4. Parallel Programming with MPI

4. Parallel Programming with MPI 1
4.1. MPI: Basic Concepts and Definitions 3
4.1.1. The Concept of Parallel Program 3
4.1.2. Data Communication Operations 3
4.1.3. Communicators 3
4.1.4. Data Types 3
4.1.5. Virtual Topologies 4
4.2. Introduction to MPI Based Development of Parallel Programs 4
4.2.1. The Fundamentals of MPI 4
4.2.1.1 MPI Programs Initialization and Termination 4
4.2.1.2 Determining the Number and the Rank of the Processes 5
4.2.1.3 Message Passing 5
4.2.1.4 Message Reception 6
4.2.1.5 The First MPI Based Parallel Program 7
4.2.2. The Determination of the MPI Based Program Execution Time 8
4.2.3. Introduction to the Collective Data Transmission Operations 9
4.2.3.1 Data Broadcasting 9
4.2.3.2 Data Transmission from All the Processes to a Process. Reduction Operation. 11
4.2.3.3 Computation Synchronization 13
4.3. Data Transmission between Two Processes 13
4.3.1. Data Transmission Modes 13
4.3.2. Nonblocking Communications 14
4.3.3. The Simultaneous Transmission and Reception 15
4.4. Collective Data Transmission Operations 16
4.4.1. Scattering Data from a Process to all the Processes 16
4.4.2. Gathering Data from All the Processes to a Process 17
4.4.3. All to All Data Transmission 17
4.4.4. Additional Data Reduction Operations 18
4.4.5. The Total List of Collective Data Operations 19
4.5. The Derived Data Types in MPI 19
4.5.1. The Derived Data Type 20
4.5.2. The Methods of Constructing the Derived Data Types 21
4.5.2.1 The Contiguous Constructing Method 21
4.5.2.2 The Vector Constructive Method 21
4.5.2.3 The Index Constructing Method 22
4.5.2.4 The Structural Constructing Method 23
4.5.3. Declaring and Deleting Derived Data Types 23
4.5.4. Creating Messages by Means of Data Packing and Unpacking 23
4.6. Managing Groups of Processes and Communicators 25
4.6.1. Managing Groups 25
4.6.2. Managing Communicators 26
4.7. Virtual Topologies 27
4.7.1. Cartesian Topologies 27
4.7.2. Graph Topology 29
4.8. Additional Information on MPI 30
4.8.1. The Development of MPI Based Parallel Programs in the Algorithmic Language Fortran 30
4.8.2. Overview of MPI Program Execution Environment 30

4.8.3. Additional Features of the Standard MPI-2 31
4.9. Summary 31
4.10. References 32
4.11. Discussions 32
4.12. Exercises 33

This Section is devoted to the description of parallel programming methods for the distributed memory computer systems with the use of MPI.

The processors in the computer systems with distributed memory (see Figure 1.6) work independently of each other. It is necessary to have a possibility to distribute the computational load and to organize the information interaction (data transmission) among the processors in order to arrange parallel computations under these conditions.

The solution to the above mentioned problems is provided by *message passing interface - MPI*.

1. Generally, it is necessary to analyze the algorithm of solving a problem, to select the fragments of the computations, which are independent with regard to the information, in order to distribute the computations among the processors. It is also necessary to perform the program implementation of the fragments and then distribute the obtained program parts on different processors. A simpler approach is used in MPI. According to this approach, *a program is developed for solving the stated problem and this single program is executed simultaneously on all the available processors*. In order to avoid the identity of computations on different processor, it is first of all possible to substitute different data for executing the program on different processors. Secondly, there are means in MPI to identify the processor, on which the program is executed. Thus, it provides the possibility to organize the computations differently depending on the processor used by the program. This method of organizing parallel computations is referred to as the model *single program multiple processes* or *SPMP*[1)].

2. The operation of data communication is sufficient for the minimum variant of organizing the information interaction among the processors (it should also be technically possible to provide communication among the processors –we mean the availability of channels or communication lines). There are many data transmission operations in MPI. They provide various means of data passing and implement practically all the communication operations discussed in Section 3. These possibilities are the main advantages of MPI (the very name of the method MPI testifies to it).

It should be noted that the attempts to create software for data transmission among the processors began practically right after the local computer networks came into being. Such software, for instance, are described in Buyya (1999), Andrews (2000) and many others. This software, however, were often incomplete and, moreover, incompatible. Thus, the portability of programs in case of transferring the software to other computer systems is one of the most serious problems in software development. This problem is strongly felt in development of parallel programs. As a result, the efforts to standardize the software for organizing message transmission in multiprocessor computational systems have been made since the 90-s. The work, which directly led to the creation of MPI, was initiated by the Workshop on Standards for Message Passing in a Distributed Memory Environment, Williamsburg, Virginia, USA, April 1992. According to the results of the workshop a special workgroup was created, which was later transformed into an international community (*MPI Forum*). The result of its activity was the creation and adaptation in 1994 of the *Message Passing Interface* standard, version 1.0. Later the standard was sequentially developed. In 1997 MPI, version 2.0, was adopted.

So now it is reasonable to explain the concept MPI. First of all, MPI is a standard for organizing message passing. Secondly, MPI is the software, which should provide the possibility of message passing and correspond to all the requirements of MPI standard. According to the standard this software should be arranged as program module libraries (*MPI libraries*) and should be comprehensible for the most widely used algorithmic languages C and Fortran. This "duality" of MPI should be taken into account while using terminology. As a rule, the abbreviation MPI is used to refer to the standard and the phrase "MPI library" points to a standard software implementation. However, the term MPI is often used to denote MPI libraries also, and thus, the correct interpretation of the term depends on the context.

There are many works devoted to the problems connected with the development of parallel programs using MPI. Brief survey of useful reference materials may be found at the end of the section. Before we started to describe MPI in details, we find useful to mention some of its advantages:

- MPI makes possible to decrease considerably the complexity of the parallel program portability among different computer systems. A parallel program developed in the algorithmic languages C or Fortran with the use of MPI library will, as a rule, operate on different computer platforms,

---

[1)] It is more often referred to as *single program multiple data* or *SPMD*. With regard to MPI it would be more logical to use the term SPMP.

- MPI contributes to the increase of parallel computation efficiency, as there are MPI library implementations for practically every type of computer system nowadays. These realizations are carried out with regard to the possibilities of the hardware being used,
- MPI decreases the complexity of parallel program development as on the one hand, the greater part of the basic data transmission operations discussed in Section 3 are provided by MPI standard. On the other hand, there are many parallel numerical libraries available nowadays created with the use of MPI.

4.1. MPI: Basic Concepts and Definitions

Let us discuss a number of concepts and definitions, which are fundamental for MPI standard.

4.1.1. The Concept of Parallel Program

Within the frame of MPI a *parallel program* means a number of simultaneously carried out *processes*. The processes may be carried out on different processors. Several processes may be located on a processor (in this case they are carried out in the time-shared mode). In the extreme case a single processor may be used to carry out a parallel program. As a rule, this method is applied for the initial testing of the parallel program correctness.

Each parallel program process is generated on the basis of the copy of the same program code (SPMP model). This program code presented as an executable file must be accessible at the moment when the parallel program is started on all the processors used. The source program code for the program being executed is developed in the algorithmic languages C or Fortran with the use of some MPI library implementation.

The number of processes and the number of the processors used are determined at the moment of the parallel program launch by the means of MPI program execution environment. These numbers must not be changed in the course of computations (the standard MPI-2 provides some means to change dynamically the number of executed processes). All the program processes are sequentially enumerated from 0 to $p-1$, where np is the total number of processes. The process number is referred to as the *process rank*.

4.1.2. Data Communication Operations

Data communication operations form the basis of MPI. Among the functions provided within MPI they usually differentiate between *point-to-point* operations, i.e. operations between two processes, and *collective* ones, i.e. communication actions for the simultaneous interaction of several processes.

Various *transmission modes* can be used to carry out point-to-point operations. Among them there are the synchronous mode, the blocking mode etc. A complete survey of possible transmission modes may be found in 4.3.

As it has been previously mentioned, MPI standard provides for the necessity to realize the majority of the basic collective data transmission operations - see subsection 4.2 and 4.4.

4.1.3. Communicators

Parallel program processes are united into *groups*. *The communicator* in MPI is a specially designed service object, which unites within itself a group of processes and a number of complementary parameters (*context*), which are used in carrying out data transmission operations.

As a rule, point-to-point data transmission operations are carried out for the processes, which belong to the same communicator. Collective operations are applied simultaneously to all the processes of the communicator. As a result, it is necessary without fail to point to the communicator being used for data transmission operations in MPI.

In the course of computations new groups may be created and the already existing groups of processes and communicators may be deleted. The same process may belong to different groups and communicators. All the processes available in a parallel program belong to the communicator with the identifier MPI_COMM_WORLD, which is created on default.

If it is necessary to transmit the data among the processes, which belong to different groups, an intercommunicator should be created.

The possibilities of MPI for operations with groups and communicators are described in detail in 4.6.

4.1.4. Data Types

It is necessary to point to the type of the transmitted data in MPI functions while carrying out data transmission operations. MPI contains a wide set of the basic data types. These data types largely coincide with the data types of the algorithmic languages C and Fortran. Besides this, MPI has possibilities for creating new derived data types for more accurate and precise description of the transmitted message content.

The possibilities of MPI for operations with the derived data types are described in detail in 4.5.

4.1.5. Virtual Topologies

As it has been mentioned previously, point-to-point data transmission operations may be carried out among any processors of the same communicator; all the processors of the communicator take part in collective operations. In this respect, the logical topology of the communication lines among the processes is a complete graph (regardless of the availability of real physical communication channels among the processors).

It is also useful (as it has been mentioned in Section 3) to logically present the available communication network as a topology for further description and analysis of a number of parallel algorithms.

MPI provides an opportunity to present a number of processes as a grid of arbitrary dimension (see Figure 1.7). The boundary processes of the grids may be referred to as neighboring, and thus, the structures of torus type may be constructed on the basis of the grids.

Moreover, MPI provides for the possibility to form logical (virtual) topologies of any desirable type. The possibilities of MPI for operations with topologies are discussed in 4.7.

And some final remarks before we start to study MPI in detail:

- the function descriptions and the program examples are given in the algorithmic language C; the peculiarities of using MPI for the algorithmic language Fortran are described in 4.8.1,
- a brief description of the available MPI library realizations and general description of MPI program execution environment are given in 4.8.2,
- the possibilities of MPI are described for standard version 1.2 (*MPI-1*); some additional characteristics of standard version 2.0 are given in 4.8.3.

Proceeding to the study of MPI it should be noted that on the one hand MPI is quite complicated, as the MPI standard provides for more than 125 functions. On the other hand, MPI structure is elaborated, so the development of parallel programs may be started after consideration of only 6 MPI functions. All the additional possibilities of MPI may be studied as the complexity of the developed algorithms and programs increases. It is exactly the way we will present the material proceeding from simple issues to more complicated ones.

4.2. Introduction to MPI Based Development of Parallel Programs

4.2.1. The Fundamentals of MPI

Let us describe the minimum necessary set of MPI functions sufficient for the development of rather simple parallel programs.

4.2.1.1 MPI Programs Initialization and Termination

The first MPI function, which has to be called, must be the following:

```
int MPI_Init ( int *agrc, char ***argv ).
```

It is called to initialize MPI program execution environment. The parameters of the function are the number of arguments in the command line and the command line text.

The last MPI function to be called must be the following one:

```
int MPI_Finalize (void).
```

As a result, it may be noted that the structure of the MPI-based parallel program should look as follows:

```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
  <program code without the use of MPI functions>
  MPI_Init ( &agrc, &argv );
    <program code with the use of MPI functions>
  MPI_Finalize();
  <program code without the use of MPI functions>
  return 0;
}
```

It should be noted:

1. File *mpi.h* contains the definition , function prototypes and data types of MPI library,
2. Functions *MPI_Init* and *MPI_Finalize* are mandatory and should be executed (only once) by each process of the parallel program,

3. Before *MPI_Init* is called, the function *MPI_Initialized* may be used to determine whether *MPI_Init* has been called or not.

The examples of the functions, which have been discussed, allow us to understand the syntax of MPI function naming. Prefix MPI precedes the name of the function. Further, there are one or more words of the function name. The first word in the function name begins with the capital symbol. The words are separated by the underline character. The names of MPI functions, as a rule, explain the designation of the operations carried out by the functions.

4.2.1.2 Determining the Number and the Rank of the Processes

The number of the processes in the parallel program being executed is carried out by means of the following function:

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

The following function is used to determine the process rank:

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

As a rule, the functions *MPI_Comm_size* and *MPI_Comm_rank* are called right after *MPI_Init*:

```
#include "mpi.h"
int main ( int argc, char *argv[] ) {
  int ProcNum, ProcRank;
  <program code without the use of MPI functions>
  MPI_Init ( &agrc, &argv );
  MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank);
    <program code with the use of MPI functions>
  MPI_Finalize();
  <program code without the use of MPI functions>
  return 0;
}
```

It should be noted:

1. Communicator *MPI_COMM_WORLD*, as it has been previously mentioned, is created on default and presents all the processes carried out by a parallel program,

2. The rank obtained by means of the function *MPI_Comm_rank* is the rank of the process, which has called this function, i.e. the variable *ProcRank* will accept different values in different processes.

4.2.1.3 Message Passing

In order to transmit data, the sending process should carry out the following function:

```
int MPI_Send(void *buf, int count, MPI_Datatype type, int dest,
  int tag, MPI_Comm comm),
where
 - buf   – the address of the memory buffer, which contains the data
           of the message to be transmitted,
 - count – the number of the data elements in the message,
 - type  - the type of the data elements of the transmitted message,
 - dest  - the rank of the process, which is to receive the message,
 - tag   - tag-value, which is used to identify messages,
 - comm  - the communicator, within of which the data is transmitted.
```

There are a number of basic types to refer to the transmitted data in MPI. The complete list of the basic types is given in Table 4.1.

Table 4.1. The basic (predefined) MPI data types for the algorithmic language C

MPI_Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float

MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

The following issues should be taken into account:

1. The message to be passed is defined by pointing to the memory block (buffer), which contains the message. The following triad is used to point to the buffer

```
( buf, count, type )
```

This triad is included into the parameters of practically all data passing functions,

2. The processes, among which data is passed, should belong to the communicator, specified in the function *MPI_Send*,

3. The parameter *tag* is used only when it is necessary to distinguish the messages being passed. Otherwise, an arbitrary integer number may be used as the parameter value (see also the description of the function MPI_*Recv*).

Immediately after the completion of the function *MPI_Send* the sending process may start to use repeatedly the memory buffer, in which the transmitted message was located. But it should be noted that the state of the message being passed may be quite different at the moment of the function MPI_Send termination. The message may be located in the sending process, it may be being transmitted, it may be stored in the receiving process, or may be received by the receiving process by means of the function MPI_Recv. Thus, the termination of the function MPI_Send means only that the data transmission operation has started to be carried out, and message passing will sooner or later be completed.

The example of this function use will be presented after the description of the function MPI_Recv.

4.2.1.4 Message Reception

In order to receive messages the receiving process should carry out the following function:

```
int MPI_Recv(void *buf, int count, MPI_Datatype type, int source,
  int tag, MPI_Comm comm, MPI_Status *status),
where
 - buf, count, type – the memory buffer for message reception, the
      designation of each separate parameter corresponds to the
      description in MPI_Send,
 - source – the rank of the process, from which message is to be
            received,
 - tag    - the tag of the message, which is to be received for the
            process,
 - comm   - communicator, within of which data is passed,
 - status – data structure, which contains the information of the results
            of carrying out the data transmission operation.
```

The following aspects should be taken into account:

1. Memory buffer should be sufficient for data reception and the element types of the transmitted and the received message must coincide. In case of memory shortage a part of the message will be lost and in the return code of the function termination there will be an overflow error registered,

2. The value *MPI_ANY_SOURCE* may be given for the parameter source, if there is a need to receive a message from any sending process,

3. If there is a need to receive a message with any tag, then the value *MPI_ANY_TAG* may be given for the parameter *tag*,

4. The parameter *status* makes possible to define a number of characteristics of the received message:

```
    - status.MPI_SOURCE – the rank of the process, which has sent the
received message,
    - status.MPI_TAG    - the tag of the received message.
```

The function

```
MPI_Get_count(MPI_Status *status, MPI_Datatype type, int *count)
```

returns in the variable *count* the number of elements of datatype *type* in the received message.

The call of the function *MPI_Recv* does not have to be in agreement with the time of the call of the corresponding message passing function *MPI_Send*. The message reception may be initiated before the moment, at the moment or even after the starting moment of sending the message.

After the termination of the function *MPI_Recv* the received message will be located in the specified buffer memory. It is essential that the function *MPI_Recv* is a *blocking* one for the receiving process, i.e. carrying out of the process is suspended till the function terminates its operation. Thus, if due to any reason the expected message is missing, then the parallel program execution will be blocked.

4.2.1.5 The First MPI Based Parallel Program

The described set of functions appears to be sufficient for parallel program development[2]. The program, which is given below, is standard example for the algorithmic language C.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[]){
  int ProcNum, ProcRank, RecvRank;
  MPI_Status Status;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
  if ( ProcRank == 0 ){
    // Code is executed only by the process of rank 0
    printf ("\n Hello from process %3d", ProcRank);
    for ( int i=1; i<ProcNum; i++ ) {
      MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
      printf("\n Hello from process %3d", RecvRank);
    }
  }
  else
    // Message is sent by all the processes except
    // the process of rank 0
    MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

Program 4.1. The first MPI based parallel program

As it is clear from the program text, each process identifies its rank, and after that all the operations in the program are separated. All the processes, except the process of rank 0, send the value of its rank to the process 0. The process of rank 0 first prints the value of its rank and then prints ranks of all the other processes. It should be noted that the order of message reception is not predetermined. It depends on the execution conditions for parallel program (moreover, the order may change in several sequential program executions). Thus, a possible variant of the process 0 print results may consist in the following (for the parallel program of 4 processes):

```
Hello from process 0
Hello from process 2
Hello from process 1
Hello from process 3
```

This “changeable” type of the obtained results complicates the developing, testing and debugging parallel programs, as one of the main programming principles, i.e. ”the reproducibility” of the executed computational experiments, does not work in this case. As a rule, if it does not lead to efficiency losses, it is necessary to provide the unambiguity of computations in case of parallel computations. Thus, for the simple example being discussed it is possible to provide the permanence of the obtained result using the rank of the sending process in the message reception operation:

---

[2] As it has been stated previously, the number of MPI functions, which are necessary to start the parallel program development, appears to be equal to six.

```
MPI_Recv(&RecvRank, 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD, &Status).
```

The rank of the sending process regulates the order of message reception. As a result, the lines of printing will appear strictly in the order of increasing the process ranks (we should remind that this regulation in certain situations may slow down the parallel computations).

One more important issue should be mentioned. An MPI based program both in this particular variant and in the most general case is used for generating all the processes of the parallel program. As a result, it should define the computations carried out on all these processes. It is possible to say that an MPI based program is a certain "*macro code"*. Different parts of this code are used by different processes. Thus, for instance, in the example being discussed, the parts of the program code marked by double frame are not used simultaneously in either of the processes. The first marked part with the sending function MPI_Send is used only by the process of rank 0. The second part with the reception function MPI_Recv is used by all the processes except the process of rank 0.

The approach, which is used in the above described program for distributing the code fragments among the processes, can be described as follows. First, the process rank is defined by means of the function *MPI_Comm_rank*. The program code parts necessary for the process are selected in accordance with the rank. The availability of code fragments of different processes in the same program complicates significantly understanding and developing an MPI based program. It is recommended to carry out the program code of different processes into separate program modules (functions), if the amount of the developed programs is significant. The general scheme of an MPI based program in this case looks as follows:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoProcess0();
else if ( ProcRank == 1 ) DoProcess1();
else if ( ProcRank == 2 ) DoProcess2();
```

In many cases, as in the above described examples, the operations differ only for the process of rank 0. In this case the general scheme of an MPI based program can be simpler:

```
MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
if ( ProcRank == 0 ) DoManagerProcess();
else DoWorkerProcesses();
```

In conclusion let us comment on the approach, which is used in MPI to control the correctness of function execution. All the MPI functions return the termination code as their value. If the function is completed successfully the return code is *MPI_SUCCESS*. The other values of the termination code testifies to the fact that some errors have been discovered in the course of function execution. To find out the type of the discovered error predetermined named constants are used. Among these constants there are the following ones:

```
- MPI_ERR_BUFFER – incorrect buffer pointer,
- MPI_ERR_COMM – incorrect communicator,
- MPI_ERR_RANK – incorrect process rank,
```

and others. The complete list of the constants for checking the termination code is in file *mpi.h*.

4.2.2. The Determination of the MPI Based Program Execution Time

The need for determining the execution time to estimate the speedup of parallel computations arises practically after the development of the first parallel program. The means, which are used to measure the execution time depend, as a rule, on the hardware, the operating system, the algorithmic language and so on. The MPI standard includes the definition of special functions for measuring time. The use of these functions makes possible to eliminate the dependence on the environment of the parallel program execution.

Obtaining time of the current moment of the program execution is provided by means of the following function:

```
double MPI_Wtime(void).
```

The result of its call is the number of seconds, which have passed since a certain moment in the past. This moment of time in the past may depend on the environment of MPI library implementation and thus, the function MPI_Wtime should be used in order to eliminate this dependence only for determining the duration of execution of parallel program code fragments. A possible scheme of application MPI_Wtime function may consist in the following:

```
double t1, t2, dt;
t1 = MPI_Wtime();
…
t2 = MPI_Wtime();
dt = t2 – t1;
```

The accuracy of time measurement may depend on the environment of the parallel program execution. The following function may be used in order to determine the current value of accuracy:

```
double MPI_Wtick(void).
```

This function allows to measure in seconds the time between two sequential ticks of time of the computer system hardware timer.

4.2.3. Introduction to the Collective Data Transmission Operations

The functions *MPI_Send* and *MPI_Recv*, discussed in 4.2.1, provide the possibility of executing data passing operations between two parallel program processes. To execute collective communication operations characterized by participating of all the processes in a communicator, MPI provides a special set of functions. This subsection discussed three functions, which are widely used even in the development of comparatively simple parallel programs. Collective operations are discussed in 4.4.

To demonstrate the example of MPI function applications we will use the problem of summing up vector x elements (see 2.5)

$$S = \sum_{i=1}^{n} x_i \ .$$

The development of parallel algorithm for solving this problem is not complicated. It is necessary to divide the data into equal blocks, to transmit these blocks to the processes, to carry out the summation of the obtained data in the processes, to collect the values of the computed partial sums on one of the processes and to add the values of partial sums to obtain the total result of the problem. In further development of the demontrational programs this algorithm will be simplified. All the vector being summed and, not only separate blocks of the vector, will be transmitted to the program processes.

4.2.3.1 Data Broadcasting

The first problem, which arises in the execution of the above discussed parallel algorithm is the need for transmitting vector x values to all the parallel program processes. Of course, it is possible to use the above discussed data transmission functions for solving the problem:

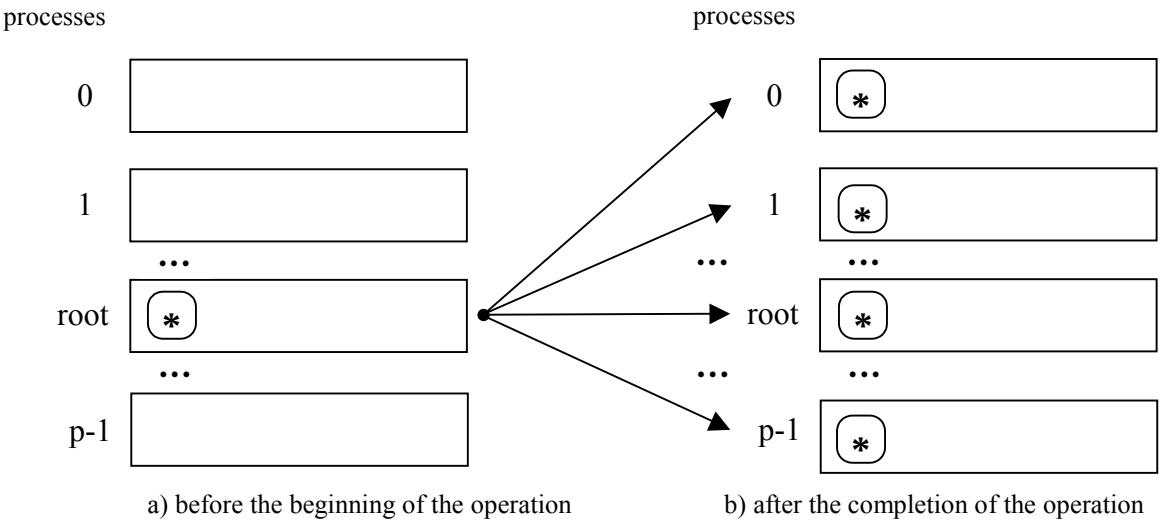
```
MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
for (i=1; i<ProcNum; i++)
  MPI_Send(&x,n,MPI_DOUBLE,i,0,MPI_COMM_WORLD);
```

However, this solution appears to be inefficient, as the repetition of the data transmission operations leads to summing up the expenses (latencies) on the preparation of the transmitted messages. Besides, as it has been shown in section 3, this operation may be executed in $log_2 p$ data transmission iterations.

To achieve efficient broadcasting the following MPI function may be used:

```
int MPI_Bcast(void *buf,int count,MPI_Datatype type,int root,MPI_Comm comm),
where
  - buf, count, type – memory buffer, which contains the transmitted message
      (for the process 0) and for message reception for the rest of the
      processes,
  - root – the rank of the process, which carries out data broadcasing,
  - comm – the communicator, within of which data broadcasting is executed.
```

The function *MPI_Bcast* carries out transmitting the data from the buffer *buf*, which contains *count* type elements from the process with the *root* number to the processes within the communicator *comm* – see Figure 4.1



a) before the beginning of the operation b) after the completion of the operation

Figure 4.1. The general scheme of the data broadcasting operation

The following aspects should be taken into consideration:

1. The function *MPI_Bcast* defines the collective operation, and thus, the call of the function MPI_Bcast, when the necessary data should be transmitted, is to be executed by all the processes of a certain communicator (see further the example of the program),

2. The memory buffer defined in the function MPI_Bcast has different designations in different processes. For the root process, from which data broadcasting is performed, this buffer should contain the transmitted message. For the rest of the processes the buffer is assigned for data reception.

We will give the example of the program for solving the problem of vector elements summation with the use of the above described function.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
int main(int argc, char* argv[]){
  double x[100], TotalSum, ProcSum = 0.0;
  int  ProcRank, ProcNum, N=100;
  MPI_Status Status;

  //initialization
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD,&ProcRank);

  // data preparation
  if ( ProcRank == 0 ) DataInitialization(x,N);

// data broadcast
  MPI_Bcast(x, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  // computing the partial sum on each of the processes
  // vector x elements from i1 to i2 are summed at each process
  int k = N / ProcNum;
  int i1 = k *   ProcRank;
  int i2 = k * ( ProcRank + 1 );
  if ( ProcRank == ProcNum-1 ) i2 = N;
  for ( int i = i1; i < i2; i++ )
    ProcSum  = ProcSum + x[i];

  // collecting partial sums on the process 0
  if ( ProcRank == 0 ) {
    TotalSum = ProcSum;
    for ( int i=1; i < ProcNum; i++ ) {
```

```
      MPI_Recv(&ProcSum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD,
        &Status);
      TotalSum = TotalSum + ProcSum;
    }
  }
  else // all the processes send their partial sums
    MPI_Send(&ProcSum, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);

  // result outprint
  if ( ProcRank == 0 )
    printf("\nTotal Sum = %10.2f",TotalSum);
  MPI_Finalize();
}
```

Program 4.2. The parallel program for summing up the numerical values

In the described program the function *DataInitialization* performs the preparation of the initial data. The necessary data may be input from the keyboard, read from a file, or generated by means of a random number generator. The preparation of the function is given to the reader for individual work.

4.2.3.2 Data Transmission from All the Processes to a Process. Reduction Operation.

The procedure of collecting and further data summation available in the above described program is an example of the high frequency operation of transmitting data from all the processes to a process. Different kinds of data processing are carried out over the collected values in this operation (to emphasize this fact the operation is often called as *data reduction*). As previously, the implementation of reduction by means of usual point-to-point data transmission operations appears to be inefficient and rather time-consuming. To execute data reduction in the best way possible, MPI provides the following function:

```
  int MPI_Reduce(void *sendbuf, void *recvbuf,int count,MPI_Datatype type,
                   MPI_Op op,int root,MPI_Comm comm),
where
  - sendbuf – memory buffer with the transmitted message,
  - recvbuf – memory buffer for the resulting message (only for the root
      rank process),
  - count   - the number of elements in the messages,
  - type    – the type of message elements,
  - op      - the operation, which should be carried out over the data,
  - root    - the rank of the process, on which the result must be obtained,
  - comm    - the communicator, within of which the operation is executed.
```

The operations predetermined in MPI (see Table 4.2) may be used as data reduction operations.

Table 4.2. The basic (predetermined) MPI operation types for data reduction functions

Operation	Description
MPI_MAX	The maximum value determination
MPI_MIN	The minimum value determination
MPI_SUM	The Determination of the sum of the values
MPI_PROD	The determination of the product of the values
MPI_LAND	The execution of the logical operation “AND” over the message values
MPI_BAND	The execution of the bit operation “AND” over the message values
MPI_LOR	The execution of the logical operation “OR” over the message values
MPI_BOR	The execution of the bit

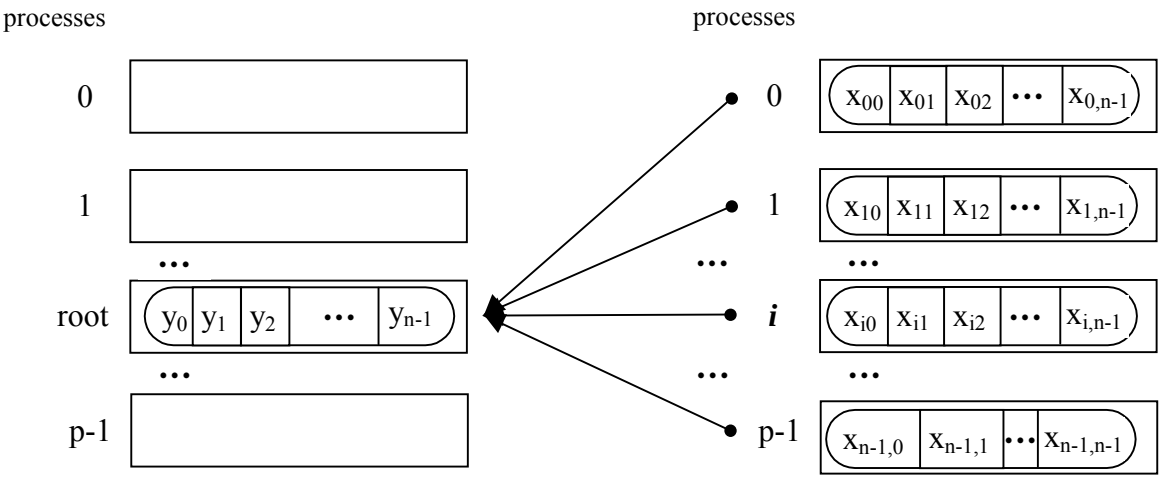
	operation “OR” over the message values
MPI_LXOR	The execution of the excluding logical operation “OR” over the message values
MPI_BXOR	The execution of the excluding bit operation “OR” over the message values
MPI_MAXLOC	The determination of the maximum values and their indices
MPI_MINLOC	The determination of the minimum values and their indices

There may be other complementary operations determined directly by the user of the MPI library besides this given standard operation set (see, for instance, Group, et al. (1994), Pacheco (1996).

The general scheme of the execution of data collecting and processing operation on a processor is shown in Figure 4.2. The elements of the received message on the root process are the results of processing the corresponding elements of the messages transmitted by the processes, i.e.

$$y_j = \overset{n-1}{\underset{i=0}{\otimes}} x_{ij}, 0 \le j < n ,$$

where $\otimes$ is the operation, which is set when the function *MPI_Reduce* is called (to clarify this we present an example of the execution of data reduction operation in Figure 4.3.).



a) after the termination of the operation b) before the beginning of the operation

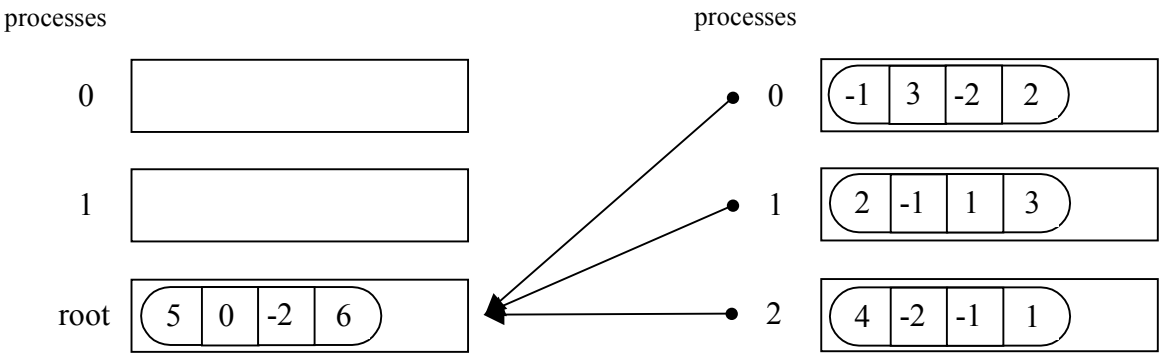
Figure 4.2. The general scheme of collecting and processing the data on a processor from all the other processes

The following aspects should be taken into consideration:

1. The function *MPI_Reduce* defines the collective operation, and thus, the call should be carried out by all the processes of the specified communicator. All the calls should contain the equal values of the parameters *count*, *type*, *op*, *root*, *comm,*

2. The data transmission should be carried out by all the processes. The operation result will be obtained only by the root rank process,

3. The execution of the reduction operation is carried out over separate elements of the transmitted messages. Thus, for instance, if messages contain two data elements each, and the summation operation MPI_SUM is executed, then the result will consist of the two values: the first will contain the sum of the first elements of all the transmitted messages, the second one will be equal to the sum of all the second message elements correspondingly.



a) after the termination of the operation b) before the beginning of the operation

Figure 4.3. The example of reduction in summing up the transmitted data for three processes (each message contains 4 elements, the messages are collected on rank 2 process)

Using the function *MPI_Reduce* to optimize the previously described program of summation the fragment marked by the double frame can be rewrite in the following form:

```
// collecting of partial sums on 0 rank process
MPI_Reduce(&ProcSum,&TotalSum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

4.2.3.3 Computation Synchronization

Sometimes the computational processes executed independently have to be sinchronized. Thus, for instance, to measure the starting time of the parallel program operation it is necessary to complete all the preparatory operations for all the processes simultaneously. Before the program termination all the processes should complete their computations etc.

Process synchronization, i.e. simultaneous achieving certain points of computations by various processes is provided by means of the following MPI function

```
int MPI_Barrier(MPI_Comm comm);
```

The function *MPI_Barrier* defines collective operation. Thus, this function should be called by all the processes of the communicator. When the function *MPI_Barrier* is called, the process execution is blocked. The computations of the process will continue only after the function *MPI_Barrier* is called by all the processes of the communicator.

4.3. Data Transmission between Two Processes

In this subsection we will continue the discussion of MPI functions for transmitting data between parallel program processes, which was partially described in 4.2.1.

4.3.1. Data Transmission Modes

The function *MPI_Send* provides the so called *Standard* message transmission mode, when (see also 4.2.1):

- the sending process is blocked during the time of the function execution,
- the buffer may be used repeatedly after the function termination,
- the state of the transmitted message after the function termination may be different, i.e. the message may be located in the sending process, may be being transmitted, may be stored in the receiving process, or may be received by the receiving process by means of the function *MPI_Recv*.

There are the following additional message transmission modes available besides the standard one:

- *The Synchronous mode* consists in the following: the message transmission function is terminated only when the confirmation of the reception beginning for the transmitted message comes from the receiving process; the transmitted message is either completely received by the receiving process, or is being received,
- *The Buffered mode* assumes the use of additional system buffers, which are used for copying the transmitted messages in them. As a result, the function of message sending is terminated immediately after the message has been copied in the system buffer,
- *The Ready mode* may be used only if the message reception operation has already been initiated. The message buffer may be repeatedly used after the termination of the message sending function.

To name the message sending function for different execution modes, the initial symbol of the corresponding mode name is added to the name of the function *MPI_Send* as its prefix, i.e.

```
- MPI_Ssend – the function of sending message in the synchronous mode,
- MPI_Bsend – the function of sending message in the buffered mode,
- MPI_Rsend – the function of sending message in the ready mode.
```

The list of all the parameters for the enumerated functions coincides with those of the *MPI_Send* function parameters.

To use the buffered transmission mode, the memory buffer for buffering messages should be created and linked with MPI. The function, which is used for that, looks as follows:

```
int MPI_Buffer_attach(void *buf, int size),
where
  - buf  - the memory buffer for buffering messages,
  - size – buffer size.
```

After all the operations with the buffer are terminated, it must be disconnected from MPI by means of the following function:

```
int MPI_Buffer_detach(void *buf, int *size).
```

The following recommendations may be given on the practical use of the modes:

1. The ready mode is formally the fastest of all, but it is used rather seldom, as it is usually quite difficult to provide the readiness of the reception,
2. The standard and the buffered modes can also be executed fast enough, but may lead to big resource expenses (memory). On the whole, they may be recommended for transmitting short messages,
3. The synchronous mode is the slowest of all, as it requires the confirmation of reception. At the same time, this mode is the most reliable one. It may be recommended for transmitting long messages.

It should be noted in conclusion, that there are no various modes of operation for the reception function *MPI_Recv*.

4.3.2. Nonblocking Communications

All the function of message communication, which have been previously discussed, are blocking. It means that they block the process execution before the called functions terminate their operations. At the same time a part of messages in executing parallel computations may be sent and received before the real need for the transmitted data emerges. In such situations it would be desirable to have the possibility to execute the functions of data exchange without blocking the processes in order to perform simultaneously the message transmission and the computations. Undoubtedly the nonblocking method of data exchange is more complicated. But if it is applied adequately, it will significantly decrease the efficiency losses for parallel computations, which arise because of rather slow (in comparison to the processor speed) communication operations.

MPI provides the possibility of the nonblocking execution of data transmission operations between two processes. The names of the nonblocking analogues are created of the corresponding function names by means of adding the prefix *I* (Immediate). The list of parameters of the nonblocking functions contains all the set of parameters of the original functions and the additional parameter *request* with the type *MPI_Request* (there is no parameter *status* in the function MPI_Irecv):

```
int MPI_Isend(void *buf, int count, MPI_Datatype type, int dest,
  int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Issend(void *buf, int count, MPI_Datatype type, int dest,
  int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Ibsend(void *buf, int count, MPI_Datatype type, int dest,
  int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irsend(void *buf, int count, MPI_Datatype type, int dest,
  int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype type, int source,
  int tag, MPI_Comm comm, MPI_Request *request)
```

The call of the nonblocking function causes the initialization of the requested transmission operation. After that the execution of the function is terminated and the process may continue its operations. The nonblocking function determines the variable *request* before its termination. This variable may be used further for checking the completion of the initialized exchange operation.

The state of the executed nonblocking data transmission operation may be checked by means of the following function:

```
int MPI_Test( MPI_Request *request, int *flag, MPI_status *status),
where
```

```
    - request – is the operation descriptor, which is defined when the
                nonblocking function is called,
    - flag    - is the result of checking (=true, if the operation is
                terminated),
    - status  - the result of the exchage operation execution (only for the
                terminated operation).
```

This operation is a nonblocking one, i.e. the process may check the state of the nonblocking exchange operation and continue its computations, if the operation is not completed according to the results of the check up. The following scheme of combining the computations and the execution of the nonblocking exchange operation is possible:

```
MPI_Isend(buf,count,type,dest,tag,comm,&request);
…
do {
  …
  MPI_Test(&request,&flag,&status)
} while ( !flag );
```

If it appears that the continuation of computations is impossible without obtaining the transmitted data, the following blocking operation of waiting for the operation termination may be used:

```
int MPI_Wait( MPI_Request *request, MPI_status *status).
```

Besides the above described ones, MPI contains a number of additional check up and waiting functions for nonblocking exchange operations:

```
   - MPI_Testall  - checking the termination of all  the enumerated exchange
operations,
   - MPI_Waitall  – waiting for the termination of all  the enumerated
exchange operations,
   - MPI_Testany  - checking the termination of at least one of the
enumerated exchange operations,
   - MPI_Waitany  – waiting for the termination of any of the enumerated
exchange operations,
   - MPI_Testsome – checking the termination of each enumerated exchange
operation,
   - MPI_Waitsome – waiting for termination of at least one of the enumerated
exchange operations and estimating the state of all the operations.
```

It is rather complicated to give a simple example of using the nonblocking functions. Matrix multiplication algorithms may be a good opportunity to study the above described function (see section 8).

4.3.3. The Simultaneous Transmission and Reception

A special form of data exchange among the processes is a frequent case of information interaction in parallel programs. In case of this data exchange it is necessary to send the data to some processes and, at the same time, to receive messages from the other processes in order to continue the computations. The simplest variant of this situation may be, for instance, the data exchange between two processes. The realization of such exchanges by means of usual data transmission operations is inefficient and time-consuming. Besides, this realization should guarantee the absence of deadlock situations, which might occur, for instance, when the two processes begin to transmit messages to each other using blocking data transmission functions.

Efficient simultaneous execution of data transmission and reception operations may be provided by means of the following MPI function:

```
int MPI_Sendrecv(void *sbuf,int scount,MPI_Datatype stype,int dest, int  stag,
                 void *rbuf,int rcount,MPI_Datatype rtype,int source,int rtag,
                 MPI_Comm comm, MPI_Status *status),
where
 - sbuf, scount, stype, dest,   stag – parameters of the transmitted message,
 - rbuf, rcount, rtype, source, rtag – parameters of the received message,
 - comm   - communicator, within of which the data transmission is executed,
 - status – data structure with the information about the operation execution.
```

As it follows from the description, the function *MPI_Sendrecv* transmits the message described by the parameters (*sbuf, scount, stype, dest, stag*), to the process with rank *dest* and receives the message into the buffer, which is defined by the parameters (*rbuf, rcount, rtype, source, rtag*), from the process with rank *source*.

Different buffers are used in the function *MPI_Sendrecv* for message transmission and reception. In case when the messages are of the same type, MPI is able to use a common buffer:

```
int MPI_Sendrecv_replace (void *buf, int count, MPI_Datatype type, int dest,
    int stag, int source, int rtag, MPI_Comm comm, MPI_Status* status).
```

An example of using functions for simultaneous execution of transmission and reception operations is given in section 8 for the development of parallel programs of matrix multiplication.

4.4. Collective Data Transmission Operations

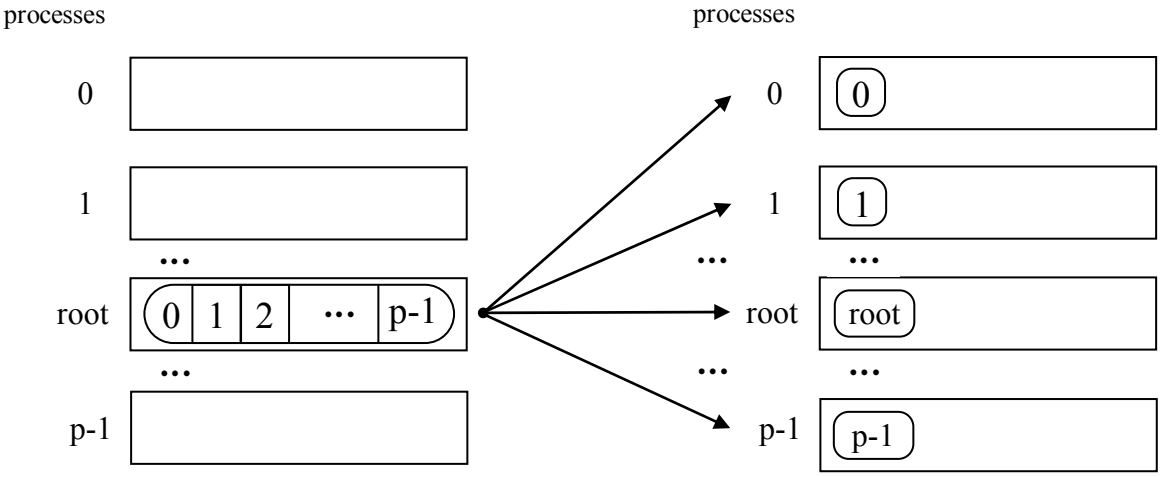
As it has been stated previously, MPI operations are called *collective*, if all the processes of the communicator take part in them. The basic types of collective operations are considered in section 3. A part of collective operations are described in 4.2.3. These are *broadcasting* operations and the operations of processing the data obtained on a process from all the other processes (*data reduction*).

Let us consider now the remaining basic collective data transmission operations.

4.4.1. Scattering Data from a Process to all the Processes

The difference between scattering data from a process to all the processes (*data distribution*) and the broadcasting operation is that in case of scattering data, the process transmits different data to the processes (see Figure 4.4). The execution of this operation may be provided by means of the following function:

```
int MPI_Scatter(void *sbuf,int scount,MPI_Datatype stype,
                void *rbuf,int rcount,MPI_Datatype rtype,
                int root, MPI_Comm comm),
where
 - sbuf, scount, stype – the parameters of the transmitted message
       (scount defines the number of elements transmitted to each process),
 - rbuf, rcount, rtype – the parameters of the message received in the
        processes,
 - root – the rank of the process, which performs data scattering,
 - comm – the communicator, within of which data scattering is performed.
```



a) before the beginning of the operation b) after the termination of the operation

Figure 4.4. The general scheme of scattering data from a process to all the other processes

When this function is called, the process with rank *root* transmits data to all the other processes in the communicator. *Scount* elements will be sent to each process. the process 0 will receive the data block of *sbuf* elements with the indices from 0 to *scount-1*; the process with rank 1 will receive the block of elements with the indices from *scount* to *2*scount-1* etc. Thus, the total size of the transmitted message should be equal to *scount*p* elements, where *p* is the number of the processes in the communicator *comm*.

It should be noted that as the function *MPI_Scatter* defines a collective operation, the call of this function in the execution of data scattering should be provided in each communicator process.

It should be also noted that the function *MPI_Scatter* transmits messages of the same size to all the processes. The execution of a more general variant of data distribution operation, when the message sizes for different processes may be different, is provided by means of the function *MPI_Scatterv*.

The use of the *MPI_Scatter* function is described in Section 7 for the development of the parallel programs of matrix-vector multiplication.

4.4.2. Gathering Data from All the Processes to a Process

Gathering data from all the processes to a process (*data gathering*) is reverse to data distribution (see Figure 4.5). The following MPI function provides the execution of this operation:

```
int MPI_Gather(void *sbuf,int scount,MPI_Datatype stype,
               void *rbuf,int rcount,MPI_Datatype rtype,
                int root, MPI_Comm comm),
where
 - sbuf, scount, stype – the parameters of the transmitted message,
 - rbuf, rcount, rtype – the parameters of the received message,
 - root – the rank of the process which performs data gathering,
 - comm – the communicator, within of which data transmission is executed.
```

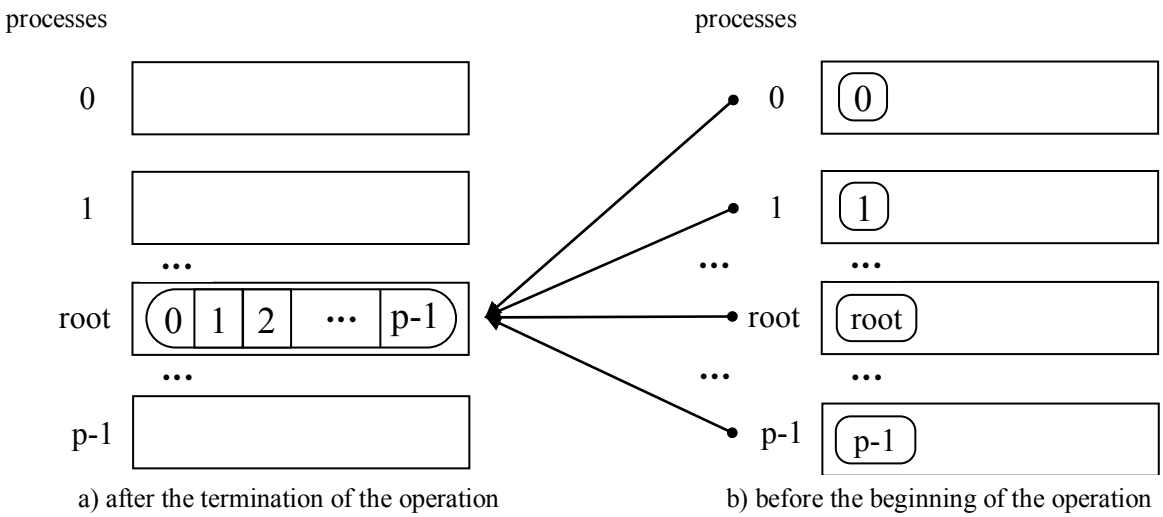


Figure 4.5. The general scheme of the data gathering from all the processes to a process

When the fuinction *MPI_Gather* is being executed, each process in the communicator transmits the data from the buffer *sbuf* to the process with rank *root*. The *root* rank process gathers all the transmitted data in the buffer *rbuf* (the data is located in the buffer in accordance with the ranks of the sending processes). In order to locate all the received data the buffer size *rbuf* should be equal to *scount*p* elements, where *p* is the number of the processes in the communicator *comm*.

The function *MPI_Gather* also defines a collective operation and its call in gathering the data must be provided in each communicator process.

It should be noted that when the *MPI_Gather* function is used, data gathering should be carried out only on one process. To obtain all the gathered data on each communicator process, it is necessary to use the function:

```
int MPI_Allgather(void *sbuf, int scount, MPI_Datatype stype,
                  void *rbuf, int rcount, MPI_Datatype rtype, MPI_Comm comm).
```

The execution of the general variant of data collecting operation, when the sizes of the messages transmitted among the processes may be different, is provided by means of the functions *MPI_Gatherv* and *MPI_Allgatherv*.

The use of function *MPI_Gather* is described in Section 7 for the development of parallel programs of matrix-vector multiplication.

4.4.3. All to All Data Transmission

All to all data transmission is the most general data transmission operation (see Figure 4.6). The execution of the operation is provided by the following function:

```
int MPI_Alltoall(void *sbuf,int scount,MPI_Datatype stype,
                 void *rbuf,int rcount,MPI_Datatype rtype,MPI_Comm comm),
where
 - sbuf, scount, stype – the parameters of the transmitted messages,
 - rbuf, rcount, rtype – the parameters of the received messages,
 - comm – the communicator, within of which the data transmission is executed.
```

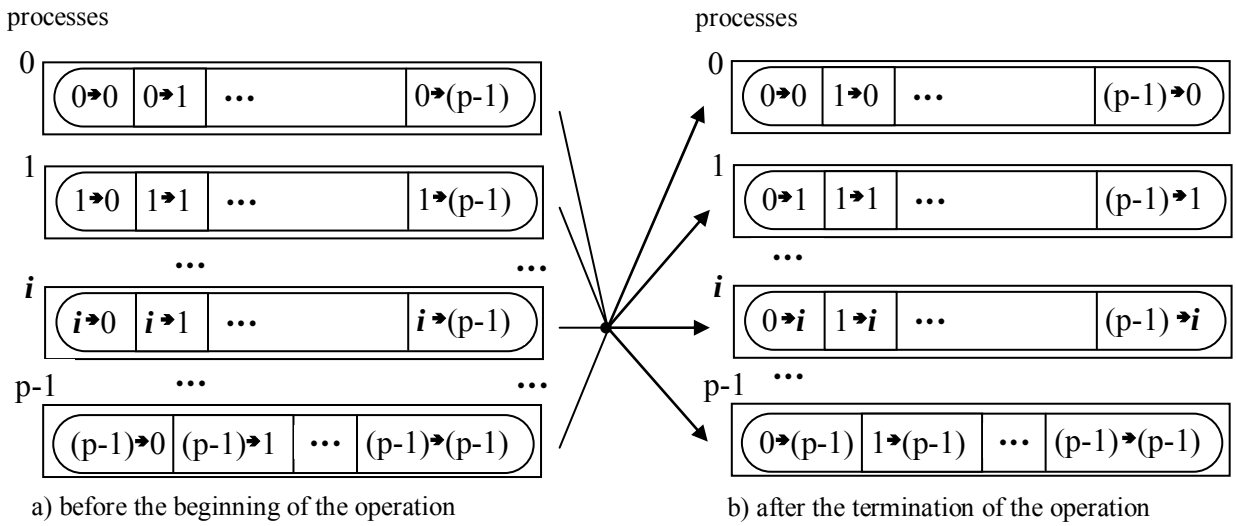


Figure 4.6. The general scheme of the all to all data transmission (the messages are shown by ***i→j***, where *i* and *j* are the ranks of the sending and receiving processes correspondingly)

When the function *MPI_Alltoall* is executed, each process in the communicator transmits the data of *scount* elements to each process (the total size of the transmitted messages in the processes should be equal to *scount*p* elements, where *p* is the number of the processes in the communicator *comm*) and receives the messages from each process.

The function *MPI_Alltoall* should be called in each communicator process during the execution of all-to-all data сщьыгтшсфешщт operation.

The variant of this operation in case when the sizes of the transmitted messages may be different is provided by means of the function *MPI_Alltoallv*.

The use of the function *MPI_Alltoall* in described in Section 7 for the development of parallel programs of matrix-vector multiplication as an exercise for individual work.

4.4.4. Additional Data Reduction Operations

The function *MPI_Reduce* described in 4.2.3.2 provides obtaining the results of data reduction only on one process. To obtain the data reduction results on each of the communicator processes, it is necessary to use the function *MPI_Allreduce*:

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

The function *MPI_AllReduce* performs the transmission of the reduction operation results among the processes. The possibility to control the distribution of the data among the processes is presented by the function *MPI_Reduce_scatter*.

One more variant of the operation of gathering and processing data, which provides obtaining also all the partial results of reduction, may be achieved by means of the following function:

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

The general scheme of the function *MPI_Scan* execution is shown in Figure 4.7. The elements of received messages are the results of processing the corresponding elements of transmitted messages. To obtain the results on the process with the rank *i*, $0 \le i < n$, the data from the processes with the ranks smaller or equal to *i* should be used, i.e.

$$y_{ij} = \bigotimes_{k=0}^{i} x_{kj}, 0 \le i, j < n,$$

where ⊗ is the operation, which is set when the function *MPI_Scan* is called.

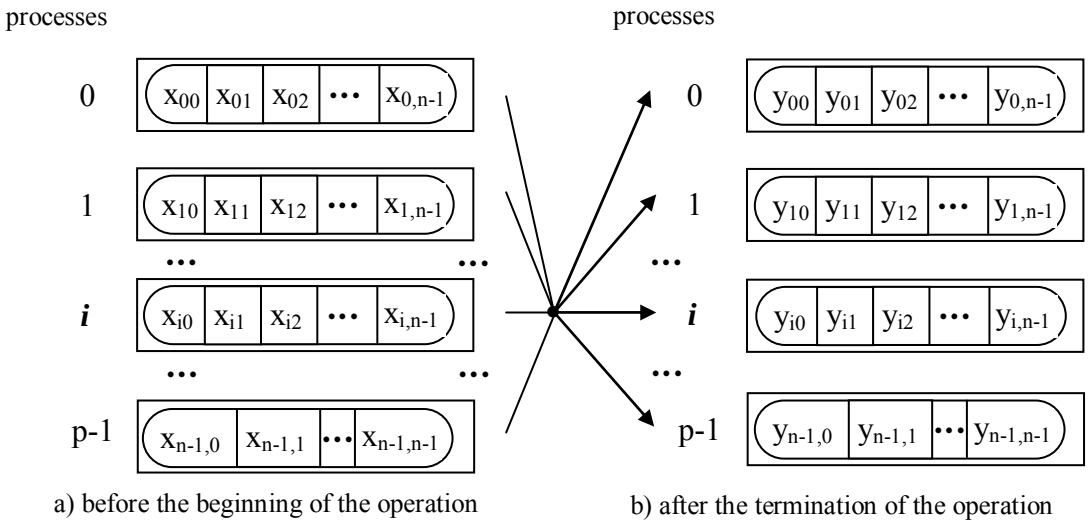


Figure 4.7. The general scheme of reduction operation including obtaining partial results of data processing

4.4.5. The Total List of Collective Data Operations

For simplicity all the information of collective data operations discussed above may be presented in the form of the Table 4.3.

Table 4.3. The list of the collective data operations and their description

The type of the collective operation	The general description and complexity estimation	The MPI function	The examples of application
Broadcasting from a process to all the processes	3.2.5	**MPI_Bcast** 4.2.3.1	4.2.3.1
Collecting and processing data on a process from all the processes (*data reduction*)	3.2.5, 3.2.6	**MPI_Reduce** 4.2.3.2	4.2.3.2
- the same including scattering the results to all the processes	3.2.5, 3.2.6	**MPI_Allreduce** **MPI_Reduce_scatter** 4.4.4	
- the same including obtaining partial results of data processing	3.2.5, 3.2.6	**MPI_Scan** 4.4.4	
Scattering data from a process to all the processes (*data distribution*)	3.2.7	**MPI_Scatter** **MPI_Scatterv** 4.4.1	Section 7
Gathering data from all the processes to a process (*gathering data*)	3.2.7	**MPI_Gather** **MPI_Gatherv** 4.4.2	Section 7
- the same including scattering results to all the processes	3.2.7	**MPI_Allgather** **MPI_Allgatherv** 4.4.2	
All to all data transmission	3.2.8	**MPI_Alltoall** **MPI_Alltoallv** 4.4.3	Section 7

4.5. The Derived Data Types in MPI

In all the above considered examples of the data transmission functions it was assumed, that the messages are a certain continuous vector of the elements of the type predetermined in MPI (the list of types available in MPI is

given in Table 4.1). It is obvious that the data necessary to be transmitted may not be located close to each other and contain the values of different types. Certainly, even in such situations the data may be transmitted using several messages. But this method will not be efficient because of the summation of latency of several executed data transmission operations. The other possible approach may be to pack the transmitted data into the format of a continuous vector. However, in this case there also be excessive operations of copying the data. Besides this, such operations are not easy to understand.

To provide more possibilities to determine the contents of the transmitted messages, MPI supports the mechanism of the so called *derived data types*. The basic concept of this approach, the possible methods of constructing the derived data types and the functions of packing and unpacking data are described in the further subsections.

4.5.1. The Derived Data Type

In the most general form the *derived data type* in MPI is the description of a set of the values of the predetermined in MPI type. The described values are not necessarily located continuously in the memory. The type is usually defined in MPI by means of the *type map* in the form of the sequential descriptions of values included into the type. Each separate value is described by pointing to the type and offset of location address from a certain origin address, i.e.

$$\mathbf{TypeMap} = \{(type_0, disp_0),...,(type_{n-1}, disp_{n-1})\} .$$

The part of the type map, which contains only the types of values, is called in MPI a *type sugnature*:

$$\mathbf{TypeSignature} = \{type_0,...,type_{n-1}\}$$

The type signature specifies which basic data types create a certain derived data type in MPI. Thus, the type signature controls the interpretation of data elements in transmitting or receiving messages. The type map offsets make possible to define, where the data values are located.

Let us explain this concept by the following example. Let the message include the following variables:

```
double a; /* address 24 */
double b; /* address 40 */
int    n; /* address 48 */
```

Then the derived type for the description of the data should have the map of the following form:

```
{(MPI_DOUBLE,0),
 (MPI_DOUBLE,16),
 (MPI_INT,24)
}
```

The following number of new concepts is used in MPI for the derived data types:

- *the lower boundary of type*

$$lb(\mathbf{TypeMap}) = \min_j(disp_j),$$

-*the upper boundary of type*

$$ub(\mathbf{TypeMap}) = \max_j(disp_j + sizeof(type_j)) + \Delta,$$

- *the extent of type*

$$extent(\mathbf{TypeMap}) = ub(\mathbf{TypeMap}) - lb(\mathbf{TypeMap}).$$

According to the definition the *lower boundary* is the offset of the considered data type values for the first byte. The *upper boundary* is correspondingly the offset for the byte, which is located next to the last element of the data type to be considered. The offset value for the upper boundary may be rounded up with regard to the requirements of address alignment. Thus, one of the most general requirements, imposed by the realization of the languages C and Fortran, is that the element address should be divisible by the element length in bytes. For instance, if the type *int* occupies four bytes, then the address for the element of *int* type should be completely divisible by four. This is the requirement, which is reflected in the determination of the upper boundary of the MPI data type. This aspect may be explained using the previously described example of the set of variables *a, b* and *n*. The lower boundary for the set is equal to zero, while the upper boundary is equal to 32 (the value of rounding is 6 or 4 depending on the size of *int* type). It should be noted that the required alignment is defined by the type of the first data element in the type map.

It should be also noted that the concept of the extent and the concept of *the type size* are different notions. The extent is the amount of memory that should be allocated for a derived type element. The size of the data type is the number of bytes occupied by the data (the difference between the addresses of the last and the first data bytes). The difference between the values of the extent and the size is the round value for the address alignment. Thus, in this example the type size is equal to 28, and the extent is equal to 32 (it is assumed that the *int* type occupies 4 bytes).

MPI provides the following functions for obtaining the values of the extent and the type size:

```
int MPI_Type_extent ( MPI_Datatype type, MPI_Aint *extent ),
int MPI_Type_size   ( MPI_Datatype type, MPI_Aint *size ).
```

The lower and the upper boundaries of the types may be determined by means of the following functions:

```
int MPI_Type_lb ( MPI_Datatype type, MPI_Aint *disp ),
int MPI_Type_ub ( MPI_Datatype type, MPI_Aint *disp ).
```

The function of calculating the address of the variable is essential in constructing the derived types:

```
int MPI_Address ( void *location, MPI_Aint *address )
```

(it should be noted that this function is portable variant of the means for obtaining addresses in the algorithmic languages C and Fortran).

4.5.2. The Methods of Constructing the Derived Data Types

The following methods of constructing the derived data types are available in MPI:

- ***The contiguous method*** makes possible to define a continuous set of the elements of some data type as a new derived type,
- ***The vector method*** provides creating a new derived type as a set of elements of some existing type among the elements of which there exist regular memory intervals. The size of the intervals is defined in the number of the elements of the initial type, while in case of ***H-vector method*** this size is given in bytes,
- ***The index method*** differs from the vector method as the intervals between the elements of the initial type may be irregular,
- ***The structural method*** provides the most general description of the derived type by pointing directly to the type map of the created data type.

These methods of constructing the derived data types are described in detail below.

4.5.2.1 The Contiguous Constructing Method

The following function is used in MPI in case of the contiguous method of constructing the derived data type:

```
int MPI_Type_contiguous(int count,MPI_Data_type oldtype,MPI_Datatype *newtype).
```

As it follows from the description, the new type *newtype* is composed as *count* elements of the initial type *oldtype*. For instance, if the type map of the initial data type is the following

```
{ (MPI_INT,0),(MPI_DOUBLE,8) },
```

then the call of the function *MPI_Type_contiguous* with the following parameters

```
MPI_Type_contiguous (2, oldtype, &newtype);
```

will cause the creation of the new data type, the type map of which looks as follows:

```
{ (MPI_INT,0),(MPI_DOUBLE,8),(MPI_INT,16),(MPI_DOUBLE,24) }.
```

To an extent the contiguous constructing method is excessive, as the use of the argument *count* in MPI procedures is equal to the use of the continuous data type of the same size.

4.5.2.2 The Vector Constructive Method

The following function is used in case of the vector constructing method:

```
int MPI_Type_vector ( int count, int blocklen, int stride,
  MPI_Data_type oldtype, MPI_Datatype *newtype ),
where
 - count    – the number of blocks,
 - blocklen – the size of each block,
 - stride   – the number of elements,located between the two neighboring
              blocks,
 - oldtype  - the initial data type,
 - newtype  - the new derived data type.
```

```
int MPI_Type_hvector ( int count, int blocklen, MPI_Aint stride,
  MPI_Data_type oldtype, MPI_Datatype *newtype ).
```

The difference of the constructing method, which is defined by the function *MPI_Type_hvector*, is only that the parameter *stride* is set in bytes (and not in the initial data type elements) for determining the interval between blocks.

As it follows from the description, the new derived type is constructed in case of the vector method as a number of blocks of the initial type elements. Additionally there may exist a regular interval in the memory between blocks. Let us give several examples of using the method for constructing data types:

- Constructing the type for selecting a half (only even or only odd ones) rows of the matrix of *nxn* size:

```
MPI_Type_vector ( n/2, n, 2*n, &StripRowType, &ElemType ),
```

- Constructing the type for selecting a column of the matrix of *nxn* size:

```
MPI_Type_vector ( n, 1, n, &ColumnType, &ElemType ),
```

- Constructing the type for selecting the main matrix diagonal for the matrix of *nxn* size:

```
MPI_Type_vector ( n, 1, n+1, &DiagonalType, &ElemType ).
```

Taking into account the type of these examples it should be mentioned the possibility to create the derived data types for the description of the subarrays of the multidimensional arrays, which is available in MPI. It is possible by means of the following function (this function is specified by the standard MPI-2):

```
int MPI_Type_create_subarray ( int ndims, int *sizes, int *subsizes,
  int *starts, int order, MPI_Data_type oldtype, MPI_Datatype *newtype ),
where
 - ndims    – the array dimension,
 - sizes    – the number of elements in each dimension of the initial
              array,
 - subsizes – the number of elements in each dimenstion of the determined
              subarray,
 - starts   – the indices of the initial elements in each dimension of the
              determined subarray,
 - order    - the parameter for pointing to the necessity of re-ordering,
 - oldtype  - the data type of the initial array elements,
 - newtype  - the new data type for the description of the subarray.
```

4.5.2.3 The Index Constructing Method

The following function is used in MPI for constructing the derived data type in this case:

```
int MPI_Type_indexed ( int count, int blocklens[], int indices[],
  MPI_Data_type oldtype, MPI_Datatype *newtype ),
where
 - count     – the number of blocks,
 - blocklens – the number of elements in each block,
 - indices   – the offset of each block with respect to the type origin
               address (in number of the initial type elements),
 - oldtype   - the initial data type,
 - newtype   - the new derived data type.
```

```
int MPI_Type_hindexed ( int count, int blocklens[], MPI_Aint indices[],
  MPI_Data_type oldtype, MPI_Datatype *newtype )
```

As it follows from the description, in case of the index method the new derived data type is created as a set of blocks of different sizes of the initial type elements and memory space between blocks may be different. The method may be illustrated by the following example of constructing a type for the description of the upper triangle matrix of *nxn* size:

```
// the constructing of the type for describing the upper triangle matrix
for ( i=0, i<n; i++ ) {
  blocklens[i] = n - i;
  indices[i]   = i * n + i;
}
MPI_Type_indexed ( n, blocklens, indices, &UTMatrixType, &ElemType ).
```

As previously, the constructing method, defined by the function *MPI_Type_hindexed*, is characterized by setting the element indices in bytes, and not in the initial data type elements.

It should be noted that there is the complimentary function *MPI_Type_create_indexed_block* for the index constructing method. This function is used for defining types with the blocks of the same size (the function is specified by the standard MPI-2).

4.5.2.4 The Structural Constructing Method

As it has been previously mentioned, this method is the most general constructing way for creating the derived data type, when the corresponding type map is set directly. The use of this method is provided by the following function:

```
int MPI_Type_struct ( int count, int blocklens[], MPI_Aint indices[],
  MPI_Data_type oldtypes[], MPI_Datatype *newtype ),
where
 - count      – the number of blocks,
 - blocklens  – the number of elements in each blocks,
 - indices    – the offset of each block with respect to the type origion
                in bytes,
 - oldtypes   - the initial data types in each block separately,
 - newtype    - the new determined data type.
```

As it follows from the description, this method in addition to the index method makes possible to specify the element types for each separate block.

4.5.3. Declaring and Deleting Derived Data Types

The constructing functions described in the previous subsection make possible to define derived data type. The created data type should be declared before being used by means of the following function:

```
int MPI_Type_commit (MPI_Datatype *type ).
```

After the termination of its use, the derived type must be annulled by means of the following function:

```
int MPI_Type_free (MPI_Datatype *type ).
```

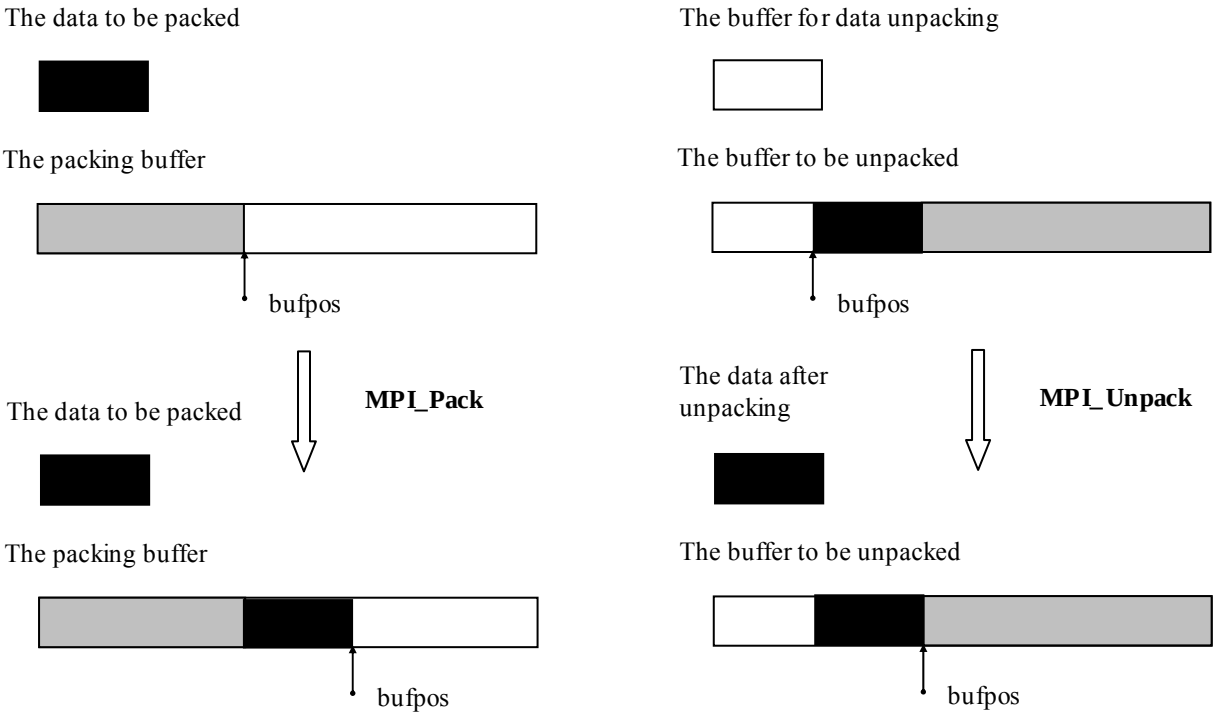
4.5.4. Creating Messages by Means of Data Packing and Unpacking

Besides the constructing methods described in 4.5.2 MPI offers an explicit method of packing and unpacking the messages, which contain values of different types and are located in different memory areas.

In order to use this method it is necessary to define the memory buffer of the size, which is sufficient for packing the message. The data contained in the message should be packed into the buffer by means of the following function:

```
int MPI_Pack ( void *data, int count, MPI_Datatype type,
  void *buf, int bufsize, int *bufpos, MPI_Comm comm),
where
 - data   – the memory buffer with the elements to be packed,
 - count  – the number of elements in the buffer,
 - type   – the data type for the elements to be packed,
 - buf    - the memory buffer for packing,
 - buflen – the buffer size in bytes,
 - bufpos – the position for the beginning of buffering (in bytes from the
            beginning of the buffer),
 - comm   - the communicator for the packed message.
```

The function *MPI_Pack* packs *count* elements from the buffer *data* to the packing buffer *buf*, starting from the position *bufpos*. The general scheme of the packing procedure is shown in Figure 4.8a.



a) data packing b) data unpacking

Figure 4.8. The general scheme of the data packing and unpacking

The initial value of the variable *bufpos* should be set before the beginning of packing. Further it is set by the function *MPI_Pack*. The function *MPI_Pack* is called sequentially for packing all the necessary data. Thus, for packing the previously discussed set of varialbles *a, b* and *n* it is necessary to carry out the following operations:

```
bufpos = 0;
MPI_Pack(a,1,MPI_DOUBLE,buf,buflen,&bufpos,comm);
MPI_Pack(b,1,MPI_DOUBLE,buf,buflen,&bufpos,comm);
MPI_Pack(n,1,MPI_INT,buf,buflen,&bufpos,comm);
```

To determine the buffer size necessary for packing, it is possible to use the following function:

```
int MPI_Pack_size (int count, MPI_Datatype type, MPI_Comm comm, int *size.
```

The parameter *size* of function specifies the necessary buffer size for packing *count* elements of the type *type*.

After packing all the necessary data the prepared buffer may be used in the data transmission functions with the type *MPI_PACKED*.

After the reception of the message with the type *MPI_PACKED,* the data may be unpacked by means of the following function:

```
int MPI_Unpack (void *buf, int bufsize, int *bufpos,
  void *data, int count, MPI_Datatype type, MPI_Comm comm),
where
 - buf    - the memory buffer with the packed data,
 - buflen – the buffer size in bytes,
 - bufpos – the position of the beginning of the data in the buffer (in
            bytes from the buffer beginning),
 - data   – the memory buffer for the data to be unpacked,
 - count  – the  number of elements in the buffer,
 - type   – the type of the unpacked data,
 - comm   - the communicator for the packed message.
```

The function *MPI_Unpack* unpacks the next portion of data from the buffer *buf* starting from the position *bufpos* and locates the unpacked data into the buffer *data*. The general scheme of the unpacking procedure is shown in Figure 4.8b.

The initial value of the variable *bufpos* should be set before the beginning of unpacking. It is further set by the function *MPI_Unpack*. The function *MPI_Unpack* is called sequentially for unpacking all the data having been

packed. The order of unpacking should correspond to the order of packing. Thus, for the previsouly described example of packing to unpack the data it is necessary to carry out the following opearations:

```
bufpos = 0;
MPI_Pack(buf,buflen,&bufpos,a,1,MPI_DOUBLE,comm);
MPI_Pack(buf,buflen,&bufpos,b,1,MPI_DOUBLE,comm);
MPI_Pack(buf,buflen,&bufpos,n,1,MPI_INT,comm);
```

Several recommendations concerning the use of packing for creating messages may be useful. As this approach causes the emergence of additional operations of packing and unpacking the data, this method may be justified, if the message sizes are comparatively small and the number of repetitions is also small. Packing and unpacking may prove to be useful, if buffers are explicitly used for the buffered method of data transmission.

4.6. Managing Groups of Processes and Communicators

Let us discuss the possibilities of MPI in managing groups of processes and communicators.

We should remind the reader of a number of concepts and definitions given at the beginning of the section.

Parallel program processes are united into groups. All the processes of a parallel program may be included into a *group*. On the other hand, a group can be formed from a part of the available processes only. Correspondingly, one process may belong to several groups. The groups of processes are managed by communicators.

A *communicator* in MPI is a specially created service object, which unites in its contents a group of processes and a number of additional parameters (*context*), which are used in data transmission operations. As a rule, point-to-point data transmission operations are carried out for the processes, which belong to the same communicator. Collective operations are applied simultaneously to all the communicator processes. Communicators are created in order to decrease the area of collective operations and to eliminate the mutual influence of different executed parts of the parallel program. It should be emphasized that the communication operations, carried out with the use of different communicators, are independent and do not influence each other.

All the processes available in the parallel program are included into the communicator with the identifier *MPI_COMM_WORLD*, which is created on default.

If it is necessary to transmit data among the processes, which belong to different groups, a global communicator (*intercommunicator*) should be created. The interaction of the processes, which belong to different groups, appears to be necessary only in comparatively rare situations. Such interaction is not discussed in this Section and may be one of the topics for individual work (see, for instance, Group, et al. (1994), Pacheco (1996)).

4.6.1. Managing Groups

The groups of processes may be composed only on basis of the available groups. The group of the predetermined communicator MPI_COMM_WORLD may be used as the source group.

To obtain such group the following function is used:

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

New groups may be further created on the basis of the existing groups:

- It is possible to create a new group *newgroup* on the basis of the group *oldgroup*, which includes *n* processes. The ranks of the processes are enumerated in the array *ranks*:

```
int MPI_Group_incl(MPI_Group oldgroup,int n, int *ranks,MPI_Group *newgroup),
```

- It is possible to create a new group *newgroup* on the basis of the group *oldgroup*, which includes *n* processes. The ranks of the processes do not coincide with the ranks enumerated in the array *ranks*:

```
int MPI_Group_excl(MPI_Group oldgroup,int n, int *ranks,MPI_Group *newgroup).
```

The following operations of union, intersection and difference may be carried out over the available groups of processes for obtaining new groups:

- Creating a new group *newgroup* by uniting the groups *group1* and *group2*:

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);
```

- Creating a new group *newgroup* by intersecting *group1* and *group2*:

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

- Creating a new group *newgroup* as difference of *group1* and *group2*:

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

The special empty group MPI_COMM_EMPTY may appear useful in constructing groups.

The following MPI functions provide obtaining information of the group of processes:

- Obtaining the number of processes in the group:

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

- Obtaining the rank of the current process in the group:

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

After the termination of its use, the group must be deleted:

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

(the execution of this operation does not concern the communicators, within of which the deleted group is used).

4.6.2. Managing Communicators

It should be first of all noted that this subsection discusses managing the intracommunicators, which are used for data transmission operation inside a group of processes. As it has been mentioned previously, the use of intercommunicators for the exchanges among different groups of processes is not to be discussed in this Section.

To create new communicators we use the two main methods:

- The duplication of the existing communicator:

```
int MPI_Comm_dup ( MPI_Comm oldcom, MPI_comm *newcomm ),
```

- The creation of a new communicator using a subset of the processes of the existing communicator:

```
int MPI_comm_create (MPI_Comm oldcom, MPI_Group group, MPI_Comm *newcomm).
```

The communicator duplication may be used, for instance, in order to eliminate the possibility of intersecting message tags in different parts of the parallel program (including the case when functions of different program libraries are used).

It should be noted that the operation of creating communicators is collective and must be executed by all the processes of initial communicator.

To explain the above described functions it is possible to give the following example of creating the communicator, which contains all the processes, except the process, which has the rank 0 in the communicator *MPI_COMM_WORLD* (such a communicator may be useful for maintaining the scheme of organizing parallel computations "manager – executors" – see Section 6):

```
MPI_Group WorldGroup, WorkerGroup;
MPI_Comm Workers;
int ranks[1];
ranks[0] = 0;
// obtaining a group of processes in MPI_COMM_WORLD
MPI_Comm_group(MPI_COMM_WORLD, &WorldGroup);
// creating a group excluding the process 0
MPI_Group_excl(WorldGroup, 1, ranks, &WorkerGroup);
// creating a communicator for the group
MPI_Comm_create(MPI_COMM_WORLD,WorkerGroup,&Workers);
...
MPI_Group_free(&WorkerGroup);
MPI_Comm_free(&Workers);
```

The following function provides a fast and useful method of simultaneous creation of several communicators:

```
int MPI_Comm_split ( MPI_Comm oldcomm, int split, int key,
  MPI_Comm *newcomm ),
where
 - oldcomm – the initial communicator,
 - split   – the number of the communicator,to which the process should
             belong,
 - key     – the order of the process rank in the  communicator being
             created,
 - newcomm – the communicator being created.
```

The creation of communicators refers to collective operations. Thus, the function *MPI_Comm_split* should be called in each process of the communicator *oldcomm*. The execution of the function leads to separating the processes into non-intersecting groups, which have the same values of the parameter *split*. On the basis of these groups a set of communicators is created. The order of enumeration for the process ranks is selected in such a way

that it corresponds to the order of the parameter *key* values (the process with the greater value of the parameter *key* should have a higher rank).

As an example we may discuss the problem of presenting a set of processes as a two-dimensional grid. Let $p=q*q$ be the total number of processes. The following program fragment provides creating communicators for each row of the created topology:

```
MPI_Comm comm;
int rank, row;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
row = rank/q;
MPI_Comm_split(MPI_COMM_WORLD,row,rank,&comm);
```

In solving this problem, when, for instance, $p=9$, the processes with the ranks (0,1,2) create the first communicator, the processes with the ranks (3,4,5) create the second one and so on.

After the termination of its use, the communicator should be deleted:

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

4.7. Virtual Topologies

The *topology* of a computer system is the structure of the network nodes and communication lines, which connect them. The topology may be presented as a graph, where the vertices are the system processors (processes), and the arcs correspond to the available communication lines (channels).

As it has been mentioned previously, point-to-point data transmission operations may be executed for any processes of the same communicator. All the processes of the communicator participate in collective operations. In this respect, the logical topology of the communication lines in a parallel program is a *complete graph* (regardless of the availability of the physical communication channels among the processors).

It is obvious that the physical system topology is hardware realizable and cannot be changed (though there are already software for networking). But leaving the physical basis unchanged, we may organize the logical presentation of any necessary *virtual topology*. It is sufficient, for instance, to form a mechanism of additional process addressing for this purpose.

The use of virtual processes may appear to be efficient due to various reasons. A virtual topology may, for instance, better correspond to the available structure of data transmission lines. The use of virtual topologies may significantly simplify in a number of cases the presentation and the implementation of parallel algorithms.

The two types of topologies are maintained in MPI: the *rectangular grid* of the arbitrary dimension (*Cartesian topology*) and the *graph* topology of any arbitrary form. It should be noted that the functions available in MPI provide only obtaining new logical systems of process addressing, which correspond to the created virtual topologies. The execution of all the communication operations should be carried out, as previously, by means of usual data transmission functions using the initial process ranks.

4.7.1. Cartesian Topologies

Cartesian topologies, which assume the presentation of a set of processes as a rectangular grid (see 1.4.1 and Figure 1.7) and the use of Cartesian coordinate system for pointing to the processes, are widely used in many problems for the description of the information dependence structure. Among the examples of such problems there are matrix algorithms (see Sections 7 and 8) and the grid method of solving differential equations in partial derivatives (see Section 12).

The following function is used in MPI for creating Cartesian topology (grid):

```
int MPI_Cart_create(MPI_Comm oldcomm, int ndims, int *dims, int *periods,
  int reorder, MPI_Comm *cartcomm),
where:
 - oldcomm  - the initial communicator,
 - ndims    - the Cartesian grid dimension,
 - dims     - the array of ndims length, it defines the number of
              processes in each dimension of the grid,
 - periods  - the array of ndims length, which defines whether the grid is
              periodical along each dimension,
 - reorder  - the parameter to allow changing the process ranks,
 - cartcomm – the communicator being created with the Cartesian topology.
```

The operation of topology creation is a collective one. Thus, it should be carried out by all the processes of the initial communicator.

To explain the designation of the function parameters of *MPI_Cart_create* we will consider the example of creating the two-dimensional grid *4x4*, the rows and columns of which have a ring structure (the first process follows the last one):

```
// creating the two-dimensional grid 4x4
MPI_Comm GridComm;
int dims[2], periods[2], reorder = 1;
dims[0]    = dims[1]    = 4;
periods[0] = periods[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reoreder, &GridComm);
```

It should be noted that the topology formed in this example is a torus due to the ring structure.

In order to determine the Cartesian process coordinates according to its rank, we may use the following function:

```
int MPI_Card_coords(MPI_Comm comm,int rank,int ndims,int *coords),
where:
 - comm   – the communicator with grid topology,
 - rank   - the rank of the process, for which Cartesian coordinates are
            determined,
 - ndims  - the grid dimansion,
 - coords – the Cartesian process coordinates returned by the function.
```

The reverse operation, i.e. determining the process rank according to its Cartesian coordinates, is provided by means of the following function:

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank),
where
 - comm   – the communicator with grid topology,
 - coords – the Cartesian coordinates of the process,
 - rank   - the process rank returned by the function.
```

The procedure of splitting the grids into subgrids of smaller dimension, which is useful in many applications, is provided by the following function:

```
int MPI_Card_sub(MPI_Comm comm, int *subdims, MPI_Comm *newcomm),
where:
 - comm    - the initial communicator with grid topology,
 - subdims – the array for pointing the dimensions that should be fixed
             in the subgrid being created,
 - newcomm – the created communicator with the subgrid.
```

The operation of creating subgrids is also collective. Thus, it must be carried out by all the processes of the initial communicator. The function *MPI_Cart_sub* defines, while it is being carried out, the communicators for each combination of the coordinates of the fixed dimensions of the initial grid.

To explain the function *MPI_Cart_sub* we will consider the above described example of creating a two-dimensional grid in more detail and define the communicators with the Cartesian topology for each grid row and column separately:

```
// creating communicators for each grid row and grid column
MPI_Comm RowComm, ColComm;
int subdims[2];
// creating communicators for rows
subdims[0] = 0;  // dimensionality fixing
subdims[1] = 1;  // the presence of the given dimension in the subgrid
MPI_Cart_sub(GridComm, subdims, &RowComm);
// creating communicatros for columns
subdims[0] = 1;
subdims[1] = 0;
MPI_Cart_sub(GridComm, subdims, &ColComm);
```

Eight communicators are created for the grid of size 4x4 in this example. A communicator for every grid row and grid column is created. For each process the defined *RowComm* and *ColComm* communicators correspond to the row and the column of the processes, to which the given process belongs.

The additional function *MPI_Cart_shift* provides the support of sequential data transmission along a grid dimension (the operation of data shift – see Section 3). Depending on the periodicity of the data shift grid dimension, the two types of this operation are differentiated:

– *The cyclic shift* on *k* elements along the grid dimension. The data from the process *i* is transmitted to the process *(i+k) mod dim*, where *dim* is the size of the dimension, along which the shift is performed,

– *The linear shift* on *k* positions along the grid dimension. In this variant of the operation the data from the processor *i* is transmitted to the processor *i+k* (if the latter is available).

The function *MPI_Cart_shift* provides obtaining the ranks of the processes, which are to exchange the data with the current process (the process, which has called up the function *MPI_Cart_shift*):

```
int MPI_Card_shift(MPI_Comm comm, int dir, int disp,
  int *source, int *dst),
where:
 - comm   – the communicator with grid topology,
 - dir    - the number of the dimension, along which the shift is
            performed,
 - disp   - the shift value (<0 – is the shift towards the beginning of
            the dimension),
 - source – the rank of the process, from which the data should be
            received,
 - dst    - the rank of the process, to which the data should be sent.
```

It should be noted, that the function *MPI_Cart_shift* only defines the rank of the processes, which are to exchange data in the course of shift operation. The direct data transmission may be carried out, for instance, with the help of the function *MPI_Sendrecv*.

4.7.2. Graph Topology

The information concerning the MPI functions for the operations with virtual topologies of graph type will be discussed briefly. Some additional information may be found in Group, et al. (1994), Pacheco (1996).

To create a communicator with the graph type topology the following function is intended in MPI:

```
int MPI_Graph_create(MPI_Comm oldcomm, int nnodes, int *index, int *edges,
  int reorder, MPI_Comm *graphcomm),
where:
 - oldcomm  - the initial communicator,
 - nnodes   - the number of the graph vertices,
 - index    - the number of the arcs outgoing from vertices,
 - edges    - the sequential list of the graph arcs,
 - reorder  - the parameter to allow changing the process ranks,
 - cartcomm – the created communicator with the graph type topology.
```

Creating topology is a collective operation and should be carried out by all the processes of the initial communicator.

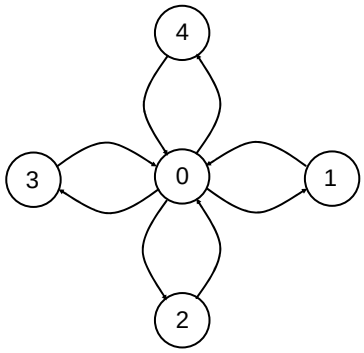


Figure 4.9. The example of the graph for the star type topology

Let us create as an example the topology of the graph, the structure of which is presented in Figure 4.9. In this case the number of processes is equal to 5, the orders of graph vertices (the numbers of the outgoing arcs) are (4,1,1,1,1) correspondingly, and the incidence matrix (the numbers of the vertices, for which the arcs are incoming) looks as follows:

```
Processes    Communication Lines
  0              1, 2, 3, 4
  1                   0
  2                   0
  3                   0
```

```
4                    0
```

To create the topology with the graph of this type, it is necessary to perform the following program code:

```
// creating the star type topology
int index[] = { 4,1,1,1,1 };
int edges[] = { 1,2,3,4,0,0,0,0 };
MPI_Comm StarComm;
MPI_Graph_create(MPI_COMM_WORLD, 5, index, edges, 1, &StarComm);
```

Let us mention two more useful functions for the operations with graph topologies. The number of the neighboring processes, which contain the outgoing arcs from the process being checked, may be obtained by the following function:

```
int MPI_Graph_neighbors_count(MPI_Comm comm,int rank, int *nneighbors).
```

Obtaining the ranks of the neighboring vertices is provided by the following function:

```
int MPI_Graph_neighbors(MPI_Comm comm,int rank,int mneighbors, int *neighbors),
```

where *mneighbors* is the size of the array *neighbors*.

4.8. Additional Information on MPI

4.8.1. The Development of MPI Based Parallel Programs in the Algorithmic Language Fortran

The development of MPI based parallel programs in Fortran does not imply as many peculiarities as in case of using the algorithmic language C:

1. The subprograms of the library MPI are procedures, and thus, they are called by means of the procedure call statement CALL,
2. The termination codes are retrurned through the additional parameter of the integer type, which is located last in the list of the procedure parameters,
3. The variable *status* is the integer type array, which consists of *MPI_STATUS_SIZE* elements,
4. The types *MPI_Comm* and *MPI_Datatype* are presented by the integer type INTEGER.

It is agreed that the names of the subprograms should be written with the use of upper-case sumbols, if the programs are developed in FORTRAN.

As an example we can cite the variant of the program from 4.2.1.5 in FORTRAN.

```
PROGRAM MAIN
  include 'mpi.h'
  INTEGER PROCNUM, PROCRANK, RECVRANK, IERR
  INTEGER STATUS(MPI_STATUS_SIZE)
  CALL MPI_Init(IERR)
  CALL MPI_Comm_size(MPI_COMM_WORLD, PROCNUM, IERR)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, PROCRANK IERR)
  IF ( PROCRANK.EQ.0 )THEN
    ! The operations carried out only by the process 0
    PRINT *,"Hello from process ", PROCRANK
    DO i = 1, PROCNUM-1
      CALL MPI_RECV(RECVRANK, 1, MPI_INT, MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, STATUS, IERR)
      PRINT *,"Hello from process ", RECVRANK
    END DO
  ELSE ! The message sent by all the processes,
       ! except the process 0
    CALL MPI_SEND(PROCRANK,1,MPI_INT,0,0,MPI_COMM_WORLD,IERR)
  END IF
  MPI_FINALIZE(IERR);
  STOP
END
```

4.8.2. Overview of MPI Program Execution Environment

The MPI program execution environment must be installed in a computer system in order to carry out parallel computations. This environment should provide the development, compilation, linkage and execution. The widely-

spread software of program development (such as, for instance Microsoft Visual Studio), are, as a rule, sufficient for carrying out the development, compilation and linkage of parallel programs. It is necessary only to have a MPI library. In order to carry out the parallel programs, the environment should have a number of additional utilities. Among these utilities there should be the means of choosing the processors being used, the tools for starting the remote programs etc. It is also desirable to have the software for profiling, tracing ands debugging parallel program in the environment.

Unfortunately, this is alomost the final point of standardization. There are several kinds of environment for carrying out MPI programs. In the majority of cases, these kinds of environment are created together with the development of some variants of MPI libraries. As a rule, the administrator of the computater system chooses the realization of MPI library, installs the environment and prepares the maintenance instructions. Usually the internet information recourses, where free MPI realizations are located, and commercial MPI versions contain enough information on MPI installation procedures. Thus, carrying out all the necessary operations does not imply any difficulties.

The start of MPI program also depends on the execution environment. In the majority of cases this operation is carried out by means of the command **mpirun**. This command may have the following possible parameters:

– *Execution mode*. It may be *local* or *distributed*. The local mode is usually indicated by means of the key -*localonly*. If a parallel program is executed in the local mode, then all the processes of the program are located on the computer, from which the program has been started. This method of execution is very useful for the initial testing of working efficiency and debugging a parallel program. A part of this work may be carried out even on a separate computer, not included into a multiprocessor computer system,

– *The number of processes,* which should be created when a parallel program is started,

– *The list of the processors* being used, which is defined by a configuration file,

– *The executable file of the parallel program*,

– *The command* line with the parameters of the executed program.

There are a number of other parameters but they are usually used in the development of rather complicated parallel programs. The description may be obtained in reference materials concerning the corresponding MPI program execution environment.

If a program is started on several computers, the executable program file must be copied on all the computers or it should be located on the resource, which may be accessible by all of them.

Additional information on the execution environment of parallel programs for cluster systems may be obtained in Sterling (2001, 2002). It should be noted that the standardization elements of the execution environment have been added in the standard MPI–2.

4.8.3. Additional Features of the Standard MPI-2

As it has already been mentioned, the standard MPI-2 was adopted in 1997. The use of this standard variant is still limited, despite the fact that it was adopted rather long ago. This situation may be caused by the following several reasons: the conservatism of the software developers, the complexity of new standard implementation etc. Another important aspect is also that MPI-1 possibilities appear to be sufficient for the realization of many parallel algorithms, while the area of application for the additional possibilities of MPI -2 appears to be not so wide.

In order to get more information on the standard MPI-2 we recommend to make use of the information resource **http://www.mpiforum.org**, and the work by Group, et al. (1999b). In this section we will briefly characterize the additional possibilities of the standard MPI-2:

– *The dynamic generation of the processes*, which assumes the creation and elimination of the parallel program processes in the course of execution,

– *The single-sided process interaction*, which allows only one process to initiate data transmission and reception,

– *The parallel input/output*, which provides a special interface for the operation of the processes with the file system,

– *The extended collective operations*, which include, for instance, the procedures for simultaneous interaction of the processes from several communicators,

– *The C++ interface*.

4.9. Summary

This section is devoted to the description of the parallel programming methods for the computational systems with the distributed memory and the use of MPI.

It was mentioned at the very beginning of the section that MPI is a *message passing interface*. It is currently one of the basic approaches to develop parallel programs for the distributed memory computer systems. The use of MPI makes possible to distribute the computational load and arrange the information interaction, data transmission among the processors. The term MPI means on the one hand the standard to which the software of arranging message passing must meet. On the other hand, this term denotes the program libraries, which provide the possibility of message passing and meet all the requirements of the standard.

Subsection 4.1 discusses a number of concepts and definitions, which are the basic ones for the standard MPI. Thus, it presents a *parallel program* as a number of simultaneously executed *processes*. These processes may be carried out on different processors but several processes may be located on a processor (in these cases they are executed in the time-sharing mode). A brief characteristic of the concept needed for the description of the message passing operations is given further. The subsection also describes the data types, the communicators and virtual topologies.

Subsection 4.2 offers a brief and simple introduction into the development of MPI based parallel programs. The material of the subsection is a good basis for starting the development of parallel programs of various complexity.

Subsection 4.3 describes the *data transmission between two processes*. The modes of operation execution, such as the Standard, the Synchronous, the Buffered and the Ready ones, are described in detail. The possibility to arrange non-blocking data exchanges between the processes is discussed for every operation.

Subsection 4.4 is devoted to *collective data transmission operations*. The sequence of the description corresponds to the order, in which the communication operations are discussed in section 3. The conclusion given in the subsection says that MPI provides maintaining practically all the basic information exchanges among the processes.

Subsection 4.5 considers the use of *derived data types* in MPI. All the basic methods of constructing the derived data types are described in the subsection: the Contiguous, the Vector, the Index and the Structural methods. The subsection also discusses the possibility of the messages forming by means of data packing and unpacking.

Subsection 4.6 focuses on the issues of *process and communicator management*. The possibilities of MPI described in subsection make possible to manage the number of processes participating in collective operations and to eliminate the mutual influence of different executed parallel program parts.

Subsection 4.7 describes the possibilities of MPI in connection with the use of *virtual topologies*. The following MPI supported topologies are described in the subsection: the *rectangular grid* of arbitrary dimension (*Cartesian topology*) and the *graph* topology of any required type.

Subsection 4.8 provides additional information of MPI. The information includes the issues of MPI based parallel program development in the algorithmic language Fortran, a brief overview of the execution environment for MPI based programs and a survey of the additional possibilities of the standard MPI-2.

4.10. References

There are a number of sources, which provide information about MPI. First of all, this is the internet resource, which describes the standard MPI: http://www.mpiforum.org. One of the most widely used MPI realizations, the library MPICH, is presented on http://www-unix.mcs.anl.gov/mpi/mpich (the library MPICH2 with the realization of the standard MPI-2 is located on http://www-unix.mcs.anl.gov/mpi/mpich2).

The following works may be recommended: Group, et al. (1994), Pacheco (1996), Snir, et al. (1996), Group, et al. (1999a). The description of the standard MPI-2 may be found in Group, et al. (1999b).

We may also recommend the work by Quinn (2003), which described a number of typical problems of parallel programming for the purpose of studying MPI. These are the problems of matrix computations, sorting, graph processing etc.

4.11. Discussions

1. What minimum set of operations of sufficient for the organization of parallel computations in the distributed memory systems?
2. Why is it important to standardize message passing?
3. How can a parallel program be defined?
4. What are the difference between the concepts of “process” and “processor”?
5. What minimum set of MPI functions makes possible to start the development of parallel progrmas?
6. How are the messages being passed described?
7. How do we organize the reception of messages from concrete processes?
8. How do we determine the execution time of an MPI based program?
9. What is the difference between point-to-point and collective data transmission operations?

10. Which MPI function provides transmitting data from a process to all the processes?
11. What is the data reduction operation?
12. In what cases should we apply barrier synchronization?
13. What data transmission modes are supported in MPI?
14. In what way is the non-blocking data exchange organized in MPI?
15. What is a deadlock? In what cases does the function of the simultaneous transmission/reception guarantee the absence of deadlock situations?
16. What collective data transmission operations are supported in MPI?
17. What is the derived data type in MPI?
18. What methods of constructing types are available in MPI?
19. In what situations may data packing and unpacking be useful?
20. What is a communicator in MPI?
21. What can new communicators be created for?
22. What is a virtual topology in MPI?
23. What types of virtual topologies are supported in MPI?
24. What may virtual topologies appear to be useful for?
25. What are the peculiarities of developing MPI based parallel programs in Fortran?
26. What are the basic additional features of the standard MPI-2?

4.12. Exercises

Subsection 4.2.

1. Develop a program for finding the minimum (maximum) value of the vector elements.
2. Develop a program for computing the scalar product of two vectors.
3. Develop a program, where two processes repeatedly exchange messages of n byte length. Carry out the experiments and estimate the dependence of the data operation execution time against the message length. Compare it to the theoretical estimations created according to the Hockney model.

Subsection 4.3.

4. Prepare the variants of the previously developed programs with different modes of data transmission. Compare the execution time of data transmission operations in cases of different modes.
5. Prepare the variants of the previously developed programs using non-blocking method of data transmission operations. Estimate the necessary amount of the computational operations, which is needed to superpose completely data transmission and computations. Develop the program, which has no computation delays caused by waiting for the transmitted data.
6. Do the exercises 3 and use the operation of simultaneous data transmission and reception. Compare the results of the computational experiments.

Subsection 4.4.

7. Develop a sample program for each collective operation available in MPI.
8. Develop the realizations of collective operations using point-to-point exchanges among processes. Carry out the computational experiments and compare the execution time of the developed programs to the functions of MPI for collective operations.
9. Develop a program, carry out the experiments and compare the results for different algorithms of data gathering, data processing and data broadcasting (the function *MPI_Allreduce*).

Subsection 4.5.

10. Develop a sample program for each method of constructing derived data types available in MPI.
11. Develop a sample program using data packing and unpacking functions. Carry out the experiments and compare the results to the results obtained in case of the use of the derived data types.
12. Develop the derived data types for the rows, columns and diagonals of matrices.
13. Develop a sample program for each of the discussed functions of managing communicators and processes.
14. Develop a program for presenting a set of processes as a rectangular grid. Create communicators for each row and each column of the processes. Perform a collective operation for all the processes and one of the created communicators. Compare the operation execution time.
15. Study by yourself and develop sample programs for transmitting data among the processes of different communicators.

Subsection 4.7.

16. Develop a sample program for the Cartesian topology.

17. Develop a sample program for a graph topology.

18. Develop subprograms for creating a set of additional virtual topologies (a star, a tree etc.).

References

Pacheco, P. (1996). Parallel Programming with MPI. - Morgan Kaufmann.

Gropp, W., Lusk, E., Skjellum, A. (1999a). Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface (Scientific and Engineering Computation). - MIT Press.

Gropp, W., Lusk, E., Thakur, R. (1999b). Using MPI-2: Advanced Features of the Message Passing Interface (Scientific and Engineering Computation). - MIT Press.

Snir, M., Otto, S., Huss-Lederman, S., Walker, D., Dongarra, J. (1996). MPI: The Complete Reference. - MIT Press, Boston, 1996.