Evaluation of Optimized Barrier Algorithms for SCI Networks with Different MPI Implementations

Boris Bierbaum, Georg Wassen, Stefan Lankes, Thomas Bemmerl
Chair for Operating Systems
RWTH Aachen University
Kopernikusstr. 16
52056 Aachen, Germany
{bierbaum,wassen,lankes,bemmerl}@lfbs.rwth-aachen.de

Abstract

The SCI Collectives Library is a new software package which implements optimized collective communication operations on SCI networks. It is designed to be coupled to different higher-level communication libraries (especially MPI implementations) by adapter modules, thereby giving them access to these optimized collectives. In this work, we present the design of the SCI Collectives Library and of adapter modules for Open MPI and NMPI. We also describe various barrier algorithms which we have implemented for this library and compare their performance to one another and to the barrier performance of MPI implementations which include support for SCI.

1. Introduction

The performance characteristics and the design of low-level interfaces vary greatly between different local area networks, such as SCI [**?**], Myrinet, and Ethernet. Therefore, for an MPI implementation to achieve good application performance on a cluster equipped with such a network, proper support for a specific network architecture must be developed. This has led to MPI implementations which are tailored to a specific high-speed network, e.g. SCI-MPICH [**?**] for SCI and MPICH-MX for Myrinet. Unfortunately, this limits the users of a cluster to a certain MPI implementation, even if other characteristics of it (like thread-safety or tool support) may be unsatisfactory. These specific MPI implementations usually contain collective operations which are highly optimized for the network architecture which the implementation supports.

The SCI Collectives Library provides optimized collective communication routines for SCI networks and is designed to be adaptable to different MPI libraries. This gives users of SCI clusters more freedom in their choice of an appropriate MPI implementation. We aim to show with the implementation of this library, that it can be coupled to different MPI libraries without a significant performance penalty. The library is also meant to serve as a tool for doing research in the area of collective algorithm design and implementation on SCI networks. So far, we have implemented various barrier algorithms in the SCI Collectives Library and evaluated their performance.

The structure of this paper is as follows: Sec. 2 refers to prior work about barrier algorithms for SCI networks and describes the MPI implementations mentioned in the later sections. Sec. 3 details the architecture of the SCI Collectives Library and the way it can be adapted to different MPI libraries. The barrier algorithms and related benchmark results are are described in Sec. 4, Sec. 5 concludes the paper.

2. Related Work

[**?**] compares various barrier implementations on an SCI cluster. Unfortunately, this comparison is biased by the best-performing barrier using direct writing to remote SCI memory (see Sec. 4) while the other ones are based on `MPI_Send` and `MPI_Recv` and therefore suffer from additional overhead. In [**?**], SCI clusters of SMPs are considered. For the implementation of a barrier on such a system, the author argues in favour of a dedicated process per node doing the network-wide synchronization with the node-internal parts of the barrier performed before and after that, an approach we followed in the SCI Collectives Library. The algorithms and data layout for barrier synchronization presented in [**?**] served as the starting point for our implementation.

2.1. SCI-MPICH

SCI-MPICH, part of the MP-MPICH software package, primarily is a channel device for MPICH, called ch_smi [**?**]. It is based on the SMI library [**?**] which in turn makes use of the low-level SISCI [**?**] interface for SCI. SCI-MPICH implements optimized point-to-point and collective operations for SCI networks [**?**, **?**]. Because of its MPICH heritage, it provides neither thread safety nor full support for the MPI-2 standard.

2.2. NMPI

NMPI [**?**] is based on MPICH2 and implements a channel device with optimized point-to-point operations for SCI. Compared to SCI-MPICH, it has the advantages which MPICH2 has over MPICH, but it does not contain optimized collective algorithms. Instead, the standard MPICH2 collective algorithms [**?**], which are designed for switched networks and not for the ring or torus topologies of SCI clusters, are used for SCI, albeit on the basis of the fast point-to-point operations.

2.3. Open MPI

Open MPI [**?**] is an MPI implementation which aims to integrate the features of several older software distributions like FT-MPI, LA-MPI, and LAM/MPI into a single package. It also differs from MPICH and MPICH2 in its component based architecture, called *Module Component Architecture* (MCA) [**?**]. The MCA allows for the inclusion of new functionality and the replacement of software components without the need to make source code changes to the Open MPI distribution, because it can detect and activate components implemented as shared libraries at runtime.

Point-to-Point communication with Open MPI on an SCI cluster can be done via sockets [**?**] and an implementation on top of SISCI has been considered [**?**], but to our knowledge no collective communication operations tailored to SCI networks are available for Open MPI yet.

3. Architecture of the SCI Collectives Library

3.1. Overview

Fig. 1 shows the design of the SCI Collectives library from a high-level point of view. The collective algorithms are implemented inside of the `scicoll` library, which is coupled to the MPI libraries via the respective adapters. For its algorithms, the `scicoll` library calls point-to-point operations from the higher-level libraries or directly uses the SISCI interface, when this is preferable (see Sec. 4).

Figure 1. Architecture of the SCI Collectives Library

3.2. Interface Design

To provide multiple MPI implementations with optimized point-to-point operations on a specific architecture, the uDAPL interface [**?**] can be implemented for this architecture, since there are several MPI implementations which can make use of this API, e.g. Intel MPI [**?**] and Open MPI. For collective communication, there is currently no such interface available, which made the development of adapter modules for different MPI implementations necessary. This situation also motivates the design of an interface for the `scicoll` library which is suitable to be used by such adapter modules. The interface for the SCI Collectives Library has the following main properties:

- Support for all MPI collective operations (which are not all implemented yet)
- Functions to register point-to-point operations used as a basis for the collective algorithms
- We plan to provide all collectives also in asynchronous versions compatible to LibNBC [**?**] (and to support non-MPI collectives which may need this)
- The possibility for the user to choose between different algorithms for a collective operation (if available)

The SCI Collectives API provides functions to initialize and finalize the library and to create and destroy groups of processes. For the creation of such a group, an adapter must provide the pointers to some communication functions (derived from MPI blocking and nonblocking send and receive) and can provide settings to override the default algorithm choice and parameters. The results of precalculations influencing the collective algorithms are stored in internal data structures. A pointer to that data is returned to the adapter and must be provided to the collective calls.

3.3. Adapter Modules

So far, we provide adapters for Open MPI and NMPI. We plan to develop additional adapter modules and provide documentation and sample source code to enable the development of third-party modules.

The adapters control the initialization of process groups during the creation of each MPI communicator. They implement send and receive functions using the same internal functions of the MPI library that are also used inside of `MPI_Send`, `MPI_Recv` etc. Furthermore, they contain

functions for the collective operations that are called by the MPI library and which in turn call the optimized algorithms of `scicoll`. This way, the collective functions are called with almost no overhead during execution.

Open MPI. In Open MPI, the collective functions are handled by the `coll` framework. It is able to deal with multiple available collective components which implement a subset or all of the collective routines. The adapter is currently based on Open MPI 1.2.1 with collective framework version 1.0.0. Upon the creation of a new MPI communicator, the `coll` framework queries the available components to find the one most suitable for this particular setup. That component is then used to create and initialize a module, which is an instance of the component. The module returns pointers to its collective functions to the framework. Functions not implemented by this module are automatically realized by generic algorithms from the included `basic` component. The current version of the collective framework can use multiple components and doesn't need to fall back to the generic functions if the best suited component provides only a few collective operations.

The Open MPI adapter is available as a shared library which maps collective routines required by Open MPI to the `scicoll` interface. If put in the correct place, it is detected by the framework and loaded automatically. Open MPI supports MCA parameters that can be set in configuration files and at the command line to influence the behaviour of components. The adapter reads its parameters and hands them to the library. This way, specific algorithms or modes can be selected by a user.

NMPI. The origin of NMPI, MPICH2 supports the replacement of its collective functions by hooks that are called each time a new communicator is created. A hook is a predefined macro which is overwritten by the adapter to initialize the library. Only those collective operations which are implemented by `scicoll` are replaced by optimized versions while the others use their original algorithms. The NMPI adapter is realized as a source code patch, therefore a rebuild of NMPI is required. Parameters can be passed to the NMPI adapter via a configuration file.

4. The Barrier Implementation

In MPI [**?**], a barrier is defined as a synchronization among a group of processes which blocks the caller until all group members have entered the corresponding call. Thus no process can proceed with execution after the barrier while there are processes which have not entered the barrier yet.

In barrier algorithms, a process marks its arrival at the barrier by emitting some kind of *signal* which must then be passed to all the other processes until every process has received enough signals to be sure that each other process has reached the barrier. Thus, a signal carries the information about the arrival of one or more processes at the barrier. The generic barrier algorithms in Open MPI and NMPI use message passing functions to send and receive such signals. To avoid the overhead of the point-to-point communication, we directly use the SISCI API for our barrier implementation.

SISCI allows the creation of SCI memory *segments* which can be *exported* by a process A and *imported* by a process B running on a different node. After B has mapped the segment into its virtual address space, it can send data to A via CPU store operations with very low latency for small messages. A signal for a barrier algorithm can thus be realized by writing via a remote pointer.

The SCI adapters used by us (see Tab. 1) contain *stream buffers*, in which write gathering is performed for outgoing data. A *sequence check* must be done to detect failed data transfers, which are then repeated until the check succeeds. Each sequence check contains by default an inherent *store barrier* to force the completion of all pending transfers. These sequence checks take more than 5 μs, but failures are rare, while a remote write operation with a size of 4 bytes stalls the sender's CPU for about 200 ns. Therefore, it is preferable to protect as many data transfer operations as possible with a single check. Furthermore, our experiments revealed that a dedicated store barrier followed by a sequence check which has its inherent store barrier deactivated is 1.5 to 2 μs faster so that the combination of store barrier and fast sequence check takes below 4 μs.

4.1. Local Synchronization

If multiple processes are running on the same node, one of them is nominated *master* to communicate with the other nodes. The *slaves* synchronize with the local master by System V shared memory. Each slave sets a flag and the master waits until all have checked in before it synchronizes with the other nodes (Fig. 2). The check-in flags are aligned at the beginning of a cache-line so that least cache misses occur. After the remote synchronization, the master sets a single check-out flag the slaves are waiting for.

Figure 2. Shared Memory Check-in/-out of Local Processes

For any node n, this intra-node synchronization scales linearly with the number of processes P_n on the node . The master reads $P_n - 1$ flags from the local memory and writes a single one in addition to the remote synchronization. The slaves issue only a single write and read operation. This is very fast compared to the remote memory access and reduces the problem of the synchronization of P processes on

connected SMP nodes to the synchronization of N nodes (with $N \leq P$). [**?**]

4.2. Remote Synchronization

For the synchronization among the nodes, each master process exports a local SCI segment and imports the segments of the other nodes. Each flag is aligned at the top of the stream buffer size so that writing to that position makes the SCI adapter issue the network transfer instantly. The flags are always located at the receiver so that setting the flag requires one data transfer and waiting for the flag can be done by polling a variable in local memory.

The communication pattern is prepared during initialization and stored as barrier-data within the MPI communicator. During the barrier call, each process just executes the precomputed list of write and read operations.

Hierarchical Shared Memory Barrier. The *hsb* algorithm described in [**?**] and implemented in SCI-MPICH concentrates the arrival signals of all processes to a tree root (a *gather* pattern forming an f_{in}-ary tree) and *broadcasts* the arrival information in the opposite direction afterwards (via an f_{out}-ary tree). In SCI-MPICH, this algorithm is performed with $f = f_{in} = f_{out} = 8$. We re-implemented it and did an experimental evaluation to find out the optimal value of f on our cluster. Our experiments did not yet show any advantages of setups with $f_{in} \neq f_{out}$.

Figure 3. Hierarchical Shared Memory Barrier

As an example for the *hsb* algorithm, Fig. 3 shows the synchronization of seven nodes with a ternary tree ($f = 3$). Node 1 waits until 4, 5 and 6 have set their flags and sets afterwards its fan-in flag at node 0. After that one has detected all flags from 1, 2 and 3 it begins the fan-out process by setting the corresponding flags in these nodes. Node 1 was waiting for this event, promotes the signal to its children and returns from the barrier call.

Each node waits for up to f children by reading local memory until a flag is set. Except the tree root, a single remote write operation with sequence check and the waiting for the fan-out flag follows. Finally, the children are released by up to f remote write operations and a single sequence check. The effort of each process is highly scalable but the further down a node is located in the tree, the longer it has to wait until the signal is promoted to the tree root and back. This algorithm performs $2 \cdot \lceil \log_f N \rceil$ steps.

Exchange Algorithms. The exchange algorithms were inspired by the binary exchange barrier also presented in [**?**]. But the number of steps is bound by O($\log_2 N$) and each step requires a time consuming sequence check. To decrease their number, we generalized the binary exchange to an n-ary exchange (*nx*) so that the number of steps is bound by O($\log_n N$) which is better for $n > 2$. Each node issues $(n-1)$ flags, but as setting a flag just creates a small data transmission, this does not congest the network for a reasonable number of nodes.

Two modes are realized. In the first one, in each of $\log_n(N)$ steps, groups of n nodes synchronize themselves. The groups are assembled in a way that in each step representatives from different groups meet (Fig. 4) and convey the synchronizations they made before.

Figure 4. n-ary Exchange

This algorithm has the disadvantage that it requires N to be a power of n. If this precondition cannot be met, additional synchronization is needed, resulting in a total of $\lfloor \log_n N \rfloor + 2$ steps.

This overhead can be avoided by a different communication pattern (mode 2) derived from the (binary) dissemination algorithms presented in [**?**] and used in MPICH2. This algorithm requires $\lceil \log_n(N) \rceil$ steps. In step i, each node nid sets $(n-1)$ flags at the nodes $nid + n^i \pmod N$, $nid + 2 \cdot n^i \pmod N$ etc. and waits for the same number of local flags to be set by other nodes. Figure 5 illustrates the dissemination of the first node's signal for $n = 3$ on 9 nodes. In the same manner, the signal of each other node is dispersed.

Figure 5. n-ary Dissemination

With the factor n, the number of steps (thus sequence checks) and network packets can be influenced. In the above example, each node sends and receives 4 signals during 2 steps and a total of $9 \cdot 4 = 36$ remote write operations are executed. With $n = 9$, only a single sequence check is done, but each node sends and receives 8 signals, a total of 72 remote write operations.

4.3. Benchmark Results

To find optimal parameter settings, we benchmarked each barrier algorithm with the Intel MPI Benchmarks (IMB) on our development cluster, which is detailed in Tab. 1, using the software with the given versions. The results for the *nx* algorithms with different values for n are illustrated in Fig. 6, where each additional step can be seen as an abrupt increase in the time taken for the barrier. Within a constant number of steps, only a slight increase is visible. The graph for $n = 16$ shows that n should not be set greater than N. Up to 16 nodes, $n = N$ is the optimal parameter

selection for this algorithm. Above 16 nodes, $n = 2$ and $n = 16$ would require an additional step, but $n = 6$ can avoid this for up to 36 nodes. By extrapolation, we assume that the graphs for $n = N$ and $n = 6$ cross at about 20 nodes.

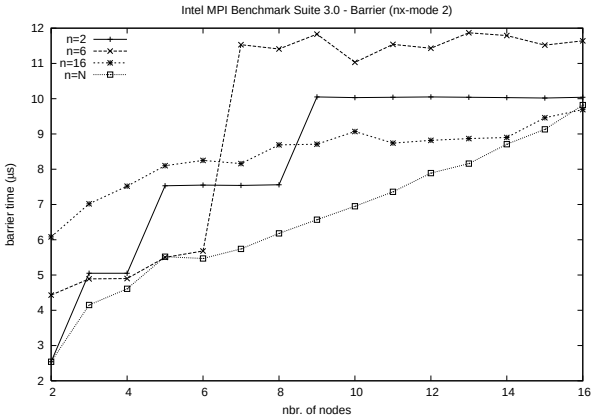

Figure 6. Optimization of the n-ary Exchange Algorithm

All presented barrier algorithms were analyzed likewise and by default, their parameters for a barrier concerning up to 16 nodes is set to the number of nodes (*hsb*: $f_{in} = f_{out} = N - 1$ and *nx*: $n = N$) in the SCI Collectives Library. We limit the parameters and set them to a much lower value (e.g. 6) above a certain number of nodes. This reduces the network traffic accepting additional steps but should be faster.

Therefore, both algorithms *hsb* and *nx* degenerate to sequential access (hsb) and all-to-all communication (nx) below 16 nodes. The additional steps do not appear until more nodes are involved, but the behaviour above that limit is currently extrapolated and must be confirmed by experiments.

We measured the performance of our algorithms in comparison to NMPI and the optimized barrier of SCI-MPICH on our cluster, the results are depicted in Fig. 7. The NMPI barrier took more than 15 μs on 3 nodes and about 37 μs on 16 nodes and is therefore not contained in that figure. The newly implemented *hsb* algorithm is slightly slower below 9 nodes than the same algorithm in SCI-MPICH. Above, SCI-MPICH needs two steps because of the fixed fan-parameter 8 so that our optimized algorithm becomes faster by using a single step. The new *nx* algorithm proved to be faster for any number of nodes up to 16 than the other barrier routines. Both *hsb* and *nx* algorithms show the same performance with the Open MPI and the NMPI adapter, demonstrating that neither of the two adapters introduces too much overhead.

If two processes are running on each node, the barrier time increases by 0.5 μs for the local synchronization (Sec. 4.1) independent of the number of nodes.

Hardware	
Processor	16 x Intel Pentium D 2.8 Ghz
RAM	2 GB per node
SCI	D352 adapter, 4x4 2D Torus
Software	
DIS Release	3.2.5
Open MPI	1.2.1
SCI-MPICH	rc-1.5
NMPI	1.2
Linux Kernel	2.6.18
IMB	3.0

Table 1. Hardware and Software used for Performance Evaluation

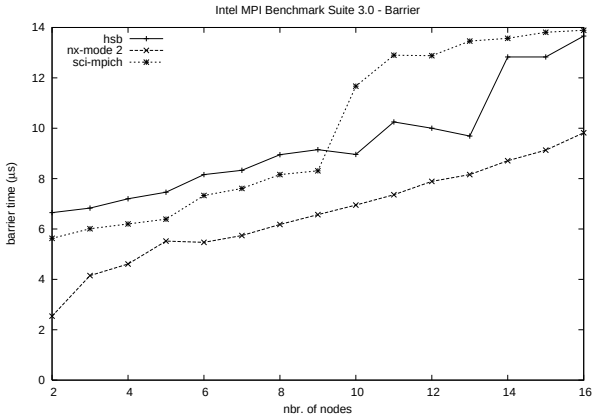

Figure 7. Comparison of Barrier Implementations

5. Conclusion, Outlook, and Acknowledgements

This work describes the SCI Collectives Library, a new software designed to provide optimized collective communication routines for SCI clusters to different MPI implementations. Our experiences in developing this library as well as the performance results we present for the barrier implementation show that this is indeed a feasible goal.

By implementing and evaluating various barrier algorithms in the SCI Collectives Library, we were able to show that the fan-parameter f for the *hsb* algorithm was set suboptimally in SCI-MPICH. In addition to that, the new *nx* algorithm shows significantly better performance on our clus-

ter than any other barrier algorithm we tested. Thus, there is now an improved barrier available for users of Open MPI and NMPI on SCI clusters.

We plan to implement a full set of collective communication patterns in this library to support all available collective functions of MPI. We also strive to develop adapter modules for MPI implementations which are not yet supported, especially Intel MPI. We will also explore the possibility to support other APIs besides MPI, which include collective communication routines, with our library. Concerning our barrier algorithms, we are currently conducting tests on a cluster with more nodes to gather new insights.

We would like to thank Intel Corporation for sponsoring this work and Dolphin Interconnect Solutions for their support.

References

[1] Dolphin Interconnect Solutions. *SISCI API User Guide, Version 1.0*, May 2001.

[2] M. Dormanns, K. Scholtyssik, and T. Bemmerl. A Shared-Memory Programming Interface for SCI Clusters. In H. Hellwagner and A. Reinefeld, editors, *SCI: Scalable Coherent Interface*, pages 281–290. Springer Verlag, 1999.

[3] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *LNCS*, pages 97–104, Budapest, Hungary, September 2004. Springer.

[4] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, Volume 17, Number 1:1–17, February 1988.

[5] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proceedings of Supercomputing 2007*, Reno, Nevada, November 2007.

[6] L. P. Huse. Collective Communication on Dedicated Clusters of Workstations. In *Proceedings of the 6th European PVM/MPI Users Group Meeting 1999 (EuroPVM/MPI)*, Barcelona, Spain, September 1999.

[7] L. P. Huse. MPI optimization for SMP based clusters interconnected with SCI. In *Proceedings of the 7th European PVM/MPI Users Group Meeting 2000 (EuroPVM/MPI)*, Lake Balaton, Hungary, September 2000.

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

[9] `http://www.intel.com/go/mpi`.

[10] J. Lentini, V. Pham, S. Sears, and R. Smith. Implementation and Analysis of the User Direct Access Programming Library. In *Proceedings of the 2nd Workshop on Novel Uses of System Area Networks, SAN-2*, February 2003.

[11] T. Mehlan, T. Hoefler, F. Mietke, and W. Rehm. Concepts for Integrating SISCI into Open MPI. In *Proceedings of the 1st Workshop Kommunikation in Clusterrechnern und Clusterverbundsystemen*, Chemnitz, Germany, November 2005.

[12] MPI Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputing Applications*, 1994.

[13] `http://www.nicevt.ru/research/nmpi`.

[14] F. Seifert and H. Kohmann. SCI SOCKET - A Fast Socket Implementation over SCI. [Available on WWW at http://www.dolphinics.com /pdf/whitepapers/sci-socket.pdf].

[15] J. M. Squyres and A. Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In V. Getov and T. Kielmann, editors, *Proc. of the 18th ACM International Conf. on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185, St. Malo, France, July 2004. Springer.

[16] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH. In *Proceedings of the 10th European PVM/MPI Users Group Meeting 2003*, volume 2840 of *LNCS*, pages 257–267, Venice,Italy, September 2003. Springer.

[17] J. Worringen. SCI-MPICH - The Second Generation. In *Proceedings of SCI-Europe 2000 (Conference Stream of Euro-Par 2000)*, pages 11–20, Munich, Germany, August 2000.

[18] J. Worringen. *Effizienter Nachrichtenaustausch auf speichergekoppelten Rechnerverbundsystemen mit SCI Verbindungsnetz*. doctoral thesis, RWTH Aachen, 2003.

[19] J. Worringen. Pipelining and Overlapping for MPI Collective Operations. In *Proceedings of the Workshop on High-Speed Local Networks (HSLN), in conjunction with 28th Annual IEEE International Conference on Local Computer Networks (LCN 2003)*, pages 548–557, Bonn/Königswinter, Germany, October 2003.