A Decomposition Approach for Optimizing the Performance of MPI Libraries

Olaf Hartmann[1], Matthias Kühnemann[1], Thomas Rauber[2], Gudula Rünger[1]

[1] Chemnitz University of Technology
Department of Computer Science
Chemnitz, Germany
ruenger@informatik.tu–chemnitz.de

[2] University Bayreuth
Department of Computer Science
Bayreuth, Germany
rauber@uni–bayreuth.de

Abstract

MPI provides a portable message passing interface for many parallel execution platforms but may lead to inefficiencies for some platforms and applications. In this article we show that the performance of both, standard libraries and vendor-specific libraries, can be improved by an orthogonal organization of the processors in 2D or 3D meshes and by decomposing the collective communication operations into several phases. We describe an adaptive approach with a configuration phase to determine for a specific execution platform and a specific MPI library which decomposition leads to the best performance. This may also depend on the number of processors and the size of the messages to be transferred. The decomposition approach has been implemented in the form of a library extension which is called for each activation of a collective MPI operation. This has the advantage that neither the application programs nor the MPI library need to be changed while leading to significant performance improvements for many collective MPI operations.

1 Introduction

MPI libraries provide a portable communication interface for message-passing programs. In addition to single-transfer operations where one processor sends a data block which is received by another processor, MPI also provides a set of collective communication operations realizing typical communication patterns with more than two participating processors. Examples are broadcast, gather or scatter operations.

Internally, MPI libraries implement collective communication operations by combining single-transfer operations according to the communication pattern required or by using several simpler collective communication operations to realize the communication pattern of more complex operations. Depending on the interconnection network of the target platform, different realizations of collective communication operations lead to different execution times. The best performance usually results if the realizations are optimized for a specific interconnection network of a parallel machine. This is usually done for vendor-specific MPI libraries.

Standard MPI libraries like LAM-MPI [6] or MPICH [4] implement collective MPI operations such that a good performance can be expected for a typical execution platform. However, this general way of implementing collective MPI operations may lead to performance degradations on some platforms, compared to an optimized vendor-specific MPI library. The advantage of these standard libraries is that they provide satisfactory performance for a wide range of execution platforms like cluster systems and that they can be easily installed without complicated adaptations.

Runtime experiments on different execution platforms show that the performance of collective MPI operations of standard MPI libraries can be significantly improved by using orthogonal processor groups or point-to-point algorithms based on virtual communication topologies. An orthogonal processor group is obtained by arranging the participating processors as a virtual 2D or 3D mesh. Based on this arrangement, a collective communication operation is then realized in two or three phases such that each phase is performed in a different dimension of the processor mesh. Since only a specific collective MPI communication operation is replaced by two or more communication phases on the processor mesh, the use of this approach is completely independent of the computation or communication structure of the parallel application considered. Therefore, all applications based on MPI can benefit from the decomposition approach for optimizing communication. The overhead caused by setting up the processor groups and the communicator handles for the orthogonal directions is negligible.

The performance improvements achieved by the decom-

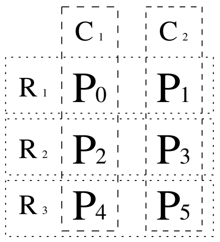


Figure 1. A set of 6 processors arranged as a two-dimensional mesh with 3 row groups and 2 column groups.

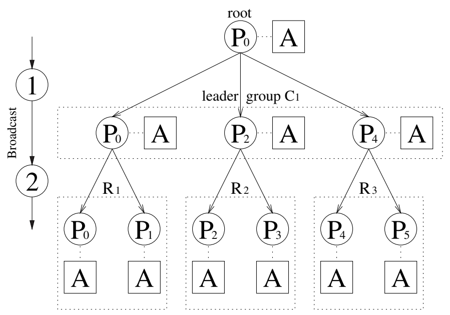


Figure 2. Illustration of an orthogonal realization of a broadcast operation with 6 processors and root processor P_0 realized by 3 concurrent groups of 2 processors each.

position into phases depend on numerous factors, like the interconnection network of the target machine, the MPI library used, the specific group layout in the mesh and the collective communication operation to be executed. There may also be a dependency on the number of participating processors and the message size. Thus, for an application programmer it is demanding to benefit from this approach. We therefore propose an adaptive implementation that automatically selects a suitable layout of the processor mesh such that the best performance improvement for a specific collective communication operation is obtained. In addition, we consider several realizations of collective communication operations using non-blocking send and receive operations on virtual processor topologies like star, hypercube and binomial tree interconnections.

The adaptive implementation of the decomposition approach has been integrated in an extension library for MPI which can be linked with an arbitrary MPI library with the effect that the collective communications operations are executed in an optimized way.

The rest of the paper is organized as follows. Section 2 describes how collective MPI operations can be implemented based on orthogonal processor groups and suitable point-to-point algorithms. Section 3 presents the adaptive selection of implementation variants for specific execution platforms and describes the design of the extension library. Section 4 contains an experimental evaluation on different target platforms. Section 5 discusses related work and Section 6 contains concluding remarks.

2 Orthogonal Processor Groups

The realization of collective communication operations in consecutive phases based on an orthogonal partitioning of the processor set can be applied for arbitrary MPI libraries. We first consider the 2D case and assume that the set of processors is arranged as a two-dimensional virtual mesh with a total number of $p = p_1 \times p_2$ processors. The mesh consists of p_1 row groups $R_1, ..., R_{p_1}$ and p_2 column groups $C_1, ..., C_{p_2}$ with $|R_q| = p_2$ for $1 \le q \le p_1$ and $|C_r| = p_1$ for $1 \le r \le p_2$. The row groups provide a partitioning into disjoint processor sets and the column groups provide a different partitioning into disjoint processor sets that are orthogonal to the row groups. Using these two partitionings, the collective communication operations can be implemented in two phases, each working on one of the partitionings of the processor mesh. Based on the processor mesh and the two partitionings induced, group and communicator handles are defined for the concurrent communication in the row and column groups. Each processor belongs to exactly one row group and analogously to exactly one column group. A row group and a column group have exactly one processor in common which can serve as communication processor between row groups and column groups. Figure 1 illustrates a set of 6 processors $P_0, P_1, ..., P_5$ arranged as a $p_1 \times p_2 = 3 \times 2$ mesh.

For the decomposition of a broadcast operation on a 2D mesh, the root processor first broadcasts the block of data within its column group (leader group). Then each of the processors in the leader group acts as a root in its row group and broadcasts the data within this group (concurrent group) concurrently to the other broadcast operations. Figure 2 illustrates the resulting two communication phases for the processor mesh from Figure 1 with processor P_0 as root of the broadcast operation. In step (1), processor P_0 sends the message A within its column group $C_1 = \{P_0, P_2, P_4\}$; this is the leader group. In step (2), each member of the leader group sends the message within its row group.

For the decomposition of a gather operation, the data blocks are first collected concurrently within the row groups by concurrent group based gather operations. In each group, the data blocks are collected by the unique processor belonging to that column group (leader group) to which the root of the global gather operation also belongs to. In a second step, a gather operation is performed within the leader group only and collects all data blocks at the root proces-

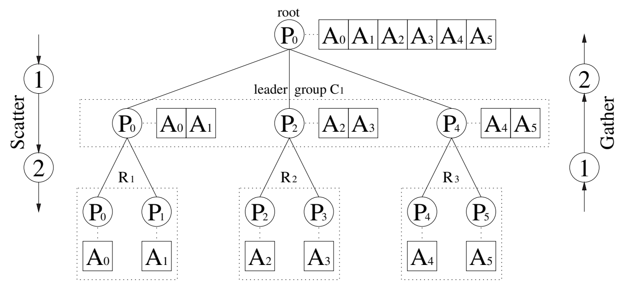


Figure 3. Illustration of an orthogonal realization of a gather operation (upward) and a scatter operation (downward) with 6 processors and root processor P_0 for 3 concurrent groups of 2 processors each.

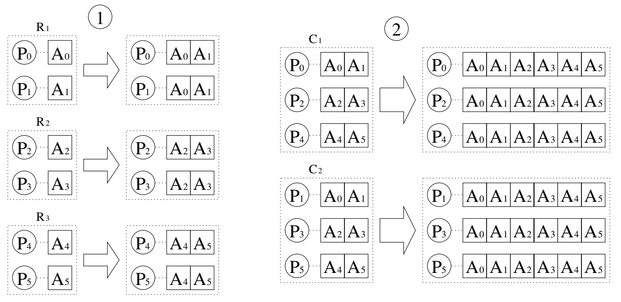


Figure 4. Illustration of an orthogonal implementation of multi-broadcast operation for the processor mesh of Figure 1.

sor specified for the global gather operation. If b is the size of the original message, each processor in the leader group contributes a data block of size $b \cdot p_2$ for the second communication step. The order of the messages collected at the root processor is preserved. Figure 3 (upward) illustrates the two phases for the processor mesh from Figure 1 where processor P_i contributes data block A_i, $i = 1, ..., 6$. In step (1), the processors P_0, P_2, P_4 concurrently collect messages from its row groups. In step (2), the leader group collects the messages built up in the previous step. A scatter operation starts at the root P_0 and can be realized by reversing the order of the two phases used for a gather operation.

For a multi-broadcast operation, each processor contributes a data block of size b and the operation makes all data blocks available in rank order for each processor. Using a 2D processor mesh, the operation can be implemented by the following two steps: first, group-based multi-broadcast operations are executed concurrently within the row groups, thus making each data block available for each processor within column groups, see Figure 4 for an illustration. Second, concurrent group-based multi-broadcast operations are performed to distribute the data blocks within the column groups. For this operation, each processor contributes messages of size $b \cdot p_2$.

The decomposition approach for 2D meshes can be applied recursively to the internal communication organization of the leader group or the concurrent groups, respectively, so that the communication in the leader group or the concurrent groups again uses an orthogonal structuring of the group. This leads to 3D or higher dimensional mesh structures. Runtime experiments have shown that 3D meshes are usually sufficient to get good performance improvements.

3 Adaptive MPI extension library

The performance improvements obtained by an orthogonal realization of collective MPI operations are provided as an extension library which can be linked with arbitrary MPI libraries. The library is organized in two phases:

- The *configuration phase* is responsible for determining for each collective MPI operation which implementation leads to the best improvements for a given execution platform and a specific MPI library. The configuration phase has to be executed only once for each combination of MPI library and execution platform.
- The *execution phase* contains the actual extension library and is activated by linking the extension library to an arbitrary MPI application. The execution phase is activated for each collective MPI operation called by the application. When activated, the execution phase selects for the given number of processors and message size the best implementations either using orthogonal processor groups, virtual topologies or the MPI implementations provided by the underlying MPI library. The execution of the collective or single-transfer MPI operations required is realized with the underlying MPI library.

In the following, we describe the two phases and the interface between the two phases in more detail.

3.1 Configuration phase

To determine which implementations are best suited for a given architecture and a given MPI library, the configuration phase executes an evaluation program which starts benchmarks for each collective MPI operation using different implementation variants and different organizations of the orthogonal processor groups in two or three dimensions. The benchmark program measures the communication times for different numbers of processors and different message sizes. The evaluation program starts the benchmark program for all possible 2D or 3D layouts of the processors and also for different virtual processor topologies suitable for the execution of the specific collective communication operation

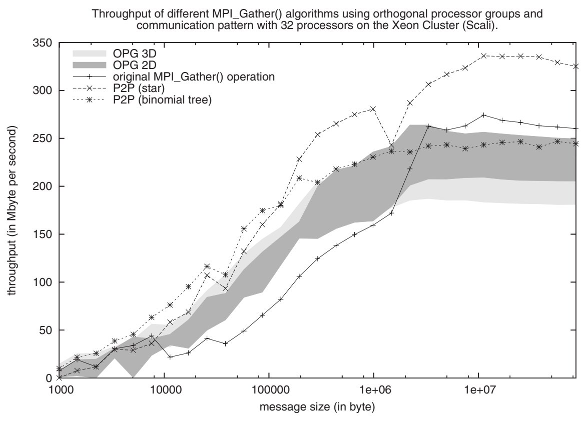


Figure 5. Throughput of different implementation variants for a gather operation on the dual Xeon cluster for 32 processors.

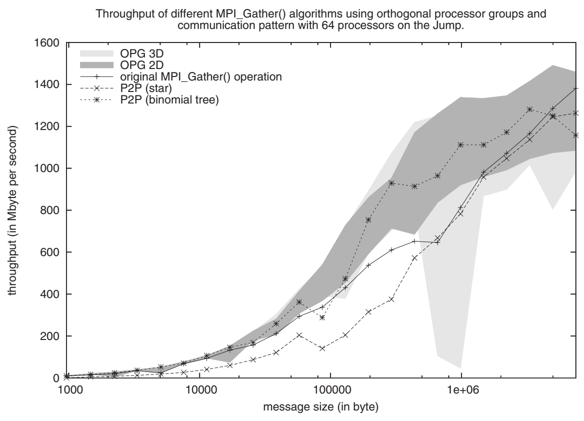


Figure 6. Throughput of different implementation variants for a gather operation on the IBM Regatta system (bottom).

considered. These topologies can be based on spanning trees for communication operations with a root processor, like broadcast or gather, or virtual interconnection networks for communication operations without a root processor, like multi-broadcast or multi-accumulation. The current implementation considers a star, a hypercube, a binomial tree, and a ring topology. By comparing all communication times measured, the evaluation program can select for each number of processors and each message size which implementations variant should be used for the given execution platform and MPI library.

The evaluation program builds intervals of message sizes and numbers of executing processors and specifies for each interval which implementation variant should be used. The results are stored in a *configuration table* which is later used by the execution phase to select the fastest implementation at program execution time. Depending on the benchmark tests it is also possible that the original implementation provided by the underlying MPI library is selected as the best implementation for a specific number of processors and a specific message size.

3.2 Execution phase

For each activation of an MPI collective communication operation, the execution phase dynamically selects the most efficient implementation for the message size specified by the MPI operation and the number of processors participating in the operation. The selection is based on the configuration table computed by the configuration phase. The execution phase is supported by an extension library which is responsible for

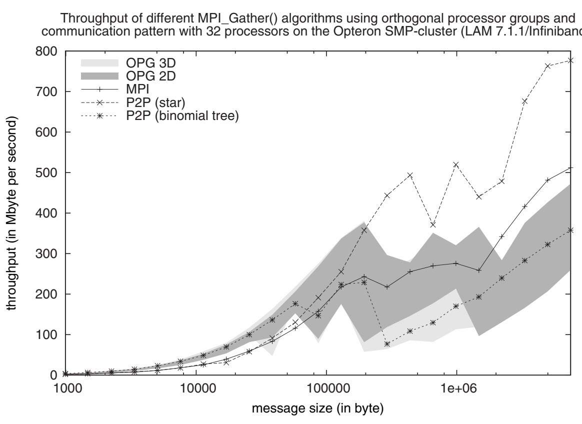
- creating the 2D or 3D processor meshes required for the best implementation for the given number of processors and the corresponding communicators in the different dimensions of the processor mesh; these communicators are used for the realizations of the different phases of the collective communication operations;
- providing additional realizations of the collective communication operations based on virtual topologies using non-blocking send and receive operations of the underlying MPI library;
- managing the selection of the best implementation of each collective MPI operation depending on the number of processors and the message size of the specific MPI operation.

Depending on the number of processors and the configuration table, there might be a large number of different 2D or 3D processor meshes leading to the best performance. If this is the case, a *lazy construction* of the processor meshes and the corresponding communicators can be useful: Instead of creating all meshes and communicators at program start, a mesh is only created when the configuration phase selects the mesh for the implementation of a collective MPI operation. In this case, the mesh and communicators are created and can be used for further activations of collective MPI operations. If a processor mesh is not needed for the execution of a program, it is not created, thus reducing the implementations overhead.

The approach presented can be applied to arbitrary MPI programs using collective communication operations.

4 Runtime Experiments

In this section, we show runtime experiments for different execution platforms and different MPI libraries. We consider the performance improvements obtained for col-



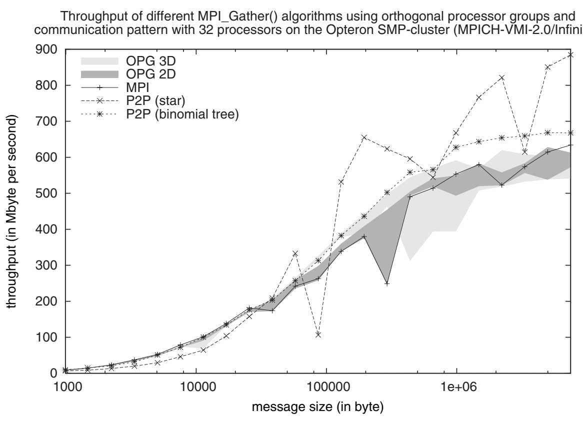
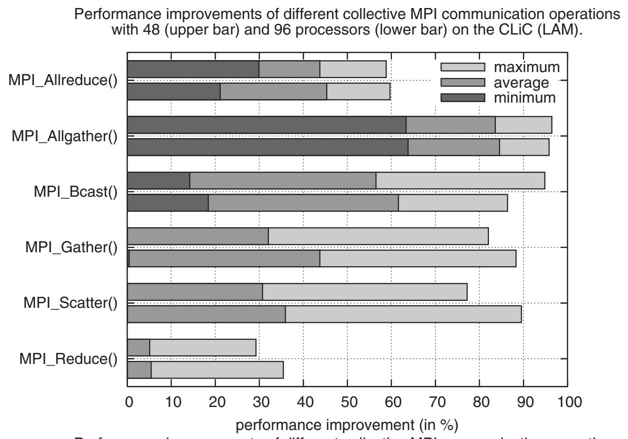


Figure 7. Throughput of different implementation variants for a gather operation on the dual Opteron cluster using LAM-MPI (top) and MPICH-VMI (bottom).



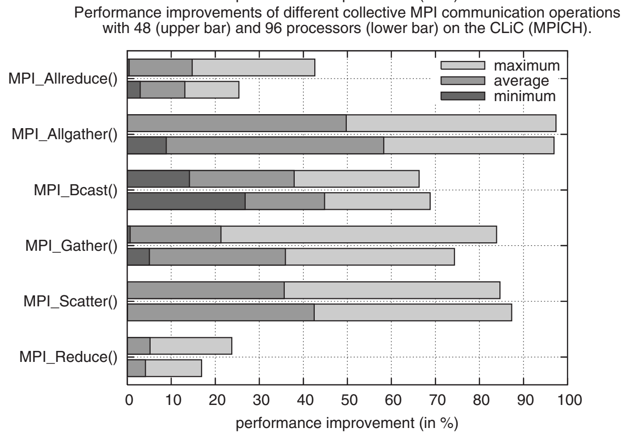


Figure 8. Performance improvements achieved for LAM-MPI (top) and MPICH (bottom) on the Beowulf cluster CLiC.

lective MPI operations in isolation and in the context of application programs from scientific computing.

4.1 MPI operations in isolation

The number of different 2D or 3D layouts of the processor mesh depends on the total number of processors available. For 96 processors, for example, there are 10 different 2D layouts to be considered $(2 \times 48, 3 \times 32, 4 \times 24, 6 \times 16, 8 \times 12, 12 \times 8, 16 \times 6, 24 \times 4, 32 \times 3, 48 \times 2)$. We have performed runtime experiments on five different platforms, the cluster systems CLiC with 528 processors, a dual Xeon cluster with 32 processors, the IBM Regatta p690+ cluster JUMP with 1312 processors, a Cray T3E 1200 with 512 processors, and an Opteron cluster with 64 processors.

As an example for benchmark results, Figures 5, 6, and 7 show the data throughput (in MBytes per second) obtained with different layouts of the processor mesh for an MPI_Gather() operation on different platforms. The grey areas depict the throughput obtained with an orthogonal realization using 2D meshes (OPG 2D) or 3D meshes (OPG 3D). The dashed lines show the throughput obtained by using a star and a binomial tree virtual processor interconnection. The figures show that for different message sizes, different implementation variants lead to the best performance. Compared to the implementation provided by the underlying MPI library, significant performance improvements can usually be obtained for a wide range of message sizes. Significant performance improvements can also be obtained for vendor-specific MPI libraries like Cray-MPI, ScaMPI [3] or mpcc on the IBM Regatta.

Figure 8 summarizes the performance improvements obtained for the CLiC cluster using LAM-MPI (top) and MPICH (bottom). The bars represent the minimum, average, and maximum performance improvements over all message sizes (between 992 and 7 480 288 Bytes) for a total number of 48 processor (upper bar) and 96 processors (lower bar). For LAM-MPI, the largest performance improvement can be obtained for MPI_Bcast() operations as well as MPI_Allgather() and MPI_Allreduce() which are both implemented by using MPI_Bcast(). The reason for the large improvements lies in the implementation of the

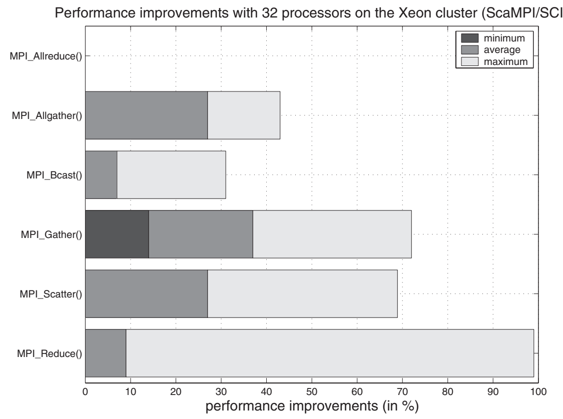


Figure 9. Performance improvements on the dual Xeon cluster with ScaMPI/SCI for 32 processors.

MPI_Bcast() operation which is based on a binomial virtual tree. A parent node sends the broadcast block to its children with non-blocking send-operations without taking into account the size of the corresponding sub-trees. Both LAM-MPI and MPICH use different protocols for different message sizes. An eager protocol is used for small messages. Using this protocol, a processor sends a data block also if a matching receive operation has not yet been issued. The receiving processor therefore has to store the message in a system buffer and then has to copy it to the user buffer of the matching receive later. This copy operation can be avoided by using a rendezvous protocol which stores the message at the sender side until the receive buffer has been specified. A rendezvous protocol is used for larger messages. The optimal change from the eager to the rendezvous protocol depends on many architectural parameters and cannot be set in advance. The adaptive approach is able to find a suitable changing point which substantially contributes to the significant performance improvements.

The performance improvements for the dual Xeon cluster using the SCI network with ScaMPI is given in Figure 9. Figure 10 contains the performance improvements for the Cray T3E-1200 (top) and the IBM Regatta system (bottom). For the IBM Regatta System, the largest performance improvements can be obtained when using specific point-to-point algorithms, especially for MPI_Reduce(), MPI_Allgather() and MPI_Allreduce(). For other operations, orthogonal realizations help to successfully reduce the execution time. Figures 11 and 12 present the performance improvements for the Opteron cluster using the Ethernet network and the Infiniband network, respectively, for both LAM-MPI and MPICH. The improvements are significantly larger for LAM-MPI, but also for MPICH good average improvements can be obtained for both networks.

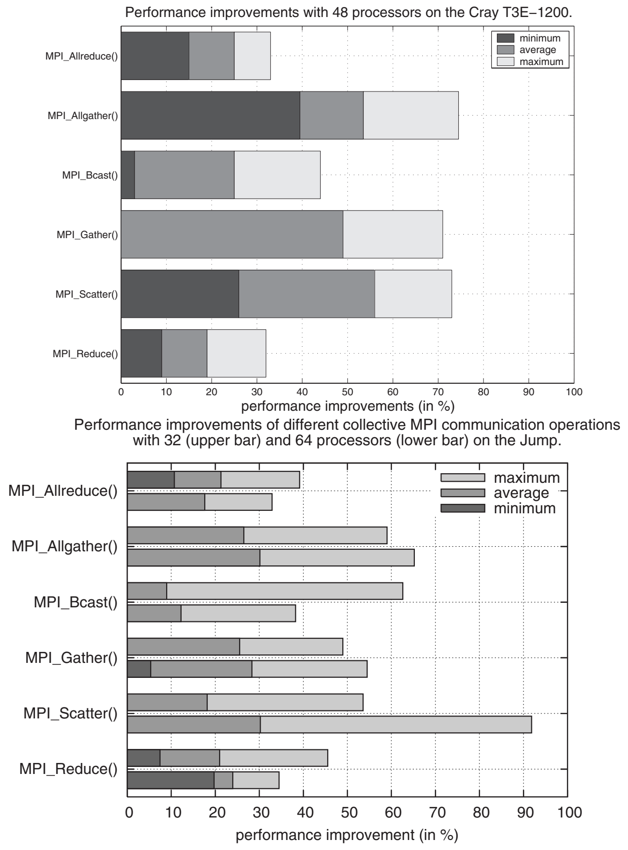


Figure 10. Performance improvements obtained on the Cray T3E-1200 (top) for a total number of 48 processors and on the IBM Regatta cluster (bottom) for a total number of 32 processors (upper bar) and 64 processors (lower bar).

4.2 Application Programs

We consider the optimized communication methods in the context of complex program applications in order to verify the performance improvements achieved in isolation. For this purpose we apply the adaptive approach to reduce the communication overhead of a Jacobi iteration and a solution method of ordinary differential equations.

For the Jacobi iteration, we consider three different implementation variants based on a row-wise and a column-wise distribution of the iteration matrix A. The row-wise realization uses an MPI_Allgather() operation to distribute the intermediate result to all processors participating in the computation. For the column-wise realization, either an MPI_Allreduce() and MPI_Allgather() operation or an MPI_Reduce() and MPI_Bcast() operation can be used to distribute the intermediate result. Figure 13 (top) shows the performance improvements of different implementations of

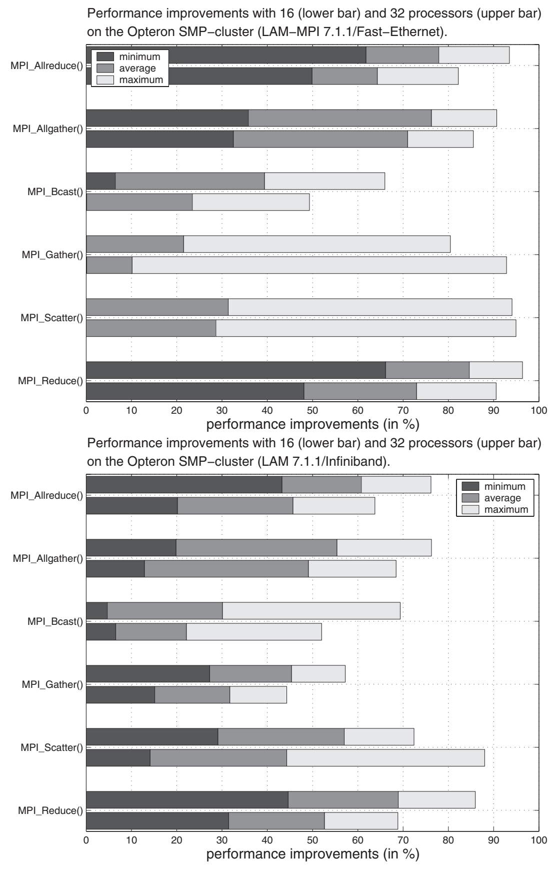


Figure 11. Performance improvements for the Opteron cluster for 64 processors using the Ethernet network (top) and the Infiniband network (bottom) for LAM MPI 7.1.1.

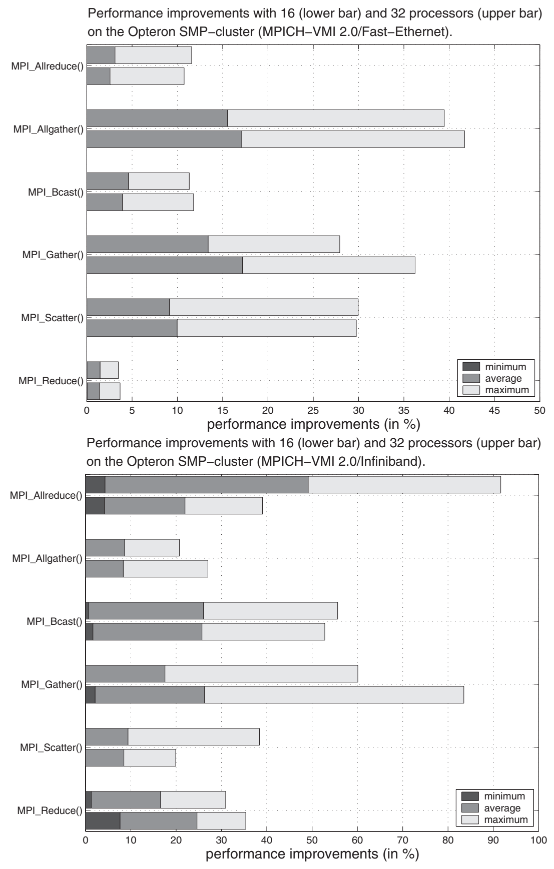


Figure 12. Performance improvements for the Opteron cluster for 64 processors using the Ethernet network (top) and the Infiniband network (bottom) for MPICH-VMI-2.0.

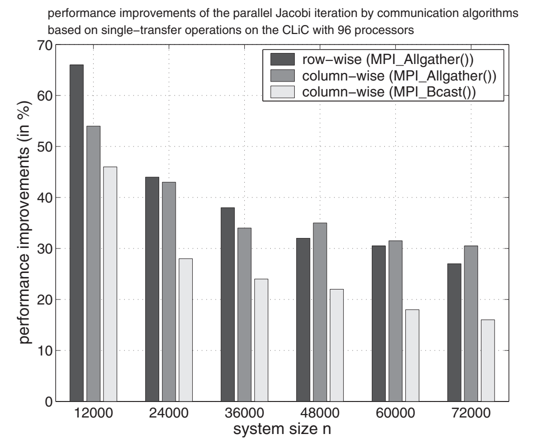
the parallel Jacobi iteration using the adaptive approach to select a suitable communication method on the CLiC (LAM-MPI) for 96 processors. In most cases, the adaptive approach selects point-to-point algorithms for performing the collective communication operations, since these algorithms lead to slightly larger performance improvements.

Parallel Adams methods are variants of general linear methods for solving ordinary differential equations (ODEs) as proposed in [10], see [7] for a more detailed description of a parallel implementation. General linear methods compute several stage values $\mathbf{y}_{\kappa,i}$ in each time step κ, $\kappa = 1, 2, \ldots$ which correspond to numerical approximations of $\mathbf{y}_{\kappa,i} = \mathbf{y}(t_\kappa + a_i h)$ with abscissa vector (a_i), $i = 1, ..., K$, and stepsize $h = t_\kappa - t_{\kappa+1}$. The stage values of one time step are combined in a vector $\mathbf{Y}_\kappa = (\mathbf{y}_{\kappa,1}, ..., \mathbf{y}_{\kappa,K})$. An MPI_Allgatherv() operation is used to distribute the intermediate result. Figure 13 (bottom) shows the average performance improvements that are obtained by an adaptive selection of communication methods to perform the MPI_Allgatherv() operation on the CLiC (96 processors, LAM-MPI and MPICH) and the IBM Regatta system (64 processors). Again, the point-to-point realizations are selected by the adaptive approach, since these are about 10% faster than the orthogonal realizations for the system sizes considered.

5 Related work

Several recent articles consider the improvement of collective communication operations. In [1] the performance of an MPI_Allgather() operation in MPICH 1.2.5 on a Linux cluster is evaluated. This implementation of MPICH improves the performance of previous versions by using a *recursive doubling* algorithm. The authors have developed a *dissemination allgather* based on the dissemination barrier algorithm. The paper experimentally evaluates MPICH allgather and the implementations of the new allgather algorithms on a Linux cluster of dual-processor nodes using both TCP over Fast Ethernet and GM over Myrinet.

In [2] the authors discuss the high-performance im-



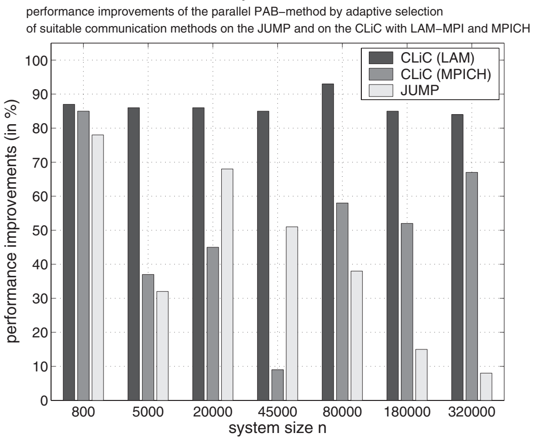


Figure 13. Runtime improvements for the parallel Jacobi iteration (top) and the parallel PAB method (bottom).

plementation of collective MPI operations on distributed-memory machines. A combination of different implementations of collective MPI operations are investigated along with an exploitation of communication modes supported by MPI. The implementations show performance improvements in most situations compared to standard MPI implementations, such as MPICH. Experimental results from a large Intel Pentium 4 processor cluster are considered.

An adaptive approach for collective communication operations is presented in [8]. Different algorithms based on virtual topologies are tested. The optimum algorithm and optimum buffer sizes are determined by conducting a series of experiments on the system. The tuning system uses the native MPI point-to-point send and receive operations. The approach is similar to our approach in the sense that different versions of implementing MPI collective operations are experimentally evaluated, but there is no extension library as for our decomposition approach. Moreover, orthogonal processor groups are not included. The search space of the adaptive approach in [8] is reduced in [9] where performance modelling techniques for collective communication operations are given. Based on the performance modelling the useful candidates for experiements are selected.

In [5] the performance behavior of collective communication operations are analyzed with models for point-to-point communication operations, such as Hockney, LogP/LogGP, and PLogP. The performance models provide useful insights into various aspects of different algorithms and their relative performance. The prediction results from the models are used to determine switching points between different available communication methods.

6 Conclusions

In this article we have shown that an adaptive approach based on the decomposition of operations can be used to successfully reduce the execution time of collective communication operations for a wide range of different platforms. For general libraries like LAM-MPI or MPICH, average improvements between 30% and 50% can usually be obtained. But also the performance of vendor-specific libraries can be improved significantly. The adaptive approach can be used via an extension library that can be linked to arbitrary MPI programs and can be used with arbitrary MPI libraries.

References

[1] G.D. Benson, Cho-Wai Chu, Q. Huang, and S. G. Caglar. A comparison of MPICH Allgather Algorithms on Switched Networks. In *10th European PVM/MPI Users' Group Meeting*. Springer LNCS 2840, 2003.

[2] E.W. Chan, M.F. Heimlich, A. Purkayastha, and R.A. van de Geijn. On Optimizing Collective Communication. In *Proc. of Int. Conference on Cluster Computing*, 2004.

[3] Scali / ScaMPI commercial MPI on SCI implementation. http://www.scali.com/.

[4] LAM/MPI Parallel Computing. http://www.lam-mpi.org/.

[5] J. Pjesivac-Grbovic and T. Angskun and G. Bosilca and G. E. Fagg and J. Dongarra. Performance Analysis of MPI Collective Operations. In *Proc. of the 4th Int. Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-IPDPS*, 2005.

[6] MPICH-A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich.

[7] T. Rauber and G. Rünger. Execution Schemes for Parallel Adams Methods. In *Proc. of Euro-Par 2004*, pages 708–717. Springer LNCS 3149, 2004.

[8] S.S. Vadhiyar, G.E. Fagg, and J. Dongarra. Automatically tuned collective communications. In *Proc. of the Supercomputing*. IEEE, 2000.

[9] S.S. Vadhiyar, G.E. Fagg, and J. Dongarra. Performance Modeling of Self Adapting Collective Communications for MPI. In *Los Alamos Computer Science Institute Symposium*, 2001.

[10] P.J. van der Houwen and E. Messina. Parallel Adams Methods. *J. of Comp. and App. Mathematics*, 101:153–165, 1999.