Chapter 5
Message-Passing Programming

The message-passing programming model is based on the abstraction of a parallel computer with a distributed address space where each processor has a local memory to which it has exclusive access, see Sect. 2.3.1. There is no global memory. Data exchange must be performed by message-passing: To transfer data from the local memory of one processor A to the local memory of another processor B, A must send a message containing the data to B, and B must receive the data in a buffer in its local memory. To guarantee portability of programs, no assumptions on the topology of the interconnection network is made. Instead, it is assumed that each processor can send a message to any other processor.

A message-passing program is executed by a set of processes where each process has its own local data. Usually, one process is executed on one processor or core of the execution platform. The number of processes is often fixed when starting the program. Each process can access its local data and can exchange information and data with other processes by sending and receiving messages. In principle, each of the processes could execute a different program (MPMD, *multiple program multiple data*). But to make program design easier, it is usually assumed that each of the processes executes the same program (SPMD, *single program, multiple data*), see also Sect. 2.2. In practice, this is not really a restriction, since each process can still execute different parts of the program, selected, for example, by its process rank.

The processes executing a message-passing program can exchange local data by using communication operations. These could be provided by a communication library. To activate a specific communication operation, the participating processes call the corresponding communication function provided by the library. In the simplest case, this could be a point-to-point transfer of data from a process A to a process B. In this case, A calls a send operation, and B calls a corresponding receive operation. Communication libraries often provide a large set of communication functions to support different point-to-point transfers and also global communication operations like broadcast in which more than two processes are involved, see Sect. 3.5.2 for a typical set of global communication operations.

A communication library could be vendor or hardware specific, but in most cases portable libraries are used, which define syntax and semantics of communication functions and which are supported for a large class of parallel computers. By far the

T. Rauber, G. Rünger, *Parallel Programming*,
DOI 10.1007/978-3-642-04818-0_5, © Springer-Verlag Berlin Heidelberg 2010

most popular portable communication library is MPI (*Message-Passing Interface*) [55, 56], but PVM (*Parallel Virtual Machine*) is also often used, see [63]. In this chapter, we give an introduction to MPI and show how parallel programs with MPI can be developed. The description includes point-to-point and global communication operations, but also more advanced features like process groups and communicators are covered.

5.1 Introduction to MPI

The Message-Passing Interface (MPI) is a standardization of a message-passing library interface specification. MPI defines the syntax and semantics of library routines for standard communication patterns as they have been considered in Sect. 3.5.2. Language bindings for C, C++, Fortran-77, and Fortran-95 are supported. In the following, we concentrate on the interface for C and describe the most important features. For a detailed description, we refer to the official MPI documents, see `www.mpi-forum.org`. There are two versions of the MPI standard: MPI-1 defines standard communication operations and is based on a static process model. MPI-2 extends MPI-1 and provides additional support for dynamic process management, one-sided communication, and parallel I/O. MPI is an interface specification for the syntax and semantics of communication operations, but leaves the details of the implementation open. Thus, different MPI libraries can use different implementations, possibly using specific optimizations for specific hardware platforms. For the programmer, MPI provides a standard interface, thus ensuring the portability of MPI programs. Freely available MPI libraries are MPICH (see `www-unix.mcs.anl.gov/mpi/mpich2`), LAM/MPI (see `www.lam-mpi.org`), and OpenMPI (see `www.open-mpi.org`).

In this section, we give an overview of MPI according to [55, 56]. An MPI program consists of a collection of processes that can exchange messages. For MPI-1, a static process model is used, which means that the number of processes is set when starting the MPI program and cannot be changed during program execution. Thus, MPI-1 does not support dynamic process creation during program execution. Such a feature is added by MPI-2. Normally, each processor of a parallel system executes one MPI process, and the number of MPI processes started should be adapted to the number of processors that are available. Typically, all MPI processes execute the same program in an SPMD style. In principle, each process can read and write data from/into files. For a coordinated I/O behavior, it is essential that only one specific process perform the input or output operations. To support portability, MPI programs should be written for an arbitrary number of processes. The actual number of processes used for a specific program execution is set when starting the program.

On many parallel systems, an MPI program can be started from the command line. The following two commands are common or widely used:

```
mpiexec -n 4 programname programarguments
mpirun -np 4 programname programarguments.
```

This call starts the MPI program `programname` with $p = 4$ processes. The specific command to start an MPI program on a parallel system can differ.

A significant part of the operations provided by MPI is the operations for the exchange of data between processes. In the following, we describe the most important MPI operations. For a more detailed description of all MPI operations, we refer to [135, 162, 163]. In particular the official description of the MPI standard provides many more details that cannot be covered in our short description, see [56]. Most examples given in this chapter are taken from these sources. Before describing the individual MPI operations, we first introduce some semantic terms that are used for the description of MPI operations:

- **Blocking operation**: An MPI communication operation is *blocking*, if return of control to the calling process indicates that all resources, such as buffers, specified in the call can be reused, e.g., for other operations. In particular, all state transitions initiated by a blocking operation are completed before control returns to the calling process.
- **Non-blocking operation**: An MPI communication operation is *non-blocking*, if the corresponding call may return before all effects of the operation are completed and before the resources used by the call can be reused. Thus, a call of a non-blocking operation only **starts** the operation. The operation itself is completed not before all state transitions caused are completed and the resources specified can be reused.

The terms *blocking* and *non-blocking* describe the behavior of operations from the *local* view of the executing process, without taking the effects on other processes into account. But it is also useful to consider the effect of communication operations from a *global* viewpoint. In this context, it is reasonable to distinguish between *synchronous* and *asynchronous* communications:

- **Synchronous communication**: The communication between a sending process and a receiving process is performed such that the communication operation does not complete before both processes have started their communication operation. This means in particular that the completion of a synchronous send indicates not only that the send buffer can be reused, but also that the receiving process has started the execution of the corresponding receive operation.
- **Asynchronous communication**: Using asynchronous communication, the sender can execute its communication operation without any coordination with the receiving process.

In the next section, we consider single transfer operations provided by MPI, which are also called point-to-point communication operations.

5.1.1 MPI Point-to-Point Communication

In MPI, all communication operations are executed using a **communicator**. A communicator represents a communication domain which is essentially a set of

processes that exchange messages between each other. In this section, we assume that the MPI default communicator MPI_COMM_WORLD is used for the communication. This communicator captures all processes executing a parallel program. In Sect. 5.3, the grouping of processes and the corresponding communicators are considered in more detail.

The most basic form of data exchange between processes is provided by point-to-point communication. Two processes participate in this communication operation: A sending process executes a send operation and a receiving process executes a corresponding receive operation. The send operation is *blocking* and has the syntax:

```
int MPI_Send(void *smessage,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm).
```

The parameters have the following meaning:

- smessage specifies a send buffer which contains the data elements to be sent in successive order;
- count is the number of elements to be sent from the send buffer;
- datatype is the data type of each entry of the send buffer; all entries have the same data type;
- dest specifies the rank of the target process which should receive the data; each process of a communicator has a unique rank; the ranks are numbered from 0 to the number of processes minus one;
- tag is a message tag which can be used by the receiver to distinguish different messages from the same sender;
- comm specifies the communicator used for the communication.

The size of the message in bytes can be computed by multiplying the number count of entries with the number of bytes used for type datatype. The tag parameter should be an integer value between 0 and 32,767. Larger values can be permitted by specific MPI libraries.

To receive a message, a process executes the following operation:

```
int MPI_Recv(void *rmessage,
             int count,
             MPI_Datatype datatype,
             int source,
             int tag,
             MPI_Comm comm,
             MPI_Status *status).
```

This operation is also blocking. The parameters have the following meaning:

- `rmessage` specifies the receive buffer in which the message should be stored;
- `count` is the maximum number of elements that should be received;
- `datatype` is the data type of the elements to be received;
- `source` specifies the rank of the sending process which sends the message;
- `tag` is the message tag that the message to be received must have;
- `comm` is the communicator used for the communication;
- `status` specifies a data structure which contains information about a message after the completion of the receive operation.

The predefined MPI data types and the corresponding C data types are shown in Table 5.1. There is no corresponding C data type for `MPI_PACKED` and `MPI_BYTE`. The type `MPI_BYTE` represents a single byte value. The type `MPI_PACKED` is used by special MPI pack operations.

Table 5.1 Predefined data types for MPI

MPI Datentyp	C-Datentyp
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing
MPI_BYTE	single byte value

By using `source = MPI_ANY_SOURCE`, a process can receive a message from any arbitrary process. Similarly, by using `tag = MPI_ANY_TAG`, a process can receive a message with an arbitrary tag. In both cases, the `status` data structure contains the information, from which process the message received has been sent and which tag has been used by the sender. After completion of `MPI_Recv()`, `status` contains the following information:

- `status.MPI_SOURCE` specifies the rank of the sending process;
- `status.MPI_TAG` specifies the tag of the message received;
- `status.MPI_ERROR` contains an error code.

The `status` data structure also contains information about the length of the message received. This can be obtained by calling the MPI function

```
int MPI_Get_count (MPI_Status *status,
                   MPI_Datatype datatype,
                   int *count_ptr),
```

where `status` is a pointer to the data structure `status` returned by `MPI_Recv()`. The function returns the number of elements received in the variable pointed to by `count_ptr`.

Internally a message transfer in MPI is usually performed in three steps:

1. The data elements to be sent are copied from the send buffer `smessage` specified as parameter into a system buffer of the MPI runtime system. The message is assembled by adding a header with information on the sending process, the receiving process, the tag, and the communicator used.
2. The message is sent via the network from the sending process to the receiving process.
3. At the receiving side, the data entries of the message are copied from the system buffer into the receive buffer `rmessage` specified by `MPI_Recv()`.

Both `MPI_Send()` and `MPI_Recv()` are *blocking*, *asynchronous* operations. This means that an `MPI_Recv()` operation can also be started when the corresponding `MPI_Send()` operation has not yet been started. The process executing the `MPI_Recv()` operation is blocked until the specified receive buffer contains the data elements sent. Similarly, an `MPI_Send()` operation can also be started when the corresponding `MPI_Recv()` operation has not yet been started. The process executing the `MPI_Send()` operation is blocked until the specified send buffer can be reused. The exact behavior depends on the specific MPI library used. The following two behaviors can often be observed:

- If the message is sent directly from the send buffer specified without using an internal system buffer, then the `MPI_Send()` operation is blocked until the entire message has been copied into a receive buffer at the receiving side. In particular, this requires that the receiving process has started the corresponding `MPI_Recv()` operation.
- If the message is first copied into an internal system buffer of the runtime system, the sender can continue its operations as soon as the copy operation into the system buffer is completed. Thus, the corresponding `MPI_Recv()` operation does not need to be started. This has the advantage that the sender is not blocked for a long period of time. The drawback of this version is that the system buffer needs additional memory space and that the copying into the system buffer requires additional execution time.

Example Figure 5.1 shows a first MPI program in which the process with rank 0 uses `MPI_Send()` to send a message to the process with rank 1. This process uses `MPI_Recv()` to receive a message. The MPI program shown is executed by all participating processes, i.e., each process executes the same program. But different processes may execute different program parts, e.g., depending on the values of local variables. The program defines a variable `status` of type `MPI_Status`, which is

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main (int argc, char *argv[]) {
   int my_rank, p, tag=0;
   char msg [20];
   MPI_Status status;

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size (MPI_COMM_WORLD, &p);
   if (my_rank == 0) {
      strcpy (msg, "Hello ");
      MPI_Send (msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
   }
   if (my_rank == 1)
      MPI_Recv (msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
   MPI_Finalize ();
}
```

Fig. 5.1 A first MPI program: message passing from process 0 to process 1

used for the `MPI_Recv()` operation. Any MPI program must include `<mpi.h>`. The MPI function `MPI_Init()` must be called before any other MPI function to initialize the MPI runtime system. The call `MPI_Comm_rank(MPI_COMM_WORLD, &my_rank)` returns the rank of the calling process in the communicator specified, which is `MPI_COMM_WORLD` here. The rank is returned in the variable `my_rank`. The function `MPI_Comm_size(MPI_COMM_WORLD, &p)` returns the total number of processes in the specified communicator in variable `p`. In the example program, different processes execute different parts of the program depending on their rank stored in `my_rank`: Process 0 executes a string copy and an `MPI_Send()` operation; process 1 executes a corresponding `MPI_Recv()` operation. The `MPI_Send()` operation specifies in its fourth parameter that the receiving process has rank 1. The `MPI_Recv()` operation specifies in its fourth parameter that the sending process should have rank 0. The last operation in the example program is `MPI_Finalize()` which should be the last MPI operation in any MPI program.

□

An important property to be fulfilled by any MPI library is that messages are delivered in the order in which they have been sent. If a sender sends two messages one after another to the same receiver and both messages fit to the first `MPI_Recv()` called by the receiver, the MPI runtime system ensures that the first message sent will always be received first. But this order can be disturbed if more than two processes are involved. This can be illustrated with the following program fragment:

```
/* example to demonstrate the order of receive operations */
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
   MPI_Send (sendbuf1, count, MPI_INT, 2, tag, comm);
   MPI_Send (sendbuf2, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
   MPI_Recv (recvbuf1, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send (recvbuf1, count, MPI_INT, 2, tag, comm);
}
else if (my_rank == 2) {
   MPI_Recv (recvbuf1, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
      &status);
   MPI_Recv (recvbuf2, count, MPI_INT, MPI_ANY_SOURCE, tag, comm,
      &status);
}
```

Process 0 first sends a message to process 2 and then to process 1. Process 1 receives a message from process 0 and forwards it to process 2. Process 2 receives two messages in the order in which they arrive using `MPI_ANY_SOURCE`. In this scenario, it can be expected that process 2 first receives the message that has been sent by process 0 directly to process 2, since process 0 sends this message first and since the second message sent by process 0 has to be forwarded by process 1 before arriving at process 2. But this must not necessarily be the case, since the first message sent by process 0 might be delayed because of a collision in the network whereas the second message sent by process 0 might be delivered without delay. Therefore, it can happen that process 2 first receives the message of process 0 that has been forwarded by process 1. Thus, if more than two processes are involved, there is no guaranteed delivery order. In the example, the expected order of arrival can be ensured if process 2 specifies the expected sender in the `MPI_Recv()` operation instead of `MPI_ANY_SOURCE`.

5.1.2 Deadlocks with Point-to-Point Communications

Send and receive operations must be used with care, since **deadlocks** can occur in ill-constructed programs. This can be illustrated by the following example:

```
/* program fragment which always causes a deadlock */
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
}
else if (my_rank == 1) {
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

Both processes 0 and 1 execute an MPI_Recv() operation before an MPI_Send() operation. This leads to a deadlock because of mutual waiting: For process 0, the MPI_Send() operation can be started not before the preceding MPI_Recv() operation has been completed. This is only possible when process 1 executes its MPI_Send() operation. But this cannot happen because process 1 also has to complete its preceding MPI_Recv() operation first which can happen only if process 0 executes its MPI_Send() operation. Thus, cyclic waiting occurs, and this program always leads to a deadlock.

The occurrence of a deadlock might also depend on the question whether the runtime system uses internal system buffers or not. This can be illustrated by the following example:

```
/* program fragment for which the occurrence of a deadlock
   depends on the implementation */
MPI_Comm_rank (comm, &my_rank);
if (my_rank == 0) {
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
}
```

Message transmission is performed correctly here without deadlock, if the MPI runtime system uses system buffers. In this case, the messages sent by processes 0 and 1 are first copied from the specified send buffer sendbuf into a system buffer before the actual transmission. After this copy operation, the MPI_Send() operation is completed because the send buffers can be reused. Thus, both processes 0 and 1 can execute their MPI_Recv() operation and no deadlock occurs. But a deadlock occurs, if the runtime system does not use system buffers or if the system buffers used are too small. In this case, none of the two processes can complete its MPI_Send() operation, since the corresponding MPI_Recv() cannot be executed by the other process.

A **secure implementation** which does not cause deadlocks even if no system buffers are used is the following:

```
/* program fragment that does not cause a deadlock */
MPI_Comm_rank (comm, &myrank);
if (my_rank == 0) {
   MPI_Send (sendbuf, count, MPI_INT, 1, tag, comm);
   MPI_Recv (recvbuf, count, MPI_INT, 1, tag, comm, &status);
}
else if (my_rank == 1) {
   MPI_Recv (recvbuf, count, MPI_INT, 0, tag, comm, &status);
   MPI_Send (sendbuf, count, MPI_INT, 0, tag, comm);
}
```

An MPI program is called **secure** if the correctness of the program does not depend on assumptions about specific properties of the MPI runtime system, like the existence of system buffers or the size of system buffers. Thus, secure MPI programs work correctly even if no system buffers are used. If more than two processes exchange messages such that each process sends and receives a message, the program must exactly specify in which order the send and receive operations are to be executed to avoid deadlocks. As example, we consider a program with p processes where process i sends a message to process $(i + 1)$ mod p and receives a message from process $(i - 1)$ mod p for $0 \leq i \leq p - 1$. Thus, the messages are sent in a logical ring. A secure implementation can be obtained if processes with an even rank first execute their send and then their receive operation, whereas processes with an odd rank first execute their receive and then their send operation. This leads to a communication with two phases and to the following exchange scheme for four processes:

Phase	Process 0	Process 1	Process 2	Process 3
1	`MPI_Send()` to 1	`MPI_Recv()` from 0	`MPI_Send()` to 3	`MPI_Recv()` from 2
2	`MPI_Recv()` from 3	`MPI_Send()` to 2	`MPI_Recv()` from 1	`MPI_Send()` to 0

The described execution order leads to a secure implementation also for an odd number of processes. For three processes, the following exchange scheme results:

Phase	Process 0	Process 1	Process 2
1	`MPI_Send()` to 1	`MPI_Recv()` from 0	`MPI_Send()` to 0
2	`MPI_Recv()` from 2	`MPI_Send()` to 2	-wait-
3		-wait-	`MPI_Recv()` from 1

In this scheme, some communication operations like the `MPI_Send()` operation of process 2 can be delayed because the receiver calls the corresponding `MPI_Recv()` operation at a later time. But a deadlock cannot occur.

In many situations, processes both send and receive data. MPI provides the following operations to support this behavior:

```
int MPI_Sendrecv (void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  int dest,
                  int sendtag,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  int source,
```

```
                    int recvtag,
                    MPI_Comm comm,
                    MPI_Status *status).
```

This operation is blocking and combines a send and a receive operation in one call. The parameters have the following meaning:

- `sendbuf` specifies a send buffer in which the data elements to be sent are stored;
- `sendcount` is the number of elements to be sent;
- `sendtype` is the data type of the elements in the send buffer;
- `dest` is the rank of the target process to which the data elements are sent;
- `sendtag` is the tag for the message to be sent;
- `recvbuf` is the receive buffer for the message to be received;
- `recvcount` is the maximum number of elements to be received;
- `recvtype` is the data type of elements to be received;
- `source` is the rank of the process from which the message is expected;
- `recvtag` is the expected tag of the message to be received;
- `comm` is the communicator used for the communication;
- `status` specifies the data structure to store the information on the message received.

Using `MPI_Sendrecv()`, the programmer does not need to worry about the order of the send and receive operations. The MPI runtime system guarantees deadlock freedom, also for the case that no internal system buffers are used. The parameters `sendbuf` and `recvbuf`, specifying the send and receive buffers of the executing process, must be disjoint, non-overlapping memory locations. But the buffers may have different lengths, and the entries stored may even contain elements of different data types. There is a variant of `MPI_Sendrecv()` for which the send buffer and the receive buffer are identical. This operation is also blocking and has the following syntax:

```
int MPI_Sendrecv_replace (void *buffer,
                          int count,
                          MPI_Datatype type,
                          int dest,
                          int sendtag,
                          int source,
                          int recvtag,
                          MPI_Comm comm,
                          MPI_Status *status).
```

Here, `buffer` specifies the buffer that is used as both send and receive buffer. For this function, `count` is the number of elements to be sent and to be received; these elements now should have identical type `type`.

5.1.3 Non-blocking Operations and Communication Modes

The use of blocking communication operations can lead to waiting times in which the blocked process does not perform useful work. For example, a process executing a blocking send operation must wait until the send buffer has been copied into a system buffer or even until the message has completely arrived at the receiving process if no system buffers are used. Often, it is desirable to fill the waiting times with useful operations of the waiting process, e.g., by overlapping communications and computations. This can be achieved by using *non-blocking communication operations*.

A **non-blocking send operation** initiates the sending of a message and returns control to the sending process as soon as possible. Upon return, the send operation has been started, but the send buffer specified cannot be reused safely, i.e., the transfer into an internal system buffer may still be in progress. A separate completion operation is provided to test whether the send operation has been completed locally. A non-blocking send has the advantage that control is returned as fast as possible to the calling process which can then execute other useful operations. A non-blocking send is performed by calling the following MPI function:

```
int MPI_Isend (void *buffer,
               int count,
               MPI_Datatype type,
               int dest,
               int tag,
               MPI_Comm comm,
               MPI_Request *request).
```

The parameters have the same meaning as for `MPI_Send()`. There is an additional parameter of type `MPI_Request` which denotes an opaque object that can be used for the identification of a specific communication operation. This request object is also used by the MPI runtime system to report information on the status of the communication operation.

A **non-blocking receive operation** initiates the receiving of a message and returns control to the receiving process as soon as possible. Upon return, the receive operation has been started and the runtime system has been informed that the receive buffer specified is ready to receive data. But the return of the call does not indicate that the receive buffer already contains the data, i.e., the message to be received cannot be used yet. A non-blocking receive is provided by MPI using the function

```
int MPI_Irecv (void *buffer,
               int count,
               MPI_Datatype type,
               int source,
               int tag,
               MPI_Comm comm,
               MPI_Request *request)
```

where the parameters have the same meaning as for MPI_Recv(). Again, a request object is used for the identification of the operation. Before reusing a send or receive buffer specified in a non-blocking send or receive operation, the calling process must test the completion of the operation. The request objects returned are used for the identification of the communication operations to be tested for completion. The following MPI function can be used to test for the completion of a non-blocking communication operation:

```
int MPI_Test (MPI_Request *request,
              int *flag,
              MPI_Status *status).
```

The call returns flag = 1 (true), if the communication operation identified by request has been completed. Otherwise, flag = 0 (false) is returned. If request denotes a receive operation and flag = 1 is returned, the parameter status contains information on the message received as described for MPI_Recv(). The parameter status is undefined if the specified receive operation has not yet been completed. If request denotes a send operation, all entries of status except status.MPI_ERROR are undefined. The MPI function

```
int MPI_Wait (MPI_Request *request, MPI_Status *status)
```

can be used to wait for the completion of a non-blocking communication operation. When calling this function, the calling process is blocked until the operation identified by request has been completed. For a non-blocking send operation, the send buffer can be reused after MPI_Wait() returns. Similarly for a non-blocking receive, the receive buffer contains the message after MPI_Wait() returns.

MPI also ensures for non-blocking communication operations that messages are non-overtaking. Blocking and non-blocking operations can be mixed, i.e., data sent by MPI_Isend() can be received by MPI_Recv() and data sent by MPI_Send() can be received by MPI_Irecv().

Example As example for the use of non-blocking communication operations, we consider the collection of information from different processes such that each process gets all available information [135]. We consider p processes and assume that each process has computed the same number of floating-point values. These values should be communicated such that each process gets the values of all other processes. To reach this goal, $p-1$ steps are performed and the processes are logically arranged in a ring. In the first step, each process sends its local data to its successor process in the ring. In the following steps, each process forwards the data that it has received in the previous step from its predecessor to its successor. After $p-1$ steps, each process has received all the data.

The steps to be performed are illustrated in Fig. 5.2 for four processes. For the implementation, we assume that each process provides its local data in an array x and that the entire data is collected in an array y of size p times the size of x.

Figure 5.3 shows an implementation with blocking send and receive operations. The size of the local data blocks of each process is given by parameter

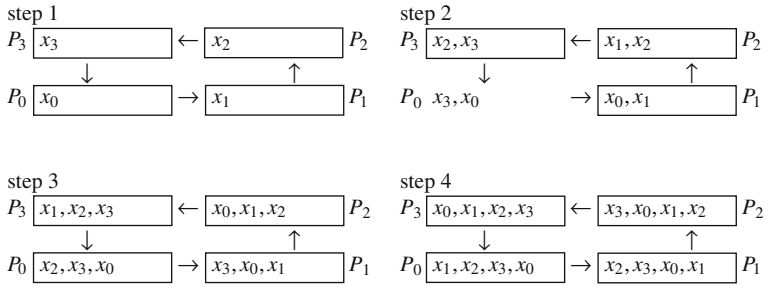

Fig. 5.2 Illustration for the collection of data in a logical ring structure for $p = 4$ processes

```
void Gather_ring (float x[], int blocksize, float y[])
{
   int i, p, my_rank, succ, pred;
   int send_offset, recv_offset;
   MPI_Status status;

   MPI_Comm_size (MPI_COMM_WORLD, &p);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   for (i=0; i<blocksize; i++)
      y[i+my_rank * blocksize] = x[i];
   succ = (my_rank+1) % p;
   pred = (my_rank-1+p) % p;
   for (i=0; i<p-1; i++) {
      send_offset = ((my_rank-i+p) % p) * blocksize;
      recv_offset = ((my_rank-i-1+p) % p) * blocksize;
      MPI_Send (y+send_offset, blocksize, MPI_FLOAT, succ, 0,
                MPI_COMM_WORLD);
      MPI_Recv (y+recv_offset, blocksize, MPI_FLOAT, pred, 0,
                MPI_COMM_WORLD, &status);
   }
}
```

Fig. 5.3 MPI program for the collection of distributed data blocks. The participating processes are logically arranged as a ring. The communication is performed with *blocking* point-to-point operations. Deadlock freedom is ensured only if the MPI runtime system uses system buffers that are large enough

blocksize. First, each process copies its local block x into the corresponding position in y and determines its predecessor process pred as well as its successors process succ in the ring. Then, a loop with $p - 1$ steps is performed. In each step, the data block received in the previous step is sent to the successor process, and a new block is received from the predecessor process and stored in the next block position to the left in y. It should be noted that this implementation requires the use of system buffers that are large enough to store the data blocks to be sent.

An implementation with non-blocking communication operations is shown in Fig. 5.4. This implementation allows an overlapping of communication with local computations. In this example, the local computations overlapped are the computations of the positions of send_offset and recv_offset of the next blocks to be sent or to be received in array y. The send and receive operations are

```
void Gather_ring_nb (float x[], int blocksize, float y[])
{
   int i, p, my_rank, succ, pred;
   int send_offset, recv_offset;
   MPI_Status status;
   MPI_Request send_request, recv_request;

   MPI_Comm_size (MPI_COMM_WORLD, &p);
   MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
   for (i=0; i<blocksize; i++)
      y[i+my_rank * blocksize] = x[i];
   succ = (my_rank+1) % p;
   pred = (my_rank-1+p) % p;
   send_offset = my_rank * blocksize;
   recv_offset = ((my_rank-1+p) % p) * blocksize;
   for (i=0; i<p-1; i++) {
      MPI_Isend (y+send_offset, blocksize, MPI_FLOAT, succ, 0,
         MPI_COMM_WORLD, &send_request);
      MPI_Irecv (y+recv_offset, blocksize, MPI_FLOAT, pred, 0,
         MPI_COMM_WORLD, &recv_request);
      send_offset = ((my_rank-i-1+p) % p) * blocksize;
      recv_offset = ((my_rank-i-2+p) % p) * blocksize;
      MPI_Wait (&send_request, &status);
      MPI_Wait (&recv_request, &status);
   }
}
```

Fig. 5.4 MPI program for the collection of distributed data blocks, see Fig. 5.3. Non-blocking communication operations are used instead of blocking operations

started with MPI_Isend() and MPI_Irecv(), respectively. After control returns from these operations, send_offset and recv_offset are re-computed and MPI_Wait() is used to wait for the completion of the send and receive operations. According to [135], the non-blocking version leads to a smaller execution time than the blocking version on an Intel Paragon and IBM SP2 machine. □

5.1.4 Communication Mode

MPI provides different **communication modes** for both blocking and non-blocking communication operations. These communication modes determine the coordination between a send and its corresponding receive operation. The following three modes are available.

5.1.4.1 Standard Mode

The communication operations described until now use the standard mode of communication. In this mode, the MPI runtime system decides whether outgoing messages are buffered in a local system buffer or not. The runtime system could, for example, decide to buffer small messages up to a predefined size, but not large messages. For the programmer, this means that he cannot rely on a buffering of messages. Hence, programs should be written in such a way that they also work if no buffering is used.

5.1.4.2 Synchronous Mode

In the standard mode, a send operation can be completed even if the corresponding receive operation has not yet been started (if system buffers are used). In contrast, in synchronous mode, a send operation will be completed not before the corresponding receive operation has been started and the receiving process has started to receive the data sent. Thus, the execution of a send and receive operation in synchronous mode leads to a form of synchronization between the sending and the receiving processes: The return of a send operation in synchronous mode indicates that the receiver has started to store the message in its local receive buffer. A blocking send operation in synchronous mode is provided in MPI by the function MPI_Ssend(), which has the same parameters as MPI_Send() with the same meaning. A non-blocking send operation in synchronous mode is provided by the MPI function MPI_Issend(), which has the same parameters as MPI_Isend() with the same meaning. Similar to a non-blocking send operation in standard mode, control is returned to the calling process as soon as possible, i.e., in synchronous mode there is **no synchronization** between MPI_Issend() and MPI_Irecv(). Instead, synchronization between sender and receiver is performed when the sender calls MPI_Wait(). When calling MPI_Wait() for a non-blocking send operation in synchronous mode, control is returned to the calling process not before the receiver has called the corresponding MPI_Recv() or MPI_Irecv() operation.

5.1.4.3 Buffered Mode

In buffered mode, the local execution and termination of a send operation is not influenced by non-local events as is the case for the synchronous mode and can be the case for standard mode if no or too small system buffers are used. Thus, when starting a send operation in buffered mode, control will be returned to the calling process even if the corresponding receive operation has not yet been started. Moreover, the send buffer can be reused immediately after control returns, even if a non-blocking send is used. If the corresponding receive operation has not yet been started, the runtime system must buffer the outgoing message. A blocking send operation in buffered mode is performed by calling the MPI function `MPI_Bsend()`, which has the same parameters as `MPI_Send()` with the same meaning. A non-blocking send operation in buffered mode is performed by calling `MPI_Ibsend()`, which has the same parameters as `MPI_Isend()`. In buffered mode, the buffer space to be used by the runtime system must be provided by the programmer. Thus, it is the programmer who is responsible that a sufficiently large buffer is available. In particular, a send operation in buffered mode may fail if the buffer provided by the programmer is too small to store the message. The buffer for the buffering of messages by the sender is provided by calling the MPI function

```
int MPI_Buffer_attach (void *buffer, int buffersize),
```

where `buffersize` is the size of the buffer `buffer` in bytes. Only one buffer can be attached by each process at a time. A buffer previously provided can be detached again by calling the function

```
int MPI_Buffer_detach (void *buffer, int *buffersize),
```

where `buffer` is the *address* of the buffer pointer used in `MPI_Buffer_attach()`; the size of the buffer detached is returned in the parameter `buffer-size`. A process calling `MPI_Buffer_detach()` is blocked until all messages that are currently stored in the buffer have been transmitted.
For receive operations, MPI provides the standard mode only.

5.2 Collective Communication Operations

A communication operation is called *collective* or *global* if all or a subset of the processes of a parallel program are involved. In Sect. 3.5.2, we have shown global communication operations which are often used. In this section, we show how these communication operations can be used in MPI. The following table gives an overview of the operations supported:

Global communication operation	MPI function
Broadcast operation	`MPI_Bcast()`
Accumulation operation	`MPI_Reduce()`
Gather operation	`MPI_Gather()`
Scatter operation	`MPI_Scatter()`
Multi-broadcast operation	`MPI_Allgather()`
Multi-accumulation operation	`MPI_Allreduce()`
Total exchange	`MPI_Alltoall()`

5.2.1 Collective Communication in MPI

5.2.1.1 Broadcast Operation

For a broadcast operation, one specific process of a group of processes sends the same data block to all other processes of the group, see Sect. 3.5.2. In MPI, a broadcast is performed by calling the following MPI function:

```
int MPI_Bcast (void *message,
               int count,
               MPI_Datatype type,
               int root,
               MPI_Comm comm),
```

where `root` denotes the process which sends the data block. This process provides the data block to be sent in parameter `message`. The other processes specify in `message` their receive buffer. The parameter `count` denotes the number of elements in the data block, `type` is the data type of the elements of the data block. `MPI_Bcast()` is a *collective* communication operation, i.e., each process of the communicator `comm` must call the `MPI_Bcast()` operation. Each process must specify the same `root` process and must use the same communicator. Similarly, the type `type` and number `count` specified by any process including the root process must be the same for all processes. Data blocks sent by `MPI_Bcast()` *cannot* be received by an `MPI_Recv()` operation.

As can be seen in the parameter list of `MPI_Bcast()`, no tag information is used as is the case for point-to-point communication operations. Thus, the receiving processes cannot distinguish between different broadcast messages based on tags.

The MPI runtime system guarantees that broadcast messages are received in the same order in which they have been sent by the root process, even if the corresponding broadcast operations are not executed at the same time. Figure 5.5 shows as example a program part in which process 0 sends two data blocks `x` and `y` by two successive broadcast operations to process 1 and process 2 [135].

Process 1 first performs local computations by `local_work()` and then stores the first broadcast message in its local variable `y`, the second one in `x`. Process 2 stores the broadcast messages in the same local variables from which they have been sent by process 0. Thus, process 1 will store the messages in other local variables as process 2. Although there is no explicit synchronization between the processes

```
if (my_rank == 0) {
   MPI_Bcast (&x, 1, MPI_INT, 0, comm);
   MPI_Bcast (&y, 1, MPI_INT, 0, comm);
   local_work ();
}
else if (my_rank == 1) {
   local_work ();
   MPI_Bcast (&y, 1, MPI_INT, 0, comm);
   MPI_Bcast (&x, 1, MPI_INT, 0, comm);
}
else if (my_rank == 2) {
   local_work ();
   MPI_Bcast (&x, 1, MPI_INT, 0, comm);
   MPI_Bcast (&y, 1, MPI_INT, 0, comm);
}
```

Fig. 5.5 Example for the receive order with several broadcast operations

executing `MPI_Bcast()`, synchronous execution semantics is used, i.e., the order of the `MPI_Bcast()` operations is such as if there were a synchronization between the executing processes.

Collective MPI communication operations are always *blocking*; no non-blocking versions are provided as is the case for point-to-point operations. The main reason for this is to avoid a large number of additional MPI functions. For the same reason, only the standard modus is supported for collective communication operations. A process participating in a collective communication operation can complete the operation and return control as soon as its local participation has been completed, no matter what the status of the other participating processes is. For the root process, this means that control can be returned as soon as the message has been copied into a system buffer and the send buffer specified as parameter can be reused. The other processes need not have received the message before the root process can continue its computations. For a receiving process, this means that control can be returned as soon as the message has been transferred into the local receive buffer, even if other receiving processes have not even started their corresponding `MPI_Bcast()` operation. Thus, the execution of a collective communication operation does not involve a synchronization of the participating processes.

5.2.1.2 Reduction Operation

An *accumulation* operation is also called *global reduction* operation. For such an operation, each participating process provides a block of data that is combined with the other blocks using a binary reduction operation. The accumulated result is collected at a root process, see also Sect. 3.5.2. In MPI, a global reduction operation is performed by letting each participating process call the function

```
int MPI_Reduce (void *sendbuf,
                void *recvbuf,
                int count,
                MPI_Datatype type,
```

```
MPI_Op op,
int root,
MPI_Comm comm),
```

where `sendbuf` is a send buffer in which each process provides its local data for the reduction. The parameter `recvbuf` specifies the receive buffer which is provided by the root process `root`. The parameter `count` specifies the number of elements provided by each process; `type` is the data type of each of these elements. The parameter `op` specifies the reduction operation to be performed for the accumulation. This must be an *associative* operation. MPI provides a number of predefined reduction operations which are also *commutative*:

Representation	Operation
`MPI_MAX`	Maximum
`MPI_MIN`	Minimum
`MPI_SUM`	Sum
`MPI_PROD`	Product
`MPI_LAND`	Logical and
`MPI_BAND`	Bit-wise and
`MPI_LOR`	Logical or
`MPI_BOR`	Bit-wise or
`MPI_LXOR`	Logical exclusive or
`MPI_BXOR`	Bit-wise exclusive or
`MPI_MAXLOC`	Maximum value and corresponding index
`MPI_MINLOC`	Minimum value and corresponding index

The predefined reduction operations `MPI_MAXLOC` and `MPI_MINLOC` can be used to determine a global maximum or minimum value and also an additional index attached to this value. This will be used in Chap. 7 in Gaussian elimination to determine a global pivot element of a row as well as the process which owns this pivot element and which is then used as the root of a broadcast operation. In this case, the additional index value is a process rank. Another use could be to determine the maximum value of a distributed array as well as the corresponding index position. In this case, the additional index value is an array index. The operation defined by `MPI_MAXLOC` is

$$(u, i) \circ_{\max} (v, j) = (w, k),$$
$$\text{where } w = \max(u, v) \text{ and } k = \begin{cases} i & \text{if } u > v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u < v \end{cases}.$$

Analogously, the operation defined by `MPI_MINLOC` is

$$(u, i) \circ_{\min} (v, j) = (w, k),$$
$$\text{where } w = \min(u, v) \text{ and } k = \begin{cases} i & \text{if } u < v \\ \min(i, j) & \text{if } u = v \\ j & \text{if } u > v \end{cases}.$$

Thus, both operations work on pairs of values, consisting of a value and an index. Therefore the data type provided as parameter of `MPI_Reduce()` must represent such a pair of values. MPI provides the following pairs of data types:

`MPI_FLOAT_INT`	`(float,int)`
`MPI_DOUBLE_INT`	`(double,int)`
`MPI_LONG_INT`	`(long,int)`
`MPI_SHORT_INT`	`(short,int)`
`MPI_LONG_DOUBLE_INT`	`(long double,int)`
`MPI_2INT`	`(int,int)`

For an `MPI_Reduce()` operation, all participating processes must specify the same values for the parameters `count`, `type`, `op`, and `root`. The send buffers `sendbuf` and the receive buffer `recvbuf` must have the same size. At the root process, they must denote disjoint memory areas. An in-place version can be activated by passing `MPI_IN_PLACE` for `sendbuf` at the root process. In this case, the input data block is taken from the `recvbuf` parameter at the root process, and the resulting accumulated value then replaces this input data block after the completion of `MPI_Reduce()`.

Example As example, we consider the use of a global reduction operation using `MPI_MAXLOC`, see Fig. 5.6. Each process has an array of 30 values of type `double`, stored in array `ain` of length 30. The program part computes the maximum value for each of the 30 array positions as well as the rank of the process that stores this

```
double ain[30], aout[30];
int ind[30];
struct {double val; int rank;} in[30], out[30];
int i, my_rank, root=0;

MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
for (i=0; i<30; i++) {
   in[i].val = ain[i];
   in[i].rank = my_rank;
}
MPI_Reduce(in,out,30,MPI_DOUBLE_INT,MPI_MAXLOC,root,MPI_COMM_WORLD);
if (my_rank == root)
   for (i=0; i<30; i++) {
      aout[i] = out[i].val;
      ind[i] = out[i].rank;
   }
```

Fig. 5.6 Example for the use of `MPI_Reduce()` using `MPI_MAXLOC` as reduction operator

maximum value. The information is collected at process 0: The maximum values are stored in array aout and the corresponding process ranks are stored in array ind. For the collection of the information based on value pairs, a data structure is defined for the elements of arrays in and out, consisting of a double and an int value. □

MPI supports the definition of user-defined reduction operations using the following MPI function:

```
int MPI_Op_create (MPI_User_function *function,
                   int commute,
                   MPI_Op *op).
```

The parameter function specifies a user-defined function which must define the following four parameters:

```
void *in, void *out, int *len, MPI_Datatype *type.
```

The user-defined function must be associative. The parameter commute specifies whether the function is also commutative (commute=1) or not (commute=0). The call of MPI_Op_create() returns a reduction operation op which can then be used as parameter of MPI_Reduce().

Example We consider the parallel computation of the scalar product of two vectors x and y of length m using p processes. Both vectors are partitioned into blocks of size local_m = m/p. Each block is stored by a separate process such that each process stores its local blocks of x and y in local vectors local_x and local_y. Thus, the process with rank my_rank stores the following parts of x and y:

```
local_x[j] = x[j + my_rank * local_m];
local_y[j] = y[j + my_rank * local_m];
```

for $0 \leq j <$ local_m.

```
int j, m, p, local_m;
float local_dot, dot;
float local_x[100], local_y[100];
MPI_Status status;

MPI_Comm_rank( MPI_COMM_WORLD, &my_rank);
MPI_Comm_size( MPI_COMM_WORLD, &p);
if (my_rank == 0) scanf("%d",&m);
local_m = m/p;
local_dot = 0.0;
for (j=0; j < local_m; j++)
    local_dot = local_dot + local_x[j] * local_y[j];
MPI_Reduce(&local_dot, &dot,1, MPI_FLOAT, MPI_SUM,0, MPI_COMM_WORLD);
```

Fig. 5.7 MPI program for the parallel computation of a scalar product

Figure 5.7 shows a program part for the computation of a scalar product. Each process executes this program part and computes a scalar product for its local blocks in `local_x` and `local_y`. The result is stored in `local_dot`. An `MPI_Reduce()` operation with reduction operation `MPI_SUM` is then used to add up the local results. The final result is collected at process 0 in variable `dot`. □

5.2.1.3 Gather Operation

For a gather operation, each process provides a block of data collected at a root process, see Sect. 3.5.2. In contrast to `MPI_Reduce()`, no reduction operation is applied. Thus, for p processes, the data block collected at the root process is p times larger than the individual blocks provided by each process. A gather operation is performed by calling the following MPI function :

```
int MPI_Gather(void *sendbuf,
               int sendcount,
               MPI_Datatype sendtype,
               void *recvbuf,
               int recvcount,
               MPI_Datatype recvtype,
               int root,
               MPI_Comm comm).
```

The parameter `sendbuf` specifies the send buffer which is provided by each participating process. Each process provides `sendcount` elements of type `sendtype`. The parameter `recvbuf` is the receive buffer that is provided by the root process. No other process must provide a receive buffer. The root process receives `recvcount` elements of type `recvtype` from each process of communicator `comm` and stores them in the order of the ranks of the processes according to `comm`. For p processes the effect of the `MPI_Gather()` call can also be achieved if each process, including the root process, calls a send operation

```
MPI_Send (sendbuf, sendcount, sendtype, root, my_rank, comm)
```

and the root process executes p receive operations

```
MPI_Recv (recvbuf+i*recvcount*extent,
          recvcount, recvtype, i, i, comm, &status),
```

where `i` enumerates all processes of `comm`. The number of bytes used for each element of the data blocks is stored in `extend` and can be determined by calling the function `MPI_Type_extent(recvtype, &extent)`. For a correct execution of `MPI_Gather()`, each process must specify the same root process `root`. Moreover, each process must specify the same element data type and the same number of elements to be sent. Figure 5.8 shows a program part in which process 0 collects 100 integer values from each process of a communicator.

```
MPI_Comm comm;
int sendbuf[100], my_rank, root = 0, gsize, *rbuf;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   rbuf = (int *) malloc (gsize*100*sizeof(int));
}
MPI_Gather(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

Fig. 5.8 Example for the application of `MPI_Gather()`

MPI provides a variant of `MPI_Gather()` for which each process can provide a *different* number of elements to be collected. The variant is `MPI_Gatherv()`, which uses the same parameters as `MPI_Gather()` with the following two changes:

- the integer parameter `recvcount` is replaced by an integer array `recvcounts` of length p where `recvcounts[i]` denotes the number of elements provided by process `i`;
- there is an additional parameter `displs` after `recvcounts`. This is also an integer array of length p and `displs[i]` specifies at which position of the receive buffer of the root process the data block of process `i` is stored. Only the root process must specify the array parameters `recvcounts` and `displs`.

The effect of an `MPI_Gatherv()` operation can also be achieved if each process executes the send operation described above and the root process executes the following p receive operations:

```
MPI_Recv(recvbuf+displs[i]*extent, recvcounts[i], recvtype, i, i,
         comm, &status).
```

For a correct execution of `MPI_Gatherv()`, the parameter `sendcount` specified by process `i` must be equal to the value of `recvcounts[i]` specified by the root process. Moreover, the send and receive types must be identical for all processes. The array parameters `recvcounts` and `displs` specified by the root process must be chosen such that no location in the receive buffer is written more than once, i.e., an overlapping of received data blocks is not allowed.

Figure 5.9 shows an example for the use of `MPI_Gatherv()` which is a generalization of the example in Fig. 5.8: Each process provides 100 integer values, but the blocks received are stored in the receive buffer in such a way that there is a free gap between neighboring blocks; the size of the gaps can be controlled by parameter `displs`. In Fig. 5.9, `stride` is used to define the size of the gap, and the gap size is set to 10. An error occurs for `stride` $<$ `100`, since this would lead to an overlapping in the receive buffer.

```
MPI_Comm comm;
int sbuf[100];
int my_rank, root = 0, gsize, *rbuf, *displs, *rcounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if  (my_rank == root) {
    MPI_Comm_size (comm, &gsize);
    rbuf = (int *) malloc(gsize*stride*sizeof(int));
    displs = (int *) malloc(gsize*sizeof(int));
    rcounts = (int *) malloc(gsize*sizeof(int));
    for (i = 0; i < gsize; i++) {
        displs[i] = i*stride;
        rcounts[i] = 100;
    }
}
MPI_Gatherv(sbuf,100,MPI_INT,rbuf,rcounts,displs,MPI_INT,root,comm);
```

Fig. 5.9 Example for the use of `MPI_Gatherv()`

5.2.1.4 Scatter Operation

For a scatter operation, a root process provides a different data block for each participating process. By executing the scatter operation, the data blocks are distributed to these processes, see Sect. 3.5.2. In MPI, a scatter operation is performed by calling

```
int MPI_Scatter (void *sendbuf,
                 int sendcount,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int recvcount,
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm),
```

where `sendbuf` is the send buffer provided by the root process `root` which contains a data block for each process of the communicator `comm`. Each data block contains `sendcount` elements of type `sendtype`. In the send buffer, the blocks are ordered in rank order of the receiving process. The data blocks are received in the receive buffer `recvbuf` provided by the corresponding process. Each participating process including the root process must provide such a receive buffer. For p processes, the effects of `MPI_Scatter()` can also be achieved by letting the root process execute p send operations

```
MPI_Send (sendbuf+i*sendcount*extent, sendcount, sendtype, i, i,
          comm)
```

for `i` $= 0, \ldots, p-1$. Each participating process executes the corresponding receive operation

```
MPI_Recv (recvbuf, recvcount, recvtype, root, my_rank, comm,
          &status).
```

For a correct execution of MPI_Scatter(), each process must specify the same root, the same data types, and the same number of elements.

Similar to MPI_Gather(), there is a generalized version MPI_Scatterv() of MPI_Scatter() for which the root process can provide data blocks of different sizes. MPI_Scatterv() uses the same parameters as MPI_Scatter() with the following two changes:

- The integer parameter sendcount is replaced by the integer array sendcounts where sendcounts[i] denotes the number of elements sent to process i for $i = 0, \ldots, p-1$.
- There is an additional parameter displs after sendcounts which is also an integer array with p entries; displs[i] specifies from which position in the send buffer of the root process the data block for process i should be taken.

The effect of an MPI_Scatterv() operation can also be achieved by point-to-point operations: The root process executes p send operations

```
MPI_Send (sendbuf+displs[i]*extent,sendcounts[i],sendtype,i,
          i,comm)
```

and each process executes the receive operation described above.

For a correct execution of MPI_Scatterv(), the entry sendcounts[i] specified by the root process for process i must be equal to the value of recvcount specified by process i. In accordance with MPI_Gatherv(), it is required that the arrays sendcounts and displs are chosen such that no entry of the send buffer is sent to more than one process. This restriction is imposed for symmetry reasons with MPI_Gatherv() although this is not essential for a correct behavior. The program in Fig. 5.10 illustrates the use of a scatter operation. Process 0 distributes

```
MPI_Comm comm;
int rbuf[100];
int my_rank, root = 0, gsize, *sbuf, *displs, *scounts, stride=110;
MPI_Comm_rank (comm, &my_rank);
if (my_rank == root) {
   MPI_Comm_size (comm, &gsize);
   sbuf = (int *) malloc(gsize*stride*sizeof(int));
   displs = (int *) malloc(gsize*sizeof(int));
   scounts = (int *) malloc(gsize*sizeof(int));
   for (i=0; i<gsize; i++) {
       displs[i] = i*stride; scounts[i]=100;
   }
}
MPI_Scatterv(sbuf,scounts,displs,MPI_INT,rbuf,100,MPI_INT,root,comm);
```

Fig. 5.10 Example for the use of an MPI_Scatterv() operation

100 integer values to each other process such that there is a gap of 10 elements between neighboring send blocks.

5.2.1.5 Multi-broadcast Operation

For a multi-broadcast operation, each participating process contributes a block of data which could, for example, be a partial result from a local computation. By executing the multi-broadcast operation, all blocks will be provided to all processes. There is no distinguished root process, since each process obtains all blocks provided. In MPI, a multi-broadcast operation is performed by calling the function

```
int MPI_Allgather (void *sendbuf,
                   int sendcount,
                   MPI_Datatype sendtype,
                   void *recvbuf,
                   int recvcount,
                   MPI_Datatype recvtype,
                   MPI_Comm comm),
```

where `sendbuf` is the send buffer provided by each process containing the block of data. The send buffer contains `sendcount` elements of type `sendtype`. Each process also provides a receive buffer `recvbuf` in which all received data blocks are collected in the order of the ranks of the sending processes. The values of the parameters `sendcount` and `sendtype` must be the same as the values of `recvcount` and `recvtype`. In the following example, each process contributes a send buffer with 100 integer values which are collected by a multi-broadcast operation at each process:

```
int sbuf[100], gsize, *rbuf;
MPI_Comm_size (comm, &gsize);
rbuf = (int*) malloc (gsize*100*sizeof(int));
MPI_Allgather (sbuf, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

For an `MPI_Allgather()` operation, each process must contribute a data block of the same size. There is a vector version of `MPI_Allgather()` which allows each process to contribute a data block of a different size. This vector version is obtained by a similar generalization as `MPI_Gatherv()` and is performed by calling the following function:

```
int MPI_Allgatherv (void *sendbuf,
                    int sendcount,
                    MPI_Datatype sendtype,
                    void *recvbuf,
                    int *recvcounts,
                    int *displs,
                    MPI_Datatype recvtype,
                    MPI_Comm comm).
```

The parameters have the same meaning as for `MPI_Gatherv()`.

5.2.1.6 Multi-accumulation Operation

For a multi-accumulation operation, each participating process performs a separate single-accumulation operation for which each process provides a different block of data, see Sect. 3.5.2. MPI provides a version of multi-accumulation with a restricted functionality: Each process provides the same data block for each single-accumulation operation. This can be illustrated by the following diagram:

$$\begin{array}{ll} P_0 : x_0 & P_0 : x_0 + x_1 + \cdots + x_{p-1} \\ P_1 : x_1 & P_1 : x_0 + x_1 + \cdots + x_{p-1} \\ \vdots \quad \overset{MPI-accumulation(+)}{\Longrightarrow} & \vdots \\ P_{p-1} : x_n & P_{p-1} : x_0 + x_1 + \cdots + x_{p-1} \end{array}$$

In contrast to the general version described in Sect. 3.5.2, each of the processes $P_0, \ldots, P_{p-1}$ only provides one data block for $k = 0, \ldots, p-1$, expressed as $P_k : x_k$. After the operation, each process has accumulated the *same* result block, represented by $P_k : x_0 + x_1 + \cdots + x_{p-1}$. Thus, a multi-accumulation operation in MPI has the same effect as a single-accumulation operation followed by a single-broadcast operation which distributes the accumulated data block to all processes. The MPI operation provided has the following syntax:

```
int MPI_Allreduce (void *sendbuf,
                   void *recvbuf,
                   int count,
                   MPI_Datatype type,
                   MPI_Op op,
                   MPI_Comm comm),
```

where `sendbuf` is the send buffer in which each process provides its local data block. The parameter `recvbuf` specifies the receive buffer in which each process of the communicator `comm` collects the accumulated result. Both buffers contain `count` elements of type `type`. The reduction operation `op` is used. Each process must specify the same size and type for the data block.

Example We consider the use of a multi-accumulation operation for the parallel computation of a matrix–vector multiplication $c = A \cdot b$ of an $n \times m$ matrix A with an m-dimensional vector b. The result is stored in the n-dimensional vector c. We assume that A is distributed in a column-oriented blockwise way such that each of the p processes stores `local_m = m/p` contiguous columns of A in its local memory, see also Sect. 3.4 on data distributions. Correspondingly, vector b is distributed in a blockwise way among the processes. The matrix–vector multiplication is performed in parallel as described in Sect. 3.6, see also Fig. 3.13. Figure 5.11 shows an outline of an MPI implementation. The blocks of columns stored by each process are stored in the two-dimensional array `a` which contains `n` rows and `local_m` columns. Each process stores its local columns consecutively in this array. The one-dimensional array `local_b` contains for each process its block

Fig. 5.11 MPI program piece to compute a matrix–vector multiplication with a column-blockwise distribution of the matrix using an `MPI_Allreduce()` operation

```
int m, local_m, n, p;
float a[MAX_N][MAX_LOC_M], local_b[MAX_LOC_M];
float c[MAX_N], sum[MAX_N];
local_m = m/p;
for (i=0; i<n; i++) {
   sum[i] = 0;
   for (j=0; j<local_m; j++)
      sum[i] = sum[i] + a[i][j]*local_b[j];
}
MPI_Allreduce (sum, c, n, MPI_FLOAT, MPI_SUM, comm);
```

of b of length `local_m`. Each process computes n partial scalar products for its local block of columns using partial vectors of length `local_m`. The global accumulation to the final result is performed with an `MPI_Allreduce()` operation, providing the result to all processes in a replicated way. □

5.2.1.7 Total Exchange

For a total exchange operation, each process provides a different block of data for each other process, see Sect. 3.5.2. The operation has the same effect as if each process performs a separate scatter operation (sender view) or as if each process performs a separate gather operation (receiver view). In MPI, a total exchange is performed by calling the function

```
int MPI_Alltoall (void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  MPI_Comm comm),
```

where `sendbuf` is the send buffer in which each process provides for each process (including itself) a block of data with `sendcount` elements of type `sendtype`. The blocks are arranged in rank order of the target process. Each process also provides a receive buffer `recvbuf` in which the data blocks received from the other processes are stored. Again, the blocks received are stored in rank order of the sending processes. For p processes, the effect of a total exchange can also be achieved if each of the p processes executes p send operations

```
MPI_Send (sendbuf+i*sendcount*extent, sendcount, sendtype,
          i, my_rank, comm)
```

as well as p receive operations

```
MPI_Recv (recvbuf+i*recvcount*extent, recvcount, recvtype,
          i, i, comm, &status),
```

where i is the rank of one of the p processes and therefore lies between 0 and $p-1$.

For a correct execution, each participating process must provide for each other process data blocks of the same size and must also receive from each other process data blocks of the same size. Thus, all processes must specify the same values for sendcount and recvcount. Similarly, sendtype and recvtype must be the same for all processes. If data blocks of different sizes should be exchanged, the vector version must be used. This has the following syntax:

```
int MPI_Alltoallv (void *sendbuf,
                   int *scounts,
                   int *sdispls,
                   MPI_Datatype sendtype,
                   void *recvbuf,
                   int *rcounts,
                   int *rdispls,
                   MPI_Datatype recvtype,
                   MPI_Comm comm).
```

For each process i, the entry scounts[j] specifies how many elements of type sendtype process i sends to process j. The entry sdispls[j] specifies the start position of the data block for process j in the send buffer of process i. The entry rcounts[j] at process i specifies how many elements of type recvtype process i receives from process j. The entry rdispls[j] at process i specifies at which position in the receive buffer of process i the data block from process j is stored.

For a correct execution of MPI_Alltoallv(), scounts[j] at process i must have the same value as rcounts[i] at process j. For p processes, the effect of Alltoallv() can also be achieved, if each of the processes executes p send operations

```
MPI_Send (sendbuf+sdispls[i]*sextent, scounts[i],
          sendtype, i, my_rank, comm)
```

and p receive operations

```
MPI_Recv (recvbuf+rdispls[i]*rextent, rcounts[i],
          recvtype, i, i, comm, &status),
```

where i is the rank of one of the p processes and therefore lies between 0 and $p-1$.

5.2.2 Deadlocks with Collective Communication

Similar to single transfer operations, different behavior can be observed for collective communication operations, depending on the use of internal system buffers by the MPI implementation. A careless use of collective communication operations may lead to **deadlocks**, see also Sect. 3.7.4 (p. 140) for the occurrence of deadlocks with single transfer operations. This can be illustrated for `MPI_Bcast()` operations: We consider two MPI processes which execute two `MPI_Bcast()` operations in opposite order

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Bcast (buf2, count, type, 1, comm);
        break;
case 1: MPI_Bcast (buf2, count, type, 1, comm);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

Executing this piece of program may lead to two different error situations:

1. The MPI runtime system may match the first `MPI_Bcast()` call of each process. Doing this results in an error, since the two processes specify different roots.
2. The runtime system may match the `MPI_Bcast()` calls with the same root, as it has probably been intended by the programmer. Then a **deadlock** may occur if no system buffers are used or if the system buffers are too small. Collective communication operations are always **blocking**; thus, the operations are *synchronizing* if no or too small system buffers are used. Therefore, the first call of `MPI_Bcast()` blocks the process with rank 0 until the process with rank 1 has called the corresponding `MPI_Bcast()` with the same root. But this cannot happen, since process 1 is blocked due to its first `MPI_Bcast()` operation, waiting for process 0 to call its second `MPI_Bcast()`. Thus, a classical deadlock situation with cyclic waiting results.

The error or deadlock situation can be avoided in this example by letting the participating processes call the matching collective communication operations in the same order.

Deadlocks can also occur when mixing collective communication and single-transfer operations. This can be illustrated by the following example:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Send (buf2, count, type, 1, tag, comm);
        break;
case 1: MPI_Recv (buf2, count, type, 0, tag, comm, &status);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

If no system buffers are used by the MPI implementation, a deadlock because of cyclic waiting occurs: Process 0 blocks when executing `MPI_Bcast()`, until process 1 executes the corresponding `MPI_Bcast()` operation. Process 1 blocks when executing `MPI_Recv()` until process 0 executes the corresponding `MPI_Send()` operation, resulting in cyclic waiting. This can be avoided if both processes execute their corresponding communication operations in the same order.

The **synchronization behavior** of collective communication operations depends on the use of system buffers by the MPI runtime system. If no internal system buffers are used or if the system buffers are too small, collective communication operations may lead to the synchronization of the participating processes. If system buffers are used, there is not necessarily a synchronization. This can be illustrated by the following example:

```
switch (my_rank) {
case 0: MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Send (buf2, count, type, 1, tag, comm);
        break;
case 1: MPI_Recv (buf2, count, type, MPI_ANY_SOURCE, tag,
           comm, &status);
        MPI_Bcast (buf1, count, type, 0, comm);
        MPI_Recv (buf2, count, type, MPI_ANY_SOURCE, tag,
           comm, &status);
        break;
case 2: MPI_Send (buf2, count, type, 1, tag, comm);
        MPI_Bcast (buf1, count, type, 0, comm);
}
```

After having executed `MPI_Bcast()`, process 0 sends a message to process 1 using `MPI_Send()`. Process 2 sends a message to process 1 before executing an `MPI_Bcast()` operation. Process 1 receives two messages from `MPI_ANY_SOURCE`, one before and one after the `MPI_Bcast()` operation. The question is which message will be received from process 1 by which `MPI_Recv()`. Two execution orders are possible:

1. Process 1 first receives the message from process 2:

process 0		**process** 1		**process** 2
		`MPI_Recv()`	$\Longleftarrow$	`MPI_Send()`
`MPI_Bcast()`		`MPI_Bcast()`		`MPI_Bcast()`
`MPI_Send()`	$\Longrightarrow$	`MPI_Recv()`		

   This execution order may occur independent of whether system buffers are used or not. In particular, this execution order is possible also if the calls of `MPI_Bcast()` are synchronizing.

2. Process 1 first receives the message from process 0:

process 0		**process** 1		**process** 2
`MPI_Bcast()`				
`MPI_Send()`	$\Longrightarrow$	`MPI_Recv()`		
		`MPI_Bcast()`		
		`MPI_Recv()`	$\Longleftarrow$	`MPI_Send()`
				`MPI_Bcast()`

This execution order can only occur, if large enough system buffers are used, because otherwise process 0 cannot finish its `MPI_Bcast()` call before process 1 has started its corresponding `MPI_Bcast()`.

Thus, a non-deterministic program behavior results depending on the use of system buffers. Such a program is correct only if both execution orders lead to the intended result. The previous examples have shown that collective communication operations are synchronizing only if the MPI runtime system does not use system buffers to store messages locally before their actual transmission. Thus, when writing a parallel program, the programmer cannot rely on the expectation that collective communication operations lead to a synchronization of the participating processes.

To synchronize a group of processes, MPI provides the operation

```
MPI_Barrier (MPI_Comm comm).
```

The effect of this operation is that all processes belonging to the group of communicator `comm` are blocked until all other processes of this group also have called this operation.

5.3 Process Groups and Communicators

MPI allows the construction of subsets of processes by defining *groups* and *communicators*. A **process group** (or **group** for short) is an ordered set of processes of an application program. Each process of a group gets an uniquely defined process number which is also called **rank**. The ranks of a group always start with 0 and continue consecutively up to the number of processes minus one. A process may be a member of multiple groups and may have different ranks in each of these groups. The MPI system handles the representation and management of process groups. For the programmer, a group is an object of type `MPI_Group` which can only be accessed via a **handle** which may be internally implemented by the MPI system as an index or a reference. Process groups are useful for the implementation of **task-parallel programs** and are the basis for the communication mechanism of MPI.

In many situations, it is useful to partition the processes executing a parallel program into disjoint subsets (groups) which perform independent tasks of the program. This is called **task parallelism**, see also Sect. 3.3.4. The execution of task-parallel program parts can be obtained by letting the processes of a program call different functions or communication operations, depending on their process numbers. But task parallelism can be implemented much easier using the group concept.

5.3.1 Process Groups in MPI

MPI provides a lot of support for process groups. In particular, collective communication operations can be restricted to process groups by using the corresponding communicators. This is important for program libraries where the communication

operations of the calling application program and the communication operations of functions of the program library must be distinguished. If the same communicator is used, an error may occur, e.g., if the application program calls `MPI_Irecv()` with communicator `MPI_COMM_WORLD` using source `MPI_ANY_SOURCE` and tag `MPI_ANY_TAG` immediately before calling a library function. This is dangerous, if the library functions also use `MPI_COMM_WORLD` and if the library function called sends data to the process which executes `MPI_Irecv()` as mentioned above, since this process may then receive library-internal data. This can be avoided by using separate communicators.

In MPI, each point-to-point communication as well as each collective communication is executed in a **communication domain**. There is a separate communication domain for each process group using the ranks of the group. For each process of a group, the corresponding communication domain is *locally* represented by a **communicator**. In MPI, there is a communicator for each process group and each communicator defines a process group. A communicator knows all other communicators of the same communication domain. This may be required for the internal implementation of communication operations. Internally, a group may be implemented as an array of process numbers where each array entry specifies the global process number of one process of the group.

For the programmer, an MPI communicator is an opaque data object of type `MPI_Comm`. MPI distinguishes between **intra-communicators** and **inter-communicators**. Intra-communicators support the execution of arbitrary collective communication operations on a single group of processes. Inter-communicators support the execution of point-to-point communication operations between two process groups. In the following, we only consider intra-communicators which we call communicators for short.

In the preceding sections, we have always used the predefined communicator `MPI_COMM_WORLD` for communication. This communicator comprises all processes participating in the execution of a parallel program. MPI provides several operations to build additional process groups and communicators. These operations are all based on existing groups and communicators. The predefined communicator `MPI_COMM_WORLD` and the corresponding group are normally used as starting point. The process group to a given communicator can be obtained by calling

```
int MPI_Comm_group (MPI_Comm comm, MPI_Group *group),
```

where `comm` is the given communicator and `group` is a pointer to a previously declared object of type `MPI_Group` which will be filled by the MPI call. A predefined group is `MPI_GROUP_EMPTY` which denotes an empty process group.

5.3.1.1 Operations on Process Groups

MPI provides operations to construct new process groups based on existing groups. The predefined empty group `MPI_GROUP_EMPTY` can also be used. The **union** of two existing groups `group1` and `group2` can be obtained by calling

```
int MPI_Group_union (MPI_Group group1,
                     MPI_Group group2,
                     MPI_Group *new_group).
```

The ranks in the new group `new_group` are set such that the processes in `group1` keep their ranks. The processes from `group2` which are not in `group1` get subsequent ranks in consecutive order. The **intersection** of two groups is obtained by calling

```
int MPI_Group_intersection (MPI_Group group1,
                            MPI_Group group2,
                            MPI_Group *new_group),
```

where the process order from `group1` is kept for `new_group`. The processes in `new_group` get successive ranks starting from 0. The **set difference** of two groups is obtained by calling

```
int MPI_Group_difference (MPI_Group group1,
                          MPI_Group group2,
                          MPI_Group *new_group).
```

Again, the process order from `group1` is kept. A `sub_group` of an existing group can be obtained by calling

```
int MPI_Group_incl (MPI_Group group,
                    int p,
                    int *ranks,
                    MPI_Group *new_group),
```

where `ranks` is an integer array with `p` entries. The call of this function creates a new group `new_group` with `p` processes which have ranks from 0 to `p-1`. Process `i` is the process which has rank `ranks[i]` in the given group `group`. For a correct execution of this operation, `group` must contain at least `p` processes, and for $0 \leq$ `i` $<$ `p`, the values `ranks[i]` must be valid process numbers in `group` which are different from each other. Processes can be deleted from a given group by calling

```
int MPI_Group_excl (MPI_Group group,
                    int p,
                    int *ranks,
                    MPI_Group *new_group).
```

This function call generates a new group `new_group` which is obtained from `group` by deleting the processes with ranks `ranks[0]`, ..., `ranks[p-1]`. Again, the entries `ranks[i]` must be valid process ranks in `group` which are different from each other.

Data structures of type MPI_Group cannot be directly accessed by the programmer. But MPI provides operations to obtain information about process groups. The **size** of a process group can be obtained by calling

```
int MPI_Group_size (MPI_Group group, int *size),
```

where the size of the group is returned in parameter size. The **rank** of the calling process in a group can be obtained by calling

```
int MPI_Group_rank (MPI_Group group, int *rank),
```

where the rank is returned in parameter rank. The function

```
int MPI_Group_compare (MPI_Group group1, MPI_Group group2, int *res)
```

can be used to check whether two group representations group1 and group2 describe the same group. The parameter value res = MPI_IDENT is returned if both groups contain the same processes in the same order. The parameter value res = MPI_SIMILAR is returned if both groups contain the same processes, but group1 uses a different order than group2. The parameter value res = MPI_UNEQUAL means that the two groups contain different processes. The function

```
int MPI_Group_free (MPI_Group *group)
```

can be used to free a group representation if it is no longer needed. The group handle is set to MPI_GROUP_NULL.

5.3.1.2 Operations on Communicators

A new intra-communicator to a given group of processes can be generated by calling

```
int MPI_Comm_create (MPI_Comm comm,
                     MPI_Group group,
                     MPI_Comm *new_comm),
```

where comm specifies an existing communicator. The parameter group must specify a process group which is a subset of the process group associated with comm. For a correct execution, it is required that all processes of comm perform the call of MPI_Comm_create() and that each of these processes specifies the same group argument. As a result of this call, each calling process which is a member of group obtains a pointer to the new communicator in new_comm. Processes not belonging to group get MPI_COMM_NULL as return value in new_comm.

MPI also provides functions to get information about communicators. These functions are implemented as local operations which do not involve communication

to be executed. The size of the process group associated with a communicator comm can be requested by calling the function

```
int MPI_Comm_size (MPI_Comm comm, int *size).
```

The size of the group is returned in parameter size. For comm = MPI_COMM_WORLD the total number of processes executing the program is returned. The rank of a process in a particular group associated with a communicator comm can be obtained by calling

```
int MPI_Comm_rank (MPI_Comm comm, int *rank).
```

The group rank of the calling process is returned in rank. In previous examples, we have used this function to obtain the global rank of processes of MPI_COMM_WORLD. Two communicators comm1 and comm2 can be compared by calling

```
int MPI_Comm_compare (MPI_Comm comm1, MPI_Comm comm2, int *res).
```

The result of the comparison is returned in parameter res; res = MPI_IDENT is returned, if comm1 and comm2 denote the same communicator data structure. The value res = MPI_CONGRUENT is returned, if the associated groups of comm1 and comm2 contain the same processes with the same rank order. If the two associated groups contain the same processes in different rank order, res = MPI_SIMILAR is returned. If the two groups contain different processes, res = MPI_UNEQUAL is returned.

For the direct construction of communicators, MPI provides operations for the duplication, deletion, and splitting of communicators. A communicator can be **duplicated** by calling the function

```
int MPI_Comm_dup (MPI_Comm comm, MPI_Comm *new_comm),
```

which creates a new intra-communicator new_comm with the same characteristics (assigned group and topology) as comm. The new communicator new_comm represents a new distinct communication domain. Duplicating a communicator allows the programmer to separate communication operations executed by a library from communication operations executed by the application program itself, thus avoiding any conflict. A communicator can be **deallocated** by calling the MPI operation

```
int MPI_Comm_free (MPI_Comm *comm).
```

This operation has the effect that the communicator data structure comm is freed as soon as all pending communication operations performed with this communicator are completed. This operation could, e.g., be used to free a communicator which has previously been generated by duplication to separate library communication from

communication of the application program. Communicators should not be assigned by simple assignments of the form `comm1 = comm2`, since a deallocation of one of the two communicators involved with `MPI_Comm_free()` would have a side effect on the other communicator, even if this is not intended. A **splitting** of a communicator can be obtained by calling the function

```
int MPI_Comm_split (MPI_Comm comm,
                    int color,
                    int key,
                    MPI_Comm *new_comm).
```

The effect is that the process group associated with `comm` is partitioned into disjoint subgroups. The number of subgroups is determined by the number of different values of `color`. Each subgroup contains all processes which specify the same value for `color`. Within each subgroup, the processes are ranked in the order defined by argument value `key`. If two processes in a subgroup specify the same value for `key`, the order in the original group is used. If a process of `comm` specifies `color = MPI_UNDEFINED`, it is not a member of any of the subgroups generated. The subgroups are not directly provided in the form of an `MPI_GROUP` representation. Instead, each process of `comm` gets a pointer `new_comm` to the communicator of that subgroup which the process belongs to. For `color = MPI_UNDEFINED`, `MPI_COMM_NULL` is returned as `new_comm`.

Example We consider a group of 10 processes each of which calls the operation `MPI_Comm_split()` with the following argument values [163]:

process	a	b	c	d	e	f	g	h	i	j
rank	0	1	2	3	4	5	6	7	8	9
color	0	$\perp$	3	0	3	0	0	5	3	$\perp$
key	3	1	2	5	1	1	1	2	1	0

This call generates three subgroups {f, g, a, d}, {e, i, c}, and {h} which contain the processes in this order. In the table, the entry $\perp$ represents `color = MPI_UNDEFINED`. □

The operation `MPI_Comm_split()` can be used to prepare a task-parallel execution. The different communicators generated can be used to perform communication within the task-parallel parts, thus separating the communication domains.

5.3.2 Process Topologies

Each process of a process group has a unique rank within this group which can be used for communication with this process. Although a process is uniquely defined by its group rank, it is often useful to have an alternative representation and access. This is the case if an algorithm performs computations and communication on a two-dimensional or a three-dimensional grid where grid points are assigned to different

processes and the processes exchange data with their neighboring processes in each dimension by communication. In such situations, it is useful if the processes can be arranged according to the communication pattern in a grid structure such that they can be addressed via two-dimensional or three-dimensional coordinates. Then each process can easily address its neighboring processes in each dimension. MPI supports such a logical arrangement of processes by defining **virtual topologies** for intra-communicators, which can be used for communication within the associated process group.

A virtual Cartesian grid structure of arbitrary dimension can be generated by calling

```
int MPI_Cart_create (MPI_Comm comm,
                     int ndims,
                     int *dims,
                     int *periods,
                     int reorder,
                     MPI_Comm *new_comm)
```

where `comm` is the original communicator without topology, `ndims` specifies the number of dimensions of the grid to be generated, `dims` is an integer array of size `ndims` such that `dims[i]` is the number of processes in dimension `i`. The entries of `dims` must be set such that the product of all entries is the number of processes contained in the new communicator `new_comm`. In particular, this product must not exceed the number of processes of the original communicator `comm`. The boolean array `periods` of size `ndims` specifies for each dimension whether the grid is periodic (entry 1 or `true`) or not (entry 0 or `false`) in this dimension. For `reorder = false`, the processes in `new_comm` have the same rank as in `comm`. For `reorder = true`, the runtime system is allowed to reorder processes, e.g., to obtain a better mapping of the process topology to the physical network of the parallel machine.

Example We consider a communicator with 12 processes [163]. For `ndims=2`, using the initializations `dims[0]=3`, `dims[1]=4`, `periods[0]=periods[1]=0`, `reorder=0`, the call

```
MPI_Cart_create (comm, ndims, dims, periods, reorder, &new_comm)
```

generates a virtual 3×4 grid with the following group ranks and coordinates:

0 (0,0)	1 (0,1)	2 (0,2)	3 (0,3)
4 (1,0)	5 (1,1)	6 (1,2)	7 (1,3)
8 (2,0)	9 (2,1)	10 (2,2)	11 (2,3)

The Cartesian coordinates are represented in the form (row, column). In the communicator, the processes are ordered according to their rank rowwise in increasing order. □

To help the programmer to select a balanced distribution of the processes for the different dimensions, MPI provides the function

```
int MPI_Dims_create (int nnodes, int ndims, int *dims)
```

where `ndims` is the number of dimensions in the grid and `nnodes` is the total number of processes available. The parameter `dims` is an integer array of size `ndims`. After the call, the entries of `dims` are set such that the `nnodes` processes are balanced as much as possible among the different dimensions, i.e., each dimension has about equal size. But the size of a dimension `i` is set only if `dims[i] = 0` when calling `MPI_Dims_create()`. The number of processes in a dimension `j` can be fixed by setting `dims[j]` to a positive value before the call. This entry is then not modified by this call and the other entries of `dims` are set by the call accordingly.

When defining a virtual topology, each process has a group rank, and also a position in the virtual grid topology which can be expressed by its Cartesian coordinates. For the translation between group ranks and Cartesian coordinates, MPI provides two operations. The operation

```
int MPI_Cart_rank (MPI_Comm comm, int *coords, int *rank)
```

translates the Cartesian coordinates provided in the integer array `coords` into a group rank and returns it in parameter `rank`. The parameter `comm` specifies the communicator with Cartesian topology. For the opposite direction, the operation

```
int MPI_Cart_coords (MPI_Comm comm,
                     int rank,
                     int ndims,
                     int *coords)
```

translates the group rank provided in `rank` into Cartesian coordinates, returned in integer array `coords`, for a virtual grid; `ndims` is the number of dimensions of the virtual grid defined for communicator `comm`.

Virtual topologies are typically defined to facilitate the determination of communication partners of processes. A typical communication pattern in many grid-based algorithms is that processes communicate with their neighboring processes in a specific dimension. To determine these neighboring processes, MPI provides the operation

```
int MPI_Cart_shift (MPI_Comm comm,
                    int dir,
                    int displ,
                    int *rank_source,
                    int *rank_dest)
```

where dir specifies the dimension for which the neighboring process should be determined. The parameter displ specifies the displacement, i.e., the distance to the neighbor. Positive values of displ request the neighbor in upward direction, negative values request for downward direction. Thus, displ = -1 requests the neighbor immediately preceding, displ = 1 requests the neighboring process which follows directly. The result of the call is that rank_dest contains the group rank of the neighboring process in the specified dimension and distance. The rank of the process for which the calling process is the neighboring process in the specified dimension and distance is returned in rank_source. Thus, the group ranks returned in rank_dest and rank_source can be used as parameters for MPI_Sendrecv(), as well as for separate MPI_Send() and MPI_Recv(), respectively.

Example As example, we consider 12 processes that are arranged in a 3×4 grid structure with periodic connections [163]. Each process stores a floating-point value which is exchanged with the neighboring process in dimension 0, i.e., within the columns of the grid:

```
int coords[2], dims[2], periods[2], source, dest, my_rank,
    reorder;
MPI_Comm comm_2d;
MPI_status status;
float a, b;
MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
dims[0] = 3; dims[1] = 4;
periods[0] = periods[1] = 1;
reorder = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods, reorder,
   &comm_2d);
MPI_Cart_coords (comm_2d, my_rank, 2, coords);
MPI_Cart_shift (comm_2d, 0, coords[1], &source, &dest);
a = my_rank;
MPI_Sendrecv (&a, 1, MPI_FLOAT, dest, 0, &b, 1, MPI_FLOAT,
   source, 0, comm_2d, &status);
```

In this example, the specification displs = coord[1] is used as displacement for MPI_Cart_shift(), i.e., the position in dimension 1 is used as displacement. Thus, the displacement increases with column position, and in each column of the grid, a different exchange is executed. MPI_Cart_shift() is used to determine the communication partners dest and source for each process. These are then used as parameters for MPI_Sendrecv(). The following diagram illustrates the exchange. For each process, its rank, its Cartesian coordinates, and its communication partners in the form source/dest are given in this order. For example, for the process with rank=5, it is coords[1]=1, and therefore source=9 (lower neighbor in dimension 0) and dest=1 (upper neighbor in dimension 0).

□

0 (0,0) 0|0	1 (0,1) 9|5	2 (0,2) 6|10	3 (0,3) 3|3
4 (1,0) 4|4	5 (1,1) 1|9	6 (1,2) 10|2	7 (1,3) 7|7
8 (2,0) 8|8	9 (2,1) 5|1	10 (2,2) 2|6	11 (2,3) 11|11

If a virtual topology has been defined for a communicator, the corresponding grid can be partitioned into subgrids by using the MPI function

```
int MPI_Cart_sub (MPI_Comm comm,
                  int *remain_dims,
                  MPI_Comm *new_comm).
```

The parameter `comm` denotes the communicator for which the virtual topology has been defined. The subgrid selection is controlled by the integer array `remain_dims` which contains an entry for each dimension of the original grid.

Setting `remain_dims[i] = 1` means that the `i`th dimension is kept in the subgrid; `remain_dims[i] = 0` means that the `i`th dimension is dropped in the subgrid. In this case, the size of this dimension determines the number of subgrids generated in this dimension. A call of `MPI_Cart_sub()` generates a new communicator `new_comm` for each calling process, representing the corresponding subgroup of the subgrid to which the calling process belongs. The dimensions of the different subgrids result from the dimensions for which `remain_dims[i]` has been set to `1`. The total number of subgrids generated is defined by the product of the number of processes in all dimensions `i` for which `remain_dims[i]` has been set to `0`.

Example We consider a communicator `comm` for which a $2 \times 3 \times 4$ virtual grid topology has been defined. Calling

```
int MPI_Cart_sub (comm_3d, remain_dims, &new_comm)
```

with `remain_dims=(1,0,1)` generates three 2×4 grids and each process gets a communicator for its corresponding subgrid, see Fig. 5.12 for an illustration. □

MPI also provides functions to inquire information about a virtual topology that has been defined for a communicator. The MPI function

```
int MPI_Cartdim_get (MPI_Comm comm,int *ndims)
```

returns in parameter `ndims` the number of dimensions of the virtual grid associated with communicator `comm`. The MPI function

```
int MPI_Cart_get (MPI_Comm comm,
                  int maxdims,
                  int *dims,
                  int *periods,
                  int *coords)
```

returns information about the virtual topology defined for communicator `comm`. This virtual topology should have `maxdims` dimensions, and the arrays `dims`, `periods`, and `coords` should have this size. The following information is returned by this call: Integer array `dims` contains the number of processes in each dimension of the virtual grid, the boolean array `periods` contains the corresponding periodicity information. The integer array `coords` contains the Cartesian coordinates of the calling process.

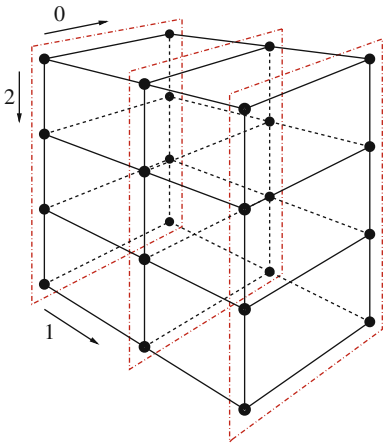

Fig. 5.12 Partitioning of a three-dimensional grid of size $2 \times 3 \times 4$ into three two-dimensional grids of size 2×4 each

This figure will be printed in b/w

5.3.3 Timings and Aborting Processes

To measure the parallel execution times of program parts, MPI provides the function

```
double MPI_Wtime (void)
```

which returns as a floating-point value the number of seconds elapsed since a fixed point in time in the past. A typical usage for timing would be:

```
start = MPI_Wtime();
part_to_measure();
end = MPI_Wtime();
```

`MPI_Wtime()` does not return a system time, but the absolute time elapsed between the start and the end of a program part, including times at which the

process executing `part_to_measure()` has been interrupted. The resolution of `MPI_Wtime()` can be requested by calling

```
double MPI_Wtick (void)
```

which returns the time between successive clock ticks in seconds as floating-point value. If the resolution is a microsecond, `MPI_Wtick()` will return 10^{-6}. The execution of all processes of a communicator can be aborted by calling the MPI function

```
int MPI_Abort (MPI_Comm comm, int error_code)
```

where `error_code` specifies the error code to be used, i.e., the behavior is as if the main program has been terminated with `return error_code`.

5.4 Introduction to MPI-2

For a continuous development of MPI, the MPI Forum has defined extensions to MPI as described in the previous sections. These extensions are often referred to as MPI-2. The original MPI standard is referred to as MPI-1. The current version of MPI-1 is described in the MPI document, version 1.3 [55]. Since MPI-2 comprises all MPI-1 operations, each correct MPI-1 program is also a correct MPI-2 program. The most important extensions contained in MPI-2 are dynamic process management, one-sided communications, parallel I/O, and extended collective communications. In the following, we give a short overview of the most important extensions. For a more detailed description, we refer to the current version of the MPI-2 document, version 2.1, see [56].

5.4.1 Dynamic Process Generation and Management

MPI-1 is based on a **static process model**: The processes used for the execution of a parallel program are implicitly created before starting the program. No processes can be added during program execution. Inspired by PVM [63], MPI-2 extends this process model to a **dynamic process model** which allows the creation and deletion of processes at any time during program execution. MPI-2 defines the interface for dynamic process management as a collection of suitable functions and gives some advice for an implementation. But not all implementation details are fixed to support an implementation for different operating systems.

5.4.1.1 `MPI_Info` Objects

Many MPI-2 functions use an additional argument of type `MPI_Info` which allows the provision of additional information for the function, depending on the spe-

cific operating system used. But using this feature may lead to non-portable MPI programs. MPI_Info provides opaque objects where each object can store arbitrary (key, value) pairs. In C, both entries are strings of type char, terminated with \0. Since MPI_Info objects are opaque, their implementation is hidden from the user. Instead, some functions are provided for access and manipulation. The most important ones are described in the following. The function

```
int MPI_Info_create (MPI_Info *info)
```

can be used to generate a new object of type MPI_Info. Calling the function

```
int MPI_Info_set (MPI_Info info, char *key, char *value)
```

adds a new (key, value) pair to the MPI_Info structure info. If a value for the same key was previously stored, the old value is overwritten. The function

```
int MPI_Info_get (MPI_Info info,
                  char *key,
                  int valuelen,
                  char *value,
                  int *flag)
```

can be used to retrieve a stored pair (key, value) from info. The programmer specifies the value of key and the maximum length valuelen of the value entry. If the specified key exists in info, the associated value is returned in parameter value. If the associated value string is longer than valuelen, the returned string is truncated after valuelen characters. If the specified key exists in info, true is returned in parameter flag; otherwise, false is returned. The function

```
int MPI_Info_delete(MPI_Info info, char *key)
```

can be used to delete an entry (key, value) from info. Only the key has to be specified.

5.4.1.2 Process Creation and Management

A number of MPI processes can be started by calling the function

```
int MPI_Comm_spawn (char *command,
                    char *argv[],
                    int maxprocs,
                    MPI_Info info,
                    int root,
                    MPI_Comm comm,
                    MPI_Comm *intercomm,
                    int errcodes[]).
```

The parameter command specifies the name of the program to be executed by each of the processes, argv[] contains the arguments for this program. In contrast to the standard C convention, argv[0] is not the program name but the first argument for the program. An empty argument list is specified by MPI_ARGV_NULL. The parameter maxprocs specifies the number of processes to be started. If the MPI runtime system is not able to start maxprocs processes, an error message is generated. The parameter info specifies an MPI_Info data structure with (key, value) pairs providing additional instructions for the MPI runtime system on how to start the processes. This parameter could be used to specify the path of the program file as well as its arguments, but this may lead to non-portable programs. Portable programs should use MPI_INFO_NULL.

The parameter root specifies the number of the root process from which the new processes are spawned. Only this root process provides values for the preceding parameters. But the function MPI_Comm_spawn() is a collective operation, i.e., all processes belonging to the group of the communicator comm must call the function. The parameter intercomm contains an intercommunicator after the successful termination of the function call. This intercommunicator can be used for communication between the original group of comm and the group of processes just spawned.

The parameter errcodes is an array with maxprocs entries in which the status of each process to be spawned is reported. When a process could be spawned successfully, its corresponding entry in errcodes will be set to MPI_SUCCESS. Otherwise, an implementation-specific error code will be reported.

A successful call of MPI_Comm_spawn() starts maxprocs identical copies of the specified program and creates an intercommunicator, which is provided to all calling processes. The new processes belong to a separate group and have a separate MPI_COMM_WORLD communicator comprising all processes spawned. The spawned processes can access the intercommunicator created by MPI_Comm_spawn() by calling the function

```
int MPI_Comm_get_parent(MPI_Comm *parent).
```

The requested intercommunicator is returned in parameter parent. Multiple MPI programs or MPI programs with different argument values can be spawned by calling the function

```
int MPI_Comm_spawn_multiple (int count,
                             char *commands[],
                             char **argv[],
                             int maxprocs[],
                             MPI_Info infos[],
                             int root,
                             MPI_Comm comm,
                             MPI_Comm *intercomm,
                             int errcodes[])
```

where count specifies the number of different programs to be started. Each of the following four arguments specifies an array with count entries where each entry has the same type and meaning as the corresponding parameters for MPI_Comm_spawn(): The argument commands[] specifies the names of the programs to be started, argv[] contains the corresponding arguments, maxprocs[] defines the number of copies to be started for each program, and infos[] provides additional instructions for each program. The other arguments have the same meaning as for MPI_comm_spawn().

After the call of MPI_Comm_spawn_multiple() has been terminated, the array errcodes[] contains an error status entry for each process created. The entries are arranged in the order given by the commands[] array. In total, errcodes[] contains

$$\sum_{i=0}^{\texttt{count}-1} \texttt{maxprocs[i]}$$

entries. There is a difference between calling MPI_Comm_spawn() multiple times and calling MPI_Comm_spawn_multiple() with the same arguments. Calling the function MPI_Comm_spawn_multiple() creates one communicator MPI_COMM_WORLD for all newly created processes. Multiple calls of MPI_Comm_spawn() generate separate communicators MPI_COMM_WORLD, one for each process group created.

The attribute MPI_UNIVERSE_SIZE specifies the maximum number of processes that can be started in total for a given application program. The attribute is initialized by MPI_Init().

5.4.2 One-Sided Communication

MPI provides single transfer and collective communication operations as described in the previous sections. For collective communication operations, each process of a communicator calls the communication operation to be performed. For single-transfer operations, a sender and a receiver process must cooperate and actively execute communication operations: In the simplest case, the sender executes an MPI_Send() operation, and the receiver executes an MPI_Recv() operation. Therefore, this form of communication is also called *two-sided communication*. The position of the MPI_Send() operation in the sender process determines at which time the data is sent. Similarly, the position of the MPI_Recv() operation in the receiver process determines at which time the receiver stores the received data in its local address space.

In addition to two-sided communication, MPI-2 supports *one-sided communication*. Using this form of communication, a source process can access the address space at a target process without an active participation of the target process. This form of communication is also called Remote Memory Access (RMA). RMA facilitates communication for applications with dynamically changing data access

patterns by supporting a flexible dynamic distribution of program data among the address spaces of the participating processes. But the programmer is responsible for the coordinated memory access. In particular, a concurrent manipulation of the same address area by different processes at the same time must be avoided to inhibit race conditions. Such race conditions cannot occur for two-sided communications.

5.4.2.1 Window Objects

If a process A should be allowed to access a specific memory region of a process B using one-sided communication, process B must expose this memory region for external access. Such a memory region is called *window*. A window can be exposed by calling the function

```
int MPI_Win_create (void *base,
                    MPI_Aint size,
                    int displ_unit,
                    MPI_Info info,
                    MPI_Comm comm,
                    MPI_Win *win).
```

This is a collective call which must be executed by each process of the communicator `comm`. Each process specifies a window in its local address space that it exposes for RMA by other processes of the same communicator.

The starting address of the window is specified in parameter `base`. The size of the window is given in parameter `size` as number of bytes. For the size specification, the predefined MPI type `MPI_Aint` is used instead of `int` to allow window sizes of more than 2^{32} bytes. The parameter `displ_unit` specifies the displacement (in bytes) between neighboring window entries used for one-sided memory accesses. Typically, `displ_unit` is set to `1` if bytes are used as unit or to `sizeof(type)` if the window consists of entries of type `type`. The parameter `info` can be used to provide additional information for the runtime system. Usually, `info=MPI_INFO_NULL` is used. The parameter `comm` specifies the communicator of the processes which participate in the `MPI_Win_create()` operation. The call of `MPI_Win_create()` returns a window object of type `MPI_Win` in parameter `win` to the calling process. This window object can then be used for RMA to memory regions of other processes of `comm`.

A window exposed for external accesses can be closed by letting all processes of the corresponding communicator call the function

```
int MPI_Win_free (MPI_Win *win)
```

thus freeing the corresponding window object `win`. Before calling `MPI_Win_free()`, the calling process must have finished all operations on the specified window.

5.4.2.2 RMA Operations

For the actual one-sided data transfer, MPI provides three *non-blocking* RMA operations: `MPI_Put()` transfers data from the memory of the calling process into the window of another process; `MPI_Get()` transfers data from the window of a target process into the memory of the calling process; `MPI_Accumulate()` supports the accumulation of data in the window of the target process. These operations are *non-blocking*: When control is returned to the calling process, this does not necessarily mean that the operation is completed. To test for the completion of the operation, additional synchronization operations like `MPI_Win_fence()` are provided as described below. Thus, a similar usage model as for non-blocking two-sided communication can be used. The local buffer of an RMA communication operation should not be updated or accessed until the subsequent synchronization call returns.

The transfer of a data block into the window of another process can be performed by calling the function

```
int MPI_Put (void *origin_addr,
             int origin_count,
             MPI_Datatype origin_type,
             int target_rank,
             MPI_Aint target_displ,
             int target_count,
             MPI_Datatype target_type,
             MPI_Win win)
```

where `origin_addr` specifies the start address of the data buffer provided by the calling process and `origin_count` is the number of buffer entries to be transferred. The parameter `origin_type` defines the type of the entries. The parameter `target_rank` specifies the rank of the target process which should receive the data block. This process must have created the window object `win` by a preceding `MPI_Win_create()` operation, together with all processes of the communicator group to which the process calling `MPI_Put()` also belongs to. The remaining parameters define the position and size of the target buffer provided by the target process in its window: `target_displ` defines the displacement from the start of the window to the start of the target buffer, `target_count` specifies the number of entries in the target buffer, `target_type` defines the type of each entry in the target buffer. The data block transferred is stored in the memory of the target process at position `target_addr := window_base + target_displ * displ_unit` where `window_base` is the start address of the window in the memory of the target process and `displ_unit` is the distance between neighboring window entries as defined by the target process when creating the window with `MPI_Win_create()`. The execution of an `MPI_Put()` operation by a process `source` has the same effect as a two-sided communication for which process `source` executes the send operation

```
int MPI_Isend (origin_addr, origin_count, origin_type,
               target_rank, tag, comm)
```

and the target process executes the receive operation

```
int MPI_Recv (target_addr, target_count, target_type,
              source, tag, comm, &status)
```

where comm is the communicator for which the window object has been defined. For a correct execution of the operation, some constraints must be satisfied: The target buffer defined must fit in the window of the target process and the data block provided by the calling process must fit into the target buffer. In contrast to MPI_Isend() operations, the send buffers of multiple successive MPI_Put() operations may overlap, even if there is no synchronization in between. Source and target processes of an MPI_Put() operation may be identical.

To transfer a data block from the window of another process into a local data buffer, the MPI function

```
int MPI_Get (void *origin_addr,
             int origin_count,
             MPI_Datatype origin_type,
             int target_rank,
             MPI_Aint target_displ,
             int target_count,
             MPI_Datatype target_type,
             MPI_Win win)
```

is provided. The parameter origin_addr specifies the start address of the receive buffer in the local memory of the calling process; origin_count defines the number of elements to be received; origin_type is the type of each of the elements. Similar to MPI_Put(), target_rank specifies the rank of the target process which provides the data and win is the window object previously created. The remaining parameters define the position and size of the data block to be transferred out of the window of the target process. The start address of the data block in the memory of the target process is given by target_addr := window_base + target_displ * displ_unit.

For the accumulation of data values in the memory of another process, MPI provides the operation

```
int MPI_Accumulate (void *origin_addr,
                    int origin_count,
                    MPI_Datatype origin_type,
                    int target_rank,
                    MPI_Aint target_displ,
                    int target_count,
                    MPI_Datatype target_type,
                    MPI_Op op,
                    MPI_Win win)
```

The parameters have the same meaning as for MPI_Put(). The additional parameter op specifies the reduction operation to be applied for the accumulation. The same predefined reduction operations as for MPI_Reduce() can be used, see Sect. 5.2, p. 215. Examples are MPI_MAX and MPI_SUM. User-defined reduction operations cannot be used. The execution of an MPI_Accumulate() has the effect that the specified reduction operation is applied to corresponding entries of the source buffer and the target buffer and that the result is written back into the target buffer. Thus, data values can be accumulated in the target buffer provided by another process. There is an additional reduction operation MPI_REPLACE which allows the replacement of buffer entries in the target buffer, without taking the previous values of the entries into account. Thus, MPI_Put() can be considered as a special case of MPI_Accumulate() with reduction operation MPI_REPLACE.

There are some constraints for the execution of one-sided communication operations by different processes to avoid race conditions and to support an efficient implementation of the operations. Concurrent conflicting accesses to the same memory location in a window are not allowed. At each point in time during program execution, each memory location of a window can be used as target of at most one one-sided communication operation. Exceptions are accumulation operations: Multiple concurrent MPI_Accumulate() operations can be executed at the same time for the same memory location. The result is obtained by using an arbitrary order of the executed accumulation operations. The final accumulated value is the same for all orders, since the predefined reduction operations are commutative. A window of a process P cannot be used concurrently by an MPI_Put() or MPI_Accumulate() operation of another process and by a local store operation of P, even if different locations in the window are addressed.

MPI provides three synchronization mechanisms for the coordination of one-sided communication operations executed in the windows of a group of processes. These three mechanisms are described in the following.

5.4.2.3 Global Synchronization

A global synchronization of all processes of the group of a window object can be obtained by calling the MPI function

```
int MPI_Win_fence (int assert, MPI_Win win)
```

where win specifies the window object. MPI_Win_fence() is a collective operation to be performed by all processes of the group of win. The effect of the call is that all RMA operations originating from the calling process and started before the MPI_Win_fence() call are locally completed at the calling process before control is returned to the calling process. RMA operations started after the MPI_Win_fence() call accesses the specified target window only after the corresponding target process has called its corresponding MPI_Win_fence() operation. The intended use of MPI_Win_fence() is the definition of program areas in which one-sided communication operations are executed. Such program areas

are surrounded by calls of MPI_Win_fence(), thus establishing communication phases that can be mixed with computation phases during which no communication is required. Such communication phases are also referred to as **access epochs** in MPI. The parameter assert can be used to specify assertions on the context of the call of MPI_Win_fence() which can be used for optimizations by the MPI runtime system. Usually, assert=0 is used, not providing additional assertions.

Global synchronization with MPI_Win_fence() is useful in particular for applications with regular communication pattern in which computation phases alternate with communication phases.

Example As example, we consider an iterative computation of a distributed data structure A. In each iteration step, each participating process updates its local part of the data structure using the function update(). Then, parts of the local data structure are transferred into the windows of neighboring processes using MPI_Put(). Before the transfer, the elements to be transferred are copied into a contiguous buffer. This copy operation is performed by update_buffer(). The communication operations are surrounded by MPI_Win_fence() operations to separate the communication phases of successive iterations from each other. This results in the following program structure:

```
while (!converged(A)) {
   update(A);
   update_buffer(A, from_buf);
   MPI_Win_fence(0, win);
   for (i=0; i<num_neighbors; i++)
      MPI_Put(&from_buf[i], size[i], MPI_INT, neighbor[i],
      to_disp[i],
              size[i], MPI_INT, win);
   MPI_Win_fence(0, win);
}
```

The iteration is controlled by the function converged().

5.4.2.4 Loose Synchronization

MPI also supports a loose synchronization which is restricted to pairs of communicating processes. To perform this form of synchronization, an accessing process defines the start and the end of an **access epoch** by a call to MPI_Win_start() and MPI_Win_complete(), respectively. The target process of the communication defines a corresponding **exposure epoch** by calling MPI_Win_post() to start the exposure epoch and MPI_Win_wait() to end the exposure epoch. A synchronization is established between MPI_Win_start() and MPI_Win_post() in the sense that all RMAs which the accessing process issues after its MPI_Win_start() call are executed not before the target process has completed its MPI_Win_post() call. Similarly, a synchronization between MPI_Win_complete() and MPI_Win_wait() is established in the sense that the MPI_Win_wait() call is

completed at the target process not before all RMAs of the accessing process in the corresponding access epoch are terminated.

To use this form of synchronization, before performing an RMA, a process defines the start of an access epoch by calling the function

```
int MPI_Win_start (MPI_Group group,
                   int assert,
                   MPI_Win win)
```

where `group` is a group of target processes. Each of the processes in `group` must issue a matching call of `MPI_Win_post()`. The parameter `win` specifies the window object to which the RMA is made. MPI supports a blocking and a non-blocking behavior of `MPI_Win_start()`:

- Blocking behavior: The call of `MPI_Win_start()` is blocked until all processes of `group` have completed their corresponding calls of `MPI_Win_post()`.
- Non-blocking behavior: The call of `MPI_Win_start()` is completed at the accessing process without blocking, even if there are processes in `group` which have not yet issued or finished their corresponding call of `MPI_Win_post()`. Control is returned to the accessing process and this process can issue RMA operations like `MPI_Put()` or `MPI_Get()`. These calls are then delayed until the target process has finished its `MPI_Win_post()` call.

The exact behavior depends on the MPI implementation. The end of an access epoch is indicated by the accessing process by calling

```
int MPI_Win_complete (MPI_Win win)
```

where `win` is the window object which has been accessed during this access epoch. Between the call of `MPI_Win_start()` and `MPI_Win_complete()`, only RMA operations to the window `win` of processes belonging to `group` are allowed. When calling `MPI_Win_complete()`, the calling process is blocked until all RMA operations to `win` issued in the corresponding access epoch have been completed at the accessing process. An `MPI_Put()` call issued in the access epoch can be completed at the calling process as soon as the local data buffer provided can be reused. But this does not necessarily mean that the data buffer has already been stored in the window of the target process. It might as well have been stored in a local system buffer of the MPI runtime system. Thus, the termination of `MPI_Win_complete()` does not imply that all RMA operations have taken effect at the target processes.

A process indicates the start of an RMA exposure epoch for a local window `win` by calling the function

```
int MPI_Win_post (MPI_Group group,
                  int assert,
                  MPI_Win win).
```

Only processes in group are allowed to access the window during this exposure epoch. Each of the processes in group must issue a matching call of the function MPI_Win_start(). The call of MPI_Win_post() is non-blocking. A process indicates the end of an RMA exposure epoch for a local window win by calling the function

```
int MPI_Win_wait (MPI_Win win).
```

This call blocks until all processes of the group defined in the corresponding MPI_Win_post() call have issued their corresponding MPI_Win_complete() calls. This ensures that all these processes have terminated the RMA operations of their corresponding access epoch to the specified window. Thus, after the termination of MPI_Win_wait(), the calling process can reuse the entries of its local window, e.g., by performing local accesses. During an exposure epoch, indicated by surrounding MPI_Win_post() and MPI_Win_wait() calls, a process should not perform local operations on the specified window to avoid access conflicts with other processes.

By calling the function

```
int MPI_Win_test (MPI_Win win, int *flag)
```

a process can test whether the RMA operation of other processes to a local window has been completed or not. This call can be considered as the non-blocking version of MPI_Win_wait(). The parameter flag=1 is returned by the call if all RMA operations to win have been terminated. In this case, MPI_Win_test() has the same effect as MPI_Win_wait() and should not be called again for the same exposure epoch. The parameter flag=0 is returned if not all RMA operations to win have been finished yet. In this case, the call has no further effect and can be repeated later.

The synchronization mechanism described can be used for arbitrary communication patterns on a group of processes. A communication pattern can be described by a directed graph $G = (V, E)$ where V is the set of participating processes. There exists an edge $(i, j) \in E$ from process i to process j, if i accesses the window of j by an RMA operation. Assuming that the RMA operations are performed on window win, the required synchronization can be reached by letting each participating process execute MPI_Win_start(target_group,0,win) followed by MPI_Win_post(source_group,0,win) where source_group= $\{i;(i, j) \in E\}$ denotes the set of accessing processes and target_group= $\{j;(i, j) \in E\}$ denotes the set of target processes.

Example This form of synchronization is illustrated by the following example, which is a variation of the previous example describing the iterative computation of a distributed data structure:

```
while (!converged (A)) {
   update(A);
   update_buffer(A, from_buf);
   MPI_Win_start(target_group, 0, win);
   MPI_Win_post(source_group, 0, win);
   for (i=0; i<num_neighbors; i++)
      MPI_Put(&from_buf[i], size[i], MPI_INT, neighbor[i], to
      _disp[i],
              size[i], MPI_INT, win);
   MPI_Win_complete(win);
   MPI_Win_wait(win);
}
```

In the example, it is assumed that `source_group` and `target_group` have been defined according to the communication pattern used by all processes as described above. An alternative would be that each process defines a set `source_group` of processes which are allowed to access its local window and a set `target_group` of processes whose window the process is going to access. Thus, each process potentially defines different source and target groups, leading to a weaker form of synchronization as for the case that all processes define the same source and target groups. □

5.4.2.5 Lock Synchronization

To support the model of a shared address space, MPI provides a synchronization mechanism for which only the accessing process actively executes communication operations. Using this form of synchronization, it is possible that two processes exchange data via RMA operations executed on the window of a third process without an active participation of the third process. To avoid access conflicts, a lock mechanism is provided as typically used in programming environments for shared address spaces, see Chap. 6. This means that the accessing process locks the accessed window before the actual access and releases the lock again afterwards. To lock a window before an RMA operation, MPI provides the operation

```
int MPI_Win_lock (int lock_type,
                  int rank,
                  int assert,
                  MPI_Win win).
```

A call of this function starts an RMA access epoch for the window `win` at the process with rank `rank`. Two lock types are supported, which can be specified by parameter `lock_type`. An *exclusive lock* is indicated by `lock_type = MPI_LOCK_EXCLUSIVE`. This lock type guarantees that the following RMA operations executed by the calling process are protected from RMA operations of other processes, i.e., exclusive access to the window is ensured. Exclusive locks should

be used if the executing process will change the value of window entries using MPI_Put() and if these entries could also be accessed by other processes.

A shared lock is indicated by lock_type = MPI_LOCK_SHARED. This lock type guarantees that the following RMA operations of the calling process are protected from *exclusive* RMA operations of other processes, i.e., other processes are not allowed to change entries of the window via RMA operations that are protected by an exclusive lock. But other processes are allowed to perform RMA operations on the same window that are also protected by a shared lock.

Shared locks should be used if the executing process accesses window entries only by MPI_Get() or MPI_Accumulate(). When a process wants to read or manipulate entries of its local window using local operations, it must protect these local operations with a lock mechanism, if these entries can also be accessed by other processes.

An access epoch started by MPI_Win_lock() for a window win can be terminated by calling the MPI function

```
int MPI_Win_unlock (int rank,
                    MPI_Win win)
```

where rank is the rank of the target process. The call of this function blocks until all RMA operations issued by the calling process on the specified window have been completed both at the calling process and at the target process. This guarantees that all manipulations of window entries issued by the calling process have taken effect at the target process.

Example The use of lock synchronization for the iterative computation of a distributed data structure is illustrated in the following example which is a variation of the previous examples. Here, an exclusive lock is used to protect the RMA operations:

```
while (!converged (A)) {
   update(A);
   update_buffer(A, from_buf);
   MPI_Win_start(target_group, 0, win);
   for (i=0; i<num_neighbors; i++) {
      MPI_Win_lock(MPI_LOCK_EXCLUSIVE, neighbor[i], 0, win);
      MPI_Put(&from_buf[i], size[i], MPI_INT, neighbor[i], to
      _disp[i],
              size[i], MPI_INT, win);
      MPI_Win_unlock(neighbor[i], win);
   }
}
```

5.5 Exercises for Chap. 5

Exercise 5.1 Consider the following incomplete piece of an MPI program:

```
int rank, p, size=8;
int left, right;
char send_buffer1[8], recv_buffer1[8];
char send_buffer2[8], recv_buffer2[8];
...
MPI_Comm_rank(MPI_COMM_WORLD, 8 rank);
MPI_Comm_size(MPI_COMM_WORLD, & p);
left = (rank-1 + p) % p;
right = (rank+1) % p;
...
MPI_Send(send_buffer1, size, MPI_CHAR, left, ...);
MPI_Recv(recv_buffer1, size, MPI_CHAR, right, ...);

MPI_Send(send_buffer2, size, MPI_CHAR, right, ...);
MPI_Recv(recv_buffer2, size, MPI_CHAR, left, ...);
...
```

(a) In the program, the processors are arranged in a logical ring and each processor should exchange its name with its neighbor to the left and its neighbor to the right. Assign a unique name to each MPI process and fill out the missing pieces of the program such that each process prints its own name as well as its neighbors' names.
(b) In the given program piece, the `MPI_Send()` and `MPI_Recv()` operations are arranged such that depending on the implementation a deadlock can occur. Describe how a deadlock may occur.
(c) Change the program such that no deadlock is possible by arranging the order of the `MPI_Send()` and `MPI_Recv()` operations appropriately.
(d) Change the program such that `MPI_Sendrecv()` is used to avoid deadlocks.
(e) Change the program such that `MPI_Isend()` and `MPI_Irecv()` are used.

Exercise 5.2 Consider the MPI program in Fig. 5.3 for the collection of distributed data block with point-to-point messages. The program assumes that all data blocks have the same size `blocksize`. Generalize the program such that each process can contribute a data block of a size different from the data blocks of the other processes. To do so, assume that each process has a local variable which specifies the size of its data block.

(*Hint*: First make the size of each data block available to each process in a pre-collection phase with a similar communication pattern as in Fig. 5.3 and then perform the actual collection of the data blocks.)

Exercise 5.3 Modify the program from the previous exercise for the collection of data blocks of different sizes such that no pre-collection phase is used. Instead, use `MPI_Get_count()` to determine the size of the data block received in each step. Compare the resulting execution time with the execution time of the program

from the previous exercise for different data block sizes and different numbers of processors. Which of the programs is faster?

Exercise 5.4 Consider the program `Gather_ring()` from Fig. 5.3. As described in the text, this program does not avoid deadlocks if the runtime system does not use internal system buffers. Change the program such that deadlocks are avoided in any case by arranging the order of the `MPI_Send()` and `MPI_Recv()` operations appropriately.

Exercise 5.5 The program in Fig. 5.3 arranges the processors logically in a ring to perform the collection. Modify the program such that the processors are logically arranged in a logical two-dimensional torus network. For simplicity, assume that all data blocks have the same size. Develop a mechanism with which each processor can determine its predecessor and successor in x and y directions. Perform the collection of the data blocks in two phases, the first phase with communication in x direction, the second phase with communication in y direction.

In both directions, communication in different rows or columns of the processor torus can be performed concurrently. For the communication in y direction, each process distributes all blocks that it has collected in the x direction phase. Use the normal blocking send and receive operations for the communication. Compare the resulting execution time with the execution time of the ring implementation from Fig. 5.3 for different data block sizes and different numbers of processors. Which of the programs is faster?

Exercise 5.6 Modify the program from the previous exercise such that non-blocking communication operations are used.

Exercise 5.7 Consider the parallel computation of a matrix–vector multiplication $A \cdot b$ using a distribution of the scalar products based on a rowwise distribution of A, see Fig. 3.10, p. 127 for a sketch of a parallel pseudo program. Transform this program into a running MPI program. Select the MPI communication operations for the multi-broadcast operations appropriately.

Exercise 5.8 Similar to the preceding exercise, consider a matrix–vector multiplication using a distribution of the linear combinations based on a columnwise distribution of the matrix. Transform the pseudo program from Fig. 3.12, p. 129 to a running MPI program. Use appropriate MPI operations for the single-accumulation and single-broadcast operations. Compare the execution time with the execution time of the MPI program from the preceding exercise for different sizes of the matrix.

Exercise 5.9 For a broadcast operation a root process sends the same data block to all other processes. Implement a broadcast operation by using point-to-point send and receive operations (`MPI_Send()` and `MPI_Recv()`) such that the same effect as `MPI_Bcast()` is obtained. For the processes, use a logical ring arrangement similar to Fig. 5.3.

Exercise 5.10 Modify the program from the previous exercise such that two other logical arrangements are used for the processes: a two-dimensional mesh and a

three-dimensional hypercube. Measure the execution time of the three different versions (ring, mesh, hypercube) for eight processors for different sizes of the data block and make a comparison by drawing a diagram. Use `MPI_Wtime()` for the timing.

Exercise 5.11 Consider the construction of conflict-free spanning trees in a d-dimensional hypercube network for the implementation of a multi-broadcast operation, see Sect. 4.3.2, p. 177, and Fig. 4.6. For $d = 3$, $d = 4$, and $d = 5$ write an MPI program with 8, 16, and 32 processes, respectively, that uses these spanning trees for a multi-broadcast operation.

(a) Implement the multi-broadcast by concurrent single-to-single transfers along the spanning trees and measure the resulting execution time for different message sizes.
(b) Implement the multi-broadcast by using multiple broadcast operations where each broadcast operation is implemented by single-to-single transfers along the usual spanning trees for hypercube networks as defined in p. 174, see Fig. 4.4. These spanning trees do not avoid conflicts in the network. Measure the resulting execution time for different message sizes and compare them with the execution times from (a).
(c) Compare the execution times from (a) and (b) with the execution time of an `MPI_Allgather()` operation to perform the same communication.

Exercise 5.12 For a global exchange operation, each process provides a potentially different block of data for each other process, see pp. 122 and 225 for a detailed explanation. Implement a global exchange operation by using point-to-point send and receive operations (`MPI_Send()` and `MPI_Recv()`) such that the same effect as `MPI_Alltoall()` is obtained. For the processes, use a logical ring arrangement similar to Fig. 5.3.

Exercise 5.13 Modify the program `Gather_ring()` from Fig. 5.3 such that synchronous send operations (`MPI_Send()` and `MPI_Recv()`) are used. Compare the resulting execution time with the execution time obtained for the standard send and receive operations from Fig. 5.3.

Exercise 5.14 Repeat the previous exercise with buffered send operations.

Exercise 5.15 Modify the program `Gather_ring()` from Fig. 5.3 such that the MPI operation `MPI_Test()` is used instead of `MPI_Wait()`. When a non-blocking receive operation is found by `MPI_Test()` to be completed, the process sends the received data block to the next process.

Exercise 5.16 Write an MPI program which implements a broadcast operation with `MPI_Send()` and `MPI_Recv()` operations. The program should use $n = 2^k$ processes which should logically be arranged as a hypercube network. Based on this arrangement the program should define a spanning tree in the network with root 0, see Fig. 3.8 and p. 123, and should use this spanning tree to transfer a message

stepwise from the root along the tree edges up to the leaves. Each node in the tree receives the message from its parent node and forwards it to its child nodes. Measure the resulting runtime for different message sizes up to 1 MB for different numbers of processors using `MPI_Wtime()` and compare the execution times with the execution times of `MPI_Bcast()` performing the same operation.

Exercise 5.17 The execution time of point-to-point communication operations between two processors can normally be described by a linear function of the form

$$t_{s2s}(m) = \tau + t_c \cdot m,$$

where m is the size of the message; τ is a startup time, which is independent of the message size; and t_c is the inverse of the network bandwidth. Verify this function by measuring the time for a ping-pong message transmission where process A sends a message to process B, and B sends the same message back to A. Use different message sizes and draw a diagram which shows the dependence of the communication time on the message size. Determine the size of τ and t_c on your parallel computer.

Exercise 5.18 Write an MPI program which arranges 24 processes in a (periodic) Cartesian grid structure of dimension $2 \times 3 \times 4$ using `MPI_Cart_create()`. Each process should determine and print the process rank of its two neighbors in x, y, and z directions.

For each of the three sub-grids in y-direction, a communicator should be defined. This communicator should then be used to determine the maximum rank of the processes in the sub-grid by using an appropriate `MPI_Reduce()` operation. This maximum rank should be printed out.

Exercise 5.19 Write an MPI program which arranges the MPI processes in a two-dimensional torus of size $\sqrt{p} \times \sqrt{p}$ where p is the number of processes. Each process exchanges its rank with its two neighbors in x and y dimensions. For the exchange, one-sided communication operations should be used. Implement three different schemes for the exchange with the following one-sided communication operations:

(a) global synchronization with `MPI_Win_fence()`;
(b) loose synchronization by using `MPI_Win_start()`, `MPI_Win_post()`, `MPI_Win_complete()`, and `MPI_Win_wait()`;
(c) lock synchronization with `MPI_Win_lock()` and `MPI_Win_unlock()`.

Test your program for $p = 16$ processors, i.e., for a 4×4 torus network.