Optimizing MPI Collective Communication by Orthogonal Structures

MATTHIAS KÜHNEMANN

Fakultät für Informatik
Technische Universität Chemnitz
09107 Chemnitz, Germany
kumat@informatik.tu–chemnitz.de

THOMAS RAUBER

Fakultät für Mathematik und Physik
Universität Bayreuth
95445 Bayreuth, Germany
rauber@uni–bayreuth.de

GUDULA RÜNGER

Fakultät für Informatik
Technische Universität Chemnitz
09107 Chemnitz, Germany
ruenger@informatik.tu–chemnitz.de

Abstract

Many parallel applications from scientific computing use MPI collective communication operations to collect or distribute data. Since the execution times of these communication operations increase with the number of participating processors, scalability problems might occur. In this article, we show for different MPI implementations how the execution time of collective communication operations can be significantly improved by a restructuring based on orthogonal processor structures with two or more levels. As platform, we consider a dual Xeon cluster, a Beowulf cluster and a Cray T3E with different MPI implementations. We show that the execution time of operations like MPI_Bcast or MPI_Allgather can be reduced by 40% and 70% on the dual Xeon cluster and the Beowulf cluster. But also on a Cray T3E a significant improvement can be obtained by a careful selection of the processor groups. We demonstrate that the optimized communication operations can be used to reduce the execution time of data parallel implementations of complex application programs without any other change of the computation and communication structure. Furthermore, we investigate how the execution time of orthogonal realization can be modeled using runtime functions. In particular, we consider the modeling of two-phase realizations of communication operations. We present runtime functions for the modeling and verify that these runtime functions can predict the execution time both for communication operations in isolation and in the context of application programs.

Contents

1 Introduction

Parallel machines with distributed address space are widely used for the implementation of applications from scientific computing, since they provide good performance for a reasonable price. Portable message-passing programs can be written using message-passing standards like MPI or PVM. For many applications, like grid-based computations, a data-parallel execution usually leads to good performance. But for target machines with a large number of processors, data parallel implementations may lead to scalability problems, in particular when collective communication operations are frequently used for exchanging data. Often scalability can be improved by re-formulating the program as a mixed task and data parallel implementation. This can be done by partitioning the computations into multiprocessor tasks and by assigning the tasks to disjoint processor groups for execution such that one task is executed by the processors of one group in a data parallel way, but different independent tasks are executed concurrently by disjoint processor groups [16]. The advantage of a group-based execution is caused by the communication overhead of collective communication operations whose execution time shows a logarithmic or linear dependence on the number of participating processors, depending on the communication operation and the target machine.

Another approach to reduce the communication overhead is the use of orthogonal processor groups [14] which are based on an arrangement of the set of processors as a virtual two- or higher-dimensional grid and a fixed number of decompositions into disjoint processor subsets representing hyper-planes. To use orthogonal processor groups the application has to be re-formulated such that it consists of tasks that are arranged in a two- or higher-dimensional task grid that is mapped onto the processor grid. The execution of the program is organized in phases. Each phase is executed on a different partitioning of the processor set and performs communication in the corresponding processor groups only. For many applications, this may reduce the communication overhead considerably but requires a specific potential of parallelism within the application and a complete rearrangement of the resulting parallel program.

In this article we consider a different approach to reduce the communication overhead. Instead of rearranging the entire program to a different communication structure, we use the communication structure given in the data parallel program. But for each collective communication operation, we introduce an internal structure that uses an orthogonal arrangement of the processor set with two or more levels. The collective communication is split into several phases each of which exploits a different level of the processor groups. Using this approach a significant reduction in the execution time can be observed for different target platforms and different MPI implementations. The most significant improvement results for *MPI_Allgather()* operations. This is especially important as these operations are often used in scientific computing. Examples are iterative methods where *MPI_Allgather()* operations are used to collect data from different processors and to make this data available to each processor for the next time step. The advantage of the approach is that the application does not have to provide a specific potential of parallelism and that all programs using collective communication can take advantage of the improved communication. Also no rearrangement of the program is necessary and no specific knowledge about the additional implementation structure is needed, so that a data parallel implementation mainly remains unchanged.

The internal rearrangement of the collective communication operations is done on top of MPI on the application programmers level. So, the optimization can be used on a large range of machines providing MPI. As target platforms we consider a Cray T3E, a Xeon cluster and a Beowulf cluster. As application programs we consider an iterative solution method for linear equation systems and solution methods for initial value problems of

ordinary differential equations.

Furthermore, we consider the modeling of the parallel runtime with runtime functions that are structured according to the orthogonal communication phases. This model is suitable for data parallel programs [15] as well as for mixed task and data parallel programs [11], and it can also be used for large and complicated application programs [8]. We investigate the use of runtime functions for modeling the runtime of MPI collective communication operations with the specific internal realization that is based on an orthogonal structuring of the processors in a two-dimensional grid. Using the runtime functions, the programmer can get an a priori estimation of the execution time to predict various runtime effects of an orthogonal realization.

The rest of the paper is organized as follows. Section 2 describes how collective communication operations can be arranged such that they consist of different steps, each performed on a subset of the entire set of processors. Section 3 presents the improvement in execution time obtained by such an arrangement on three different target platforms. Section 4 presents runtime functions to predict the behavior of execution time for collective communication operation based on orthogonal group structure. Section 5 applies the improved operations in the context of larger application programs and shows the resulting improvements. Section 6 discusses related work and Section 7 concludes the paper.

2 Orthogonal structures for realizing communication operations

The Message Passing Interface (MPI) standard has been defined in an effort to standardize the programming interface presented to developers arcoss a wide variety of parallel architectures. Many implementations of the standard are available, including highly-tuned versions for proprietary massively-parallel processors (MPPs), such as the Cray T3E, as well as hardware-independent implementations such as MPICH [4] and LAM-MPI [13], which have been ported to run on a large variety of machine types.

Most MPI implementations have been ported to cluster platforms, since clusters of commodity systems connected by a high-speed network in a rather loosely-coupled MPP are a cost effective alternative to supercomputers. As the implementations are not tuned towards a specific architecture interconnection network, such realizations of MPI communication operations can be inefficient for some communication operations on some platforms. Especially the network topology is crucial for an efficient realization of a given communication pattern. Furthermore, the latency and bandwidth of the interconnection network determine the switch between different communication protocols, e.g. for short and long messages.

In this section we describe how collective communication operations can be realized in consecutive phases based on an orthogonal partitioning of the processor set. The resulting orthogonal realizations can be used for arbitrary communication libraries that provide collective communication operations. We demonstrate this for MPI considering the following collective communication operations: a *single-broadcast* operation (*MPI_Bcast()*), a *gather* operation (*MPI_Gather()*), a *scatter* operation (*MPI_Scatter()*), a *single-accumulation* operation (*MPI_Reduce()*), a *multi-accumulation* operation (*MPI_Allreduce()*) and a *multi-broadcast* operation (*MPI_Allgather()*).

2.1 Realization using a two-dimensional processor grid

We assume that the set of processors is arranged as a two-dimensional virtual grid with a total number of $p = p_1 \times p_2$ processors. The grid consists of p_1 row groups $R_1, ..., R_{p_1}$
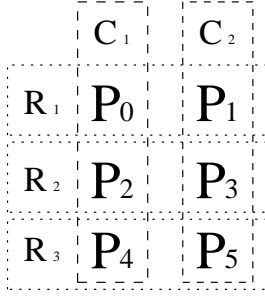
4

Figure 1: A set of 6 processors arranged as a two-dimensional grid with $p_1 = 3$ row groups and $p_2 = 2$ column groups in row-oriented mapping.
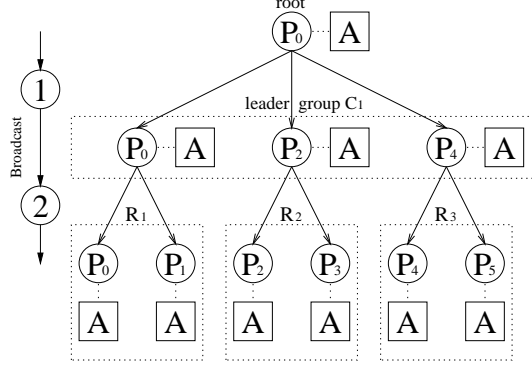

Figure 2: Illustration of an orthogonal realization of an *single-broadcast* operation with 6 processors and root processor P_0 realized by 3 concurrent groups of 2 processors each. In step (1), processor P_0 sends the message A within its column group C_1; this is the leader group. In step (2), each member of the leader group sends the message within its row group.

and p_2 column groups $C_1, ..., C_{p_2}$ with $|R_q| = p_2$ for $1 \leq q \leq p_1$ and $|C_r| = p_1$ for $1 \leq r \leq p_2$. The row groups provide a partitioning into disjoint processor sets. The disjoint processor sets resulting from column groups are orthogonal to the row groups. Using these two partitionings, the communication operations can be implemented in two phases, each working on a different partitioning of the processor grid. Based on the processor grid and the two partitionings induced, group and communicator handles are defined for the concurrent communication in the row and column groups. Based on the 2D grid arrangement, each processor belongs to one row group and to one column group. A row group and a column group have exactly one communication processor. Figure 1 illustrates a set of 6 processors $P_0, P_1, ..., P_5$ arranged as $p_1 \times p_2 = 3 \times 2$ grid.

The overhead for the processor arrangement itself is very small. Only two functions to create the groups are required and the arrangement has to be performed only once for an entire application program.

Single-Broadcast In a single-broadcast operation, a root processor sends a block of data to all processors in the communicator domain. Using the 2D processor grid as communication domain, the root processor first broadcasts the block of data within its column group C_1 (leader group). Then each of the receiving processors acts as a root in its corresponding 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 grid from Figure 1 with processor P_0 as root of the broadcast operation. We assume that the processors are organized into three concurrent groups of two processors each, i.e., there are $p_1 = 3$ row groups, each having $p_2 = 2$ members. Processors P_0, P_2 and P_4 form the leader group.

Gather For a gather operation, each processor contributes a block of data and the root processor collects the blocks in rank order. For an orthogonal realization, the data blocks are first collected within the row groups by concurrent group based gather operations such that 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 processor 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
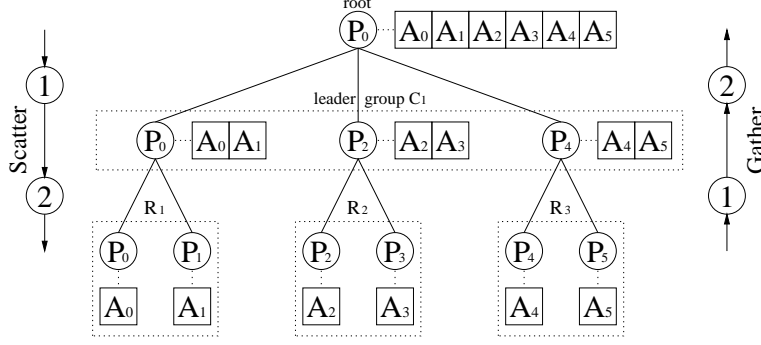
Figure 3: Illustration of an orthogonal realization of a *gather* operation (upward) and a *scatter* operation (downward) with 6 processors and root processor P_0 for 3 concurrent groups R_1, R_2 and R_3 of 2 processors each. In step (1), for the *gather* operation, processors P_0, P_2, P_4 concurrently collect messages from its row groups. In step (2), the leader group collects the messages built up in the previous step.

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 grid from Figure 1 where processor P_i contributes data block A_i, $i = 1, ..., 6$.

Scatter A scatter operation is the dual operation to a gather operation. Thus, a scatter operation can be realized by reversing the order of the two phases used for a gather operation: first, the messages are scattered in the leader group such that each processor in the leader group obtains all messages for processors in the row group to which it belongs to; then the messages are scattered in the row groups by concurrent group-based scatter operation, see Figure 3 (downward).

Single-Accumulation For a single-accumulation operation, each processor contributes a buffer with n elements and the root processor accumulates the values of the buffer with a specific reduction operation, like MPI_SUM. For an orthogonal realization, the buffers are first reduced within the row group by concurrent group-based reduce operations such that the elements are accumulated in that column group to which the root of the global reduce operation belongs to. In a second step, a reduce operation is performed within the leader group, thus accumulating all values of the buffer at the specific root processor. The numbers of elements are always the same which means that all messages in both phases have the same size.

Multi-Accumulation For a multi-accumulation operation, each processor contributes a buffer with n elements and the operation makes the result buffer available for each processor. Using a 2D processor grid, the operation can also be implemented by the following two steps: first, a group-based multi-accumulation operation is executed concurrently within the row groups, thus making the result buffer available to each processor of every row group. Second, concurrent group-based multi-accumulation operation are performed to reduce this buffer within the column groups. The messages have the same size in both phases.

Multi-Broadcast 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 grid, 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 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 $b \cdot p_2$. Again, the original rank order of data blocks is preserved.
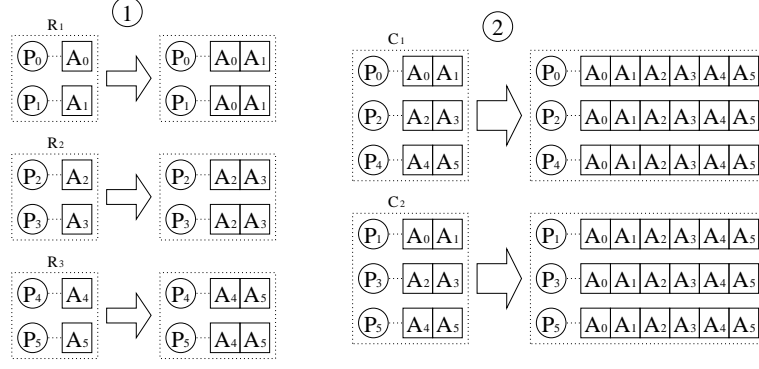
Figure 4: Illustration of an orthogonal implementation of *multi-broadcast* operation with 6 processors and root processor P_0. The operation may be realized by 3 concurrent groups R_1, R_2 and R_3 of 2 processors each and 2 orthogonal groups C_1 and C_2 of 3 processors each. Step (1) shows concurrent multi-broadcast operations on row groups and step (2) shows concurrent multi-broadcast operations on column groups.
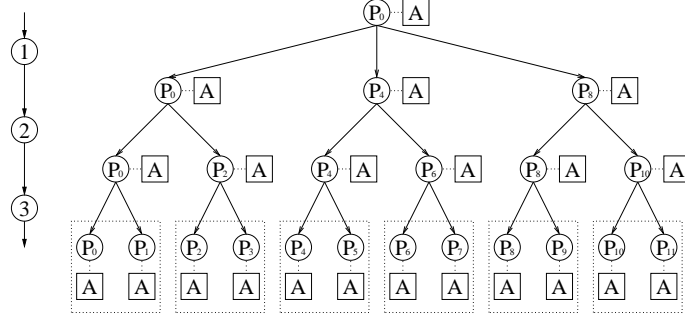

Figure 5: Illustration of an *MPI_Bcast()* operation with 12 processors and root processor P_0 using three communication phases.

2.2   Realization using a hierarchical processor grid

The idea from Section 2.1 can be applied recursively to the internal communication organization of the leader group or the concurrent groups, so that the communication in the leader group or the concurrent groups can be performed by again applying an orthogonal structuring of the group. This is illustrated in Figure 5 for a *single-broadcast* operation with 12 processors $P_0, P_1, ..., P_{11}$ and root processor P_0. We assume that the processors are organized in 6 concurrent groups of two processors each. The processors $P_0, P_2, ..., P_{10}$ forming the original leader group are again arranged as three concurrent groups of two processors such that the processors P_0, P_4 and P_8 form a first-level leader group. This results in three communication phases for the 12 processors as shown in Figure 5. Each hierarchical decomposition of a processor group leads to a new communication phase. For a fixed number of processors, the hierarchical decomposition can be selected such that the best performance improvement results. For three decompositions, we use a total number of $p = p_1 \cdot p_2 \cdot p_3$ processors, where p_1 denotes the size of the leader group; p_2 and p_3 denotes the size of the concurrent groups in the communication phases 2 and 3. In Figure 5, the numbers are $p1 = 3$ and $p_2 = p_3 = 2$.

3   MPI Performance results in isolation

To investigate the performance of the implementation described in Section 2.1, we consider communication on different distributed memory machines, a Cray T3E-1200, a Beowulf cluster and a dual Xeon cluster. The T3E uses a three-dimensional torus network. The six communication links of each node are able to simultaneously support hardware transfer rates of 600 MB/s. The Beowulf Cluster CLiC ('**C**hemnitzer **Li**nux **C**luster') is built up of 528 Pentium III processors clocked at 800 MHz. The processors are connected by two different networks, a communication network and a service network. Both are based on the fast-Ethernet-standard, i.e. the processing elements (PEs) can swap 100 MBit per second. The service network (Cisco Catalyst) allows external access to the cluster. The communication network (Extreme Black Diamond) is used for inter-process communication between the PEs. On the CLiC, LAM MPI 6.3 b2 and MPICH 1.2.4 were used for the experiments.

The Xeon cluster is built up of 16 nodes and each node consists of two Xeon processors clocked at 2 GHz. The nodes are connected by three different networks, a service network and two commmunication networks. The service network and one communication network are based on the fast-Ethernet-standard and the functionality is similar to the two interconnection networks of the CLiC. Additionally, a high performance interconnection network based on Dolphin SCI interface cards is available. The SCI network is connected as 2-dimensional torus topology and can be used by the ScaMPI (SCALI MPI) [3] library. The fast-Ethernet based networks are connected by a switch and can be used by two portable MPI libraries, LAM MPI 6.3 b2 and MPICH 1.2.4.

In the following, we present runtime tests on the three platforms. On the CliC and the Cray T3E we present the results for 48 and 96 processors. For other processor numbers, similar results have been obtained. For 48 processors 8 different two-dimensional virtual grid layouts (2×24, 3×16, ..., 24×2) and for 96 processors 10 different grid layouts (2×48, 3×32, ..., 48×2) are possible. For the runtime tests, we have used message sizes between 10 KBytes and 500 KBytes, which is the size of the block of data contributed (e.g. *MPI_Gather()*) or obtained (e.g. *MPI_Scatter()*) by each participating processor. The following figures show the minimum, average and maximum performance improvements achieved by the orthogonal implementation described in Section 2.1 compared with the original MPI implementation over the entire interval of message sizes.

For the dual Xeon cluster we present runtime tests for 16 and 32 processors for both communication networks, i.e. for the Ethernet and the SCI interconnection network. For the Ethernet network LAM MPI 6.3 b2 and for the SCI interface ScaMPI 4.0.0 have been used. The processor layouts are similarly chosen, i.e. for 16 processors 3 different two-dimensional grid layouts (2×8, 4×4, 8×2) and for 32 processors 4 different layouts (2×16, 4×8, 8×4, 16×2) are possible.

3.1   Orthogonal realization using LAM-MPI on the CLiC

On the CLiC, the orthogonal implementations based on the LAM-MPI library lead to the highest performance improvements for most collective communication operations.

The orthogonal realizations of an *MPI_Bcast()*, *MPI_Allgather()* and *MPI_Allreduce()* operation show the most significant performance improvements. All partitions show a considerable improvement, but the largest improvements can be obtained when using a layout for which the number of row and column groups are about the same. The *MPI_Bcast()* operation shows significant average improvements of more than 20% for 48 and 40% for 96 processors, respectively, using balanced grid layouts, see Figure 7 (left). The orthogonal implementation of a *MPI_Allreduce()* operation shows average improvements of more than
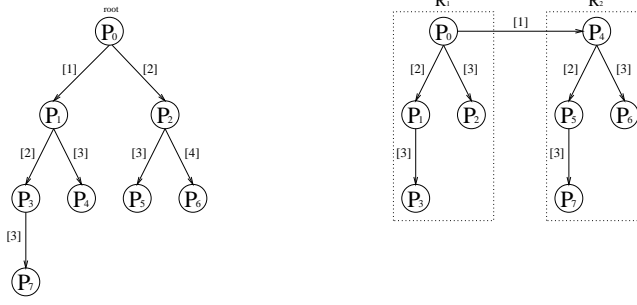
Figure 6: Illustration of a *MPI_Bcast()* operation with 8 processors $P_0, ..., P_7$ and root processor P_0 using a binary tree algorithm. The figure shows a standard implementation using in LAM-MPI (left) and an orthogonal realization with two groups R_1 and R_2 of 4 processors each (right). The processors P_0 and P_4 form the leader group. The number in the squared bracket denotes the message passing step to distribute the data block. The standard algorithm needs 4 and the orthogonal realization 3 message passing steps to distribute the block of data.
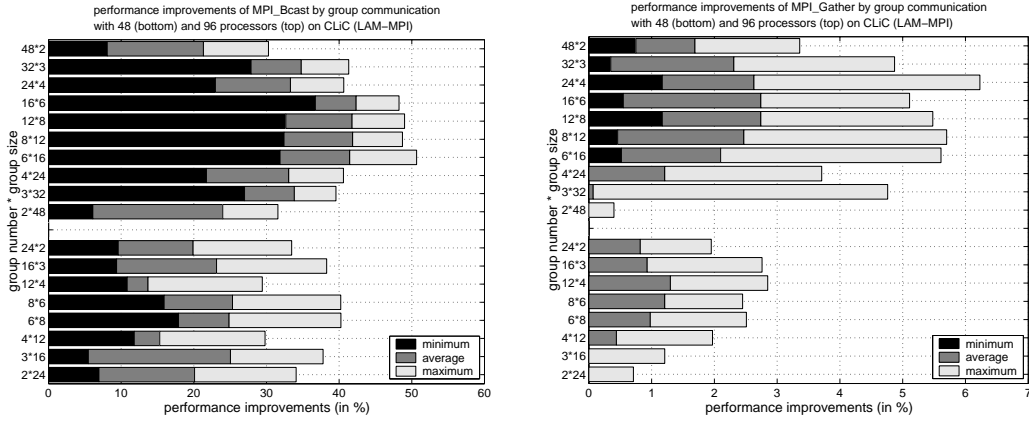

Figure 7: Performance improvements by group-based realization of *MPI_Bcast()* (left) and *MPI_Gather()* (right) with 48 and 96 processors on the CLiC (LAM-MPI).

20% for 48 and 30% for 96 processors, respectively, again using balanced group sizes, see Figure 8 (left). The execution time of the *MPI_Allgather()* operation can be dramatically improved by an orthogonal realization, see Figure 8 (right). For some of the group partitionings, improvements of over 60% for 48 and 70% for 96 processors, respectively, can be obtained. The difference between the minimum and maximum performance enhancements are extremely small, which means that this method leads to a reliable improvement for all message sizes.

The main reason for the significant performance improvements of these three collective communication operations achieved by orthogonal realization is the specific implementation of the *MPI_Bcast()* operation in LAM-MPI. The algorithm of the *MPI_Bcast()* operation to distribute the block of data does not exploit the star network topology of the CLiC, but uses a structure describing a tree topology. In general, the orthogonal realization leads to a better utilization of the network caused by a more balanced communcation pattern. Figure 6 illustrates the message passing steps of a binary broadcast-tree with 8 processors $P_0, ..., P_7$ and demonstrates the benefits of an orthogonal realization.

Both the *MPI_Allgather()* and the *MPI_Allreduce()* operation in the LAM implementation use a *MPI_Bcast()* operation to distribute the block of data to all participating processors. The *MPI_Allreduce()* operation is composed of an *MPI_Reduce()* and an *MPI_Bcast()* operation. First the root processor reduces the blocks of data from all members of the processor group and broadcasts the result buffer to all processors participating in the
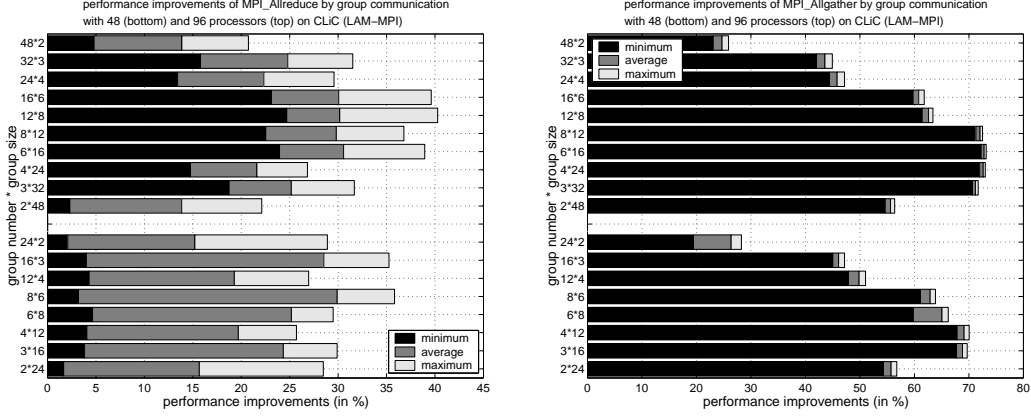
Figure 8: Performance improvements of *MPI_Allreduce()* (left) and *MPI_Allgather* (right) by group communication with 48 and 96 processors on the CLiC (LAM-MPI).

communication operation. The message size is constant for both operations. The improvements correspond to the performance enhancements of the *MPI_Bcast()* operation, since the preceding *MPI_Reduce()* operation with orthogonal structure leads to a small performance degradation. The *MPI_Allgather()* operation is composed of an *MPI_Gather()* and an *MPI_Bcast()* operation. First the root processor collects blocks of data from all members of the processor group and broadcasts the entire message to all processors participating in the communication operation. The root processor broadcasts a considerably larger message of size $b \cdot p$, when b denotes the original message size and p is the number of participating processors. The dramatic improvements are again caused by the execution of the *MPI_Bcast()* operation for the larger message size.

The orthogonal implementation of the *MPI_Gather()* operation shows a small, but persistent average performance improvement for all grid layouts of more than 1% for 48 and 2% for 96 processors, respectively, see Figure 7 (right). There are only small variations of the improvements obtained for different layouts, but using the same number of row and column groups again leads to the best average performance. An average performance degradation can be observed for the *MPI_Scatter()* and the *MPI_Reduce()* operation. Only for specific message sizes, a small performance improvement can be obtained, not shown in a figure.

3.2   Orthogonal realization using MPICH on the CLiC

The performance improvements on the CLiC based on the MPICH library are not as significant as with LAM-MPI, but also with MPICH persistent enhancements by an orthogonal realization can be obtained for some collective communication operations.
The orthogonal implementations of the *MPI_Gather()* and *MPI_Scatter()* operations lead to small, but persistent performance enhancements. For the *MPI_Gather()* operation more than 1% for 48 and 2% for 96 processors, respectively, can be obtained using balanced grid layouts, see Figure 9 (left). Similar results are shown in Figure 9 (right) for the *MPI_Scatter()* operation. Depending on the message size up to 5% performance enhancements can be obtained with an optimal grid layout in the best case.

The orthogonal realization of the *MPI_Allgather()* operation leads to very large performance improvements for message sizes in the range of 32 KByte and 128 KByte, see Figure 12 (left); for larger message sizes up to 500 KByte slight performance degradation between 1 % and 2 % can be observed. The main reason for the large differences in the improvements depending on the message size are the different protocols used for short and long messages. Both protocols are realized using non-blocking *MPI_Isend()* and
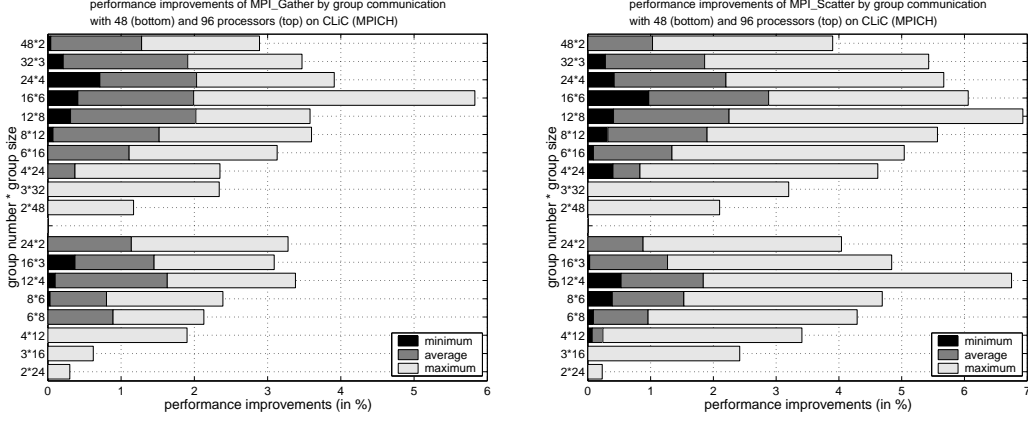
Figure 9: Performance improvements by group-based realization of *MPI_Gather()* (left) and *MPI_Scatter()* (right) with 48 and 96 processors on the CLiC (MPICH).

MPI_Irecv() operations. For messages up to 128 KBytes, an *eager* protocol is used where the receiving processor stores messages that arrive before the corresponding *MPI_Irecv()* operation has been activated in a system buffer. This buffer is allocated each time that such a message arrives. Issuing the *MPI_Irecv()* operation leads to copying the message from the system buffer to the user buffer. For messages that are larger than 128 KBytes, a *rendezvous* protocol is used that is based on request messages send by the destination processor to the source processor as soon as a receive operation has been issued, so that the message can be directly copied into the user buffer. The reason for the large improvements for short messages shown in Figure 12 (left) is caused by the fact that the asynchronous standard realization of the *MPI_Allgather()* operation leads to an allocation of a temporary buffer and a succeeding copy operation for a large number of processors whereas the orthogonal group-based realization uses the *rendezvous* protocol for larger messages in the second communication phase because of an increased message size $b \cdot p_2$. Slight performance degradations between 1% and 2% can also be observed for the *MPI_Reduce()*, *MPI_Allreduce()* and the *MPI_Bcast()* operations by an orthogonal realization which is not shown in a figure.

3.3 Orthogonal realization using LAM-MPI on the dual Xeon cluster

The Xeon Cluster consists of 16 nodes with two processors per node. The processors participating in a communication operation are assigned to the nodes in a cyclic order to achieve a reasonable utilization of both interconnection networks. For runtime tests with 16 processors all 16 nodes are involved, i.e., processor i uses one physical processor of node i for $0 \leq i \leq 15$. When 32 processors participate in the communication operation, node i provides the processors i and $i + 16$ for $0 \leq i \leq 15$.

The performance results of the different communication operations are similar to the performance enhancements using LAM-MPI on the CLiC. The main reason is that both platforms use a star network topology, the same interconnection network (fast-Ethernet) and the same realization of communication operation based on the LAM-MPI library. Figure 10 shows as example that for an *MPI_Bcast()* (left) and an *MPI_Allgather* (right) operation similar performance improvement as on the Beowulf cluster can be obtained. Because of the specific processor arrangement of the cluster the performance improvements of the various two-dimensional group layouts differ from the performance results on the CLiC, such that a balanced grid layout does not necessarily lead to the best average performance improvement. Concerning performance improvements and grid layouts sim-
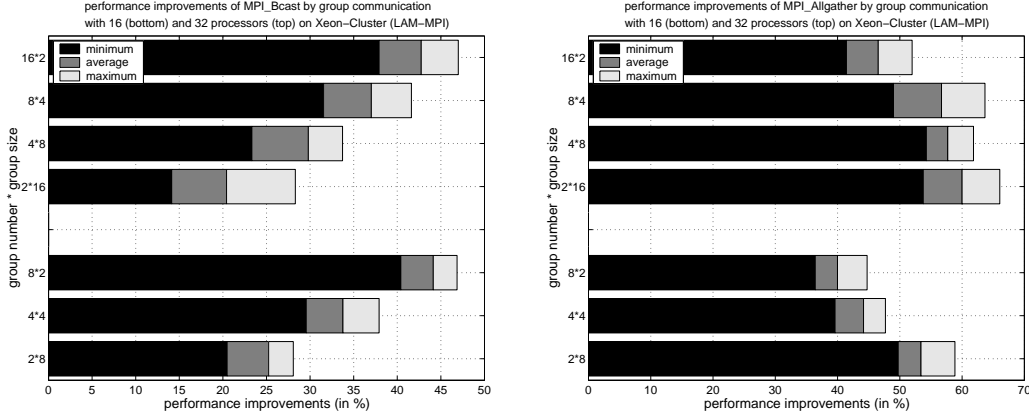
Figure 10: Performance improvements by group-based realization of *MPI_Bcast()* (left) and *MPI_Allgather()* (right) with 16 and 32 processors on the dual Xeon cluster (LAM-MPI).

ilar performance improvements on the CLiC can be observed for the remaining collective MPI communication operations.

3.4   Orthogonal realization using ScaMPI on the Xeon cluster

In general, collective communication operations using the two-dimensional SCI torus are significantly faster than operations using an Ethernet network. Depending on the specific communication operation the SCI interface is by a factor of 100 faster than the Ethernet network. Several collective communication operations using ScaMPI on SCI still show performance improvements obtained by orthogonal group realization, see Figure 11 for an *MPI_Gather* (left) and *MPI_Allgather* (right) operation for smaller message sizes. For *MPI_Scatter* similar performance results like for *MPI_Gather* can be observed. For *MPI_Bcast* and the accumulation operations slight performance degradations can be observed. The assignment of processors participating in the communication operation to the cluster nodes is done as described in Section 3.3.
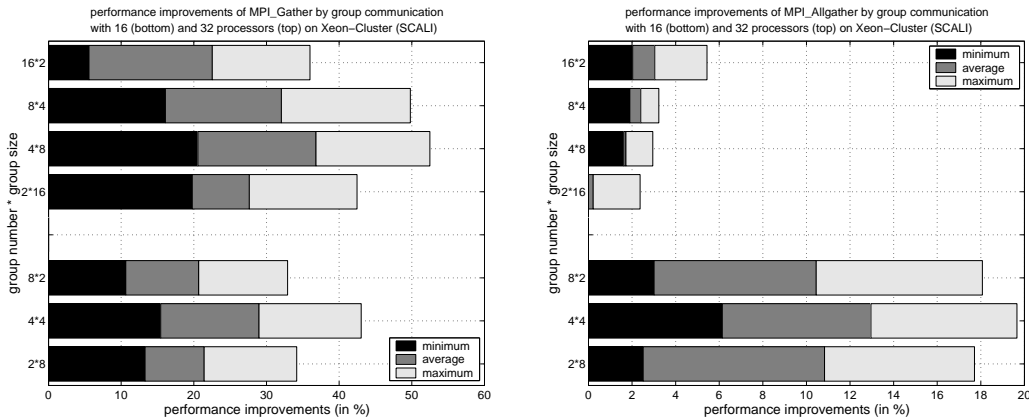

Figure 11: Performance improvements by group-based realization of *MPI_Gather()* (left) for message sizes in the range of 560 Byte and 64 KByte and *MPI_Allgather()* (right) for message sizes between 100 KByte and 500 KByte on the Xeon cluster (ScaMPI).

3.5   Orthogonal realization on the Cray T3E-1200

The Cray T3E is a distributed shared memory system in which the nodes are interconnected through a bidirectional 3D torus network. The T3E network uses a deterministic,
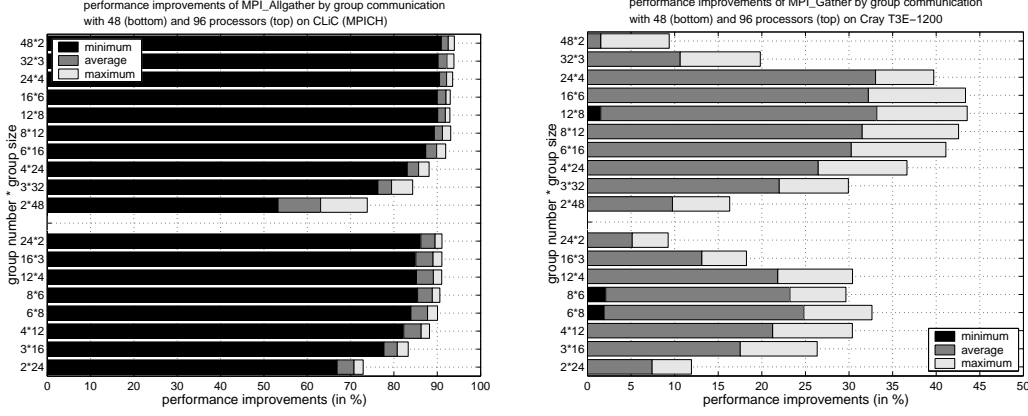
Figure 12: Performance improvements by group-based realization of *MPI_Allgather()* for message sizes in the range of 32 KByte and 128 KByte on the CLiC (MPICH) (left) and *MPI_Gather()* for messages sizes between 10 KByte and 500 KByte on the Cray T3E-1200 (right).

dimension-order, k-ary n-cube wormhole routing scheme [17]. Each node contains a table, stored in dedicated hardware, that provides a routing tag for every destination node. In deterministic routing, the path from source to destination is determined by the current node address and the destination node address so that for the same source-destination pair all packets follow the same path. Dimension-order routing is a deterministic routing scheme in which the path is selected that the network is traversed in a predefined monotonic order of the dimensions in the torus. Deadlocks are avoided in deterministic routing by ordering the virtual channels that a message needs to traverse. Dimension-order routing is not optimal for k-ary n-cubes. Because of the wraparound connections, messages may get involved in deadlocks while routing through the shortest paths. In fact, messages being routed along the same dimension (a single dimension forms a ring) may be involved in a deadlock due to a cyclic dependency. A non-minimal deadlock-free deterministic routing algorithm can be developed for k-ary n-cubes by restricting the use of certain edges so as to prevent the formation of cycles [5]. In general, when an algorithm restricts message routing to a fixed path, it cannot exploit possible multiple paths between source-destination pairs during congestion.

There considerations show that a standard MPI communication operation does not automatically lead to an optimal execution time on the T3E. Furthermore, a rearrangement of a processor set as smaller groups of processors may prevent congestions thus leading to smaller execution times. This will be shown in the following runtime tests. The application uses the virtual PE number to reference each PE in a partition and has *no* direct access to logical or physical PE addresses. The hardware is responsible for the conversion of virtual, logical and physical PE numbers.

For *MPI_Bcast()* and *MPI_Allgather()* operations, good performance improvements up to 20% can be obtained when using suitable grid layouts for messages in the range of 10 KByte and 500 KByte. The execution times of the orthogonal realizations are quite sensible to the grid layout and the specific message size, i.e. other grid layouts lead to smaller improvements or may even lead to performance degradation. Moreover, there is a large variation of performance improvements especially for large messages where messages of similar size may lead to a significant difference in the performance improvement obtained. This leads to large differences between the minimum and maximum improvement. In contrast, smaller message sizes in the range of 10 KByte and 100 KByte lead to persistent average performance improvements for both operations, see Figure 13 for the *MPI_Bcast()* (left) and *MPI_Allgather()* (right) operations.

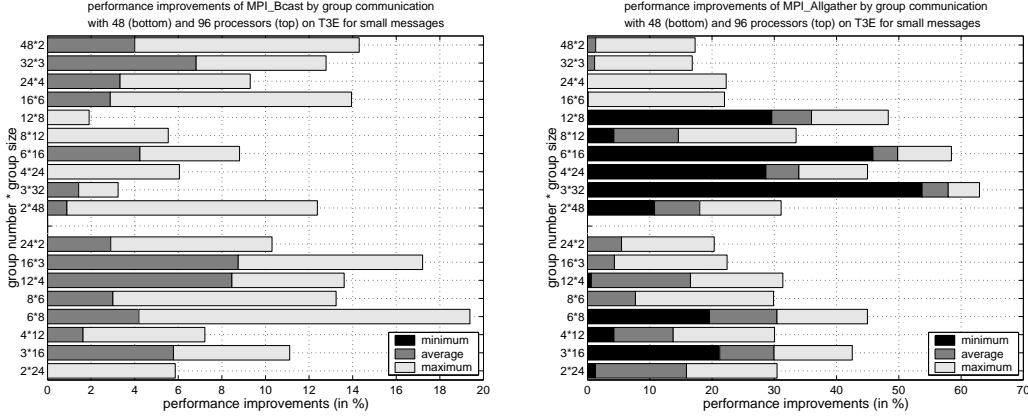For the *MPI_Gather()* operation a significant performance improvement of more than

Figure 13: Performance improvements by group communication of *MPI_Bcast()* (left) and *MPI_Allgather()* (right) for message sizes in the range of 10 KByte and 100 KByte on the Cray T3E-1200.

20% for 48 and 30% for 96 processors, respectively, can be obtained, see Figure 12 (right). For small message sizes, a slight performance degradation can sometimes be observed. Therefore there is no minimum improvement shown for most of the layouts in the figure. For message sizes between 128 KBytes and 500 KBytes, the improvements obtained are nearly constant. The runtimes for *MPI_Gather()* operations increase more than linearly with the number p of processors. which is caused by the fact that the root processor becomes a bottleneck when gathering larger messages. This bottleneck is avoided when using orthogonal group communication.

Slight performance degradations between 1% and 2% are obtained for the *MPI_Scatter()*, *MPI_Reduce()* and *MPI_Allreduce()* operations. Neither for smaller nor for larger message sizes performance enhancements can be observed.

3.6 Performance results for hierarchical orthogonal organization

On the CLiC and T3E a sufficiently large number of processors is available to arrange different grid layouts for three communication phases. For up to 96 processors, up to three hierarchical decompositions according to Section 2.2 are useful and we present runtime tests for 96 processors on the CLiC (with LAM-MPI) and on the Cray T3E. In particular, if the original leader group contains 16 or more processors, it is reasonable to decompose this again and the communication is performed in three instead of two phases. Compared to the two-phase realization, a hierarchical realization of the *MPI_Bcast()* and *MPI_Gather()* operation leads to additional and persistent performance improvements on the CLiC and T3E.

Hierarchical realization using LAM-MPI on the CLiC   Comparing a two-dimensional with a three-dimensional realization for the *MPI_Bcast()* operation, an *additional* performance improvement of up to 15% can be obtained for the CLiC using LAM-MPI. The additional average performance improvement lies above 10% for some of the group partitionings, see Figure 14 (left). The hierarchical realization for the *MPI_Gather()* operation shows no additional performance improvements compared to the two-dimensional realization, see Figure 14 (right). The resulting differences between the minimum and maximum performance improvements are larger for all message sizes than for two-phase realization.

Hierarchical realization on the Cray T3E-1200   For the *MPI_Bcast()* operation *all* group partitionings show an average performance improvement compared to the runtime
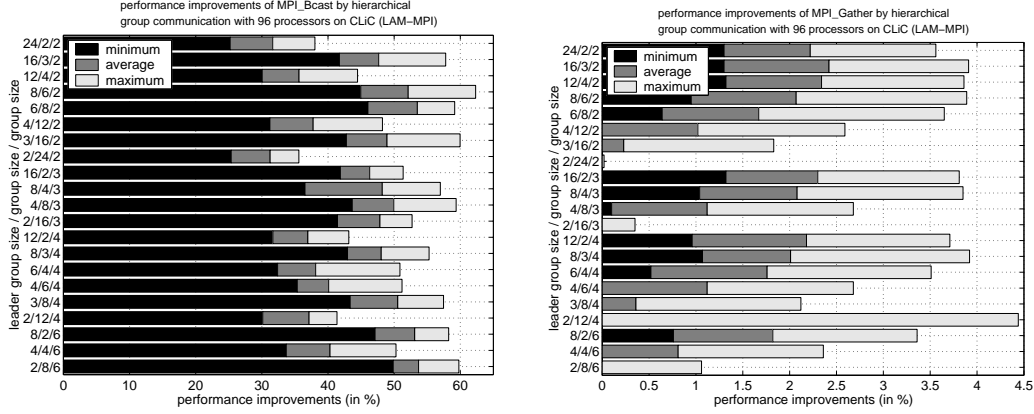
Figure 14: Performance improvements by hierarchical group communication of *MPI_Bcast()* (left) and *MPI_Gather()* (right) on the CLiC (LAM-MPI).
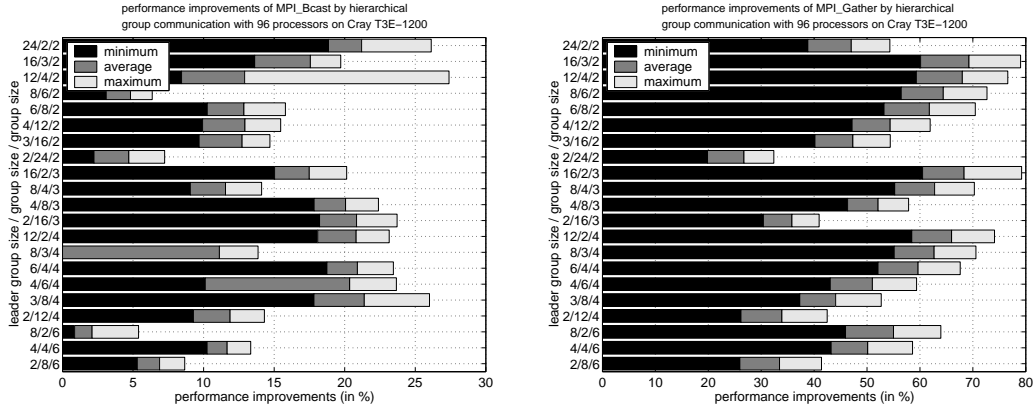

Figure 15: Performance improvements by hierarchical group communication of *MPI_Bcast()* (left) and *MPI_Gather()* (right) on the Cray T3E-1200.

tests with two communication phases for message sizes up to 500 KByte on the T3E, see Figure 15 (left). For suitable grid layouts average improvements of more than 20% can be obtained. Also for *MPI_Gather()* the hierarchical realization with three communication phases leads to additional performance improvements, see Figure 15 (right). The improvements vary depending on the group partitionings. For some of the group partitionings, additional improvements of over 60 % can be obtained.

Hierarchical realization on the Xeon cluster Figure 16 shows performance enhancements for four MPI communication operations obtained by a hierarchical orthogonal grid layout with three communication phases. Since 32 processors are available three different group layouts ($2 \times 8 \times 4$, $4 \times 4 \times 2$, $8 \times 2 \times 2$) are chosen for the Xeon cluster. Figure 16 shows the additional performance improvements for the orthogonal realization of *MPI_Bcast*, *MPI_Allgather* using LAM-MPI (left) and *MPI_Gather*, *MPI_Scatter* using ScaMPI (right). The orthogonal realizations using ScaMPI are obtained for smaller message sizes in the range of 560 Byte and 64 KByte, see also Section 3.4.

3.7 Grid Selection

For a given machine and a given MPI implementation a different layout of the processor grid lead to the largest performance improvement. A good layout of the processor grid can be selected by performing measurements with different grid layouts and different message sizes for each of the collective communication operations to be optimized.
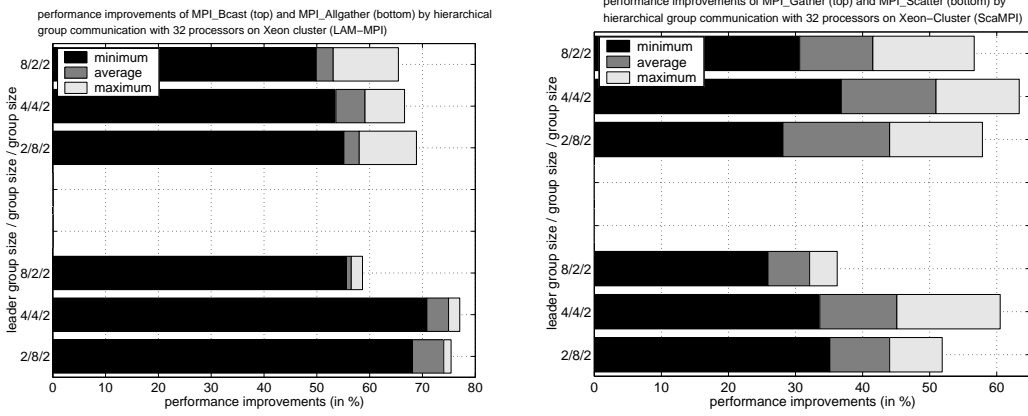
Figure 16: Performance improvements by hierarchical group communication of *MPI_Bcast()*, *MPI_Allgather()* using LAM-MPI (left) and *MPI_Gather()*, *MPI_Scatter()* using ScaMPI (right) on the Xeon cluster. The improvements for orthogonal realizations using ScaMPI (right Figure) are obtained for message sizes between 560 Byte and 64 KByte.

The process of obtaining and analyzing the measurements can be automated such that for each communication operation, a suitable layout is determined that leads to small execution times. This process has to be done only once for each target machine and each MPI implementation and the selected layout can then be used for all communication operations in all application programs. In general, different optimal layouts may result for different communication operations, but our experiments with LAM-MPI, MPICH and Cray-MPI show that using the same number of row and column groups usually leads to good and persistent improvements compared to the standard implementation of the MPI operations.

Also, different grid layouts may lead to the best performance improvement when considering different message sizes for the same communicaton operation. Based on the measured execution times of the communication operation, it is also possible to identify intervals of message sizes such that a different grid layout is selected for a different interval of message sizes, depending on the expected performance improvement. The measured data also shows whether for a specific communication operation and for a specific message size, no performance improvement is expected by an orthogonal realization so that the original MPI implementation can be used. Based on the selection of appropriate grid layouts in two- or multi-dimensional forms an optimized collective communication operation is realized offering the best improvements possible using the orthogonal approach. The result of the analysis step is a realization of the collective communication operations that uses orthogonal realization with a suitable layout whenever it is expected that this leads to a performance improvement compared to the given MPI implememtation.

4 Performance modeling of orthogonal group structures

In this section, we consider the performance modeling of the orthogonal realization of collective communication operations using runtime functions. Runtime functions have been successfully used to model the execution time of communication operation for various communication libraries [7, 15]. The execution of an MPI_Bcast broadcast operation, for

example, on the CLiC using LAM-MPI 6.3 b2 can be described by the runtime function

$$t_{sb}(p, b) = (0.0383 + 0.474 \cdot 10^{-6} \cdot log_2(p)) \cdot b,$$

where b denotes the message size in bytes and p is the number of processors participating in the communication operation. For the performance modeling of orthogonal implementations of collective communication operations we adopt the approach for modeling the execution time of each phase of the orthogonal implementation in isolation. For each phase we use the runtime functions of the monolithic standard MPI communication operations from Table 1. The coefficients τ_1 and t_c can be considered as startup time and byte-transfer time, respectively, and are determined by applying curve fitting with the least-squares method to measured execution times. For the measurements, message sizes between 10 KBytes and 500 KBytes have been used. In some of the formulas the startup time is very small and can be ignored. In the following, we consider the modeling of orthogonal realizations of some important MPI operations.

MPI_Bcast For the broadcast operation the communication time is modeled by adding the runtime function for the broadcast in the leader group (using the formula from Table 1 with $p = p_1$) and the runtime function for the concurrent broadcast in the row groups (using the formula from Table 1 with $p = p_2$). The accurate predictions for the concurrent groups and leader groups show that this approach can be used for all collective MPI communication operations. The good prediction results of the two-phase performance modeling also show that there is no interferences of concurrent communication operations in the second communication phase. A possible interference would lead to a delayed communication time and, thus, would require a different modeling approach.

For the single-broadcast operations, LAM-MPI uses two different internal realizations, one for $p \le 4$ and one for $p > 4$. If up to 4 processors participate in the broadcast operation, a formula results that depends linearly on p. For more than 4 processors a formula with a logarithmic dependence on p is used, because the broadcast transmissions are based on broadcast trees with logarithmic depth. The corresponding coefficients are given in Table 2.

Figure 17 (left) shows the deviations between measured and predicted execution times for single-broadcast on the CLiC for an orthogonal realization. For different group layouts the deviations are gives separately for the leader group (LG) used in the first phase and the for the concurrent groups (CG) used in the second phase. The bar *total* shows the accumulated deviation of both communication phases. The figure shows minimum, maximum, and average deviations between measured and predicted runtimes over the entire interval of message sizes. The predictions are quite accurate but not absolutely precise for some groups, because the depth of the broadcast tree remains constant for a specific interval of processor sizes. This means that the communication time increases in stages and the runtime formulas do not model these stages exactly. Figure 17 (right)

operation	runtime function
MPI_Bcast	$t_{sb\_lin}(p, b) = (\tau + t_c \cdot p) \cdot b$
	$t_{sb\_log}(p, b) = (\tau + t_c \cdot log_2(p)) \cdot b$
MPI_Gather	$t_{sc}(p, b) = \tau_1 + (\tau_2 + t_c \cdot p) \cdot b$
MPI_Scatter	$t_{ga}(p, b) = \tau_1 + (\tau_2 + t_c \cdot p) \cdot b$
MPI_Reduce	$t_{acc\_lin}(p, b) = (\tau + t_c \cdot p) \cdot b$
	$t_{acc\_log}(p, b) = (\tau + t_c \cdot \lfloor log_2(p - 1) \rfloor) \cdot b$
MPI_Allreduce	$t_{macc}(p, b) = t_{acc}(p, b) + t_{sb}(p, b)$
MPI_Allgather	$t_{mb}(p, b) = t_{ga}(p, b) + t_{sb}(p, p \cdot b)$

Table 1: Runtime functions for collective communication operations on the CLiC

coefficients for broadcast and accumulation							
			CLiC			Xeon cluster	
operation	formula	p	$\tau[\mu s]$	$t_c[\mu s]$	p	$\tau[\mu s]$	$t_c[\mu s]$
MPI_Bcast	$t_{sb\_lin}(p,b)$	≤ 4	-0.085	0.092	-	-	-
	$t_{sb\_log}(p,b)$	> 4	0.038s	0.474	> 1	-0.0005	0.0042
MPI_Reduce	$t_{acc\_lin}(p,b)$	≤ 4	-0.103	0.105	-	-	-
	$t_{acc\_log}(p,b)$	> 4	0.141	0.101	> 1	0.0116	0.0002

Table 2: Coefficients of the runtime function for *MPI_Bcast()* and *MPI_Reduce()* on the CLiC (LAM-MPI) and on the dual Xeon cluster (ScaMPI).
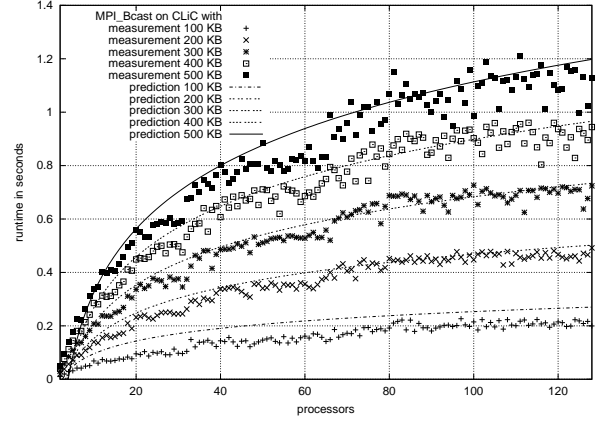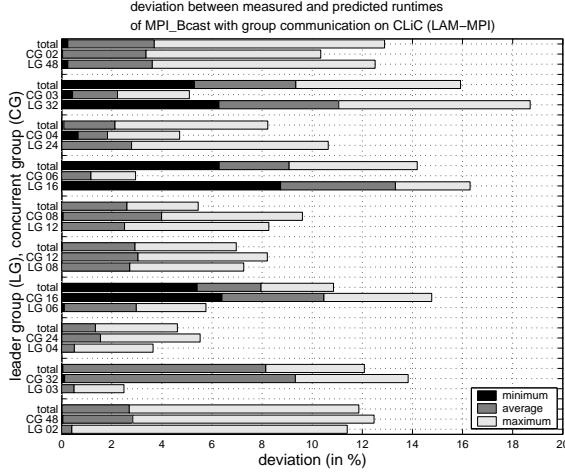

Figure 17: Deviations between measured and predicted runtimes for 96 processors (left) and modeling (right) of *MPI_Bcast()* on the CLiC.

shows the measured and predicted runtime of the single-broadcast operation as a function of the number of processors for fixed message sizes given in the key.

Figure 18 (left) shows the predictions for the dual Xeon cluster using ScaMPI and in Table 2 the coefficients of the runtime functions are shown.

MPI_Gather The coefficients of the runtime functions for the CLiC are shown in Table 3. The predictions fit the measured runtimes quite accurately, see Figure 19 (left). The approximations are quite accurate for the entire interval of message sizes. The average deviations between measured and predicted runtimes lie clearly below 3 % in most cases. On the T3E, the runtimes of *MPI_Gather()* operations increase more than linearly with the number p of processors. This effect might be caused by the fact that the root processor is a bottleneck when gathering larger messages. This observation can be used to obtain good performance improvements by orthogonal realizations. To capture the sharp increases of the runtimes we use different runtime functions for different message sizes. Each increase can be captured by a specific formula, see Table 4. The use of a specific formula depends on the root message size which is the size of the message that the root processor is gathering

coefficients for gather and scatter						
	CLiC			Xeon cluster		
operation	$\tau_1(V)[s]$	$\tau_2(V)[\mu s]$	$t_c(V)[\mu s]$	$\tau_1(V)[s]$	$\tau_2(V)[\mu s]$	$t_c(V)[\mu s]$
MPI_Gather	0.009	-0.0825	0.0929	0.00	-0.0056	0.0040
MPI_Scatter	0.00	-0.0730	0.0897	0.00	-0.0032	0.0039

Table 3: Coefficients for runtime function of *MPI_Gather()* and *MPI_Scatter()* on the CLiC (LAM-MPI) and on the dual Xeon cluster (ScaMPI).

runtime functions of gather/scatter operations on Cray T3E-1200			
MPI_Gather			
No.	p	$n[kbyte]$	runtime function
1	002 - 128	≤ 8448	$T_1(p,b) = (\tau_2 + t_c \cdot p) \cdot b$
2	017 - 032	> 8448	$T_2(p,b) = \tau_1 + (\tau_2 + t_c \cdot p) \cdot b$
3	033 - 128	> 8448	$T_3(p,b) = \tau_1 + (\tau_2 + t_c \cdot p) \cdot b$
MPI_Scatter			$T(p,b) = (\tau_2 + t_c \cdot p) \cdot b$

Table 4: Runtime functions of *MPI_Gather()* and *MPI_Scatter()* on Cray T3E

operation	No.	$\tau_1(V)[s]$	$\tau_2(V)[\mu s]$	$t_c(V)[\mu s]$
MPI_Gather	1	-	-0.00134	0.00308
	2	-0.020	0.0157	0.00505
	3	-0.036	0.0265	0.00617
MPI_Scatter	-	-	0.0002453	0.00297

Table 5: Coefficients for runtime function of *MPI_Gather()* and *MPI_Scatter()* on Cray T3E-1200.

from all members of the processor group. Above 8448 KBytes a different runtime formula is used. For the first formula no startup-time is necessary. The values of the coefficients are shown in Table 5. The prediction for the dual Xeon cluster using ScaMPI are given in Figure 18 (right), the coefficients of the runtime function are shown in Table 3. Figure 19 (right) shows the deviations between measured and predicted runtimes on the T3E. The approximations are quite accurate for the entire interval of message sizes. The average deviations of 16 different processor groups (leader and concurrent groups are modeled separately) are clearly below 3 %.

MPI_Scatter The runtime formulas for the predictions of the scatter operations are also given in Table 4. The values of the coefficients are shown in Table 5. The approximations of the scatter operation show the best results on both systems. On the CLiC the predictions fit the measured runtimes very accurately with an average deviations below 2 % for most processor groups over the entire interval of all message sizes, see also Figure 20 (left). The deviations of the leader group lies below 1 % for almost all group sizes. The deviations between measurements and predictions of the *MPI_Scatter()* operation on the Cray T3E-1200 are shown in Figure 20 (right).

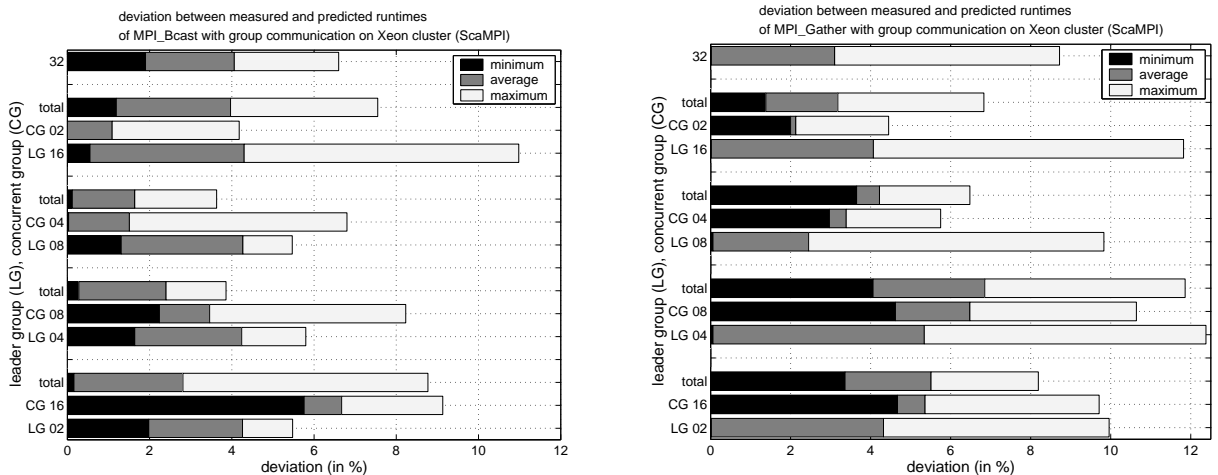**MPI_Reduce** The modeling of *MPI_Reduce()* operations is performed with the formula

Figure 18: Deviations between measured and predicted runtimes for *MPI_Bcast()* (left) and *MPI_Gather()* (right) on the dual Xeon cluster using ScaMPI.
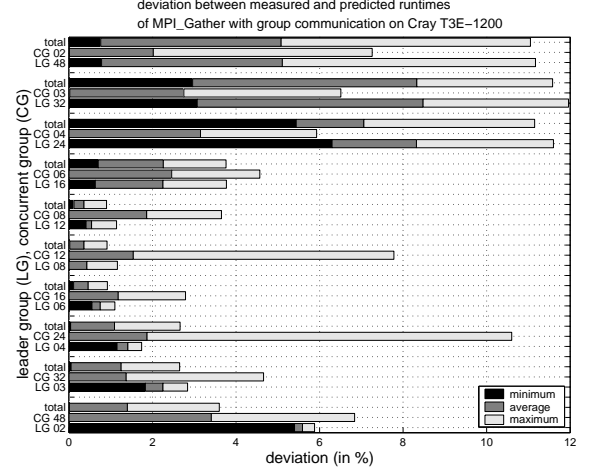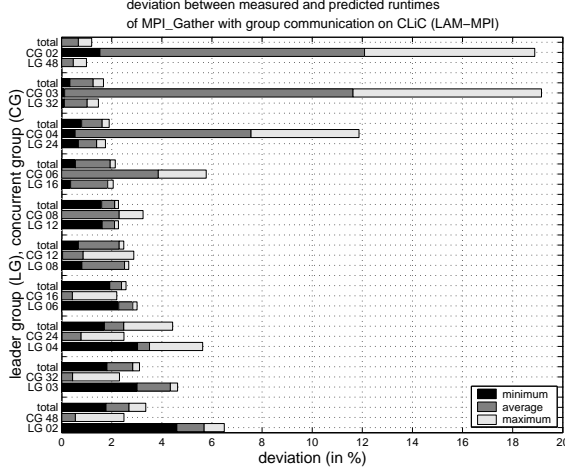
Figure 19: Deviations between measured and predicted runtimes for *MPI_Gather()* on the CLiC (left) and Cray T3E-1200 (right) for 96 processors.
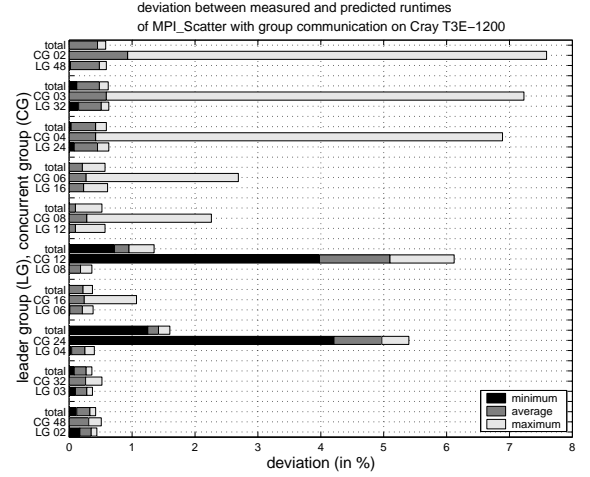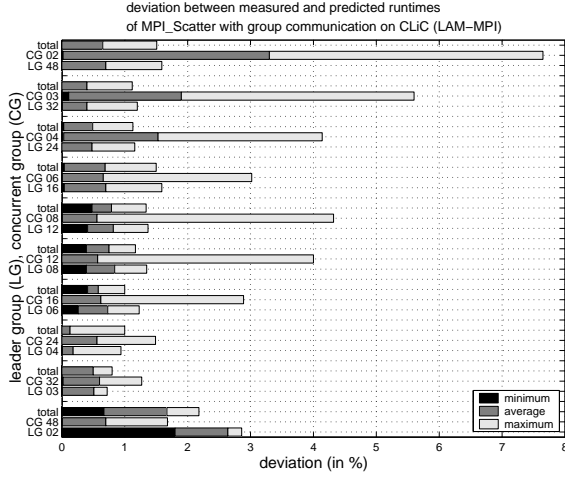

Figure 20: Deviations between measured and predicted runtimes for *MPI_Scatter()* on the CLiC (left) and Cray T3E-1200 (right) for 96 processors.

from Table 1. LAM-MPI uses a different internal realization for $p \leq 4$ than for $p > 4$. The specific values for the coefficients are shown in Table 2 for both cases. The communication time of a reduce operation increases in stages, because the number of time steps to accumulate an array depends on the depth of the reduce tree. A detailed analysis shows that the number of processors can be partitioned into intervals such that for all processor numbers within an interval reduce trees with the same depth are used. The predictions are very accurate and the average deviations between measured and predicted runtimes lie below 3 % for most cases.

MPI_Allreduce In LAM-MPI the *MPI_Allreduce()* operation is composed of an *MPI_Reduce()* and an *MPI_Bcast()* operation. First the root processor reduces the block of data from all members of the processor group und broadcasts the reduced array to all processors participating in the communication operation. The size of the array is constant in both phases. Figure 21 shows specific measured and predicted runtimes with fixed message sizes.

MPI_Allgather In LAM-MPI the *MPI_Allgather()* operation is composed of an *MPI_Gather()* and an *MPI_Bcast()* operation. At first the root processor gathers a block of data from each member of the processor group und broadcasts the entire message to all processors participating in the communication operation. The entire message has size $p \cdot b$,
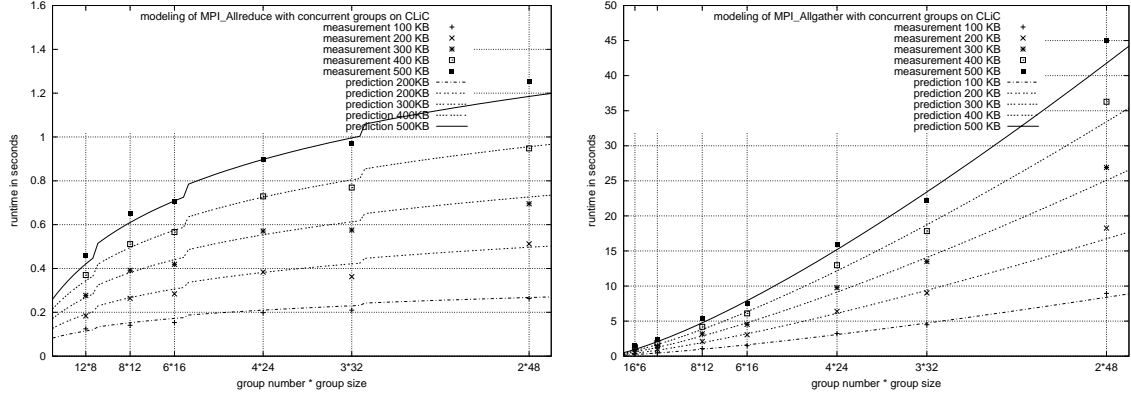
20

Figure 21: Measured and predicted runtimes for concurrent groups of *MPI_Allreduce()* (left) and *MPI_Allgather()* (right) on the CLiC.

when b denotes the original message size and p the number of involved processors. Figure 21 shows measured and prediced runtimes with fixed message sizes. The predictions fit the measured runtimes quite accurately. The deviations between measured and predicted runtimes lies below 5 % in most cases.

5   Applications and runtime tests

To investigate the efficiency improvement of the approach for entire application programs, we consider parallel implementations of the Jacobi iteration with and without optimized communication in section 5.1. In section 5.2 we consider a complex application program, the parallel Adams methods PAB and PABM to show the performance improvements by concurrent group communication.

5.1   Parallel Jacobi iteration

We consider three different ways to implement the Jacobi iteration in a data parallel way based on a row-wise and a column-wise distribution of the matrix A. For both distributions the computational work for computing the new entries of the next iteration vector $x^{(k)}$ is the same and is equally allocated to the processors. For systems of size n each processor performs $\lceil \frac{n}{p} \rceil \times n$ multiplications and about the same number of additions in each iteration. But because each processor computes different parts and each processor needs the entire new iteration vector $x^{(k)}$ in the next iteration step, different communication operations are required for the implementations. In the row-wise distribution of A each processor computes $\lceil \frac{n}{p} \rceil$ scalar products yielding $\lceil \frac{n}{p} \rceil$ components of the new iteration vector. To provide the entire vector to each processor for the next step a multi-broadcast operation (*MPI_Allgather()*) is performed. The execution time of the row-wise Jacobi iteration with p processors and a system size n can be modeled by the formula

$$T_{row}(p,n) = 2 \cdot \frac{n^2}{p} \cdot t_{op} + T_{mb}(p, \frac{n}{p}) \tag{1}$$

where t_{op} denotes the time for the execution of an arithmetic operation and T_{mb} denotes the runtime formula of the multi-broadcast operation, see Table 1.
In the column-wise distribution of A each processor computes a new vector d of size n. Addition of all those vectors gives the new iteration vector x. Since the vectors d are
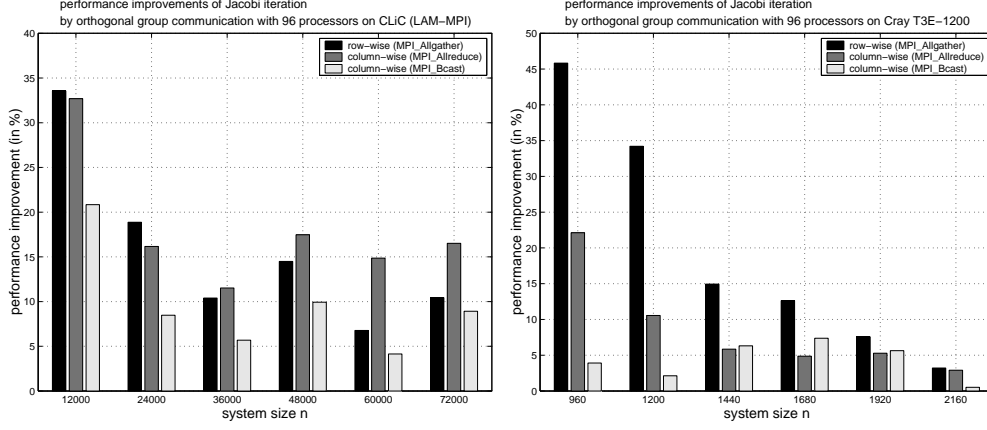
21

Figure 22: Performance improvements of the Jacobi iteration by orthogonal group communication on the CLiC (LAM-MPI) (left) and for smaller system sizes on the T3E (right).
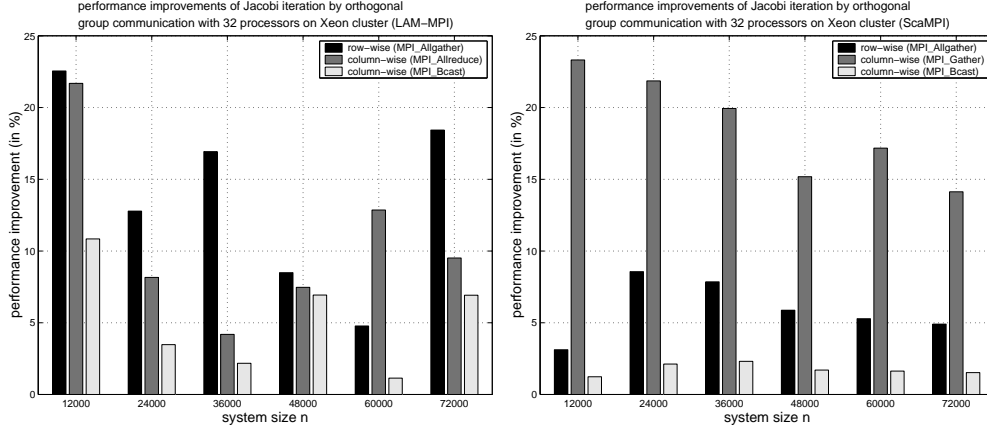

Figure 23: Performance improvements of the Jacobi iteration by orthogonal group communication on the Xeon cluster with LAM-MPI (left) and with ScaMPI (right).

located in different address spaces, collective communication is required to perform the addition. There are two possibilities.

Using an *MPI_Allreduce()* and an *MPI_Allgather()* operation results in the execution time

$$T_{col\_mb}(p, n) = 2 \cdot \frac{n^2}{p} \cdot t_{op} + T_{macc}(p, n) + T_{mb}(p, \frac{n}{p}) \qquad (2)$$

where T_{macc} denotes the time of an *MPI_Allreduce()* operation.

Using an *MPI_Reduce()* operation and then an *MPI_Bcast()* operation results in the runtime

$$T_{col\_sb}(p, n) = 2 \cdot \frac{n}{p} \cdot (n - 1) \cdot t_{op} + T_{acc}(p, n) + T_{sb}(p, n) \qquad (3)$$

where T_{acc} denotes the runtime formula of an *MPI_Reduce()* operation, see Table 1.

Optimized communication operations   Each original collective communication operation can be replaced by the optimized version using concurrent communications on disjoint subsets of processors.

Thus, when using a 2D orthogonal structure the multi-broadcast operation T_{mb} in Equation (1) and (2) can be replaced by:

$$T_{mb}(p, \frac{n}{p}) = T_{mb}(p_1, \frac{n}{p}) + T_{mb}(p_2, \frac{n \cdot p_1}{p}) \qquad (4)$$
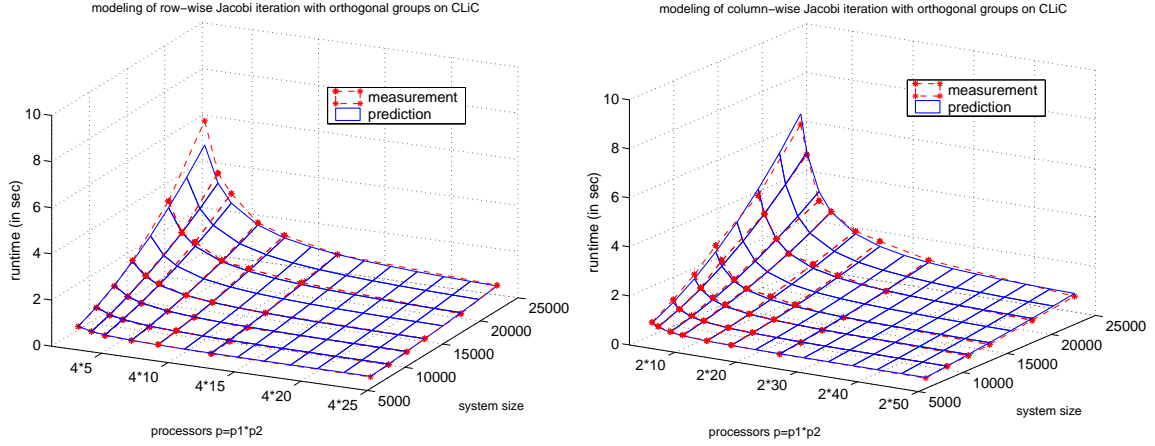
Figure 24: Modeling of row-wise (left) and column-wise (right) orthogonal realization of Jacobi iteration on the CLiC (LAM-MPI).

where $p = p_1 \cdot p_2$. The runtime formula of the single-broadcast operation in equation (3) can be replaced analogously by

$$T_{sb}(p, n) = T_{sb}(p_1, n) + T_{sb}(p_2, n). \tag{5}$$

Figure 22 shows the performance improvements obtained by a 2D orthogonal structure for the CLiC with LAM-MPI (left) and the T3E (right) for the three implementation variants described. Figure 23 shows the improvements for the dual Xeon cluster using LAM-MPI (left) and ScaMPI (right). For the parallel realization using ScaMPI a specific implementation with *MPI_Gather* and *MPI_Scatter* instead *MPI_Allreduce* is shown in Figure 23 (right). The improvements are obtained for a large range of system sizes using balanced grid layouts. Figure 24 compares the performance measurements and predictions obtained by a 2D orthogonal processor structure for the CLiC. The figure shows that the predictions fit the measurements quite good.

5.2   Parallel Adams methods PAB und PABM

Parallel Adams methods are variants of general linear methods for solving ordinary differential equations (ODEs) $\mathbf{y}'(t) = \mathbf{f}(t, \mathbf{y}(t))$ proposed in [19]; the name was chosen due to a similarity of the stage equations with classical Adams formulas. General linear methods compute several stage values $\mathbf{y}_{\kappa,i}$ in each time step κ 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 the vector $\mathbf{Y}_\kappa = (\mathbf{y}_{\kappa,1}, ..., \mathbf{y}_{\kappa,K})$; for an ODE system of size n, this vector has size $n \cdot K$.
We consider an explicit Parallel Adams-Bashforth (PAB) and an implicit parallel Adams-Moulton (PABM) method. The implicit methods use fix-point iteration with the PAB method as predictor. The resulting methods have the advantage that the computations of the parallel stages within each time step are completely independent from each other. Strong data dependencies only occur at the end of each time step.   In a data parallel implementation of the PAB method, the stage values are computed one after another with all processors available. The computation includes K function evaluations of function $\mathbf{f}$, the computation of new stage values, and K multi-broadcast operations. The resulting communication overhead within one time step is given by

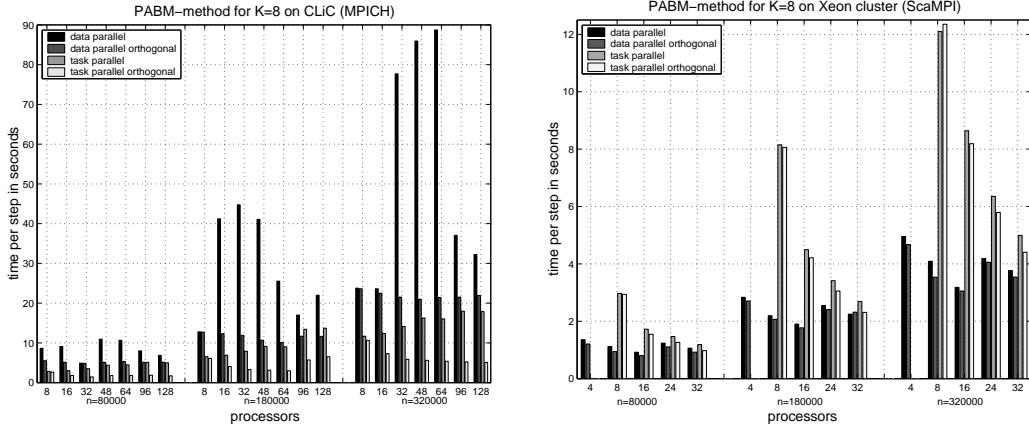$$C_{PAB}(n, p) = K \cdot T_{mb}(p, n/p).$$

Figure 25: Runtime per time step of the PABM-method for $K = 8$ by orthogonal group communication on the CLiC using MPICH (left) and on the dual Xeon cluster using ScaMPI (right).
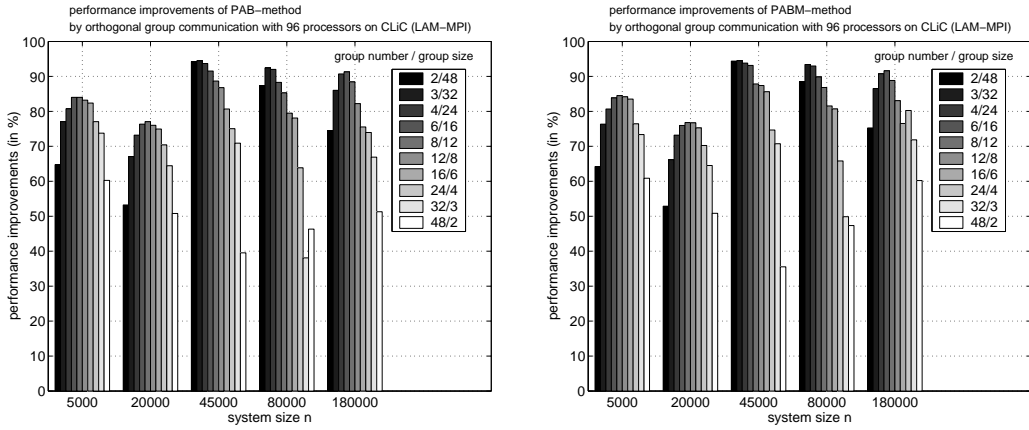

Figure 26: Performance improvements of the *data parallel* PAB-method (left) and PABM-method (right) by orthogonal group communication on the CLiC (LAM-MPI).

The computation time of one time step is given by

$$T_{PAB}(n, p) = K \cdot (n/p \cdot T_{eval}(\mathbf{f}) + (2K + 1) \cdot n/p \cdot t_{op})$$

where $T_{eval}(\mathbf{f})$ is the time for evaluating one component of $\mathbf{f}$.

The PABM method uses the PAB method as predictor and uses the PAM method for a fixed number I of iterations in the corrector step. This implementation strategy results in the following communication overhead within one time step:

$$C_{PABM}(n, p) = K \cdot I \cdot T_{mb}(p, n/p).$$

The computation time is:

$$T_{PABM}(n, p) = K \cdot I \cdot n/p \cdot T_{eval}(f) + K \cdot (2K + 1) \cdot n/p \cdot t_{op} + K \cdot I \cdot 3 \cdot n/p \cdot t_{op}.$$

Figure 25 (left) shows the runtime per step of the *PABM*-method with and without orthogonal structure on the CLiC based on the MPICH library for different implementations. As application, an ODE system has been used that results from the spatial discretization of a reaction-diffusion equation. This is a *sparse* ODE system, i.e., the evaluation time of one component of $\mathbf{f}$ is constant. We consider a data parallel, an orthogonal data parallel, a task parallel and an orthogonal task parallel implementation
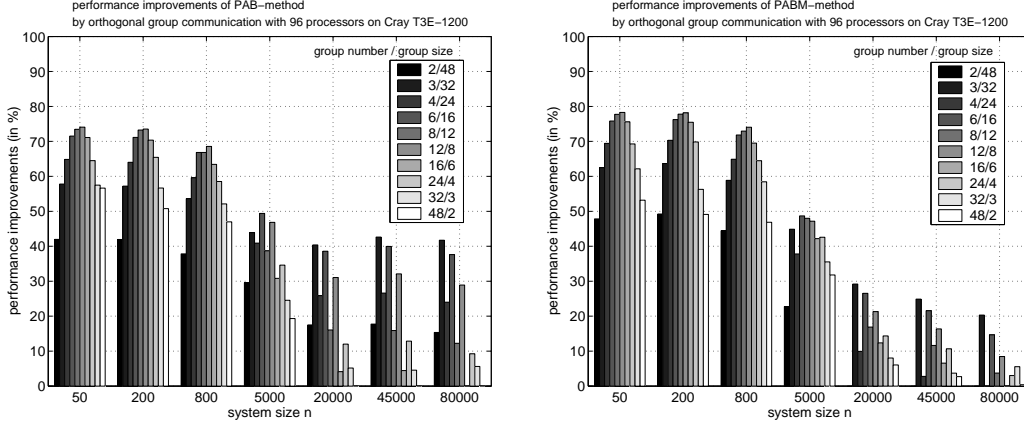
Figure 27: Performance improvements of the *data parallel* PAB-method (left) and PABM-method (right) by orthogonal group communication for small system sizes with 96 processors on the Cray T3E-1200.

for varying processor numbers and different system sizes. The orthogonal realizations are obtained by replacing the original MPI communication operation by the corresponding group-based operations. The program structure of the original data parallel implementation is not rearranged. Figure 25 shows a significant performance improvement based on the orthogonal realization of the *MPI_Allgatherv()* operation. For the orthogonal task parallel implementation a rearrangement of the program structure is required, yet this variant is the fastest implementation.

For the sparse ODE systems considered, only the orthogonal realizations lead to speedup values. For *dense* ODE systems that arise, e.g., from spectral methods, all versions lead to speedup values. But the difference between the execution time is not as significant as for the *sparse* case, since the computation time dominates the communication time.

Figures 26 and 27 show the performance improvements obtained for the PAB and PABM methods by a 2D orthogonal realization of the communication operations on the CLiC (LAM-MPI) and the T3E, respectively. The figures show the improvements of the data parallel implementation with orthogonal structure compared to the original data parallel variant for different system sizes. Figure 27 shows the PAB and PABM methods for small system sizes on the Cray T3E, confirming the performance improvements of orthogonal communication operations for small message sizes in isolation on this platform. Figure 25 (right) presents the execution times of the PABM method on the dual Xeon cluster using ScaMPI showing that the execution times can be improved by an orthogonal realization of the communication operations. Figure 28 shows the performance measurements and predictions obtained by a 2D orthogonal structure for the PAB-method (left) and the PABM-method (right) for a wide range of system sizes and participating processors on the Cray T3E-1200. The figure shows that the predictions fit the measurements quite good. The deviations between the predicted and measured runtime lie below 12% for the most cases.

6 Related Work

Related work comes from different research directions, including programming models and software support for scientific computing, parallel languages and libraries, and mixed task and data parallelism [2, 18]. Many environments for scientific computing are extensions to the HPF data parallel language. A good overview can be found in [6]. The HPF-2 standard supports the execution of tasks defined as computations on subranges of arrays
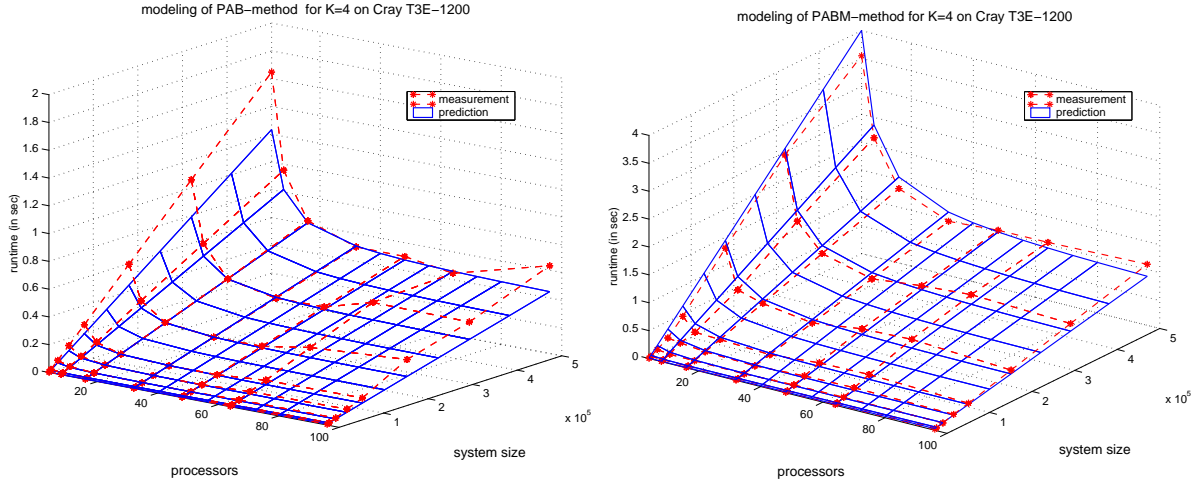
Figure 28: The measured and predicted runtime of the PAB-method (left) and PABM-method for K=4 on the Cray T3E-1200.

on processor groups also defined as parts of the processor grid. Another example is HP-Java which adopts the data distribution concepts of HPF but uses a high level SPMD programming model with a fixed number of logical control threads and includes collective communication operations encapsulated in a communication library. A language description is given in [20]. The concept of processor groups is supported in the sense that global data distributed over one process group can be defined and that the program execution control can choose one of the process groups to be active. In contrast, our approach provides processor groups which can work simultaneously and, thus, can exploit the potential parallelism of the application and the machine resources allocated more efficiently. Hence, orthogonal processor groups seem to provide the right level for applications with medium or fine-grained potential parallelism.

LPARX is a parallel programming system for the development of dynamic, nonuniform scientific computations supporting block-irregular data distributions [9]. KeLP extends LPARX to support the development of efficient programs for hierarchical parallel computers such as clusters of SMPs [1, 6]. In comparison to our approach, LPARX and KeLP are more directed towards the realization of irregular grid computations whereas our approach is based on regular grids using different partitions of the same set of processors. KeLP has been extended to KeLP-HPF which uses an SPMD program to coordinate multiple HPF tasks and, thus, combines regular data parallel computations in HPF with a coordination layer for irregular block-structured features on one grid [12]. An API for adaptive mesh algorithms based on LPARX is presented in [10].

Orthogonal processor groups have been used in [14] for the implementation of application programs by restructuring these programs as task grids such that the application consists of communication phases in the hyper-planes of the grid. This reduces the communication overhead considerably, but requires the restructuring of the application. In contrast, the approach presented in this paper restructures the internal organization of the communication operation, so that no restructuring of the application is required at all. This allows the use in arbitrary data parallel programs.

7   Conclusion

In this paper, we have shown that the execution time of MPI collective communication operations can be significantly reduced by a restructuring of the communication operation

based on a hierarchical decomposition into phases such that each phase realizes a part of the communication operation. As platforms, we have used a Cray T3E, a Beowulf cluster and a dual Xeon cluster. On all platforms, large performance improvements have been observed for optimized communication operations in isolation and also for entire application programs using those communication operations. The orthogonal realization of collective communication operations can be used to reduce scalability problems in data parallel implementations by replacing the communication operations. This avoids a restructuring the entire communication and computation structure of the application, thus reducing the programming effort for parallel machines with a large number of processors significantly.

Acknowledgment

References

[1] S.B. Baden and S.J. Fink. A Programming Methodology for Dual-Tier Multicomputers. *IEEE Transactions on Software Engineering*, 26(3):212–226, 2000.

[2] H. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3):74–84, July-August 1998.

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

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

[5] W.J. Dally and C.L. Seitz. Deadlock free message routing in multiprocessor interconnection networks. *IEEE Trans. on Computers*, 36(5):547–553, 1987.

[6] S.J. Fink. *A Programming Model for Block-Structured Scientific Calculations on SMP Clusters*. PhD thesis, University of California, San Diego, 1998.

[7] K. Hwang, Z. Xu, and M. Arakawa. Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):522–536, 1996.

[8] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proc. of IEEE/ACM SC2001*, 2001.

[9] S.R. Kohn and S.B. Baden. Irregular Coarse-Grain Data Parallelism under LPARX. *Scientific Programming*, 5:185–201, 1995.

[10] S.R. Kohn and S.B. Baden. Parallel Software Abstractions for Structured Adaptive Mesh Methods. *Journal of Parallel and Distributed Computing*, 61(6):713–736, 2001.

[11] M. Kühnemann, T. Rauber, and G. Rünger. Performance Modelling for Task-Parallel Programs. In *Communication Networks and Distributed Systems Modeling and Simulation (CNDS'02)*, pages 148–154, 2002.

[12] J. Merlin, S.Baden, St. Fink, and B. Chapman. Multiple data parallelsim with HPF and KeLP. *J. Future Generation Computer Science*, 15(3):393–405, 1999.

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

[14] T. Rauber, R. Reilein, and G. Rünger. ORT – A Communication Library for Orthogonal Processor Groups. In *Proc. of the ACM/IEEE SC 2001*. IEEE Press, 2001.

[15] T. Rauber and G. Rünger. PVM and MPI Communication Operations on the IBM SP2: Modeling and Comparison. In *Proc. 11th Symp. on High Performance Computing Systems (HPCS'97)*, 1997.

[16] T. Rauber and G. Rünger. Library Support for Hierarchical Multi-Processor Tasks. In *Proc. of the Supercomputing 2002*, Baltimore, USA, 2002.

[17] Cray Research Web Server. http://www.cray.org/.

[18] D. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, 1998.

[19] P.J. van der Houwen and E. Mesina. Parallel adams methods. *J. of Comp. and App. Mathematics*, 101:153–165, 1999.

[20] G. Zhang, B. Carpenter, G.Fox, X. Li, and Y. Wen. A high level SPMD programming model: HPspmd and its Java language binding. Technical report, NPAC at Syracuse University, 1998.