Ministry of Education and Science of the Russian Federation

Federal State Autonomous
Educational Institution of Higher Education
"SOUTHERN FEDERAL UNIVERSITY"

M. E. Abramyan

# Parallel Programming Based on MPI 2.0 Technology

*Textbook on the course*
*" Parallel Programming"*
*for students of the directions 02.03.02 and 02.04.02*
*"Fundamental informatics and information technology"*
*(bachelor's and master's degrees)*

Rostov-on-Don - Taganrog
Southern Federal University Publishing House
2018

UDC 004.42(075.8)

BBK 32.973я73

A 13

Published by decision of the educational and methodological commission
of the I.I. Vorovich Institute of Mathematics, Mechanics and Computer Science of the
Southern Federal University (minutes No. 3 of March 20, 2018)

Reviewers:

Doctor of Physical and Mathematical Sciences, Professor,
Head of the Department of Digital Technologies, Faculty of Computer Science, Voronezh State University
**S. D. Kurgalin**

Doctor of Technical Sciences, Professor of the Department of Computer Science
of the Rostov State Transport University (RSTU)
**M. A. Butakova**

*The textbook was prepared and published with the support of the Scholarship Program of the V. Potanin Charitable Foundation as part of the project "Electronic Problem Book on Parallel MPI Programming" — one of the winning projects of the grant competition for teachers of the Master's program in 2016–2017.*

**Abramyan, M.E.**

Parallel Programming Based on MPI 2.0 Technology: textbook / M. E. Abramyan. - Rostov-on-Don; Taganrog: Publishing house of Southern Federal University, 2018. - 357 p.

ISBN

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 means of MPI technology are used. The description is accompanied by solutions of typical tasks. The second section contains 250 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 describes the constructor of tasks on parallel MPI-programming for the electronic problem book. The fourth section gives an overview of the capabilities of the Programming Taskbook for MPI-2 and provides a set of 24 variants of individual tasks, 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.

For undergraduate and graduate students studying in the field of "Fundamental Informatics and Information Technology".

UDC 004

BBK 32.973

ISBN

# Table of contents

# Preface

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

The textbook consists of three main sections. The first section provides a systematic description of the capabilities of the MPI interface. The basic capabilities of this technology included in all versions of the standard 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 intercommunicators and dynamic creation of processes. 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 the implementation of which various means of MPI technology are used. All topics discussed are accompanied by examples of program code associated with solving typical tasks.

The second section contains 250 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 debugging tools provided in integrated environments for parallel programs. In order to facilitate the development of MPI technology, the author has developed a specialized training system–an electronic problem book on parallel programming on MPI-2 **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 environments Microsoft Visual Studio (version 2008 and higher) and Code::Blocks (version 13 and higher). It is the presence of a large number of training tasks related to all aspects of MPI technology and the possibility of using specialized

software tools that significantly speed up the process of solving problems that are the distinctive features of this taskbook and the approach to teaching parallel programming based on it.

The third section describes the tools of the training task designer for the PT for MPI-2 taskbook, allowing the development of new groups of tasks in parallel MPI programming. The development of new groups of tasks can be a useful type of educational tasks of increased complexity, which can be offered to students, including as part of coursework and graduation projects.

The fourth, 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 index included in the textbook, which contains constants, types, and functions of the MPI interface, allows it to be used as a reference for MPI technology of the 1.1 and 2.x standards.

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–the new capabilities of the MPI-2 standard and parallel matrix algorithms. 150 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 electronic problem book PT for MPI-2, which replaced 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 universal electronic programming taskbook Programming Taskbook http://ptaskbook.com/.

# 1. MPI technology: description and examples of use

## *1.1. Introduction to MPI technology*

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

MPI technology (*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, and in 2012 the MPI 3.0 standard was released. 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 standard-2.

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 usually parallel programs using MPI technology 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 technology, it is quite sufficient to use a local computer, launching all the processes of a parallel application on it. In such a situation, one should not expect a significant gain in the speed of parallel algorithms, but with the help of such educational programs one can become familiar with the mechanisms of MPI 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 4.1.

PT for MPI-2 taskbook allows developing parallel programs in C++ using MPI technology for C. Additional capabilities of the C++ language are used mainly for more convenient organization of input-output (using streams and iterators—see Section 4.1.2), although in some situations other C++ tools are also useful, for example template functions (see tasks MPI2Send22–MPI2Send25 in Section 2.2.1). Since the PT for MPI-2 taskbook is a specialized extension for the universal programming problem book Programming Taskbook, it can be used together with all programming environments for the C++ language that the

basic taskbook supports. For version 4.17 of the basic Programming Taskbook, starting from which you can use the PT MPI-2 extension, environments Microsoft Visual Studio (version 2008 and above) and Code::Blocks (version 13 and above) are available.

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 programming environments.

However, to run parallel programs developed on the basis of MPI technology, 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. One of the popular freely distributed MPI support systems is the MPICH system, developed at the Argonne National Laboratory in the USA. The PT for MPI-2 taskbook can be used in conjunction with two versions of this system for Windows:

- MPICH 1.2.5 (ftp://ftp.mcs.anl.gov/pub/mpi/nt/mpich.nt.1.2.5.exe), supports the MPI 1.2 standard;
- MPICH 2 1.3 (http://www.mpich.org/static/downloads/1.3/mpich2-1.3-win-ia32.msi), supports the MPI 2.1 standard.

When using the MPICH 1.2.5 system, you can only perform those tasks that are intended for studying the MPI tools of the 1.1 standard. The MPICH2 1.3 system allows you to perform all the tasks included in the PT for MPI-2 taskbook.

> **Note:** To install MPICH 1.2.5, simply run the installation file and follow its instructions. The system is installed by default in the MPICH subdirectory of the Program Files directory for 32-bit programs: c:\Program Files (x86).
>
> To install the MPICH2 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. By default, the MPICH2 system is installed in the MPICH2 subdirectory of the Program Files directory for 32-bit programs.

> 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 MPICH system are installed, the basic version of the electronic problem book Programming Taskbook version not lower than 4.17 and electronic taskbook PT for MPI-2 should be installed (in the order specified). Installation programs for these problem books can be downloaded from the website of the electronic problem book ptaskbook.com (either in the section "Download" or on the main pages of the sections "Main" and "PT for MPI-2"). The main page of the section "PT for MPI-2" also contains links for downloading distributions of both versions of the MPICH system 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 versions of the MPICH system that are found on the computer will additionally appear (Fig. 1).

If there are two versions of MPICH, one will be active and the other (with a gray checkbox) will be temporarily disabled. To activate the other version of MPICH, 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 2017 environment is used when solving tasks, and MPICH2 1.3 is selected as the active version of the MPICH system.

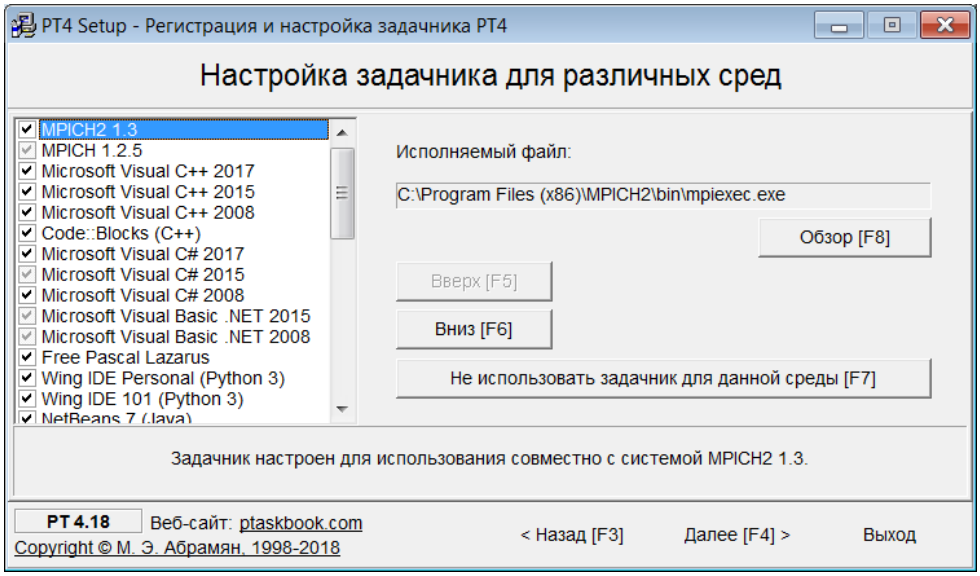


**Fig. 1**PT4Setup program window with a list of found IDEs

### 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 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 completing a task using the PT for MPI-2 taskbook starts with creating a project template for the selected task. All necessary libraries (associated with the taskbook and with the selected MPICH 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 designed to create a template. The easiest way to call this program is with the Load.lnk shortcut, which is automatically created in the working directory (by default, the working directory is called PT4Work and is located on the C drive). After starting the program, its window will appear on the screen (Fig. 2).

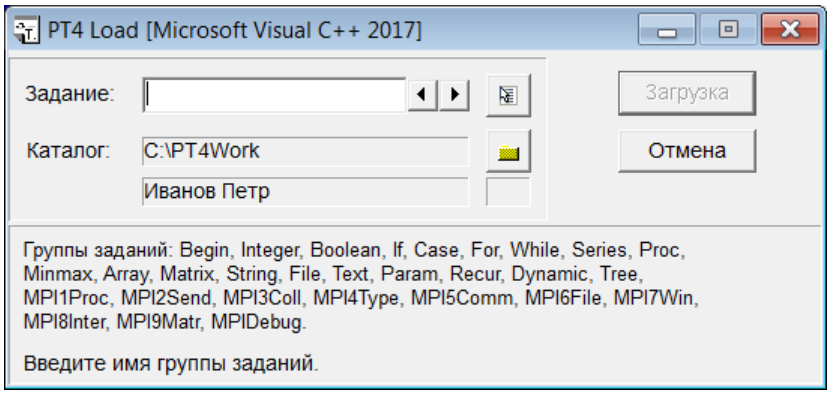


**Fig. 2.** Window of the PT4Load program

This is what the window looks like if the current IDE is Microsoft Visual Studio 2017 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, **Code::Blocks (C++))**; the name of the selected environment will appear in the window title.

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 MPICH 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.

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

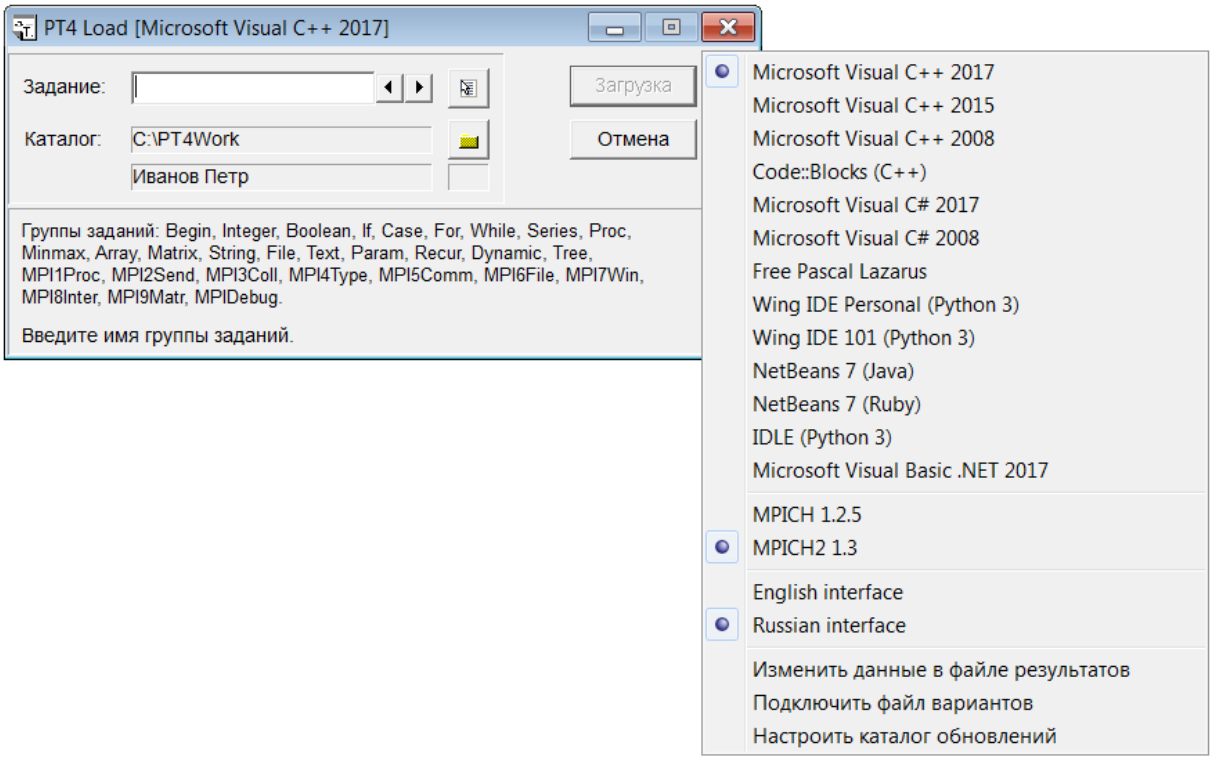


**Fig. 3.** PT4Load window with expanded pop-up menu

Let us select the required IDE, and then enter the text MPI1Proc2 in the **Task** field (it is not necessary to enter the full name of the group; it is enough to enter 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 number of tasks included in it will be given at the bottom of the window (Fig. 4).

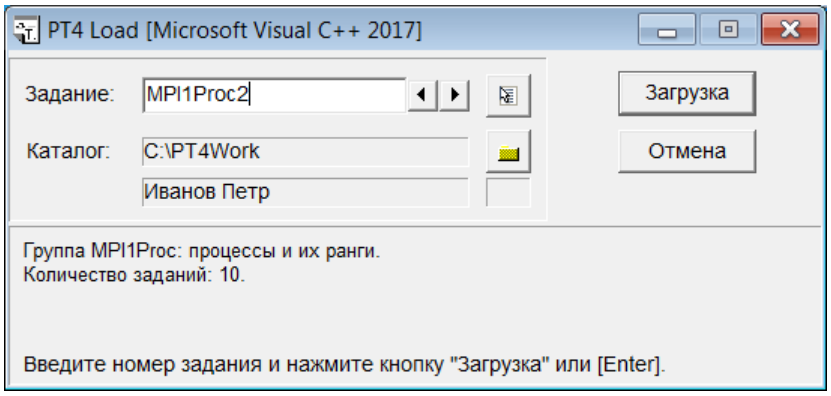


**Fig. 4.** The PT4Load window after entering the task name

Pressing the **Load** button or the [Enter] key, we will create a template 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, the name of 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; it is in this file that you must enter the solution of the task. Let us give the text of file MPI1Proc2.cpp:

```
#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 to 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 with the solution. 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 that the student does not have to type them anew each time, 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 statement for the Task function, which initializes the required task (see Section 4.1.2). This operator 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 (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 file pt4.cpp, 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 operators 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 system, a widely used free software implementation of the MPI standard for various operating systems, including 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 lib object file for the MPICH2 1.3 system is contained in the MPICH2\lib subdirectory and has the name mpi.lib , but the taskbook uses the name mpich.lib for this library, 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 environment: 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.

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 value different from zero; otherwise, the flag parameter is assumed to be zero. 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. For example, 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?")), 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. If the current process is not included in the comm communicator, then the value MPI_UNDEFINED is returned in the rank parameter. 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 4.1.3). Some MPI capabilities related to error handling are discussed in MPI5Comm23–24 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 (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 (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 MPICH system. In MPICH 1.2.5, it is named MPIRun.exe (and is contained in the MPICH\mpd\bin directory), in MPICH2 1.3, it is named mpiexec.exe (and is contained in the MPICH2\bin directory). 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.
>
> 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 IDEs (in particular, in Code::Blocks), 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 under test.

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 already 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.

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 (x86)\MPICH2\bin\mpiexec.exe"
    -nopopup_debug -localonly 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. The -nopopup_debug parameter disables the output of error messages

in a separate window (since these messages will eventually be displayed in the taskbook window), the -localonly parameter ensures that all instances of the process are launched on the local computer.

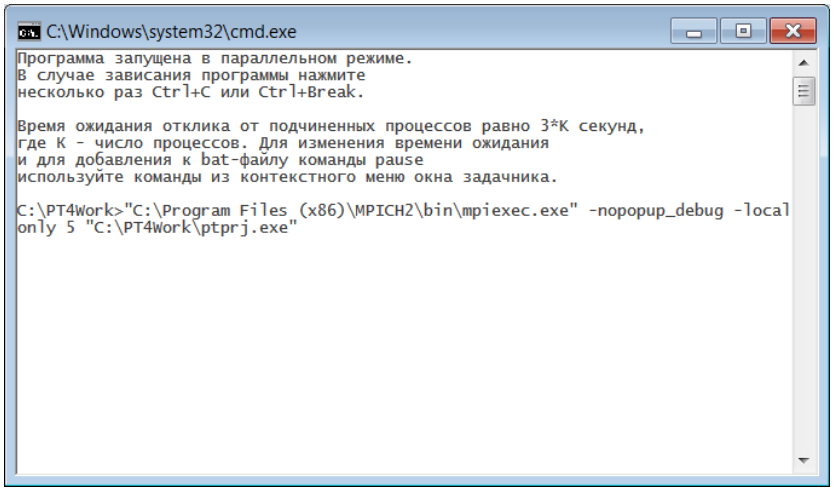


**Fig. 5**. Window with information about running the program in parallel mode

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** option.

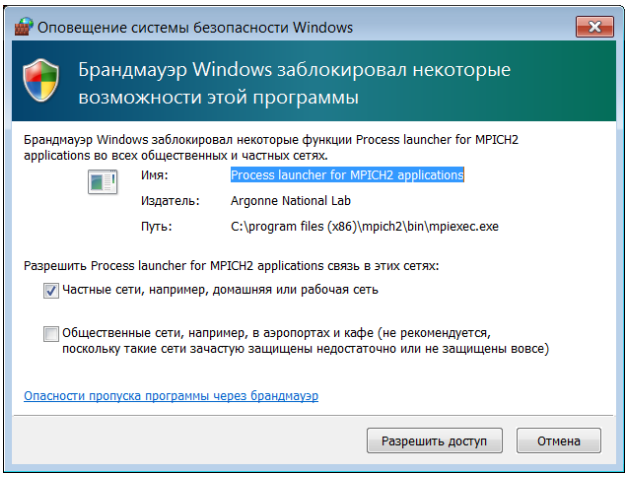


**Fig. 6.** Window with a request to block a running parallel program

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 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 (details are given in Section 3.1).

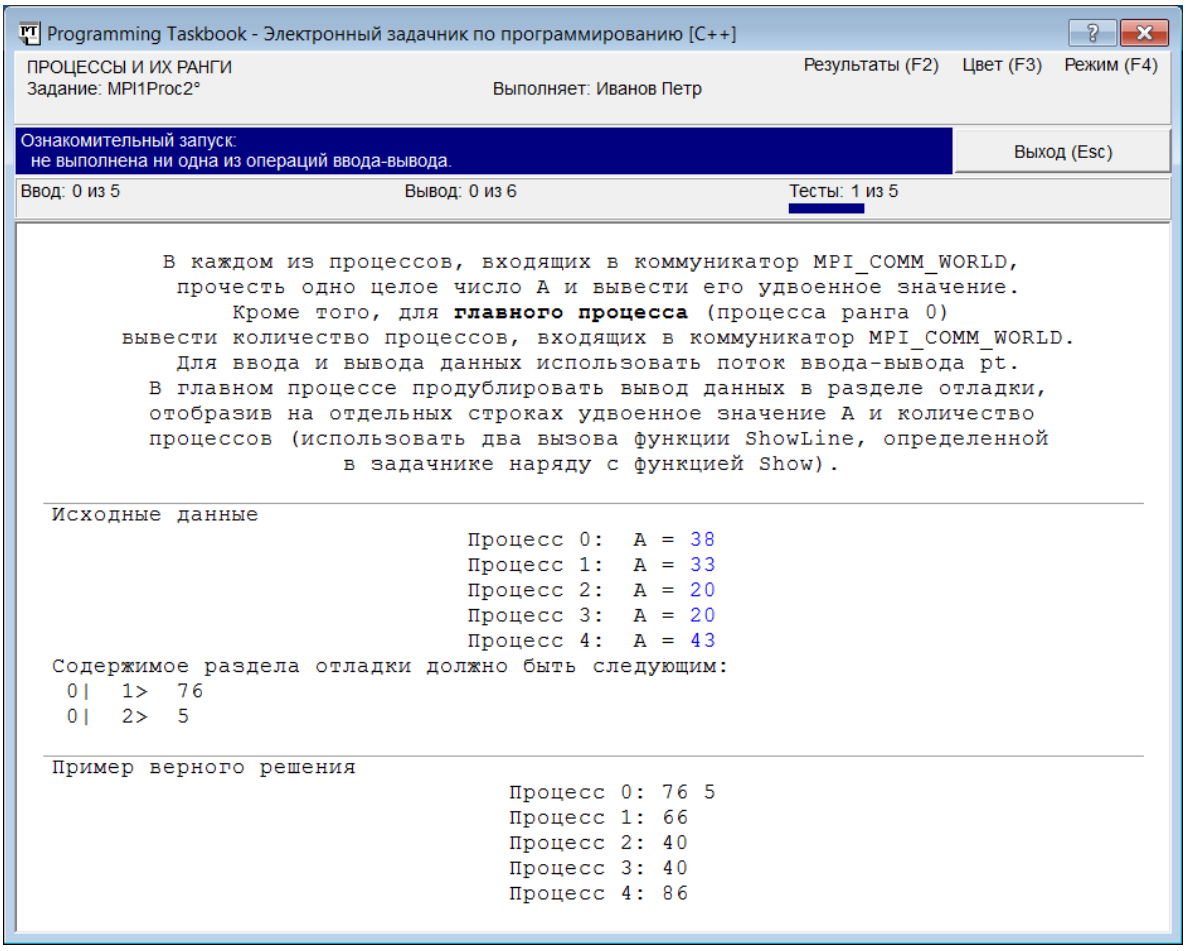


**Fig. 7.** Introductory run of MPI task 1 Proc 2

"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 discovered that the task belongs to the group of parallel tasks, it only creates a batch file $pt_run$.bat, writes comment lines and a command line to it that calls the program mpiexec.exe with the necessary parameters, and 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 out 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 console window header). If the starting copy of the program detects that the batch file has completed its work, and hung parallel program processes remain in memory, *it will automatically unload 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 [Crtl]+[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 very 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, al-

though the solution to the problem is correct. In such cases, you can increase the waiting time using the y command in the pop-up menu of the taskbook window **Increase the waiting time for a response from slave processes** (the menu also contains the command **Decrease the waiting time for a response from slave processes)**.

### 1.1.5. Executing MPI1Proc2 task

Let us return to our task. Now that we have become familiar with the mechanism of the program's operation in parallel mode, solving this simple 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 4.1.2). This stream allows you to input data of any scalar types, 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.*"

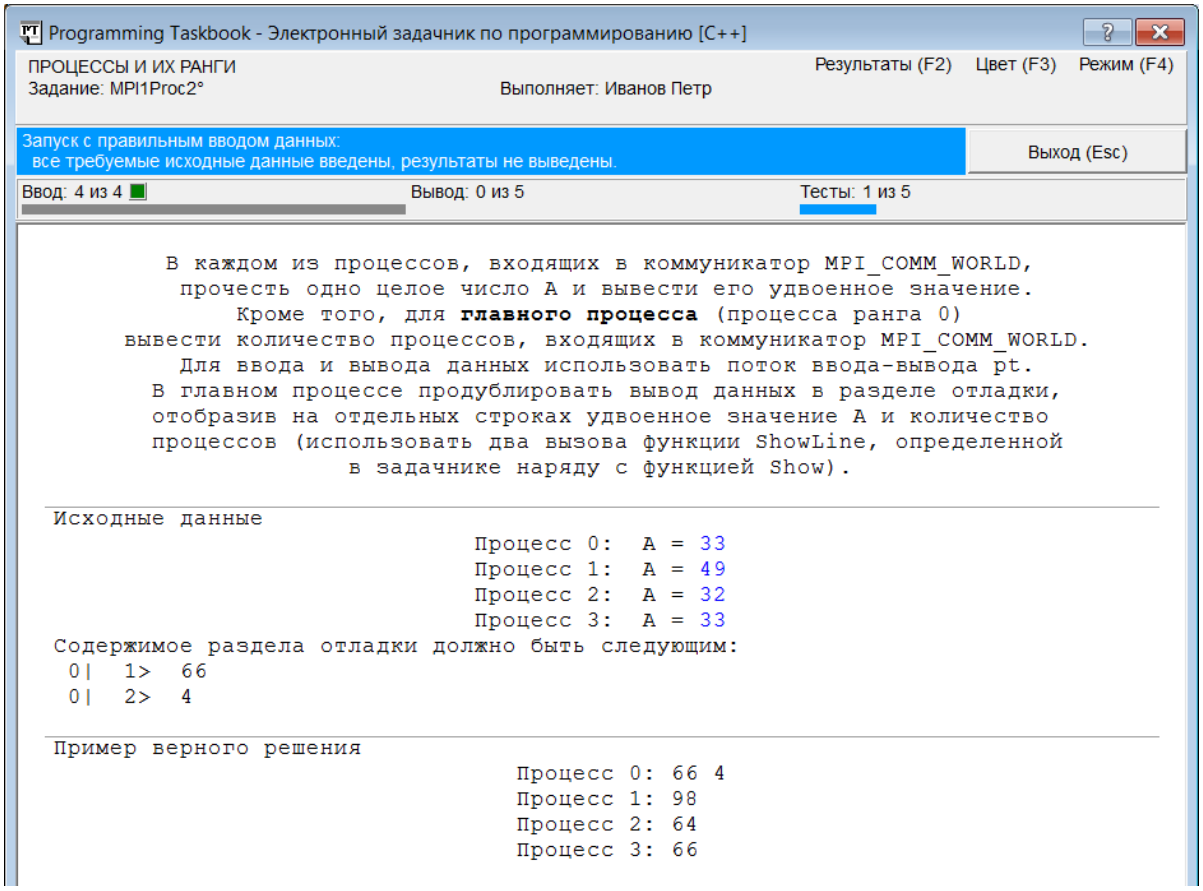


**Fig. 8**taskbook window with information about the correct input of the initial data

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'll add the following statement to the end of the Solve function:

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

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 occured 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.

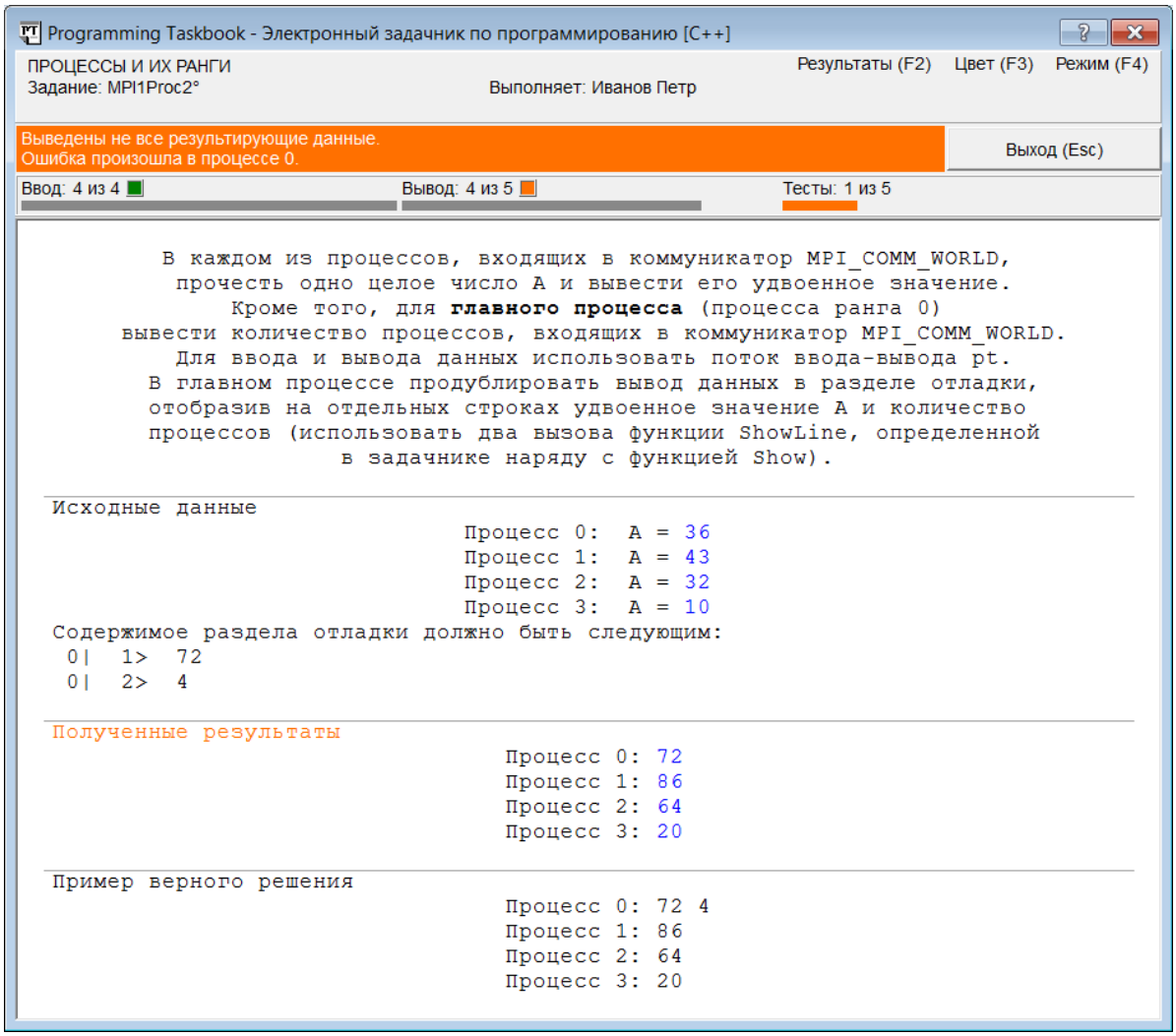


**Fig. 9.** Taskbook window with information about the error in the master process

The number of processes is stored in the variable size. 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 color magenta is used to highlight errors related to an attempt to input or output superfluous data.

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

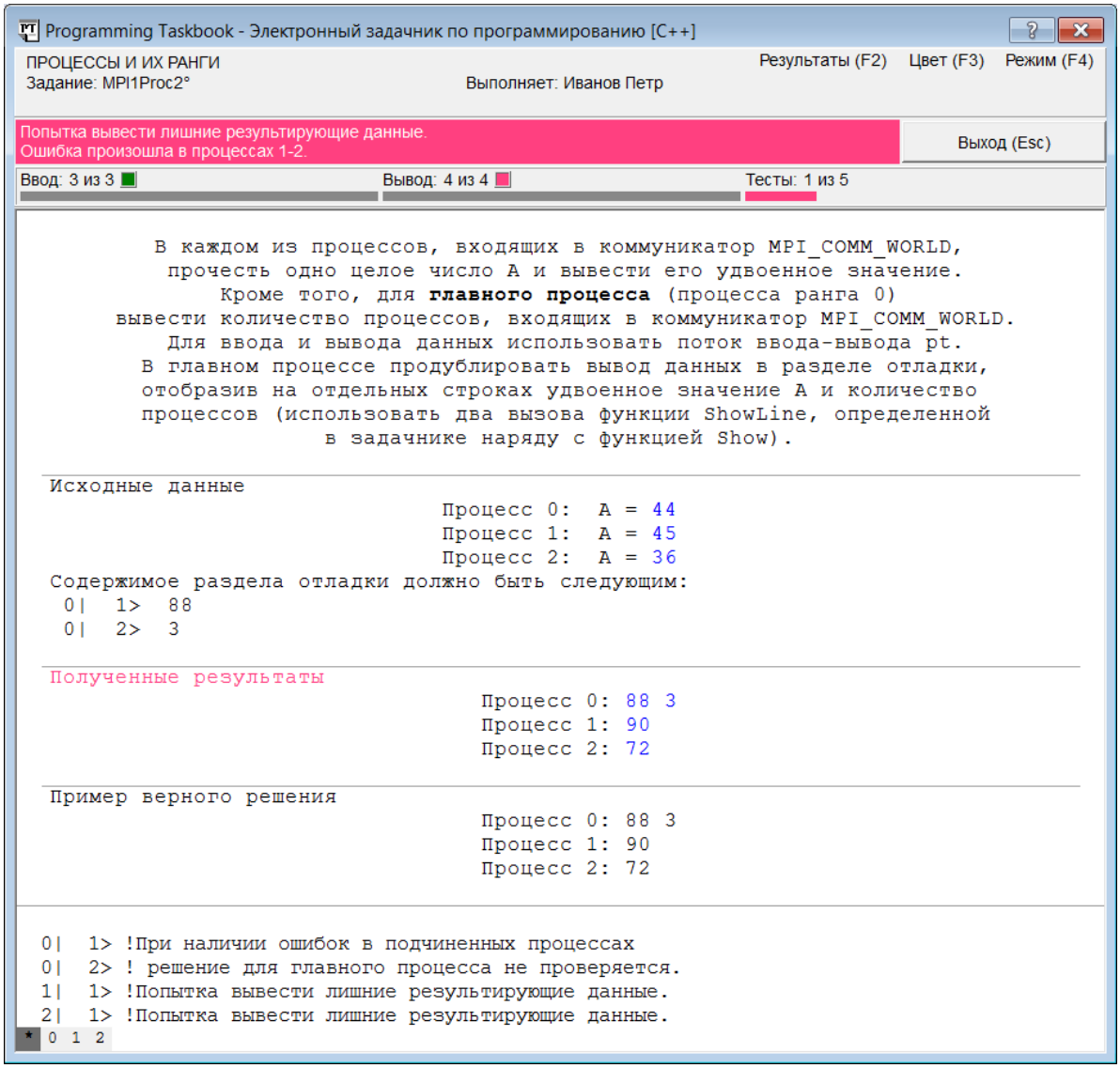


**Fig. 10**window with information about an attempt to output superfluous data

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.

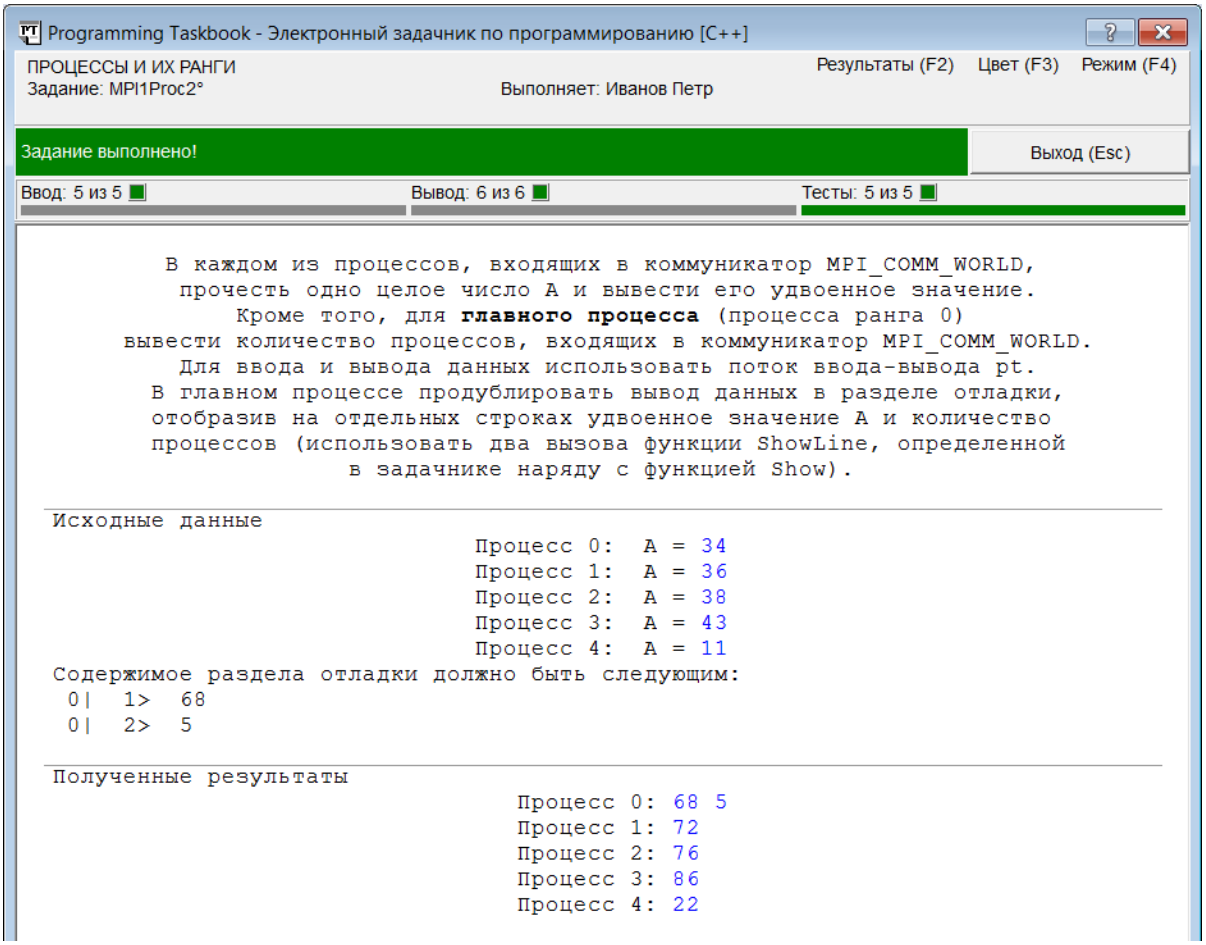


**Fig. 11.** Window with a message about successful execution of the MPI task 1 Proc 2

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 approximately the following text:

```
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.

### 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 not by the taskbook itself, but by the teacher. 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 (in this case, 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 group tasks related to studying *nonblocking data transfer*, as well as in the MPI8Inter group tasks devoted to *dynamic process creation*.

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 taskbook window, in the input data section, 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 sections).

To output data in the debug section, the taskbook provides two functions: Show and ShowLine , each of which has several overloaded options that allow you to customize the appearance of the output data and provide them with additional comments (details are given in Section 4.1.4). These functions differ in that the ShowLine function automatically moves to the next line of 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 win-

dow, simply click the button  on the right side of the taskbook window title 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).

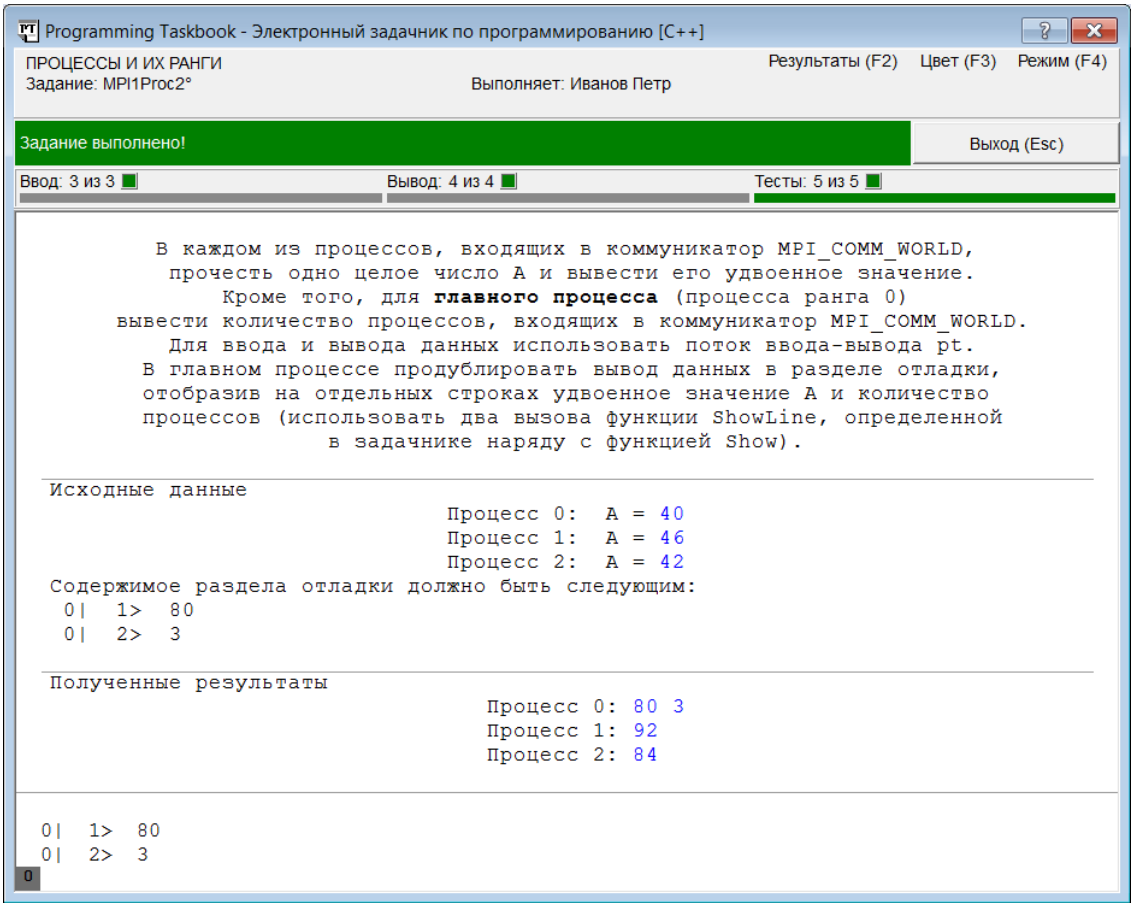


**Fig. 12.** Window with the complete version of solution

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 complete. 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.

So, we have solved the MPI1Proc2 task. In the process of solving it, we got acquainted with the actions for creating a template, studied the features of executing parallel programs and those capabilities of the taskbook that simplify their launch from the IDE. We learned about the task book 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 exchanging messages 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 an 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 data types included in the MPI library or with user data types 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 (double) and MPI_CHAR (signed char). 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 type, and the float type corresponds to the MPI_FLOAT type. The MPI_BYTE type corresponds *to a byte*, an integer in the range from 0 to 255. There are also standard composite types 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 received message;

MPI_Comm comm – communicator;

MPI_Status * status – additional information about the received 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* identifier.

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

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

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

> **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 1.2.5 system, which implements the MPI-1 standard, *this constant should not be used* (despite the fact that it is present in the MPICH 1.2.5 library).

Sometimes it is desirable to obtain additional information about the expected message before it is directly 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 given 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 standard 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 scount – 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 – parameters of the receiving message (output parameter).

It should be emphasized that for the MPI_Sendrecv function, y*ou cannot use* the same (or even just 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 – parameters of 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 will look like the one shown in Fig. 13.

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 data received from it will be written (see the previous section for a description of the parameters of these functions).

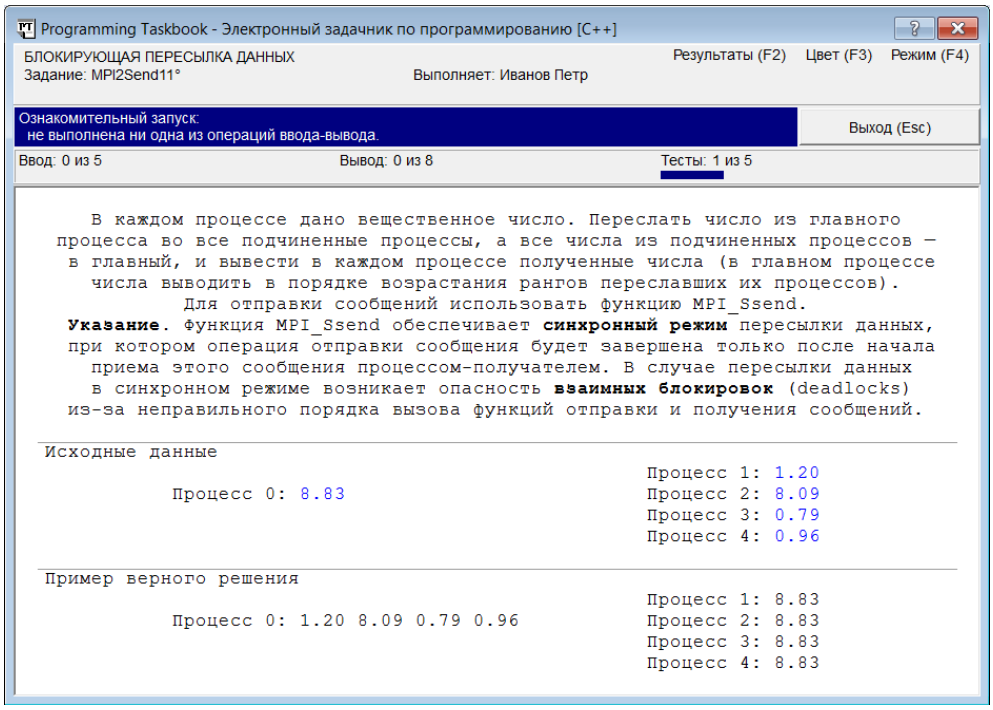


**Fig. 13.** Acquaintance run of the MPI2Send11 task

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.

Let us 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. 20–30 seconds after the console window appears with information that the program has been launched in parallel mode, a task window will appear on the screen with an error message in the slave processes (Fig. 14).

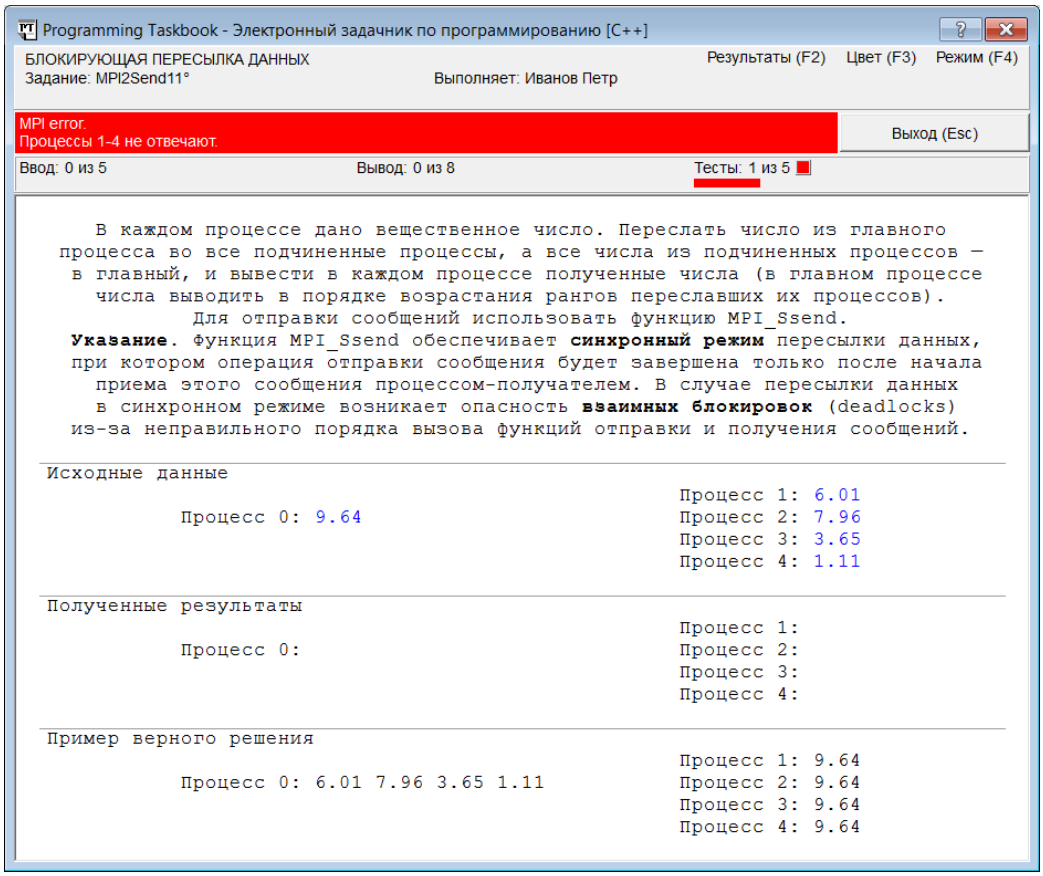


**Fig. 14.** Taskbook window with information about locking slave processes

Error message of the type "*MPI error. The processes 1–4 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 (four 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 inself (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 ("natural") 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.

> **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* rather than *synchronous* data transfer

mode. This is due to the fact that in the MPI library of the MPICH system, 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 problem 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 which the state of this 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 was 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.*

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 the exchange request (input and output parameter) and returns the output parameter status, which is typically used only for non-blocking receivings.

All other functions accept an *array* of exchange 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.

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,* return-

ing 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 the *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 variant differs from the blocking variant 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 variant MPI_Probe, is usually used in a situation where the number of elements in the sending message is not known in advance.

The persistent requests are a special type of non-blocking operations. 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 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 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.

All operations related to collective interaction of processes are performed in blocking standard mode. 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, this means that 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 message broadcast/receive buffer; in the root process this parameter is an input parameter, and in other processes it is an output parameter. The other parameters are input in all processes: count specifies the number of elements to be broadcast, and datatype 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 )
```

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 scount – 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; it should be emphasized that this parameter *is not equal to the size of the* rbuf *buffer*);

MPI_Datatype rtype – type of elements of the received message;

int root – the rank of the process receiving data;

MPI_Comm comm – communicator.

The root process also accepts its own data.

*Example* (scount = 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 count, MPI_Datatype stype, void * rbuf, int * rcounts, int * displs, MPI_Datatype rtype, int root, MPI_Comm comm)

In this case, the scount 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 rbuf data receiving buffer. 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 either 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 scount – the number of elements sent to each process (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 performing the data broadcast;

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 same combined set. Here is a list of the parameters of these functions and examples.

MPI_Allgather (void * sbuf, int count, 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_Allgatherv(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 data to all processes of the communicator. The MPI_Alltoall function sends the same amount of data from each process:

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* amount of data to all other 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 (Sectons 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 complex data types, it is desirable to specify displacements in *bytes* rather than 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 in the section devoted to derived types is due to the fact that this function is intended, first of all, for collective exchange of complex data types. 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).

### 1.2.5. Reduction operations and using composite data types

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 provided by 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 MPI_Op_create(MPI_User_function * function, int commute, MPI_Op * op) function is pro-

vided 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 parameter-flag 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 elements of the arrays invec[i] and inoutvec [i], i = 0, …, len–1, are considered, respectively, the left and right 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:*

```
            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 contains 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:*

```
            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.** In the MPI-2 standard, the set of functions related to reduction operations was expanded. We will describe one of the new functions, MPI_Reduce_scatter_block, since it is used in the MPI8Inter group tasks (see Section 2.8.3). 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* * rcount elements, where *K* is the number of processes in the communicator comm.

The second subgroup of the MPI3Coll group (see Section 2.3.2) is devoted to collective reduction operations. Let us consider one of the tasks included in it, during solving of which we will also become acquainted with an example of the use of composite data types (structures) in MPI programs.

MPI3 Coll 23. 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 task window on the screen similar to the one shown in Fig. 15.

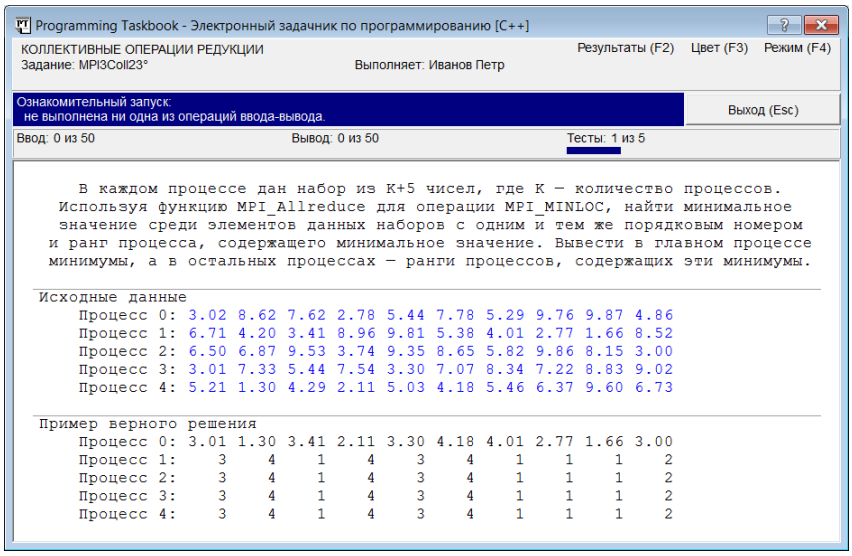


**Fig. 15.** MPI3Coll23 task demo running

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, each element of which is the result of applying the reduction operation to the elements of the original arrays with the same index.

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 will be real, and the second will 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 knows the number of processes size), or use static arrays, the size of which will be sufficient for any sets of initial data. When executing 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 for this task 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 has been successfully input (Fig. 16).

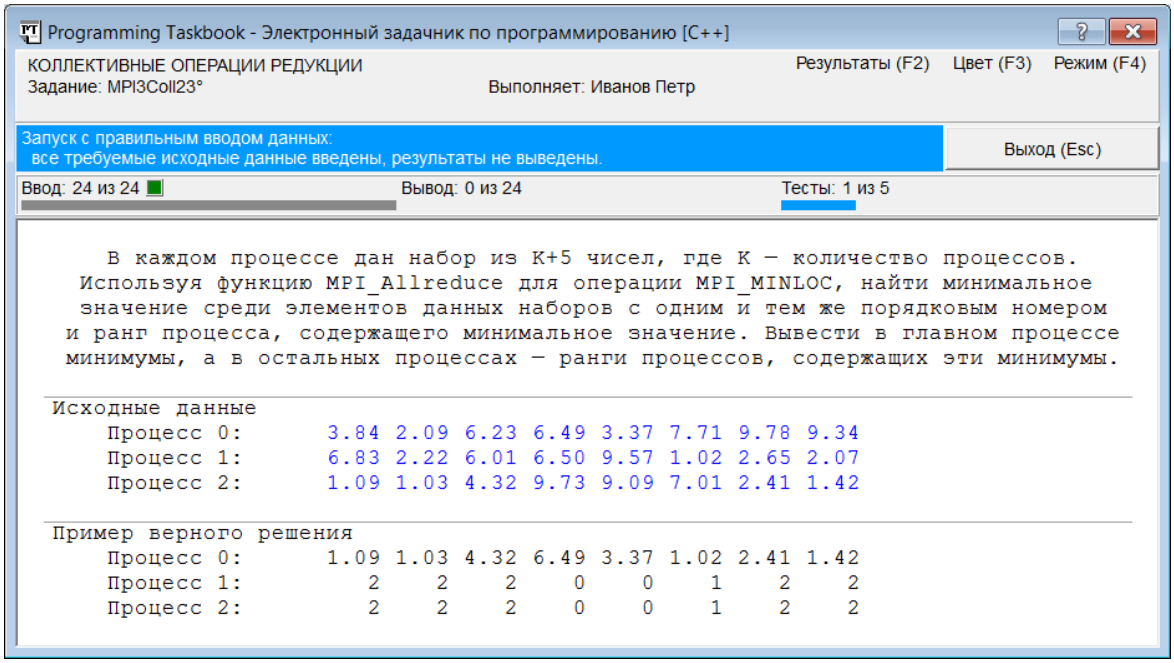


**Fig. 16.** The taskbook window with information about the successful input of the initial data

Before output the results, it is necessary to perform the corresponding collective reduction operation. It must be performed in all processes, after which 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 original 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 where the standard data types provided by the MPI library are insufficient, new MPI types must be defined (this topic is covered in the MPI4Type task group).

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 which 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 type: [T1]
Derived type: [T1][T1][T1][T1][T1]
```

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

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 type: [T1]
A memory area equal to the length of the original type: [..]
Derived type: [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 type: [T1]
A memory area equal to the length of the original type: [..]
Derived type:
[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 type being defined contains offsets in *bytes,* and second, each

block has *its own base type* (the base types 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 types: [T1], [T2], [T3], [T4]
A section of memory equal to 1 byte (denoted by a dot): .
Derived type: [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 type being defined (the distances are specified in the number of elements of the base type and are contained in the displs array of size count). This function differs from MPI_Type_indexed in that all blocks in the type 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 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 1.2.5 system. However, in the case of the MPICH2 1.3 system, an attempt to specify an unregistered type when sending 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 type are extent and size. *Extent* is the number of bytes that the type occupies in memory (including all empty spaces between its blocks). *Size* is the total size (in bytes) of all blocks, *excluding spaces between them.* 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 types (MPI_INT, MPI_DOUBLE, MPI_CHAR, etc.), 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 ("hole") for it. The MPI-1 standard provides special 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 type 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, defining the blocklens and displs arrays as follows: blocklens = {1, 1, 1}, displs = {-3, 0, 6}, then the type1 type will contain one integer element, before which there will be an initial interval of 3 bytes, and the upper limit of the type will be located at a distance of 6 bytes from the first byte occupied by the integer element. Thus, the size of the created type 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 type type2, this 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 of MPI_LB and the rightmost boundary of MPI_UB.

The way of defining initial and final intervals 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 interval when defining a new datatype was proposed in the MPI-2 standard. It is based on the use of a special function MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype * newtype). In it, to define a new type newtype, the base type 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 type oldtype already had initial and final empty intervals, the MPI_Type_create_resized function removes them and creates new ones; thus, for the new type, they can be either increased or decreased. If only the final empty interval is required, the lb parameter should be set equal to 0.

A new function was also added to the MPI-2 standard that allows one 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 obsolete.

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.

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_PACK is specified, and the size is specified in bytes.

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 from which the unpacked data was received.

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 incount, MPI_Datatype datatype, MPI_Comm comm, int * size) that allows you to determine the memory size (in bytes) that *is sufficient* to store incount 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 meaningful 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 $R$ ($R = 1, 2, \ldots, K$) starting with the $A_R$ and increasing the ordinal number by 2

($R$, $R$ + 2, $R$ + 4, …). 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 the MPI_Type_struct 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 a version 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).

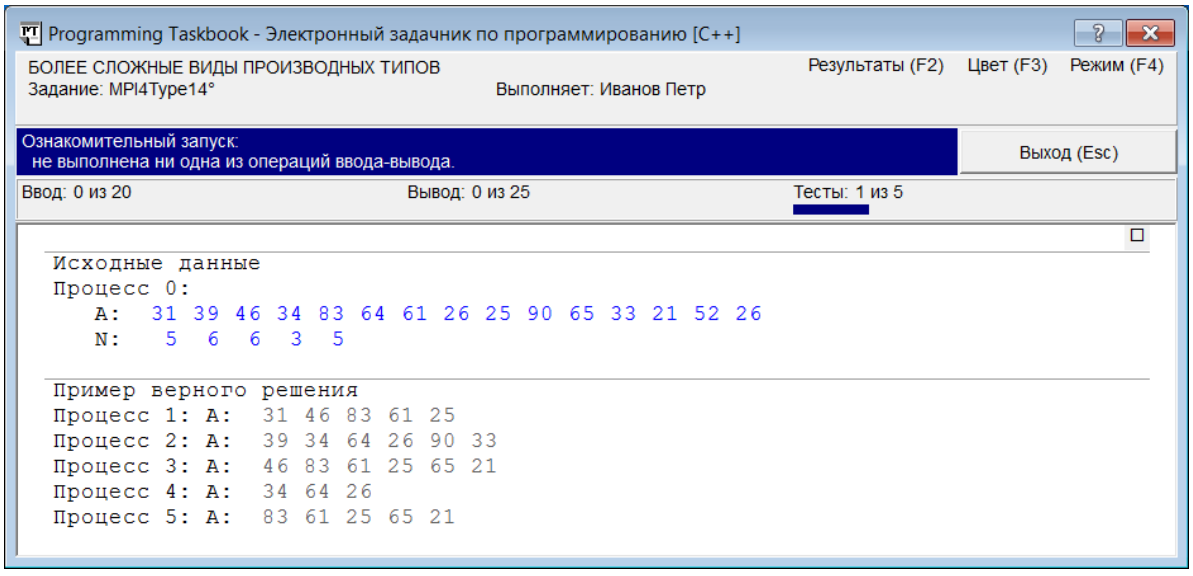


**Fig. 17.** MPI4Type14 task demo running

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 types, 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 type, we will be able to form a message containing only the necessary elements of the array *A*. For the receiving process, a new type 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 operators 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 };
int 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 data type t. Recall that the offsets for each block (specified in the displs array) are specified 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:

```
int t_sz, t_ext;
MPI_Type_size(t, &t_sz);
MPI_Type_extent(t, &t_ext);
Show("size = ", t_sz);
Show("extent = ", 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.

If we use the tools introduced in the MPI-2 standard, then when defining the type t we can do without auxiliary arrays:

```
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.

To complete the program fragment corresponding to the master process, we only need to use the created type 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 2.** If the MPI_Type_commit function had not been called in the program, then when using the MPICH2 1.3 system, the following MPI error message would have been displayed in the taskbook window: "*Error MPI_ERR_TYPE: Datatype has not been committed*".

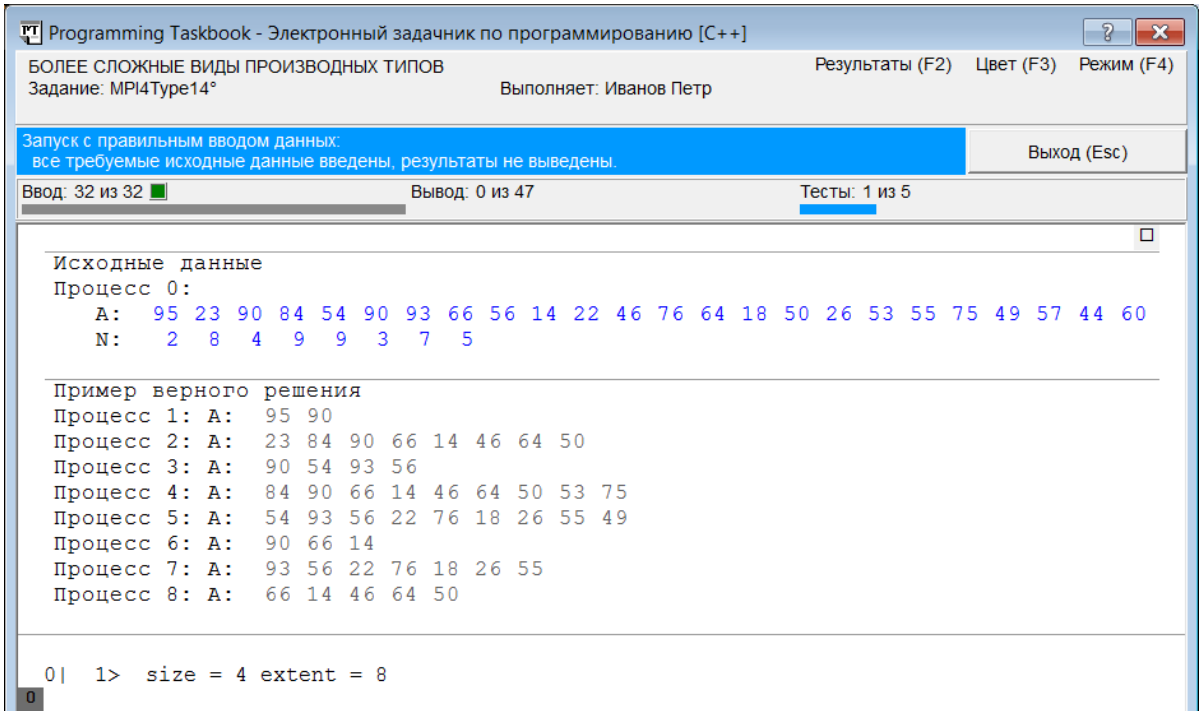


**Fig. 18.** Window with information about the created datatype

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 encounter for the first time a situation where 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 type 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, 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, you can use *the "vector" container* std::vector<T> from the Standard Template Library STL [2]. This will allow using additional capabilities for input/output related to the pt stream *iterators* (see Section 4.1.2). 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;
    }
}
```

Let us comment on the corrections made.

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 library *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 template iterator ptin_iterator<T> is defined in the taskbook, allowing you to organize the reading of data of type T. Constructor with one parameter ptin_iterator<T>(int 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—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—pointers to functions. 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 sufficient to turn the parameter of the first iterator into 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>());
```

There is another way to solve the problem mentioned above—*enclose one of the constructor parameters in parentheses:*

```
std::vector<int> n((ptin_iterator<int>(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]).

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.

Using C++ template library (and the related means of the electronic taskbook—the iterators ptin_iterator and ptout_iterator), 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. Creating 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 three 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 *intracommunicators*, associated with one group of processes. In MPI, it is possible to create another type of communicators, namely, *intercommunicators*, which are associated not with one, but with two groups of processes. The MPI8Inter task group (Section 2.8) is devoted to

intercommunicators, most of which can be executed only in the MPICH2 1.3 system, which supports 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 designed 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 this 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 method for creating 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 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 any 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 functions MPI_Group_incland MPI_Group_excl 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, …, 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 it 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 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 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 initial 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 "Process groups and communicators" (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 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 a window of the task book that was displayed on the screen during the acquaintance running of the program template for this task (Fig. 19).

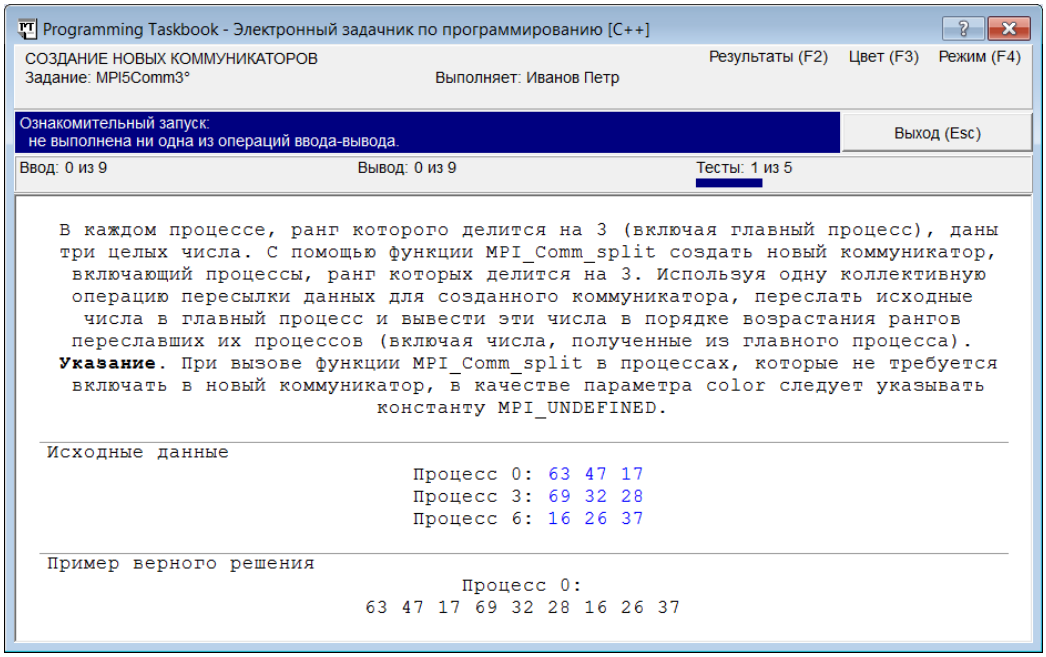


**Fig. 19.** Acquaintance running of the MPI5Comm3 task

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

```
C:\PT4Work>"C:\Program Files (x86)\MPICH2\bin\mpiexec.exe"
    -nopopup_debug -localonly 8 "C:\PT4Work\ptprj.exe"
```

Thus, in the task 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 MPI_Comm_split function's ability 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 capability, it is mentioned in the note for this task.

Taking into account the features of the MPI_Comm_split function, we will use it to create a communicator that will include only processes of rank multiple of three:

```
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 we 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 original 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 dynamic 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 iterators 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 remind you 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 $k_1 \times k_2 \times \ldots \times k_n$ (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 (arcs) 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 used 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 (MPI_Cart_create function, as well as the helper function MPI_Dims_create);
- characterization of the existing Cartesian topology (functions MPI_Cartdim_get, MPI_Cart_get, MPI_Cart_rank, MPI_Cart_coords);
- splitting the original Cartesian grid into subgrids of lower dimension (function MPI_Cart_sub);

- finding the ranks of sources and receivers when *shifting data* along some coordinate of the Cartesian grid (function MPI _Cart_shift).

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.

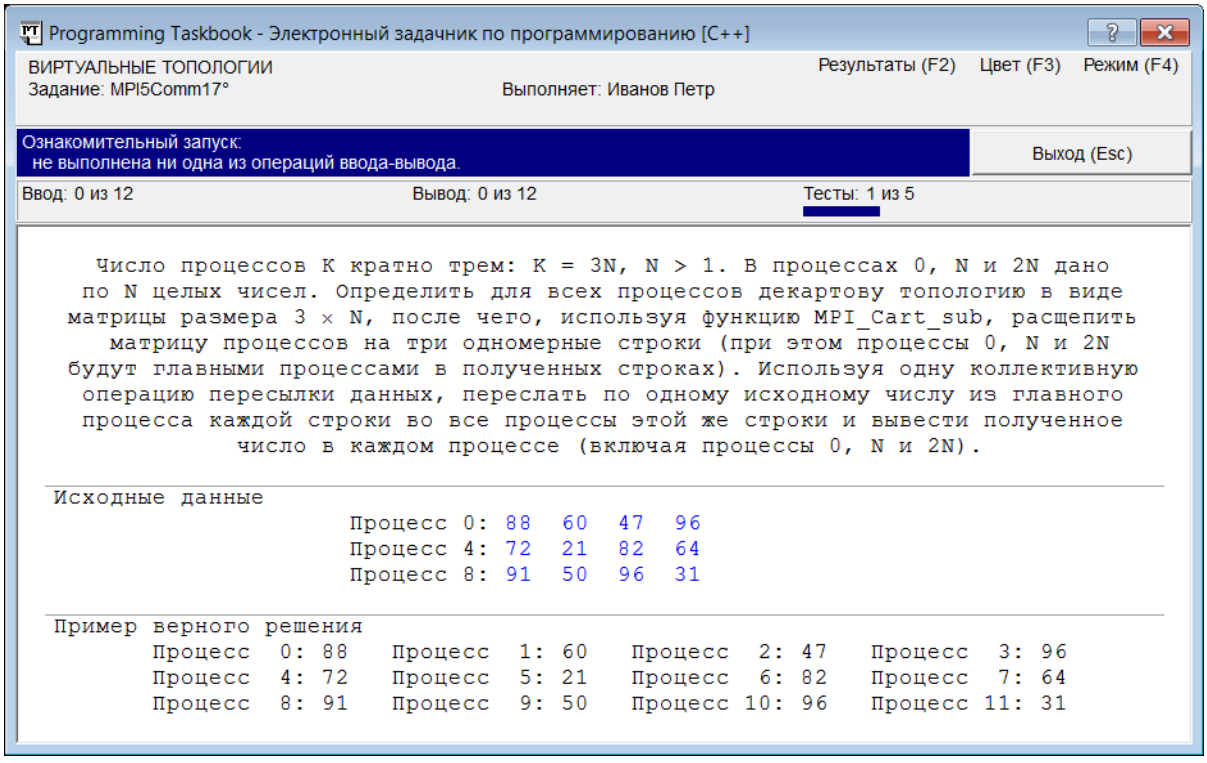


**Fig. 20.** Acquaintance run of the MPI5Comm17 task

This example corresponds to case $N = 4$: there are 12 processes that should be interpreted as elements of a $3 \times 4$ 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 of processes.

The first step in solving the task is to determine the required Cartesian topology. For this purpose, the MPI_Cart_create function is intended, 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 each dimension (in our case, the array must consist of two elements with values 3 and 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 process numbering (in our case, we need to set this parameter to 0);

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

It is convenient to use periodicity for some dimensions of the Cartesian grid, for example, when performing *cyclic* data transfer between processes included in these dimensions (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 renumbering of processes 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 renumbering 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 as the original communicator MPI_COMM_WORLD, and in the same order. However, these communicators are *different*: the data transfer operations performed using the communicators

MPI_COMM_WORLD and comm are performed independently and do not affect each other. In addition, the communicator comm is associated with a virtual topology, while the communicator MPI_COMM_WORLD 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 program text with the following statements:

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

When you launch the supplemented 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 output one line containing two numbers: its coordinates in the Cartesian topology.

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 (fourth) element of the last (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, sending fragments of the data set from one process to all processes included in the communicator.

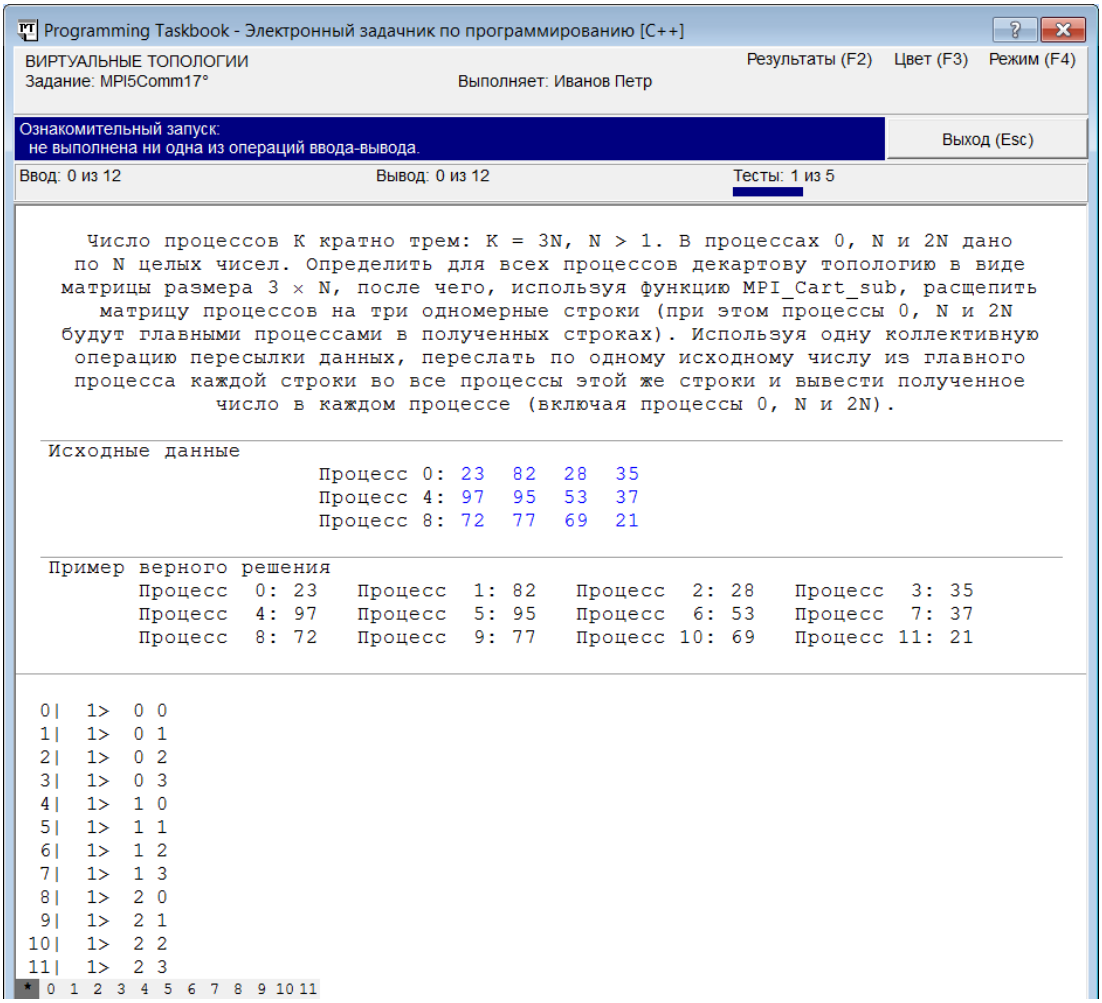


**Fig. 21.** Output of Cartesian coordinates of processes in the debug section

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 should be an array of flags remain_dims, which defines the numbers of those dimensions that should remain in the subgrids: if the corresponding dimension should *remain* in each subgrid, then a *nonzero* 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 execution, *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 of which is equal to 0, and the second is 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) will have a rank of 0 in the obtained communicator comm_sub. 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 full 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, in the given program we swap the elements with values 0 and 1 in the remain_dims array, 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 were 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 will be true for processes 0, 1, 2, and 3, and it is for them that the input operators of the initial data will be executed. 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 used 0 input data item(s) (the amount of the required items is 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 problem statement), 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 taskbookr 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.

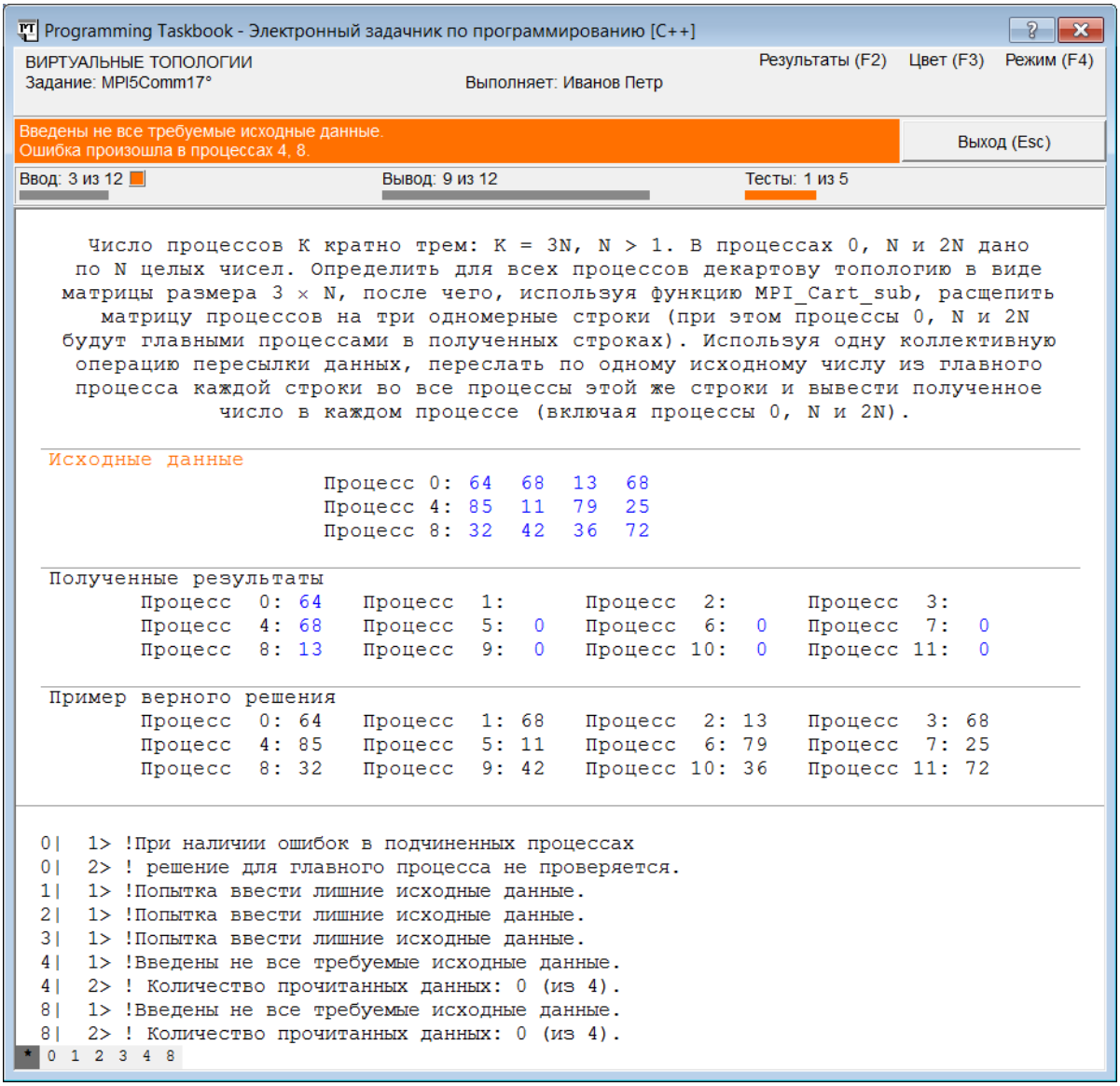


**Fig. 22.** Taskbook window when the MPI5Comm17 task is executed incorrectly

Having completed the discussion of the MPI5Comm17 task, we will 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* ndims (the number of dimensions) 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 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 output array dims.

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 descending (or rather, 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 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 – an array with the number of processes along each dimension of the Cartesian grid;

periods – an 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: fast finding the ranks of the source and destination processes for the current process when sending data along a specified coordinate (i. e., during a normal or cyclic *shift* of data). 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 number of the Cartesian coordinate along which the shift is performed (numbering 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, then a situation is possible in which 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 of 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* (the final tasks of the MPI5Comm group are devoted to this topology—see Section 2.5.2). 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 a function MPI_Cart_shift, 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 the MPI_Graphdims_get and MPI_Graph_get functions), 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 intended 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 — … — (2*N* − 2); each process with odd rank *R* (1, 3, …, 2*N* − 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—see 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.

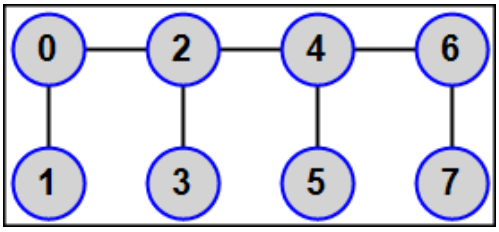


**Fig. 23.** Example of graph topology from MPI5Comm29 task

When you run this task for the acquaintance run, 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.

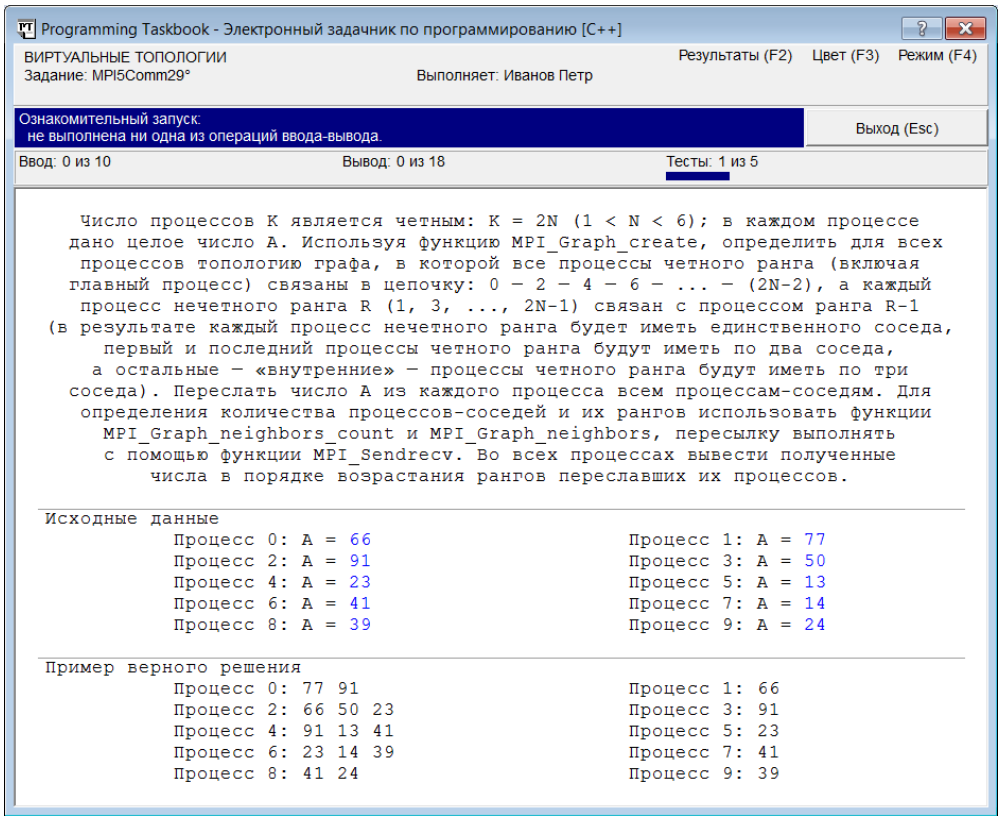


**Fig. 24.** MPI5Comm29 task acquaintance run

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

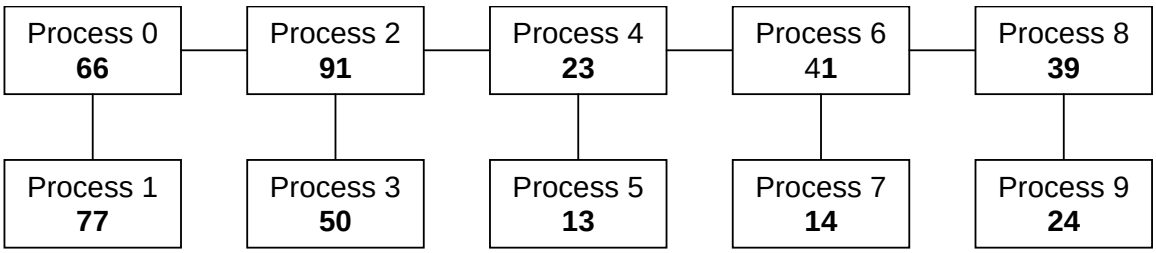


**Fig. 25.** Example of Initial data for the MPI5Comm29 task

Since process 0 has two neighbors (processes of rank 1 and 2), it must send them the number 66 and receive from them the numbers 77 and 91. Process 1 has only one neighbor (process 0), so it must send it the number 77 and receive from it the number 66. Process 2, which has three neighbors, must send them the number 91 and receive from them the numbers 66, 50, and 23, and so on.

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, and for this it is necessary to use the MPI_Graph_create function. This function has the following parameters:

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

int nnodes – 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 – an integer array of *edges* containing an ordered list of edges 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 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 vertex degree array 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 edge array, it is necessary to indicate the ranks of all neighbors for each vertex (for greater clarity, we will highlight 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 will be 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 will output the values of the elements of the obtained arrays to the debug section of the task-book 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 formulation is hidden in the figure.

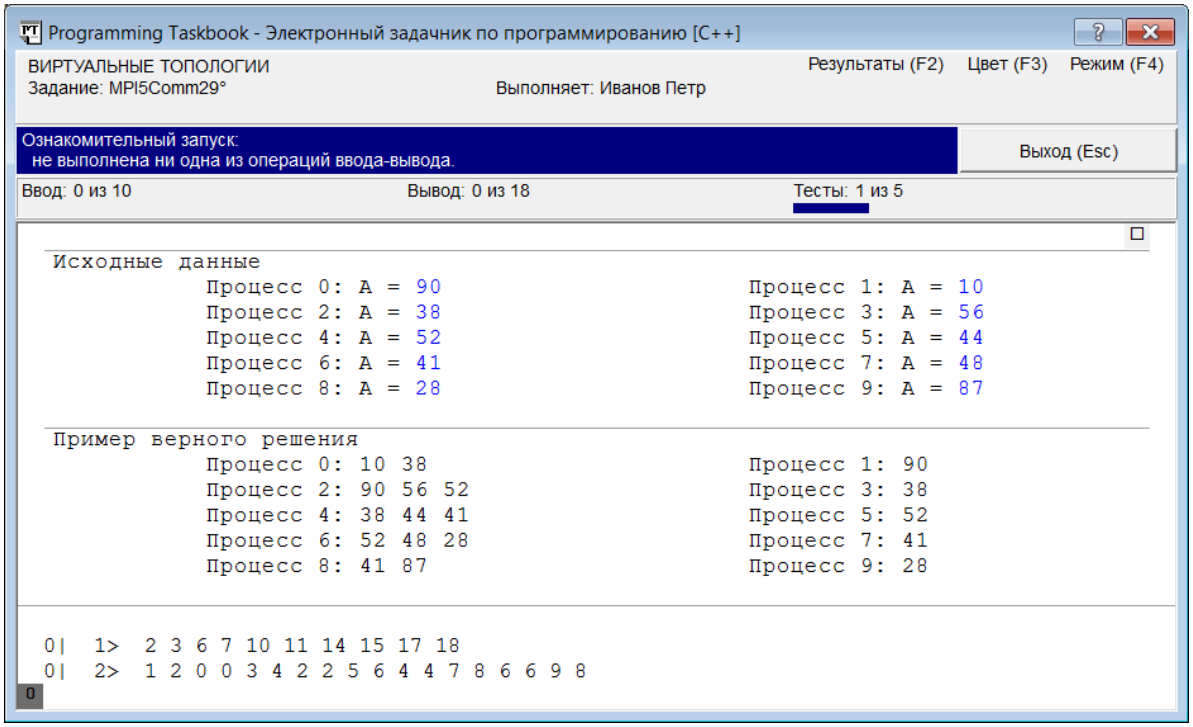


**Fig. 26.** Debug output of elements of the vertex degree array and the edge array

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, directly 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 organized.

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 corresponding 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 using 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.