Chapter 9

Distributed-memory parallel programming with MPI

Ever since parallel computers hit the HPC market, there was an intense discussion about what should be an appropriate programming model for them. The use of explicit *message passing* (MP), i.e., communication between processes, is surely the most tedious and complicated but also the most flexible parallelization method. Parallel computer vendors recognized the wish for efficient message-passing facilities, and provided proprietary, i.e., nonportable libraries up until the early 1990s. At that point in time it was clear that a joint standardization effort was required to enable scientific users to write parallel programs that were easily portable between platforms. The result of this effort was MPI, the Message Passing Interface. Today, the MPI standard is supported by several free and commercial implementations [W125, W126, W127], and has been extended several times. It contains not only communication routines, but also facilities for efficient parallel I/O (if supported by the underlying hardware). An MPI library is regarded as a necessary ingredient in any HPC system installation, and numerous types of interconnect are supported.

The current MPI standard in version 2.2 (to which we always refer in this book) defines over 500 functions, and it is beyond the scope of this book to even try to cover them all. In this chapter we will concentrate on the important concepts of message passing and MPI in particular, and provide some knowledge that will enable the reader to consult more advanced textbooks [P13, P14] or the standard document itself [W128, P15].

9.1 Message passing

Message passing is required if a parallel computer is of the distributed-memory type, i.e., if there is no way for one processor to directly access the address space of another. However, it can also be regarded as a *programming model* and used on shared-memory or hybrid systems as well (see Chapter 4 for a categorization). MPI, the nowadays dominating message-passing standard, conforms to the following rules:

- The same program runs on all processes (Single Program Multiple Data, or *SPMD*). This is no restriction compared to the more general *MPMD* (Multiple Program Multiple Data) model as all processes taking part in a parallel calcu-

lation can be distinguished by a unique identifier called *rank* (see below). Most modern MPI implementations allow starting different binaries in different processes, however. An MPMD-style message passing library is *PVM*, the Parallel Virtual Machine [P16]. Since it has waned in importance in recent years, it will not be covered here.

- The program is written in a sequential language like Fortran, C or C++. Data exchange, i.e., sending and receiving of messages, is done via calls to an appropriate library.

- All variables in a process are local to this process. There is no concept of shared memory.

One should add that message passing is not the only possible programming paradigm for distributed-memory machines. Specialized languages like High Performance Fortran (HPF), Co-Array Fortran (CAF) [P17], Unified Parallel C (UPC) [P18], etc., have been created with support for distributed-memory parallelization built in, but they have not developed a broad user community and it is as yet unclear whether those approaches can match the efficiency of MPI.

In a message passing program, messages carry data between processes. Those processes could be running on separate compute nodes, or different cores inside a node, or even on the same processor core, time-sharing its resources. A message can be as simple as a single item (like a DP word) or even a complicated structure, perhaps scattered all over the address space. For a message to be transmitted in an orderly manner, some parameters have to be fixed in advance:

- Which process is sending the message?

- Where is the data on the sending process?

- What kind of data is being sent?

- How much data is there?

- Which process is going to receive the message?

- Where should the data be left on the receiving process?

- What amount of data is the receiving process prepared to accept?

All MPI calls that actually transfer data have to specify those parameters in some way. Note that above parameters strictly relate to point-to-point communication, where there is always exactly one sender and one receiver. As we will see, MPI supports much more than just sending a single message between two processes; there is a similar set of parameters for those more complex cases as well.

MPI is a very broad standard with a huge number of library routines. Fortunately, most applications merely require less than a dozen of those.

Listing 9.1: A very simple, fully functional "Hello World" MPI program in Fortran 90.

```
1  program mpitest
2
3  use MPI
4
5  integer :: rank, size, ierror
6
7  call MPI_Init(ierror)
8  call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
9  call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
10
11 write(*,*) 'Hello World, I am ',rank,' of ',size
12
13 call MPI_Finalize(ierror)
14 end
```


9.2 A short introduction to MPI

9.2.1 A simple example

MPI is always available as a library. In order to compile and link an MPI program, compilers and linkers need options that specify where include files (i.e., C headers and Fortran modules) and libraries can be found. As there is considerable variation in those locations among installations, most MPI implementations provide compiler wrapper scripts (often called `mpicc`, `mpif77`, etc.), which supply the required options automatically but otherwise behave like "normal" compilers. Note that the way that MPI programs should be compiled and started is not fixed by the standard, so please consult the system documentation by all means.

Listing 9.1 shows a simple "Hello World"-type MPI program in Fortran 90. (See Listing 9.2 for a C version. We will mostly stick to the Fortran MPI bindings, and only describe the differences to C where appropriate. Although there are C++ bindings defined by the standard, they are of limited usefulness and will thus not be covered here. In fact, they are deprecated as of MPI 2.2.) In line 3, the `MPI` module is loaded, which provides required globals and definitions (the preprocessor is used to read in the `mpi.h` header in C; there is an equivalent header file for Fortran 77 called `mpif.h`). All Fortran MPI calls take an `INTENT(OUT)` argument, here called `ierror`, which transports information about the success of the MPI operation to the user code, a value of `MPI_SUCCESS` meaning that there were no errors. In C, the return code is used for that, and the `ierror` argument does not exist. Since failure resiliency is not built into the MPI standard today and checkpoint/restart features are usually implemented by the user code anyway, the error code is rarely used at all in practice.

The first statement, apart from variable declarations, in any MPI code should be

Listing 9.2: A very simple, fully functional "Hello World" MPI program in C.

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  int main(int argc, char** argv) {
5    int rank, size;
6
7    MPI_Init(&argc, &argv);
8    MPI_Comm_size(MPI_COMM_WORLD, &size);
9    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10
11   printf("Hello World, I am %d of %d\n", rank, size);
12
13   MPI_Finalize();
14   return 0;
15 }
```


a call to `MPI_Init()`. This initializes the parallel environment (line 7). If thread parallelism of any kind is used together with MPI, calling `MPI_Init()` is not sufficient. See Section 11 for details.

The MPI bindings for the C language follow the case-sensitive name pattern `MPI_Xxxxx...`, while Fortran is case-insensitive, of course. In contrast to Fortran, the C binding for `MPI_Init()` takes pointers to the `main()` function's arguments so that the library can evaluate and remove any additional command line arguments that may have been added by the MPI startup process.

Upon initialization, MPI sets up the so-called *world communicator*, which is called `MPI_COMM_WORLD`. A communicator defines a group of MPI processes that can be referred to by a communicator *handle*. The `MPI_COMM_WORLD` handle describes all processes that have been started as part of the parallel program. If required, other communicators can be defined as subsets of `MPI_COMM_WORLD`. Nearly all MPI calls require a communicator as an argument.

The calls to `MPI_Comm_size()` and `MPI_Comm_rank()` in lines 8 and 9 serve to determine the number of processes (`size`) in the parallel program and the unique identifier (`rank`) of the calling process, respectively. Note that the C bindings require output arguments (like `rank` and `size` above) to be specified as pointers. The ranks in a communicator, in this case `MPI_COMM_WORLD`, are consecutive, starting from zero. In line 13, the parallel program is shut down by a call to `MPI_Finalize()`. Note that no MPI process except rank 0 is guaranteed to execute any code beyond `MPI_Finalize()`.

In order to compile and run the source code in Listing 9.1, a "common" implementation may require the following steps:

```
1 $ mpif90 -O3 -o hello.exe hello.F90
2 $ mpirun -np 4 ./hello.exe
```

This would compile the code and start it with four processes. Be aware that pro-

MPI type	Fortran type
MPI_CHAR	CHARACTER(1)
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_BYTE	N/A

Table 9.1: Standard MPI data types for Fortran.

cessors may have to be allocated from some resource management (batch) system before parallel programs can be launched. How exactly MPI processes are started is entirely up to the implementation. Ideally, the start mechanism uses the resource manager's infrastructure (e.g., daemons running on all nodes) to launch processes. The same is true for process-to-core affinity; if the MPI implementation provides no direct facilities for affinity control, the methods described in Appendix A may be employed.

The output of this program could look as follows:

```
1 Hello World, I am 3 of 4
2 Hello World, I am 0 of 4
3 Hello World, I am 2 of 4
4 Hello World, I am 1 of 4
```

Although the `stdout` and `stderr` streams of MPI programs are usually redirected to the terminal where the program was started, the order in which outputs from different ranks will arrive there is undefined if the ordering is not enforced by other means.

9.2.2 Messages and point-to-point communication

The "Hello World" example did not contain any real communication apart from starting and stopping processes. An MPI message is defined as an array of elements of a particular MPI data type. Data types can either be basic types (corresponding to the standard types that every programming language knows) or *derived types*, which must be defined by appropriate MPI calls. The reason why MPI needs to know the data types of messages is that it supports heterogeneous environments where it may be necessary to do on-the-fly data conversions. For any message transfer to proceed, the data types on sender and receiver sides must match. See Tables 9.1 and 9.2 for nonexhaustive lists of available MPI data types in Fortran and C, respectively.

If there is exactly one sender and one receiver we speak of *point-to-point communication*. Both ends are identified uniquely by their ranks. Each point-to-point message can carry an additional integer label, the so-called *tag*, which may be used to identify the type of a message, and which must match on both ends. It may carry

MPI type	C type
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	N/A

Table 9.2: A selection of the standard MPI data types for C. Unsigned variants exist where applicable.

any accompanying information, or just be set to some constant value if it is not needed. The basic MPI function to send a message from one process to another is `MPI_Send()`:

```
1 <type> buf(*)
2 integer :: count, datatype, dest, tag, comm, ierror
3 call MPI_Send(buf,      ! message buffer
4               count,    ! # of items
5               datatype, ! MPI data type
6               dest,     ! destination rank
7               tag,      ! message tag (additional label)
8               comm,     ! communicator
9               ierror)   ! return value
```

The data type of the message buffer may vary; the MPI interfaces and prototypes declared in modules and headers accommodate this.[1] A message may be received with the `MPI_Recv()` function:

```
1  <type> buf(*)
2  integer :: count, datatype, source, tag, comm,
3  integer :: status(MPI_STATUS_SIZE), ierror
4  call MPI_Recv(buf,      ! message buffer
5                count,    ! maximum # of items
6                datatype, ! MPI data type
7                source,   ! source rank
8                tag,      ! message tag (additional label)
9                comm,     ! communicator
10               status,   ! status object (MPI_Status* in C)
11               ierror)   ! return value
```

Compared with `MPI_Send()`, this function has an additional output argument, the `status` object. After `MPI_Recv()` has returned, the `status` object can be used to determine parameters that have not been fixed by the call's arguments. Primarily, this pertains to the length of the message, because the `count` parameter is

[1] While this is no problem in C/C++, where the `void*` pointer type conveniently hides any variation in the argument type, the Fortran MPI bindings are explicitly inconsistent with the language standard. However, this can be tolerated in most cases. See the standard document [P15] for details.

only a maximum value at the receiver side; the message may be shorter than `count` elements. The `MPI_Get_count()` function can retrieve the real number:

```
integer :: status(MPI_STATUS_SIZE), datatype, count, ierror
call MPI_Get_count(status,     ! status object from MPI_Recv()
                   datatype,   ! MPI data type received
                   count,      ! count (output argument)
                   ierror)     ! return value
```

However, the `status` object also serves another purpose.The `source` and `tag` arguments of `MPI_Recv()` may be equipped with the special constants ("wildcards") `MPI_ANY_SOURCE` and `MPI_ANY_TAG`, respectively. The former specifies that the message may be sent by anyone, while the latter determines that the message tag should not matter. After `MPI_Recv()` has returned, `status(MPI_SOURCE)` and `status(MPI_TAG)` contain the sender's rank and the message tag, respectively. (In C, the `status` object is of type `struct MPI_Status`, and access to source and tag information works via the "`.`" operator.)

Note that `MPI_Send()` and `MPI_Recv()` have *blocking* semantics, meaning that the buffer can be used safely after the function returns (i.e., it can be modified after `MPI_Send()` without altering any message in flight, and one can be sure that the message has been completely received after `MPI_Recv()`). This is not to be confused with *synchronous* behavior; see below for details.

Listing 9.3 shows an MPI program fragment for computing an integral over some function `f(x)` in parallel. In contrast to the OpenMP version in Listing 6.2, the distribution of work among processes must be handled manually in MPI. Each MPI process gets assigned a subinterval of the integration domain according to its rank (lines 9 and 10), and some other function `integrate()`, which may look similar to Listing 6.2, can then perform the actual integration (line 13). After that each process holds its own partial result, which should be added to get the final integral. This is done at rank 0, who executes a loop over all ranks from 1 to `size` -1 (lines 18–29), receiving the local integral from each rank in turn via `MPI_Recv()` (line 19) and accumulating the result in `res` (line 28). Each rank apart from 0 has to call `MPI_Send()` to transmit the data. Hence, there are `size` -1 send and `size` -1 matching receive operations. The data types on both sides are specified to be `MPI_DOUBLE_PRECISION`, which corresponds to the usual `double precision` type in Fortran (cf. Table 9.1). The message tag is not used here, so we set it to zero.

This simple program could be improved in several ways:

- MPI does not preserve the temporal order of messages unless they are transmitted between the same sender/receiver pair (and with the same tag). Hence, to allow the reception of partial results at rank 0 without delay due to different execution times of the `integrate()` function, it may be better to use the `MPI_ANY_SOURCE` wildcard instead of a definite source rank in line 23.

- Rank 0 does not call `MPI_Recv()` before returning from its own execution of `integrate()`. If other processes finish their tasks earlier, communication cannot proceed, and it cannot be overlapped with computation. The MPI

Listing 9.3: Program fragment for parallel integration in MPI.

```
integer, dimension(MPI_STATUS_SIZE) :: status
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya,myb)

! rank 0 collects partial results
if(rank.eq.0) then
   res=psum
   do i=1,size-1
      call MPI_Recv(tmp, &    ! receive buffer
                    1,   &    ! array length
                         ! data type
                    MPI_DOUBLE_PRECISION,&
                    i,   &    ! rank of source
                    0,   &    ! tag (unused here)
                    MPI_COMM_WORLD,& ! communicator
                    status,& ! status array (msg info)
                    ierror)
      res=res+tmp
   enddo
   write(*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
else
   call MPI_Send(psum,   &    ! send buffer
                 1,      &    ! message length
                 MPI_DOUBLE_PRECISION,&
                 0,      &    ! rank of destination
                 0,      &    ! tag (unused here)
                 MPI_COMM_WORLD,ierror)
endif
```

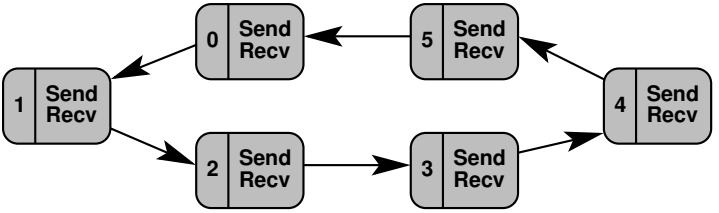

Figure 9.1: A ring shift communication pattern. If sends and receives are performed in the order shown, a deadlock can occur because `MPI_Send()` may be synchronous.

standard provides *nonblocking point-to-point communication* facilities that allow multiple outstanding receives (and sends), and even let implementations support asynchronous messages. See Section 9.2.4 for more information.

- Since the final result is needed at rank 0, this process is necessarily a communication bottleneck if the number of messages gets large. In Section 10.4.4 we will demonstrate optimizations that can significantly reduce communication overhead in those situations. Fortunately, nobody is required to write explicit code for this. In fact, the global sum is an example for a *reduction operation* and is well supported within MPI (see Section 9.2.3). Vendor implementations are assumed to provide optimized versions of such global operations.

While `MPI_Send()` is easy to use, one should be aware that the MPI standard allows for a considerable amount of freedom in its actual implementation. Internally it may work completely synchronously, meaning that the call can not return to the user code before a message transfer has at least started after a handshake with the receiver. However, it may also copy the message to an intermediate buffer and return right away, leaving the handshake and data transmission to another mechanism, like a background thread. It may even change its behavior depending on any explicit or hidden parameters. Apart from a possible performance impact, *deadlocks* may occur if the possible synchronousness of `MPI_Send()` is not taken into account. A typical communication pattern where this may become crucial is a "ring shift" (see Figure 9.1). All processes form a closed ring topology, and each *first* sends a message to its "left-hand" and *then* receives a message from its "right-hand" neighbor:

```
integer :: size, rank, left, right, ierror
integer, dimension(N) :: buf
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)
left = rank+1                ! left and right neighbors
right = rank-1
if(right<0)    right=size-1   ! close the ring
if(left>=size) left=0
call MPI_Send(buf, N, MPI_INTEGER, left, 0, &
                 MPI_COMM_WORLD,ierror)
call MPI_Recv(buf,N,MPI_INTEGER,right,0, &
                 MPI_COMM_WORLD,status,ierror)
```

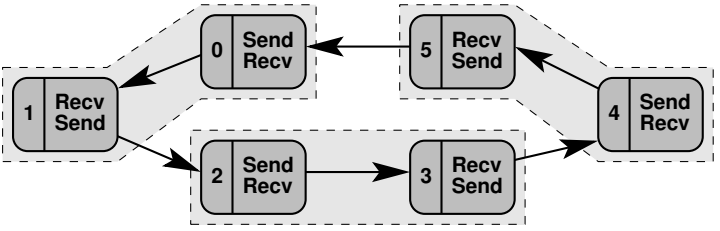

Figure 9.2: A possible solution for the deadlock problem with the ring shift: By changing the order of `MPI_Send()` and `MPI_Recv()` on all odd-numbered ranks, pairs of processes can communicate without deadlocks because there is now a matching receive for every send operation (dashed boxes).

If `MPI_Send()` is synchronous, all processes call it first and then wait forever until a matching receive gets posted. However, it may well be that the ring shift runs without problems if the messages are sufficiently short. In fact, most MPI implementations provide a (small) internal buffer for short messages and switch to synchronous mode when the buffer is full or too small (the situation is actually a little more complex in reality; see Sections 10.2 and 10.3 for details). This may lead to sporadic deadlocks, which are hard to spot. If there is some suspicion that a sporadic deadlock is triggered by `MPI_Send()` switching to synchronous mode, one can substitute all occurrences of `MPI_Send()` by `MPI_Ssend()`, which has the same interface but is synchronous by definition.

A simple solution to this deadlock problem is to interchange the `MPI_Send()` and `MPI_Recv()` calls on, e.g., all odd-numbered processes, so that there is a matching receive for every send executed (see Figure 9.2). Lines 9–12 in the code above should thus be replaced by:

```
1  if(MOD(rank,2)/=0) then
2    call MPI_Recv(buf,N,MPI_INTEGER,right,0, &     ! odd rank
3                  MPI_COMM_WORLD,status,ierror)
4    call MPI_Send(buf, N, MPI_INTEGER, left, 0, &
5                  MPI_COMM_WORLD,ierror)
6  else
7    call MPI_Send(buf, N, MPI_INTEGER, left, 0, & ! even rank
8                  MPI_COMM_WORLD,ierror)
9    call MPI_Recv(buf,N,MPI_INTEGER,right,0, &
10                 MPI_COMM_WORLD,status,ierror)
11 endif
```

After the messages sent by the even ranks have been transmitted, the remaining send/receive pairs can be matched as well. This solution does not exploit the full bandwidth of a nonblocking network, however, because only half the possible communication links can be active at any time (at least if `MPI_Send()` is really synchronous). A better alternative is the use of nonblocking communication. See Section 9.2.4 for more information, and Problem 9.1 for some more aspects of the ring shift pattern.

Since ring shifts and similar patterns are so ubiquitous, MPI has some direct support for them even with blocking communication. The `MPI_Sendrecv()` and `MPI_Sendrecv_replace()` routines combine the standard send and receive in one call, the latter using a single communication buffer in which the received message overwrites the data sent. Both routines are guaranteed to not be subject to the deadlock effects that occur with separate send and receive.

Finally we should add that there is also a blocking send routine that is guaranteed to return to the user code, regardless of the state of the receiver (`MPI_Bsend()`). However, the user must explicitly provide sufficient buffer space at the sender. It is rarely employed in practice because nonblocking communication is much easier to use (see Section 9.2.4).

9.2.3 Collective communication

The accumulation of partial results as shown above is an example for a *reduction* operation, performed on all processes in the communicator. Reductions have been introduced already with OpenMP (see Section 6.1.5), where they have the same purpose. MPI, too, has mechanisms that make reductions much simpler and in most cases more efficient than looping over all ranks and collecting results. Since a reduction is a procedure which all ranks in a communicator participate in, it belongs to the so-called *collective*, or *global communication* operations in MPI. Collective communication, as opposed to point-to-point communication, requires that every rank calls the same routine, so it is impossible for a point-to-point message sent via, e.g., `MPI_Send()`, to match a receive that was initiated using a collective call.

The simplest collective in MPI, and one that does not actually perform any real data transfer, is the barrier:

```
integer :: comm, ierror
call MPI_Barrier(comm,     ! communicator
                 ierror)   ! return value
```

The barrier *synchronizes* the members of the communicator, i.e., all processes must call it before they are allowed to return to the user code. Although frequently used by beginners, the importance of the barrier in MPI is generally overrated, because other MPI routines allow for implicit or explicit synchronization with finer control. It is sometimes used, though, for debugging or profiling.

A more useful collective is the *broadcast*. It sends a message from one process (the "root") to all others in the communicator:

```
<type> buf(*)
integer :: count, datatype, root, comm, ierror
call MPI_Bcast(buffer,     ! send/receive buffer
               count,      ! message length
               datatype,   ! MPI data type
               root,       ! rank of root process
               comm,       ! communicator
               ierror)     ! return value
```

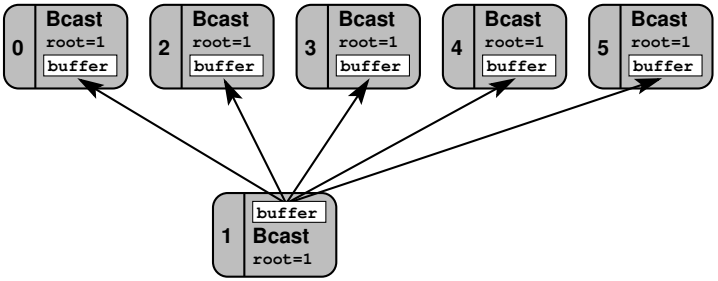

Figure 9.3: An MPI broadcast: The "root" process (rank 1 in this example) sends the same message to all others. Every rank in the communicator must call `MPI_Bcast()` with the same `root` argument.

The concept of a "root" rank, at which some general data source or sink is located, is common to many collective routines. Although rank 0 is a natural choice for "root," it is in no way different from other ranks. The `buffer` argument to `MPI_Bcast()` is a send buffer on the root and a receive buffer on any other process (see Figure 9.3). As already mentioned, every process in the communicator must call the routine, and of course the `root` argument to all those calls must be the same. A broadcast is needed whenever one rank has information that it must share with all others; e.g., there may be one process that performs some initialization phase after the program has started, like reading parameter files or command line options. This data can then be communicated to everyone else via `MPI_Bcast()`.

There are a number of more advanced collective calls that are concerned with global data distribution: `MPI_Gather()` collects the send buffer contents of all processes and concatenates them in rank order into the receive buffer of the root process. `MPI_Scatter()` does the reverse, distributing equal-sized chunks of the root's send buffer. Both exist in variants (with a "v" appended to their names) that support arbitrary per-rank chunk sizes. `MPI_Allgather()` is a combination of `MPI_Gather()` and `MPI_Bcast()`. See Table 9.3 for more examples.

Coming back to the integration example above, we had stated that there is a more effective method to perform the global reduction. This is the `MPI_Reduce()` function:

```
1  <type> sendbuf(*), recvbuf(*)
2  integer :: count, datatype, op, root, comm, ierror
3  call MPI_Reduce(sendbuf,     ! send buffer
4                  recvbuf,     ! receive buffer
5                  count,       ! number of elements
6                  datatype,    ! MPI data type
7                  op,          ! MPI reduction operator
8                  root,        ! root rank
9                  comm,        ! communicator
10                 ierror)      ! return value
```

`MPI_Reduce()` combines the contents of the `sendbuf` array on all processes,

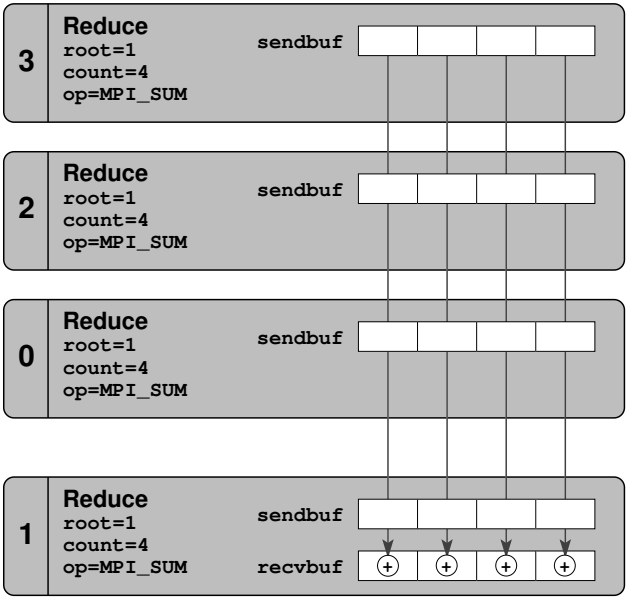

Figure 9.4: A reduction on an array of length `count` (a sum in this example) is performed by `MPI_Reduce()`. Every process must provide a send buffer. The receive buffer argument is only used on the root process. The local copy on root can be prevented by specifying `MPI_IN_PLACE` instead of a send buffer address.

element-wise, using an operator encoded by the `op` argument, and stores the result in `recvbuf` on root (see Figure 9.4). There are twelve predefined operators, the most important being `MPI_MAX`, `MPI_MIN`, `MPI_SUM` and `MPI_PROD`, which implement the global maximum, minimum, sum, and product, respectively. User-defined operators are also supported.

Now it is clear that the whole `if ...else ...endif` construct between lines 16 and 39 in Listing 9.3 (apart from printing the result in line 30) could have been written as follows:

```
1 call MPI_Reduce(psum, &           ! send buffer (partial result)
2                 res,  &           ! recv buffer (final result @ root)
3                 1, &              ! array length
4                 MPI_DOUBLE_PRECISION, &
5                 MPI_SUM, &        ! type of operation
6                 0, &              ! root (accumulate result there)
7                 MPI_COMM_WORLD,ierror)
```

Although a receive buffer (the `res` variable here) must be specified on all ranks, it is only relevant (and used) on root. Note that `MPI_Reduce()` in its plain form requires separate send and receive buffers on the root process. If allowed by the program semantics, the local accumulation on root can be simplified by setting the `sendbuf` argument to the special constant `MPI_IN_PLACE`. `recvbuf` is then used as the send buffer and gets overwritten with the global result. This can be good for performance if `count` is large and the additional copy operation leads to significant overhead. The behavior of the call on all nonroot processes is unchanged.

There are a few more global operations related to `MPI_Reduce()` worth noting.

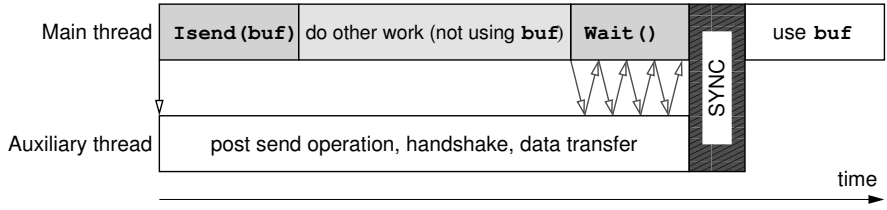

Figure 9.5: Abstract timeline view of a nonblocking send (`MPI_Isend()`). Whether there is actually an auxiliary thread is not specified by the standard; the whole data transfer may take place during `MPI_Wait()` or any other MPI function.

For example, `MPI_Allreduce()` is a fusion of a reduction with a broadcast, and `MPI_Reduce_scatter()` combines `MPI_Reduce()` with `MPI_Scatter()`.

Note that collectives are not required to, but still may synchronize all processes (the barrier is synchronizing by definition, of course). They are thus prone to similar deadlock hazards as blocking point-to-point communication (see above). This means, e.g., that collectives must be executed by all processes in the same order. See the MPI standard document [P15] for examples.

In general it is a good idea to prefer collectives over point-to-point constructs or combinations of simpler collectives that "emulate" the same semantics (see also Figures 10.15 and 10.16 and the corresponding discussion in Section 10.4.4). Good MPI implementations are optimized for data flow on collective communication and (should) also have some knowledge about network topology built in.

9.2.4 Nonblocking point-to-point communication

All MPI functionalities described so far have the property that the call returns to the user program only after the message transfer has progressed far enough so that the send/receive buffer can be used without problems. This means that, received data has arrived completely and sent data has left the buffer so that it can be safely modified without inadvertently changing the message. In MPI terminology, this is called *blocking communication*. Although collective communication in MPI is always blocking in the current MPI standard (version 2.2 at the time of writing), point-to-point communication can be performed with *nonblocking* semantics as well. A nonblocking point-to-point call merely initiates a message transmission and returns very quickly to the user code. In an efficient implementation, waiting for data to arrive and the actual data transfer occur in the background, leaving resources free for computation. Synchronization is ruled out (see Figure 9.5 for a possible timeline of events for the nonblocking `MPI_Isend()` call). In other words, nonblocking MPI is a way in which communication may be overlapped with computation if implemented efficiently. The message buffer must not be used as long as the user program has not been notified that it is safe to do so (which can be checked by suitable MPI calls). Nonblocking and blocking MPI calls are mutually compatible, i.e., a message sent via a blocking send can be matched by a nonblocking receive.

The most important nonblocking send is `MPI_Isend()`:

```
<type> buf(*)
integer :: count, datatype, dest, tag, comm, request, ierror
call MPI_Isend(buf,         ! message buffer
               count,       ! # of items
               datatype,    ! MPI data type
               dest,        ! destination rank
               tag,         ! message tag
               comm,        ! communicator
               request,     ! request handle (MPI_Request* in C)
               ierror)      ! return value
```

As opposed to the blocking send (see page 208), `MPI_Isend()` has an additional output argument, the *request handle*. It serves as an identifier by which the program can later refer to the "pending" communication request (in C, it is of type `struct MPI_Request`). Correspondingly, `MPI_Irecv()` initiates a nonblocking receive:

```
<type> buf(*)
integer :: count, datatype, source, tag, comm, request, ierror
call MPI_Irecv(buf,         ! message buffer
               count,       ! # of items
               datatype,    ! MPI data type
               source,      ! source rank
               tag,         ! message tag
               comm,        ! communicator
               request,     ! request handle
               ierror)      ! return value
```

The `status` object known from `MPI_Recv()` is missing here, because it is not needed; after all, no actual communication has taken place when the call returns to the user code. Checking a pending communication for completion can be done via the `MPI_Test()` and `MPI_Wait()` functions. The former only tests for completion and returns a flag, while the latter blocks until the buffer can be used:

```
logical :: flag
integer :: request, status(MPI_STATUS_SIZE), ierror
call MPI_Test(request,      ! pending request handle
              flag,         ! true if request complete (int* in C)
              status,       ! status object
              ierror)       ! return value
call MPI_Wait(request,      ! pending request handle
              status,       ! status object
              ierror)       ! return value
```

The `status` object contains useful information only if the pending communication is a completed receive (i.e., in the case of `MPI_Test()` the value of `flag` must be `true`). In this sense, the sequence

```
call MPI_Irecv(buf, count, datatype, source, tag, comm, &
               request, ierror)
call MPI_Wait(request, status, ierror)
```

is completely equivalent to a standard `MPI_Recv()`.

A potential problem with nonblocking MPI is that a compiler has no way to know that `MPI_Wait()` can (and usually will) modify the contents of `buf`. Hence, in the following code, the compiler may consider it legal to move the final statement in line 3 before the call to `MPI_Wait()`:

```
1 call MPI_Irecv(buf, ..., request, ...)
2 call MPI_Wait(request, status, ...)
3 buf(1) = buf(1) + 1
```

This will certainly lead to a race condition and the contents of `buf` may be wrong. The inherent connection between the `MPI_Irecv()` and `MPI_Wait()` calls, mediated by the request handle, is invisible to the compiler, and the fact that `buf` is not contained in the argument list of `MPI_Wait()` is sufficient to assume that the code modification is legal. A simple way to avoid this situation is to put the variable (or buffer) into a `COMMON` block, so that potentially all subroutines may modify it. See the MPI standard [P15] for alternatives.

Multiple requests can be pending at any time, which is another great advantage of nonblocking communication. Sometimes a group of requests belongs together in some respect, and one would like to check not one, but any one, any number, or all of them for completion. This can be done with suitable calls that are parameterized with an array of handles. As an example we choose the `MPI_Waitall()` routine:

```
1 integer :: count, requests(*)
2 integer :: statuses(MPI_STATUS_SIZE,*), ierror
3 call MPI_Waitall(count,          ! number of requests
4                  requests,       ! request handle array
5                  statuses,       ! statuses array (MPI_Status* in C)
6                  ierror)         ! return value
```

This call returns only after all the pending requests have been completed. The status objects are available in `array_of_statuses(:,:)`.

The integration example in Listing 9.3 can make use of nonblocking communication by overlapping the local interval integration on rank 0 with receiving results from the other ranks. Unfortunately, collectives cannot be used here because there are no nonblocking collectives in MPI. Listing 9.4 shows a possible solution. The reduction operation has to be done manually (lines 33–35), as in the original code. Array sizes for the status and request arrays are not known at compile time, hence those must be allocated dynamically, as well as separate receive buffers for all ranks except 0 (lines 11–13). The collection of partial results is performed with a single `MPI_Waitall()` in line 32. Nothing needs to be changed on the nonroot ranks; `MPI_Send()` is sufficient to communicate the partial results (line 39).

Nonblocking communication provides an obvious way to overlap communication, i.e., overhead, with useful work. The possible performance advantage, however, depends on many factors, and may even be nonexistent (see Section 10.4.3 for a discussion). But even if there is no real overlap, multiple outstanding nonblocking requests may improve performance because the MPI library can decide which of them gets serviced first.

Listing 9.4: Program fragment for parallel integration in MPI, using nonblocking point-to-point communication.

```
integer, allocatable, dimension(:,:) :: statuses
integer, allocatable, dimension(:) :: requests
double precision, allocatable, dimension(:) :: tmp
call MPI_Comm_size(MPI_COMM_WORLD, size, ierror)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierror)

! integration limits
a=0.d0 ; b=2.d0 ; res=0.d0

if(rank.eq.0) then
  allocate(statuses(MPI_STATUS_SIZE, size-1))
  allocate(requests(size-1))
  allocate(tmp(size-1))
! pre-post nonblocking receives
  do i=1,size-1
    call MPI_Irecv(tmp(i), 1, MPI_DOUBLE_PRECISION, &
                   i, 0, MPI_COMM_WORLD,       &
                   requests(i), ierror)
  enddo
endif

! limits for "me"
mya=a+rank*(b-a)/size
myb=mya+(b-a)/size

! integrate f(x) over my own chunk - actual work
psum = integrate(mya,myb)

! rank 0 collects partial results
if(rank.eq.0) then
   res=psum
   call MPI_Waitall(size-1, requests, statuses, ierror)
   do i=1,size-1
      res=res+tmp(i)
   enddo
   write (*,*) 'Result: ',res
! ranks != 0 send their results to rank 0
else
   call MPI_Send(psum, 1, &
                 MPI_DOUBLE_PRECISION, 0, 0, &
                 MPI_COMM_WORLD,ierror)
endif
```

	Point-to-point	**Collective**

Blocking	`MPI_Send()`
	`MPI_Ssend()`
	`MPI_Bsend()`
	`MPI_Recv()`
		`MPI_Barrier()`
		`MPI_Bcast()`
		`MPI_Scatter()`/
		`MPI_Gather()`
		`MPI_Reduce()`
		`MPI_Reduce_scatter()`
		`MPI_Allreduce()`

Nonblocking	`MPI_Isend()`
	`MPI_Irecv()`
	`MPI_Wait()`/`MPI_Test()`
	`MPI_Waitany()`/
	`MPI_Testany()`
	`MPI_Waitsome()`/
	`MPI_Testsome()`
	`MPI_Waitall()`/
	`MPI_Testall()`
		N/A

Table 9.3: MPI's communication modes and a nonexhaustive overview of the corresponding subroutines.

Table 9.3 gives an overview of available communication modes in MPI, and the most important library functions.

9.2.5 Virtual topologies

We have outlined the principles of domain decomposition as an example for data parallelism in Section 5.2.1. Using the MPI functions covered so far, it is entirely possible to implement domain decomposition on distributed-memory parallel computers. However, setting up the process grid and keeping track of which ranks have to exchange halo data is nontrivial. Since domain decomposition is such an important pattern, MPI contains some functionality to support this recurring task in the form of *virtual topologies*. These provide a convenient process naming scheme, which fits the required communication pattern. Moreover, they potentially allow the MPI library to optimize communications by employing knowledge about network topology. Although arbitrary graph topologies can be described with MPI, we restrict ourselves to Cartesian topologies here.

As an example, assume there is a simulation that handles a big double precision array `P(1:3000,1:4000)` containing $3000 \times 4000 = 1.2 \times 10^7$ words. The simulation runs on $3 \times 4 = 12$ processes, across which the array is distributed "naturally," i.e., each process holds a chunk of size 1000×1000. Figure 9.6 shows a possible Cartesian topology that reflects this situation: Each process can either be identified by its rank or its Cartesian coordinates. It has a number of neighbors, which depends on

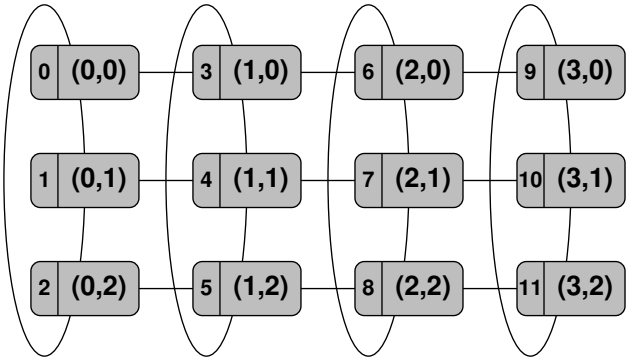

Figure 9.6: Two-dimensional Cartesian topology: 12 processes form a 3×4 grid, which is periodic in the second dimension but not in the first. The mapping between MPI ranks and Cartesian coordinates is shown.

the grid's dimensionality. In our example, the number of dimensions is two, which leads to at most four neighbors per process. Boundary conditions on each dimension can be closed (cyclic) or open.

MPI can help with establishing the mapping between ranks and Cartesian coordinates in the process grid. First of all, a new communicator must be defined to which the chosen topology is "attached." This is done via the `MPI_Cart_create()` function:

```
integer :: comm_old, ndims, dims(*), comm_cart, ierror
logical :: periods(*), reorder
call MPI_Cart_create(comm_old,      ! input communicator
                     ndims,         ! number of dimensions
                     dims,          ! # of processes in each dim.
                     periods,       ! periodicity per dimension
                     reorder,       ! true = allow rank reordering
                     comm_cart,     ! new cartesian communicator
                     ierror)        ! return value
```

It generates a new, "Cartesian" communicator `comm_cart`, which can be used later to refer to the topology. The `periods` array specifies which Cartesian directions are periodic, and the `reorder` parameter allows, if true, for rank reordering so that the rank of a process in communicators `comm_old` and `comm_cart` may differ. The MPI library may choose a different ordering by using its knowledge about network topology and anticipating that next-neighbor communication is often dominant in a Cartesian topology. Of course, communication between any two processes is still allowed in the Cartesian communicator.

There is no mention of the actual problem size (3000×4000) because it is entirely the user's job to care for data distribution. All MPI can do is keep track of the topology information. For the topology shown in Figure 9.6, `MPI_Cart_create()` could be called as follows:

```
call MPI_Cart_create(MPI_COMM_WORLD,          ! standard communicator
                     2,                       ! two dimensions
                     (/ 4, 3 /),              ! 4x3 grid
                     (/ .false., .true. /),   ! open/periodic
                     .false.,                 ! no rank reordering
```

```
6                       comm_cart,              ! Cartesian communicator
7                       ierror)
```

If the number of MPI processes is given, finding an "optimal" extension of the grid in each direction (as needed in the `dims` argument to `MPI_Cart_create()`) requires some arithmetic, which can be offloaded to the `MPI_Dims_create()` function:

```
1 integer :: nnodes, ndims, dims(*), ierror
2 call MPI_Dims_create(nnodes, ! number of nodes in grid
3                       ndims,  ! number of Cartesian dimensions
4                       dims,   ! input: /=0 # nodes fixed in this dir.
5                               !        ==0 # calculate # nodes
6                               ! output: number of nodes each dir.
7                       ierror)
```

The `dims` array is both an input and an output parameter: Each entry in `dims` corresponds to a Cartesian dimension. A zero entry denotes a dimension for which `MPI_Dims_create()` should calculate the number of processes, and a nonzero entry specifies a fixed number of processes. Under those constraints, the function determines a balanced distribution, with all `ndims` extensions as close together as possible. This is optimal in terms of communication overhead only if the overall problem grid is cubic. If this is not the case, the user is responsible for setting appropriate constraints, since MPI has no way to know the grid geometry.

Two service functions are responsible for the translation between Cartesian process coordinates and an MPI rank. `MPI_Cart_coords()` calculates the Cartesian coordinates for a given rank:

```
1 integer :: comm_cart, rank, maxdims, coords(*), ierror
2 call MPI_Cart_coords(comm_cart,     ! Cartesian communicator
3                       rank,         ! process rank in comm_cart
4                       maxdims,      ! length of coords array
5                       coords,       ! return Cartesian coordinates
6                       ierror)
```

(If rank reordering was allowed when producing `comm_cart`, a process should always obtain its rank by calling `MPI_Comm_rank(comm_cart,...)` first.) The output array `coords` contains the Cartesian coordinates belonging to the process of the specified rank.

This mapping function is needed whenever one deals with domain decomposition. The first information a process will obtain from MPI is its rank in the Cartesian communicator. `MPI_Cart_coords()` is then required to determine the coordinates so the process can calculate, e.g., which subdomain it should work on. See Section 9.3 below for an example.

The reverse mapping, i.e., from Cartesian coordinates to an MPI rank, is performed by `MPI_Cart_rank()`:

```
1 integer :: comm_cart, coords(*), rank, ierror
2 call MPI_Cart_rank(comm_cart,     ! Cartesian communicator
3                     coords,       ! Cartesian process coordinates
```

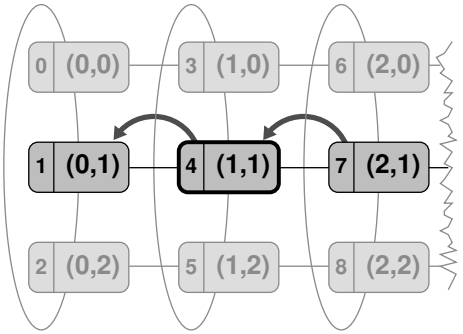

Figure 9.7: Example for the result of `MPI_Cart_shift()` on a part of the Cartesian topology from Figure 9.6. Executed by rank 4 with `direction`=0 and `disp`=−1, the function returns `rank_source`=7 and `rank_dest`=1.

```
4                       rank,          ! return process rank in comm_cart
5                       ierror)
```

Again, the return value in `rank` is only valid in the `comm_cart` communicator if reordering was allowed.

A regular task with domain decomposition is to find out who the next neighbors of a certain process are along a certain Cartesian dimension. In principle one could start from its Cartesian coordinates, offset one of them by one (accounting for open or closed boundary conditions) and map the result back to an MPI rank via `MPI_Cart_rank()`. The `MPI_Cart_shift()` function does it all in one step:

```
1 integer :: comm_cart, direction, disp, rank_source,
2 integer :: rank_dest, ierror
3 call MPI_Cart_shift(comm_cart,    ! Cartesian communicator
4                     direction,    ! direction of shift (0..ndims-1)
5                     disp,         ! displacement
6                     rank_source, ! return source rank
7                     rank_dest,    ! return destination rank
8                     ierror)
```

The `direction` parameter specifies within which Cartesian dimension the shift should be performed, and `disp` determines the distance and direction (positive or negative). `rank_source` and `rank_dest` return the "neighboring" ranks, according to the other arguments. Figure 9.7 shows an example for a shift along the negative first dimension, executed on rank 4 in the topology given in Figure 9.6. The source and destination neighbors are 7 and 1, respectively. If a neighbor does not exist because it would extend beyond the grid's boundary in a noncyclic dimension, the rank will be returned as the special value `MPI_PROC_NULL`. Using `MPI_PROC_NULL` as a source or destination rank in any communication call is allowed and will effectively render the call a dummy statement — no actual communication will take place. This can simplify programming because the boundaries of the grid do not have to be treated in a special way (see Section 9.3 for an example).

9.3 Example: MPI parallelization of a Jacobi solver

As a nontrivial example for virtual topologies and other MPI functionalities we use a simple Jacobi solver (see Sections 3.3 and 6.2) in three dimensions. As opposed to parallelization with OpenMP, where inserting a couple of directives was sufficient, MPI parallelization by domain decomposition is much more complex.

9.3.1 MPI implementation

Although the basic algorithm was described in Section 5.2.1, we require some more detail now. An annotated flowchart is shown in Figure 9.8. The central part is still the sweep over all subdomains (step 3); this is where the computational effort goes. However, each subdomain is handled by a different MPI process, which poses two difficulties:

1. The convergence criterion is based on the maximum deviation between the current and the next time step across all grid cells. This value can be easily obtained for each subdomain separately, but a reduction is required to get a global maximum.

2. In order for the sweep over a subdomain to yield the correct results, appropriate boundary conditions must be implemented. This is no problem for cells that actually neighbor real boundaries, but for cells adjacent to a domain cut, the boundary condition changes from sweep to sweep: It is formed by the cells that lie right across the cut, and those are not available directly because they are owned by another MPI process. (With OpenMP, all data is always visible by all threads, making access across "chunk boundaries" trivial.)

The first problem can be solved directly by an `MPI_Allreduce()` call after every process has obtained the maximum deviation `maxdelta` in its own domain (step 4 in Figure 9.8).

As for the second problem, so-called *ghost* or *halo layers* are used to store copies of the boundary information from neighboring domains. Since only a single ghost layer per subdomain is required per domain cut, no additional memory must be allocated because a boundary layer is needed anyway. (We will see below, however, that some supplementary arrays may be necessary for technical reasons.) Before a process sweeps over its subdomain, which involves updating the $T = 1$ array from the $T = 0$ data, the $T = 0$ boundary values from its neighbors are obtained via MPI and stored in the ghost cells (step 2 in Figure 9.8). In the following we will outline the central parts of the Fortran implementation of this algorithm. The full code can be downloaded from the book's Web site.[2] For clarity, we will declare important variables with each code snippet.

[2] http://www.hpc.rrze.uni-erlangen.de/HPC4SE/

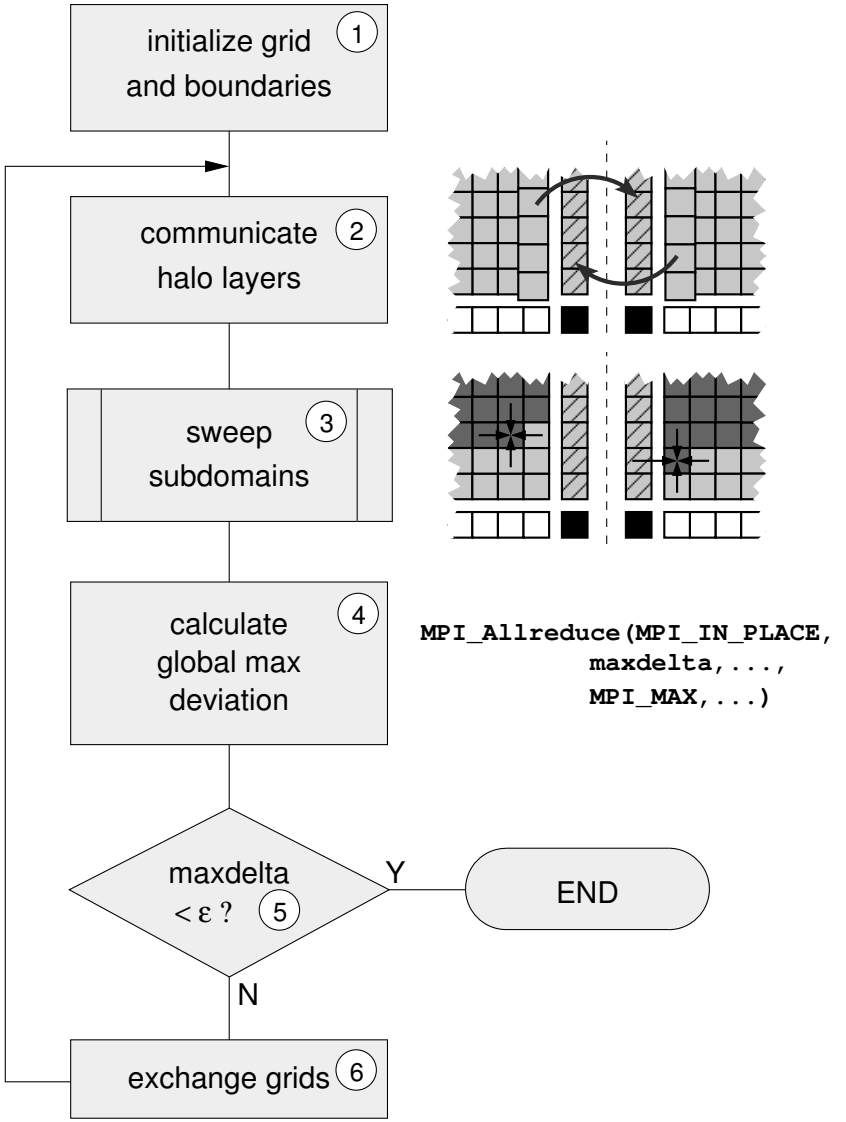

Figure 9.8: Flowchart for distributed-memory parallelization of the Jacobi algorithm. Hatched cells are ghost layers, dark cells are already updated in the $T = 1$ grid, and light-colored cells denote $T = 0$ data. White cells are real boundaries of the overall grid, whereas black cells are unused.

First the required parameters are read by rank zero from standard input (line 10 in the following listing): problem size (`spat_dim`), possible presets for number of processes (`proc_dim`), and periodicity (`pbc_check`), each for all dimensions.

```
1  logical, dimension(1:3) :: pbc_check
2  integer, dimension(1:3) :: spat_dim, proc_dim
3
4  call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
5  call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)
6
7  if(myid.eq.0) then
8    write(*,*) ' spat_dim , proc_dim, PBC ? '
9    do i=1,3
10     read(*,*) spat_dim(i), proc_dim(i), pbc_check(i)
11   enddo
12 endif
13
14 call MPI_Bcast(spat_dim , 3, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
15 call MPI_Bcast(proc_dim , 3, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
16 call MPI_Bcast(pbc_check, 3, MPI_LOGICAL, 0, MPI_COMM_WORLD, ierr)
```

Although many MPI implementations have options to allow the standard input of rank zero to be seen by all processes, a portable MPI program cannot rely on this feature, and must broadcast the data (lines 14–16). After that, the Cartesian topology can be set up using `MPI_Dims_create()` and `MPI_Cart_create()`:

```
1  call MPI_Dims_create(numprocs, 3, proc_dim, ierr)
2
3  if(myid.eq.0) write(*,'(a,3(i3,x))') 'Grid: ', &
4      (proc_dim(i),i=1,3)
5
6  l_reorder = .true.
7  call MPI_Cart_create(MPI_COMM_WORLD, 3, proc_dim, pbc_check, &
8      l_reorder, GRID_COMM_WORLD, ierr)
9
10 if(GRID_COMM_WORLD .eq. MPI_COMM_NULL) goto 999
11
12 call MPI_Comm_rank(GRID_COMM_WORLD, myid_grid, ierr)
13 call MPI_Comm_size(GRID_COMM_WORLD, nump_grid, ierr)
```

Since rank reordering is allowed (line 6), the process rank must be obtained again using `MPI_Comm_rank()` (line 12). Moreover, the new Cartesian communicator `GRID_COMM_WORLD` may be of smaller size than `MPI_COMM_WORLD`. The "surplus" processes then receive a communicator value of `MPI_COMM_NULL`, and are sent into a barrier to wait for the whole parallel program to complete (line 10).

Now that the topology has been created, the local subdomains can be set up, including memory allocation:

```
1  integer, dimension(1:3) :: loca_dim, mycoord
2
3  call MPI_Cart_coords(GRID_COMM_WORLD, myid_grid, 3,
4      mycoord,ierr)
5
```

```
6  do i=1,3
7    loca_dim(i) = spat_dim(i)/proc_dim(i)
8    if(mycoord(i) < mod(spat_dim(i),proc_dim(i))) then
9      local_dim(i) = loca_dim(i)+1
10   endif
11 enddo
12
13 iStart = 0 ; iEnd = loca_dim(3)+1
14 jStart = 0 ; jEnd = loca_dim(2)+1
15 kStart = 0 ; kEnd = loca_dim(1)+1
16
17 allocate(phi(iStart:iEnd, jStart:jEnd, kStart:kEnd,0:1))
```

Array `mycoord` is used to store a process' Cartesian coordinates as acquired from `MPI_Cart_coords()` in line 3. Array `loca_dim` holds the extensions of a process' subdomain in the three dimensions. These numbers are calculated in lines 6–11. Memory allocation takes place in line 17, allowing for an additional layer in all directions, which is used for fixed boundaries or halo as needed. For brevity, we are omitting the initialization of the array and its outer grid boundaries here.

Point-to-point communication as used for the ghost layer exchange requires consecutive message buffers. (Actually, the use of derived MPI data types would be an option here, but this would go beyond the scope of this introduction.) However, only those boundary cells that are consecutive in the inner (*i*) dimension are also consecutive in memory. Whole layers in the *i*-*j*, *i*-*k*, and *j*-*k* planes are never consecutive, so an intermediate buffer must be used to gather boundary data to be communicated to a neighbor's ghost layer. Sending each consecutive chunk as a separate message is out of the question, since this approach would flood the network with short messages, and latency has to be paid for every request (see Chapter 10 for more information on optimizing MPI communication).

We use two intermediate buffers per process, one for sending and one for receiving. Since the amount of halo data can be different along different Cartesian directions, the size of the intermediate buffer must be chosen to accommodate the largest possible halo:

```
1  integer, dimension(1:3) :: totmsgsize
2
3  ! j-k plane
4  totmsgsize(3) = loca_dim(1)*loca_dim(2)
5  MaxBufLen=max(MaxBufLen,totmsgsize(3))
6  ! i-k plane
7  totmsgsize(2) = loca_dim(1)*loca_dim(3)
8  MaxBufLen=max(MaxBufLen,totmsgsize(2))
9  ! i-j plane
10 totmsgsize(1) = loca_dim(2)*loca_dim(3)
11 MaxBufLen=max(MaxBufLen,totmsgsize(1))
12
13 allocate(fieldSend(1:MaxBufLen))
14 allocate(fieldRecv(1:MaxBufLen))
```

At the same time, the halo sizes for the three directions are stored in the integer array `totmsgsize`.

Now we can start implementing the main iteration loop, whose length is the maximum number of iterations (sweeps), `ITERMAX`:

```
1  t0=0 ; t1=1
2  tag = 0
3  do iter = 1, ITERMAX
4    do disp = -1, 1, 2
5      do dir = 1, 3
6
7        call MPI_Cart_shift(GRID_COMM_WORLD, (dir-1), &
8                            disp, source, dest, ierr)
9
10       if(source /= MPI_PROC_NULL) then
11         call MPI_Irecv(fieldRecv(1), totmsgsize(dir), &
12              MPI_DOUBLE_PRECISION, source, &
13              tag, GRID_COMM_WORLD, req(1), ierr)
14       endif   ! source exists
15
16       if(dest /= MPI_PROC_NULL) then
17         call CopySendBuf(phi(iStart, jStart, kStart, t0), &
18                          iStart, iEnd, jStart, jEnd, kStart, kEnd, &
19                          disp, dir, fieldSend, MaxBufLen)
20
21         call MPI_Send(fieldSend(1), totmsgsize(dir), &
22                       MPI_DOUBLE_PRECISION, dest, tag, &
23                       GRID_COMM_WORLD, ierr)
24       endif   ! destination exists
25
26       if(source /= MPI_PROC_NULL) then
27         call MPI_Wait(req, status, ierr)
28
29         call CopyRecvBuf(phi(iStart, jStart, kStart, t0), &
30                          iStart, iEnd, jStart, jEnd, kStart, kEnd, &
31                          disp, dir, fieldRecv, MaxBufLen)
32       endif   ! source exists
33
34     enddo   ! dir
35   enddo   ! disp
36
37   call Jacobi_sweep(loca_dim(1), loca_dim(2), loca_dim(3), &
38                     phi(iStart, jStart, kStart, 0), t0, t1, &
39                     maxdelta)
40
41   call MPI_Allreduce(MPI_IN_PLACE, maxdelta, 1, &
42                      MPI_DOUBLE_PRECISION, &
43                      MPI_MAX, 0, GRID_COMM_WORLD, ierr)
44   if(maxdelta<eps) exit
45   tmp=t0; t0=t1; t1=tmp
46 enddo   ! iter
47
48 999 continue
```

Halos are exchanged in six steps, i.e., separately per positive and negative Cartesian direction. This is parameterized by the loop variables `disp` and `dir`. In line 7, `MPI_Cart_shift()` is used to determine the communication neighbors along the

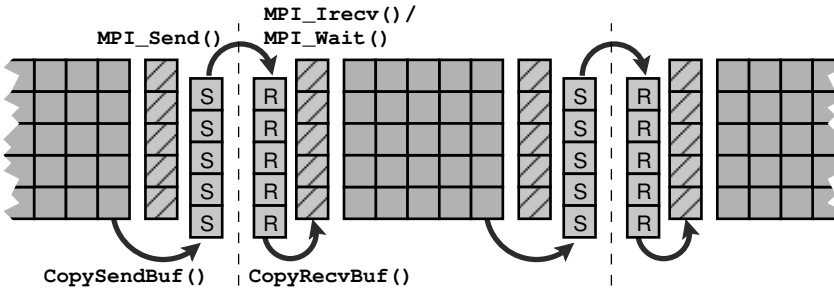

Figure 9.9: Halo communication for the Jacobi solver (illustrated in two dimensions here) along one of the coordinate directions. Hatched cells are ghost layers, and cells labeled "R" ("S") belong to the intermediate receive (send) buffer. The latter is being reused for all other directions. Note that halos are always provided for the grid that gets read (not written) in the upcoming sweep. Fixed boundary cells are omitted for clarity.

current direction (`source` and `dest`). If a subdomain is located at a grid boundary, and periodic boundary conditions are not in place, the neighbor will be reported to have rank `MPI_PROC_NULL`. MPI calls using this rank as source or destination will return immediately. However, as the copying of halo data to and from the intermediate buffers should be avoided for efficiency in this case, we also mask out any MPI calls, keeping overhead to a minimum (lines 10, 16, and 26).

The communication pattern along a direction is actually a ring shift (or a linear shift in case of open boundary conditions). The problems inherent to a ring shift with blocking point-to-point communication were discussed in Section 9.2.2. To avoid deadlocks, and possibly utilize the available network bandwidth to full extent, a nonblocking receive is initiated before anything else (line 11). This data transfer can potentially overlap with the subsequent halo copy to the intermediate send buffer, done by the `CopySendBuf()` subroutine (line 17). After sending the halo data (line 21) and waiting for completion of the previous nonblocking receive (line 27), `CopyRecvBuf()` finally copies the received halo data to the boundary layer (line 29), which completes the communication cycle in one particular direction. Figure 9.9 again illustrates this chain of events.

After the six halo shifts, the boundaries of the current grid `phi(:,:,:,t0)` are up to date, and a Jacobi sweep over the local subdomain is performed, which updates `phi(:,:,:,t1)` from `phi(:,:,:,t0)` (line 37). The corresponding subroutine `Jacobi_sweep()` returns the maximum deviation between the previous and the current time step for the subdomain (see Listing 6.5 for a possible implementation in 2D). A subsequent `MPI_Allreduce()` (line 41) calculates the global maximum and makes it available on all processes, so that the decision whether to leave the iteration loop because convergence has been reached (line 44) can be made on all ranks without further communication.

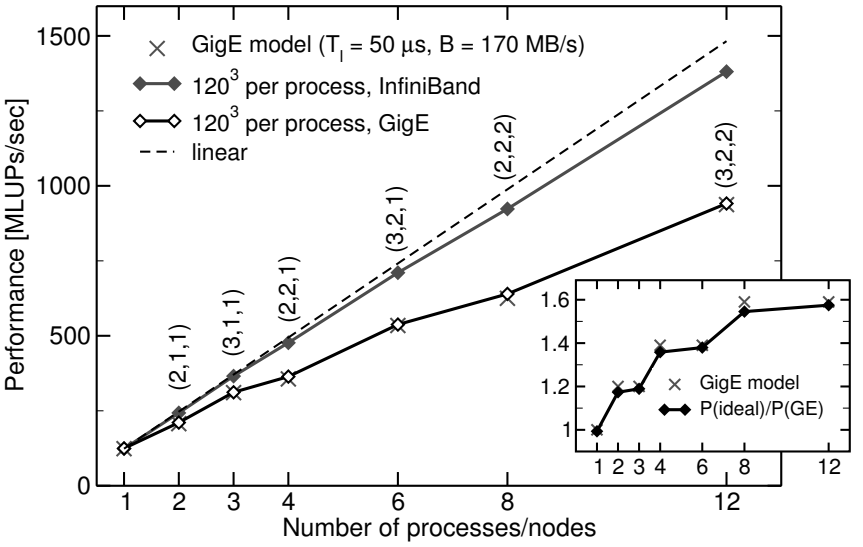

Figure 9.10: Main panel: Weak scaling of the MPI-parallel 3D Jacobi code with problem size 120^3 per process on InfiniBand vs. Gigabit Ethernet networks. Only one process per node was used. The domain decomposition topology (number of processes in each Cartesian direction) is indicated. The weak scaling performance model (crosses) can reproduce the GigE data well. Inset: Ratio between ideally scaled performance and Gigabit Ethernet performance vs. process count. (Single-socket cluster based on Intel Xeon 3070 at 2.66 GHz, Intel MPI 3.2.)

9.3.2 Performance properties

The performance characteristics of the MPI-parallel Jacobi solver are typical for many domain decomposition codes. We distinguish between weak and strong scaling scenarios, as they show quite different features. All benchmarks were performed with two different interconnect networks (Intel MPI Version 3.2 over DDR InfiniBand vs. Gigabit Ethernet) on a commodity cluster with single-socket nodes based on Intel Xeon 3070 processors at 2.66 GHz. A single process per node was used throughout in order to get a simple scaling baseline and minimize intranode effects.

Weak scaling

In our performance models in Section 5.3.6 we have assumed that 3D domain decomposition at weak scaling has constant communication overhead. This is, however, not always true because a subdomain that is located at a grid boundary may have fewer halo faces to communicate. Fortunately, due to the inherent synchronization between subdomains, the overall runtime of the parallel program is dominated by the slowest process, which is the one with the largest number of halo faces if all processes work on equally-sized subdomains. Hence, we can expect reasonably linear scaling behavior even on slow (but nonblocking) networks, once there is at least one subdomain that is surrounded by other subdomains in all Cartesian directions.

The weak scaling data for a constant subdomain size of 120^3 shown in Figure 9.10 substantiates this conjecture:

Scalability on the InfiniBand network is close to perfect. For Gigabit Ethernet, communication still costs about 40% of overall runtime at large node counts, but this fraction gets much smaller when running on fewer nodes. In fact, the performance graph shows a peculiar "jagged" structure, with slight breakdowns at 4 and 8 processes. These breakdowns originate from fundamental changes in the communication characteristics, which occur when the number of subdomains in any coordinate direction changes from one to anything greater than one. At that point, internode communication along this axis sets in: Due to the periodic boundary conditions, every process always communicates in all directions, but if there is only one process in a certain direction, it exchanges halo data only with itself, using (fast) shared memory. The inset in Figure 9.10 indicates the ratio between ideal scaling and Gigabit Ethernet performance data. Clearly this ratio gets larger whenever a new direction gets cut. This happens at the decompositions (2,1,1), (2,2,1), and (2,2,2), respectively, belonging to node counts of 2, 4, and 8. Between these points, the ratio is roughly constant, and since there are only three Cartesian directions, it can be expected to not exceed a value of ≈ 1.6 even for very large node counts, assuming that the network is nonblocking. The same behavior can be observed with the InfiniBand data, but the effect is much less pronounced due to the much larger ($\times 10$) bandwidth and lower ($/20$) latency. Note that, although we use a performance metric that is only relevant in the parallel part of the program, the considerations from Section 5.3.3 about "fake" weak scalability do not apply here; the single-CPU performance is on par with the expectations from the STREAM benchmark (see Section 3.3).

The communication model described above is actually good enough for a quantitative description. We start with the assumption that the basic performance characteristics of a point-to-point message transfer can be described by a simple latency/bandwidth model along the lines of Figure 4.10. However, since sending and receiving halo data on each MPI process can overlap for each of the six coordinate directions, we must include a maximum bandwidth number for full-duplex data transfer over a single link. The (half-duplex) PingPong benchmark is not accurate enough to get a decent estimate for full-duplex bandwidth, even though most networks (including Ethernet) claim to support full-duplex. The Gigabit Ethernet network used for the Jacobi benchmark can deliver about 111 MBytes/sec for half-duplex and 170 MBytes/sec for full-duplex communication, at a latency of 50 μs.

The subdomain size is the same regardless of the number of processes, so the raw compute time T_s for all cell updates in a Jacobi sweep is also constant. Communication time T_c, however, depends on the number and size of domain cuts that lead to internode communication, and we are assuming that copying to/from intermediate buffers and communication of a process with itself come at no cost. Performance on $N = N_x N_y N_z$ processes for a particular overall problem size of $L^3 N$ grid points (using cubic subdomains of size L^3) is thus

$$P(L,\vec{N}) = \frac{L^3 N}{T_\mathrm{s}(L) + T_\mathrm{c}(L,\vec{N})}\ , \tag{9.1}$$

where

$$T_{\mathrm{c}}(L,\vec{N}) = \frac{c(L,\vec{N})}{B} + kT_{\ell} \; . \tag{9.2}$$

Here, $c(L,\vec{N})$ is the maximum bidirectional data volume transferred over a node's network link, B is the full-duplex bandwidth, and k is the largest number (over all subdomains) of coordinate directions in which the number of processes is greater than one. $c(L,\vec{N})$ can be derived from the Cartesian decomposition:

$$c(L,\vec{N}) = L^2 \cdot k \cdot 2 \cdot 8 \tag{9.3}$$

For $L = 120$ this leads to the following numbers:

N	(N_z,N_y,N_x)	k	$c(L,\vec{N})$ [MB]	$P(L,\vec{N})$ [MLUPs/sec]	$\frac{NP_1(L)}{P(L,\vec{N})}$
1	(1,1,1)	0	0.000	124	1.00
2	(2,1,1)	2	0.461	207	1.20
3	(3,1,1)	2	0.461	310	1.20
4	(2,2,1)	4	0.922	356	1.39
6	(3,2,1)	4	0.922	534	1.39
8	(2,2,2)	6	1.382	625	1.59
12	(3,2,2)	6	1.382	938	1.59

$P_1(L)$ is the measured single-processor performance for a domain of size L^3. The prediction for $P(L,\vec{N})$ can be seen in the third column, and the last column quantifies the "slowdown factor" compared to perfect scaling. Both are shown for comparison with the measured data in the main panel and inset of Figure 9.10, respectively. The model is clearly able to describe the performance features of weak scaling well, which is an indication that our general concept of the communication vs. computation "workflow" was correct. Note that we have deliberately chosen a small problem size to emphasize the role of communication, but the influence of latency is still minor.

Strong scaling

Figure 9.11 shows strong scaling performance data for periodic boundary conditions on two different problem sizes (120^3 vs. 480^3). There is a slight penalty for the smaller size (about 10%) even with one processor, independent of the interconnect. For InfiniBand, the performance gap between the two problem sizes can mostly be attributed to the different subdomain sizes. The influence of communication on scalability is minor on this network for the node counts considered. On Gigabit Ethernet, however, the smaller problem scales significantly worse because the ratio of halo data volume (and latency overhead) to useful work becomes so large at larger node counts that communication dominates the performance on this slow network. The typical "jagged" pattern in the scaling curve is superimposed by the communication volume changing whenever the number of processes changes. A simple predictive model as with weak scaling is not sufficient here; especially with small grids, there is

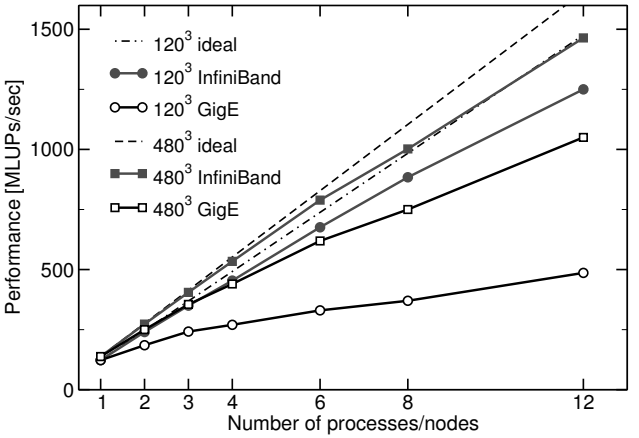

Figure 9.11: Strong scaling of the MPI-parallel 3D Jacobi code with problem size 120^3 (circles) and 480^3 (squares) on IB (filled symbols) vs. GigE (open symbols) networks. Only one process per node was used. (Same system and MPI topology as in Figure 9.10.)

a strong dependence of single-process performance on the subdomain size, and intranode communication processes (halo exchange) become important. See Problem 9.4 and Section 10.4.1 for some more discussion.

Note that this analysis must be refined when dealing with multiple MPI processes per node, as is customary with current parallel systems (see Section 4.4). Especially on fast networks, intranode communication characteristics play a central role (see Section 10.5 for details). Additionally, copying of halo data to and from intermediate buffers within a process cannot be neglected.

Problems

For solutions see page 304 *ff.*

9.1 *Shifts and deadlocks.* Does the remedy for the deadlock problem with ring shifts as shown in Figure 9.2 (exchanging send/receive order) also work if the number of processes is odd?

What happens if the chain is open, i.e., if rank 0 does not communicate with the highest-numbered rank? Does the reordering of sends and receives make a difference in this case?

9.2 *Deadlocks and nonblocking MPI.* In order to avoid deadlocks, we used nonblocking receives for halo exchange in the MPI-parallel Jacobi code (Section 9.3.1). An MPI implementation is actually not required to support overlapping of communication and computation; MPI progress, i.e., real data transfer, might happen only if MPI library code is executed. Under such conditions, is it still guaranteed that deadlocks cannot occur? Consult the MPI standard if in doubt.

9.3 *Open boundary conditions.* The performance model for weak scaling of the Jacobi code in Section 9.3.2 assumed periodic boundary conditions. How would the model change for open (Dirichlet-type) boundaries? Would there still be plateaus in the inset of Figure 9.10? What would happen when going from 12 to 16 processes? What is the minimum number of processes for which the ratio between ideal and real performance reaches its maximum?

9.4 *A performance model for strong scaling of the parallel Jacobi code.* As mentioned in Section 9.3.2, a performance model that accurately predicts the strong scaling behavior of the MPI-parallel Jacobi code is more involved than for weak scaling. Especially the dependence of the single-process performance on the subdomain size is hard to predict since it depends on many factors (pipelining effects, prefetching, spatial blocking strategy, copying to intermediate buffers, etc.). This was no problem for weak scaling because of the constant subdomain size. Nevertheless one could try to establish a partly "phenomenological" model by measuring single-process performance for all subdomain sizes that appear in the parallel run, and base a prediction for parallel performance on those baselines. What else would you consider to be required enhancements to the weak scaling model? Take into account that T_s becomes smaller and smaller as N grows, and that halo exchange is not the only internode communication that is going on.

9.5 *MPI correctness.* Is the following MPI program fragment correct? Assume that only two processes are running, and that `my_rank` contains the rank of each process.

```
1 if(my_rank.eq.0) then
2    call MPI_Bcast(buf1, count, type, 0, comm, ierr)
3    call MPI_Send(buf2, count, type, 1, tag, comm, ierr)
4 else
5    call MPI_Recv(buf2, count, type, 0, tag, comm, status, ierr)
6    call MPI_Bcast(buf1, count, type, 0, comm, ierr)
7 endif
```

(This example is taken from the MPI 2.2 standard document [P15].)