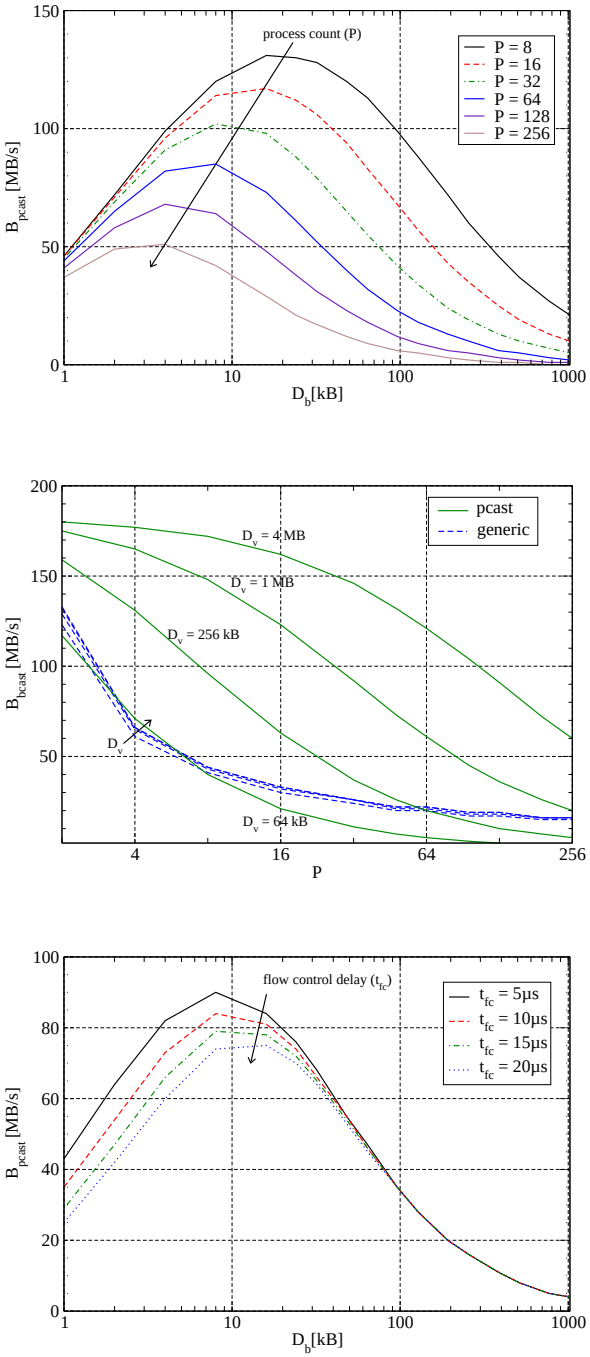

Figure 5. Simulated performance comparison of *pcast* protocols, varied over pipeline block size, process count, and flow control delays (from top to bottom)

from about $50\,\mathrm{KB}$ down to $4\,\mathrm{KB}$ if 256 instead of 8 processes are used. This means that choosing the value for D_b becomes more important the more processes are used as the negative performance impacts for the same absolute deviance from the optimal value for D_b increase. Therefore, D_b needs to be chosen dynamically for each MPI communicator, depending on the number of processes in this communicator. SCI-MPICH does this currently using a simple approximation via a linear equation; more sophisticated methods are possible.

The middle chart compares the bandwidth per process B_{bcast} of the *pcast* protocol with the generic algorithm for different vector lengths D_v, varying over the process count P. For the generic algorithm, B_{bcast} remains nearly constant for different value of D_v because the point-to-point message bandwidth is nearly constant for the chosen values of D_v, too. In contrast, the performance of the *pcast* protocol depends heavily on both the process count P and the vector length D_v. This difference in the scaling characteristics doesn't come suprising as the generic algorithm scales with $O(logP)$, while the *pcast* protocol scales with $O(P)$. Again, it depends on the performance characteristics of the specific platform which algorithm is suited better for given values of P and D_v. For the shown platform, the *pcast* protocol should be chosen for $D_v > 64\,\mathrm{KB}$.

The bottom chart answers the last of our questions by showing B_{pcast} over D_b for different flow control delays t_{fc} between each transfered block for $D_v = 1\,\mathrm{MB}$. On the tested platform, $t_{fc} = 5\,\mu s$ applies. We can see that for the optimal block size $D_b = 8\,\mathrm{KB}$, the achieved bandwidth by about 20% if the flow control delay is increased by a factor of 4. This indicates that it is feasible, but not without impact, to use less efficient means of flow control.

6 Summary and Outlook

We have shown that it is possible to efficiently implement a number of collective operations in MPI by overlapping CPU-driven data transfer and combine operations inside a node and DMA-driven data transfers between nodes. Using theses transfers, we could employ new communication protocols for pipelined 1:N- and N:1-operations like `MPI_Bcast`, `MPI_Reduce` and `MPI_Scan`. Even for the N:N-operation `MPI_Allreduce`, we could achieve improvements in comparison with highly optimized non-pipelined algorithms. This shows most significantly for process numbers which are not a power of two.

Next to the experiments with the implementation of the pipelining protocols for MPI collective operations, we presented a model of these protocols which allows to predict the throughput for arbitrary process counts and data transfer performance settings. This way, we could estimate for which cases pipelining will be more efficient than the con-

ventional tree-based algorithm. However, the wide range of parameters which influence the performance and their mutual dependencies make it difficult to determine the best choice as the underlying performance characteristics vary between the platforms. An approach of automatically tuning as presented by Vadhiyar et.al. [21] seems to be a solution for this problem.

For large process counts, the concept of a single linear pipeline was shown to be less efficient than the tree-based algorithms if the vector length is not big enough. It might be worthwile to switch to more complex pipeline concepts which combine the advantages of pipelined transfers and reduced communication steps by splitting up the single pipeline into multiple sub-pipelines as it is done in the "fractional tree" algorithm proposed in [17].

While our implementation is based on SCI, the concept can be transfered to other high-speed interconnects with remote memory access (RDMA) capabilities. The complete source of the software used to achieve the results presented in this paper is available at http://www.mp-mpich.de.

References

[1] R. Bhoedjang, T. Rühl, and H. E. Bal. Efficient Multicast on Myrinet using Link-Level Flow Control. In *Proceedings of the 1998 International Conference on Parallel Processing*, pages 381–390, Minneapolis, August 1998.

[2] Dolphin Interconnect Solutions. *SISCI Interface Specification*, 2.1.1 edition, May 1999.

[3] S. Fleischmann and D. Golan. MPI Collective Communication on the Convex Exemplar SPP-1000 Series Scalable Parallel Computer. Presentation at MPI Developers Conference (MPIDC95), 1995.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[5] L. P. Huse. Collective Communication on Dedicated Clusters of Workstations. In J. D. et.al., editor, *Recent Advances in PVM and MPI*, number 1697 in LNCS, pages 469–476, September 1999.

[6] L. P. Huse, K. Omang, H. Bugge, H. Ry, A. Haugsdal, and E. Rustad. *SCI: Scalable Coherent Interface. Architecture software for high-performance compute clusters*, chapter ScaMPI - Design and Implementation. Number 1734 in LNCS. Springer, 1999.

[7] IEEE, editor. *Standard for Scalable Coherent Interface (SCI)*. Number 1596 in IEEE Standards. The Institute of Electrical and Electronics Engineers, Inc., 1992.

[8] M. C. Liaanen and H. Kohmann. *SCI: Architecture and Software for High Peformance Compute Clusters*, chapter Dolphin SCI Adapter Cards, pages 71–86. Number 1734 in LNCS. Springer, 1999.

[9] G. R. Luecke, B. Raffin, and J. J. Coyle. The Performance of MPI Collective Communication Routines for Large Messages on the Cray T3E-600, the Cray Origin 2000, and the IBM SP. *Journal of Performance Evaluation and Modelling for Computer Systems*, July 1999.

[10] Message-Passing Interface Forum. MPI: A Message-Passing Interface Standard . http://www.mpi-forum.org, 1995.

[11] Message-Passing Interface Forum. MPI-2: Extentsions to the Message-Passing Interface. http://www.mpi-forum.org, 1997.

[12] P. Mitra, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts. Fast Collective Communication Libraries, Please. Technical Report TR-95-22, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712-1188, June 1995.

[13] S. Oral and A. George. Multicast performance analysis for high-speed torus networks. In *27th Annual IEEE Conference on on Local Computer Networks (LCN), Workshop on High Speed Local Networks*, pages 619 – 628. IEEE Computer Society, November 2002.

[14] Pallas GmbH. Pallas MPI Benchmark 2.0. http://www.pallas.com, 2002.

[15] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. Hardware- and Software-Based Collective Communication on the Quadrics Network. In *IEEE International Symposium on Network Computing and Applications 2001 (NCA 2001)*, Boston, MA, February 2002.

[16] R. Rabenseifner. A new optimized MPI reduce algorithm. High-Performance Computing-Center, November 1997. http://www.hlrs.de/mpi/myreduce.html.

[17] P. Sanders and J. F. Sibeyn. A Bandwidth Latency Tradeoff for Broadcast and Reduction. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Proceedings of 6th International EuroPar Conference*, number 1900 in LNCS, Munich, August 2000.

[18] F. Seiffert, J. Worringen, and W. Rehm. Using Arbitrary Memory Regions for SCI Communication. In *SCI Europe Conference*, Dublin, October 2001. Trinity College.

[19] SISCI Consortium. Standard Software Infrastructures for SCI-based Parallel Systems (SISCI). http://www.parallab.uib.no/projects/sisci, August 1997.

[20] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH. Submitted to publication for Euro-PVM/MPI 2003, May 2003.

[21] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra. Automatically Tuned Collective Communications. In J. E. Donnelley, editor, *Proceedings of Supercomputing 2000*, Dallas, November 2000.

[22] J. Worringen and T. Bemmerl. MPICH for SCI-connected Clusters. In *SCI Europe Conference*, Toulouse, September 1999. Conference Stream of EuroPar Conference.

[23] J. Worringen, F. Seiffert, and T. Bemmerl. Efficient Asynchronous Message Passing via SCI with Zero-Copying. In *SCI Europe Conference*, Dublin, October 2001. Trinity College.