

M. E. Abramyan

Parallel Programming Based on MPI 2.0

*Textbook for students
of computer science and programming*

The first section of the textbook describes Message Passing Interface (MPI) standards 1.1 and 2.0 for the C language, and also considers algorithms of parallel matrix multiplication, in the implementation of which various tools of MPI are used. The description is accompanied by solutions of typical tasks. The second section contains 265 training tasks included in the electronic problem book Programming Taskbook for MPI-2 developed by the author and covering all topics of the first section. The third section gives an overview of the capabilities of the Programming Taskbook for MPI-2 and provides a set of 24 variants of individual assignments, compiled from the training tasks of the second section. The textbook is supplied with an index containing all the considered constants, types and functions of the MPI-1 and MPI-2 interfaces.

The textbook is intended for students specializing in science and engineering.

Contents

Preface	6
1. MPI: description and examples of use	8
1.1. Introduction to MPI	8
1.1.1. MPI and its study with the help of the electronic problem book PT for MPI-2.....	8
1.1.2. Basic concepts of MPI programming	11
1.1.3. Creating a template for a parallel program.....	12
1.1.4. Running a program in parallel mode	17
1.1.5. Executing MPI1Proc2 task	23
1.1.6. Using additional information in the debug section.....	28
1.2. Basic capabilities of the MPI interface (MPI-1 standard)	31
1.2.1. Blocking point-to-point communication: basic features	31
1.2.2. Blocking point-to-point communication: examples. Mutual process deadlocks.....	35
1.2.3. Non-blocking point-to-point communications. Persistent requests for interaction. Timing functions	42
1.2.4. Collective communications	45
1.2.5. Reduction operations and using compound datatypes.....	49
1.2.6. Defining derived datatypes and packing data using dynamic arrays and vector containers	56
1.2.7. Creation of new communicators	69
1.2.8. Cartesian topology	76
1.2.9. Graph topology	85
1.3. Additional features of the MPI interface (MPI-2 standard)	91
1.3.1. Distributed graph topology	91
1.3.2. Parallel input-output. File access functions	94
1.3.3. Parallel input-output: an example. Setting up the file view	99
1.3.4. One-sided communications: general description.....	105
1.3.5. One-sided communications: an example using the simplest synchronization option.....	111
1.3.6. One-sided communications: an example of a more complex version of synchronization.....	115
1.3.7. Inter-communicators.....	121
1.3.8. Dynamic process creation.....	130
1.4. Parallel matrix algorithms.....	139

1.4.1. Band and block algorithms for parallel matrix multiplication: general description	139
1.4.2. Implementation of a non-parallel matrix multiplication algorithm	143
1.4.3. Scattering source data: an example of implementation	148
1.4.4. Redistribution of blocks at the initial stage of Cannon's algorithm	153
1.4.5. Result gathering stage: an example of file-based output implementation	159
1.5. Additional techniques for developing parallel programs	165
1.5.1. Debugging parallel programs using taskbook tools	165
1.5.2. Developing and running parallel programs without the taskbook ..	174
1.5.3. Additional debug features. Output redirection.....	178
2. Learning tasks	185
2.1. Processes and their ranks.....	185
2.2. Point-to-point communication.....	186
2.2.1. Blocking communications.....	186
2.2.2. Non-blocking communications	191
2.3. Collective communications	194
2.3.1. Collective data transfer.....	194
2.3.2. Global reduction operations	196
2.4. Derived datatypes and data packing.....	198
2.4.1. The simplest derived datatypes	198
2.4.2. Data packing.....	200
2.4.3. Additional ways of derived datatypes creation	201
2.4.4. The MPI_Alltoallw function (MPI-2).....	205
2.5. Process groups and communicators	205
2.5.1. Creation of new communicators	205
2.5.2. Virtual topologies	208
2.5.3. The distributed graph topology (MPI-2).....	214
2.5.4. Non-blocking collective functions (MPI-3).....	215
2.6. Parallel file input-output (MPI-2)	220
2.6.1. Local functions for file input-output.....	220
2.6.2. Collective functions for file input-output.....	222
2.6.3. File view setting for file input-output	224
2.7. One-sided communications (MPI-2).....	228
2.7.1. One-sided communications with the simplest synchronization	229
2.7.2. Additional types of synchronization	233
2.8. Inter-communicators and process creation.....	238
2.8.1. Inter-communicator creation	239
2.8.2. Collective communications for inter-communicators.....	244
2.8.3. Process creation	246
2.9. Parallel matrix algorithms	250
2.9.1. Non-parallel matrix multiplication algorithm	251

2.9.2. Band algorithm 1 (horizontal bands)	251
2.9.3. Band algorithm 2 (horizontal and vertical bands)	255
2.9.4. Cannon's block algorithm	261
2.9.5. Fox's block algorithm	267
3. Additions	275
3.1. Programming Taskbook for MPI-2	275
3.1.1. General description	275
3.1.2. Taskbook tools for initializing tasks and data input-output	278
3.1.3. Debug section	279
3.1.4. Functions for outputting and configuring debug information	282
3.2. Options for individual assignments	284
3.2.1. Series of similar tasks	284
3.2.2. Set of 24 variants of tasks	288
References.....	304
Index	305

Preface

The textbook offered to your attention is a practical introduction to parallel programming based on MPI (*Message Passing Interface*). Currently, MPI is one of the main parallel programming tools for cluster systems and distributed memory computers [5–10]. The textbook describes the MPI standards of versions 1.1 [11] and 2.x [12] (and partially 3.x [13]), on which most modern software implementations are based. A version of the MPI interface for the C language is used; for input-output, C++ language tools are used.

The textbook consists of two main sections. The first section provides a systematic description of the capabilities of the MPI interface. The basic capabilities of MPI are considered in detail, including blocking and non-blocking message exchange between two processes, collective interactions of processes, definition of derived types, work with groups of processes and communicators, application of virtual topologies. In addition, new capabilities introduced (or significantly expanded) in the MPI-2 standard are studied: parallel file input-output, one-sided communications, use of inter-communicators and dynamic creation of processes, and some others. Along with various capabilities of the MPI interface, the first section also considers an important class of parallel algorithms, namely, parallel matrix multiplication algorithms, for which various MPI capabilities are used. All topics discussed are accompanied by examples of program code associated with solving typical tasks.

The second section contains 265 training tasks on all the topics considered in the first section. It should be noted that for practical study of the main components of MPI it is sufficient to use a local computer, simulating parallel execution of processes on it. However, even in this simplest version, the student inevitably encounters additional difficulties in developing parallel programs, due to the complexity of organizing input-output of data for various processes of a parallel program and the impossibility of using standard IDE debugging tools for parallel programs. To facilitate studying MPI technologies in practice, the author has developed a specialized training system — an electronic problem book on parallel programming **Programming Taskbook for MPI-2** (PT for MPI-2). All tasks included in the second section of the textbook can be solved using the PT for MPI-2 in various IDEs including **Microsoft Visual Studio** (2017, 2019, 2022), **Code::Blocks**, **Dev-C++**, and **Visual Studio Code** editor. Thus, the distinctive features of the approach to teaching parallel programming used in this textbook are the presence of a large number of training tasks related

to all aspects of MPI and the possibility of using specialized software tools that significantly accelerate the process of solving tasks.

The third, additional section provides a general description of the PT for MPI-2 taskbook. It also provides information about the series of similar training tasks presented in the second section of the book. This information may be useful in compiling various sets of individual tasks. As an example, this section presents 24 variants of individual tasks that cover all the topics covered in the textbook.

The textbook index includes constants, types, and functions of the MPI interface. The presence of this index allows the textbook to be used as a reference for MPI technologies.

The textbook is a substantially revised and expanded version of the book [1]. It examines the capabilities of the MPI -1 standard in more detail and covers new topics, namely, the new capabilities of the MPI-2 and MPI-3 standard and parallel matrix algorithms. 165 new problems have been added to the 100 given in the book [1]; and some previous problems have been provided with new wordings. A large number of new features have also appeared in the PT for MPI-2 taskbook, which is an extension of the PT for MPI taskbook used in [1].

You can get more information about the PT for MPI-2 taskbook and download its distribution from the website of the Programming Taskbook <http://ptaskbook.com/>.

1. MPI: description and examples of use

1.1. Introduction to MPI

1.1.1. MPI and its study with the help of the electronic problem book PT for MPI-2

MPI (*Message Passing Interface*) provides means for transferring information between different processes of a parallel application. The first version of the MPI standard (MPI-1) was developed in 1993–1995 [11]; already in 1997 the second version (MPI-2) appeared, supplemented with a large number of new features [12]. The MPI-2 standard was subsequently revised in 2008 and 2009 [12]. Starting from 2012, versions of standard 3 appear [13]; standard 4 is being developed from 2021. Currently, the most common version of MPI is 1.1, but an increasing number of implementations are beginning to support the capabilities of the MPI-2 and MPI-3 standards.

MPI standard is defined for two languages: Fortran and C (the C variant can be used without any changes in C++ programs). There are MPI implementations for other languages (for example, Python and C#), but, as a rule, parallel programs using MPI are developed in C/C++ and Fortran.

In order to achieve maximum efficiency, parallel programs should be executed on supercomputers or computing clusters that allow for efficient distribution of the launched processes across the supercomputer processors or cluster nodes. However, to study the capabilities of MPI, it is quite sufficient to use a local computer for launching all the processes of a parallel application. In such a situation, one should not expect a significant gain in the speed of parallel algorithms, but with the help of such learning programs a student can become familiar with the MPI tools and try them out in action. For this purpose, the author of this textbook has developed *an electronic problem book on parallel programming* **Programming Taskbook for MPI-2** (PT for MPI-2). Detailed description of the PT for MPI-2 taskbook is contained in Section 3.1. This textbook describes the PT for MPI-2 version 1.6 released in 2024.

The PT for MPI-2 taskbook allows developing parallel programs in C++ using MPI interface for C. Additional capabilities of the C++ language are used mainly for more convenient organization of input-output (using streams and iterators, see Section 3.1.2), although in some situations other C++ tools are also useful, e. g. template functions (see the MPI2Send22–MPI2Send25 tasks in Section 2.2.1). Since the PT for MPI-2 taskbook is a specialized extension for the universal problem book *Programming Taskbook*, it can be used together with all

IDEs for the C++ language that the basic taskbook supports. For the Programming Taskbook version 4.24 released in 2024, the following C++ IDEs are supported: Microsoft Visual Studio (version 2017, 2019, 2022), Code::Blocks, Dev-C++, Visual Studio Code. For more information about the IDEs supported by the taskbook, see its website ptaskbook.com.

Thus, in order to be able to solve training tasks on parallel programming using the PT for MPI-2 taskbook, you must first install one of the specified IDEs.

However, to run parallel programs developed on the basis of MPI, the presence of a programming environment (even with additional MPI libraries) is not enough. A system is needed that allows you to run parallel program processes and provides message exchange between them. Among the popular freely distributed MPI support systems are the MPICH/MPICH2 systems developed at the Argonne National Laboratory (USA) and MS-MPI system developed by Microsoft. The PT for MPI-2 taskbook can be used in conjunction with the following MPI support systems:

- **MPICH 1.2.5** (<ftp://ftp.mcs.anl.gov/pub/mpi/nt/mpich.nt.1.2.5.exe>), supports the MPI 1.2 standard;
- **MPICH2 1.3** (<http://www.mpich.org/static/downloads/1.3/mpich2-1.3-win-ia32.msi>), supports the MPI 2.1 standard;
- **MS-MPI 10.1.2** (<https://www.microsoft.com/en-us/download/details.aspx?id=100593>), supports MPI-2.1 (and partially MPI-3), provides faster operation of parallel programs for Windows 10 (requires downloading the `mssmpisetup.exe` installation file).

When using the MPICH system, you can only perform those tasks that are intended for studying the MPI tools of the 1.1 standard. The MPICH2 system allows you to perform almost all the tasks included in the PT for MPI-2 taskbook (except the MPI5Comm33–MPI5Comm47 tasks on non-blocking collective functions from the MPI-3 standard). The MS-MPI system allows you to perform *all* the tasks included in the PT for MPI-2 taskbook.

Note. To install MPICH 1.2.5 or MS-MPI 10.1.2, simply run the installation file and follow its instructions.

To install the MPICH2 1.3 system correctly, *you must run the installation file* `mpich2-1.3-win-ia32.msi` *with administrator rights*. If the corresponding pop-up menu item for this file is missing, you can, for example, run the command line with administrator rights (**Start | All Programs | Standard | Command Line**, use the **Run as administrator** command from the pop-up menu of this program), and run the installation file `mpich2-1.3-win-ia32.msi` in this command line. If you have the **FAR** file manager on your computer, it is more convenient to run this program with administrator rights and run the installation file in it. If you do not use administrator rights when installing the MPICH2 system, the installation will proceed

normally, however, when you try to run a parallel application using the `mpiexec.exe` program, the message *"Unknown option: -d"* will be displayed, caused by the fact that the system will not be able to start the `smpd.exe` process manager, which is part of MPICH2.

Sometimes a situation arises when the Windows system starts blocking the call of MPICH2 system components that ensure the launch of programs in parallel mode. In this case, it is usually sufficient to *reinstall* the MPICH2 system by running the installation program and selecting the **Repair MPICH2** option in it. Some types of antivirus applications may also try to block the execution of parallel programs, considering them suspicious.

After the programming environment for C++ and the MPI support system (MPICH, MPICH2 or MS-MPI) are installed, the basic version of the electronic problem book Programming Taskbook and the PT for MPI-2 taskbook should be installed (in the order specified). Installation programs for these problem books can be downloaded from the website of the Programming Taskbook `ptaskbook.com` (either in the **Download** section of the **Main** page or in the **Overview** section of the **PT for MPI-2** page). The **Overview** section of the **PT for MPI-2** page also contains links for downloading distributions of all MPI systems supported by the PT for MPI-2 taskbook.

After installing the PT for MPI-2 taskbook, the **PT4Setup** program window will appear on the screen, listing all programming environments in which the taskbook can be used. In this window, those MPI systems that are found on the computer will additionally appear (Fig. 1).

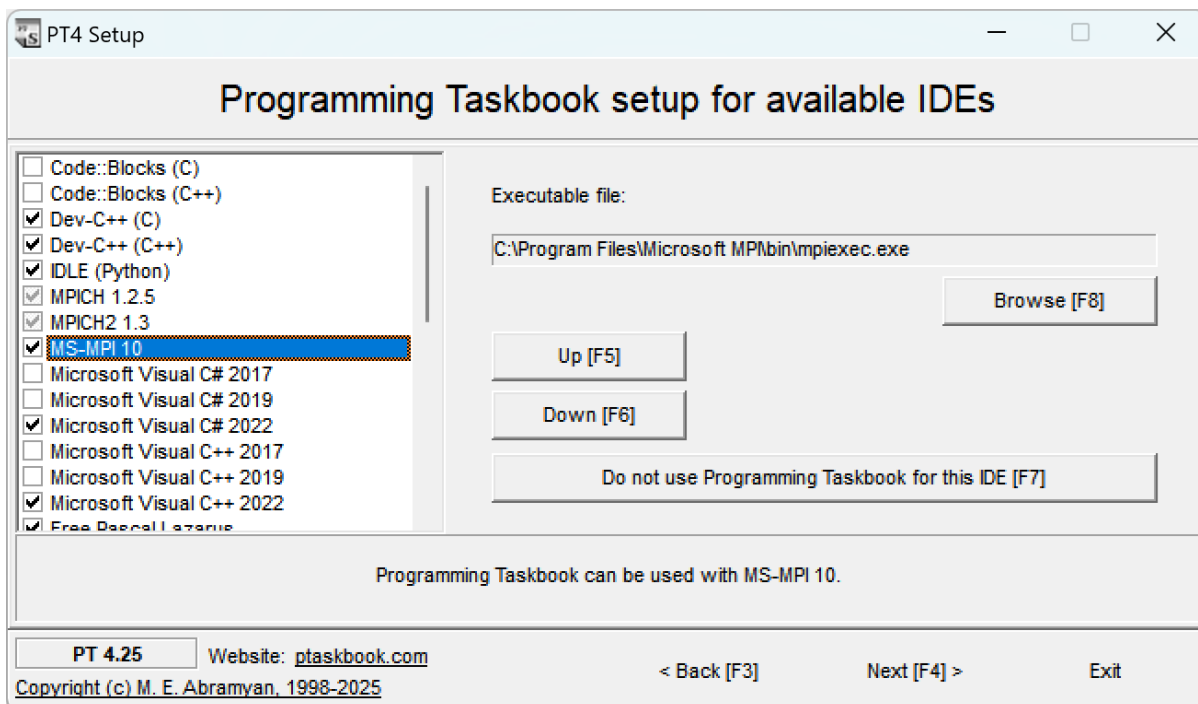


Fig. 1. The PT4Setup program window with a list of found IDEs

If there are several MPI support systems, one will be active and the others (with a gray checkbox) will be temporarily disabled. To activate the other MPI support system, click on its gray checkbox.

After installing all the specified programs, you can start solving tasks from the PT for MPI-2 taskbook.

Throughout the textbook, we will assume that the Microsoft Visual Studio 2022 environment is used when solving tasks, and MS-MPI is selected as the active MPI support system.

1.1.2. Basic concepts of MPI programming

We will begin our introduction to parallel programming by examining the following simple task from the initial group MPI1Proc (see Section 2.1). This will allow us not only to become familiar with the basic concepts of parallel programming based on message passing, but also to study the capabilities of the electronic taskbook related to data input and output, as well as debug output.

MPI1Proc2. Input an integer A in each process of the `MPI_COMM_WORLD` communicator and output doubled value of A . Also output the total number of processes in the *master process* (that is, a rank-zero process). For data input and output use the input-output stream `pt`. In the master process, duplicate the data output in the debug section by displaying on separate lines the doubled value of A and the total number of processes (use two calls of the `ShowLine` function, which is defined in the taskbook along with the `Show` function).

First of all, let us clarify the basic terms of parallel MPI programming. When a program is executed in parallel, several instances of the program are launched. Each launched instance is a separate *process* that can interact with other processes by exchanging messages. MPI functions provide a variety of means for implementing such interaction.

To identify each process in a process group, the concept of rank is used. The *rank of a process* is the ordinal number of the process in the process group, counted from zero (thus, the first process has rank 0, and the last process has rank $K - 1$, where K is the number of processes in the group). In this case, a process *group* may include only a part of all running processes of the parallel application. Note that in task formulations, the letter K is usually used to denote the number of processes.

A special entity of the MPI library, called a *communicator*, is associated with a group of processes. Any interaction between processes is possible only within a particular communicator. The standard communicator, which contains all processes launched during parallel execution of a program, has the name `MPI_COMM_WORLD`. The constant `MPI_COMM_NULL` corresponds to an "empty" communicator, which cannot be used to send messages. Each process also has a communicator `MPI_COMM_SELF`, which is associated only with this process. A communicator can be interpreted as a channel connecting processes included in

a certain group. It is often convenient to organize additional channels that, for example, do not contain all processes or in which the order of their sequence is changed. In this situation, new communicators are created, information about which is stored in descriptor variables of the MPI_Comm type. Working with communicators is discussed in MPI5Comm and MPI8Inter task groups (see also Sections 1.2.7–1.2.9, 1.3.1, and 1.3.7). The tasks of the initial four groups always use the standard communicator MPI_COMM_WORLD.

A process of rank 0 is often called the *master process*, and the remaining processes are *slave processes*. Typically, the master process plays a special role with respect to slave processes, passing its data to them or receiving data from all (or some) slave processes. In the MPI1Proc2 task under consideration, all processes must perform the same action—read one integer and output its double value, and the master process, in addition, must perform an additional action—output the number of all running processes (in other words, the number of all processes included in the communicator MPI_COMM_WORLD). Note that in this simple task, the processes do not need to exchange messages with each other (all tasks of the MPI1Proc group are like this).

1.1.3. Creating a template for a parallel program

The process of solving a task using the PT for MPI-2 taskbook starts with creation of a project template for the selected task. All necessary libraries (associated with the taskbook and with the selected MPI support system) will already be connected to this project; in addition, the main file of this project will contain code fragments necessary for the execution of any parallel program.

The **PT4Load** program, which is part of the taskbook, is intended to create a template project. The easiest way to run this program is with the Load.lnk or Panel.lnk shortcuts that are automatically created in the working directory (by default, the working directory is called PT4Work and is located on the C drive). After running the program, its window will appear on the screen (Fig. 2).

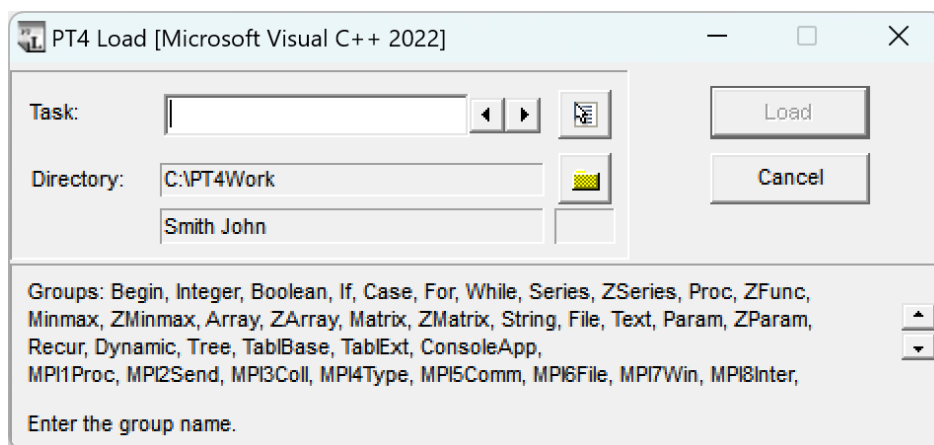



Fig. 2. Window of the **PT4Load** program

This is what the window looks like if the current IDE is Microsoft Visual Studio 2022 for C++. To change the current environment, simply right-click in the window (or press the button  or key [Shift]+[F10]) and select a new environment from the pop-up menu that appears (for example, **Dev-C++ (C++)**); the name of the selected environment will appear in the window header.

The pop-up menu is shown in Fig. 3. In addition to the list of available environments, the pop-up menu contains a list of available MPI support systems (indicating the selected one). Also it allows you to select the interface language (Russian or English), and perform a number of additional actions to configure the working directory and the PT4Load program.

You should check for task groups that start with the MPI prefix (MPI1Proc, etc.). They will appear in the task list only after installing the PT for MPI-2 taskbook and only if the C++ language environment is selected as the current IDE.

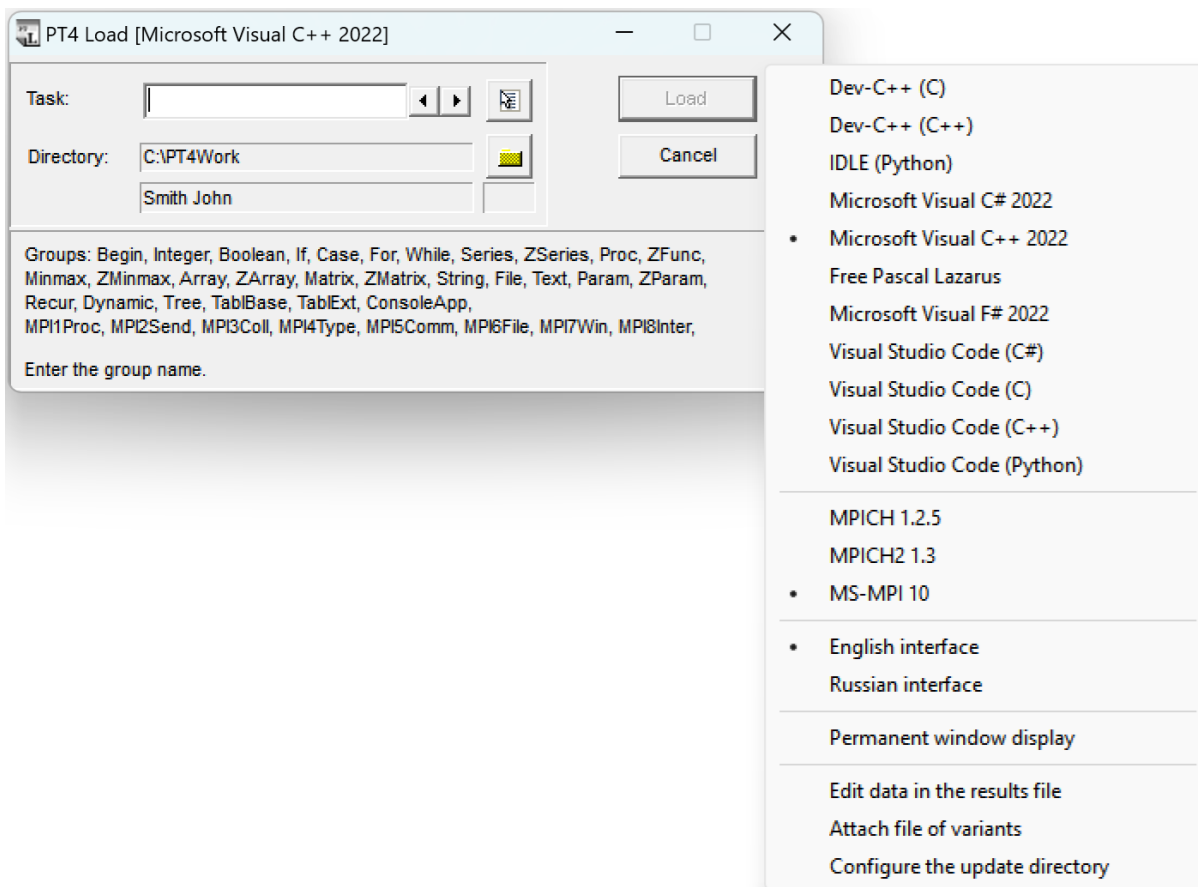


Fig. 3. PT4Load window with expanded pop-up menu

Let us select the required IDE, and then input the text MPI1Proc2 in the **Task** field (it is not necessary to input the full name of the group; it is enough to input the text MPI1, which uniquely identifies the group, then press the space bar and specify the task number 2). As a result, the **Load** button will become available;

in addition, a brief description of the selected group and the amount of tasks included in it will be shown at the bottom of the window (Fig. 4).

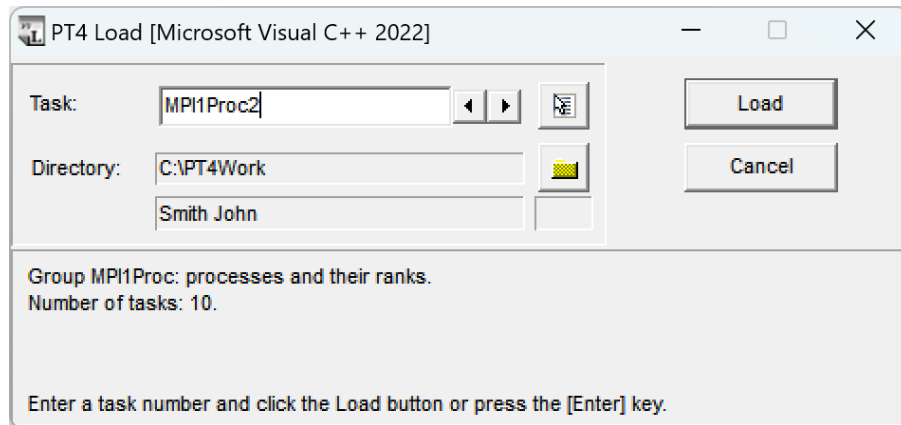


Fig. 4. The **PT4Load** window after the task name input

Pressing the **Load** button or the [Enter] key, we will create a template project for the specified task, which will be immediately loaded into the selected IDE.

The project created for the C++ language always has the name `ptprj`; this allows, in particular, to significantly reduce the number of files created in the working directory when performing various tasks. It includes a number of files, the main one of which is the `cpp` file with the name which coincides with the name of the task being performed (in our case, `MPI1Proc2.cpp`). This file is automatically loaded into the IDE code editor; the solution of the task must be input in this file. Here are the contents of the `MPI1Proc2.cpp` file:

```
#include "pt4.h"
#include "mpi.h"

void Solve()
{
    Task("MPI1Proc2");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
}
```

At the beginning of the program, there are directives for connecting auxiliary header files `pt4.h` and `mpi.h`. Then there is the `Solve` function, which should contain the solution of the task.

When analyzing the `MPI1Proc2.cpp` file, a natural question arises: where is the "start" function of the application (usually named `main` or `WinMain`)? This

function is located in another file of the project, since its contents do not require editing. In it, the initialization of the taskbook is performed, after which the Solve function is called. Then, if necessary, exceptions that may arise during the execution of the Solve function are caught, and at the end, final actions related to the analysis of the obtained solution are performed.

The program template for parallel programming tasks contains additional statements that are not present in the templates for "non-parallel" tasks. These statements must be used in almost any parallel MPI program, so they are automatically added to the program when it is created.

Let us discuss the contents of the Solve function in more detail. Its first statement is the call for the Task function, which initializes the required task (see Section 3.1.2). This statement is present in the template programs for all tasks, including those not related to parallel programming. The Task function is implemented in the core of the Programming Taskbook (the dynamic library) and is available in the program due to the header file pt4.h connected to it. In addition to the header file pt4.h, the working directory must contain the pt4.cpp file, which contains definitions of the functions declared in the file pt4.h (all these files are automatically added to the working directory when creating a template project).

The remaining statements of the Solve function are associated with the MPI library. In Section 1.1.1, it was noted that the taskbook uses the MPI library, which is part of the MPICH or MS-MPI systems, a widely used free software implementation of the MPI for Windows. The functions and constants of the MPI library are available to the program due to the header file mpi.h connected to it. The implementation of the functions from the mpi.h file is contained in the object file mpich.lib, which must be connected to any project in C/C++ languages that uses the MPI library. However, in our case, this connection *has already been made* during the creation of the template project, so no additional actions related to this connection are required.

Note 1. The object lib file for the MPICH2 and MS-MPI systems has the name mpi.lib, but the taskbook uses the name mpich.lib for these libraries, which coincides with the name of the similar library for the MPICH 1.2.5 version. This allows you to specify the same settings for projects regardless of which version of the MPICH system should be used (the version of the mpich.lib library that is contained in the working directory is always linked to the project).

Note 2. To connect an additional lib file to the project in the Visual Studio, you need to call the project properties window (**Project** < *project name* > **Properties...** command), go to the **Configuration Properties | Linker | Input** section in this window and specify the name of the required file in the **Additional Dependencies** input field, for instance, mpich.lib; (with a trailing semicolon).

Similar actions need to be performed in the Code::Blocks and Dev-C++ environment. In Code::Blocks, execute the **Project | Build options...** command; in the window that appears, go to the **Linker settings** tab and specify the required library in the **Link libraries** section. In Dev-C++, execute the **Project | Project options...** command; in the window that appears, go to the **Parameters** tab and specify the required library in the **Linker** section.

As for the Visual Studio Code editor, an additional compilation option must be input in the tasks.json file located in the .vscode subdirectory of the working directory. This option has the form "\${fileDirname}\\mpich.lib" and should be located at the end of the args section.

The MPI_Initialized(int *flag) function call allows us to determine whether the parallel mode is initialized for the program or not. If the mode is initialized, the output parameter flag takes a non-zero value. It should be noted that the parallel mode is initialized by the MPI_Init function (see Note 3), which is missing in the given code. This is because the taskbook itself is responsible for the initialization, and it is performed *before* the program proceeds to executing the code contained in the Solve function. However, such initialization is not always performed by the taskbook. If the program is launched in *demo mode* (for this, it is sufficient to supplement the task name with the "?" symbol when calling the Task function, for example, Task("MPI1Proc2?")), then the taskbook *does not initialize the parallel mode*. In this situation, calling MPI functions (other than MPI_Initialized) in the Solve function may lead to incorrect program working. The call to the MPI_Initialized function and the conditional statement that follows it are intended to "skip" all other statements of the Solve function during program execution if the program is not running in parallel mode.

Note 3. The MPI_Init function has two parameters: (int *argc, char ***argv); the first parameter specifies the number of command line parameters, and the second contains these parameters themselves as an array of type char*. The parameters are passed by reference; this is due to the fact that the MPI standard provides for the possibility of implementing this function in such a way that the parameters are passed not *from* the parallel program *to* the MPI environment, but *vice versa*: *from* the MPI environment *to* the parallel program. Note also that the MPI_Init function must be called by *all* processes of the parallel application.

The last two program statements allow us to define two characteristics necessary for the normal working of any process of any parallel program: the total number of processes (function MPI_Comm_size(MPI_Comm comm, int *size)) and the rank of the current process (function MPI_Comm_rank(MPI_Comm comm, int *rank)). The current process is the one that called this function. The required characteristic is returned in the second parameter of the corresponding function (which is a *pointer*); the first parameter is the comm *communicator*, which specifies the

group of processes. By calling these functions, we can immediately use the size (the total number of processes in the MPI_COMM_WORLD communicator) and rank values in our program (the rank value must be in the range from 0 to size – 1).

Note 4. Any MPI function returns information about the success of its execution. In particular, upon successful completion, the function returns the value MPI_SUCCESS. However, as a rule, the return values of MPI functions are not analyzed, and errors that occur are processed by a special *error handler*. When solving tasks on parallel programming using the PT for MPI-2 taskbook, a special error handler is used, which is defined in the taskbook and provides output of information about errors in a special section of the taskbook window, namely, *the debug section* (see Section 3.1.3). Some MPI capabilities related to error handling are discussed in MPI5Comm23–MPI5Comm24 tasks; a more detailed description of the MPI facilities related to error handling is given, for example, in [8, Chapter 8] and [10, Chapter 11].

Note 5. The MPI library also provides the MPI_Finalize() function without parameters, which finishes the parallel part of the program (after calling this function, other functions of the MPI library cannot be used). However, in the part of the program that is developed by the student, this function cannot be called, since after executing this part of the program, the taskbook must "collect" all the results obtained in the slave processes (in order to analyze them and display them in the window of the master process), and for this purpose, the program must be in parallel mode. Therefore, the taskbook takes on the responsibility not only to *initialize* the parallel mode (by calling the MPI_Init function at the beginning of the program execution), but also to *terminate* it (by calling the MPI_Finalize function at the end of the program).

As noted above, the MPI_Initialized function returns a non-zero flag if the MPI_Init function was called in the program. However, calling the MPI_Finalize function does not affect the result of the MPI_Initialized function. The ability to check whether the MPI_Finalize function was called was implemented only in the MPI-2 standard. It added the MPI_Finalized(int *flag) function, which returns a non-zero value for the flag parameter if the program has already called the MPI_Finalize function.

1.1.4. Running a program in parallel mode

Now let us find out how this project can be launched in parallel mode. When compiling and launching any program from the integrated environment (even with the MPI library connected), it will be launched in *a single copy*. It will also be launched in a single copy if we exit the integrated environment and launch the compiled exe file of this program.

To run a program in parallel mode, a control program (*a host application*) is required, which, firstly, ensures that the required number of instances of the original program are launched and, secondly, intercepts messages sent by these instances (*processes*) and forwards them to their destination.

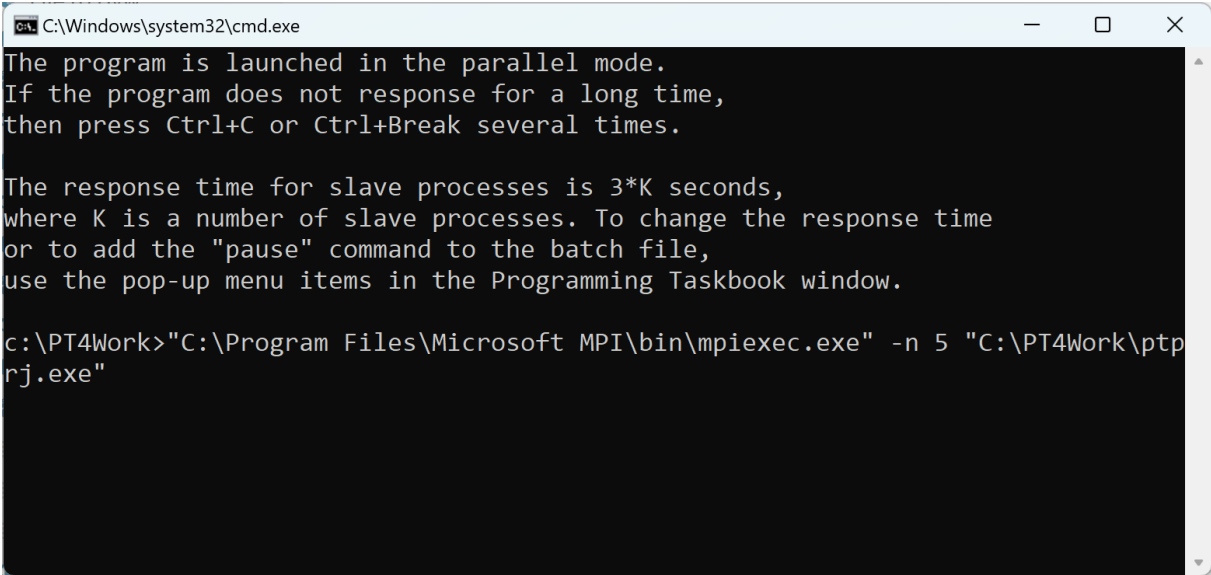
In Section 1.1.1, it was already noted that instances of "real" parallel programs are usually launched on different computers connected in a network (*a computer cluster*), or on supercomputers equipped with a large number of processors. It is in the situation where each process is executed on its own processor that the maximum efficiency of parallel programs is ensured. Of course, to check the correctness of our learning programs, it is enough to launch all their instances on one local computer. However, the control program is necessary in this case too.

As a control program for parallel programs, the PT for MPI-2 taskbook uses an application included in the MPI support system. In MPICH, it is named MPIRun.exe (and is contained in the MPICH\mpd\bin directory), in MPICH2 and MS-MPI, it is named mpiexec.exe (and is contained in the MPICH2\bin and Microsoft MPI\bin directories respectively). To run an executable file in parallel mode, it is sufficient to run the corresponding control program (MPIRun.exe or mpiexec.exe), passing it the full file name, the required number of processes, and some additional parameters. Since such runs will have to be performed repeatedly during program testing, it is advisable to create a *batch file* (a file with the .bat extension) containing a call to the control program with all the necessary parameters. However, even in this case, the process of testing a parallel program will not be very convenient: each time after making the necessary corrections to the program, it will have to be recompiled, after which, leaving the IDE, the batch file will have to be run. After analyzing the results of the program's work, you will need to return to the IDE to make further changes to it, then compile it again and run the batch file, etc.

Note 1. Microsoft Visual Studio provides a mechanism that simplifies testing programs that require a control program to run. In the project settings (menu command **Project** | < **project name** > **Properties...**) in the **Debugging** section, you can specify this control program in the **Command** field; in our case, it will be **MPIRun.exe** or **mpiexec.exe**. The program launch parameters are specified in the **Command Arguments** field; the parameters required in our case are described further in this section. You should also configure the **Working Directory** field by specifying the directory where the executable file for our parallel program is located. After making such settings, launching the application under development will lead to launching the control program. Thus, there is no need to launch a separate batch file. However, in this case, it will be necessary to add a fragment to the program that ensures its suspension at the end of execution, since without it, the control program window will be immediately closed,

and it will not be possible to view the results obtained. It should also be noted that in many other IDEs (in particular, in Code::Blocks and Dev-C++), the control program can only be specified when testing *libraries*, so when using such environments, it will not be possible to do without auxiliary batch files to launch the program in parallel mode.

To ensure that actions to launch a parallel program do not distract from solving the task, the PT for MPI-2 taskbook performs them itself. Let us demonstrate this by means of the example of our project for solving the MPI1Proc2 task, which is ready to run. Press the [F5] key in the Visual Studio; as a result, the program will be compiled and, if the compilation is successfully completed, the program will be launched. Since we have not made any changes to the template, the compilation should be completed successfully. When the program is launched, a console window similar to the one shown in Fig. 5 will appear on the screen.

A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. The text inside the window reads: "The program is launched in the parallel mode. If the program does not response for a long time, then press Ctrl+C or Ctrl+Break several times." followed by "The response time for slave processes is 3*K seconds, where K is a number of slave processes. To change the response time or to add the 'pause' command to the batch file, use the pop-up menu items in the Programming Taskbook window." At the bottom, a command is entered: "c:\PT4Work>"C:\Program Files\Microsoft MPI\bin\mpiexec.exe" -n 5 "C:\PT4Work\ptprj.exe".

```
C:\Windows\system32\cmd.exe
The program is launched in the parallel mode.
If the program does not response for a long time,
then press Ctrl+C or Ctrl+Break several times.

The response time for slave processes is 3*K seconds,
where K is a number of slave processes. To change the response time
or to add the "pause" command to the batch file,
use the pop-up menu items in the Programming Taskbook window.

c:\PT4Work>"C:\Program Files\Microsoft MPI\bin\mpiexec.exe" -n 5 "C:\PT4Work\ptprj.exe"
```

Fig. 5. Console window with information about running the program in parallel mode

After a few lines of informational message, this window displays a command line that runs the `ptprj.exe` program in parallel mode under the control of `mpiexec.exe`:

```
C:\PT4Work>"C:\Program Files\Microsoft MPI\bin\mpiexec.exe"
-n 5 "C:\PT4Work\ptprj.exe"
```

The number "5" specified before the full name of the exe file (`C:\PT4work\ptprj.exe`) means that the corresponding process will be launched in five copies. MPICH systems may use some additional parameters, for example, the `-nopopup_debug` parameter, which disables the output of error messages in a separate window, and the `-localonly` parameter, which ensures that all instances of the program are launched on the local computer.

Immediately after the console window appears, if the parallel program named `ptprj.exe` has not been launched before, another window may appear on the screen (Fig. 6), in which you should select the **Allow access (Разрешить доступ)** option.

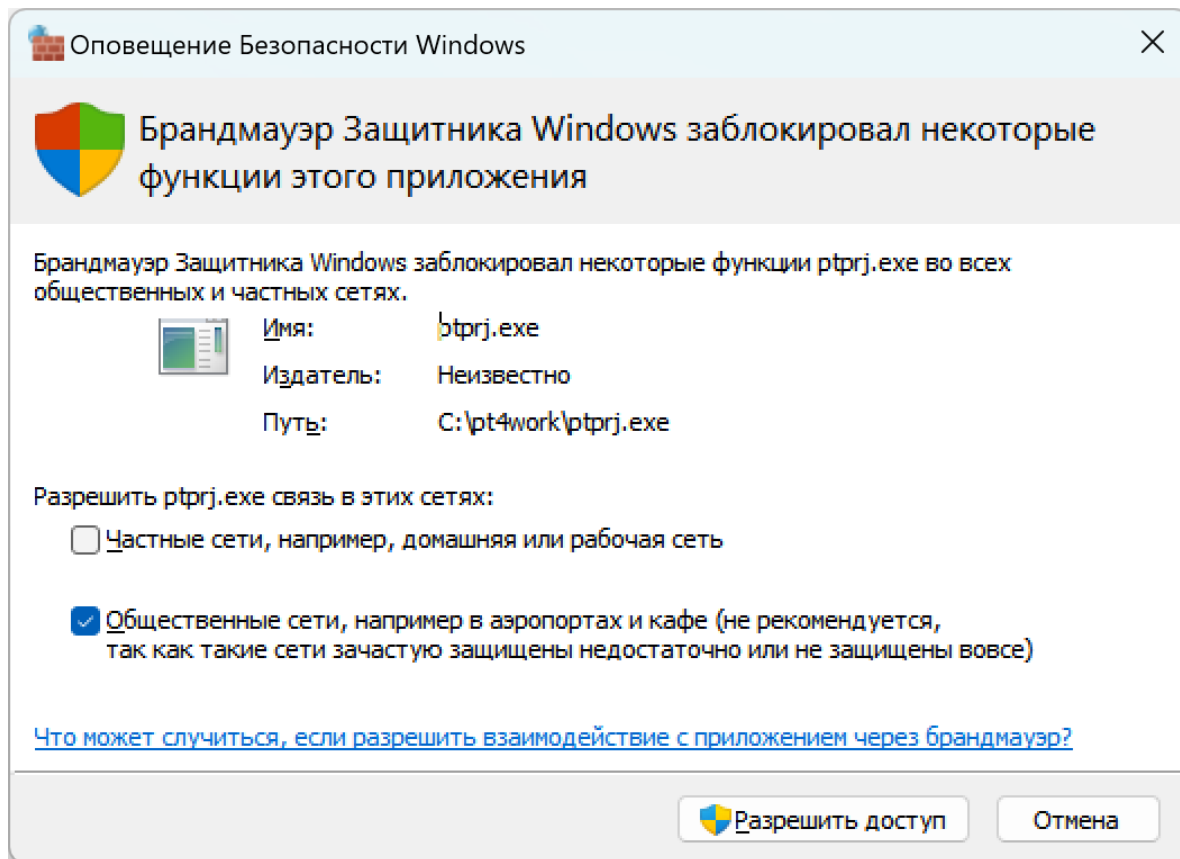


Fig. 6. Window with a request to block a running parallel program

Finally, the taskbook window will appear on the screen (Fig. 7). This window is no different from the window that appears when executing a usual, "non-parallel" program. However, in this case, the information that none of the input-output operations were performed applies *to all processes launched in parallel mode*.

To complete the program, you must, as usual, close the taskbook window (for example, by clicking the **Exit (Esc)** button or pressing the [Esc] key). After closing the taskbook window, the console window will immediately close too, and we will return to the IDE from which our program was launched.

Thus, having compiled and launched the program from the IDE, we are able to immediately ensure its execution in parallel mode. This happens due to a rather complicated mechanism that is implemented in the core of the Programming Taskbook. In order to successfully solve the training tasks, a detailed understanding of this mechanism is not required, so we will only give a brief description of it here.

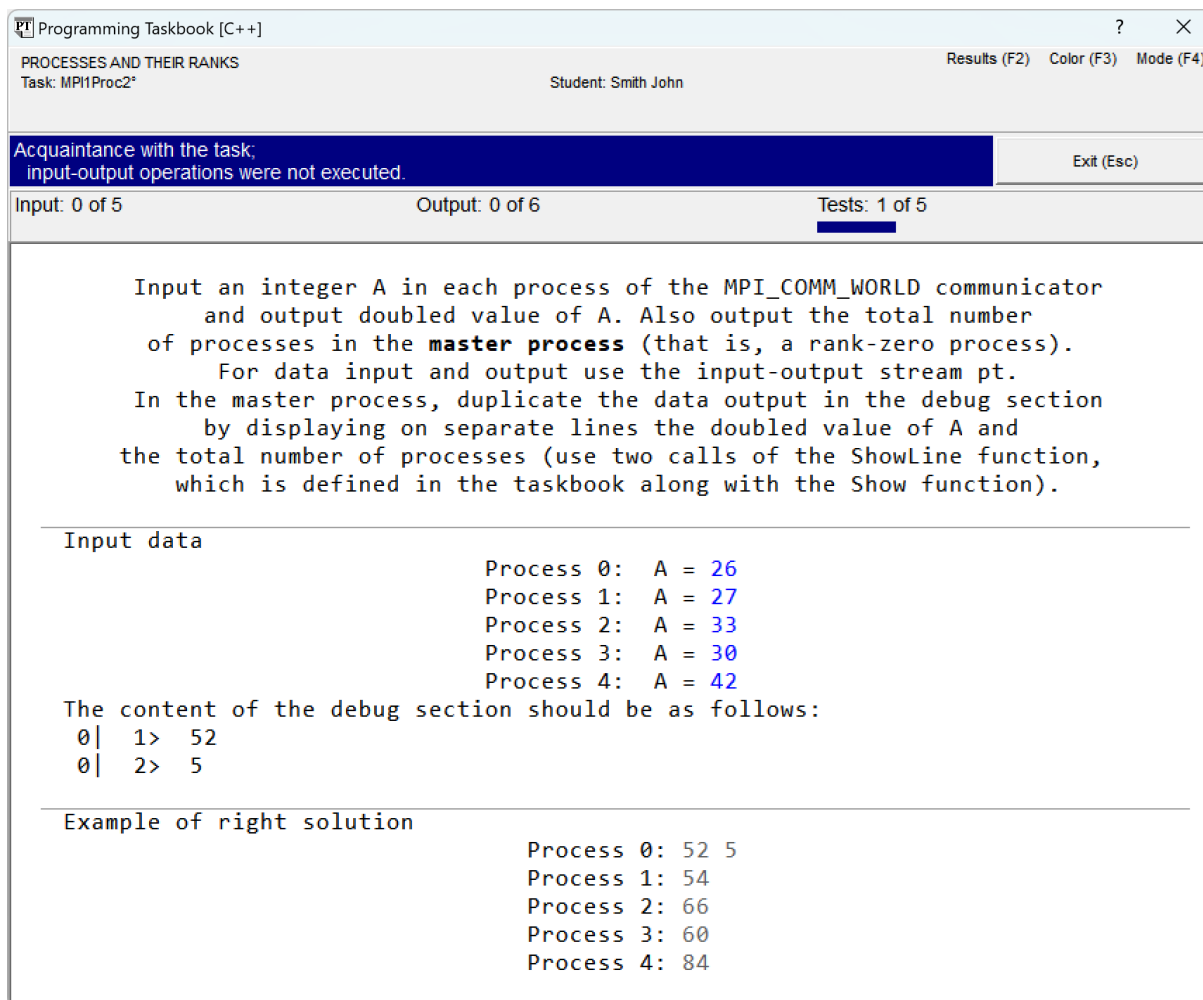


Fig. 7. Acquaintance running of the MPI1Proc2 task

In fact, the program launched from the IDE *does not try to solve the task* and is executed in the usual, "non-parallel" mode. Having found that the task belongs to the group of tasks on parallel programming, it only creates a batch file \$pt_run\$.bat with comment lines and a command line that calls the program mpiexec.exe with the necessary parameters. Then it launches this batch file for execution and goes into the mode of waiting for the completion of the batch file. The program mpiexec.exe launched by the batch file, in turn, launches the required number of program instances (*processes*) in parallel mode, and these processes actually try to solve the task. In particular, the taskbook offers each process its set of initial data and expects a set of results from it.

Because, in our program, no input-output operation was specified in any process, this launch of the parallel program is considered as acquaintance one, and the corresponding message is shown in the information section of the taskbook window. Note that this window is displayed by *the main process* of the parallel program, while all slave processes (as well as the first instance of the program that created and launched the batch file) work in "invisible" mode.

When the task window is closed, all processes of the parallel program are terminated, after which the batch file is terminated too, and finally, having discovered that the batch file has successfully completed its work, the instance of our program that was launched from the IDE also terminates its work.

Note 2. The "starting" copy of the program performs one more action: it automatically *unloads* all parallel program processes from memory if they "hang" as a result of incorrect programming. If, during the execution of a parallel program, the taskbook window does not appear within 20–30 seconds, this usually means that the program has hung (sometimes a program hangs after closing the taskbook window; in this case, the console window does not close immediately, i. e., the batch file does not complete its work). In any of these situations, you must close the console window by following the instructions given in it, namely, by pressing the key combination [Ctrl]+[C] or [Ctrl]+[Break] several times (or simply by clicking the close button **X** on the window header). If the starting copy of the program detects that the batch file has completed its work, but hung parallel program processes remain in memory, *it automatically unloads all these processes from memory*. Note that while hung processes remain in memory, they do not allow you to change the executable file of the program (in particular, replace the executable file with a new compiled version). In such a situation, it is necessary to call the **Windows Task Manager** (using the combination [Ctrl]+[Alt]+[Del]) and manually terminate the execution of all hung processes in the **Processes** tab. The automatic unloading of hung processes performed by the starting copy of the program saves the student from having to perform such actions.

Note 3. Sometimes only some of the slave processes of a parallel application hang. In this case, the master process usually displays its window and reports which slave processes are hanging (and also displays the results from those slave processes that are not hanging). This information can be useful when troubleshooting errors.

The master process considers a slave process to be hung if it does not receive a response from it within a certain period of time (proportional to the number of processes). By default, the interval is $3 * K$ seconds, where K is the number of processes (this is reported in the comment that is displayed in the console window). In some rare cases, when executing tasks on low-performance computers, a situation may arise when some slave processes do not have time to complete their part of the work within the allotted waiting time, and the master process considers them to be hung, although the solution to the problem is correct. In such cases, you can increase the waiting time using the command in the pop-up menu of the taskbook window **Increase the waiting time for a response from slave processes**.

1.1.5. Executing MPI1Proc2 task

Now we have become familiar with the mechanism of the program's operation in parallel mode, and solving our task will not be a problem for us.

Let us start with the input data. By task condition, one integer is given in each process. Let us go to the empty line located below the call of the `MPI_Comm_rank` function. If this section of code is reached during the execution of the program, it means that the program was launched as one of the processes of the parallel application (otherwise, the return statement specified in the conditional statement would have been executed). Thus, in this place of the program, you can input an element of the initial data, having previously described it (here and below, we will only provide the contents of the `Solve` function):

```
Task("MPI1Proc2");
int flag;
MPI_Initialized(&flag);
if (flag == 0)
    return;
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int n;
pt >> n;
```

The added operators are highlighted in bold. To input the initial data, we used a special input stream `pt` defined in the taskbook (see Section 3.1.2). This stream allows you to input data of any scalar type, in particular, `int`, `double`, and `char*` (note that in the tasks included in the PT for MPI-2 taskbook, only data of these types are used). Having launched the new version of the program, we will see the taskbook window on the screen (Fig. 8).

The taskbook has detected that the input data has been completed, and thus the program has started solving the task. However, no resulting data element has been output. Strictly speaking, this indicates an erroneous solution, but the first step towards the correct solution has been taken: *all the initial data has been input correctly*. In such a situation, the taskbook displays the message on a light blue background "*Correct data input: all required data are input, no data are output*".

Note that the data input is performed *in all processes of the parallel application*. Also note that the number of processes is different for each running of the program. The number of processes changes for all runs of the program; this allows us to test the solution for different numbers of processes.

Let us close the taskbook window and return to our program code. In each process, we need to output the doubled value of the input number, so we add the following statement to the end of the `Solve` function:

```
pt << 2 * n;
```

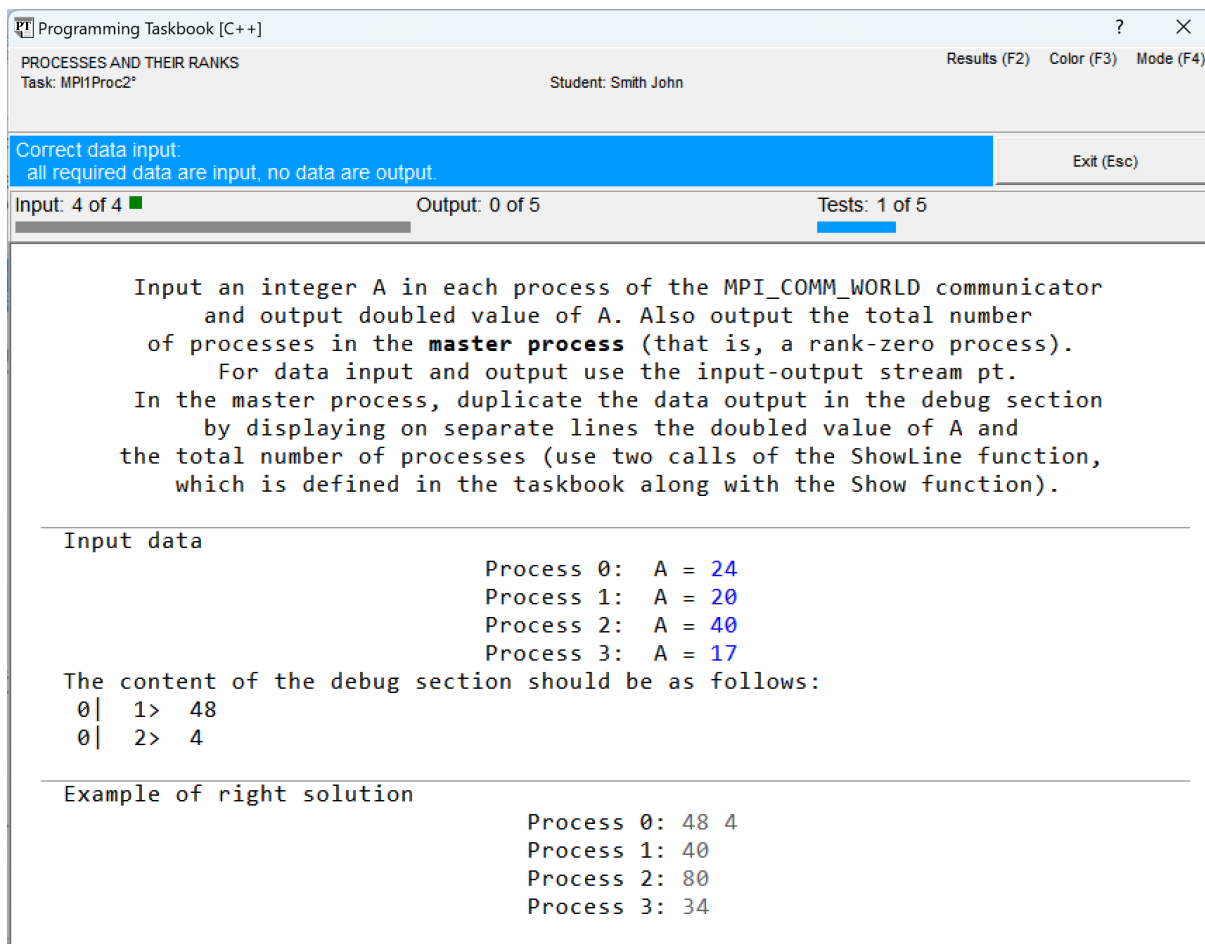


Fig. 8. The taskbook window with information about the correct input of the initial data

The same pt stream is used to output data when solving problems; thus, this stream is *an input-output stream*.

Running the new version will result in an error message (Fig. 9).

Now all the slave processes output the required results. In addition, the doubled number has been output in the master process. This data is correct, as can be verified by comparing the values output in the results section and those shown in the section with the example of the correct solution.

However, the master process also needed to output the number of processes included in the communicator, and this was not done. Therefore, the information panel contains the message *"Some data are not output. The error has occurred in the process 0"*, and the message is displayed on an orange background. Orange is used to highlight errors related to the input or output of an *insufficient* amount of data. When trying to input or output *excess* data, the information panel is highlighted in crimson; if errors occur related to the use of data *of the wrong type*, the color of the panel becomes purple. The red background color is used for all other errors.

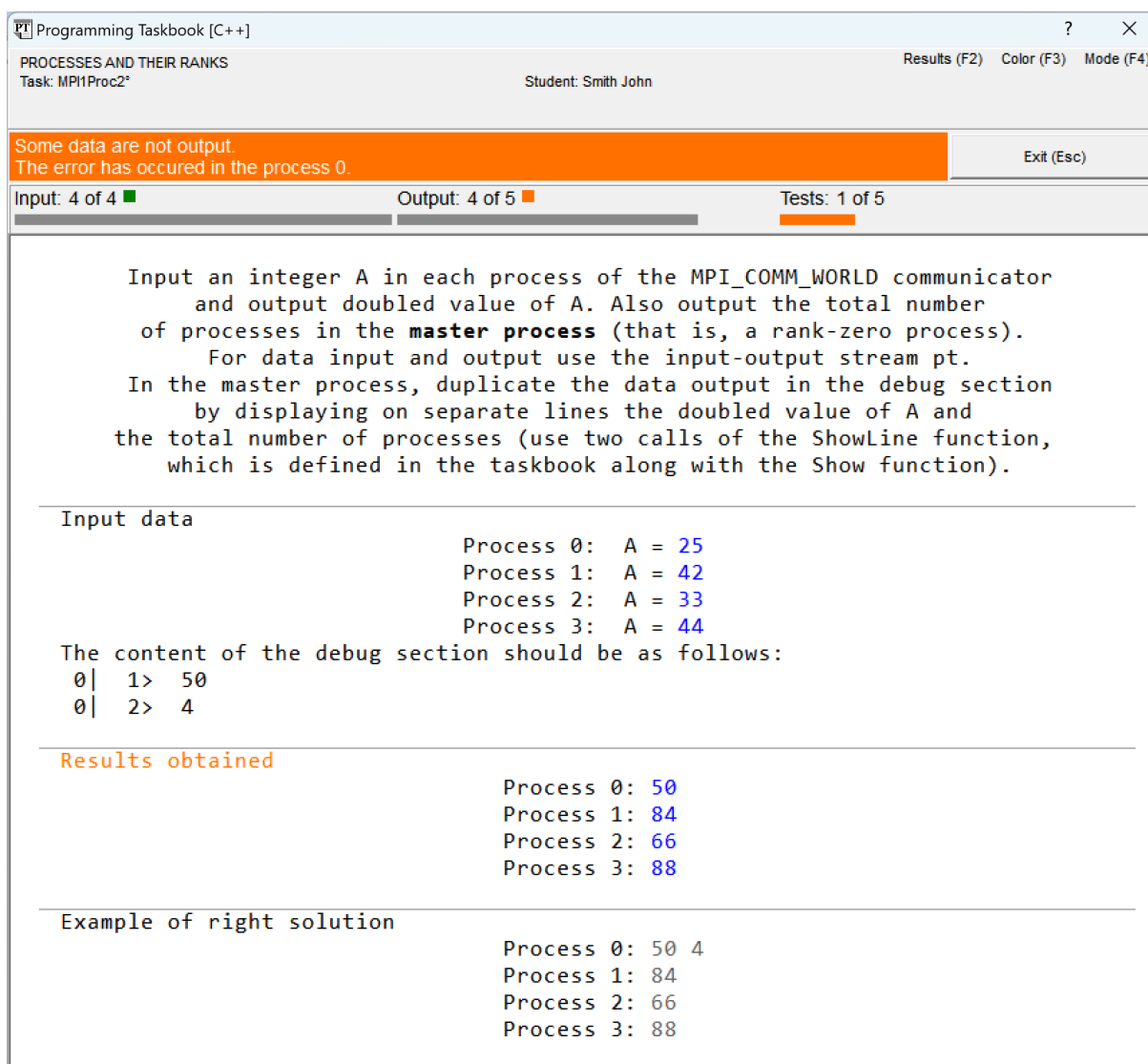


Fig. 9. The taskbook window with information about the error in the master process

The number of processes is stored in the size variable. Let us try to output the value of this variable at the end of the Solve function:

```
pt << size;
```

The taskbook window will look like the one shown in Fig. 10.

We can verify that all the resulting data has been output. However, the solution is still considered to be erroneous, since we have now attempted to output *superfluous* data (namely the size value) in the *slave processes*. As noted above, the crimson color is used to highlight errors related to an attempt to input or output superfluous data.

If errors are found in slave processes, then the taskbook window displays an additional *debug section*, which displays more detailed error information for each slave process.

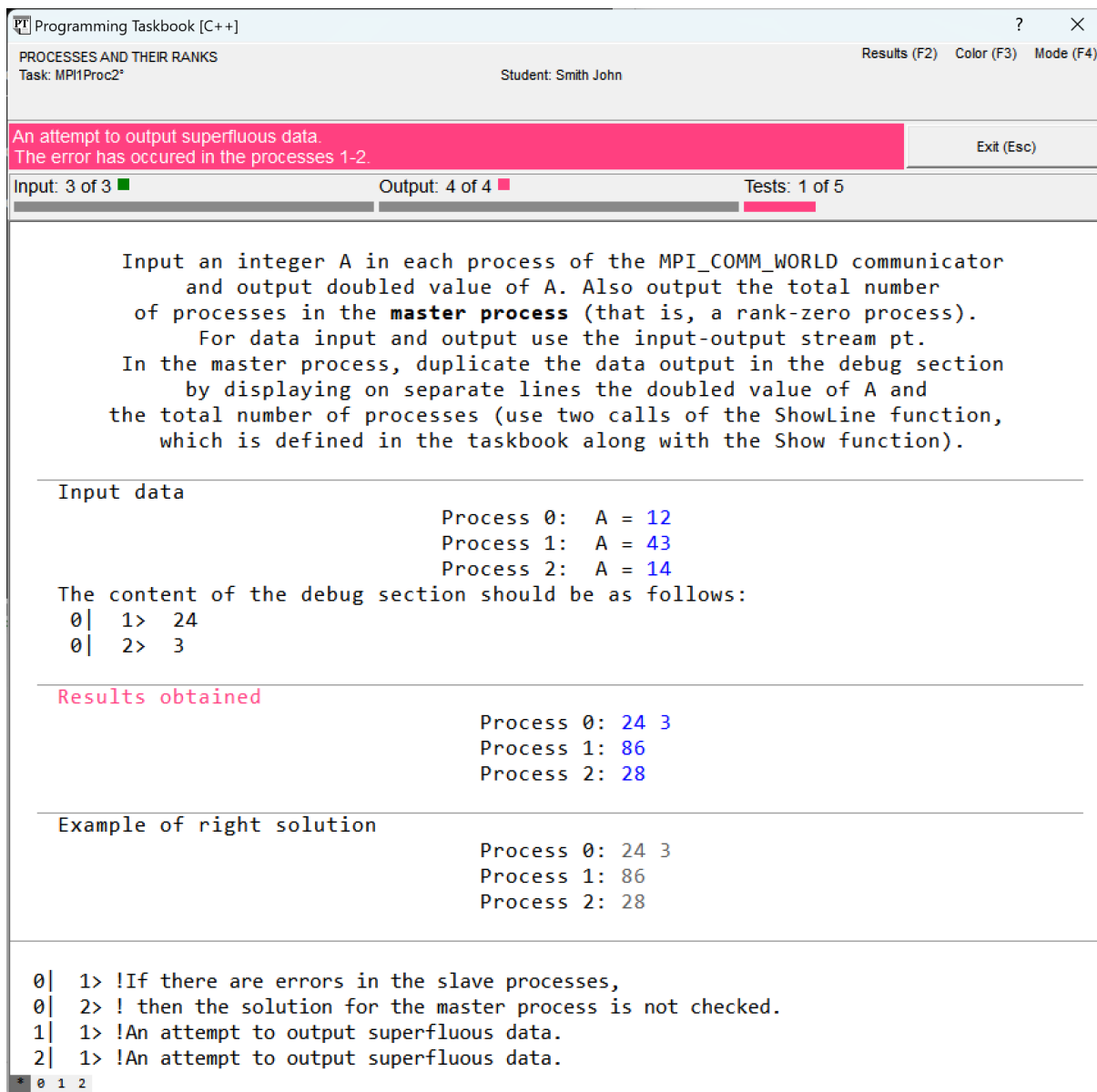


Fig. 10. The taskbook window with information about an attempt to output superfluous data

You can determine the process associated with a particular message displayed in the debug section by the number indicated on the left side of the line (before the "|" symbol). All lines associated with a particular process are numbered independently; their numbers are indicated after the process number and separated from the message text by the ">" symbol. To display only messages associated with a particular process in the debug section, simply click the marker with the number (rank) of this process (all markers are located on the lower border of the window) or press the corresponding numeric key. To display summary information on all processes, select the marker with the "*" symbol or enter this symbol from the keyboard (you can also cycle through the markers using the arrow keys [←] and [→]). If a message line in the debug section begins with the "!" symbol, then this means that this message is an *error message* and

was added to the debug section by the taskbook itself. The program can output its own messages to the debug section; This possibility will be discussed in detail below (see also Section 4.1.4).

If the taskbook detects an error in at least one slave process, it does not analyze the result obtained in the master process (this is also reported in the debug section, see Fig. 10).

In order for the size value to be output only in the master process, it is necessary to make sure that the rank of the current process is 0 before performing this action. By adding the appropriate check, we get a solution that the taskbook will consider correct:

```
Task("MPI1Proc2");
int flag;
MPI_Initialized(&flag);
if (flag == 0)
    return;
int rank, size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
int n;
pt >> n;
pt << 2 * n;
if (rank == 0)
    pt << size;
```

When a new solution version is launched, five console windows will be displayed on the screen in sequence, each of which is associated with the parallel program being executed. Thus, a single launch of the program from the IDE leads to a whole series of launches of this program in parallel mode, which allows you to immediately test the resulting solution on *several* sets of input data. The test series is completed either when an error is detected or when the required number of tests is successfully completed (for all tasks included in the PT for MPI-2 taskbook, the number of tests is five). This feature further simplifies the process of checking the correctness of the task solution.

After five successful test runs, the taskbook window will appear on the screen with a message that the task has been solved (Fig. 11).

In this case, all square markers located on the *indicator panel* (under the information panel) are green.

Each time the program is launched, the taskbook saves the results of its work in a special *results file* named results.dat. This file can be viewed using the **PT4Results** program, which is part of the taskbook (to launch this program, there is a shortcut Results.lnk in the working directory). In addition, the results can be viewed directly from the taskbook window by clicking on the **Results (F2)** label or the [F2] key. A window with a protocol of all program runnings for all tasks will appear on the screen. In our case, it will contain text like this:

```

MPI1Proc2 c12/14 15:44 Acquaintance with the task.
MPI1Proc2 c12/14 15:49 Correct data input.
MPI1Proc2 c12/14 15:54 Some data are not output.
MPI1Proc2 c12/14 15:59 An attempt to output superfluous data.
MPI1Proc2 c12/14 16:03 The task is solved!

```

After the task name, there is a symbol corresponding to the programming language used (in this case, the symbol "c", meaning that the C++ language was used), the date and time of the program launch, and a description of the result of its execution.

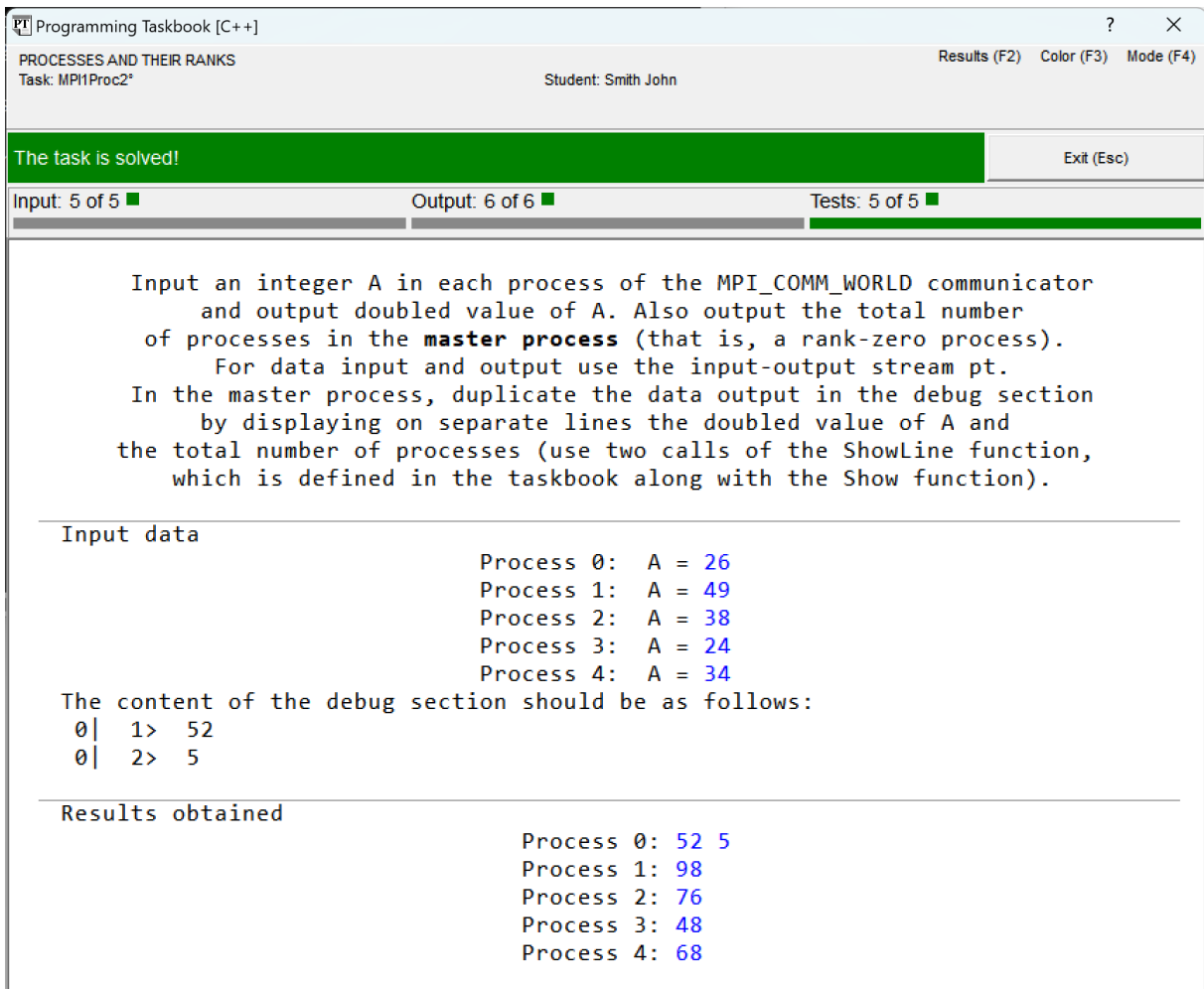


Fig. 11. The taskbook window with a message about successful solving the MPI1Proc2 task

1.1.6. Using additional information in the debug section

If you analyze the resulting solution, you will notice that it is still incomplete, since the task requires that some data be output not only in the results section, but also in the *debug section*.


We have already encountered the use of the debug section: it displays additional information about errors that occurred in slave processes. The second purpose of this section is to provide the ability to display various debug data on the

screen during the solution of a task. This ability is especially important when developing *parallel* programs, since such standard debugging tools of the integrated environment as breakpoints and watches of variables cannot be used for them.

The additional part of the MPI1Proc2 task (and other initial tasks of the MPI1Proc group) is devoted to familiarization with various options for debug information output. Although the taskbook does not analyze the contents of the debug section, this part of the task is as mandatory as the output of the obtained results, and it will be checked by the teacher. It should be taken into account that the taskbook only notes that "from its point of view" the task is solved; the final decision on whether to accept this solution is made by the teacher; he/she, in particular, pays attention to what MPI tools are used to solve the problem, whether the solution is efficient, etc. Note that displaying data in the debug section is also required in the MPI2Send and MPI5Comm group tasks related to studying *non-blocking data transfer* (Sections 2.2.2 and 2.5.4), as well as in the MPI8Inter group tasks devoted to *dynamic process creation* (Section 2.8.3).

Recall the final part of the MPI1Proc2 task formulation: "*In the master process, duplicate the data output in the debug section by displaying on separate lines the doubled value of A and the total number of processes (use two calls of the ShowLine function, which is defined in the taskbook along with the Show function)*". Note that in the input data section of the taskbook window, a comment is displayed explaining how the debug section should look if the solution is correct (see any of the figures with the taskbook window given in the previous section).

To output data in the debug section, the taskbook provides two functions: Show and ShowLine, each of which has several overloaded versions that allow you to customize the appearance of the output data and provide them with additional comments (details are given in Section 3.1.4). These functions differ in that the ShowLine function provides the line break in the debug section after data output, while the Show function does not do this (however, when the right border of the debug section is reached, an automatic move to a new line also occurs).

Note. A full description of the capabilities associated with the output of debug information is given in the information window in the **Debugging** section. If the taskbook window is active, then to display the information window, simply click the button  on the right-hand side of the taskbook window header or press the [F1] key.

To obtain the required contents of the debug section, we only need to add two calls of the ShowLine function at the end of the Solve function. Since the required data should be output only in the part of the debug section that is associated with the *master* process, the calls to these functions should be placed in the conditional statement already present in the program. Here is the final part of the Solve function, containing the full text of the task solution:

```

int n;
pt >> n;
pt << 2 * n;
if (rank == 0)
{
    pt << size;
    ShowLine(2 * n);
    ShowLine(size);
}

```

After launching a new version and testing it on five test data sets, a taskbook window will appear on the screen (Fig. 12).

By comparing the contents of the debug section with the sample shown in the source data section, we can verify that the task is now completely solved. Note that since the debug section in this case only contains data output from the master process, the bottom border of the window displays a single marker, "0", corresponding to this process.

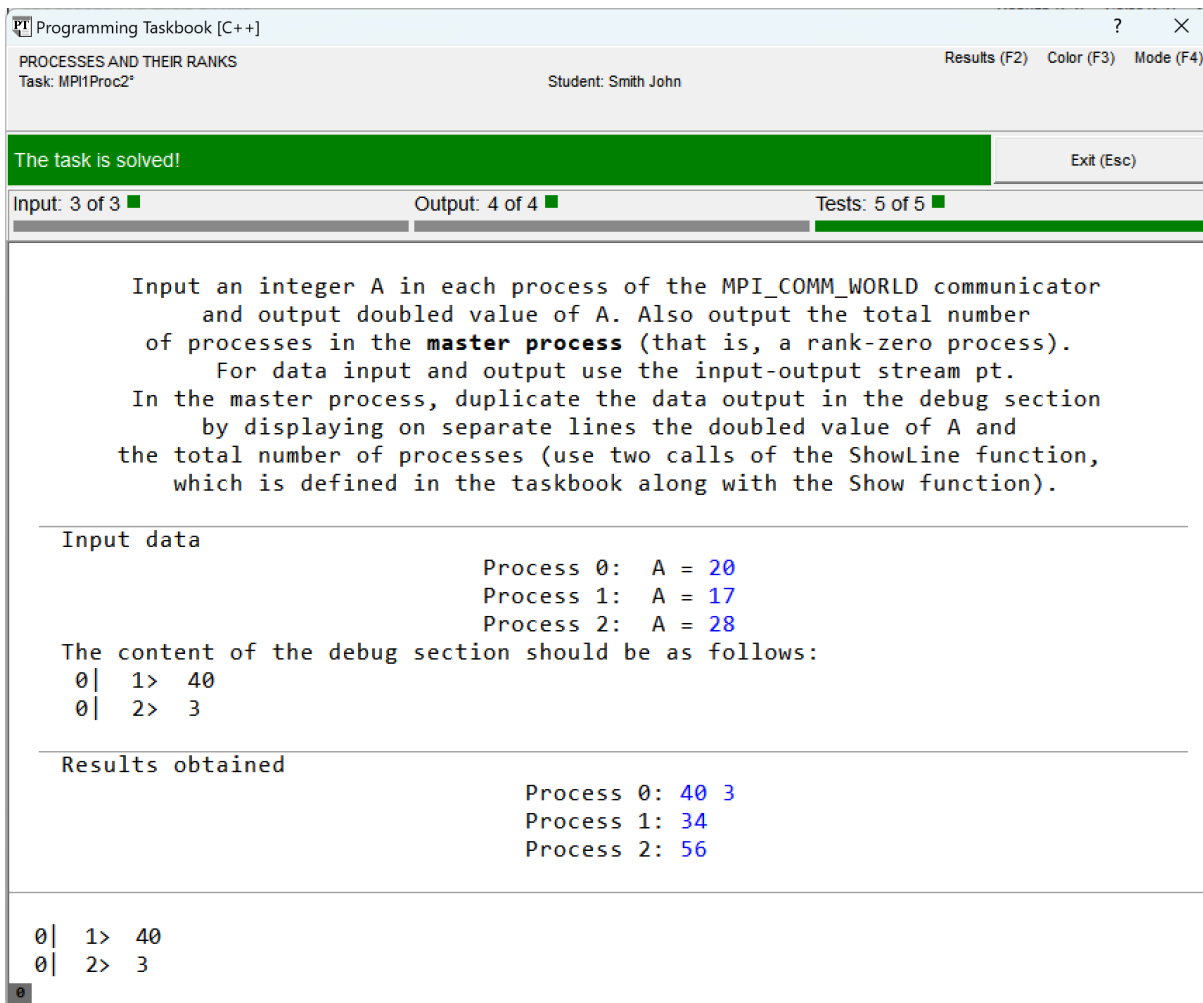


Fig. 12. The taskbook window with the final solution of the MPI1Proc2 task

So, we have solved the MPI1Proc2 task. In the process of solving it, we got acquainted with the actions for creating a template project, studied the features of executing parallel programs and those capabilities of the taskbook that simplify their launch from the IDE. We learned about the taskbook tools intended for input initial data, output results and displaying additional information in the debug section. In addition, we saw how the taskbook handles various types of errors. All this information will be useful when solving tasks devoted to various methods of message passing between parallel application processes.

1.2. Basic capabilities of the MPI interface (MPI-1 standard)

1.2.1. Blocking point-to-point communication: basic features

MPI library includes a large number of functions that implement various options for sending data between two processes. Such interaction between processes is called *point-to-point communication*.

There are two main ways for point-to-point communication: blocking and non-blocking.

In *blocking communication*, any function associated with a message sending or receiving operation exits only after that operation has completed. There are four functions for blocking message sending:

- MPI_Send – standard mode;
- MPI_Bsend – buffered mode;
- MPI_Ssend – synchronous mode;
- MPI_Rsend – ready mode.

All these functions have the same set of parameters, and all parameters are input:

- void *buf – message sending buffer;
- int count – the number of sending elements;
- MPI_Datatype datatype – type of sending elements;
- int dest – the rank of the receiving process;
- int msgtag – message identifier (non-negative number not exceeding the constant MPI_TAG_UB);
- MPI_Comm comm – communicator.

Note that according to the MPI standard, the constant MPI_TAG_UB cannot be less than 32767.

Here and below, when describing parameters, the fact that a parameter is an input parameter is not specifically mentioned; only the situation is noted when the parameter is an output parameter or both an input and output parameter. Output parameters are always passed as *pointers* to a variable whose value is changed.

The parameters datatype and comm have special types defined in the MPI library. We are already familiar with the MPI_Comm type; this type is used for

communicator descriptors. The `MPI_Datatype` type is intended to store information about the type of data being sent. Variables of this type are *descriptors* associated either with standard datatypes included in the MPI library or with user datatypes defined using the corresponding MPI functions (see Section 1.2.6). When solving tasks, we will use the standard types `MPI_INT` (corresponds to the signed type `int` of C language), `MPI_DOUBLE` (the double type) and `MPI_CHAR` (the char signed type). Other numeric types of C language also are associated with standard MPI types, for instance, the long int signed type corresponds to the `MPI_LONG` datatype, and the float type corresponds to the `MPI_FLOAT` datatype. The `MPI_BYTE` datatype corresponds to *a byte*, an integer in the range from 0 to 255. There are also standard compound datatypes designed to store *pairs of numbers*, for instance, `MPI_2INT` and `MPI_DOUBLE_INT` (the type `MPI_DOUBLE_INT` will be used when solving the `MPI3Coll23` task in Section 1.2.5).

Let us return to the blocking message sending modes and describe their main features.

In *standard mode*, the MPI environment itself determines whether a special system buffer (which is created automatically in this case) will be used. If a system buffer is used, then the send operation completes after the data has been sent to this buffer, regardless of whether the receiving process has started receiving the message (thus, in this case, standard mode works similarly to the buffered mode). If a system buffer is not used, then the send operation completes only after the receiving process has started receiving the message (in this case, standard mode works similarly to the synchronous mode). The send operation in standard mode is *non-local*, i. e., its completion may depend on the actions of another process.

Before using *buffered mode*, the sending process must define a *user buffer* of sufficient size (using the `MPI_Buffer_attach` function). The send operation completes after the data has been sent to this buffer, regardless of whether the receiving process has started receiving the message, so the buffered send operation is *local*.

In *synchronous mode*, the send operation can begin regardless of whether the receiving process has initiated the message, but will not complete until the receiving process has begun receiving the message. This operation is non-local.

In *ready mode*, the send operation can only begin if the receiving process has already initiated the receiving the message (otherwise the send operation is considered as an erroneous and its result is undefined). This operation is non-local and is used quite rarely.

For blocking message receiving, the `MPI_Recv` function is used with the following parameters:

```
void *buf – message receiving buffer (output parameter);
```


int count – the maximum number of elements in the received message (or, in other words, the size of the buffer buf in elements of the received message);

MPI_Datatype datatype – type of receiving elements;

int source – the rank of the sending process;

int msgtag – identifier of the receiving message;

MPI_Comm comm – communicator;

MPI_Status *status – additional information about the receiving message (output parameter).

The exit from the MPI_Recv function is completed only after the buffer buf is filled. As the parameters source and msgtag, you can use the special constants MPI_ANY_SOURCE and MPI_ANY_TAG, meaning, respectively, that the message can be received from *any* process or can have *any* message identifier.

The status parameter, the last parameter of the MPI_Recv function, has a structured type MPI_Status, all fields of which are integer. By accessing these fields, we can determine:

- rank of the process that sent the message (the MPI_SOURCE field);
- message identifier (the MPI_TAG field);
- error code associated with this message (the MPI_ERROR field).

In addition, the MPI_Status type contains a special field that allows you to determine the *number of elements* in the message. Instead of directly accessing this field, you should use the function MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count) with input parameters status and datatype and output parameter count, which is the number of received elements of type datatype.

Note. If the program does not need to use the information provided by the status parameter, then the constant MPI_STATUS_IGNORE, which appeared in the MPI-2 standard, can be specified instead. Note that when using the MPICH system, which implements the MPI-1 standard, *this constant should not be used* (despite the fact that it is present in the MPICH library).

Sometimes it is desirable to obtain additional information about the expected message before it is received by the MPI_Recv function (for example, to determine the size of the buffer buf, sufficient to store the received message). For this purpose, the auxiliary function MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status *status) can be used, the parameters of which have the same meaning as the parameters of the MPI_Recv function with the same names. This function, like the MPI_Recv function, is blocking; exit from it is performed only after completion of receiving data from the sending process.

When organizing the sending of a message in buffered mode, it is necessary to use the auxiliary functions MPI_Buffer_attach and MPI_Buffer_detach.

The MPI_Buffer_attach(void *buf, int size) function allows you to define a buffer buf, which is used later when sending messages in buffered mode. The buffer size is specified in bytes and must be sufficient to store both the messages being sent

and the service information. The memory size (in bytes) required to contain the service information is determined by the constant `MPI_BSEND_OVERHEAD`. At any time, a process can use only one buffer, and after it has been defined and until it has been freed, *it should not be accessed by the program itself*.

The `MPI_Buffer_detach` (`void *buf`, `int *size`) function is used to free a previously defined buffer; both of its parameters are output (the `buf` parameter returns the address of the beginning of the freed buffer, and the `size` parameter returns its size in bytes).

Here is a program fragment demonstrating the correct creating, attaching and subsequent detaching the buffer `buf` used in the buffering sending mode. It is assumed that messages containing not more than 10 real numbers will be sent in this mode. Once again, we emphasize that the `buf` buffer *cannot be specified when calling the `MPI_Bsend` function*.

```
int bufsize = 10 * sizeof(double) + MPI_BSEND_OVERHEAD;
char *buf = new char[bufsize];
MPI_Buffer_attach(buf, bufsize);
// ...
// The MPI_Bsend function can be called here to send data.
// The buf buffer must not be used!
MPI_Buffer_detach(buf, &bufsize);
delete[] buf;
```

MPI library provides the functions `MPI_Sendrecv` and `MPI_Sendrecv_replace` for *combined communication requests*, which, when called, both send and receive messages simultaneously (not necessarily for the same pair of communicating processes). Both functions perform simultaneous sending and receiving of messages in standard blocking mode. The difference is that `MPI_Sendrecv_replace` uses a *single* buffer, which initially contains the message being sent, and, after exiting the function, the received message (thus, this buffer is an input and output parameter).

Here is a list of parameters of these functions (in fact, it is a combined list of parameters of the functions for sending and receiving messages). Parameters of the `MPI_Sendrecv` function:

```
void *sbuf – message sending buffer;
int scout – the number of sending elements;
MPI_Datatype stype – type of sending elements;
int dest – the rank of the receiving process;
int stag – identifier of the sending message;
void *rbuf – message receiving buffer (output parameter);
int rcount – the maximum number of elements in the receiving message;
MPI_Datatype rtype – type of receiving elements;
int source – the rank of the sending process;
int rtag – identifier of the receiving message;
MPI_Comm comm – communicator;
```

MPI_Status *status – additional information about the receiving message (output parameter).

It should be emphasized that for the MPI_Sendrecv function, *you cannot use* the same (or even overlapping) sbuf and rbuf buffers.

The number of parameters in the MPI_Sendrecv_replace function is smaller, since in this case, the message sending and receiving buffer is common, and therefore has common characteristics (size and type of elements):

void *buf – common buffer for sending and receiving messages (input and output parameter);

int count – the size of the message sending and receiving buffer (determining the number of elements in the sending message, as well as the maximum number of elements in the receiving message);

MPI_Datatype datatype – type of sending and receiving elements;

int dest – the rank of the receiving process;

int stag – identifier of the sending message;

int source – the rank of the sending process;

int rtag – identifier of the receiving message;

MPI_Comm comm – communicator;

MPI_Status *status – additional information about the receiving message (output parameter).

In some cases, combined requests for interaction make it possible to avoid *mutual deadlocking* (see the next section) that could occur when using separate requests for sending and receiving messages.

All possibilities related to blocking point-to-point communication are studied in the first subgroup of the MPI2Send group (see Section 2.2.1).

1.2.2. Blocking point-to-point communication: examples. Mutual process deadlocks

To get acquainted with the features of the functions used to exchange messages between individual processes, let us consider one of the tasks of the MPI2Send group.

MPI2Send11. A real number is given in each process. Send the given number from the master process to all slave processes and send the given numbers from the slave processes to the master process. Output the received numbers in each process. The numbers received by the master process should be output in ascending order of ranks of sending processes. Use the MPI_Ssend function to send data.

Note. The MPI_Ssend function provides a *synchronous* data transfer mode, in which the operation of sending a message will be completed only after the receiving process starts to receive this message. In the case of data transfer in synchronous mode, there is a danger of *deadlocks* because of the incorrect order of the function calls for sending and receiving data.

Let us create a project template for solving this task and run the resulting program. The taskbook window that appears on the screen (Fig. 13).

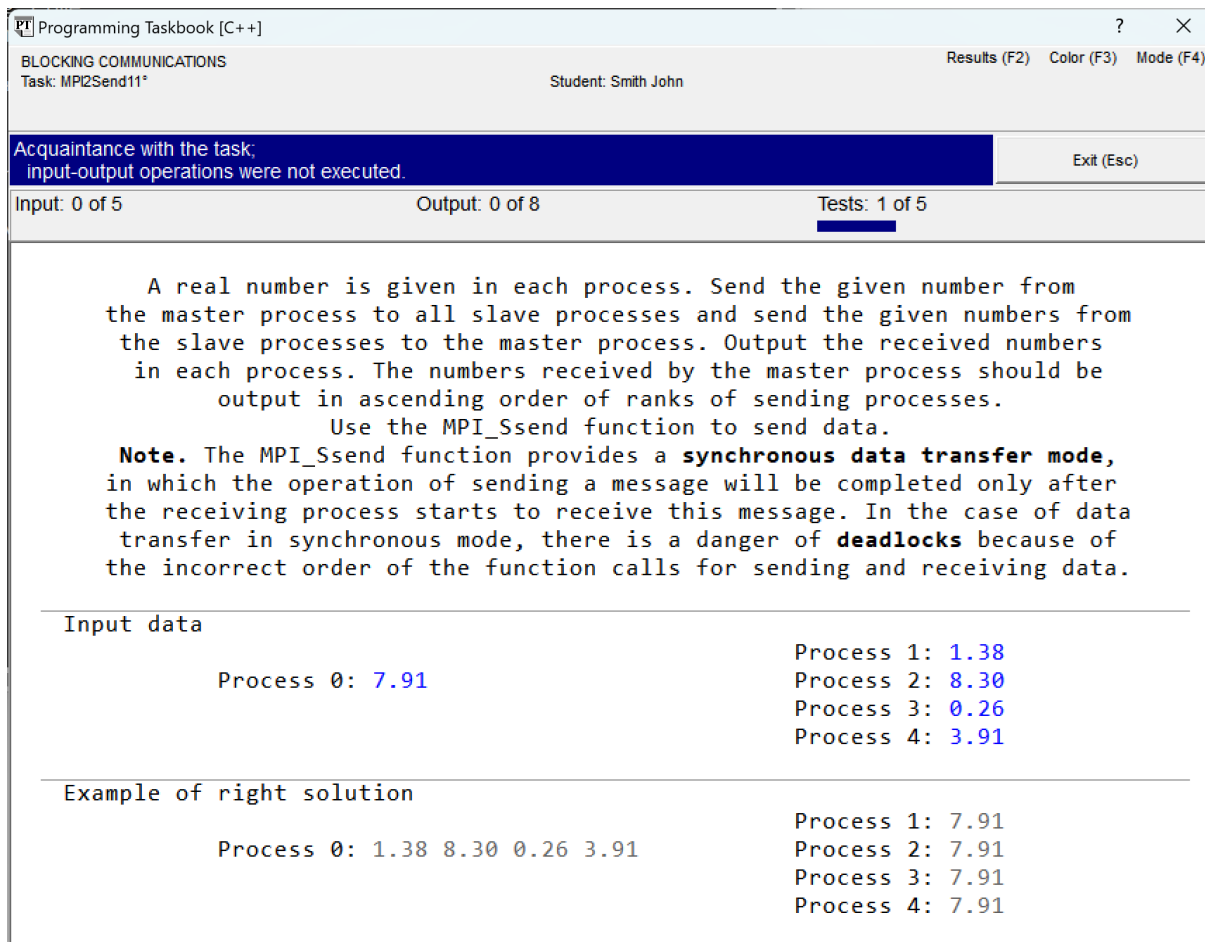


Fig. 13. Acquaintance running of the MPI2Send11 task

To read the initial data, it will be enough for us to use a single variable of real type, since in each process only one real number is given.

The initial data must be sent to other processes of the parallel program. To do this, you need to use a pair of MPI library functions: one for sending the message, the other for receiving it. Since this subgroup of the MPI2Send group studies *blocking* message sending options, you must use the MPI_Recv function for receiving. To send a message in blocking mode, four types of functions are provided (see the previous section). The most commonly used function is MPI_Send, but in our case, we must use the MPI_Ssend function, since this is explicitly stated in the task.

The MPI_Ssend function (like other functions for sending messages, such as MPI_Send) is called by the sending process and specifies which process will receive sending data. The MPI_Recv function is called by the receiving process; it specifies the sending process and the buffer variable into which the received data will be written (see the previous section for a description of the parameters of these functions).

Let us first deal with receiving and sending data for slave processes, without implementing the actions that need to be performed in the master process.

Add the following code fragment to the end of the Solve function:

```
double a;
MPI_Status s;
if (rank > 0)
{
    pt >> a;
    MPI_Ssend(&a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(&a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &s);
    pt << a;
}
```

Note that the first parameter of both functions is the *address* of the variable that contains (or should receive) the sending data.

Let us run our program. The console window appears immediately, and after 20–30 seconds, the taskbook window appears on the screen with an information about an error message in the slave processes (Fig. 14).

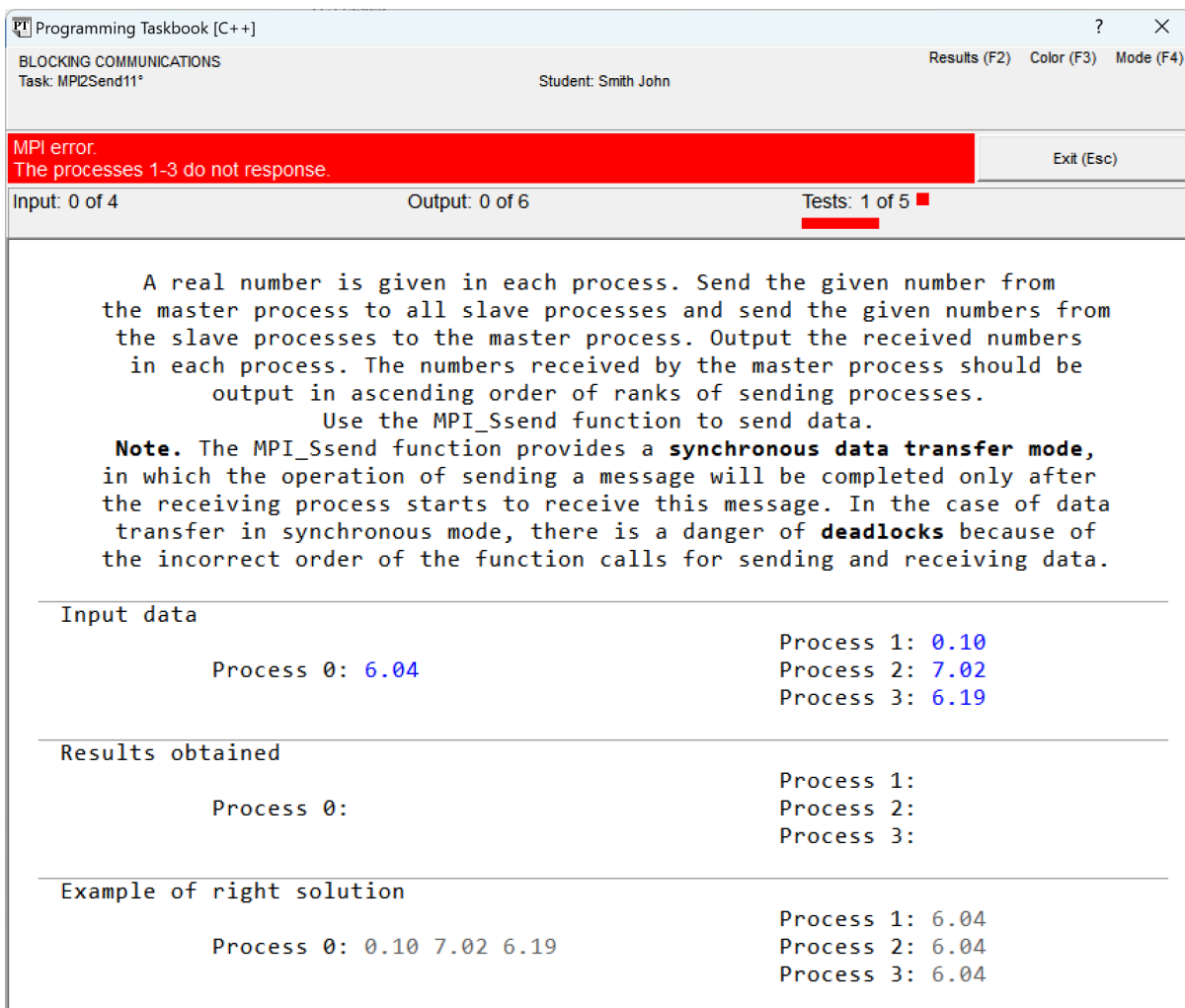


Fig. 14. The taskbook window with information about hung slave processes

Error message of the type "*MPI error. The processes 1–3 do not response*" means that the master process of our parallel program was unable to "contact" the slave processes within a certain time in order to obtain information from them about their input and output data (for information on how to determine and change the response time from slave processes, see Note 3 in Section 1.1.4).

As follows from the second line of the message, the error occurred when trying to contact *all* slave processes (three processes at this program launch).

The reason for the error is that the MPI_Ssend function for sending a message waits until the receiving (in this case, the master) process calls the corresponding receive function (MPI_Recv), and only after that it sends the data and completes its work (*synchronous send mode*). But our program does not yet contain a call to the MPI_Recv function in the master process. Therefore, the wait for the MPI_Ssend function will last forever (more precisely, until the execution of the slave processes will be terminated). This is an example of a parallel program *hang*, which usually occurs because one or more processes are blocked waiting for information that has not been sent to them (in this case, the MPI_Ssend function is waiting for information that the master process has started receiving data).

Note that if we had used another function to send the message, for example MPI_Bsend, which does not wait for information from the receiving process, but simply sends the data to a special send buffer and immediately terminates (*buffered send mode*), then the program would still hang, but for a different reason: now the MPI_Recv function would forever wait for the data that the master process should have sent to it.

When you close the taskbook window, the console window will remain on the screen. The reason is clear: the console window is controlled by the mpiexec program, which terminates only when *all* processes of the running parallel program terminate, but in this case, only the master process terminated (the slave processes remain blocked). To terminate the mpiexec program and close the console window, you need to press the key combination [Ctrl]+[C] or [Ctrl]+[Break] several times (as stated in the comment displayed in the console window).

Note 1. When the mpiexec program terminates, hung processes of the parallel program remain in memory. This will prevent our program from being recompiled in the future, since while the process is in memory, the exe file associated with it is not available for modification. However, when performing tasks using the PT for MPI-2 taskbook, this problem is solved automatically by the taskbook itself (see Note 2 in Section 1.1.4).

So, we have become familiar with the situation when one or more slave processes are blocked. A similar situation can happen to the master process. Let us supply our program with a fragment related to the master process, and in this

fragment we will also organize the call of MPI functions in the same order (first sending data, then receiving it):

```
else
{
    pt >> a;
    for (int i = 1; i < size; i++)
        MPI_Ssend(&a, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
    for (int i = 1; i < size; i++)
    {
        MPI_Recv(&a, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &s);
        pt << a;
    }
}
```

If you run this program, you can wait as long as you like after the console window appears, but the taskbook window will not appear on the screen. This is due to the fact that the master process of our parallel program was blocked: before displaying the taskbook window, the master process must execute the fragment of the program developed by the student, and in our case, this fragment led to the blocking. Therefore, the master process simply did not reach the place in the program where the taskbook window is displayed on the screen.

Why does the blocking occur again? It would seem that every process is now ready to both send and receive a message. However, in order for the MPI_Ssend function to complete, the MPI_Recv function must have already been called in the receiving process, but the receiving process cannot reach this function, since the MPI_Ssend function has also been called in it. This phenomenon is called a *deadlock*.

If the task window does not appear within 20–30 seconds, then it can be assumed that the master process has hung. In this situation, as in the situation described earlier, it is necessary to explicitly interrupt the execution of the parallel program by pressing the keyboard combination [Ctrl]+[C] or [Ctrl]+[Break] several times in the console window.

Note 2. Any of the above-described "emergency" methods of program termination is recorded by the taskbook in the results file. However, if only the slave processes hang (and the taskbook window appears on the screen), the text "*MPI error*" will be written to the results file, whereas in case of a hang of the master process, the text will be different: "*The test run is interrupted*".

The simplest way to fix our program is to delete the second letter "s" in the name of *at least one* MPI_Ssend function, i. e. replace the call to the MPI_Ssend function (either in the slave or in the master process) with a call to the MPI_Send function, which implements the *standard* data transfer mode. This is due to the fact that in the MPI library of the MPICH, MPICH2, and MS-MPI systems, the standard mode, like the buffered mode, uses a buffer to store the data being sent

(and, unlike the buffered mode, the buffer for the standard mode is created *automatically*). After sending the data to the buffer, the `MPI_Send` function terminates, even if by this time the receiving process has not called the `MPI_Recv` function.

Let us describe the sequence of actions in this situation, assuming that we have changed the `MPI_Ssend` function to `MPI_Send` in the *slave* processes. In each of the slave processes, the `MPI_Send` function is called; it copies the data being sent to the buffer, and then immediately terminates; after that, the `MPI_Recv` function is called, which waits for data to be received from the master process. At the same time, the `MPI_Ssend` function is called in a loop in the master process, which suspends execution until the `MPI_Recv` function is called in the slave processes. But the `MPI_Recv` function in the slave processes will definitely be called, at which point the `MPI_Ssend` function in the master process sends the data and terminates. Thus, all `MPI_Ssend` functions in the loop will work successfully, after which the `MPI_Recv` functions will be called in the second loop in the master process, which will receive the data from the slave processes that were previously placed in the buffer. Finally, after the `MPI_Ssend` functions in the master process complete their work, the `MPI_Recv` functions in the slave processes that were waiting to receive data will also be successfully executed. So, no mutual blocking will occur.

When you run the corrected program, it will be successfully tested on five sets of input data, and a taskbook window will appear on the screen with a message that the task has been solved.

However, the correction described above does not fully correspond to the task condition, since the task requires using only the `MPI_Ssend` functions. A variant of the correction with preservation of the `MPI_Ssend` function is in *changing the order of calling* the functions for sending and receiving messages either in the program fragment for the slave processes or in the program fragment for the master process. For example, you can change the order of calling the functions in the master process (the changed code fragment is highlighted in bold):

```
double a;
MPI_Status s;
if (rank > 0)
{
    pt >> a;
    MPI_Ssend(&a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(&a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &s);
    pt << a;
}
else
{
    for (int i = 1; i < size; i++)
    {
        MPI_Recv(&a, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &s);
    }
}
```



```

    pt << a;
}
pt >> a;
for (int i = 1; i < size; i++)
    MPI_Ssend(&a, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
}

```

In this situation, mutual blocking will not occur. Indeed, the MPI_Recv functions are immediately called in the master process, so the corresponding MPI_Ssend functions in the slave process will successfully work and transfer data to the master process. Then, in turn, the MPI_Recv functions will be called in the slave processes, which will allow the MPI_Ssend functions in the master process to work successfully.

Note 3. The described version of the correction has another advantage. The fact is that in the MPI standard it is not guaranteed that the MPI_Send function will *necessarily* use a buffer for intermediate storage of the data being sent. This is determined by the MPI runtime environment itself, so it is possible that the MPI_Send function will use a synchronous mode rather than a buffered mode of sending; in such a situation, a deadlock will still occur.

The resulting program can be simplified if in the else section we use an auxiliary real variable b to receive data from slave processes. This will allow us to place the input statement `pt >> a` *before* the conditional statement, and will also make it possible to perform all the actions in the else section *in a single loop*. Here is a corresponding solution:

```

double a;
MPI_Status s;
pt >> a;
if (rank > 0)
{
    MPI_Ssend(&a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
    MPI_Recv(&a, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &s);
    pt << a;
}
else
    for (int i = 1; i < size; i++)
    {
        double b;
        MPI_Recv(&b, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD, &s);
        pt << b;
        MPI_Ssend(&a, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
    }
}

```

Another small simplification can be achieved by removing the declaration of the variable s of type MPI_Status and replacing the parameter &s in the MPI_Recv functions to the special "stub" constant MPI_STATUS_IGNORE (see the note in Section 1.2.1). The constant MPI_STATUS_IGNORE is convenient to use in a situation

where the program does not need to access the information provided by the parameter of type `MPI_Status`.

Note that a more efficient solution to this task can be obtained by using *collective communications* (see Section 1.2.4).

1.2.3. Non-blocking point-to-point communications. Persistent requests for interaction. Timing functions

This section describes the capabilities of the MPI interface related to non-blocking communications and persistent requests. These capabilities are covered in the second subgroup of the `MPI2Send` group (see Section 2.2.2).

In non-blocking point-to-point communications, the send/receive message operations only *initiate* the corresponding actions, and then immediately terminate returning a special MPI object, *an exchange request* of the `MPI_Request` type. With the help of the exchange request, the state of this non-blocking operation can be checked later, using either the Wait group functions, which block the program execution until the operation is completed, or the non-blocking Test group functions. When a non-blocking operation completes, the associated exchange request is "reset", taking the value `MPI_REQUEST_NULL` (this occurs either upon return from the Wait function, or upon such a call to the Test function, in which information about the completion of the operation is returned).

As for blocking communications, there are four non-blocking message sending functions with identical parameter sets and one non-blocking message receiving function. The names of these functions coincide with the names of the corresponding functions for blocking message sending or receiving, with the prefix "I" ("immediate") added: `MPI_Isend`, `MPI_Ibsend`, `MPI_Issend`, `MPI_Irsend`, `MPI_Irecv`.

The functions `MPI_Isend`, `MPI_Ibsend`, `MPI_Issend`, `MPI_Irsend` initiate a non-blocking message sending operation in one of four possible modes (see Section 1.2.1), returning an exchange request of type `MPI_Request` associated with this operation. The output parameter request is the last parameter of these functions; all preceding parameters coincide with the parameters of the blocking message sending functions: `buf`, `count`, `datatype`, `dest`, `msgtag`, and `comm`. Until the exchange request is reset (i. e., until the non-blocking operation is completed), the buffer `buf` *cannot be reused*.

The `MPI_Irecv` function initiates a non-blocking message receiving operation, returning an associated exchange request of type `MPI_Request`. This parameter is the last one and is located in place of the status parameter of the `MPI_Recv` function; all other parameters (`buf`, `count`, `datatype`, `source`, `msgtag`, and `comm`) are the same for these functions. Before the exchange request is reset, the buffer `buf` *cannot be used to read the received data*.

The Wait blocking group contains four functions:

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

```
MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses),  
MPI_Waitany(int count, MPI_Request *requests, int *index, MPI_Status *status),  
MPI_Waitsome(int count, MPI_Request *requests, int *outcount, int *indices,  
             MPI_Status *statuses).
```

The `MPI_Wait` function waits for the completion of a non-blocking message sending or receiving operation associated with a request (input and output parameter) and returns the output parameter status, which is typically used only for non-blocking receiveings.

All other functions accept an *array* requests of size count.

The `MPI_Waitall` function blocks the execution of a process until *all* communication operations associated with the specified requests are completed (the statuses parameter of size count returns an array of elements of type `MPI_Status` with additional information about each of the completed operations).

The `MPI_Waitany` function blocks execution of the process until *any* exchange operation associated with the specified requests is completed. The index parameter returns the index of the completed operation, and the status parameter returns additional information about this operation (all other exchange requests in the requests array are unchanged).

Finally, the `MPI_Waitsome` function blocks the process until *at least one* of the communication operations associated with the specified requests is completed. Unlike the `MPI_Waitany` function, this function can return information about *multiple* completed operations: the number of completed operations is returned in the outcount parameter, the indices of the completed operations are returned in the first outcount elements of the indices array, and additional information about the completed operations is returned in the first outcount elements of the statuses array.

The Test group also includes four functions:

```
MPI_Test(MPI_Request *request, int *flag, MPI_Status *status),  
MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses),  
MPI_Testany(int count, MPI_Request *requests, int *index, int *flag, MPI_Status *status),  
MPI_Testsome(int count, MPI_Request *requests, int *outcount, int *indices,  
            MPI_Status *statuses).
```

The `MPI_Test` function checks the completion of a non-blocking sending or receiving operation associated with a request and *immediately terminates*, returning the result of the check in the output parameter flag. If the operation is complete, the flag parameter returns a non-zero value (in this case, the exchange request value is reset to `MPI_REQUEST_NULL`, and additional information about the completed operation is returned in the status parameter); otherwise, a zero value is returned in the flag parameter (in this case, the request and status parameters are not changed). The other functions behave the same as the `MPI_Test` function, i. e. they check the completion of non-blocking operations (in this case, associated with an *array* requests) and immediately terminate, returning the result in the flag

parameter (or, for the MPI_Testsome function, in the outcount parameter). The meaning of the remaining parameters of these functions is similar to the meaning of the parameters of the corresponding functions of the Wait group.

There is also a non-blocking version of the MPI_Probe function:

```
MPI_Iprobe(int source, int msgtag, MPI_Comm comm, int *flag, MPI_Status *status)
```

This version differs from the blocking version by the presence of the output parameter flag. The MPI_Iprobe function does not wait for the message receiving operation to complete. If the receiving operation is not completed, then a zero value is returned in the flag parameter (in this case, the status parameter should not be used). This function, like its blocking version MPI_Probe, is usually used in a situation where the number of elements in the sending message is not known in advance.

A special type of non-blocking operations are persistent requests. These requests are formed using the functions MPI_Send_init, MPI_Bsend_init, MPI_Ssend_init, MPI_Rsend_init, MPI_Recv_init, which have the same parameters as the previously considered non-blocking functions MPI_Isend, MPI_Ibsend, MPI_Issend, MPI_Irsend, MPI_Irecv. However, unlike the "usual" non-blocking functions, the functions of the Init group do not immediately execute the corresponding operation; they only return a *persistent request*, the request parameter of the MPI_Request type, which contains all the settings for the required operation.

To start persistent requests generated using the Init group functions, the MPI_Start(MPI_Request *request) and MPI_Startall(int count, MPI_Request *requests) functions are provided. The first of them starts (in non-blocking mode) the operation associated with the request (input and output parameter), and the second starts (also in non-blocking mode) all operations associated with the array requests of size count. In the future, to check the completion of the operations, it is necessary to use the previously discussed functions of the Wait and Test groups.

The request returned by the Init group functions has one important feature: it is *persistent*. This means that after the completion of the corresponding non-blocking operation, the value of the request associated with it is *not reset to MPI_REQUEST_NULL*, but remains valid. Therefore, the persistent request *can be reused* later by calling the starting function MPI_Start or MPI_Startall for it *again* (of course, before this, the contents of the buffer buf containing the data being sent must be changed). To reset the request generated by one of the Init group functions, use the function MPI_Request_free(MPI_Request *request).

Some tasks involving non-blocking operations (see Section 2.2.2) require the use of a special MPI function designed for *measuring time*: double MPI_Wtime(). This function is one of two special MPI functions that return not information about the success of their launch, but the result itself, namely, the time in seconds that has passed since some point in the past. Thus, to determine the duration of execution of some program fragment, it is sufficient to call this function at the beginning and at the end of this fragment, and then find the difference

between the obtained values. The second special MPI function is also related to measuring time: this is the double `MPI_Wtick()` function, which returns the duration in seconds between successive timer ticks and, thus, characterizes the *accuracy* of the time measurement.

1.2.4. Collective communications

A large group of MPI functions is intended for organizing *collective interaction of processes*. "Collective" MPI functions, in contrast to the previously considered functions `MPI_Send`, `MPI_Recv`, etc., allow organizing the exchange of messages not between two separate processes (sender and receiver), but between *all* processes included in a certain communicator. In particular, when using the communicator `MPI_COMM_WORLD`, it is possible to organize collective exchange of messages between all running processes of a parallel program.

The use of collective communications is more preferable than multiple calls of individual point-to-point operations, which is due to two circumstances. First, when implementing collective functions in the MPI library, efficient algorithms are used and, second, in supercomputer or cluster systems, collective exchange operations can be implemented at the hardware level, which can be taken into account when developing MPI libraries for these systems.

In MPI-1 and MPI-2, all operations related to collective interaction of processes are performed in *blocking* standard mode (the MPI-3 standard added *non-blocking* functions for collective operations, these functions are briefly described at the end of this section). For successful execution of a collective operation, it is necessary that the corresponding function be called in *all* processes of the communicator for which this collective operation is performed.

If a process that plays a special role is associated with a collective operation, then the corresponding function has the root parameter containing the rank of this process (and, in addition, some parameters of this function will be used only in a process of rank root). If the root parameter is absent in a collective function, then all processes are equal when executing the collective operation.

The simplest collective function with equal processes is `MPI_Barrier(MPI_Comm comm)`. It blocks the work of the processes that called it until *all* processes of the communicator `comm` also call this function. Thus, the `MPI_Barrier` function allows *synchronization* of processes of a parallel application.

The simplest collective function with a selected root process is `MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`. This function broadcasts data from the root process to all processes of the communicator `comm`. The `buf` parameter specifies the buffer for message broadcast/receive; this parameter is an input parameter in the root process, and it is an output parameter in other processes. The other parameters are input in all processes: the `count` parameter specifies the number of elements to be broadcast, and the `datatype` parameter is their type.

Example:

	buf		buf
Process 0:			(b0 b1 b2 b3)
Process 1:	(b0 b1 b2 b3)	==>	(b0 b1 b2 b3)
Process 2:			(b0 b1 b2 b3)

The MPI_Gather function *collects* data from all communicator processes into the buffer of the receiver process root, with the same number of data elements received from each process. Its parameters are:

void *sbuf – send buffer;

int scout – the number of elements in the sending message;

MPI_Datatype stype – type of elements of the sending message;

void *rbuf – data collection buffer (this is an output parameter that is used only in the root process);

int rcount – the number of elements received from each process (this and the next parameter are also used only in the root process; note that this parameter *is not equal to the size of the rbuf buffer*);

MPI_Datatype rtype – type of elements of the receiving message;

int root – the rank of the process receiving data;

MPI_Comm comm – communicator.

The root process also accepts its own data.

Example (scout = 2, rcount = 2):

	sbuf		rbuf
Process 0:	(a0 a1)		
Process 1:	(b0 b1)	==>	(a0 a1 b0 b1 c0 c1)
Process 2:	(c0 c1)		

A more complicated version of the MPI_Gather function is the MPI_Gatherv function, which allows a different number of data elements to be received from each process. This function has the following set of parameters:

MPI_Gatherv(void *sbuf, int scout, MPI_Datatype stype, void *rbuf, int *rcounts, int *displs, MPI_Datatype rtype, int root, MPI_Comm comm)

In this case, the scout parameters may have *different* values in different processes, and instead of the integer rcount parameter, the rcounts array is used, which specifies the number of elements received from each process. Additional flexibility of this function is provided by the displs parameter, an *array of offsets* (in elements) from the beginning of the data receiving buffer rbuf. Data received from each process is written to the buffer of the receiving process with an offset determined by the corresponding element of the displs array. The displs parameter is taken into account only in the root process; offsets can be positive or negative.

Example (rcounts = {2, 1, 3}, displs = {0, 2, 3}):

	sbuf		rbuf
Process 0:	(a0 a1)		
Process 1:	(b0)	==>	(a0 a1 b0 c0 c1 c2)
Process 2:	(c0 c1 c2)		

The "inverse" operation to the gather operation is the scatter operation, which sends data from the selected root process to all processes of the given communicator. This operation is also implemented as two functions: `MPI_Scatter` and `MPI_Scatterv`. The parameters of the `MPI_Scatter` function are:

`void *sbuf` – broadcast buffer (this and the next two parameters are used only in the root process);
`int scout` – the number of elements sent to each process (note that this parameter *is not equal to the size of the sbuf buffer*);
`MPI_Datatype stype` – type of elements of the sending message;
`void *rbuf` – data receiving buffer (output parameter);
`int rcount` – the number of elements in the data receiving buffer;
`MPI_Datatype rtype` – type of elements of the receiving message;
`int root` – the rank of the process sending data;
`MPI_Comm comm` – communicator.

Example (`scount = 2`, `rcount = 2`):

	sbuf		rbuf
Process 0:			(b0 b1)
Process 1:	(b0 b1 b2 b3 b4 b5)	==>	(b2 b3)
Process 2:			(b4 b5)

The `MPI_Scatterv` function has the following parameters:

`MPI_Scatterv(void *sbuf, int *scounts, int *displs, MPI_Datatype stype, void *rbuf, int rcount, MPI_Datatype rtype, int root, MPI_Comm comm)`

In this case, the array parameters are `scounts` (an array that specifies the number of elements sent to each process) and `displs` (an array of offsets, in elements, from the start of the sending buffer).

Example (`scounts = {2, 1, 3}`, `displs = {0, 2, 3}`):

	sbuf		rbuf
Process 0:			(a0 a1)
Process 1:	(a0 a1 b0 c0 c1 c2)	==>	(b0)
Process 2:			(c0 c1 c2)

The gather operation has a modification in which the collected data is sent to *all* processes. This operation is implemented as the `MPI_Allgather` and `MPI_Allgatherv` functions. These functions do not have the root parameter, since all processes are equal: they provide their part of the data and receive the combined data. Here is a list of the parameters of these functions and an example of their action:

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

Example (`scount = 2`, `rcount = 2`):

	sbuf		rbuf
Process 0:	(a0 a1)		(a0 a1 b0 b1 c0 c1)
Process 1:	(b0 b1)	==>	(a0 a1 b0 b1 c0 c1)
Process 2:	(c0 c1)		(a0 a1 b0 b1 c0 c1)

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

Example (rcounts = {2, 1, 3}, displs = {0, 2, 3}):

	sbuf		rbuf
Process 0:	(a0 a1)		(a0 a1 b0 c0 c1 c2)
Process 1:	(b0)	==>	(a0 a1 b0 c0 c1 c2)
Process 2:	(c0 c1 c2)		(a0 a1 b0 c0 c1 c2)

The most complex collective operation is *the all-to-all operation*, in which each process sends *different data* to all processes of the communicator. The MPI_Alltoall function sends the same amount of data from each process to all processes:

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

Example (scount = 2, rcount = 2):

	sbuf		rbuf
Process 0:	(a0 a1 a2 a3 a4 a5)		(a0 a1 b0 b1 c0 c1)
Process 1:	(b0 b1 b2 b3 b4 b5)	==>	(a2 a3 b2 b3 c2 c3)
Process 2:	(c0 c1 c2 c3 c4 c5)		(a4 a5 b4 b5 c4 c5)

The MPI_Alltoallv function causes each process to broadcast *different* data to all other processes, and each process may receive a *different* number of data elements from different processes. The data sent to each process must be placed in the source buffer at an offset determined by the corresponding element of the sdispls array. The data received from each process is written to the destination buffer at an offset determined by the corresponding element of the rdispls array:

```
MPI_Alltoallv(void *sbuf, int *scounts, int *sdispls, MPI_Datatype stype, void *rbuf,
              int *rcounts, int *rdispls, MPI_Datatype rtype, MPI_Comm comm)
```

Example:

	sbuf		rbuf
Process 0:	(a0 a1 a2 a3 a4)		(a0 a1 b0 c0 c1 c2)
Process 1:	(b0 b1 b2 b3 b4 b5)	==>	(a2 b1 b2 b3 c3)
Process 2:	(c0 c1 c2 c3 c4)		(a3 a4 b4 b5 c4)

The collective functions described above (except for the MPI_Barrier function) are considered in the tasks of the first subgroup of the MPI3Coll group (see Section 2.3.1). In addition, they are actively used in subsequent task groups. In particular, when solving the MPI5Comm3 task (see Section 1.2.7), the MPI_Gather function is used, and when solving the MPI5Comm17 task (see Section 1.2.8), the MPI_Scatter function is used. The MPI_Barrier function is used in the final tasks of the MPI7Win and MPI8Inter groups (Sections 2.7.2 and 2.8.3).

In all collective functions of the MPI-1 standard that contain array of displacements displs, these displacements are specified *in elements of the sending/receiving data*. However, in some situations involving the exchange of compound datatypes, it is desirable to specify displacements *in bytes* rather than in elements. Therefore, in the MPI-2 standard, the set of collective functions was

supplemented by the MPI_Alltoallw function, which performs the same action as the MPI_Alltoallv function, but allows displacements to be specified in bytes. In addition, in this version of the collective function, *each process can send data of different types to different processes*:

```
MPI_Alltoallw(void *sbuf, int *scounts, int *sdispls, MPI_Datatype *stypes, void *rbuf,
             int *rcounts, int *rdispls, MPI_Datatype *rtypes, MPI_Comm comm)
```

The MPI_Alltoallw function can be used to implement variants of the gather and scatter operations, in which offsets are specified in bytes, and data of different types is gathered or scattered. A special subgroup of the MPI4Type group is devoted to this function (see Section 2.4.4). The inclusion of tasks for the MPI_Alltoallw function into the section devoted to derived types is due to the fact that this function is intended, first of all, for collective exchange of compound datatypes. This function is subsequently used in the subgroup of the MPI9Matr group, devoted to the Fox's block algorithm for matrix multiplication (see Section 2.9.5).

The MPI-3 standard added *non-blocking* functions for collective operations. All blocking collective functions discussed above have non-blocking versions. As in the case of non-blocking point-to-point operations (see Section 1.2.3), the names of non-blocking collective functions are derived from the names of the corresponding blocking functions by adding the prefix "l" (MPI_lbarrier, MPI_lbroadcast, MPI_lgather, MPI_lscatter, MPI_lallgather, MPI_lalltoall, etc.), and an additional output parameter MPI_Request *request is added at the end of the parameter list. The request parameter is further used in the Wait and Test group functions (see Section 1.2.3) to check that a given collective operation has completed in a given process.

The PT for MPI-2 taskbook, starting with version 1.6, includes tasks connected with non-blocking collective functions (except for MPI_lbarrier and MPI_lalltoallw functions). They are contained in the last part of the MPI5Comm group (see Section 2.5.4). The inclusion of these tasks into the MPI5Comm group, which is devoted to creation of new communicators, is due to the fact that these tasks require to apply a non-blocking collective operation to *a part* of processes of a parallel application, and for this purpose you need to create a new communicator containing this part of processes. To solve tasks on non-blocking collective functions, it is necessary to use the MS-MPI system, because the MPICH and MPICH2 systems do not support the MPI-3 standard.

1.2.5. Reduction operations and using compound datatypes

MPI functions includes a group of functions that perform *reduction operations*, i. e. operations associated with sending not the original data, but the results of their processing by some *group operation* of the MPI_Op type. The most frequently used operations are finding the sum MPI_SUM, the product MPI_PROD, the maximum MPI_MAX or the minimum MPI_MIN value. The logical operations

MPI LAND, MPI_LOR, MPI_LXOR and their bitwise analogs MPI_BAND, MPI_BOR, MPI_BXOR are also provided. Among the reduction operations, a special place is occupied by the operations MPI_MAXLOC and MPI_MINLOC, which allow finding not only the maximum or minimum element among the elements of each process, but also its number (the rank of the process containing this extremal element is usually used as the number).

The user can define a new reduction operation *op*; the function `MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)` is provided for this purpose. The first parameter *function* is a pointer to the function in which the new operation is defined. The operation being defined must necessarily be *associative*. If it is also *commutative*, then the flag parameter *commute* must be non-zero. The prototype of the function parameter has the following form:

```
typedef void MPI_User_function(void *invec, void *inoutvec, int *len,
                               MPI_Datatype *datatype);
```

The parameters *invec* and *inoutvec* are pointers to arrays containing *len* elements of type *datatype*. The array elements *invec*[*i*] and *inoutvec*[*i*], *i* = 0, ..., *len*−1, are considered, respectively, as the left-hand and right-hand operands of the user-defined operation; the result of applying this operation to the elements *invec*[*i*] and *inoutvec*[*i*] is to be stored in the element *inoutvec*[*i*]. Thus, the array *invec* is the input parameter, and the array *inoutvec* is both the input and output parameter (which explains the choice of their names).

If a user operation is intended to be applied only to data of a fixed type, then when defining it, you can assume that the *invec* and *inoutvec* arrays have the required type and not analyze the *datatype* parameter.

The MPI-1 standard defines four functions that perform reduction operations: `MPI_Reduce`, `MPI_Allreduce`, `MPI_Reduce_scatter`, and `MPI_Scan`.

The `MPI_Reduce` function performs a global operation, returning the results to the specified destination process *root*. The parameters of this function are:

```
void *sbuf – buffer for arguments;
void *rbuf – buffer for the result (output parameter that is used only in the root
           process);
int count – the number of arguments for each process;
MPI_Datatype datatype – type of arguments;
MPI_Op op – operation identifier;
int root – the rank of the process receiving data;
MPI_Comm comm – communicator.
```

Example (count = 3):

	sbuf		rbuf
Process 0:	(a0 a1 a2)		
Process 1:	(b0 b1 b2)	==>	(a0+b0+c0 a1+b1+c1 a2+b2+c2)
Process 2:	(c0 c1 c2)		

The MPI_Allreduce function is a version of the MPI_Reduce function in which the result of the global operation is returned to *all* processes. Therefore, the MPI_Allreduce function does not have the root parameter, and the output parameter rbuf is used in *all* processes of the comm communicator:

```
MPI_Allreduce(void *sbuf, void *rbuf, int count, MPI_Datatype datatype, MPI_Op op,
             MPI_Comm comm)
```

Example (count = 3):

	sbuf		rbuf
Process 0:	(a0 a1 a2)		(a0+b0+c0 a1+b1+c1 a2+b2+c2)
Process 1:	(b0 b1 b2)	==>	(a0+b0+c0 a1+b1+c1 a2+b2+c2)
Process 2:	(c0 c1 c2)		(a0+b0+c0 a1+b1+c1 a2+b2+c2)

Another version of the MPI_Reduce function is the MPI_Reduce_scatter function. It also does not contain the root parameter. Unlike the MPI_Allreduce function, it does not send the full set of results of the global operation to all processes, but distributes the obtained results among the processes, and each process can receive a different number of result elements:

```
MPI_Reduce_scatter(void *sbuf, void *rbuf, int *rcounts, MPI_Datatype datatype,
                  MPI_Op op, MPI_Comm comm)
```

In this case, the third parameter rcounts is *an array* that specifies the number of result elements sent to each process (note that the sum of the values of the rcounts array elements determines the size of the sbuf buffer in each process).

Example (rcounts = { 1, 3, 2 }):

	sbuf		rbuf
Process 0:	(a0 a1 a2 a3 a4 a5)		(a0+b0+c0)
Process 1:	(b0 b1 b2 b3 b4 b5)	==>	(a1+b1+c1 a2+b2+c2 a3+b3+c3)
Process 2:	(c0 c1 c2 c3 c4 c5)		(a4+b4+c4 a5+b5+c5)

Finally, the MPI_Scan function performs a sequence of *partial* global operations: the result of the global operation for processes from zero to *i* inclusive is sent to the *i*-th process of the communicator. This function has the same set of parameters as the MPI_Allreduce function.

Example (count = 3):

	sbuf		rbuf
Process 0:	(a0 a1 a2)		(a0 a1 a2)
Process 1:	(b0 b1 b2)	==>	(a0+b0 a1+b1 a2+b2)
Process 2:	(c0 c1 c2)		(a0+b0+c0 a1+b1+c1 a2+b2+c2)

Note 1. In the MPI-2 standard, two new functions have been added to the set of functions related to reduction operations: MPI_Reduce_scatter_block and MPI_Exscan. The function MPI_Reduce_scatter_block(void *sbuf, void *rbuf, int rcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) is a simplified version of the MPI_Reduce_scatter function, in which each process is sent a *block* of the result data of the *same size* rcount (as opposed to the rcounts array used in the MPI_Reduce_scatter function). The value of rcount determines the size of the result buffer rbuf. The buffer for the arguments sbuf must contain

$K \cdot \text{rcount}$ elements, where K is the number of processes in the communicator `comm`. The `MPI_Reduce_scatter_block` function is used in the `MPI8Inter` group tasks (see Section 2.8.3).

The `MPI_Exscan` function has the same set of parameters as the `MPI_Scan` function and, like `MPI_Scan`, performs a sequence of partial global operations, but these operations are *exclusive* ("exclusive scan"): each process receives the result of a partial global operation performed on the elements of all *previous* processes (i. e., processes with lower ranks). In particular, a process of rank 0 receives no results at all. The `MPI_Exscan` function is more universal than the `MPI_Scan` function ("inclusive scan"), as it allows to model the `MPI_Scan` function without performing additional collective operations: it is enough, after applying the `MPI_Exscan` function, to call the local function `MPI_Reduce_local`, which also appeared in the MPI-2 standard). It should be noted that the inverse action (modeling the `MPI_Exscan` function with the `MPI_Scan` function) is not possible for some reduction operations (in particular, such an action is not possible for the operations of finding the minimum or maximum).

Note 2. The MPI-3 standard introduced *non-blocking* versions of all reduction operations, namely, the functions `MPI_Ireduce`, `MPI_Iallreduce`, `MPI_Ireduce_scatter`, `MPI_Ireduce_scatter_block`, `MPI_Iscan`, and `MPI_Iexscan`. They have the same features as the usual non-blocking collective functions (see their brief description at the end of the previous section). In particular, the output parameter `MPI_Request *request` is added at the end of their parameter list. Tasks for all non-blocking reduction operations, like tasks for usual non-blocking collective operations, are contained in the final part of the `MPI5Comm` group (see Section 2.5.4).

The second subgroup of the `MPI3Coll` group (see Section 2.3.2) is devoted to blocking collective reduction operations. Let us consider one of the tasks included in it. During solving of this task, we will also become acquainted with an example of the use of compound datatypes (structures) in MPI programs.

MPI3Coll23. A sequence of $K + 5$ real numbers is given in each process; K is the number of processes. Using the `MPI_Allreduce` function with the `MPI_MINLOC` operation, find the minimal value among the elements of all given sequences with the same order number and also the rank of process that contains this minimal value. Output received minimal values in the master process and output corresponding ranks in each slave process.

When we run the template program created to solve the `MPI3Coll23` task, we will see a window on the screen similar to the one shown in Fig. 15.

Each process provides an array of numbers of the same size, and the reduction operation is applied individually to the elements of the provided arrays with the same index; the result is an array of the same size.

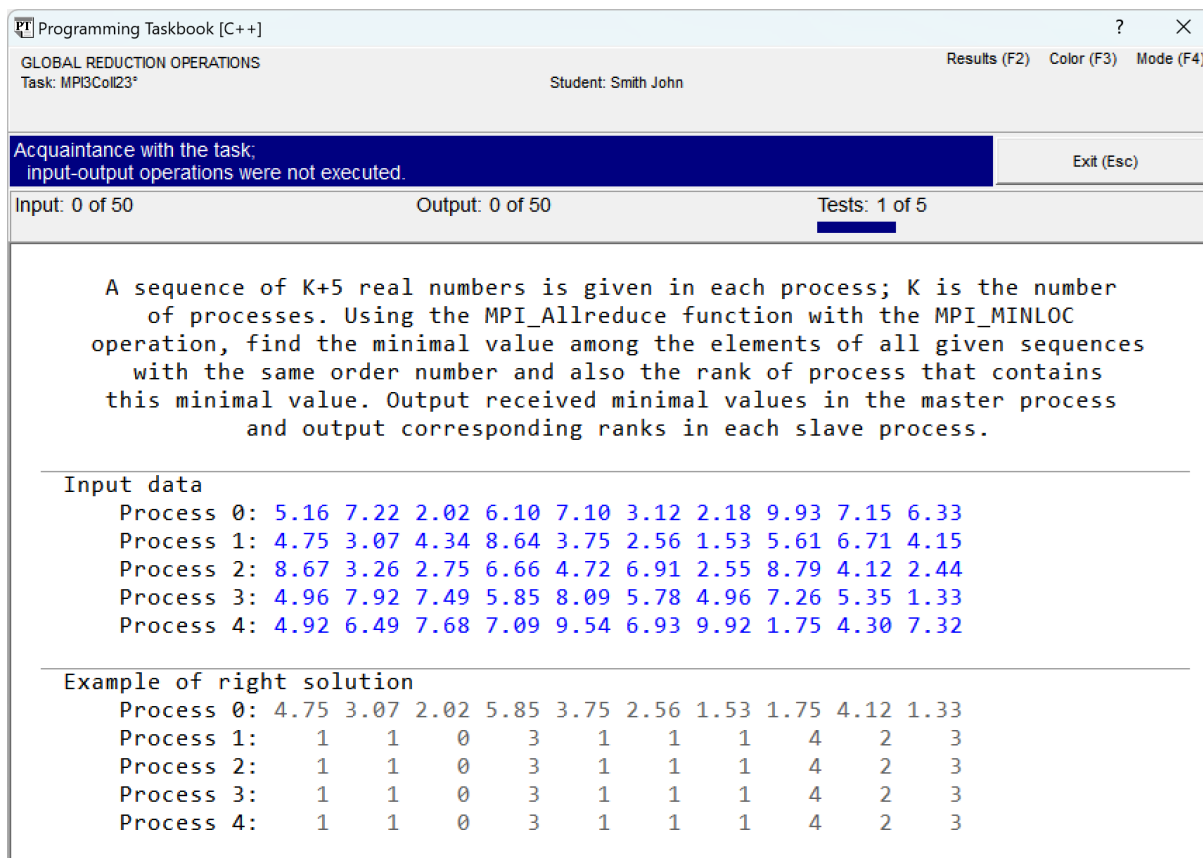


Fig. 15. Acquaintance running of the MPI3Coll23 task

When using the `MPI_MAXLOC` and `MPI_MINLOC` operations, the source data sets must contain *pairs* of numbers: the actual number to be processed and its index. Therefore, the program must define an auxiliary structure for storing such pairs. In our case, real numbers must be processed, so the first element of the pair must be real, and the second must be integer:

```
struct MINLOC_Data
{
    double a;
    int n;
};
```

To store the initial data, each process must allocate an array of elements of the `MINLOC_Data` type, and the same array must be used to store the results of the reduction operation. The size of the data set that will have to be stored in these arrays is not known in advance, since it is related to the number of processes in the parallel program. Therefore, you can either allocate memory for arrays dynamically (after the program obtains the value of the size variable), or use static arrays, the size of which will be sufficient for any sets of initial data. When solving the MPI3Coll23 task, we will use static arrays (the features associated with the use of dynamic arrays, as well as `vector<T>` containers, will be discussed in the next section). Having run the created template program several times, we can

see that the number of processes can vary in the range from 3 to 5. Thus, given that the size of the initial data sets is $K + 5$, where K is the number of processes, it is enough for us to declare arrays of size 10 in the Solve function:

```
MINLOC_Data d[10], res[10];
```

Initialization of the source array d must be performed in each process of the parallel program:

```
for (int i = 0; i < size + 5; i++)
{
    pt >> d[i].a;
    d[i].n = rank;
}
```

After running this version of the program, we will receive a message that all initial data have been successfully input (Fig. 16).

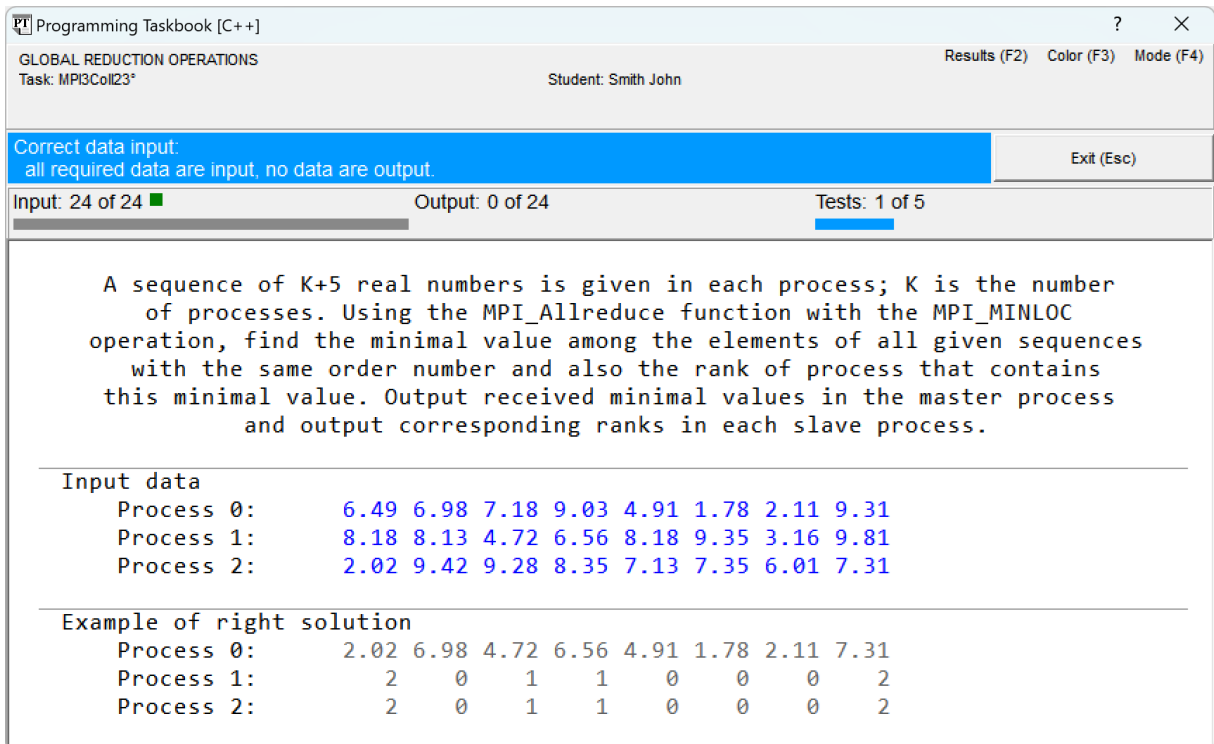


Fig. 16. The taskbook window with information about the successful input of the initial data

Before output the results, it is necessary to perform the corresponding collective reduction operation. It must be performed in all processes. Then in the master process (of rank 0), it is necessary to output the field a of each element of the resulting array res (i. e., the minimum value selected from all elements of the initial arrays with a given index), and in the remaining (slave) processes, it is necessary to output the field n (i. e., the rank of the process with this minimum value):

```
MPI_Allreduce(d, res, size + 5, MPI_DOUBLE_INT, MPI_MINLOC,
             MPI_COMM_WORLD);
for (int i = 0; i < size + 5; i++)
```

```
if (rank == 0)
    pt << res[i].a;
else
    pt << res[i].n;
```

Note two important points. First, the source and result arrays are passed to MPI functions as *pointers to their initial element*, so the first two parameters of the MPI_Allreduce function are simply the array identifiers d and res. Second, the type name specified as the fourth parameter must match the element type of the arrays being processed (in this case, one of the standard MPI types must be specified: MPI_DOUBLE_INT, which corresponds to a structure of two fields, a real field and an integer field). In situations when the standard datatypes provided by the MPI library are insufficient, new MPI datatypes must be defined (this topic is covered in the MPI4Type task group; see the next section).

When you run the resulting program, a message will be displayed stating that the task has been solved. In conclusion, we present the full text of the solution to the MPI3Coll23 task:

```
struct MINLOC_Data
{
    double a;
    int n;
};

void Solve()
{
    Task("MPI3Coll23");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MINLOC_Data d[10], res[10];
    for (int i = 0; i < size + 5; i++)
    {
        pt >> d[i].a;
        d[i].n = rank;
    }
    MPI_Allreduce(d, res, size + 5, MPI_DOUBLE_INT,
        MPI_MINLOC, MPI_COMM_WORLD);
    for (int i = 0; i < size + 5; i++)
        if (rank == 0)
            pt << res[i].a;
        else
            pt << res[i].n;
}
```

1.2.6. Defining derived datatypes and packing data using dynamic arrays and vector containers

MPI library provides a large set of functions for defining new types (derived datatypes). The use of derived datatypes allows to simplify and speed up the actions on sending complex data. Examples of complex data are structures consisting of fields of different types, as well as fragments of multidimensional arrays with "empty" gaps (for example, any column of a two-dimensional matrix). In order to take into account both of these features when defining a new datatype, two sets of characteristics are associated with the new datatype: a sequence of base *types* and a sequence of *displacements*. Thus, a derived datatype can contain elements of different base types and, in addition, these elements may not be located consecutively, but with some displacements relative to each other (the displacements can be both positive and negative). Not only standard MPI types (for example, MPI_INT or MPI_DOUBLE) can be used as base types, but also previously defined derived datatypes.

The simplest of the MPI functions for defining new datatypes is MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype), which creates a derived datatype newtype consisting of count consecutive elements of the base type oldtype. In this and all subsequent functions for defining a new type, the only output parameter is the last parameter, a reference to the derived datatype.

Example (count = 5):

```
Original datatype: [T1]
Derived datatype: [T1][T1][T1][T1][T1]
```

Datatypes created with the MPI_Type_contiguous function are typically used as "building blocks" in defining more complex datatypes.

More useful features are provided by the function MPI_Type_vector(int count, int blocklen, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype), which creates a derived datatype newtype consisting of count *blocks*, each of which contains the same number blocklen of elements of the base type oldtype and is located at the same distance stride from the beginning of the previous block (the distance is specified in the number of elements of the base type).

Example (count = 3, blocklen = 2, stride = 5):

```
Original datatype: [T1]
A memory area equal to the length of the original datatype: [...]
Derived datatype:
[T1][T1][...][...][...][T1][T1][...][...][...][T1][T1]
```

The datatype given in the example can be interpreted as *two adjacent columns* of a 3 by 5 matrix (3 rows, 5 columns), not necessarily the *first* two columns. If, for example, the position of the *third* element in the first row of the matrix is specified as the starting address, then this type will contain elements of the third and fourth columns.

If the blocks in the new type are of different sizes or there should be different distances between them, then the more complex `MPI_Type_indexed` function should be used with array parameters: `MPI_Type_indexed(int count, int *blocklens, int *displs, MPI_Datatype oldtype, MPI_Datatype *newtype)`. This function creates a derived datatype `newtype` consisting of `count` blocks, each of which can contain a different number of elements of the base type `oldtype` and is located at a specified distance from the starting position of the datatype being defined. The number of elements for the different blocks is specified in the `blocklens` array of size `count`, and the distances are measured in elements of the base type and are contained in the `displs` array of size `count`.

Example (`count = 4`, `blocklens = {2, 3, 1, 2}`, `displs = {0, 3, 8, 12}`):

```
Original datatype: [T1]
A memory area equal to the length of the original datatype: [...]
Derived datatype:
[T1][T1][...][T1][T1][T1][...][...][T1][...][...][...][T1][T1]
```

Note that the `MPI_Type_vector` function specifies the distance between the *beginnings of adjacent blocks*, while the `MPI_Type_indexed` function specifies an array of distances from *the starting position* of the derived datatype.

The most flexible of the functions for defining new datatypes is `MPI_Type_struct` with parameters (`int count`, `int *blocklens`, `MPI_Aint *displs`, `MPI_Datatype *oldtypes`, `MPI_Datatype *newtype`). In the MPI-2 standard, the name of this function was changed to `MPI_Type_create_struct`. This function differs from `MPI_Type_indexed` in two ways: first, the array `displs` of offsets from the starting position of the datatype being defined contains offsets *in bytes*, and second, each block has *its own base datatype* (the base datatypes are specified in the `oldtypes` array). The elements of the array of offsets `displs` have the `MPI_Aint` type; this type is intended to store offsets between different *addresses* in memory and is implemented as a signed integer type, the size of which is sufficient to store any possible offset in the address space.

Example:

```
Initial datatypes: [T1], [T2], [T3], [T4]
A section of memory equal to 1 byte (denoted by a dot): .
Derived datatype: [T1][T1].[T2][T2][T2]...[T3].....[T4][T4]
```

Note that the MPI library provides versions of the `MPI_Type_vector` and `MPI_Type_indexed` functions, for which the offsets are also specified in bytes rather than elements (and are of type `MPI_Aint`). In the MPI-1 standard, these functions are named `MPI_Type_hvector` and `MPI_Type_hindexed`, and in the MPI-2 standard, they are named `MPI_Type_create_hvector` and `MPI_Type_create_hindexed`.

In the MPI-2 standard, the set of functions for defining new datatypes was expanded. Without describing all the added functions, we will note one of them, which occupies an intermediate position between `MPI_Type_vector` and `MPI_Type_indexed`. This is the function `MPI_Type_create_indexed_block(int count, int`

blocklen, int *displs, MPI_Datatype oldtype, MPI_Datatype *newtype). It defines a derived datatype newtype consisting of count blocks, each of which consists of blocklen elements of the base type oldtype and is at a specified distance from the starting position of the datatype being defined (the distances are specified in the number of elements of the base datatype and are contained in the displs array of size count). This function differs from MPI_Type_indexed in that all blocks in the datatype it defines have *the same size*, and therefore it is specified not by an array, but by the scalar parameter blocklen of an integer type (as in MPI_Type_vector).

All the functions described are *local*, i. e. they can be called only in some parallel processes, which subsequently use new datatypes (to define other new datatypes or send/receive data).

If a new type is to be used when sending/receiving messages, it must be additionally *registered* by calling the MPI_Type_commit(MPI_Datatype *datatype) function for it. In this case, the datatype parameter is both input and output. Unregistered types can be used when defining new datatypes, but they cannot be used when sending/receiving data.

Note 1. The absence of a call to the MPI_Type_commit function does not prevent the execution of parallel programs on the local computer when using the MPICH system. However, in the case of the MPICH2 and MS-MPI systems, an attempt to specify an unregistered type when sending/receiving data results in an error.

A derived datatype can be destroyed by releasing the descriptor (of type MPI_Datatype) associated with it. This is done by the function MPI_Type_free(MPI_Datatype *datatype), in which the parameter datatype is both input and output. After calling this function, the value MPI_DATATYPE_NULL is assigned to its parameter. It should be emphasized that derived types defined using this datatype are preserved even after its destruction.

The two main characteristics of an MPI datatype are extent and size. *Extent* is the number of bytes that the type occupies in memory (including all empty spaces before and after its blocks). *Size* is the total size (in bytes) of all blocks, *excluding empty spaces*. While extent characterizes the amount of memory allocated to *store* an element of the given datatype, size determines the number of bytes used to *send* an element of the given datatype to other processes (since empty spaces are not included in the generated message). For standard MPI datatypes (MPI_INT, MPI_DOUBLE, MPI_CHAR, etc.), their extent and size are the same.

Two functions are provided in the MPI-1 standard for determining extent and size: MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent) and MPI_Type_size(MPI_Datatype datatype, int *size). The first returns the extent of the type datatype, and the second returns its size.

Sometimes, when defining a new datatype, it is desirable to specify a starting or ending empty space (a "hole") for it. The MPI -1 standard provides spe-

cial base types (*pseudotypes*) for this purpose: MPI_LB and MPI_UB, which are not associated with any actual data and have zero size and extent. They can be used in the MPI_Type_struct function as "markers" for the starting and ending position of the datatype being defined. For example, if in the MPI_Type_struct function, when defining the type1 type, we include three elements in the oldtypes array: MPI_LB, MPI_INT, MPI_UB and define the blocklens and displs arrays as follows: blocklens = {1, 1, 1}, displs = {-3, 0, 6}, then the type1 type will contain one integer element with an *initial* empty space (a hole) of 3 bytes, and the upper limit of the datatype will be located at a distance of 6 bytes from the first byte occupied by the integer element. Thus, the size of the created datatype will be equal to the size of the MPI_INT type (usually 4 bytes), and the extent will be equal to 9 bytes (byte number 6 is not included in the extent, since the MPI_UB type, like MPI_LB, has no extent):

```

MPI_LB  MPI_INT          MPI_UB
|  .  .  [  .  .  .  ]  .  |
-3 -2 -1 0  1  2  3  4  5  6

```

If we now use the MPI_Type_contiguous function with parameters (2, type1, &type2) to define a new type2 datatype, then this new type will contain two integer elements and its length will be 18 bytes:

```

MPI_LB  MPI_INT          MPI_INT          MPI_UB
|  .  .  [  .  .  .  ]  .  .  .  .  [  .  .  .  ]  .  |
-3 -2 -1 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15

```

Note that when merging multiple types with explicitly specified boundary markers, all markers are removed except for the leftmost boundary marker MPI_LB and the rightmost boundary marker MPI_UB.

The way of defining initial and final empty spaces based on the use of markers used in the MPI-1 standard has a drawback: explicitly specified boundaries of the original type *cannot be reduced* when defining a new datatype; they can only be *increased* by specifying new markers MPI_LB and MPI_UB. For this reason, a new, more flexible and convenient method of specifying the initial and final empty spaces when defining a new datatype was proposed in the MPI-2 standard. It uses the function MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype *newtype). In it, to define a new datatype newtype, the base datatype oldtype, the new position of the left boundary lb and the new extent are specified. For example, to define the datatype type1 described above, it is sufficient to use the following call:

```

MPI_Type_create_resized(MPI_INT, -3, 9, &type1);

```

If the original datatype oldtype already had initial and final empty spaces, then the MPI_Type_create_resized function removes them and creates new ones; thus, for the new datatype, initial and final empty spaces can be either increased or decreased. If only the final empty space is required, then the lb parameter should be set equal to 0.

A new function was also added to the MPI-2 standard that allows us to simultaneously determine the left bound `lb` and the extent of a datatype: `MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint *lb, MPI_Aint *extent)`. The previous function `MPI_Type_extent` was declared deprecated in MPI-2.

Note 2. Additional features related to the size and extent of MPI datatypes are also discussed in the note to the MPI4Type5 task (see Section 2.4.1).

MPI interface also provides another way to form messages containing data of different types. This method is based on *packing* data in the sending process, sending data and then *unpacking* it in the receiving process. The advantage of this method is that it does not require defining new datatypes, and the disadvantage is the need to use an additional buffer to store the packed data.

`MPI_Pack` function is used for data packing with the following parameters:

`void *inbuf` – input buffer with initial data;

`int incount` – the number of elements in the input buffer;

`MPI_Datatype datatype` – type of elements in the input buffer;

`void *outbuf` – output buffer with packed data (output parameter);

`int outsize` – output buffer size (in bytes);

`int *position` – current position in the output buffer in bytes (input and output parameter);

`MPI_Comm comm` – the communicator for which data is packed.

The `MPI_Pack` function packs `incount` elements of type `datatype` into the output buffer `outbuf`, starting at the specified position. After this operation, the position parameter is incremented, defining the new current position in the output buffer. The first time the function is called for a given output buffer, the position parameter should be set to 0. After the last call to the function for a given output buffer, the position parameter will be equal to the size of its filled part (in bytes). Care must be taken to ensure that the `outsize` of the output buffer is large enough to hold all the packed data (i. e., that the final value of the position parameter does not exceed the `outsize` value).

Note that when packing (and subsequently unpacking) you must specify the communicator used to send the packed data.

When sending packed data, a special type `MPI_PACKED` must be specified, and the size is specified in bytes.

The `MPI_Unpack` function unpacks the received message on the receiving process side. Its parameters are:

`void *inbuf` – input buffer (with packed data);

`int insize` – input buffer size (in bytes);

`int *position` – current position in the input buffer in bytes (input and output parameter);

`void *outbuf` – buffer with unpacked data (output parameter);

`int outcount` – the number of elements extracted from the input buffer;

`MPI_Datatype datatype` – the type of elements extracted from the input buffer;

MPI_Comm comm – the communicator used to receive the packed message.

Unpacking starts at the specified position of the input buffer. After this operation, the value of the position parameter is incremented, defining the new current position in the input buffer. The first time MPI_Unpack is called for a given input buffer, the position parameter should be set to 0.

There is also a function MPI_Pack_size(int incout, MPI_Datatype datatype, MPI_Comm comm, int *size) that allows you to determine the memory size (in bytes) that *is sufficient* to store incout packed data of type datatype. It should be noted, however, that the returned value size may be *larger* than what is actually required to store the specified number of packed data.

Tasks that allow you to get acquainted with all the capabilities described above are collected in the MPI4Type group (see Section 2.4). The first subgroup of this group examines basic methods for defining new datatypes, the second subgroup is devoted to sending packed data. The third subgroup presents more complex examples of defining new datatypes, associated mainly with parts of two-dimensional arrays (matrices); in these examples, in particular, it is necessary to additionally define types with final empty spaces. Let us consider the first task from the third subgroup of the MPI4Type group.

MPI4Type14. Two sequences of integers are given in the master process: the sequence A of the size $3K$ and the sequence N of the size K , where K is the number of slave processes. The elements of sequences are numbered from 1. Send N_R elements of the sequence A to each slave process R ($R = 1, 2, \dots, K$) starting with the A_R and increasing the ordinal number by 2 ($R, R + 2, R + 4, \dots$). For example, if N_2 is equal to 3, then the process 2 should receive the elements A_2, A_4, A_6 . Output all received data in each slave process. Use one call of the MPI_Send, MPI_Probe, and MPI_Recv functions for sending numbers to each slave process; the MPI_Recv function should return an array that contains only elements that should be output. To do this, define a new datatype that contains a single integer and an additional empty space (*a hole*) of a size that is equal to the size of integer datatype. Use the following data as parameters for the MPI_Send function: the given array A with the appropriate displacement, the amount N_R of sending elements, a new datatype. Use an integer array of the size N_R and the MPI_INT datatype in the MPI_Recv function. To determine the number N_R of received elements, use the MPI_Get_count function in the slave processes.

Note. Use the MPI_Type_create_resized function to define the hole size for a new datatype (this function should be applied to the MPI_INT datatype). In the MPI-1, the zero-size upper-bound marker MPI_UB should be used jointly with the MPI_Type_struct function for this purpose (in MPI-2, the MPI_UB pseudo-datatype is deprecated).

When you run the template program created for this task, a window will appear on the screen with an example of the initial data and an example of the

correct results (Fig. 17). In order to reduce the size of the window, the section with the task formulation is hidden in it (to hide and then restore the section with the formulation, simply press the [Del] key).

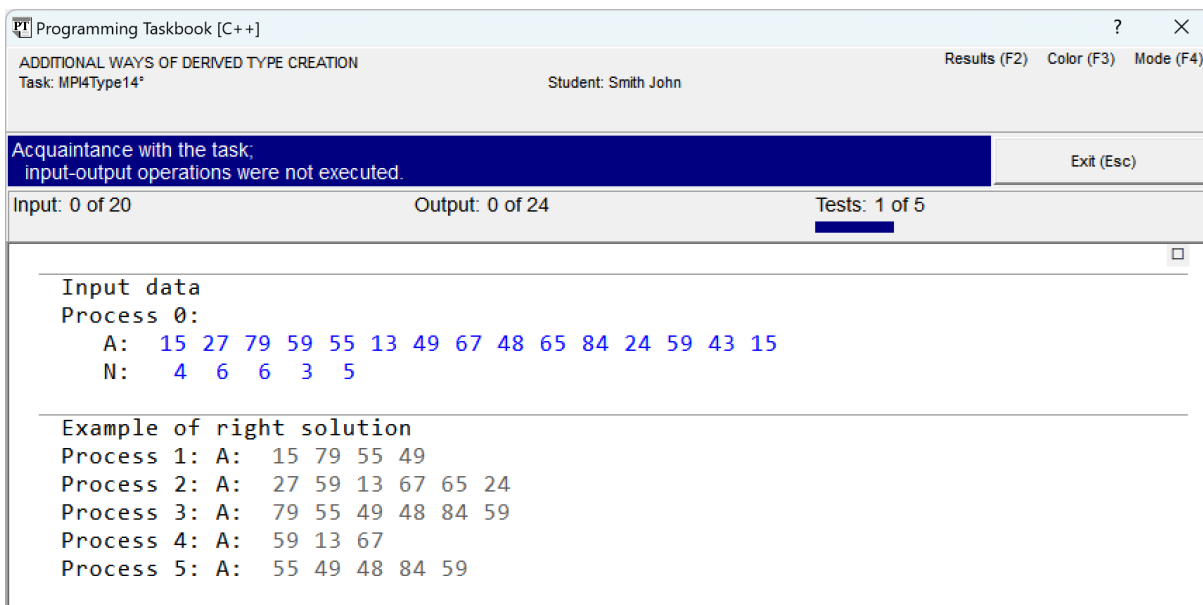


Fig. 17. Acquaintance running of the MPI4Type14 task

In this task, it is necessary to send elements of array *A* from the master process to the slave processes, going through "every other one" of them. If you do not create new datatypes, you will have to either send "extra" data (which will lead to an increase in the size of the messages being sent, as well as the need to allocate additional memory in the receiving processes), or preliminarily, before sending, copy the required elements into an auxiliary buffer (which will require the allocation of additional memory in the sending process, as well as additional actions in this process to copy the necessary data into the auxiliary buffer).

In order to implement the transfer of the required data in an efficient manner both on the sender and on the receiver side, an auxiliary datatype should be defined, and only for the sending process. Using this datatype, we will be able to form a message containing only the necessary elements of the array *A*. For the receiving process, a new datatype is not required, since the message received by this process will not contain "extra" data.

At the first stage of the solution, we will deal with the input of the initial data in the master process. Since the sizes of the initial arrays depend on the number of parallel processes, we will use dynamic memory allocation for them:

```
if (rank == 0)
{
    int k = size - 1;
    int *a = new int[3 * k];
    int *n = new int[k];
```

```

    for (int i = 0; i < 3 * k; i++)
        pt >> a[i];
    for (int i = 0; i < k; i++)
        pt >> n [i];
    // Define a new datatype and send a message
    delete[] a;
    delete[] n;
}

```

After finishing working with the created dynamic arrays, we free the memory allocated for them using the `delete[]` operator.

When you launch a new version of the program, the taskbook window will display the message *"Correct data input: all required data are input, no data are output"*.

Now we will define the new datatype (the corresponding statements should be placed in the position marked with a comment). To illustrate the capabilities of both the MPI-1 and MPI-2 standards, we will describe two versions of such a definition.

In the first version, we will use only the means of the MPI-1 standard:

```

MPI_Datatype t;
int int_sz;
MPI_Type_size(MPI_INT, &int_sz);
int blocklens[] = {1, 1};
MPI_Datatype oldtypes[] = {MPI_INT, MPI_UB};
MPI_Aint displs[] = {0, 2 * int_sz};
MPI_Type_struct(2, blocklens, displs, oldtypes, &t);

```

First, we use the `MPI_Type_size` function to determine the size of an element of the integer type `MPI_INT`. Then, using the `MPI_Type_struct` function, we create a structure of two blocks (each of length 1), the first block containing a single integer and the second block containing an `MPI_UB` element (upper bound marker) that can be used to specify the final empty space for the defined datatype `t`. Recall that the offsets for each block (specified in the `displs` array) are *in bytes* and are counted from the beginning of the first block.

To check the correctness of the created datatype, we will display its characteristics (size and extent) in the debug section:

```

if (rank == 0)
{
    int t_sz;
    MPI_Aint t_ext;
    MPI_Type_size(t, &t_sz);
    MPI_Type_extent(t, &t_ext);
    Show("size = ", t_sz);
    Show("extent = ", (int)t_ext);
}

```

When running this version of the program, the taskbook window will look like the one shown in Fig. 18. We see that the extent of the created type is indeed twice the size of the base type MPI_INT (equal to 4 bytes). The sizes of the new type and the MPI_INT type coincide, since the created type contains a single integer element.

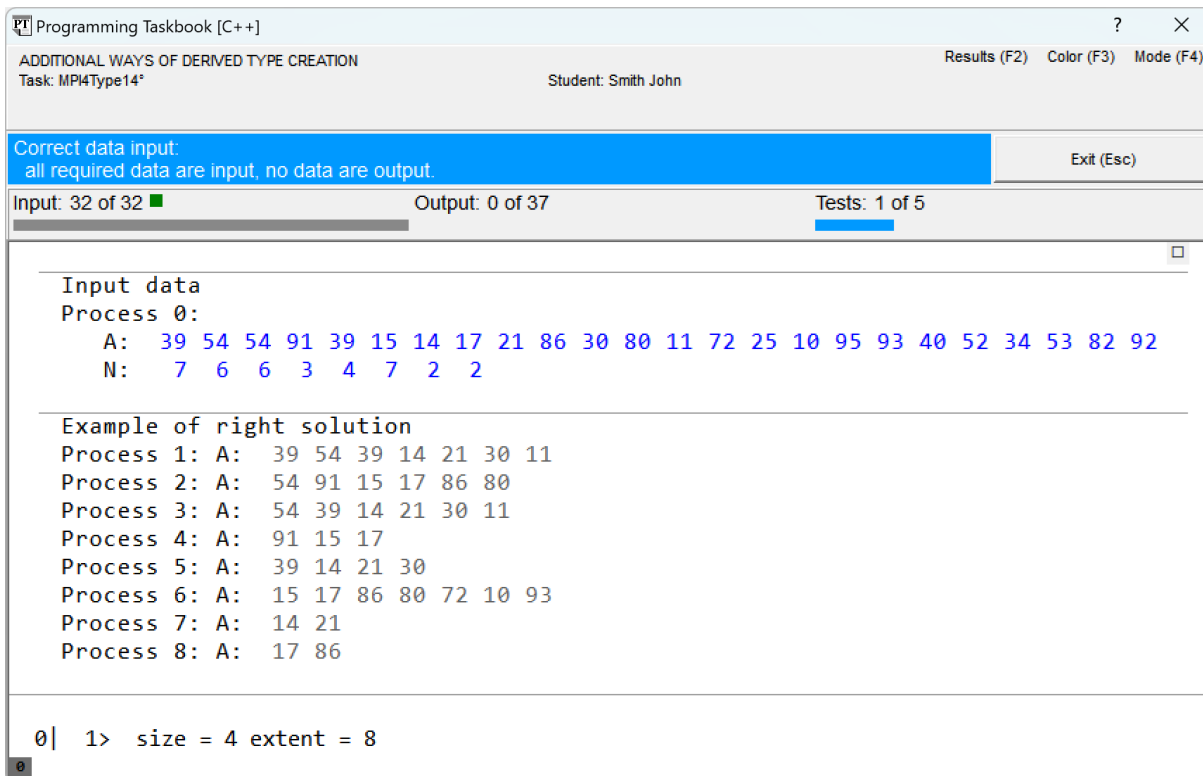


Fig. 18. The taskbook window with information about the created datatype

If we use the tools introduced in the MPI-2 standard, then we do not need auxiliary arrays when defining a new datatype `t`:

```
MPI_Datatype t;
int int_sz;
MPI_Type_size(MPI_INT, &int_sz);
MPI_Type_create_resized(MPI_INT, 0, 2 * int_sz, &t);
```

The characteristics of a type created using the `MPI_Type_create_resized` function will, of course, coincide with the corresponding characteristics of a type created using the MPI-1 standard (see Fig. 18).

To complete the program fragment corresponding to the master process, we only need to use the created datatype to send the required data to the slave processes, having previously registered it using the `MPI_Type_commit` function:

```
MPI_Type_commit(&t);
for (int i = 1; i < size; i++)
    MPI_Send(&a[i - 1], n[i - 1], t, i, 0, MPI_COMM_WORLD);
```

Note 3. If the `MPI_Type_commit` function had not been called in the program, then when using the MPICH2 or MS-MPI system, the following MPI error

message would have been displayed in the taskbook window: *"Error MPI_ERR_TYPE: Datatype has not been committed"*.

It remains to define the fragment corresponding to the slave processes by adding the else branch to the if (rank == 0) statement:

```
else
{
    MPI_Status s;
    MPI_Probe(0, 0, MPI_COMM_WORLD, &s);
    int n;
    MPI_Get_count(&s, MPI_INT, &n);
    int *a = new int[n];
    MPI_Recv(a, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &s);
    for (int i = 0; i < n; i++)
        pt << a[i];
    delete[] a;
}
```

To determine the amount of data to receive, we use the MPI_Probe function, then create a receiving buffer of the required size and filled it in the MPI_Recv function.

Note that when solving this task, we, for the first time, encounter a situation when the type of data being sent (t) does not match the type of data being received (MPI_INT). In addition, it should be emphasized that in the slave processes we did not use the new datatype and at the same time we received exactly the data that needed to be sent from the master process, and in the receiving buffer (unlike the sending buffer) the received data are located without any "holes".

When you run the final version of the program, a message will be displayed stating that the task has been solved. Here is the full text of the resulting solution (without debug output), which uses the capabilities added to the MPI-2 standard:

```
#include "pt4.h"
#include "mpi.h"
void Solve()
{
    Task("MPI4Type14");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0)
    {
        int k = size - 1;
        int *a = new int[3 * k];
```

```

    int *n = new int[k];
    for (int i = 0; i < 3 * k; i++)
        pt >> a[i];
    for (int i = 0; i < k; i++)
        pt >> n[i];
    MPI_Datatype t;
    int int_sz;
    MPI_Type_size(MPI_INT, &int_sz);
    MPI_Type_create_resized(MPI_INT, 0, 2 * int_sz, &t);
    MPI_Type_commit(&t);
    for (int i = 1; i < size; i++)
        MPI_Send(&a[i - 1], n[i - 1], t, i, 0,
                MPI_COMM_WORLD);
    delete[] a;
    delete[] n;
}
else
{
    MPI_Status s;
    MPI_Probe(0, 0, MPI_COMM_WORLD, &s);
    int n;
    MPI_Get_count(&s, MPI_INT, &n);
    int *a = new int[n];
    MPI_Recv(a, n, MPI_INT, 0, 0, MPI_COMM_WORLD, &s);
    for (int i = 0; i < n; i++)
        pt << a[i];
    delete[] a;
}
}

```

Instead of arrays (static or dynamic) in C++ programs, we can use *the "vector" container* `std::vector<T>` from the Standard Template Library STL [2]. This will allow us using additional capabilities for input-output related to the `pt` stream *iterators* (see Section 3.1.2). Here is a solution to the MPI4Type14 task, in which vectors are used instead of arrays (added program fragments are highlighted in bold, and deleted fragments are striked out):

```

#include "pt4.h"
#include "mpi.h"
#include <vector>
#include <algorithm>
void Solve()
{
    Task("MPI4Type14");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;

```

```

MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0)
{
    int k = size - 1;
int *a = new int[3 * k];
int *n = new int[k];
for (int i = 0; i < 3 * k; i++)
    pt->> a[i];
for (int i = 0; i < k; i++)
    pt->> n[i];
    std::vector<int> a(ptin_iterator<int>(3 * k),
                    ptin_iterator<int>()),
                    n(ptin_iterator<int>(1 * k), ptin_iterator<int>());
    MPI_Datatype t;
    int int_sz;
    MPI_Type_size(MPI_INT, &int_sz);
    MPI_Type_create_resized(MPI_INT, 0, 2 * int_sz, &t);
    MPI_Type_commit(&t);
    for (int i = 1; i < size; i++)
        MPI_Send(&a[i - 1], n[i - 1], t, i, 0,
                MPI_COMM_WORLD);
    delete[] a;
    delete[] n;
}
else
{
    MPI_Status s;
    MPI_Probe(0, 0, MPI_COMM_WORLD, &s);
    int n;
    MPI_Get_count(&s, MPI_INT, &n);
int *a = new int[n];
    std::vector<int> a(n);
    MPI_Recv(&a[0], n, MPI_INT, 0, 0, MPI_COMM_WORLD, &s);
    copy(a.begin(), a.end(), ptout_iterator<int>());
for (int i = 0; i < n; i++)
    pt << a[i];
    delete[] a;
}
}

```

To be able to work with vectors, the standard header `<vector>` must be included in the program. In addition, we have included the header `<algorithm>`, which allows the use of STL *algorithms* in the program.

When creating a vector, you can immediately fill it with the initial data by specifying the iterator of the beginning and end of the input stream in the constructor. In our case, we use the input stream `pt`, for which the input iterator `ptin_iterator<T>` is defined in the taskbook, allowing you to organize the reading of

data of type T . Constructor `ptin_iterator<T>(int count)` with one parameter `count` creates an iterator for reading the required number of elements from the `pt` stream, the parameterless constructor `ptin_iterator<T>()` creates an iterator for the end of the input stream. If the vector is intended to store a data set received from another process, then to create it, it is sufficient to use a constructor with one parameter, namely, the required vector size (we used this constructor option in the else branch of the conditional statement). In this case, the vector is filled with zero values of type T .

It is worth paying special attention to the fact that to create the vector `n` we specified *the expression* as the parameter of the first iterator `1 * k` instead of *variable* `k`. This is explained by the fact that the declaration

```
std::vector<int> n(ptin_iterator<int>(k), ptin_iterator<int>());
```

is interpreted by C++ lexical analyzer as a declaration of a *function prototype* `n` with two parameters. In order for this declaration to be interpreted in the way we need (i. e., as a declaration of the vector `n` initialized with two iterators), it is enough to represent the parameter of the first iterator as *an expression*, since in this case the resulting declaration can no longer be interpreted as a function prototype:

```
std::vector<int> n(ptin_iterator<int>(1 * k),  
ptin_iterator<int>());
```

When passing a vector as a buffer for sending or receiving data, it is necessary to specify *the address of the initial element of the buffer* (in particular, in the else branch we had to change the first parameter `a` of the `MPI_Recv` function to `&a[0]`). The required buffer address can also be specified using the data function of the `vector<T>` class: `a.data()`.

To output all elements of a vector, it is sufficient to use the copy algorithm, specifying the begin and end iterators of the beginning and end of the vector as the first two parameters, and the `ptout_iterator` iterator for the output stream `pt` as the last parameter.

If you use the loop by container elements, which appeared in the C++11 standard, you can organize the output of vector elements as follows:

```
for (auto e : a)  
pt << e;
```

Starting with the taskbook version 4.22, to output all elements of a vector, it is enough to pass the vector name to the `pt` stream:

```
pt << a;
```

Using C++ template library (and the related means of the taskbook), we are able to describe the actions for input and output of data sets more briefly. In addition, we did not need to perform special actions related to *freeing memory*, since the memory allocated for vectors is freed in their destructors, which are called automatically.

1.2.7. Creation of new communicators

Often, for efficient implementation of data transfer, it is convenient to use auxiliary communicators, which include not all processes of the parallel application, but only the required part of them (*a group* of processes). Tasks for using auxiliary communicators are collected in four subgroups of the MPI5Comm group (see Section 2.5). It should be noted that the tasks of the MPI5Comm group consider only the so-called *intra-communicators*, associated with one group of processes. In MPI, it is possible to create another type of communicators, namely, *inter-communicators*, which are associated not with one, but with two groups of processes. The MPI8Inter task group (Section 2.8) is devoted to inter-communicator. Most of tasks of this group can be executed only in the MPICH2 and MS-MPI systems that support the MPI-2 standard.

New communicators can be created in three ways.

The simplest way to create a new communicator is to create *a copy* of an existing communicator. The `MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)` function is intended for this purpose, which must be called in all processes of the original communicator `comm`. The new communicator `newcomm` includes the same group of processes and has the same additional characteristics (in particular, some virtual topology, see Section 1.2.8) as the original communicator `comm`. Messages sent using one of these communicators do not affect messages sent using the other in any way; they are sent "over different channels". Copies of the communicator `MPI_COMM_WORLD` are often created in additional parallel libraries and are used to send internal information between processes that is necessary for the normal operation of these libraries. The user of the libraries does not have access to these copies and therefore cannot influence the data transfer performed using them.

Let us emphasize that a usual assignment of the form

```
MPI_Comm newcomm = comm;
```

does not create a copy of the communicator `comm`, it only creates *a copy of the handle* associated with the *same* communicator.

To compare communicators, the function `MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)` is provided. When comparing different descriptors associated with the same communicator, this function returns the value `MPI_IDENT` in the result variable. If different communicators containing the same set of processes are compared, and these processes are ordered in the same way, then the value `MPI_CONGRUENT` is returned (this is the value that will be returned when comparing the original communicator and its copy created using the `MPI_Comm_dup` function). If two communicators contain the same sets of processes, but the order of the processes in them is different, then the value `MPI_SIMILAR` is returned. If communicators contain different sets of processes, then the value `MPI_UNEQUAL` is returned.

The second way to create a new communicator requires a preliminary definition of a new *group* of processes within an existing communicator. Having such a group included in the original communicator `comm`, it is possible to create a new communicator `newcomm` that will contain only processes from the group. The function `MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)` is intended for this purpose. It must be called in all processes included in the communicator `comm`; for those processes that are not included in the specified group, the value `MPI_COMM_NULL` will be returned in the `newcomm` parameter.

Note. In the MPI-2 standard, the capabilities of the `MPI_Comm_create` function were extended so that, when it is called once, it is possible to create *several* new communicators associated with *disjoint* groups of processes from the original communicator. To do this, in the processes of each of these groups, it is sufficient to call the `MPI_Comm_create` function with the group parameter equal to the required group (the `MPI_Comm_create` function must still be called in all processes of the original communicator `comm`). Note that the new capabilities of the `MPI_Comm_create` function make it close to the `MPI_Comm_split` function (see below for a description of the third way to create new communicators).

To work with process groups (objects of type `MPI_Group`), the MPI library provides many different functions, as well as two constants: `MPI_GROUP_EMPTY` (corresponds to an empty group, i. e. a group that does not contain processes) and `MPI_GROUP_NULL` (a value used to indicate an erroneous group).

To create a group of *all* processes of the communicator `comm`, the function `MPI_Comm_group(MPI_Comm comm, MPI_Group *group)` is provided.

For groups, as well as for communicators, there are functions that allow you to determine the *size* of the group, i. e. the number of processes included in it (the function `MPI_Group_size(MPI_Group group, int *size)`), as well as the *rank* of the current (i. e., calling this function) process in the specified group (the function `MPI_Group_rank(MPI_Group group, int *rank)`). If the current process is not in the specified group, then the value `MPI_UNDEFINED` is returned in the `rank` parameter.

There is also the function `MPI_Group_translate_ranks(MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)`, which allows us to determine the ranks of processes in `group2` if their ranks in `group1` are known. In this case, the known ranks of processes in `group1` are specified in the `ranks1` array (of size `n`), and the ranks of the same processes in `group2` are returned in the `ranks2` array of the same size (the output parameter). If some of the processes in the first group is not included in the second group, then the corresponding element of the `ranks2` array is assigned the value `MPI_UNDEFINED`.

Groups, like communicators, can be compared. The function `MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)` returns one of three values in the `result` variable:

MPI_IDENT – two groups contain identical sets of processes, and these sets are ordered identically;

MPI_SIMILAR – two groups contain the same sets of processes, but the order of the processes in them is different;

MPI_UNEQUAL – two groups contain different sets of processes.

Given a group, you can create a new group containing only *a part* of the processes of the original group. For this purpose, the MPI_Group_incl and MPI_Group_excl functions are intended, with the same set of parameters: (MPI_Group group, int n, int *ranks, MPI_Group *newgroup).

When using the MPI_Group_incl function, the new group includes those processes of the original group whose ranks are specified in the array ranks of size n; therefore, the new group will contain n processes. The order of the processes in the new group corresponds to the order of the ranks in the array ranks; thus, a process of the new group of rank i , $i = 0, \dots, n-1$, will coincide with a process of rank ranks[i] of the original group (the array ranks cannot contain identical elements). If the parameter n is 0, then an empty group equal to the constant MPI_GROUP_EMPTY is returned.

When using the MPI_Group_excl function, the new group includes those processes of the original group whose ranks are *not specified* in the ranks array of size n; therefore, the new group will contain n fewer processes than the original group. The order of the processes in the new group corresponds to the order of the processes in the original group; the order of the elements in the ranks array does not matter, it is only required that the ranks array does not contain identical elements. If the parameter n is 0, then a group equal to the original group is returned.

There are versions of the functions MPI_Group_incl and MPI_Group_excl that are convenient to use if the ranks of the included (or, respectively, excluded) processes form regular ranges. These versions have the names MPI_Group_range_incl and MPI_Group_range_excl and the same set of parameters: (MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup). The ranges parameter is an array of size n, and its elements are *triples*, that is, arrays of three integers. Each such a triple defines a *range* of ranks of the form (*first*, *last*, *step*), which includes ranks from the *first* up to and including the *last* with the step *step* (step *cannot* be zero, but can be negative; in this case, *first* must be greater than *last*). "Degenerate" ranges are allowed, consisting of one process of rank R and defined by a triple of the form (R, R, 1). For the MPI_Group_range_incl function, the ranges array defines the ranks of processes from the group group included in the group newgroup (in the specified order), and for the MPI_Group_range_excl function, the ranges array defines the ranks of processes excluded from the group group to obtain the group newgroup. The ranges in the ranges array must be pairwise disjoint.

Given two original groups `group1` and `group2`, one can apply one of the *set operations* to them: union, intersection, difference, resulting in a new group `newgroup`. For this purpose, the functions `MPI_Group_union`, `MPI_Group_intersection`, `MPI_Group_difference` are provided, with the same set of parameters: (`MPI_Group group1`, `MPI_Group group2`, `MPI_Group *newgroup`).

The union consists of all processes of the first group (taken in the same order) supplemented by those processes of the second group (in the same order) that are not in the first group. *The intersection* consists of those processes of the first group (taken in the same order) that are in the second group. *The difference* consists of those processes of the first group (taken in the same order) that are *not* in the second group. The intersection and difference operations may result in an empty group; in this case, the `newgroup` parameter returns the value `MPI_GROUP_EMPTY`. The union and intersection operations are not commutative, since swapping the original groups may change the *order* of the processes in the new group.

Groups and communicators created in the program can be destroyed by freeing the descriptors associated with them. The functions `MPI_Group_free(MPI_Group *group)` and `MPI_Comm_free(MPI_Comm *comm)` are intended for this purpose. As a result of executing these functions, the value `MPI_GROUP_NULL` is returned in the group parameter, and the value `MPI_COMM_NULL` is returned in the comm parameter.

The third way to create a new communicator is associated with the `MPI_Comm_split` function, which splits the original communicator into a set of communicators with pairwise disjoint process groups. We will demonstrate the use of this function using the example of solving one of the tasks included in the first subgroup of the `MPI5Comm` group (see Section 2.5.1). The tasks from the next two subgroups, associated with virtual topologies, are discussed in Sections 1.2.8 and 1.2.9.

MPI5Comm3. Three integers are given in each process whose rank is a multiple of 3 (including the master process). Using the `MPI_Comm_split` function, create a new communicator that contains all processes with ranks that are a multiple of 3. Send all given numbers to the master process using one collective operation with the created communicator. Output received integers in the master process in ascending order of ranks of sending processes (including integers received from the master process).

Note. When calling the `MPI_Comm_split` function in processes that are not required to include in the new communicator, one should specify the constant `MPI_UNDEFINED` as the color parameter.

Here is the taskbook window that was displayed on the screen during the acquaintance running of the program template for this task (Fig. 19).

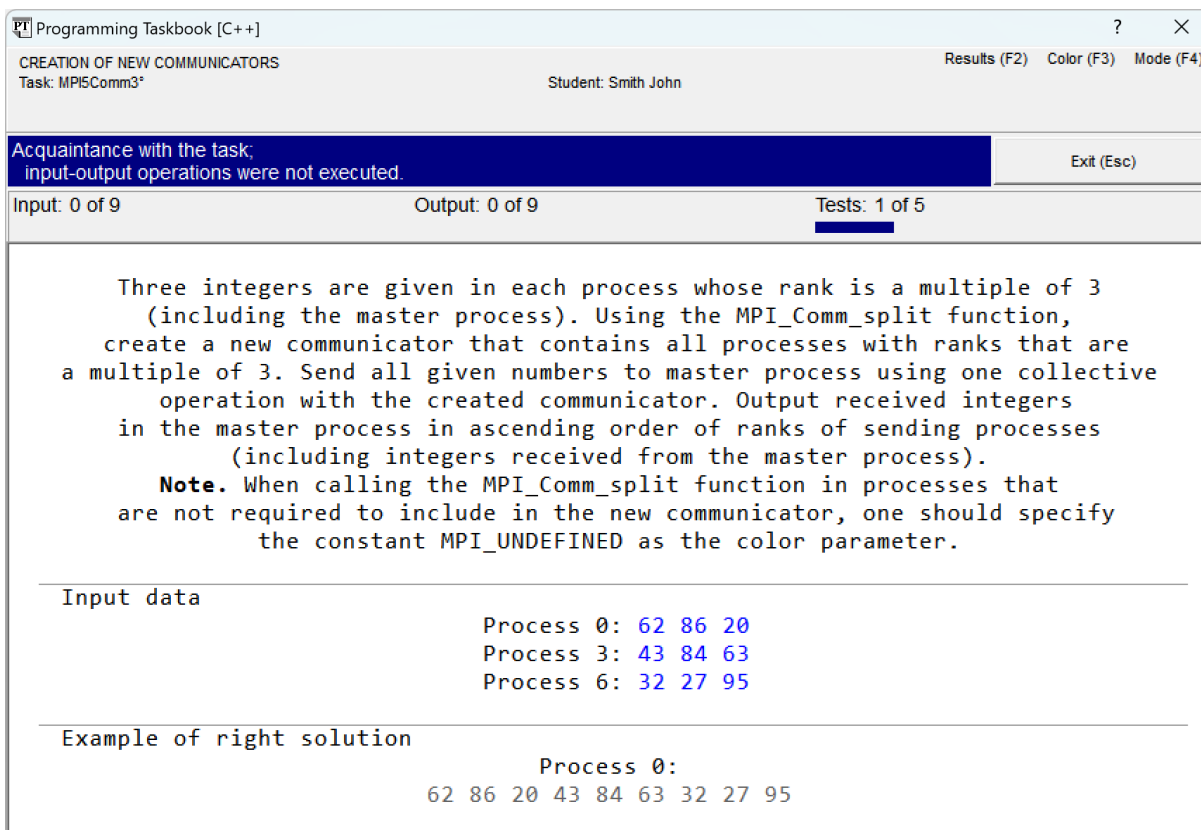


Fig. 19. Acquaintance running of the MPI5Comm3 task

Note that the console window displayed text indicating that eight processes were running in the parallel program:

```
C:\PT4Work>"C:\Program Files\Microsoft MPI\bin\mpiexec.exe"
-n 8 "C:\PT4Work\ptprj.exe"
```

Thus, we need to organize interaction only between *some* of the existing processes. Of course, we can use MPI functions that provide data exchange between two processes (as in the solution to the MPI2Send11 task given in Section 1.2.2), but a more efficient way would be with a suitable collective data transfer operation. However, collective operations are performed for *all* processes included in a certain communicator, so the program must first create a communicator that includes only processes whose rank is divisible by 3. This can be done in various ways; we will use the `MPI_Comm_split` function mentioned in the task formulation.

The function `MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)` splits the set of processes included in the communicator `comm` into separate communicators. This function must be called in all processes included in the communicator `comm`.

As a result of executing this function, each process of the communicator `comm` receives *one* new communicator `newcomm` from the created set, which includes this process. A situation is also possible when some processes will not be

included in any of the created communicators; for such processes, the function `MPI_Comm_split` returns an "empty" communicator `MPI_COMM_NULL`.

The `MPI_Comm_split` function uses the `color` parameter to split processes into new groups. All processes that specify the same `color` parameter when calling `MPI_Comm_split` are included in the same new communicator. Any `color` must be specified as a non-negative number. There is also an "undefined color" `MPI_UNDEFINED`; it must be specified for processes that should not be included in any of the new communicators.

The second characteristic used in the `MPI_Comm_split` function when creating a new set of communicators is the `key` parameter. It determines the order in which the processes will be located in each of the new communicators: the processes in each communicator are ordered by their keys (if some processes have the same keys, their order is determined by the MPI environment that controls the parallel program). To preserve the original order of the processes in each of the newly created communicators, it is sufficient to specify the rank of this process in the original communicator as the `key` parameter for each process.

The ability of the `MPI_Comm_split` function to use the `MPI_UNDEFINED` constant allows new communicators to be created for only *some* of the existing processes. Because of the importance of this feature, it is mentioned in the note for this task.

Taking into account the features of the `MPI_Comm_split` function, we can use it to create a communicator that includes only processes of rank multiple of 3:

```
MPI_Comm comm;
int color = rank % 3 == 0 ? 0: MPI_UNDEFINED;
MPI_Comm_split(MPI_COMM_WORLD, color, rank, &comm);
if (comm == MPI_COMM_NULL)
    return;
```

You can run this version of the program to make sure that you did not make any mistakes when creating a new communicator (the taskbook will still consider the program launch as acquaintance one, since no input or output of data is performed in it).

The last conditional statement ensures immediate exit from the process if the communicator `MPI_COMM_NULL` is associated with it. Of course, in our case, the exit condition could analyze the remainder of dividing rank by 3, but the checking the communicator `MPI_COMM_NULL` is more universal.

In all other processes, it remains to input three integers, send all the input numbers to the master process using the collective function `MPI_Gather`, and output the resulting numbers. To input the given numbers in each process, you can use an array `data` of three elements. The size of the resulting array `res`, which will be obtained in the master process, depends on the number of processes in the parallel application. When discussing the `MPI3Coll23` task, we noted that in such a situation you can use either a static array of a sufficiently large size, or a dynam-

ic array (or a vector `std::vector<T>`), the size of which will be determined after the number of processes becomes known. In Section 1.2.5, when solving the `MPI3Coll23` task, we used a static array. When solving the `MPI4Type14` task, we used dynamic arrays, as well as their alternative from the standard C++ template library, `std::vector<T>` vectors. In this program we will once again use the STL library tools, describing the original and resulting data sets (`data` and `res`) as vectors and using stream iterator `pt` for their input and output:

```
MPI_Comm_size(comm, &size);
std::vector<int> res(3 * size),
    data(ptin_iterator<int>(3), ptin_iterator<int>());
MPI_Gather(&data[0], 3, MPI_INT, &res[0], 3, MPI_INT,
    0, comm);
if (rank == 0)
    copy(res.begin(), res.end(), ptout_iterator<int>());
```

Let us recall that if you use vectors and algorithms from the STL library in your program, you need to include the standard headers `<vector>` and `<algorithm>` to it.

To find the total number of elements received, we first determined the number of processes in the created communicator `comm` (using the `MPI_Comm_size` function), writing this number to the `size` variable. Since each process of the communicator `comm` sends three elements to the master process, the size of the vector `res` is assumed to be equal to `3 * size`.

Then the `MPI_Gather` function is called (see Section 1.2.4). Recall that in the `MPI_Gather` function, the fifth parameter is not the size of the `res` buffer, but the number of elements received from *each* process. Note also that the `MPI_Gather` function receives data from *all* processes of the `comm` communicator, including the root process that is the receiver of all data. When specifying the root process, we took into account that the process of rank 0 in the `MPI_COMM_WORLD` communicator is also the process of rank 0 in the `comm` communicator.

After running the resulting program, we will receive a message that the task has been solved. Here is the full text of the resulting solution:

```
#include "pt4.h"
#include "mpi.h"
#include <vector>
#include <algorithm>
void Solve()
{
    Task("MPI5Comm3");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm comm;
int color = rank % 3 == 0 ? 0 : MPI_UNDEFINED;
MPI_Comm_split(MPI_COMM_WORLD, color, rank, &comm);
if (comm == MPI_COMM_NULL)
    return;
MPI_Comm_size(comm, &size);
std::vector<int> res(3 * size),
    data(ptin_iterator<int>(3), ptin_iterator<int>());
MPI_Gather(&data[0], 3, MPI_INT, &res[0], 3, MPI_INT,
    0, comm);
if (rank == 0)
    copy(res.begin(), res.end(), ptout_iterator<int>());
}

```

1.2.8. Cartesian topology

When executing a parallel program, each process can exchange data with any other process via the standard communicator `MPI_COMM_WORLD`. If it is necessary to use some *part* of the existing processes for organizing interaction between them (for example, for collective data exchange within only this part of the processes), then it is necessary to define a new communicator for the required processes (see Section 1.2.7). However, in a number of situations it is desirable not only to use the required part of the processes (and/or arrange the processes in a different order), but also to establish additional connections between them. For these purposes, the MPI library provides tools that allow you to define *a virtual topology*.

A virtual topology defines a structure on a set of processes that allows these processes to be ordered in a more complex way than in usual communicators (in which processes are ordered linearly). There are two types of virtual topology: Cartesian topology and graph topology. In the case of *Cartesian topology*, all processes are interpreted as nodes of some *n*-dimensional *grid* of size $k_1 \times k_2 \times \dots \times k_n$ (if $n = 2$, then the processes can be considered as elements of *a rectangular matrix* of size $k_1 \times k_2$). In the case of *graph topology*, processes are interpreted as *vertices* of some graph; in this case, connections between processes are defined by specifying a set of *edges* for this graph. In the MPI-2 standard, a special type of graph topology was added, namely, *the distributed graph topology*.

Information about the virtual topology is connected with the communicator. To check the presence of a virtual topology for the `comm` communicator, one can use the function `MPI_Topo_test(MPI_Comm comm, int *status)`, which returns the detected topology type in the output parameter `status`. The `status` parameter can take the following values:

- `MPI_CART` – the Cartesian topology is associated with the communicator;
- `MPI_GRAPH` – the graph topology is associated with the communicator;

MPI_DIST_GRAPH – the distributed graph topology is associated with the communicator (this constant appeared in the MPI-2 standard);

MPI_UNDEFINED – no virtual topology is associated with the communicator.

In this section, we will consider the functions of the MPI library related to the Cartesian topology. They can be divided into four groups:

- creation of Cartesian topology for some communicator (the MPI_Cart_create function, as well as the helper function MPI_Dims_create);
- characterization of the existing Cartesian topology (the MPI_Cartdim_get, MPI_Cart_get, MPI_Cart_rank, MPI_Cart_coords functions);
- splitting the original Cartesian grid into subgrids of lower dimension (the MPI_Cart_sub function);
- finding the ranks of senders and receivers when *shifting data* along some coordinate of the Cartesian grid (the MPI_Cart_shift function).

Some of these functions (MPI_Cart_create, MPI_Cart_coords, MPI_Cart_rank, MPI_Cart_sub) will be considered when discussing the MPI5Comm17 task, and the rest functions will be described at the end of the section, after completing the discussion of the task.

MPI5Comm17. The number of processes K is a multiple of 3: $K = 3N$, $N > 1$. A sequence of N integers is given in the processes 0, N , and $2N$. Define a Cartesian topology for all processes as a $(3 \times N)$ grid. Using the MPI_Cart_sub function, split this grid into three one-dimensional subgrids (namely, *rows*) such that the processes 0, N , and $2N$ were the master processes in these rows. Send one given integer from the master process of each row to each process of the same row using one collective operation. Output the received integer in each process (including the processes 0, N , and $2N$).

When you run the program template for this task, the taskbook window will look similar to that shown in Fig. 20.

This example corresponds to case $N = 4$: there are 12 processes that should be interpreted as elements of a 3×4 matrix. In this case, in the processes that are the initial elements of the rows (in other words, in the processes included in the first column of the matrix), four numbers are given, each of which must be sent to the corresponding process of *the same row* of the matrix.

The first stage in solving the task is to determine the required Cartesian topology. For this purpose, the MPI_Cart_create function is provided, which has the following parameters:

MPI_Comm oldcomm – the original communicator for whose processes the Cartesian topology is defined (in our case, MPI_COMM_WORLD);

int ndims – the number of dimensions of the created Cartesian grid (in our case, 2);

int *dims – an integer array, each element of which defines the size of the corresponding dimension (in our case, the array must consist of two elements with values 3 and $\text{size} / 3$);

int *periods – an integer array of flags that determine *the periodicity* of each dimension (in our case, it is sufficient to use an array of two zero elements);

int reorder – an integer flag that determines whether the MPI environment can automatically change the order of processes (in our case, we need to set this parameter to 0);

MPI_Comm *cartcomm – the resulting communicator with Cartesian topology (output parameter).

Acquaintance with the task;
input-output operations were not executed. Exit (Esc)

Input: 0 of 12 Output: 0 of 12 Tests: 1 of 5

The number of processes K is a multiple of 3: $K = 3N$, $N > 1$. A sequence of N integers is given in the processes 0, N , and $2N$. Define a Cartesian topology for all processes as a $(3 \times N)$ grid. Using the `MPI_Cart_sub` function, split this grid into three one-dimensional subgrids (namely, **rows**) such that the processes 0, N , and $2N$ were the master processes in these rows. Send one given integer from the master process of each row to each process of the same row using one collective operation. Output the received integer in each process (including the processes 0, N , and $2N$).

Input data

Process 0:	80	42	43	71
Process 4:	49	88	10	39
Process 8:	82	66	78	78

Example of right solution

Process 0:	80	Process 1:	42	Process 2:	43	Process 3:	71
Process 4:	49	Process 5:	88	Process 6:	10	Process 7:	39
Process 8:	82	Process 9:	66	Process 10:	78	Process 11:	78

Fig. 20. Acquaintance running of the MPI5Comm17 task

It is convenient to use periodicity for some dimension of the Cartesian grid, for example, when performing *cyclic* data transfer between processes along this dimension (see the description of the `MPI_Cart_shift` function at the end of this section); in this case, the corresponding element in the periods flag array must be set to something other than 0.

Automatic process reordering when creation of the Cartesian topology allows taking into account the physical configuration of the computer system on which the parallel program is executed, and thereby increasing the efficiency of its execution. However, in learning programs executed under the control of the PT for MPI-2 taskbook, the order of processes in the generated Cartesian topologies must remain unchanged, so process reordering should be disabled.

Here is a program fragment that defines the Cartesian topology and connects it to the new communicator `comm` (this fragment should be placed at the end of the `Solve` function):

```
MPI_Comm comm;
int dims[] = {3, size / 3},
periods[] = {0, 0};
MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
```

The communicator `comm` created as a result of executing the `MPI_Cart_create` function contains the same processes (and in the same order) as the original communicator `MPI_COMM_WORLD`. However, these communicators are *different*: the data transfer operations performed using the communicators `comm` and `MPI_COMM_WORLD` are performed independently and do not affect each other. In addition, the `comm` communicator is associated with a virtual topology, while the `MPI_COMM_WORLD` communicator does not have any virtual topology.

Due to the presence of Cartesian topology, each process of the communicator `comm` is associated not only with an ordinal number (*the rank* of the process), but also with a set of integers defining *the coordinates* of this process in the corresponding Cartesian grid. The coordinates, like the rank, are numbered from 0.

The coordinates of a process in a Cartesian topology can be determined by its rank using the `MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)` function, and the `MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)` function allows you to solve the inverse problem. Note that in the `MPI_Cart_coords` function, you must specify an additional parameter `maxdims`, the size of the output array `coords`.

To solve our task, we do not need to use the `MPI_Cart_coords` function, but in some cases (in particular, when debugging parallel programs), it may be useful. Therefore, we will give an example of its use, displaying the coordinates of all processes included in the Cartesian grid in the debug section of the taskbook window. To do this, we will supply the `Solve` function with the following statements:

```
int coords[2];
MPI_Cart_coords(comm, rank, 2, coords);
Show(coords[0]);
Show(coords[1]);
```

When you launch the modified program, the taskbook window will look like that shown in Fig. 21.

Recall that the first number in each line of the debug section (before the `"|"` symbol) denotes the rank of the process that output the data specified in that line. The second number (followed by the `">"` symbol) denotes the order number of the output line for that process. In our case, each process outputs one line containing two numbers: its coordinates in the Cartesian topology.

Programming Taskbook [C++]

VIRTUAL TOPOLOGIES Student: Smith John Results (F2) Color (F3) Mode (F4)

Task: MPI5Comm17*

Acquaintance with the task; input-output operations were not executed. Exit (Esc)

Input: 0 of 12 Output: 0 of 12 Tests: 1 of 5

The number of processes K is a multiple of 3: $K = 3N$, $N > 1$. A sequence of N integers is given in the processes 0, N , and $2N$. Define a Cartesian topology for all processes as a $(3 \times N)$ grid. Using the `MPI_Cart_sub` function, split this grid into three one-dimensional subgrids (namely, **rows**) such that the processes 0, N , and $2N$ were the master processes in these rows. Send one given integer from the master process of each row to each process of the same row using one collective operation. Output the received integer in each process (including the processes 0, N , and $2N$).

Input data

```

Process 0: 59 77 72 57
Process 4: 41 55 81 64
Process 8: 65 50 46 51

```

Example of right solution

```

Process 0: 59 Process 1: 77 Process 2: 72 Process 3: 57
Process 4: 41 Process 5: 55 Process 6: 81 Process 7: 64
Process 8: 65 Process 9: 50 Process 10: 46 Process 11: 51

```

```

0| 1> 0 0
1| 1> 0 1
2| 1> 0 2
3| 1> 0 3
4| 1> 1 0
5| 1> 1 1
6| 1> 1 2
7| 1> 1 3
8| 1> 2 0
9| 1> 2 1
10| 1> 2 2
11| 1> 2 3

```

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

Fig. 21. Output of Cartesian coordinates of processes in the debug section

We see that the process of rank 0 has coordinates $(0, 0)$, i. e. it is the first element of the first row of the matrix, and the process of rank 11 has coordinates $(2, 3)$, i. e. it is the last (the fourth) element of the last (the third) row. In addition, in this case, the first row of the matrix includes processes of ranks 0, 1, 2, 3, and the first column includes processes of ranks 0, 4, and 8.

Let us return to our task. To solve it, we must first split the resulting matrix of processes into separate rows, associating a new communicator with each row. After that, we must perform the collective operation `MPI_Scatter` (see Section 1.2.4) for all processes included in one row for sending fragments of the data set from one process to all processes included in the communicator.

Splitting a Cartesian grid into a set of subgrids of lower dimension (in particular, splitting a matrix into a set of rows or columns) and associating a new

communicator with each resulting subgrid is performed using the function `MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)`.

Its first parameter `comm` should be the original communicator with Cartesian topology, and the second parameter `remain_dims` should be an array of flags: if the corresponding dimension should remain in each subgrid, then a *non-zero* flag is indicated in its place in the array, and if the original grid is split along this dimension (and, consequently, this dimension "disappears" in the resulting subgrids), then the value of the flag associated with this dimension must be zero.

`MPI_Cart_sub` function must be called in all processes of the original communicator `comm`. As a result of its call, *a set* of new communicators is created, each of which is connected to one of the obtained subgrids (all created communicators are automatically supplied with a Cartesian topology). However, this function returns (as the third, output parameter `newcomm`) only *one of the created communicators*, namely, the communicator that includes the process that called this function. Note that the `MPI_Comm_split` function, considered in the previous section, behaves in a similar way.

To split the original process matrix into a set of rows, you need to specify an array of two integer elements as the second parameter of the `MPI_Cart_sub` function; the first element must be equal to 0, and the second element must be non-zero (for example, equal to 1). In this case, all matrix elements with the same value of the *first* (deleted) coordinate will be combined in a new communicator (let us name it `comm_sub`).

The first process of each row (the one that, according to the task conditions, must send its data to all other processes of the same row) has a rank of 0 in the `comm_sub` communicator. To determine the rank, use the `MPI_Comm_rank` function. After that, if the rank is 0, you need to read the original data and send one data element to each process of the same communicator using the `MPI_Scatter` function. At the end, it remains to output the element received by each process.

Here is the final part of the solution:

```
MPI_Comm comm_sub;
int remain_dims[] = {0, 1};
MPI_Cart_sub(comm, remain_dims, &comm_sub);
MPI_Comm_size(comm_sub, &size);
MPI_Comm_rank(comm_sub, &rank);
int b, *a = new int[size];
if (rank == 0)
    for (int i = 0; i < size; i++)
        pt >> a[i];
MPI_Scatter(a, 1, MPI_INT, &b, 1, MPI_INT, 0, comm_sub);
pt << b;
delete[] a;
```

Having launched the new version of the program, we will receive a message that the task has been solved. There is no need to remove the fragment that

provides debug output of process coordinates, since the output of debug data does not affect the verification of the correctness of the solution.

Here is the final text of the solution (without debug output of coordinates):

```
void Solve()
{
    Task("MPI5Comm17");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm comm;
    int dims[] = {3, size / 3},
        periods[] = {0, 0};
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
    MPI_Comm comm_sub;
    int remain_dims[] = {0, 1};
    MPI_Cart_sub(comm, remain_dims, &comm_sub);
    MPI_Comm_size(comm_sub, &size);
    MPI_Comm_rank(comm_sub, &rank);
    int b, *a = new int[size];
    if (rank == 0)
        for (int i = 0; i < size; i++)
            pt >> a[i];
    MPI_Scatter(a, 1, MPI_INT, &b, 1, MPI_INT, 0, comm_sub);
    pt << b;
    delete[] a;
}
```

Note. A common error associated with the use of the `MPI_Cart_sub` function is the incorrect specification of its second parameter, the `remain_dims` flag array. If, for example, we swap the elements with values 0 and 1 in the `remain_dims` array in the given program, then when the program is run, the taskbook window will display error messages similar to those shown in Fig. 22.

Let us analyze these messages. Due to an incorrect flag array specification, the `MPI_Cart_sub` function split the original matrix into *columns* instead of rows; as a result, 4 new communicators are created, each of which contains 3 processes included in the same column of the matrix. In this case, the process that is the first in the column is considered to be a process of rank 0 for the corresponding communicator. Therefore, the condition in the last if statement of our program will be true for processes 0, 1, 2, and 3, and it is for them that the input operators of the given data will be executed.

Programming Taskbook [C++]

VIRTUAL TOPOLOGIES Student: Smith John Results (F2) Color (F3) Mode (F4)

Task: MPI5Comm17

Some required data are not input.
The error has occurred in the processes 4, 8.

Input: 3 of 12 Output: 9 of 12 Tests: 1 of 5

The number of processes K is a multiple of 3: $K = 3N$, $N > 1$. A sequence of N integers is given in the processes 0 , N , and $2N$. Define a Cartesian topology for all processes as a $(3 \times N)$ grid. Using the `MPI_Cart_sub` function, split this grid into three one-dimensional subgrids (namely, **rows**) such that the processes 0 , N , and $2N$ were the master processes in these rows. Send one given integer from the master process of each row to each process of the same row using one collective operation. Output the received integer in each process (including the processes 0 , N , and $2N$).

Input data

Process 0:	42	83	63	40
Process 4:	36	18	88	31
Process 8:	37	74	96	46

Results obtained

Process 0:	42	Process 1:		Process 2:		Process 3:	
Process 4:	83	Process 5:	0	Process 6:	0	Process 7:	0
Process 8:	63	Process 9:	0	Process 10:	0	Process 11:	0

Example of right solution

Process 0:	42	Process 1:	83	Process 2:	63	Process 3:	40
Process 4:	36	Process 5:	18	Process 6:	88	Process 7:	31
Process 8:	37	Process 9:	74	Process 10:	96	Process 11:	46

```

0| 1> !If there are errors in the slave processes,
0| 2> ! then the solution for the master process is not checked.
1| 1> !An attempt to input superfluous data.
2| 1> !An attempt to input superfluous data.
3| 1> !An attempt to input superfluous data.
4| 1> !Some required data are not input.
4| 2> ! The program has input 0 data item(s) (of 4).
8| 1> !Some required data are not input.
8| 2> ! The program has input 0 data item(s) (of 4).

```

Fig. 22. Taskbook window when the MPI5Comm17 task is executed incorrectly

However, in processes 1, 2, and 3, the initial data are not provided, therefore, when executing the program, the error message *"An attempt to input superfluous data"* is displayed for these processes. On the other hand, processes 4 and 8 (which are the initial processes in the second and third rows of the matrix) have a non-zero rank in the new communicators, and therefore no data input is performed for them, which is noted in the error message for these processes: *"Some required data are not input. The program has input 0 data item(s) (of 4)"*. Note also that process 0 sent its initial data not to the processes in the first row of the matrix (as required by the task), but to the processes in the first *column*. Since processes 1, 2, and 3 did not have any initial data, zeros were sent to the other processes in

the corresponding columns. Note that the received zeros were not output in processes 1, 2, and 3, since the taskbook had previously detected an input error in each of these processes and therefore blocked all subsequent input-output operations for these processes.

Thus, the information provided in the taskbook window is sufficient to identify the cause of the error and make the necessary corrections to the program.

Having completed the discussion of the MPI5Comm17 task, we now describe those functions associated with the Cartesian topology that were not required in its solution.

When defining a Cartesian topology using the MPI_Cart_create function, two main characteristics must be specified: the Cartesian topology *size* (the number of dimensions, ndims) and the *number of nodes* (i. e. processes) in each dimension, an array of integers dims of size ndims. The MPI library provides an auxiliary function MPI_Dims_create(int nnode, int ndims, int *dims), which allows us to determine the optimal number of nodes in each dimension of the Cartesian grid if the total number of nodes nnodes and the number of dimensions ndims are known. The found number of nodes is returned in the dims array.

The elements of the dims array that need to find must have *zero* initial values; the initial *positive* values of the elements of the dims array are considered fixed and *do not change*. The values of the elements of the dims array determined by the function are always *sorted in non-ascending order* and are chosen *as close to each other as possible* (for example, from the options {6, 1} and {3, 2}, the option {3, 2} will be chosen). In the case of negative initial values or the impossibility of choosing at least one option of the required Cartesian grid, an error occurs (recall that in this case, the function returns a value different from MPI_SUCCESS).

Here are some examples (the initial values of the parameters are indicated to the left of the ==> arrow, and the resulting contents of the dims array are indicated to the right):

```
nnodes = 6, ndims = 2, dims = {0, 0}    ==> dims = {3, 2}
nnodes = 7, ndims = 2, dims = {0, 0}    ==> dims = {7, 1}
nnodes = 6, ndims = 3, dims = {0, 0, 0} ==> dims = {3, 2, 1}
nnodes = 6, ndims = 3, dims = {0, 3, 0} ==> dims = {2, 3, 1}
nnodes = 7, ndims = 3, dims = {0, 3, 0} ==> error
```

Functions MPI_Cartdim_get(MPI_Comm comm, int *ndims) and MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims, int *periods, int *coords) allow us to obtain the characteristics of the Cartesian grid for an existing communicator comm with Cartesian topology. The first of them returns the size of the Cartesian grid in the ndims parameter. The second function contains three output parameters:

`dims` – array with the number of processes along each dimension of the Cartesian grid;

`periods` – array of flags that define the periodicity of each dimension (a dimension is periodic if the corresponding flag is not equal to 0);

`coords` – array of Cartesian coordinates of the current process.

All these parameters are integer arrays of size `maxdims`.

We still have one more useful feature provided by the Cartesian topology to describe: finding the ranks of the source and destination processes for the current process when sending data along a specified coordinate (i. e., when performing *data shifting*). This feature is provided by the `MPI_Cart_shift` function, which has the following parameters:

`MPI_Comm comm` – communicator with Cartesian topology;

`int direction` – the index of the Cartesian coordinate along which the shift is performed (indexing starts from 0);

`int disp` – shift step along the selected coordinate;

`int *rank_source` – rank of the source process (output parameter);

`int *rank_dest` – rank of the destination process (output parameter).

The returned data will correspond to a *cyclic shift* if the coordinate along which the shift is performed is *periodic* (this means that when defining the communicator `comm`, a *non-zero* element corresponding to this coordinate was specified in the `periods` array). Due to the `disp` parameter, the shift can be performed with any step, including negative (in the case of a negative step, the shift is performed in the *decreasing direction* of the given coordinate).

If the shift is not cyclic, it is possible that the current process does not have a source process and/or a destination process. For example, when shifting with a step 1, processes with a shift coordinate 0 do not have a source, and when shifting with a step -1, these processes do not have a destination. In such a situation, the corresponding output parameter takes the value `MPI_PROC_NULL`.

1.2.9. Graph topology

Now let us consider another type of virtual topology, *the graph topology*. It should be noted that the MPI library provides significantly fewer tools for working with graph topologies than for working with Cartesian topologies. Recall that for processes included in a Cartesian topology, it is possible to determine Cartesian coordinates by their ranks (and ranks by Cartesian coordinates); in addition, it is possible to create subgrids of smaller dimension (with each subgrid automatically associated with a new communicator); there is also the `MPI_Cart_shift` function, which simplifies message sending along a certain coordinate of the Cartesian grid.

As for the graph topology, after its definition using the `MPI_Graph_create` function, it is only possible to restore its characteristics (using functions `MPI_Graphdims_get` and `MPI_Graph_get`), as well as obtain information about the

number and ranks of all *neighboring processes* of a certain process in the graph defined by this topology (the `MPI_Graph_neighbors_count` and `MPI_Graph_neighbors` functions are provided for this).

To get acquainted with the possibilities associated with graph topology, let us solve the following task.

MPI5Comm29. The number of processes K is an even number: $K = 2N$ ($1 < N < 6$). An integer A is given in each process. Using the `MPI_Graph_create` function, define a graph topology for all processes as follows: all even-rank processes (including the master process) are linked in a chain $0 - 2 - 4 - 6 - \dots - (2N - 2)$; each process with odd rank R ($1, 3, \dots, 2N - 1$) is connected by edge to the process with the rank $R - 1$. Thus, each odd-rank process has a single neighbor, the first and the last even-rank processes have two neighbors, and other even-rank processes (the "inner" ones) have three neighbors (Fig. 23). Using the `MPI_Sendrecv` function, send the given integer A from each process to all its neighbors. The amount and ranks of neighbors should be determined by means of the `MPI_Graph_neighbors_count` and `MPI_Graph_neighbors` functions respectively. Output received data in each process in ascending order of ranks of sending processes.

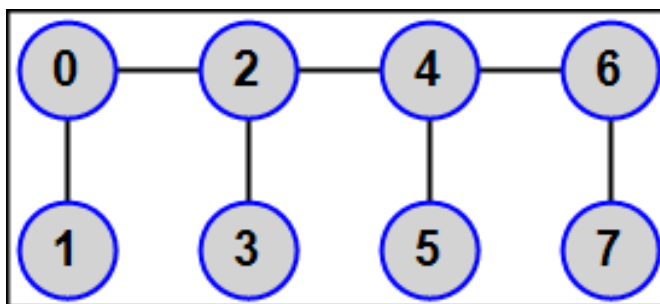


Fig. 23. Example of graph topology from the MPI5Comm29 task

When you run the program template for this task, the taskbook window will look like the one shown in Fig. 24. Simultaneously with the taskbook window, a picture from the task formulation will be displayed in the upper right corner of the screen, illustrating the topology used.

For greater clarity, let us show the processes together with their initial data in the form of a graph of the structure described in the task formulation (Fig. 25).

Since process 0 has two neighbors (processes of rank 1 and 2), it must send them the number 39 and receive from them the numbers 76 and 20. Process 1 has only one neighbor (process 0), so it must send it the number 76 and receive from it the number 39. Process 2, which has three neighbors, must send them the number 20 and receive from them the numbers 39, 31, and 96, and so on.

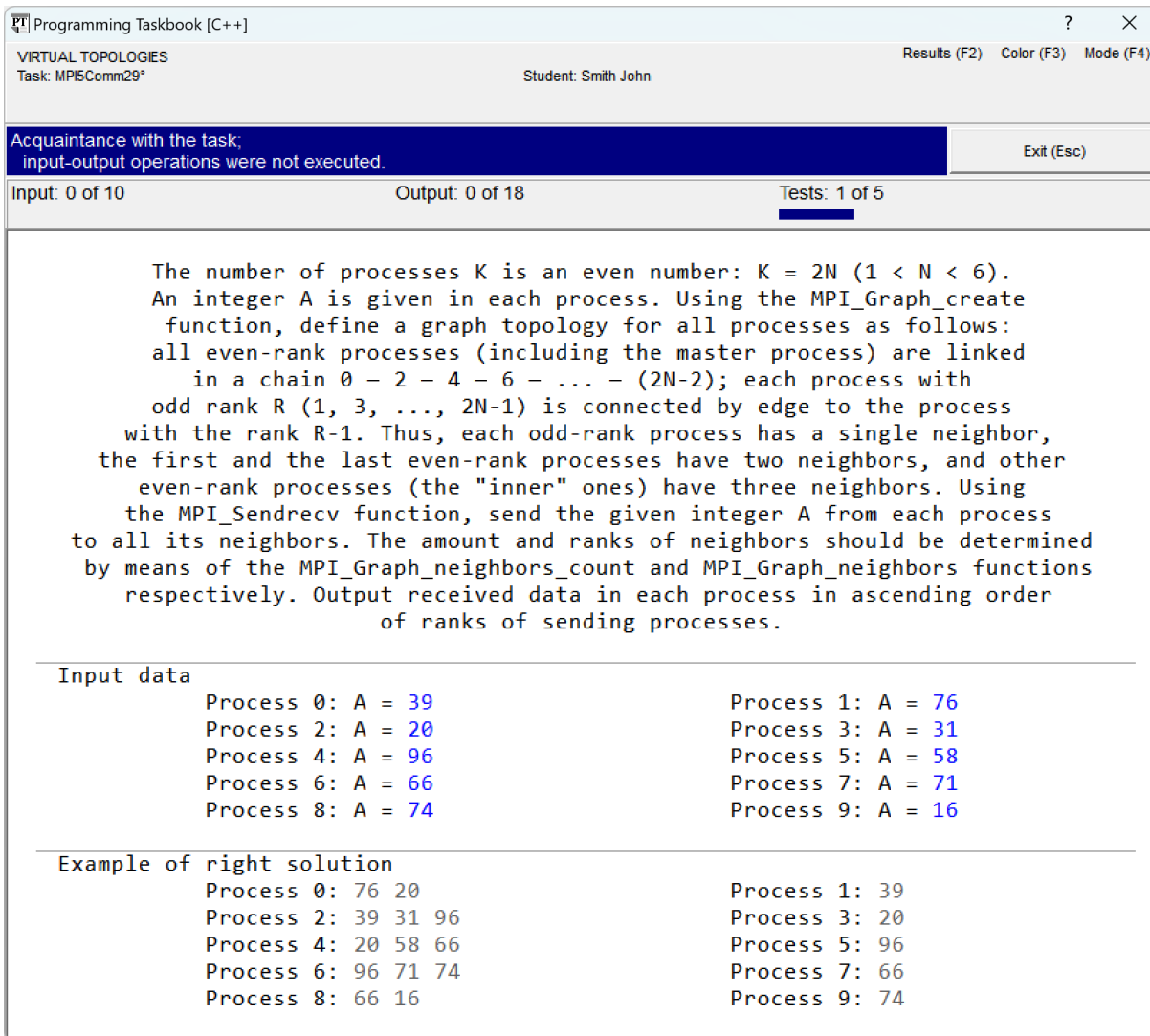


Fig. 24. Acquaintance running of the MPI5Comm29 task

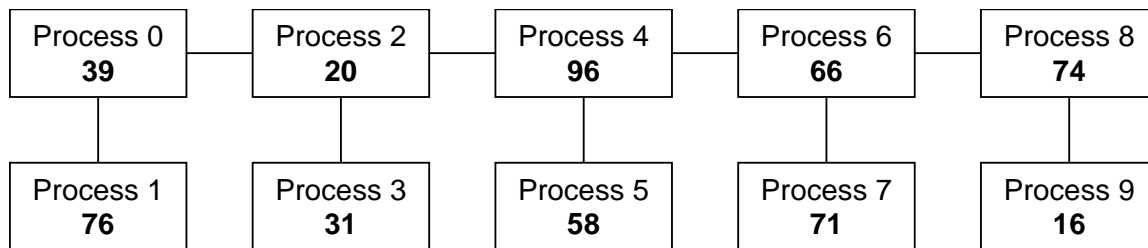


Fig. 25. Example of initial data for the MPI5Comm29 task

If each process had information about the number of its neighbors, as well as their ranks (in ascending order), then this would allow for a programming of data transfer actions for any process uniformly, regardless of how many neighbors it has. The required information about neighbors can be easily obtained if the appropriate graph topology is defined on the set of all processes by means of the `MPI_Graph_create` function, which has the following parameters:

MPI_Comm oldcomm – the original communicator for whose processes the graph topology is defined;
 int nnodes – the number of graph vertices;
 int *index – integer array of *vertex degrees*, the i -th element of which is equal to the total number of neighbors for the first i vertices of the graph;
 int *edges – integer array of *edges* containing an ordered list of neighbors for all vertices (vertices are numbered from 0);
 int reorder – an integer flag that determines whether the MPI environment can automatically reorder processes;
 MPI_Comm *graphcomm – the resulting communicator with graph topology (output parameter).

As for the Cartesian topology tasks (see Section 1.2.8), reordering of processes should be disabled by setting the reorder flag to 0.

To better understand the meaning of the array parameters that define the characteristics of the graph being created, let us list their elements for the graph shown in Fig. 25. The first vertex of the graph (a process of rank 0) has two neighbors, so the first element of the index array of vertex degrees will be equal to 2. The second vertex of the graph (a process of rank 1) has one neighbor, so the second element of the vertex degree array will be equal to 3 (1 is added to the value of the previous element). The third vertex (a process of rank 2) has three neighbors, so the third element of the vertex degree array will be equal to 6, and so on. We obtain the following set of values: 2, 3, 6, 7, 10, 11, 14, 15, 17, 18 (the last but one element of the array is 17, since a process of rank 8, like a process of rank 0, has two neighbors). Note that the value of the last element of the vertex degree array will always be twice the total number of edges in the graph.

In the edges array, it is necessary to indicate the ranks of all neighbors for each vertex (for greater clarity, we separate the groups of neighbors of each vertex with additional spaces, and indicate the rank of the vertex whose neighbors are listed below in brackets above):

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

The size of the resulting edge array is equal to the value of the last element of the vertex degree array.

If the number of processes is $size$, then the vertex degree array must contain $size$ elements. The size of the edge array depends on the graph structure; in our case, the edge array size is $2(size - 1)$, where $size$ is the number of processes.

When filling the index and edges arrays, it is convenient to separately process the first two (ranks 0 and 1) and the last two (ranks $size - 2$ and $size - 1$) graph vertices, and to use a loop for the rest vertices, processing two vertices (rank 2 and 3, 4 and 5, ..., $size - 4$ and $size - 3$) at each iteration. It is convenient

to use an auxiliary variable n equal to half the number of processes. Here is a fragment of the program that fills the arrays `index` and `edges`:

```
int n = size / 2;
int *index = new int[size],
    *edges = new int[2 * (size - 1)];
index[0] = 2;
index[1] = 3;
edges[0] = 1;
edges[1] = 2;
edges[2] = 0;
int j = 3;
for (int i = 1; i <= n - 2; i++)
{
    index[2 * i] = index[2 * i - 1] + 3;
    edges[j] = 2 * i - 2;
    edges[j + 1] = 2 * i + 1;
    edges[j + 2] = 2 * i + 2;
    index[2 * i + 1] = index[2 * i] + 1;
    edges[j + 3] = 2 * i;
    j += 4;
}
index[2 * n - 2] = index[2 * n - 3] + 2;
index[2 * n - 1] = index[2 * n - 2] + 1;
edges[j] = 2 * n - 4;
edges[j + 1] = 2 * n - 1;
edges[j + 2] = 2 * n - 2;
```

To check the correctness of this part of the algorithm, we output the values of the elements of the obtained arrays to the debug section of the taskbook window (since these arrays are formed in the same way in all processes, it is sufficient to output their values only in the master process):

```
if (rank == 0)
{
    for (int i = 0; i < size; i++)
        Show(index[i]);
    ShowLine();
    for (int i = 0; i < j + 3; i++)
        Show(edges[i]);
}
```

If the number of processes is 10 when the program is launched, then in the debug section we will see sets of values that match those that we obtained earlier (Fig. 26). To reduce the size of the window, the section with the task formula is hidden in the figure.

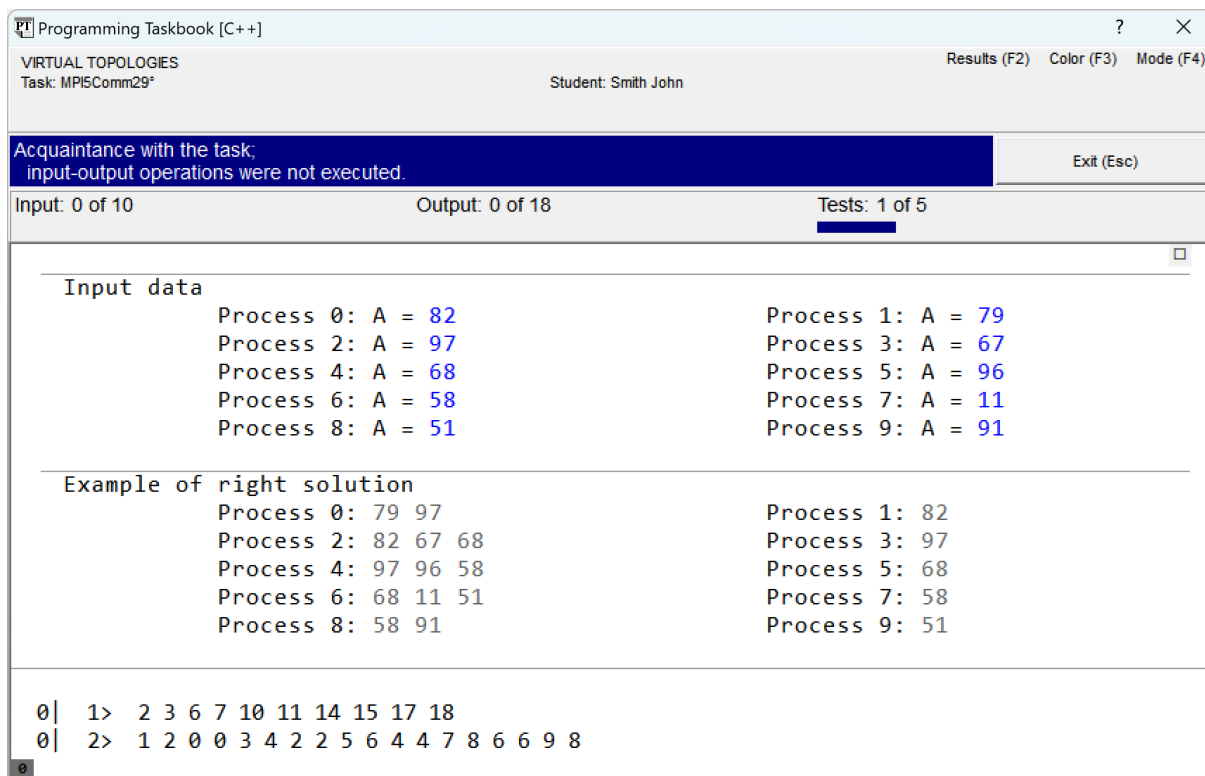


Fig. 26. Debug output of elements of the vertex degree array and the edge array

Once we have verified that the arrays are formed correctly, we create the graph topology by calling the `MPI_Graph_create` function in each process of the parallel application:

```
MPI_Comm g_comm;
MPI_Graph_create(MPI_COMM_WORLD, size, index, edges, 0, &g_comm);
```

Note that for obtaining characteristics of existing communicator `comm` with graph topology, MPI provides two functions: `MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)` and `MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index, int *edges)`. The first of them allows to obtain the number of vertices `nnodes` and the number of edges `nedges` of the graph for the communicator `comm`, and the second returns the vertex degree array `index` (the size of the array is specified in the variable `maxindex`) and the edge array `edges` (the size of this array is specified in the variable `maxedges`). The functions `MPI_Graphdims_get` and `MPI_Graph_get` play the same role for the graph topology as the functions `MPI_Cartdim_get` and `MPI_Cart_get` for the Cartesian topology.

It remains to implement the final part of the algorithm related to data transfer. In this part, for the current process (process of rank `rank`), the number count of its neighbors and the array neighbors of their ranks in the current graph topology should be determined, after which data exchange between the current process and each of its neighbors should be performed.

To determine the number of neighbors count of a process of rank `rank` included in a communicator `comm` with graph topology, the function

`MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *count)` is provided. Knowing the number of neighbors, one can allocate memory of the required size for the neighbors array of ranks of neighbors and determine these ranks using the function `MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int *neighbors)`, where the `maxneighbors` parameter specifies the size of the neighbors array. Note that by means of these functions, any process can determine not only its neighbors, but also the neighbors of any other process from this communicator.

To send data between a process and its neighbors, according to the task formulation, the `MPI_Sendrecv` function should be used, which ensures both receiving a message from a certain process and sending another message to it or to another process (see Section 1.2.1).

Thus, the final part of the solution will take the following form:

```
int count;
MPI_Graph_neighbors_count(g_comm, rank, &count);
int *neighbors = new int[count];
MPI_Graph_neighbors(g_comm, rank, count, neighbors);
int a, b;
MPI_Status s;
pt >> a;
for (int i = 0; i < count; i++)
{
    MPI_Sendrecv(&a, 1, MPI_INT, neighbors[i], 0,
                &b, 1, MPI_INT, neighbors[i], 0, g_comm, &s);
    pt << b;
}
delete[] index;
delete[] edges;
delete[] neighbors;
```

After running the program, we will receive a message that the task has been solved.

1.3. Additional features of the MPI interface (MPI-2 standard)

1.3.1. Distributed graph topology

The MPI-2 standard includes a new type of virtual topology, the *distributed graph topology*. The tasks included in the third subgroup of the `MPI5Comm` group are devoted to it (see Section 2.5.3).

Distributed graphs have the following main differences compared to "regular" graphs (discussed in Section 1.2.9):

- while to define the topology of a regular graph it is necessary to completely describe its structure in *all* processes of the defined communicator, a distributed graph can be defined "in parts", specifying in each process only those components of the structure of the distributed graph

that are "known" to this process (it is for this reason that the new type of graph is called a *distributed* graph);

- the edges of a distributed graph, unlike the edges of a regular graph, are *oriented*; each edge connects a source process and a destination process; it can be considered that each edge of a distributed graph is supplied with an "arrow" that determines the preferential direction of data transfer;
- each edge of the distributed graph can be supplied with an additional numerical characteristic, *weight*, which can be taken into account in one way or another by the MPI system when choosing optimal data transfer routes. Weights are non-negative; in addition to its numerical values, additional information in a parameter of the MPI_Info type can be associated with weights.

To create a communicator with a distributed graph topology, the MPI-2 standard provides two functions: `MPI_Dist_graph_create_adjacent` and `MPI_Dist_graph_create`. Both of these functions are collective; they must be called for all processes that are members of the original communicator `oldcomm`.

When using the function `MPI_Dist_graph_create_adjacent(MPI_Comm oldcomm, int indegree, int *sources, int *sourceweights, int outdegree, int *destinations, int *destweights, MPI_Info info, int reorder, MPI_Comm *distcomm)`, each process must determine only the set of graph edges that are associated with it, i. e. those edges, for which the given process is either a source or a receiver. For the first set of edges, their number `indegree` and an array `sources` containing the ranks of *source* processes are specified; the second set defines *destination* processes (receivers), namely, its size `outdegree` and an array of its ranks `destinations`; in addition, for each of these sets, the values of the edge weights are specified (the arrays `sourceweights` and `destweights`, respectively). If it is not necessary to specify weights, then the constant `MPI_UNWEIGHTED` must be specified as the `sourceweights` and `destweights` parameters in *all* processes of the `oldcomm` communicator.

Of course, the information specified must be consistent between the source and destination processes; in particular, both the source and receiver must specify the same weight for their common edge.

The `info` and `reorder` parameters are the same for both functions; they will be described below, along with the other parameters of the `MPI_Dist_graph_create` function.

`MPI_Dist_graph_create` function provides more flexible capabilities for defining the topology of a distributed graph (note that this is the function that is required to be used in the tasks devoted to the distributed graph topology, see Section 2.5.3). Using the `MPI_Dist_graph_create` function, each process can define *any* part of the distributed graph being created. Its parameters are as follows:

`MPI_Comm oldcomm` – the original communicator for whose processes the distributed graph topology is defined;

int n – the number of source processes for which edges are defined in this process;
 int *sources – array of size n containing the ranks of the defined source processes;
 int *degrees – array of size n that specifies the number of destination processes for each of the defined source processes;
 int *destinations – array containing the ranks of all destination processes for the defined source processes (its size is equal to the sum of all elements of the degrees array);
 int *weights – array containing the weights of all defined edges (its size is also equal to the sum of all elements of the degrees array), or the constant MPI_UNWEIGHTED if weights do not need to be specified;
 MPI_Info info – additional information related to the defined weights; if it is not provided, then the constant MPI_INFO_NULL is specified as this parameter;
 int reorder – an integer flag that determines whether the MPI environment can automatically reorder processes;
 MPI_Comm distcomm – the resulting communicator with distributed graph topology (output parameter).

If the constant MPI_UNWEIGHTED is used, it must be specified when calling the MPI_Dist_graph_create function in *all* processes of the oldcomm communicator.

When defining a distributed graph using the MPI_Dist_graph_create function, various strategies can be used. We illustrate them using the example of a simple distributed graph shown in Fig. 27 (assuming for simplicity that weights are not specified).

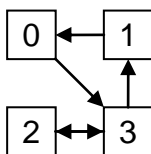


Fig. 27. Example of a distributed graph

We can define for each process only those edges for which it is a source:

```

Process 0: n = 1, sources = {0}, degrees = {1}, destinations = {3}
Process 1: n = 1, sources = {1}, degrees = {1}, destinations = {0}
Process 2: n = 1, sources = {2}, degrees = {1}, destinations = {3}
Process 3: n = 1, sources = {3}, degrees = {2}, destinations = {1, 2}
  
```

Another way (which is required to be applied in the MPI5Comm group tasks) is that the graph is completely defined in some single process:

```

Process 0: n = 4, sources = {0, 1, 2, 3}, degrees = {1, 1, 1, 2},
           destinations = {3, 0, 3, 1, 2}
Process 1: n = 0, sources = {}, degrees = {}, destinations = {}
Process 2: n = 0, sources = {}, degrees = {}, destinations = {}
Process 3: n = 0, sources = {}, degrees = {}, destinations = {}
  
```

Any intermediate variants are also possible.

Here is a variant of defining the same distributed graph using the `MPI_Dist_graph_create_adjacent` function:

```
Process 0: indegree = 1, sources = {1},
           outdegree = 1, destinations = {3}
Process 1: indegree = 1, sources = {3},
           outdegree = 1, destinations = {0}
Process 2: indegree = 1, sources = {3},
           outdegree = 1, destinations = {3}
Process 3: indegree = 2, sources = {0, 2},
           outdegree = 2, destinations = {1, 2}
```

For the distributed graph topology, as for the regular graph topology, there are two functions that allow restoring its characteristics. However, unlike the corresponding functions for the regular graph topology, which allow *any* process to obtain complete information about the neighbors *of any* process, the functions for the distributed graph topology return information about the neighbors *of the process in which they are called*.

The function `MPI_Dist_graph_neighbors_count(MPI_Comm comm, int *indegree, int *outdegree, int *weighted)` allows to determine the number of source and destination processes (output parameters `indegree` and `outdegree`, respectively). In addition, the output flag parameter `weighted` takes a non-zero value if the constant `MPI_UNWEIGHTED` was *not used* when defining the distributed graph topology.

The function `MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int *sources, int *sourceweights, int maxoutdegree, int destinations, int *destweights)` returns information about the ranks of source and destination processes (output arrays `sources` and `destinations` of size `maxindegree` and `maxoutdegree`, respectively). In addition, if the `MPI_Dist_graph_neighbors_count` function returned a non-zero `weighted` flag, then the weights of the edges associated with the source and destination processes are returned in the `sourceweights` and `destweights` arrays.

1.3.2. Parallel input-output. File access functions

One of the important innovations of the MPI-2 standard is support for *parallel file input-output*.

In parallel programs, the initial data are usually read from files, and the results are written to files. Due to the lack of parallel file access facilities in MPI-1, it was necessary to organize data reading in some selected (usually master) process, and then send them to all other processes of the parallel application. Similarly, to save the obtained results, it was necessary to first send these results to some process and organize their writing to a file in this process. If in a parallel application it was necessary to provide access to the same file for several processes, then special synchronization actions had to be performed to correctly organize such an access.

In the MPI-2 standard, it became possible to read or write file data in each process of a parallel application without making any special efforts to synchronize access to the file.

The file input-output mechanisms defined in the MPI-2 standard are extremely flexible. They include both *local* and *collective* file access functions, with three *positioning options* provided for each type of access: either *explicit*, by specifying the file position in a special function parameter, or implicit, using the current value of an *individual* (own for each process) or *shared* (by all processes) *file pointer*; both types of file pointers can be used for both local and collective access functions. In addition, all the types of parallel file access described above are implemented in two variants: *blocking* and *non-blocking* (the differences between them are similar to the differences between blocking and non-blocking data transfer operations, see Sections 1.2.1 and 1.2.3).

Thus, by combining the options described above, we can obtain $2 \cdot 2 \cdot 3 \cdot 2 = 24$ variants of parallel file access:

- reading *or* writing (2 options); the name of the reading functions uses the word read, the name of the writing functions uses the word write;
- local *or* collective access (2 options); the word all is added to the name of collective access functions (except for functions that use shared file pointers, for which the word shared is used in the case of local access, and the word ordered is used in the case of collective access);
- explicit positioning *or* individual file pointer *or* shared file pointer (3 options); the word at is added to the name of functions for explicit positioning, the word shared or ordered is added to the name of functions using a shared pointer;
- blocking *or* non-blocking access (2 options); for local functions, in case of non-blocking access, the prefix i is added to the words read and write (iread and iwrite); collective non-blocking functions are paired, the name of the first of them ends with the word begin, and the name of the second ends with the word end.

Each file access variant has its own MPI function (or pair of functions for collective non-blocking access options), so the total number of functions is 30.

The name of a function can easily determine the associated file access variant. For example, MPI_File_read_at_all provides a *blocking read* option (read) based on *explicit positioning* (at) and is *collective* (all), while MPI_iwrite_shared provides a *non-blocking write* option (iwrite), uses a *shared file pointer*, and is *local* (both of which are specified by the word shared). The simplest file read and write functions are named MPI_File_read and MPI_File_write; these functions are *blocking*, *local*, and use *individual file pointers*.

Table 1 lists the names of all functions related to file access. The functions are grouped into the categories described above: *Positioning* (explicit position-

ing, local file pointer, or shared file pointer), *Access* (blocking or non-blocking), *Coordination* (local or collective functions).

Table 1

MPI file access functions

Positioning	Access	Coordination (input and output)	
		<i>Local</i>	<i>Collective</i>
<i>Explicit</i>	<i>Blocking</i>	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	<i>Non-blocking</i>	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_read_at_all_begin MPI_File_read_at_all_end MPI_File_write_at_all_begin MPI_File_write_at_all_end
<i>Individual file pointer</i>	<i>Blocking</i>	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	<i>Non-blocking</i>	MPI_File_iread MPI_File_iwrite	MPI_File_read_all_begin MPI_File_read_all_end MPI_File_write_all_begin MPI_File_write_all_end
<i>Shared file pointer</i>	<i>Blocking</i>	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	<i>Non-blocking</i>	MPI_File_iread_shared MPI_File_iwrite_shared	MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end

Let us describe the blocking functions in more detail, since they are the ones that need to be used in learning tasks.

Blocking functions with explicit positioning (MPI_File_read_at, MPI_File_write_at, MPI_File_read_at_all, MPI_File_write_at_all) have the following parameters:

MPI_File *f* – file variable (file descriptor);

MPI_Offset *offset* – file position from which data reading/writing begins;

void **buf* – buffer for storing read (for read functions) or written data (for write functions); for read functions, it is an output parameter;

int *count* – the size of the buffer *buf* (in elements of type *datatype*);

MPI_Datatype *datatype* – type of elements of the buffer *buf*;

MPI_Status **status* – additional information about the completed read/write operation (output parameter).

A file descriptor *f* of type MPI_File is defined when opening a file, which is performed by the function MPI_File_open. This function is collective and must be called for all processes of some communicator; as a result, all processes of this communicator gain access to the file. A detailed description of the function MPI_File_open is given in Section 1.3.3.

The `MPI_Offset` type is similar to the previously discussed `MPI_Aint` type (see Section 1.2.6); it is intended to store offsets between different file positions and is implemented as a signed integer type, the size of which is sufficient to store any possible offset in the disk address space.

Using the output parameter `status`, you can determine the number of elements actually read/written (for this, as usual, you need to use the `MPI_Get_count` function, see Section 1.2.1), and also get the error code (using the field `MPI_ERROR`). Other fields of the `MPI_Status` structure are not used in file access functions.

Calling functions with explicit positioning does not affect the current position of the individual and shared file pointer.

All other blocking functions (`MPI_File_read`, `MPI_File_write`, `MPI_File_read_all`, `MPI_File_write_all`, `MPI_File_read_shared`, `MPI_File_write_shared`, `MPI_File_read_ordered`, `MPI_File_write_ordered`) have the same set of parameters, differing only absence the offset parameter: when using these functions, the starting position for reading/writing file data is determined by the position of the individual or shared file pointer.

The position of the file pointer used in the function is automatically changed during read/write operations. In addition, this position can be set explicitly using the functions `MPI_File_seek` (for the individual file pointer) and `MPI_File_seek_shared` (for the shared file pointer). These functions have the same set of parameters:

`MPI_File f` – file variable;

`MPI_Offset offset` – the required offset of the file pointer (can be either positive or negative);

`int whence` – the offset mode to use.

Three constants are provided to specify the `whence` offset mode:

`MPI_SEEK_SET` – the offset is counted from the starting file position;

`MPI_SEEK_CUR` – the offset is counted from the current position of the file pointer;

`MPI_SEEK_END` – the offset is counted from the end-of-file marker.

In the `MPI_SEEK_SET` mode, only positive values of the offset parameter are allowed; in the `MPI_SEEK_END` mode, only negative values are allowed.

To determine the current position of the individual and shared file pointer, the functions `MPI_File_get_position` and `MPI_File_get_position_shared` are provided, which have the same set of parameters:

`MPI_File f` – file variable;

`MPI_Offset *offset` – current position of the file pointer (output parameter).

It is necessary to take into account that when using *local* functions associated with a *shared* file pointer (`MPI_File_read_shared` and `MPI_File_write_shared`), no additional synchronization is performed, i. e. the order of reading data will

depend on the moment of time at which the functions were called in different processes.

Collective functions associated with a *shared* file pointer (MPI_File_read_ordered and MPI_File_write_ordered) provide the synchronization. When they are called (the function call, as for any collective functions, must be performed in *all* processes of the communicator for which the file was opened), the actions to read/write data using the shared file pointer are performed sequentially, in *ascending order of the ranks* of the processes in this communicator.

For all the functions discussed above, the offset position is specified *in the elements of the elementary type* etype and is calculated taking into account *the initial offset* disp, specified when defining the file datatype (see the description of the MPI_File_set_view function in Section 1.3.3).

To convert the offset position (in elements of elementary type) to the disp offset in *bytes* measured from the beginning of the file, the helper function MPI_File_get_byte_offset(MPI_File f, MPI_Offset offset, MPI_Offset *disp) is provided.

The function MPI_File_get_size(MPI_File f, MPI_Offset *size) returns the size of the file f in the size parameter (in bytes). Its "pair" is the function MPI_File_set_size(MPI_File f, MPI_Offset size), which allows changing the size of the file f by setting it equal to size bytes. It is allowed to both decrease the file size (in this case, its final part is deleted) and increase it (in this case, the content up to the added final part is undefined). The MPI_File_set_size function is collective and must be called (with the same value of the size parameter) by all processes of the communicator in which the file was opened.

In addition to various options for organizing reading and writing, MPI-2 provides a flexible way to configure the type of file data (*file view*): from the simplest, in which the file is interpreted as a set of sequentially located *bytes*, to very complex ones, in which the file view can include not necessarily contiguous groups of elements (empty spaces are allowed at the beginning, at the end, and between some elements). In addition, each process can define its own file data view. These possibilities are discussed in detail in the next section.

By executing the tasks of the MPI6File group (see Section 2.6), you can become familiar with most of the features of parallel file input-output. The first subgroup of this group (MPI6File1–MPI6File8) studies *local* file operations, the second subgroup (MPI6File9–MPI6File16) studies *collective* file operations, and the final subgroup (MPI6File17–MPI6File30) is devoted to various methods of defining *complex types of file data*. The tasks of each of the first two subgroups use all three *positioning options* (based on explicit indication of the position or on the use of individual or shared file pointers); the third group uses collective file operations, which mainly use individual file pointers. Only additional features related to rather rarely used non-blocking file access remain outside the MPI6File group.

1.3.3. Parallel input-output: an example. Setting up the file view

As an example of a task related to parallel file input-output, we consider the MPI6File26 task, which is part of the third subgroup of the MPI6File group (see Section 2.6.3).

MPI6File26. The name of file is given in the master process. In addition, four real numbers, namely, A, B, C, D , are given in each process. The number of processes is equal to K . Create a new file of real numbers with the given name and write the given real numbers to this file as follows: $A_0, A_1, \dots, A_{K-1}, B_{K-1}, \dots, B_1, B_0, C_0, C_1, \dots, C_{K-1}, D_{K-1}, \dots, D_0$ (an index indicates the process rank). To do this, use one call of the `MPI_File_write_all` collective function and a new file view with the `MPI_DOUBLE` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of two real numbers (with an additional empty space between these numbers) and a terminal empty space of the appropriate size.

In this task, we will learn about the steps required to open a file in parallel mode, to use one of the most common collective file operations, `MPI_File_write_all`, and to define complex file datatypes. When you run this task for the first time, the taskbook window will look like the one shown in Fig. 28.

Tasks related to parallel file input-output have a number of special features. First, these tasks are the first to include *string* input data containing a file name. This name is always given in the master process; the easiest way to send it to other processes of the parallel application is to use the collective function `MPI_Bcast`. The preamble to the MPI6File group states that it is sufficient to use the `char[12]` array to store the file name. When sending this array, the `MPI_CHAR` type should be used. Note that to input a string s (of the `char*` or `char[]` type) into the taskbook, it is sufficient to use a *single* call to the `pt` input stream: `pt >> s`.

The second feature of the task is the inclusion of *file data*. In some tasks, file data is specified in the input data section (if the task requires processing an existing file), in others, file data is specified in the results section (if a new file is required to be created or an existing file must be transformed). File data is displayed in the taskbook window on several lines, with the current file element number indicated at the beginning of each line (elements are numbered from 1). All tasks use *typed binary files* consisting of either integers or real numbers. Note that the contents of such files cannot be viewed in text editors, so the ability to display them in the taskbook window is especially useful. The method for splitting file elements into lines in the taskbook window depends on the features of the task. In our case, we need to write to the file the first elements given in each process, then the second elements (in reverse order), then the third elements, and finally the fourth elements (also in reverse order). Therefore, it is most convenient to represent the contents of the file as *four* lines, each of which contains elements of all processes that have the same order number.

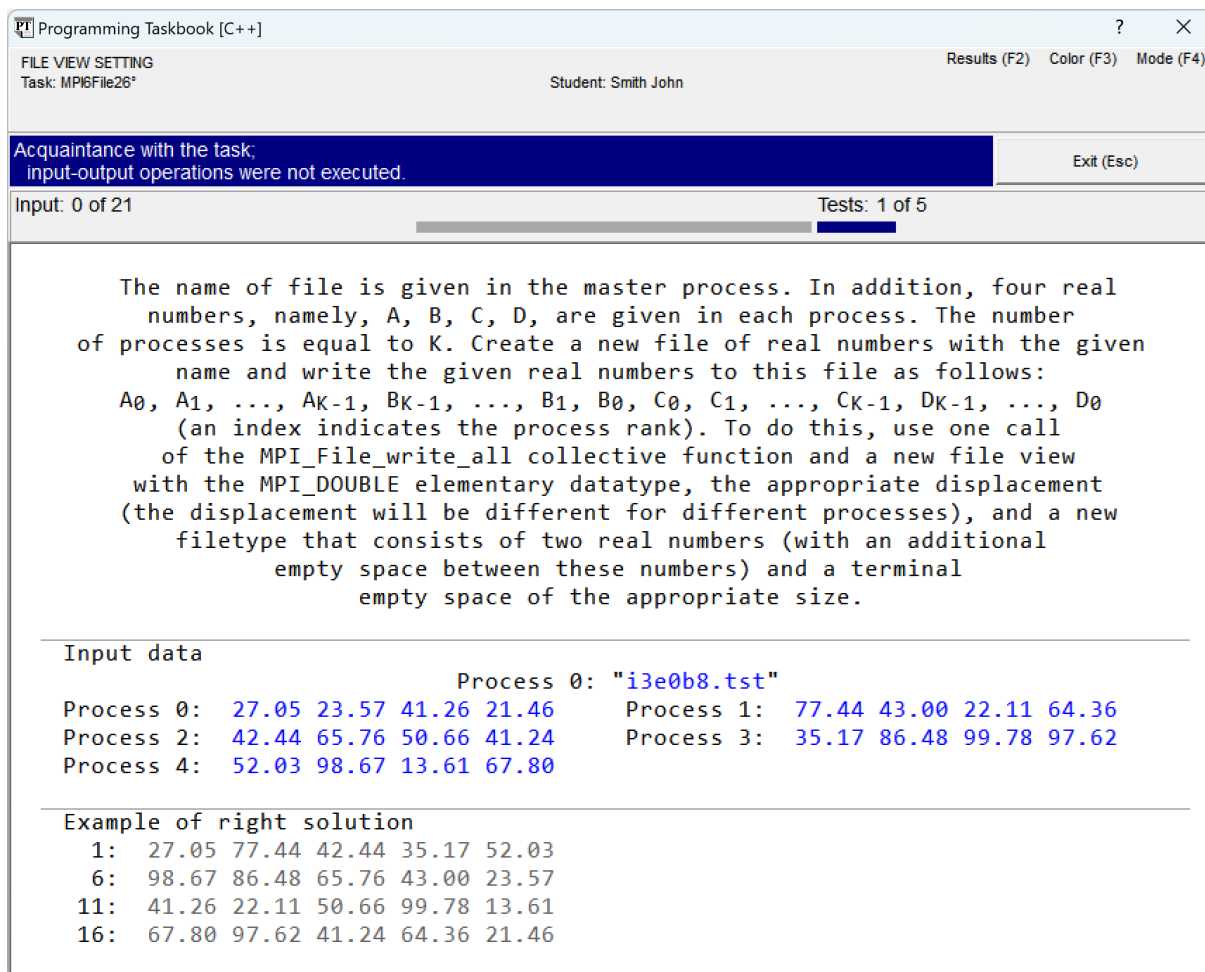


Fig. 28. Acquaintance running of the MPI6File26 task

You should also pay attention to the special type of *output indicator* in the indicators section (located above the task formulation section). This indicator is displayed in gray and does not contain any accompanying text. This means that the results obtained during the task solving do not need to be sent to the taskbook (using the `pt` output stream); you only need to write the required data to a file with the specified name, after which the taskbook itself will check the correctness of the created file.

Let us start to solve the task. At the first stage, we will organize the input of the given data and the transfer of the file name to all processes:

```

char name[12];
if (rank == 0)
    pt >> name;
MPI_Bcast(name, 12, MPI_CHAR, 0, MPI_COMM_WORLD);
double a[4];
for (int i = 0; i < 4; i++)
    pt >> a[i];

```

After launching a new version of the program, a message will be displayed in the taskbook window stating that all the data are input, but the resulting file is not created (Fig. 29).

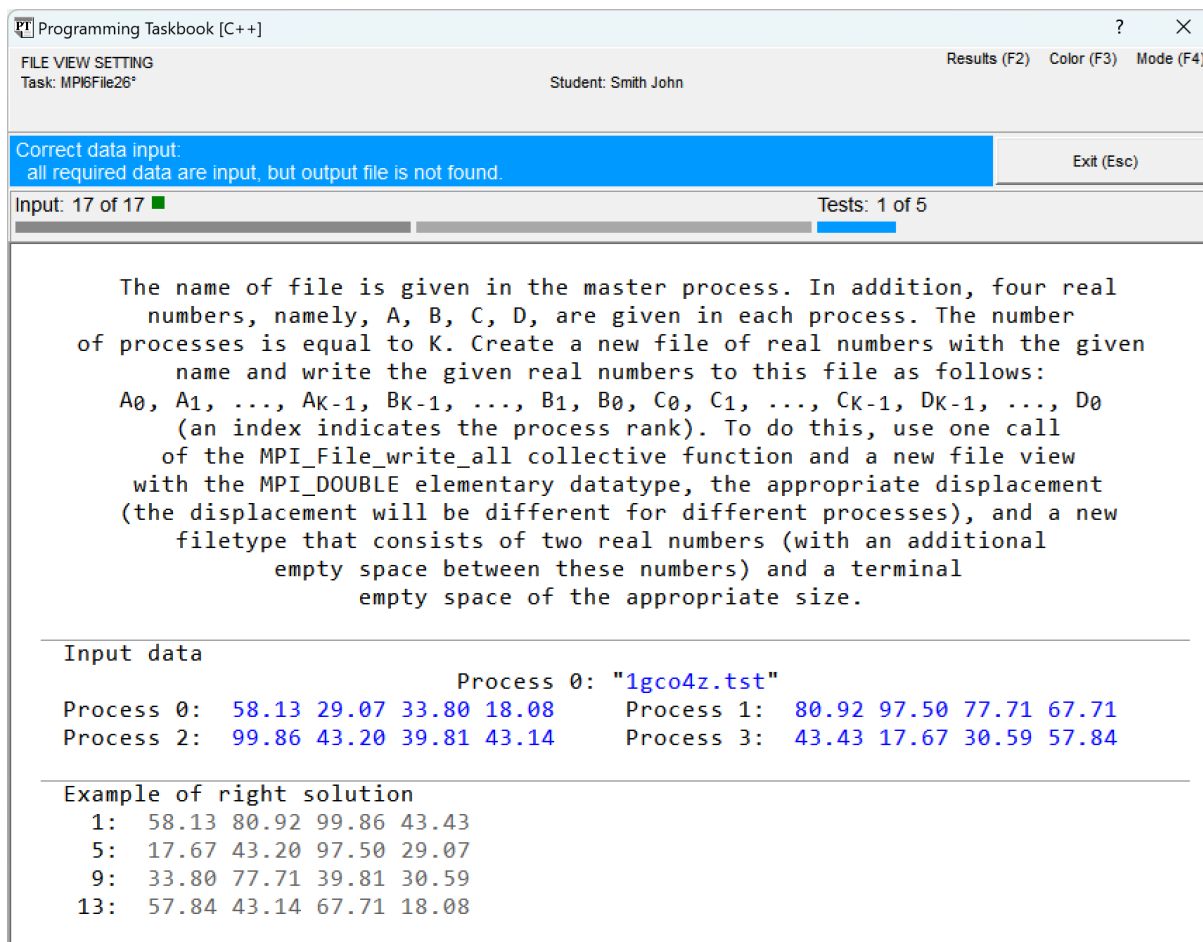


Fig. 29. The first stage of solving the MPI6File26 task (input of given data)

In the next stage, we will perform actions related to opening and closing a file:

```
MPI_File f;
MPI_File_open(MPI_COMM_WORLD, name,
              MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &f);
// Access to the contents of file f
MPI_File_close(&f);
```

First, a file variable `f` of type `MPI_File` is declared, which will be used in all functions related to file operations. Then, the function `MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info info, MPI_File *f)` is called. This function is collective and must be called in all processes of the communicator `comm` specified as its first parameter. The file name `filename` is specified in the second parameter, and the access mode `amode` is specified in the third parameter; both of these parameters must have the same values in all processes. To combine the options of the access mode, they must be combined with the `|` operator. For ex-

ample, in our case, it is necessary *to create* a file (MPI_MODE_CREATE), and then open it *for writing only* (MPI_MODE_WRONLY). Here are some more of the available options of the access mode:

- MPI_MODE_RDONLY – open for reading only;
- MPI_MODE_RDWR – open for reading and writing;
- MPI_MODE_APPEND – automatically move all file pointers (both individual and shared) to the end of the file when it is opened;
- MPI_MODE_DELETE_ON_CLOSE – automatically delete a file when it is closed.

The info parameter of the MPI_File type allows you to specify additional file characteristics; if this is not necessary (as is usually the case), then the constant MPI_INFO_NULL is specified.

The last parameter of the MPI_File_open function is a file variable *f* associated with the open file. This parameter is an output parameter, so it must be passed using a pointer.

After finishing working with the file, it must be closed with the function MPI_File_close(MPI_File *f), which, like the MPI_File_open function, is collective and must be called in all processes in which the file was opened. The file variable *f* is passed to this function as a pointer; as a result of executing the MPI_File_close function, the value of the file variable *f* becomes equal to MPI_FILE_NULL.

When you run a new version of the program, the message on the information panel will change slightly: *"Correct data input: all required data are input, resulting file is empty"*. This message indicates that the file was created.

Note that after the program is completed, the taskbook automatically deletes all files created during its working. You can verify that the required file was actually created by viewing the contents of the working directory *before* closing the taskbook window.

To simplify the actions of filling the file with the required data as much as possible, it is necessary to define *the file data view*. For this purpose, the MPI_File_set_view function is provided with a large number of parameters: (MPI_File *f*, MPI_Offset *disp*, MPI_Datatype *etype*, MPI_Datatype *filetype*, char **datarep*, MPI_Info *info*).

The first parameter *f* specifies a file variable associated with the open file. The next three parameters define the basic characteristics of the file datatype:

- disp* – the starting offset (in bytes) that will be performed in this process before reading the first file element;
- etype* – the *elementary* datatype on the basis of which the *filetype* is defined (in all processes using this file, the same elementary datatype must be specified);
- filetype* – a *filetype* used when directly reading or writing file data; it is a set of elements of the elementary type, as well as empty spaces (the size of which must also be a multiple of the size of the elementary type).

Positioning in a file is always performed in elements of the elementary type, and empty spaces included in the filetype are not taken into account. Note that by default (if the `MPI_File_set_view` function is not called), the initial offset `disp` is assumed to be 0, and the `etype` and `filetype` types are assumed to be `MPI_BYTE`.

The `datarep` parameter determines how file data should be interpreted. In the simplest case, when all processes are running on the same computer (or on different computers with the same architecture), the "native" option can be used, in which the file representation exactly matches the representation of the same data in RAM. In more complex situations (in particular, when the program is running on heterogeneous hardware), other data representation options are used, in which the data stored in memory undergoes additional transformation before being written to the file (and a similar transformation is performed when reading data from the file into memory). In our case, it is sufficient to use the "native" option.

Finally, the `info` parameter allows you to specify additional information related to the file view being defined. Typically, this parameter is set to `MPI_INFO_NULL`.

Let us define the type of file data that will be most convenient for solving our task. If we denote by $[R]$ the file elements (of type `MPI_DOUBLE`) that need to be written in the process R ($R = 0, 1, \dots, K-1$), then the distribution of these elements in the file will be as follows:

[0][1][2]...[K-1][K-1]...[2][1][0][0][1][2]...[K-1][K-1]...[2][1][0]

Thus, the file will contain two fragments of the same structure (the first fragment is highlighted in bold). We only need to define a filetype that allows us to correctly read the data from the first fragment (the second fragment that match the structure of the first fragment will also be read correctly).

It is natural to select the `MPI_DOUBLE` type as the elementary type. The filetype will include two real numbers with an empty space between them. The size of the space depends on the rank of the process: for process 0 this size is the largest and equal to the size of $2 \cdot K - 2$ real numbers, for process $K-1$ there is no space between the elements.

Thus, if for a process of rank R we set the initial offset `disp` to be equal to $R \cdot d$, where d is equal to the size of the `MPI_DOUBLE` type in bytes, then the filetypes for different processes can be represented as follows (an element of the `MPI_DOUBLE` type is denoted by $[*]$, and the size of the empty interval in elements of the `MPI_DOUBLE` type is indicated in parentheses):

process 0: $[*](2 \cdot K - 2)[*]$
process 1: $[*](2 \cdot K - 4)[*](2)$
process 2: $[*](2 \cdot K - 6)[*](4)$
...
process $K-1$: $[*][*](2 \cdot K - 2)$

Note that the filetype *extent* is the same for all processes and equals $2 * K$ (in elements of type MPI_DOUBLE); the total size of empty spaces is also the same, it is equal to $2 * K - 2$ (also in elements of type MPI_DOUBLE).

When defining data types containing initial or final empty spaces, we will use the new convenient and flexible approach introduced in the MPI-2 standard (another approach, based on the use of the capabilities of the MPI -1 standard, is described in Section 1.2.6).

Recall that in our program the number of processes K is stored in the variable `size`, and the rank of the process R is stored in the variable `rank`. We find the size of the MPI_DOUBLE type using the `MPI_Type_size` function and save it in the variable `dbl_sz`.

First, we define an auxiliary type `t0` containing two real numbers with the required space between them, and then we supplement this type with a final empty space using the `MPI_Type_create_resized` function (see Section 1.2.6), obtaining the final type `t`:

```
int dbl_sz;
MPI_Type_size(MPI_DOUBLE, &dbl_sz);
MPI_Datatype t0, t;
MPI_Type_vector(2, 1, 2 * (size - rank) - 1, MPI_DOUBLE, &t0);
MPI_Type_create_resized(t0, 0, 2 * size * dbl_sz, &t);
```

Once we have defined the type `t`, we can use it in the `MPI_File_set_view` function:

```
MPI_File_set_view(f, rank * dbl_sz, MPI_DOUBLE, t, "native",
MPI_INFO_NULL);
```

Now, to write the required data to the file, we only need to perform a *single call* to the collective function `MPI_File_write_all`, passing to it an array `a` of 4 given real numbers:

```
MPI_File_write_all(f, a, 4, MPI_DOUBLE, MPI_STATUS_IGNORE);
```

The actions for setting the file view and writing data to the file must be performed between the file opening and closing operations (this position of our program was marked with the comment *"Access to the contents of file f"*).

Here is the full text of the resulting solution:

```
void Solve()
{
    Task("MPI6File26");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    char name[12];
```



```
if (rank == 0)
    pt >> name;
MPI_Bcast(name, 12, MPI_CHAR, 0, MPI_COMM_WORLD);
double a[4];
for (int i = 0; i < 4; i++)
    pt >> a[i];
MPI_File f;
MPI_File_open(MPI_COMM_WORLD, name,
    MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);
int dbl_sz;
MPI_Type_size(MPI_DOUBLE, &dbl_sz);
MPI_Datatype t0, t;
MPI_Type_vector(2, 1, 2 * (size - rank) - 1, MPI_DOUBLE, &t0);
MPI_Type_create_resized(t0, 0, 2 * size * dbl_sz, &t);
MPI_File_set_view(f, rank * dbl_sz, MPI_DOUBLE, t, "native",
    MPI_INFO_NULL);
MPI_File_write_all(f, a, 4, MPI_DOUBLE, MPI_STATUS_IGNORE);
MPI_File_close (&f);
}
```

After running the program, we will receive a message that the task has been solved.

1.3.4. One-sided communications: general description

One-sided communications, introduced in the MPI-2 standard, allow data transfer between processes to be organized without performing special actions on the side of both participants in the interaction. Recall that in the traditional scheme of interaction between two processes, it is necessary to call one of the sending functions (`MPI_Send` or its variants, both blocking and non-blocking) on the side of the *source process* (*sender*), as well as one of the receiving functions (`MPI_Recv` or its non-blocking variant) on the side of the *destination process* (*receiver*). However, a situation is possible when the source process "does not know" what part of its data will be needed by other processes, or, conversely, the destination process "does not know" what data will be sent to it by other processes. In the latter case, however, it is possible to use the `MPI_ANY_SOURCE` and `MPI_ANY_TAG` parameters, which allow receiving data from arbitrary sources, as well as determining the nature of the received data based on the `msgtag` tag. But such a possibility usually requires preliminary analysis of the received message (by calling the `MPI_Probe` function or its non-blocking version) and thus complicates the algorithm on the side of the destination process. If the source process "does not know" who may need its data, then the use of the traditional MPI "send–receive" mechanism becomes impossible.

Meanwhile, in *multithreaded programming*, it is precisely one-sided interactions that are the standard way of exchanging information: a thread can write data to some area of *shared memory*, after which any other thread will be able to

access this memory and obtain the required part of the data. To implement this type of interaction in MPI, it is necessary that the processes of a parallel MPI application have the ability to define "shared areas" of their memory, which other processes of the same application could directly access. For this reason, the mechanism of one-sided communications in MPI is also called *remote memory access* (RMA).

A section of memory of some process that can be accessed by any process of a parallel application is called *a window*.

To **create a window**, there is a collective function `MPI_Win_create(void *base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win *win)`, which must be called for all processes of the communicator `comm`. One call to this function allows creating shared memory areas (i. e. windows) in *all processes* included in the specified communicator; all these areas will be associated with the same *window handle* `win` of type `MPI_Win`, which is returned in the last parameter of the function. Using this handle, any process can access the window of any other process.

The starting address base of the window for the current process and its size (in bytes) are specified as the first two parameters of the `MPI_Win_create` function. Windows defined in different processes may have different sizes; if it is not necessary to create a window for some processes, then when calling the `MPI_Win_create` function in these processes, it is sufficient to set the size parameter to zero.

The third parameter of the `MPI_Win_create` function, named `disp_unit`, is intended to simplify the address arithmetic used when calling the window access functions (described below): the offset (in bytes) from the beginning of the window specified in the access functions is automatically multiplied by the `disp_unit` value. If the window is intended to store elements of some array, then it is convenient to specify the size of the array element as the `disp_unit` parameter. This will allow us to specify the element *index* in the access functions when accessing a particular array element. If we set the `disp_unit` parameter to 1, then the offset in the access functions must be specified in *bytes*.

The fourth parameter `info` allows additional information to be associated with the window being created; it is usually not used and in this case is equal to the constant `MPI_INFO_NULL`.

After finishing working with the access window, it must be destroyed by calling the function `MPI_Win_free(MPI_Win *win)`; the handle of the destroyed window receives the value `MPI_WIN_NULL`, meaning that no operations can be performed with this window.

To **access a window**, the MPI library provides three functions: `MPI_Get` (read access), `MPI_Put` (write access), and `MPI_Accumulate` (modify access). The process that calls these functions is called *the origin process*; the process containing the window being accessed is called *the target process*. It is the origin

process that provides one-sided communication, while the target process plays a passive role, not performing any special actions. Both the origin and target processes can act as either a source or a receiver of data. If the MPI_Get function is used, then the origin process is the receiver of data, and the target process is the source; if the MPI_Put or MPI_Accumulate function is used, then the origin process is the source, and the target process is the receiver of data.

Most of the parameters of all access functions (MPI_Get, MPI_Put, MPI_Accumulate) are the same. The first three parameters define the data "on the side" of the origin process: these are void *origin_addr, int origin_count, MPI_Datatype origin_datatype. The first parameter specifies the address of the beginning of the data buffer, the second specifies the number of elements in the buffer, and the third specifies the type of elements in the buffer. The next four parameters define the data "on the side" of the target process: int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype. The first parameter specifies the rank of the target process, the second is the offset from the start of the window in this target process (in disp_unit units specified when defining the window, see the description of the MPI_Win_create function above), the third and fourth determine the number and type of window elements to be accessed (note that the number is measured in elements of the specified type and is not related to disp_unit units).

If the types of elements in the buffer of the origin process (origin_datatype) and in the window of the target process (target_datatype) match, as is most often the case, then the specified sizes (origin_count and target_count) must also match.

The last parameter in all access functions specifies the window descriptor win of the MPI_Win type.

The MPI_Accumulate function has the same parameters and one additional parameter op of type MPI_Op, specified before the win parameter. The op parameter specifies the operation used to change the contents of the window: the operation is applied to the buffer element in the origin process and to the corresponding window element in the target process, and the result of the operation is written to the same window element.

Any standard reduction operation defined in the MPI library (see Section 1.2.5) can be used as the operation op. User-defined operations cannot be used.

The MPI_Accumulate function is implemented in such a way that it can be safely used in the situation when some window section of the target process is modified by *several* origin processes. Note that the MPI_Put function *does not* have this property: if two origin processes try to write their data to the same window section of the target process during one access period, then only the data of one of the origin processes will be stored in this window, and it is impossible to say in advance which one).

When calling the MPI_Get function, the contents of the buffer in the origin process are changed; when calling the MPI_Put and MPI_Accumulate functions, the

contents of the specified section of the window in the target process are changed.

An important aspect of the one-sided communication mechanism is **the synchronization of access to the window**. Since, unlike the standard two-sided "send–receive" exchange scheme, the target process does not perform special actions in one-sided exchanges, it is necessary to take additional efforts to *coordinate* access to the data stored in the window. In particular, if the target process plays the role of a data receiver (in this case, it is called *an active target process*), then it must know when it can access the window to read the received data, and if the data receiver is the origin process, then it must know when it can access the data received from the target process. In both cases, synchronization is required. The only exception, in which synchronization is not required on the side of the target process, is the variant of one-sided interaction with the so-called *passive target process*, in which the target process does not access its window at all. The window of the passive target process is used as *a data store* to which other processes of the parallel application access (this type of one-sided communication is closest to the shared memory model used in multithreaded programming).

Synchronization functions allow you to set so-called *access epochs* on the side of origin processes and *exposure epochs* on the side of active target processes. All results of one-sided interactions performed during the access epoch (and associated exposure epoch) will be available to the process *only when this epoch ends*. In other words, until the current access epoch has ended, the origin process should not access data obtained using MPI_Get functions, and until the exposure epoch has ended, the active target process should not access its window to read data passed to this window using MPI_Put or MPI_Accumulate functions.

The simplest synchronization option is provided by the function MPI_Win_fence(int assert, MPI_Win win). This is a collective function that must be called in all processes in which the window win is defined.

In addition to the win parameter, which defines the window for which the access epoch is set, this function includes the assert parameter, which may contain a set of constants specifying properties of the access epoch being defined (for example, the MPI_MODE_NOPUT constant means that no actions related to *changing* the window contents by the MPI_Put or MPI_Accumulate functions will be performed during this access epoch). Such constants allow the MPI environment to optimize actions performed during one-sided communications. If properties of the access epoch does not need to be specified, then the number 0 is specified as the assert parameter. Note that the assert parameter is also included in other synchronization functions (described below).

The first call to MPI_Win_fence begins the first access epoch (and associated exposure epoch) for the window. Each subsequent call to this function ends the

previous access epoch (and associated exposure epoch) and simultaneously begins a new access epoch (and associated new exposure epoch). Thus, when using this synchronization function, at least *two* calls to this function must be made in each process with a window. This type of synchronization is used when the same processes act as both origin processes and active target processes during an access epoch. Its limitation is that it is global: a single synchronization is established for all processes for which the window is defined.

Note 1. If the `MPI_Win_fence` function is called only to terminate the last access epoch, then this fact can be noted by specifying the special value `MPI_MODE_NOSUCCEED` as the `assert` parameter:

```
MPI_Win_fence(MPI_MODE_NOSUCCEED, w);
```

Another option for synchronization allows it to be done *locally*, defining an access epoch (and an associated exposure epoch) for only *some* of the processes in which the window is defined. This flexibility comes at the cost of a more complex way of configuring access epochs, in which special functions are provided for the beginning and end of both the access epoch and the exposure epoch:

- the function `MPI_Win_start(MPI_Group group, int assert, MPI_Win win)` function starts the access epoch for all processes in which it is called, and it specifies the group of processes `group` that can act as active target processes for this epoch;
- the function `MPI_Win_complete(MPI_Win win)` ends the access epoch started by the `MPI_Win_start` function;
- the function `MPI_Win_post(MPI_Group group, int assert, MPI_Win win)` begins an exposure epoch for all processes in which it is called, and it specifies the group of processes `group` that can act as origin processes for this epoch;
- the function `MPI_Win_wait(MPI_Win win)` ends the exposure epoch started by the `MPI_Win_post` function.

Exit from the `MPI_Win_wait` function means that all origin processes have completed the access epoch by calling the `MPI_Win_complete` function.

Note 2. There is also a non-blocking version of the `MPI_Win_wait` function, `MPI_Win_test`, which has an additional output parameter `flag`. The `MPI_Win_test(MPI_Win win, int *flag)` function returns a non-zero `flag` value if all origin processes have completed the access epoch, which in turn means that the exposure epoch has ended. The `MPI_Win_test` function should be called again within a current exposure epoch only if its previous call returned a zero `flag` value. In the tasks included in the PT for MPI-2 taskbook, the `MPI_Win_test` function is not used.

Both considered synchronization options assume that the target processes are active. For the one-sided communications, in which the target processes are passive, i. e. do not access their windows, a third synchronization option based

on *blocking* is provided. The main feature of the blocking synchronization option is that in this option the target process does not have to call any synchronization functions (and, therefore, no exposure epoch is specified for it). Special synchronization functions are intended only for the beginning and end of the access epoch in the origin processes. To begin the blocking access epoch, the origin process must call the `MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)` function, and to end the blocking access epoch, the `MPI_Win_unlock(int rank, MPI_Win win)` function is intended. The rank parameter determines the target process. The lock can be *exclusive* or *shared*; the type of lock is specified by the `lock_type` parameter, which can take the value of one of two constants: `MPI_LOCK_EXCLUSIVE` and `MPI_LOCK_SHARED`. If any process attempts to organize access with an exclusive lock to a target process for which a lock (of any type) is currently set by another process, then granting access is postponed until the previously set lock is released (the exclusive lock mode is usually used if several origin processes access the window of the target process for *writing* or *modifying*). Unlike an exclusive lock, several processes can simultaneously organize access with a shared lock to the same target process (the shared lock mode is used if the origin processes access the window of the target process only for *reading*).

The tasks of the `MPI7Win` group allow you to get acquainted with all aspects of one-sided communications. The first subgroup of this group (`MPI7Win1–MPI7Win17`, Section 2.7.1) uses the simplest synchronization based on the use of the collective function `MPI_Win_fence`; various options for using all three types of one-sided communications (with read, write, and modify access) are considered. In the initial tasks of this subgroup (`MPI7Win1–MPI7Win6`), shared memory within the access window is created only in one process, and in the remaining tasks, shared memory areas are created in groups of processes or in all processes of the application. In the second subgroup (see Section 2.7.2), more complex types of synchronization are studied: local synchronization based on the use of four functions `MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_post`, `MPI_Win_wait` (`MPI7Win18–MPI7Win23`), and blocking synchronization based on the use of the functions `MPI_Win_lock` and `MPI_Win_unlock` (`MPI7Win24–MPI7Win27`, `MPI7Win29`); in `MPI7Win28` and `MPI7Win30` tasks, it is necessary to use both synchronization options considered in the second subgroup.

To demonstrate the various features associated with one-sided communications, in the next two sections we will consider one task from each subgroup of the `MPI7Win` group.

1.3.5. One-sided communications: an example using the simplest synchronization option

Let us start with the MPI7Win13 task, which uses the simplest synchronization based on the `MPI_Win_fence` function, and at the same time uses the most complex of the access functions, `MPI_Accumulate`.

MPI7Win13. Three integers N_1, N_2, N_3 are given in each process; each given integer is in the range 0 to $K - 1$, where K is the number of processes (the values of some of these integers in each process may coincide). In addition, an array A of $R + 1$ real numbers is given in each process, where R is the process rank ($0, \dots, K - 1$). Create an access window containing the array A in all the processes. Using three calls of the `MPI_Accumulate` function in each process, add the integer $R + 1$ to all elements of the arrays A given in the processes N_1, N_2, N_3 , where R is the rank of the process that calls the `MPI_Accumulate` function (for instance, if the number N_i in the process 3 is equal to 2, then a real number 4.0 should be added to all the elements of array A in the process 2). If some of the integers N_1, N_2, N_3 coincide in the process R , then the number $R + 1$ should be added to the elements of the corresponding arrays several times. Output the changed arrays A in each process.

Here is the taskbook window when the program is launched with a template for this task (Fig. 30). To reduce the size of the window, the section with the task formulation is hidden.

```

ONE-SIDED COMMUNICATIONS WITH THE SIMPLEST SYNCHRONIZATION
Task: MPI7Win13
Student: Smith John

Acquaintance with the task;
input-output operations were not executed.

Input: 0 of 49          Output: 0 of 28          Tests: 1 of 5

Input data
Process 0: 4 6 2 5.92          Process 1: 1 3 4 1.55 1.00
Process 2: 4 3 4 2.25 4.82 8.59 Process 3: 2 6 3 3.64 7.00 1.68 7.46
Process 4: 4 0 4 7.58 9.14 2.08 3.34 3.29
Process 5: 1 4 0 6.62 5.50 4.03 7.42 4.95 8.15
Process 6: 2 1 4 2.01 1.87 8.69 2.37 2.66 4.57 6.11

Example of right solution
Process 0: 16.92          Process 1: 16.55 16.00
Process 2: 14.25 16.82 20.59 Process 3: 12.64 16.00 10.68 16.46
Process 4: 39.58 41.14 34.08 35.34 35.29
Process 5: 6.62 5.50 4.03 7.42 4.95 8.15
Process 6: 7.01 6.87 13.69 7.37 7.66 9.57 11.11

```

Fig. 30. Acquaintance running of the MPI7Win13 task

The transformation of initial array A required in the task is difficult (although possible) to implement using traditional MPI tools. The main problem is that the receiving process does not know from which source processes it will re-

ceive data that must be added to the elements of its array A . At the same time, using the one-sided communication mechanism, the required transformation is implemented quite simply. Since in this case all processes act as origin processes and, in addition, most processes are also target processes, the collective synchronization option is the most natural.

In the example shown in Fig. 30, only one process is not the target: this is process 5, for which the contents of array A will not change (this is due to the fact that among the integers given to the processes there is no number equal to 5). It should also be noted that in some cases the same process acts as both the origin and the target process (for example, process 1 must increase the elements of its array, since among the integers given to this process there is the number 1). In addition, for most processes, the arrays they contain will be modified by several origin processes (for example, the array from process 4 will be modified by processes 0, 1, 2, 4, 5, and 6, with processes 2 and 4 modifying this array twice; as a result, each element of array from process 4 will increased by the number $32 = 1 + 2 + 3 + 3 + 5 + 5 + 6 + 7$).

At the initial stage of the solution, we will declare all the necessary arrays and ensure that they are filled with the initial data. In addition to the array a given in the task formulation, we will declare an array n of three elements containing the given integers N_1 , N_2 and N_3 , as well as an array b , which will contain the values added by the origin process to the arrays a of target processes. As the size of the array b , it is sufficient to specify the maximum of the sizes of the arrays a (equal to the maximum rank plus 1, or, in other words, equal to $size$, the number of processes).

Here is the first part of the solution:

```
int n[3];
for (int i = 0; i < 3; i++)
    pt >> n[i];
double *a = new double[rank + 1];
for (int i = 0; i < rank + 1; i++)
    pt >> a[i];
double *b = new double[size];
for (int i = 0; i < size; i++)
    b[i] = rank + 1;
```

When we run this version of the program, we will receive a message that all the required initial data are input, but no results are output.

Now we need to define a window that contains the memory areas that other processes of the application can access. In our case, the window should include arrays a from each process. Let us give the corresponding code fragment and then comment on it:

```
int dbl_sz;
MPI_Type_size(MPI_DOUBLE, &dbl_sz);
MPI_Win win;
```



```
MPI_Win_create(a, (rank + 1) * dbl_sz, dbl_sz, MPI_INFO_NULL,  
MPI_COMM_WORLD, &win);
```

First we define, in the variable `dbl_sz`, the size of the `MPI_DOUBLE` type in bytes (using the debug output, we can verify that it is 8). Then we create a window `win` using the `MPI_Win_create` function. In Section 1.3.4, we noted that this function is collective and therefore must be called in all processes of the communicator for which the window is created (we create a window for the `MPI_COMM_WORLD` communicator, which is specified as the last but one parameter of this function).

The first parameter of the function specifies the address of the beginning of the memory area associated with the window in the given process; in our case, this is always the beginning of the array `a`. Then, the size of this memory area in bytes is specified (recall that it is permissible to specify a size equal to 0; this means that no memory area is associated with the window in the given process). As the third parameter of the `MPI_Win_create` function, we specify the size (in bytes) of the array elements stored in the window; this will allow us to specify *the index of the required array element* in the window access functions. The fourth parameter is assumed to be equal to the constant `MPI_INFO_NULL`. The last parameter `win` returns the window handle, which must be specified in all functions used when working with this window.

Recall that after finishing working with the access window, it must be destroyed by the `MPI_Win_free` function. In addition, in our program, it is necessary to free the memory allocated for the dynamic arrays `a` and `b`. Therefore, we add to the `Solve` function the following final statements:

```
MPI_Win_free(&win);  
delete[] a;  
delete[] b;
```

The result of running the new version of the program will not differ from the previous one. All subsequent additions to the solution must be specified *before* the fragment that leads to the destruction of the access window.

Now we need to organize access to the created window from different processes. Since such access is possible only within the access epoch, we should start such an epoch by calling the `MPI_Win_fence` function:

```
MPI_Win_fence(0, win);
```

Since the additional properties of this access epoch do not need to be specified, the number 0 is used as the first parameter.

Once the access epoch has started, we can call the window access functions. In this case, we need to call the `MPI_Accumulate` function three times in each process. For this we organize a loop:

```
for (int i = 0; i < 3; i++)  
    MPI_Accumulate(b, n[i] + 1, MPI_DOUBLE, n[i], 0, n[i] + 1,  
        MPI_DOUBLE, MPI_SUM, win);
```

Recall that the first three parameters of any window access function define the data "on the side" of the origin process: the address of the start of the data buffer, the number of elements in the buffer, and the type of the buffer elements. The next four parameters define the data "on the side" of the target process: the rank of the target process, the offset from the start of the window in this target process, and the number and type of window elements to be accessed. The last but one parameter defines the operation used to modify the window elements. Since in our case we need to add new terms to the original values of the window elements, we used the MPI_SUM operation.

In order to be able to access the modified elements of the window, the access epoch (and the associated exposure epoch on the target processes side) during which the window modification actions were performed must be completed. Therefore, before outputting the modified array `a`, the synchronization function `MPI_Win_fence` must be called one more time:

```
MPI_Win_fence(0, win);
for (int i = 0; i < rank + 1; i++)
    pt << a[i];
```

After running the latest version of the program, we will receive a message that the task has been solved.

Here is the full text of the solution:

```
void Solve ()
{
    Task("MPI7Win13");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int n[3];
    for (int i = 0; i < 3; i++)
        pt >> n[i];
    double *a = new double[rank + 1];
    for (int i = 0; i < rank + 1; i++)
        pt >> a[i];
    double *b = new double[size];
    for (int i = 0; i < size; i++)
        b[i] = rank + 1;
    int dbl_sz;
    MPI_Type_size(MPI_DOUBLE, &dbl_sz);
    MPI_Win win;
    MPI_Win_create(a, (rank + 1) * dbl_sz, dbl_sz, MPI_INFO_NULL,
        MPI_COMM_WORLD, &win);
    MPI_Win_fence(0, win);
```

```

    for (int i = 0; i < 3; i++)
        MPI_Accumulate(b, n[i] + 1, MPI_DOUBLE, n[i], 0, n[i] + 1,
            MPI_DOUBLE, MPI_SUM, win);
    MPI_Win_fence(0, win);
    for (int i = 0; i < rank + 1; i++)
        pt << a[i];
    MPI_Win_free(&win);
    delete[] a;
    delete[] b;
}

```

Note: Both calls to `MPI_Win_fence` in the above program are *mandatory*. If you comment out at least one of them, the program will terminate without reporting any MPI errors, but the results will be different from what is required.

1.3.6. One-sided communications: an example of a more complex version of synchronization

Now let us turn to the `MPI7Win23` task from the second subgroup, which requires using a different synchronization method.

MPI7Win23. An array A of 5 real numbers is given in each process. In addition, two arrays N and M of 5 integers are given in the master process. All the elements of the array N are in the range 1 to K , where K is the number of slave processes, all the elements of the array M are in the range 0 to 4. Some elements of both the array N and the array M may have the same value. Create an access window containing the array A in each slave process. Using the required number of calls of the `MPI_Get` function in the master process, receive the element of A with the index M_I from the process N_I ($I = 0, \dots, 4$) and add the received element to the element A_I in the master process. After changing the array A in the master process, change all the arrays A in the slave processes as follows: if some element of the array A from the slave process is greater than the element, with the same index, of the array A from the master process, then replace this element in the slave process by the corresponding element from the master process (to do this, use the required number of calls of the `MPI_Accumulate` function in the master process). Output the changed arrays A in each process. Use two calls of the `MPI_Win_post` and `MPI_Win_wait` synchronization functions in the slave processes and two calls of the `MPI_Win_start` and `MPI_Win_complete` synchronization functions in the master process.

Fig. 31 shows the taskbook window for this task. As with the previous task, the window hides a section with the task formulation.

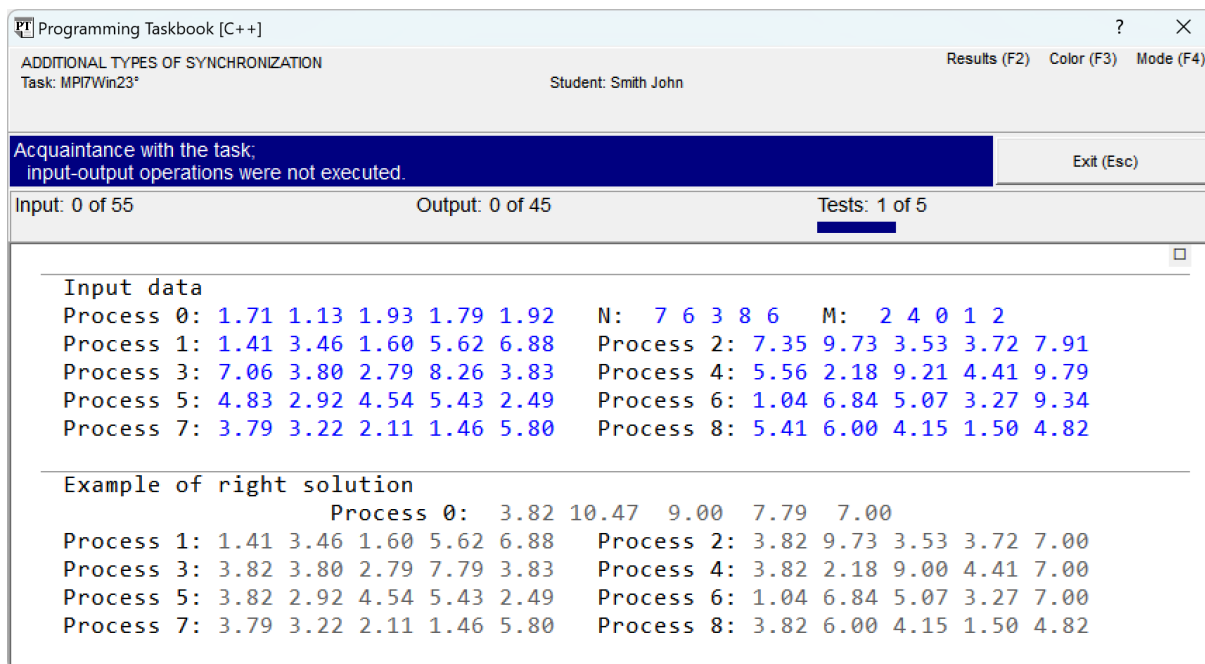


Fig. 31. Acquaintance running of the MPI7Win23 task

In this case, it is necessary to create a window to access arrays A in slave processes; thus, the master process acts as the origin process, and the slave processes act as the target processes. In such a situation, it makes sense to use synchronization that takes into account the specified features of exchange operations.

Another feature of this task is that it requires two series of sequentially executed one-sided exchanges. First, it is necessary to modify array A in the master process. Since this process acts as the origin process, the `MPI_Get` function should be used to perform this action. Then, the modified array A from the master process should be used to change some elements of array A in the slave processes. Since the master process still acts as the origin process, the `MPI_Accumulate` function should be used in this case. Since it is necessary to start changing the arrays in the slave processes only after the array in the master process has been modified, the program must use *two access epochs*: in the first epoch, the array in the master process is modified, and in the second epoch, the arrays in the slave processes are changed.

At the initial stage, we organize the input of all initial data and the definition of the access window:

```
int win_sz = 5;
int n[5], m[5];
double a[5];
for (int i = 0; i < 5; i++)
    pt >> a[i];
if (rank == 0)
{
```

```
win_sz = 0;
for (int i = 0; i < 5; i++)
    pt >> n[i];
for (int i = 0; i < 5; i++)
    pt >> m[i];
}
int dbl_sz;
MPI_Type_size(MPI_DOUBLE, &dbl_sz);
MPI_Win win;
MPI_Win_create(a, win_sz * dbl_sz, dbl_sz, MPI_INFO_NULL,
    MPI_COMM_WORLD, &win);
// Perform one-sided exchanges and output the result
MPI_Win_free(&win);
```

Running this version of the program will result in a message stating that all the initial data are input, but no results are output.

It should be noted that some initial data (arrays *n* and *m*) are input only in the master process, and also that a shared memory area is not created in the master process (the second parameter size of the `MPI_Win_create` function in the master window has the value 0).

The rest of the solution should be placed in the position marked with the comment *"Perform one-sided exchanges and output the result"*.

In the synchronization functions `MPI_Win_start` and `MPI_Win_post`, which are to be used in this task, it is necessary to specify the target process group and the origin process group, respectively. Such groups are most conveniently obtained from the process group associated with the communicator `MPI_COMM_WORLD`:

```
MPI_Group g0, g;
MPI_Comm_group(MPI_COMM_WORLD, &g0);
```

Given a group *g0* of all processes, we can simply *remove* the process of rank 0 from the group *g0* to obtain a group of target processes, and take the first element of the group *g0* (i. e., the process of rank 0) to obtain a group of origin processes.

Thus, to implement the first access epoch in the master process (and the associated exposure epoch in the slave processes), it is sufficient to perform the following actions:

```
int b = 0;
if (rank == 0)
{
    MPI_Group_excl(g0, 1, &b, &g);
    MPI_Win_start(g, 0, win);
    // Call to access functions
    MPI_Win_complete(win);
    // Output of results
}
else
```

```

{
    MPI_Group_incl(g0, 1, &b, &g);
    MPI_Win_post(g, 0, win);
    MPI_Win_wait(win);
}

```

Let us recall that the first parameter of the `MPI_Win_start` and `MPI_Win_post` functions is a group of processes, the second is the assert parameter, which has the same meaning as the parameter of the `MPI_Win_fence` function of the same name (it is enough to set it equal to 0), the third parameter determines the access window used. The `MPI_Win_complete` and `MPI_Win_wait` functions, which complete the current access epoch, are simpler: they only specify the access window.

Between the calls to the `MPI_Win_start` and `MPI_Win_complete` functions, it is possible to call access functions, in our case, the `MPI_Get` functions, which allow us to obtain elements from slave processes that should be added to the elements of the array `a` of the master process. To store the data obtained from the slave processes, we allocate an auxiliary buffer of real numbers `a0` of size 5 (the buffer size corresponds to the number of numbers obtained from the slave processes). The contents of the buffer can be accessed only after the end of the access epoch (i. e., after calling the `MPI_Win_complete` function). Thus, to obtain data from the slave processes, it is sufficient to add the following fragment *before* the `MPI_Win_complete(win)` call (in the position marked with the comment "*Call to access functions*"):

```

double a0[5];
for (int i = 0; i < 5; i++)
    MPI_Get(&a0[i], 1, MPI_DOUBLE, n[i], m[i], 1, MPI_DOUBLE,
           win);

```

In this fragment, the elements of the auxiliary array `a0` are filled in a loop: the value of the element with index `i` is assumed to be equal to the value of the element of array `a` with index `m[i]` located in the process of rank `n[i]`.

After the `MPI_Win_complete(win)` call (in the position marked with the comment "*Output of results*"), it is necessary to add a fragment that ensures the modification and output of the array `a` in the master process:

```

for (int i = 0; i < 5; i++)
    a[i] += a0[i];
for (int i = 0; i < 5; i++)
    pt << a[i];

```

In this fragment, the obtained elements of array `a0` are added to the corresponding elements of array `a` of the master process, after which the modified array is output.

Note that between the calls to the `MPI_Win_post` and `MPI_Win_wait` functions we did not need to perform any actions in the slave processes.

When running this version of the program, a message will appear in the taskbook window stating that the resulting data has not been output in the slave

processes, but the contents of array *a* in the master process will be correct (Fig. 32).

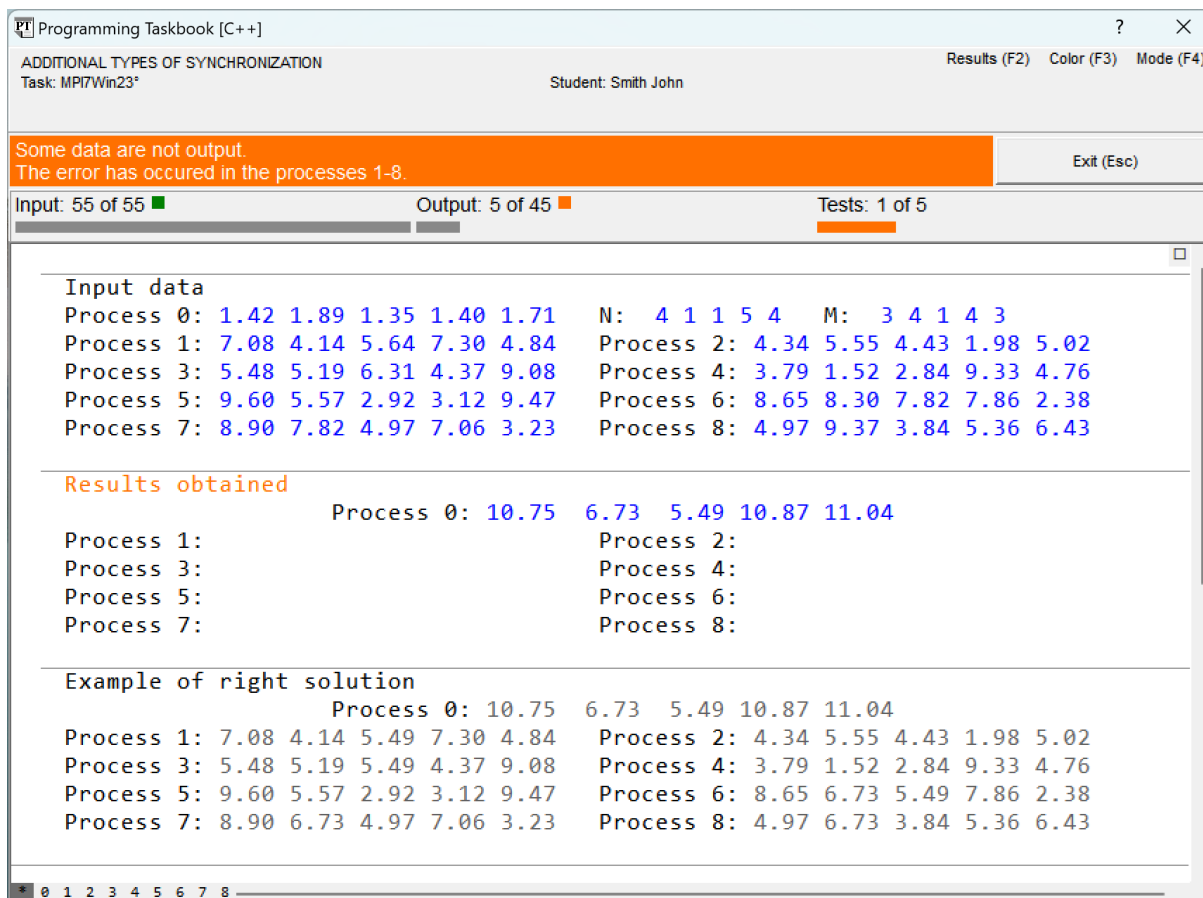


Fig. 32. The first stage of solving the MPI7Win23 task

The second part of the one-sided exchanges remains to be performed: adding the modified contents of array *a* from the master process to all arrays in the slave processes. To do this, a new access epoch must be started in the master process and an associated exposure epoch must be started in the slave processes.

Within the new access epoch, the master process must execute the `MPI_Accumulate` function, transferring data from array *a* of the master process to the windows of all slave processes:

```
MPI_Win_start(g, 0, win);
for (int i = 1; i < size; i++)
    MPI_Accumulate(a, 5, MPI_DOUBLE, i, 0, 5, MPI_DOUBLE,
        MPI_MIN, win);
MPI_Win_complete(win);
```

In accordance with the task formulation, we use the `MPI_MIN` operation in the `MPI_Accumulate` function.

Within the new exposure epoch (in slave processes), as for the first exposure epoch, no action needs to be performed, but after this epoch ends, we can output the changed contents of the array *a*:

```
MPI_Win_post(g, 0, win);
MPI_Win_wait(win);
for (int i = 0; i < 5; i++)
    pt << a[i];
```

By running this version of the program, we will receive a message that the task has been solved.

The output of results in the master process and in the slave processes can be combined by taking the corresponding loop out of the two parts of the conditional statement and placing it immediately before the window destruction statement `MPI_Win_free(&win)`.

Here is the final version of the solution:

```
void Solve ()
{
    Task("MPI7Win23");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int win_sz = 5;
    int n[5], m[5];
    double a[5];
    for (int i = 0; i < 5; i++)
        pt >> a[i];
    if (rank == 0)
    {
        win_sz = 0;
        for (int i = 0; i < 5; i++)
            pt >> n[i];
        for (int i = 0; i < 5; i++)
            pt >> m[i];
    }
    int dbl_sz;
    MPI_Type_size(MPI_DOUBLE, &dbl_sz);
    MPI_Win win;
    MPI_Win_create(a, win_sz * dbl_sz, dbl_sz, MPI_INFO_NULL,
        MPI_COMM_WORLD, &win);
    MPI_Group g0, g;
    MPI_Comm_group(MPI_COMM_WORLD, &g0);
    int b = 0;
    if (rank == 0)
    {
        MPI_Group_excl(g0, 1, &b, &g);
        MPI_Win_start(g, 0, win);
        double a0[5];
```



```

        for (int i = 0; i < 5; i++)
            MPI_Get(&a0[i], 1, MPI_DOUBLE, n[i], m[i], 1,
                  MPI_DOUBLE, win);
        MPI_Win_complete(win);
        for (int i = 0; i < 5; i++)
            a[i] += a0[i];
        MPI_Win_start(g, 0, win);
        for (int i = 1; i < size; i++)
            MPI_Accumulate(a, 5, MPI_DOUBLE, i, 0, 5, MPI_DOUBLE,
                          MPI_MIN, win);
        MPI_Win_complete(win);
    }
    else
    {
        MPI_Group_incl(g0, 1, &b, &g);
        MPI_Win_post(g, 0, win);
        MPI_Win_wait(win);
        MPI_Win_post(g, 0, win);
        MPI_Win_wait(win);
    }
    for (int i = 0; i < 5; i++)
        pt << a[i];
    MPI_Win_free(&win);
}

```

1.3.7. Inter-communicators

Inter-communicators (or *intercommunicators*) appeared already in the MPI-1 standard. Unlike the "regular" communicator, also called the *intra-communicator* (or *intracommunicator*), which connects to a certain group of processes and provides various types of interaction between any processes included in this group, the inter-communicator connects to *two* groups of processes and is intended to provide interaction between processes from *different* groups; in this case, the ranks of the processes in these groups are used. This type of interaction is convenient if the parallel algorithm assumes the distribution of actions between several groups of processes and at the same time requires the exchange of information between processes included in different groups.

In the MPI-2 standard, the concept of inter-communicators was further developed: the possibilities for *creating inter-communicators* were expanded, *collective interactions* of processes within inter-communicators became possible, and, finally, inter-communicators became the tool that was used as the basis for the mechanism for *dynamic process creation*.

Any inter-communicator is connected to two groups of processes. A process from any group can initiate data exchange with a process from another group within the inter-communicator that connects these groups. In this case, the group, to which the process that calls the function to send or receive a message

belongs, is called *the local group*, and the group, containing the processes with which the connection is established, is called *the remote group*. Thus, for the sending process, the remote group is the one in which the receiving process is located, and for the receiving process, the remote group is the one in which the sending process is located.

The function `MPI_Comm_size(MPI_Comm comm, int *size)` to determine the number size of processes included in the communicator `comm`, can also be used for the inter-communicator; in this case, it returns the number of processes of the *local group*, i. e. the inter-communicator group to which the process that called this function belongs. Similarly, for an inter-communicator, the function `MPI_Comm_rank(MPI_Comm comm, int *rank)` returns the rank `rank` of the process in the local group. There is also an additional function `MPI_Comm_remote_size(MPI_Comm comm, int *size)`, which is available only to inter-communicators; it returns the size of the remote group, i. e. the group of the inter-communicator `comm` to which the process calling this function *does not belong*.

The function `MPI_Comm_group(MPI_Comm comm, MPI_Group *group)`, in the case of an inter-communicator `comm`, returns its local group `group`, and to get a remote group, the function `MPI_Comm_remote_group` is provided with the same set of parameters.

You can check whether the communicator `comm` is an inter-communicator using the function `MPI_Comm_test_inter(MPI_Comm comm, int *flag)`, which returns a non-zero value of the flag parameter for inter-communicators and a zero value for intracommunicators.

To get acquainted with the basic method of creating an inter-communicator and the simplest techniques for organizing interaction between its groups, let us consider the following task.

MPI8Inter9. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the `MPI_Comm_split` function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer -1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. Input an integer X in the processes of the first group, input an integer Y in the processes of the second group. Using the required number of calls of the `MPI_Send` and `MPI_Recv` functions for all the processes of the inter-communicator, send all the integers X to each process of the second group and send all the integers Y to each process of the

first group. Output all received numbers in ascending order of ranks of sending processes.

When you run the program with a template for a given task, a window similar to the one shown in Fig. 33 will appear on the screen (this window hides a section with the task formulation).

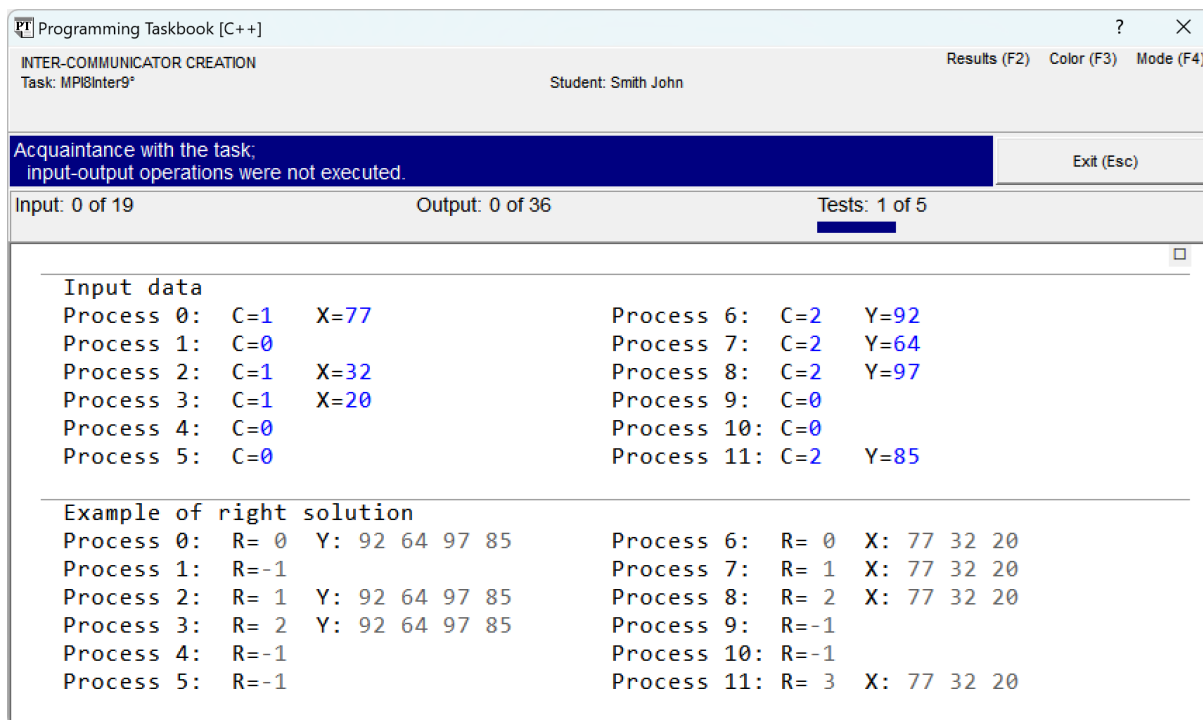


Fig. 33. Acquaintance running of the MPI8Inter9 task

At the first stage of solving the task, it is necessary to create two new communicators containing processes with identical non-zero values of C . An indication of successful completion of this stage will be the output of the correct values of ranks R of processes in the new communicators.

Since at this stage there is no need to use the new MPI library tools, we will immediately provide the corresponding program fragment (the features of using the `MPI_Comm_split` function were previously discussed in detail in Section 1.2.7 devoted to creating new communicators):

```
int c;
pt >> c;
if (c == 0)
    c = MPI_UNDEFINED;
MPI_Comm local;
MPI_Comm_split(MPI_COMM_WORLD, c, rank, &local);
if (local == MPI_COMM_NULL)
{
    pt << -1;
    return;
}
int local_rank;
```

```
MPI_Comm_rank(local, &local_rank);
pt << local_rank;
```

After input of the number C , we immediately correct its value if it is 0: the zero value is replaced by `MPI_UNDEFINED`, so as not to create a communicator for processes with $C = 0$. The new communicator containing the current process is associated with the variable named `local` (the variable name indicates that the group of this communicator will later become the *local group* of the inter-communicator). If a new communicator is not associated with the process, then the value -1 is output in it and the program exits; otherwise, the rank `local_rank` of the process in the new communicator is obtained and output.

When the program is launched, a message will be displayed stating that in some processes not all the given data has been input (since the input of numbers X and Y has not yet been performed in our program), however, the values R will be found correctly for all processes (Fig. 34).

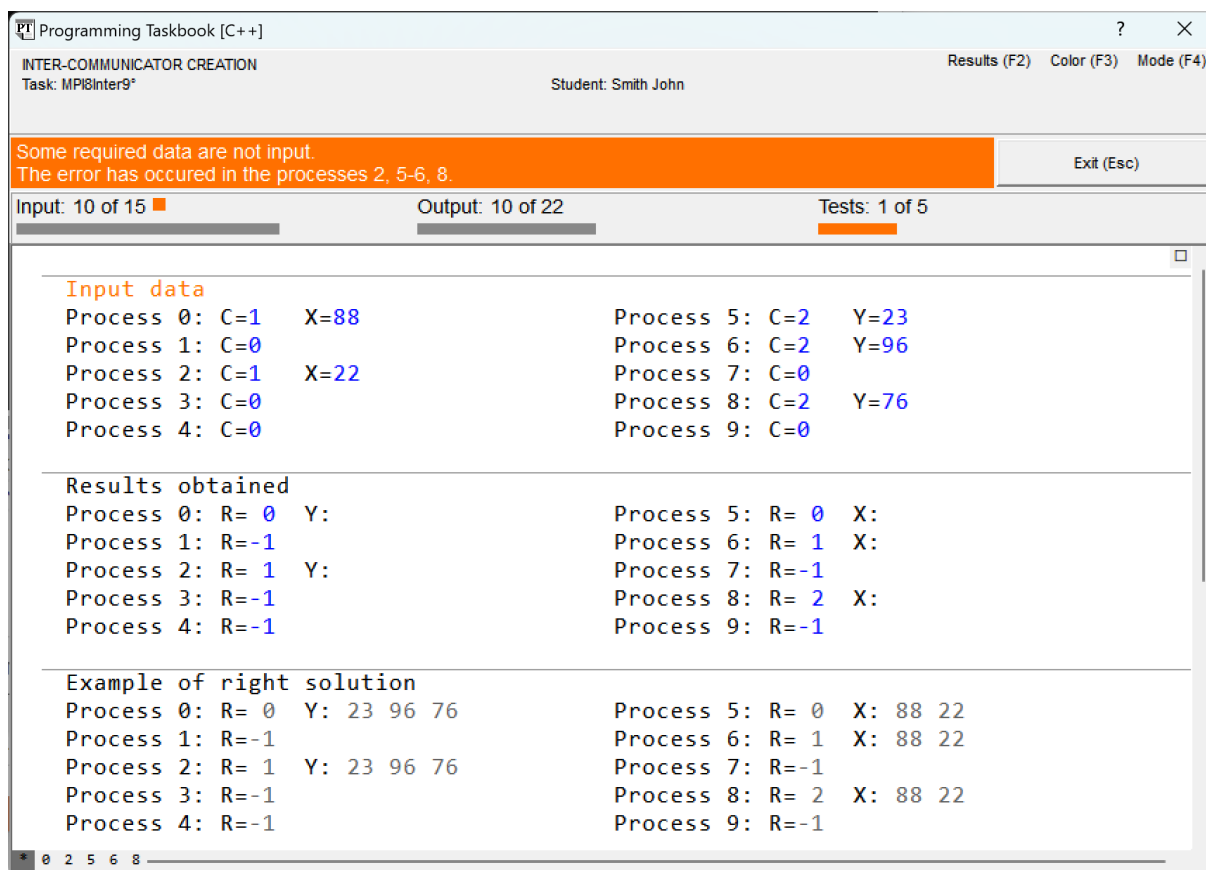


Fig. 34. The first stage of solving the MPI8Inter9 task

Let us proceed to the main stage of the solution: defining an inter-communicator that contains both previously created groups of processes. The main tool for creating inter-communicators is the `MPI_Intercomm_create` function. This is a collective function that must be called in all processes that need to be included in the inter-communicator being created. In other words, it must be

called in all processes of those two "regular" communicators (intra-communicators) that contain groups of processes included in the inter-communicator.

The main problem in creating an inter-communicator is defining a remote group. As a local group, it is sufficient to specify the corresponding communicator, which includes the process calling the `MPI_Intercomm_create` function. However, a communicator created for processes of another (remote) group is not available in this process. The problem is solved by specifying the "representative" processes (*leaders*) of each of the two groups. In this case, it is necessary that both selected leaders are included in some common communicator, a *peer*. A natural candidate for the role of a peer is the universal communicator `MPI_COMM_WORLD`, however, in order to avoid possible conflicts when sending data, it is desirable to use a *copy* of the communicator `MPI_COMM_WORLD`, by means of the `MPI_Comm_dup` function.

Let us list the parameters of the `MPI_Intercomm_create` function:

`MPI_Comm local` – a communicator associated with the local group of the created inter-communicator;

`int local_leader` – rank of the local group leader (in the local communicator);

`MPI_Comm peer_comm` – a peer communicator; this parameter is taken into account only in the process that is a leader of the local group;

`int remote_leader` – the rank of the leader of the remote group (in the `peer_comm` communicator); this parameter, like the previous one, is taken into account only in the process that is a leader of the local group;

`int tag` – an integer "security tag" that must be the same in all processes that calls the `MPI_Intercomm_create` function to create this inter-communicator (other values for this tag should be used when creating other inter-communicators);

`MPI_Comm *intercomm` – pointer to the created inter-communicator (output parameter).

Thus, to create an inter-communicator, a leader of each group must know the rank of the leader of the other group in the peer communicator. In our case, we can use the following part of the task formulation: "*The first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$* " (K denotes the total number of processes). This means that the first of the groups we created (for processes with $C = 1$) is guaranteed to include a process of rank 0 of the communicator `MPI_COMM_WORLD`, and this process is *the first process* of the created group (i. e., it has rank 0 in the corresponding communicator), and the second of the groups we created (for processes with $C = 2$) is guaranteed to include a process of rank $K/2$ of the communicator `MPI_COMM_WORLD`, and this process is also *the first process* of the created group.

Therefore, when calling the `MPI_Intercomm_create` function in all processes, we can set the `local_leader` parameter to 0. As for the `remote_leader` parameter, its

value can be determined by the rank of the calling process in the MPI_COMM_WORLD communicator: if this rank is 0 (this means that the process is a leader of *the first group*), then the remote_leader parameter should be set to $K/2$, and if the rank of the calling process in the MPI_COMM_WORLD communicator is $K/2$, then the remote_leader parameter should be set to 0 (in the remaining processes, the value of the remote_leader parameter can be arbitrary, for example, equal to 0). Another way to do this is to analyze not the rank of the process in the MPI_COMM_WORLD communicator, but the value of C ; in this case, for processes with $C = 1$, the remote_leader parameter should be set equal to $K/2$, and for the remaining processes (with $C = 2$), it should be set equal to 0.

To make sure that the required inter-communicator has been created correctly, you can use the following simple test: determine the remote group size for each inter-communicator process by calling the MPI_Comm_remote_size function (and displaying this size in the debug section). Note that we will need this size later when organizing data transfers.

When implementing the second stage of solving the task, it is easy to make a serious mistake by trying to create a copy of the MPI_COMM_WORLD communicator *after* some processes exit the Solve function. This will inevitably lead to a hang of the parallel application. It is necessary to define an auxiliary peer communicator *before* the conditional statement in which the return statement is executed (for example, at the very beginning of the solution).

Here is an expanded version of the solution, in which new fragments are highlighted in bold:

```

MPI_Comm peer;
MPI_Comm_dup(MPI_COMM_WORLD, &peer);
int c;
MPI_Comm local;
pt >> c;
if (c == 0)
    c = MPI_UNDEFINED;
MPI_Comm_split(MPI_COMM_WORLD, c, rank, &local);
if (local == MPI_COMM_NULL)
{
    pt << -1;
    return;
}
int local_rank;
MPI_Comm_rank(local, &local_rank);
pt << local_rank;
MPI_Comm inter;
int lead = 0;
if (rank == 0)
    lead = size / 2;
MPI_Intercomm_create(local, 0, peer, lead, 100, &inter);
int remote_size;

```

```
MPI_Comm_remote_size(inter, &remote_size);
Show(remote_size);
```

When running this solution, the message in the information section will not change, but in the debug section, along with error messages, for each process with a non-zero C value, the correct size of the corresponding remote group in the created communicator will be displayed (see Fig. 35; in this case, for processes with $C = 1$, the size of the remote group is 2, and for processes with $C = 2$, the size is 1).

```
0| 1> 2
0| 2> !If there are errors in the slave processes,
0| 3> ! then the solution for the master process is not checked.
4| 1> 1
4| 2> !Some required data are not input.
4| 3> ! The program has input 1 data item(s) (of 2).
7| 1> 1
7| 2> !Some required data are not input.
7| 3> ! The program has input 1 data item(s) (of 2).
```

Fig. 35. The second stage of solving the MPI8Inter9 task

The final stage of the task solution is the simplest, since it requires using the well-known `MPI_Send` and `MPI_Recv` functions to exchange messages between two processes. The only feature is that, in this case, these functions are called for the inter-communicator, and therefore the rank of the receiving process (in the `MPI_Send` function) and the rank of the sending process (in the `MPI_Recv` function) should be specified as the rank of the process in *the remote group*:

```
MPI_Status s;
```

```
int a, b;
pt >> a;
for (int i = 0; i < remote_size; i++)
{
    MPI_Send(&a, 1, MPI_INT, i, 0, inter);
    MPI_Recv(&b, 1, MPI_INT, i, 0, inter, &s);
    pt << b;
}
```

Note that the number of iterations in the loop is equal to the size of the remote group.

After running this version of the program, we will receive a message that the task has been solved.

Note 1. The solution uses only those MPI library tools that are already available in the MPI-1 standard, so this program will also work successfully under the control of the MPICH system. As noted in the preamble to the MPI8Inter task group, five tasks in this group (numbers 1–4 and 9) can be executed in the MPI-1 version.

The MPI-2 standard introduced a number of new features related to **the creation of inter-communicators** (see tasks MPI8Inter5–MPI8Inter8). In this standard, the functions `MPI_Comm_create` and `MPI_Comm_split` can be used not only to create new intra-communicators based on existing ones, but also to create new inter-communicators (also based on existing ones).

If the function `MPI_Comm_create` (`MPI_Comm comm`, `MPI_Group group`, `MPI_Comm *newcomm`) is called for processes of the inter-communicator `comm`, then in the processes of each group of the inter-communicator `comm` it is necessary to specify the same parameter group, which determines the subset of processes from this group, which will become the corresponding group of the new inter-communicator `newcomm`. This function must be called in all processes included in the communicator `comm`; in this case, for those processes of each group that are not included in the specified group, the value `MPI_COMM_NULL` will be returned in the parameter `newcomm`.

Flexible capabilities for creating a whole *family* of new inter-communicators with pairwise disjoint groups of processes are provided by the function `MPI_Comm_split` (`MPI_Comm comm`, `int color`, `int key`, `MPI_Comm *newcomm`) function. When it is called for the inter-communicator `comm`, a set of new inter-communicators is created, the groups of which include the processes of the original inter-communicator with the same values of the color parameter (the key parameter, as usual, is used to determine the order of processes in the created groups; see the description of this parameter for this function in Section 1.2.7). If one of the groups of the inter-communicator `comm` does not contain processes with the color value specified in any processes of another group, then `MPI_COMM_NULL` is returned in the processes with this color value (this is due to the fact that the inter-communicator must contain two *non-empty* groups of processes). As usual, the

MPI_COMM_NULL value is also returned for those processes in which the color parameter has the value MPI_UNDEFINED.

Note 2. It should be noted that in the MPICH2 system, the MPI_Comm_split function behaves incorrectly in some special situations related to the creation of new inter-communicators. These situations are described in detail in the note to the MPI8Inter7 task (see Section 2.8.1). In the MS-MPI system, the MPI_Comm_split function behaves correctly in all special situations.

Another important innovation of the MPI-2 standard is the possibility of **using collective operations for inter-communicators**. The tasks MPI8Inter10–MPI8Inter15 are devoted to these possibilities. When performing collective exchanges, the same functions are used for both intra-communicators and inter-communicators (see Sections 1.2.4 and 1.2.5); it should only be taken into account that collective exchanges are always performed between *different groups* of the inter-communicator (in other words, the processes of one group exchange data only with the processes of another group).

As an example of applying the collective operation to the inter-communicator we can show the second variant of the final fragment of the MPI8Inter9 task solution, in which, instead of multiple calls of the MPI_Send and MPI_Recv functions in the loop, we use one collective MPI_Allgather function call in each process:

```
int a;
pt >> a;
int *b = new int[remote_size];
MPI_Allgather(&a, 1, MPI_INT, b, 1, MPI_INT, inter);
for (int i = 0; i < remote_size; i++)
    pt << b[i];
delete[] b;
```

Note that in the case of inter-communicators, collective functions without the root parameter act "bidirectionally" by sending data from processes of each inter-communicator group to processes of the other group.

Note 3. In the case where collective functions for inter-communicators use a special process specified by the root parameter (such as the MPI_Bcast, MPI_Scatter, and MPI_Gather functions), special rules apply. Recall that for intra-communicators, the root parameter defines the rank of the special process and must have the same value in all processes. For inter-communicators, the rank of the special process is specified only in the processes of the group that *does not contain* the special process. In the processes of the group that contains the special process, one of two predefined values must be specified as the root parameter: in the special process, the root parameter must have the MPI_ROOT value, and in the other processes of this group, the root parameter must have the MPI_PROC_NULL value. Ex-

amples of using such collective functions will be given in the next section when solving the MPI8Inter15 task.

1.3.8. Dynamic process creation

The inter-communicators discussed in the previous section are also used for *dynamically creating new processes* during the execution of a parallel application. The ability to create new processes appeared in the MPI-2 standard. Such an ability allows implementing parallel algorithms for which the number of processes can be increased while the application is running. In addition, inclusion of this feature into the MPI standard simplifies the transition to MPI technologies for those developers who previously used other parallel technologies allowing to generate processes dynamically (for example, Parallel Virtual Machine, PVM).

There are two functions for creating new processes: `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`. The first of these functions allows you to create the required number of new processes by running *the same executable file with the same command line parameters* (thus, it acts similarly to the MPI environment, which performs the initial launch of a parallel application). The second function is more flexible: it allows you to use *different sets of command line parameters* and even *different executable files* for different processes in the created group.

The `MPI_Comm_spawn` function has the following set of parameters:

`char *command` – a string defining the file to be launched (this parameter is only taken into account in the root process; see below for a description of the root parameter);

`char **argv` – a string array containing command line parameters (this parameter is also taken into account only in the root process); if parameters are not required, then it is sufficient to specify a null pointer (NULL or `nullptr`) as `argv`;

`int maxproc` – maximum number of launched processes (the parameter is taken into account only in the root process);

`MPI_Info info` – additional information related to the file being launched (the parameter is taken into account only in the root process); if additional information is not used, then it is sufficient to specify the constant `MPI_INFO_NULL` as `info`;

`int root` – the rank of the process from the parent communicator `comm`, in which the values of the four previous parameters must be specified;

`MPI_Comm comm` – parent communicator that provides creation of a new process group;

`MPI_Comm *intercomm` – a pointer to the resulting inter-communicator (output parameter); one of its groups is the group of all processes of the parent communicator `comm`, and the other is the group of all created processes;

int **array_of_errcodes* – an integer array containing error codes associated with each of the created processes (output parameter). If all processes are created successfully, then all elements of the array are equal to `MPI_SUCCESS`. If the program does not use error codes, the constant `MPI_ERRCODES_IGNORE` can be specified as this parameter.

The `MPI_Comm_spawn` function is collective; it must be called by all processes of the parent communicator `comm` (however, the values of the parameters defining the properties of the created processes need only be specified in one process of this communicator, which has the rank root). Successful exit from this function occurs only if all the required new processes are launched and each new process calls the `MPI_Init` function, which initializes the parallel mode. The exit from all `MPI_Init` functions in the new processes will occur simultaneously with the exit from the `MPI_Comm_spawn` functions in the parent processes.

The `MPI_Comm_spawn_multiple` function differs from the `MPI_Comm_spawn` function only in that all the settings for the parameters of the processes being launched are specified in *arrays*, each of which has a size equal to `count` (the first parameter of the function). There is a parameter list for the function `MPI_Comm_spawn_multiple`: (int `count`, char ***array_of_commands*, char ****array_of_argv*, int **array_of_maxprocs*, `MPI_Info` **array_of_info*, int `root`, `MPI_Comm` `comm`, `MPI_Comm` **intercomm*, int **array_of_errcodes*).

All features of the `MPI_Comm_spawn` function that are described below also apply to the `MPI_Comm_spawn_multiple` function. In the tasks included in the PT for MPI-2 taskbook, the `MPI_Comm_spawn_multiple` function is not used.

By default, call to the `MPI_Comm_spawn` function is considered successful if all `maxproc` processes are launched. It is possible to change this behavior by specifying additional settings in the `info` parameter; in this case, for the success of the `MPI_Comm_spawn` function, it is enough to launch fewer processes than specified in the `maxproc` parameter (we will not discuss this option in more detail).

Communication between the initial (parent) and new (child) processes is established using a new *inter-communicator* `intercomm`, returned by the `MPI_Comm_spawn` function.

However, as a result of calling the `MPI_Comm_spawn` function, this inter-communicator will be accessible only to parent processes. Child processes access this inter-communicator using the special function `MPI_Comm_get_parent`(`MPI_Comm` **parent*). If this function is called by one of the child processes, i. e. processes created *after* the parallel application has been launched, then its parameter `parent` returns a descriptor of the inter-communicator that connects the parent and child processes. If the `MPI_Comm_get_parent` function is called by one of the initial processes of the parallel application, then the constant `MPI_COMM_NULL` is returned in the `parent` parameter.

Since the same executable file is often used to start both the initial and new (child) processes, it is the `MPI_Comm_get_parent` function that allows you to rec-

ognize the initial processes and thus ensure that different code fragments are executed for the initial and new processes.

For new processes, there is also a communicator `MPI_COMM_WORLD` defined, which includes all processes created during the call of the `MPI_Comm_spawn` function. We can think of the inter-communicator returned by the `MPI_Comm_spawn` function as combining two "regular" intra-communicators `MPI_COMM_WORLD`, one containing all parent processes, and the other containing all child processes.

If two calls to `MPI_Comm_spawn` are made in the parent communicator, then two new process groups will be created in the parallel application, each of which will be connected to the parent group via its own inter-communicator. Another scheme is possible in which the child process group (`child1`) itself calls `MPI_Comm_spawn`; as a result, a new process group (`child2`) is created, for which the parent group will be `child1`.

Thus, to create new processes and ensure their subsequent interaction with parent processes, it is sufficient to use *two* MPI functions (`MPI_Comm_spawn` and `MPI_Comm_get_parent`) and to apply the data transfer functions that are provided for *inter-communicators*.

As an illustration of the described possibilities, let us consider the first task from the subgroup of the `MPI8Inter` group, which is devoted to the creation of new processes (see Section 2.8.3).

MPI8Inter15. A real number is given in each process. Using the `MPI_Comm_spawn` function with the first parameter "ptprj.exe", create one new process. Using the `MPI_Reduce` collective function, send the sum of the given numbers to the new process. Output the received sum in the debug section using the `Show` function in the new process. Then, using the `MPI_Bcast` collective function, send this sum to the initial processes and output it in each process.

When we launch the template program for this task, we will see a window similar to the one shown in Fig. 36.

Note the sample content of the debug section given in the input data section. The process rank specified there contains, in addition to the numeric value "0", the prefix "a". This prefix means that this process was created during the execution of the parallel application (if *several* groups of new processes are created in the parallel application, then different letter prefixes are associated with them in the debug section: "a", "b", "c", etc.).

Let us start to solve the task. First of all, we need to create a new process. In all tasks of this subgroup, the standard communicator `MPI_COMM_WORLD` should be used as the parent communicator. In addition, in all tasks, we need to specify the same name of the executable file "ptprj.exe", since any project created by the `PT4Load` program has such a name. The number of processes to be created is defined in the task formulation, the values of other settings for new

process groups are defined in the preamble to the MPI8Inter group (see Section 2.8).

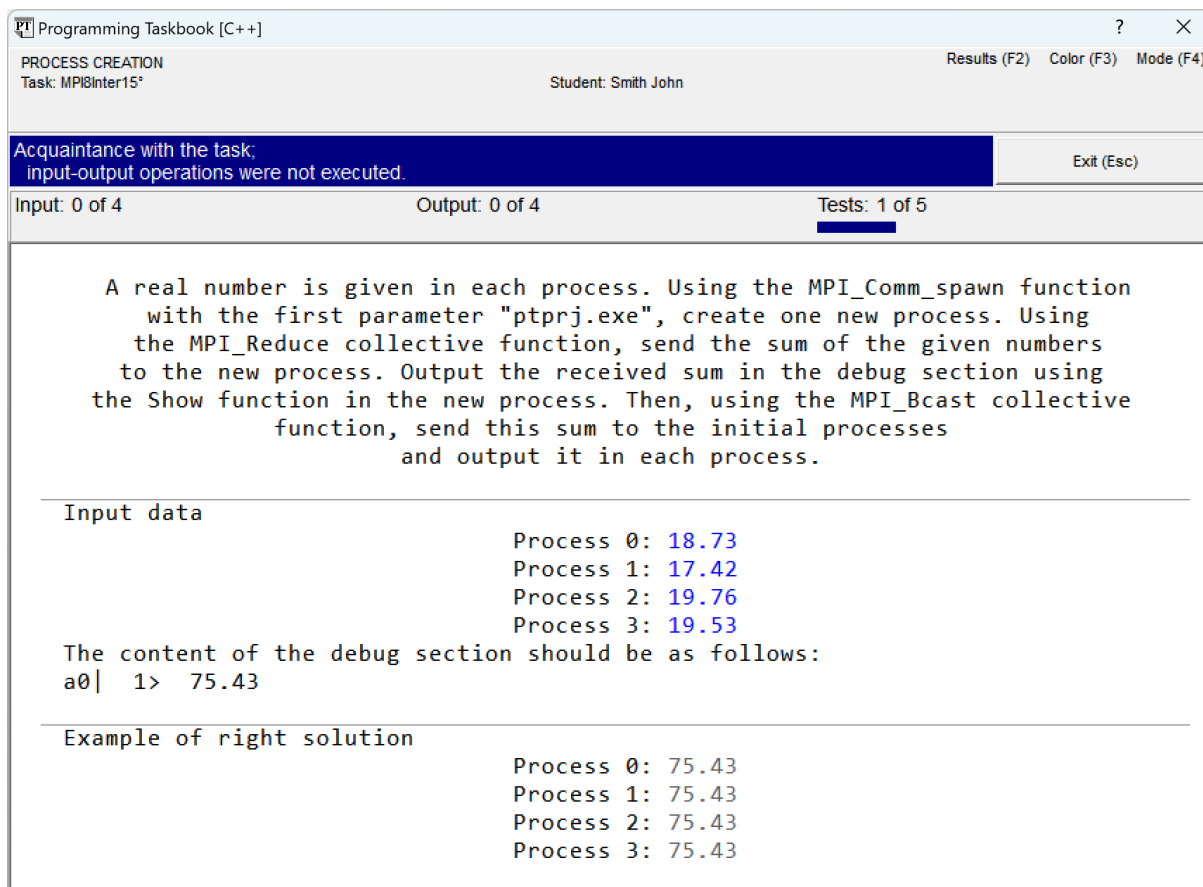


Fig. 36. Acquaintance running of the MPI8Inter15 task

When executing tasks to create processes, it is necessary to take into account that the code of the Solve function will be executed not only in the initial processes of the parallel application, but also in the created processes. Note that for the created processes, as well as for the initial ones, you *cannot* call the MPI_Init function, since this call is automatically executed by the taskbook *before* calling the Solve function in each process.

To ensure that the MPI_Comm_spawn function is called only in the initial processes, it is necessary to call the MPI_Comm_get_parent function at the beginning of the Solve function and analyze its result. If it returns the value MPI_COMM_NULL, then this process is one of the initial processes, and the MPI_Comm_spawn function should be called for it; if a non-empty communicator is returned, this means that the process is a child process, and this communicator can be used to communicate with parent processes.

To check that the new process was created successfully, we can output the size and rank values for *each* process of the parallel application in the debug section. We get the following code fragment:

```
MPI_Comm inter;
```

```

MPI_Comm_get_parent(&inter);
if (inter == MPI_COMM_NULL)
{
    MPI_Comm_spawn("ptprj.exe", NULL, 1, MPI_INFO_NULL, 0,
        MPI_COMM_WORLD, &inter, MPI_ERRCODES_IGNORE);
}
Show(size);
Show(rank);

```

When you launch this version of the program, the taskbook window will look like that shown in Fig. 37.

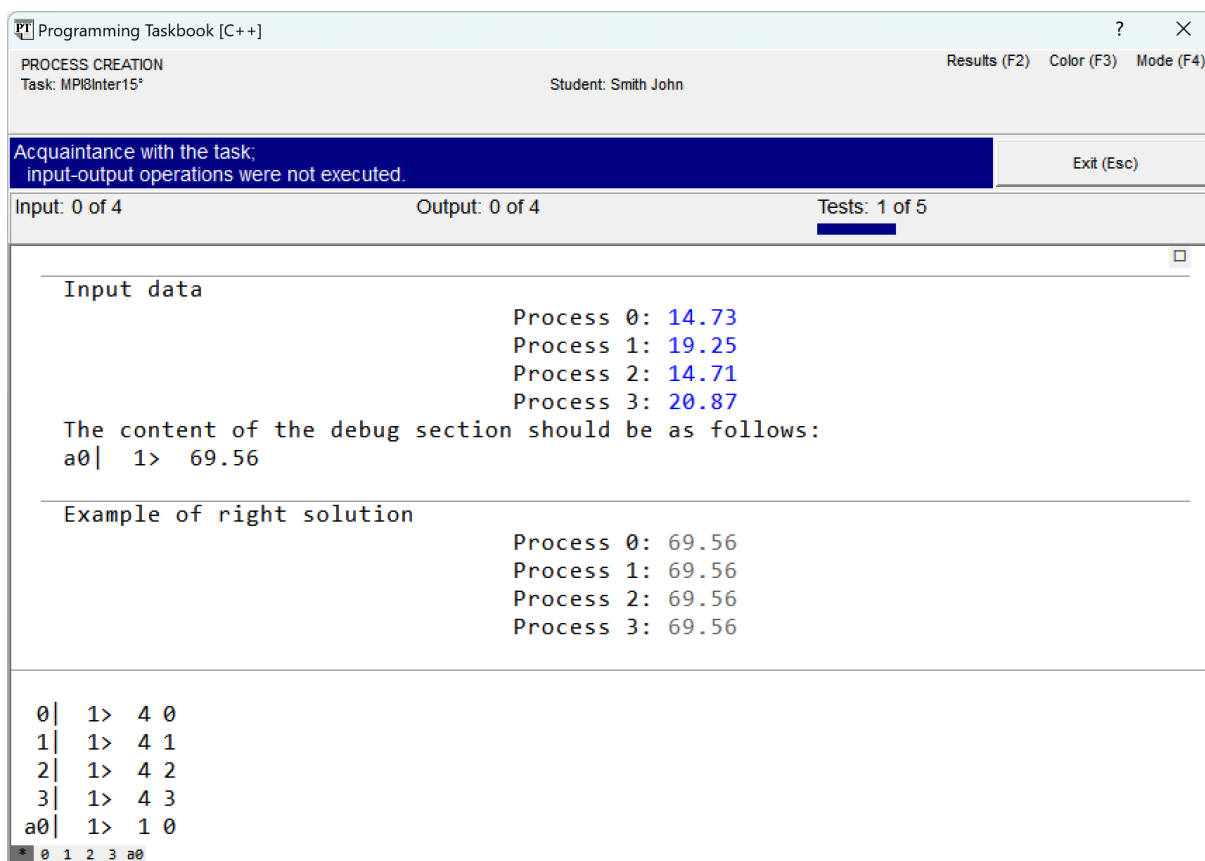


Fig. 37. The first stage of solving the MPI8Inter15 task

At the beginning of each line in the debug section, the rank of the process is indicated. Along with the "usual" ranks 0–3, the section contains a rank starting with the prefix "a". As was said above, dynamically created processes are marked in this way. So, in our case, the program includes 4 initial processes (for them, size = 4, and ranks vary from 0 to 3) and one new process. The standard communicator MPI_COMM_WORLD is also associated with this process, but it includes only one process (since when calling the MPI_Comm_spawn function, we specified the value 1 as the maxproc parameter), so the line with the label "a0" displays the numbers 1 (the number of processes in the communicator) and 0 (the rank of the process). So, our program correctly creates a new process.

At the second stage of the solution, we must obtain in the new process the sum of the numbers specified in the initial processes. According to the task formulation, the collective function `MPI_Reduce` must be used for this. Obviously, it must be applied to the inter-communicator `inter`, which is defined both in the initial processes and in the new process.

When defining the root parameter for the `MPI_Reduce` function, it is necessary to take into account the peculiarities of using collective operations for inter-communicators (see Note 3 in Section 1.3.7): in all processes sending their data to another inter-communicator group, we must specify the "actual" rank of the receiving root process in the remote group (in this case, 0), and in the receiving process, a special value `MPI_ROOT` must be specified. If the group of new processes contains more than one process, then in the remaining processes, the value `MPI_PROC_NULL` must be specified as the root parameter.

Thus, before calling the `MPI_Reduce` function, we need not only to perform the data input (in the initial processes), but also to determine the value of the root parameter in all processes. After calling this function, we need to output the obtained result in the debug section corresponding to the new process.

Let us present a new version of the solution, highlighting the added fragments in bold:

```
double a, sum;
int root;
MPI_Comm inter;
MPI_Comm_get_parent(&inter);
if (inter == MPI_COMM_NULL)
{
    MPI_Comm_spawn("ptprj.exe", NULL, 1, MPI_INFO_NULL, 0,
        MPI_COMM_WORLD, &inter, MPI_ERRCODES_IGNORE);
    pt >> a;
    root = 0;
}
else
    root = MPI_ROOT;
Show(size);
Show(rank);
MPI_Reduce(&a, &sum, 1, MPI_DOUBLE, MPI_SUM, root, inter);
if (root == MPI_ROOT)
    Show(sum);
```

We replaced the previous `Show(size)` and `Show(rank)` functions with a single `Show(sum)` statement that outputs the resulting sum of the given numbers in the new process. Note that we also used the root value to "distinguish" the new process from the initial ones.

As a result of launching this version of the program, the window will look like that shown in Fig. 38.

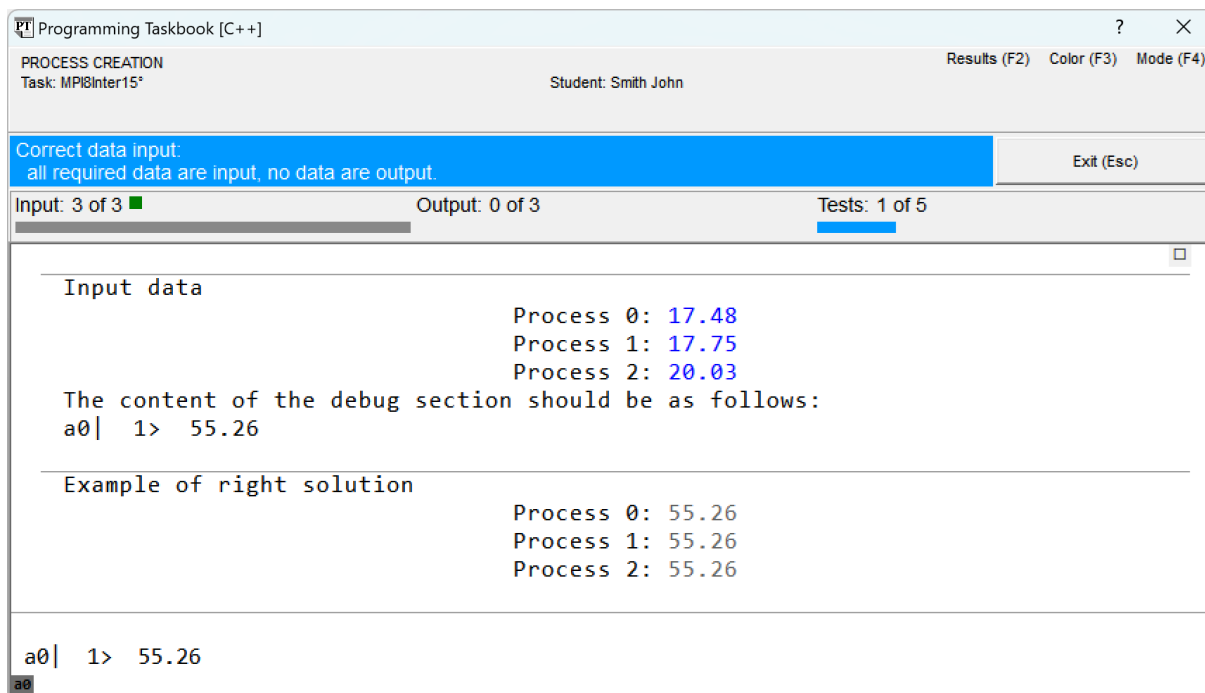


Fig. 38. The second stage of solving the MPI8Inter15 task

Note that the contents of the debug section exactly match the sample shown in the input data section. The message in the information section is "*Correct data input*" because the program has all the required input, but no output has been performed in any of the initial processes.

Let us perform the last part of the task: send the found sum from the new process to all initial processes and output the received data. To do this, we also need to use a collective function (in this case, `MPI_Bcast`), but now the sending process is the new (child) process, and the receiving processes are the initial (parent) processes. For the `MPI_Bcast` function, you need to set *the same values* for the root parameter as for the previously called `MPI_Reduce` function; indeed, in this case, the process sending data (the root process) is the new process, so we need to use the root parameter equal to `MPI_ROOT` in this process, and all processes in the other (receiving) group must indicate that they receive data from process 0 of the sending group.

Thus, after calling `MPI_Reduce`, we only need to add a call to the new collective function `MPI_Bcast` and provide the last conditional statement with an else branch (executed in the initial processes), in which we organize the output of the resulting sum. Here is the final version of the solution, highlighting the new statements in bold:

```
double a, sum;
int root;
MPI_Comm inter;
MPI_Comm_get_parent(&inter);
if (inter == MPI_COMM_NULL)
```



```
{
    MPI_Comm_spawn("ptprj.exe", NULL, 1, MPI_INFO_NULL, 0,
                  MPI_COMM_WORLD, &inter, MPI_ERRCODES_IGNORE);
    pt >> a;
    root = 0;
}
else
    root = MPI_ROOT;
MPI_Reduce(&a, &sum, 1, MPI_DOUBLE, MPI_SUM, root, inter);
MPI_Bcast(&sum, 1, MPI_DOUBLE, root, inter);
if (root == MPI_ROOT)
    Show(sum);
else
    pt << sum;
```

After launching this version of the program, we will receive a message that the task has been solved.

In the final part of this section, we describe two more features related to inter-communicators and dynamic process creation. These features are the subject of the four final tasks of the MPI8Inter group.

The communicator that is created when new processes are created is an *inter-communicator*; one its group includes the parent processes and the other its group includes the child processes. Sometimes, after creating new processes, it is convenient *to merge* the parent and child processes into one common *intra-communicator*. For this purpose, MPI provides a special function `MPI_Intercomm_merge(MPI_Comm comm, int high, MPI_Comm *newcomm)` with three parameters: the original inter-communicator `comm`, the `high` parameter, which determines the order of processes in the created intra-communicator, and the output parameter `newcomm`, a pointer to the created intra-communicator. The `MPI_Intercomm_merge` function must be called in all processes of the original inter-communicator.

The `high` parameter is an integer flag; all processes in each group of the original inter-communicator must specify *the same value* for the `high` parameter. If the `high` parameter is 0 in one of the groups and 1 in the another, then the processes of the first group (with `high = 0`) are placed first in the created intra-communicator in the order in which they appear in this group, and then the processes of the second group (with `high = 1`) are placed in the intra-communicator, also in the order in which they appear in this group. If the value of the `high` parameter is the same in *all* processes of the inter-communicator, then the order of the groups is undefined, but even in this case the order of the processes from each group coincides with the order of the processes in this group.

The `MPI_Intercomm_merge` function is used in the MPI8Inter19–MPI8Inter20 tasks.

A situation is possible when two groups of processes do not have a common inter-communicator (for example, if each of these groups was created using a separate call to the `MPI_Comm_spawn` function). To establish communication between such groups, a special *client-server interaction mechanism* implemented in MPI-2 can be used. One of the groups of processes between which communication needs to be established plays the role of *a server*, and the other plays the role of *a client*. The processes of the server group create *a port* for communication (using the function `MPI_Open_port(MPI_Info info, char *port_name)`, the `port_name` parameter is output) and define *the public name* of this port (using the function `MPI_Publish_name(char *service_name, MPI_Info info, char *port_name)`, all parameters are input), after which they start listening to this port waiting for the client to connect. For this purpose, the processes of the server group use the function `MPI_Comm_accept(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)`. This function is collective and must be called for all processes of the communicator `comm`, however, the port name `port_name` only needs to be specified in the process of rank `root`.

Processes of the client group obtain the port created by the server group using the *public name* `service_name` of this port (by means of the function `MPI_Lookup_name(char *service_name, MPI_Info info, char *port_name)`; the last parameter is output), after which they connect to the server on this port using the function `MPI_Comm_connect(char *port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm *newcomm)`. This function, like the `MPI_Comm_accept` function, is collective and must be called for all processes of the `comm` communicator, and, like for the `MPI_Comm_accept` function, the port name `port_name` only needs to be specified in the process of the rank `root`.

When the `MPI_Comm_accept` and `MPI_Comm_connect` functions complete successfully, they return an inter-communicator `newcomm` that joins the client group and the server group.

The port (the `port_name` parameter) is a text string generated by the MPI environment when calling the `MPI_Open_port` function; its maximum length (with the terminating null character) does not exceed the `MPI_MAX_PORT_NAME` constant. It is sufficient to create the port in one of the processes of the server group and receive it in one of the processes of the client group. The public name of the port (the `service_name` parameter), unlike the parameter `port_name`, must be known in advance to both the processes of the server group and the processes of the client group; this is a kind of "password" exchanged between these groups.

All these functions provide an additional parameter `info`, which can be set to `MPI_INFO_NULL`.

Let us give a standard sequence of actions on the server group side:

```
char port[MPI_MAX_PORT_NAME];
if (rank == 0)
{
```

```
MPI_Open_port(MPI_INFO_NULL, port);
MPI_Publish_name("password", MPI_INFO_NULL, port);
}
MPI_Comm_accept(port, MPI_INFO_NULL, 0, comm, &inter);
```

The sequence of actions on the client group side is as follows:

```
char port[MPI_MAX_PORT_NAME];
if (rank == 0)
    MPI_Lookup_name("password", MPI_INFO_NULL, port);
MPI_Comm_connect(port, MPI_INFO_NULL, 0, comm, &inter);
```

It is important *to coordinate* the function calls so that the `MPI_Publish_name` function in the root process of the server group is called *earlier in time* than the `MPI_Lookup_name` function in the root process of the client group.

Three functions are intended for releasing resources allocated during a client-server connection:

- `MPI_Close_port(char *port_name)` closes the port `port_name` created by the `MPI_Open_port` function;
- `MPI_Unpublish_name(char *service_name, MPI_Info info, char *port_name)` releases the port public name `service_name` previously connected with the port `port_name` (the public name of the port should be released before the closing the port itself);
- `MPI_Comm_disconnect(MPI_Comm *comm)` destroys communicator `comm` created for client-server connection and returns `MPI_COMM_NULL` in the `comm` parameter. Unlike the standard function `MPI_Comm_free`, the function `MPI_Comm_disconnect` waits for all communication operations performed using the communicator `comm` to complete, and only then destroys this communicator.

The client-server interaction mechanism is used in the tasks MPI8Inter21–MPI8Inter22. The notes to these tasks describe ways to coordinate the order of calling the `MPI_Publish_name` and `MPI_Lookup_name` functions based on the `MPI_Barrier` function.

1.4. Parallel matrix algorithms

1.4.1. Band and block algorithms for parallel matrix multiplication: general description

The MPI9Matr group, unlike the previous groups, is devoted not to some specific section of the MPI library, but *to parallel matrix algorithms*, which use various tools of this library, including various options for interaction between processes, new derived datatypes, communicators with virtual topologies, and parallel file input-output. Thus, the MPI9Matr group can be considered as a final group, allowing you to repeat and consolidate most of the previously studied topics related to MPI technologies.

Matrix processing is one of those types of computational tasks for which efficient parallelization methods have been developed, including those based on MPI technologies. It should be noted that the MPI library contains a number of tools specifically designed for working with matrices; such tools include derived datatypes associated with various matrix fragments (columns, sets of columns, blocks), as well as additional capabilities of communicators with Cartesian topology.

A typical representative of matrix algorithms is the *matrix multiplication algorithm*. The MPI9Matr group considers two main types of parallel distributed matrix multiplication algorithms: *band algorithms*, in which the distribution of computations between processes is achieved by dividing matrices into *bands*, including sets of adjacent rows or columns, and *block algorithms*, in which matrix division into rectangular *blocks* is used. In addition, the group includes the introductory task MPI9Matr1 (see Section 2.9.1), which describes the matrix data storage format, provides the necessary formulas, and requires the implementation of the simplest non-parallel matrix multiplication algorithm.

For each type of parallel matrix multiplication algorithm, two variants are considered, differing in implementation details.

In *the band algorithm 1* (Section 2.9.2, MPI9Matr2–MPI9Matr10), only horizontal bands (sets of adjacent rows) are used, and their transfer does not require new derived datatypes.

In *the band algorithm 2* (Section 2.9.3, MPI9Matr11–MPI9Matr20), both horizontal and vertical bands are used, which, on the one hand, somewhat simplifies the implementation of the multiplication algorithm itself (since it is based on multiplying *the rows* of one matrix by *the columns* of another), and on the other hand, requires the use of new datatypes that provide more efficient transfer of vertical bands (i. e., sets of adjacent columns).

In the first version of the block algorithm (*Cannon's algorithm*, Section 2.9.4, MPI9Matr21–MPI9Matr31), before the iterative calculation of the fragments of the final matrix product, an initial stage of block *redistribution* between processes is performed, which simplifies subsequent actions for sending data. To send blocks at any stage of the algorithm, an auxiliary communicator equipped with a topology of a square matrix of processes is used.

In the second version of the block algorithm (*Fox's algorithm*, Section 2.9.5, MPI9Matr32–MPI9Matr44), the special stage of initial redistribution is absent and, as a consequence, each step of calculating the final matrix product requires more complex actions for sending blocks. In this sending, it is proposed to use not only a communicator with the topology of a square matrix of processes, but also communicators generated on its basis and associated with individual rows and columns of this matrix.

In each of the variants of the block algorithm, it is necessary to define a new derived datatype that simplifies the transfer of matrix blocks; in Cannon's

algorithm, it is proposed to use the `MPI_Send` and `MPI_Recv` functions to transfer blocks, and in Fox's algorithm, it is proposed to use the collective function `MPI_Alltoallw` (provided that the MPI-2 library is used).

In each of the considered matrix multiplication algorithms, three main stages can be distinguished:

- *source data scattering stage*: initial scattering source matrix fragments (bands or blocks) to all processes;
- *computation stage*: sequential computation of fragments of the final matrix product, each step of which is accompanied by the transfer of fragments of the original matrices between processes (for Cannon's algorithm, the computation stage is preceded by an *initialization stage*, which performs the initial redistribution of blocks between processes);
- *result gathering stage*: sending the calculated fragments of the matrix product to the master process in order to obtain the final matrix.

Each of these stages is associated with a separate task (or series of tasks); in this case, the initial data in each task is a set of data that must be formed as a result of the execution of the *previous* stage. This simplifies the development and testing of each stage, and also makes it possible to implement separate stages of the algorithm without first developing all the previous stages.

The task series that are associated with the computation stage are the most complex. The initial task of each series requires the development of the simplest version of the computation used in the first step of the algorithm, and in subsequent tasks, this version is modified to be applicable to each step of the computational stage. An exception is Cannon's algorithm, for which the actions at each step do not depend on the step number (see the note to the `MPI9Matr25` task).

There are also tasks associated with the modified initial and final stages of each algorithm using *parallel file input-output*:

- *file data reading stage*: each process obtains fragments of the source matrices directly from the files containing these matrices;
- *final file writing stage*: each process writes the received fragments of the final product to the corresponding part of the resulting file.

In addition, for those algorithms that require the use of new datatypes or communicators with Cartesian topology, additional tasks are provided related to the creation of the corresponding objects (of type `MPI_Datatype` or `MPI_Comm`).

In all tasks devoted to the implementation of various stages of matrix algorithms, as well as the creation of auxiliary objects (new derived datatypes or communicators), it is necessary to formalize the corresponding actions in the form of an *auxiliary function*. Functions that create new objects are used later when implementing various stages of the algorithm, and functions associated with the stages themselves are used in final tasks in which it is necessary to implement the corresponding matrix algorithm in full.

Tables 2 and 3 below list the task numbers associated with implementing the various stages of each algorithm and also provide the names of the functions that must be developed in these tasks.

Table 2

MPI9Matr group tasks related to band algorithms

Algorithm stage	Band algorithm 1	Band algorithm 2
Defining a new derived datatype	<i>Absent</i>	MPI9Matr11: Matr2CreateTypeBand (p, k, q, t)
Source data scattering stage	MPI9Matr2: Matr1ScatterData()	MPI9Matr12: Matr2ScatterData()
Computation stage	MPI9Matr3: Matr1Calc() MPI9Matr4–MPI9Matr5: Matr1Calc(l)	MPI9Matr13: Matr2Calc() MPI9Matr14–MPI9Matr15: Matr2Calc(l)
Result gathering stage	MPI9Matr6: Matr1GatherData()	MPI9Matr16: Matr2GatherData()
Full implementation of the algorithm	MPI9Matr7	MPI9Matr17
File data reading stage	MPI9Matr8: Matr1ScatterFile()	MPI9Matr18: Matr2ScatterFile()
Final file writing stage	MPI9Matr9: Matr1GatherFile()	MPI9Matr19: Matr2GatherFile()
Full implementation of the algorithm using file input-output	MPI9Matr10	MPI9Matr20

Table 3

MPI9Matr group tasks related to block algorithms

Algorithm stage	Cannon's block algorithm	Fox's block algorithm
Defining a new derived datatype	MPI9Matr21: Matr3CreateTypeBlock (m0, p0, p, t)	MPI9Matr32: Matr4CreateTypeBlock (m0, p0, p, t)
Defining new communicators with Cartesian topology	MPI9Matr22: Matr3CreateCommGrid (comm)	MPI9Matr33: Matr4CreateCommGrid (comm), Matr4CreateCommRow (grid, row) MPI9Matr34: Matr4CreateCommCol (grid, col)
Source data scattering stage	MPI9Matr23: Matr3ScatterData()	MPI9Matr35: Matr4ScatterData()

Table 3 (continued)

Algorithm stage	Cannon's block algorithm	Fox's block algorithm
Initialization stage	MPI9Matr24: Matr3Init()	<i>Absent</i>
Computation stage	MPI9Matr25–MPI9Matr26: Matr3Calc()	MPI9Matr36: Matr4Calc1() MPI9Matr37: Matr4Calc2() MPI9Matr38–MPI9Matr39: Matr4Calc1(l), Matr4Calc2()
Result gathering stage	MPI9Matr27: Matr3GatherData()	MPI9Matr40: Matr4GatherData()
Full implementation of the algorithm	MPI9Matr28	MPI9Matr41
File data reading stage	MPI9Matr29: Matr3ScatterFile()	MPI9Matr42: Matr4ScatterFile()
Final file writing stage	MPI9Matr30: Matr3GatherFile()	MPI9Matr43: Matr4GatherFile()
Full implementation of the algorithm using file input-output	MPI9Matr31	MPI9Matr44

Another feature of the tasks of the MPI9Matr group is the creation of *specialized template projects*, which already contain declarations of global variables for storing various objects used when solving the task (this feature is noted in the preamble to this group of tasks, see Section 2.9).

1.4.2. Implementation of a non-parallel matrix multiplication algorithm

Let us start by looking at the introductory task of the MPI9Matr group, which is intended to introduce the techniques of working with matrices that are necessary when solving any tasks in this group.

MPI9Matr1. Integers M , P , Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process. Find and output a $M \times Q$ matrix C that is the product of the matrices A and B .

The formula for calculating the elements of the matrix C under the assumption that the rows and columns of all matrices are numbered from 0 is as follows: $C_{I,J} = A_{I,0} \cdot B_{0,J} + A_{I,1} \cdot B_{1,J} + \dots + A_{I,P-1} \cdot B_{P-1,J}$, where $I = 0, \dots, M - 1$, $J = 0, \dots, Q - 1$.

To store the matrices A , B , C , use one-dimensional arrays of size $M \cdot P$, $P \cdot Q$, and $M \cdot Q$ placing elements of matrices in a row-major order (that is, the matrix element with indices I and J will be stored in the element of the corresponding array with the index $I \cdot N + J$, where N is the number of columns of the matrix). The slave processes are not used in this task.

MPI9Matr1.cpp file, created as a template for solving this task, contains the following code:

```

#include "pt4.h"
#include "mpi.h"
# include <cmath>

int k;      // number of processes
int r;      // rank of the current process

int m, p, q; // sizes of the given matrices

int *a_, *b_, *c_;
    // arrays to store matrices in the master process

void Solve()
{
    Task("MPI9Matr1");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    k = size;
    r = rank;
}

```

Let us describe the features of the created template. First, it includes the standard header `<cmath>`, since many tasks of the MPI9Matr group require the use of the rounding functions `ceil` and `floor` declared in this header. Second, the file contains declarations of a set of global variables associated with the task being solved (the purpose of the variables is indicated in the comments located next to the declarations). In particular, shorter names are introduced for the number of processes and the rank of the current process: `k` and `r`. The variables `k` and `r`, unlike the variables `size` and `rank` added to the template of any program associated with the PT for MPI-2 taskbook, can be used not only in the `Solve` function, but also in auxiliary functions that need to be developed when solving most of the tasks in the MPI9Matr group.

It should be noted that the names of the pointer variables `a_`, `b_`, `c_`, which must be associated with the arrays containing the given matrices A and B , as well as the result of their multiplication $C = AB$, are provided with the underscore character. This is explained by the fact that the shorter names of the variables `a`, `b`, `c` are associated with *matrix fragments* (bands or blocks) that are processed in each process. The variables `a_`, `b_`, `c_`, unlike the variables `a`, `b`, `c`, must be used only in the master process. Besides the MPI9Matr1 task, the variables `a_`, `b_`, `c_` are required only in those tasks in which fragments of the given

matrices are sent to all processes, as well as fragments of the resulting matrix product are sent to the master process.

When you launch this template, a taskbook window will appear on the screen (Fig. 39). To reduce the size of the window, the section with the task formulation is hidden in this and subsequent figures.

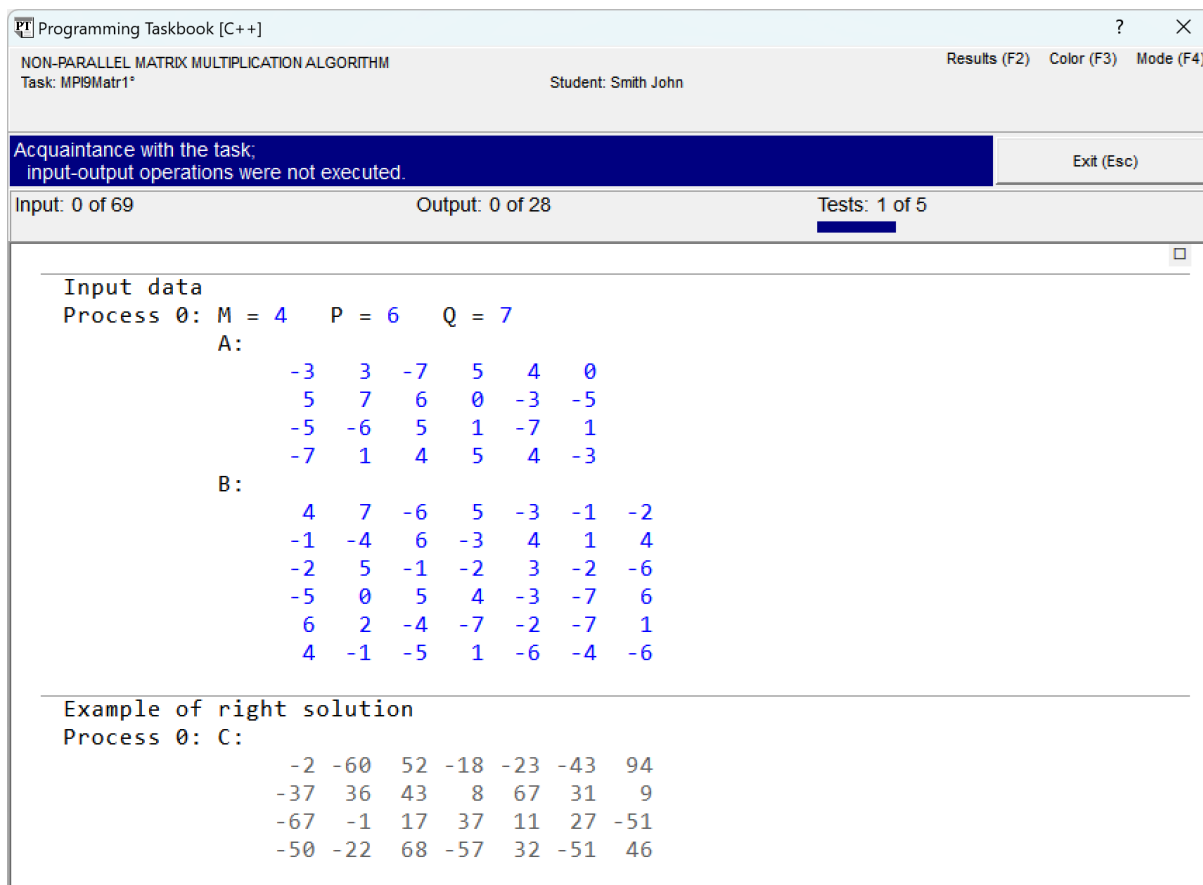


Fig. 39. Acquaintance running of the MPI9Matr1 task

Although the created project (like all other projects for tasks included in the PT for MPI-2 taskbook) runs as a parallel application, it does not require the use of slave processes: data input, processing, and output of results must be performed only in the master process.

In all MPI9Matr group tasks, matrices must be stored in *one-dimensional* dynamic arrays (row by row). Two-dimensional arrays should not be used in this case, since this complicates the transfer of matrix fragments between processes.

At the first stage of the solution, we organize the input of the initial data. To do this, we add the following statements to the Solve function:

```

if (r != 0)
    return;
pt >> m >> p >> q;
a_ = new int[m*p];
b_ = new int[p*q];
for (int i = 0; i < m*p; i++)

```

```

    pt >> a_[i];
for (int i = 0; i < p*q; i++)
    pt >> b_[i];

```

At the beginning of this fragment, we analyze the rank of the process and immediately exit the Solve function if the process is not the master process (i. e., a process of rank 0). Then the matrix sizes are input and memory is allocated for the given arrays `a_` and `b_`, after which the elements of the matrices A and B are read into these arrays. Since the taskbook always passes matrix elements row by row, it is sufficient to use a *single* loop to input these elements into each of the one-dimensional arrays.

When you launch this version of the program, a message will appear in the taskbook window stating that all initial data are successfully input.

At the second stage of the solution, we allocate memory for the array `c_` intended for storing the resulting matrix product C , set its elements equal to zero and perform matrix multiplication using the formula given in the formulation of the task (the element of matrix C with indices I and J is obtained as a result of pairwise multiplication of the elements of the I -th row of matrix A and the J -th column of matrix B and summing the resulting products):

$$C_{I,J} = A_{I,0} B_{0,J} + A_{I,1} B_{1,J} + \dots + A_{I,P-1} B_{P-1,J}$$

Since all matrices are stored in one-dimensional arrays (by rows), and indexing of both matrix elements and array elements starts from 0, to access a matrix element with indices I and J , one should access the array element with index $I \cdot N + J$, where the symbol N denotes *the number of matrix columns*.

After finding the matrix product, all its elements must be output. Thus, the second part of the solution looks like this:

```

c_ = new int [m*q];
for (int i = 0; i < m*q; i++)
    c_[i] = 0;
for (int i = 0; i < m; i++)
    for (int j = 0; j < q; j++)
        for (int n = 0; n < p; n++)
            c_[i*q+j] += a_[i*p+n] * b_[n*q+j];
for (int i = 0; i < m*q; i++)
    pt << c_[i];

```

Note that to output the obtained results (as well as to input the elements of the given matrices), it is sufficient to use a single loop; in this case, the taskbook itself provides a visual display of the resulting matrix in its window.

In the final part of the solution, we free the memory allocated for arrays `a_`, `b_` and `c_`:

```

delete[] a_;
delete[] b_;
delete[] c_;

```

As a result of launching this version of the program, a message will be displayed in the taskbook window stating that the task has been solved.

So, this task allowed us to become familiar with the methods of input and output of matrix data, and also demonstrated the standard algorithm of matrix multiplication.

Note that instead of dynamic arrays, when solving tasks on implementing matrix algorithms, you can use `std::vector<T>` containers (see Sections 1.2.6 and 1.2.7). This allows you to represent the matrix input-output actions in a more concise and visual manner. Let us present the corresponding solution to the MPI9Matr1 task, highlighting the modified or added fragments in bold:

```
#include "pt4.h"
#include "mpi.h"
#include <cmath>
#include <vector>
#include <algorithm>

int k;      // number of processes
int r;      // rank of the current process

int m, p, q; // sizes of the given matrices

std::vector<int> a_, b_, c_;
    // arrays to store matrices in the master process

void Solve()
{
    Task("MPI9Matr1");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    k = size;
    r = rank;
    if (r != 0)
        return;
    pt >> m >> p >> q;
a.assign(ptin_iterator<int>(m*p), ptin_iterator<int>());
b.assign(ptin_iterator<int>(p*q), ptin_iterator<int>());
c.assign(m*q, 0);
    for (int i = 0; i < m; i++)
        for (int j = 0; j < q; j++)
            for (int n = 0; n < p; n++)
                c_[i*q+j] += a_[i*p+n] * b_[n*q+j];
}
```

```

    copy(c_.begin(), c_.end(), ptout_iterator<int>());
//or:
// pt << c_;
}

```

1.4.3. Scattering source data: an example of implementation

Now let us turn to the MPI9Matr2 task, which is related to the implementation of the first stage of the algorithm, namely, input of the source data in the master process and sending them to other processes of the parallel application. This task is the first in the subgroup devoted to the first version of the band algorithm (in which both matrices are divided into horizontal bands).

MPI9Matr2. Integers M , P , Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process. In the first variant of the band algorithm of matrix multiplication, each matrix multiplier is divided into K horizontal bands, where K is the number of processes (hereinafter bands are distributed by processes and used to calculate a part of the total matrix product in each process).

The band of the matrix A contains N_A rows, the band of the matrix B contains N_B rows. The numbers N_A and N_B are calculated as follows: $N_A = \text{ceil}(M/K)$, $N_B = \text{ceil}(P/K)$, where the operation "/" means the division of real numbers and the function `ceil` performs rounding up. If the matrix contains insufficient number of rows to fill the last band, then the zero-valued rows should be added to this band.

Add, if necessary, the zero-valued rows to the initial matrices, save them in one-dimensional arrays in the master process, and then send the matrix bands from these arrays to all processes as follows: a band with the index R is sent to the process of rank R ($R = 0, 1, \dots, K - 1$), all the bands A_R are of the size $N_A \times P$, all the bands B_R are of the size $N_B \times Q$. In addition, create a band C_R in each process to store the part of the matrix product $C = AB$ which will be calculated in this process. Each band C_R is of the size $N_A \times Q$ and is filled with zero-valued elements.

The bands, like the initial matrices, should be stored in one-dimensional arrays in a row-major order. To send the matrix sizes, use the `MPI_Bcast` collective function, to send the bands of the matrices A and B , use the `MPI_Scatter` collective function.

Include all the above mentioned actions in a `Matr1ScatterData` function (without parameters). As a result of the call of this function, each process will receive the values N_A , P , N_B , Q , as well as one-dimensional arrays filled with the corresponding bands of the matrices A , B , C . Output all obtained data (that is, the numbers N_A , P , N_B , Q and the bands of the matrices A , B , C) in each process after calling the `Matr1ScatterData` function. Perform the input of initial data in the `Matr1ScatterData` function, perform the output of the results in the `Solve` function.

Note. To reduce the number of the MPI_Bcast function calls, all matrix sizes may be sent as a single array.

The template for this task is as follows:

```
#include "pt4.h"
#include "mpi.h"
# include <cmath>

int k;      // number of processes
int r;      // rank of the current process

int m, p, q; // sizes of the given matrices
int na, nb;  // sizes of the matrix bands

int *a_, *b_, *c_;
    // arrays to store matrices in the master process
int *a, *b, *c;
    // arrays to store matrix bands in each process

void Solve()
{
    Task("MPI9Matr2");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    k = size;
    r = rank;
}
```

In this case, the global variables additionally include arrays a, b, c for storing matrix bands in each process, as well as variables na and nb, which determine the sizes of these bands.

When you launch the created template, the taskbook window will appear on the screen (Figs. 40–41). The first figure shows the section with the input data, the second figure shows the section with an example of correct results.

As follows from the figures, the tasks of the MPI9Matr group can use a large number of given and resulting data (for the variant shown in the figures, the number of given numbers is 133, and the number of resulting data is 198). However, despite the large amount of data, all of them are displayed in the taskbook window in a clear form thanks to special formatting and comments.

```

Programming Taskbook [C++]
BAND ALGORITHM 1 (HORIZONTAL BANDS)
Task: MPI9Matr2
Student: Smith John
Results (F2) Color (F3) Mode (F4)
Acquaintance with the task;
input-output operations were not executed.
Exit (Esc)
Input: 0 of 133 Output: 0 of 198 Tests: 1 of 5
-
+
/
Input data
Process 0: M = 6 P = 10 Q = 7
A:
-3 -2 -6 -6 -7 2 7 1 -7 -4
7 -7 4 4 -4 -4 3 -5 1 -4
-1 -6 7 -7 2 4 -2 5 -4 4
4 1 2 0 1 0 2 0 -1 7
-5 4 -3 7 1 -1 -1 6 5 -7
-4 2 -7 -4 -7 7 7 -7 5 3
B:
2 -2 3 -5 -5 -6 1
4 2 7 2 0 -6 -6
-1 6 4 -1 -5 3 -1
-7 -7 3 3 6 5 -5
0 -6 5 -4 2 6 0
-4 4 0 -5 0 4 2
-6 4 -6 -6 2 -3 2
2 -5 3 -3 6 -3 -7
-3 2 -5 -4 2 2 2
-6 -5 -7 4 -6 2 6

```

Fig. 40. Acquaintance running of the MPI9Matr 2 task (input data section)

All given data must be input in the master process. The output for each process must be the band sizes (these values are the same for all processes) and the bands themselves; the bands for the final product C must be zero. In some processes, the final rows of the bands associated with the initial matrices A and B may also be zero. In our case, two zero rows are contained in the band of matrix B associated with the last process (of rank 2).

According to the task condition, all actions related to the input of given data and their distribution must be implemented in the function `Matr1ScatterData` without parameters (global variables declared in the program template must be used in this function). At the first stage, we will perform data input in the master process:

```

void Matr1ScatterData()
{
    if (r == 0)
    {
        int m;
        pt >> m >> p >> q;
        na = (int)ceil(m / (k*1.0));
        nb = (int)ceil(p / (k*1.0));
        a_ = new int[na*k*p];
    }
}

```

```

    b_ = new int[nb*k*q];
    for (int i = 0; i < m*p; i++)
        pt >> a_[i];
    for (int i = m*p; i < na*k*p; i++)
        a_[i] = 0;
    for (int i = 0; i < p*q; i++)
        pt >> b_[i];
    for (int i = p*q; i < nb*k*q; i++)
        b_[i] = 0;
}
}

```

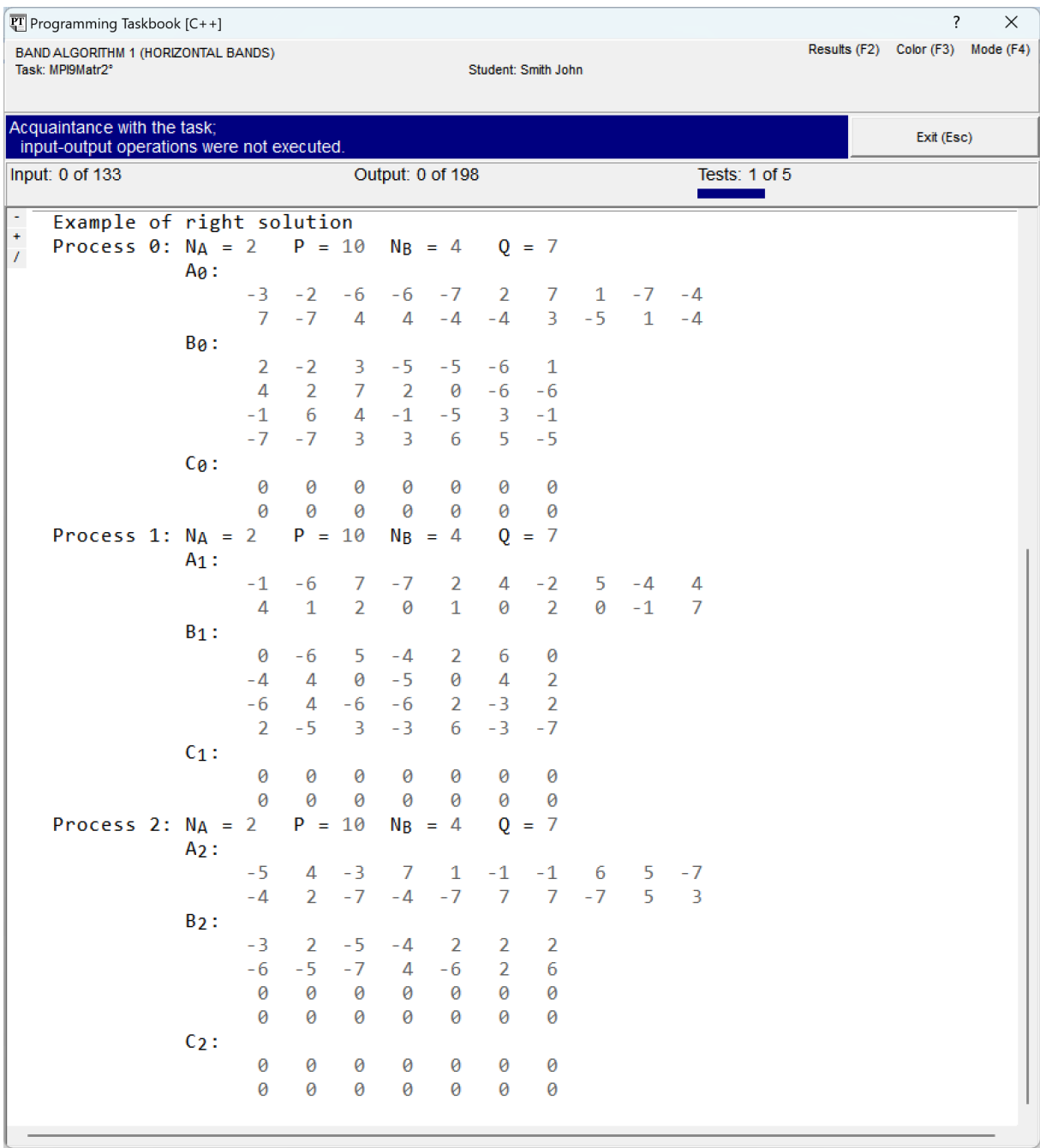


Fig. 41. Acquaintance running of the MPI9Matr2 task (example of the correct solution)

Note that it is necessary to input into arrays `a_` and `b_` not only the elements of the given matrices, but also the terminating zero rows, which will allow us to obtain bands of the same size in each process.

The `Solve` function must be supplemented by calling the `Matr1ScatterData` function.

When you run this version of the program, a message will be displayed stating that all given data are input.

Now let us add the final fragment to the `Matr1ScatterData` function, which ensures data transfer:

```
MPI_Bcast(&na, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&p, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&nb, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&q, 1, MPI_INT, 0, MPI_COMM_WORLD);
a = new int[na*p];
b = new int[nb*q];
c = new int[na*q];
MPI_Scatter(a_, p*na, MPI_INT, a, p*na, MPI_INT, 0,
           MPI_COMM_WORLD);
MPI_Scatter(b_, q*nb, MPI_INT, b, q*nb, MPI_INT, 0,
           MPI_COMM_WORLD);
for (int i = 0; i < na*q; i++)
    c [i] = 0;
```

In this fragment, we have not used the recommendation from the task note and have not placed the transferred sizes in an auxiliary array (the reader is advised to perform such a modification of the algorithm). In addition to transferring the bands of matrices *A* and *B*, we create a band for the final product *C* in each process and set its elements to zero.

The result of launching a new version will not differ from the previous one.

All that remains is to output the results. This action should not be included in the `Matr1ScatterData` function, since this function will be used later in the final `MPI9Matr7` task, which does not require outputting the results obtained in the first stage of the algorithm. Therefore, we will place the output statements at the end of the `Solve` function (the added statements are highlighted in bold):

```
void Solve()
{
    Task("MPI9Matr2");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    k = size;
    r = rank;
```



```

Matr1ScatterData();
pt << na << p << nb << q;
for (int i = 0; i < na*p; i++)
    pt << a[i];
for (int i = 0; i < nb*q; i++)
    pt << b[i];
for (int i = 0; i < na*q; i++)
    pt << c [ i ];
}

```

After running the program, we will see a message that the task has been solved.

1.4.4. Redistribution of blocks at the initial stage of Cannon's algorithm

When implementing the next stages of matrix multiplication algorithms, we are not need to perform input of given data in the master process and send them to other application processes, since the corresponding tasks in each process already provide the initial data associated with this process.

As an example of such a task, let us consider the MPI9Matr24 task, which is devoted to the step that must be performed immediately after the initial scattering the source data (in this case, we select the task from the subgroup associated with one of the block algorithms, namely Cannon's algorithm).

MPI9Matr24. Integers M_0 , P_0 , Q_0 and one-dimensional arrays filled with the corresponding blocks of matrices A , B , C are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr23 task). Implement the initial block redistribution used in the Cannon's algorithm for block matrix multiplication.

To do this, define a Cartesian topology for all processes as a two-dimensional periodic $K_0 \times K_0$ grid, where $K_0 \cdot K_0$ is equal to the number of processes (ranks of processes should not be reordered), and perform a cyclic shift of the blocks A_R given in all processes of each grid row I_0 by I_0 positions left (that is, in descending order of ranks of processes), $I_0 = 0, \dots, K_0 - 1$, and perform a cyclic shift of the blocks B_R given in all processes of each grid column J_0 by J_0 positions up (that is, in descending order of ranks of processes), $J_0 = 0, \dots, K_0 - 1$.

To create the MPI_COMM_GRID communicator associated with the Cartesian topology, use the Matr3CreateCommGrid function implemented in the MPI9Matr22 task. Use the MPI_Cart_coords, MPI_Cart_shift, MPI_Sendrecv_replace functions to perform the cyclic shifts (compare with MPI9Matr22).

Include all the above mentioned actions in a Matr3Init function (without parameters). Output the received blocks A_R and B_R in each process; perform data input and output in the Solve function.

The template for this task is as follows:

```
#include "mpi.h"
```

```

#include "pt4.h"
#include <cmath>

int k;          // number of processes
int r;          // rank of the current process

int m, p, q;    // sizes of the given matrices
int m0, p0, q0; // sizes of the matrix blocks
int k0;         // order of the Cartesian grid (equal to sqrt(k))

int *a_, *b_, *c_;
    // arrays to store matrices in the master process
int *a, *b, *c;
    // arrays to store matrix blocks in each process

MPI_Datatype MPI_BLOCK_A;
    // datatype for the block of the matrix A
MPI_Datatype MPI_BLOCK_B;
    // datatype for the block of the matrix B
MPI_Datatype MPI_BLOCK_C;
    // datatype for the block of the matrix C

MPI_Comm MPI_COMM_GRID = MPI_COMM_NULL;
// communicator associated with a two-dimensional Cartesian grid

void Solve()
{
    Task("MPI9Matr24");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    k = size;
    r = rank;
    k0 = (int)floor(sqrt((double)k) + 0.1);
}

```

The global variables already described in the template include not only the arrays `a`, `b`, `c` for storing the matrix blocks in each process and the variables `m0`, `p0`, `q0` that determine the sizes of these blocks, but also objects associated with the new datatypes and communicators used in Cannon's algorithm. Note that in the tasks for implementing block algorithms, the number of processes K is a perfect square: $K = K_0 \cdot K_0$; in this case, a special variable `k0` is provided for storing

the value of K_0 (the order of the Cartesian grid of processes), which already contains the required value (see the last statement in the Solve function).

When you launch the created template, a taskbook window similar to the one shown in Figs. 42–43 will appear on the screen (the first figure contains the initial part of the section with the input data, and the second one contains a section with an example of correct results). The data are distributed among 9 processes, K_0 (the order of the Cartesian grid) is 3.

```

Input data
Process 0:
M0= 3 P0= 3 Q0= 4
A0:
  6 -6  7
 -1 -2 -5
  5 -3  0
B0:
  2  7  3 -5
 -3 -4  6 -2
 -3 -5  7 -3
C0:
  0  0  0  0
  0  0  0  0
  0  0  0  0
Process 1:
M0= 3 P0= 3 Q0= 4
A1:
 -2  1  0
 -1  4 -2
  1  3 -2
B1:
 -6  6  0 -1
  0  0  2 -7
 -3  5  1 -4
C1:
  0  0  0  0
  0  0  0  0
  0  0  0  0
Process 2:
M0= 3 P0= 3 Q0= 4
A2:
 -7 -2  4
  0  4 -2
 -6  2  1
B2:
  2 -4 -7  0
  7 -1  6  0
 -2  6 -2  0
C2:
  0  0  0  0
  0  0  0  0
  0  0  0  0
Process 3:
M0= 3 P0= 3 Q0= 4
A3:
  3 -2 -5
  0  7 -3
  0  6 -5
B3:
  1  4  7 -7
  7 -6 -1 -2
 -2  1 -5 -2
C3:
  0  0  0  0
  0  0  0  0
  0  0  0  0
Process 4:
M0= 3 P0= 3 Q0= 4
A4:
 -4  0  3
 -3  3  7
  0  7  6
B4:
  2  1  7 -5
  4  7  5 -1
  0  4 -5  0
C4:
  0  0  0  0
  0  0  0  0
  0  0  0  0
Process 5:
M0= 3 P0= 3 Q0= 4
A5:
  7  2  7
  2  4 -3
  1  3  0
B5:
  6 -6 -5  0
 -4  0 -6  0
  0 -2 -5  0
C5:
  0  0  0  0
  0  0  0  0
  0  0  0  0
Process 6:
M0= 3 P0= 3 Q0= 4
A6:
  0 -3  7
  1  5  0
 -3  4 -3
B6:
 -3 -1 -1 -2
  1  6 -1 -6
 -1 -4 -5 -7
C6:
  0  0  0  0
  0  0  0  0
  0  0  0  0
Process 7:
M0= 3 P0= 3 Q0= 4
A7:
  5  2 -4
 -1  4 -3
  0 -6 -4
B7:
  2  3  2 -1
  7 -6  0  0
 -1 -5 -2 -7
C7:
  0  0  0  0
  0  0  0  0
  0  0  0  0
Process 8:
M0= 3 P0= 3 Q0= 4
A8:
 -4  0  4
  5  2  3
 -4 -7 -3
B8:
 -3  3 -7  0
 -4  5  2  0
  1 -2 -7  0
C8:
  0  0  0  0
  0  0  0  0
  0  0  0  0

```

Fig. 42. Acquaintance running of the MPI9Matr24 task (input data section)

```

Example of right solution
Process 0:          Process 1:          Process 2:
A0:                A1:                A2:
  6  -6  7          -2  1  0          -7  -2  4
 -1  -2  -5         -1  4  -2         0  4  -2
  5  -3  0          1  3  -2         -6  2  1
B0:                B1:                B2:
  2  7  3  -5       2  1  7  -5       -3  3  -7  0
 -3  -4  6  -2       4  7  5  -1       -4  5  2  0
 -3  -5  7  -3       0  4  -5  0       1  -2  -7  0
Process 3:          Process 4:          Process 5:
A3:                A4:                A5:
 -4  0  3           7  2  7           3  -2  -5
 -3  3  7           2  4  -3          0  7  -3
  0  7  6           1  3  0           0  6  -5
B3:                B4:                B5:
  1  4  7  -7       2  3  2  -1       2  -4  -7  0
  7  -6  -1  -2     7  -6  0  0       7  -1  6  0
 -2  1  -5  -2     -1  -5  -2  -7     -2  6  -2  0
Process 6:          Process 7:          Process 8:
A6:                A7:                A8:
 -4  0  4           0  -3  7           5  2  -4
  5  2  3           1  5  0          -1  4  -3
 -4  -7  -3         -3  4  -3          0  -6  -4
B6:                B7:                B8:
 -3  -1  -1  -2     -6  6  0  -1       6  -6  -5  0
  1  6  -1  -6       0  0  2  -7       -4  0  -6  0
 -1  -4  -5  -7     -3  5  1  -4       0  -2  -5  0

```

Fig. 43. Acquaintance running of the MPI9Matr24 task (example of the correct solution)

The actions required to input the given data are no different from those previously discussed in MPI9Matr1 and MPI9Matr2, but now they must be performed for each of the processes in the parallel application (the following fragment must be added to the end of the Solve function):

```

pt >> m0 >> p0 >> q0;
a = new int[m0*p0];
b = new int[p0*q0];
c = new int[m0*q0];
for (int i = 0; i < m0*p0; i++)
    pt >> a[i];
for (int i = 0; i < p0*q0; i++)
    pt >> b[i];
for (int i = 0; i < m0*q0; i++)
    pt >> c[i];

```

It should be noted that the contents of the blocks of matrix C are not used in this task, but they must be input, since all tasks associated with the matrix

product computation stage offer the same set of given data. This set coincides with the results obtained at the stage of scattering the initial data to all application processes (see the solution of the MPI9Matr2 task given in Section 1.4.3).

When you launch a new version of the program, a message will be displayed stating that all given data are input.

Let us turn to the implementation of the initial redistribution of blocks.

First of all, it is necessary to implement the auxiliary function `Matr3CreateCommGrid`, which creates a communicator with the topology of a two-dimensional square cyclic grid of order `k0`. This function is associated with a special task `MPI9Matr22`, which is, in fact, a simpler version of the task `MPI9Matr24`. Since we did not solve the `MPI9Matr22` task, the function `Matr3CreateCommGrid` has to be implemented during the solving our task, using the recommendations given in the `MPI9Matr22` task:

```
void Matr3CreateCommGrid(MPI_Comm &comm)
{
    int dims[] = {k0, k0},
        periods[] = {1, 1};
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
}
```

In the declaration of the `Matr3CreateCommGrid` function, we used a global variable `k0`, which contains the order of the Cartesian grid.

Now we describe the main function `Matr3Init`. First, it is necessary to create a communicator with the required Cartesian topology (note that the variable `MPI_COMM_GRID` associated with this communicator has already been declared in the template program). Then it is necessary to determine the coordinates of the current process in this communicator using the `MPI_Cart_coords` function. The coordinates will be stored in the auxiliary array `coord` with two elements: `coord[0]`, `coord[1]`. Using these coordinates, it is necessary to transfer the blocks of matrix *A* to `coord[0]` positions *to the left*, i. e. in the direction of decreasing *the second coordinate* of the grid (corresponding to the column numbers), taking into account its cyclicity along this coordinate, and the blocks of matrix *B* to `coord[1]` positions *up*, i. e. in the direction of decreasing *the first coordinate* of the grid (corresponding to the row numbers), also taking into account its cyclicity. To satisfy the conditions of the task, the ranks of the sending and receiving processes should be determined using the `MPI_Cart_shift` function, and the transfer itself should be carried out using the `MPI_Sendrecv_replace` function (by this function, the received block will be copied to the location of the block sent to another process):

```
void Matr3Init()
{
    Matr3CreateCommGrid(MPI_COMM_GRID);
    int coord[2];
    MPI_Cart_coords(MPI_COMM_GRID, r, 2, coord);
}
```

```

int src, dst;
MPI_Cart_shift(MPI_COMM_GRID, 1, -coord[0], &src, &dst);
MPI_Sendrecv_replace(a, m0*p0, MPI_INT, dst, 0, src, 0,
    MPI_COMM_GRID, MPI_STATUS_IGNORE);
MPI_Cart_shift(MPI_COMM_GRID, 0, -coord[1], &src, &dst);
MPI_Sendrecv_replace(b, p0*q0, MPI_INT, dst, 0, src, 0,
    MPI_COMM_GRID, MPI_STATUS_IGNORE);
}

```

In the `Matr3Init` function, we used, along with the local variables `coord`, `src` and `dst`, the global variables `MPI_COMM_GRID`, `r` (the rank of the current process), `a`, `b` (arrays with blocks of matrices A and B), and `m0`, `p0`, `q0` (the sizes of the block data).

It remains to call this function in the `Solve` function and output the new contents of the matrix blocks A and B in each process. Here is the final content of the `Solve` function (the added statements are highlighted in bold):

```

void Solve()
{
    Task("MPI9Matr24");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    k = size;
    r = rank;
    k0 = (int)floor(sqrt((double)k) + 0.1);
    pt >> m0 >> p0 >> q0;
    a = new int[m0*p0];
    b = new int[p0*q0];
    c = new int[m0*q0];
    for (int i = 0; i < m0*p0; i++)
        pt >> a[i];
    for (int i = 0; i < p0*q0; i++)
        pt >> b[i];
    for (int i = 0; i < m0*q0; i++)
        pt >> c[i];
    Matr3Init();
    for (int i = 0; i < m0*p0; i++)
        pt << a[i];
    for (int i = 0; i < p0*q0; i++)
        pt << b[i];
}

```

After launching this version of the program, a message will be displayed in the taskbook window stating that the task has been solved.

Note 1. The meaning of the initializing transfer of blocks of matrices A and B is that, as a result of this transfer, each process will receive a pair of blocks, the multiplication of which will determine the summand for the elements of the corresponding block of the matrix product C .

Note 2. The `Matr3Init` function implemented in this task will be used further in tasks `MPI9Matr28` and `MPI9Matr31`, which are devoted to the implementation of Cannon's algorithm in full.

1.4.5. Result gathering stage: an example of file-based output implementation

At the final stage of any matrix algorithm, it is necessary to combine the fragments of the final matrix product C obtained in different processes. The result of such a combination may be, for example, an array obtained in the master process and containing all the elements of the matrix C (which can be displayed on the screen or saved in a file). When implementing this stage using the MPI-2 library, you can immediately write fragments of the matrix C to a binary file, using the parallel file input-output tools that appeared in this standard (see Sections 1.3.2–1.3.3). This avoids additional actions associated with sending the resulting data to the master process.

As an example, let us consider the `MPI9Matr19` task, which requires saving in the resulting file the matrix product obtained using the second version of the band algorithm.

MPI9Matr19. Integers N_A , N_B and one-dimensional arrays filled with the $(N_A \cdot K) \times N_B$ bands C_R are given in each process (the given bands C_R are obtained as a result of K steps of the band algorithm of matrix multiplication—see the `MPI9Matr15` task). In addition, an integer M (the number of rows of the matrix product) and the name of file (to store this product) are given in the master process. The number of columns Q of the matrix product is a multiple of the number of processes K (and, therefore, is equal to $N_B \cdot K$).

Send the number M and the file name to all processes using the `MPI_Bcast` function. Write all the parts of the matrix product contained in the bands C_R to the resulting file, which will eventually contain a matrix C of the size $M \times Q$.

To write the bands to the file, set the appropriate file view using the `MPI_File_set_view` function and the `MPI_BAND_C` filetype defined with the `Matr2CreateTypeBand` function (see the `MPI9Matr11` task), and then use the `MPI_File_write_all` function.

Include all these actions (namely, the input of file name, sending number M and the file name, and writing all bands to the file) in a `Matr2GatherFile` function. Perform the input of all initial data, except the file name, in the `Solve` function.

Note. When writing data to the resulting file, it is necessary to take into account that the bands C_R may contain final zero-valued rows that are not re-

lated to the resulting matrix product (the number M should be sent to all processes in order to control this situation).

The template for this task contains the same global variables as the template for the MPI9Matr2 task given in Section 1.4.3, in particular, k (the number of processes), r (the rank of the current process), m , p , q , na , nb (the sizes of the matrices A , B , C and their bands), a , b , c (arrays for storing the bands of the matrices A , B , C in each process). In addition, the template includes the variables MPI_BAND_B and MPI_BAND_C intended for storing new datatypes associated with the vertical bands of the matrices B and C .

Fig. 44 shows the taskbook window that will appear when you launch the created template.

```

Programming Taskbook [C++]
BAND ALGORITHM 2 (HORIZONTAL AND VERTICAL BANDS)
Task: MPI9Matr19
Student: Smith John
Results (F2) Color (F3) Mode (F4)

Acquaintance with the task;
input-output operations were not executed. Exit (Esc)

Input: 0 of 80 Tests: 1 of 5

Input data
Process 0: NA = 2 NB = 4
C0:
    38  83  53 -20
    19 -46  19 -42
    64   9 -15 -12
    42  16 -55  59
   101  13  -3 -42
     0   0   0   0
M = 5
Name of file for matrix C: "Ca09r8.tst"
Process 1: NA = 2 NB = 4
C1:
   -22  39  -2 -18
    30 -67   3  12
     -3  -8  26  28
   -16  13  76 -39
     -7  -1  37   7
     0   0   0   0
Process 2: NA = 2 NB = 4
C2:
    29  31 -23   6
    52 -18 -35 -11
     9  -7 -12  11
    47 -52   6  41
    35   9 -48  32
     0   0   0   0

Example of right solution
File with matrix C:
 1:  38  83  53 -20 -22  39  -2 -18  29  31 -23   6
13:  19 -46  19 -42  30 -67   3  12  52 -18 -35 -11
25:  64   9 -15 -12  -3  -8  26  28   9  -7 -12  11
37:  42  16 -55  59 -16  13  76 -39  47 -52   6  41
49: 101  13  -3 -42  -7  -1  37   7  35   9 -48  32

```

Fig. 44. Acquaintance running of the MPI9Matr19 task

This task does not require outputting the results using the `pt` output stream; it only requires writing the elements of the resulting matrix C to a binary integer file (note that the indicators section of the taskbook window does not specify the number of data elements to be output). The taskbook displays the contents of the created file and checks if it is correct. Note that the master process specifies additional input data that are absent in the slave processes, namely, the number of rows M of the resulting matrix C and the name of the file for storing this matrix. It should also be noted that the bands of the matrix C given in each process contain not only its elements, but also "extra" zero rows that should not be saved in the resulting file (the presence of these rows is explained by the fact that the product $N_A \cdot K$, where K is the number of processes, in this case *exceeds* the number of rows M of the matrix C).

At the first stage of the solution, as usual, it is necessary to organize the input of the given data. The task condition states that all the given data, except for the file name (which is given in the master process), must be input in the `Solve` function. It should also be taken into account that the number of rows M of the resulting matrix C is also given only in the master process. The file name should be input at the beginning of the `Matr2GatherFile` function. The requirement to input all the data, except for the file name, in the `Solve` function is related to the fact that when implementing the matrix multiplication algorithm in full, all this data will already have been input (or calculated) in the corresponding processes *before* calling the `Matr2GatherFile` function, so their repeated input in the `Matr2GatherFile` function would lead to incorrect execution of the program.

So, let us implement the `Matr2GatherFile` function, in which we input the file name in the master process, and add to the `Solve` function a fragment containing the input of other data and the call to the `Matr2GatherFile` function (the `Matr2GatherFile` function and the statements added to the `Solve` function are highlighted in bold):

```
void Matr2GatherFile()
{
    char cname[12];
    if (r == 0)
        pt >> cname;
}

void Solve()
{
    Task("MPI9Matr19");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
k = size;
r = rank;
pt >> na >> nb;
c = new int[na*k*nb];
for (int i = 0; i < na*k*nb; i++)
    pt >> c[i];
if (r == 0)
    pt >> m;
Matr2GatherFile();
}

```

When declaring the file name `cname`, we took into account that this name consists of no more than 11 characters (see the preamble to the MPI9Matr task group, Section 2.9).

After running this version of the program, we will receive a message that all the initial data are input, but the resulting file has not been created.

The remaining actions will be implemented in the `Matr2GatherFile` function. First of all, it is necessary to transfer the number M and the file name to all processes:

```

MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(cname, 12, MPI_CHAR, 0, MPI_COMM_WORLD);

```

Then you need to declare the file variable `f` and open the file with the given name using the `MPI_File_open` function. It is advisable to immediately add a function for closing this file at the end of the `Matr2GatherFile` function:

```

MPI_File f;
MPI_File_open(MPI_COMM_WORLD, cname,
    MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);

MPI_File_close(&f);

```

All actions with the file must be performed between calls to the `MPI_File_open` and `MPI_File_close` functions.

Running this version of the program will result in a new message which, in addition to informing you that all the input data are input, will also note that the resulting file is empty. This indicates that the transfer of the file name to all processes and the actions to create and close it in each process have been successfully completed.

It remains to ensure that the bands of the resulting matrix are correctly written to the created file. To do this, it is necessary to define a special file datatype, since each process must write to the file not a set of sequentially located elements (as would be the case if the process contained matrix *rows*), but several fragments with empty spaces (each fragment corresponds to the elements of a separate row, and the spaces provide a transition to the corresponding elements of the next row). Each band C_R has the size $(N_A \cdot K) \times N_B$; thus, it contains N_B ad-

adjacent columns. This means that the fragment for each row must contain N_B adjacent elements. What is the number of such fragments? It would seem that it should be equal to the number of rows in the band C_R , i. e. the value $N_A \cdot K$, but we should not forget that the number of rows in the band can be *greater* than the number of rows M in the resulting matrix (the "extra" rows are zero and should not be written to the resulting file). Thus, the number of fragments written to the file should be equal to M .

So, each process must write M fragments into the file with N_B consecutive integers in each fragment, and the distance between the initial positions of adjacent fragments must be equal to Q , i. e. the number of columns of the resulting matrix (this will lead to the fact that the fragments written into the file by each process will form columns of the resulting matrix).

Such compound data types (associated with a set of adjacent matrix *columns*) are also required in the implementation of other stages of the band algorithm using vertical bands. In particular, such a data type simplifies scattering the bands of the given matrix B (which are also sets of columns) and gathering the bands of the resulting matrix C in the master process (if the algorithm does not use file input-output). Therefore, it would be useful to describe an auxiliary function `Matr2CreateTypeBand` for the creation of types associated with vertical bands.

The implementation of the `Matr2CreateTypeBand` function is the subject of a special task `MPI9Matr11`, which is the first in a series of tasks related to the second variant of the band algorithm (see Section 2.9.3). Since we did not solve the `MPI9Matr11` task, the `Matr2CreateTypeBand` function has to be implemented when solving our task, using the recommendations given in the `MPI9Matr11` task (recall that we encountered a similar situation earlier in Section 1.4.4 when solving the `MPI9Matr24` task, in which we had to implement the auxiliary `Matr3CreateCommGrid` function, which defines a new communicator with a Cartesian topology).

As stated in `MPI9Matr11`, to implement a derived datatype associated with matrix columns, the `MPI_Type_vector` function should be used. Let us recall the meaning of the first three parameters of this function (all of them are of integer type):

- count – the number of blocks containing consecutive elements;
- blocklen – the size of each block (in elements);
- stride – the distance between the starting positions of adjacent blocks (also in elements).

The next parameter `oldtype` defines the base type of the elements (in our case, it is `MPI_INT`), and the last parameter `newtype` is an output parameter: it is a pointer to the created datatype.

When implementing the auxiliary function `Matr2CreateTypeBand`, it is more convenient to use parameters related to the characteristics of vertical bands. In

the MPI9Matr11 task, it is proposed to use the parameters p , n , q , where p and n determine the number of rows and columns of the band, and q is the number of columns of the matrix from which this band is extracted. It is easy to see that these parameters exactly correspond to the first three parameters of the MPI_Type_vector function. Therefore, we obtain the following implementation of the Matr2CreateTypeBand function:

```
void Matr2CreateTypeBand(int p, int n, int q, MPI_Datatype &t)
{
    MPI_Type_vector(p, n, q, MPI_INT, &t);
    MPI_Type_commit(&t);
}
```

The MPI_Type_commit function call included in this function is required if the new datatype must be used in data transfer. When solving the MPI9Matr19 task, the call to MPI_Type_commit is not required, but we have provided this implementation of the Matr2CreateTypeBand function because it can be used in all tasks associated with the second variant of the band algorithm.

Having implemented the Matr2CreateTypeBand function, let us define the file view. The MPI_File_set_view function, which was previously discussed in detail in Section 1.3.3, is intended for this definition. To define the file view, we must specify the initial offset disp (in bytes), the elementary type etype (in our case, MPI_INT), and the file datatype filetype, which can be obtained using the Matr2CreateTypeBand function. As for the initial offset, it should be equal to $\text{int_sz} * \text{nb} * r$, where r is the process rank, int_sz is the size of the base type MPI_INT in bytes, and nb is the number of columns in each band. The easiest way to determine the value of int_sz is to use the MPI_Type_size function. Thus, the fragment of the Matr2GatherFile function associated with defining the file view will look as follows:

```
Matr2CreateTypeBand(m, nb, nb*k, MPI_BAND_C);
int int_sz;
MPI_Type_size(MPI_INT, &int_sz);
MPI_File_set_view(f, int_sz*nb*r, MPI_INT, MPI_BAND_C,
    "native", MPI_INFO_NULL);
```

Recall that the variable MPI_BAND_C does not need to be declared, since it is already declared in our program as a global variable. When defining the type MPI_BAND_C, we took into account that the vertical band with elements of the matrix C (without terminating zero rows) has a size of $M \times N_B$ in each process, and the number of columns Q of the matrix C is equal to the product $N_B \cdot K$, where K is the number of processes.

Once the file view has been defined, we can implement writing all elements of each band to the file with *a single call* to the collective function MPI_File_write_all:

```
MPI_File_write_all(f, c, m*nb, MPI_INT, MPI_STATUS_IGNORE);
```

In this call, we specify the array containing the band and the number of its elements that need to be written to the file (recall that the array can also contain terminating zero elements that do not need to be written to the file).

Let us present the final version of the `Matr2GatherFile` function. In this version, we highlight in bold the operators associated with defining the file view and writing data to the file:

```
void Matr2GatherFile()
{
    char cname[12];
    if (r == 0)
        pt >> cname;
    MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(cname, 12, MPI_CHAR, 0, MPI_COMM_WORLD);
    MPI_File f;
    MPI_File_open(MPI_COMM_WORLD, cname,
        MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);
    Matr2CreateTypeBand(m, nb, nb*k, MPI_BAND_C);
    int int_sz;
    MPI_Type_size(MPI_INT, &int_sz);
    MPI_File_set_view(f, int_sz*nb*r, MPI_INT, MPI_BAND_C,
        "native", MPI_INFO_NULL);
    MPI_File_write_all(f, c, m*nb, MPI_INT, MPI_STATUS_IGNORE);
    MPI_File_close(&f);
}
```

After launching this version of the program, a message will be displayed in the taskbook window stating that the task has been solved.

1.5. Additional techniques for developing parallel programs

1.5.1. Debugging parallel programs using taskbook tools

The examples of solving parallel programming tasks discussed above show that using the taskbook ensures a significant acceleration of parallel program development. This is achieved primarily due to a special mechanism that ensures a quick launch of a parallel program directly from the integrated environment. In addition, debugging a parallel program is simplified by using the `Show` function, which ensures displaying any required data from various processes in the taskbook window and subsequently viewing the resulting debug information for both a separate process and all processes of the parallel program.

The noted features of the taskbook may be useful not only in solving tasks on parallel MPI programming, but also in developing general-purpose parallel programs. Therefore, along with the "usual" groups of tasks that allow you to study the capabilities of the MPI library, the Programming Taskbook for MPI-2 includes a special group `MPIDebug`, intended not for solving specific tasks, but for debugging *any* parallel program. This group consists of 36 "tasks", and task

number N ensures the launch of a parallel program with N processes. Thus, if you create a template program for one of the tasks of the MPIDebug group, then when you launch this program from the integrated environment, it will automatically be executed in parallel mode with the required number of processes.

Let us use MPIDebug group to develop a program that implements one of the parallel algorithms for multiplying a matrix A by a vector b , the so-called *self-scheduling algorithm* (see [10, Chapter 7]), in which the master process coordinates the work of slave processes, sending them the initial data sets and receiving results.

The initial data are input in the master process. Then the master process sends the vector b and one row of the matrix A to each slave process. After this, a loop is started in which the master process receives the results of multiplying a row of the matrix by a vector from the slave processes and sends them the remaining rows of the matrix A . The loop ends when the master process sends all the rows of the matrix A to the slave processes and receives all the elements of the resulting vector Ab from them.

It should be noted that when implementing this algorithm, it is impossible to determine in advance which matrix rows will be processed by a particular slave process. If, for example, the program uses 5 processes, then matrix row number 5 will be sent to the process (of slave processes with ranks from 1 to 4) that first completes the multiplication of the row sent to it earlier and returns the result to the master process.

To test the algorithm, we will process a matrix A of order $N = 20$, each line of which contains identical elements equal to the ordinal numbers of their rows: $a_{ij} = i$ for $i, j = 1, \dots, N$. Vector b of the same size N will contain identical elements equal to 0.01. Thus, the elements of vector $c = Ab$ will have the form $c_i = 0.2i$, $i = 1, \dots, N$:

$$c = (0.2, 0.4, 0.6, \dots, 3.8, 4.0)$$

For the algorithm to work correctly, it is necessary that the number of slave processes does not exceed the order of the matrix N , since at the initial stage of the algorithm, each slave process is sent one of the rows of this matrix. For certainty, we choose the number of processes of our program equal to 10; in order to ensure automatic launch of a parallel program with this number of processes, we should use the MPIDebug10 task.

When you run the template program for the MPIDebug10 task, the taskbook window will look like the one shown in Fig. 45. Instead of the initial and resulting data, this window displays a description of those taskbook tools that are intended for outputting debug information.

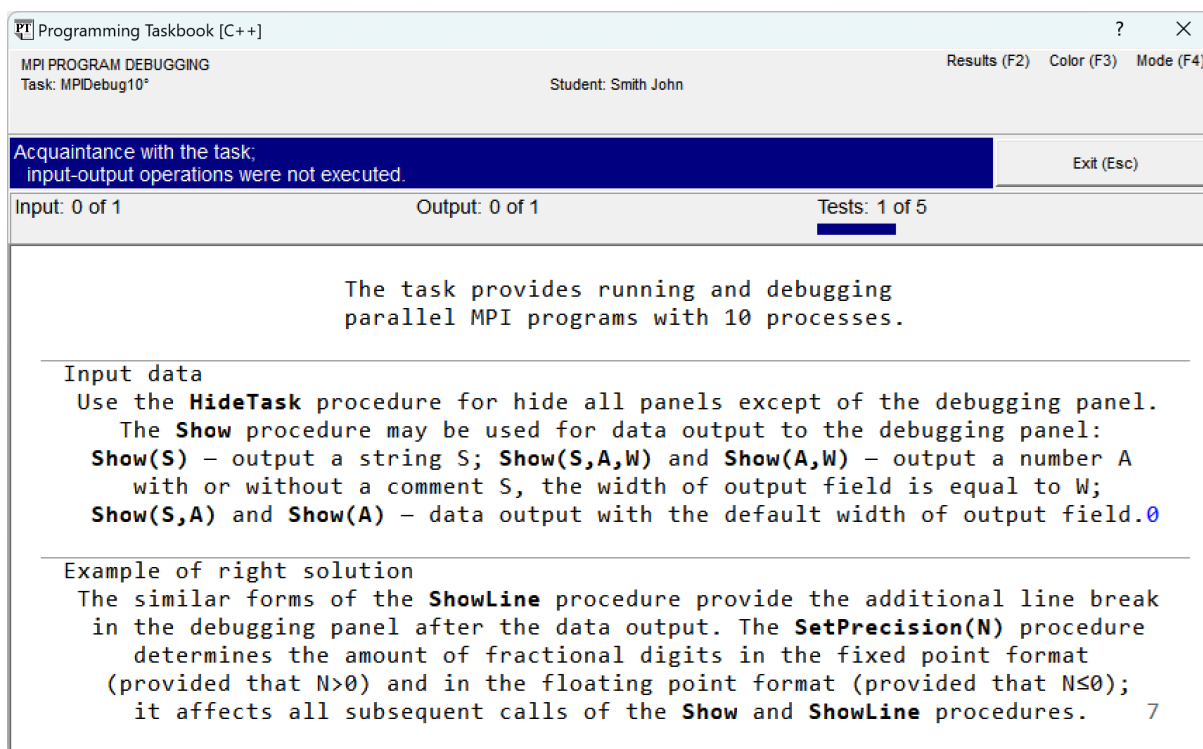


Fig. 45. Acquaintance running of the MPIDebug10 task

Let us supplement the created template with a fragment that describes all the necessary variables (including those that will be required at the next stages of the algorithm implementation). Then the creation of the initial matrix A and vector b is performed in the master process, and vector b is sent to all slave processes (added operators are highlighted in bold):

```
void Solve()
{
    Task("MPIDebug10");
    int flag;
    MPI_Initialized(&flag);
    if (flag == 0)
        return;
    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    const int n = 20;
    double a[n][n], b[n], c[n], d;
    int cur_row;
    MPI_Status s;
    // Creation of initial data in the master process:
    if (rank == 0)
        for (int i = 0; i < n; i++)
        {
            b[i] = 0.01;
            for (int j = 0; j < n; j++)
```

```

        a[i][j] = i + 1;
    }
    // Sending vector b to all processes:
    MPI_Bcast(b, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    ShowLine("Vector b:..., ", b[n-1]);
}

```

A two-dimensional static array (array of arrays) is used to store the matrix, since the order of the matrix ($N = 20$) is known in advance.

When you launch a new version of the program, a debug section will appear in the taskbook window (Fig. 46).

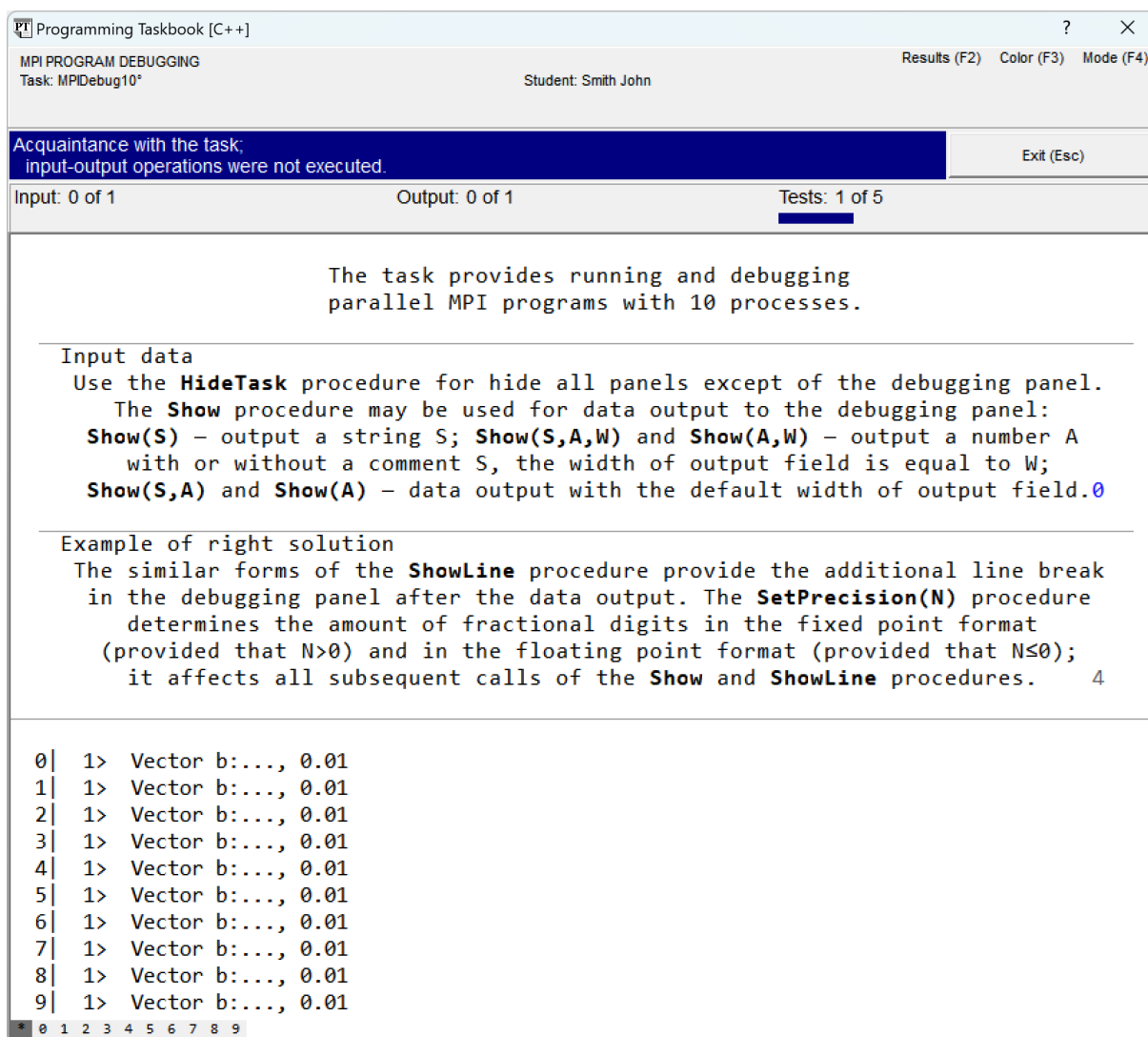



Fig. 46. Debug data output

Since the new version of the program uses calls to the ShowLine function, the taskbook window displays a debug section with information that was output using this function. In our case, each of the processes (including the master process) output information about successful creation of the vector b. For brevi-

ty, only the value of *the last element* of the *b* array (with index $n - 1$) is output. To emphasize that the only last element is displayed, an ellipsis is placed before it.

Since the main sections of the taskbook window in our case contain only reference information, it is convenient *to hide* all sections of the window except the debug section. If the taskbook window is already displayed on the screen, then to do this, it is enough to press the key combination [Ctrl]+[Space]. However, it is even more convenient to automatically hide unnecessary sections when displaying the taskbook window on the screen. To do this, it is enough to call the auxiliary function `HideTask` anywhere in the program. If we add a call to this function to the end of our program and re-run it, then the taskbook window will contain only the debug section (Fig. 47).



```

0| 1> Vector b:..., 0.01
1| 1> Vector b:..., 0.01
2| 1> Vector b:..., 0.01
3| 1> Vector b:..., 0.01
4| 1> Vector b:..., 0.01
5| 1> Vector b:..., 0.01
6| 1> Vector b:..., 0.01
7| 1> Vector b:..., 0.01
8| 1> Vector b:..., 0.01
9| 1> Vector b:..., 0.01

```

Fig. 47. Hiding the main sections of the taskbook window when displaying debug data

Let us return to the implementation of the matrix-vector multiplication algorithm and present its final part:

```

if (rank == 0)
{
    // Sending the initial rows of matrix A to slave processes:
    for (int i = 1; i < size; i++)
    {
        MPI_Send(a[i-1], n, MPI_DOUBLE, i, i-1, MPI_COMM_WORLD);
        Show("Sending initial data: dest=", i);
        Show("tag=", i - 1);
        ShowLine("->..., ", a[i - 1][0]);
    }
    cur_row = size - 2;
    // Receiving the elements of the product
    // and sending the remaining rows:
    for (int i = 0; i < n; i++)
    {
        MPI_Recv(&d, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &s);
        Show("Receiving result: source=", s.MPI_SOURCE);
        Show("tag=", s.MPI_TAG, 2);
        Show("<- ", d);
        c[s.MPI_TAG] = d;
    }
}

```

```

        cur_row++;
        if (cur_row < n)
            MPI_Send(a[cur_row], n, MPI_DOUBLE, s.MPI_SOURCE,
                    cur_row, MPI_COMM_WORLD);
        else
            MPI_Send(&d, 1, MPI_DOUBLE, s.MPI_SOURCE,
                    cur_row, MPI_COMM_WORLD);
        Show("| Sending data: dest=", s.MPI_SOURCE);
        ShowLine(" tag = ", cur_row, 2);
    }
    // Output of the obtained vector Ab:
    SetPrecision(1);
    ShowLine("Resulting vector Ab:");
    for (int i = 0; i < n; i++)
        Show(c[i], 3);
}
else
    // Data processing in slave processes:
    while (true)
    {
        MPI_Recv(c, n, MPI_DOUBLE, 0, MPI_ANY_TAG,
                MPI_COMM_WORLD, &s);
        Show("Receiving data: tag=", s.MPI_TAG, 2);
        if (s.MPI_TAG >= n)
            break;
        d = 0;
        for (int i = 0; i < n; i++)
            d += c[i] * b[i];
        MPI_Send(&d, 1, MPI_DOUBLE, 0, s.MPI_TAG,
                MPI_COMM_WORLD);
        Show("| Sending result: tag=", s.MPI_TAG, 2);
        ShowLine ("->", d);
    }
}

```

This part consists of two large fragments, the first of which must be executed in the master process and the second must be executed in each of the slave processes.

The master process first sends one row of the given matrix to each slave process and initializes the variable `cur_row`, which contains the index of the last processed (i. e., sent to the slave process) matrix row; rows are indexed from zero. After this, the main loop is started, in which the master process receives the resulting elements of the product from the slave processes and sends them the next rows of the matrix.

Since the order of receiving data from slave processes is not determined, the source parameter of the `MPI_Recv` function is set to `MPI_ANY_SOURCE`, which ensures receiving data from *any* process that sent them. In the message tag, the slave process passes additional information to the master process: *the index of*

the calculated product element. In order for the master process to be able to receive messages with any tags, the `msgtag` parameter of the `MPI_Recv` function is set to `MPI_ANY_TAG`. The master process receives information about the rank of the process that sent the message and about the message tag using `MPI_SOURCE` and `MPI_TAG` fields of the structure `s` of type `MPI_Status`.

After receiving a message from a slave process, the master process sends it a new message, which either contains the next unprocessed row of the matrix (if such rows still remain) or a special termination indicator. If the next row is sent, then the tag of the message being sent is assumed to be equal to the row index `cur_row`; if the termination indicator is sent, then the tag contains a number that *is greater than the maximum matrix row index*.

After the for loop completes, all elements of the calculated product are output in the master process.

In the slave process, a loop is started, in which the next row of the matrix is received from the master process (its index can be obtained from the message tag), then the received matrix row is multiplied by the previously received vector, and finally, the result of the multiplication is sent to the master process. If, when receiving the next message from the master process, a tag is received that exceeds the number of the maximum index of the matrix row, then an immediate exit from the loop occurs, and the slave process completes its work.

All stages of the algorithm described above are accompanied by debug output. For this, the `Show` function and its `ShowLine` version are used (the `ShowLine` function provides a transition to a new line in the debug section after data output). Before outputting the next numeric element, as a rule, a string comment preceding it is output; the amount of screen positions that should be used for data output is additionally indicated (this ensures *vertical alignment of the output data*).

Before the output of the elements of the resulting product in the master process, the `SetPrecision(1)` function is called, ensuring that all subsequent data of the real type is output with *one* digit after the decimal separator. Note that by default the number of fractional digits is assumed to be equal to 2, but using this value it would not be possible to output all the resulting elements in one line, since the width of the output data area in the debug section is equal to 80 positions.

When formatting the output, it was taken into account that the data output by the `Show` or `ShowLine` function is separated by one space from the previously output data located on the same screen line.

Let us present the debug data that was obtained when the program was launched (recall that before the first symbol `|` the rank of the process that output the debug information is indicated, and before the symbol `>` the ordinal number of the output line is indicated; in this case, the lines associated with different processes are numbered independently).

```

0| 1> Vector b: ..., 0.01
0| 2> Sending initial data: dest=1 tag=0 -> ..., 1.00
0| 3> Sending initial data: dest=2 tag=1 -> ..., 2.00
0| 4> Sending initial data: dest=3 tag=2 -> ..., 3.00
0| 5> Sending initial data: dest=4 tag=3 -> ..., 4.00
0| 6> Sending initial data: dest=5 tag=4 -> ..., 5.00
0| 7> Sending initial data: dest=6 tag=5 -> ..., 6.00
0| 8> Sending initial data: dest=7 tag=6 -> ..., 7.00
0| 9> Sending initial data: dest=8 tag=7 -> ..., 8.00
0| 10> Sending initial data: dest=9 tag=8 -> ..., 9.00
0| 11> Receiving result: source=1 tag= 0 <- 0.20 | Sending data: dest=1 tag= 9
0| 12> Receiving result: source=2 tag= 1 <- 0.40 | Sending data: dest=2 tag=10
0| 13> Receiving result: source=3 tag= 2 <- 0.60 | Sending data: dest=3 tag=11
0| 14> Receiving result: source=1 tag= 9 <- 2.00 | Sending data: dest=1 tag=12
0| 15> Receiving result: source=2 tag=10 <- 2.20 | Sending data: dest=2 tag=13
0| 16> Receiving result: source=3 tag=11 <- 2.40 | Sending data: dest=3 tag=14
0| 17> Receiving result: source=2 tag=13 <- 2.80 | Sending data: dest=2 tag=15
0| 18> Receiving result: source=1 tag=12 <- 2.60 | Sending data: dest=1 tag=16
0| 19> Receiving result: source=3 tag=14 <- 3.00 | Sending data: dest=3 tag=17
0| 20> Receiving result: source=1 tag=16 <- 3.40 | Sending data: dest=1 tag=18
0| 21> Receiving result: source=2 tag=15 <- 3.20 | Sending data: dest=2 tag=19
0| 22> Receiving result: source=3 tag=17 <- 3.60 | Sending data: dest=3 tag=20
0| 23> Receiving result: source=1 tag=18 <- 3.80 | Sending data: dest=1 tag=21
0| 24> Receiving result: source=2 tag=19 <- 4.00 | Sending data: dest=2 tag=22
0| 25> Receiving result: source=6 tag= 5 <- 1.20 | Sending data: dest=6 tag=23
0| 26> Receiving result: source=7 tag= 6 <- 1.40 | Sending data: dest=7 tag=24
0| 27> Receiving result: source=8 tag= 7 <- 1.60 | Sending data: dest=8 tag=25
0| 28> Receiving result: source=9 tag= 8 <- 1.80 | Sending data: dest=9 tag=26
0| 29> Receiving result: source=5 tag= 4 <- 1.00 | Sending data: dest=5 tag=27
0| 30> Receiving result: source=4 tag= 3 <- 0.80 | Sending data: dest=4 tag=28
0| 31> Resulting vector Ab:
0| 32> 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0
1| 1> Vector b: ..., 0.01
1| 2> Receiving data: tag= 0 | Sending result: tag= 0 -> 0.20
1| 3> Receiving data: tag= 9 | Sending result: tag= 9 -> 2.00
1| 4> Receiving data: tag=12 | Sending result: tag=12 -> 2.60
1| 5> Receiving data: tag=16 | Sending result: tag=16 -> 3.40
1| 6> Receiving data: tag=18 | Sending result: tag=18 -> 3.80
1| 7> Receiving data: tag=21
2| 1> Vector b: ..., 0.01
2| 2> Receiving data: tag= 1 | Sending result: tag= 1 -> 0.40
2| 3> Receiving data: tag=10 | Sending result: tag=10 -> 2.20
2| 4> Receiving data: tag=13 | Sending result: tag=13 -> 2.80
2| 5> Receiving data: tag=15 | Sending result: tag=15 -> 3.20
2| 6> Receiving data: tag=19 | Sending result: tag=19 -> 4.00
2| 7> Receiving data: tag=22
3| 1> Vector b: ..., 0.01
3| 2> Receiving data: tag= 2 | Sending result: tag= 2 -> 0.60
3| 3> Receiving data: tag=11 | Sending result: tag=11 -> 2.40
3| 4> Receiving data: tag=14 | Sending result: tag=14 -> 3.00
3| 5> Receiving data: tag=17 | Sending result: tag=17 -> 3.60
3| 6> Receiving data: tag=20
4| 1> Vector b: ..., 0.01
4| 2> Receiving data: tag= 3 | Sending result: tag= 3 -> 0.80
4| 3> Receiving data: tag=28
5| 1> Vector b: ..., 0.01
5| 2> Receiving data: tag= 4 | Sending result: tag= 4 -> 1.00
5| 3> Receiving data: tag=27
6| 1> Vector b: ..., 0.01
6| 2> Receiving data: tag= 5 | Sending result: tag= 5 -> 1.20
6| 3> Receiving data: tag=23
7| 1> Vector b: ..., 0.01
7| 2> Receiving data: tag= 6 | Sending result: tag= 6 -> 1.40
7| 3> Receiving data: tag=24

```

```

8| 1> Vector b: ..., 0.01
8| 2> Receiving data: tag= 7 | Sending result: tag= 7 -> 1.60
8| 3> Receiving data: tag=25
9| 1> Vector b: ..., 0.01
9| 2> Receiving data: tag= 8 | Sending result: tag= 8 -> 1.80
9| 3> Receiving data: tag=26

```

It is clear from the given text that during the execution of the algorithm, processes of ranks 1 and 2 processed 5 rows of the given matrix, the process of rank 3 processed 4 rows, and the remaining 6 slave processes processed 1 row each. The last, 32nd row, associated with the master process, contains the elements of the found product.

Note that the text displayed in the debug section of the taskbook window can be copied to the Windows clipboard; to do this, simply press the standard key combination [Ctrl]+[C] or call the window's pop-up menu (by mouse right-clicking) and execute its command **Copy the debug data to clipboard**.

By changing the task name in the Task function to "MPIDebug5", we can test the developed algorithm in parallel mode using 5 processes. Here is the debug information that was output when running the program in this way:

```

0| 1> Vector b: ..., 0.01
0| 2> Sending initial data: dest=1 tag=0 -> ..., 1.00
0| 3> Sending initial data: dest=2 tag=1 -> ..., 2.00
0| 4> Sending initial data: dest=3 tag=2 -> ..., 3.00
0| 5> Sending initial data: dest=4 tag=3 -> ..., 4.00
0| 6> Receiving result: source=4 tag= 3 <- 0.80 | Sending data: dest=4 tag= 4
0| 7> Receiving result: source=2 tag= 1 <- 0.40 | Sending data: dest=2 tag= 5
0| 8> Receiving result: source=3 tag= 2 <- 0.60 | Sending data: dest=3 tag= 6
0| 9> Receiving result: source=4 tag= 4 <- 1.00 | Sending data: dest=4 tag= 7
0| 10> Receiving result: source=1 tag= 0 <- 0.20 | Sending data: dest=1 tag= 8
0| 11> Receiving result: source=2 tag= 5 <- 1.20 | Sending data: dest=2 tag= 9
0| 12> Receiving result: source=3 tag= 6 <- 1.40 | Sending data: dest=3 tag=10
0| 13> Receiving result: source=4 tag= 7 <- 1.60 | Sending data: dest=4 tag=11
0| 14> Receiving result: source=1 tag= 8 <- 1.80 | Sending data: dest=1 tag=12
0| 15> Receiving result: source=2 tag= 9 <- 2.00 | Sending data: dest=2 tag=13
0| 16> Receiving result: source=3 tag=10 <- 2.20 | Sending data: dest=3 tag=14
0| 17> Receiving result: source=4 tag=11 <- 2.40 | Sending data: dest=4 tag=15
0| 18> Receiving result: source=2 tag=13 <- 2.80 | Sending data: dest=2 tag=16
0| 19> Receiving result: source=4 tag=15 <- 3.20 | Sending data: dest=4 tag=17
0| 20> Receiving result: source=2 tag=16 <- 3.40 | Sending data: dest=2 tag=18
0| 21> Receiving result: source=3 tag=14 <- 3.00 | Sending data: dest=3 tag=19
0| 22> Receiving result: source=1 tag=12 <- 2.60 | Sending data: dest=1 tag=20
0| 23> Receiving result: source=4 tag=17 <- 3.60 | Sending data: dest=4 tag=21
0| 24> Receiving result: source=2 tag=18 <- 3.80 | Sending data: dest=2 tag=22
0| 25> Receiving result: source=3 tag=19 <- 4.00 | Sending data: dest=3 tag=23
0| 26> Resulting vector Ab:
0| 27> 0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0
1| 1> Vector b: ..., 0.01
1| 2> Receiving data: tag= 0 | Sending result: tag= 0 -> 0.20
1| 3> Receiving data: tag= 8 | Sending result: tag= 8 -> 1.80
1| 4> Receiving data: tag=12 | Sending result: tag=12 -> 2.60
1| 5> Receiving data: tag=20
2| 1> Vector b: ..., 0.01
2| 2> Receiving data: tag= 1 | Sending result: tag= 1 -> 0.40
2| 3> Receiving data: tag= 5 | Sending result: tag= 5 -> 1.20
2| 4> Receiving data: tag= 9 | Sending result: tag= 9 -> 2.00
2| 5> Receiving data: tag=13 | Sending result: tag=13 -> 2.80
2| 6> Receiving data: tag=16 | Sending result: tag=16 -> 3.40
2| 7> Receiving data: tag=18 | Sending result: tag=18 -> 3.80

```

```

2| 8> Receiving data: tag=22
3| 1> Vector b: ..., 0.01
3| 2> Receiving data: tag= 2 | Sending result: tag= 2 -> 0.60
3| 3> Receiving data: tag= 6 | Sending result: tag= 6 -> 1.40
3| 4> Receiving data: tag=10 | Sending result: tag=10 -> 2.20
3| 5> Receiving data: tag=14 | Sending result: tag=14 -> 3.00
3| 6> Receiving data: tag=19 | Sending result: tag=19 -> 4.00
3| 7> Receiving data: tag=23
4| 1> Vector b: ..., 0.01
4| 2> Receiving data: tag= 3 | Sending result: tag= 3 -> 0.80
4| 3> Receiving data: tag= 4 | Sending result: tag= 4 -> 1.00
4| 4> Receiving data: tag= 7 | Sending result: tag= 7 -> 1.60
4| 5> Receiving data: tag=11 | Sending result: tag=11 -> 2.40
4| 6> Receiving data: tag=15 | Sending result: tag=15 -> 3.20
4| 7> Receiving data: tag=17 | Sending result: tag=17 -> 3.60
4| 8> Receiving data: tag=21

```

1.5.2. Developing and running parallel programs without the taskbook

The example given in the previous section shows that the use of the Programming Taskbook for MPI-2 simplifies the development of parallel programs not related to solving tasks.

However, our ultimate goal is to create parallel programs whose execution does not depend on the presence of the taskbook on the computer. In this section, we will describe in detail the process of creating such programs in the **Microsoft Visual Studio** environment. In a similar way, parallel programs can be developed in other environments on a local computer (for example, in the **Code::Blocks** or **Dev-C++** environment). Of course, parallel programs are usually intended to be run on a supercomputer or a computing cluster, but if you have a parallel program created and tested on a local computer, then transferring it to another computing platform is quite simple and requires only knowledge of how to compile and run the program on this platform (see, for example, Chapter 4 in the book [4], which describes the steps that allow you to compile a parallel program on a Unix server using **Intel** and **GCC** compilers and run it on a set of cluster nodes using the **PBS** job management system).

As before, we assume that the program is being developed in the **Microsoft Visual Studio 2022** environment. Create a new console application in this environment by following the standard steps:

- start with the menu command **File | New | Project...**;
- in the right-hand part of the **Create a new project** window, select the following values in the comboboxes: **C++**, **Windows**, **Console**, then select the **Console App** project option and click the **Next** button;
- configure the properties of the project being created: **Name** (we assume that the name `ParallelApplication1` is specified) and the top-level directory **Location** (we assume that the working directory `C:\PT4Work` of the taskbook is selected); in addition, check the **Place solution and project in the same directory** checkbox to avoid creating an extra level of directories, and finally click the **Create** button.

Other versions of Visual Studio may require slightly different steps. Let us describe the steps required to create a console application in **Visual Studio 2017**:

- start with the menu command **File | New | Project...**;
- in the **New Project** window, select the **Visual C++** section and set the **Win 32 Console Application** project type in it;
- configure the properties of the project being created: **Name** (ParallelApplication1) and the top-level directory **Location** (C:\PT4Work); in addition, uncheck the **Create directory for solution** checkbox to avoid creating an extra level of directories;
- after setting up the project properties, click **OK** and set up additional properties of the console application in the **Win32 Application Wizard** window: immediately click the **Next** button in this window; in the **Application Settings** section, leave the **Console application** option selected and *uncheck* all the checkboxes in the **Additional options** group (in particular, uncheck the **Precompiled header** checkbox), then click **Finish**.

As a result, the ParallelApplication1 project will be created, and the main file of this project, ParallelApplication1.cpp, will be loaded into the code editor with the following contents (we do not specify additional comments that are included in this file):

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}
```

In other versions of Visual Studio, the contents of the main file may differ from the above. For example, in **Visual Studio 2017**, the file contains the following statements:

```
#include "stdafx.h"

int main()
{
    return 0;
}
```

In the second drop-down list on the toolbar (the **Solution Platforms** tooltip is displayed for this list), you must set the value to **x86**.

Correct the ParallelApplication1.cpp file as follows:

```
#include <iostream>

int main(int argc, char *argv[])
{
```

```
std::cout << "Hello World!\n";  
return 0;  
}
```

The resulting program should compile successfully. When you run it (by pressing the [F5] key), you will see a console window with the text "Hello World!".

Now let us add the following components to the project, which allow us to use the MPI library (the easiest way is to copy all these components from the PT4Work directory to the project directory, i. e. to the directory that already contains the ParallelApplication1.cpp file):

- mpich.lib (compiled library);
- all header files (h-files) whose name starts with the text "mpi", in particular mpi.h.

It is necessary to add a lib file to the project. The steps required for this have already been described earlier, see Note 2 in Section 1.1.3: you need to start with the menu command **Project | <project name> Properties...**, select the **Configuration Properties | Linker | Input** section in the project properties window that appears, and specify the name of the mpich.lib file to be added in the **Additional Dependencies** textbox, separating it from subsequent names with a semicolon. No special steps are required to include header files in the project.

Let us add directives for including the necessary headers to the main project file and append statements that initialize the parallel mode, print the process rank, and terminate the parallel mode (the added text is highlighted in bold):

```
#include <iostream>  
#include "mpi.h"  
  
int main(int argc, char *argv[])  
{  
    MPI_Init(&argc, &argv);  
    int rank;  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    std::cout << rank << '\n';  
    MPI_Finalize();  
    return 0;  
}
```

Note that here we used the MPI_Init and MPI_Finalize functions for the first time (in all previous programs, these functions were called by the taskbook itself).

If all actions for connecting MPI components were performed correctly, this program will be successfully compiled and launched. When it is launched for the first time, the Windows security system window will appear, in which you need to click the **Allow access (Разрешить доступ)** button (Fig. 48).

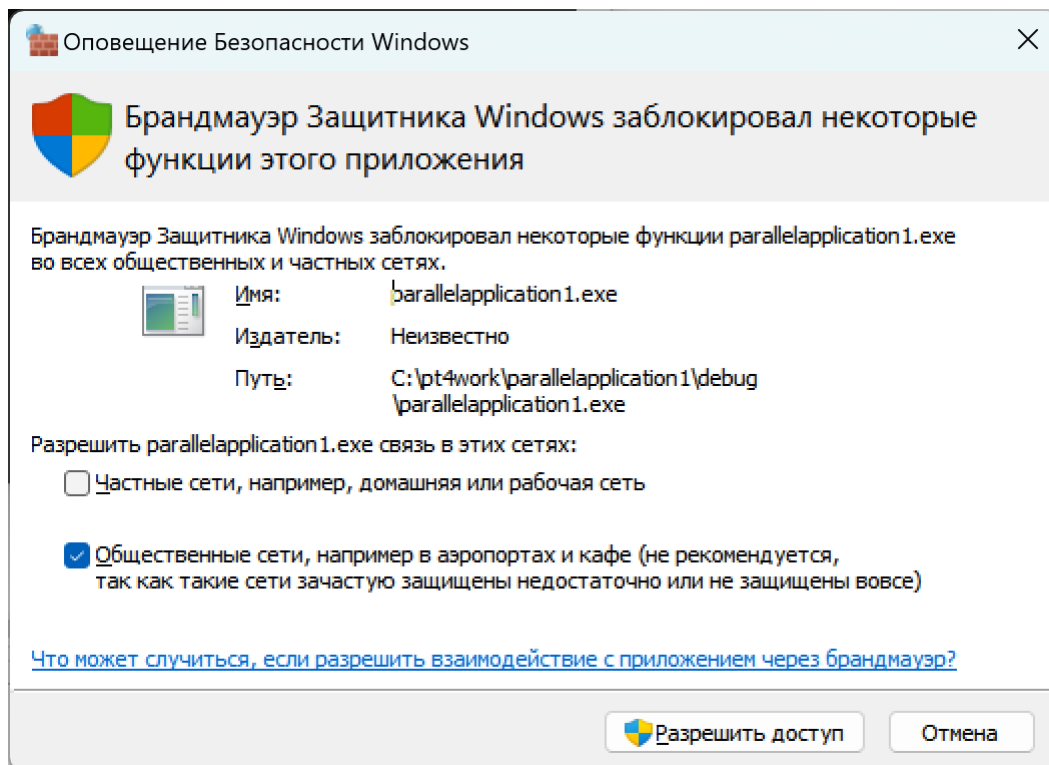


Fig. 48. Window with a request to block a running parallel program

Now we need to prepare a batch file that will launch the resulting program in parallel mode. Let us name this file `mpi_run.bat` and save it in the same directory, where the executable file `ParallelApplication1.exe` is stored (this will allow us not to specify the full path to this file). Recall that by default the executable file of the application is created in the debug subdirectory (or `bin\debug`) of the application directory (see the path to the launched application, indicated in Fig. 48).

The `mpi_run.bat` file must contain two commands, each command must be located *on one line* (to emphasize this fact, here and further in similar situations before the continuation of the previous line we will indicate a special symbol ↵):

```
"C:\Program Files\Microsoft MPI\bin\mpiexec.exe" -n 10
  ↵ "ParallelApplication1.exe"
pause
```

After running this file, a console window will appear on the screen with the result of our program running in parallel mode (Fig. 49).

We see that each process has output its rank in the console window. However, the order of data output from different processes is non-deterministic: when the program is launched again, we will receive a different sequence of numbers in the range from 0 to 9.

So, we have developed a parallel program that can be compiled and run independently of the taskbook. Of course, this program is very simple.

```

C:\Windows\system32\cmd.exe
C:\PT4Work\ParallelApplication1\Debug>"C:\Program Files\Microsoft MPI\bin\mpiexec.exe" -n 10 "ParallelApplication1.exe"
2
0
5
4
6
9
7
8
3
1
C:\PT4Work\ParallelApplication1\Debug>pause
Для продолжения нажмите любую клавишу . . .

```

Fig. 49. Result of execution of the parallel program ParallelApplication1

Note. When developing parallel programs in the **Microsoft Visual Studio** environment, you can avoid using the auxiliary bat-file if you specify the program `mpiexec.exe` and its parameters in the project settings (see Note 1 in Section 1.1.4). In this case, however, in some versions of **Visual Studio**, you need to add a fragment to the program that ensures its suspension at the end of execution (e. g., you can use the function call `system("pause")`).

1.5.3. Additional debug features. Output redirection

In Section 1.5.1, we developed a complex parallel program implementing a self-scheduling algorithm for matrix-vector multiplication. The capabilities of the taskbook helped us in developing and debugging it, but now we want to run this program independently of the taskbook. It turns out that this is not difficult.

First of all, we will include the `MPIDebug10.cpp` file in our console application, which contains the text of the previously developed program. To do this, we will copy it from the `PT4Work` directory to the console application directory (which already contains the `ParallelApplication1.cpp` file) and execute the menu command **Project | Add Existing Item...**, selecting the `MPIDebug10.cpp` file in the window that appears.

Load the `MPIDebug10.cpp` file into the code editor. Now it looks like this (we omit the contents of the `Solve` function for brevity):

```

File MPIDebug10.cpp
#include "pt4.h"
#include "mpi.h"

void Solve()
{
    ...
}

```

The Solve function contains a large number of function calls related to the taskbook and taken from the pt4.h header file. Of course, you can simply delete the directive for including the pt4.h file, and also delete (or comment out) all such calls. However, the taskbook provides a more convenient option: a *stub* for the pt4.h header file. The simplest version of such a stub is designed as a pt4null.h file and contains "empty" implementations of all functions related to the taskbook. Thus, without changing the contents of the Solve function, you can replace the #include "pt4.h" directive with the following directive:

```
#include "pt4null.h"
```

It is also necessary to copy the stub file pt4null.h to the project directory. This file is contained in the PTforMPI2stubs directory of the Programming Taskbook system directory (by default, this is the PT4 directory located in the directory for 32-bit programs: c:\Program Files (x86)).

After completing the above steps, we will find that all the underlines related to the detected errors in the MPIDebug10.cpp file disappear. Of course, in doing so we have lost all the capabilities of our program associated with displaying data on the screen.

We will not (yet) try to restore all the debug information that was output in the Solve function, limiting ourselves to printing the elements of the final product: the *Ab* vector. To do this, add the #include <iostream> directive to the beginning of the MPIDebug10.cpp file, and after the comment "*output of the resulting Ab vector*" in the Solve function, add a fragment that provides the output of the array *c*. As a result, the contents of the MPIDebug10.cpp file will be as follows (all statements added in this file are highlighted in bold):

```
File MPIDebug10.cpp
#include <iostream>
#include "pt4null.h"
#include "pt4.h"
#include "mpi.h"

void Solve ()
{
    ...
    // Output of the obtained vector Ab:
    for (int i = 0; i < n; i++)
        std::cout << c[i] << ' ';
    std::cout << '\n';
    ...
}
```

We only need to correct the contents of the main application file ParallelApplication1.cpp so that instead of printing the process ranks, it calls the Solve function (changed or added fragments in this file are also highlighted in bold):

```

File ParallelApplication1.cpp
#include <iostream>
#include "mpi.h"

void Solve();

int main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    Solve();
    MPI_Finalize ();
    return 0;
}

```

Now all that remains is to compile and build the resulting version of the program by running the command **Build | Build solution** or by pressing the key combination [Ctrl]+[Shift]+[B], after which, having made sure that the compilation and build were successful, run the file `mpi_run.bat` (in which no changes need to be made). The result of the work is shown in Fig. 50. We see that the found elements of the vector Ab coincide with those obtained in the program from Section 1.5.1.

```

C:\Windows\system32\cmd.exe
C:\PT4Work\ParallelApplication1\Debug>"C:\Program Files\Microsoft MPI\bin\mpiexec.exe" -n 10 "ParallelApplication1.exe"
0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2 2.2 2.4 2.6 2.8 3 3.2 3.4 3.6 3.8 4
C:\PT4Work\ParallelApplication1\Debug>pause
Для продолжения нажмите любую клавишу . . .

```

Fig. 50. Result of executing the `ParallelApplication1` program with the `pt4null.h` stub

The Programming Taskbook for MPI-2 provides another version of the stub for the file `pt4.h`, which allows us to apply *all* the debug tools of the taskbook to an "ordinary" console parallel program. This version of the stub has the name `pt4console` and consists of two files: `pt4console.cpp` and `pt4console.h`. Let us demonstrate its use.

First of all, let us copy the `pt4console.cpp` and `pt4console.h` files from the taskbook directory to the console application directory (these files are located in the taskbook directory in the same place as the `pt4null.h` file). Then let us connect the

pt4console.cpp file to the console application by executing the menu command **Project | Add Existing Item...** for it.

After this, in the MPIDebug10.cpp file, replace the `#include "pt4null.h"` directive with the `#include "pt4console.h"` directive and add a call to the `ShowAll` function (without parameters) to the end of the `Solve` function:

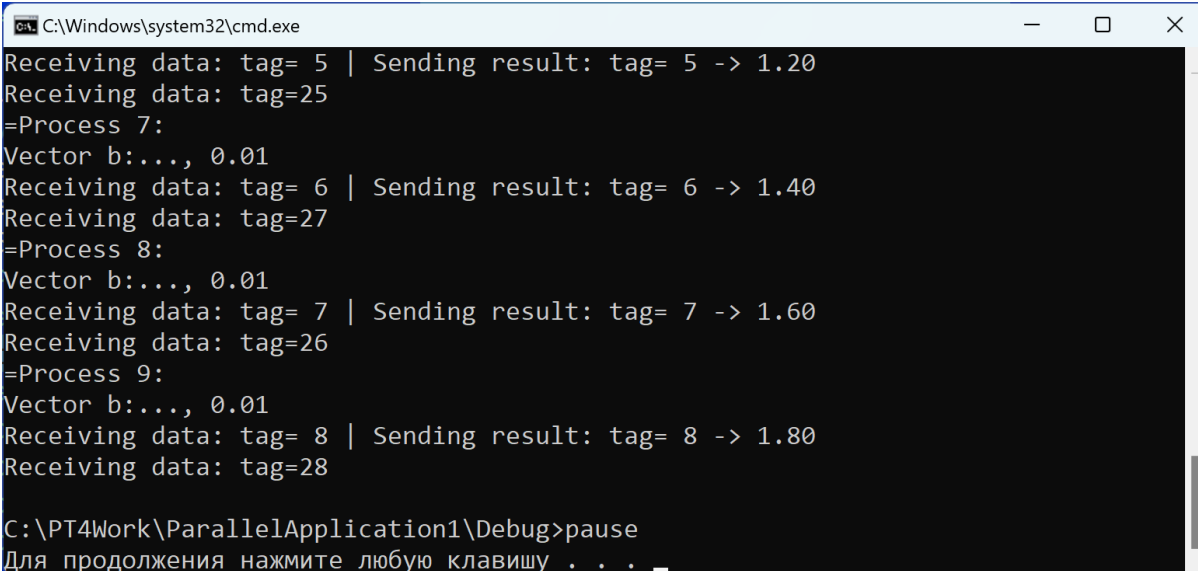
```
File MPIDebug10.cpp
#include <iostream>
#include "pt4console.h"
#include "mpi.h"

void Solve ()
{
    ...
    ShowAll();
}
```

Note that the `ShowAll` function is *collective* function and must be called for all processes in the parallel application.

The `ParallelApplication1.cpp` file does not need to be changed.

After compiling and building a new version of the program using the **Build | Build solution** command and running the file `mpi_run.bat`, we will receive, in the console window, the same information that was previously (see Section 1.5.1) displayed in the debug section of the taskbook window (Fig. 51).



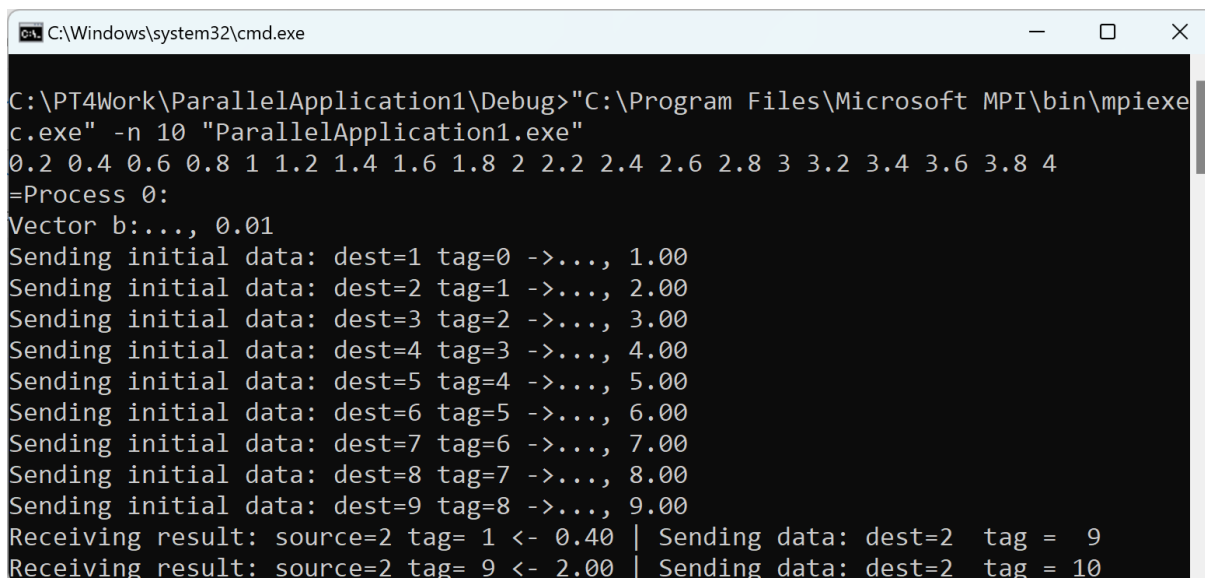
```
C:\Windows\system32\cmd.exe
Receiving data: tag= 5 | Sending result: tag= 5 -> 1.20
Receiving data: tag=25
=Process 7:
Vector b:..., 0.01
Receiving data: tag= 6 | Sending result: tag= 6 -> 1.40
Receiving data: tag=27
=Process 8:
Vector b:..., 0.01
Receiving data: tag= 7 | Sending result: tag= 7 -> 1.60
Receiving data: tag=26
=Process 9:
Vector b:..., 0.01
Receiving data: tag= 8 | Sending result: tag= 8 -> 1.80
Receiving data: tag=28

C:\PT4Work\ParallelApplication1\Debug>pause
Для продолжения нажмите любую клавишу . . .
```

Fig. 51. Result of executing the `ParallelApplication1` program with the `pt4console.h` stub

The only difference is that the output lines are not numbered, and the process rank is indicated in a special line starting with the equal sign "`=`". Note that this output option is *deterministic*: all data is grouped by the processes in which they are output, and the processes are sorted in ascending order of their ranks.

If you scroll the contents of the console window to the beginning, you can see that the master process first output the obtained vector Ab , and then began outputting the data collected from the Show functions (Fig. 52).



```

C:\Windows\system32\cmd.exe
C:\PT4Work\ParallelApplication1\Debug>"C:\Program Files\Microsoft MPI\bin\mpiexec.exe" -n 10 "ParallelApplication1.exe"
0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2 2.2 2.4 2.6 2.8 3 3.2 3.4 3.6 3.8 4
=Process 0:
Vector b:..., 0.01
Sending initial data: dest=1 tag=0 ->..., 1.00
Sending initial data: dest=2 tag=1 ->..., 2.00
Sending initial data: dest=3 tag=2 ->..., 3.00
Sending initial data: dest=4 tag=3 ->..., 4.00
Sending initial data: dest=5 tag=4 ->..., 5.00
Sending initial data: dest=6 tag=5 ->..., 6.00
Sending initial data: dest=7 tag=6 ->..., 7.00
Sending initial data: dest=8 tag=7 ->..., 8.00
Sending initial data: dest=9 tag=8 ->..., 9.00
Receiving result: source=2 tag= 1 <- 0.40 | Sending data: dest=2 tag = 9
Receiving result: source=2 tag= 9 <- 2.00 | Sending data: dest=2 tag = 10

```

Fig. 52. The initial part of the console window for the ParallelApplication1 program with the pt4console.h stub

The reader is advised to analyze the contents of the pt4console.cpp file (and especially the ShowAll function) to find out how such data output was organized. Only the master process performs console output in the ShowAll function, which is why the output is completely deterministic. In this function, the master process first receives all debug data (obtained using the Show and ShowLine functions) from the slave processes, and then it outputs them to the console in the required order. In slave processes, the ShowAll function sends all debugging data to the master process.

In conclusion of the review of examples related to the development of parallel programs without connecting a taskbook to them, we will note one more useful feature.

It is often desirable to organize the output of data in such a way that it is saved after the program has finished. Usually this is done by saving the results in a file. For console applications, such saving can be done very easily using *output stream redirection*. In the case of our program, we only need to slightly supplement the first command from the mpi_run.bat file (the added fragment is highlighted in bold and underlined):

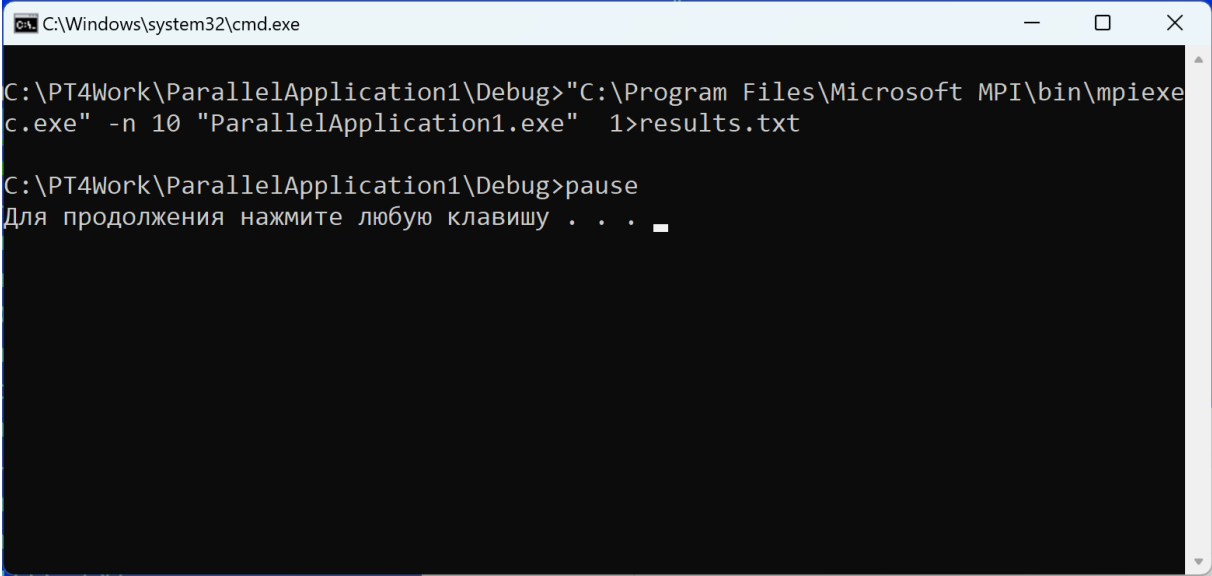
```

"C:\Program Files\Microsoft MPI\bin\mpiexec.exe" -n 10
  ↪ "ParallelApplication1.exe" > results.txt

```

The ">" symbol means that the output stream will be redirected from the console window to the file whose name is specified after this symbol (we chose the name results.txt). If such a file does not exist, it will be created; if it already exists, then its contents will be overwritten.

When you run the modified `mpi_run.bat` file, the console window will not contain any output data (Fig. 53). In such a situation, it is advisable to remove the `pause` command from the bat file, since you no longer need to view the contents of the console window.



```

C:\Windows\system32\cmd.exe
C:\PT4Work\ParallelApplication1\Debug>"C:\Program Files\Microsoft MPI\bin\mpiexec.exe" -n 10 "ParallelApplication1.exe" 1>results.txt

C:\PT4Work\ParallelApplication1\Debug>pause
Для продолжения нажмите любую клавишу . . .

```

Fig. 53. Result of executing the `ParallelApplication1` program with output stream redirection

Now on the disk in the same directory, where the `mpi_run.bat` file is located, the `results.txt` file will appear, containing all the data output by our program. Here is a variant of the contents of this file (compare it with the variants given at the end of Section 1.5.1):

```

0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2 2.2 2.4 2.6 2.8 3 3.2 3.4 3.6 3.8 4
=Process 0:
Vector b:..., 0.01
Sending initial data: dest=1 tag=0 ->..., 1.00
Sending initial data: dest=2 tag=1 ->..., 2.00
Sending initial data: dest=3 tag=2 ->..., 3.00
Sending initial data: dest=4 tag=3 ->..., 4.00
Sending initial data: dest=5 tag=4 ->..., 5.00
Sending initial data: dest=6 tag=5 ->..., 6.00
Sending initial data: dest=7 tag=6 ->..., 7.00
Sending initial data: dest=8 tag=7 ->..., 8.00
Sending initial data: dest=9 tag=8 ->..., 9.00
Receiving result: source=1 tag= 0 <- 0.20 | Sending data: dest=1 tag= 9
Receiving result: source=2 tag= 1 <- 0.40 | Sending data: dest=2 tag=10
Receiving result: source=4 tag= 3 <- 0.80 | Sending data: dest=4 tag=11
Receiving result: source=5 tag= 4 <- 1.00 | Sending data: dest=5 tag=12
Receiving result: source=6 tag= 5 <- 1.20 | Sending data: dest=6 tag=13
Receiving result: source=7 tag= 6 <- 1.40 | Sending data: dest=7 tag=14
Receiving result: source=7 tag=14 <- 3.00 | Sending data: dest=7 tag=15
Receiving result: source=7 tag=15 <- 3.20 | Sending data: dest=7 tag=16
Receiving result: source=7 tag=16 <- 3.40 | Sending data: dest=7 tag=17
Receiving result: source=7 tag=17 <- 3.60 | Sending data: dest=7 tag=18
Receiving result: source=7 tag=18 <- 3.80 | Sending data: dest=7 tag=19
Receiving result: source=7 tag=19 <- 4.00 | Sending data: dest=7 tag=20

```

```

Receiving result: source=8 tag= 7 <- 1.60 | Sending data: dest=8 tag=21
Receiving result: source=9 tag= 8 <- 1.80 | Sending data: dest=9 tag=22
Receiving result: source=1 tag= 9 <- 2.00 | Sending data: dest=1 tag=23
Receiving result: source=2 tag=10 <- 2.20 | Sending data: dest=2 tag=24
Receiving result: source=3 tag= 2 <- 0.60 | Sending data: dest=3 tag=25
Receiving result: source=4 tag=11 <- 2.40 | Sending data: dest=4 tag=26
Receiving result: source=5 tag=12 <- 2.60 | Sending data: dest=5 tag=27
Receiving result: source=6 tag=13 <- 2.80 | Sending data: dest=6 tag=28
Resulting vector Ab:
0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8
4.0
=Process 1:
Vector b:..., 0.01
Receiving data: tag= 0 | Sending result: tag= 0 -> 0.20
Receiving data: tag= 9 | Sending result: tag= 9 -> 2.00
Receiving data: tag=23
=Process 2:
Vector b:..., 0.01
Receiving data: tag= 1 | Sending result: tag= 1 -> 0.40
Receiving data: tag=10 | Sending result: tag=10 -> 2.20
Receiving data: tag=24
=Process 3:
Vector b:..., 0.01
Receiving data: tag= 2 | Sending result: tag= 2 -> 0.60
Receiving data: tag=25
=Process 4:
Vector b:..., 0.01
Receiving data: tag= 3 | Sending result: tag= 3 -> 0.80
Receiving data: tag=11 | Sending result: tag=11 -> 2.40
Receiving data: tag=26
=Process 5:
Vector b:..., 0.01
Receiving data: tag= 4 | Sending result: tag= 4 -> 1.00
Receiving data: tag=12 | Sending result: tag=12 -> 2.60
Receiving data: tag=27
=Process 6:
Vector b:..., 0.01
Receiving data: tag= 5 | Sending result: tag= 5 -> 1.20
Receiving data: tag=13 | Sending result: tag=13 -> 2.80
Receiving data: tag=28
=Process 7:
Vector b:..., 0.01
Receiving data: tag= 6 | Sending result: tag= 6 -> 1.40
Receiving data: tag=14 | Sending result: tag=14 -> 3.00
Receiving data: tag=15 | Sending result: tag=15 -> 3.20
Receiving data: tag=16 | Sending result: tag=16 -> 3.40
Receiving data: tag=17 | Sending result: tag=17 -> 3.60
Receiving data: tag=18 | Sending result: tag=18 -> 3.80
Receiving data: tag=19 | Sending result: tag=19 -> 4.00
Receiving data: tag=20
=Process 8:
Vector b:..., 0.01
Receiving data: tag= 7 | Sending result: tag= 7 -> 1.60
Receiving data: tag=21
=Process 9:
Vector b:..., 0.01
Receiving data: tag= 8 | Sending result: tag= 8 -> 1.80
Receiving data: tag =22

```


2. Learning tasks

If the number of processes is not defined in a task, then this number is assumed to be not greater than 16. A zero-rank process in the MPI_COMM_WORLD communicator is called a *master process* throughout all task groups. All other processes are called *slave processes*.

If a task does not specify the maximal size of an input sequence of number, then this size should be considered as 20.

Tasks of increased difficulty level are marked with * and ** symbols. Tasks whose solutions are given in Section 1 are marked with the symbol °.

2.1. Processes and their ranks

Before solving the tasks in this section, you should study the MPI1Proc2 task solution given in Section 1.1.

MPI1Proc1. Input a real number X in each process of the MPI_COMM_WORLD communicator and output its opposite value $-X$. Also output the total number of processes in the *master process* (that is, a rank-zero process). For data input and output use the input-output stream pt. Also output the value $-X$ in the debug section using the Show function, which is also defined in the taskbook.

MPI1Proc2°. Input an integer A in each process of the MPI_COMM_WORLD communicator and output doubled value of A . Also output the total number of processes in the *master process* (that is, a rank-zero process). For data input and output use the input-output stream pt. In the master process, duplicate the data output in the debug section by displaying on separate lines the doubled value of A and the total number of processes (use two calls of the ShowLine function, which is defined in the taskbook along with the Show function).

Remark. The solution of this task is given in Section 1.1.

MPI1Proc3. Input a real number X and output its opposite value in the master process. Also output the rank of *slave processes* (which are processes whose rank is greater than 0); the rank of each process should be output in the process of this rank. In addition, duplicate the data output in the debug section by displaying the value of $-X$ with the "-X=" comment and the rank values with the "rank=" comments (use the Show function calls with two parameters).

- MPI1Proc4.** Input one integer in processes with even rank (inclusive of the master process) and output the doubled value of input number. Do not perform any action in processes with odd rank.
- MPI1Proc5.** Input one integer in processes with even rank (inclusive of the master process), input one real number in processes with odd rank. Output doubled value of input number in each process.
- MPI1Proc6.** Input one integer in slave processes with even rank, input one real number in processes with odd rank. Output doubled value of input number in each slave process. Do not perform any action in the master process.
- MPI1Proc7.** An integer $N (> 0)$ and a sequence of N real numbers are given in each process with even rank (inclusive of the master process). Output the sum of given numbers in each process. Do not perform any action in processes with odd rank.
- MPI1Proc8.** An integer $N (> 0)$ and a sequence of N real numbers are given in each process. Output the sum of given numbers in each process with even rank (inclusive of the master process), output the average of given numbers in each process with odd rank.
- MPI1Proc9.** An integer $N (> 0)$ and a sequence of N real numbers are given in each process. Output the sum of given numbers in each slave process with even rank, output the average of given numbers in each process with odd rank, output the product of given numbers in the master process.
- MPI1Proc10.** An integer $N (> 0)$ and a sequence of N numbers are given in each process. The sequence contains real numbers in the slave processes with odd rank (1, 3, ...) and integers in the slave processes with even rank (2, 4 ...). The type of elements in the master process depends on the number of processes K : if K is an odd number, then the sequence contains integers, otherwise the sequence contains real numbers. Output the minimal element of the given sequence in each even-rank process (inclusive of the master process), output the maximal element of the given sequence in each odd-rank process.

2.2. Point-to-point communication

Before solving the tasks in this section, you should study the MPI2Send11 task solution given in Section 1.2.2.

2.2.1. Blocking communications

- MPI2Send1.** An integer is given in each process. Send all given integers to the master process using the MPI_Send and MPI_Recv functions (the blocking functions for standard communication mode) and output received integers in the master process in ascending order of ranks of sending processes.

- MPI2Send2.** A real number is given in each slave process. Send all given numbers to the master process using the `MPI_Bsend` (the blocking function for buffered sending mode) and `MPI_Recv` functions and output received numbers in the master process in descending order of ranks of sending processes. Use the `MPI_Buffer_attach` function for attaching a buffer to a process.
- MPI2Send3.** Four integers are given in each slave process. Send all given integers to the master process using one call of the `MPI_Send` function for each sending process. Output received integers in the master process in ascending order of ranks of sending processes.
- MPI2Send4.** An integer N ($0 < N < 5$) and a sequence of N integers are given in each slave process. Send all given sequences to the master process using one call of the `MPI_Bsend` function for each sending process. Output received integers in the master process in ascending order of ranks of sending processes. Use the `MPI_Get_count` to determine the size of received sequences.
- MPI2Send5.** A sequence of real numbers is given in the master process; the size of sequence is equal to the number of slave processes. Send each element of given sequence to corresponding slave process using the `MPI_Send` function: the first number should be sent to the process 1, the second number should be sent to the process 2, and so on. Output received numbers in the slave processes.
- MPI2Send6.** A sequence of real numbers is given in the master process; the size of sequence is equal to the number of slave processes. Send each element of given sequence to corresponding slave process (in inverse order) using the `MPI_Bsend` function: the first number should be sent to the last process, the second number should be sent to the last but one process, and so on. Output received numbers in the slave processes.
- MPI2Send7.** An integer N and a sequence of N real numbers is given in the master process; $K - 1 \leq N < 10$, K is the number of processes. Send elements of given sequence with order number 1, 2, ..., $K - 2$ to slave process of rank 1, 2, ..., $K - 2$ respectively, and send remaining elements of the sequence to the process $K - 1$. Output received numbers in the slave processes. Use the `MPI_Send` function to send data, use the `MPI_Get_count` function to determine the size of received sequences in the process $K - 1$.
- MPI2Send8.** An integer is given in each slave process; only one of given integers is nonzero-valued. Send nonzero integer to the master process. Output the received number and the rank of sending process in the master process. Use the `MPI_Recv` function with the `MPI_ANY_SOURCE` parameter to receive data in the master process.

MPI2Send9. An integer N is given in each slave process; one of given integers is positive, others are zero-valued. Also a sequence of N real numbers is given in the slave process with nonzero integer N . Send the given sequence to the master process. Output the received numbers and the rank of sending process in the master process. Use the `MPI_Recv` function with the `MPI_ANY_SOURCE` parameter to receive data in the master process.

MPI2Send10. An integer N is given in each slave process, an integer $K (> 0)$ is given in the master process; the number K is equal to number of slave processes whose given integers N are positive. Send all positive integers N to the master process. Output sum of received numbers in the master process. Use the `MPI_Recv` function with the `MPI_ANY_SOURCE` parameter to receive data in the master process.

MPI2Send11°. A real number is given in each process. Send the given number from the master process to all slave processes and send the given numbers from the slave processes to the master process. Output the received numbers in each process. The numbers received by the master process should be output in ascending order of ranks of sending processes. Use the `MPI_Ssend` function to send data.

Note. The `MPI_Ssend` function provides a *synchronous data transfer mode*, in which the operation of sending a message will be completed only after the receiving process starts to receive this message. In the case of data transfer in synchronous mode, there is a danger of *deadlocks* because of the incorrect order of the function calls for sending and receiving data.

Remark. The solution of this task is given in Section 1.2.2.

MPI2Send12. An integer is given in each process. Using the `MPI_Ssend` and `MPI_Recv` functions, perform the right cyclic shift of given data by step 1 (that is, the given integer should be sent from the process 0 to the process 1, from the process 1 to the process 2, ..., from the last process to the process 0). Output the received number in each process.

Note. See note to `MPI2Send11`.

MPI2Send13. An integer is given in each process. Using the `MPI_Ssend` and `MPI_Recv` functions, perform the left cyclic shift of given data by step -1 (that is, the given integer should be sent from the process 1 to the process 0, from the process 2 to the process 1, ..., from the process 0 to the last process). Output the received number in each process.

Note. See note to `MPI2Send11`.

MPI2Send14. Two integers are given in each process. Send the first integer to the previous process and the second integer to the next process (use the `MPI_Ssend` and `MPI_Recv` functions). The last process is assumed to be the previous one for the master process, the master process is assumed to be the previous one for the last process. Output the received numbers in each

process in the following order: the number received from the previous process, then the number received from the next process.

Note. See note to MPI2Send11.

MPI2Send15. A real number A and an integer N are given in each process; the set of given integers N contains all values in the range 0 to $K - 1$, K is the number of processes. Send the number A to the process N in each process (use the `MPI_Send` and `MPI_Recv` functions and the `MPI_ANY_SOURCE` parameter). Output the received number and the rank of sending process in each process.

MPI2Send16. An integer N is given in each process; the value of N is equal to 1 for one process and is equal to 0 for others. Also a sequence of $K - 1$ real numbers is given in the process with nonzero integer N ; K is the number of processes. Send each number from the given sequence to one of other processes in ascending order of ranks of receiving processes. Output the received number in each process.

MPI2Send17. A sequence of $K - 1$ integers is given in each process; K is the number of processes. Send one of the integers from the given sequence in each process to the corresponding process in ascending order of ranks of receiving processes. Output the received numbers in each process in ascending order of ranks of sending processes.

MPI2Send18. The number of processes is an even number. An integer N ($0 < N < 5$) and a sequence of N real numbers are given in each process. Exchange given sequences of processes 0 and 1, 2 and 3, and so on, using the `MPI_Sendrecv` function. Output the received sequence of real numbers in each process.

MPI2Send19. A real number is given in each process. Change the order of given numbers to inverse one by sending the given numbers from the process 0 to the last process, from the process 1 to the last but one process, ..., from the last process to the process 0. Use the `MPI_Sendrecv_replace` function. Output the received number in each process.

MPI2Send20. A real number A and its order number N (as an integer) are given in each slave process; the set of integers N contains all values in the range 0 to $K - 1$, K is the number of processes. Send all numbers A to the master process and output the received numbers in ascending order of their order numbers N . Do not use arrays. Use the order number N as a `msgtag` parameter of the `MPI_Send` function.

MPI2Send21. An integer L (≥ 0) and a sequence of L pairs (A, N) are given in each slave process; A is a real number and N is the order number of A . The sum of all integers L is equal to $2K$, where K is the number of processes; the set of integers N contains all values in the range 1 to $2K$. Send all numbers A to the master process and output the received numbers in ascending

order of their order numbers N . Do not use arrays. Use the order number N as a msgtag parameter of the MPI_Send function.

MPI2Send22*. A sequence of pairs (T, A) is given in the master process; the size of sequence is equal to the number of slave processes. An integer T is equal to 0 or 1; if $T = 0$, then A is an integer, otherwise A is a real number. Send one of the numbers A to the corresponding slave process (the first number to the process 1, the second number to the process 2, and so on) and output received numbers in the slave processes. Use the value of T as a msgtag parameter of the MPI_Send function to send information about the type of number A ; use the MPI_Probe function with the parameter MPI_ANY_TAG to receive this information.

Note. To avoid the code duplication, use the auxiliary *template functions* `template<typename T> void send(int t, int dest, MPI_Datatype d)` to send data and `template<typename T> void recv(MPI_Datatype d)` to receive data. Use a number equal to 0 or 1 for the t parameter and the rank of receiving process for the $dest$ parameter.

MPI2Send23*. Two integers T, N and a sequence of N numbers are given in each slave process. An integer T is equal to 0 or 1; if $T = 0$, then the given sequence contains integers, otherwise it contains real numbers. Send all given sequences to the master process and output received numbers in the ascending order of ranks of sending processes. Use the value of T as a msgtag parameter of the MPI_Send function to send information about the sequence type; use the MPI_Probe function with the parameter MPI_ANY_TAG to receive this information.

Note. To avoid the code duplication, use the auxiliary *template functions* `template<typename T> void send(int t, MPI_Datatype d)` to send data and `template<typename T> void recv(MPI_Datatype d, MPI_Status s)` to receive data. Use a number equal to 0 or 1 for the t parameter and the result returned by the MPI_Probe function for the s parameter.

MPI2Send24*. The number of processes K is an even number: $K = 2N$. A sequence of N real numbers is given in each even-rank process $(0, 2, \dots, K - 2)$, a sequence of N integers is given in each odd-rank process $(1, 3, \dots, K - 1)$. Using the MPI_Sendrecv_replace function, perform the cyclic shift of all real-valued sequences in the direction of increasing the ranks of processes and the cyclic shift of all integer sequences in the direction of decreasing the ranks of processes (that is, the real-valued sequences should be sent from the process 0 to the process 2, from the process 2 to the process 4, ..., from the process $K - 2$ to the process 0 and the integer sequences should be sent from the process $K - 1$ to the process $K - 3$, from the process $K - 3$ to the process $K - 5$, ..., from the process 1 to the process $K - 1$). Output received data in each process. To determine the rank of the

receiving process, use the expression containing the % operator of taking the remainder after integer division. Use the MPI_ANY_SOURCE parameter as the rank of sending process.

Note. To avoid the code duplication, use the auxiliary *template function* `template<typename T> void sendrecv(int rank, int size, MPI_Datatype d, int step)`. The step parameter specifies a shift value, which should be equal to 2 for real-valued sequences and equal to -2 for integer ones.

MPI2Send25*. The number of processes K is an even number: $K = 2N$. A sequence of $R + 1$ real numbers is given in the first half of the processes, where R is the process rank ($R = 0, 1, \dots, N - 1$). A sequence of $2N - R$ integers is given in the second half of the processes, where R is the process rank ($R = N, N + 1, \dots, 2N - 1$). Using the MPI_Sendrecv function, send the given sequences from each half of the processes to the corresponding process of the other half (that is, the sequence from the process 0 should be sent to the process N , from the process 1 — to the process $N + 1$, from the process N — to the process 0, from the process $2N - 1$ — to the process $N - 1$, and so on). Output received data in each process.

Note. To avoid the code duplication, use the auxiliary *template function* `template<typename T1, typename T2> void sendrecv(MPI_Datatype d1, int cnt1, int rank2, MPI_Datatype d2, int cnt2)`, where the $d1$ and $cnt1$ parameters define the properties of the process that calls the function (the type and the number of elements of sending sequence) and the parameters $rank2$, $d2$, $cnt2$ are the rank and the similar properties of the process involved in data exchange.

2.2.2. Non-blocking communications

MPI2Send26*. An integer N is given in each process. The value of N is equal to 0 in all processes, except for one, and it is equal to 1 in some selected process. Also an integer sequence A of size $K - 1$ is given in the selected process, where K is the number of processes. Do not save the sequence A in array. Send one element of the sequence A at a time to other processes in ascending order of their ranks and output the received number in each process. Use the required number of the MPI_Isend and MPI_Wait function calls (sending a message in synchronous non-blocking mode) in the selected process and the MPI_Recv function call in the other processes. Additionally, display the duration of each MPI_Wait function call (in milliseconds) in the debug section. To do this, call the MPI_Wtime function before and after the MPI_Wait call and use the Show function to display the difference between returned values of the MPI_Wtime function multiplied by 1000. Check how the debugging information changes if the MPI_Isend function (sending a message in standard non-blocking mode) will be used instead of the MPI_Isend function.

- MPI2Send27***. An integer N is given in each process. The value of N is equal to -1 in some selected process of the rank R and it is equal to R in the other processes. A real number A is also given in all processes, except for the selected one. Send the numbers A to the selected process and output received numbers in ascending order of ranks of sending processes. Use the required number of the `MPI_Recv` function calls in the selected process and the `MPI_Issend` and `MPI_Test` function call in the other processes. Repeat the `MPI_Test` function call until it returns a nonzero flag, and display the required number of iterations of the loop in the debug section using the `Show` function. Check how the debugging information changes if the `MPI_Isend` function will be used instead of the `MPI_Issend` function.
- MPI2Send28***. An integer N is given in each process. The value of N is equal to -1 in some selected process of the rank R and it is equal to R in the other processes. A real number A is also given in all processes, except for the selected one. Send the numbers A to the selected process and output received numbers in descending order of ranks of sending processes. Use the required number of the `MPI_lrecv` and `MPI_Test` function calls (receiving a message in non-blocking mode) in the selected process and the `MPI_Ssend` function call in the other processes. Repeat the `MPI_Test` function call after each `MPI_lrecv` function call until `MPI_Test` returns a nonzero flag, and display the required number of iterations of the loop in the debug section using the `Show` function. Check how the debugging information changes if the `MPI_Send` function will be used instead of the `MPI_Ssend` function.
- MPI2Send29***. An integer N is given in each process. The value of N is equal to -1 in some selected process of the rank R and it is equal to R in the other processes. A real number A is also given in all processes, except for the selected one. Send the numbers A to the selected process and output the sum S of received numbers. Use the required number of the `MPI_lrecv` and `MPI_Waitany` function calls in the selected process and the `MPI_Ssend` function call in the other processes. Declare an array Q of the `MPI_Request` type in the selected process and call the `MPI_lrecv` functions in a loop with the a separate element of Q for each function call. Then call the `MPI_Waitany` function in a second loop to accumulate the sum S . Additionally, display the following data in the debug section in each iteration of the second loop (using the `Show` and `ShowLine` function call): the value of A added to the sum at this iteration, and the rank of the process that sent this value.
- MPI2Send30***. An integer N is given in each process. The value of N is equal to 0 in all processes, except for two, and it is equal to 1 in the first selected process (*the sender*) and it is equal to 2 in the second selected process (*the receiver*). Also an integer R and a sequence of K integers are given in the sender, where R is the rank of the receiver and K is the number of

processes. Do not save the sequence A in array. Send all elements of the sequence A to the receiver and output the received numbers in the same order. Use the single call of the `MPI_Ssend_init` function and the required number of the `MPI_Start` and `MPI_Wait` function calls in the sender, and the single call of the `MPI_Recv_init` function and the required number of the `MPI_Start` and `MPI_Wait` function calls in the receiver. Additionally, display the duration of each `MPI_Wait` function call (in milliseconds) in the debug section (for both the sender and the receiver). To do this, call the `MPI_Wtime` function before and after the `MPI_Wait` call and use the `Show` function to display the difference between returned values of the `MPI_Wtime` function multiplied by 1000. Check how the debugging information changes if the `MPI_Send_init` function will be used instead of the `MPI_Ssend_init` function.

MPI2Send31*. An integer N is given in each process. The value of N is equal to 2 in the first selected process (*the receiver*), it is equal to 1 in some other selected processes (*the senders*), it is equal to 0 in all other processes. Also an integer R and a sequence A of K integers are given in each sender, where R is the rank of the receiver and K is the number of processes, and the number of senders C is given in the receiver. Send all sequences A to the receiver and output the sums S of elements of all sequences A with the same indices (in ascending order of indices). Use the single call of the `MPI_Ssend` function in each sender. Declare an array Q of the `MPI_Request` type in the receiver and call the `MPI_Recv_init` functions in a loop with a separate element of Q for each function call. Then call the `MPI_Startall` function in the receiver and, after that, call the `MPI_Waitany` function in a second loop to accumulate the sums S . Additionally, display the following data in the debug section in each iteration of the second loop (using two `Show` function and one `ShowLine` function calls): the duration of each `MPI_Waitany` function call (in milliseconds), the returned value of the third parameter (named `index`) of the `MPI_Waitany` function, and the rank of current sender that corresponds to the `index` parameter. To find the duration, call the `MPI_Wtime` function before and after the `MPI_Waitany` call and calculate the difference between returned values of the `MPI_Wtime` function multiplied by 1000. To find the rank of the current sender, use the value of the last parameter (of the `MPI_Status` type) returned by the `MPI_Waitany` function.

MPI2Send32*. An integer N is given in each process. The value of N is equal to 1 in the first selected process (*the sender*), it is equal to 2 in some other selected processes (*the receivers*), it is equal to 0 in all other processes. Also a real number A , an integer C , and a sequence R of C integers are given in the sender, where C is the number of receivers and R contains ranks of all receivers. Send the number A to all receivers and output it in each receiver. Use the single call of the `MPI_Recv` function in each receiver. Declare an array Q of the `MPI_Request` type in the sender and call the

MPI_Ssend_init functions in a loop with a separate element of Q for each function call. Then call the MPI_Startall function in the sender and, after that, call the MPI_Testany function in a second loop (the MPI_Testany function should be called in a nested loop until it returns a nonzero flag). Additionally, display the following data in the debug section in each iteration of the second loop (using the Show and ShowLine function call): the returned value of the third parameter (named index) of the MPI_Testany function (when it returns a nonzero flag), and the number of MPI_Testany function calls (that is, the number of iterations of the nested loop). Check how the debugging information changes if the MPI_Send_init function will be used instead of the MPI_Ssend_init function.

2.3. Collective communications

Before solving the tasks in this section, you should study the MPI3Coll23 task solution given in Section 1.2.5.

2.3.1. Collective data transfer

MPI3Coll1. An integer is given in the master process. Send the given integer to all slave processes using the MPI_Bcast function. Output the received integer in all slave processes.

MPI3Coll2. A sequence of 5 real numbers is given in the master process. Send the given sequence to all slave processes using the MPI_Bcast function. Output received data in all slave processes.

MPI3Coll3. A real number is given in each process. Send the given numbers to master process using the MPI_Gather function. Output received numbers in the master process in ascending order of ranks of sending processes (starting with the number that is given in the master process).

MPI3Coll4. A sequence of 5 integers is given in each process. Send the given sequences to master process using the MPI_Gather function. Output received data in the master process in ascending order of ranks of sending processes (starting with the sequence that is given in the master process).

MPI3Coll5. A sequence of $R + 2$ integers is given in each process; the integer R is equal to rank of the process (there are given 2 integers in the process 0, 3 integers in the process 1, and so on). Send the given sequences to master process using the MPI_Gatherv function. Output received data in the master process in ascending order of ranks of sending processes (starting with the sequence that is given in the master process).

MPI3Coll6. A sequence of K real numbers is given in the master process; K is the number of processes. Send one element of given sequence to each process (inclusive of the master process) using the MPI_Scatter function. Output the received number in each process.

- MPI3Coll7.** A sequence of $3K$ real numbers is given in the master process, K is the number of processes. Send three elements of given sequence to each process (inclusive of the master process) using the `MPI_Scatter` function. Output received numbers in each process.
- MPI3Coll8.** A sequence of K real numbers is given in the master process; K is the number of processes. Using the `MPI_Scatterv` function, send elements of given sequence to all processes as follows: the first element should be sent to the process $K - 1$, the second element should be sent to the process $K - 2$, ..., the last element should be sent to the process 0). Output the received number in each process.
- MPI3Coll9.** A sequence of $K(K + 3)/2$ integers is given in the master process; K is the number of processes. Using the `MPI_Scatterv` function, send $R + 2$ elements of given sequence to the process of rank R , where $R = 0, \dots, K - 1$: the first two elements should be sent to the process 0, the next three elements should be sent to the process 1, and so on. Output received numbers in each process.
- MPI3Coll10.** A sequence of $K + 2$ real numbers is given in the master process; K is the number of processes. Using the `MPI_Scatterv` function, send three elements of given sequence to each process as follows: elements with order numbers in the range $R + 1$ to $R + 3$ should be sent to the process of rank R , where $R = 0, \dots, K - 1$ (the initial three elements should be sent to the process 0; the second, the third, and the fourth element should be sent to the process 1, and so on). Output received numbers in each process.
- MPI3Coll11.** A real number is given in each process. Send given numbers to all process using the `MPI_Allgather` function. Output received data in each process in ascending order of ranks of sending processes (inclusive of the number received from itself).
- MPI3Coll12.** Four integers are given in each process. Send given integers to all processes using the `MPI_Allgather` function. Output received data in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).
- MPI3Coll13.** A sequence of $R + 2$ integers is given in each process; R is the rank of process (that is, two integers are given in the process 0, three integers are given in the process 1, and so on). Send given integers to all processes using the `MPI_Allgatherv` function. Output received data in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).
- MPI3Coll14.** A sequence of K real numbers is given in each process; K is the number of processes. Using the `MPI_Alltoall` function, send one element of each given sequence to each process as follows: first element of each sequence should be sent to the process 0, second element of each sequence

should be sent to the process 1, and so on. Output received numbers in each process in ascending order of ranks of sending processes (inclusive of the number received from itself).

MPI3Coll15. A sequence of $3K$ integers is given in each process; K is the number of processes. Using the `MPI_Alltoall` function, send three elements of each given sequence to each process as follows: the initial three elements of each sequence should be sent to the process 0, the next three elements of each sequence should be sent to the process 1, and so on. Output received numbers in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).

MPI3Coll16*. A sequence of $K(K + 1)/2$ integers is given in each process; K is the number of processes. Using the `MPI_Alltoallv` function, send some elements of each given sequence to each process as follows: the first element of each sequence should be sent to the process 0, the next two elements of each sequence should be sent to the process 1, the next three elements of each sequence should be sent to the process 2, and so on. Output received numbers in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).

MPI3Coll17*. A sequence of $K + 1$ real numbers is given in each process; K is the number of processes. Using the `MPI_Alltoallv` function, send two elements of each given sequence to each process as follows: the initial two elements of each sequence should be sent to the process 0, the second and the third element of each sequence should be sent to the process 1, ..., the last two elements of each sequence should be sent to the last process. Output received numbers in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).

MPI3Coll18*. A sequence of $K + 1$ real numbers is given in each process; K is the number of processes. Using the `MPI_Alltoallv` function, send two elements of each given sequence to each process as follows: the last two elements of each sequence (with the order numbers $K + 1$ and K) should be sent to the process 0, the elements of each sequence with the order numbers $K - 1$ and K should be sent to the process 1, ..., the initial two elements of each sequence should be sent to the last process. Output received numbers in each process in ascending order of ranks of sending processes (inclusive of the numbers received from itself).

2.3.2. Global reduction operations

MPI3Coll19. A sequence of $K + 5$ integers is given in each process; K is the number of processes. Find sums of elements of all given sequences with the same order number using the `MPI_Reduce` function with the `MPI_SUM` operation. Output received sums in the master process.

MPI3Coll20. A sequence of $K + 5$ real numbers is given in each process; K is the number of processes. Find the minimal value among the elements of all given sequences with the same order number using the `MPI_Reduce` function with the `MPI_MIN` operation. Output received minimal values in the master process.

MPI3Coll21. A sequence of $K + 5$ integers is given in each process; K is the number of processes. Using the `MPI_Reduce` function with the `MPI_MAXLOC` operation, find the maximal value among the elements of all given sequences with the same order number and also the rank of process that contains this maximal value. Output received maximal values and ranks in the master process (first, output all maximal values, then output all corresponding ranks).

MPI3Coll22. A sequence of $K + 5$ real numbers is given in each process; K is the number of processes. Find products of elements of all given sequences with the same order number using the `MPI_Allreduce` function with the `MPI_PROD` operation. Output received products in each process.

MPI3Coll23°. A sequence of $K + 5$ real numbers is given in each process; K is the number of processes. Using the `MPI_Allreduce` function with the `MPI_MINLOC` operation, find the minimal value among the elements of all given sequences with the same order number and also the rank of process that contains this minimal value. Output received minimal values in the master process and output corresponding ranks in each slave process.

Remark. The solution of this task is given in Section 1.2.5.

MPI3Coll24. A sequence of K integers is given in each process; K is the number of processes. Using the `MPI_Reduce_scatter` function, find sums of elements of all given sequences with the same order number and send one sum to each process as follows: the first sum should be sent to the process 0, the second sum should be sent to the process 1, and so on. Output the received sum in each process.

MPI3Coll25. A sequence of $2K$ real numbers is given in each process; K is the number of processes. Using the `MPI_Reduce_scatter` function, find maximal values among elements of all given sequences with the same order number and send two maximal values to each process as follows: the initial two maximums should be sent to the process 0, the next two maximums should be sent to the process 1, and so on. Output received data in each process.

MPI3Coll26. A sequence of $K(K + 3)/2$ integers is given in each process; K is the number of processes. Using the `MPI_Reduce_scatter` function, find minimal values among elements of all given sequences with the same order number and send some minimal values to each process as follows: the initial two minimums should be sent to the process 0, the next three mini-

mums should be sent to the process 1, ..., the last $K + 1$ minimums should be sent to the process $K - 1$. Output received data in each process.

MPI3Coll27. A sequence of $K + 5$ real numbers is given in each process; K is the number of processes. Using the `MPI_Scan` function, find products of elements of given sequences with the same order number as follows: the products of elements of sequences given in the processes of rank 0, ..., R should be found in the process R ($R = 0, 1, \dots, K - 1$). Output received data in each process; in particular, products of elements of all given sequences should be output in the process $K - 1$.

MPI3Coll28. A sequence of $K + 5$ integers is given in each process; K is the number of processes. Using the `MPI_Scan` function, find maximal values among elements of given sequences with the same order number as follows: the maximal values of elements of sequences given in the processes of rank 0, ..., R should be found in the process R ($R = 0, 1, \dots, K - 1$). Output received data in each process.

2.4. *Derived datatypes and data packing*

Before solving the tasks in this section, you should study the `MPI4Type14` task solution given in Section 1.2.6.

2.4.1. The simplest derived datatypes

MPI4Type1. A sequence of $K - 1$ triples of integers is given in the master process; K is the amount of processes. Send all given data to each slave process using derived datatype with three integer elements and one collective operation with the derived datatype. Output received data in each slave process in the same order.

MPI4Type2. A sequence of $K - 1$ triples of integers is given in the master process; K is the amount of processes. Send one given triple at a time to each slave process using derived datatype with three integer elements and one collective operation with the derived datatype. Output received integers in each slave process in the same order.

MPI4Type3. A triple of integers is given in each slave process. Send all given triples to the master process using derived datatype with three integer elements and one collective operation with the derived datatype. Output received data in the master process in ascending order of ranks of sending processes.

MPI4Type4. A sequence of $K - 1$ triples of numbers is given in the master process; K is the amount of processes. Two initial items of each triple are integers, the last item is a real number. Send all given triples to each slave process using derived datatype with three elements (two integers and a real

number) and one collective operation with the derived datatype. Output received data in each slave process in the same order.

MPI4Type5. A sequence of $K - 1$ triples of numbers is given in the master process; K is the amount of processes. The first item and the last item of each triple are integers, the middle item is a real number. Send one given triple at a time to each slave process using derived datatype with three elements (an integer, a real number, an integer) and one collective operation with the derived datatype. Output received data in each slave process in the same order.

Note. When solving this task, it may be necessary to solve an additional problem related to the special way of field alignment in data structures for C/C++ languages. In some situations, the alignment results in additional *empty spaces* between the fields of the structure (and at its end), which must be taken into account when defining the corresponding MPI derived datatypes. To recognize such situations, it is enough to find the size of the created structure (using the `sizeof` operator) and compare it with the total size of its fields. If these sizes do not coincide, then it means that there are empty spaces between the fields and/or at the end of the structure. Here is an example of such a structure corresponding to the current task:

```
struct s1
{
    int a;
    double b;
    int c;
};
```

The total size of its fields is $4 + 8 + 4 = 16$ bytes, but the `sizeof(s1)` call will return the value 24. This is explained by the fact that data of double type in structures are aligned on the boundary of 8 bytes (the so-called *double-word alignment*), and this alignment must be preserved in *arrays of structures*, so an empty space can be added *to the end* of the structure as well. In our case, in order to ensure the double-word alignment of field `b`, an empty space equal to 4 bytes is added after field `a`, and in order to preserve such alignment for consecutive structures of type `s1`, an empty space equal to 4 bytes is also added to its end. Thus, the distribution of fields in memory for two instances of structure `s1` will be as follows (here the symbols `x`, `y`, `z` denote bytes occupied by fields `a`, `b`, `c`, and the symbol `.` (dot) denotes a byte included in the empty space:

a	b	c	a	b	c	
0	8	16	24	32	40	48
xxxx....yyyyyyzzzz....xxxx....yyyyyyzzzz....						

It is easy to see that only the presence of empty spaces provides double-word alignment for field `b`.

This must be taken into account when defining the appropriate MPI datatypes. In our case, it is necessary not only to specify the correct offset between the first and the second data block, but also to define the final empty space. Here is an example of correct definition of datatype `t1` for structure `s1`, in which functions from MPI-2 standard are used:

```
MPI_Datatype t10, t1;
int blocklen[3] = {1, 1, 1};
MPI_Aint displ[3] = {0, 8, 16};
MPI_Datatype types[3] = {MPI_INT, MPI_DOUBLE, MPI_INT};
MPI_Type_create_struct(3, blocklen, displ, types, &t10);
MPI_Type_create_resized(t10, 0, 24, &t1);
```

Standard MPI datatypes also take into account the double-word alignment. In particular, the *extent* of the `MPI_DOUBLE_INT` type is 16, while its *size* is 12.

MPI4Type6. A triple of numbers is given in each slave process. The first item of each triple is a real number, the other items are integers. Send all given triples to the master process using derived datatype with three elements (a real number and two integers) and one collective operation with the derived datatype. Output received data in the master process in ascending order of ranks of sending processes.

MPI4Type7. A triple of numbers is given in each process. The first item and the last item of each triple are integers, the middle item is a real number. Send the given triples from each process to all processes using derived datatype with three elements (an integer, a real number, an integer) and one collective operation with the derived datatype. Output received data in each process in ascending order of ranks of sending processes (inclusive of data received from itself).

MPI4Type8. A sequence of R triples of numbers is given in each slave process; R is the rank of process. Two initial items of each triple are integers, the last item is a real number. Send all given triples to the master process using derived datatype with three elements (two integers and a real number) and one collective operation with the derived datatype. Output received data in the master process in ascending order of ranks of sending processes.

2.4.2. Data packing

MPI4Type9. Two sequences of K numbers are given in the master process; K is the amount of processes. The first given sequence contains integers, the second given sequence contains real numbers. Send all data to each slave process using the `MPI_Pack` and `MPI_Unpack` functions and one collective operation. Output received data in each slave process in the same order.

- MPI4Type10.** A sequence of $K - 1$ triples of numbers is given in the master process; K is the amount of processes. The first item and the last item of each triple are integers, the middle item is a real number. Send one given triple at a time to each slave process using the pack/unpack functions and one collective operation. Output received numbers in each slave process in the same order.
- MPI4Type11.** A sequence of $K - 1$ triples of numbers is given in the master process; K is the amount of processes. Two initial items of each triple are integers, the last item is a real number. Send all given triples to each slave process using the pack/unpack functions and one collective operation. Output received data in each slave process in the same order.
- MPI4Type12.** A triple of numbers is given in each slave process. Two initial items of each triple are integers, the last item is a real number. Send the given triples from each slave process to the master process using the pack/unpack functions and one collective operation. Output received data in the master process in ascending order of ranks of sending processes.
- MPI4Type13.** A real number and a sequence of R integers are given in each slave process; R is the rank of process (one integer is given in the process 1, two integers are given in the process 2, and so on). Send all given data from each slave process to the master process using the pack/unpack functions and one collective operation. Output received data in the master process in ascending order of ranks of sending processes.

2.4.3. Additional ways of derived datatypes creation

- MPI4Type14*°.** Two sequences of integers are given in the master process: the sequence A of the size $3K$ and the sequence B of the size K , where K is the number of slave processes. The elements of sequences are numbered from 1. Send N_R elements of the sequence A to each slave process R ($R = 1, 2, \dots, K$) starting with the A_R and increasing the ordinal number by 2 ($R, R + 2, R + 4, \dots$). For example, if N_2 is equal to 3, then the process 2 should receive the elements A_2, A_4, A_6 . Output all received data in each slave process. Use one call of the MPI_Send, MPI_Probe, and MPI_Recv functions for sending numbers to each slave process; the MPI_Recv function should return an array that contains only elements that should be output. To do this, define a new datatype that contains a single integer and an additional empty space (*a hole*) of a size that is equal to the size of integer datatype. Use the following data as parameters for the MPI_Send function: the given array A with the appropriate displacement, the amount N_R of sending elements, a new datatype. Use an integer array of the size N_R and the MPI_INT datatype in the MPI_Recv function. To determine the number N_R of received elements, use the MPI_Get_count function in the slave processes.

Note. Use the `MPI_Type_create_resized` function to define the hole size for a new datatype (this function should be applied to the `MPI_INT` datatype). In the MPI-1, the zero-size *upper-bound marker* `MPI_UB` should be used jointly with the `MPI_Type_struct` for this purpose (in MPI-2, the `MPI_UB` pseudo-datatype is deprecated).

Remark. The solution of this task is given in Section 1.2.6.

MPI4Type15*. An real-valued square matrix of order K is given in the master process; K is the number of slave processes. Elements of the matrix should be stored in a one-dimensional array A in a row-major order. The columns of matrix are numbered from 1. Send R -th column of matrix to the process of rank R ($R = 1, 2, \dots, K$) and output all received elements in each slave process. Use one call of the `MPI_Send` and `MPI_Recv` functions for sending elements to each slave process; the `MPI_Recv` function should return an array that contains only elements that should be output. To do this, define a new datatype that contains a single real number and an additional empty space (*a hole*) of the appropriate size. Use the following data as parameters for the `MPI_Send` function: the given array A with the appropriate displacement, the amount K of sending elements (i. e., the size of column), a new datatype. Use a real-valued array of the size K and the `MPI_DOUBLE` datatype in the `MPI_Recv` function.

Note. See the note to `MPI4Type14`.

MPI4Type16*. R -th column of a real-valued square matrix of order K is given in the slave process of rank R ($R = 1, 2, \dots, K$); K is the number of slave processes, the columns of matrix are numbered from 1. Send all columns to the master process and store them in a one-dimensional array A in a row-major order. Output all elements of A in the master process. Use one call of the `MPI_Send` and `MPI_Recv` functions for sending elements of each column; the resulting array A with the appropriate displacement should be the first parameter for the `MPI_Recv` function, and a number 1 should be its second parameter. To do this, define a new datatype (in the master process) that contains K real numbers and an empty space (*a hole*) of the appropriate size after each number. Define a new datatype in two steps. In the first step, define auxiliary datatype that contains one real number and additional hole (see the note to `MPI4Type14`). In the second step, define the final datatype using the `MPI_Type_contiguous` function (this datatype should be the third parameter for the `MPI_Recv` function). The `MPI_Type_commit` function is sufficient to call only for the final datatype. Use a real-valued array of size K and the `MPI_DOUBLE` datatype in the `MPI_Send` function.

MPI4Type17*. The number of slave processes K is a multiple of 3 and does not exceed 9. An integer N is given in each process, all the numbers N are the same and are in the range from 3 to 5. Also an integer square matrix of or-

der N (*a block*) is given in each slave process; the block should be stored in a one-dimensional array B in a row-major order. Send all arrays B to the master process and compose a block matrix of the size $(K/3) \times 3$ (the size is indicated in blocks) using a row-major order for blocks (i. e., the first row of blocks should include blocks being received from the processes 1, 2, 3, the second row of blocks should include blocks from the processes 4, 5, 6, and so on). Store the block matrix in the one-dimensional array A in a row-major order. Output all elements of A in the master process. Use one call of the `MPI_Send` and `MPI_Recv` functions for sending each block B ; the resulting array A with the appropriate displacement should be the first parameter for the `MPI_Recv` function, and a number 1 should be its second parameter. To do this, define a new datatype (in the master process) that contains N sequences, each sequence contains N integers, and an empty space (*a hole*) of the appropriate size should be placed between the sequences. Define the required datatype using the `MPI_Type_vector` function (this datatype should be the third parameter for the `MPI_Recv` function). Use the array B of size $N \cdot N$ and the `MPI_INT` datatype in the `MPI_Send` function.

MPI4Type18*. The number of slave processes K is a multiple of 3 and does not exceed 9. An integer N in the range from 3 to 5 and an integer *block matrix* of the size $(K/3) \times 3$ (the size is indicated in blocks) are given in the master process. Each block is *a lower triangular matrix* of order N , the block contains all matrix elements, inclusive of zero-valued ones. The block matrix should be stored in the one-dimensional array A in a row-major order. Send a non-zero part of each block to the corresponding slave process in a row-major order of blocks (i. e., the blocks of the first row should be sent to the processes 1, 2, 3, the blocks of the second row should be sent to the processes 4, 5, 6, and so on). Output all received elements in each slave process (in a row-major order). Use one call of the `MPI_Send`, `MPI_Probe`, and `MPI_Recv` functions for sending each block; the resulting array A with the appropriate displacement should be the first parameter for the `MPI_Send` function, and a number 1 should be its second parameter. To do this, define a new datatype (in the master process) that contains N sequences, each sequence contains non-zero part of the next row of a lower triangular block (the first sequence consists of 1 element, the second sequence consists of 2 elements, and so on), and an empty space (*a hole*) of the appropriate size should be placed between the sequences. Define the required datatype using the `MPI_Type_indexed` function (this datatype should be the third parameter for the `MPI_Send` function). Use an integer array B , which contains a non-zero part of received block, and the `MPI_INT` datatype in the `MPI_Recv` function. To determine the number of received elements, use the `MPI_Get_count` function in the slave processes.

MPI4Type19*. The number of slave processes K is a multiple of 3 and does not exceed 9. An integer N is given in each process, all the numbers N are the same and are in the range from 3 to 5. Also an integer P and a non-zero part of an integer square matrix of order N (*a Z-block*) are given in each slave process. The given elements of Z-block should be stored in a one-dimensional array B in a row-major order. These elements are located in the Z-block in the form of the symbol "Z", i. e. they occupy the first and last row, and also the antidiagonal. Define a zero-valued integer matrix of the size $N \cdot (K/3) \times 3N$ in the master process (all elements of this matrix are equal to 0 and should be stored in a one-dimensional array A in a row-major order). Send a non-zero part of the given Z-block from each slave process to the master process in ascending order of ranks of sending processes and write each received Z-block in the array A starting from the element of array A with index P (the positions of Z-blocks can overlap, in this case the elements of blocks received from processes of higher rank will replace some of the elements of previously written blocks). Output all elements of A in the master process. Use one call of the MPI_Send and MPI_Recv functions for sending each Z-block; the array A with the appropriate displacement should be the first parameter for the MPI_Recv function, and a number 1 should be its second parameter. To do this, define a new datatype (in the master process) that contains N sequences, the first and the last sequences contain N integers, the other sequences contain 1 integer, and an empty space (*a hole*) of the appropriate size should be placed between the sequences. Define the required datatype using the MPI_Type_indexed function (this datatype should be the third parameter for the MPI_Recv function). Use the array B , which contains a non-zero part of a Z-block, and the MPI_INT datatype in the MPI_Send function.

Note. Use the msgtag parameter to send the Z-block insertion position P to the master process. To do this, set the value of P as the msgtag parameter for the MPI_Send function in slave processes, call the MPI_Probe function with the MPI_ANY_TAG parameter in the master process (before calling the MPI_Recv function), and analyze its returned parameter of the MPI_Status type.

MPI4Type20*. The number of slave processes K is a multiple of 3 and does not exceed 9. An integer N is given in each process, all the numbers N are the same and are in the range from 3 to 5. Also an integer P and a non-zero part of an integer square matrix of order N (*an U-block*) are given in each slave process. The given elements of U-block should be stored in a one-dimensional array B in a row-major order. These elements are located in the U-block in the form of the symbol "U", i. e. they occupy the first and last column, and also the last row. Define a zero-valued integer matrix of the size $N \cdot (K/3) \times 3N$ in the master process (all elements of this matrix are

equal to 0 and should be stored in a one-dimensional array A in a row-major order). Send a non-zero part of the given U-block from each slave process to the master process in ascending order of ranks of sending processes and write each received U-block in the array A starting from the element of array A with index P (the positions of U-blocks can overlap, in this case the elements of blocks received from processes of higher rank will replace some of the elements of previously written blocks). Output all elements of A in the master process. Use one call of the `MPI_Send` and `MPI_Recv` functions for sending each U-block; the array A with the appropriate displacement should be the first parameter for the `MPI_Recv` function, and a number 1 should be its second parameter. To do this, define a new datatype (in the master process) that contains appropriate number of sequences with empty spaces (*holes*) between them. Define the required datatype using the `MPI_Type_indexed` function (this datatype should be the third parameter for the `MPI_Recv` function). Use the array B , which contains a non-zero part of an U-block, and the `MPI_INT` datatype in the `MPI_Send` function.

Note. See the note to `MPI4Type19`.

2.4.4. The `MPI_Alltoallw` function (MPI-2)

`MPI4Type21**`. Solve the `MPI4Type15` task by using one collective operation instead of the `MPI_Send` and `MPI_Recv` functions to transfer data.

Note. You cannot use the functions of the Scatter group, since the displacements for the data items (columns of the matrix) should be specified in *bytes* rather than in elements. Therefore, you should use the function `MPI_Alltoallw` introduced in MPI-2, which allows you to configure the collective communications in the most flexible way. In this case, the `MPI_Alltoallw` function should be used to implement a data transfer of the Scatter type (and most of the array parameters used in this function need to be defined differently in the master and slave processes).

`MPI4Type22**`. Solve the `MPI4Type16` task by using one collective operation instead of the `MPI_Send` and `MPI_Recv` functions to transfer data.

Note. See the note to `MPI4Type21`. In this case, the `MPI_Alltoallw` function should be used to implement a data transfer of the Gather type.

2.5. *Process groups and communicators*

Before solving the tasks in this section, you should study the `MPI5Comm3`, `MPI5Comm17`, `MPI5Comm29` task solutions given in Sections 1.2.7–1.2.9.

2.5.1. Creation of new communicators

`MPI5Comm1`. A sequence of K integers is given in the master process; K is the number of processes whose rank is an even number (0, 2, ...). Create a new communicator that contains all even-rank processes using the

MPI_Comm_group, MPI_Group_incl, and MPI_Comm_create functions. Send one given integer to each even-rank process (including the master process) using one collective operation with the created communicator. Output received integer in each even-rank process.

MPI5Comm2. Two real numbers are given in each process whose rank is an odd number (1, 3, ...). Create a new communicator that contains all odd-rank processes using the MPI_Comm_group, MPI_Group_excl, and MPI_Comm_create functions. Send all given numbers to each odd-rank process using one collective operation with the created communicator. Output received numbers in each odd-rank process in ascending order of ranks of sending processes (including numbers received from itself).

MPI5Comm3°. Three integers are given in each process whose rank is a multiple of 3 (including the master process). Using the MPI_Comm_split function, create a new communicator that contains all processes with ranks that are a multiple of 3. Send all given numbers to master process using one collective operation with the created communicator. Output received integers in the master process in ascending order of ranks of sending processes (including integers received from the master process).

Note. When calling the MPI_Comm_split function in processes that are not required to include in the new communicator, one should specify the constant MPI_UNDEFINED as the color parameter.

Remark. The solution of this task is given in Section 1.2.7.

MPI5Comm4. A sequence of 3 real numbers is given in each process whose rank is an even number (including the master process). Find the minimal value among the elements of the given sequences with the same order number using a new communicator and one global reduction operation. Output received minimums in the master process.

Note. See the note to MPI5Comm3.

MPI5Comm5. A real number is given in each process. Using the MPI_Comm_split function and one global reduction operation, find the maximal value among the numbers given in the even-rank processes (including the master process) and the minimal value among the numbers given in the odd-rank processes. Output the maximal value in the process 0 and the minimal value in the process 1.

Note. The program should contain a single MPI_Comm_split call, which creates the both required communicators (each for the corresponding group of processes).

MPI5Comm6. An integer K and a sequence of K real numbers are given in the master process, an integer N is given in each slave process. The value of N is equal to 1 for some processes and is equal to 0 for others; the number of processes with $N = 1$ is equal to K . Send one real number from the master

process to each slave process with $N = 1$ using the `MPI_Comm_split` function and one collective operation. Output the received numbers in these slave processes.

Note. See the note to `MPI5Comm3`.

MPI5Comm7. An integer N is given in each process; the value of N is equal to 1 for at least one process and is equal to 0 for others. Also a real number A is given in each process with $N = 1$. Send all numbers A to the first process with $N = 1$ using the `MPI_Comm_split` function and one collective operation. Output received numbers in this process in ascending order of ranks of sending processes (including the number received from this process).

Note. See the note to `MPI5Comm3`.

MPI5Comm8. An integer N is given in each process; the value of N is equal to 1 for at least one process and is equal to 0 for others. Also a real number A is given in each process with $N = 1$. Send all numbers A to the last process with $N = 1$ using the `MPI_Comm_split` function and one collective operation. Output received numbers in this process in ascending order of ranks of sending processes (including the number received from this process).

Note. See the note to `MPI5Comm3`.

MPI5Comm9. An integer N is given in each process; the value of N is equal to 1 for at least one process and is equal to 0 for others. Also a real number A is given in each process with $N = 1$. Send all numbers A to each process with $N = 1$ using the `MPI_Comm_split` function and one collective operation. Output received numbers in these processes in ascending order of ranks of sending processes (including the number received from itself).

Note. See the note to `MPI5Comm3`.

MPI5Comm10. An integer N is given in each process; the value of N is equal to 1 for some processes and is equal to 2 for others, there are at least one process with $N = 1$ and one process with $N = 2$. Also an integer A is given in each process. Using the `MPI_Comm_split` function and one collective operation, send integers A from all processes with $N = 1$ to each process with $N = 1$ and from all processes with $N = 2$ to each process with $N = 2$. Output received integers in each process in ascending order of ranks of sending processes (including the integer received from itself).

Note. See the note to `MPI5Comm5`.

MPI5Comm11. An integer N is given in each process; the value of N is equal to 1 for at least one process and is equal to 0 for others. Also a real number A is given in each process with $N = 1$. Find the sum of all real numbers A using the `MPI_Comm_split` function and one global reduction operation. Output the received sum in each process with $N = 1$.

Note. See the note to MPI5Comm3.

MPI5Comm12. An integer N is given in each process; the value of N is equal to 1 for some processes and is equal to 2 for others, there are at least one process with $N = 1$ and one process with $N = 2$. Also a real number A is given in each process. Using the `MPI_Comm_split` function and one global reduction operation, find the minimal value among the numbers A given in the processes with $N = 1$ and the maximal value among the numbers A given in the processes with $N = 2$. Output the minimal value in each process with $N = 1$ and the maximal value in each process with $N = 2$.

Note. See the note to MPI5Comm5.

2.5.2. Virtual topologies

MPI5Comm13. An integer $N (> 1)$ is given in the master process; the number of processes K is assumed to be a multiple of N . Send the integer N to all processes and define a Cartesian topology for all processes as a $(N \times K/N)$ grid using the `MPI_Cart_create` function (ranks of processes should not be reordered). Find the process coordinates in the created topology using the `MPI_Cart_coords` function and output the process coordinates in each process.

MPI5Comm14. An integer $N (> 1)$ is given in the master process; the number N is not greater than the number of processes K . Send the integer N to all processes and define a Cartesian topology for the initial part of processes as a $(N \times K/N)$ grid using the `MPI_Cart_create` function (the symbol `"/"` denotes the operator of integer division, ranks of processes should not be reordered). Output the process coordinates in each process included in the Cartesian topology.

MPI5Comm15. The number of processes K is an even number: $K = 2N$, $N > 1$. A real number A is given in the processes 0 and N . Define a Cartesian topology for all processes as a $(2 \times N)$ grid. Using the `MPI_Cart_sub` function, split this grid into two one-dimensional subgrids (namely, *rows*) such that the processes 0 and N were the master processes in these rows. Send the given number A from the master process of each row to each process of the same row using one collective operation. Output the received number in each process (including the processes 0 and N).

MPI5Comm16. The number of processes K is an even number: $K = 2N$, $N > 1$. A real number A is given in the processes 0 and 1. Define a Cartesian topology for all processes as a $(N \times 2)$ grid. Using the `MPI_Cart_sub` function, split this grid into two one-dimensional subgrids (namely, *columns*) such that the processes 0 and 1 were the master processes in these columns. Send the given number A from the master process of each column to each process of the same column using one collective operation. Output the received number in each process (including the processes 0 and 1).

MPI5Comm17°. The number of processes K is a multiple of 3: $K = 3N$, $N > 1$. A sequence of N integers is given in the processes 0, N , and $2N$. Define a Cartesian topology for all processes as a $(3 \times N)$ grid. Using the `MPI_Cart_sub` function, split this grid into three one-dimensional subgrids (namely, *rows*) such that the processes 0, N , and $2N$ were the master processes in these rows. Send one given integer from the master process of each row to each process of the same row using one collective operation. Output the received integer in each process (including the processes 0, N , and $2N$).

Remark. The solution of this task is given in Section 1.2.8.

MPI5Comm18. The number of processes K is a multiple of 3: $K = 3N$, $N > 1$. A sequence of N integers is given in the processes 0, 1, and 2. Define a Cartesian topology for all processes as a $(N \times 3)$ grid. Using the `MPI_Cart_sub` function, split this grid into three one-dimensional subgrids (namely, *columns*) such that the processes 0, 1, and 2 were the master processes in these columns. Send one given integer from the master process of each column to each process of the same column using one collective operation. Output the received integer in each process (including the processes 0, 1, and 2).

MPI5Comm19. The number of processes K is equal to 8 or 12. An integer is given in each process. Define a Cartesian topology for all processes as a three-dimensional $(2 \times 2 \times K/4)$ grid (ranks of processes should not be reordered), which should be considered as 2 two-dimensional $(2 \times K/4)$ subgrids (namely, *matrices*) that contain processes with the identical first coordinate in the Cartesian topology. Split each matrix into two one-dimensional rows of processes. Send given integers from all processes of each row to the master process of the same row using one collective operation. Output received integers in the master process of each row (including integer received from itself).

MPI5Comm20. The number of processes K is equal to 8 or 12. An integer is given in each process. Define a Cartesian topology for all processes as a three-dimensional $(2 \times 2 \times K/4)$ grid (ranks of processes should not be reordered), which should be considered as $K/4$ two-dimensional (2×2) subgrids (namely, *matrices*) that contain processes with the identical third coordinate in the Cartesian topology. Split this grid into $K/4$ matrices of processes. Send given integers from all processes of each matrix to the master process of the same matrix using one collective operation. Output received integers in the master process of each matrix (including integer received from itself).

MPI5Comm21. The number of processes K is equal to 8 or 12. A real number is given in each process. Define a Cartesian topology for all processes as a three-dimensional $(2 \times 2 \times K/4)$ grid (ranks of processes should not be

reordered), which should be considered as 2 two-dimensional ($2 \times K/4$) subgrids (namely, *matrices*) that contain processes with the identical first coordinate in the Cartesian topology. Split each matrix into $K/4$ one-dimensional columns of processes. Using one global reduction operation, find the product of all numbers given in the processes of each column. Output the product in the master process of the corresponding column.

MPI5Comm22. The number of processes K is equal to 8 or 12. A real number is given in each process. Define a Cartesian topology for all processes as a three-dimensional ($2 \times 2 \times K/4$) grid (ranks of processes should not be reordered), which should be considered as $K/4$ two-dimensional (2×2) subgrids (namely, *matrices*) that contain processes with the identical third coordinate in the Cartesian topology. Split this grid into $K/4$ matrices of processes. Using one global reduction operation, find the sum of all numbers given in the processes of each matrix. Output the sum in the master process of the corresponding matrix.

MPI5Comm23. Positive integers M and N are given in the master process; the product of the numbers M and N is less than or equal to the number of processes. Also integers X and Y are given in each process whose rank is in the range 0 to $M \cdot N - 1$. Send the numbers M and N to all processes and define a Cartesian topology for initial $M \cdot N$ processes as a two-dimensional ($M \times N$) grid, which is periodic in the first dimension (ranks of processes should not be reordered). Using the `MPI_Cart_rank` function, output the rank of process with the coordinates X , Y (taking into account periodicity) in each process included in the Cartesian topology. Output -1 in the case of erroneous coordinates.

Note. If invalid coordinates are specified when calling the `MPI_Cart_rank` function (for instance, in the case of negative coordinates for non-periodic dimensions), then the function itself returns an error code (instead of the successful return code `MPI_SUCCESS`) whereas the return value of the rank parameter is undefined. So, in this task, the number -1 should be output when the `MPI_Cart_rank` function returns a value that differs from `MPI_SUCCESS`. To suppress the output of error messages in the debug section of the Programming Taskbook window, it is enough to set the special error handler named `MPI_ERROR_RETURN` before calling a function that may be erroneous (use the `MPI_Comm_set_errhandler` function or, in MPI-1, the `MPI_Errhandler_set` function). When an error occurs in some function, this error handler performs no action except setting an error return value for this function. In MPICH version 1.2.5, the `MPI_Cart_rank` function returns the rank parameter equal to -1 when the process coordinates are invalid. This feature may simplify the solution; however, in this case, one also should suppress the output of error messages by means of special error handler setting.

MPI5Comm24. Positive integers M and N are given in the master process; the product of the numbers M and N is less than or equal to the number of processes. Also integers X and Y are given in each process whose rank is in the range 0 to $M \cdot N - 1$. Send the numbers M and N to all processes and define a Cartesian topology for initial $M \cdot N$ processes as a two-dimensional ($M \times N$) grid, which is periodic in the second dimension (ranks of processes should not be reordered). Using the `MPI_Cart_rank` function, output the rank of process with the coordinates X, Y (taking into account periodicity) in each process included in the Cartesian topology. Output -1 in the case of erroneous coordinates.

Note. See the note to `MPI5Comm23`.

MPI5Comm25. A real number is given in each slave process. Define a Cartesian topology for all processes as a one-dimensional grid. Using the `MPI_Send` and `MPI_Recv` functions, perform a shift of given data by step -1 (that is, the real number given in each process should be sent to the process of the previous rank). Ranks of source and destination process should be determined by means of the `MPI_Cart_shift` function. Output received data in each destination process.

MPI5Comm26. The number of processes K is an even number: $K = 2N, N > 1$. A real number A is given in each process. Define a Cartesian topology for all processes as a two-dimensional ($2 \times N$) grid (namely, *matrix*); ranks of processes should not be reordered. Using the `MPI_Sendrecv` function, perform a cyclic shift of data given in all processes of each row of the matrix by step 1 (that is, the number A should be sent from each process in the row, except the last process, to the next process in the same row and from the last process in the row to the first process in the same row). Ranks of source and destination process should be determined by means of the `MPI_Cart_shift` function. Output received data in each process.

MPI5Comm27. The number of processes K is equal to 8 or 12. A real number A is given in each process. Define a Cartesian topology for all processes as a three-dimensional ($2 \times 2 \times K/4$) grid (ranks of processes should not be reordered), which should be considered as $K/4$ two-dimensional (2×2) subgrids (namely, *matrices*) that contain processes with the identical third coordinate in the Cartesian topology and are ordered by the value of this third coordinate. Using the `MPI_Sendrecv_replace` function, perform a cyclic shift of data given in all processes of each matrix by step 1 (that is, the numbers A should be sent from all processes of each matrix, except the last matrix, to the corresponding processes of the next matrix and from all processes of the last matrix to the corresponding processes of the first matrix). Ranks of source and destination process should be deter-

mined by means of the `MPI_Cart_shift` function. Output received data in each process.

MPI5Comm28.** The number of processes K is an odd number: $K = 2N + 1$ ($1 < N < 5$). An integer A is given in each process. Using the `MPI_Graph_create` function, define a graph topology for all processes as follows: the master process must be connected by edges to all odd-rank processes (that is, to the processes 1, 3, ..., $2N - 1$); each process of the rank R , where R is a positive even number (2, 4, ..., $2N$), must be connected by edge to the process of the rank $R - 1$. Thus, the graph represents N -beam star whose center is the master process, each star beam consists of two slave processes of ranks R and $R + 1$, and each odd-rank process R is adjacent to star center (namely, the master process), see Fig. 54.

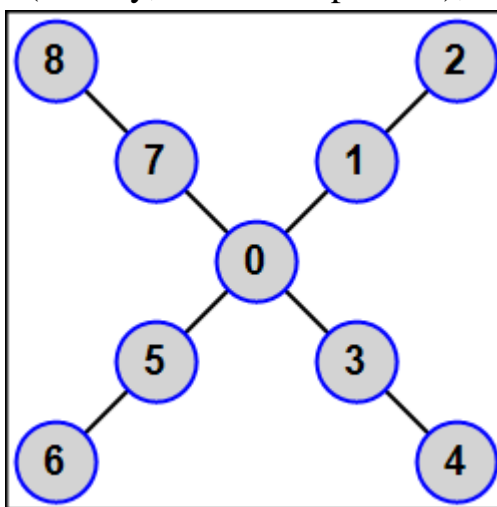


Fig. 54. Example of graph topology from the MPI5Comm28 task

Using the `MPI_Send` and `MPI_Recv` functions, send the given integer A from each process to all its adjacent processes (its *neighbors*). The amount and ranks of neighbors should be determined by means of the `MPI_Graph_neighbors_count` and `MPI_Graph_neighbors` functions respectively. Output received data in each process in ascending order of ranks of sending processes.

MPI5Comm29°.** The number of processes K is an even number: $K = 2N$ ($1 < N < 6$). An integer A is given in each process. Using the `MPI_Graph_create` function, define a graph topology for all processes as follows: all even-rank processes (including the master process) are linked in a chain $0 - 2 - 4 - 6 - \dots - (2N - 2)$; each process with odd rank R (1, 3, ..., $2N - 1$) is connected by edge to the process with the rank $R - 1$. Thus, each odd-rank process has a single neighbor, the first and the last even-rank processes have two neighbors, and other even-rank processes (the "inner" ones) have three neighbors (Fig. 55).

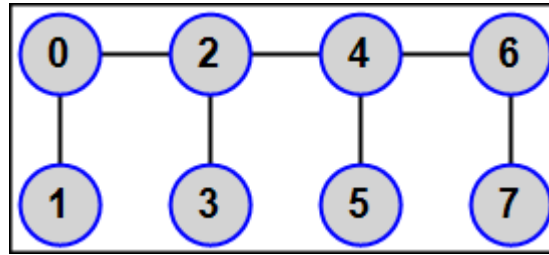


Fig. 55. Example of graph topology from the MPI5Comm29 task

Using the `MPI_Sendrecv` function, send the given integer A from each process to all its neighbors. The amount and ranks of neighbors should be determined by means of the `MPI_Graph_neighbors_count` and `MPI_Graph_neighbors` functions respectively. Output received data in each process in ascending order of ranks of sending processes.

Remark. The solution of this task is given in Section 1.2.9.

MPI5Comm30.** The number of processes K is equal to $3N + 1$ ($1 < N < 5$). An integer A is given in each process. Using the `MPI_Graph_create` function, define a graph topology for all processes as follows: processes of ranks R , $R + 1$, $R + 2$, where $R = 1, 4, 7, \dots$, are linked by edges and, moreover, each process whose rank is positive and is a multiple of 3 (that is, the process 3, 6, ...) is connected by edge to the master process. Thus, the graph represents N -beam star whose center is the master process, each star beam consists of three linked slave processes, and each slave process with rank that is a multiple of 3 is adjacent to star center (namely, the master process), see Fig. 56.

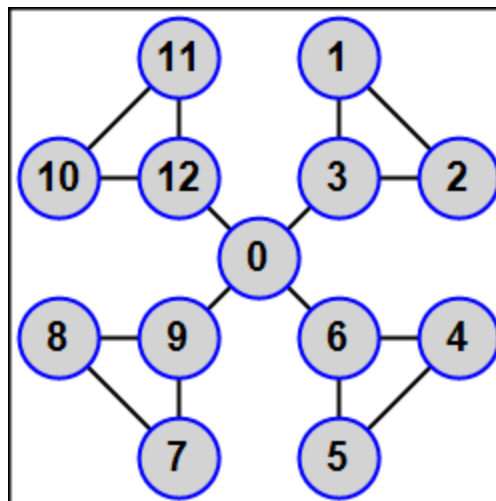


Fig. 56. Example of graph topology from the MPI5Comm30 task

Using the `MPI_Sendrecv` function, send the given integer A from each process to all its neighbors. The amount and ranks of neighbors should be determined by means of the `MPI_Graph_neighbors_count` and `MPI_Graph_neighbors` functions respectively. Output received data in each process in ascending order of ranks of sending processes.

2.5.3. The distributed graph topology (MPI-2)

MPI5Comm31.** The number of processes K is a multiple of 3; an integer A is given in each process. Using the `MPI_Dist_graph_create` function, define a distributed graph topology for all processes as follows: all processes whose rank is a multiple of 3 ($0, 3, \dots, K - 3$) are linked in a ring, each process in this ring is the source for the next process of the ring (that is, the process 0 is the source for the process 3, the process 3 is the source for the process 6, ..., the process $K - 3$ is the source for the process 0); besides, the process $3N$ ($N = 0, 1, \dots, K/3 - 1$) is the source for the processes $3N + 1$ and $3N + 2$, and the process $3N + 1$ is the source for the process $3N + 2$ (Fig. 57).

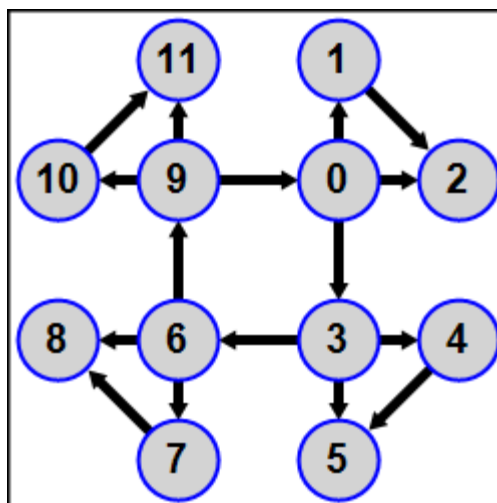


Fig. 57. Example of graph topology from the MPI5Comm31 task

The complete definition of the graph topology should be given in the master process (whereas the second parameter of the `MPI_Dist_graph_create` function should be equal to 0 in all slave processes). The weights parameter should be equal to `MPI_UNWEIGHTED`, the info parameter should be equal to `MPI_INFO_NULL`, ranks of processes should not be reordered. Using the `MPI_Send` and `MPI_Recv` functions, send the given numbers from the source processes to the destination processes and output the sum of the given number A and all received numbers in each process. The amount and ranks of source and destination processes should be determined by means of the `MPI_Dist_graph_neighbors_count` and `MPI_Dist_graph_neighbors` functions.

MPI5Comm32.** The number of processes is in the range 4 to 15; an integer A is given in each process. Using the `MPI_Dist_graph_create` function, define a distributed graph topology for all processes as follows: all processes form a binary tree with the process 0 as a tree root, the processes 1 and 2 as a tree nodes of level 1, the processes from 3 to 6 as a tree nodes of level 2 (the processes 3 and 4 are the child nodes of the process 1 and the processes 5 and 6 are the child nodes of the process 2), and so on. Each process is the

source for all its child nodes, so each process has 0 to 2 destination processes (Fig. 58).

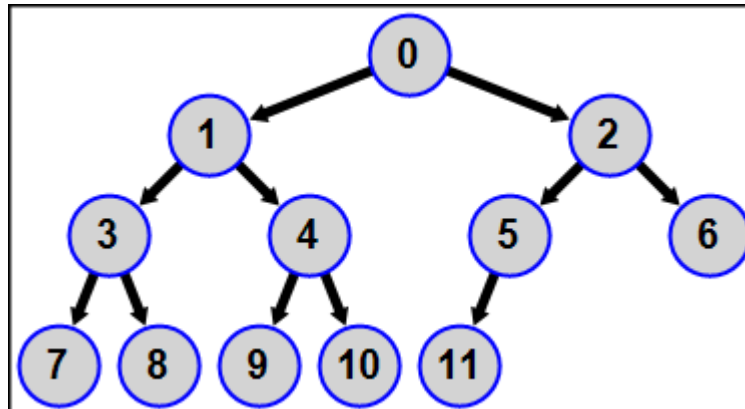


Fig. 58. Example of graph topology from the MPI5Comm32 task

The complete definition of the graph topology should be given in the master process (whereas the second parameter of the `MPI_Dist_graph_create` function should be equal to 0 in all slave processes). The weights parameter should be equal to `MPI_UNWEIGHTED`, the info parameter should be equal to `MPI_INFO_NULL`, ranks of processes should not be reordered. Using the `MPI_Send` and `MPI_Recv` functions in each process, find and output the sum of the given number A and all the numbers that are given in ancestors of all levels — from the tree root (the master process) to the nearest ancestor (the parent process). The amount and ranks of source and destination processes should be determined by means of the `MPI_Dist_graph_neighbors_count` and `MPI_Dist_graph_neighbors` functions.

2.5.4. Non-blocking collective functions (MPI-3)

MPI5Comm33*. A sequence of 5 real numbers is given in the process of rank $K/2$, where K is the number of processes. Use the `MPI_lbcst` function for an appropriate communicator to send these numbers to processes with lower ranks and output the numbers in processes with ranks from 0 to $K/2$. Use the `MPI_Wait` function to check if the `MPI_lbcst` non-blocking operation finishes. Additionally, display the duration of each call of `MPI_Wait` function in milliseconds for processes with ranks from 0 to $K/2$ in the debug section; for this purpose, call `MPI_Wtime` before and after `MPI_Wait` and use the `Show` function to display the difference of values returned by `MPI_Wtime` multiplied by 1000.

MPI5Comm34*. A sequence of 5 integers is given in each process of odd rank. Using the `MPI_lgather` function for an appropriate communicator, send these numbers to a process of rank 1 and output them in descending order of ranks of the processes that sent them (first output the sequence of numbers given in the process with the highest odd rank, last — the sequence of

numbers given in the process of rank 1). Use the `MPI_Test` function to check if the `MPI_lgather` non-blocking operation finishes; call the `MPI_Test` function in a loop until it returns a non-zero flag. Additionally, display the number of required loop iterations with `MPI_Test` calls in the debug section for all odd-rank processes using the `Show` function.

MPI5Comm35*. A sequence of $R + 2$ integers is given in each process of even rank, where the number R equals the rank of the process (process 0 has 2 numbers, process 2 has 4 numbers, and so on). Use the `MPI_lgatherv` function for an appropriate communicator to send these numbers to master process. Output these numbers in ascending order of the ranks of the processes that sent them (first output the sequence of numbers given in the master process). Use `MPI_Wait` function to check if `MPI_lgatherv` non-blocking operation finishes. Additionally, display the duration of each call to the `MPI_Wait` function in milliseconds for even-rank processes in the debug section; for this purpose, call `MPI_Wtime` before and after `MPI_Wait` and use the `Show` function to display the difference of the values returned by `MPI_Wtime` multiplied by 1000.

MPI5Comm36*. The number of processes K is an even number. A sequence of $K/2$ real numbers is given in the master process. Use the `MPI_lscatter` function for an appropriate communicator to send a number to each process of even rank (including the master process) and output the received number in each process; the numbers must be sent to the processes in reverse order: the first number must be sent to the last process of even rank, the last number — to the process of rank 0. Use the `MPI_Test` function to check if the `MPI_lscatter` non-blocking operation finishes; call the `MPI_Test` function in a loop until it returns a non-zero flag. Additionally, display the number of required loop iterations with `MPI_Test` calls in the debug section for all even-rank processes using the `Show` function.

MPI5Comm37*. The number of processes K is an even number. A sequence of $K/2 + 2$ real numbers is given in the process of rank 1. Using the `MPI_lscatterv` function for an appropriate communicator, send three numbers of given sequence to each process of odd rank, with the first three numbers sent to the process of rank 1, the second, the third and the fourth numbers sent to the process of rank 3, and so on. Output the received numbers in each process. Use the `MPI_Wait` function to check if the `MPI_lscatterv` non-blocking operation finishes. Additionally, display the duration of each call to the `MPI_Wait` function in milliseconds for odd-rank processes in the debug section; for this purpose, call `MPI_Wtime` before and after `MPI_Wait` and use the `Show` function to display the difference of values returned by `MPI_Wtime` multiplied by 1000.

- MPI5Comm38***. An integer N is given in each process; the number N can take two values: 0 and 1 (there is at least one process with $N = 1$). Also a real number A is given in each process with $N = 1$. Use the `MPI_lallgather` function to send the number A to each process for which $N = 1$ and output the numbers in each process in ascending order of the ranks of the processes that sent the numbers (including the number received from the same process). Use the `MPI_Test` function to check if the `MPI_lallgather` non-blocking operation finishes; call the `MPI_Test` function in a loop until it returns a non-zero flag. Additionally, display the number of required loop iterations with `MPI_Test` calls in the debug section for all processes with $N = 1$ using the `Show` function.
- MPI5Comm39***. An integer N is given in each process; the number N can take two values: 0 and 1 (there is at least one process with $N = 1$). Also, a sequence A of integers is given in each process with $N = 1$ and it is known that the sequence A in the first process with $N = 1$ includes one number, the sequence A in the second process with $N = 1$ includes two numbers, and so on. Use the `MPI_lallgatherv` function to send the sequences of numbers A to all processes with $N = 1$ and output the numbers in each process in ascending order of the ranks of the processes that sent them (including the numbers received from the same process). Use the `MPI_Wait` function to check if the `MPI_lallgatherv` non-blocking operation finishes. Additionally, display the duration of each call of `MPI_Wait` in milliseconds for all processes with $N = 1$ in the debug section. For this purpose, call `MPI_Wtime` before and after `MPI_Wait` and use the `Show` function to display the difference of values returned by `MPI_Wtime` multiplied by 1000.
- MPI5Comm40***. A sequence of $2K$ integers is given in each process of even rank; K is the number of even-rank processes. Using the `MPI_lalltoall` function, send two elements of each given sequence to each process of even rank as follows: the initial two elements of each sequence should be sent to the master process, the next two elements of each sequence should be sent to the process of rank 2, and so on. Output received numbers in each even-rank process in ascending order of ranks of sending processes (including the numbers received from the same process). Use the `MPI_Test` function to check if the `MPI_lalltoall` non-blocking operation finishes; call the `MPI_Test` function in a loop until it returns a non-zero flag. Additionally, display the number of required loop iterations with `MPI_Test` calls in the debug section for all even-rank processes using the `Show` function.
- MPI5Comm41***. A sequence of $K + 1$ integers is given in each process of odd rank; K is the number of odd-rank processes. Using the `MPI_lalltoallv` function, send two elements of each given sequence to each odd-rank process as follows: the initial two elements of each sequence should be sent to the

process 1, the second element and the third element of each sequence should be sent to the process 3, ..., the last two elements of each sequence should be sent to the last odd-rank process. Output received numbers in each odd-rank process in ascending order of ranks of sending processes (including the numbers received from the same process). Use `MPI_Wait` function to check if `MPI_alltoallv` non-blocking operation finishes. Additionally, display the duration of each call to the `MPI_Wait` function in milliseconds for all odd-rank processes in the debug section; for this purpose, call `MPI_Wtime` before and after `MPI_Wait` and use the `Show` function to display the difference of the values returned by `MPI_Wtime` multiplied by 1000.

MPI5Comm42*. A sequence of $K + 5$ real numbers is given in each process of even rank; K is the number of even-rank processes. Find the maximal value among the elements of all given sequences with the same order number using the `MPI_reduce` function with the `MPI_MAX` operation. Output received maximal values in the master process. Use the `MPI_Test` function to check if the `MPI_reduce` non-blocking operation finishes; call the `MPI_Test` function in a loop until it returns a non-zero flag. Additionally, display the number of required loop iterations with `MPI_Test` calls in the debug section for all even-rank processes using the `Show` function.

MPI5Comm43*. An integer N is given in each process; the number N can take two values: 0 and 1 (there is at least one process with $N = 1$). Also a sequence of K real numbers, where K is the number of processes with $N = 1$, is given in each process with $N = 1$. Find products of elements of all given sequences with the same order number using the `MPI_allreduce` function with the `MPI_PROD` operation. Output received products in each process with $N = 1$. Use the `MPI_Wait` function to check if the `MPI_allreduce` non-blocking operation finishes. Additionally, display the duration of each call of `MPI_Wait` in milliseconds for all processes with $N = 1$ in the debug section. For this purpose, call `MPI_Wtime` before and after `MPI_Wait` and use the `Show` function to display the difference of values returned by `MPI_Wtime` multiplied by 1000.

MPI5Comm44*. The number of processes is $2K$, and it is known that the number K is odd. A sequence of N integers is given in each process with ranks from 0 to K , where $N = 3(1 + K)/2$. Using the `MPI_reduce_scatter` function, find sums of elements of all given sequences with the same order number and send sums to processes with ranks from 0 to K as follows: the first two sums should be sent to the process 0, the third sum should be sent to the process 1, the fourth and the fifth sum should be sent to the process 2, the sixth sum should be sent to the process 3, and so on (two sums are sent to the even-rank processes and one sum is sent to the odd-rank processes). Output the received sums in each process. Use the `MPI_Test` function to

check if the `MPI_lreduce_scatter` non-blocking operation finishes; call the `MPI_Test` function in a loop until it returns a non-zero flag. Additionally, display the number of required loop iterations with `MPI_Test` calls in the debug section for processes with ranks from 0 to K using the `Show` function.

MPI5Comm45*. A sequence of $3K$ integers is given in each process of even rank; K is the number of even-rank processes. Using the `MPI_lreduce_scatter_block` function, find sums of elements of all given sequences with the same order number and send three sums to each even-rank process as follows: the first three sums should be sent to the process 0, the next three sums should be sent to the process 2, and so on. Output the received sums in each process. Use the `MPI_Wait` function to check if the `MPI_lreduce_scatter_block` non-blocking operation finishes. Additionally, display the duration of each call of `MPI_Wait` in milliseconds for all even-rank processes in the debug section. For this purpose, call `MPI_Wtime` before and after `MPI_Wait` and use the `Show` function to display the difference of values returned by `MPI_Wtime` multiplied by 1000.

Note. The corresponding blocking collective function `MPI_Reduce_scatter_block` appeared in the MPI-2 standard. It simplifies the distribution of reduction results to different processes (compared to the `MPI_Reduce_scatter` function) if each process must receive a reduction dataset of the same size.

MPI5Comm46*. A sequence of $K + 5$ integers is given in each even-rank process; $2K$ is the number of processes. Using the `MPI_lscan` function, find maximal values among elements of given sequences with the same order number as follows: the maximal values of elements of sequences given in the processes of even ranks 0, 2, ..., R should be found in the process R ($R = 0, 2, \dots, 2K - 2$). Output received data in each process. Use the `MPI_Test` function to check if the `MPI_lscan` non-blocking operation finishes; call the `MPI_Test` function in a loop until it returns a non-zero flag. Additionally, display the number of required loop iterations with `MPI_Test` calls in the debug section for even-rank processes using the `Show` function.

MPI5Comm47*. A sequence of $K + 5$ integers is given in each odd-rank process; $2K$ is the number of processes. Using the `MPI_lexscan` function, find minimal values among elements of given sequences with the same order number as follows: the minimal values of elements of sequences given in the processes of odd ranks 1, 3, ..., $R - 2$ should be found in the process of rank R , $R = 3, 5, \dots, 2K - 1$). Output received data in processes of rank R . Use `MPI_Wait` function to check if `MPI_lexscan` non-blocking operation finishes. Additionally, display the duration of each call to the `MPI_Wait` function in milliseconds for even-rank processes in the debug section; for this purpose, call `MPI_Wtime` before and after `MPI_Wait` and use the `Show` function

to display the difference of the values returned by `MPI_Wtime` multiplied by 1000.

Note. The corresponding blocking collective function `MPI_Exscan` ("exclusive scan") appeared in the MPI-2 standard. It is more general than the `MPI_Scan` function, because it allows to model the `MPI_Scan` function ("inclusive scan") without performing additional collective operations.

2.6. Parallel file input-output (MPI-2)

Use the `char[12]` array to store the filename, use the `MPI_Bcast` function with the `MPI_CHAR` datatype parameter to send the filename from the master process to the slave processes.

You do not need to set the view of the data in the file by means the `MPI_File_set_view` function in the initial two subgroups (`MPI6File1–MPI6File16`); it is enough to use the default view, in which both the elementary datatype and the filetype are of the `MPI_BYTE` type, the initial displacement is equal to 0 for all processes, and the "native" data representation is used. The same data representation should be specified for the file view in the tasks of the third subgroup (`MPI6File17–MPI6File30`).

Use the `MPI_Type_size` function to determine the size of the `MPI_INT` and `MPI_DOUBLE` types.

Use the function `MPI_Type_create_resized` to specify the additional empty space in tasks devoted to the file view setting. One can use also *the zero-size upper-bound marker* `MPI_UB`, but this pseudo-datatype is deprecated in the MPI-2 standard.

Before solving the tasks in this section, you should study the `MPI6File26` task solution given in Section 1.3.3.

2.6.1. Local functions for file input-output

MPI6File1. The name of existing file of integers is given in the master process; the amount of file items, which should be read, and their ordinal numbers are given in each slave process (the file items are numbered from 1). File items with some numbers may be missing in the source file. Using the required number of the `MPI_File_read_at` local function calls in each process, read existing file items with the specified ordinal numbers from the source file and output them in the same order. To check existence of file item with the specified ordinal number, you can either use the `MPI_File_get_size` function, or analyze the `MPI_Status` parameter of the `MPI_File_read_at` function.

MPI6File2. The name of file is given in the master process; the amount of pairs of integers and the pairs themselves are given in each slave process; the first term of the pair is the ordinal number of the file item, the second term of the pair is the value of this file item (the file items are numbered from 1,

all ordinal numbers are different and cover range from 1 to some integer). Create a new file of integers with the given name and write the given data to this file using the required number of the `MPI_File_write_at` local function calls in each slave process.

MPI6File3. The name of existing file of real numbers is given in the master process. The file contains elements of the $K \times N$ matrix, where K is the number of slave processes. Using one call of the `MPI_File_read_at` local function in each slave process, read and output elements of R th matrix row in the process of rank R (rows are numbered from 1). Use the `MPI_File_get_size` function to determine the file size.

MPI6File4. The name of file is given in the master process; a sequence of R real numbers is given in each slave process, where R is the process rank. Create a new file of real numbers with the given name and write the given data to this file in ascending order of ranks of processes containing these data. Use one call of the `MPI_File_write_at` local function in each slave process.

MPI6File5. The name of existing file of integers is given in the master process. The file contains all integers in the range from 1 to K , where K is the maximal rank of process. Read and output two sequences of the file items in each slave process. The first sequence contains the *initial* part of the file items until the first item whose value is equal to the process rank (including this item); the second sequence contains the *final* part of the file items and has the same size as the first one. Elements of each sequence should be output in the order they are stored in the file. Use the required number of the `MPI_File_read` local function calls in each process (without using arrays) and also the `MPI_File_get_position` function call to determine the size of the first sequence and the `MPI_File_seek` function call with the `MPI_SEEK_END` parameter to move the file pointer to the beginning of the second sequence.

MPI6File6. The name of file is given in the master process; an integer is given in each slave process. Create a new file of integers with the given name and write K successive copies of each given integer to this file, where K is the number of slave processes. The order of integers in the file should be the inverse of the order of the slave processes (i. e., K copies of the integer from the process 1 should be written at the end of file, K copies of the integer from the process 2 should be written before them, and so on). Use one call of the `MPI_File_write` local function in each slave process and also the `MPI_File_seek` function with the `MPI_SEEK_SET` parameter.

MPI6File7. The name of existing file of integers is given in the master process. The sum of all file item values is greater than K , where K is the number of slave processes. Read initial file items in each slave process until the sum of their values exceeds the rank of process and output this sum and the amount N of read numbers. After that, in addition, read and output the val-

ues of the *last* N file items in the order they are stored in the file. Use the required number of the `MPI_File_read` local function calls in each slave process (without using arrays) and also the `MPI_File_get_position` function call to determine the amount N and the `MPI_File_seek` function call with the `MPI_SEEK_END` parameter to move the file pointer to beginning of the group of the last N items.

MPI6File8. The name of file is given in the master process; a real number is given in each slave process. Create a new file of real numbers with the given name and write R successive copies of each given real number to this file, where R is equal to the rank of the process in which this number is given. The order of numbers in the file should be the *inverse* of the order of the slave processes (i. e., single copy of the real number from the process 1 should be written at the end of file, two copies of the real number from the process 2 should be written before it, and so on). Use one call of the `MPI_File_write` local function in each slave process and also the `MPI_File_seek` function with the `MPI_SEEK_SET` parameter.

2.6.2. Collective functions for file input-output

MPI6File9. The name of existing file of integers is given in the master process. Read and output the $R + 1$ file items in each process starting with the item with the ordinal number $R + 1$, where R is the process rank (0, 1, ...). File items are numbered from 1; thus, you should read only the initial file item in the process 0, the two following items in the process 1, the three items starting with the third one in the process 2, and so on. If the file does not contain enough items, then, in some processes, the number of output items may be less than the required one. Use one call of the `MPI_File_read_all` collective function call and also the `MPI_File_get_size` function to determine the file size.

Note. In MPICH2 1.3, the function `MPI_File_read_all` does not allow to determine the number of actually read file items based on the information contained in the `MPI_Status` parameter: this parameter always contain the number of items to be read, and if there are not enough items in the file, then zero values are appended to the output array.

MPI6File10. The name of file is given in the master process; a sequence of R integers is given in each slave process, where R is the process rank (1, 2, ...). Create a new file of integers with the given name and write the given sequences to this file in ascending order of ranks of processes containing these sequences. Use one call of the `MPI_File_write_all` collective function (for all processes including the process 0) and also the `MPI_File_seek` function with the `MPI_SEEK_SET` parameter.

- MPI6File11***. The name of existing file of real numbers is given in the master process. In addition, an integer is given in each process. This integer is equal to 0 or it is equal to the ordinal number of one of the existing items of the file (the file items are numbered from 1). Using the `MPI_Comm_split` function, create a new communicator containing only those processes in which a non-zero integer is given. Using the `MPI_File_read_at_all` collective function for all processes of this communicator, read and output a file item located at a position with the given ordinal number.
- MPI6File12***. The name of existing file of real numbers is given in the master process. In addition, an integer is given in each process. This integer is equal to 0 or it is equal to the ordinal number of one of the existing items of the file (the file items are numbered from 1). Using the `MPI_Comm_split` function, create a new communicator containing only those processes in which a non-zero integer is given. Using the `MPI_File_write_at_all` collective function for all processes of this communicator, replace the value of the file item that has the given ordinal number, by the value of the process rank in the new communicator (the rank should be converted to a real number).
- MPI6File13**. The name of existing file of integers is given in the master process. In addition, an integer is given in each process. This integer is equal to 0 or 1. Using the `MPI_Comm_split` function, create a new communicator containing only those processes in which the number 1 is given. Using the `MPI_File_read_ordered` collective function for all processes of this communicator, read and output the $R + 1$ file item, where R is the process rank in the new communicator (items should be read in ascending order: the first item in the process 0, two next items in the process 1, three next items in the process 2, and so on). If the file does not contain enough items, then, in some processes, the number N of output items may be less than required one or even may be equal to zero. In addition, output the number N of actually read items and the new value P of the shared file pointer in each process of the new communicator. Use the `MPI_Status` parameter of the `MPI_File_read_ordered` function to determine the number N of actually read items. Use the `MPI_File_get_position_shared` function to determine the current value of P (this value should be the same in all processes).
- MPI6File14**. The name of file is given in the master process. In addition, an integer N is given in each process. Create a new file of integers with the given name. Using the `MPI_Comm_split` function, create a new communicator containing only those processes in which a non-zero integer is given. Using the `MPI_File_write_ordered` collective function for all processes of the new communicator, write K successive copies of each given integer N to this file, where K is the number of processes in the *new* communicator. The in-

tegers N should be written to the file in the ascending order of ranks of processes containing these integers.

MPI6File15*. The name of existing file of integers is given in the master process. The file contains at least K items, where K is the number of processes. Using the `MPI_Comm_split` function, create a new communicator containing only processes with odd rank (1, 3, ...). Using one call of the `MPI_File_seek_shared` and `MPI_File_read_ordered` collective functions, read and output two file items at a time in each process of the new communicator: the second and the first item from the end (in this order) should be read and output in the process of the rank 1 in the `MPI_COMM_WORLD` communicator, the fourth and third item from the end should be read and output in the process of the rank 3, and so on.

Note. To ensure the required order of data reading in the `MPI_File_read_ordered` function, you should inverse the order of the processes in the created communicator (in comparison with the processes in the `MPI_COMM_WORLD` communicator).

MPI6File16*. The name of file is given in the master process. In addition, an integer N (≥ 0) and N real numbers are given in each process. Create a new file of real numbers with the given name. Using the `MPI_Comm_split` function, create a new communicator containing only those processes in which a non-zero integer N is given. Using one call of the `MPI_File_write_ordered` collective function for all processes of the new communicator, write the given real numbers to the file in the *inverse* order: at first, all the real numbers from the process with the maximal rank in the communicator `MPI_COMM_WORLD` should be written (in inverse order), after that, all the numbers from the process with the previous rank, and so on.

Note. To ensure the required order of data writing in the `MPI_File_write_ordered` function, you should inverse the order of the processes in the created communicator (in comparison with the processes in the `MPI_COMM_WORLD` communicator).

2.6.3. File view setting for file input-output

MPI6File17. The name of existing file of integers is given in the master process. The file contains $2K$ items, where K is the number of processes. Using one call of the `MPI_File_read_all` collective function (and without using the `MPI_File_seek` function), read and output two file items at a time in each process. The file items should be read and output in the order in which they are stored in a file. To do this, use the `MPI_File_set_view` function to define a new file view with the `MPI_INT` elementary datatype, the same filetype, and the appropriate displacement (the displacement will be different for different processes).

- MPI6File18.** The name of existing file of integers is given in the master process. The file contains elements of the $K \times 5$ matrix, where K is the number of processes. In addition, an integer N ($1 \leq N \leq 5$) is given in each process; this integer determines the ordinal number of a *selected* element in some matrix row, namely, in the first row for the process 0, in the second row for the process 1, and so on (the row elements are numbered from 1). Using one call of the `MPI_File_write_at_all` collective function with the second parameter equal to $N - 1$, replace the value of the selected element in each matrix row by the rank of the corresponding process (the selected element in the first row should be replaced by 0, the selected element in the second row should be replaced by 1, and so on). To do this, use the `MPI_File_set_view` function to define a new file view with the `MPI_INT` elementary datatype, the same filetype, and the appropriate displacement (the displacement will be different for different processes).
- MPI6File19.** The name of existing file of real numbers is given in the master process. The file contains elements of the $K \times 6$ matrix, where K is the number of processes. In addition, an integer N ($1 \leq N \leq 6$) is given in each process; this integer determines the ordinal number of a *selected* element in some matrix row, namely, in the last row for the process 0, in the last but one row for the process 1, and so on (the row elements are numbered from 1). Using one call of the `MPI_File_read_at_all` collective function with the second parameter equal to $N - 1$, read and output the value of the selected row element in the corresponding process (the selected element in the first row should be output in the last process, the selected element in the second row should be output in the last but one process, and so on). To do this, use the `MPI_File_set_view` function to define a new file view with the `MPI_DOUBLE` elementary datatype, the same filetype, and the appropriate displacement (the displacement will be different for different processes).
- MPI6File20.** The name of file is given in the master process. In addition, a sequence of $R + 1$ real numbers is given in each process, where R is the process rank (0, 1, ...). Create a new file of real numbers with the given name. Using one call of the `MPI_File_write_all` collective function (and without using the `MPI_File_seek` function), write all given numbers to the file in the order that is inverse to the order in which they are given in processes: at first, the numbers from the last process (in the inverse order) should be written, after that, the numbers from the last but one process (in the inverse order), and so on. To do this, use the `MPI_File_set_view` function to define a new file view with the `MPI_DOUBLE` elementary datatype, the same filetype, and the appropriate displacement (the displacement will be different for different processes).

- MPI6File21***. The name of existing file of integers is given in the master process. The file contains $3K$ items, where K is the number of processes. Read and output three file items, namely, A, B, C , in each process. These items are located in the given file as follows: $A_0, A_1, \dots, A_{K-1}, B_0, B_1, \dots, B_{K-1}, C_0, C_1, \dots, C_{K-1}$ (an index indicates the process rank). To do this, use one call of the `MPI_File_read_all` collective function and a new file view with the `MPI_INT` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of an integer and a terminal empty space of a size equal to the extent of $K - 1$ integers.
- MPI6File22***. The name of file is given in the master process. In addition, three integers, namely, A, B, C , are given in each process. The number of processes is equal to K . Create a new file of integers with the given name and write the given integers to this file as follows: $A_{K-1}, A_{K-2}, \dots, A_0, B_{K-1}, B_{K-2}, \dots, B_0, C_{K-1}, C_{K-2}, \dots, C_0$ (an index indicates the process rank). To do this, use one call of the `MPI_File_write_all` collective function and a new file view with the `MPI_INT` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of an integer and a terminal empty space of a size equal to the extent of $K - 1$ integers.
- MPI6File23****. The name of existing file of real numbers is given in the master process. The file contains $6K$ items, where K is the number of processes. Read and output six file items, namely, A, B, C, D, E, F , in each process. These items are located in the given file as follows: $A_0, B_0, C_0, A_1, B_1, C_1, \dots, A_{K-1}, B_{K-1}, C_{K-1}, D_0, E_0, F_0, D_1, E_1, F_1, \dots, D_{K-1}, E_{K-1}, F_{K-1}$ (an index indicates the process rank). To do this, use one call of the `MPI_File_read_all` collective function and a new file view with the `MPI_DOUBLE` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of three real numbers and a terminal empty space of the appropriate size.
- MPI6File24****. The name of file is given in the master process. In addition, four real numbers, namely, A, B, C, D , are given in each process. The number of processes is equal to K . Create a new file of real numbers with the given name and write the given real numbers to this file as follows: $A_{K-1}, B_{K-1}, A_{K-2}, B_{K-2}, \dots, A_0, B_0, C_{K-1}, D_{K-1}, C_{K-2}, D_{K-2}, \dots, C_0, D_0$ (an index indicates the process rank). To do this, use one call of the `MPI_File_write_all` collective function and a new file view with the `MPI_DOUBLE` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of two real numbers and a terminal empty space of the appropriate size.

MPI6File25.** The name of file is given in the master process. In addition, $3 \cdot (R + 1)$ integers are given in the process of rank R ($R = 0, 1, \dots, K - 1$, where K is the number of processes): 3 integers, namely, A, B, C , are given in the process 0, 6 integers, namely, A, A', B, B', C, C' , are given in the process 1, 9 integers, namely, $A, A', A'', B, B', B'', C, C', C''$, are given in the process 2, and so on. Create a new file of real numbers with the given name and write the given integers to this file as follows: $A_0, A_1, A'_1, A_2, A'_2, A''_2, \dots, B_0, B_1, B'_1, B_2, B'_2, B''_2, \dots$ (an index indicates the process rank). To do this, use one call of the `MPI_File_write_all` collective function and a new file view with the `MPI_INT` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of $R + 1$ integers and a terminal empty space of the appropriate size.

MPI6File26°.** The name of file is given in the master process. In addition, four real numbers, namely, A, B, C, D , are given in each process. The number of processes is equal to K . Create a new file of real numbers with the given name and write the given real numbers to this file as follows: $A_0, A_1, \dots, A_{K-1}, B_{K-1}, \dots, B_1, B_0, C_0, C_1, \dots, C_{K-1}, D_{K-1}, \dots, D_0$ (an index indicates the process rank). To do this, use one call of the `MPI_File_write_all` collective function and a new file view with the `MPI_DOUBLE` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of two real numbers (with an additional empty space between these numbers) and a terminal empty space of the appropriate size.

Remark. The solution of this task is given in Section 1.3.3.

MPI6File27*. The name of existing file of real numbers is given in the master process. The file contains elements of the $(K/2) \times K$ matrix, where K is the number of processes (K is an even number). Read and output elements of $(R + 1)$ th matrix column in the process of rank R ($R = 0, \dots, K - 1$, columns are numbered from 1). To do this, use one call of the `MPI_File_read_all` collective function and a new file view with the `MPI_DOUBLE` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of one real number and a terminal empty space of the appropriate size.

MPI6File28*. The name of file is given in the master process. In addition, an integer N and a sequence of $K/2$ real numbers are given in each process, where K is the number of processes (K is an even number). The numbers N are different for all processes and are in the range from 1 to K . Create a new file of real numbers with the given name. Write a $(K/2) \times K$ matrix to this file; each process should write its sequence of real numbers into a column of the matrix with the ordinal number N (the columns are numbered

from 1). To do this, use one call of the `MPI_File_write_all` collective function and a new file view with the `MPI_DOUBLE` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of one real number and a terminal empty space of the appropriate size.

MPI6File29.** The name of existing file of integers is given in the master process. The file contains elements of a block matrix of the size $(K/3) \times 3$ (the size is indicated in blocks), where K is the number of processes (K is a multiple of 3). Each block is a square matrix of order N (all the numbers N are the same and are in the range from 2 to 5). Read and output one block of the given matrix in each process in a row-major order of blocks. To do this, use one call of the `MPI_File_read_all` collective function and a new file view with the `MPI_INT` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of N integers and a terminal empty space of the appropriate size. Use the `MPI_File_get_size` function to determine the number N .

MPI6File30.** The name of file and an integer N are given in the master process (N is in the range 2 to 5). In addition, two integers I and J are given in each process. The integers I and J determine a position (that is, row and column numbers) of some square block of a block matrix of the size $(K/3) \times 3$ (the size is indicated in blocks), where K is the number of processes (K is a multiple of 3). The integers I are in the range of 1 to $K/3$, the integers J are in the range 1 to 3; all processes contain different positions of blocks. Each block is a square matrix of order N . Create a new file of integers with the given name. Write a $(K/3) \times 3$ block matrix to this file; each process should write a matrix block to the block position (I, J) . All the elements of the block written by the process of rank R ($R = 0, 1, \dots, K - 1$) should be equal to the number R . To do this, use one call of the `MPI_File_write_all` collective function and a new file view with the `MPI_INT` elementary datatype, the appropriate displacement (the displacement will be different for different processes), and a new filetype that consists of N integers and a terminal empty space of the appropriate size. Use the `MPI_Bcast` collective function to send the value of N to all processes.

2.7. One-sided communications (MPI-2)

When defining an access window object using the `MPI_Win_create` function, it is recommended to specify the `disp_unit` displacement (the third parameter) equal to the size of the data item of the corresponding type (it is always either `MPI_INT` or `MPI_DOUBLE` in all tasks, so the size can be obtained using the `MPI_Type_size` function). In this case, one can indicate the `target_disp` displacement (the fifth parameter of the `MPI_Get`, `MPI_Put`, `MPI_Accumulate` functions) equal to

the initial index of the used part of the array (rather than the number of bytes from the beginning of the array to the required part, as in the case of the `disp_unit` parameter equal to 1).

It is suffice to specify the `MPI_INFO_NULL` constant as the info parameter (the fourth parameter of the `MPI_Win_create` function).

If you do not need to create a window in some processes, then the value 0 should be specified as the size parameter (the second parameter of the `MPI_Win_create` function) in these processes.

It is suffice to specify a constant 0 as the assert parameter in all synchronizing functions (`MPI_Win_fence`, `MPI_Win_start`, `MPI_Win_post`, `MPI_Win_lock`); this parameter is the last but one parameter in all these functions.

In the first subgroup (`MPI7Win1–MPI7Win17`), you should use the `MPI_Win_fence` *collective* function as a synchronizing function that should be called both before the actions related to the one-sided data transfer and after these actions, but before the actions related to access to the transferred data.

The tasks of the second subgroup (`MPI7Win18–MPI7Win30`) require the use of *local* synchronization: the `MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_post`, `MPI_Win_wait` functions or a pair of the `MPI_Win_lock`, `MPI_Win_unlock` functions. In the tasks of this subgroup, there is always specified which kind of the local synchronization you should use.

Before solving the tasks in this section, you should study the `MPI7Win13` and `MPI7Win23` task solutions given in Sections 1.3.5–1.3.6.

2.7.1. One-sized communications with the simplest synchronization

MPI7Win1. An integer is given in each slave process. Create an access window of the size K of integers in the master process (K is the number of slave processes). Using the `MPI_Put` function call in the slave processes, send all the given integers to the master process and output received integers in the ascending order of ranks of sending processes.

MPI7Win2. A sequence of R real numbers is given in each slave process, where R is the process rank (1, 2, ...). Create an access window of the appropriate size in the master process. Using the `MPI_Put` function call in the slave processes, send all the given real numbers to the master process and output received numbers in the ascending order of ranks of sending processes.

MPI7Win3. An array A of K integers is given in the master process, where K is the number of slave processes. Create an access window containing the array A in the master process. Using the `MPI_Get` function call in the slave processes, receive and output one element of the array A in each slave process. Elements of the array A should be received in the slave processes in descending order of their indices (that is, the element A_0 should be re-

ceived in the last process, the element A_1 should be received in the last but one process, and so on).

MPI7Win4. An array A of $K + 4$ real numbers is given in the master process, where K is the number of slave processes. Create an access window containing the array A in the master process. Using the `MPI_Get` function call in the slave processes, receive and output five elements of the array A in each slave process starting with the element of index $R - 1$, where R is the slave process rank ($R = 1, 2, \dots, K - 1$).

MPI7Win5. An array A of K integers is given in the master process, where K is the number of slave processes. In addition, an index N (an integer in the range 0 to $K - 1$) and an integer B are given in each slave process. Create an access window containing the array A in the master process. Using the `MPI_Accumulate` function call in the slave processes, multiply the element A_N by the number B and then output the modified array A in the master process.

Note. Some slave processes can contain the same value of N ; in this case the element A_N will be multiplied several times. This circumstance does not require additional synchronization due to the features of the `MPI_Accumulate` function implementation.

MPI7Win6. An array A of $2K - 1$ real numbers is given in the master process (K is the number of slave processes), and array B of R real numbers is given in each slave process (R is the process rank, $R = 1, 2, \dots, K - 1$). Create an access window containing the array A in the master process. Using the `MPI_Accumulate` function call in the slave processes, add the values of all the elements of array B from the process of rank R to the elements of array A starting with the index $R - 1$ (the single element B_0 from the process 1 should be added to the element A_0 , the elements B_0 and B_1 from the process 2 should be added to the elements A_1 and A_2 respectively, the elements B_0, B_1 , and B_2 from the process 3 should be added to elements A_2, A_3 , and A_4 respectively, and so on). Output the modified array A in the master process.

Note. Elements of array A , starting from the index 2, will be modified several times by adding values from the different slave processes. This circumstance does not require additional synchronization due to the features of the `MPI_Accumulate` function implementation.

MPI7Win7*. An array A of $2K$ integers is given in the master process, where K is the number of slave processes. Create an access window containing two integers in each slave process. Using the `MPI_Put` function call in the master process, send and output two elements of the array A in each slave process. Elements of the array A should be sent to slave processes in ascending order of their indices (that is, the elements A_0 and A_1 should be sent to the

process 1, the elements A_2 and A_3 should be sent to the process 2, and so on).

MPI7Win8*. An integer R and a real number B are given in each process. All the integers R are different and are in the range from 0 to $K - 1$, where K is the number of processes. Create an access window containing one real number in each process. Using the `MPI_Put` function call in each process, send the number B from this process to the process R and output received numbers in all processes.

MPI7Win9*. An array A of K integers is given in each process, where K is the number of processes. Create an access window containing the array A in each process. Using several calls of the `MPI_Get` function in each process R ($R = 0, \dots, K - 1$), receive and output elements of all arrays A with the index R . Received elements should be output in descending order of ranks of sending processes (that is, the element received from the process $K - 1$ should be output first, the element received from the process $K - 2$ should be output second, and so on).

Note. The function `MPI_Get`, as well as other one-way communication functions, can also be used to access the window created in the calling process.

MPI7Win10*. An array A of 5 real numbers and integers N_1 and N_2 are given in each process. Each of the numbers N_1 and N_2 is in the range 0 to 4. Create an access window containing the array A in each process. Using two calls of the `MPI_Get` function in each process, receive and output the element of index N_1 from the array A of the previous process and then receive and output the element of index N_2 from the array A of the next process (the numbers N_1 and N_2 are taken from the calling process, processes are taken in a cyclic order).

MPI7Win11*. The number of processes K is an even number. An array A of $K/2$ integers is given in each process. Create an access window containing the array A in all the odd-rank processes (1, 3, ..., $K - 1$). Using the required number of calls of the `MPI_Accumulate` function in each even-rank process, add the element $A[I]$ of the process $2J$ to the element $A[J]$ of the process $2I + 1$ and output the changed arrays A in all the odd-rank processes.

Note. The required changing of the given arrays can be described in the another way: if B denotes a matrix of order $K/2$ whose rows coincide with the arrays A given in the even-rank processes and C denotes a matrix of the same order whose rows coincide with the arrays A given in the odd-rank processes, then the matrix C should be transformed as follows: elements of the row I of the matrix B should be added to the corresponding elements of the column I of the matrix C .

MPI7Win12*. Solve the MPI7Win11 task by creating access windows in even-rank processes and using the MPI_Get function calls instead of the MPI_Accumulate function calls in odd-rank processes.

Note. Since the numbers received from the even-rank processes must be added to the elements of array A after the second MPI_Win_fence synchronization function call, it is convenient to use an auxiliary array to store the received numbers.

MPI7Win13*°. Three integers N_1, N_2, N_3 are given in each process; each given integer is in the range 0 to $K - 1$, where K is the number of processes (the values of some of these integers in each process may coincide). In addition, an array A of $R + 1$ real numbers is given in each process, where R is the process rank (0, ..., $K - 1$). Create an access window containing the array A in all the processes. Using three calls of the MPI_Accumulate function in each process, add the integer $R + 1$ to all elements of the arrays A given in the processes N_1, N_2, N_3 , where R is the rank of the process that calls the MPI_Accumulate function (for instance, if the number N_1 in the process 3 is equal to 2, then a real number 4.0 should be added to all the elements of array A in the process 2). If some of the integers N_1, N_2, N_3 coincide in the process R , then the number $R + 1$ should be added to the elements of the corresponding arrays several times. Output the changed arrays A in each process.

Remark. The solution of this task is given in Section 1.3.5.

MPI7Win14*. An array of K real numbers is given in each process, where K is the number of processes. The given array contains a row of an upper triangular matrix A , inclusive of its zero-valued part (the process of rank R contains the R th row of the matrix, the rows are numbered from 0). Create an access window containing the given array in all the processes. Using the required number of calls of the MPI_Get function in each process, write the rows of the matrix transposed to the given matrix A (inclusive of its zero-valued part) in the given arrays. Then output the changed arrays in each process. Do not use auxiliary arrays.

Notes. (1) The rows of the transposed matrix coincide with the columns of the original matrix, so the resulting matrix will be the lower triangular one.

(2) You should write zero values to the required array elements only after the second call of the MPI_Win_fence function. (3) You do not need to create an access window for the last process.

MPI7Win15*. Solve the MPI7Win14 task by using the MPI_Put function calls instead of the MPI_Get function calls.

Note. In this case, you do not need to create an access window for the master process.

MPI7Win16.** One row of the square real-valued matrix A of order K is given in each process, where K is the number of processes (the process of rank R contains the R th row of the matrix, the rows are numbered from 0). In addition, a real number B is given in each process. Create an access window containing the given row of the matrix A in all the processes. Using the required number of calls of the `MPI_Accumulate` function in each process R ($R = 0, \dots, K - 1$), change the matrix row given in the next process as follows: all row elements that are less than the number B from the process R should be replaced by this number B (processes are taken in a cyclic order). Then, using K calls of the `MPI_Get` function in each process, receive and output the R th column of the transformed matrix A in the process R ($R = 0, \dots, K - 1$, the columns are numbered from 0).

Note. You should call the `MPI_Win_fence` synchronization function *three times* in each process.

MPI7Win17.** One row of the square real-valued matrix A of order K is given in each process, where K is the number of processes (the process of rank R contains the R th row of the matrix, the rows are numbered from 0). In addition, a real number B is given in each process. Create an access window containing the given row of the matrix A in all the processes. Using the required number of calls of the `MPI_Accumulate` function in each process R ($R = 0, \dots, K - 1$), change the matrix row given in the previous process as follows: all row elements that are greater than the number B from the process R should be replaced by this number B (processes are taken in a cyclic order). Then, using K calls of the `MPI_Accumulate` function in each slave process, add the first element of the row from each slave process R ($1, \dots, K - 1$) to all the elements of the R th column of the transformed matrix A (the columns are numbered from 0). Output the new contents of the given row of the matrix A in each process after all transformations.

Note. You should call the `MPI_Win_fence` synchronization function *three times* in each process.

2.7.2. Additional types of synchronization

MPI7Win18. The number of processes K is an even number. An integer A is given in each even-rank process ($0, 2, \dots, K - 2$). Create an access window containing one integer in all the odd-rank processes ($1, 3, \dots, K - 1$). Using the `MPI_Put` function call in each even-rank process $2N$, send the integer A to the process $2N + 1$ and output the received integers. Use the `MPI_Win_start` and `MPI_Win_complete` synchronization functions in the even-rank processes and the `MPI_Win_post` and `MPI_Win_wait` synchronization functions in the odd-rank processes. Use the `MPI_Group_incl` function to create a group of processes specified as the first parameter of the `MPI_Win_start` and `MPI_Win_post` functions. The `MPI_Group_incl` function should be applied to the

group of the `MPI_COMM_WORLD` communicator (use the `MPI_Comm_group` function to obtain the group of the `MPI_COMM_WORLD` communicator).

Note. Unlike the `MPI_Win_fence` *collective* synchronization function, used in previous tasks, the synchronization functions used in this and the subsequent tasks are *local* ones and, in addition, allow to specify the groups of origin and target processes for one-way communications.

MPI7Win19. An array A of K real numbers is given in the master process, where K is the number of slave processes. Create an access window containing the array A in the master process. Using the `MPI_Get` function call in each slave process, receive and output one of elements of the array A . The elements should be received in descending order of their indices (that is, the element with the index $K - 1$ should be received in the process 1, the element with the index $K - 2$ should be received in the process 2, and so on). Use the `MPI_Win_start` and `MPI_Win_complete` synchronization functions in the slave processes and the `MPI_Win_post` and `MPI_Win_wait` synchronization functions in the master process. Use the `MPI_Group_incl` function to create a group of processes specified as the first parameter of the `MPI_Win_start` function, use the `MPI_Group_excl` function to create a group of processes specified as the first parameter of the `MPI_Win_post` function. The `MPI_Group_incl` and `MPI_Group_excl` functions should be applied to the group of the `MPI_COMM_WORLD` communicator.

MPI7Win20. The number of processes K is a multiple of 3. An array A of 3 real numbers is given in the processes of rank $3N$ ($N = 0, \dots, K/3 - 1$). Create an access window containing the array A in all processes in which this array is given. Using one call of the `MPI_Get` function in the processes of rank $3N + 1$ and $3N + 2$ ($N = 0, \dots, K/3 - 1$), receive and output one element A_0 and two elements A_1, A_2 respectively from the process $3N$ (namely, the process 1 should output the element A_0 received from the process 0, the process 2 should output the elements A_1 and A_2 received from the process 0, the process 4 should output the element A_0 received from the process 3, and so on). Use the `MPI_Win_post` and `MPI_Win_wait` synchronization functions in the processes of rank $3N$ and the `MPI_Win_start` and `MPI_Win_complete` synchronization functions in the other processes.

MPI7Win21*. The number of processes K is an even number. An array A of $K/2$ real numbers and an array N of $K/2$ integers are given in the master process. All the elements of the array N are distinct and are in the range 1 to $K - 1$. Create an access window containing one real number in each slave process. Using the required number of calls of the `MPI_Put` function in the master process, send the real number A_I to the slave process of rank N_I ($I = 0, \dots, K/2 - 1$). Output the received number (or 0.0 if the process did not receive data) in each slave process. Use the `MPI_Win_post` and `MPI_Win_wait` synchro-

nization functions in the slave processes and the `MPI_Win_start` and `MPI_Win_complete` synchronization functions in the master process.

MPI7Win22*. An array A of K real numbers (where K is the number of slave processes) and an array N of 8 integers are given in the master process. All the elements of the array N are in the range 1 to K ; some elements of this array may have the same value. In addition, an array B of R real numbers is given in the slave process of rank R ($R = 1, \dots, K$). Create an access window containing the array B in each slave process. Using the required number of calls of the `MPI_Accumulate` function in the master process, add all the elements of the array A to the corresponding elements of the array B from the process of rank N_I , $I = 0, \dots, 7$ (that is, the element A_0 should be added to the element B_0 , the element A_1 should be added to the element B_1 , and so on). Elements of the array A can be added several times to some arrays B . Output the array B (which may be changed or not) in each slave process. Use the `MPI_Win_post` and `MPI_Win_wait` synchronization functions in the slave processes and the `MPI_Win_start` and `MPI_Win_complete` synchronization functions in the master process.

MPI7Win23*°. An array A of 5 real numbers is given in each process. In addition, two arrays N and M of 5 integers are given in the master process. All the elements of the array N are in the range 1 to K , where K is the number of slave processes, all the elements of the array M are in the range 0 to 4. Some elements of both the array N and the array M may have the same value. Create an access window containing the array A in each slave process. Using the required number of calls of the `MPI_Get` function in the master process, receive the element of A with the index M_I from the process N_I ($I = 0, \dots, 4$) and add the received element to the element A_I in the master process. After changing the array A in the master process, change all the arrays A in the slave processes as follows: if some element of the array A from the slave process is greater than the element, with the same index, of the array A from the master process, then replace this element in the slave process by the corresponding element from the master process (to do this, use the required number of calls of the `MPI_Accumulate` function in the master process). Output the changed arrays A in each process. Use two calls of the `MPI_Win_post` and `MPI_Win_wait` synchronization functions in the slave processes and two calls of the `MPI_Win_start` and `MPI_Win_complete` synchronization functions in the master process.

Remark. The solution of this task is given in Section 1.3.6.

MPI7Win24. An integer N is given in each slave process, all the integers N are distinct and are in the range 0 to $K - 1$, where K is the number of slave processes. Create an access window containing an array A of K integers in each slave process. Without performing any synchronization function calls

in the master process (except calling the `MPI_Barrier` function) and using a sequence of calls of the `MPI_Win_lock`, `MPI_Win_unlock`, `MPI_Barrier`, `MPI_Win_lock`, `MPI_Win_unlock` synchronization functions in the slave processes, change element of the array A with index N by assigning the rank of the slave process, which contains the integer N , to this element (to do this, use the `MPI_Put` function) and then receive and output all the elements of the changed array A in each slave process (to do this, use the `MPI_Get` function). Use the `MPI_LOCK_SHARED` constant as the first parameter of the `MPI_Win_lock` function.

Note. The `MPI_Win_lock` and `MPI_Win_unlock` synchronization functions are used mainly for one-way communications with *passive targets*. In such a kind of one-way communications, the target process does not process the data transferred to it but acts as their storage, which is accessible to other processes.

MPI7Win25. The number of processes K is a multiple of 3. An array A of 5 real numbers is given in the processes of rank $3N$ ($N = 0, \dots, K/3 - 1$), an integer M and a real number B are given in the processes of rank $3N + 1$. The given integers M are in the range 0 to 4. Create an access window containing the array A in all processes in which this array is given. Using the `MPI_Accumulate` function call in the processes of rank $3N + 1$ ($N = 0, \dots, K/3 - 1$), change the array A from the process $3N$ as follows: if the array element with the index M is greater than the number B , then this element should be replaced by the number B (the numbers M and B are taken from the process $3N + 1$). Then send the changed array A from the process $3N$ to the process $3N + 2$ and output the received array in the process $3N + 2$; to do this, use the `MPI_Get` function call in the process of rank $3N + 2$. Use the `MPI_Win_lock`, `MPI_Win_unlock`, `MPI_Barrier` synchronization functions in the processes of rank $3N + 1$, the `MPI_Barrier`, `MPI_Win_lock`, `MPI_Win_unlock` synchronization functions in the processes of rank $3N + 2$, and the `MPI_Barrier` function in the processes of rank $3N$. Use the `MPI_LOCK_EXCLUSIVE` constant as the first parameter of the `MPI_Win_lock` function.

MPI7Win26. An array A of 5 positive real numbers is given in each slave process. Create an access window containing an array B of 5 zero-valued real numbers in the master process. Without performing any synchronization function calls in the master process (except calling the `MPI_Barrier` function) and using a sequence of calls of the `MPI_Win_lock`, `MPI_Win_unlock`, `MPI_Barrier`, `MPI_Win_lock`, `MPI_Win_unlock` synchronization functions in the slave processes, change elements of the array B by assigning the maximal value of the array A elements with the index I ($I = 0, \dots, 4$) to the array B element with the same index (to do this, use the `MPI_Accumulate` function) and then receive and output all the elements of the changed array B in each slave process (to do this, use the `MPI_Get` function). Use the

MPI_LOCK_SHARED constant as the first parameter of the MPI_Win_lock function.

MPI7Win27*. Two real numbers X , Y (the coordinates of a some point on a plane) are given in each slave process. Using the MPI_Get function in the master process, receive real numbers X_0 , Y_0 in this process that are equal to the coordinates of the point that is the most remote from the origin among all the points given in the slave processes. Then send the numbers X_0 , Y_0 from the master process to all the slave processes and output these numbers in the slave processes; to do this, use the MPI_Get function call in the slave processes. Use the MPI_Win_lock, MPI_Win_unlock, MPI_Barrier synchronization functions in the master process and the MPI_Barrier, MPI_Win_lock, MPI_Win_unlock synchronization functions in the slave processes.

Note. This task cannot be solved by using one-way communications only on the side of the slave processes by means of the lock/unlock synchronizations.

MPI7Win28*. Solve the MPI7Win27 task using the single access window containing the numbers X_0 , Y_0 in the master process. Use the MPI_Get and MPI_Put functions in the slave processes to find the numbers X_0 , Y_0 (for some processes, the MPI_Put function is not required), use the MPI_Get function to send the numbers X_0 , Y_0 to all the slave processes (as in the MPI7Win27 task). To synchronize exchanges when find the numbers X_0 , Y_0 , use two calls of each of the MPI_Win_start and MPI_Win_complete functions in the slave processes and calls of the MPI_Win_post and MPI_Win_wait functions *in a loop* in the master process (it is necessary to define *a new group of processes* at each iteration of the loop; this group should be used in the MPI_Win_post function call). To synchronize sending numbers X_0 , Y_0 to slave processes, use the MPI_Barrier function in the master process and the MPI_Barrier, MPI_Win_Lock, MPI_Win_unlock functions in the slave processes (as in the MPI7Win27 task).

Note. The solution method described in this task allows one-way communications to be used only on the side of the slave processes (in contrast to the method described in the MPI7Win27 task) but it requires to apply a synchronizations that different from the lock/unlock ones.

MPI7Win29*. One row of the square integer-valued matrix of order K is given in each process, where K is the number of processes (the process of rank R contains the R th row of the matrix, the rows are numbered from 0). Using the MPI_Get function calls in the master process, receive a matrix row with the minimal sum S of elements in this process and also find the number N of matrix rows with this minimal sum (if $N > 1$, then the last of such rows, that is, the row with the maximal ordinal number, should be saved in the master process). Then send this matrix row, the sum S , and the number N to

each slave process using the `MPI_Get` function in these processes. Output all received data in each process. To do this, create an access window containing $K + 2$ integers in each process; the first K elements of the window should contain the elements of the matrix row, the next element should contain the sum S of its elements, and the last element should contain the number N . Use the `MPI_Win_lock`, `MPI_Win_unlock`, `MPI_Barrier` synchronization functions in the master process and the `MPI_Barrier`, `MPI_Win_lock`, `MPI_Win_unlock` synchronization functions in the slave processes.

Note. This task cannot be solved by using one-way communications only on the side of the slave processes by means of the lock/unlock synchronizations.

MPI7Win30*. Solve the `MPI7Win29` task using the single access window containing the matrix row and the numbers S and N in the master process. Use the `MPI_Get` and `MPI_Put` functions in the slave processes to find the matrix row with the minimal sum and the related numbers S and N (for some processes, the `MPI_Put` function is not required), use the `MPI_Get` function to send the row with the minimal sum and the numbers S and N to all the slave processes (as in the `MPI7Win29` task). To synchronize exchanges when find the matrix row, use two calls of each of the `MPI_Win_start` and `MPI_Win_complete` functions in the slave processes and calls of the `MPI_Win_post` and `MPI_Win_wait` functions *in a loop* in the master process (it is necessary to define *a new group of processes* at each iteration of the loop; this group should be used in the `MPI_Win_post` function call). To synchronize sending the row with the minimal sum and the numbers S and N to slave processes, use the `MPI_Barrier` function in the master process and the `MPI_Barrier`, `MPI_Win_Lock`, `MPI_Win_unlock` functions in the slave processes (as in the `MPI7Win29` task).

Note. The solution method described in this task allows one-way communications to be used only on the side of the slave processes (in contrast to the method described in the `MPI7Win29` task) but it requires to apply a synchronizations that different from the lock/unlock ones.

2.8. Inter-communicators and process creation

The basic tools for creation of inter-communicators and their use for point-to-point communication are defined in the MPI-1 standard. Therefore, 5 tasks of this group (`MPI8Inter1`–`MPI8Inter4` and `MPI8Inter9`) can be solved using the MPICH 1.2.5 system. Other tasks are devoted to the new functions for inter-communicator creation (`MPI8Inter5`–`MPI8Inter8`), to the collective communications via inter-communicators (`MPI8Inter10`–`MPI8Inter14`) and to use inter-communicators for process creation (`MPI8Inter15`–`MPI8Inter22`). All these features have appeared in the MPI-2 standard, so you should use the MPICH2 1.3 or MS-MPI 10 system to solve these tasks.

You should use a *copy* of the communicator `MPI_COMM_WORLD` as the *peer* communicator (the third parameter of the `MPI_Intercomm_create` function). Use the `MPI_Comm_dup` function to create this copy.

The parameters of the `MPI_Comm_spawn` function used for process creation in the `MPI8Inter15`–`MPI8Inter22` tasks should be as follows: the first parameter should be the name of the executable file `ptprj.exe`, the second parameter `argv` is enough to specify the `NULL` constant, the fourth parameter `info` should be the `MPI_NULL_INFO` constant, the last parameter `array_of_errcodes` should be the `MPI_ERRCODES_IGNORE` constant. If the task does not specify the source communicator for the process creation, then it is assumed that this communicator should be the `MPI_COMM_WORLD` one.

Instead of the string `"ptprj.exe"`, you can use the function `char *GetExename()` that is implemented in the Programming Taskbook and returns the full name of the executable file.

Before solving the tasks in this section, you should study the `MPI8Inter9` and `MPI8Inter15` task solutions given in Sections 1.3.7–1.3.8.

2.8.1. Inter-communicator creation

MPI8Inter1. The number of processes K is an even number. An integer X is given in each process. Using the `MPI_Comm_group`, `MPI_Group_range_incl`, and `MPI_Comm_create` functions, create two communicators: the first one contains the even-rank processes in the same order $(0, 2, \dots, K/2 - 2)$, the second one contains the odd-rank processes in the same order $(1, 3, \dots, K/2 - 1)$. Output the ranks R of the processes included in these communicators. Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. Using the `MPI_Send` and `MPI_Recv` functions for this inter-communicator, send the integer X from each process to the process with the same rank from the other group of the inter-communicator and output the received integers.

MPI8Inter2. The number of processes K is an even number. An integer C and a real number X are given in each process. The numbers C are equal to 0 or 1, the amount of integers 1 is equal to the amount of integers 0. The integer C is equal to 0 in the process of rank 0 and is equal to 1 in the process of rank $K - 1$. Using one call of the `MPI_Comm_split` function, create two communicators: the first one contains processes with $C = 0$ (in the same order) and the second one contains processes with $C = 1$ (in the inverse order). Output the ranks R of the processes included in these communicators (note that the first and the last processes of the `MPI_COMM_WORLD` communicator will receive the value $R = 0$). Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. Using the `MPI_Send` and `MPI_Recv` functions for this inter-communicator, send the real

number X from each process to the process with the same rank from the other group of the inter-communicator and output the received numbers.

MPI8Inter3*. The number of processes K is a multiple of 3. A real number X is given in the processes of rank $3N$ ($N = 0, \dots, 3K - 3$), real numbers X and Y are given in the processes of rank $3N + 1$, a real number Y is given in the processes of rank $3N + 2$. Using the `MPI_Comm_group`, `MPI_Group_range_incl`, and `MPI_Comm_create` functions, create three communicators: the first one contains processes of rank $3N$ in the same order ($0, 3, \dots, K - 3$), the second one contains processes of rank $3N + 1$ in the inverse order ($K - 2, K - 5, \dots, 1$), the third one contains processes of rank $3N + 2$ in the same order ($2, 5, \dots, K - 1$). Output the ranks R of the processes included in these communicators. Then combine these communicators into two inter-communicators using the `MPI_Intercomm_create` function. The first inter-communicator contains the first and second group of processes, the second one contains the second and third group of processes. Using the `MPI_Send` and `MPI_Recv` functions for these inter-communicators, exchange the numbers X in the processes with the same rank in the first and second group and the numbers Y in the processes with the same rank in the second and third group. Output the received number in each process.

Note. The `MPI_Intercomm_create` function should be called once for processes of the first and third groups, and twice for processes of the second group, and this number of calls should be performed for the `MPI_Send` and `MPI_Recv` functions.

MPI8Inter4*. The number of processes K is a multiple of 3. Three integers are given in each process. The first integer (named C) is in the range 0 to 2, the amount of each value 0, 1, 2 is equal to $K/3$, processes 0, 1, 2 contain C with the value 0, 1, 2 respectively. Using one call of the `MPI_Comm_split` function, create three communicators: the first one contains processes with $C = 0$ (in the same order), the second one contains processes with $C = 1$ (in the same order), the third one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (note that the processes 0, 1, 2 of the `MPI_COMM_WORLD` communicator will receive the value $R = 0$). Then combine these communicators into three inter-communicators using two calls of the `MPI_Intercomm_create` function in each process. The first inter-communicator contains groups of processes with C equal to 0 and 1, the second one contains groups of processes with C equal to 1 and 2, the third one contains groups of processes with C equal to 0 and 2 (thus, the created inter-communicators will form a *ring* connecting all three previously created groups). Denoting two next given integers in the first group as X and Y , in the second group as Y and Z , and in the third group as Z and X (in this order) and using two calls of the `MPI_Send` and `MPI_Recv` functions for these inter-communicators, exchange the num-

bers X in the processes with the same rank in the first and second group, the numbers Y in the processes with the same rank in the second and third group, and the numbers Z in the processes with the same rank in the first and third group. Output the received numbers in each process.

MPI8Inter5*. The number of processes K is a multiple of 4. An integer X is given in each process. Using the `MPI_Comm_group`, `MPI_Group_range_incl` and `MPI_Comm_create` function, create two communicators: the first one contains the first half of the processes (of rank 0, 1, ..., $K/2 - 1$ in this order), the second one contains the second half of the processes (of rank $K/2$, $K/2 + 1$, ..., $K - 1$ in this order). Output the ranks R_1 of the processes included in these communicators. Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. Using the `MPI_Comm_create` function for this inter-communicator, create a new inter-communicator whose the first group contains the even-rank processes of the first group of the initial inter-communicator (in the same order) and the second group contains the odd-rank processes of the second group of the initial inter-communicator (in the inverse order). Thus, the first and second groups of the new inter-communicator will include the processes of the `MPI_COMM_WORLD` communicator with ranks 0, 2, ..., $K/2 - 2$ and $K - 1$, $K - 3$, ..., $K/2 + 1$ respectively. Output the ranks R_2 of the processes included in the new inter-communicator. Using the `MPI_Send` and `MPI_Recv` functions for the new inter-communicator, send the integer X from each process to the process with the same rank from the other group of the inter-communicator and output the received numbers.

MPI8Inter6*. The number of processes K is a multiple of 4. A real number X is given in each process. Using the `MPI_Comm_group`, `MPI_Group_range_incl` and `MPI_Comm_create` function, create two communicators: the first one contains the first half of the processes (of rank 0, 1, ..., $K/2 - 1$ in this order), the second one contains the second half of the processes (of rank $K/2$, $K/2 + 1$, ..., $K - 1$ in this order). Output the ranks R_1 of the processes included in these communicators. Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. Using one call of the `MPI_Comm_split` function for this inter-communicator, create two new inter-communicators: the first one contains the even-rank processes of the initial inter-communicator, the second one contains the odd-rank processes of the initial inter-communicator; the processes of the second group of each new inter-communicator should be in the inverse order. Thus, the first new communicator will include groups of the processes of the `MPI_COMM_WORLD` communicator with ranks 0, 2, ..., $K/2 - 2$ and $K - 2$, $K - 4$, ..., $K/2$, the first new communicator will include groups of the processes of the `MPI_COMM_WORLD` communicator with ranks 1, 3, ..., $K/2 - 1$ and $K - 1$, $K - 3$, ..., $K/2 + 1$. Output the ranks R_2 of the processes

included in the new inter-communicators. Using the `MPI_Send` and `MPI_Recv` functions for the new inter-communicators, send the integer X from each process to the process with the same rank from the other group of this inter-communicator and output the received numbers.

MPI8Inter7.** The number of processes K is an even number. An integer C is given in each process. The numbers C are equal to 0 or 1. A single value of $C = 1$ is given in the first half of the processes, the number of values of $C = 1$ is greater than one in the second half of the processes and, in addition, there is at least one value $C = 0$ in the second half of the processes. Using the `MPI_Comm_split` function, create two communicators: the first one contains the first half of the processes (of rank 0, 1, ..., $K/2 - 1$ in this order), the second one contains the second half of the processes (of rank $K/2$, $K/2 + 1$, ..., $K - 1$ in this order). Output the ranks R_1 of the processes included in these communicators. Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. Using the `MPI_Comm_split` function for this inter-communicator, create a new inter-communicator with groups which contain processes from the corresponding groups of the initial inter-communicator with the values $C = 1$ (in the inverse order). Thus, the first group of the new inter-communicator will include a single process, and the number of processes in the second group will be in the range 2 to $K/2 - 1$. Output the ranks R_2 of the processes that are included in the second group of the new inter-communicator (this group contains more than one process). Input an array Y of K_2 integers in the single process of the first group of the new inter-communicator, where K_2 is the number of the processes in the second group. Input an integer X in each process of the second group of the new inter-communicator. Using the required number of calls of the `MPI_Send` and `MPI_Recv` functions for all the processes of the new inter-communicator, send all the integers X to the single process of the first group and send the element of the array Y with the index R_2 to the process R_2 of the second group ($R_2 = 0, 1, \dots, K_2 - 1$). Output all received numbers (the integers X should be output in ascending order of ranks of sending processes).

Note. In the MPICH 2 version 1.3, the `MPI_Comm_split` function call for some inter-communicator is erroneous if some of the values of its color parameter are equal to `MPI_UNDEFINED`. Thus, you should use only *non-negative* values of color in this situation. In addition, the program can behave incorrectly if the `MPI_Comm_split` function create *empty groups* for some inter-communicators (this is possible if the same color values are specified for all processes of one of the groups of the initial inter-communicator and these color values are different from color values for some processes of the other group).

MPI8Inter8.** An integer C is given in each process. The integer C is in the range 0 to 2, all the values of C (0, 1, 2) are given for the even-rank processes and for the odd-rank processes. Using one call of the `MPI_Comm_split` function, create two communicators: the first one contains the even-rank processes (in ascending order of ranks), the second one contains the odd-rank processes (in ascending order of ranks). Output the ranks R_1 of the processes included in these communicators. Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. Using one call of the `MPI_Comm_split` function for this inter-communicator, create three new inter-communicators with groups which contain processes from the corresponding groups of the initial inter-communicator with the same values of C (in the same order). Thus, for instance, the first group of the first new inter-communicator will include the even-rank processes with $C = 0$ and the second group of the third new inter-communicator will include the odd-rank processes with $C = 2$. Output the ranks R_2 of the processes included in the new inter-communicators. Input an integer X in the processes of the first group of each new inter-communicator, input an integer Y in the processes of the second group of each new inter-communicator. Using the required number of calls of the `MPI_Send` and `MPI_Recv` functions for all the processes of all the new inter-communicators, send all the integers X to each process of the second group of the same inter-communicator and send all the integers Y to each process of the first group of the same inter-communicator. Output all received numbers in ascending order of ranks of sending processes.

MPI8Inter9°.** The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the `MPI_Comm_split` function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer -1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. Input an integer X in the processes of the first group, input an integer Y in the processes of the second group. Using the required number of calls of the `MPI_Send` and `MPI_Recv` functions for all the processes of the inter-communicator, send all the integers X to each process of the second group and send all the integers Y to each process of the first group. Output all received numbers in ascending order of ranks of sending processes.

Remark. The solution of this task is given in Section 1.3.7.

2.8.2. Collective communications for inter-communicators

MPI8Inter10*. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the `MPI_Comm_split` function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer -1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. Input integers R_1 and R_2 in each process of the inter-communicator. The values of the numbers R_1 coincide in all processes and indicate the rank of the selected process of the first group; the values of the numbers R_2 also coincide in all processes and indicate the rank of the selected process of the second group. A sequence of three integers X is given in the selected process of the first group, a sequence of three integers Y is given in the selected process of the second group. Using two calls of the `MPI_Bcast` collective function in each process of the inter-communicator, send the numbers X to all the processes of the second group, send the numbers Y to all the processes of the first group, and output the received numbers.

MPI8Inter11*. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the `MPI_Comm_split` function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer -1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. Input an integer R_1 in each process of the inter-communicator. The values of the number R_1 coincide in all processes and indicate the rank of the selected process of the first group. An array X of K_2 integers is given in the selected process of the first group, where K_2 is the number of processes in the second group. Using one call of the `MPI_Scatter` collective function in each

process of the inter-communicator, send the element $X[R_2]$ to the process R_2 of the second group ($R_2 = 0, \dots, K_2 - 1$) and output the received numbers.

MPI8Inter12*. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the `MPI_Comm_split` function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer -1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. Input an integer R_2 in each process of the inter-communicator. The values of the number R_2 coincide in all processes and indicate the rank of the selected process of the second group. An integer X is given in all the processes of the first group. Using one call of the `MPI_Gather` collective function in each process of the inter-communicator, send all the integers X to the selected process of the second group. Output the received numbers in this process in ascending order of ranks of sending processes.

MPI8Inter13*. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K/2$. Using the `MPI_Comm_split` function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the same order). Output the ranks R of the processes included in these communicators (output the integer -1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. An integer X is given in each process of the first group, an integer Y is given in each process of the second group. Using one call of the `MPI_Allreduce` collective function in each process of the inter-communicator, receive the number Y_{min} in each process of the first group and the number X_{max} in each process of the second group, where the number Y_{min} is the minimal value of the given integers Y and the number X_{max} is the maximal value of the given integers X . Output the received numbers.

MPI8Inter14*. The number of processes K is an even number. An integer C is given in each process. The integer C is in the range 0 to 2, the first value $C = 1$ is given in the process 0, the first value $C = 2$ is given in the process $K - 1$. Using the `MPI_Comm_split` function, create two communicators: the first one contains processes with $C = 1$ (in the same order), the second one contains processes with $C = 2$ (in the inverse order). Output the ranks R of the processes included in these communicators (output the integer -1 if the process is not included into the created communicators). Then combine these communicators into an inter-communicator using the `MPI_Intercomm_create` function. A group containing processes with $C = 1$ is considered to be the first group of the created inter-communicator and the group of processes with $C = 2$ is considered to be its second group. An array X of K_2 integers is given in each process of the first group, where K_2 is the number of processes in the second group; an array Y of K_1 integers is given in each process of the second group, where K_1 is the number of processes in the first group. Using one call of the `MPI_Alltoall` collective function in each process of the inter-communicator, send the element $Y[R_1]$ of each array Y to the process R_1 of the first group ($R_1 = 0, \dots, K_1 - 1$) and send the element $X[R_2]$ of each array X to the process R_2 of the second group ($R_2 = 0, \dots, K_2 - 1$). Output the received numbers in ascending order of ranks of sending processes.

2.8.3. Process creation

MPI8Inter15°. A real number is given in each process. Using the `MPI_Comm_spawn` function with the first parameter "ptprj.exe", create one new process. Using the `MPI_Reduce` collective function, send the sum of the given numbers to the new process. Output the received sum in the debug section using the `Show` function in the new process. Then, using the `MPI_Bcast` collective function, send this sum to the initial processes and output it in each process.

Remark. The solution of this task is given in Section 1.3.8.

MPI8Inter16. An array A of K real numbers is given in each process, where K is the number of processes. Using one call of the `MPI_Comm_spawn` function with the first parameter "ptprj.exe", create K new processes. Using the `MPI_Reduce_scatter_block` collective function, send the maximal value of the elements $A[R]$ of the given arrays to the new process of rank R ($R = 0, \dots, K - 1$). Output the received maximal value in the debug section using the `Show` function in each new process. Then, using the `MPI_Send` and `MPI_Recv` functions, send the maximal value from the new process of rank R ($R = 0, \dots, K - 1$) to the initial process of the same rank and output the received numbers in the initial processes.

MPI8Inter17*. The number of processes K is an even number. Arrays of $K/2$ real numbers are given in the processes of rank 0 and 1. Using one call of the `MPI_Comm_spawn` function with the first parameter "ptprj.exe", create two new processes. Using one call of the `MPI_Comm_split` function for the inter-communicator connected with the new processes, create two new inter-communicators: the first one contains the group of even-rank initial processes (0, ..., $K - 2$) and the new process of rank 0 as the second group, the second one contains the group of odd-rank initial processes (1, ..., $K - 1$) and the new process of rank 1 as the second group. Using the `MPI_Send` function in the initial processes and the `MPI_Recv` function in the new processes, send all the given numbers from the first process of the first group of each inter-communicator to the single process of its second group. Output the received numbers in the debug section using the `Show` function in the new processes. Then, using the `MPI_Scatter` collective function for inter-communicators, send one number from the new process to each process of the first group of the corresponding inter-communicator (in ascending order of ranks of receiving processes) and output the received numbers.

MPI8Inter18*. The number of processes K is an even number. Arrays A of $K/2$ real numbers are given in each process. Using one call of the `MPI_Comm_spawn` function with the first parameter "ptprj.exe", create K new processes. Using one call of the `MPI_Comm_split` function for the inter-communicator connected with the new processes, create two new inter-communicators: the first one contains the group of even-rank initial processes (0, ..., $K - 2$) and the even-rank new processes as the second group, the second one contains the group of odd-rank initial processes (1, ..., $K - 1$) and the odd-rank new processes as the second group. Perform the following actions for each created inter-communicator: (1) find the minimal value (for the first inter-communicator) or the maximal value (for the second one) of the elements $A[R]$ ($R = 0, \dots, K/2 - 1$) of all the arrays A given in the first group of this inter-communicator; (2) send the found value to the new process of rank R in the second group of the corresponding inter-communicator. For instance, the minimal of the first elements of the arrays given in the even-rank initial processes should be sent to the first of the new processes, the maximal of the first elements of the arrays given in the odd-rank initial processes should be sent to the second of the new processes (since this process has rank 0 in the corresponding inter-communicator). To do this, use the `MPI_Reduce_scatter_block` collective function. Output the received values in the debug section using the `Show` function in each new process. Then, using the `MPI_Reduce` collective function, find the minimum of the values received in the second group of the first inter-communicator, send the found minimum to the first process of the first group of this inter-communicator (that is, to the process 0 in the

MPI_COMM_WORLD communicator), and output the received minimum. Also, find the maximum of the values received in the second group of the second inter-communicator, send the found maximum to the first process of the first group of this inter-communicator (that is, to the process 1 in the MPI_COMM_WORLD communicator), and output the received maximum.

MPI8Inter19*. An array A of $2K$ integers is given in the master process, where K is the number of processes. Using one call of the `MPI_Comm_spawn` function with the first parameter "ptprj.exe", create K new processes. Using the `MPI_Intercomm_merge` function for the inter-communicator connected with the new processes, create a new intra-communicator which include both the initial and the new processes. The order of the processes in the new intra-communicator should be as follows: the initial processes, then the new ones (to specify this order, use the appropriate value of the parameter `high` of the `MPI_Intercomm_merge` function). Using the `MPI_Scatter` collective function for the new intra-communicator, send the element $A[R]$ of the array A to the process of rank R in this intra-communicator ($R = 0, \dots, 2K - 1$). Output the numbers received in the initial processes in the section of results, output the numbers received in the new processes in the debug section using the `Show` function. Then, using the `MPI_Reduce` collective function in this intra-communicator, find and output the sum of all numbers in the process of rank 1 in this intra-communicator.

MPI8Inter20*. The number of processes K is *not* a multiple of 4. An integer A is given in each process. Using one call of the `MPI_Comm_spawn` function with the first parameter "ptprj.exe", create such a number of new processes (1, 2 or 3) that the total number of processes K_0 in the application would be a multiple of 4. Define an integer A equal to $-R - 1$ in each new process, where R is the process rank. Using the `MPI_Intercomm_merge` function for the inter-communicator connected with the new processes, create a new intra-communicator which include both initial and new processes. The order of the processes in the new intra-communicator should be as follows: the initial processes, then the new ones (to specify this order, use the appropriate value of the parameter `high` of the `MPI_Intercomm_merge` function). Using the `MPI_Cart_create` function for the new intra-communicator, define a Cartesian topology for all processes as a two-dimensional $(K_0/4 \times 4)$ grid, which is periodic in the second dimension (ranks of processes should not be reordered). Find the process coordinates in the created topology using the `MPI_Cart_coords` function. Output the coordinates found in the initial processes in the section of results, output the coordinates found in the new processes in the debug section with the "X =" and "Y =" comments using the `Show` function. Using the `MPI_Comm_shift` and `MPI_Sendrecv_replace` functions, perform a cyclic shift of the integers A given in all processes of each column of the grid by step -1 (that is, the number A should be sent from each

process in the column, with the exception of the first process, to the previous process in the same column and from the first process in the column to the last process in the same column). Output the integers A received in the initial processes in the section of results, output the integers A received in the new processes in the debug section with the "A =" comment using the Show function.

MPI8Inter21.** A real number is given in each process; this number is denoted by the letter A in the master process and by the letter B in the slave processes. Using two calls of the `MPI_Comm_spawn` function with the first parameter "ptprj.exe", create two groups of new processes as follows: the first group (named the *server* group) should include one process, the second group (named the *client* group) should include $K - 1$ processes, where K is the number of initial processes. Send the number A from the master process to the single new process of the server group, send the number B from each slave process to the corresponding new process of the client group (in ascending order of the process ranks). Output the number received in each new process in the debug section using the Show function. Using the `MPI_Open_port`, `MPI_Publish_name`, and `MPI_Comm_accept` functions on the server side and the `MPI_Lookup_name` and `MPI_Comm_connect` functions in the client side, establish a connection between two new groups of processes by means of a new inter-communicator. Using the `MPI_Send` and `MPI_Recv` functions for this inter-communicator, receive the number A in each process of the client group from the process of the server group. Found the sum of the received number A and the number B , which is received earlier from the initial slave process, and output the sum $A + B$ in the debug section using the Show function in each process of the client group. Send this sum to the corresponding initial slave process and output the received sum in this process (the sum found in the process of rank R of the client group should be sent to the initial process of rank $R + 1$).

Note. The `MPI_Lookup_name` function call in the client processes should be performed *after* the function `MPI_Publish_name` call in the server process. You can, for example, use the `MPI_Barrier` function for the initial processes and the server process: in the server process, the `MPI_Barrier` function should be called after the call of the `MPI_Publish_name` function, whereas in the initial processes, the `MPI_Barrier` function should be called before the call of the `MPI_Comm_spawn` function which create the client group.

MPI8Inter22.** An integer N is given in each process. The integer N can take three values: 0, 1 and K ($K > 1$). There is exactly one process with the value $N = 1$ and exactly K processes with the value $N = K$. In addition, an integer A is given in the processes with the non-zero integer N . Using the `MPI_Comm_split` function, split the initial communicator `MPI_COMM_WORLD` into two ones: the first new communicator should include the process with

$N = 1$, the second one should include the processes with $N = K$. Using one call of the `MPI_Comm_spawn` function with the first parameter "ptprj.exe" for each new communicator, create two groups of new processes. The number of processes in each new group must coincide with the number of processes in the corresponding communicator (that is, the first group, named the *server group*, should include one process and the second one, named the *client group*, should include K processes). Send the integer A from each initial process to the new process; the rank of the receiving process should coincide with the rank of the sending process in the new communicator. Output the received integers in the debug section using the `Show` function. Using the `MPI_Open_port`, `MPI_Publish_name`, and `MPI_Comm_accept` functions on the server side and the `MPI_Lookup_name` and `MPI_Comm_connect` functions in the client side, establish a connection between two new groups of processes by means of a new inter-communicator. Using the `MPI_Gather` collective function for this inter-communicator, send all the integers A from the processes of the client group to the single process of the server group and output the received numbers in the debug section using several calls of the `Show` function in the process of the server group. Then, using the `MPI_Send` and `MPI_Recv` functions, send all these numbers from the process of the server group to the initial process that has created the server group. Output the received numbers in this initial process.

Note. The `MPI_Lookup_name` function call in the client processes should be performed *after* the `MPI_Publish_name` function call in the server process. You can, for example, send the number A to the process of the server group using the `MPI_Ssend` function and call the `MPI_Barrier` function for the `MPI_COMM_WORLD` communicator after the call of the `MPI_Ssend` function (on the side of the receiving process, you should receive the number A only after the call of the `MPI_Publish_name` function). In the other processes of the `MPI_COMM_WORLD` communicator, you should call the `MPI_Barrier` function and then send the numbers A to the processes of the client group. Thus, any of the processes of the client group will receive the number A only when the process of the server group has already called the `MPI_Publish_name` function.

2.9. Parallel matrix algorithms

All numeric data in tasks are integers. Matrices should be input and output by rows. Files with the matrix elements also contain them in a row-major order.

The number of processes in tasks related to the band algorithms (`MPI9Matr2`–`MPI9Matr20`) does not exceed 5. The number of processes in tasks related to the block algorithms (`MPI9Matr21`–`MPI9Matr44`) does not exceed 16.

Use the `char[12]` array to store the file name, use the `MPI_Bcast` function with the `MPI_CHAR` datatype parameter to send the file name from the master process to the slave processes.

The program templates for each task already contain descriptions of integer variables for storing the numeric data mentioned in tasks (in particular, the matrix sizes), pointers to arrays for storing the matrices themselves, as well as variables of the `MPI_Datatype` and `MPI_Comm` type. These variables should be used in all the functions that you need to implement when solving tasks. All names of variables correspond to the notations used in the task formulations. For arrays associated with bands or blocks of matrices, the names `a`, `b`, `c`, `t` are used; for arrays associated with the initial matrices A , B , and their resulting product C , the names with the underline are used (namely, `a_`, `b_`, `c_`).

Tasks with the file input-output (`MPI9Matr8–MPI9Matr10`, `MPI9Matr18–MPI9Matr20`, `MPI9Matr29–MPI9Matr31`, `MPI9Matr42–MPI9Matr44`) require the use of the MPI-2 library. To solve the other tasks in this group, you can use any version of MPI.

Before solving the tasks in this section, you should study the `MPI9Matr1`, `MPI9Matr2`, `MPI9Matr24`, `MPI9Matr19` task solutions given in Sections 1.4.2–1.4.5.

2.9.1. Non-parallel matrix multiplication algorithm

MPI9Matr1°. Integers M , P , Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process. Find and output a $M \times Q$ matrix C that is the product of the matrices A and B .

The formula for calculating the elements of the matrix C under the assumption that the rows and columns of all matrices are numbered from 0 is as follows: $C_{I,J} = A_{I,0} \cdot B_{0,J} + A_{I,1} \cdot B_{1,J} + \dots + A_{I,P-1} \cdot B_{P-1,J}$, where $I = 0, \dots, M - 1$, $J = 0, \dots, Q - 1$.

To store the matrices A , B , C , use one-dimensional arrays of size $M \cdot P$, $P \cdot Q$, and $M \cdot Q$ placing elements of matrices in a row-major order (that is, the matrix element with indices I and J will be stored in the element of the corresponding array with the index $I \cdot N + J$, where N is the number of columns of the matrix). The slave processes are not used in this task.

Remark. The solution of this task is given in Section 1.4.2.

2.9.2. Band algorithm 1 (horizontal bands)

MPI9Matr2*°. Integers M , P , Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process. In the first variant of the band algorithm of matrix multiplication, each matrix multiplier is divided into K horizontal bands, where K is the number of processes (hereinafter bands are distributed by processes and used to calculate a part of the total matrix product in each process).

The band of the matrix A contains N_A rows, the band of the matrix B contains N_B rows. The numbers N_A and N_B are calculated as follows: $N_A = \text{ceil}(M/K)$, $N_B = \text{ceil}(P/K)$, where the operation "/" means the division of real numbers and the function `ceil` performs rounding up. If the matrix contains insufficient number of rows to fill the last band, then the zero-valued rows should be added to this band.

Add, if necessary, the zero-valued rows to the initial matrices, save them in one-dimensional arrays in the master process, and then send the matrix bands from these arrays to all processes as follows: a band with the index R is sent to the process of rank R ($R = 0, 1, \dots, K - 1$), all the bands A_R are of the size $N_A \times P$, all the bands B_R are of the size $N_B \times Q$. In addition, create a band C_R in each process to store the part of the matrix product $C = AB$ which will be calculated in this process. Each band C_R is of the size $N_A \times Q$ and is filled with zero-valued elements.

The bands, like the initial matrices, should be stored in one-dimensional arrays in a row-major order. To send the matrix sizes, use the `MPI_Bcast` collective function, to send the bands of the matrices A and B , use the `MPI_Scatter` collective function.

Include all the above mentioned actions in a `Matr1ScatterData` function (without parameters). As a result of the call of this function, each process will receive the values N_A , P , N_B , Q , as well as one-dimensional arrays filled with the corresponding bands of the matrices A , B , C . Output all obtained data (that is, the numbers N_A , P , N_B , Q and the bands of the matrices A , B , C) in each process after calling the `Matr1ScatterData` function. Perform the input of initial data in the `Matr1ScatterData` function, perform the output of the results in the `Solve` function.

Note. To reduce the number of the `MPI_Bcast` function calls, all matrix sizes may be sent as a single array.

Remark. The solution of this task is given in Section 1.4.3.

MPI9Matr3. Integers N_A , P , N_B , Q and one-dimensional arrays filled with the corresponding bands of matrices A , B , C are given in each process (thus, the given data coincide with the results obtained in the `MPI9Matr2` task). Implement the first step of the band algorithm of matrix multiplication as follows: multiply elements in the bands A_R and B_R of each process and perform the cyclic sending each band B_R to the process of the previous rank (that is, from the process 1 to the process 0, from the process 2 to the process 1, ..., from the process 0 to the process $K - 1$, where K is the number of processes).

Use the `MPI_Sendrecv_replace` function to send the bands. To determine the ranks of the sending and receiving processes, use the expression containing the `%` operator that gives the remainder of a division.

Include all the above mentioned actions in a `Matr1Calc` function (without parameters). Output the new contents of the bands C_R and B_R in each process; perform data input and output in the `Solve` function.

Note. As a result of multiplying the bands A_R and B_R , each element of the band C_R will contain a *part* of the terms included in the elements of the product AB ; all elements of the band B_R and some of the elements of the band A_R will be used (in particular, the first N_B elements of the band A_0 will be used in the process 0 in the first step and the last N_B elements of the band A_{K-1} will be used in the process $K - 1$ in the first step).

MPI9Matr4. Integers N_A , P , N_B , Q and one-dimensional arrays filled with the corresponding bands of matrices A , B , C are given in each process (thus, the given data coincide with the results obtained in the `MPI9Matr2` task). Modify the `Matr1Calc` function, which was implemented in the previous task; the modified function should provide execution of any step of the band algorithm of matrix multiplication.

To do this, add the parameter named `step` to the function (this parameter specifies the step number and may be in the range 0 to $K - 1$, where K is the number of processes) and use the value of this parameter in the part of the algorithm that deals with the recalculation of the elements of the band C_R (the cyclic sending of the bands B_R does not depend on the value of the parameter `step`).

Using two calls of the modified `Matr1Calc` function with the parameters 0 and 1, execute two initial steps of the band algorithm and output the new contents of the bands C_R and B_R in each process. Perform data input and output in the `Solve` function.

Note. The parameter `step` determines which part of the band A_R will be used for the next recalculation of the elements of the band C_R (note that these parts should be selected cyclically).

MPI9Matr5*. Integers N_A , P , N_B , Q and one-dimensional arrays filled with the corresponding bands of matrices A , B , C are given in each process (thus, the given data coincide with the results obtained in the `MPI9Matr2` task). In addition, a number L with the same value is given in each process. The value of L is in the range 3 to K , where K is the number of processes, and determines the number of steps of the band algorithm.

Using the function `Matr1Calc(I)` (see the previous task) in a loop with the parameter I ($I = 0, \dots, L - 1$), execute the initial L steps of the band algorithm and output the new contents of the bands C_R and B_R in each process. Perform data input and output in the `Solve` function.

Remark. If the value of L is equal to K , then the bands C_R will contain the corresponding parts of the final matrix product AB .

MPI9Matr6*. An integer M (the number of rows of the matrix product) is given in the master process. In addition, integers N_A , Q and one-dimensional arrays filled with the $N_A \times Q$ bands of matrix C are given in each process (the given bands of C are obtained as a result of K steps of the band algorithm — see the MPI9Matr5 task). Send all the bands C_R to the master process and output the received matrix C of the size $M \times Q$ in this process. To store the resulting matrix C in the master process, use a one-dimensional array sufficient to store the matrix of the size $(N_A \cdot K) \times Q$. To send data to this array, use the MPI_Gather collective function.

Include all the above mentioned actions in a Matr1GatherData function (without parameters). Perform the input of initial data in the Solve function, perform the output of the resulting matrix in the Matr1GatherData function.

MPI9Matr7**. Integers M , P , Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process (thus, the given data coincide with the given data in the MPI9Matr2 task).

Using successively the Matr1ScatterData, Matr1Calc (in a loop), and Matr1GatherData functions, that are developed in the MPI9Matr2–MPI9Matr6 tasks, find a matrix C , which is equal to the product of the initial matrices A and B , and output this matrix in the master process.

In addition, output the current contents of the band C_R in each process after each call of the Matr1Calc function. Modify the Matr1Calc function (see the MPI9Matr4 task), before using in this task, as follows: the bands B_R should not be sent when the parameter step is equal to $K - 1$.

MPI9Matr8*. Integers M , P , Q and two file names are given in the master process. The given files contain elements of a matrix A of the size $M \times P$ and a matrix B of the size $P \times Q$. Modify the initial stage of the band algorithm of matrix multiplication (see the MPI9Matr2 task) as follows: each process should read the corresponding bands of the matrices A and B directly from the given files using the MPI_File_seek and MPI_File_read_all collective functions (a new file view is not required).

To send the sizes of matrices and file names, use the MPI_Bcast collective function.

Include all these actions in a Matr1ScatterFile function (without parameters). As a result of the call of this function, each process will receive the values N_A , P , N_B , Q , as well as one-dimensional arrays filled with the corresponding bands of the matrices A , B , C . Output all obtained data (that is, the numbers N_A , P , N_B , Q and the bands of the matrices A , B , C) in each process after calling the Matr1ScatterFile function. Perform the input of initial data in the Matr1ScatterFile function, perform the output of the results in the Solve function.

Remark. For some bands, some of their elements (namely, the last rows) or even the entire bands should not be read from the source files and will remain zero-valued ones. However, this situation does not require special processing, since the `MPI_File_read_all` function automatically stops reading the data (without generating any error message) when the end of the file is reached.

MPI9Matr9*. Integers N_A , Q and one-dimensional arrays filled with the $N_A \times Q$ bands C_R are given in each process (the given bands C_R are obtained as a result of K steps of the band algorithm of matrix multiplication — see the `MPI9Matr5` task). In addition, an integer M (the number of rows of the matrix product) and the name of file (to store this product) are given in the master process.

Send the number M and the file name to all processes using the `MPI_Bcast` function. Write all the parts of the matrix product contained in the bands C_R to the resulting file, which will eventually contain a matrix C of the size $M \times Q$. To write the bands to the file, use the `MPI_File_seek` and `MPI_File_write_all` collective functions.

Include all these actions (namely, the input of file name, sending number M and the file name, and writing all bands to the file) in a `Matr1GatherFile` function. Perform the input of all initial data, except the file name, in the `Solve` function.

Note. When writing data to the resulting file, it is necessary to take into account that some of the bands C_R may contain trailing zero-valued rows that are not related to the resulting matrix product (the number M should be sent to all processes in order to control this situation).

MPI9Matr10.** Integers M , P , Q and three file names are given in the master process. The first two file names are related to the existing files containing the elements of matrices A and B of the size $M \times P$ and $P \times Q$, respectively, the third file should be created to store the resulting matrix product $C = AB$. Using successively the `Matr1ScatterFile`, `Matr1Calc` (in a loop), and `Matr1GatherFile` functions (see the `MPI9Matr8`, `MPI9Matr5`, `MPI9Matr9` tasks), find a matrix C and write its elements to the resulting file.

In addition, output the current value of the `c[step]` in each process after each call of the `Matr1Calc` function, where `c` is a one-dimensional array containing the band C_R , and `step` is the algorithm step number ($0, 1, \dots, K - 1$). Thus, the element `c[0]` should be output on the first step of the algorithm, the element `c[1]` should be output on the second step of the algorithm, and so on.

2.9.3. Band algorithm 2 (horizontal and vertical bands)

MPI9Matr11. Integers P and Q are given in each process; in addition, a matrix B of the size $P \times Q$ is given in the master process. The number Q is a multiple of the number of processes K . Input the matrix B into a one-

dimensional array of the size $P \cdot Q$ in the master process and define a new datatype named `MPI_BAND_B` that contains a vertical band of the matrix B . The width of the vertical band should be equal to $N_B = Q/K$ columns. When defining the `MPI_BAND_B` datatype, use the `MPI_Type_vector` and `MPI_Type_commit` functions.

Include this definition in a `Matr2CreateTypeBand(p, n, q, t)` function with the input integer parameters p , n , q and the output parameter t of the `MPI_Datatype` type; the parameters p and n determine the size of the vertical band (the number of its rows and columns), and the parameter q determines the number of columns of the matrix from which this band is extracted.

Using the `MPI_BAND_B` datatype, send to each process (inclusive of the master process) the corresponding band of the matrix B in the ascending order of ranks of receiving processes. Sending should be performed using the `MPI_Send` and `MPI_Recv` functions; the bands should be stored in one-dimensional arrays of the size $P \cdot N_B$. Output the received band in each process.

Remark. In the MPICH2 version 1.3, the `MPI_Send` function call is erroneous if the sending and receiving processes are the same. You may use the `MPI_Sendrecv` function to send a band to the master process. You may also fill a band in the master process without using tools of the MPI library.

MPI9Matr12*. Integers M , P , Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process. In the second variant of the band algorithm of matrix multiplication, the first multiplier (the matrix A) is divided into K horizontal bands and the second multiplier (the matrix B) is divided into K vertical bands, where K is the number of processes (hereinafter bands are distributed by processes and used to calculate a part of the total matrix product in each process).

The band of the matrix A contains N_A rows, the band of the matrix B contains N_B columns. The numbers N_A and N_B are calculated as follows: $N_A = \text{ceil}(M/K)$, $N_B = \text{ceil}(Q/K)$, where the operation $"/$ means the division of real numbers and the function `ceil` performs rounding up. If the matrix contains insufficient number of rows (or columns) to fill the last band, then the zero-valued rows (or columns) should be added to this band.

Add, if necessary, the zero-valued rows or columns to the initial matrices, save them in one-dimensional arrays in the master process, and then send the matrix bands from these arrays to all processes as follows: a band with the index R is sent to the process of rank R ($R = 0, 1, \dots, K - 1$), all the bands A_R are of the size $N_A \times P$, all the bands B_R are of the size $P \times N_B$. In addition, create a band C_R in each process to store the part of the matrix product $C = AB$ which will be calculated in this process. Each band C_R is of the size $(N_A \cdot K) \times N_B$ and is filled with zero-valued elements.

The bands, like the initial matrices, should be stored in one-dimensional arrays in a row-major order. To send the matrix sizes, use the `MPI_Bcast` collective function, to send the bands of the matrix A , use the `MPI_Scatter` collective function, to send the bands of the matrix B , use the `MPI_Send` and `MPI_Recv` functions and also the `MPI_BAND_B` datatype created by the `Matr2CreateTypeBand` function (see the previous task and a note to it).

Include all the above mentioned actions in a `Matr2ScatterData` function (without parameters). As a result of the call of this function, each process will receive the values N_A , P , N_B , as well as one-dimensional arrays filled with the corresponding bands of the matrices A , B , C . Output all obtained data (that is, the numbers N_A , P , N_B and the bands of the matrices A , B , C) in each process after calling the `Matr2ScatterData` function. Perform the input of initial data in the `Matr2ScatterData` function, perform the output of the results in the `Solve` function.

Notes. (1) When input the matrix B into an array in the master process, it should be taken into account that this array may contain elements corresponding to additional zero-valued columns.

(2) To reduce the number of the `MPI_Bcast` function calls, all matrix sizes may be sent as a single array.

MPI9Matr13. Integers N_A , P , N_B and one-dimensional arrays filled with the corresponding bands of matrices A , B , C are given in each process (thus, the given data coincide with the results obtained in the `MPI9Matr12` task). Implement the first step of the band algorithm of matrix multiplication as follows: multiply elements in the bands A_R and B_R of each process and perform the cyclic sending each band A_R to the process of the previous rank (that is, from the process 1 to the process 0, from the process 2 to the process 1, ..., from the process 0 to the process $K - 1$, where K is the number of processes).

Use the `MPI_Sendrecv_replace` function to send the bands. To determine the ranks of the sending and receiving processes, use the expression containing the `%` operator that gives the remainder of a division.

Include all the above mentioned actions in a `Matr2Calc` function (without parameters). Output the new contents of the bands C_R and A_R in each process; perform data input and output in the `Solve` function.

Note. In this variant of the band algorithm, the bands A_R contain the full rows of the matrix A and the bands B_R contain the full columns of the matrix B , so, as a result of their multiplication, the band C_R will contain part of the elements of the final matrix product already at the first step of the algorithm (the other elements of the band C_R will remain zero-valued). The location of the found elements in the band C_R depends on the rank of the process (in particular, the first N_A rows of the band C_0 in the process 0 will

be filled in the first step and the last N_A rows of the band C_{K-1} in the process $K - 1$ will be filled in the first step).

MPI9Matr14. Integers N_A , P , N_B and one-dimensional arrays filled with the corresponding bands of matrices A , B , C are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr12 task). Modify the `Matr2Calc` function, which was implemented in the previous task; the modified function should provide execution of any step of the band algorithm of matrix multiplication.

To do this, add the parameter named `step` to the function (this parameter specifies the step number and may be in the range 0 to $K - 1$, where K is the number of processes) and use the value of this parameter in the part of the algorithm that deals with the recalculation of the elements of the band C_R (the cyclic sending of the bands A_R does not depend on the value of the parameter `step`).

Using two calls of the modified `Matr2Calc` function with the parameters 0 and 1, execute two initial steps of the band algorithm and output the new contents of the bands C_R and A_R in each process. Perform data input and output in the `Solve` function.

Note. The parameter `step` determines which rows of the band C_R will be calculated in this step of the algorithm (note that these rows are selected cyclically).

MPI9Matr15*. Integers N_A , P , N_B and one-dimensional arrays filled with the corresponding bands of matrices A , B , C are given in each process (thus, the given data coincide with the results obtained in the MPI9Matr12 task). In addition, a number L with the same value is given in each process. The value of L is in the range 3 to K , where K is the number of processes, and determines the number of steps of the band algorithm.

Using the function `Matr2Calc(I)` (see the previous task) in a loop with the parameter I ($I = 0, \dots, L - 1$), execute the initial L steps of the band algorithm and output the new contents of the bands C_R and A_R in each process. Perform data input and output in the `Solve` function.

Remark. If the value of L is equal to K , then the bands C_R will contain the corresponding parts of the final matrix product AB .

MPI9Matr16*. Integers M and Q (the numbers of rows and columns of the matrix product) are given in the master process. In addition, integers N_A , N_B and one-dimensional arrays filled with the $(N_A \cdot K) \times N_B$ bands of the matrix C are given in each process (the given bands of C are obtained as a result of K steps of the band algorithm — see the MPI9Matr15 task). Send all the bands C_R to the master process and output the received matrix C of the size $M \times Q$ in this process.

To store the resulting matrix C in the master process, use a one-dimensional array sufficient to store the matrix of the size $(N_A \cdot K) \times (N_B \cdot K)$. To send data to this array, use the `MPI_Send` and `MPI_Recv` functions and the `MPI_BAND_C` datatype created by the `Matr2CreateTypeBand` function (see the `MPI9Matr11` task and a note to it).

Include all the above mentioned actions in a `Matr2GatherData` function (without parameters). Perform the input of initial data in the `Solve` function, perform the output of the resulting matrix in the `Matr2GatherData` function.

Note. When output the matrix C in the master process, it should be taken into account that an array, which is intended for matrix storage, may contain elements corresponding to additional zero-valued columns.

MPI9Matr17.** Integers M , P , Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process (thus, the given data coincide with the given data in the `MPI9Matr12` task).

Using successively the `Matr2ScatterData`, `Matr2Calc` (in a loop), and `Matr2GatherData` functions, that are developed in the `MPI9Matr12`–`MPI9Matr16` tasks, find a matrix C , which is equal to the product of the initial matrices A and B , and output this matrix in the master process.

In addition, output the current contents of the band C_R in each process after each call of the `Matr2Calc` function.

Modify the `Matr2Calc` function (see the `MPI9Matr14` task), before using in this task, as follows: the bands A_R should not be sent when the parameter step is equal to $K - 1$.

MPI9Matr18*. Integers M , P , Q and two file names are given in the master process. The given files contain elements of a matrix A of the size $M \times P$ and a matrix B of the size $P \times Q$. The number Q is a multiple of the number of processes K . Modify the initial stage of the band algorithm of matrix multiplication (see the `MPI9Matr12` task) as follows: each process should read the corresponding bands of the matrices A and B directly from the given files.

To send the sizes of matrices and file names, use the `MPI_Bcast` collective function. Use the `MPI_File_seek` and `MPI_File_read_all` collective functions to read the horizontal bands of the matrix A . To read the vertical bands of the matrix B , set the appropriate file view using the `MPI_File_set_view` function and the `MPI_BAND_B` filetype defined with the `Matr2CreateTypeBand` function (see the `MPI9Matr11` task), and then use the `MPI_File_read_all` function.

Include all these actions in a `Matr2ScatterFile` function (without parameters). As a result of the call of this function, each process will receive the values N_A , P , N_B , as well as one-dimensional arrays filled with the corresponding bands of the matrices A , B , C . Output all obtained data (that is, the numbers N_A , P , N_B and the bands of the matrices A , B , C) in each process after call-

ing the `Matr2ScatterFile` function. Perform the input of initial data in the `Matr2ScatterFile` function, perform the output of the results in the `Solve` function.

Note. A condition that the number Q is a multiple of K allows us to perform reading of the bands B_R using the same filetype in all processes.

If this condition is not fulfilled, then it would be necessary to use special types that ensure the correct reading from the file and write to the array of "truncated" bands of the matrix B in the last processes (in addition, in this case it would be necessary to send to each process the value of Q which is necessary for the correct type definition for "truncated" bands).

MPI9Matr19*°. Integers N_A , N_B and one-dimensional arrays filled with the $(N_A \cdot K) \times N_B$ bands C_R are given in each process (the given bands C_R are obtained as a result of K steps of the band algorithm of matrix multiplication — see the `MPI9Matr15` task). In addition, an integer M (the number of rows of the matrix product) and the name of file (to store this product) are given in the master process. The number of columns Q of the matrix product is a multiple of the number of processes K (and, therefore, is equal to $N_B \cdot K$).

Send the number M and the file name to all processes using the `MPI_Bcast` function. Write all the parts of the matrix product contained in the bands C_R to the resulting file, which will eventually contain a matrix C of the size $M \times Q$.

To write the bands to the file, set the appropriate file view using the `MPI_File_set_view` function and the `MPI_BAND_C` filetype defined with the `Matr2CreateTypeBand` function (see the `MPI9Matr11` task), and then use the `MPI_File_write_all` function.

Include all these actions (namely, the input of file name, sending number M and the file name, and writing all bands to the file) in a `Matr2GatherFile` function. Perform the input of all initial data, except the file name, in the `Solve` function.

Note. When writing data to the resulting file, it is necessary to take into account that the bands C_R may contain trailing zero-valued rows that are not related to the resulting matrix product (the number M should be sent to all processes in order to control this situation).

Remark. The solution of this task is given in Section 1.4.5.

MPI9Matr20.** Integers M , P , Q and three file names are given in the master process. The first two file names are related to the existing files containing the elements of matrices A and B of the size $M \times P$ and $P \times Q$, respectively, the third file should be created to store the resulting matrix product $C = AB$. The number Q is a multiple of the number of processes K . Using successively the `Matr2ScatterFile`, `Matr2Calc` (in a loop), and `Matr2GatherFile` functions

(see the MPI9Matr18, MPI9Matr15, MPI9Matr19 tasks), find a matrix C and write its elements to the resulting file.

In addition, output the current value of the $c[\text{step}]$ in each process after each call of the `Matr2Calc` function, where c is a one-dimensional array containing the band C_R , and step is the algorithm step number ($0, 1, \dots, K - 1$). Thus, the element $c[0]$ should be output on the first step of the algorithm, the element $c[1]$ should be output on the second step of the algorithm, and so on.

2.9.4. Cannon's block algorithm

MPI9Matr21. Integers M and P are given in each process; in addition, a matrix A of the size $M \times P$ is given in the master process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$, the numbers M and P are multiples of K_0 . Input the matrix A into a one-dimensional array of the size $M \cdot P$ in the master process and define a new datatype named `MPI_BLOCK_A` that contains a $M_0 \times P_0$ block of the matrix A , where $M_0 = M/K_0$, $P_0 = P/K_0$.

When defining the `MPI_BLOCK_A` type, use the `MPI_Type_vector` and `MPI_Type_commit` functions. Include this definition in a `Matr3CreateTypeBlock(m0, p0, p, t)` function with the input integer parameters m_0 , p_0 , p and the output parameter t of the `MPI_Datatype` type; the parameters m_0 and p_0 determine the size of the block, and the parameter p determines the number of columns of the matrix from which this block is extracted.

Using the `MPI_BLOCK_A` datatype, send to each process (in ascending order of ranks of processes, inclusive of the master process) the corresponding block of the matrix A in a row-major order of blocks (that is, the first block should be sent to the process 0, the next block in the same row of blocks should be sent to the process 1, and so on). Sending should be performed using the `MPI_Send` and `MPI_Recv` functions; the blocks should be stored in one-dimensional arrays of the size $M_0 \cdot P_0$. Output the received block in each process.

Remark. In the MPICH2 version 1.3, the `MPI_Send` function call is erroneous if the sending and receiving processes are the same. You may use the `MPI_Sendrecv` function to send a block to the master process. You may also fill a block in the master process without using tools of the MPI library.

MPI9Matr22. Integers M_0 , P_0 and a matrix A of the size $M_0 \times P_0$ are given in each process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$. Input the matrix A into a one-dimensional array of the size $M_0 \cdot P_0$ in each process and create a new communicator named `MPI_COMM_GRID` using the `MPI_Cart_create` function. The `MPI_COMM_GRID` communicator defines a Cartesian topology for all processes as a two-dimensional periodic $K_0 \times K_0$ grid (ranks of processes should not be reordered).

Include the creation of the `MPI_COMM_GRID` communicator in a `Matr3CreateCommGrid(comm)` function with the output parameter `comm` of the

MPI_Comm type. Using the MPI_Cart_coords function for this communicator, output the process coordinates (I_0, J_0) in each process.

Perform a cyclic shift of the matrices A given in all processes of each grid row I_0 by I_0 positions left (that is, in descending order of ranks of processes) using the MPI_Cart_shift and MPI_Sendrecv_replace functions. Output the received matrix in each process.

MPI9Matr23*. Integers M, P, Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$. In the block algorithms of matrix multiplication, the initial matrices are divided into K blocks and are interpreted as square block matrices of the order K_0 (hereinafter blocks are distributed by processes and used to calculate a part of the total matrix product in each process).

The block of the matrix A is of the size $M_0 \times P_0$, the block of the matrix B is of the size $P_0 \times Q_0$, the numbers M_0, P_0, Q_0 are calculated as follows: $M_0 = \text{ceil}(M/K_0)$, $P_0 = \text{ceil}(P/K_0)$, $Q_0 = \text{ceil}(Q/K_0)$, where the operation "/" means the division of real numbers and the function ceil performs rounding up. If the matrix contains insufficient number of rows (or columns) to fill the last blocks, then the zero-valued rows (or columns) should be added to these blocks.

Add, if necessary, the zero-valued rows or columns to the initial matrices (as a result, the matrices A and B will have the size $(M_0 \cdot K_0) \times (P_0 \cdot K_0)$ and $(P_0 \cdot K_0) \times (Q_0 \cdot K_0)$ respectively), save them in one-dimensional arrays in the master process, and then send the matrix blocks (in a row-major order) from these arrays to all processes (in ascending order of its ranks): the process R will receive the blocks A_R and B_R , $R = 0, \dots, K - 1$. In addition, create a block C_R in each process to store the part of the matrix product $C = AB$ which will be calculated in this process. Each block C_R is of the size $M_0 \times Q_0$ and is filled with zero-valued elements.

The blocks, like the initial matrices, should be stored in one-dimensional arrays in a row-major order. To send the matrix sizes, use the MPI_Bcast collective function, to send the blocks of the matrices A and B , use the MPI_Send and MPI_Recv functions and also the MPI_BLOCK_A and MPI_BLOCK_B datatypes created by the Matr3CreateTypeBlock function (see the MPI9Matr21 task and a note to it).

Include all the above mentioned actions in a Matr3ScatterData function (without parameters). As a result of the call of this function, each process will receive the values M_0, P_0, Q_0 , as well as one-dimensional arrays filled with the corresponding blocks of the matrices A, B, C . Output all obtained data (that is, the numbers M_0, P_0, Q_0 and the blocks of the matrices A, B, C) in each process after calling the Matr3ScatterData function. Perform the input of

initial data in the `Matr3ScatterData` function, perform the output of the results in the `Solve` function.

Notes. (1) When input the matrices A and B into arrays in the master process, it should be taken into account that these arrays may contain elements corresponding to additional zero-valued columns.

(2) To reduce the number of the `MPI_Bcast` function calls, all matrix sizes may be sent as a single array.

MPI9Matr24°. Integers M_0 , P_0 , Q_0 and one-dimensional arrays filled with the corresponding blocks of matrices A , B , C are given in each process (thus, the given data coincide with the results obtained in the `MPI9Matr23` task). Implement the initial block redistribution used in the Cannon's algorithm for block matrix multiplication.

To do this, define a Cartesian topology for all processes as a two-dimensional periodic $K_0 \times K_0$ grid, where $K_0 \cdot K_0$ is equal to the number of processes (ranks of processes should not be reordered), and perform a cyclic shift of the blocks A_R given in all processes of each grid row I_0 by I_0 positions left (that is, in descending order of ranks of processes), $I_0 = 0, \dots, K_0 - 1$, and perform a cyclic shift of the blocks B_R given in all processes of each grid column J_0 by J_0 positions up (that is, in descending order of ranks of processes), $J_0 = 0, \dots, K_0 - 1$.

To create the `MPI_COMM_GRID` communicator associated with the Cartesian topology, use the `Matr3CreateCommGrid` function implemented in the `MPI9Matr22` task. Use the `MPI_Cart_coords`, `MPI_Cart_shift`, `MPI_Sendrecv_replace` functions to perform the cyclic shifts (compare with `MPI9Matr22`).

Include all the above mentioned actions in a `Matr3Init` function (without parameters). Output the received blocks A_R and B_R in each process; perform data input and output in the `Solve` function.

Remark. The solution of this task is given in Section 1.4.4.

MPI9Matr25. Integers M_0 , P_0 , Q_0 and one-dimensional arrays filled with the corresponding blocks of matrices A , B , C are given in each process. The blocks C_R are zero-valued, the initial redistribution for the blocks A_R and B_R has already been performed in accordance with the Cannon's algorithm (see the previous task). Implement one step of the Cannon's algorithm of matrix multiplication as follows: multiply elements in the blocks A_R and B_R of each process and perform a cyclic shift of the blocks A_0 given in all processes of each row of the Cartesian periodic grid by 1 position left (that is, in descending order of ranks of processes) and perform a cyclic shift of the blocks B_0 given in all processes of each grid column by 1 position up (that is, in descending order of ranks of processes).

To create the `MPI_COMM_GRID` communicator associated with the Cartesian topology, use the `Matr3CreateCommGrid` function implemented in the `MPI9Matr22` task. Use the `MPI_Cart_shift` and `MPI_Sendrecv_replace` functions to perform the cyclic shifts (compare with `MPI9Matr22`).

Include all the above mentioned actions in a `Matr3Calc` function (without parameters). Output the new contents of the blocks C_R , A_R , and B_R in each process; perform data input and output in the `Solve` function.

Remark. A special feature of the Cannon's algorithm is that the actions at each step are not depend on the step number.

MPI9Matr26*. Integers M_0 , P_0 , Q_0 and one-dimensional arrays filled with the corresponding blocks of matrices A , B , C are given in each process. The blocks C_R are zero-valued, the initial redistribution for the blocks A_R and B_R has already been performed in accordance with the Cannon's algorithm (see the `MPI9Matr24` task). In addition, a number L with the same value is given in each process. The value of L is in the range 2 to K_0 , where $K_0 \cdot K_0$ is the number of processes, and determines the number of steps of the Cannon's algorithm.

Using the function `Matr3Calc` (see the previous task) in a loop, execute the initial L steps of the Cannon's algorithm and output the new contents of the blocks C_R , A_R , and B_R in each process. Perform data input and output in the `Solve` function.

Note. If the value of L is equal to K_0 , then the blocks C_R will contain the corresponding parts of the final matrix product AB .

MPI9Matr27*. Integers M and Q (the numbers of rows and columns of the matrix product) are given in the master process. In addition, integers M_0 , Q_0 and one-dimensional arrays filled with the $M_0 \times Q_0$ blocks of the matrix C are given in each process (the given blocks of C are obtained as a result of K_0 steps of the Cannon's algorithm — see the `MPI9Matr26` task). Send all the blocks C_R to the master process and output the received matrix C of the size $M \times Q$ in this process.

To store the resulting matrix C in the master process, use a one-dimensional array sufficient to store the matrix of the size $(M_0 \cdot K_0) \times (Q_0 \cdot K_0)$. To send the blocks C_R to this array, use the `MPI_Send` and `MPI_Recv` functions and the `MPI_BLOCK_C` datatype created by the `Matr3CreateTypeBlock` function (see the `MPI9Matr21` task and a note to it).

Include all the above mentioned actions in a `Matr3GatherData` function (without parameters). Perform the input of initial data in the `Solve` function, perform the output of the resulting matrix in the `Matr3GatherData` function.

Note. When output the matrix C in the master process, it should be taken into account that an array, which is intended for matrix storage, may contain elements corresponding to additional zero-valued columns.

MPI9Matr28.** Integers M , P , Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process (thus, the given data coincide with the given data in the MPI9Matr23 task).

Using successively the `Matr3ScatterData`, `Matr3Init`, `Matr3Calc` (in a loop), and `Matr3GatherData` functions, that are developed in the MPI9Matr23–MPI9Matr27 tasks, find a matrix C , which is equal to the product of the initial matrices A and B , and output this matrix in the master process.

In addition, output the current contents of the block C_R in each process after each call of the `Matr3Calc` function.

The `MPI_COMM_GRID` communicator, which is used in the `Matr3Init` and `Matr3Calc` functions, should not be created several times. To do this, modify the `Matr3CreateCommGrid` function; the modified function should not perform any actions when it is called with the parameter `comm` that is not equal to `MPI_COMM_NULL`.

In addition, modify the `Matr3Calc` function (see the MPI9Matr25 task), before using in this task, as follows: add the parameter named `step` to this function (`step = 0, \dots, K_0 - 1`); the blocks A_R and B_R should not be sent when the parameter `step` is equal to $K_0 - 1$.

MPI9Matr29*. Integers M , P , Q and two file names are given in the master process. The given files contain elements of a matrix A of the size $M \times P$ and a matrix B of the size $P \times Q$. Modify the stage of receiving blocks for the Cannon's algorithm of matrix multiplication (see the MPI9Matr23 task) as follows: each process should read the corresponding blocks of the matrices A and B directly from the given files. In this case, all processes should receive not only the sizes M_0 , P_0 , Q_0 of the blocks, but also the sizes M , P , Q of the initial matrices, which are needed to determine correctly the positions of blocks in the source files.

To send the sizes of matrices and file names, use the `MPI_Bcast` collective function. Use the `MPI_File_read_at` local function to read each row of the block (a new file view is not required).

Include all these actions in a `Matr3ScatterFile` function (without parameters). As a result of the call of this function, each process will receive the values M , P , Q , M_0 , P_0 , Q_0 , as well as one-dimensional arrays filled with the corresponding blocks of the matrices A , B , C . Output all obtained data (that is, the numbers M , P , Q , M_0 , P_0 , Q_0 and the blocks of the matrices A , B , C) in each process after calling the `Matr3ScatterFile` function. Perform the input of initial data in the `Matr3ScatterFile` function, perform the output of the results in the `Solve` function.

Note. For some blocks, some of their elements (namely, the last rows and/or columns) should not be read from the source files and will remain zero-valued ones. To determine the actual size of the block being read (the

number of rows and columns), it is required to use the sizes of the initial matrices and the coordinates (I_0, J_0) of the block in a square Cartesian grid of order K_0 (note that $I_0 = R/K_0$, $J_0 = R\%K_0$, where R is the process rank).

Remark. Whereas the values of P and Q are necessary to ensure the correct reading of the file blocks, the value of M is not required for this purpose, since the attempt to read data beyond the end of file is ignored (without generating any error message). However, the value of M is required at the final stage of the algorithm (see the next task), so it must also be sent to all processes.

MPI9Matr30*. Integers M , Q , M_0 , Q_0 and one-dimensional arrays filled with the $M_0 \times Q_0$ blocks C_R are given in each process (the given blocks C_R are obtained as a result of K_0 steps of the Cannon's block algorithm of matrix multiplication — see the MPI9Matr25 task). In addition, the name of file to store the matrix product is given in the master process.

Send the file name to all processes using the MPI_Bcast function. Write all the parts of the matrix product contained in the blocks C_R to the resulting file, which will eventually contain a matrix C of the size $M \times Q$.

Use the MPI_File_write_at local function to write each row of the block to the file (a new file view is not required).

Include all these actions (namely, the input of file name, sending the file name, and writing all blocks to the file) in a Matr3GatherFile function. Perform the input of all initial data, except the file name, in the Solve function.

Note. When writing data to the resulting file, it is necessary to take into account that some of the blocks C_R may contain trailing zero-valued rows and/or columns that are not related to the resulting matrix product (see also the note and the remark for the previous task).

MPI9Matr31**. Integers M , P , Q and three file names are given in the master process. The first two file names are related to the existing files containing the elements of matrices A and B of the size $M \times P$ and $P \times Q$, respectively, the third file should be created to store the resulting matrix product $C = AB$. Using successively the Matr3ScatterFile, Matr3Init, Matr3Calc (in a loop), and Matr3GatherFile functions (see the MPI9Matr29, MPI9Matr24, MPI9Matr25, and MPI9Matr30 tasks), find a matrix C and write its elements to the resulting file.

In addition, output the current value of the $c[\text{step}]$ in each process after each call of the Matr3Calc function, where c is a one-dimensional array containing the block C_R , and step is the algorithm step number $(0, 1, \dots, K_0 - 1)$. Thus, the element $c[0]$ should be output on the first step of the algorithm, the element $c[1]$ should be output on the second step of the algorithm, and so on.

2.9.5. Fox's block algorithm

MPI9Matr32. Integers M and P are given in each process; in addition, a matrix A of the size $M \times P$ is given in the master process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$, the numbers M and P are multiples of K_0 . Input the matrix A into a one-dimensional array of the size $M \cdot P$ in the master process and define a new datatype named `MPI_BLOCK_A` that contains a $M_0 \times P_0$ block of the matrix A , where $M_0 = M/K_0$, $P_0 = P/K_0$.

When defining the `MPI_BLOCK_A` type, use the `MPI_Type_vector` and `MPI_Type_commit` functions. Include this definition in a `Matr4CreateTypeBlock(m0, p0, p, t)` function with the input integer parameters m_0 , p_0 , p and the output parameter t of the `MPI_Datatype` type; the parameters m_0 and p_0 determine the size of the block, and the parameter p determines the number of columns of the matrix from which this block is extracted.

Using the `MPI_BLOCK_A` datatype, send to each process (in ascending order of ranks of processes, inclusive of the master process) the corresponding block of the matrix A in a row-major order of blocks (that is, the first block should be sent to the process 0, the next block in the same row of blocks should be sent to the process 1, and so on). Sending should be performed using the `MPI_Alltoallw` function; the blocks should be stored in one-dimensional arrays of the size $M_0 \cdot P_0$. Output the received block in each process.

Notes. (1) Use the `MPI_Send` and `MPI_Recv` functions instead of the `MPI_Alltoallw` function when solving this task using the MPI-1 library.

(2) The `MPI_Alltoallw` function introduced in MPI-2 is the only collective function that allows you to specify the displacements for the sent data in *bytes* (not in elements). This gives opportunity to use it in conjunction with complex data types to implement any variants of collective communications (in our case, we need to implement a communication of the scatter type).

It should be note that all array parameters of the `MPI_Alltoallw` function associated with the sent data must be differently defined in the master and slave processes. In particular, the array `scounts` (which determines the number of sent elements) must contain the values 0 in all the slave processes and the value 1 in the master process (the sent elements are of the `MPI_BLOCK_A` datatype).

At the same time, arrays associated with the received data will be defined in the same way in all processes; in particular, the zero-indexed element of the array `rcounts` (which determines the number of received elements) must be equal to $M_0 \cdot P_0$, and all other elements of this array must be equal to 0 (the received elements are of the `MPI_INT` datatype).

It is necessary to pay special attention to the correct definition of elements in the array `sdispls` of displacements for the sent data in the master process (in the slave processes, it is enough to use the zero-valued array `sdispls`).

MPI9Matr33. Integers M_0 , P_0 and a matrix A of the size $M_0 \times P_0$ are given in each process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$. Input the matrix A into a one-dimensional array of the size $M_0 \cdot P_0$ in each process and create a new communicator named `MPI_COMM_GRID` using the `MPI_Cart_create` function. The `MPI_COMM_GRID` communicator defines a Cartesian topology for all processes as a two-dimensional periodic $K_0 \times K_0$ grid (ranks of processes should not be reordered).

Include the creation of the `MPI_COMM_GRID` communicator in a `Matr4CreateCommGrid(comm)` function with the output parameter `comm` of the `MPI_Comm` type. Using the `MPI_Cart_coords` function for this communicator, output the process coordinates (I_0, J_0) in each process.

On the base of the `MPI_COMM_GRID` communicator, create a set of communicators named `MPI_COMM_ROW`, which are associated with the rows of the initial two-dimensional grid. Use the `MPI_Cart_sub` function to create the `MPI_COMM_ROW` communicators.

Include the creation of the `MPI_COMM_ROW` communicators in a `Matr4CreateCommRow(grid, row)` function with the input parameter `grid` (the communicator associated with the initial two-dimensional grid) and the output parameter `row` (both parameters are of the `MPI_Comm` type). Output the process rank R_0 for the `MPI_COMM_ROW` communicator in each process (this rank must be equal to J_0).

In addition, send the matrix A from the grid element (I_0, I_0) to all processes of the same grid row I_0 ($I_0 = 0, \dots, K_0 - 1$) using the `MPI_Bcast` collective function for the `MPI_COMM_ROW` communicator. Save the received matrix in the auxiliary matrix T of the same size as the matrix A (it is necessary to copy the matrix A to the matrix T in the sending process before the call of the `MPI_Bcast` function). Output the received matrix T in each process.

MPI9Matr34. Integers P_0 , Q_0 and a matrix B of the size $P_0 \times Q_0$ are given in each process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$. Input the matrix B into a one-dimensional array of the size $P_0 \cdot Q_0$ in each process and create a new communicator named `MPI_COMM_GRID`, which defines a Cartesian topology for all processes as a two-dimensional periodic $K_0 \times K_0$ grid.

Use the `Matr4CreateCommGrid` function (see the `MPI9Matr33` task) to create the `MPI_COMM_GRID` communicator. Using the `MPI_Cart_coords` function for this communicator, output the process coordinates (I_0, J_0) in each process.

On the base of the `MPI_COMM_GRID` communicator, create a set of communicators named `MPI_COMM_COL`, which are associated with the columns of

the initial two-dimensional grid. Use the `MPI_Cart_sub` function to create the `MPI_COMM_COL` communicators.

Include the creation of the `MPI_COMM_COL` communicators in a `Matr4CreateCommCol(grid, col)` function with the input parameter `grid` (the communicator associated with the initial two-dimensional grid) and the output parameter `col` (both parameters are of the `MPI_Comm` type). Output the process rank R_0 for the `MPI_COMM_COL` communicator in each process (this rank must be equal to I_0).

In addition, perform a cyclic shift of the matrices B given in all processes of each grid column J_0 by 1 position up (that is, in descending order of ranks of processes) using the `MPI_Sendrecv_replace` function for the `MPI_COMM_COL` communicator (to determine the ranks of the sending and receiving processes, use the expression containing the `%` operator that gives the remainder of a division). Output the received matrix in each process.

MPI9Matr35*. Integers M, P, Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process. The number of processes K is a perfect square: $K = K_0 \cdot K_0$. In the block algorithms of matrix multiplication, the initial matrices are divided into K blocks and are interpreted as square block matrices of the order K_0 (hereinafter blocks are distributed by processes and used to calculate a part of the total matrix product in each process).

The block of the matrix A is of the size $M_0 \times P_0$, the block of the matrix B is of the size $P_0 \times Q_0$, the numbers M_0, P_0, Q_0 are calculated as follows: $M_0 = \text{ceil}(M/K_0)$, $P_0 = \text{ceil}(P/K_0)$, $Q_0 = \text{ceil}(Q/K_0)$, where the operation `"/` means the division of real numbers and the function `ceil` performs rounding up. If the matrix contains insufficient number of rows (or columns) to fill the last blocks, then the zero-valued rows (or columns) should be added to these blocks.

Add, if necessary, the zero-valued rows or columns to the initial matrices (as a result, the matrices A and B will have the size $(M_0 \cdot K_0) \times (P_0 \cdot K_0)$ and $(P_0 \cdot K_0) \times (Q_0 \cdot K_0)$ respectively), save them in one-dimensional arrays in the master process, and then send the matrix blocks (in a row-major order) from these arrays to all processes (in ascending order of its ranks): the process R will receive the blocks A_R and B_R , $R = 0, \dots, K - 1$. In addition, create two blocks C_R and T_R filled with zero-valued elements in each process: the block C_R is intended to store the part of the matrix product $C = AB$, which will be calculated in this process, the block T_R is an auxiliary one. Each block C_R and T_R is of the size $M_0 \times Q_0$.

The blocks, like the initial matrices, should be stored in one-dimensional arrays in a row-major order. To send the matrix sizes, use the `MPI_Bcast` collective function, to send the blocks of the matrices A and B , use the `MPI_Alltoallw` collective function and also the `MPI_BLOCK_A` and `MPI_BLOCK_B`

datatypes created by the `Matr4CreateTypeBlock` function (see the `MPI9Matr32` task and notes to it).

Include all the above mentioned actions in a `Matr4ScatterData` function (without parameters). As a result of the call of this function, each process will receive the values M_0 , P_0 , Q_0 , as well as one-dimensional arrays filled with the blocks A_R , B_R , C_R , T_R . Output all obtained data (that is, the numbers M_0 , P_0 , Q_0 and the blocks A_R , B_R , C_R , T_R) in each process after calling the `Matr4ScatterData` function. Perform the input of initial data in the `Matr4ScatterData` function, perform the output of the results in the `Solve` function.

Notes. (1) When input the matrices A and B into arrays in the master process, it should be taken into account that these arrays may contain elements corresponding to additional zero-valued columns.

(2) To reduce the number of the `MPI_Bcast` function calls, all matrix sizes may be sent as a single array.

MPI9Matr36. Integers M_0 , P_0 , Q_0 and one-dimensional arrays filled with the blocks A_R , B_R , C_R , T_R are given in each process (thus, the given data coincide with the results obtained in the `MPI9Matr35` task). A virtual Cartesian topology in the form of a square grid of order K_0 is used for all processes, the value of $K_0 \cdot K_0$ is equal to the number of the processes. Each step of the Fox's block algorithm of matrix multiplication consists of two stages.

In the first stage of the first step, the block A_R is sent from the process with the grid coordinates (I_0, I_0) to all processes of the same grid row I_0 ($I_0 = 0, \dots, K_0 - 1$). The received block is saved in the block T_R in the receiving processes. Then the block T_R is multiplied by the block B_R from the same process and the result is added to the block C_R .

Implement the first stage of the first step of the Fox's algorithm. To do this, create the `MPI_COMM_GRID` and `MPI_COMM_ROW` communicators using the `Matr4CreateCommGrid` and `Matr4CreateCommRow` functions implemented in the `MPI9Matr33` task. Use the `MPI_Bcast` function for the `MPI_COMM_ROW` communicator to send the blocks A_R (compare with `MPI9Matr33`).

Include all the above mentioned actions in a `Matr4Calc1` function (without parameters). Output the new contents of the blocks T_R and C_R in each process; perform data input and output in the `Solve` function.

MPI9Matr37. Integers M_0 , P_0 , Q_0 and one-dimensional arrays filled with the blocks A_R , B_R , C_R , T_R are given in each process (thus, the given data coincide with the results obtained in the `MPI9Matr35` task).

Implement the second stage of the first step of the Fox's algorithm of matrix multiplication as follows: perform a cyclic shift of the blocks B_R given in all processes of each column of the Cartesian periodic grid by 1 position up (that is, in descending order of ranks of processes).

To do this, create the `MPI_COMM_GRID` and `MPI_COMM_COL` communicators using the `Matr4CreateCommGrid` and `Matr4CreateCommCol` functions implemented in the `MPI9Matr34` task, then use the `MPI_Bcast` function for the `MPI_COMM_COL` communicator to perform the cyclic shift of the blocks B_R (compare with `MPI9Matr34`).

Include all the above mentioned actions in a `Matr4Calc2` function (without parameters). Output the received blocks B_R in each process; perform data input and output in the `Solve` function.

MPI9Matr38. Integers M_0 , P_0 , Q_0 and one-dimensional arrays filled with the blocks A_R , B_R , C_R , T_R are given in each process (thus, the given data coincide with the results obtained in the `MPI9Matr35` task).

Modify the `Matr4Calc1` function, which was implemented in the `MPI9Matr36` task; the modified function should provide execution of the first stage of any step of the Fox's algorithm. To do this, add the parameter named `step` to the function (this parameter specifies the step number and may be in the range 0 to $K_0 - 1$, where K_0 is the order of the Cartesian grid of processes) and use the value of this parameter in the part of the algorithm that deals with the sending the blocks A_R : the block A_R should be sent from the process with the grid coordinates $(I_0, (I_0 + \text{step})\%K_0)$ to all processes of the same grid row I_0 , $I_0 = 0, \dots, K_0 - 1$ (the recalculation of the elements of the block C_R does not depend on the value of the parameter `step`).

Using successively the calls of `Matr4Calc1(0)`, `Matr4Calc2()`, `Matr4Calc1(1)` (the `Matr4Calc2` function provides the second stage of each step of the algorithm — see the `MPI9Matr37` task), execute two initial steps of the Fox's algorithm and output the new contents of the blocks T_R , B_R , and C_R in each process. Perform data input and output in the `Solve` function.

MPI9Matr39*. Integers M_0 , P_0 , Q_0 and one-dimensional arrays filled with the blocks A_R , B_R , C_R , T_R are given in each process (thus, the given data coincide with the results obtained in the `MPI9Matr35` task). In addition, a number L with the same value is given in each process. The value of L is in the range 3 to K_0 and determines the number of steps of the Fox's algorithm.

Using successively the calls of `Matr4Calc1(0)`, `Matr4Calc2()`, `Matr4Calc1(1)`, `Matr4Calc2()`, ..., `Matr4Calc1(L - 1)` (see the `MPI9Matr38` and `MPI9Matr37` tasks), execute the initial L steps of the Fox's algorithm and output the new contents of the blocks T_R , B_R , and C_R in each process. Perform data input and output in the `Solve` function.

Remark. If the value of L is equal to K_0 , then the blocks C_R will contain the corresponding parts of the final matrix product AB . Note that the second stage (associated with the call of the `Matr4Calc2` function) is not necessary at the last step of the algorithm.

MPI9Matr40*. Integers M and Q (the numbers of rows and columns of the matrix product) are given in the master process. In addition, integers M_0 , Q_0 and one-dimensional arrays filled with the $M_0 \times Q_0$ blocks of the matrix C are given in each process (the given blocks of C are obtained as a result of K_0 steps of the Fox's algorithm — see the MPI9Matr39 task).

Send all the blocks C_R to the master process and output the received matrix C of the size $M \times Q$ in this process. To store the resulting matrix C in the master process, use a one-dimensional array sufficient to store the matrix of the size $(M_0 \cdot K_0) \times (Q_0 \cdot K_0)$. To send the blocks C_R to this array, use the MPI_Alltoallw collective function and the MPI_BLOCK_C datatype created by the Matr4CreateTypeBlock function (see the MPI9Matr32 task and notes to it). Include all the above mentioned actions in a Matr4GatherData function (without parameters). Perform the input of initial data in the Solve function, perform the output of the resulting matrix in the Matr4GatherData function.

Note. When output the matrix C in the master process, it should be taken into account that an array, which is intended for matrix storage, may contain elements corresponding to additional zero-valued columns.

MPI9Matr41**. Integers M , P , Q , a matrix A of the size $M \times P$, and a matrix B of the size $P \times Q$ are given in the master process (thus, the given data coincide with the given data in the MPI9Matr35 task).

Using successively the Matr4ScatterData, Matr4Calc1, Matr4Calc2, and Matr4GatherData functions, that are developed in the MPI9Matr35–MPI9Matr40 tasks, find a matrix C , which is equal to the product of the initial matrices A and B , and output this matrix in the master process. The Matr4Calc1 and Matr4Calc2 functions should be called in a loop, the number of Matr4Calc2 function calls must be one less than the number of Matr4Calc1 function calls.

In addition, output the current contents of the block C_R in each process after each call of the Matr4Calc1 function.

The MPI_COMM_GRID, MPI_COMM_ROW, and MPI_COMM_COL communicators that are used in the Matr4Calc1 and Matr4Calc2 functions, should not be created several times. To do this, modify the Matr4CreateCommGrid, Matr4CreateCommRow, and Matr4CreateCommCol functions (see the MPI9Matr33 and MPI9Matr34 tasks); the modified functions should not perform any actions when it is called with the parameter comm that is not equal to MPI_COMM_NULL.

MPI9Matr42*. Integers M , P , Q and two file names are given in the master process. The given files contain elements of a matrix A of the size $M \times P$ and a matrix B of the size $P \times Q$. The numbers M , P , and Q are multiples of the order K_0 of square grid of processes.

Modify the stage of receiving blocks for the Fox's algorithm of matrix multiplication (see the MPI9Matr35 task) as follows: each process should read

the corresponding blocks of the matrices A and B directly from the given files.

To send the sizes of matrices and file names, use the `MPI_Bcast` collective function. To read the blocks from the files, set the appropriate file views using the `MPI_File_set_view` function and the `MPI_BLOCK_A` and `MPI_BLOCK_B` filetypes defined with the `Matr4CreateTypeBlock` function (see the `MPI9Matr32` task), and then use the `MPI_File_read_all` function.

Include all these actions in a `Matr4ScatterFile` function (without parameters). As a result of the call of this function, each process will receive the values M_0 , P_0 , Q_0 , as well as one-dimensional arrays filled with the blocks A_R , B_R , C_R , T_R (the blocks C_R and T_R should contain zero-valued elements). Output all obtained data (that is, the numbers M_0 , P_0 , Q_0 and the blocks A_R , B_R , C_R , T_R) in each process after calling the `Matr4ScatterFile` function. Perform the input of initial data in the `Matr4ScatterFile` function, perform the output of the results in the `Solve` function.

Remark. A condition that the numbers M , P , Q are multiples of K_0 means that you do not need to add zero-valued rows and/or zero-valued columns to the blocks obtained from the matrices A and B , and therefore you may perform reading of the blocks A_R and B_R using the same filetypes (namely, `MPI_BLOCK_A` and `MPI_BLOCK_B`) in all processes.

If this condition is not fulfilled, then it would be necessary to use special types that ensure the correct reading from the file and write to the array of "truncated" blocks of the matrices A and B in some processes (in addition, in this case it would be necessary to send to each process the values of P and Q that are necessary for the correct type definition for "truncated" blocks).

MPI9Matr43*. Integers M_0 , Q_0 and one-dimensional arrays filled with the $M_0 \times Q_0$ blocks C_R are given in each process (the given blocks C_R are obtained as a result of K_0 steps of the Fox's block algorithm of matrix multiplication — see the `MPI9Matr39` task). In addition, the name of file to store the matrix product is given in the master process. The numbers M and Q (the numbers of rows and columns of the matrix product) are multiples of the order K_0 of square grid of processes (thus, $M = M_0 \cdot K_0$, $Q = Q_0 \cdot K_0$).

Send the file name to all processes using the `MPI_Bcast` function. Write all the parts of the matrix product contained in the blocks C_R to the resulting file, which will eventually contain a matrix C of the size $M \times Q$.

To write the blocks to the files, set the appropriate file view using the `MPI_File_set_view` function and the `MPI_BLOCK_C` filetype defined with the `Matr4CreateTypeBlock` function (see the `MPI9Matr32` task), and then use the `MPI_File_write_all` collective function.

Include all these actions (namely, the input of file name, sending the file name, and writing all blocks to the file) in a `Matr4GatherFile` function. Perform the input of all initial data, except the file name, in the `Solve` function.

Remark. A condition that the numbers M and Q are multiples of K_0 means that the blocks C_R do not contain "extra" zero-valued rows and/or columns, and therefore you may perform writing of the blocks C_R to the file using the same filetype (namely, `MPI_BLOCK_C`) in all processes.

MPI9Matr44.** Integers M , P , Q and three file names are given in the master process. The first two file names are related to the existing files containing the elements of matrices A and B of the size $M \times P$ and $P \times Q$, respectively, the third file should be created to store the resulting matrix product $C = AB$. The numbers M , P , and Q are multiples of the order K_0 of square grid of processes.

Using successively the `Matr4ScatterFile`, `Matr4Calc1`, `Matr3Calc2`, and `Matr4GatherFile` functions (see the `MPI9Matr42`, `MPI9Matr38`, `MPI9Matr37`, and `MPI9Matr43` tasks), find a matrix C and write its elements to the resulting file. The `Matr4Calc1` and `Matr4Calc2` functions should be called in a loop, the number of `Matr4Calc2` function calls must be one less than the number of `Matr4Calc1` function calls.

In addition, output the current value of the `c[step]` in each process after each call of the `Matr4Calc1` function, where `c` is a one-dimensional array containing the block C_R , and `step` is the algorithm step number ($0, 1, \dots, K_0 - 1$). Thus, the element `c[0]` should be output on the first step of the algorithm, the element `c[1]` should be output on the second step of the algorithm, and so on.

3. Additions

3.1. *Programming Taskbook for MPI-2*

3.1.1. General description

Electronic problem book on parallel programming **Programming Taskbook for MPI-2** (PT for MPI-2) is an extension of the universal electronic problem book **Programming Taskbook**. PT for MPI-2 allows you to solve tasks on developing parallel programs using MPI of the standard 1.1, 2.x, and partially 3.x.

To be able to use PT for MPI-2, it should be installed in the system directory of the basic version of the Programming Taskbook version not lower than 4.17 (usually the system directory of the taskbook is C:\Program Files (x86)\PT4). The MPI support system for Windows, which ensures the launch of programs in parallel mode, should also be installed on the computer. The taskbook can be used together with the following MPI support systems:

- **MPICH 1.2.5** (<ftp://ftp.mcs.anl.gov/pub/mpi/nt/mpich.nt.1.2.5.exe>), supports the MPI 1.2 standard;
- **MPICH2 1.3** (<http://www.mpich.org/static/downloads/1.3/mpich2-1.3-win-ia32.msi>), supports the MPI 2.1 standard;
- **MS-MPI 10.1.2** (<https://www.microsoft.com/en-us/download/details.aspx?id=100593>), supports MPI-2.1 (and partially MPI-3), provides faster operation of parallel programs for Windows 10 (requires downloading the mspisetup.exe installation file).

For MPI system installation, see the note in Section 1.1.1. If several MPI systems are installed on a computer, then you can use the **PT4Setup** and **PT4Load** applications to select the required version for use with your programs.

The PT for MPI-2 taskbook is a freeware; it can be used both with the full version of the Programming Taskbook (PT4Complete) and with the freely distributed mini-version (PT4Mini).

The tasks included in the PT for MPI-2 taskbook can be executed in C++ in all programming environments for this language supported by the Programming Taskbook. For the Programming Taskbook version 4.24 released in 2024, the following IDEs are supported:

- Microsoft Visual Studio (version 2017, 2019, 2022),
- Code::Blocks 20.03,
- Dev-C++ 5.11 and 6.30,

- Visual Studio Code.

The dynamic library included in the MPI support system must be connected to the learning programs. Access to the library is provided using a lib file and a set of header files. The taskbook performs all actions to copy additional files to the working directory and connect them to the student's project automatically.

The PT for MPI-2 taskbook provides the same capabilities as the basic Programming Taskbook when solving tasks. In particular, it passes the initial data to the student's program, checks the correctness of the results obtained by the program by executing a series of test runs of the student's program, and saves information about each test run in a special file. The PT for MPI-2 taskbook also provides additional capabilities related to the specifics of solving tasks on parallel programming (these capabilities are described in more detail in Section 1.1):

- task demo running that does not require the use of parallel mode;
- creation of a template project for the selected task with MPI library modules connected to it;
- a special mechanism that ensures the execution of a student's program in parallel mode when it is launched from the IDE: the launched program launches the host application from the MPI support system, which, in turn, launches the program in parallel mode (all processes are executed on the local computer);
- passing to each process of a parallel program its own set of initial data;
- receiving the required results from each process and automatically sending them to the master process for checking and displaying in the taskbook window;
- output of information about run-time errors (including errors that occurred while executing MPI functions) and input-output errors, indicating the ranks of the processes in which these errors occurred;
- the ability to output debugging information for each process in a special section of the taskbook window;
- automatic unloading from memory of all running processes even if a parallel program hangs.

The above features eliminate the need for the student to perform additional actions associated with running his program in parallel mode and make it easier to identify and correct common errors that occur in parallel programs.

The use of initial data prepared by the taskbook for each process of the parallel program, a visual output in one window of all the results obtained by each process and their automatic checking, as well as additional debugging tools allow the student to focus on the implementation of the algorithm for solving the task and ensure reliable testing of the developed algorithm.

The PT for MPI-2 taskbook is an extended version of the Programming Taskbook for MPI (PT for MPI), which uses the MPI-1 standard and was developed in 2009 [1]. In addition to 100 training tasks of the PT for MPI taskbook, the PT for MPI-2 taskbook includes 165 new tasks. Some of the new tasks complement previous topics, some are related to new features of the MPI-2 and MPI-3 standard.

Along with the groups devoted to specific sections of the MPI library, the taskbook includes a group of tasks on developing *parallel algorithms* (namely, band and block matrix multiplication algorithms) that use various MPI tools studied in the previous groups.

The formulations of all tasks included in the PT for MPI-2 taskbook, are given in Section 2.

The tools for automatic launch and debugging of parallel applications implemented in the PT for MPI-2 taskbook allow it to be used for developing and testing parallel programs not related to specific learning tasks. With this purpose, the PT for MPI-2 taskbook (as previously the PT for MPI taskbook) includes an auxiliary MPIDebug group of 36 "tasks", each of which provides automatic launch of a parallel program, and the number of processes is determined by the ordinal number of the task (an example of using this group is given in Section 1.5.1). Thus, the "tasks" of the MPIDebug group allow launching any parallel program with the required number of processes directly from IDE and provide the debugging tools available in the taskbook.

The debug features provided in the taskbook can also be used when developing console parallel programs that do not require connecting the taskbook core. For these purposes, it is sufficient to connect the `pt4console.h` and `pt4console.cpp` files to the program, located in the `PTforMPI2\stubs` subdirectory of the Programming Taskbook system directory. An example of using the `pt4console.h` and `pt4console.cpp` files is given in Section 1.5.3.

The PT for MPI-2 taskbook includes a hypertext help system `PTforMPI2.chm`, which contains the same information as the section of the `ptaskbook.com` website related to the PT for MPI-2 taskbook.

The software "**Electronic taskbook on parallel programming Programming Taskbook for MPI-2**" was registered in the Register of computer programs on February 2, 2018 (certificate of state registration of computer program No. 2018611548).

Detailed information about the capabilities of the PT for MPI-2 taskbook is available on the website `ptaskbook.com`. From this website, you can download the latest versions of the Programming Taskbook and its PT for MPI-2 extension.

3.1.2. Taskbook tools for initializing tasks and data input-output

All the tools of the taskbook described below are implemented in the files `pt4.h` and `pt4.cpp`. When creating a template project using the PT4Load application, these files are connected to the project automatically.

To **initialize a task**, the function `void Task(const char *name)` is used. The `Task` function must be called at the beginning of the program solving the task (before performing input-output operations). If the `Task` function is not specified in the program that solves the task, the message *"The Task procedure with a task name is not called"* will appear in the taskbook window when the program is started.

If the `Task` function is called several times in a program, all subsequent calls to it are ignored, except when the `Task` function is used to generate html pages with task text (see below).

The task name must include the task group name and the task number within the group (e. g. "MPI1Proc2"). The group name is case-insensitive. If an invalid group is specified, the program will display the message *"Invalid task group"*. If an invalid task number is specified, the program will display a message indicating the range of acceptable task numbers for the group. If the task name is followed by the "?" symbol in the name parameter (e. g. "MPI1Proc 2?"), then the program will be launched *in the demo mode*, which has the following features:

- even if the program contains a solution to the task, this solution is not analyzed and the information is not saved in the results file;
- when you launch the program, you can view *several* versions of the initial and control data; to change the data set, you need to press the **New data** button in the taskbook window or the space bar;
- when you launch the program, you can sequentially view *all* the tasks in a given group; to move to a task with the next number, you need to press the **Next task** button or the [Enter] key, and to move to a task with the previous number, you need to press the **Previous task** button or the [Backspace] key.

The demo run of the parallel programming task is performed in non-parallel mode, without access to the MPI support system.

The `Task` function can also be used to generate and display an html page with the text of a task or a group of tasks. To do this, you need to specify the name of a particular task or group of tasks and the "#" symbol as the name parameter, for example, "MPI1Proc2#" or "MPI1Proc#". To include several tasks or groups of tasks in an html page, it is enough to call the `Task` function for each task or group with a parameter ending with the # symbol.

The main tool of **data input-output** when solving C++ tasks using the Programming Taskbook is the `pt` input-output stream defined in the `pt4.h` header file. It can be used to input and output data of all standard types, including `int`, `double`, `char*` used in the PT for MPI-2 taskbook. The `pt4.h` header file also contains functions intended for input and output of separate elements, but the use of

the `pt` stream is a more convenient way for input-output. To input data from the `pt` stream, the `>>` operator is provided (e.g. `pt >> a >> b;`); the `<<` operator is used to output the results to this stream (e.g. `pt << 2 * a << 3 + b;`).

Starting with the taskbook version 4.22, to output all elements of a vector `std::vector<T>` and other STL containers, it is enough to pass the container name to the `pt` stream. Examples of this feature are given in the program fragments at the end of Sections 1.2.6 and 1.4.2.

Input and output of data must be performed after the Task function is called, otherwise the error message *"The Task procedure with a task name is not called at beginning of the program"* will be displayed.

The *iterators* `ptin_iterator<T>` and `ptout_iterator<T>` associated with the stream `pt` are provided for input-output of *sequences* with elements of type `T`. These iterators have the same properties as the standard stream iterators `istream_iterator<T>` and `ostream_iterator<T>` (see [2]).

The `ptin_iterator<T>` iterator has two constructors. The constructor without parameters creates an end-of-sequence iterator. The constructor with the count parameter of the unsigned int type creates an input iterator that allows to read count of elements of type `T` from the `pt` stream and then passes to the end-of-sequence state (i. e., it becomes equal to the iterator `ptin_iterator<T>()`).

In case of the special value of the count parameter equal to 0, the size of the sequence is read from the `pt` stream before reading the first element of the sequence. If, when reading the size, it turns out that the element read is not an integer or this number is not positive, then the iterator immediately passes to the end-of-sequence state.

These features of the input iterators `ptin_iterator<T>` makes it easy to implement reading of several sequences if their size is known in advance or if the size is specified immediately before the beginning of the sequence.

The `ptout_iterator<T>` output iterator is created using the constructor without parameters and allows writing an arbitrary amount of data of type `T` to the stream `pt`.

3.1.3. Debug section

Programming Taskbook includes tools that allow you to output debug information directly to its window (in a special *debug section*). The need for such additional tools arises, first of all, when debugging parallel programs, since such standard debugging tools as breakpoints, step-by-step program execution, and variable watch cannot be used for them. It should also be noted that the ability to output information to the debug section allows you to use the PT for MPI-2 taskbook for developing and debugging parallel programs that are not related to solving specific tasks (see Section 1.5.1).

The debugging tools of the taskbook can also be useful for non-parallel programs. In this case, they can be used as a supplement to the tools of the built-in debugger.

The debug section consists of one or more multi-line text output areas. It is located under the main sections of the taskbook window and is displayed on the screen only if it contains some text (Fig. 59).

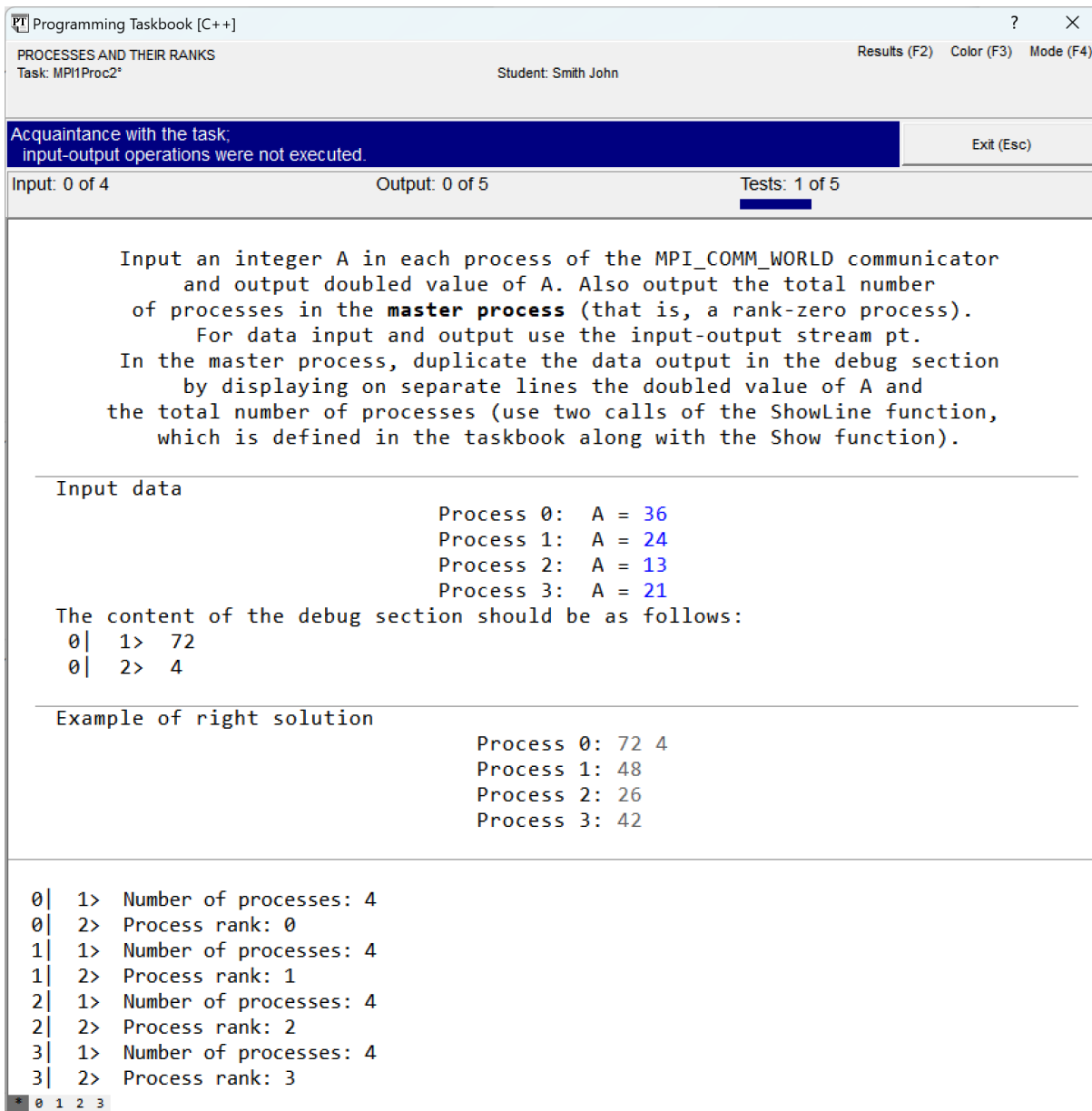


Fig. 549. Taskbook window with the debug section

When running the program in the demo mode, all functions related to the debug section are ignored, so the debug section is not displayed on the screen.

It is possible to hide all sections of the taskbook window except for the debug section; to do this, simply press the key combination [Ctrl]+[space]. Pressing this shortcut again restores previously hidden sections in the taskbook win-

dow. You can use the corresponding pop-up menu command to hide/show the main sections of the taskbook window. You can also hide all sections of the taskbook window except for the debug section programmatically by calling the `HideTask` function (see Section 3.1.4).

For non-parallel tasks, the debug section contains a single output area. For parallel tasks, the number of output areas is determined by the number of parallel application processes that have output data to the debug section. At any time, one of the areas is displayed in the debug section. If there is more than one output area, a set of *markers* is displayed at the bottom of the debug section, allowing you to switch to any of the available output areas.

Markers with numbers allow you to view the contents of the output area associated with the process of the corresponding rank (Fig. 60); a marker with the "*" symbol allows you to view the output area containing the combined text of all other areas (see Fig. 59).



```
Programming Taskbook [C++] ? X
3| 1> Number of processes: 4
3| 2> Process rank: 3
* 0 1 2 3
```

Fig. 60. Debug information for a process of rank 3

Each screen line in the debug section begins with the process rank followed by the "|" character. Then comes the line number, which ends with the ">" character (lines for each process are numbered independently).

For processes from the original communicator `MPI_COMM_WORLD`, their ranks in this communicator are specified. For dynamically created groups of processes (see Section 1.3.8), their ranks in the `MPI_COMM_WORLD` communicator associated with these groups are specified, with the letter prefixes "a", "b", etc. specified before the rank value. All processes included in one group have the same prefix. The presence of prefixes allows us to distinguish between original and new processes, as well as new processes included in different groups. An example of a debug section with information output in dynamically created processes is given in Section 1.3.8 (see Figs. 37 and 38).

The number of debug lines for each process must not exceed 999; if a process attempts to output data to a line number greater than 999, an error message is output in the area associated with that process and further debug output for that process is blocked. This limitation avoids problems that arise when a slave process endlessly outputs debug information.

To switch to the required output area, you can simply click the mouse on the corresponding marker. In addition, to sequentially cycle through the markers from left to right or from right to left, you can use the arrow keys [→] and [←]. You can also immediately go to the required output area by pressing the corresponding key: for the "*" area, press the [*] key; for areas "0"–"9", press the numeric keys [0]–[9]; for areas "10"–"35", press the letter keys from [A] to [Z] (for areas associated with dynamically created processes, the keys are not defined).

Debugging information received from slave processes is pre-saved in special temporary files in the student's directory, so it will be available for viewing even if some slave processes hang at some stage of the parallel program execution. Debugging information received from the master process is output directly to the debug section.

The contents of the output area displayed in the debug section can be copied to the Windows clipboard; for this purpose, use the standard [Ctrl]+[C] shortcut or the corresponding command in the pop-up menu of the taskbook window.

3.1.4. Functions for outputting and configuring debug information

The Show function is used to output data to the debug section. Its simplest version contains a single string parameter *s* of type `std::string` or `char*`:

```
void Show(const char *s);  
void Show(std::string s);
```

This function outputs the string *s* to the output area of the debug section.

If the current screen line already contains some text, then the string *s* is supplied with an initial space and appended to this text, except for the case when, with such appending, the size of the resulting text would exceed the width of the output area (equal to 80 characters). In the latter case, the output of the string *s* is performed from the beginning of the next screen line; if, in this situation, the string *s* exceeds the width of the output area, then the string *s* will be output on several screen lines, and the text will be broken at the whitespace characters of the string *s*, and if there are no spaces, when the next fragment of the string of length equal to 80 is reached.

The string *s* may contain explicit commands to move to a new screen line. Such commands may be either the '\r' (carriage return) character, or the '\n' (new-line) character, or a combination of both in that order "\r\n".

There is a modification of the Show function, the ShowLine function, which, after displaying the line *s*, automatically goes to a new line of the debug section:

```
void ShowLine(const char *s);  
void ShowLine(std::string s);
```

The ShowLine function can be called without a string parameter; in this case, it only goes to a new line in the debug section:

```
void ShowLine();
```

In addition to the Show function with a single string parameter, its overloaded versions are provided for outputting numeric debugging data. Using these versions allows you to simplify actions related to the output of numeric data, because it eliminates the necessity to use standard C++ tools intended for converting numbers into their string representations.

There are several overloaded versions of the Show function intended for outputting numerical data. Below are the versions containing the largest number of parameters (other versions are obtained from the given ones by removing parameter *s*, parameter *w*, or both):

```
void Show(const char *s, int a, int w);
void Show(const char *s, double a, int w);
void Show(std::string s, int a, int w);
void Show(std::string s, double a, int w);
```

The string parameter *s* specifies an optional *comment* that appears before the number to be output; if *s* is omitted, the comment is assumed to be an empty string.

The numeric parameter *a* specifies the number to be output.

The optional integer parameter *w* specifies *the width of the output field* (i. e., the number of screen positions allocated for displaying the number). If the specified output field width *w* is not enough, then the value of parameter *w* is ignored; in this case (and also if parameter *w* is missing), the minimum output field width required to display the given number is used. If the number does not occupy the entire output field, it is padded on the left with spaces (i. e., aligned with *the right* border of the output field). A dot is used as a decimal separator for numbers with a fractional part. Real numbers are output by default in a format with a fixed point and two fractional digits. You can change the output format of real numbers using the SetPrecision function described below.

Similar overloaded versions are provided for the ShowLine function. In these versions, after the debug data is output, the transition to the new line of the debug section is automatically performed.

Note. There are also overloaded *template versions* of the Show and ShowLine functions with *iterator* parameters *first* and *last* intended for outputting elements of *STL containers* (see [2]). The iterators *first* and *last*, as usual, define the initial element to be output and the position after the final element to be output. These versions may contain an optional third parameter *s* of type `std::string`, which defines a comment that will be output before the output sequence of elements. The template version of the Show function outputs the elements of the sequence on one line, separating them with spaces; after the end of the output, a transition to a new line is automatically performed. The template version of the ShowLine function outputs each element of the sequence on a new line; the string comment is output on the same line as the first element.

There are also two helper functions associated with debugging output: HideTask and SetPrecision.

Calling a void HideTask() function ensures that all sections of the taskbook window are hidden except for the debug section. If the debug section is not displayed in the taskbook window (in particular, if the program is running in demo mode), then the call to the HideTask function is ignored.

You can hide/restore the main sections of the taskbook window after it is displayed on the screen using the combination [Ctrl]+[space] or the corresponding command in the pop-up menu.

The void SetPrecision(int n) function is used to configure the format of real debug data. If parameter n is positive, then it defines the number of fractional digits to be output; the number is output in fixed-point format. If parameter n is zero or negative, then real numbers are output in floating-point (exponential) format. The number of fractional digits is assumed to be equal to *the absolute value* of parameter n; if parameter n is zero, the number of fractional digits is set to 6.

The current numeric format setting defined by the SetPrecision function lasts until the next call to this function. Before the first call of the SetPrecision function, real numbers are output in fixed-point format with two fractional digits.

3.2. Options for individual assignments

3.2.1. Series of similar tasks

Tasks included in the PT for MPI-2 taskbook and presented in Section 2 can be divided into *series* of several (2 to 4) similar tasks. Below is a list of such series. It contains all the tasks from Section 2 except those solved in Section 1. Additionally, the comparative *complexity* of each series is indicated. Complexity is estimated in points (1 to 3) that are given after the name of the group of tasks and separated from it by dash "-" (note that the tasks estimated at 2 and 3 points are marked in Section 2 with the symbols * and ** respectively).

```
-Processes and their ranks
%(task 2 is solved in Section 1.1)
MPI1Proc-1 1 3
MPI1Proc-1 4 5 6 7
MPI1Proc-1 8 9 10

-Blocking data transfer
%(task 11 is solved in Section 1.2.2)
MPI2Send-1 1 2 3 4
MPI2Send-1 5 6 7
MPI2Send-1 8 9 10
MPI2Send-1 12 13 14
MPI2Send-1 15 16 17
MPI2Send-1 18 19
MPI2Send-1 20 21
```

```
MPI2Send-2 22 23
MPI2Send-2 24 25

-Non-blocking data transfer
MPI2Send-2 26 27
MPI2Send-2 28 29
MPI2Send-2 30 31 32

-Collective data transfer
MPI3Coll-1 1 2 3
MPI3Coll-1 4 5 6 7
MPI3Coll-1 8 9 10
MPI3Coll-1 11 12 13
MPI3Coll-1 14 15
MPI3Coll-2 16 17 18

-Reduction operations
%(task 23 is solved in Section 1.2.5)
MPI3Coll-1 19 20 21 22
MPI3Coll-1 24 25 26
MPI3Coll-1 27 28

-Using the simplest derived datatypes
MPI4Type-1 1 2 3
MPI4Type-1 4 5 6
MPI4Type-1 7 8

-Packed data transfer
MPI4Type-1 9 10 11
MPI4Type-1 12 13

-More complex derived datatypes
%(task 14 is solved in Section 1.2.6)
MPI4Type-2 15 16
MPI4Type-2 17 18
MPI4Type-2 19 20

-Collective function MPI_Alltoallw (MPI-2)
MPI4Type-3 21 22

-Creating new communicators
%(task 3 is solved in Section 1.2.7)
MPI5Comm-1 1 2
MPI5Comm-1 4 5
MPI5Comm-1 6 7
MPI5Comm-1 8 9
MPI5Comm-1 10 11 12
```

```
-Virtual topologies  
%(tasks 17, 29 are solved in Sections 1.2.8, 1.2.9)  
MPI5Comm-1 13 14  
MPI5Comm-1 15 16 18  
MPI5Comm-1 19 20 21 22  
MPI5Comm-1 23 24  
MPI5Comm-1 25 26 27  
MPI5Comm-3 28 30  
  
-Distributed graph topology (MPI-2)  
MPI5Comm-3 31 32  
  
-Non-blocking collective functions (MPI-3)  
MPI5Comm-2 33 35 37  
MPI5Comm-2 34 36 38  
MPI5Comm-2 39 40 41  
MPI5Comm-2 42 43 44  
MPI5Comm-2 45 46 47  
  
-Local file input-output (MPI-2)  
MPI6File-1 1 2 3 4  
MPI6File-1 5 6 7 8  
  
-Collective file input-output (MPI-2)  
MPI6File-1 9 10  
MPI6File-2 11 12  
MPI6File-1 13 14  
MPI6File-2 15 16  
  
-Setting up the file view (MPI-2)  
%(task 26 is solved in Section 1.3.3)  
MPI6File-1 17 20  
MPI6File-1 18 19  
MPI6File-2 21 22  
MPI6File-3 23 24 25  
MPI6File-2 27 28  
MPI6File-3 29 30  
  
-One-side communications (MPI-2)  
%(task 13 is solved in Section 1.3.5)  
MPI7Win-1 1 2  
MPI7Win-1 3 4  
MPI7Win-1 5 6  
MPI7Win-2 7 8 9 10  
MPI7Win-2 11 12  
MPI7Win-2 14 15  
MPI7Win-3 16 17
```

```
-Additional synchronization types (MPI-2)
%(task 23 is solved in Section 1.3.6)
MPI7Win-1 18 19 20
MPI7Win-2 21 22
MPI7Win-1 24 25 26
MPI7Win-2 27 29
MPI7Win-2 28 30

-Creation of inter-communicators (MPI-2)
%(task 9 is solved in Section 1.3.7)
MPI8Inter-1 1 2
MPI8Inter-2 3 4
MPI8Inter-2 5 6
MPI8Inter-3 7 8

-Collective operations for inter-communicators (MPI-2)
MPI8Inter-2 10 11 12
MPI8Inter-2 13 14

-Dynamic process creation (MPI-2)
%(task 15 is solved in Section 1.3.8)
MPI8Inter-1 16
MPI8Inter-2 17 18
MPI8Inter-2 19 20
MPI8Inter-3 21 22

%Parallel matrix algorithms (task 1 is solved in Section 1.4.2)

-Defining new datatypes and communicators
MPI9Matr-1 11 21 32
MPI9Matr-1 22 33 34

-Source data scattering stage
%(task 2 is solved in Section 1.4.3)
MPI9Matr-2 12 23 35

-Computation stage
%(task 24 is solved in Section 1.4.4)
MPI9Matr-1 3 13 36 37
MPI9Matr-1 4 14 25 38
MPI9Matr-2 5 15 26 39

-Result gathering stage
MPI9Matr-2 6 16 27 40

-Full implementation of the algorithm
MPI9Matr-3 7 17 28 41
```

```

-File input-output (MPI-2)
%(task 19 is solved in Section 1.4.5)
MPI9Matr-2 8 18 29 42
MPI9Matr-2 9 30 43
MPI9Matr-3 10 20 31 44

```

To form a set of tasks covering all the capabilities of MPI discussed in the book, it is sufficient to select one task from each series of similar tasks. Summary information on the resulting sets of tasks is given in Table 4.

Table 4

Summary of task sets

Task group	Number of tasks	Total points
Part 1	24	30
MPI1Proc	3	3
MPI2Send	12	17
MPI3Coll	9	10
Part 2	21	30
MPI4Type	9	14
MPI5Comm	12	16
Part 3	39	70
MPI5Comm (MPI-3)	5	10
MPI6File	12	20
MPI7Win	12	20
MPI8Inter	10	20
Part 4	11	20
MPI9Matr	11	20
Total	95	150

Each set can be divided into 4 parts. The first part includes tasks from the groups MPI1Proc, MPI2Send, and MPI3Coll, which are devoted to the basic capabilities of MPI related to sending data between processes. The second part includes tasks devoted to additional capabilities of MPI (mainly the standard 1.1): defining new datatypes and new communicators, including those related to virtual topologies. The third part contains tasks related to new capabilities of the MPI-2 and MPI-3 standard, and the fourth part is devoted to matrix algorithms.

Depending on the number of hours allocated for laboratory classes and the MPI topics covered, individual assignment sets may include only some of the specified parts.

3.2.2. Set of 24 variants of tasks

Table 5 contains a set of 24 variants of tasks generated from a groups of similar tasks (see Section 3.2.1). The PTVarMaker program from the **Teacher Pack** system was used to create the variants. The **Teacher Pack** system is one

of the extensions of the Programming Taskbook and includes utilities designed to manage and control group laboratory classes. The description of the **Teacher Pack** system and, in particular, of the PTVarMaker program is presented on the ptaskbook.com website (the **Teacher Pack** section), as well as in [3].

If more than one point is given for a task, the number of points is indicated after the task number in square brackets. If the number of points is not indicated after the task number, then it is considered to be equal to 1.

Table 5

24 variants of tasks

VARIANT 1

- MPI1Proc Processes and their ranks: 1, 4, 9
- MPI2Send(1) Blocking data transfer: 3, 6, 8, 12, 17, 18, 20, 22[2], 25[2]
- MPI2Send(2) Non-blocking data transfer: 26[2], 28[2], 30[2]
- MPI3Coll(1) Collective data transfer: 2, 7, 9, 11, 14, 17[2]
- MPI3Coll(2) Reduction operations: 20, 24, 27
- MPI4Type(1) Using the simplest derived datatypes: 2, 5, 7
- MPI4Type(2) Packed data transfer: 9, 12
- MPI4Type(3) More complex derived datatypes: 16[2], 18[2], 19[2]
- MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]
- MPI5Comm(1) Creating new communicators: 1, 4, 7, 9, 11
- MPI5Comm(2) Virtual topologies: 13, 15, 20, 23, 27, 30[3]
- MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
- MPI5Comm(4) Non-blocking collective functions (MPI-3): 37[2], 34[2], 40[2], 44[2], 46[2]
- MPI6File(1) Local file input-output (MPI-2): 3, 6
- MPI6File(2) Collective file input-output (MPI-2): 9, 12[2], 13, 16[2]
- MPI6File(3) Setting up the file view (MPI-2): 17, 18, 22[2], 24[3], 27[2], 29[3]
- MPI7Win(1) One-side communications (MPI-2): 1, 4, 5, 7[2], 11[2], 15[2], 17[3]
- MPI7Win(2) Additional synchronization types (MPI-2): 20, 21[2], 25, 29[2], 28[2]
- MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 4[2], 6[2], 7[3]
- MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 12[2], 14[2]
- MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 19[2], 22[3]
- MPI9Matr(1) Defining new datatypes and communicators: 32, 22
- MPI9Matr(2) Source data scattering stage: 12[2]
- MPI9Matr(3) Computation stage: 13, 14, 15[2]
- MPI9Matr(4) Result gathering stage: 27[2]
- MPI9Matr(5) Full implementation of the algorithm: 41[3]
- MPI9Matr(6) File input-output (MPI-2): 8[2], 9[2], 10[3]

VARIANT 2

- MPI1Proc Processes and their ranks: 1, 7, 8
- MPI2Send(1) Blocking data transfer: 3, 5, 9, 14, 15, 19, 20, 22[2], 25[2]
- MPI2Send(2) Non-blocking data transfer: 27[2], 28[2], 32[2]
- MPI3Coll(1) Collective data transfer: 3, 4, 8, 12, 15, 18[2]
- MPI3Coll(2) Reduction operations: 20, 24, 27
- MPI4Type(1) Using the simplest derived datatypes: 1, 6, 7
- MPI4Type(2) Packed data transfer: 10, 12
- MPI4Type(3) More complex derived datatypes: 15[2], 18[2], 20[2]
- MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]

MPI5Comm(1) Creating new communicators: 2, 4, 7, 9, 11
 MPI5Comm(2) Virtual topologies: 13, 16, 19, 24, 26, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 37[2], 34[2], 39[2], 44[2], 47[2]
 MPI6File(1) Local file input-output (MPI-2): 4, 7
 MPI6File(2) Collective file input-output (MPI-2): 10, 11[2], 13, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 18, 22[2], 23[3], 28[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 3, 6, 9[2], 12[2], 14[2], 16[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 20, 22[2], 26, 27[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 4[2], 6[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 12[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 19[2], 22[3]
 MPI9Matr(1) Defining new datatypes and communicators: 32, 34
 MPI9Matr(2) Source data scattering stage: 35[2]
 MPI9Matr(3) Computation stage: 37, 25, 26[2]
 MPI9Matr(4) Result gathering stage: 16[2]
 MPI9Matr(5) Full implementation of the algorithm: 28[3]
 MPI9Matr(6) File input-output (MPI-2): 29[2], 43[2], 20[3]

VARIANT 3

MPI1Proc Processes and their ranks: 1, 5, 10
 MPI2Send(1) Blocking data transfer: 2, 6, 8, 14, 15, 18, 20, 22[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 29[2], 32[2]
 MPI3Coll(1) Collective data transfer: 1, 6, 9, 13, 15, 17[2]
 MPI3Coll(2) Reduction operations: 19, 26, 27
 MPI4Type(1) Using the simplest derived datatypes: 3, 5, 8
 MPI4Type(2) Packed data transfer: 11, 13
 MPI4Type(3) More complex derived datatypes: 16[2], 18[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 2, 4, 6, 9, 10
 MPI5Comm(2) Virtual topologies: 13, 18, 22, 23, 25, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 33[2], 38[2], 39[2], 43[2], 47[2]
 MPI6File(1) Local file input-output (MPI-2): 1, 5
 MPI6File(2) Collective file input-output (MPI-2): 10, 11[2], 13, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 19, 21[2], 24[3], 27[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 4, 5, 9[2], 12[2], 15[2], 16[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 19, 21[2], 25, 27[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 3[2], 5[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 11[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 19[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 21, 22
 MPI9Matr(2) Source data scattering stage: 35[2]
 MPI9Matr(3) Computation stage: 3, 25, 39[2]
 MPI9Matr(4) Result gathering stage: 6[2]
 MPI9Matr(5) Full implementation of the algorithm: 28[3]
 MPI9Matr(6) File input-output (MPI-2): 8[2], 9[2], 44[3]

VARIANT 4

MPI1Proc Processes and their ranks: 1, 5, 9

MPI2Send(1) Blocking data transfer: 1, 5, 8, 12, 16, 19, 20, 22[2], 25[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 29[2], 30[2]
 MPI3Coll(1) Collective data transfer: 2, 6, 8, 12, 15, 16[2]
 MPI3Coll(2) Reduction operations: 22, 25, 28
 MPI4Type(1) Using the simplest derived datatypes: 2, 6, 8
 MPI4Type(2) Packed data transfer: 9, 12
 MPI4Type(3) More complex derived datatypes: 15[2], 17[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 2, 5, 7, 8, 11
 MPI5Comm(2) Virtual topologies: 13, 15, 21, 24, 25, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 33[2], 38[2], 41[2], 43[2], 46[2]
 MPI6File(1) Local file input-output (MPI-2): 3, 6
 MPI6File(2) Collective file input-output (MPI-2): 9, 11[2], 14, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 19, 22[2], 25[3], 27[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 3, 6, 10[2], 11[2], 15[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 18, 22[2], 24, 29[2], 28[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 3[2], 5[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 11[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 20[2], 22[3]
 MPI9Matr(1) Defining new datatypes and communicators: 21, 33
 MPI9Matr(2) Source data scattering stage: 23[2]
 MPI9Matr(3) Computation stage: 36, 4, 15[2]
 MPI9Matr(4) Result gathering stage: 16[2]
 MPI9Matr(5) Full implementation of the algorithm: 7[3]
 MPI9Matr(6) File input-output (MPI-2): 8[2], 43[2], 44[3]

VARIANT 5

MPI1Proc Processes and their ranks: 1, 4, 8
 MPI2Send(1) Blocking data transfer: 4, 7, 10, 13, 17, 19, 21, 22[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 29[2], 32[2]
 MPI3Coll(1) Collective data transfer: 1, 5, 8, 12, 15, 17[2]
 MPI3Coll(2) Reduction operations: 19, 24, 27
 MPI4Type(1) Using the simplest derived datatypes: 1, 4, 7
 MPI4Type(2) Packed data transfer: 10, 12
 MPI4Type(3) More complex derived datatypes: 15[2], 18[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 1, 4, 7, 9, 12
 MPI5Comm(2) Virtual topologies: 14, 16, 22, 24, 26, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 33[2], 36[2], 40[2], 42[2], 45[2]
 MPI6File(1) Local file input-output (MPI-2): 2, 8
 MPI6File(2) Collective file input-output (MPI-2): 9, 11[2], 14, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 19, 21[2], 24[3], 27[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 3, 5, 7[2], 12[2], 14[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 18, 21[2], 26, 27[2], 28[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 3[2], 6[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 12[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 20[2], 21[3]

MPI9Matr(1) Defining new datatypes and communicators: 32, 34
 MPI9Matr(2) Source data scattering stage: 12[2]
 MPI9Matr(3) Computation stage: 36, 4, 39[2]
 MPI9Matr(4) Result gathering stage: 6[2]
 MPI9Matr(5) Full implementation of the algorithm: 17[3]
 MPI9Matr(6) File input-output (MPI-2): 42[2], 30[2], 44[3]

VARIANT 6

MPI1Proc Processes and their ranks: 3, 7, 8
 MPI2Send(1) Blocking data transfer: 4, 7, 10, 13, 17, 18, 21, 23[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 28[2], 31[2]
 MPI3Coll(1) Collective data transfer: 1, 7, 9, 11, 14, 17[2]
 MPI3Coll(2) Reduction operations: 22, 26, 28
 MPI4Type(1) Using the simplest derived datatypes: 2, 4, 8
 MPI4Type(2) Packed data transfer: 10, 13
 MPI4Type(3) More complex derived datatypes: 15[2], 18[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]
 MPI5Comm(1) Creating new communicators: 2, 5, 6, 9, 10
 MPI5Comm(2) Virtual topologies: 14, 15, 22, 24, 27, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 35[2], 36[2], 41[2], 43[2], 46[2]
 MPI6File(1) Local file input-output (MPI-2): 4, 6
 MPI6File(2) Collective file input-output (MPI-2): 10, 12[2], 13, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 18, 21[2], 23[3], 28[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 4, 5, 10[2], 11[2], 14[2], 16[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 19, 21[2], 25, 29[2], 28[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 3[2], 5[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 10[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 20[2], 22[3]
 MPI9Matr(1) Defining new datatypes and communicators: 21, 33
 MPI9Matr(2) Source data scattering stage: 35[2]
 MPI9Matr(3) Computation stage: 36, 14, 15[2]
 MPI9Matr(4) Result gathering stage: 40[2]
 MPI9Matr(5) Full implementation of the algorithm: 28[3]
 MPI9Matr(6) File input-output (MPI-2): 18[2], 30[2], 20[3]

VARIANT 7

MPI1Proc Processes and their ranks: 3, 6, 9
 MPI2Send(1) Blocking data transfer: 3, 6, 9, 13, 15, 19, 21, 23[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 28[2], 31[2]
 MPI3Coll(1) Collective data transfer: 1, 4, 10, 11, 15, 16[2]
 MPI3Coll(2) Reduction operations: 19, 25, 28
 MPI4Type(1) Using the simplest derived datatypes: 1, 4, 7
 MPI4Type(2) Packed data transfer: 11, 13
 MPI4Type(3) More complex derived datatypes: 15[2], 18[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 2, 4, 7, 8, 12
 MPI5Comm(2) Virtual topologies: 13, 16, 21, 23, 27, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 35[2], 34[2], 40[2], 42[2], 45[2]

MPI6File(1) Local file input-output (MPI-2): 3, 8
 MPI6File(2) Collective file input-output (MPI-2): 9, 11[2], 14, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 18, 21[2], 25[3], 28[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 4, 5, 9[2], 11[2], 14[2], 16[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 19, 22[2], 24, 29[2], 28[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 3[2], 5[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 10[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 19[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 11, 34
 MPI9Matr(2) Source data scattering stage: 23[2]
 MPI9Matr(3) Computation stage: 37, 25, 26[2]
 MPI9Matr(4) Result gathering stage: 16[2]
 MPI9Matr(5) Full implementation of the algorithm: 7[3]
 MPI9Matr(6) File input-output (MPI-2): 29[2], 43[2], 10[3]

VARIANT 8

MPI1Proc Processes and their ranks: 1, 6, 10
 MPI2Send(1) Blocking data transfer: 1, 6, 10, 12, 15, 18, 20, 23[2], 25[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 29[2], 31[2]
 MPI3Coll(1) Collective data transfer: 2, 4, 10, 13, 14, 18[2]
 MPI3Coll(2) Reduction operations: 21, 25, 27
 MPI4Type(1) Using the simplest derived datatypes: 1, 4, 8
 MPI4Type(2) Packed data transfer: 11, 12
 MPI4Type(3) More complex derived datatypes: 16[2], 17[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]
 MPI5Comm(1) Creating new communicators: 2, 5, 6, 8, 12
 MPI5Comm(2) Virtual topologies: 14, 15, 21, 24, 26, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 35[2], 38[2], 40[2], 42[2], 45[2]
 MPI6File(1) Local file input-output (MPI-2): 1, 5
 MPI6File(2) Collective file input-output (MPI-2): 9, 12[2], 13, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 18, 21[2], 23[3], 28[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 3, 5, 8[2], 12[2], 14[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 18, 22[2], 25, 29[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 3[2], 5[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 11[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 19[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 11, 33
 MPI9Matr(2) Source data scattering stage: 12[2]
 MPI9Matr(3) Computation stage: 13, 38, 5[2]
 MPI9Matr(4) Result gathering stage: 40[2]
 MPI9Matr(5) Full implementation of the algorithm: 41[3]
 MPI9Matr(6) File input-output (MPI-2): 18[2], 9[2], 20[3]

VARIANT 9

MPI1Proc Processes and their ranks: 3, 6, 10
 MPI2Send(1) Blocking data transfer: 2, 5, 10, 14, 16, 18, 21, 22[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 29[2], 31[2]
 MPI3Coll(1) Collective data transfer: 2, 5, 9, 13, 14, 16[2]
 MPI3Coll(2) Reduction operations: 20, 25, 27

MPI4Type(1) Using the simplest derived datatypes: 3, 6, 7
 MPI4Type(2) Packed data transfer: 10, 12
 MPI4Type(3) More complex derived datatypes: 16[2], 17[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]
 MPI5Comm(1) Creating new communicators: 1, 5, 7, 8, 11
 MPI5Comm(2) Virtual topologies: 14, 18, 20, 24, 25, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 35[2], 36[2], 39[2], 43[2], 45[2]
 MPI6File(1) Local file input-output (MPI-2): 2, 7
 MPI6File(2) Collective file input-output (MPI-2): 10, 12[2], 13, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 19, 22[2], 23[3], 28[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 3, 6, 8[2], 11[2], 15[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 18, 21[2], 24, 29[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 4[2], 6[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 10[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 20[2], 22[3]
 MPI9Matr(1) Defining new datatypes and communicators: 11, 34
 MPI9Matr(2) Source data scattering stage: 23[2]
 MPI9Matr(3) Computation stage: 13, 4, 5[2]
 MPI9Matr(4) Result gathering stage: 27[2]
 MPI9Matr(5) Full implementation of the algorithm: 41[3]
 MPI9Matr(6) File input-output (MPI-2): 29[2], 9[2], 10[3]

VARIANT 10

MPI1Proc Processes and their ranks: 3, 7, 8
 MPI2Send(1) Blocking data transfer: 4, 7, 8, 12, 16, 19, 21, 23[2], 25[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 28[2], 32[2]
 MPI3Coll(1) Collective data transfer: 3, 7, 8, 13, 15, 18[2]
 MPI3Coll(2) Reduction operations: 21, 26, 28
 MPI4Type(1) Using the simplest derived datatypes: 3, 5, 8
 MPI4Type(2) Packed data transfer: 9, 13
 MPI4Type(3) More complex derived datatypes: 16[2], 17[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 1, 5, 6, 8, 10
 MPI5Comm(2) Virtual topologies: 14, 18, 20, 23, 27, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 37[2], 34[2], 41[2], 42[2], 46[2]
 MPI6File(1) Local file input-output (MPI-2): 4, 5
 MPI6File(2) Collective file input-output (MPI-2): 9, 12[2], 14, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 19, 22[2], 25[3], 28[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 3, 6, 8[2], 12[2], 15[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 20, 22[2], 24, 27[2], 28[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 4[2], 5[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 12[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 20[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 21, 33
 MPI9Matr(2) Source data scattering stage: 23[2]
 MPI9Matr(3) Computation stage: 37, 38, 5[2]
 MPI9Matr(4) Result gathering stage: 6[2]

MPI9Matr(5) Full implementation of the algorithm: 17[3]
 MPI9Matr(6) File input-output (MPI-2): 42[2], 30[2], 31[3]

VARIANT 11

MPI1Proc Processes and their ranks: 3, 4, 9
 MPI2Send(1) Blocking data transfer: 1, 7, 9, 13, 16, 18, 21, 23[2], 25[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 29[2], 30[2]
 MPI3Coll(1) Collective data transfer: 3, 5, 10, 11, 14, 16[2]
 MPI3Coll(2) Reduction operations: 22, 26, 28
 MPI4Type(1) Using the simplest derived datatypes: 2, 5, 7
 MPI4Type(2) Packed data transfer: 9, 13
 MPI4Type(3) More complex derived datatypes: 16[2], 17[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]
 MPI5Comm(1) Creating new communicators: 1, 4, 6, 8, 12
 MPI5Comm(2) Virtual topologies: 14, 18, 19, 23, 25, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 37[2], 38[2], 41[2], 44[2], 47[2]
 MPI6File(1) Local file input-output (MPI-2): 2, 8
 MPI6File(2) Collective file input-output (MPI-2): 10, 12[2], 14, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 19, 21[2], 24[3], 27[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 4, 6, 7[2], 12[2], 14[2], 16[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 19, 22[2], 26, 27[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 4[2], 6[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 10[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 19[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 32, 22
 MPI9Matr(2) Source data scattering stage: 35[2]
 MPI9Matr(3) Computation stage: 3, 38, 39[2]
 MPI9Matr(4) Result gathering stage: 40[2]
 MPI9Matr(5) Full implementation of the algorithm: 17[3]
 MPI9Matr(6) File input-output (MPI-2): 42[2], 43[2], 31[3]

VARIANT 12

MPI1Proc Processes and their ranks: 3, 5, 10
 MPI2Send(1) Blocking data transfer: 2, 5, 9, 14, 17, 19, 20, 23[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 28[2], 30[2]
 MPI3Coll(1) Collective data transfer: 3, 6, 10, 12, 14, 18[2]
 MPI3Coll(2) Reduction operations: 21, 24, 28
 MPI4Type(1) Using the simplest derived datatypes: 3, 6, 8
 MPI4Type(2) Packed data transfer: 11, 13
 MPI4Type(3) More complex derived datatypes: 15[2], 17[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 1, 5, 6, 9, 10
 MPI5Comm(2) Virtual topologies: 13, 16, 19, 23, 26, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 33[2], 36[2], 39[2], 44[2], 47[2]
 MPI6File(1) Local file input-output (MPI-2): 1, 7
 MPI6File(2) Collective file input-output (MPI-2): 10, 11[2], 14, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 18, 22[2], 25[3], 27[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 4, 6, 10[2], 11[2], 15[2], 16[3]

MPI7Win(2) Additional synchronization types (MPI-2): 20, 21[2], 26, 27[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 4[2], 6[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 11[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 20[2], 22[3]
 MPI9Matr(1) Defining new datatypes and communicators: 11, 22
 MPI9Matr(2) Source data scattering stage: 12[2]
 MPI9Matr(3) Computation stage: 3, 14, 26[2]
 MPI9Matr(4) Result gathering stage: 27[2]
 MPI9Matr(5) Full implementation of the algorithm: 7[3]
 MPI9Matr(6) File input-output (MPI-2): 18[2], 30[2], 31[3]

VARIANT 13

MPI1Proc Processes and their ranks: 3, 6, 10
 MPI2Send(1) Blocking data transfer: 1, 5, 8, 12, 15, 18, 20, 22[2], 25[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 28[2], 31[2]
 MPI3Coll(1) Collective data transfer: 3, 7, 8, 13, 15, 16[2]
 MPI3Coll(2) Reduction operations: 21, 25, 27
 MPI4Type(1) Using the simplest derived datatypes: 2, 5, 8
 MPI4Type(2) Packed data transfer: 10, 12
 MPI4Type(3) More complex derived datatypes: 15[2], 17[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]
 MPI5Comm(1) Creating new communicators: 1, 4, 7, 8, 12
 MPI5Comm(2) Virtual topologies: 13, 16, 22, 24, 27, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 33[2], 34[2], 40[2], 42[2], 45[2]
 MPI6File(1) Local file input-output (MPI-2): 3, 5
 MPI6File(2) Collective file input-output (MPI-2): 9, 11[2], 13, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 18, 22[2], 24[3], 27[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 4, 5, 9[2], 11[2], 15[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 18, 21[2], 26, 27[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 3[2], 6[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 11[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 20[2], 22[3]
 MPI9Matr(1) Defining new datatypes and communicators: 11, 33
 MPI9Matr(2) Source data scattering stage: 12[2]
 MPI9Matr(3) Computation stage: 37, 14, 5[2]
 MPI9Matr(4) Result gathering stage: 40[2]
 MPI9Matr(5) Full implementation of the algorithm: 7[3]
 MPI9Matr(6) File input-output (MPI-2): 42[2], 30[2], 20[3]

VARIANT 14

MPI1Proc Processes and their ranks: 3, 5, 8
 MPI2Send(1) Blocking data transfer: 4, 7, 10, 14, 17, 19, 20, 23[2], 25[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 29[2], 32[2]
 MPI3Coll(1) Collective data transfer: 1, 7, 10, 11, 15, 18[2]
 MPI3Coll(2) Reduction operations: 22, 26, 28
 MPI4Type(1) Using the simplest derived datatypes: 1, 4, 7
 MPI4Type(2) Packed data transfer: 11, 12
 MPI4Type(3) More complex derived datatypes: 16[2], 18[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]

MPI5Comm(1) Creating new communicators: 2, 4, 6, 9, 12
 MPI5Comm(2) Virtual topologies: 14, 16, 21, 24, 27, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 37[2], 36[2], 41[2], 43[2], 47[2]
 MPI6File(1) Local file input-output (MPI-2): 4, 8
 MPI6File(2) Collective file input-output (MPI-2): 9, 11[2], 13, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 18, 21[2], 24[3], 27[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 4, 6, 8[2], 12[2], 14[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 20, 21[2], 25, 27[2], 28[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 3[2], 5[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 10[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 19[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 32, 34
 MPI9Matr(2) Source data scattering stage: 35[2]
 MPI9Matr(3) Computation stage: 36, 14, 15[2]
 MPI9Matr(4) Result gathering stage: 27[2]
 MPI9Matr(5) Full implementation of the algorithm: 28[3]
 MPI9Matr(6) File input-output (MPI-2): 8[2], 30[2], 44[3]

VARIANT 15

MPI1Proc Processes and their ranks: 1, 7, 9
 MPI2Send(1) Blocking data transfer: 4, 7, 9, 12, 16, 18, 21, 23[2], 25[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 29[2], 32[2]
 MPI3Coll(1) Collective data transfer: 2, 4, 9, 13, 15, 17[2]
 MPI3Coll(2) Reduction operations: 19, 24, 27
 MPI4Type(1) Using the simplest derived datatypes: 1, 6, 8
 MPI4Type(2) Packed data transfer: 11, 12
 MPI4Type(3) More complex derived datatypes: 16[2], 18[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]
 MPI5Comm(1) Creating new communicators: 1, 4, 6, 8, 11
 MPI5Comm(2) Virtual topologies: 14, 16, 20, 23, 26, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 35[2], 38[2], 39[2], 42[2], 47[2]
 MPI6File(1) Local file input-output (MPI-2): 1, 6
 MPI6File(2) Collective file input-output (MPI-2): 9, 11[2], 14, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 18, 21[2], 25[3], 28[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 3, 5, 9[2], 11[2], 14[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 18, 21[2], 26, 27[2], 28[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 3[2], 5[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 11[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 19[2], 22[3]
 MPI9Matr(1) Defining new datatypes and communicators: 21, 34
 MPI9Matr(2) Source data scattering stage: 12[2]
 MPI9Matr(3) Computation stage: 3, 25, 39[2]
 MPI9Matr(4) Result gathering stage: 27[2]
 MPI9Matr(5) Full implementation of the algorithm: 7[3]
 MPI9Matr(6) File input-output (MPI-2): 42[2], 9[2], 10[3]

VARIANT 16

MPI1Proc Processes and their ranks: 3, 4, 10

MPI2Send(1) Blocking data transfer: 1, 7, 10, 14, 16, 19, 21, 22[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 28[2], 30[2]
 MPI3Coll(1) Collective data transfer: 2, 5, 9, 13, 14, 18[2]
 MPI3Coll(2) Reduction operations: 20, 24, 28
 MPI4Type(1) Using the simplest derived datatypes: 2, 5, 8
 MPI4Type(2) Packed data transfer: 10, 12
 MPI4Type(3) More complex derived datatypes: 15[2], 17[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 2, 4, 7, 8, 12
 MPI5Comm(2) Virtual topologies: 14, 18, 22, 24, 27, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 33[2], 38[2], 40[2], 42[2], 45[2]
 MPI6File(1) Local file input-output (MPI-2): 4, 5
 MPI6File(2) Collective file input-output (MPI-2): 9, 11[2], 14, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 18, 21[2], 23[3], 27[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 3, 6, 8[2], 12[2], 14[2], 16[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 19, 22[2], 24, 29[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 4[2], 6[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 12[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 19[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 32, 22
 MPI9Matr(2) Source data scattering stage: 35[2]
 MPI9Matr(3) Computation stage: 3, 25, 39[2]
 MPI9Matr(4) Result gathering stage: 40[2]
 MPI9Matr(5) Full implementation of the algorithm: 17[3]
 MPI9Matr(6) File input-output (MPI-2): 8[2], 43[2], 10[3]

VARIANT 17

MPI1Proc Processes and their ranks: 1, 4, 10
 MPI2Send(1) Blocking data transfer: 2, 5, 8, 13, 17, 19, 20, 22[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 28[2], 32[2]
 MPI3Coll(1) Collective data transfer: 1, 7, 8, 12, 15, 16[2]
 MPI3Coll(2) Reduction operations: 21, 25, 28
 MPI4Type(1) Using the simplest derived datatypes: 2, 6, 7
 MPI4Type(2) Packed data transfer: 9, 12
 MPI4Type(3) More complex derived datatypes: 15[2], 17[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 1, 5, 7, 9, 11
 MPI5Comm(2) Virtual topologies: 13, 18, 19, 24, 26, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 35[2], 38[2], 40[2], 44[2], 46[2]
 MPI6File(1) Local file input-output (MPI-2): 2, 6
 MPI6File(2) Collective file input-output (MPI-2): 10, 12[2], 13, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 19, 22[2], 23[3], 28[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 3, 6, 10[2], 11[2], 15[2], 16[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 20, 21[2], 26, 27[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 3[2], 5[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 10[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 19[2], 22[3]

MPI9Matr(1) Defining new datatypes and communicators: 11, 34
 MPI9Matr(2) Source data scattering stage: 23[2]
 MPI9Matr(3) Computation stage: 36, 4, 26[2]
 MPI9Matr(4) Result gathering stage: 6[2]
 MPI9Matr(5) Full implementation of the algorithm: 17[3]
 MPI9Matr(6) File input-output (MPI-2): 29[2], 30[2], 31[3]

VARIANT 18

MPI1Proc Processes and their ranks: 1, 7, 8
 MPI2Send(1) Blocking data transfer: 3, 5, 10, 12, 16, 18, 21, 22[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 28[2], 30[2]
 MPI3Coll(1) Collective data transfer: 3, 6, 8, 13, 15, 16[2]
 MPI3Coll(2) Reduction operations: 19, 24, 27
 MPI4Type(1) Using the simplest derived datatypes: 1, 6, 8
 MPI4Type(2) Packed data transfer: 11, 13
 MPI4Type(3) More complex derived datatypes: 16[2], 18[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 1, 5, 7, 9, 10
 MPI5Comm(2) Virtual topologies: 14, 18, 19, 23, 27, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 33[2], 34[2], 39[2], 43[2], 45[2]
 MPI6File(1) Local file input-output (MPI-2): 2, 7
 MPI6File(2) Collective file input-output (MPI-2): 10, 12[2], 13, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 19, 21[2], 25[3], 27[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 4, 6, 7[2], 11[2], 14[2], 16[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 19, 21[2], 24, 29[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 4[2], 5[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 12[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 20[2], 22[3]
 MPI9Matr(1) Defining new datatypes and communicators: 32, 22
 MPI9Matr(2) Source data scattering stage: 12[2]
 MPI9Matr(3) Computation stage: 13, 4, 5[2]
 MPI9Matr(4) Result gathering stage: 16[2]
 MPI9Matr(5) Full implementation of the algorithm: 28[3]
 MPI9Matr(6) File input-output (MPI-2): 18[2], 43[2], 20[3]

VARIANT 19

MPI1Proc Processes and their ranks: 1, 6, 10
 MPI2Send(1) Blocking data transfer: 2, 6, 9, 12, 17, 18, 21, 23[2], 25[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 28[2], 31[2]
 MPI3Coll(1) Collective data transfer: 1, 6, 9, 11, 14, 17[2]
 MPI3Coll(2) Reduction operations: 19, 25, 28
 MPI4Type(1) Using the simplest derived datatypes: 3, 6, 8
 MPI4Type(2) Packed data transfer: 10, 13
 MPI4Type(3) More complex derived datatypes: 16[2], 18[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]
 MPI5Comm(1) Creating new communicators: 2, 4, 6, 8, 11
 MPI5Comm(2) Virtual topologies: 14, 15, 20, 24, 25, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 35[2], 34[2], 41[2], 44[2], 47[2]

MPI6File(1) Local file input-output (MPI-2): 2, 7
 MPI6File(2) Collective file input-output (MPI-2): 10, 11[2], 13, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 19, 22[2], 24[3], 27[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 4, 6, 10[2], 12[2], 15[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 19, 22[2], 24, 29[2], 28[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 3[2], 6[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 10[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 20[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 32, 22
 MPI9Matr(2) Source data scattering stage: 23[2]
 MPI9Matr(3) Computation stage: 13, 14, 5[2]
 MPI9Matr(4) Result gathering stage: 6[2]
 MPI9Matr(5) Full implementation of the algorithm: 41[3]
 MPI9Matr(6) File input-output (MPI-2): 29[2], 30[2], 44[3]

VARIANT 20

MPI1Proc Processes and their ranks: 1, 7, 9
 MPI2Send(1) Blocking data transfer: 4, 6, 9, 13, 16, 18, 20, 23[2], 25[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 28[2], 31[2]
 MPI3Coll(1) Collective data transfer: 3, 5, 8, 11, 14, 16[2]
 MPI3Coll(2) Reduction operations: 21, 26, 27
 MPI4Type(1) Using the simplest derived datatypes: 3, 5, 8
 MPI4Type(2) Packed data transfer: 9, 13
 MPI4Type(3) More complex derived datatypes: 15[2], 18[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]
 MPI5Comm(1) Creating new communicators: 2, 4, 7, 9, 10
 MPI5Comm(2) Virtual topologies: 13, 15, 21, 23, 25, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 37[2], 38[2], 39[2], 42[2], 47[2]
 MPI6File(1) Local file input-output (MPI-2): 1, 8
 MPI6File(2) Collective file input-output (MPI-2): 9, 12[2], 14, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 18, 22[2], 25[3], 28[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 4, 5, 8[2], 12[2], 14[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 19, 21[2], 25, 29[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 4[2], 5[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 11[2], 14[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 20[2], 22[3]
 MPI9Matr(1) Defining new datatypes and communicators: 21, 33
 MPI9Matr(2) Source data scattering stage: 23[2]
 MPI9Matr(3) Computation stage: 3, 38, 39[2]
 MPI9Matr(4) Result gathering stage: 40[2]
 MPI9Matr(5) Full implementation of the algorithm: 41[3]
 MPI9Matr(6) File input-output (MPI-2): 8[2], 9[2], 31[3]

VARIANT 21

MPI1Proc Processes and their ranks: 3, 4, 8
 MPI2Send(1) Blocking data transfer: 2, 6, 9, 13, 15, 18, 21, 22[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 29[2], 31[2]
 MPI3Coll(1) Collective data transfer: 2, 5, 10, 12, 14, 17[2]
 MPI3Coll(2) Reduction operations: 22, 24, 27

MPI4Type(1) Using the simplest derived datatypes: 3, 4, 7
 MPI4Type(2) Packed data transfer: 10, 12
 MPI4Type(3) More complex derived datatypes: 15[2], 17[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 1, 5, 6, 8, 11
 MPI5Comm(2) Virtual topologies: 13, 15, 21, 23, 26, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 33[2], 34[2], 40[2], 43[2], 46[2]
 MPI6File(1) Local file input-output (MPI-2): 3, 5
 MPI6File(2) Collective file input-output (MPI-2): 9, 12[2], 13, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 18, 21[2], 24[3], 27[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 3, 5, 7[2], 11[2], 15[2], 16[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 18, 22[2], 24, 27[2], 28[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 3[2], 5[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 12[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 19[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 11, 33
 MPI9Matr(2) Source data scattering stage: 23[2]
 MPI9Matr(3) Computation stage: 13, 25, 26[2]
 MPI9Matr(4) Result gathering stage: 16[2]
 MPI9Matr(5) Full implementation of the algorithm: 28[3]
 MPI9Matr(6) File input-output (MPI-2): 29[2], 43[2], 31[3]

VARIANT 22

MPI1Proc Processes and their ranks: 3, 5, 9
 MPI2Send(1) Blocking data transfer: 3, 7, 10, 14, 15, 19, 21, 23[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 29[2], 32[2]
 MPI3Coll(1) Collective data transfer: 2, 6, 10, 12, 14, 18[2]
 MPI3Coll(2) Reduction operations: 22, 25, 28
 MPI4Type(1) Using the simplest derived datatypes: 3, 4, 7
 MPI4Type(2) Packed data transfer: 11, 13
 MPI4Type(3) More complex derived datatypes: 15[2], 18[2], 19[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 2, 5, 6, 8, 12
 MPI5Comm(2) Virtual topologies: 13, 16, 22, 24, 26, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 35[2], 36[2], 41[2], 43[2], 46[2]
 MPI6File(1) Local file input-output (MPI-2): 4, 6
 MPI6File(2) Collective file input-output (MPI-2): 10, 12[2], 14, 16[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 19, 21[2], 25[3], 28[2], 30[3]
 MPI7Win(1) One-side communications (MPI-2): 2, 3, 5, 7[2], 12[2], 15[2], 17[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 20, 22[2], 25, 29[2], 30[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 1, 4[2], 6[2], 8[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 12[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 18[2], 20[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 11, 33
 MPI9Matr(2) Source data scattering stage: 35[2]
 MPI9Matr(3) Computation stage: 36, 38, 26[2]
 MPI9Matr(4) Result gathering stage: 16[2]

MPI9Matr(5) Full implementation of the algorithm: 41[3]
 MPI9Matr(6) File input-output (MPI-2): 42[2], 43[2], 44[3]

VARIANT 23

MPI1Proc Processes and their ranks: 1, 5, 8
 MPI2Send(1) Blocking data transfer: 1, 5, 8, 13, 15, 19, 20, 22[2], 24[2]
 MPI2Send(2) Non-blocking data transfer: 26[2], 29[2], 30[2]
 MPI3Coll(1) Collective data transfer: 1, 4, 9, 11, 14, 17[2]
 MPI3Coll(2) Reduction operations: 20, 26, 27
 MPI4Type(1) Using the simplest derived datatypes: 1, 4, 7
 MPI4Type(2) Packed data transfer: 9, 13
 MPI4Type(3) More complex derived datatypes: 16[2], 17[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 21[3]
 MPI5Comm(1) Creating new communicators: 1, 5, 6, 9, 10
 MPI5Comm(2) Virtual topologies: 13, 15, 19, 23, 25, 28[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 31[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 37[2], 36[2], 39[2], 44[2], 45[2]
 MPI6File(1) Local file input-output (MPI-2): 1, 7
 MPI6File(2) Collective file input-output (MPI-2): 10, 11[2], 14, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 17, 19, 22[2], 23[3], 28[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 3, 5, 9[2], 11[2], 14[2], 16[3]
 MPI7Win(2) Additional synchronization types (MPI-2): 18, 22[2], 25, 27[2], 28[2]
 MPI8Inter(1) Creation of inter-communicators (MPI-2): 2, 4[2], 6[2], 7[3]
 MPI8Inter(2) Collective operations for inter-communicators (MPI-2): 10[2], 13[2]
 MPI8Inter(3) Dynamic process creation (MPI-2): 16, 17[2], 19[2], 21[3]
 MPI9Matr(1) Defining new datatypes and communicators: 21, 22
 MPI9Matr(2) Source data scattering stage: 12[2]
 MPI9Matr(3) Computation stage: 37, 4, 15[2]
 MPI9Matr(4) Result gathering stage: 6[2]
 MPI9Matr(5) Full implementation of the algorithm: 7[3]
 MPI9Matr(6) File input-output (MPI-2): 18[2], 9[2], 10[3]

VARIANT 24

MPI1Proc Processes and their ranks: 3, 6, 9
 MPI2Send(1) Blocking data transfer: 3, 6, 8, 14, 17, 19, 20, 23[2], 25[2]
 MPI2Send(2) Non-blocking data transfer: 27[2], 29[2], 30[2]
 MPI3Coll(1) Collective data transfer: 3, 4, 10, 12, 15, 18[2]
 MPI3Coll(2) Reduction operations: 20, 26, 28
 MPI4Type(1) Using the simplest derived datatypes: 2, 5, 7
 MPI4Type(2) Packed data transfer: 9, 13
 MPI4Type(3) More complex derived datatypes: 16[2], 17[2], 20[2]
 MPI4Type(4) Collective function MPI_Alltoallw (MPI-2): 22[3]
 MPI5Comm(1) Creating new communicators: 2, 5, 7, 9, 10
 MPI5Comm(2) Virtual topologies: 14, 18, 20, 23, 25, 30[3]
 MPI5Comm(3) Distributed graph topology (MPI-2): 32[3]
 MPI5Comm(4) Non-blocking collective functions (MPI-3): 37[2], 36[2], 41[2], 44[2], 46[2]
 MPI6File(1) Local file input-output (MPI-2): 3, 8
 MPI6File(2) Collective file input-output (MPI-2): 10, 12[2], 14, 15[2]
 MPI6File(3) Setting up the file view (MPI-2): 20, 19, 22[2], 23[3], 28[2], 29[3]
 MPI7Win(1) One-side communications (MPI-2): 1, 4, 6, 10[2], 12[2], 15[2], 16[3]

MPI7Win(2)	Additional synchronization types (MPI-2): 20, 22[2], 26, 29[2], 28[2]
MPI8Inter(1)	Creation of inter-communicators (MPI-2): 2, 4[2], 6[2], 8[3]
MPI8Inter(2)	Collective operations for inter-communicators (MPI-2): 11[2], 13[2]
MPI8Inter(3)	Dynamic process creation (MPI-2): 16, 17[2], 20[2], 22[3]
MPI9Matr(1)	Defining new datatypes and communicators: 21, 34
MPI9Matr(2)	Source data scattering stage: 35[2]
MPI9Matr(3)	Computation stage: 37, 38, 15[2]
MPI9Matr(4)	Result gathering stage: 27[2]
MPI9Matr(5)	Full implementation of the algorithm: 17[3]
MPI9Matr(6)	File input-output (MPI-2): 18[2], 9[2], 20[3]

References

1. *Abramyan M. E.* Workshop on parallel programming using the Programming Taskbook for MPI. Rostov-on-Don: SFedU Publishing House, 2010. 172 p. (In Russian.)
2. *Abramyan M. E.* Introduction to the C++ Standard Template Library. Description, examples of use, training tasks. Rostov-On-Don; Taganrog: SFedU Publishing House, 2017. 178 p. (In Russian.)
3. *Abramyan M. E.* Tools and methods for developing electronic educational resources in computer science. Rostov-on-Don; Taganrog: SFedU Publishing House, 2018. 259 p. (In Russian.)
4. *Abramyan M. E., Steinberg B. Ya., Steinberg O. B.* Technologies of program parallelization. MPI and OpenMP, loop vectorization, memory usage optimization. Rostov-on-Don: SFedU Publishing House, 2014. 148 p. (In Russian.)
5. *Antonov A. S.* Parallel programming technologies MPI and OpenMP. Moscow: MSU Publishing House, 2012. 344 p. (In Russian.)
6. *Borzunov S. V., Kurgalin S. D., Flegel A. V.* Practical training in parallel programming. St. Petersburg: BHV, 2017. 236 p. (In Russian.)
7. *Bukatov A. A., Datsyuk V. N., Zhegulo A. N.* Programming for multiprocessor computing systems. Rostov-on-Don: CVVR Publishing, 2003. 208 p. (In Russian.)
8. *Korneev V. D.* Parallel programming in MPI. Novosibirsk: Publishing house of the Institute of Computational Mathematics and Mathematical Geophysics SB RAS, 2002. 215 p. (In Russian.)
9. *Nemnyugin S. A., Stesik O. L.* Parallel programming for multiprocessor computing systems. St. Petersburg: BHV-Petersburg, 2002. 396 p. (In Russian.)
10. *Shpakovsky G. I., Serikova N. V.* Programming for multiprocessor systems in the MPI standard. Minsk: BSU, 2002. 323 p. (In Russian.)
11. *MPI: A Message-Passing Interface Standard.* Version 1.1: June, 1995. Message Passing Interface Forum, 2003. 238 p. [Electronic resource] URL: <http://mpi-forum.org/docs/mpi-1.1/mpi1-report.pdf> (date accessed: 24.10.2024).
12. *MPI: A Message-Passing Interface Standard.* Version 2.2. Message Passing Interface Forum, 2009. 647 p. [Electronic resource] URL: <http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf> (date accessed: 24.10.2024).
13. *MPI: A Message-Passing Interface Standard.* Version 3.2. Message Passing Interface Forum, 2015. 636 p. [Electronic resource] URL: <http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf> (date accessed: 24.10.2024).

Index

The index contains all the functions, types, and constants of the MPI library described in this textbook, as well as the tools of the Programming Taskbook and Standard Template Library used in it. The "Task solutions" group lists all the tasks from the Programming Taskbook for MPI-2 that were solved in Section 1.

mpi.h, 14, 15, 176
mpi.lib and mpich.lib, 15, 176
MPI_2INT, 32
MPI_Accumulate, 106, 113, 119
MPI_Aint, 57
MPI_Allgather, 47
MPI_Allgatherv, 47
MPI_Allreduce, 51
MPI_Alltoall, 48
MPI_Alltoallv, 48
MPI_Alltoallw, 48
MPI_ANY_SOURCE, 33
MPI_ANY_TAG, 33, 171
MPI_BAND, 50
MPI_Barrier, 45
MPI_Bcast, 45, 129, 136
MPI_BOR, 50
MPI_Bsend, 31
MPI_Bsend_init, 44
MPI_BSEND_OVERHEAD, 34
MPI_Buffer_attach, 33
MPI_Buffer_detach, 34
MPI_BXOR, 50
MPI_BYTE, 32, 103
MPI_CART, 76
MPI_Cart_coords, 79
MPI_Cart_create, 77
MPI_Cart_get, 84
MPI_Cart_rank, 79, 210
MPI_Cart_shift, 85
MPI_Cart_sub, 81, 82
MPI_Cartdim_get, 84
MPI_CHAR, 32
MPI_Close_port, 139
MPI_Comm, 12
MPI_Comm_accept, 138
MPI_Comm_compare, 69
MPI_Comm_connect, 138
MPI_Comm_create, 70, 128
MPI_Comm_disconnect, 139
MPI_Comm_dup, 69, 125
MPI_Comm_free, 72, 139
MPI_Comm_get_parent, 131
MPI_Comm_group, 70, 122
MPI_COMM_NULL, 11, 70, 72, 128, 131
MPI_Comm_rank, 16, 122
MPI_Comm_remote_group, 122
MPI_Comm_remote_size, 122, 126
MPI_COMM_SELF, 11
MPI_Comm_set_errhandler, 210
MPI_Comm_size, 16, 122
MPI_Comm_spawn, 130
MPI_Comm_spawn_multiple, 130, 131
MPI_Comm_split, 73, 123, 128
MPI_Comm_test_inter, 122
MPI_COMM_WORLD, 11, 132
MPI_CONGRUENT, 69
MPI_Datatype, 32
MPI_DATATYPE_NULL, 58
MPI_Dims_create, 84
MPI_DIST_GRAPH, 77
MPI_Dist_graph_create, 92
MPI_Dist_graph_create_adjacent, 92
MPI_Dist_graph_neighbors, 94
MPI_Dist_graph_neighbors_count, 94
MPI_DOUBLE, 32
MPI_DOUBLE_INT, 32, 200
MPI_ERRCODES_IGNORE, 131
MPI_Errhandler_set, 210
MPI_ERROR, 33, 97
MPI_ERROR_RETURN, 210
MPI_Exscan, 52, 220
MPI_File, 96, 101
MPI_File_close, 102

MPI_File_get_byte_offset, 98
MPI_File_get_position, 97
MPI_File_get_position_shared, 97
MPI_File_get_size, 98
MPI_File_iread, 96
MPI_File_iread_at, 96
MPI_File_iread_shared, 96
MPI_File_iwrite, 96
MPI_File_iwrite_at, 96
MPI_File_iwrite_shared, 96
MPI_FILE_NULL, 102
MPI_File_open, 101
MPI_File_read, 96, 97
MPI_File_read_all, 96, 97
MPI_File_read_all_begin, 96
MPI_File_read_all_end, 96
MPI_File_read_at, 96
MPI_File_read_at_all, 96
MPI_File_read_at_all_begin, 96
MPI_File_read_at_all_end, 96
MPI_File_read_ordered, 96, 97, 98
MPI_File_read_ordered_begin, 96
MPI_File_read_ordered_end, 96
MPI_File_read_shared, 96, 97
MPI_File_seek, 97
MPI_File_seek_shared, 97
MPI_File_set_size, 98
MPI_File_set_view, 102
MPI_File_write, 96, 97
MPI_File_write_all, 96, 97, 104
MPI_File_write_all_begin, 96
MPI_File_write_all_end, 96
MPI_File_write_at, 96
MPI_File_write_at_all, 96
MPI_File_write_at_all_begin, 96
MPI_File_write_at_all_end, 96
MPI_File_write_ordered, 96, 97, 98
MPI_File_write_ordered_begin, 96
MPI_File_write_ordered_end, 96
MPI_File_write_shared, 96, 97
MPI_Finalize, 17, 176
MPI_Finalized, 17
MPI_FLOAT, 32
MPI_Gather, 46, 74, 129
MPI_Gatherv, 46
MPI_Get, 106, 118
MPI_Get_count, 33, 97
MPI_GRAPH, 76
MPI_Graph_create, 87
MPI_Graph_get, 90
MPI_Graph_neighbors, 91
MPI_Graph_neighbors_count, 91
MPI_Graphdims_get, 90
MPI_Group, 70
MPI_Group_compare, 70
MPI_Group_difference, 72
MPI_GROUP_EMPTY, 70, 72
MPI_Group_excl, 71
MPI_Group_free, 72
MPI_Group_incl, 71
MPI_Group_intersection, 72
MPI_GROUP_NULL, 70, 72
MPI_Group_range_excl, 71
MPI_Group_range_incl, 71
MPI_Group_rank, 70
MPI_Group_size, 70
MPI_Group_translate_ranks, 70
MPI_Group_union, 72
MPI_Iallgather, 49, 217
MPI_Iallgatherv, 217
MPI_Iallreduce, 52, 218
MPI_Ialltoall, 49, 217
MPI_Ialltoallv, 217
MPI_Ialltoallw, 49
MPI_Ibarrier, 49
MPI_Ibcast, 49, 215
MPI_Ibsend, 42
MPI_IDENT, 69, 71
MPI_Iexscan, 52, 219
MPI_Igather, 49, 215
MPI_Igatherv, 216
MPI_Info, 102
MPI_INFO_NULL, 93, 102, 106, 138
MPI_Init, 16, 131, 176
MPI_Initialized, 16
MPI_INT, 32
MPI_Intercomm_create, 125
MPI_Intercomm_merge, 137
MPI_Iprobe, 44
MPI_Irecv, 42
MPI_Ireduce, 52, 218
MPI_Ireduce_scatter, 52, 218
MPI_Ireduce_scatter_block, 52, 219
MPI_Irsend, 42
MPI_Iscan, 52, 219
MPI_Iscatter, 49, 216
MPI_Iscatterv, 216
MPI_Isend, 42

- MPI_Issend, 42
- MPI_LAND, 50
- MPI_LB, 59
- MPI_LOCK_EXCLUSIVE, 110
- MPI_LOCK_SHARED, 110
- MPI_LONG, 32
- MPI_Lookup_name, 138
- MPI_LOR, 50
- MPI_LXOR, 50
- MPI_MAX, 49
- MPI_MAXLOC, 50, 53
- MPI_MIN, 49, 119
- MPI_MINLOC, 50, 53
- MPI_MODE_APPEND, 102
- MPI_MODE_CREATE, 102
- MPI_MODE_DELETE_ON_CLOSE, 102
- MPI_MODE_NOPUT, 108
- MPI_MODE_NOSUCCEED, 109
- MPI_MODE_RDONLY, 102
- MPI_MODE_RDWR, 102
- MPI_MODE_WRONLY, 102
- MPI_Offset, 97
- MPI_Op, 49, 107
- MPI_Op_create, 50
- MPI_Open_port, 138
- MPI_Pack, 60
- MPI_Pack_size, 61
- MPI_PACKED, 60
- MPI_Probe, 33
- MPI_PROC_NULL, 85, 129, 135
- MPI_PROD, 49
- MPI_Publish_name, 138
- MPI_Put, 106
- MPI_Recv, 32, 127
- MPI_Recv_init, 44
- MPI_Reduce, 50, 135
- MPI_Reduce_local, 52
- MPI_Reduce_scatter, 51
- MPI_Reduce_scatter_block, 51
- MPI_Request, 42, 49, 52
- MPI_Request_free, 44
- MPI_REQUEST_NULL, 42
- MPI_ROOT, 129, 135, 136
- MPI_Rsend, 31
- MPI_Rsend_init, 44
- MPI_Scan, 51
- MPI_Scatter, 47, 81, 129
- MPI_Scatterv, 47
- MPI_SEEK_CUR, 97
- MPI_SEEK_END, 97
- MPI_SEEK_SET, 97
- MPI_Send, 31, 127
- MPI_Send_init, 44
- MPI_Sendrecv, 34, 91
- MPI_Sendrecv_replace, 34
- MPI_SIMILAR, 69, 71
- MPI_SOURCE, 33
- MPI_Ssend, 31
- MPI_Ssend_init, 44
- MPI_Start, 44
- MPI_Startall, 44
- MPI_Status, 33, 97, 171
- MPI_STATUS_IGNORE, 33
- MPI_SUCCESS, 17
- MPI_SUM, 49, 114
- MPI_TAG, 33
- MPI_TAG_UB, 31
- MPI_Test, 43
- MPI_Testall, 43
- MPI_Testany, 43
- MPI_Testsome, 43
- MPI_Topo_test, 76
- MPI_Type_commit, 58
- MPI_Type_contiguous, 56
- MPI_Type_create_hindexed, 57
- MPI_Type_create_hvector, 57
- MPI_Type_create_indexed_block, 57
- MPI_Type_create_resized, 59, 200
- MPI_Type_create_struct, 57, 200
- MPI_Type_extent, 58
- MPI_Type_free, 58
- MPI_Type_get_extent, 60
- MPI_Type_hindexed, 57
- MPI_Type_hvector, 57
- MPI_Type_indexed, 57
- MPI_Type_size, 58
- MPI_Type_struct, 57
- MPI_Type_vector, 56
- MPI_UB, 59
- MPI_UNDEFINED, 70, 77, 124
- MPI_UNEQUAL, 69, 71
- MPI_Unpack, 60
- MPI_Unpublish_name, 139
- MPI_UNWEIGHTED, 93
- MPI_User_function, 50
- MPI_Wait, 42
- MPI_Waitall, 43
- MPI_Waitany, 43

- MPI_Waitsome, 43
- MPI_Win, 106
- MPI_Win_complete, 109, 118
- MPI_Win_create, 106, 113, 117
- MPI_Win_fence, 108, 113
- MPI_Win_free, 106, 120
- MPI_Win_lock, 110
- MPI_Win_post, 109, 117
- MPI_Win_start, 109, 117
- MPI_Win_test, 109
- MPI_Win_unlock, 110
- MPI_Win_wait, 109, 118
- MPI_Wtick, 45
- MPI_Wtime, 44
- PT for MPI-2 taskbook
 - GetExename, 239
 - HideTask, 169, 284
 - pt, 23, 278
 - pt4.h, 14, 15, 179, 278
 - pt4console.h, 180, 277
 - pt4null.h, 179
 - ptin_iterator<T>, 67, 279
 - ptout_iterator<T>, 68, 279
 - SetPrecision, 284
 - Show, 29, 282
 - ShowAll, 181
 - ShowLine, 29, 282
 - Task, 15, 278
- STL library
 - <algorithm>, 67
 - <vector>, 67
 - assign, 147
 - begin, 68, 148
 - copy, 68, 75, 148
 - end, 68, 75, 148
 - vector<T>, 66, 75, 147
- Task solutions
 - MPI1Proc2, 11
 - MPI2Send11, 35
 - MPI3Coll23, 52
 - MPI4Type14, 61
 - MPI5Comm3, 72
 - MPI5Comm17, 77
 - MPI5Comm29, 86
 - MPI6File26, 99
 - MPI7Win13, 111
 - MPI7Win23, 115
 - MPI8Inter9, 122
 - MPI8Inter15, 132
 - MPI9Matr1, 143
 - MPI9Matr2, 148
 - MPI9Matr19, 159
 - MPI9Matr24, 153