MPI Processes and Messaging

4

Chapter Summary
Distributed memory computers cannot communicate through a shared memory. Therefore, messages are used to coordinate parallel tasks that eventually run on geographically distributed but interconnected processors. Processes as well as their management and communication are well defined by a platform-independent message passing interface (MPI) specification. MPI is introduced from the practical point of view, with a set of basic operations that enable implementation of parallel programs. We will give simple example programs that will serve as an aid for a smooth start of using MPI and as motivation for developing more complex applications.

4.1 Distributed Memory Computers Can Execute in Parallel

We know from previous chapters that there are two main differences between the shared memory and distributed memory computer architectures. The first difference is in the price of communication: the time needed to exchange a certain amount of data between two or more processors is in favor of shared memory computers, as these can usually communicate much faster than the distributed memory computers. The second difference, which is in the number of processors that can cooperate efficiently, is in favor of distributed memory computers. Usually, our primary choice when computing complex tasks will be to engage a large number of fastest available processors, but the communication among them poses additional limitations. Cooperation among processors implies communication or data exchange among them. When the number of processors must be high (e.g., more than eight) to reduce the execution time, the speed of communication becomes a crucial performance factor.

There is a significant difference in the speed of data movement between two computing cores within a single multi-core computer, depending on the location of data to be communicated. This is because the data can be stored in registers, cache

R. Trobec et al., *Introduction to Parallel Computing*,
Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-98833-7_4

memory, or system memory, which can differ by up to two orders of magnitude if their access times are considered. The differences in the communication speed get even more pronounced in the interconnected computers, again by orders of magnitude, but this now depends on the technology and topology of the interconnection networks and on the geographical distance of the cooperating computers.

Taking into account the above facts, complex tasks can be executed efficiently either (i) on a small number of extremely fast computers or (ii) on a large number of potentially slower interconnected computers. In this chapter, we focus on the presentation and usage of the **Message Passing Interface (MPI)**, which enables system-independent parallel programming. The well-established MPI standard[1] includes process creation and management, language bindings for C and Fortran, point-to-point and collective communications, and group and communicator concepts. Newer MPI standards are trying to better support the scalability in future extreme-scale computing systems, because currently, the only feasible option for increasing the computing power is to increase the number of cooperating processors. Advanced topics, as one-sided communications, extended collective operations, process topologies, external interfaces, etc., are also covered by these standards, but are beyond the scope of this book.

The final goal of this chapter is to advise users how to employ the basic MPI principles in the solution of complex problems with a large number of processes that exchange application data through messages.

4.2 Programmer's View

Programmers have to be aware that the cooperation among processes implies the data exchange. The total execution time is consequently a sum of computation and communication time. Algorithms with only local communication between neighboring processors are faster and more scalable than the algorithms with the global communication among all processors. Therefore, the programmer's view of a problem that will be parallelized has to incorporate a wide number of aspects, e.g., data independency, communication type and frequency, balancing the load among processors, balancing between communication and computation, overlapping communication and computation, synchronous or asynchronous program flow, stopping criteria, and others.

Most of the above issues that are related to communication are efficiently solved by the MPI specification. Therefore, we will identify the mentioned aspects and describe efficient solutions through the standardized MPI operations. Further sections should not be considered as an MPI reference guide or MPI library implementation manual.

[1] Against potential ambiguities, some segments of text are reproduced from A Message-Passing Interface Standard (Version 3.1), © 1993, 1994, 1995, 1996, 1997, 2008, 2009, 2012, 2015, by University of Tennessee, Knoxville, Tennessee.

We will just try to rise the interest of readers, through simple and illustrative examples, and to show how some of the typical problems can be efficiently solved by the MPI methodology.

4.3 Message Passing Interface

The standardization effort of a message passing interface (MPI) library began in 90s and is one of the most successful projects of the software standardization. Its driving force was, from the beginning, a cooperation between academia and industry that has been created with the MPI standardization forum.

The MPI library interface is a specification, not an implementation. The MPI is not a language, and all MPI "**operations**" are expressed as functions, subroutines, or methods, according to the appropriate language bindings for C and Fortran, which are a part of the MPI standard. The MPI standard defines the syntax and semantics of library operations that support the message passing model, independently of program language or compiler specification.

Since the word "`PARAMETER`" is a keyword in the Fortran language, the MPI standard uses the word "**argument**" to denote the arguments to a subroutine. It is expected that C programmers will understand the word "argument", which has no specific meaning in C, as a "parameter", thus allowing to avoid unnecessary confusion for Fortran programmers.

An MPI program consists of autonomous processes that are able to execute their own code in the sense of multiple instruction multiple data (MIMD) paradigm. An MPI "**process**" can be interpreted in this sense as a program counter that addresses their program instructions in the system memory, which implies that the program codes executed by each process have not to be the same.

The processes communicate via calls to MPI communication operations, independently of operating system. The MPI can be used in a wide range of programs written in C or Fortran. Based on the MPI library specifications, several efficient MPI library implementations have been developed, either in open-source or in a proprietary domain. The success of the project is evidenced by a coherent development of the parallel software projects that are portable between different computing environments, e.g., parallel computers, clusters, and heterogeneous networks, and scalable along wide numbers of cooperating processors, from one to millions. Finally, the MPI interface is designed for end users, parallel library writers and developers of parallel software tools.

Any MPI program should have operations to initialize execution environment and to control starting and terminating procedures of all generated processes. MPI processes can be collected into groups of specific `size` that can communicate in its own environment where each message sent in a context must be received only in the same context. A **process group** and **context** together form an MPI `communicator`. A process is identified by its `rank` in the group associated with a communicator. There is a default communicator `MPI_COMM_WORLD` whose group encompasses all initial processes, and whose context is default. Two essential questions arise early in any MPI parallel program: “How many processes are participating in computation?” and “Which are their identities?” Both questions will be answered after calling two specialized MPI operations.

The basic MPI communication is characterized by two fundamental MPI operations `MPI_SEND` and `MPI_RECV` that provide sends and receives of process data, represented by numerous data types. Besides the data transfer these two operations synchronize the cooperating processes in time instants where communication has to be established, e.g., a process cannot proceed if the expected data has not arrived. Further, a sophisticated addressing is supported within a group of ranked processes that are a part of a communicator. A single program may use several communicators, which manage common or separated MPI processes. Such a concept enables to use different MPI based parallel libraries that can communicate independently, without interference, within a single parallel program.

Even that the most of parallel algorithms can be implemented by just a few MPI operations, the MPI-1 standard offers a set of more than 120 operations for elegant and efficient programming, including operations for collective and asynchronous communication in numerous topologies of interconnected computers. The MPI library is well documented from its beginning and constantly developing. The MPI-2 provides standardized process start-up, dynamic process creation and management, improved data types, one-sided communication, and versatile input/output operations. The MPI-3 standard introduces non-blocking collective communication that enables communication-computation overlapping and the MPI shared memory (SHM) model that enables efficient programming of hybrid architectures, e.g., a network of multi-core computing nodes.

Complete MPI is quite a large library with 128 MPI-1 operations, with twice as much in MPI-2 and even more in MPI-3. We will start with only six basic operations and further add a few from the complete MPI set for greater flexibility in the parallel programming. However, to fulfill the desires of this textbook one need to master just a few dozens of MPI operations that will be described in more detail in the following sections.

Very well organized documentation can be found on several web pages, for example, on the following link: http://www.mcs.anl.gov/research/projects/mpi/tutorial/mpiexmpl/contents.html with assignments, solution, program output and many useful hints and additional links. The latest MPI standard and further information about MPI are available on http://www.mpi-forum.org/.

Example 4.1 Hello World MPI program

We will proceed with a minimal MPI program in C programming language. Its implementation is shown in Listing 4.1.

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

int main(int argc, char **argv)
```

```
//int main(argc, argv)
//int  argc;
//char **argv;
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Hello world from process %d of %d processes.\n", rank, size);
  MPI_Finalize();
  return 0;
}
```

Listing 4.1 "Hello world" MPI program `MSMPIHello.ccp` in C programming syntax.

The "Hello World" has been written in C programming language; hence, the three-line preamble should be commented and replaced by `int main(int argc, char **argv)`, if C++ compiler is used. The "Hello World" code looks like a standard C code with several additional lines with `MPI_` prefix, which are calls to global MPI operations that are executed on all processes. Note that some MPI operations that will be introduced later could be local, i.e., executed on a single process.

The "Hello World" code in Listing 4.1 is the same for all processes. It has to be compiled only once to be executed on all active processes. Such a methodology could simplify the development of parallel programs. Run the program with:

```
$ mpiexec -n 3 MSMPIHello
```

from Command prompt of the host process, at the path of directory where `MSMPIHello.exe` is located. The program should output three "Hello World" messages, each with a process identification data.

All non-MPI procedures are local, e.g., `printf` in the above example. It runs on each process and prints separate "Hello World" notice. If one would prefer to have only a notice from a specific process, e.g., `0`, an extra `if(rank == 0)` statement should be inserted. Let us comment the rest of the above code:

- `#include "stdafx.h"` is needed because the MS Visual Studio compiler has been used,
- `#include <stdio.h>` is needed because of `printf`, which is used later in the program, and
- `#include "mpi.h"` provides basic MPI definition of named constants, types, and function prototypes, and must be included in any MPI program.

The above MPI program, including the definition of variables, will be executed in all active processes. The number of processes will be determined by parameter `-n` of the MPI execution utility `mpiexec`, usually provided by the MPI library implementation.

- `MPI_Init` initializes the MPI execution environment and `MPI_Finalize` exits the MPI.

- `MPI_Comm_size(MPI_COMM_WORLD, & size)` returns `size`, which is the number of started processes.
- `MPI_Comm_rank(MPI_COMM_WORLD, & rank)` that returns `rank`, i.e., an ID of each process.
- MPI operations return a status of the execution success; in C routines as the value of the function, which is not considered in the above C program, and in Fortran routines as the last argument of the function call (see Listing 4.2).

Depending on the number of processes, the `printf` function will run on each process, which will print a separate “Hello World” notice. If all processes will print the output, we expect `size` lines with “Hello World” notice, one from each process. Note that the order of the printed notices is not known in advance, because there is no guaranty about the ordering of the MPI processes. We will address this topic, in more detail, later in this chapter. Note also that in this simple example no communication between processes has been required. □

For comparison, a version of “Hello World” MPI program in Fortran programming language is given in Listing 4.2:

```
program hello_world
include ’/usr/include/mpif.h’
integer ierr, num_procs, my_id

call MPI_INIT (ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD, my_id, ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, num_procs, ierr)
print *, "Hello world from process ", my_id, " of ", num_procs
call MPI_FINALIZE (ierr)
stop
end
```

Listing 4.2 ”Hello world” MPI program `OMPIHello.f` in Fortran programming language.

Note that capitalized `MPI_` prefix is used again in the names of MPI operations, which are also capitalized in Fortran syntax, but the different header file `mpif.h` is included. MPI operations return a status of execution success, i.e., `ierr` in the case of Fortran program.

4.3.1 MPI Operation Syntax

The MPI standard is independent of specific programming languages. To stress this fact, capitalized MPI operation names will be used in the definition of MPI operations. MPI operation arguments, in a language-independent notation, are marked as:
IN—for input values that may be used by the operation, but not updated;
OUT—for output values that may be updated by the operation, but not used as input value;
INOUT—for arguments that may be used and/or updated by the MPI operation.

An argument used as IN by some processes and as OUT by other processes is also marked as INOUT, even that it is not used for input and for output in a single call.

For shorter specifications of MPI operations, the following notation is used for descriptive names of arguments:
IN arguments are in normal text, e.g., buf, sendbuf, MPI_COMM_WORLD, etc.
OUT arguments are in underlined text, e.g., rank, recbuf, etc.
INOUT arguments are in underlined italic text, e.g., *inbuf*, *request*, etc.

The examples of MPI programs, in the rest of this chapter, are given in C programming language. Below are some terms and conventions that are implemented with C program language binding:

- Function names are equal to the MPI definitions but with the MPI_ prefix and the first letter of the function name in uppercase, e.g., MPI_Finalize().
- The status of execution success of MPI operations is returned as integer return codes, e.g., ierr = MPI_Finalize(). The return code can be an error code or MPI_SUCCESS for successful competition, defined in the file mpi.h. Note that all predefined constants and types are fully capitalized.
- Operation arguments IN are usually passed by value with an exception of the send buffer, which is determined by its initial address. All OUT and INOUT arguments are passed by reference (as pointers), e.g.,
  MPI_Comm_size (MPI_COMM_WORLD, &size).

4.3.2 MPI Data Types

MPI communication operations specify the message data length in terms of number of data elements, not in terms of number of bytes. Specifying message data elements is machine independent and closer to the application level. In order to retain machine independent code, the MPI standard defines its own basic data types that can be used for the specification of message data values, and correspond to the basic data types of the host language.

As MPI does not require that communicating processes use the same representation of data, i.e., data types, it needs to keep track of possible data types through the build-in basic MPI data types. For more specific applications, MPI offers operations to construct custom data types, e.g., array of (int, float) pairs, and many other options. Even that the typecasting between a particular language and the MPI library may represent a significant overhead, the portability of MPI programs significantly benefits.

Some basic MPI data types that correspond to the adequate C or Fortran data types are listed in Table 4.1. Details on advanced structured and custom data types can be found in the before mentioned references.

The data types MPI_BYTE and MPI_PACKED do not correspond to a C or a Fortran data type. A value of type MPI_BYTE consists of a byte, i.e., 8 binary digits. A byte is uninterpreted and is different from a character. Different machines may have different representations for characters or may use more than one byte to represent

Table 4.1 Some MPI data types corresponding to C and Fortran data types

MPI data type	C data type	MPI data type	Fortran data type
MPI_INT	int	MPI_INTEGER	INTEGER
MPI_SHORT	short int	MPI_REAL	REAL
MPI_LONG	long int	MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_FLOAT	float	MPI_COMPLEX	COMPLEX
MPI_DOUBLE	double	MPI_LOGICAL	LOGICAL
MPI_CHAR	char	MPI_CHARACTER	CHARACTER
MPI_BYTE	/	MPI_BYTE	/
MPI_PACKED	/	MPI_PACKED	/

characters. On the other hand, a byte has the same binary value on all machines. If the size and representation of data are known, the fastest way is the transmission of raw data, for example, by using an elementary MPI data type MPI_BYTE.

The MPI communication operations have involved only buffers containing a continuous sequence of identical basic data types. Often, one wants to pass messages that contain values with different data types, e.g., a number of integers followed by a sequence of real numbers; or one wants to send noncontiguous data, e.g., a subblock of a matrix. The type MPI_PACKED is maintained by MPI_PACK or MPI_UNPACK operations, which enable to pack different types of data into a contiguous send buffer and to unpack it from a contiguous receive buffer.

A more efficient alternative is a usage of derived data types for construction of custom message data. The derived data types allow, in most cases, to avoid explicit packing and unpacking, which requires less memory and time. A user specifies in advance the layout of data types to be sent or received and the communication library can directly access a noncontinuous data. The simplest noncontiguous data type is the vector type, constructed with MPI_Type_vector. For example, a sender process has to communicate the main diagonal of an $N \times N$ array of integers, declared as:

```
int matrix[N][N];
```

which is stored in a row-major layout. A continuous derived data type diagonal can be constructed:

```
MPI_Datatype MPI_diagonal;
```

that specifies the main diagonal as a set of integers:

```
MPI_Type_vector (N, 1, N+1, MPI_INT, &diagonal);
```

where their count is N, block length is 1, and stride is N+1. The receiver process receives the data as a contiguous block. There are further options that enable the construction of sub-arrays, structured data, irregularly strided data, etc.

If all data of an MPI program is specified by MPI types it will support data transfer between processes on computers with different memory organization and different interpretations of elementary data items, e.g., in heterogeneous platforms. The parallel programs, designed with MPI data types, can be easily ported even between computers with unknown representations of data. Further, the custom application-oriented data types can reduce the number of memory-to-memory copies or can be tailored to a dedicated hardware for global communication.

4.3.3 MPI Error Handling

The MPI standard assumes a reliable and error-free underlying communication platform; therefore, it does not provide mechanisms for dealing with failures in the communication system. For example, a message sent is always received correctly, and the user need not check for transmission errors, time-outs, or similar. Similarly, MPI does not provide mechanisms for handling processor failures. A program error can follow an MPI operation call with incorrect arguments, e.g., non-existing destination in a send operation, exceeding available system resources, or similar.

Most of MPI operation calls return an error code that indicates the completion status of the operation. Before the error value is returned, the current MPI error handler is called, which, by default, aborts all MPI processes. However, MPI provides mechanisms for users to change this default and to handle recoverable errors. One can specify that no MPI error is fatal, and handle the returned error codes by custom error-handling routines.

4.3.4 Make Your Computer Ready for Using MPI

In order to test the presented theory, we need to install first the necessary software that will make our computer ready for running and testing MPI programs. In Appendix A of this book, readers will find short instructions for the installation of free MPI supporting software for either for Linux, macOS, or MS Windows-powered computers. Beside a compiler for selected program language, an MPI implementation of the MPI standard is needed with a method for running MPI programs. Please refer the instruction in Appendix A and run your first "Hello Word" MPI program. Then you can proceed here in order to find some further hints for running and testing simple MPI programs, either on a single multi-core computer or on a set of interconnected computers.

4.3.5 Running and Configuring MPI Processes

Any MPI library will provide you with the `mpiexec` (or `mpirun`) program that can launch one or more MPI applications on a single computer or on a set of interconnected computers (hosts). The program has many options that are standardized to some extent, but one is advised to check actual program options with `mpiexec -help`. Most common options are `—n <num_processes>`, `-host` or `-machinefile`.

An MPI program executable `MyMPIprogram.exe` can be launched on a local host and on three processes with:

```
$ mpiexec -n 3 MyMPIprogram
```

MPI will automatically distribute processes among the available cores, which can be specified by option `—cores <num_cores_per_host>` Alternatively, the program can be launched on two interconnected computers, on each with four processes, with:

```
$ mpiexec -host 2 host1 4 host2 4 MyMPIprogram
```

For more complex managing of cooperation processes, a separate configuration file can be used. The processes available for the MPI can be specified by using `-machinefile` option to `mpiexec`. With this option, a text file, e.g., `myhostsfile`, lists computers on which to launch MPI processes. The hosts are listed one per line, identified either with a computer name or with its IP address. An MPI program, e.g., `MyMPIprogram`, can be executed, for example, on three processes, with:

```
$ mpiexec -machinefile myhostsfile -n 3 MyMPIprogram
```

Single Computer

The configuration file can be used for a specification of processes on a single computer or on a set of interconnected computers. For each host, the number of processes to be used on that host can be defined by a number that follows a computer name. For example, on a computer with a single core, the following configuration file defines four processes per computing core:

```
localhost 4
```

If your computer has, for example, four computing cores, MPI processes will be distributed among the cores automatically, or in a way specified by the user in the MPI configuration file, which supports, in this case, the execution of the MPI parallel program on a single computer. The configuration file could have the following structure:

```
localhost
localhost
localhost
localhost
```

specifying that a single process will run on each computing core if mpiexec option -n 4 is used, or two processes will run on each computing core if -n 8 is used, etc. Note that there are further alternative options for configuring MPI processes that are usually described in more detail in -help options of a specific MPI implementation.

Your computer is now ready for the coding and testing more useful MPI programs that will be discussed in following sections. Before that, some further hints are given for the execution of MPI programs on a set of interconnected computers.

Interconnected Computers

If you are testing your program on a computer network you may select several computers to perform defined processes and run and test your code. The configuration file must be edited in a way that all cooperating computers are listed. Suppose that four computers will cooperate, each with two computing cores. The configuration file: myhostsfile should contain names or IP addresses of these computers, e.g.:

```
computer_name1
computer_name2
192.20.301.77
computer_name4
```

each in a separate line, and with the first name belonging to the name of the local host, i.e., the computer from which the MPI program will be started, by mpiexec.

Let us execute our MPI program MyMPIprogram on a set of computers in a network, e.g., connected with an Ethernet. Editing, compiling, and linking processes are the same as in the case of a single computer. However, the MPI executable should be available to all computers, e.g., by a manual copying of the MPI executable on the same path on all computers, or more systematically, through a shared disk.

On MS Windows, a service for managing the local MPI processes, e.g., smpd daemons should be started by smpd -d on all cooperating computers before launching MPI programs. The cooperating computers should have the same version of the MPI library installed, and the compiled MPI executable should be compatible with the computing platforms (32 or 64 bits) on all computers. The command from the master host:

```
$mpiexec -machinefile myhostsfile \\MasterHost\share\
MyMPIprog
```

will enable to run the program on a set of processes, eventually located on different computers, as has been specified in the configuration file myhostsfile.

Note also that the potential user should be granted with rights for executing the programs on selected computers. One will need a basic user account and an access to the MPI executable that must be located on the same path on all computers. In Linux, this can be accomplished automatically by placing the executable in /home/username/ directory. Finally, a method that allows automatic login, e.g., in Linux, SSH login without password, is needed, to enable automatic login between cooperating computers.

The described approach is independent on the technology of the interconnection network. The interconnected computers can be multi-core computers, computing clusters connected by Gigabit Ethernet or Infiniband, or computers in a home network connected by Wi-Fi.

4.4 Basic MPI Operations

Let us recall the presented issues in a more systematic way by a brief description of four basic MPI operations. Two trivial operations without MPI arguments will initiate and shut down the MPI environment. Next two operations will answer the questions: "How many processes will cooperate?" and "Which is my ID among them?" Note that all four operations are called from all processes of the current communicator.

4.4.1 MPI_INIT (int *argc, char ***argv)

The operation initiates an MPI library and environment. The arguments argc and argv are required in C language binding only, where they are parameters of the main C program.

4.4.2 MPI_FINALIZE ()

The operation shuts down the MPI environment. No MPI routine can be called before MPI_INIT or after MPI_FINALIZE, with one exception MPI_INITIALIZED (flag), which queries if MPI_INIT has been called.

4.4.3 MPI_COMM_SIZE (comm, size)

The operation determines the number of processes in the current communicator. The input argument comm is the handle of communicator; the output argument size returned by the operation MPI_COMM_SIZE is the number of processes in the group of comm. If comm is MPI_COMM_WORLD, then it represents the number of all active MPI processes.

4.4.4 MPI_COMM_RANK (comm, rank)

The operation determines the identifier of the current process within a communicator. The input argument comm is the handle of the communicator; the output argument rank is an ID of the process from comm, which is in the range from 0 to size-1.

In the following sections, some of the frequently used communication MPI operations are described briefly. There is no intention to provide an MPI user manual in its complete version, instead, this short description should be just a first motivation for beginners to write an MPI program that will effectively harness his computer, and to further explore the beauty and usefulness of the MPI approach.

4.5 Process-to-Process Communication

We know from previous chapters that a traditional process is associated with a private program counter of its private address space. Processes may have multiple program threads, associated with separate program counters, which share a single process' address space. The message passing model formalizes the communication between processes that have separate address spaces. The process-to-process communication has to implement two essential tasks: data movement and synchronization of processes; therefore, it requires cooperation of sender and receiver processes. Consequently, every send operation expects a pairing/matching receive operation. The cooperation is not always apparent in the program, which may hinder the understanding of the MPI code.

A schematic presentation of a communication between sender `Process_0` and receiver `Process_1` is shown in Fig. 4.1. In this case, optional intermediate message buffers are used in order to enable sender `Process_0` to continue immediately after it initiates the send operation. However, `Process_0` will have to wait on the return from the previous call, before it can send a new message. On the receiver side, `Process_1` can do some useful work instead of idling while waiting on the matching message reception. It is a communication system that must ensure that the message will be reliably transferred between both processes. If the processes have been created on a single computer, the actual communication will be probably implemented through a shared memory. If the processes reside on two distant computers, then the actual communication might be performed through an existing interconnection network using, e.g., TCP/IP communication protocol.

Although that blocking send/receive operations enable a simple way for synchronization of processes, they could introduce unnecessary delays in cases where sender and receiver do not reach communication point at the same real time. For example, if `Process_0` issues a send call significantly before the matching receives call in `Process_1`, `Process_0` will start waiting to the actual message data transfer. In the same way, processes' idling can happen if a process that produces many messages is much faster than the consumer process. Message buffering may alleviate the idling to some extent, but if the amount of data exceeds the capacity of the message buffer, which can always happen, `Process_0` will be blocked again.

The next concern of the blocking communication are deadlocks. For example, if `Process_0` and `Process_1` initiate their send calls in the same time, they will be blocked forever by waiting matching receive calls. Fortunately, there are several

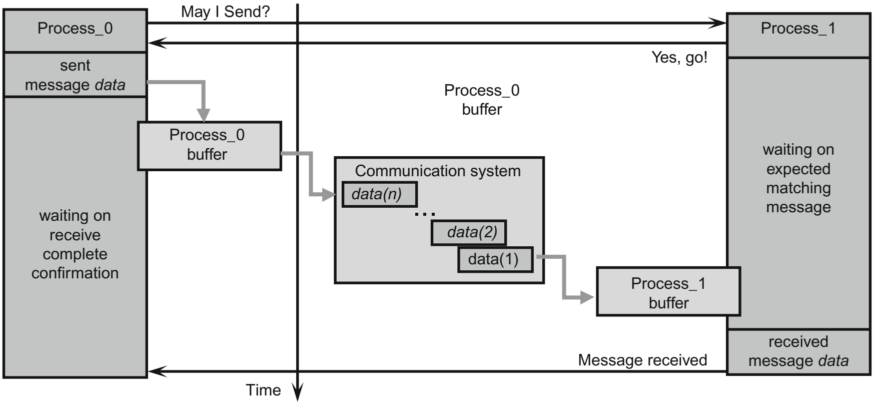


Fig. 4.1 Communication between two processes awakes both of them while transferring data from sender `Process_0` to receiver `Process_1`, possibly with a set of shorter sub-messages

ways for alleviating such situations, which will be described in more detail near the end of Sect. 4.7.

Before an actual process-to-process transfer of data happens, several issues have to be specified, e.g., how will message data be described, how processes will be identified, and how the receiver recognizes/screens messages, when the operations will complete. The `MPI_SEND` and `MPI_RECV` operations are responsible for the implementation of the above issues.

4.5.1 `MPI_SEND (buf, count, datatype, dest, tag, comm)`

The operation, invoked by a blocking call `MPI_SEND` in the sender process `source`, will not complete until there is a matching `MPI_RECV` in receiver process `dest`, identified by a corresponding `rank`. The `MPI_RECV` will empty the input send buffer `buf` of matching `MPI_SEND`. The `MPI_SEND` will return when the message data has been delivered to the communication system and the send buffer `buf` of the sender process `source` can be reused. The send buffer is specified by the following arguments: `buf` - pointer to the send buffer, `count` - number of data items, and `datatype` - type of data items. The receiver process is addressed by an envelope that consists of arguments `dest`, which is the rank of receiver process within all processes in the communicator `comm`, and of a message `tag`.

The **message tags** provide a mechanism for distinguishing between different messages for the same receiver process identified by destination `rank`. The `tag` is an integer in the range `[0, UB]` where `UB`, defined in `mpi.h`, can be found by querying the predefined constant `MPI_TAG_UB`. When a sender process has to send more separate messages to a receiver process, the sender process will distinguish them by using `tags`, which will allow receiver process to efficiently screening its messages. For example, if a receiver process has to distinguish between messages from a single source process, a message tag will serve an additional means for messages differentiation. MPI_ANY_TAG is a constant predefined in mpi.h, which can be considered as a "wild-card", where all tags will be accepted.

4.5.2 MPI_RECV (buf, count, datatype, source, tag, comm, status)

This operation waits until the communication system delivers a message with matching datatype, source, tag, and comm. Messages are screened at the receiving part based on specific source, which is a rank of the sender process within communicator comm, or not screened at all on source by equating it with MPI_ANY_SOURCE. The same screening is performed with tag, or if screening on tag is not necessary, by using MPI_ANY_TAG, instead. After return from MPI_RECV the output buffer buf is emptied and can be reused.

The number of received data items of datatype must be equal or fewer as specified by count, which must be positive or zero. Receiving more data items results in an error. In such cases, the output argument status contains further information about the error. The entire set of arguments: count, datatype, source, tag and comm, must match between the sender process and the receiver process to initiate actual message passing. When a message, posted by a sender process, has been collected by a receiver process, the message is said to be completed, and the program flows of the receiver and the sender processes may continue.

Most implementations of the MPI libraries copy the message data out of the user buffer, which was specified in the MPI program, into some other intermittent system or network buffer. When the user buffer can be reused by the application, the call to MPI_SEND will return. This may happen before the matching MPI_RECV is called or it may not, depending on the message data length.

Example 4.2 Ping-pong message transfer

Let us check the behavior of the MPI_SEND and MPI_RECV operations on your computer, if the message length grows. Two processes will exchange messages that will become longer and longer. Each process will report when the expected message has been sent, which means that it was also received. The code of the MPI program MSMPImessage.cpp is shown in Listing 4.3. Note that there is a single program for both processes. A first part of the program, that determines the number of cooperating processes, is executed on both processes, which must be two. Then the program splits in two parts, first for process of rank = 0 and second of process of rank = 1. Each process sends and receives a message with appropriate calls to the MPI operations. We will see in the following, how the order of these two calls impacts the program execution.

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
  int   numprocs, rank, tag = 100, msg_size=64;
  char  *buf;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs != 2) {
    printf("The number of processes must be two!\n");
    MPI_Finalize();
    return(0);
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("MPI process %d started...\n", rank);
  fflush(stdout);
  while (msg_size  < 10000000) {
    msg_size = msg_size *2;
    buf = (char *)malloc(msg_size * sizeof(char));
    if (rank == 0) {
      MPI_Send(buf, msg_size, MPI_BYTE, rank+1, tag, MPI_COMM_WORLD);
      printf("Message of length %d to process %d\n",msg_size,rank+1);
      fflush(stdout);
      MPI_Recv(buf, msg_size, MPI_BYTE, rank+1, tag,MPI_COMM_WORLD,
               &status);
    }
    if (rank == 1) {
//      MPI_Recv(buf, msg_size, MPI_BYTE, rank-1, tag, MPI_COMM_WORLD,
//               &status);
      MPI_Send(buf, msg_size, MPI_BYTE, rank-1, tag, MPI_COMM_WORLD);
      printf("Message of length %d to process %d\n",msg_size,rank-1);
      fflush(stdout);
      MPI_Recv(buf, msg_size, MPI_BYTE, rank-1, tag, MPI_COMM_WORLD,
               &status);
    }
    free(buf);
  }
  MPI_Finalize();
}
```

Listing 4.3 Verification of MPI_SEND and MPI_RECV operations on your computer.

The output of this program should be as follows:

```
$ mpiexec -n 2 MPImessage
MPI process 0 started...
MPI process 1 started...
Message of length 128 send to process 1.
Message of length 128 returned to process 0.
Message of length 256 send to process 1.
Message of length 256 returned to process 0.
Message of length 512 send to process 1.
Message of length 512 returned to process 0.
Message of length 1024 send to process 1.
Message of length 1024 returned to process 0.
Message of length 2048 send to process 1.
Message of length 2048 returned to process 0.
Message of length 4096 returned to process 0.
Message of length 4096 send to process 1.
Message of length 8192 returned to process 0.
Message of length 8192 send to process 1.
Message of length 16384 send to process 1.
Message of length 16384 returned to process 0.
Message of length 32768 send to process 1.
Message of length 32768 returned to process 0.
Message of length 65536 send to process 1.
Message of length 65536 returned to process 0.
```

The program blocks at the message length 65536, which is in some relation with the capacity of the MPI data buffer in the actual MPI implementation. When the message exceeds it, MPI_Send in both processes block and enter a deadlock. If we just change the order of MPI_Send and MPI_Recv by comment lines 36–37 and uncomment lines 31–32 in process with rank = 1, all expected messages until the length 16777216 are transferred correctly. Some further discussion about the reasons for such a behavior will be provided later, in Sect. 4.7. □

4.5.3 MPI_SENDRECV (sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

The MPI standard specifies several additional operations for message transfer that are a combination of basic MPI operations. They are useful for writing more compact programs. For example, operation MPI_SENDRECV combines a sending of message to destination process dest and a receiving of another message from process source, in a single call in sender and receiver process; however, with two distinct message buffers: sendbuf, which acts as an input, and recvbuf, which is an output. Note that buffers' sizes and types of data can be different.

The send-receive operation is particularly effective for executing a shift operation across a chain of processes. If blocking MPI_SEND and MPI_RECV are used, then one needs to order them correctly, for example, even processes send, then receive, odd processes receive first, then send - so as to prevent cyclic dependencies that may lead to deadlocks. By using MPI_SENDRECV, the communication subsystem will manage these issues alone.

There are further advanced communication operations that are a composition of basic MPI operations. For example, MPI_SENDRECV_REPLACE (buf, count, datatype, dest, sendtag, source, recvtag, comm, status) operation implements the functionality the MPI_SENDRECV, but uses only a single message buffer. The operation is therefore useful in cases with send and receive messages of the same length and of the same data type.

Seven basic MPI operations

Many parallel programs can be written and evaluated just by using the following seven MPI operations that have been overviewed in the previous sections:

```
MPI_INIT,
MPI_FINALIZE,
MPI_COMM_SIZE,
MPI_COMM_RANK,
MPI_SEND,
MPI_RECV,
MPI_WTIME.
```

4.5.4 Measuring Performances

The elapsed time (wall clock) between two points in an MPI program can be measured by using operation `MPI_WTIME ()`. Its use is self-explanatory through a short segment of an MPI program example:

```
double start, finish;
start = MPI_Wtime ();
  ... //MPI program segment to be clocked
finish = MPI_Wtime ();
printf ("Elapsed time is %f\n", finish - start);
```

We are now ready to write a simple example of a useful MPI program that will measure the speed of communication channel between two processes. The program is presented, in more detail, in the next example.

Example 4.3 Measuring communication bandwidth

Let us design a simple MPI program, which will measure the communication channel bandwidth, i.e., the amount of data transferred in a specified time interval, by using MPI communication operations `MPI_SEND` and `MPI_RECV`. As shown in Fig. 4.2, we will generate two processes, either on a single computer or on two interconnected computers. In the first case, the communication channel will be a databus that “connects” the processes through their shared memory, while in the second case the communication channel will be an Ethernet link between computers.

The process with `rank = 0` will send a message, with a specified number of doubles, to the process with `rank = 1`. The communication time is a sum of the communication start-up time t_s and the message transfer time, i.e., the transfer time per word t_w times message length. We could expect that with shorter messages the bandwidth will be lower because a significant part of communication time will be

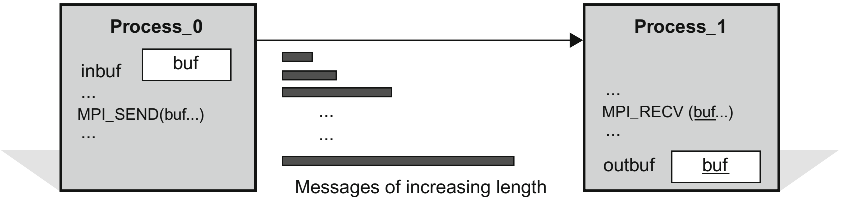


Fig. 4.2 A simple methodology for measurement of process-to-process communication bandwidth

spent on setting up the software and hardware of the message communication channel, i.e., on the start-up time t_s. On the other hand, with long messages, the data transfer time will dominate, hence, we could expect that the communication bandwidth will approach to a theoretical value of the communication channel. Therefore, the length of messages will vary from just a few data items to very long messages. The test will be repeated `nloop` times, with shorter messages, in order to get more reliable average results.

Considering the above methodology, an example of MPI program `MSMPIbw.cpp`, for measuring the communication bandwidth, is given in Listing 4.4. We have again a single program but slightly different codes for the sender and the receiver process. The essential part, message passing, starts in the sender process with a call to `MPI_Send`, which will be matched in the receiver process by a call to corresponding `MPI_Recv`.

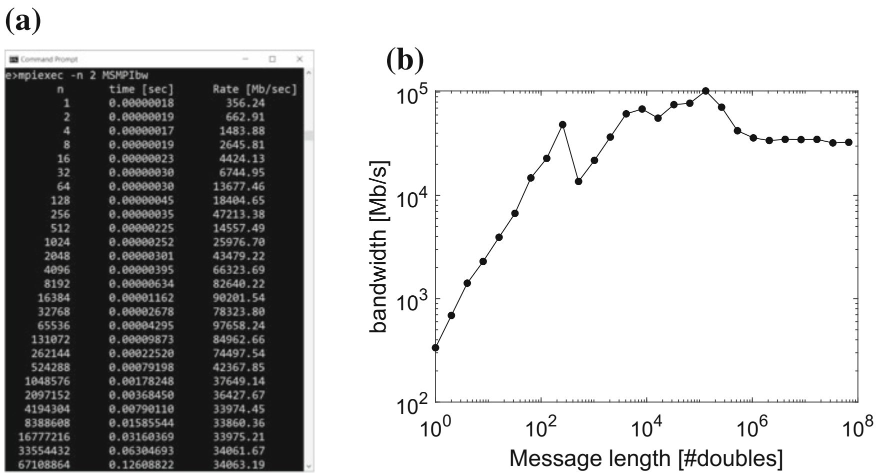
```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define NUMBER_OF_TESTS 10  //for more reliable average results

int main(int argc, char* argv[])
{
  double        *buf;
  int           rank, numprocs;
  int           n;
  double        t1, t2;
  int           j, k, nloop;
  MPI_Status    status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if (numprocs != 2) {
    printf("The number of processes must be two!\n");
    return(0);
  }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    printf("\tn\ttime [sec]\tRate [Mb/sec]\n");
  }
  for (n = 1; n < 100000000; n *= 2) { //message length doubles
    nloop = 1000000 / n;
    if (nloop < 1) nloop = 1;  //just a single loop for long messages.
    buf = (double *)malloc(n * sizeof(double));
    if (!buf) {
      printf("Could not allocate message buffer of size %d\n", n);
      MPI_Abort(MPI_COMM_WORLD, 1);
    }
    for (k = 0; k < NUMBER_OF_TESTS; k++) {
      if (rank == 0) {
        t1 = MPI_Wtime();
        for (j = 0; j < nloop; j++) {//send message nloop times
          MPI_Send(buf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD);
        }
        t2 = (MPI_Wtime() - t1) / nloop;
      }
      else if (rank == 1) {
        for (j = 0; j < nloop; j++) {//receive message nloop times
          MPI_Recv(buf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD, &status);
        }
      }
    }
    if (rank == 0) { //calculate bandwidth
      double bandwidth;
      bandwidth = n * sizeof(double)*1.0e-6 * 8 / t2; //in Mb/sec
      printf("\t%10d\t%10.8f\t%8.2f\n", n, t2, bandwidth);
    }
    free(buf);
  }
  MPI_Finalize();
  return 0;
}
```

Listing 4.4 MPI program for measuring bandwidth of a communication channel.

The output of the MPI program from Listing 4.4, which has been executed on two processes, each running on one of two computer cores that communicate through the shared memory, is shown in Fig. 4.3a with a screenshot of `rank = 0` process user terminal, and in Fig. 4.3b with a corresponding bandwidth graph. The results confirmed our expectations. The bandwidth is poor with short messages and reaches the whole capacity of the memory access with longer messages.

If we assume that with very short messages, the majority of time is spent on the communication setup, we can read from Fig. 4.3a (first line of data) that the setup time was 0.18 μs. The setup time starts increasing when the messages become longer than 16 of doubles. A reason could be that processes communicate until know through the fastest cache memory. Then the bandwidth increases until message length 512 of doubles. A reason for a drop at this length could be cache memory incoherences. The bandwidth converges then to 43 Gb/s, which could be a limit of cache memory access. If message lengths are increased above 524 thousands of doubles, the bandwidth is becoming lower and stabilizes at around 17 Gb/s, eventually because of a limit in shared memory access. Note that the above merits are strongly related to a specific computer architecture and may therefore significantly differ among different computers. □

You are encouraged to run the same program on your computer and compare the obtained results with the results from Fig. 4.3. You may also run the same program on two interconnected computers, e.g., by Ethernet or Wi-Fi, and try to explain the obtained differences in results, taking into account a limited speed of your connection. Note that the maximum message lengths `n` could be made shorter in the case of slower communication channels.



```
e>mpiexec -n 2 MSMPIbw
         n        time [sec]       Rate [Mb/sec]
         1        0.00000018          356.24
         2        0.00000019          662.91
         4        0.00000017         1483.88
         8        0.00000019         2645.81
        16        0.00000023         4424.13
        32        0.00000030         6744.95
        64        0.00000030        13677.46
       128        0.00000045        18404.65
       256        0.00000035        47213.38
       512        0.00000225        14557.49
      1024        0.00000252        25976.70
      2048        0.00000301        43479.22
      4096        0.00000395        66323.69
      8192        0.00000634        82640.22
     16384        0.00001162        90201.54
     32768        0.00002678        78323.80
     65536        0.00004295        97658.24
    131072        0.00009873        84962.66
    262144        0.00022520        74497.54
    524288        0.00079198        42367.85
   1048576        0.00178248        37649.14
   2097152        0.00368450        36427.67
   4194304        0.00790110        33974.45
   8388608        0.01585544        33860.36
  16777216        0.03160369        33975.21
  33554432        0.06304693        34061.67
  67108864        0.12608822        34063.19
```

Fig. 4.3 The bandwidth of a communication channel between two processes on a single computer that communicate through shared memory. **a**) Message length, communication time, and bandwidth, all in numbers; **b**) corresponding graph of the communication bandwidth

4.6 Collective MPI Communication

The communication operations, described in the previous sections, are called from a single process, identified by a `rank`, which has to be explicitly expressed in the MPI program, e.g., by a statement `if(my_id == rank)`. The MPI collective operations are called by all processes in a communicator. Typical tasks that can be elegantly implemented in this way are as follows: global synchronization, reception of a local data item from all cooperating processes in the communicator, and a lot of others, some of them described in this section.

4.6.1 `MPI_BARRIER (comm)`

This operation is used to synchronize the execution of a group of processes specified within the communicator `comm`. When a process reaches this operation, it has to wait until all other processes have reached the `MPI_BARRIER`. In other words, no process returns from `MPI_BARRIER` until all processes have called it. Note that the programmer is responsible that all processes from communicator `comm` will really call to `MPI_BARRIER`.

The barrier is a simple way of separating two phases of a computation to ensure that messages generated in different phases do not interfere. Note again that the `MPI_BARRIER` is a global operation that invokes all processes; therefore, it could be time-consuming. In many cases, the call to `MPI_BARRIER` should be avoided by an appropriate use of explicit addressing options, e.g., `tag`, `source`, or `comm`.

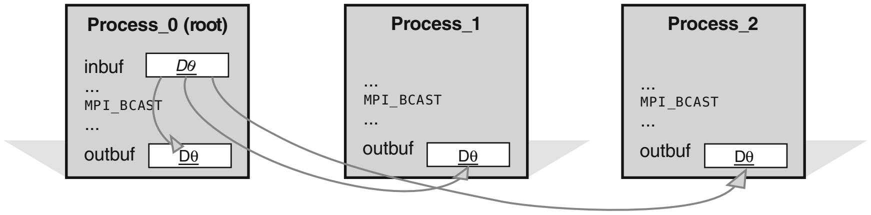


Fig. 4.4 Root process broadcasts the data from its input buffer in the output buffers of all processes

4.6.2 MPI_BCAST (*inbuf*, incnt, intype, root, comm)

The operation implements a one-to-all broadcast operation whereby a single named process root sends its data to all other processes in the communicator, including to itself. Each process receives this data from the root process, which can be of any rank. At the time of call, the input data are located in inbuf of process root and consists of incnt data items of a specified intype. This implies that the number of data items must be exactly the same at input and output side. After the call, the data are replicated in inbuf as output data of all remaining processes. As *inbuf* is used as an input argument at the root process, but as an output argument in all remaining processes, it is of the INOUT type.

A schematic presentation of data broadcast after the call to MPI_BCAST is shown in Fig. 4.4 for a simple case of three processes, where the process with rank = 0 is the root process. Arrows symbolize the required message transfer. Note that all processes have to call MPI_BCAST to complete the requested data relocation.

Note that the functionality of MPI_BCAST could be implemented, in the above example, by three calls to MPI_SEND in the root process and by a single corresponding MPI_RECV call in any remaining process. Usually, such an implementation will be less efficient than the original MPI_BCAST. All collective communications could be time-consuming. Their efficiency is strongly related with the topology and performance of interconnection network.

4.6.3 MPI_GATHER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

All-to-one collective communication is implemented by MPI_GATHER. This operation is also called by all processes in the communicator. Each process, including root process, sends its input data located in inbuf that consists of incnt data items of a specified intype, to the root process, which can be of any rank. Note that the communication data can be different in count and type for each process. However, the root process has to allocate enough space, through its output buffer, that suffices for all expected data. After the return from MPI_GATHER in all processes, the data are collected in outbuf of the root processes.

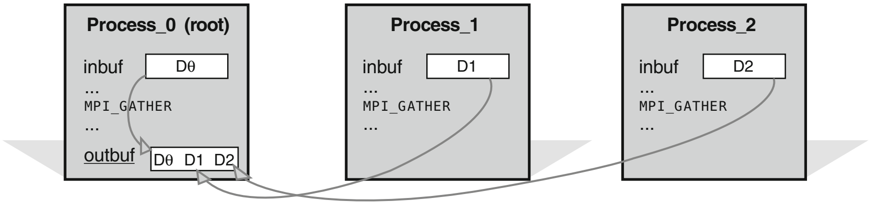


Fig. 4.5 Root process gathers the data from input buffers of all processes in its output buffer

A schematic presentation of data relocation after the call to MPI_GATHER is shown in Fig. 4.5 for the case of three processes, where process with rank = 0 is the root process. Note again that all processes have to call MPI_GATHER to complete the requested data relocation.

4.6.4 MPI_SCATTER (inbuf, incnt, intype, outbuf, outcnt, outtype, root, comm)

This operation works inverse to MPI_GATHER, i.e., it scatters data from inbuf of process root to outbuf of all remaining processes, including itself. Note that the count outcnt and type outtype of the data in each of the receiver processes are the same, so, data is scattered into equal segments.

A schematic presentation of data relocation after the call to MPI_SCATTER is shown in Fig. 4.6 for the case of three processes, where process with rank = 0 is the root process. Note again that all processes have to call MPI_SCATTER to complete the requested data relocation.

There are also more complex collective operations, e.g., MPI_GATHERV and MPI_SCATTERV that allow a varying count of process data from each process and permit some options for process data placement on the root process. Such extensions are possible by changing the incnt and outcnt arguments from a single integer to an array of integers, and by providing a new array argument displs for specifying the displacement relative to root buffers at which to place the processes' data.

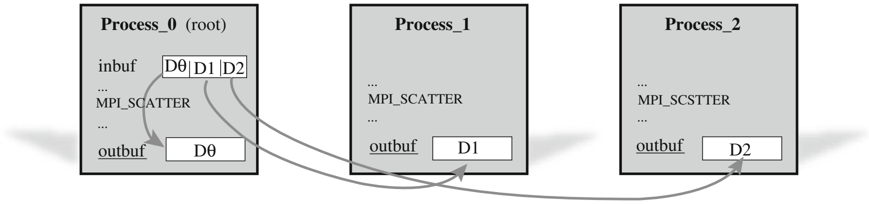


Fig. 4.6 Root process scatters the data from its input buffer to output buffers of all processes in its output buffer

4.6.5 Collective MPI Data Manipulations

Instead of just relocating data between processes, MPI provides a set of operations that perform several simple manipulations on the transferred data. These operations represent a combination of collective communication and computational manipulation in a single call and therefore simplify MPI programs.

Collective MPI operations for data manipulation are based on data reduction paradigm that involves reducing a set of numbers into a smaller set of numbers via a data manipulation. For example, three pairs of numbers: {5, 1}, {3, 2}, {7, 6}, each representing the local data of a process, can be reduced in a pair of maximum numbers, i.e., {7, 6}, or in a sum of all pair numbers, i.e., {15, 9}, and in the same way for other reduction operations defined by MPI:

- `MPI_MAX`, `MPI_MIN`; return either maximum or minimum data item;
- `MPI_SUM`, `MPI_PROD`; return either sum or product of aligned data items;
- `MPI_LAND`, `MPI_LOR`, `MPI_BAND`, `MPI_BOR`; return logical or bitwise AND or OR operation across the data items;
- `MPI_MAXLOC`, `MPI_MINLOC`; return the maximum or minimum value and the rank of the process that owns it;
- The MPI library enables to define custom reduction operations, which could be interesting for advanced readers (see references in Sect. 4.10 for details).

The MPI operation that implements all kind of data reductions is
`MPI_REDUCE (inbuf, outbuf, count, type, op, root, comm)`.
The `MPI_REDUCE` operation implements manipulation `op` on matching data items in input buffer `inbuf` from all processes in the communicator `comm`. The results of the manipulation are stored in the output buffer `outbuf` of process `root`. The functionality of `MPI_REDUCE` is in fact an `MPI_GATHER` followed by manipulation `op` in process `root`. Reduce operations are implemented on a per-element basis, i.e., ith elements from each process' `inbuf` are combined into the ith element in `outbuf` of process `root`.

A schematic presentation of the `MPI_REDUCE` functionality before and after the call:
`MPI_REDUCE (inbuf,outbuf,2,MPI_INT, MPI_SUM,0,MPI_COMM_WORLD)`
is shown in Fig. 4.7. Before the call, `inbuf` of three processes with ranks 0, 1, and 2 were: {5, 1}, {3, 2}, and {7, 6}, respectively. After the call to the `MPI_REDUCE` the value in `outbuf` of `root` process is {15, 9}.

In many parallel calculations, a global problem domain is divided into sub-domains that are assigned to corresponding processes. Often, an algorithm requires that all processes take a decision based on the global data. For example, an iterative calculation can stop when the maximal solution error reaches a specified value. An approach to the implementation of the stopping criteria could be the calculation of maximal sub-domain errors and collection of them in a `root` process, which will evaluate stopping criteria and broadcast the final result/decision to all processes. MPI

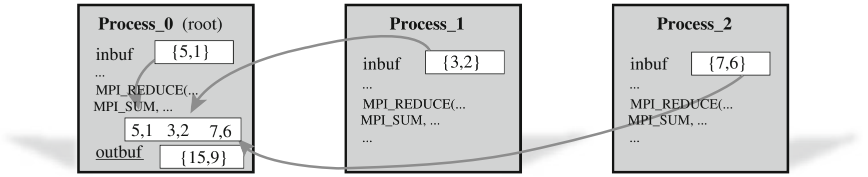


Fig. 4.7 Root process collects the data from input buffers of all processes, performs per-element `MPI_SUM` manipulation, and saves the result in its output buffer

provides a specialized operation for this task, i.e.:
`MPI_ALLREDUCE (inbuf, outbuf, count, type, op, comm)`,
which improves simplicity and efficiency of MPI programs. It works as `MPI_REDUCE` followed by `MPI_BCAST`. Note that the argument `root` is not needed anymore because the final result has to be available to all processes in the communicator. For the same `inbuf` data as in Fig. 4.7 and with `MPI_SUM` manipulation, a call to `MPI_ALLREDUCE` will produce the result $\{15, 9\}$, in output buffers of all processes in the communicator.

Example 4.4 Parallel computation of π

We know that an efficient parallel execution on multiple processors implies that a complex task has to be decomposed in subtasks of similar complexity that have to be executed in parallel on all available processors. Consider again a computation of π by a numerical integration of $4\int_0^1\sqrt{1-x^2}dx$, which represents the area of a circle with radius one that is equal to π. A detailed description of this task is given in Section 2. We divide the interval [0, 1] into N subintervals of equal width. The area of subintervals is calculated in this case slightly different, by a multiplication of subinterval width with the evaluated integrand `yi` in the central point `xi` of each subinterval. Finally, all subareas are summed-up by a partial sum. The schematic presentation of the described methodology is shown in Fig. 4.8a with ten subintervals with central points $xi = [0.05, 0.1, \ldots, 0.95]$.

If we have p available processes and p is much smaller than the number of subintervals N, which is usually the case if we need an accurate solution, the calculation load has to be distributed among all available processes, in a balanced way, for efficient execution. One possible approach is to assign each pth subinterval to a specific process. For example, a process with `rank = myID` will calculate the following subintervals: $i = myID + 1, i + p, i + 2p$, until $i + (k - 1)p \leq N$, where $k = \lceil N/p \rceil$. In the case of $N \gg p$, and N is dividable by p, $k = N/p$, and each process calculates N/p intervals. Otherwise, a small unbalance in the calculation is introduced, because some processes have to calculate one additional subinterval, while remaining processes will already finish their calculation. After the calculation of the area of subintervals, p partial sums are reduced to a global sum, i.e., by sending MPI messages to a root process. The global sum approximates π, which should be now computed faster and more accurate if more intervals are used.

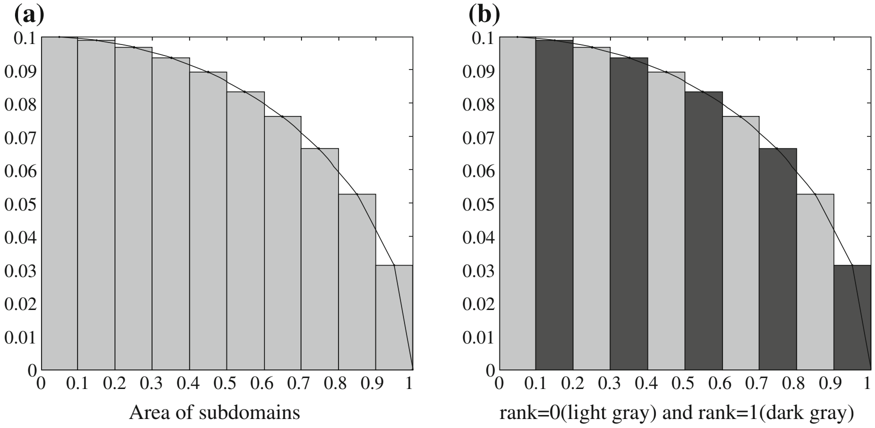


Fig. 4.8 **a**) Discretization of interval [0, 1] in 10 subintervals for numerical integration of quarter circle area; **b**) decomposition for two parallel processes: light gray subintervals are sub-domain of `rank` 0 process; dark gray subintervals of `rank` 1 process

A simple case for two processes and ten intervals is shown in Fig. 4.8b. Five subintervals {1,3,5,7,9}, marked in gray, are integrated by `rank` 0 process and the other five subintervals {2,4,6,8,10}, marked in dark, are integrated by `rank` 1 process.

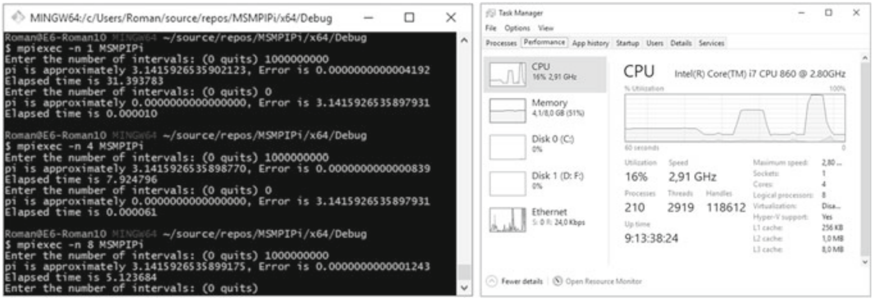
An example of an MPI program that implements parallel computation of π, for an arbitrary p and N, in C programming language, is given in Listing 4.5:

```
#include "stdafx.h"
#include <stdio.h>
#include <math.h>
#include "mpi.h"
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i;
  double PI25DT = 3.141592653589793238462643;
  double pi, h, sum, x, start, finish;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  while (!done) {
    if (myid == 0) {
      printf("Enter the number of intervals: (0 quits) ");
      fflush(stdout);
      scanf_s("%d", &n);
      start = MPI_Wtime();
    }
    //execute in all active processes
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (n == 0) done = 1;
    h = 1.0 / (double)n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
      x = h * ((double)i - 0.5);
      sum += 4.0 * h * sqrt(1.0 - x*x);
    }
    MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) {
      finish = MPI_Wtime();
      printf("Pi is app. %.16f, Error is %.16f\n",pi,fabs(pi-PI25DT));
      printf("Elapsed time is %f\n", finish - start);
    }
  }
  MPI_Finalize();
  return 0;
}
```

Listing 4.5 MPI program in C for parallel computation of π.

Let us open a Terminal window and a Task Manager window (see Fig. 4.9), where we see that the computer used has four cores, eight logical processors and is utilized by a background task for 17%. After running the compiled program for the calculation of π on a single process and with 10^9 intervals, the execution time is about 31.4 s and the CPU utilization increases to 30%. In the case of four processes, the execution time drops to 7.9 s and utilization increases to 70%. Running the program on 8 processes a further speedup is noticed, by the execution time 5.1 s and CPU utilization 100%, because all computational resources of the computer are now fully utilized. From prints in the Terminal window, it is evident that the number π was calculated with similar accuracy in all cases. With our simple MPI program, we achieved a speedup a bit higher than 6, which is excellent! □

Recall, that we have parallelized the computation of π by distributing the computation of subintervals areas among cooperating processes. In this simple example, our, initially continuous, computation domain was interval [0, 1]. The domain was discretized into N subintervals. Then a 1-D domain decomposition was used to divide the whole domain into p sub-domains, where p is the number of cooperating processes that did the actual computation. Finally, the partial results have been assembled in a selected host process and output as a final result. This is the most often used approach for the parallelization in numerical analysis. It can be applied for the operations on large vectors or matrices, for solutions of systems of equations, for solutions of partial differential equations (PDE), and similar. A more detailed methodology and analysis of the parallel program is given in Part III.



```
Roman@E6-Roman10 MINGW64 ~/source/repos/MSMPIPi/x64/Debug
$ mpiexec -n 1 MSMPIPi
Enter the number of intervals: (0 quits) 1000000000
pi is approximately 3.1415926535902123, Error is 0.0000000000004192
Elapsed time is 31.393783
Enter the number of intervals: (0 quits) 0
pi is approximately 0.0000000000000000, Error is 3.1415926535897931
Elapsed time is 0.000010

Roman@E6-Roman10 MINGW64 ~/source/repos/MSMPIPi/x64/Debug
$ mpiexec -n 4 MSMPIPi
Enter the number of intervals: (0 quits) 1000000000
pi is approximately 3.1415926535898770, Error is 0.0000000000000839
Elapsed time is 7.924796
Enter the number of intervals: (0 quits) 0
pi is approximately 0.0000000000000000, Error is 3.1415926535897931
Elapsed time is 0.000061

Roman@E6-Roman10 MINGW64 ~/source/repos/MSMPIPi/x64/Debug
$ mpiexec -n 8 MSMPIPi
Enter the number of intervals: (0 quits) 1000000000
pi is approximately 3.1415926535899175, Error is 0.0000000000001243
Elapsed time is 5.123684
Enter the number of intervals: (0 quits)
```

Intel(R) Core(TM) i7 CPU 860 @ 2.80GHz

Utilization 16%
Speed 2,91 GHz
Processes 210
Threads 2919
Handles 118612
Up time 9:13:38:24

Maximum speed: 2,80 ...
Sockets: 1
Cores: 4
Logical processors: 8
Virtualization: Disa...
Hyper-V support: Yes
L1 cache: 256 KB
L2 cache: 1,0 MB
L3 cache: 8,0 MB

Fig. 4.9 Screenshots of Terminal window and Task Manager indicating timing of the program for calculation of π and the computer utilization history

Quite enough MPI operations

We are now quite familiar with enough operations for coding simpler MPI programs and for evaluating their performances. A list of corresponding MPI operations is shown below:

Basic MPI operations:

```
MPI_INIT, MPI_FINALIZE,
MPI_COMM_SIZE, MPI_COMM_RANK,
MPI_SEND, MPI_RECV,
```

MPI operations for collective communication:

```
MPI_BARRIER,
MPI_BCAST, MPI_GATHER, MPI_SCATTER,
MPI_REDUCE, MPI_ALLREDUCE,
```

Control MPI operations:

```
MPI_WTIME, MPI_STATUS,
MPI_INITIALIZED.
```

4.7 Communication and Computation Overlap

Contemporary computers have separate communication and calculation resources; therefore, they are able to execute both tasks in parallel, which is a significant potential for improving an MPI program efficiency. For example, instead of just waiting for a data transmission to be completed, a certain part of calculation could be done that could be eventually required in the next computing step. If a process can perform useful work while some long communication is in progress, the overall execution time might be reduced. This approach is often termed as a **hiding latency**.

Various communication modes are available in MPI that enable hiding latency, but they require correct usage to avoid communication deadlock or program shutdown. The measures for managing potential deadlocks of communication operations are addressed in more detail in a separate subsection. Finally, a single MPI program in the ecosystems of more communicators is presented. More advanced topics, e.g., a virtual shared memory emulation through the so-called MPI windows, which could simplify the programming and improve the execution efficiency, are beyond the scope of this book and are well covered by continual evolving MPI standard, which should be ultimate reference of enthusiastic programmers.

4.7.1 Communication Modes

MPI processes can communicate in four different communication modes: standard, buffered, synchronous and ready. Each of these modes can be performed in blocking or in non-blocking type, first being less eager to the amount of required memory for message buffering, and second being often more efficient, because of an ability to overlap the communication and computation tasks.

Blocking Communication
A **standard mode** send call, described in Sect. 4.5 with operation MPI_SEND, should be assumed as a blocking send, which will not return until the message data and envelope have been safely stored away. The sender process can access and overwrite the send buffer with a new message. However, depending on the MPI implementation, short messages might still be buffered while longer messages might be split and sent in shorter fragments, or they might be copied into a temporary communication buffer (see Fig. 4.1 for details).

Because the message buffering requires extra memory space and memory-to-memory copying, implementations of MPI libraries do not guarantee the amount of buffering; therefore, one has always to count on the possibility that send call will not complete until a matching receive has been posted, and the data has been moved to the receiver. In other words, the standard send call is **nonlocal**, i.e., may require execution of an MPI operation in another process.

According to the MPI standard, a program is correct and portable if it does not rely on system buffering in the standard mode. Buffering may improve the performance of a correct program, but does not affect the result of the program. There are three blocking send call modes, indicated by a single-letter prefix: MPI_BSEND, MPI_SSEND, MPI_RSEND, with B for buffered, S for synchronous, and R for ready, respectively. The send operation syntax is the same as in the standard send, e.g., MPI_BSEND (buf, count, datatype, dest, tag, comm).

The **buffered mode** send is a standard send with a user-supplied message buffering. It will start independent of a matching receive and can complete before a matching receive is posted. However, unlike the standard send, this operation is **local**, i.e., its completion is independent on the matching receive. Thus, if a buffered send is executed and no matching receive is posted, then the MPI will buffer the outgoing message, to allow the send call to complete. It is a responsibility of the programmer to allocate enough buffer space for all subsequent MPI_BSEND by calling MPI_BUFFER_ATTACH (bbuf, bsize). The buffer space bbuf cannot be reused by subsequent MPI_BSENDs if they have not been completed by matching MPI_RECVs; therefore, it must be large enough to store all subsequent messages.

The **synchronous** mode send can start independently of a matching receive. However, the send will complete successfully only if a matching receive operation has started to receive the message sent by the synchronous send. Thus, the completion of a synchronous send not only indicates that the send buffer can be reused but also indicates that the receiver has reached a certain point in its execution, i.e., it has started executing the matching receive. If both sends and receives are blocking operations then the use of the synchronous mode provides synchronous communication semantics: a communication does not complete at either end before both processes rendezvous at the communication. A send executed in this mode is **nonlocal**, because its competition requires a cooperation of sender and receiver processes.

The **ready mode** send may be started only if the matching receive has been already called. Otherwise, the operation is erroneous and its outcome is undefined. On some systems, this allows the removal of a handshake operation that is otherwise required, which could result in improved performance. In a correct program, a ready send can be replaced by a standard send with no effect on the program results, but with eventually improved performances.

The receive call `MPI_RECV` is always blocking, because it returns only after the receive buffer contains the expected received message.

Non-blocking Communication

Non-blocking send start calls are denoted by a leading letter `I` in the name of MPI operation. They can use the same four modes as blocking sends: standard, buffered, synchronous, and ready, i.e., `MPI_ISEND`, `MPI_IBSEND`, `MPI_ISSEND`, `MPI_IRSEND`. Sends of all modes, except ready, can be started whether a matching receive has been posted or not; a non-blocking ready send can be started only if a matching receive is posted. In all cases, the non-blocking send start call is **local**, i.e., it returns immediately, irrespective of the status of other processes. Non-blocking communications return immediately request handles that can be waited on, or queried, by specialized MPI operations that enables to wait or to test for their completion.

The syntax of the non-blocking MPI operations are the same as in the standard communication mode, e.g.:
`MPI_ISEND (buf, count, datatype, dest, tag, comm, request)`, or
`MPI_IRECV (buf, count,datatype, dest, tag, comm, request),`
except with an additional request handle that is used for later querying by send-complete calls, e.g.:
`MPI_WAIT (request, status)`, or
`MPI_TEST (request, flag, status).`

A non-blocking **standard send** call `MPI_ISEND` initiates the send operation, but does not complete it, in a sense that it will return before the message is copied out of the send buffer. A later separate call is needed to complete the communication, i.e., to verify that the data has been copied out of the send buffer. In the meantime, a computation can run concurrently. In the same way, a non-blocking receive call `MPI_IRECV` initiates the receive operation, but does not complete it. The call will return before a message is stored into the receive buffer. A later separate call is needed to verify that the data has been received into the receive buffer. While querying about the reception of the complete message, a computation can run concurrently.

We can expect that a non-blocking send `MPI_ISEND` immediately followed by send-complete call `MPI_WAIT` is functionally equivalent to a blocking send `MPI_SEND`. One can wait on multiple requests, e.g., in a master/slave MPI program, where the master waits either for all or for some slaves' messages, using MPI operations:

MPI_WAITALL (count, *array_of_requests*, array_of_statuses), or
MPI_WAITSOME (incount, *array_of_requests*, outcount, array_of_indices, array_of_statuses).

A send-complete call returns when data has been copied out of the send buffer. It may carry additional meaning, depending on the send mode. For example, if the send mode is synchronous, then the send can complete only if a matching receive has started, i.e., a receive has been posted, and has been matched with the send. In this case, the send-complete call is nonlocal. Note that a synchronous, non-blocking send may complete, if matched by a non-blocking receive, before the receive complete call occurs. It can complete as soon as the sender "knows" that the transfer will complete, but before the receiver "knows" that the transfer will complete.

If the non-blocking send is in **buffered mode**, then the message must be buffered if there is no pending receive. In this case, the send-complete call is local and must succeed irrespective of the status of a matching receive. If the send mode is standard then the send-complete call may return before a matching receive occurred, if the message is buffered. On the other hand, the send-complete may not complete until a matching receive occurred, and the message was copied into the receive buffer.

Some further facts or implications of the non-blocking communication mode are listed below. Non-blocking sends can be matched with blocking receives, and vice versa. The completion of a send operation may be delayed, for a standard mode, and must be delayed, for synchronous mode, until a matching receive is posted. The use of non-blocking sends in these two cases allows the sender to proceed ahead of the receiver, so that the computation is more tolerant of fluctuations in the speeds of the two processes.

Non-blocking sends in the buffered and ready modes have a more limited impact. A non-blocking send will return as soon as possible, whereas a blocking send will return after the data has been copied out of the sender memory. The use of non-blocking sends is advantageous in these cases only if data copying can be concurrent with computation.

The message passing model implies that a communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication, e.g., message data can be moved directly into the receive buffer, and there is no need to queue a pending send request. However, a receive operation can complete only after the matching send has occurred. The use of non-blocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send. There are further, more advanced, approaches for optimized use of the communication modes that are beyond the scope of this chapter; however, they are well documented elsewhere (see Sect. 4.10).

4.7.2 Sources of Deadlocks

We know from previous sections that after a call to receive operation, e.g., MPI_RECV, the process will wait patiently until a matching MPI_SEND is posted. If the matching send is never posted, the receive operation will wait forever in a deadlock. In practice, the program will become unresponsive until some time limit is exceeded, or the operating system will report a crash. The above situation can appear if two `MPI_RECV` are issued in approximately the same time, on two different processes, that mutually expect a matching send and are waiting to the matching messages that will be never delivered. Such a situation is shown below with a segment from an MPI program, in C language, for process with `rank = 0` and `rank =1`, respectively:

```
if (rank == 0) {
  MPI_Recv (rec_buf, count, MPI_BYTE, 1, tag, comm, &status);
  MPI_Send (send_buf, count, MPI_BYTE, 1, tag, comm);
}
if (rank == 1) {
  MPI_Recv (rec_buf, count, MPI_BYTE, 0, tag, comm, &status);
  MPI_Send (send_buf, count, MPI_BYTE, 0, tag, comm);
}
```

In the same way, if two blocking `MPI_SEND`s are issued in approximately the same time, on process, e.g., with `rank = 0` and `rank =1`, respectively, both followed by a matching `MPI_RECV`, they will never finish if `MPI_SEND`s are implemented without buffers. Even in the case that message buffering is implemented, it will usually suffice only for shorter messages. With longer messages, a deadlock situation could be expected, when the buffer space is exhausted, which was already demonstrated in Listing 4.3.

The above situations are called "unsafe" because they depend on the implementation of the MPI communication operations and on the availability of system buffers. The portability of such unsafe programs may be limited.

Several solutions are available that can make an unsafe program "correct". The simplest approach is to use the order of communication operations more carefully. For example, in the given example, by a call to `MPI_SEND`, in process with `rank = 0`, first. Consequently, with exchanging the order of two lines in the program segment for process with `rank = 0`:

```
if (rank == 0) {
  MPI_Send (send_buf, count, MPI_BYTE, 1, tag, comm);
  MPI_Recv (rec_buf, count, MPI_BYTE, 1, tag, comm, &status);
}
if (rank == 1) {
  MPI_Recv (rec_buf, count, MPI_BYTE, 0, tag, comm, &status);
  MPI_Send (send_buf, count, MPI_BYTE, 0, tag, comm);
}
```

send and receive operations are automatically matched and deadlocks are avoided in both processes.

An alternative approach is to supply receive buffer in the same time as the send buffer, which can be done by operation MPI_SENDRECV. If we replace the MPI_RECV and MPI_SEND pair by MPI_SENDRECV, in both processes, the deadlock is not possible, because four buffers will prevent eventual mutual waiting.

Next possibility is to use a pair of non-blocking operations MPI_IRECV, MPI_ISEND in each process, with subsequent waiting in both processes to both requests by MPI_WAITALL:

```
...
MPI_Request requests[2]
...
if (rank == 0) {
  MPI_Irecv (rec_buf,count,MPI_BYTE,1,tag,comm,&requests[0]);
  MPI_Isend(send_buf,count,MPI_BYTE,1,tag,comm,&requests[1]);
}
else if (rank == 1) {
  MPI_Irecv (rec_buf,count,MPI_BYTE,0,tag,comm,&requests[0]);
  MPI_Isend(send_buf,count,MPI_BYTE,0,tag,comm,&requests[1]);
}
MPI_Waitall (2, request, MPI_STATUSES_IGNORE);
```

The call to MPI_IRECV is issued first, which provides a receive data buffer that is ready for the message that will arrive. This approach avoids extra memory copies of data buffers, avoids deadlock situations and could therefore speed up the program execution.

Finally, non-blocking buffered send can be used MPI_BSEND with explicit allocation of separate send buffers by MPI_BUFFER_ATTACH, however, this approach needs extra memory.

Example 4.5 Hiding latency

We have learned that a blocking send will continue to wait until a matching receive will signal that it is ready to receive. In situations where a significant calculation work follows a send of a large message, and it does not interfere with the send buffer, it might be more efficient to use non-blocking send. Now, the calculation work following the send operation can start almost immediately after the send process is initiated, and can continue to run while the send operation is pending. Similarly, a non-blocking receive could be more efficient than its blocking counterpart if work following the receive operation does not depend on the received message.

In some MPI programs, communication and calculation tasks can run concurrently, and consequently, can speed up the program execution. Suppose that a master process has to receive large messages from all slaves. Then, all processes have to do an extensive calculation that is independent of data in the messages. If blocking communication is used, the execution time will be a sum of communication and calculation time. If asynchronous, non-blocking communication is used, a part of communication and calculation tasks could overlap, which could result in a shorter execution time.

One way to implement the above task is to start a master process that will receive messages from all slave processes, and then proceed with its calculation work. The slave processes will send their messages and then start to calculate. The program runs until all communication and calculation are done. A simple demonstration code of overlapping communication and calculation is given in Listing 4.6.

```
#include <mpi.h> #include <stdlib.h> #include <math.h> #include
<stdio.h>

double other_work(int numproc)
{
    int i;  double a;
    for (i = 0; i < 100000000/numproc; i++) {
        a = sin(sqrt(i));   //different amount of calculation
    }
    return a;
}

int main(int argc, char* argv[])   //number of processes must be > 1
{
    int p, i, myid, tag=1, proc, ierr;
    double start_p, run_time, start_c, comm_t, start_w, work_t, work_r;
    double *buff = nullptr;

    MPI_Request request;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    start_p = MPI_Wtime();
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &p);

    #define master      0
    #define  MSGSIZE    100000000 //5000000  //different sizes of ↩
     messages
    buff = (double*)malloc(MSGSIZE * sizeof(double));   //allocate

    if (myid == master) {
        for (i = 0; i < MSGSIZE; i++) { //initialize message
            buff[i] = 1;
        }
        start_c = MPI_Wtime();
        for (proc = 1; proc<p; proc++) {
#if 1
            MPI_Irecv(buff, MSGSIZE,     //non-blocking receive
                MPI_DOUBLE, MPI_ANY_SOURCE,  tag, MPI_COMM_WORLD, &↩
     request);
#endif
#if 0
            MPI_Recv(buff, MSGSIZE,     //blocking receive
                 MPI_DOUBLE, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &status↩
     );
#endif
        }
        comm_t = MPI_Wtime() - start_c;
        start_w = MPI_Wtime();
        work_r = other_work(p);
        work_t = MPI_Wtime() - start_w;
        MPI_Wait(&request, &status);    //block until Irecv is done
    }
    else {    //slave processes
        start_c = MPI_Wtime();
#if 1
        MPI_Isend(buff, MSGSIZE,      //non-blocking send
             MPI_DOUBLE, master, tag, MPI_COMM_WORLD, &request);
#endif #if 0
        MPI_Send(buff, MSGSIZE,      //blocking send
             MPI_DOUBLE, master, tag, MPI_COMM_WORLD);
#endif
        comm_t = MPI_Wtime()-start_c;
        start_w = MPI_Wtime();
        work_r = other_work(p);
        work_t = MPI_Wtime()-start_w;
        MPI_Wait(&request, &status);     //block until Isend is done
    }
    run_time = MPI_Wtime() - start_p;
    printf("Rank \t Comm[s] \t Calc[s] \t Total[s] \t Work_result\n");
    printf(" %d\t %e\t %e\t %e\t %e\t\n", myid, comm_t, work_t, run_time,↩
      work_r);
    fflush(stdout);  //to correctly finish all prints
    free(buff);
    MPI_Finalize();
}
```

Listing 4.6 Communication and calculation overlap.

The program from Listing 4.6 has to be executed with at least two processes: one master and one or more slaves. The non-blocking `MPI_Isend` call, in all processes, returns immediately to the next program statement without waiting for the communication task to complete. This enables `other_work` to proceed without delay. Such a usage of non-blocking send (or receive), to avoid processor idling, has the effect of "latency hiding", where MPI latency is the elapsed time for an operation, e.g., `MPI_Isend`, to complete. Note that we have used `MPI_ANY_SOURCE` in the master process to specify message source. This enables an arbitrary arrival order of messages, instead of a predefined sequence of processes that can further speed up the program execution.

The output of this program should be as follows:

```
$ mpiexec -n 2 MPIhiding
Rank   Comm[s]        Calc[s]        Total[s]       Work_result
 1     2.210910e-04   1.776894e+00   2.340671e+00   6.109991e-01
Rank   Comm[s]        Calc[s]        Total[s]       Work_result
 0     1.692562e-05   1.747064e+00   2.340667e+00   6.109991e-01
```

Note that the total execution time is longer than the calculation time. The communication time is negligible, even that we have sent 100 millions of doubles. Please use blocking MPI communication, compare the execution time, and explain the differences. Please experiment with different numbers of processes, different message lengths, and different amount of calculation, and explain the behavior of the execution time. □

4.7.3 Some Subsidiary Features of Message Passing

The MPI communication model is by default **nondeterministic**. The arrival order of messages sent from two processes, A and B, to a third process, C, is not known in advance.

The MPI communication is **unfair**. No matter how long a send process has been pending, it can always be overtaken by a message sent from another sender process. For example, if process A sends a message to process C, which executes a matching receive operation, and process B sends a competing message that also matches the receive operation in process C, only one of the sends will complete. It is the programmer's responsibility to prevent "starvation" by ensuring that a computation is deterministic, e.g., by forcing a reception of specific number of messages from all competing processes.

The MPI communication is **non-overtaking**. If a sender process posts successive messages to a receiver process and a receive operation matches all messages, the messages will be managed in the order as they were sent, i.e., the first sent message will be received first, etc. Similarly, if a receiver process posts successive receives, and all match the same message, then the messages will be received in the same order as they have been sent. This requirement facilitates correct matching of a send to a receive operation and guarantees that an MPI program is deterministic, if the cooperating processes are single-threaded.

On the other hand, if an MPI process is multi-threaded, then the semantics of thread execution may not define a relative order between send operations from distinct program threads. In the case of multi-threaded MPI processes, the messages sent from different threads can be received in an arbitrary order. The same is valid also for multi-threaded receive operations, i.e., successively sent messages will be received in an arbitrary order.

Example 4.6 Fairness and overtaking of MPI communication

A simple demonstration example of some MPI communication features is given in Listing 4.7. The master process is ready to receive `10*(size-1)` messages, while each of the slave processes wants to send 10 messages to the master process, each with a larger tag. The master process lists all received messages with their source process ranks and their tags.

```
#include "stdafx.h"
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
  int rank, size, i, buf[1];
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if (rank == 0) {
    for (i = 0; i < 10*(size-1); i++) {
        MPI_Recv(buf, 1, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      printf("Msg from %d with tag %d\n",
        status.MPI_SOURCE, status.MPI_TAG);
    }
  }
  else { //rank > 0
    for (i = 0; i < 10; i++)
      MPI_Send(buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD);
  }
  MPI_Finalize();
  return 0;
}
```

Listing 4.7 Demonstration of unfairness and non-overtaking in MPI communication.

The output of this program depends on the number of cooperating processes. For the case of 3 processes it could be as follows:

```
$ mpiexec -n 3 MPIfairness
Msg from 1 with tag 0
Msg from 1 with tag 1
Msg from 1 with tag 2
Msg from 1 with tag 3
Msg from 1 with tag 4
Msg from 1 with tag 5
Msg from 1 with tag 6
Msg from 1 with tag 7
Msg from 1 with tag 8
Msg from 1 with tag 9
Msg from 2 with tag 0
Msg from 2 with tag 1
Msg from 2 with tag 2
Msg from 2 with tag 3
Msg from 2 with tag 4
Msg from 2 with tag 5
Msg from 2 with tag 6
Msg from 2 with tag 7
Msg from 2 with tag 8
Msg from 2 with tag 9
```

We see that all messages from the process with rank 1 have been received first, even that the process with rank 2 has also attempted to send its messages, so the communication was unfair. The order of received messages, identified by tags, is the same as the order of sent messages, so the communication was non-overtaking. □

4.7.4 MPI Communicators

All communication operations introduced in previous sections have used the default communicator `MPI_COMM_WORLD`, which incorporates all processes involved and defines a default context. More complex parallel programs usually need more process groups and contexts to implement various forms of sequential or parallel decomposition of a program. Also, the cooperation of different software developer groups is much easier if they develop their software modules in distinct contexts. The MPI library supports modular programming via its communicator mechanism that provides the "information hiding" and "local name space", which are both needed in modular programs.

We know from previous sections that any MPI communication operation specifies a **communicator**, which identifies a **process group** that can be engaged in the communication and a **context** (tagging space) in which the communication occurs. Different communicators can encapsulate the same or different process groups but always with different contexts. The message context can be implemented as an extended tag field, which enables to distinguish between messages from different contexts. A communication operation can receive a message only if it was sent in the same context; therefore, MPI processes that run in different contexts cannot be interfered by unwanted messages.

For example, in master–slave parallelization, master process manages the tasks for slave processes. To distinguish between master and slave tasks, statements like `if(rank==master)` and `if(rank>master)` for ranks in a default communicator `MPI_COMM_WORLD` can be used. Alternatively, the processes of a default communicator can be splitted into two new sub-communicators, each with a different group of processes. The first group of processes, eventually with a single process, performs master tasks, and the second group of processes, eventually with a larger number of processes, executes slave tasks. Note that both sub-communicators are encapsulated into a new communicator, while the default communicator still exists. A collective communication is possible now in the default communicator or in the new communicator.

In a further example, a sequentially decomposed parallel program is schematically shown in Fig. 4.10. Each of the three vertical lines with blocks represents a single process of the parallel program, i.e., `P_0`, `P_1`, and `P_2`. All three processes form a single process group. The processes are decomposed in consecutive sequential program modules shown with blocks. Process-to-process communication calls are shown with arrows. In Fig. 4.10a, all processes and their program modules run in the same context, while in Fig. 4.10b, program modules, encircled by dashed curves, run in two different contexts that were obtained by a duplication of the default communicator.

Figure 4.10a shows that MPI processes `P_0` and `P_2` have finished sooner than `P_1`. Dashed arrows denote messages that have been generated during subsequent computation in `P_0` and `P_2`. The messages could be accepted by a sequential program module `P1_mod_1` of MPI process `P_1`, which is eventually NOT correct. A problem solution is shown in Fig. 4.10b. The program modules run here in two different contexts, `New_comm(1)` and `New_comm(2)`. The early messages will be accepted now correctly by MPI receive operations in program module `P1_mod_2` from communicator `New_comm(2)`, which uses a distinct tag space that will correctly match the problematic messages.

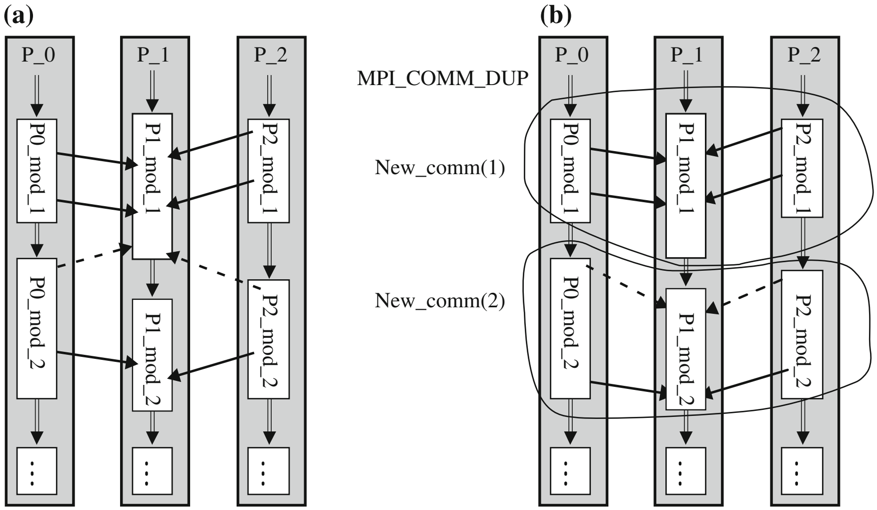


Fig. 4.10 Sequentially decomposed parallel program that runs on three processes. **a**) processes run in the same context; **b**) processes run in two different contexts

The MPI standard specifies several operations that support modular programming. Two basic operations implement duplication or splitting of an existing communicator `comm`.

`MPI_COMM_DUP (comm, new_comm)`

is executed by each process from the parent communicator `comm`. It creates a new communicator `new_comm` comprising the same process group but a new context. This mechanism supports sequential composition of MPI programs, as shown in Fig. 4.10, by separating communication that is performed for different purposes. Since all MPI communication is performed within a specified communicator, `MPI_COMM_DUP` provides an effective way to create a new user-specified communicator, e.g., for use by a specific program module or by a library, in order to prevent interferences of messages.

`MPI_COMM_SPLIT (comm, color, key, new_comm)`

creates a new communicator `new_comm` from the initial communicator `comm`, comprising disjoint subgroups of processes with optional reordering of their ranks. Each subgroup contains all processes of the same `color`, which is a nonnegative argument. It can be `MPI_UNDEFINED`; in this case, its corresponding process will not be included in any of the new communicators. Within each subgroup, the processes are ranked in the order defined by the value of corresponding argument `key`, i.e., a lower value of `key` implies a lower value of `rank`, while equal process `keys` preserve the original order of `ranks`. A new sub-communicator is created for each subgroup and returned as a component of a new communicator new_comm. This mechanism supports parallel decomposition of MPI programs. The MPI_COMM_SPLIT is a collective communication operation with a functionality similar to MPI_ALLGATHER to collect color and key from each process. Consequently, the MPI_COMM_SPLIT operation must be executed by every process from the parent communicator comm, however, every process is permitted to apply different values for color and key.

Remember that every call to MPI_COMM_DUP or MPI_COMM_SPLIT should be followed by a call to MPI_COMM_FREE, which deallocates a communicator that can be reused later. The MPI library can create a limited number of objects at a time and not freeing them could result in a runtime error.

More flexible ways to create communicators are based on MPI object MPI_GROUP. A process group is an ordered set of process identifiers associated with an integer rank. Process groups allow a subset of processes to communicate among themselves using local names and identifiers without interfering with other processes, because groups do not have a context. Dedicated MPI operations can create groups in a communicator by MPI_COMM_GROUP, obtain a group size of a calling process by MPI_GROUP_SIZE, perform set operations between groups, e.g., union by MPI_GROUP_UNION, etc., and create a new communicator from the existing group by MPI_COMM_CREATE_GROUP.

There are many advanced MPI operations that support the creation of communicators and structured communication between processes within a communicator, i.e., intracommunicator communication, and between processes from two different communicators, i.e., intercommunicator communication. These topics are useful for advanced programming, e.g., in the development of parallel libraries, which are not covered in this book.

Example 4.7 Splitting MPI communicators

Let visualize now the presented concepts with a simple example. Suppose that we would like to split a default communicator with eight processes ranked as rank = {0 1 2 3 4 5 6 7} to create two sets of process by a call to MPI_COMM_SPLIT, as shown in Fig. 4.11. Two disjoint sets should include processes with odd and even ranks, respectively. We therefore need two colors that can be created, for example, with division of original ranks by modulo 2: color = rank%2, which results in corresponding processes' colors = {0 1 0 1 0 1 0 1}. Ranks of processes in new groups are assigned according to process key. If corresponding keys are {0 0 0 0 0 0 0 0}, new process ranks in groups, new_g1 and new_g2, are sorted in the ascending order as in the initial communicator.

A call to MPI_COMM_SPLIT (MPI_COMM_WORLD, rank%2, 0, & new_comm); will partition the initial communicator with eight processes in two groups with four processes, based on the color, which is, in this example, either 0 or 1. The groups, identified by their initial ranks, are new_g1 = {0 2 4 6} and new_g2 = {1 3 5 7}. Because all process keys are 0, new process ranks of both groups are sorted in ascending order as rank = {0 1 2 3}.

A simple MPI program MSMPIsplitt.cpp in Listing 4.8 implements the above ideas. MPI_COMM_SPLIT is called by color = rank%2, which is either 0 or 1.

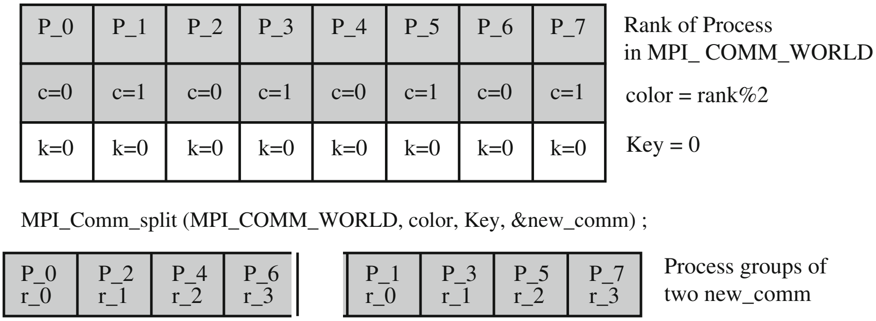


Fig. 4.11 Visualization of splitting the default communicator with eight processes into two subcommunicators with disjoint sets of four processes

Consequently, we get two process groups with four processes per group. Note that the new process ranks in both groups are equal to {0 1 2 3}, because the `key = 0` in all processes, and consequently, the original order of ranks remain the same. For an additional test, the master process calculates the sum of processes' ranks in each new group of a new communicator, using the `MPI_REDUCE` operation. In this simple example, the sum of ranks in both groups should be equal to $0 + 1 + 2 + 3 = 6$.

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

int main(int argc, char **argv)
{
  int  numprocs, org_rank, new_size, new_rank;
  MPI_Comm new_comm;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &org_rank);

  MPI_Comm_split(MPI_COMM_WORLD, org_rank%2, 0, &new_comm);
//  MPI_Comm_split(MPI_COMM_WORLD,org_rank>=2,org_rank<=3,&new_comm);
  MPI_Comm_size(new_comm, &new_size);
  MPI_Comm_rank(new_comm, &new_rank);
  printf("'MPI_COMM_WORLD' process rank/size %d/%d has rank/size %d/%d in↩
      'new_comm'\n", org_rank, numprocs, new_rank, new_size);

  int sum_ranks; //calculate sum of ranks in both new groups of new_com
  MPI_Reduce(&new_rank, &sum_ranks, 1, MPI_INT, MPI_SUM, 0, new_comm);
  if (new_rank == 0) {
    printf("Sum of ranks in 'new_com': %d\n", sum_ranks);
  }

  MPI_Comm_free(&new_comm);
  MPI_Finalize();
  return 0;
}
```

Listing 4.8 Splitting a default communicator in two process groups of a new communicator. First and second process groups include, respectively, processes with even and odd ranks from the default communicator.

The output of compiled program from Listing 4.8, after running it on eight processes, should be similar to:

```
$ mpiexec -n 8 MSMPIsplitt
'MPI_COMM_WORLD' process rank/size 4/8 has rank/size 2/4 in 'new_comm'
'MPI_COMM_WORLD' process rank/size 6/8 has rank/size 3/4 in 'new_comm'
'MPI_COMM_WORLD' process rank/size 5/8 has rank/size 2/4 in 'new_comm'
'MPI_COMM_WORLD' process rank/size 0/8 has rank/size 0/4 in 'new_comm'
Sum of ranks in 'new_com': 6
'MPI_COMM_WORLD' process rank/size 7/8 has rank/size 3/4 in 'new_comm'
'MPI_COMM_WORLD' process rank/size 1/8 has rank/size 0/4 in 'new_comm'
Sum of ranks in 'new_com': 6
'MPI_COMM_WORLD' process rank/size 3/8 has rank/size 1/4 in 'new_comm'
'MPI_COMM_WORLD' process rank/size 2/8 has rank/size 1/4 in 'new_comm'
```

The above output confirms our expectations. We have two process groups in the new communicator, each comprising four processes with ranks 0 to 3. Both sums of ranks in process groups are 6, as expected. □

For an exercise, suppose that we have seven processes in the default communicator `MPI_COMM_WORLD` with initial `ranks` = {0 1 2 3 4 5 6}. Note that for this case, the program should be executed by `mpiexec` option `-n 7`. Let the `color` be (`rank >= 2`) and `key` be (`rank <= 3`), which results in process `colors` = {0 0 1 1 1 1 1} and `keys` = {1 1 1 1 0 0 0}. After a call to `MPI_COMM_SPLIT` operation, two process groups are created in `new_comm`, with two and five members, respectively. By using initial `rank` for the processes identification, the processes in new groups are `new_g1` = {0 1} and `new_g2` = {2 3 4 5 6}.

The new ranks of processes in both groups are determined according to corresponding values of keys. Aligning the initial `rank` and `key`, we see, for example, that process with initial `rank` = 0 is aligned with `key` = 1, or process with initial `rank` = 4 is aligned with `key` = 0, etc. Now, the keys can be assigned to process groups as: `key_g1` = {1 1} and `key_g2` = {1 1 0 0 0}. Because smaller values of `keys` relate with smaller values of ranks, and because equal `keys` does not change the original rank's order, we get: `rank_g1` = {0 1} and `rank_g2` = {3 4 0 1 2}. For example, process with initial `rank` = 4 becomes a member of `new_g2` with `rank` = 0. Obviously, the sums of ranks in both groups of the new communicator are 1 and 10, respectively. Please, feel free to adapt MPI program `MSMPIsplitt.cpp` from Listing 4.8 in a way that it will implement the described example.

4.8 How Effective Are Your MPI Programs?

Already in the simple cases of MPI programs, one can analyze the speedup as a function of the problem size and as a function of the number of cooperating processes.

The parallelization of sequential problems can be guided by various methodologies that provide the same quantitative results, however, in different execution time or with different memory requirements. Some parallelization approaches are better for a smaller number of computing nodes and other for a larger number of nodes. We are looking for an optimal solution that is simple, efficient, and scalable. A simple parallelization methodology, proposed by Ian Foster in his famous book "Designing and Building Parallel Programs", is performed in four distinct stages: Partitioning, Communication, Agglomeration, and Mapping (PCAM).

In the first two stages, the sequential problem is decomposed into, as small as possible, tasks and the required communication among the tasks is identified. The available parallel execution platform is ignored for these two phases, because the aim is a maximal decomposition, with the final goal, to improve concurrency and scalability of the discovered parallel algorithms.

The third and fourth stages respect the ability of targeted parallel computer. The identified fine-grained tasks have to be agglomerated to improve performance and to reduce development costs. The last stage is devoted to the mapping of tasks on real computers, taking into account the locality of communication and balancing of calculation load.

The developed parallel program speedup and, consequently, its efficiency and scalability depend mainly on the following three issues:

- balancing of computing and communication loads among processes,
- ratio between computing and communication loads, and
- computer architecture.

Further improvements in the parallelization efficiency could be obtained by an overlapping of calculation with communication, in particular in problems with large messages. Some of the approaches to measure the performance of MPI programs are presented in Part III.

4.9 Exercises and Mini Projects

Test Questions

1. True or false:
   (a) MPI is a message passing library specification not a language or compiler specification.
   (b) In the MPI model processes communicate only by shared memory.
   (c) MPI is useful for an implementation of MIMD/SPMD parallelism.
   (d) A single MPI program is usually written that can run with a general number of processes.
   (e) It is necessary to specify explicitly, which part of the MPI code will run with specific processes.
2. True or false:
   (a) A group and context together form a communicator.
   (b) A default communicator `MPI_COMM_WORLD` contains in its group all initial

processes and its context is default.
(c) A process is identified by its rank in the group associated with a communicator.
(d) Maximal rank is equal to size.

3. List, in the required order, (a) MPI functions to control starting and terminating procedures of MPI processes.
(b) MPI functions for determining the number of participating processes and the identifier of the current process.
4. Suppose that a process with rank 1 started the execution of `MPI_SEND (buf, 5, MPI_INT, 4, 7, MPI_COMM_WORLD)`.
(a) Which process has to start matching `MPI_RECV` to finish this communication?
(b) Write the adequate `MPI_RECV`.
(c) What will be received?
5. Name the following definitions of the MPI communication semantics:
(a) An operation may return before its completion, and before the user is allowed to reuse resources (such as buffers) specified in the call.
(b) Return from an operation call indicates that resources can safely be reused.
(c) A call may require execution of an operation on another process, or communication with another process.
(d) All processes in a group need to invoke the procedure.
6. When a process makes a call to `MPI_RECV`, it will wait patiently until a matching send is posted. If the matching send is never posted, the receiver will wait forever.
(a) Name this situation.
(b) Describe a solution to the problem?
7. Give a functional equivalent program segment using non-blocking send to implement blocking MPI send operation: `MPI_SEND`.
8. Name the following definitions of the MPI communication semantics:
(a) If a sender posts two messages to the same receiver, and a receive operation matches both messages, the message first posted will be chosen first.
(b) No matter how long a send has been pending, it can always be overtaken by a message sent from another process.
(c) Does the MPI implementation by itself guarantee fairness?
9. (a) Implement a one-to-all MPI broadcast operation whereby a single named process (root) sends the same data to all other processes.
(b) Which process(es) has(have) to call this operation?
10. Suppose an $M \times N$ `array` of doubles stored in a C row-major layout in the sender system memory.
(a) Construct a continuous derived datatype `MPI_newtype` specifying a column of the array.
(b) Write an `MPI_Send` to send the first column of `array`. Try the same for the second column. Note that the first stride starts now at `array[0][1]`.
11. Suppose four processes a, b, c, d, with corresponding `oldrank` in `comm`: 0, 1, 2, 3. Let `color=oldrank%2` and corresponding `key`= 7, 1, 0, 3. Identify

newgroups of newcomm, sorted by newranks, after the execution of: MPI_COMM_SPLIT (comm, color, key, newcomm).

12. Which types of parallel program composition are supported by:
(a) MPI_COMM_DUP (comm, newcomm) and by
(b) MPI_COMM_SPLIT (comm, color, key, newcomm)?
(c) Are the above operations examples of collective operations?

Mini Projects

P1. Implement MPI program for a 2-D finite difference algorithm on a square domain with $n \times n = N$ points. Assume 5 points stencil (actual point and four neighbors). Assume ghost boundary points in order to simplify the calculation in border points (all stencils, including boundary points, are equal). Compare the obtained results, after a specified number of iterations, on a single MPI process and on a parallel multi-core computer, e.g., with up to eight cores. Use the performance models for calculation and communication to explain your results. Plot the execution time as a function of the number of points N and as a function of the number of processes p for, e.g., 10^4 time steps.

P2. Use MPI point-to-point communication to implement the broadcast and reduce functions. Compare the performance of your implementation with that of the MPI global operations MPI_BCAST and MPI_REDUCE for different data sizes and different numbers of processes. Use data sizes up to 10^4 doubles and up to all available number of processes. Plot and explain the obtained results.

P3. Implement the summation of four vectors, each of N doubles, with an algorithm similar to the reduction algorithm. The final sum should be available on all processes. Use four processes. Each of them will initially generate its own vector. Use MPI point-to-point communication to implement your version of the summation of the generated vector. Test your program for small and large vectors. Comment results and compare the performance of your implementation with that of the MPI_ALLREDUCE. Explain any differences.

4.10 Bibliographical Notes

The primary source of MPI information is available at MPI Forum website: https://www.mpi-forum.org/ where the complete MPI library specifications and documents are available. MPI features of Version 2.0 are mostly referenced in this book as later versions include more advanced options, however, they are backward compatible with MPI 2.0.

Newer MPI standards [10] are trying to better support the scalability in future extreme-scale computing systems using advanced topics as: one-sided communications, extended collective operations, process topologies, external interfaces, etc. Advanced topics, e.g., a virtual shared memory emulation through so-called MPI windows, which could simplify the programming and improve the execution efficiency, are beyond the scope of this book and are well covered by the continual evolving MPI standard, which should be an ultimate reference of enthusiastic programmers.

More demanding readers are adviced to check several well-documented open-source references for further reading, e.g., for the MPI standard [16], for MPI implementations [1,2], and many other internet sources for advanced MPI programming.

Note that besides the parallel algorithm, parallelization methodology [9], and the computational performance of the cooperating computers, the parallel program efficiency depends also on the topology and speed of the interconnection network [26].