Collective Operations in NEC's High-performance MPI Libraries

Hubert Ritzdorf, Jesper Larsson Träff

C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
{ritzdorf,traff}@ccrl-nece.de

Abstract

We give an overview of the algorithms and implementations in the high-performance MPI libraries MPI/SX and MPI/ES of some of the most important collective operations of MPI (the Message Passing Interface*). The infrastructure of MPI/SX makes it easy to incorporate new algorithms and algorithms for common special cases (e.g. a single SX node, or a single MPI process per SX node). Algorithms that are among the best known are employed, and special hardware features of the SX architecture and Internode Crossbar Switch (IXS) are exploited wherever possible. We discuss in more detail the implementation of* `MPI_Barrier`*,* `MPI_Bcast`*, the MPI reduction collectives,* `MPI_Alltoall`*, and the gather/scatter collectives.*

Performance figures and comparisons to straightforward algorithms are given for a large SX-8 system, and for the Earth Simulator*. The measurements show excellent absolute performance, and demonstrate the scalability of MPI/SX and MPI/ES to systems with large numbers of nodes.*

1 Introduction

As usage of MPI, the *Message Passing Interface* [5, 14], in applications is maturing, also the so-called *collective operations* are gaining in importance, and MPI libraries should strive to incorporate the best possible algorithms with the best possible implementations. This is reflected in much recent activity on improving the performance of the collective operations in both public domain and vendor MPI libraries, see for instance [1, 3, 10, 15, 16, 17]. From the outset the NEC MPI/SX implementation for the SX series of parallel vector supercomputers has emphasized efficient, hardware specific implementations of the MPI collectives [4, 8], and in this respect MPI/SX is among the most advanced implementations of the MPI standard. In this paper we summarize recent developments to the implementation of the MPI collectives in MPI/SX and MPI/ES, and illustrate the performance of basic collective operations on recent NEC systems, namely the 72 node SX-8 system at HLRS (*Hochleistungsrechenzentrum Stuttgart*, Germany, and the *Earth Simulator* (ES) in Yokohama, Japan.

MPI/SX is developed at the NEC C&C Research Laboratories in St. Augustin, Germany, in collaboration with 1st Computers Software Division in Fuchu. The starting point for MPI/SX in 1997 was the then current MPICH implementation [7], but MPI/SX has since then evolved in its own directions. Since 2000 MPI/SX implements the full MPI-2 standard [23].

2 Principles and techniques

The collective operations of MPI perform synchronization, communication, and computation operations over given sets of MPI processes which in MPI are represented by distributed objects called *communicators*. Each collective operation must be called by all processes in the given communicator, and upon return the operation is completed from the returning process' point of view. Examples of such collective operations are barrier synchronization (`MPI_Barrier`), broadcast (`MPI_Bcast`), in which data from a given root process are distributed to the other processes in the communicator, all-to-all communication (`MPI_Alltoall`), in which all processes exchange data with all other processes, and reduction-to-root (`MPI_Reduce`). The MPI standard has 16 collective operations [14], capturing common communication and reduction patterns on both regular (i.e. same amount of data per process) and irregular data sets.

Multi-node SX systems are clustered systems of shared memory nodes with 8 vector processors per node, interconnected by the high-performance *Internode Crossbar Switch* (IXS) for communication between processors on different nodes. Processors on the same shared memory node communicate via the shared memory, and the memory system is sufficiently powerful that all processors can read and write

1-4244-0054-6/06/$20.00

MPI/SX on SX-8			
Comm.	Buf.	Latency	Bandwidth
Intra	local	$1.6\mu s$	$28.1\text{GBytes}/s$
	global	$1.6\mu s$	$30.7\text{GBytes}/s$
Inter	local	$4.7\mu s$	$11.7\text{GBytes}/s$
	global	$4.7\mu s$	$15.0\text{GBytes}/s$
MPI/ES on Earth Simulator			
Comm.	Comm. buf.	Latency	Bandwidth
Intra	local	$2.0\mu s$	$14.4\text{GBytes}/s$
	global	$2.0\mu s$	$15.8\text{GBytes}/s$
Inter	local	$5.2\mu s$	$9.3\text{GBytes}/s$
	global	$5.2\mu s$	$12.2\text{GBytes}/s$

Table 1. Communication latency/bandwidth for intra- and inter-node communication with MPI/SX and MPI/ES. Communication buffers are placed either in MPI *process local memory* **or in special, non-swappable** *global memory***.**

to/from memory at the same time. The shared memory nodes are not cache-coherent, but vector loads read directly from memory, which can be exploited to implement the necessary MPI level coherence. The IXS is a full, single-stage bidirectional crossbar across the nodes. For the SX-8 the nodes can be equipped with either one or two Remote Control Units (RCU), which control the IXS. With one RCU, a node can be involved in one ingoing and one outgoing data transfer at a time, and any two processors on different nodes can communicate simultaneously with all other such pairs of processors on different nodes. With two RCU's the communication capability is doubled. The ES has a single RCU on the nodes. The shared-memory communication latency is lower and bandwidth higher than for communication via the IXS. The MPI communication bandwidths achieved with MPI/SX and MPI/ES are shown in Table 1, with communication buffers allocated either in *process local memory* (by `malloc`) or in special, non-swappable *global memory* by the MPI-2 `MPI_Alloc_mem` function.

Efficient implementations of the MPI collectives must take these system characteristics into account. Since MPI processes running on the same node can simultaneously read/write to the shared memory, whereas only one process at a time can be involved in inter-node communication, different algorithms are applicable for intra-node than for inter-node collective communication. Furthermore, inter-node collective communication algorithms must take into account that MPI processes can be arbitrarily distributed over the nodes with, for pure MPI applications, typically more than one MPI process per node. Communication algorithms must aim to schedule communication across the IXS to keep the switch busy without causing contention or starvation, at the same time exploiting as much as possible the more powerful intra-node communication capabilities. IXS traffic should be minimized by efficiently distributing data and computation e.g. for the reduction collectives, and by avoiding redundant communication. To reduce the impact of latency for small messages, data to be exchanged between nodes can be combined. The following sections will show concrete examples of these techniques.

The NEC SUPER-UX operating system does not give an MPI process direct access to the user memory of other MPI processes, whether on the same node or on different nodes. Thus, instead of direct memory copy between the MPI processes, MPI communication of user data in process local memory is via specially allocated, non-swappable global memory segments. For processes on the same node a single extra copy is required per block transferred, whereas for the IXS transfer the blocks on both sending and receiving node have to be in global memory segments which entails a memory copy by both sending an receiving process. For large buffers pipelining is employed, and the extra memory copies can be effectively hidden. For data in non-swappable, global memory MPI communication can in most cases be done by a single memory copy operation. This is exploited by the algorithms for the collectives. To reduce collective latency a pre-allocated global memory buffer is used for small messages. To reduce global memory consumption dynamically allocated buffers are used for long messages, up to a certain limit. Beyond the limit the collective operations have to break data into blocks, and revert to pipelined or blocked algorithms. The actual sizes of static and dynamic buffers have been fine-tuned for the SX architecture.

Across nodes, the collective communication algorithms use point-to-point communication. The low-level communication primitives for this have the same, and in some cases extended functionality as the standard point-to-point primitives of MPI. The primitives realize communication between the aforementioned global memory segments by a single IXS read operation, and revert to the otherwise used methods for point-to-point communication if the communication buffers are not placed in global memory. Extended functionality can track completion of send and receive operations by message counters that are incremented upon successful completion of the operation. Further communication primitives makes it possible to perform operations concurrently with the IXS communication, like for instance preparing the next block for a pipelined algorithm. By this design porting the collective algorithms of MPI/SX to new systems is at first a matter of adapting these low level primitives. An example of such a communication primitive is

```
MPIR_Recv_func(buffer,count,type,
               rank,tag,
               comm,status,func,state);
```

which performs a blocking receive exactly as the `MPI_Recv` call. To save latency, no argument error checking is performed, and data type and communicator arguments are pointers instead of MPI objects in need of dereferencing. The argument `func` is a pointer to a function that is executed on the `state` argument, which (for inter-node communication) can be done concurrently with the IXS read.

The MPI/SX collectives have a component based design to further reduce latency and ease maintainability. Since the allocation of MPI processes to processors is static, some amount of case analysis can be performed at communicator creation time instead of at invocation time. New communicators are classified into cases according to whether the communicator spans only a single SX node, whether only one process per node is assigned (*flat* case), whether a global barrier counter has been assigned to the communicator (see Section 3.2), and others. Each of these special cases has a corresponding set of collective implementations, represented by a function table, which is assigned to (or built for) the communicator. Since the algorithms for the special cases are sufficiently different (single-node, flat case, multi-node), there is only a limited amount of code replication, and the code for each case can be kept simple(r) since it does not have to cater for all possible situations. Some cases are important enough that it makes sense to implement algorithms that might be difficult or less efficient for arbitrary communicators (see discussion in Section 3.4). One such case is the *flat* case, where the communicator contains only a single MPI process per node. This is a frequent case in applications using a hybrid programming style like for instance OpenMP coupled with MPI.

For collective intra-node communication, whether for the single-node or the multi-node case with more than one process per node, memory accesses are synchronized using combinations of binary trees (for propagating the value of a synchronization flag to a local root), vectorization (for resetting flag vectors), or symmetric barrier synchronization algorithms [9]. Synchronization across nodes is implicit via the special point-to-point communication primitives. Also in the case where a process fails to allocate the global memory segment for inter-node communication, explicit synchronization between nodes is not needed. Instead, the inter-node communication is done via memory allocated in process local memory, and only the intra-node part need be aware that insufficient global memory was available.

3 Algorithms, implementations and performance

In this section we describe in some detail the algorithms and implementation of some of the more important MPI collectives. We give absolute performance for fixed numbers of nodes, varying over the message length, and demonstrate scalability by keeping the message length fixed, and varying instead over the number of nodes. A proprietary benchmark, which tries to adhere to good benchmarking principles [6, 12], has been used for performance measurement. To get reproducible figures, the running times reported are minimum times over a number of repetitions of the time for the slowest process to finish. In some cases we compare to the performance of a straight-forward implementation (however, implemented in the framework of MPI/SX) as found for instance in the original MPICH implementation [7] in order to illustrate the gains possible by the use of more sophisticated algorithms tailored to the particularities of the SX architecture.

We let p denote the number of MPI processes in the communicator. We consider only regular distributions of the p processes over the nodes, and let N denote the number of nodes, and n the number of processes per node, such that $p = n \times N$. The message size per process is denoted by m (and reported in Bytes, where for instance one GBytes is 2^{30} Bytes). We are concerned only with data that are contiguous in memory (arrays of integers). Except where noted communication buffers have been allocated in global memory with `MPI_Alloc_mem`.

3.1 Single-node communicators

For communicators spanning only a single SX node, all communication and synchronization is performed in shared (global) memory. As explained, shared communication buffers are allocated outside of the process local memory. Synchronization flags are likewise pre-allocated in global memory. For `MPI_Barrier` a symmetric barrier synchronization algorithm is used as described for instance in [9] yielding an MPI software barrier time of less than 7μseconds for 8 processes on a single node. The necessary synchronization for other collectives are either by binary tree or by symmetric barrier synchronization algorithms. The collective algorithms themselves can take full use of the shared-memory, and pipelining is used for larger messages to exploit memory bandwidth better, and to limit the size of the reserved global memory segment. For single-node reduction operations like `MPI_Reduce` all processes on the node participates equally in the reduction of each block, e.g. each block is divided into n equal-sized segments. Since single-node communicators are of lesser importance for large, multi-node SX systems, this paper will from now on deal only with collectives for multi-node communicators.

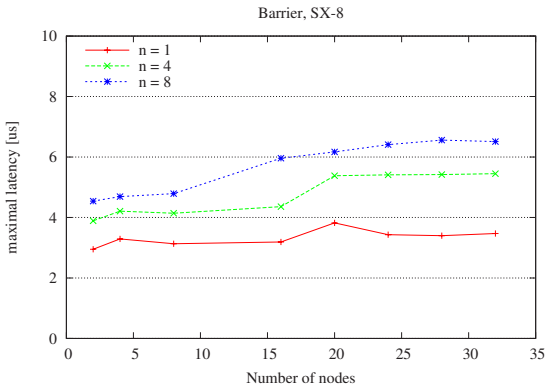

Figure 1. SX-8 barrier time with $n = 1, 4, 8$ processes/node for $N = 2, 4, \ldots, 32$ nodes.

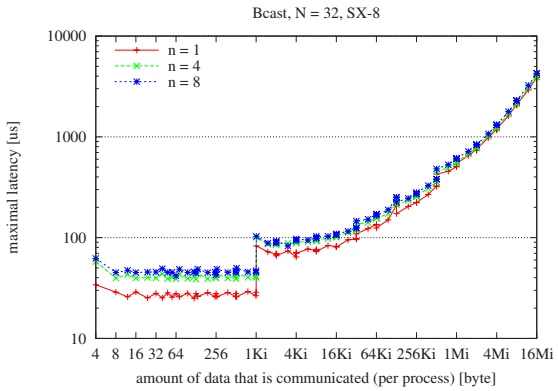

Figure 2. SX-8 broadcast time for fixed $N = 32$ and $n = 1, 4, 8$ processes/node and data size from 4 Bytes to 16 MBytes.

3.2 Multi-node barrier

For multi-node communicators barrier synchronization can rely on special hardware support with a software barrier based on simultaneous binomial trees [2] as fall back. The IXS provides a limited number of *global barrier counter/flag* (GBCF) registers that can be associated with an MPI job. The counter can be preset to a given value, and provides a constant time (i.e. independent of the number of processes) atomic decrement operation. When the counter reaches zero, the flag value is toggled (from 0 to 1 or *vice versa*), and the counter is set back to the preset value. To implement `MPI_Barrier` the counter is preset at communicator creation time to the number of nodes spanned by the communicator. For nodes with more than one MPI process an intra-node barrier is performed, using pairs of *communication registers* (CR) to emulate the functionality of the GBCF, and the process detecting successful intra-node synchronization is responsible for decrementing the global barrier counter. This process waits for the global barrier flag to toggle, and signals successful intra-node synchronization to the other processes on the node by toggling the CR flag.

The performance of the hardware based `MPI_Barrier` for the SX-8 is shown in Figure 1. The synchronization time is close to constant, and ranges from about 3μ`seconds` for one process/node to about 6μ`seconds` for 8 processes/node.

3.3 Multi-node broadcast

Two different algorithms are used for multi-node communicators depending on the size of the message to be broadcast. For small messages the best possible algorithm is a binomial tree, whereas a pipelined binary tree algorithm is used for messages beyond a certain threshold. To reduce inter-node traffic, broadcast trees are across nodes, not MPI processes, with an arbitrary process chosen as local root on each node. The pipelined binary tree algorithm is well-suited to the case with more than one MPI process per node. When a new block is received in a global memory segment, it can be read by the non-root processes on the node concurrently with the local root sending to its child nodes, and with the reception of the next block. Thus, for longer messages the broadcast time should ideally be independent of n, the number of processes per node. This behavior is shown Figure 2. The difference between $n = 4$ and $n = 8$ is less than 10%.

For the binomial tree broadcast the broadcast time is proportional to $\lceil \log_2 N \rceil$, whereas for large enough m the broadcast time of the pipelined binary tree is independent of N. The scalability behavior of `MPI_Bcast` for 8 Bytes, 1 MBytes and 16 MBytes is shown in Figure 3. For $m = 16$ MBytes the increase from 16 to 32 nodes is less than 10%, and this decreases as m and N increase.

Recently, an optimal algorithm which has the theoretical potential of being a factor of two faster than the pipelined binary tree by exploiting fully the bidirectional communication capability of the IXS has been developed [22, 21], but is not yet included in MPI/SX. Like the pipelined binary tree this algorithm also sends and receives the broadcast data in smaller blocks. Copying previous and next block in and out of the global memory segment used for communication can be overlapped with reception of the current block (using the `MPI_Recv_func` primitive), and other processes on the node can concurrently read the previous block received by the local root processes.

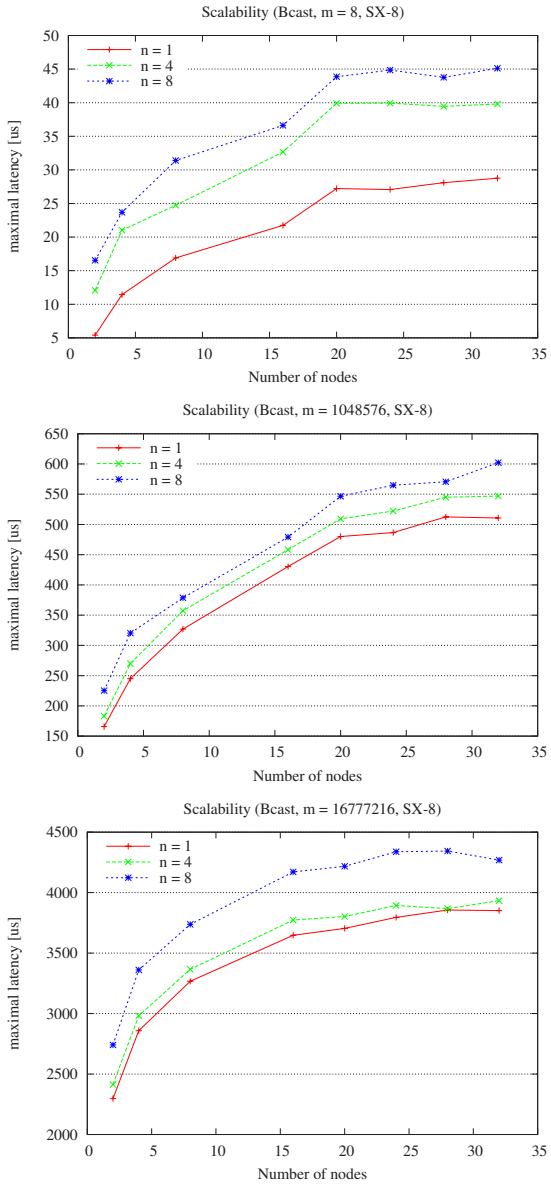

Figure 3. SX-8 scalability for `MPI_Bcast` for 8 Bytes (top), 1 MBytes (middle) and 16 MBytes (bottom) for $N = 2, 4, \ldots, 32$ nodes and $n = 1, 4, 8$ processes/node.

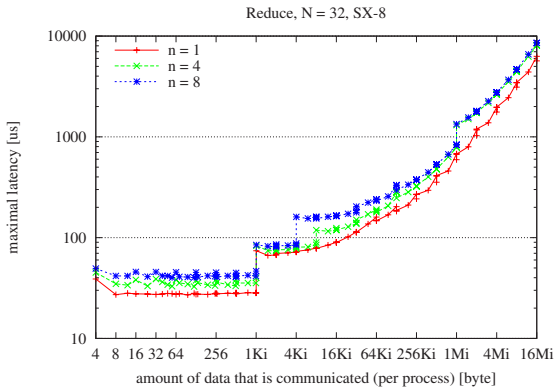

Figure 4. SX-8 reduction time for fixed $N = 32$, $n = 1, 4, 8$ processes/node and data size from 4 Bytes to 16 MBytes. For $n = 1$ a special, *flat* algorithm is used.

3.4 Reduction operations

The reduction-to-root collective `MPI_Reduce` can be thought of as an inverted broadcast, and currently similar algorithms are used in MPI/SX. For small data sizes binomial reduction trees are optimal, while pipelined binary trees are used for longer messages. For nodes with more than one MPI process the reduction to be performed before data are passed upwards in the tree is done in parallel using a shared memory algorithm like in the single-node case. As shown in Figure 4, beyond 1 MBytes there is virtually no difference between the cases with $n = 4$ and $n = 8$ processes per node, even less so than was the case for `MPI_Bcast`. Scalability of `MPI_Reduce` is illustrated for 8 Bytes, 1 MBytes and 16 MBytes in Figure 5.

For communicators spanning multiple nodes with exactly one process per node (the *flat* case), special algorithms based on *recursive halving* with a butterfly communication pattern have been implemented for `MPI_Reduce` (see [26] for an early application of this technique for hypercubes), as well as for `MPI_Allreduce` and `MPI_Reduce_scatter` with very good results [11, 20] (for arbitrary numbers of nodes, not only powers of two). Especially for `MPI_Allreduce` (and `MPI_Reduce_scatter`, not shown here) the corresponding reduction time is significantly smaller for the flat case, as can be seen in Figures 4 and 6. For $m = 16$ MBytes the difference between the flat and the non-flat case is more than a factor 5. These algorithms are also used for possibly non-commutative user-defined operators, where a canonical reduction order is mandated by the MPI standard.

For large data the recursive halving algorithms are not

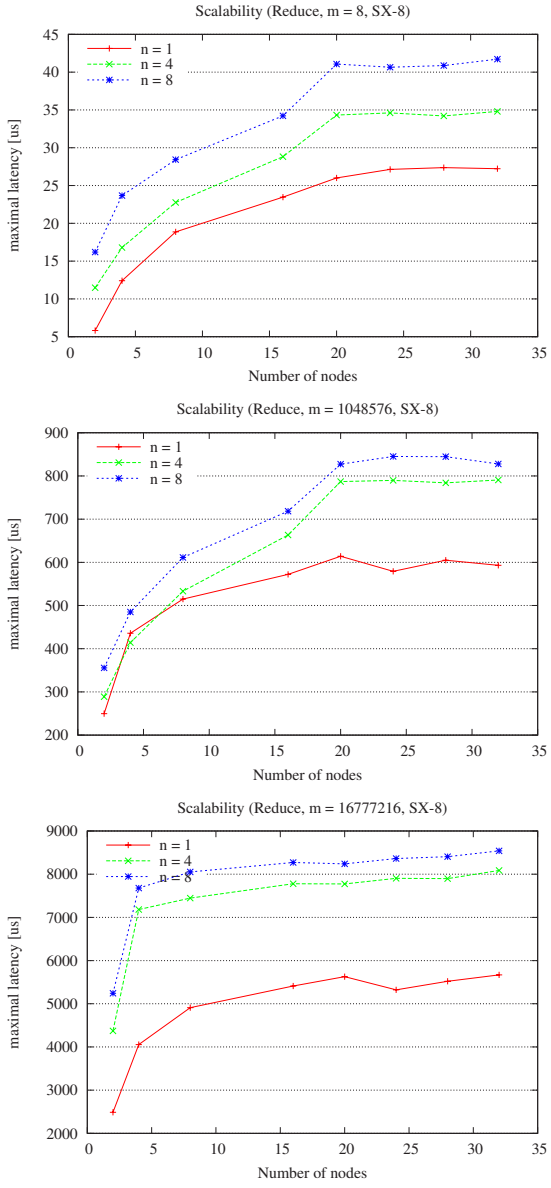

Figure 5. SX-8 scalability of `MPI_Reduce` for 8 Bytes (top), 1 MBytes (middle) and 16 MBytes (bottom). For $n = 1$ the *flat* algorithm is used.

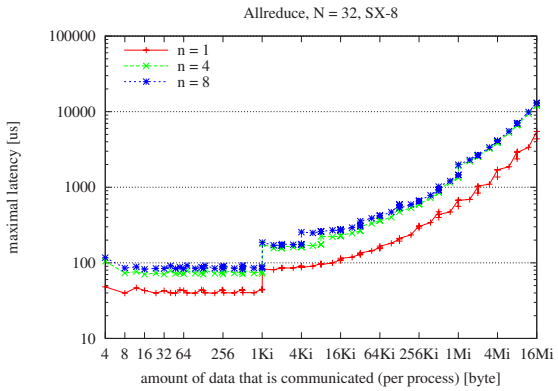

Figure 6. Performance of `MPI_Allreduce` on the SX-8 for fixed $N = 32$. For $n = 1$ the *flat* algorithm is used.

naturally well-suited to the case with more than one process per node, although it would be possible to employ pipelining here also. For this case the tree based algorithms are used, and `MPI_Allreduce` is implemented as a reduction followed by a broadcast, and `MPI_Reduce_scatter` as a reduction followed by a(n irregular) scatter operation.

3.5 Multi-node all-to-all

Despite the different algorithms used for the reduction collectives for the flat and the general multi-node case, the collectives discussed so far can be reasonably implemented by combining an algorithm for the flat case with algorithm(s) for the single-node case. These simple, *hierarchical decompositions* are outlined in Table 2. This is not the case for the *all-to-all* communication primitives (or the irregular collectives), where such an approach would easily lead to significant load imbalances [13]. For the flat case where $p = N$ the algorithm of Bruck et.al. [2] reduces the number of communication rounds to $\lceil \log p \rceil$ and thus reduces latency by combining messages, however at the expense of redundant transmission of a factor of $\lceil \log p \rceil$ more data than in a straight-forward linear algorithm. This algorithm is therefore only useful up to a certain, not too large message size limit. Beyond that limit, a linear algorithm that explicitly pairs nodes so as to avoid contention and starvation in the IXS is used. A similar pairing is used for the general multi-node case, but over a number of rounds in a fashion that reduces the significant load imbalance that would arise in the case where the number of processes per node differ significantly [13, 18]. For small data sizes, data are collected within the nodes to reduce the latency of each node pair data exchange. As shown in Figure 7 and Figure 8 the explicit pairing is efficient in utilizing the IXS com-

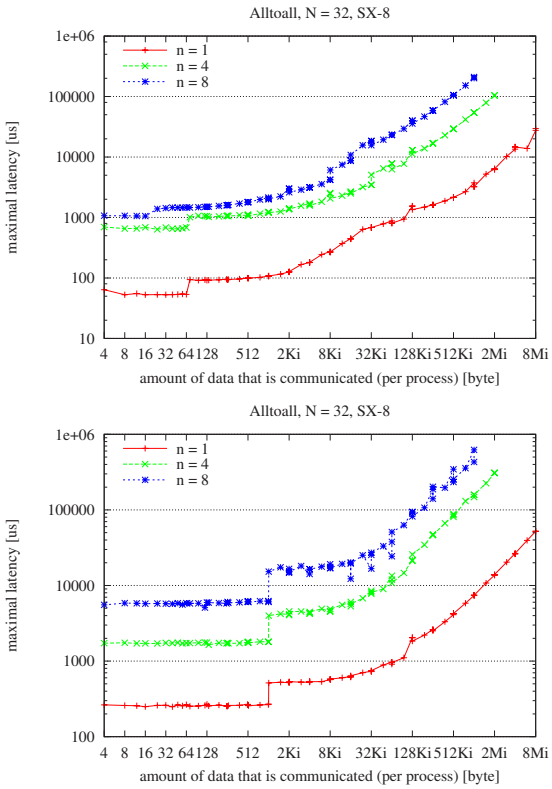

Figure 7. SX-8 performance of `MPI_Alltoall` for fixed $N = 32$, MPI/SX (top) versus naive (bottom) algorithm.

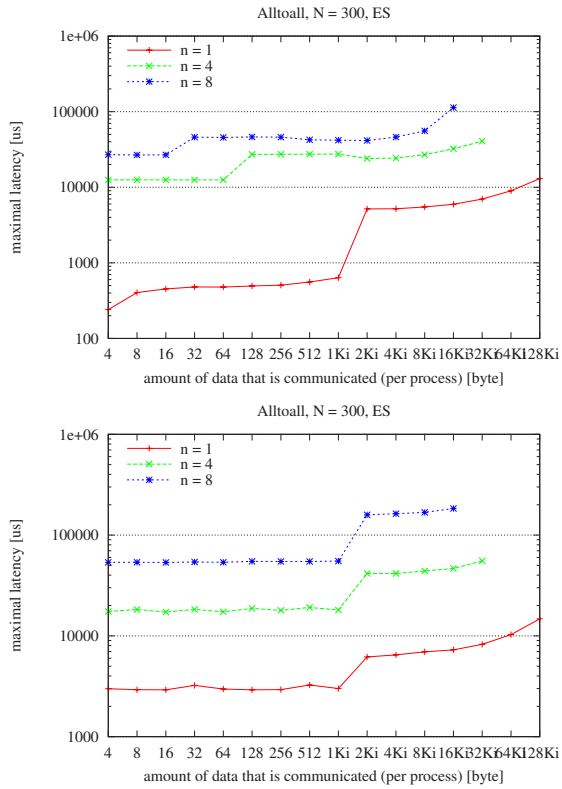

Figure 8. ES performance of `MPI_Alltoall` for fixed $N = 300$, MPI/ES (top) versus naive (bottom) algorithm. Communication buffers are in local memory.

pared to a naive approach that simply sends and receives data from other processes in a non-controlled fashion (using non-blocking point-to-point communication). For the SX-8 with $N = 32$ and $n = 8$ the improved algorithm is a factor 9 faster than the naive approach for $m = 8$ Bytes, and a factor 3 faster for $m = 1$ MByte; similarly for the ES.

4 Other collective operations

The MPI standard has altogether 16 collective communication and computation operations, only a few of which have been touched upon above. In addition, a number of other important MPI calls are collective in the sense that all processes must participate in call and collectively exchange information. These must also be given efficient implementations, but can mostly rely on the collective communication/computation operations. We mention that in MPI/SX the collective operations for creating new communicators (`MPI_Comm_split`, `MPI_Comm_create` etc.) must take care of distributing and reusing the scarce GBCF's (see Section 3.2).

4.1 Gather/scatter

The operations for gathering or scattering data to/from a given root use binomial trees over the processes (not nodes), but sorted according to the identity of the node to which they belong. In [19] it is argued that this can give better performance than algorithms which perform gather/scatter locally followed/preceded by a gather/scatter over the nodes, which can lead to load imbalance. As data size m increases, the trees gradually become flatter, in the limit turning into a linear algorithm. In Figure 9 this algorithm is contrasted to a previous, hierarchical, linear algorithm which first gathers data node locally, and then uses a linear algorithm for gathering the blocks to the global root.

4.2 Multi-node allgather

For small data sizes the (flat) catenation algorithm of Bruck et.al. [2] which can be adopted also to the case with more than one process per node is used for the implemen-

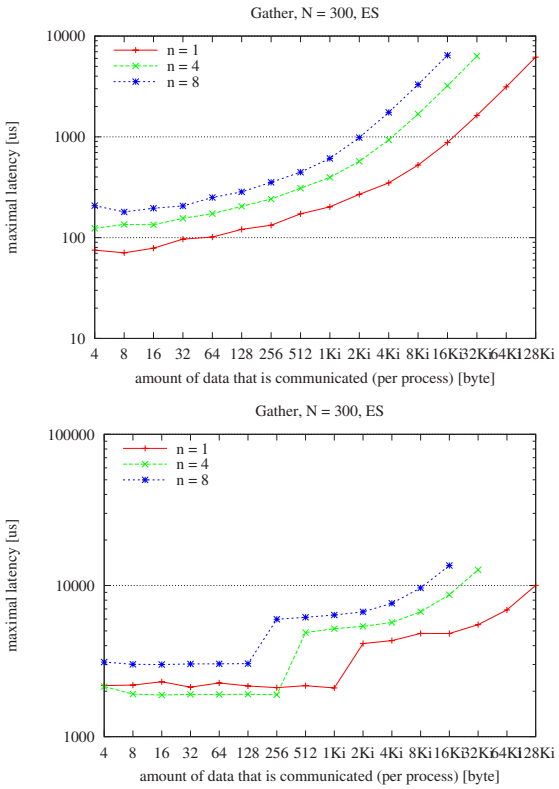

Figure 9. ES performance of MPI/ES binomial tree with graceful degradation `MPI_Gather` for fixed $N = 300$ (top) versus simple, hierarchical, linear (bottom) algorithm. Communication buffers are in local memory. For small data MPI/ES is more than 10 times faster than the linear, hierarchical algorithm.

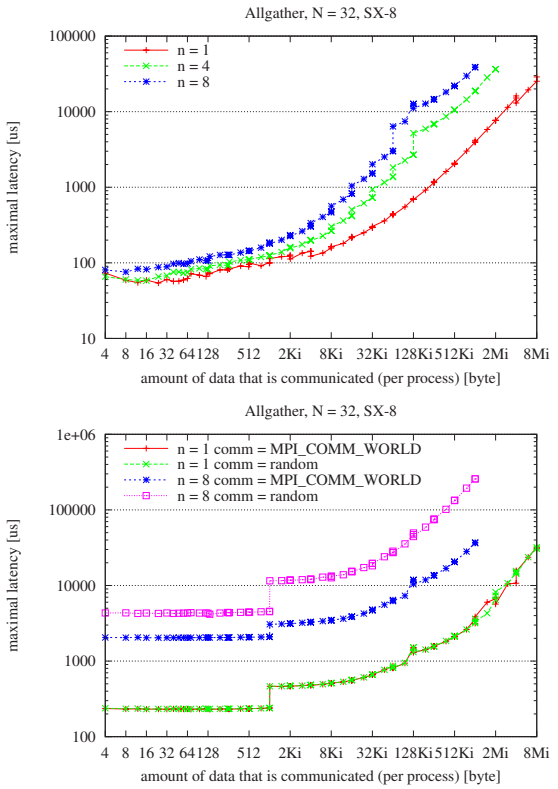

Figure 10. Performance of MPI/SX `MPI_Allgather` for $N = 32$ on a random communicator for $n = 1, 4, 8$ (top) compared to a naive (bottom) algorithm on `MPI_COMM_WORLD` and a random communicator for $n = 1, 8$.

tation of `MPI_Allgather` and `MPI_Allgatherv`. The algorithm collects data in a logarithmic number of rounds, which reduces latency. For large data sizes, a linear (ring) algorithm is used with processes ordered according to their node identity, which ensures that exactly one process on each node receives data from another node and exactly one process on each node sends data to another node. The naive algorithm which does not pay attention to the process to node allocation is extremely sensitive to the actual ordering of the MPI processes. Figure 10 contrasts the MPI/SX implementation to a naive algorithm, which is executed over the `MPI_COMM_WORLD` communicator and over a communicator in which the processes have been randomly permuted. The degradation for the naive algorithm for the random communicator is about a factor of 7, which is as expected since almost all processes per node have their previous and next process on different nodes. The sharp decrease in performance at 1 KBytes is due to a protocol change in the underlying point-to-point communication. The MPI/SX algorithm, on the other hand, is not sensitive to the process to processor allocation, and is more than a factor 20 faster than the naive algorithm for small data, even on the regular `MPI_COMM_WORLD` communicator. The logarithmic algorithm of Bruck et.al. which for each process gathers all data in a global memory buffer, has been modified to degrade gracefully towards the linear algorithm as data grow larger than the maximum size of the global memory buffer. This is nicely illustrated in Figure 10, and can be contrasted to an earlier implementation shown in Figure 11 which switches sharply between logarithmic and linear algorithm.

4.3 Irregular all-to-all collectives

For the remaining irregular all-to-all collectives, `MPI_Alltoallv`, `MPI_Alltoallw` and to some ex-

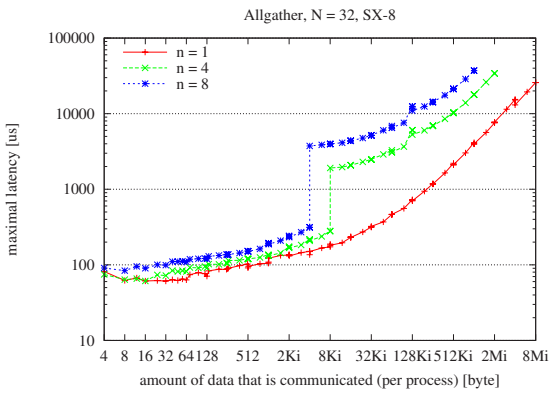

Figure 11. SX-8 allgather with a previous version of MPI/SX for fixed $N = 32$ and $n = 1, 4, 8$ processes/node. The sharp performance decrease at 4 KBytes for $n = 8$ and 8 KBytes for $n = 4$ is due to the switch from logarithmic to linear algorithm.

tent `MPI_Allgatherv` efficient algorithms are either not known or complicated and would entail large overheads (for preprocessing). MPI/SX uses algorithms derived from the regular cases for the irregular cases also, which in extreme cases can lead to load imbalance or have other undesired effects. For MPI/SX, efficient algorithms have been developed and implemented for the `MPI_Gatherv`, `MPI_Scatterv` and `MPI_Reduce_scatter` irregular collectives [19, 20].

5 Concluding remarks and future work

We discussed algorithms and implementation in MPI/SX and MPI/ES for most of the collectives of the MPI standard. Algorithms that are among the best known are employed, and the implementations efficiently utilize the SX hardware. We also pointed out cases where performance improvements are possible, and areas where new algorithmic developments are needed. The developments described here will be transferred to NEC MPI implementations for other platforms.

The semantics of the MPI collectives is, especially for irregular collectives like `MPI_Alltoallv`, a somewhat complex issue, and give ample opportunities for the application programmer to make mistakes. MPI/SX performs error checks that do not require communication between the processes (which would severely compromise a high-performance MPI implementation), such as (locally) correct count and datatype arguments, process ranks and root arguments in range, etc.. To assist the user, an extended

Collective	Schematic implementation
`MPI_Barrier`	1. barrier-single to root
	2. barrier-node
	3. barrier-single from root
`MPI_Bcast`	1. bcast-node
	2. bcast-single
`MPI_Reduce`	1. reduce-single
	2. reduce-node
`MPI_Allreduce`	1. reduce-single
	2. allreduce-node
	3. bcast-single
`MPI_Reduce_scatter`	
	1. reduce-single
	2. reduce-scatter-node
	3. scatter(v)-single
`MPI_Allgather`	1. gather-single
	2. allgather-node
	3. bcast-single

Table 2. Algorithm schemes for *hierarchical decomposition* of collectives that can be reasonably implemented as a combination of intra-node and inter-node algorithm. The gather/scatter and all-to-all collectives do not have this property and are differently implemented in MPI/SX. For large m pipelining is employed in all steps.

MPI/SX library with *collective verification* has recently been developed for checking non-local semantics of collective calls [24, 25], including matching buffer sizes and globally consistent use of other arguments.

Acknowledgments

We thank Takeshi Hayasaka and Masoni Tamura of 1st CSD for continued support for the MPI/SX development. We thank HLRS for the opportunity to perform measurements on the newly installed 72-node NEC SX-8 system in Stuttgart, Germany, and the Earth Simulator Center for the opportunity to perform measurements on the Earth Simulator in Yokohama, Japan. We also thank Joachim Worringen, whose `perfbase` system was used to extract, compare and present data [27].

References

[1] G. Almási, P. Heidelberger, C. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. D. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication

on BlueGene/L systems. In *19th ACM International Conference on Supercomputing (ICS 2005)*, pages 253–262, 2005.

[2] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.

[3] E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. van de Geijn. On optimizing collective communication. In *Cluster 2004*, 2004.

[4] M. Gołebiewski, H. Ritzdorf, J. L. Träff, and F. Zimmermann. The MPI/SX implementation of MPI for NEC's SX-6 and other NEC platforms. *NEC Research & Development*, 44(1):69–74, 2003.

[5] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI – The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998.

[6] W. Gropp and E. Lusk. Reproducible measurements of MPI performance characteristics. 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 11–18, 1999.

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

[8] R. Hempel, H. Ritzdorf, and F. Zimmermann. Efficient message passing interface implementations for NEC parallel computers. *NEC Research & Development*, 39(4):408–413, 1998.

[9] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.

[10] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. Dongarra. Performance analysis of MPI collective operations. In *International Parallel and Distributed Processing Symposium (IPDPS 2005), Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO)*, 2005.

[11] R. Rabenseifner and J. L. Träff. More efficient reduction algorithms for message-passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 36–46. Springer, 2004.

[12] R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A detailed, accurate MPI benchmark. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 5th European PVM/MPI Users' Group Meeting*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59, 1998.

[13] P. Sanders and J. L. Träff. The hierarchical factor algorithm for all-to-all communication. In *Euro-Par 2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 799–803. Springer, 2002.

[14] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.

[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, *18th ACM International Conference on Supercomputing (ICS), Workshop on Component Models and Systems for Grid Applications*, pages 167–185. Springer-Verlag, 2004.

[16] R. Thakur, W. D. Gropp, and R. Rabenseifner. Improving the performance of collective operations in MPICH. *International Journal on High Performance Computing Applications*, 19:49–66, 2004.

[17] V. Tipparaju, J. Nieplocha, and D. K. Panda. Fast collective operations using shared and remote memory access protocols on clusters. In *17th International Parallel and Distributed Porcessing Symposium (IPDPS03)*, page 84, 2003.

[18] J. L. Träff. Improved MPI all-to-all communication on a Giganet SMP cluster. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 9th European PVM/MPI Users' Group Meeting*, volume 2474 of *Lecture Notes in Computer Science*, pages 392–400. Springer, 2002.

[19] J. L. Träff. Hierarchical gather/scatter algorithms with graceful degradation. In *International Parallel and Distributed Processing Symposium (IPDPS 2004)*, page 80, 2004.

[20] J. L. Träff. An improved algorithm for (non-commutative) reduce-scatter with an application. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 129–137. Springer, 2005.

[21] J. L. Träff and A. Ripke. An optimal broadcast algorithm adapted to SMP-clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, pages 48–56. Springer, 2005.

[22] J. L. Träff and A. Ripke. Optimal broadcast for fully connected networks. In *High Performance Computing and Communications (HPCC'05)*, volume 3726 of *Lecture Notes in Computer Science*, pages 45–56. Springer, 2005.

[23] J. L. Träff, H. Ritzdorf, and R. Hempel. The implementation of MPI-2 one-sided communication for the NEC SX-5. In *Supercomputing*, 2000. `http://www.sc2000.org/proceedings/techpapr/index.htm#01`.

[24] J. L. Träff and J. Worringen. Verifying collective MPI calls. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 18–27. Springer, 2004.

[25] J. L. Träff and J. Worringen. The MPI/SX collectives verification library. In *ParCo*, 2005.

[26] R. van de Geijn. On global combine operations. *Journal of Parallel and Distributed Computing*, 22:324–328, 1994.

[27] J. Worringen. Experiment management and analysis with perfbase. In *Proceedings of the IEEE International Conference on Cluster Computing*, Boston, September 2005. IEEE Computer Society Press.