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. MPI collective communication operations to distribute or gather data are used for many parallel applications from scientific computing, but they may lead to scalability problems since their execution times increase with the number of participating processors. In this article, we show how the execution time of collective communication operations can be improved significantly by an internal restructuring based on orthogonal processor structures with two or more levels. The execution time of operations like *MPI_Bcast()* or *MPI_Allgather()* can be reduced by 40% and 70% on a dual Xeon cluster and a Beowulf cluster with single-processor nodes. But also on a Cray T3E a significant performance improvement can be obtained by a careful selection of the processor structure. The use of these optimized communication operations can reduce the execution time of data parallel implementations of complex application programs significantly without requiring any other change of the computation and communication structure. We present runtime functions for the modeling of two-phase realizations and verify that these runtime functions can predict the execution time both for communication operations in isolation and in the context of application programs.

Keywords: MPI communication operation, orthogonal processor groups, parallel application, modeling of communication time.

1. Introduction

Parallel machines with distributed address space are widely used for the implementation of applications from scientific computing and portable message-passing programs can be written using message-passing standards like MPI or PVM. For many regular applications a data-parallel execution usually leads to good performance. But for a large number of processors, data parallel implementations may also lead to scalability problems, in particular when collective communication operations are used frequently for exchanging data. The reason is that the communication overhead of collective communication operations increases with the number of participating processors and depending on the communication operation and the target machine there is a logarithmic or linear dependence of the communication overhead on the number of participating processors.

Often scalability can be improved by restructuring a parallel program so that communication occurs on smaller subsets of processors. One possibility is the exploitation of mixed task and data parallelism by partitioning the computations into multiprocessor tasks. The tasks are assigned to disjoint processor groups for execution such that each task is executed by the processors of one subgroup in a data parallel way, but different independent tasks are executed concurrently by disjoint processor groups [16]. Using multiprocessor tasks requires a specific potential of task parallelism in the application algorithm. Another approach 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 of the processor grid. To use orthogonal processor groups the application has to be reformulated such that it consists of tasks that are arranged in a two- or higher-dimensional task grid which 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 can reduce the communication overhead considerably. But this approach also 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 which avoids a program restructuring and works without any constraints about the application algorithm. Instead of rearranging the entire program structure, we use the communication structure given in the data parallel program and exchange collective

communication operations by a more efficient realization. For each collective communication operation, we introduce an internal structure based on an orthogonal arrangement of the processor set with two or more levels. Each collective communication operation is split into several phases and each phase exploits a different partitioning into processor groups. Runtime experiments show that a significant reduction in the execution time can be achieved for different target platforms and different MPI implementations. The most significant improvement are observed for *MPI_Allgather()* operations which is especially important since this operation is often used in scientific computing. Examples are iterative methods where *MPI_Allgather()* operations are used to collect data from different processors and to distribute data for the next iteration step.

Furthermore, we consider the modeling of the parallel runtime of the optimized communication operations with runtime functions. This performance 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 based on an orthogonal structuring of the processors in a two-dimensional grid. We present the runtime functions and show that they are suitable to predict the performance of the optimized communication operations in isolation and within application programs. Using these runtime functions, the programmer can get an a priori estimation of the execution time to predict various runtime effects of an orthogonal realization.

The optimized realization of collective communication operations is done on top of MPI on the application programmers level but the implementation strategy is not limited to MPI. The internal rearrangement is encapsulated in library functions which can be used on a large range of machines providing MPI. As target platforms we use a Cray T3E, a dual 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.

An advantage of the approach presented in this paper is that all programs using collective communication can benefit from the optimized communications operations and only a basic communication library like MPI needs to be available. Also no rearrangement of the program is necessary and no specific knowledge about the additional orthogonal implementation structure is needed, so that the original data parallel implementation remains unchanged. The performance prediction with runtime functions illustrates the runtime improvements but can also

be used by the application programmer to decide about the use of optimized collective communication by orthogonal structures.

The rest of the paper is organized as follows. Section 2 describes how collective communication operations can be arranged into phases performed on subsets of processors. Section 3 presents the performance modeling by runtime functions for collective communication operations based on orthogonal group structures. Runtime experiments showing the improvement in execution time obtained by the optimized communication operations are given in Section 4. 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 to standardize the programming interface for a wide variety of parallel architectures. Many implementations of the MPI standard are available, including highly-tuned proprietary versions for specific massively-parallel platforms, like the Cray T3E, as well as hardware-independent implementations such as MPICH [4] and LAM-MPI [13], which have been implemented on a large variety of machine types. Most MPI implementations are also available on clusters consisting of commodity systems connected by a high-speed network in a loosely coupled way, which are a cost effective alternative to supercomputers. As the implementations are not tuned towards a specific interconnection network, such realizations of MPI communication operations can be inefficient for some communication operations on some platforms.

In this section we describe the realization of collective communication operations in consecutive phases based on an orthogonal partitioning of the processor set. The orthogonal realizations can be used for arbitrary communication libraries that provide collective communication operations. In particular, we consider the following collective communication operations:

operation	MPI realization
single-broadcast operation	*MPI_Bcast()*
single-accumulation operation	*MPI_Reduce()*
gather operation	*MPI_Gather()*
scatter operation	*MPI_Scatter()*
multi-accumulation operation	*MPI_Allreduce()*
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 $R_1, ..., R_{p_1}$ 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 and the column groups provide a different partitioning into disjoint processor sets that are orthogonal to the row 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 this processor in common which can serve as communication processor between rows and columns. Figure 1 (left) illustrates a set of 6 processors $P_0, P_1, ..., P_5$ arranged as a $p_1 \times p_2 = 3 \times 2$ grid. For the realization in MPI, appropriate group and communicator handles are defined for the concurrent communication in the row and column groups. The overhead for creating the processor arrangement as 2D grid is quite small. Only two functions to create the groups are required and the arrangement has to be performed only once for an entire application program.

Using the 2D processor grid and the two partitionings, the communication operations can be implemented in two phases, each working on one of the partitionings of the processor grid.

Single-Broadcast In a single-broadcast operation (*MPI_Bcast()*), a root processor sends the same 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, which is also called *leader group* of the two-phase realization since this group starts the broadcasting of the message. In the second phase, each of the receiving processors in the leader group acts as a root in its corresponding row group and broadcasts the data within this row group concurrently to the other broadcast operations. The groups are also called *concurrent groups* of the two-phase communication realization. Figure 1 (right) illustrates the resulting realization *MPI_Bcast_2D()* using two communication phases for the processor grid from Figure 1 (left) with processor P_0 as root of the broadcast operation.

Single-Accumulation For a single-accumulation operation (*MPI_Reduce()*), each processor contributes an input buffer with n elements and the root processor receives a buffer with n elements containing the accumulated values that are obtained by applying a specific reduction operation, like *MPI_SUM*, component-wise to the input buffers. For the orthogonal realization *MPI_Reduce_2D()*, the local values are first reduced within the row groups by concurrent

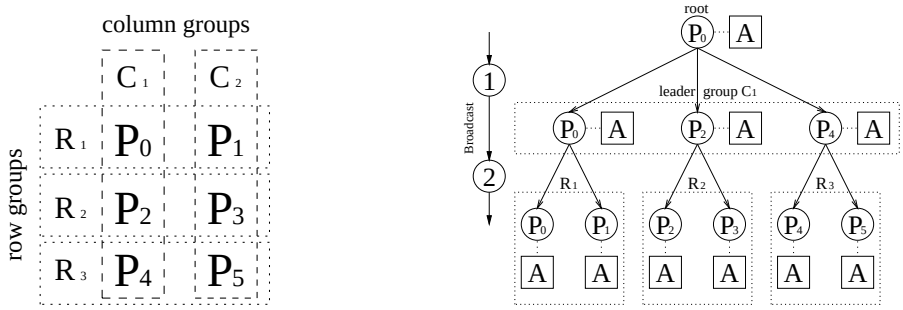

Figure 1. A set of 6 processors arranged as a two-dimensional grid with $p_1 = 3$ row groups and $p_2 = 2$ column groups in row-oriented mapping (left). An orthogonal realization of a *single-broadcast* operation on this 2D grid and root processor P_0 is illustrated to the right. In phase (1), processor P_0 broadcasts the message A within its column group C_1; this is the leader group. In phase (2), each member of the leader group broadcasts the message within its row group.

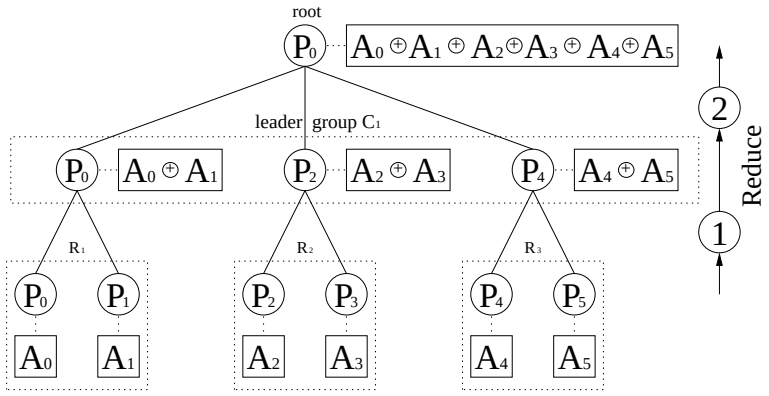

Figure 2. Illustration of an orthogonal realization of a *single-accumulation* operation with 6 processors arranged according to Figure 1 and root processor P_0. The symbol $\oplus$ represents an arbitrary reduction operation in MPI. In phase (1) processors P_0, P_2, P_4 concurrently reduce operands from their row groups. In phase (2), the leader group accumulates the results built up in the previous step.

group-based accumulation operations such that partial results are available in the column group to which the root of the global accumulation operation belongs to. In the second phase, a single-accumulation operation within the leader group accumulates all partial results at the specific root processor. Due to the reduction operation all messages in both phases have the same size, see Figure 2.

Gather For a gather operation (*MPI_Gather()*), each processor contributes a block of data and the root processor collects the blocks in rank order. For the orthogonal realization *MPI_Gather_2D()*, concur-

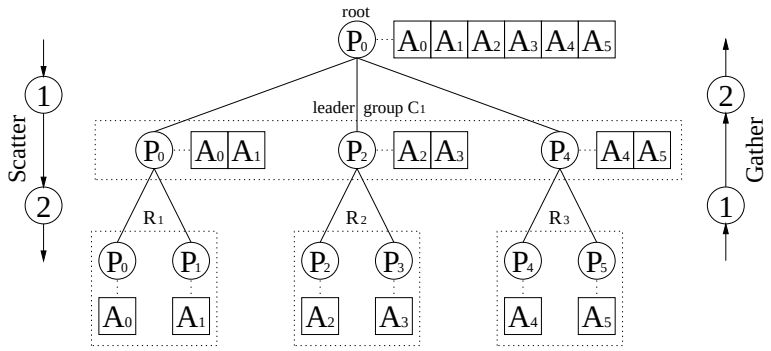

Figure 3. Illustration of an orthogonal realization of a *gather* operation (upward) and a *scatter* operation (downward) with 6 processors arranged according to Figure 1 (left) and root processor P_0. For the *gather* operation processors P_0, P_2, P_4 concurrently collect messages from processors in the corresponding row groups in phase (1). In phase (2), the leader group collects the messages built up in phase (1) at the root processor.

rent group-based gather operations first collect the data within the row groups at the unique processor belonging to the column group (leader group) to which the root of the global gather operation also belongs.

In the second phase, 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 the original message contains n elements, each processor in the leader group contributes a data block with $n \cdot p_2$ elements for the second communication phase. The order of the messages collected at the root processor is preserved because of the row-oriented mapping of processors to grids. Figure 3 (upward) illustrates the two phases for the processor grid from Figure 1 (left) where processor P_i contributes data block A_i, $i = 1, ..., 6$.

Scatter A scatter operation (*MPI_Scatter()*) is the dual operation of a gather operation, which means that the orthogonal realization *MPI_Scatter_2D()* can be obtained by reversing the order of the two phases used for a gather operation: First, the messages are scattered within the leader group such that each processor in the leader group obtains a message containing all p_2 messages for processors in the row group to which it belongs. Then the messages are scattered in the row groups by concurrent group-based scatter operation, see Figure 3 (downward).

Multi-Accumulation For a multi-accumulation operation (*MPI_Allreduce()*), each processor contributes a buffer with n elements to be accumulated component-wise according to a reduction operation and the result buffer is made available for each processor. Using a

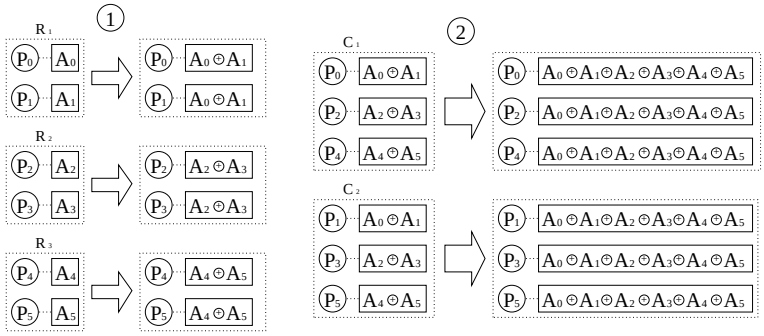

Figure 4. Illustration of an orthogonal implementation of *multi-accumulation* operation with 6 processors arranged according to Figure 1 and root processor P_0. The symbol $\oplus$ represents an arbitrary reduction operation in MPI. Phase (1) realizes concurrent multi-accumulation operations on row groups and phase (2) realizes concurrent multi-accumulation operations on column groups.

2D processor grid, the orthogonal realization *MPI_Allreduce_2D()* can be obtained by the following two steps: first, a group-based multi-accumulation operation is executed concurrently within the row groups computing partial results which are available to each processor of the corresponding row group. Second, concurrent group-based multi-accumulation operation are performed to reduce this buffer within the column groups, see Figure 4. The messages have the same size in both phases.

Multi-Broadcast For a multi-broadcast operation (*MPI_Allgather()*), each processor contributes a data block with n elements and after the operation all data blocks are available in rank order for each processor. Using a 2D processor grid, the orthogonal realization *MPI_Allgather_2D()* can be obtained 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 5. Second, concurrent group-based multi-broadcast operations are performed to distribute the data blocks within the column groups. For the second phase, each processor contributes messages with $n \cdot p_2$ elements. Again, the original rank order of data blocks is preserved.

2.2. Realization using a hierarchical processor grid

The usage of communication phases in Section 2.1 can be applied recursively to the internal communication organization of the leader group or the concurrent groups, so that the communication within the leader group or the concurrent groups is again based on an orthog-

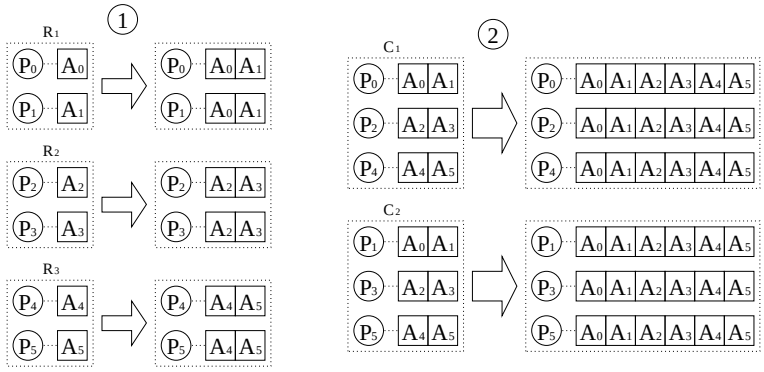

Figure 5. Illustration of an orthogonal implementation of a *multi-broadcast* operation with 6 processors arranged according to Figure 1 and root processor P_0. Phase (1) shows concurrent multi-broadcast operations on row groups and phase (2) shows concurrent multi-broadcast operations on column groups.

onal structuring of the group. Each hierarchical decomposition of a processor group leads to an additional communication phase. For three decompositions, a total number of $p = p_1 \cdot p_2 \cdot p_3$ processors is used, where p_1 denotes the size of the leader group and p_2 and p_3 denote the size of the concurrent groups in the communication phases 2 and 3. For a fixed number of processors, the hierarchical decomposition can be selected such that the best performance improvement results.

Figure 6 illustrates a *single-broadcast* operation with root processor P_0 and an additional orthogonal structure for 12 processors $P_0, P_1, ..., P_{11}$ which 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 new leader group. This results in three internal communication phases for the resulting *MPI_Bcast_3D()* operation.

3. 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 used successfully to model the execution times of monolithic collective communication operations for various communication libraries [7, 15]. Here we consider three parallel platforms: a Cray T3E-1200, a Beowulf cluster and a dual Xeon cluster. We start with a short presentation of the machines and runtime formulas for monolithic standard MPI communication operations.

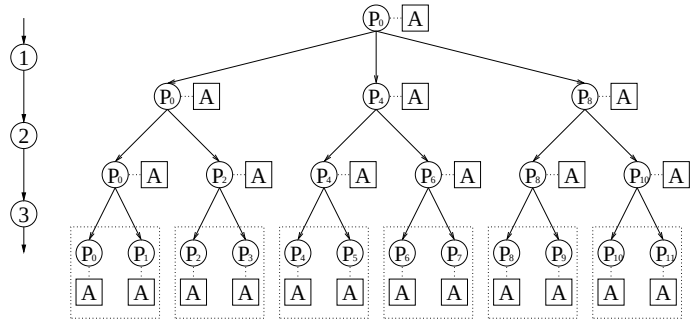

Figure 6. Illustration of the *MPI_Bcast_3D()* operation with 12 processors and root processor P_0 using three communication phases.

3.1. Parallel platforms

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 are used for the experiments.

The **dual 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 communication 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.

The **Cray T3E-1200** uses a bidirectional three-dimensional torus network to connect the nodes each containing a DEC Alpha 21164 processor with 600 MHz. The six communication links of each node are able to simultaneously support hardware transfer rates of 600 MB/s.

3.2. Performance modeling for MPI collective operations

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. Figure 7 shows the measured and predicted runtime of the *MPI_Bcast()* operation as a function of the number of processors for fixed message sizes. The figure shows that the deviations from the idealized runtime functions increase with the size of the message transmitted.

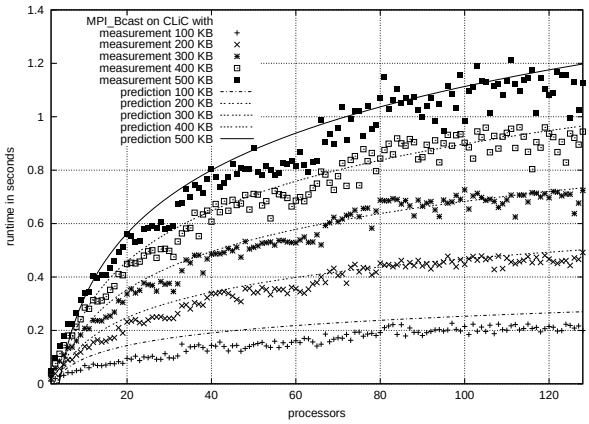

Figure 7. Performance modeling of *MPI_Bcast()* on the CLiC for fixed message sizes.

Table I presents the runtime functions of the monolithic standard MPI communication operations on the Beowulf cluster CLiC and the dual Xeon Cluster. 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 *MPI_Bcast()* operations, LAM-MPI uses two different internal realizations, one for $p \leq 4$ 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 II.

On the T3E, the runtimes of *MPI_Gather()* operations can be described by a stepwise linear function with an increasing slope. This

Table I. Runtime functions for collective communication operations on the CLiC.

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_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_Gather()	$t_{ga}(p,b) = \tau_1 + (\tau_2 + t_c \cdot p) \cdot b$
MPI_Scatter()	$t_{sc}(p,b) = \tau_1 + (\tau_2 + t_c \cdot p) \cdot b$
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 II. Coefficients of the runtime function for *MPI_Bcast()* and *MPI_Reduce()* on the CLiC (LAM-MPI), on the dual Xeon cluster (ScaMPI) and on the Cray T3E-1200.

			CLiC		dual Xeon		Cray T3E	
operation	formula	p	$\tau[\mu s]$	$t_c[\mu s]$	$\tau[\mu s]$	$t_c[\mu s]$	$\tau[\mu s]$	$t_c[\mu s]$
MPI_Bcast()	t_{sb_log}	> 1	-	-	-0.0005	0.0042	0.00065	0.0033
	t_{sb_lin}	≤ 4	-0.085	0.092	-	-	-	-
	t_{sb_log}	> 4	0.038s	0.474	-	-	-	-
MPI_Reduce()	t_{acc_log}	> 1	-	-	0.0116	0.0002	0.00048	0.0029
	t_{acc_lin}	≤ 4	-0.103	0.105	-	-	-	-
	t_{acc_log}	> 4	0.141	0.101	-	-	-	-

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 significant performance improvements by orthogonal realizations. To capture the sharp increases of the runtimes we use different runtime functions for different message sizes and different numbers of processors, see Table IV. 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 from all members of the processor group. For messages greater than 8448 KBytes a different runtime formula is used. For the

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

	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	0.000	-0.0056	0.0040
MPI_Scatter()	0.000	-0.0730	0.0897	0.000	-0.0032	0.0039

Table IV. Runtime functions of *MPI_Gather()* and *MPI_Scatter()* on the 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 V. Coefficients for runtime function of *MPI_Gather()* and *MPI_Scatter()* on Cray T3E-1200.

operation	No.	$\tau_1(V)[s]$	$\tau_2(V)[\mu s]$	$t_c(V)[\mu s]$
MPI_Gather()	1	0.000	-0.00134	0.00308
	2	-0.020	0.01573	0.00505
	3	-0.036	0.02652	0.00617
MPI_Scatter()	-	0.000	0.00024	0.00297

first formula no startup-time is necessary. The values of the coefficients are shown in Table V.

3.3. Modeling approach for two-phase realizations

For the performance modeling of orthogonal implementations of collective communication operations we model 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 introduced in Table I.

For the single-broadcast operation the communication time is modeled by adding the runtime function for the broadcast in the leader group (using the formula from Table I with $p = p_1$) and the runtime function for the concurrent broadcast in the row groups (using the formula from Table I with $p = p_2$). For a single-broadcast realized on a processor grid of size $p = p_1 \times p_2$ with p_1 row groups $R_1, ..., R_{p_1}$ and p_2 column groups $C_1, ..., C_{p_2}$ the combined runtime formula is:

$$\begin{aligned} t_{sb\_2D}(b,p) &= t_{sb\_lin}(p_1,b) + t_{sb\_lin}(p_2,b) \\ &= (\tau + t_c \cdot p_1) \cdot b + (\tau + t_c \cdot p_2) \cdot b \ , \end{aligned} \tag{1}$$

assuming a linear dependence on p. The dual operation single-accumulation is modeled by the same function. The formulas with a logarithmic dependence on p are adjusted similarly for the two-phase realizations.

For gather and scatter operations the message size changes within the phases and this is modeled in the following formulas:

$$t_{ga\_2D}(b,p) = t_{ga}(p_2,b) + t_{ga}(p_1, b \cdot p_2) \ .$$

For multi-accumulation and multi-broadcast we use:

$$t_{mb\_2D}(b,p) = t_{mb}(p_2,b) + t_{mb}(p_1, b \cdot p_2) \ . \tag{2}$$

Comparing those formulas with the times for monolithic collective communication operations with p processors shows the potential for performance improvements.

For an accurate modeling it is important to verify, whether concurrent group communication operations can interfere with each other which might lead to delayed transmission times, especially when larger message sizes are transmitted concurrently. In the following subsection we consider the performance measurements and performance predictions for several orthogonal realizations. 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. The accurate predictions for the concurrent groups and leader groups show that this modeling approach can be used for all collective 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.

3.4. Performance prediction for two-phase realizations

In the following, we compare the predicted with the measured execution times of the orthogonal realizations of the collective communication operations. On the CLiC and the Cray T3E we present the results with 48 and with 96 processors. For other numbers of processors, 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, message sizes between 10 KBytes and 500 KBytes have been used, describing the size of the data block contributed (e.g. *MPI_Gather()*) or obtained (e.g. *MPI_Scatter()*) by each participating processor. For the dual Xeon cluster we present runtime tests with 16 and with 32 processors for both communication networks, 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. We show a selection of the diagrams demonstrating the most interesting effects.

Single-Broadcast Figure 8 (left) shows the deviations between measured and predicted execution times for single-broadcast with an orthogonal realization on the Beowulf cluster CLiC. For different group

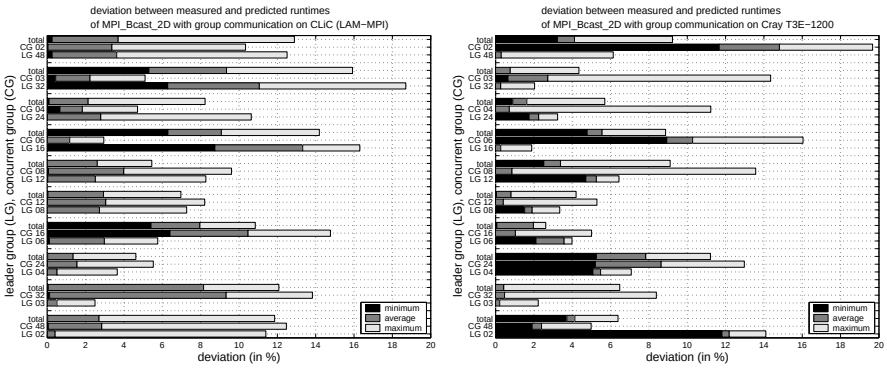

Figure 8. Deviations between measured and predicted runtimes of *MPI_Bcast_2D()* for 96 processors and varying processor grid layouts on the CLiC (left) and on the Cray T3E (right).

layouts the deviations are given 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. This is caused by the depth of the broadcast tree which remains constant for a specific interval of numbers of processors so that the communication time increases stepwise. The runtime formulas do not model these steps exactly. Figure 8 (right) shows the resulting deviations for the T3E-1200. Figure 9 (left) shows deviations between measured and predicted execution times for the dual Xeon cluster using ScaMPI.

Single-Accumulation The modeling of the *MPI_Reduce_2D()* operations is performed with an formula equivalent to formula (1). The specific values for the coefficients are shown in Table II for both cases. The communication time of a single accumulation operation increases stepwise, because the number of time steps to accumulate an array depends on the depth of the accumulation tree. A detailed analysis shows that the number of processors can be partitioned into intervals such that for all numbers of processors within an interval, accumulation 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. Figure 9 (right) shows the results for the dual Xeon cluster.

Gather operation The predictions fit the measured runtimes of the gather operation quite accurately on all three test platforms. For the CLiC, the approximations are quite accurate for the entire interval of message sizes, see Figure 10 (left). The average deviations between measured and predicted runtimes lie clearly below 3 % in most cases. On the T3E, using the runtime functions from Table IV, the deviations between measured and predicted runtimes in Figure 10 (right) are obtained. The prediction for the dual Xeon cluster using ScaMPI are given in Figure 11 (left); the coefficients for the runtime function from Table III are used.

Scatter operation The runtime prediction of the scatter operation shows the best results on all systems. On the CLiC the predictions fit the measured runtimes very accurately with an average deviations below 2 % for most processor layouts over the entire interval of all message sizes, see also Figure 12 (left). The deviations of the leader group lie below 1 % for almost all group sizes. The deviations between measurements and predictions on the Cray T3E-1200 are shown in Figure 12 (right). The deviations for the dual Xeon cluster are show in Figure 11 (right).

Multi-Accumulation and Multi-Broadcast Figure 13 shows measured and predicted runtimes for multi-accumulation and multi-broadcast with fixed message sizes on the CLiC. The predictions fit the measured runtimes quite accurately and the deviations are below 5 %. The orthogonal two-phase realization use the LAM-MPI functions *MPI_Allreduce()* and *MPI_Allgather()* for the orthogonal two-phase realization. In LAM-MPI the function *MPI_Allreduce()* is built up

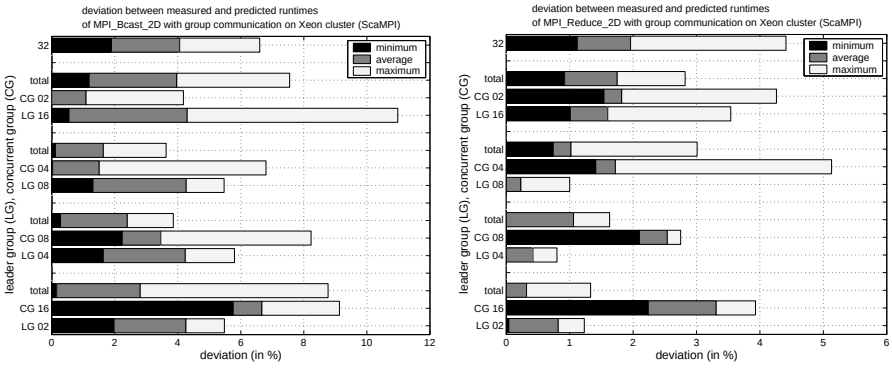

Figure 9. Deviations between measured and predicted runtimes for *MPI_Bcast_2D()* (left) and *MPI_Reduce_2D()* (right) on the dual Xeon cluster using ScaMPI.

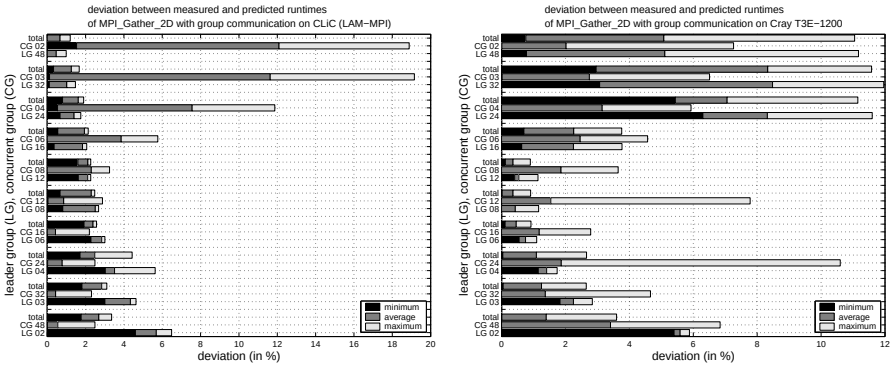

Figure 10. Deviations between measured and predicted runtimes for *MPI_Gather_2D()* on the CLiC (left) and Cray T3E-1200 (right) for 96 processors.

from an *MPI_Reduce()* and an *MPI_Bcast()* operations; the function *MPI_Allgather()* operation is composed of an *MPI_Gather()* and an *MPI_Bcast()* operation. Thus the prediction results are closely related to the results presented for those operations.

4. MPI Performance results in isolation

In this section, we present an experimental evaluation of the orthogonal realizations of the collective communication operations on the three platforms using the number of processors and grid layouts described in Section 3.4. The following figures show the minimum, average and

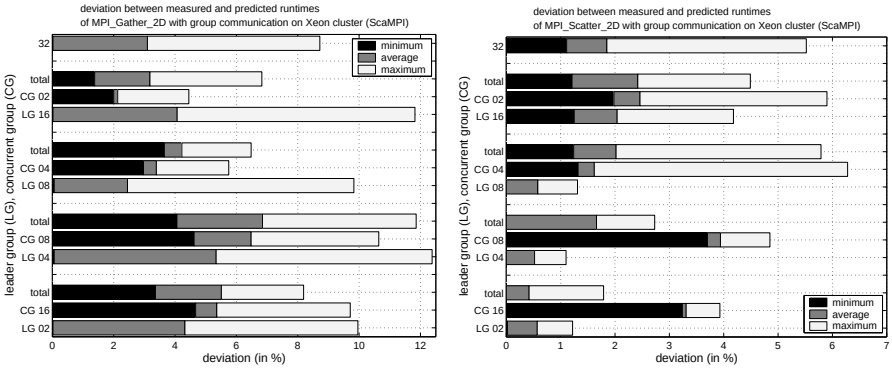

Figure 11. Deviations between measured and predicted runtimes for *MPI_Gather_2D()* (left) and *MPI_Scatter_2D()* (right) on the dual Xeon cluster using ScaMPI.

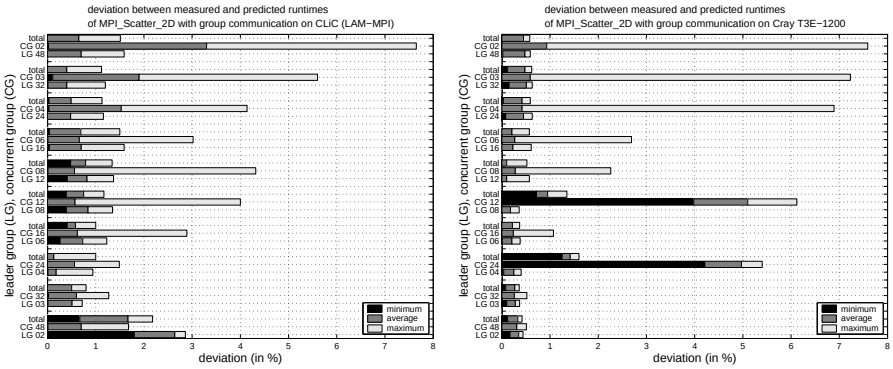

Figure 12. Deviations between measured and predicted runtimes for *MPI_Scatter_2D()* on the CLiC (left) and Cray T3E-1200 (right) for 96 processors.

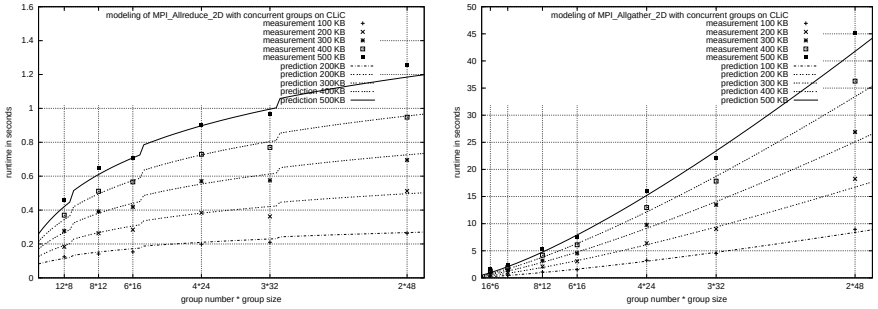

Figure 13. Measured and predicted runtimes for concurrent groups of *MPI_Allreduce_2D()* (left) and *MPI_Allgather_2D()* (right) on the CLiC.

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.

4.1. 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 *MPI_Bcast_2D()*, *MPI_Allgather_2D()* and *MPI_Allreduce_2D()* operation lead to 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_2D()* operation shows significant average improvements of more than 20% for 48 and 40% for 96 processors, respectively, using

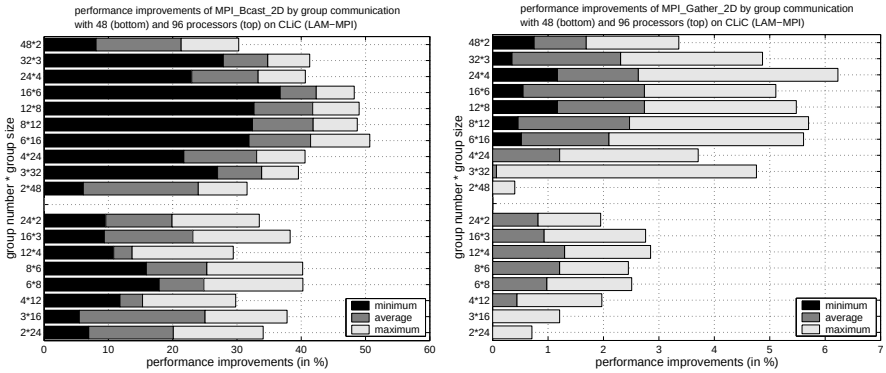

Figure 14. Performance improvements by group-based realization of *MPI_Bcast()* (left) and *MPI_Gather()* (right) with 48 and 96 processors on the CLiC (LAM-MPI).

balanced grid layouts, see Figure 14 (left). The deviations between the minimum and maximum improvements mainly occur because the figures show the improvements for a large range of message sizes between 10 KBytes and 500 KBytes. Large variations in the improvements especially occur for small message sizes. The orthogonal implementation *MPI_Allreduce_2D()* shows average improvements of more than 20% for 48 and 30% for 96 processors, respectively, again using balanced group sizes, see Figure 15 (left). The execution time of the *MPI_Allgather()* operation can be dramatically improved by an orthogonal realization, see Figure 15 (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 communication pattern. Figure 16 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()* op-

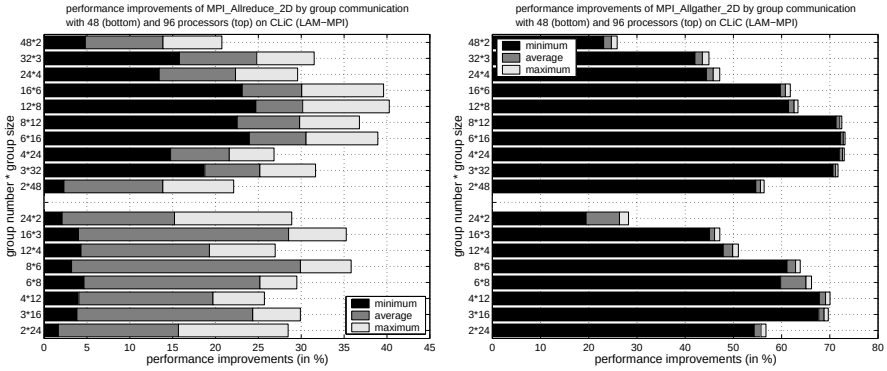

Figure 15. Performance improvements of *MPI_Allreduce_2D()* (left) and *MPI_Allgather_2D()* (right) by group communication with 48 and 96 processors on the CLiC (LAM-MPI).

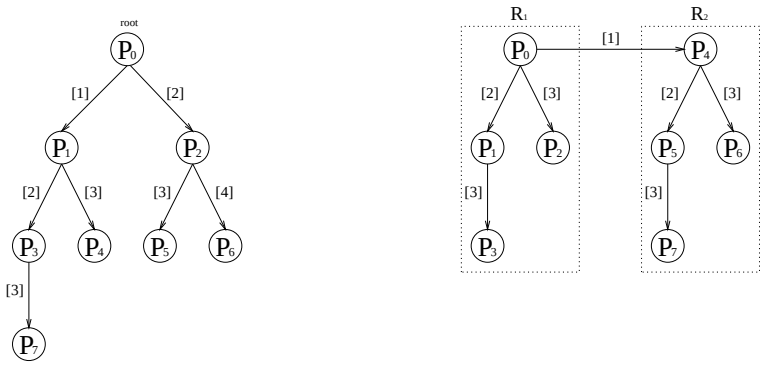

Figure 16. Illustration of an *MPI_Bcast()* operation with 8 processors $P_0, ..., P_7$ and root processor P_0 using a binary tree algorithm. The figure shows a standard implementation using in LAM-MPI (left) and an orthogonal realization with two groups R_1 and R_2 of 4 processors each (right). The processors P_0 and P_4 form the leader group. The number in the squared brackets 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.

eration. 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 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 much larger message size of $b \cdot p$ used for this *MPI_Bcast()* is also the reason for the much larger improvements of the *MPI_Allgather_2D()* operation compared to the *MPI_Allreduce_2D()* operation, where only a message of size b needs to be broadcasted.

The orthogonal implementation *MPI_Gather_2D()* 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 14 (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_2D()* and the *MPI_Reduce_2D()* operation. Only for specific message sizes, a small performance improvement can be obtained, not shown in a figure.

4.2. 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 *MPI_Gather_2D()* and *MPI_Scatter_2D()* lead to small, but persistent performance enhancements. For the *MPI_Gather_2D()* operation more than 1% for 48 and 2% improvement for 96 processors, respectively, can be obtained using balanced grid layouts, see Figure 17 (left). Similar results are shown in Figure 17 (right) for the *MPI_Scatter_2D()* 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 *MPI_Allgather_2D()* leads to large performance improvements for message sizes in the range of 32 KByte and 128 KByte, see Figure 18 (left); for larger message sizes up to 500 KByte slight performance degradation between 1 % and 2 % can be observed. The main reason for the dependence of the improvements obtained on the message size are different protocols used for short and long messages. Both protocols are realized using non-blocking *MPI_Isend()* and *MPI_Irecv()* operations. For messages up to 128

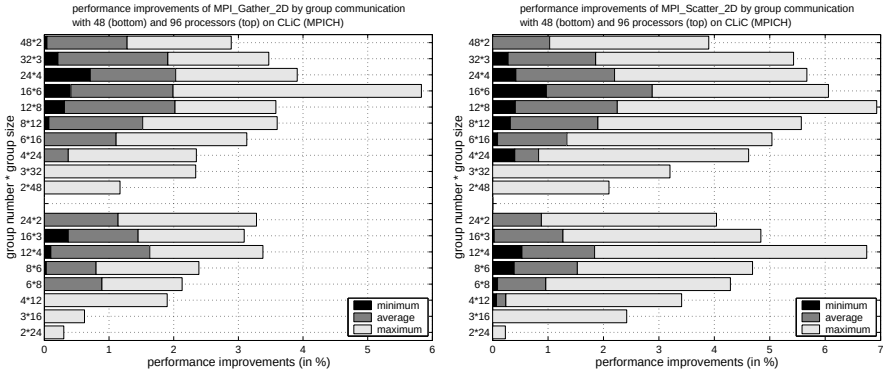

Figure 17. Performance improvements by group communication of *MPI_Gather_2D()* (left) and *MPI_Scatter_2D()* (right) with 48 and 96 processors on the CLiC (MPICH).

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 when 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 sent from the destination processor to the source processor as soon as a receive operation has been issued, so that the message can directly be copied into the user buffer. Thus, the reason for the large improvements for short messages shown in Figure 18 (left) is caused by the asynchronous standard realization of the *MPI_Allgather()* operation which 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 orthogonal *MPI_Reduce_2D()*, *MPI_Allreduce_2D()* and the *MPI_Bcast_2D()* operations which is not shown in a figure.

4.3. Orthogonal realization using LAM-MPI on the dual Xeon cluster

On the dual Xeon cluster, 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

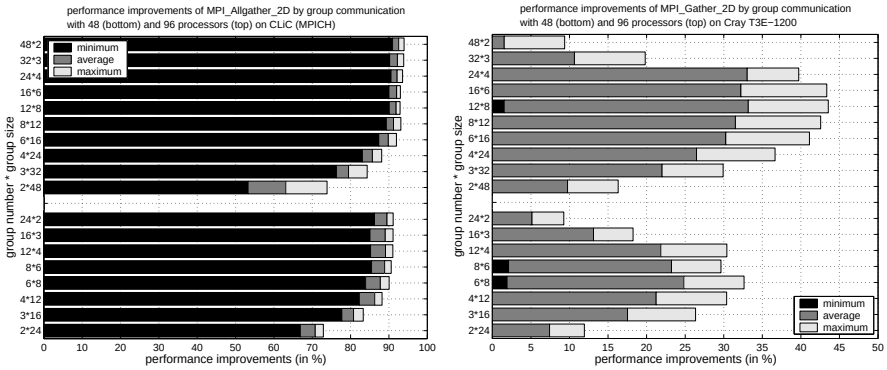

Figure 18. Performance improvements by group-based realization of *MPI_Allgather_2D()* for message sizes in the range of 32 KByte and 128 KByte on the CLiC (MPICH) (left) and *MPI_Gather_2D()* for messages sizes between 10 KByte and 500 KByte on the Cray T3E-1200 (right).

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 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 19 shows as example that for an *MPI_Bcast_2D()* (left) and an *MPI_Allgather_2D()* (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 similar performance improvements as on the CLiC can be observed for the remaining collective MPI communication operations.

4.4. 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

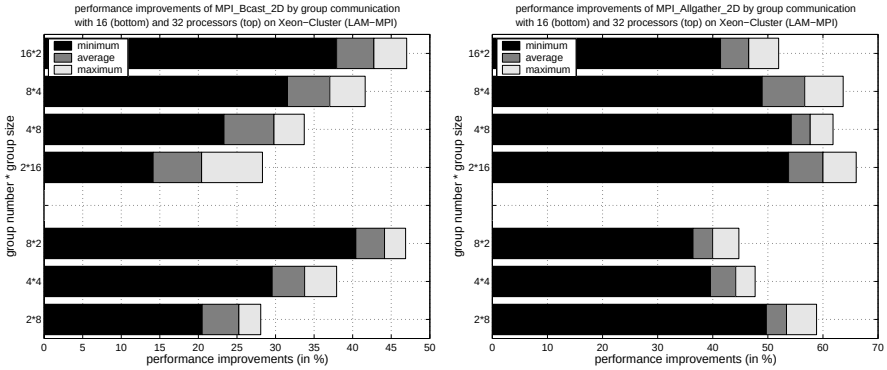

Figure 19. Performance improvements by group communication of *MPI_Bcast_2D()* (left) and *MPI_Allgather_2D()* (right) with 16 and 32 processors on the dual Xeon cluster (LAM-MPI).

network. Several collective communication operations using ScaMPI on SCI still show good performance improvements obtained by orthogonal group realization, see Figure 20 for an *MPI_Gather_2D()* (left) and *MPI_Allgather_2D()* (right) operation for smaller message sizes. For *MPI_Scatter_2D()* similar performance results like for *MPI_Gather_2D()* can be observed. For *MPI_Bcast_2D()* 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 4.3.

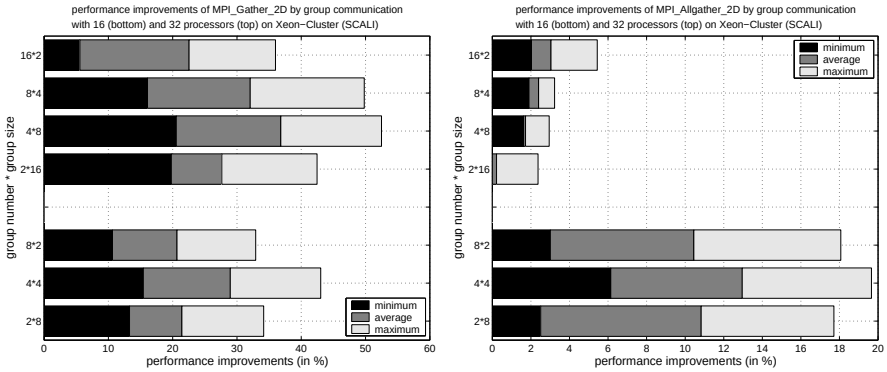

Figure 20. Performance improvements by group-based realization of *MPI_Gather_2D()* (left) for message sizes in the range of 560 Byte and 64 KByte and *MPI_Allgather_2D()* (right) for message sizes between 100 KByte and 500 KByte on the Xeon cluster (ScaMPI).

4.5. Orthogonal realization on the Cray T3E-1200

The T3E network uses a deterministic, dimension-order, k-ary n-cube wormhole routing scheme [17, 5]. Each node contains a table, stored in dedicated hardware that provides a routing tag for every destination node. The path from source to destination is determined from the current node address and the destination node address so that for the same source-destination pair all packets follow the same path. Each path traverses the network in a predefined monotonic order of the dimensions in the torus. Messages from different source-destination pairs may get involved in deadlocks while routing through the shortest paths.
Therefore, a standard MPI communication operation does not necessarily lead to automatically to an optimal execution time on the T3E. Furthermore, a rearrangement of a processor set into smaller groups of processors may prevent congestion thus leading to smaller execution times. This is shown in the following runtime tests.

The *MPI_Bcast_2D()* and *MPI_Allgather_2D()* operations lead to good performance improvements of up to 20% 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 sensitive 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 also 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 21 for the *MPI_Bcast_2D()* (left) and *MPI_Allgather_2D()* (right) operations.

For the *MPI_Gather_2D()* operation a significant performance improvement of more than 20% for 48 and 30% for 96 processors, respectively, can be obtained, see Figure 18 (right). For small message sizes, a slight performance degradation can sometimes be observed, and 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 can be described by a stepwise linear function with an increasing slope, 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

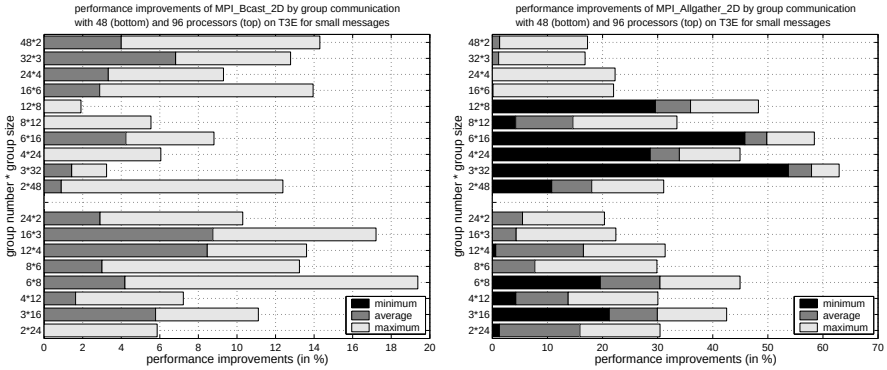

Figure 21. Performance improvements by group communication of *MPI_Bcast_2D()* (left) and *MPI_Allgather_2D()* (right) for message sizes in the range of 10 KByte and 100 KByte on the Cray T3E-1200.

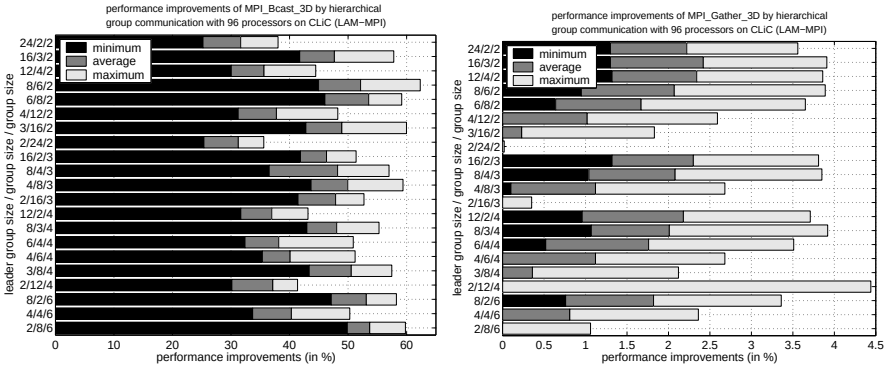

Figure 22. Overall performance improvements by hierarchical group communication of *MPI_Bcast_3D()* (left) and *MPI_Gather_3D()* (right) on the CLiC (LAM-MPI).

for the *MPI_Scatter_2D()*, *MPI_Reduce_2D()* and *MPI_Allreduce_2D()* operations. Neither for smaller nor for larger message sizes performance enhancements can be observed.

4.6. 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,

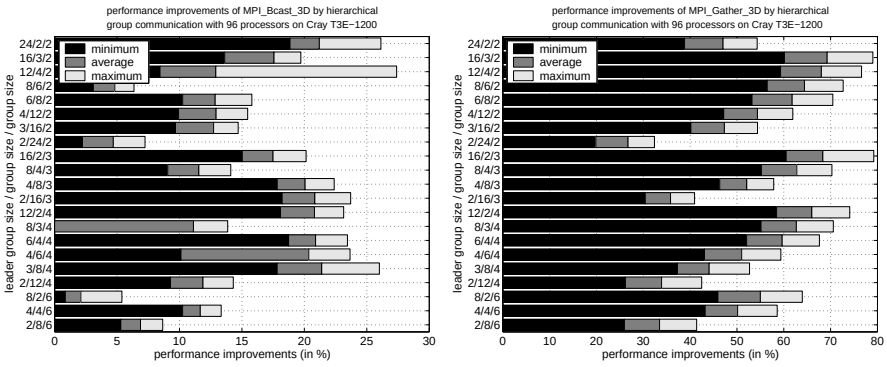

Figure 23. Overall performance improvements by hierarchical group communication of *MPI_Bcast_3D()* (left) and *MPI_Gather_3D()* (right) on the Cray T3E-1200.

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 single broadcast and gather operation leads to additional and persistent performance improvements on the CLiC and T3E. But also on the dual Xeon cluster, an additional improvement can be obtained.

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. Figure 22 (left) shows the overall improvement. The hierarchical realization for the *MPI_Gather()* operation shows no additional performance improvements compared to the two-dimensional realization, see Figure 22 (right) for the overall improvement. The resulting differences between the minimum and maximum performance improvements are larger for all message sizes than for the 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 tests with two communication phases for message sizes up to 500 KByte on the T3E, see Figure 23 (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 23 (right) for the overall improvement. The improvements vary depending on the group partitionings. For some of

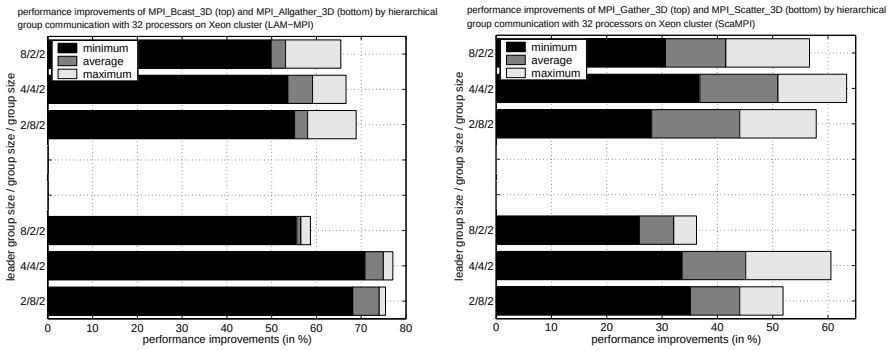

Figure 24. Overall performance improvements by hierarchical group communication of *MPI_Bcast_3D()*, *MPI_Allgather_3D()* using LAM-MPI (left) and *MPI_Gather_3D()*, *MPI_Scatter_3D()* using ScaMPI (right) on the Xeon cluster. The improvements for orthogonal realizations using ScaMPI (right Figure) are obtained for message sizes between 560 Byte and 64 KByte.

the group partitionings, additional improvements of over 60 % can be obtained.

Hierarchical realization on the dual Xeon cluster Figure 24 shows performance enhancements for four MPI communication operations realized with 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. Figure 24 shows the overall 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 4.4.

4.7. Grid Selection

For a given machine and a given MPI implementation a different layout of the processor grid leads 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. 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 communication operation. Based on the measured execution times of the communication operation, it is also possible to determine 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 implementation.

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

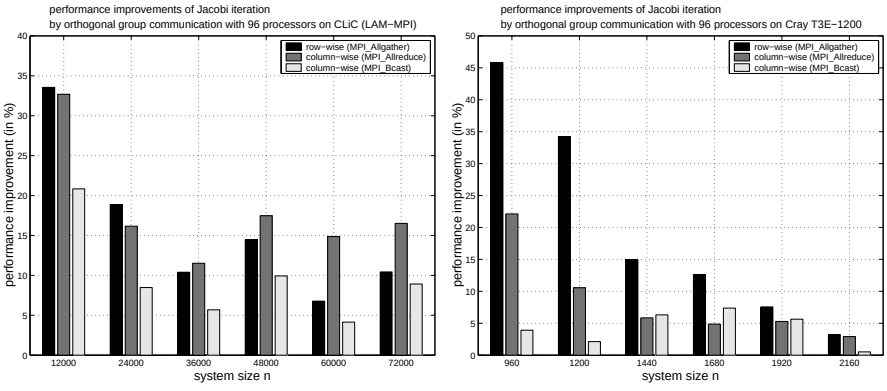

Figure 25. Performance improvements of the Jacobi iteration by orthogonal group communication on the CLiC (LAM-MPI) (left) and for smaller system sizes on the T3E (right).

$x^{(k)}$ in the next iteration step, different communication operations are required for the implementations. In the row-wise distribution of A

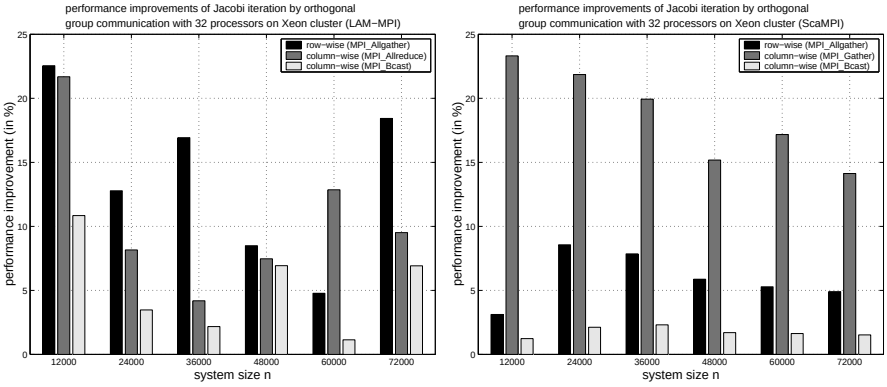

Figure 26. Performance improvements of the Jacobi iteration by orthogonal group communication on the Xeon cluster with LAM-MPI (left) and with ScaMPI (right).

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

$$T_{row}(p,n) = 2 \cdot \frac{n^2}{p} \cdot t_{op} + t_{mb}(p, \frac{n}{p}) \qquad (3)$$

where t_{op} denotes the time for the execution of an arithmetic operation and t_{mb} denotes the runtime formula of a multi-broadcast operation, see Table I.

In the column-wise distribution of A each processor computes a new vector d of size n. The addition of all those vectors gives the new iteration vector x. Since the vectors d are 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

$$T_{col\_mb}(p,n) = 2 \cdot \frac{n^2}{p} \cdot t_{op} + t_{macc}(p,n) + t_{mb}(p, \frac{n}{p}) \qquad (4)$$

where t_{macc} denotes the time of a multi-accumulation operation.

Alternatively, using an *MPI_Reduce()* operation and then an *MPI_Bcast()* operation results in the runtime

$$T_{col\_sb}(p,n) = 2 \cdot \frac{n}{p} \cdot (n-1) \cdot t_{op} + t_{acc}(p,n) + t_{sb}(p,n) \qquad (5)$$

where t_{acc} denotes the runtime formula of a single-accumulation operation, see Table I.

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 (3) and (4) are replaced by Formula (2). The runtime formula of the single-broadcast operation in Equation (5) is replaced analogously by Formula (1). Figure 25 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 26 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 26 (right). The improvements are obtained for a large range of system sizes using balanced grid layouts. Figure 27 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.

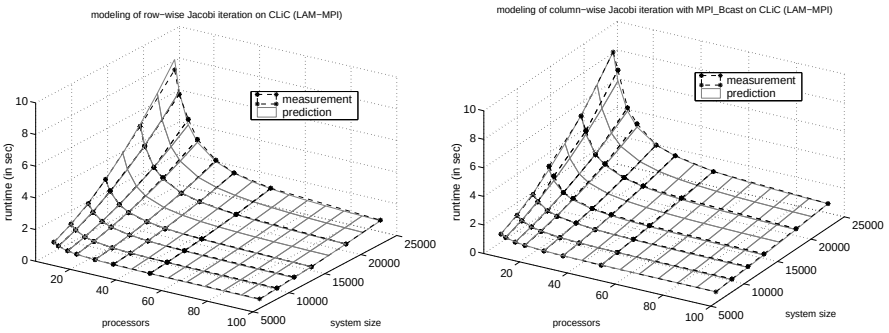

Figure 27. Modeling of row-wise (left) and column-wise (right) orthogonal realization of Jacobi iteration on the CLiC (LAM-MPI).

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 explicit Parallel Adams-Bashforth (PAB) and implicit parallel Adams-Moulton (PABM) methods. The implicit methods use fixed 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 occur only 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

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

The parallel 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 $\mathbf{f}$.

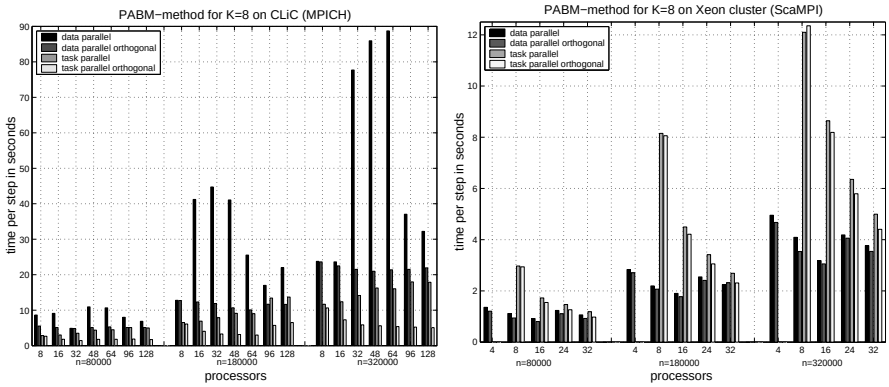

Figure 28. Runtime per time step of the PABM-method for $K = 8$ by orthogonal group communication on the CLiC using MPICH (left) and on the dual Xeon cluster using ScaMPI (right).

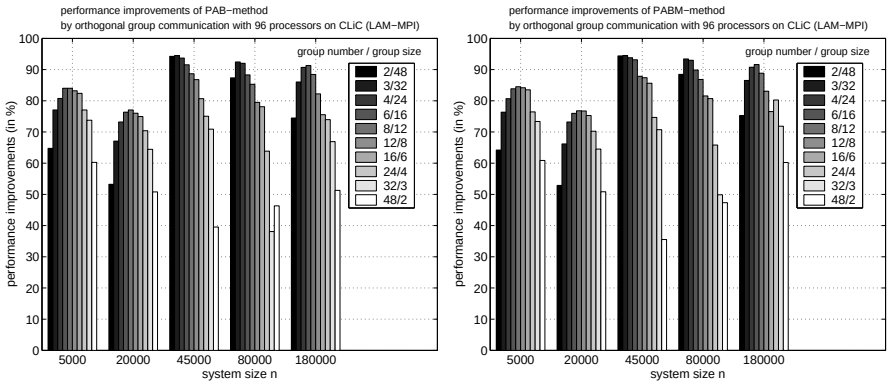

Figure 29. Performance improvements of the *data parallel* PAB-method (left) and PABM-method (right) by orthogonal group communication on the CLiC (LAM-MPI).

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 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 28 (left) shows the runtime per step of the *PABM*-method with and without optimized communication on the CLiC based on the

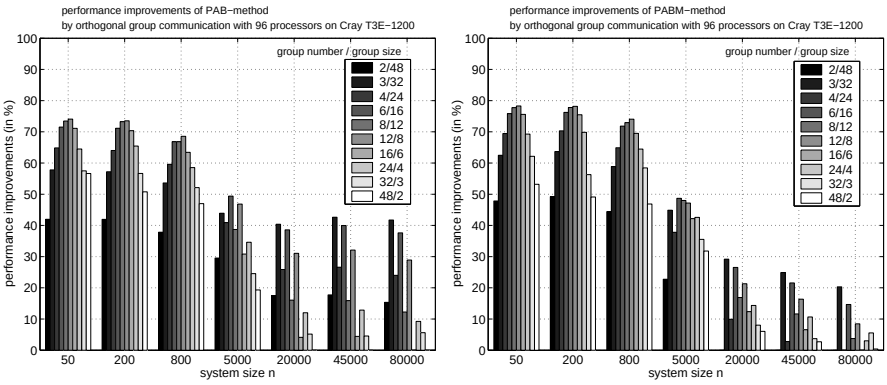

Figure 30. Performance improvements of the *data parallel* PAB-method (left) and PABM-method (right) by orthogonal group communication for small system sizes with 96 processors on the Cray T3E-1200.

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 for a varying number of processors and different system sizes. The orthogonal versions are obtained by replacing the original MPI communication operation by the corresponding group-based 2D communication operations. The program structure of the original data parallel implementation is not rearranged. Figure 28 shows a significant performance improvement based on the orthogonal realization of the *MPI_Allgatherv()* operation.

For the sparse ODE systems considered, only the programs with 2D communication operations 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 29 and 30 show the performance improvements obtained for the PAB and PABM methods by using 2D orthogonal communication operations on the CLiC (LAM-MPI) and the T3E, respectively. The figures show the improvements of the data parallel implementation with 2D orthogonal communication operations compared to the original data parallel variant for different system sizes. Figure 30 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 28

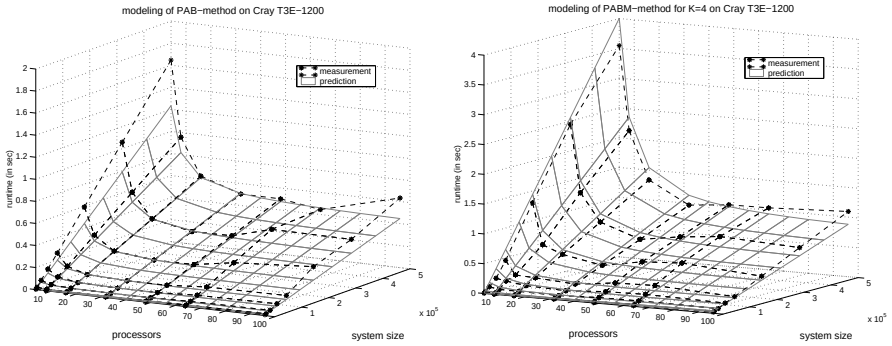

Figure 31. The measured and predicted runtime of the PAB-method (left) and PABM-method for K=4 on the Cray T3E-1200.

(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 using 2D orthogonal communication operations. Figure 31 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 on processor groups also defined as parts of the processor grid. Another example is HPJava 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 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 reduced significantly by a restructuring of the communication operations based on a hierarchical decomposition into phases such that each phase realizes a part of the communication operation on a subgroup of processors. 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. No restructuring of the communication and computation structure of the application, and thus no additional programming effort is required. The orthogonal realization of collective communication operations can

be used to reduce scalability problems in data parallel implementations by just replacing the communication operations.

Acknowledgements

We thank the NIC Jülich for providing access to a Cray T3E. The project is supported by the Deutsche Forschungsgemeinschaft.

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 implementation. 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 Transactions 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 Modeling for Task-Parallel Programs. In *Proc. of 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 parallelism 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. of the 11th Symposium 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. Messina. Parallel Adams methods. *Journal of Computational and Applied 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.