Оглавление

1. Выполнение заданий в параллельном режиме: MPI1Proc3

1.1. Установка задачника PT for MPI-2 и связанных с ним сред программирования

Стандарт MPI определен для двух языков: Фортрана и C (вариант для языка C может быть использован без каких-либо изменений и в программах на языке C++). До версии 2.2 стандарт MPI содержал также объектно-ориентированный вариант библиотеки MPI для языка C++, но затем данный вариант был исключен из стандарта. Имеются реализации MPI для других языков (например, для языка C#), однако в подавляющем большинстве ситуаций параллельные программы, использующие технологию MPI, разрабатываются на языках C/C++ и Фортран.

Для того чтобы достичь максимальной эффективности, параллельные программы должны выполняться на суперкомпьютерах или вычислительных кластерах, позволяющих эффективно распределить запускаемые процессы по процессорам суперкомпьютера или узлам кластера. Однако для начального изучения возможностей технологии MPI вполне достаточно использовать локальный компьютер, запуская на нем все процессы параллельного приложения. Ожидать в такой ситуации существенного выигрыша в быстродействии алгоритмов не приходится, но с помощью подобных учебных программ можно познакомиться с механизмами MPI и опробовать их в действии. Именно с этой целью и был разработан *электронный задачник по параллельному программированию* **Programming Taskbook for MPI-2** (**PT for MPI-2**).

Задачник PT for MPI-2 позволяет разрабатывать параллельные программы на языке C++ с применением интерфейса MPI для языка C. Дополнительные возможности языка C++ используются, в основном, для более удобной организации ввода-вывода (с применением потоков), хотя в некоторых ситуациях оказываются удобными и другие средства C++, например шаблонные функции (см. задания MPI2Send22–25). Поскольку задачник PT for MPI-2 является специализированным дополнением для универсального задачника по программированию Programming Taskbook, он может использоваться совместно со всеми средами программирования для языка C++, которые поддерживает базовый задачник. Для версии 4.17 базового задачника Programming Taskbook, начиная с которой к нему можно подключать задачник PT for MPI-2, это среды Microsoft Visual Studio 2008–2017 и Code::Blocks версии 13 и выше.

Таким образом, прежде всего на компьютере должна быть установлена одна из указанных сред программирования.

Однако для запуска параллельных программ, разработанных на основе технологии MPI, наличия среды программирования (даже при условии подключения дополнительных библиотек MPI) недостаточно. Необходима система, позволяющая запускать процессы параллельной программы и обеспечивающая обмен сообщениями между ними. Одной из популярных свободно распространяемых систем поддержки MPI является система MPICH, разрабатываемая в Аргоннской национальной лаборатории США. Задачник PT for MPI-2 может использоваться совместно с двумя версиями данной системы для Windows:

- MPICH 1.2.5 (ftp://ftp.mcs.anl.gov/pub/mpi/nt/mpich.nt.1.2.5.exe), поддерживает стандарт MPI 1.2;
- MPICH2 1.3 (http://www.mpich.org/static/downloads/1.3/mpich2-1.3-win-ia32.msi), поддерживает стандарт MPI 2.1.

При использовании системы MPICH 1.2.5 можно выполнять только те задания, которые предназначены для изучения средств MPI стандарта 1.x. Система MPICH2 1.3 позволяет выполнять все задания, входящие в задачник PT for MPI-2.

Для установки MPICH 1.2.5 достаточно запустить установочный файл и следовать его рекомендациям (система по умолчанию устанавливается в подкаталоге MPICH каталога Program Files для 32-разрядных программ).

Для корректной установки системы MPICH2 *необходимо запустить установочный файл mpich2-1.3-win-ia32.msi с правами администратора*. Если соответствующий пункт контекстного меню для данного файла отсутствует, то можно, например, запустить командную строку с правами администратора («Пуск | Все программы | Стандартные | Командная строка», использовать команду «Запуск от имени администратора» из контекстного меню данной программы), а в ней осуществить запуск установочного файла mpich2-1.3-win-ia32.msi. При наличии на компьютере файлового менеджера FAR удобнее запустить с правами администратора эту программу, и в ней выполнить запуск установочного файла. Если при установке системы MPICH2 не использовать права администратора, то установка пройдет нормально, однако при попытке запуска параллельного приложения с помощью программы mpiexec.exe будет выведено сообщение «Unknown option: -d», вызванное тем, что система не сможет запустить менеджер процессов smpd.exe, входящий в состав MPICH2. По умолчанию система MPICH2 устанавливается в подкаталоге MPICH2 каталога Program Files для 32-разрядных программ.

> **Примечание**. Иногда возникает ситуация, когда система Windows начинает блокировать вызов компонентов системы MPICH2, обеспечивающих запуск программ в параллельном режиме. В этом случае обычно бывает достаточно *переустановить* систему MPICH2, запустив установочную программу и выбрав в ней вариант «Repair

MPICH2». Некоторые виды антивирусных приложений также могут пытаться блокировать выполнение параллельных программ, считая их подозрительными.

После того как среда программирования для C++ и система MPICH установлена, следует установить базовый вариант электронного задачника Programming Taskbook версии не ниже 4.17 и электронный задачник PT for MPI-2 (в указанном порядке). Установочные программы для этих задачников можно скачать с сайта электронного задачника ptaskbook.com (либо в разделе «Скачивание дистрибутивов», либо на главных страницах разделов «Главная» и «PT for MPI-2». На главной странице раздела «PT for MPI-2» содержатся также ссылки для скачивания дистрибутивов обеих версий системы MPICH, поддерживаемых задачником PT for MPI-2.

После установки базового варианта задачника на экране появится окно программы PT4Setup, в котором будут перечислены все среды программирования, в которых может использоваться задачник. После установки задачника PT for MPI-2 в этом окне дополнительно появятся те версии системы MPICH, которые обнаружены на компьютере (рис. 1).

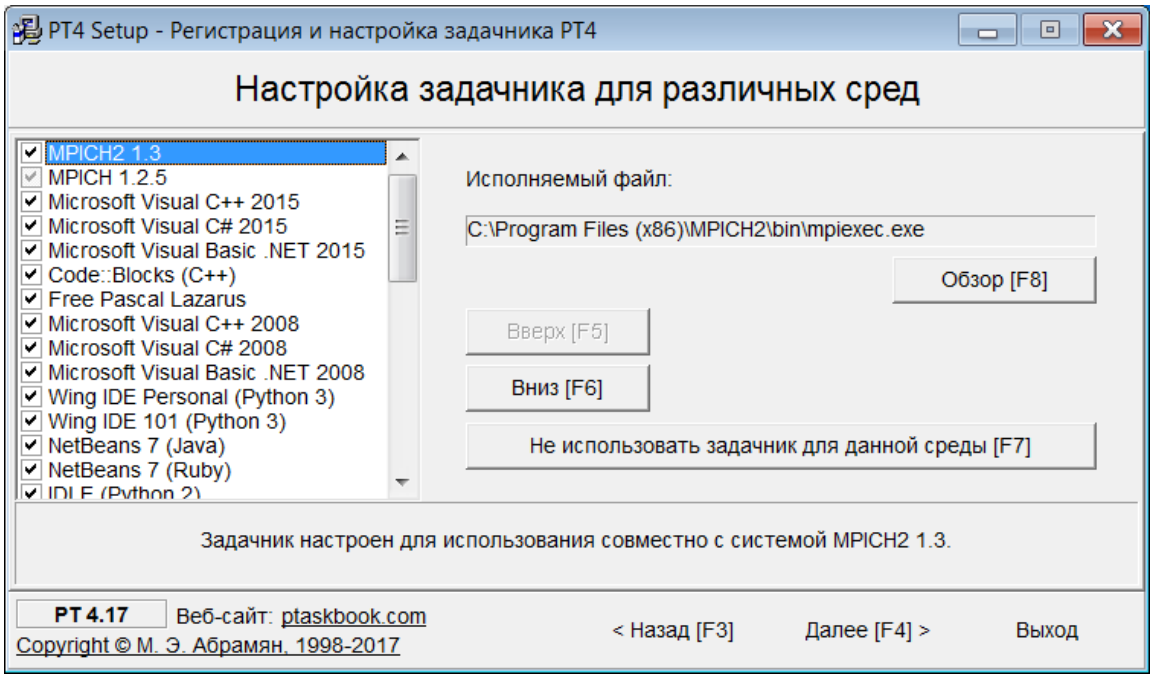


Рис. 1. Окно программы PT4Setup со списков найденных сред разработки

При наличии двух версий MPICH одна будет активной, а другая (с затененным флажком) — временно отключенной. Для активизации другой версии MPICH следует выполнить щелчок на ее затененном флажке.

После установки всех указанных программ можно приступать к выполнению заданий из задачника PT for MPI-2.

Будем считать для определенности, что при выполнении заданий используется среда Microsoft Visual Studio .NET 2015, а в качестве активной версии системы MPICH выбрана версия MPICH2 1.3.

1.2. Основные понятия MPI-программирования

Знакомство с параллельным программированием начнем с рассмотрения следующей простой задачи из начальной группы MPI1Proc. Это позволит нам не только ознакомиться с основными понятиями параллельного программирования, основанного на передаче сообщений, но и изучить возможности электронного задачника, связанные с вводом и выводом данных, а также с печатью отладочной информации.

MPI1Proc2. В каждом из процессов, входящих в коммуникатор MPI_COMM_WORLD, прочесть одно целое число *A* и вывести его удвоенное значение. Кроме того, для *главного процесса* (процесса ранга 0) вывести количество процессов, входящих в коммуникатор MPI_COMM_WORLD. Для ввода и вывода данных использовать поток ввода-вывода pt. В главном процессе продублировать вывод данных в разделе отладки, отобразив на отдельных строках удвоенное значение *A* и количество процессов (использовать два вызова функции ShowLine, определенной в задачнике наряду с функцией Show).

Прежде всего разъясним термины параллельного MPI-программирования. При параллельном выполнении программы запускается несколько экземпляров этой программы. Каждый запущенный экземпляр представляет собой отдельный *процесс* (англ. process), который может взаимодействовать с другими процессами, обмениваясь *сообщениями* (messages). MPI-функции предоставляют разнообразные средства для реализации такого взаимодействия (аббревиатура MPI расшифровывается как «Message Passing Interface» — *интерфейс передачи сообщений*).

Для идентификации каждого процесса в группе процессов используется понятие ранга (rank). *Ранг процесса* — это порядковый номер процесса в группе процессов, отсчитываемый от нуля (таким образом, первый процесс имеет ранг 0, а последний процесс — ранг $K - 1$, где K — количество процессов в группе). При этом группа процессов может включать лишь часть всех запущенных процессов параллельного приложения. Заметим, что в формулировках заданий буква *K* используется обычно для обозначения количества процессов.

С группой процессов связывается особая сущность библиотеки MPI, называемая *коммуникатором* (communicator). Любое взаимодействие процессов возможно только в рамках того или иного коммуникатора. Стандартный коммуникатор, содержащий все процессы, запущенные при параллельном выполнении программы, имеет имя MPI_COMM_WORLD. «Пустой» коммуникатор, не содержащий ни одного процесса, имеет имя

MPI_COMM_NULL. Коммуникатор можно представлять себе как канал, соединяющий между собой некоторую группу процессов. В некоторых случаях бывает удобно организовывать дополнительные каналы, которые, например, содержат не все исходные процессы или в которых изменен порядок их следования. В этой ситуации создаются новые коммуникаторы. Более подробно работа с коммуникаторами рассматривается в группах заданий MPI5Comm и MPI8Inter. В заданиях начальных четырех групп всегда используется стандартный коммуникатор MPI_COMM_WORLD.

Процесс ранга 0 часто называют *главным процессом* (master process), а остальные процессы — *подчиненными процессами* (slave processes). Как правило, главный процесс играет особую роль по отношению к подчиненным процессам, передавая им свои данные или получая данные от всех (или некоторых) подчиненных процессов. В рассматриваемом задании MPI1Proc2 все процессы должны выполнить одно и то же действие — прочесть одно целое число и вывести его удвоенное значение, а главный процесс, кроме этого, должен выполнить дополнительное действие — вывести количество всех запущенных процессов (иными словами, количество всех процессов, входящих в коммуникатор MPI_COMM_WORLD). Обратите внимание на то, что в этом простом задании процессам не требуется обмениваться сообщениями друг с другом (таковы все задания вводной группы MPI1Proc).

1.3. Создание заготовки для параллельной программы

Процесс выполнения задания с применением задачника PT for MPI-2 начинается с создания заготовки проекта для выбранного задания. К этому проекту уже будут подключены все необходимые библиотеки (связанные с задачником и с выбранной системой MPICH), кроме того, основной файл этого проекта будет содержать важные фрагменты кода, необходимые при выполнении любой параллельной программы.

Для создания заготовки предназначен программный модуль PT4Load, входящий в состав задачника. Проще всего вызвать этот модуль с помощью ярлыка Load.lnk, который автоматически создается в рабочем каталоге учащегося (по умолчанию рабочий каталог имеет имя PT4Work и находится на диске C). После запуска модуля на экране появится его окно (рис. 2).

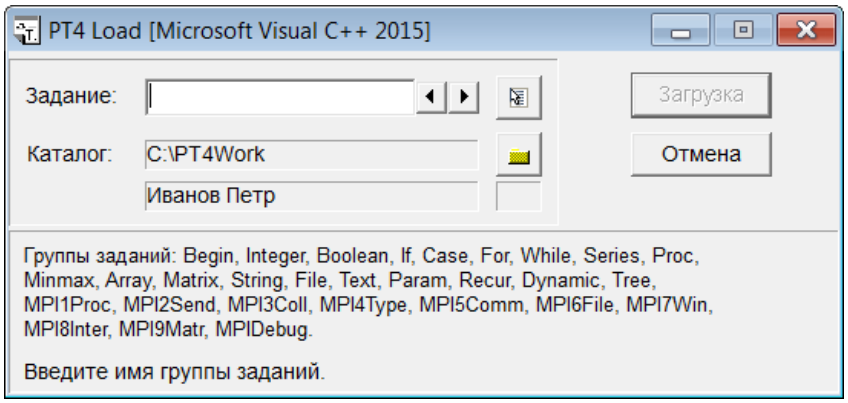


Рис. 2. Окно модуля PT4Load

Так выглядит окно, если текущей программной средой задачника является среда Microsoft Visual Studio 2015 для языка C++. Для изменения текущей среды достаточно выполнить в окне щелчок правой кнопкой мыши (или нажать кнопку  или клавишу Shift+F10) и выбрать из появившегося контекстного меню новую среду (например «Code::Blocks (C++)»; при этом в заголовке окна появится название выбранной среды).

Вид контекстного меню приведен на рис. . Кроме списка доступных сред контекстное меню содержит список доступных вариантов системы MPICH (с указанием текущего варианта), позволяет выбрать язык интерфейса (русский или английский), а также выполнить ряд дополнительных действий по настройке рабочего каталога.

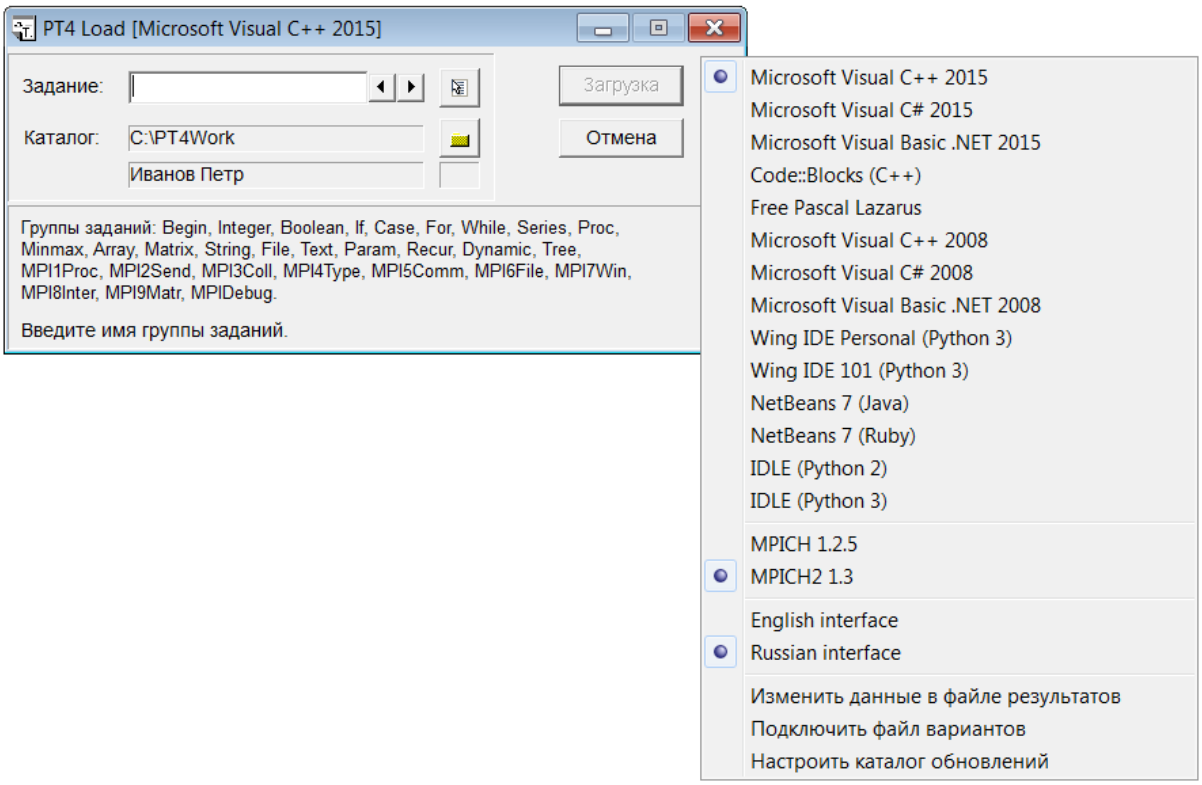


Рис. 3. Модуль PT4Load с развернутым контекстным меню

Обратите внимание на группы заданий, начинающиеся с префикса MPI (MPI1Proc и т. д.). Они появятся в списке только после установки задачника по параллельному программированию PT for MPI-2 и только в том случае, если в качестве текущей среды программирования выбрана среда для языка C++.

Определимся с выбором среды программирования, после чего введем в поле «Задание» текст MPI1Proc2 (полное имя группы вводить не обязательно; достаточно ввести текст MPI1, однозначно определяющий группу, после чего нажать пробел и указать номер задания 2). В результате кнопка «Загрузка» станет доступной; кроме того, в нижней части окна будет приведено краткое описание выбранной группы и количество входящих в нее заданий (см. рис.).

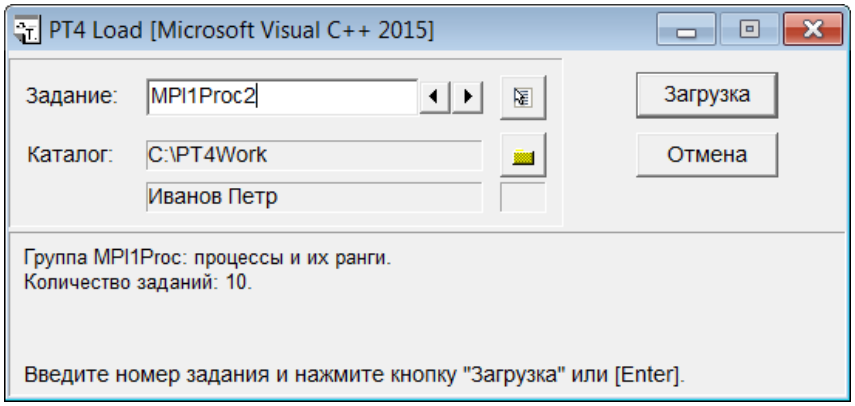


Рис. 4. Вид окна PT4Load после ввода имена задания

Нажав кнопку «Загрузка» или клавишу Enter, мы создадим заготовку для указанного задания, которая будет немедленно загружена в выбранную программную среду.

Проект, созданный задачником для языка C++, всегда имеет имя ptprj; это позволяет, в частности, существенно уменьшить количество файлов, создаваемых в рабочем каталоге при выполнении большого количества различных заданий. Он включает ряд файлов, основным из которых является cpp-файл, имя которого совпадает с именем выполняемого задания (в нашем случае MPI1Proc2.cpp). Этот файл автоматически загружается в редактор кода среды программирования; именно в нем необходимо ввести решение задачи. Приведем текст файла 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);

}
```

В начале программы содержатся директивы подключения к ней вспомогательных заголовочных файлов pt4.h и mpi.h. Затем располагается функция Solve, которая должна содержать решение задачи.

При анализе файла MPI1Proc2.cpp возникает естественный вопрос: где находится «стартовая» функция приложения (обычно имеющая имя main или WinMain)? Данная функция размещается в другом файле проекта, поскольку ее содержимое не требует редактирования. В ней производится инициализация задачника, после чего происходит вызов функции Solve с решением, при необходимости перехватываются исключения, которые могут возникнуть при выполнении функции Solve, и в конце выполняются завершающие действия, связанные с анализом полученного решения.

Программа-заготовка для заданий по параллельному программированию содержит дополнительные операторы, отсутствующие в заготовках для «непараллельных» заданий. Эти операторы должны использоваться практически в любой параллельной MPI-программе, поэтому, чтобы учащемуся не требовалось набирать их каждый раз заново, они автоматически добавляются к программе при ее создании.

Обсудим содержимое функции Solve подробнее. Первым ее оператором является оператор вызова функции Task, инициализирующей требуемое задание (см. п.). Этот оператор имеется в программах-заготовках для всех заданий, в том числе и не связанных с параллельным программированием. Функция Task реализована в ядре задачника Programming Taskbook (динамической библиотеке) и доступна в программе учащегося благодаря подключенному к ней заголовочному файлу pt4.h. Помимо заголовочного файла pt4.h в рабочем каталоге учащегося должен находиться файл pt4.cpp, содержащий определения функций, объявленных в файле pt4.h.

Оставшиеся операторы функции Solve связаны с библиотекой MPI. Мы уже отмечали, что задачник использует библиотеку MPI, входящую в систему MPICH — широко распространенную бесплатную программную реализацию стандарта MPI для различных операционных систем, в том числе и для Windows. Функции и константы библиотеки MPI доступны программе благодаря подключенному к ней заголовочному файлу mpi.h. Реализация функций из файла mpi.h содержится в объектном файле

mpich.lib, который требуется подключать к любому проекту на языках C/C++, использующему библиотеку MPI. Однако в нашем случае это подключение *уже выполнено* в ходе создания проекта-заготовки, поэтому дополнительных действий, связанных с этим подключением, выполнять не требуется.

> **Примечание 1.** Объектный lib-файл для системы MPICH2 1.3 содержится в подкаталоге MPICH2\lib и имеет имя mpi.lib, однако задачник использует для этой библиотеки название mpich.lib, совпадающее с названием совпадающее с названием аналогичной библиотеки для версии MPICH 1.2.5 (это позволяет задавать для проектов одинаковые настройки независимо от того, с какой версией системы MPICH они должны быть связаны: к проекту всегда подключается тот вариант библиотеки mpich.lib, который содержится в рабочем каталоге).
>
> **Примечание 2**. Для подключения к проекту дополнительного lib-файла в среде Visual Studio надо вызвать окно свойств проекта (команда «Project | <*имя проекта*> Properties…»), перейти в этом окне в раздел «Configuration Properties | Linker | Input» и указать имя подключаемого файла в поле ввода «Additional Dependencies». Если перейти на данный раздел в созданном проекте, то начальный текст в этом поле ввода будет иметь вид «mpich.lib;».
>
> Аналогичные действия надо проделать и в среде Code::Blocks; в ней надо выполнить команду «Project | Build options…», в появившемся окне перейти на вкладку «Linker settings» и указать требуемую библиотеку в разделе «Link libraries».

Вызов функции MPI_Initialized позволяет определить, инициализирован для программы параллельный режим или нет. Если режим инициализирован, то выходной параметр функции принимает значение, отличное от нуля; в противном случае параметр полагается равным нулю. Следует отметить, что инициализация параллельного режима выполняется функцией MPI_Init, которая в приведенном коде отсутствует. Это объясняется тем, что за инициализацию отвечает сам задачник, и выполняется она перед тем, как программа переходит к выполнению кода учащегося. Однако такая инициализация выполняется задачником не всегда. Например, если программа запущена в демо-режиме (для этого достаточно при вызове функции Task дополнить имя задания символом «?»: Task("MPI1Proc2?")), задачник *не выполняет* инициализацию параллельного режима, поскольку в нем нет необходимости. В этой ситуации вызов в коде учащегося функций MPI (отличных от MPI_Initialized) может привести к некорректной работе программы. Вызов функции MPI_Initialized и следующий за ним условный оператор предназначены для того, чтобы «пропустить» при выполнении программы все операторы, введенные учащимся, если программа запущена не в параллельном режиме.

Два последних оператора программы позволяют определить две характеристики, необходимые для нормальной работы любого процесса любой содержательной параллельной программы: общее количество процессов (функция MPI_Comm_size) и ранг текущего процесса (функция MPI_Comm_rank). Текущим считается процесс, вызвавший данную функцию. Требуемая характеристика возвращается во втором параметре соответствующей функции; первым параметром является коммуникатор, задающий набор процессов. Благодаря вызову этих функций мы можем сразу использовать в нашей программе значения size (общее число процессов в коммуникаторе MPI_COMM_WORLD) и rank (ранг текущего процесса в коммуникаторе MPI_COMM_WORLD; значение ранга обязательно лежит в диапазоне от 0 до size – 1). Обратите внимание на то, что вторые параметры этих функций являются *указателями* на соответствующие переменные.

Примечание 3. Любая функция MPI возвращает информацию об успешности своего выполнения. В частности, при успешном завершении функция возвращает значение MPI_SUCCESS. Однако, как правило, возвращаемые значения функций MPI не анализируются, а возникающие ошибки обрабатываются специальным *обработчиком ошибок* (error handler). При выполнении заданий по параллельному программированию с применением задачника PT for MPI-2 используется специальный обработчик ошибок, который определен в задачнике и обеспечивает вывод информации об ошибках в особом разделе окна задачника — *разделе отладки*. Некоторые возможности MPI, связанные с обработкой ошибок, рассматриваются в заданиях MPI5Comm23–24, более подробное описание средства MPI, связанных с обработкой ошибок, приводится, например, в [4, гл. 8]; [6, гл. 11].

Примечание 4. В библиотеке MPI предусмотрена функция MPI_Finalize, завершающая параллельную часть программы (после вызова этой функции нельзя использовать остальные функции библиотеки MPI). Однако в той части программы, которая разрабатывается учащимся, вызывать эту функцию нельзя, так как после выполнения данной части программы задачник должен «собрать» все результаты, полученные в подчиненных процессах (чтобы проанализировать их и отобразить в окне главного процесса), а для этого программа должна находиться в параллельном режиме. Поэтому задачник берет на себя обязанность не только инициализировать параллельный режим (вызовом функции MPI_Init в начале выполнения программы), но и завершить его (вызовом функции MPI_Finalize в конце программы).

1.4. Запуск программы в параллельном режиме

Теперь выясним, каким образом данный проект можно запустить в параллельном режиме. При компиляции и запуске из интегрированной среды любой программы (даже с подключенной библиотекой MPI) она будет запущена в единственном экземпляре. В единственном экземпляре она будет запущена и в случае, если мы выйдем из интегрированной среды и запустим на выполнение откомпилированный exe-файл данной программы.

Для запуска программы в параллельном режиме необходима «управляющая» программа (англ. host application), которая, во-первых, обеспечивает запуск нужного количества экземпляров исходной программы и, во-вторых, перехватывает сообщения, отправленные этими экземплярами (*процессами*) и пересылает их по назначению.

Мы уже отмечали, что экземпляры «настоящих» параллельных программ обычно запускаются на разных компьютерах, объединенных в сеть (вычислительный кластер), или на суперкомпьютерах, снабженных большим числом процессоров. Именно в ситуации, когда каждый процесс выполняется на своем собственном процессоре, и обеспечивается максимальная эффективность параллельных программ. Разумеется, для проверки правильности наших учебных программ все их экземпляры достаточно запускать на одном локальном компьютере. Однако управляющая программа необходима и в этом случае.

В качестве управляющей программы для параллельных программ задачник PT for MPI использует приложение, входящее в систему MPICH. В версии MPICH 1.2.5 оно имеет имя MPIRun.exe (и содержится в каталоге MPICH\mpd\bin), в версии MPICH2 1.3 — имя mpiexec.exe (и содержится в каталоге MPICH2\bin). Для запуска исполняемого файла в параллельном режиме достаточно запустить соответствующую управляющую программу (MPIRun.exe или mpiexec.exe), передав ей полное имя файла, требуемое количество процессов (т. е. запущенных экземпляров программы) и некоторые дополнительные параметры. Поскольку при тестировании программы такие запуски придется осуществлять многократно, желательно создать пакетный файл (bat-файл), содержащий вызов управляющей программы со всеми необходимыми параметрами. Однако и в этом случае процесс тестирования параллельной программы будет не слишком удобным: каждый раз после внесения необходимых исправлений в программу ее придется перекомпилировать, после чего, покинув интегрированную среду, запускать bat-файл. Проанализировав результаты работы программы, потребуется опять вернуться в интегрированную среду для внесения в нее очередных изменений, и т. д.

> **Примечание 1**. В среде Microsoft Visual Studio предусмотрен механизм, упрощающий тестирование программ, для запуска которых тре-

буется управляющая программа. В настройках проекта (команда меню «Project | *<имя проекта>* Properties…») в разделе «Debugging» можно указать эту управляющую программу (поле «Command»; в нашем случае это будет MPIRun.exe или mpiexec.exe) и параметры ее запуска (поле «Command Arguments»; требуемые в нашем случае параметры описываются далее в этом пункте). После задания подобных настроек запуск разрабатываемого приложения будет приводить к запуску управляющей программы. Таким образом, необходимость в отдельном запуске bat-файла отпадает. Правда, при этом потребуется добавить в программу фрагмент, обеспечивающий ее приостановку в конце выполнения, поскольку при его отсутствии окно управляющей программы будет немедленно закрыто, и не удастся ознакомиться с полученными результатами. Следует также отметить, что в большинстве сред разработки (в частности, в Code::Blocks) управляющую программу можно указывать только при тестировании библиотек, поэтому при использовании подобных сред обойтись без вспомогательных bat-файлов для запуска тестируемой программы не удастся.

Для того чтобы действия по запуску параллельной программы не отвлекали от решения задачи, задачник PT for MPI-2 выполняет их самостоятельно. Продемонстрируем это на примере нашего проекта для решения задачи MPI1Proc2, который уже готов к запуску. Нажмем клавишу F5 в среде Visual Studio; в результате будет выполнена компиляция программы и, в случае ее успешного завершения, программа будет запущена на выполнение. Поскольку мы не вносили в заготовку никаких изменений, компиляция должна завершиться успешно. При запуске программы на экране появится консольное окно, подобное приведенному на рис.

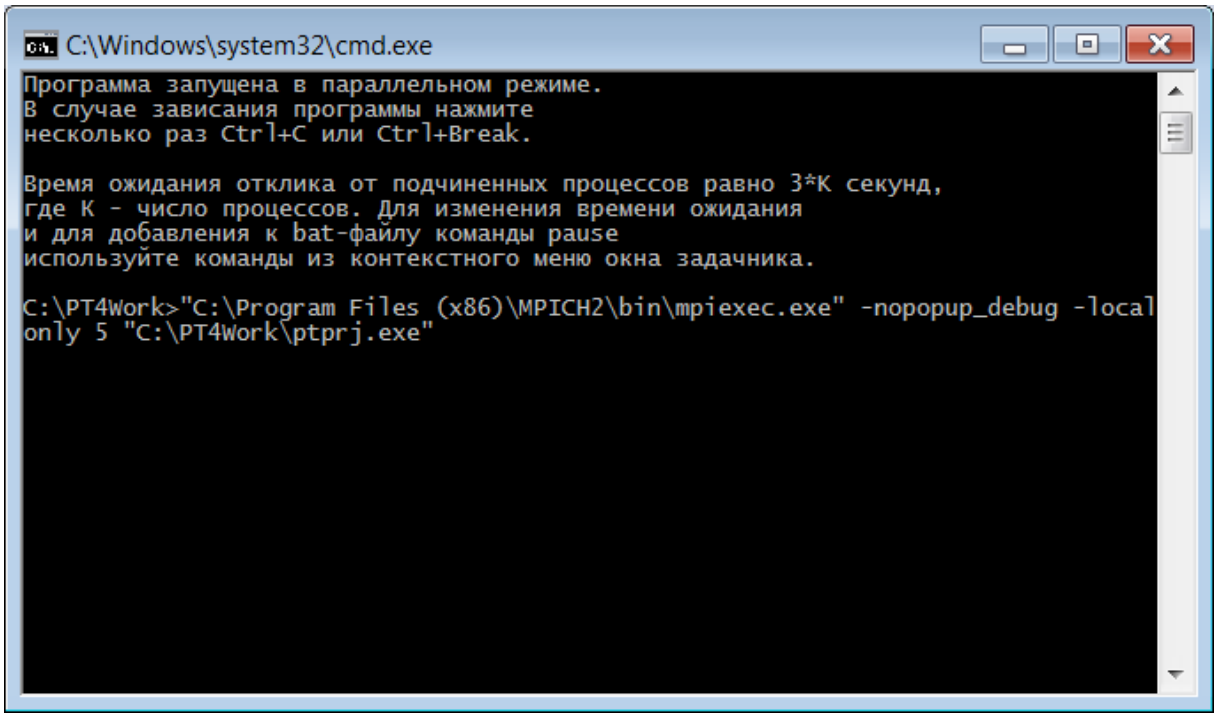


Рис. 5. Консольное окно с информацией о запуске программы в параллельном режиме

После нескольких строк информационного сообщения в этом окне отображается командная строка, которая обеспечивает запуск программы ptprj.exe в параллельном режиме под управлением mpiexec.exe. Число «5», указанное перед полным именем exe-файла, означает, что соответствующий процесс будет запущен в пяти экземплярах. Параметр –nopopup_debug отключает вывод сообщений об ошибках в отдельном окне (поскольку эти сообщения в конечном итоге будут выведены в окне задачника), параметр –localonly обеспечивает запуск всех экземпляров процесса на локальном компьютере.

Сразу после появления консольного окна, если ранее параллельная программа с именем ptprj.exe не запускалась, на экране может появиться еще одно окно (рис.).

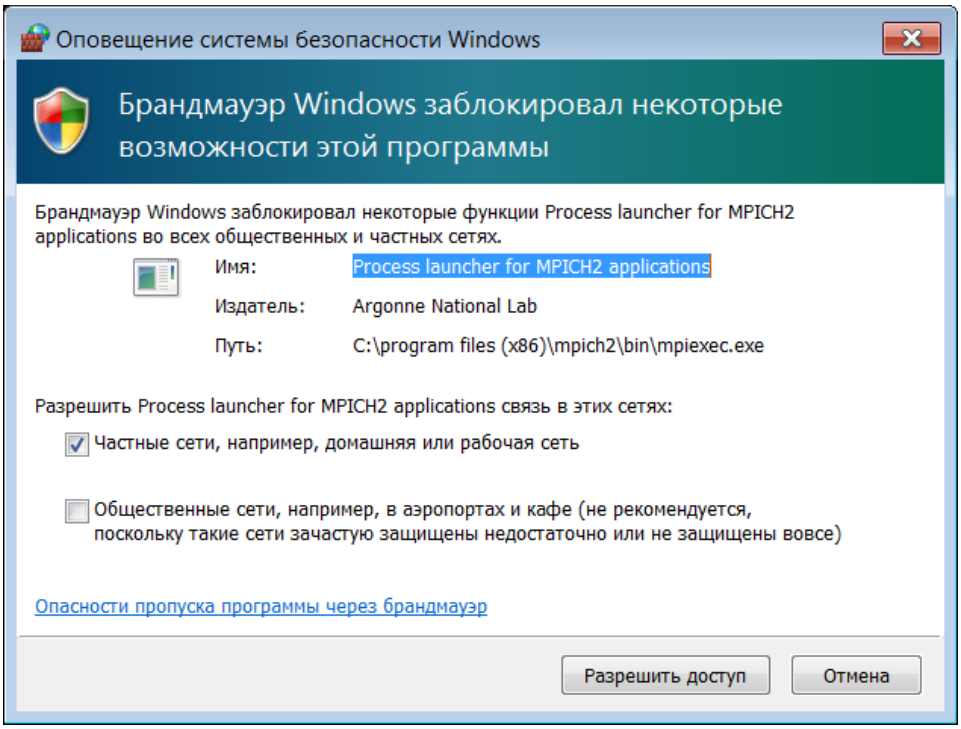


Рис. 6. Окно с запросом о блокировке запущенной параллельной программы

В этом окне следует выбрать вариант «Разрешить доступ». Наконец, на экране появится окно задачника (рис.).

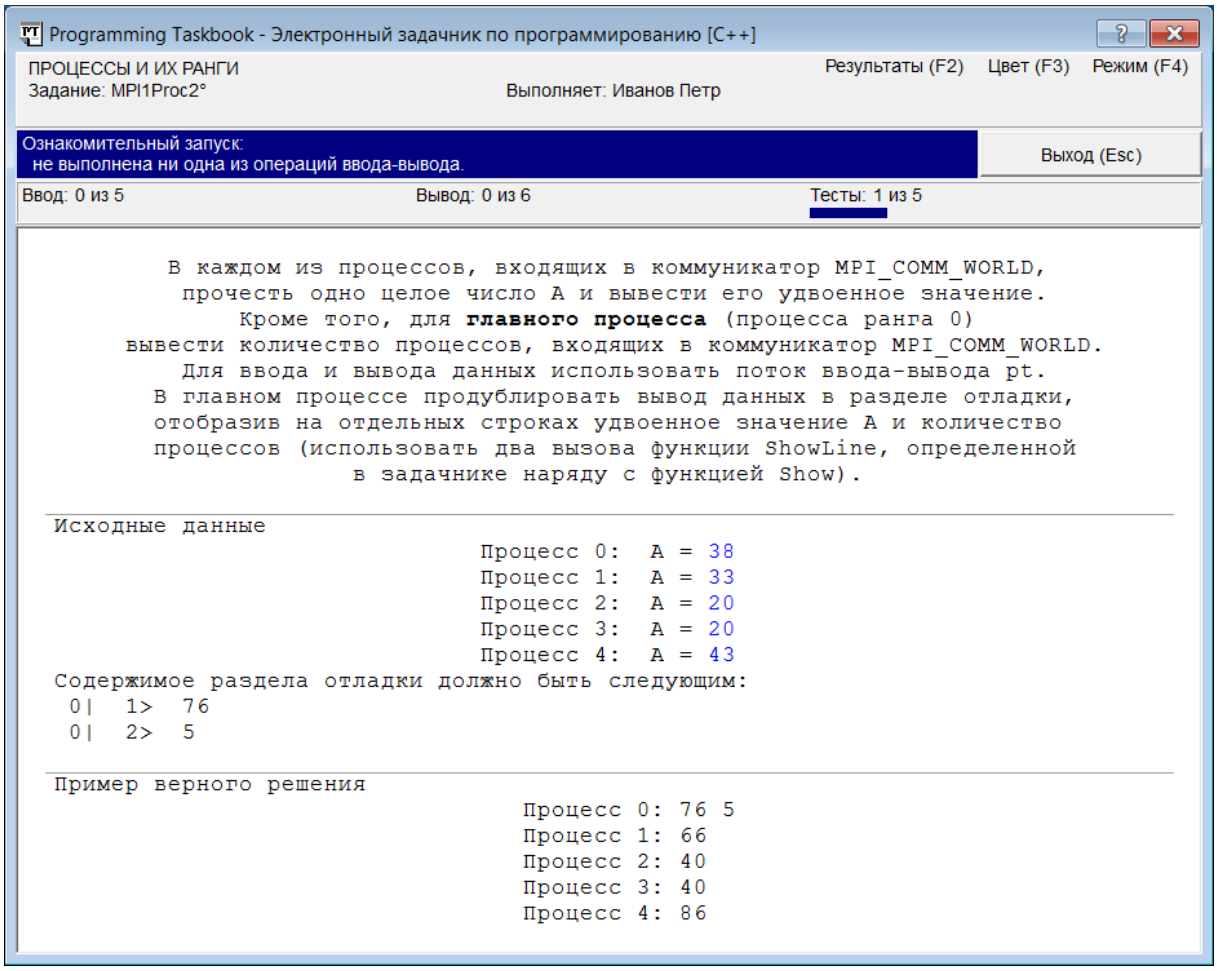


Рис. 7. Ознакомительный запуск задания MPI1Proc2

Внешне это окно ничем не отличается от окна, возникающего при выполнении обычной, «непараллельной» программы. Однако отличие имеется: в данном случае информация о том, что не была выполнена ни одна из операций ввода-вывода, относится *ко всем процессам, запущенным в параллельном режиме*.

Для завершения работы программы надо, как обычно, закрыть окно задачника (например, щелкнув мышью на кнопке «Выход (Esc)» или нажав клавишу Esc). После закрытия окна задачника немедленно закроется и консольное окно, и мы вернемся в интегрированную среду, из которой была запущена наша программа.

Таким образом, откомпилировав и запустив программу из интегрированной среды, мы смогли сразу обеспечить ее выполнение в параллельном режиме. Это происходит благодаря достаточно сложному механизму, который реализован в ядре задачника Programming Taskbook. Для того чтобы успешно выполнять учебные задания, не требуется детального понимания этого механизма, поэтому дадим здесь лишь его краткое описание (подробности приведены в п.).

«На самом деле» программа, запущенная из интегрированной среды, не пытается решить задачу и выполняется в обычном, «непараллельном» режиме. Обнаружив, что задача относится к группе заданий по параллельному программированию, она лишь создает пакетный файл pt_run.bat, записывая в него строки комментария и командную строку, обеспечивающую вызов программы mpiexec.exe с необходимыми параметрами, после чего запускает этот пакетный файл на выполнение и переходит в режим ожидания завершения работы пакетного файла. Запущенная с помощью пакетного файла программа mpiexec.exe запускает, в свою очередь, нужное количество экземпляров программы (процессов) в параллельном режиме, и эти процессы действительно пытаются решить задачу. В частности, задачник предлагает каждому процессу его набор исходных данных и ожидает от него набор результатов.

Поскольку в нашем случае ни в одном процессе не была указана ни одна операция ввода-вывода, данный запуск параллельной программы был признан ознакомительным, о чем и было сообщено в информационном разделе окна задачника. Отметим, что данное окно отображается *главным процессом* параллельной программы, в то время как все подчиненные процессы (а также самый первый экземпляр программы, обеспечивший создание и запуск пакетного файла) работают в «невидимом» режиме.

При закрытии окна задачника происходит завершение всех процессов параллельной программы, после этого завершается выполнение пакетного файла и, наконец, обнаружив, что пакетный файл успешно завершил работу, завершает работу и тот экземпляр нашей программы, который был запущен из интегрированной среды.

> **Примечание 2**. «Стартовый» экземпляр программы обеспечивает выполнение еще одного действия: он автоматически выгружает из памяти все процессы параллельной программы, если в результате неправильного программирования происходит их «зависание». Если при выполнении параллельной программы в течение 20–30 секунд окно задачника не появляется, то это, как правило, означает, что она зависла (иногда зависание программы проявляется в том, что после закрытия окна задачника не происходит немедленного закрытия консольного окна, т. е. завершения работы пакетного файла). В этой ситуации надо закрыть консольное окно, следуя приведенным в нем указаниям — нажав несколько раз комбинацию клавиш Ctrl+C или Ctrl+Break. Если стартовый экземпляр программы обнаружит, что пакетный файл завершил свою работу, а в памяти остались зависшие процессы параллельной программы, то он автоматически выгрузит из памяти все эти процессы. Это действие является важным, так как, пока в памяти остаются процессы, связанные с исполняемым файлом, этот файл нельзя

изменить (в частности, удалить или заменить на новый откомпилированный вариант).

Примечание 3. Иногда зависают лишь некоторые подчиненные процессы параллельного приложения. В этом случае главный процесс обычно выводит свое окно и сообщает, в каких именно подчиненных процессах произошло зависание (а также выводит результаты, полученные из тех подчиненных процессов, которые не зависли). Такая информация может оказаться полезной при поиске и исправлении ошибок.

Главный процесс считает подчиненный процесс зависшим, если он не получает от него отклика в течение некоторого промежутка времени (пропорционального количеству процессов). По умолчанию это промежуток равен $3*K$ секунд, где K — количество процессов (об этом сообщается в комментарии, который выводится в консольном окне). В некоторых, очень редких, случаях при выполнении заданий на компьютерах с низким быстродействием возможна ситуация, когда некоторые подчиненные процессы «не успеют» завершить свою часть работы за отведенное время ожидания, и главный процесс сочтет их зависшими, хотя решение задачи является правильным. В таких случаях можно увеличить время ожидания, используя команды контекстного меню окна задачника «Увеличить время ожидания отклика от подчиненных процессов» и «Уменьшить время ожидания отклика от подчиненных процессов».

1.5. Выполнение задания MPI1Proc2

Перейдем к выполнению задания. Теперь, когда мы подробно познакомились с механизмом работы программы в параллельном режиме, решение этой простой задачи не будет представлять для нас особых проблем.

Начнем с ввода исходных данных. По условию в каждом процессе дано по одному целому числу. Перейдем на пустую строку, расположенную ниже вызова функции MPI_Comm_rank. Если при выполнении программы будет достигнут данный участок кода, то это означает, что программа была запущена как один из процессов параллельного приложения (в противном случае был бы выполнен оператор выхода, указанный в условном операторе). Таким образом, в этом месте программы можно ввести элемент исходных данных, предварительно описав его (здесь и далее будем приводить только содержимое функции 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);
int n;
pt >> n;
```

Для ввода исходных данных мы использовали специальный поток ввода pt, определенный в задачнике (см. п. 8.2). Это поток позволяет вводить данные любых скалярных типов, в частности, int и double, которые обычно и требуются при выполнении заданий из задачника PT for MPI-2.

Запустив полученную программу, мы увидим на экране окно задачника (см. рис.).

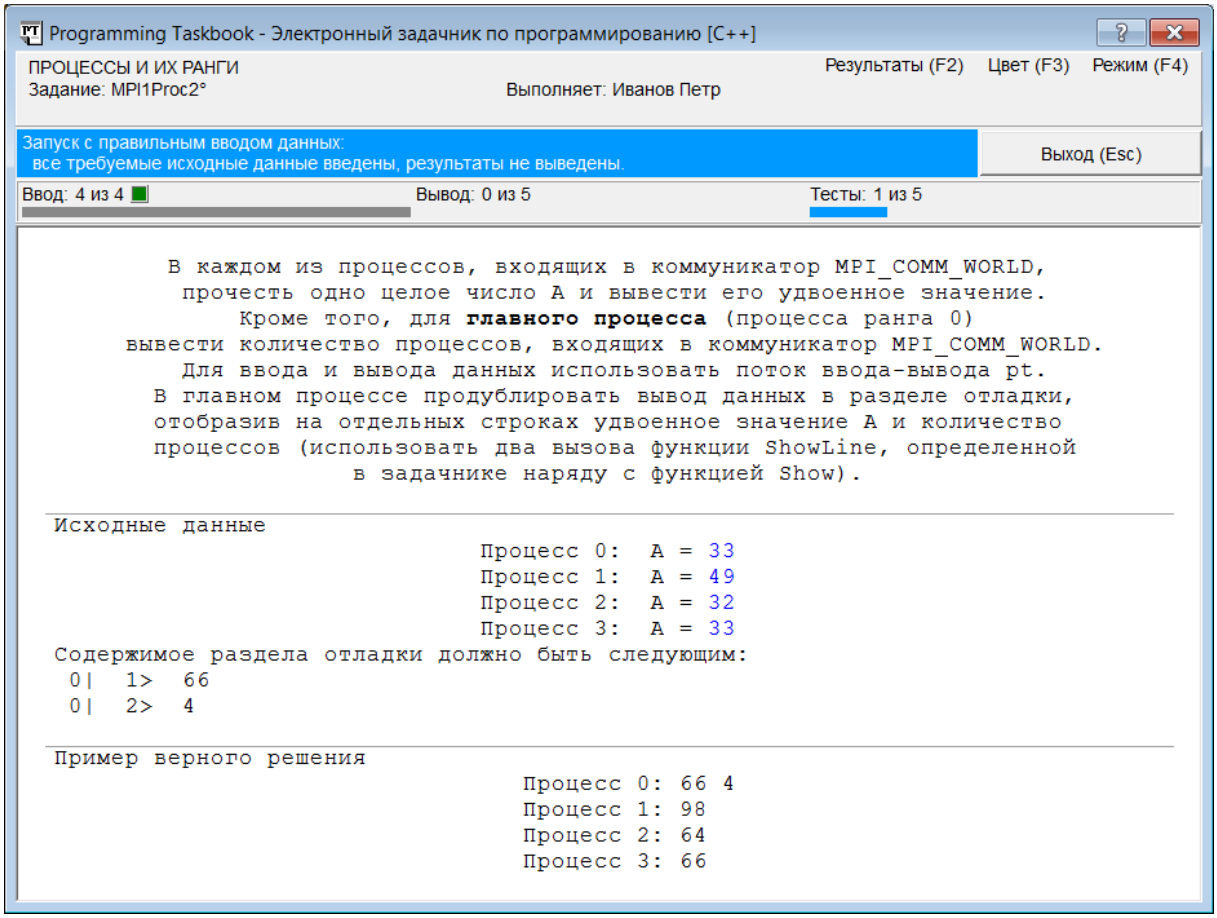


Рис. 8. Окно задачника с информацией о правильном вводе исходных данных

Задачник обнаружил, что ввод данных выполнен, и, таким образом, программа приступила к решению задачи. Однако ни один результирующий элемент данных не был выведен. Строго говоря, это свидетельствует об ошибочном решении, однако первый шаг к правильному решению нами сделан: правильно введены все исходные данные. В подобной ситуации задачник выводит сообщение на светло-синем фоне «*Запуск с правильным вводом данных: все требуемые данные введены, результаты не выведены*»

Обратите внимание на то, что добавленные нами действия по вводу данных выполняются *во всех процессах параллельного приложения*. Отметим также, что при данном запуске программы количество процессов стало другим. При выполнении задания число процессов может изменяться, поэтому решение будет считаться правильным только в случае, если оно дает верный результат при любом допустимом количестве процессов.

Закроем окно задачника и вернемся к нашей программе. В каждом процессе нам надо вывести удвоенное значение введенного числа, поэтому добавим в конец функции Solve еще один оператор:

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

Для вывода данных при решении задач используется тот же поток pt; таким образом, этот поток является *потоком ввода-вывода*.

Запуск исправленного варианта приведет к появлению окна с сообщением об ошибке (рис.).

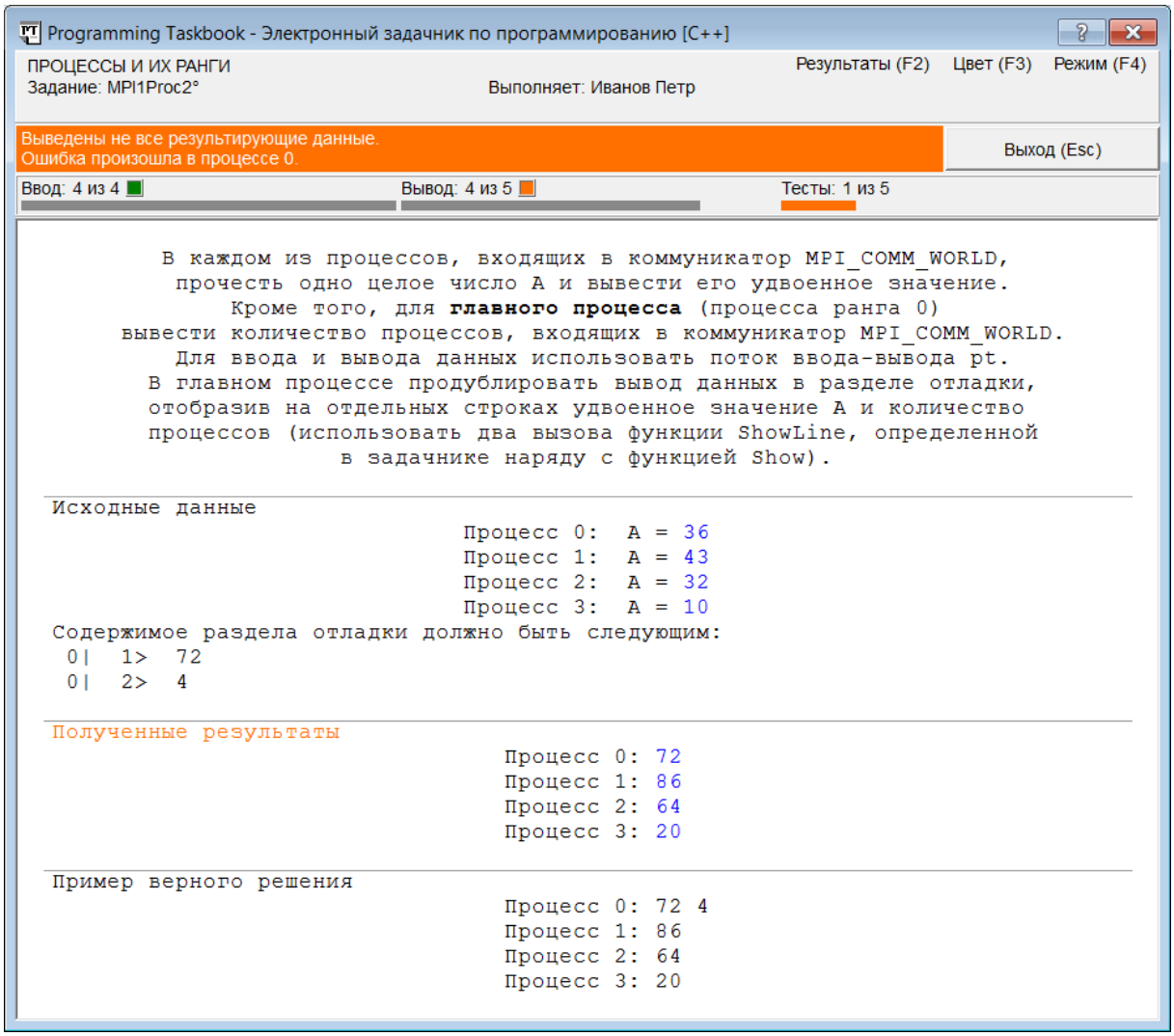


Рис. 9. Окно задачника с информацией об ошибке в главном процессе

Теперь во всех подчиненных процессах выведены требуемые результаты. Кроме того, удвоенное число выведено и в главном процессе. Эти данные являются правильными, в чем можно убедиться, сравнив значения, выведенные в разделе результатов и показанные в разделе с примером верного решения.

Однако в главном процессе требовалось также вывести количество процессов, входящих в коммуникатор, а это сделано не было. Поэтому информационная панель содержит сообщение «*Выведены не все результирующие данные. Ошибка произошла в процессе 0*», причем сообщение выводится на оранжевом фоне. Оранжевый цвет используется для выделения ошибок, связанных с вводом или выводом *недостаточного* количества данных. При попытке ввести или вывести *лишние* данные информационная панель выделяется малиновым цветом, если возникают ошибки, связанные с использованием данных *неверного типа*, то цвет панели становится фиолетовым. Для всех прочих ошибок используется красный цвет.

Количество процессов хранится в переменной size. Попытаемся вывести значение этой переменной в конце функции Solve:

```
pt << size;
```

Окно задачника примет вид, приведенный на рис. .

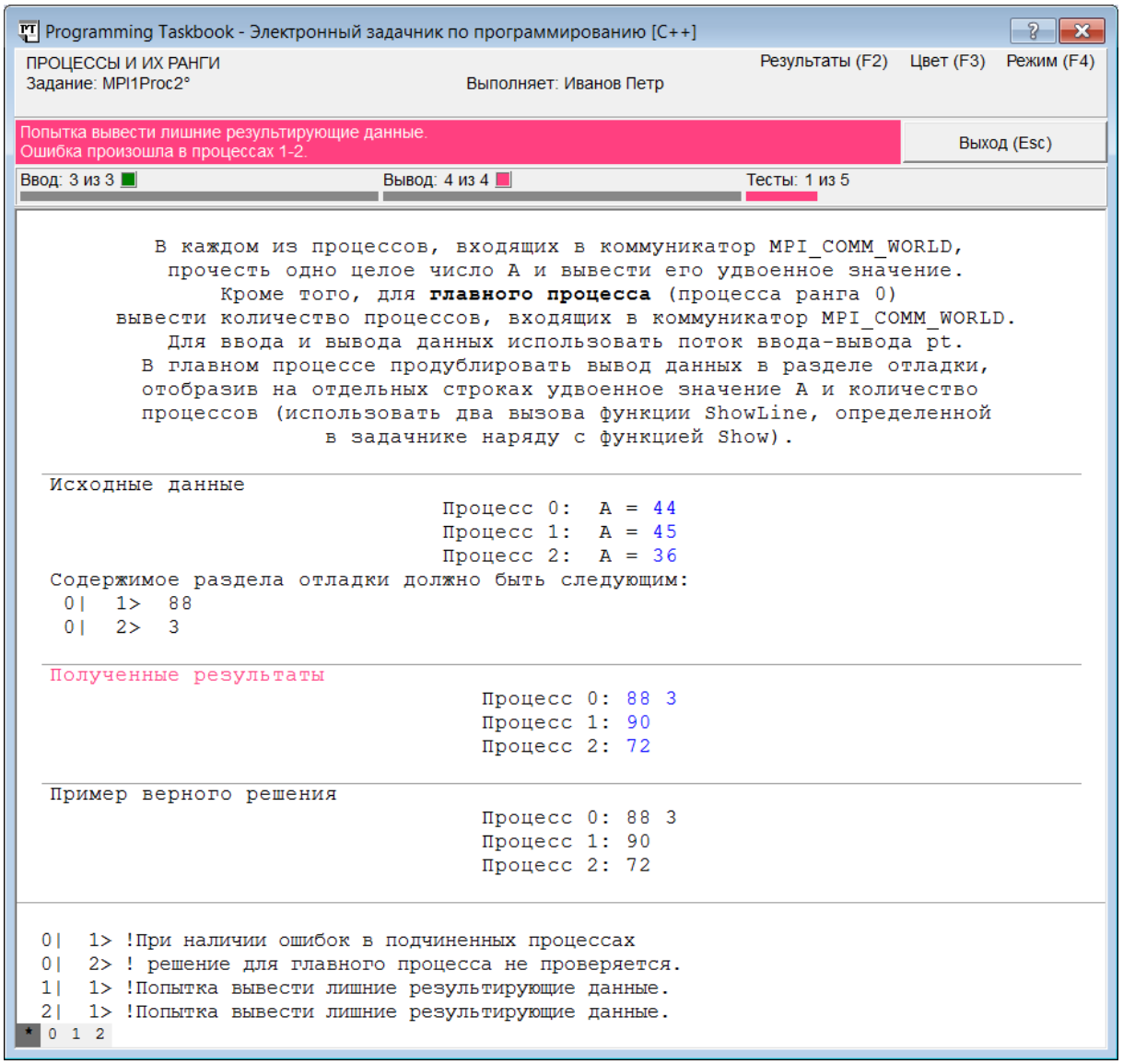


Рис. 10. Окно задачника с информацией о попытке вывода лишних данных

Можно убедиться в том, что все результирующие данные выведены. Однако решение по-прежнему считается ошибочным, поскольку теперь мы попытались вывести лишние данные (а именно значение size) в *подчиненных* процессах. Как уже отмечалось выше, для выделения ошибок, связанных с попыткой ввода или вывода лишних данных, используется малиновый цвет.

Если ошибки обнаружены в подчиненных процессах, то в окне задачника отображается дополнительный *раздел отладки*, в котором для каждого подчиненного процесса выводится более подробная информация об ошибке.

Определить, с каким процессом связано то или иное сообщение, выведенное в разделе отладки, можно по номеру, указываемому в левой части строки (перед символом «|»). Все строки, связанные с определенным процессом, нумеруются независимо от остальных строк; их номера указы-

ваются после номера процесса и отделяются от текста сообщения символом «>». Для того чтобы отобразить в разделе отладки только сообщения, связанные с каким-либо одним процессом, достаточно щелкнуть мышью на маркере с номером (рангом) этого процесса (все маркеры расположены на нижней границе окна) или нажать соответствующую цифровую клавишу. Для отображения сводной информации по всем процессам надо выбрать маркер с символом «*» или ввести этот символ с клавиатуры (перебирать маркеры можно также с помощью клавиш со стрелками [←] и [→]). Если строка сообщения в разделе отладки начинается с символа «!», то это означает, что данное сообщение является *сообщением об ошибке* и добавлено в раздел отладки самим задачником. Программа учащегося может выводить в раздел отладки свои собственные сообщения; об этой возможности будет подробно рассказано далее (см. также п. 8.3).

Если задачник обнаружил ошибку хотя бы в одном подчиненном процессе, то он не анализирует результат, полученный в главном процессе (об этом также сообщается в разделе отладки).

Для того чтобы значение size было выведено только в главном процессе, необходимо перед выполнением этого действия убедиться, что ранг текущего процесса равен 0. Добавив соответствующую проверку, мы получим решение, который задачник сочтет правильным:

```
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;
```

При запуске этого варианта решения на экране будут последовательно выводиться пять консольных окон, каждое из которых связано с выполняемой параллельной программой. Таким образом, однократный запуск программы из среды разработки приводит к целой серии запусков этой программы в параллельном режиме, что позволяет сразу протестировать полученное решение на нескольких наборах исходных данных. Серия тестовых испытаний завершается либо при обнаружении какой-либо ошибки, либо при успешном прохождении требуемого количества тестов (для всех заданий, входящих в задачник PT for MPI-2 количество тестовых испыта-

ний равно пяти). Данная возможность еще более упрощает процесс проверки правильности полученного решения задачи.

После пяти успешных тестовых испытаний на экране появится окно задачника с сообщением об успешном выполнении задания (рис.).

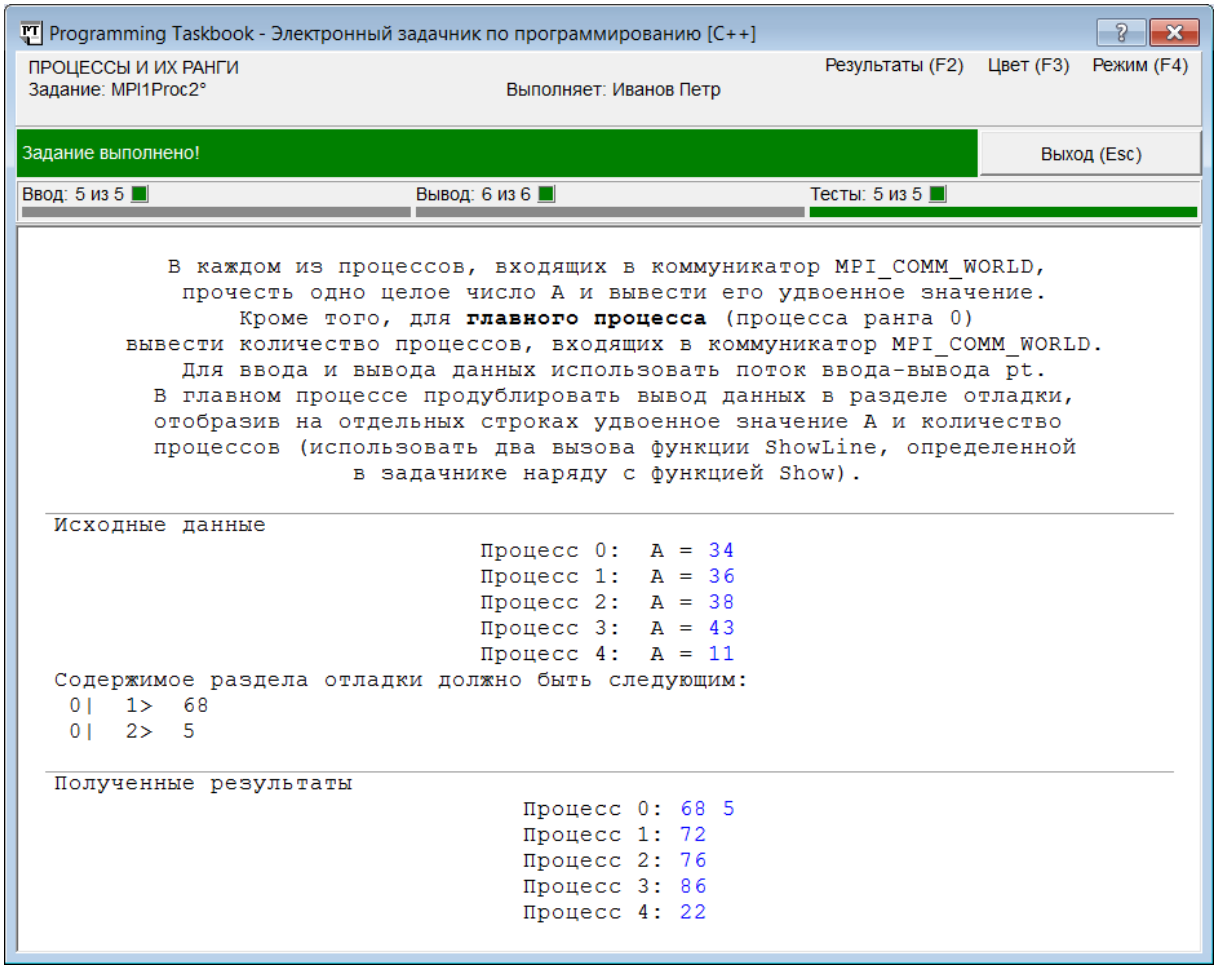


Рис. 11. Окно с сообщением об успешном выполнении задания MPI1Proc2

В данном случае все индикаторы, расположенные на панели индикаторов (под информационной панелью), имеют зеленый цвет.

При каждом запуске учебной программы задачник сохраняет результаты ее работы в специальном файле результатов results.dat. Этот файл можно просмотреть с помощью модуля PT4Results, входящего в состав задачника (для запуска этого модуля в рабочем каталоге предусмотрен ярлык Results.lnk). Кроме того, результаты можно просмотреть и непосредственно из окна задачника, нажав в нем метку «Результаты (F2)» или клавишу F2. На экране появится окно с протоколом выполнения всех заданий. В нашем случае оно будет содержать примерно такой текст:

```
MPI1Proc2   c04/09 15:44 Ознакомительный запуск.
MPI1Proc2   c04/09 15:49 Запуск с правильным вводом данных.
MPI1Proc2   c04/09 15:54 Выведены не все результирующие данные.
MPI1Proc2   c04/09 15:59 Попытка вывести лишние результирующие данные.
```

```
MPI1Proc2   c04/09 16:03 Задание выполнено!
```

После имени задания указывается символ, соответствующей использованному языку программирования (в данном случае — символ «c», означающий, что использовался язык C++), дата и время запуска программы и описание результата ее выполнения.

1.6. Использование дополнительной информации в раздел отладки

Если внимательно проанализировать полученное решение, то можно заметить, что оно является неполным, поскольку в задании требуется вывести некоторые данные не только в разделе результатов, но и в разделе отладки.

Мы уже встречались с использованием раздела отладки: в нем выводилась дополнительная информация об ошибках, возникших в подчиненных процессах. Вторая задача этого раздела — дать возможность в процессе решения задачи выводить на экран различные отладочные данные. Эта возможность является особенно важной при разработке параллельных программ, так как для них нельзя использовать такие стандартные средства отладки, как точки останова и окна просмотра содержимого.

Дополнительная часть задания MPI1Proc2 (и других начальных заданий группы MPI1Proc) посвящена знакомству с различными вариантами вывода отладочной информации. Хотя задачник не анализирует содержимое раздела отладки, эта часть выполнения задания является такой же обязательной, как и вывод полученных результатов, просто проверяться она будет не самим задачником, а преподавателем. Задачник лишь отмечает, что «с его точки зрения» задание выполнено, окончательное решение о том, следует ли зачесть данный вариант решения, принимает преподаватель (при этом он, в частности, обращает внимание на то, какими средствами MPI решалась задача, является ли полученное решение эффективным, и т. п.). Заметим, что отображать данные в разделе отладки требуется также в заданиях группы MPI2Send, связанных с изучением неблокирующих функций пересылки данных, а также в заданиях группы MPI8Inter, связанных с возможностями динамического порождения процессов в ходе выполнения параллельной программы.

Напомним заключительную часть задания MPI1Proc2: «*В главном процессе продублировать вывод данных в разделе отладки, отобразив на отдельных строках удвоенное значение A и количество процессов (использовать два вызова функции ShowLine, определенной в задачнике наряду с функцией Show)*». Обратите также внимание на то, что в разделе результатов окна задачника выводится комментарий, поясняющий, как должен выглядеть раздел отладки при правильном решении (см. любой из приведенных в предыдущем пункте рисунков с окном задачника).

Для вывода данных в окне отладки в задачнике предусмотрены две функции: Show и ShowLine, каждая из которых имеет несколько перегруженных вариантов, позволяющих настраивать вид выводимых данных и снабжать их дополнительными комментариями (подробности приведены в п.). Отличаются эти функции тем, что функция ShowLine после вывода данных автоматически переходит на следующую строку раздела отладки, тогда как функция Show этого не делает (однако при достижении правой границы раздела отладки также происходит переход на новую строку).

> **Примечание.** Полное описание возможностей, связанных с выводом отладочной информации, приведено в информационном окне в разделе «Отладка». Если окно задачника активно, то для отображения информационного окна достаточно нажать кнопку  в правой части заголовка окна задачника или клавишу F1.

Для получения нужного содержимого окна отладки нам достаточно добавить в конец функции Solve по одному вызову функций ShowLine и Show. Поскольку требуемые данные надо вывести только в той части раздела отладки, которая связана с главным процессом, вызовы этих функций следует поместить в уже имеющийся в программе условный оператор. Приведем заключительную часть функции Solve, содержащую полный текст решения задачи:

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

После запуска нового варианта и проверки его на пяти тестовых наборах данных на экране появится окно задачника (см. рис.).

Сравнивая содержимое окна отладки с образцом, приведенным в разделе результатов, убеждаемся, что теперь задание выполнено полностью. Обратите внимание на то, что поскольку в данном случае раздел отладки содержит только данные, выведенные в главном процессе, на нижней границе окна отображается единственный маркер «0», соответствующий этому процессу.

Итак, мы полностью выполнили задание MPI1Proc2. В процессе решения мы познакомились с действиями по созданию заготовки, изучили особенности выполнения параллельных программ и те возможности задачника, которые упрощают их запуск из среды разработки. Мы узнали о средствах задачника, предназначенных для ввода исходных данных, выво-

да результатов и отображения дополнительной информации в разделе отладки. Кроме того, мы увидели, как задачник реагирует на различные виды ошибок. Вся эта информация окажется полезной, когда мы будем выполнять задания, посвященные главному в технологии MPI: различным способам обмена сообщениями между процессами параллельного приложения.

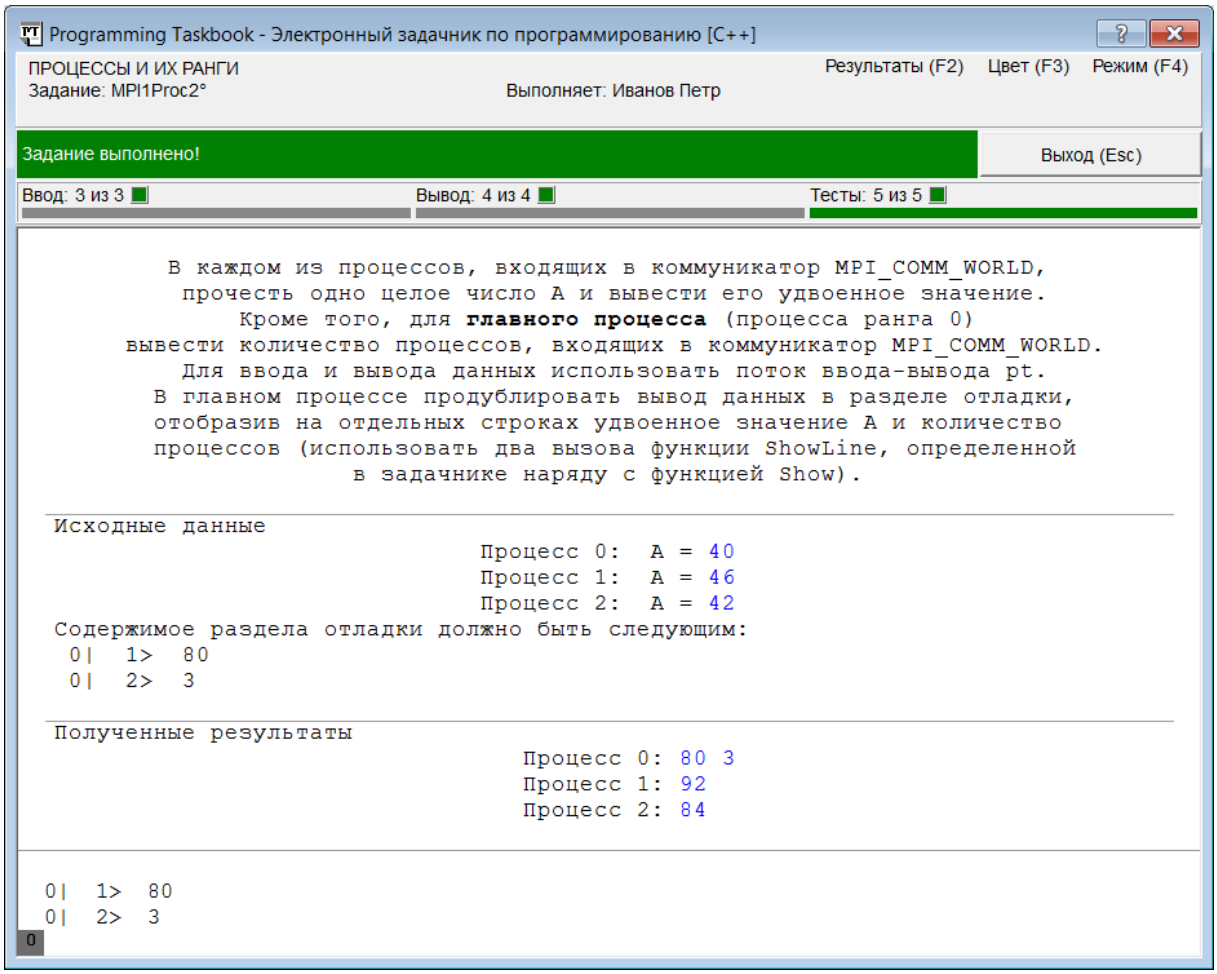


Рис. 12. Окно с полным вариантом решения

2. Примеры разработки параллельных программ

2.1. Пересылка сообщений между двумя процессами: MPI2Send11

Теперь рассмотрим задание, связанное с пересылкой сообщений между различными процессами параллельной программы, и познакомимся с особенностями используемых для этого функций MPI.

MPI2Send11. В каждом процессе дано вещественное число. Переслать число из главного процесса во все подчиненные процессы, а все числа из подчиненных процессов — в главный, и вывести в каждом процессе полученные числа (в главном процессе числа выводить в порядке возрастания рангов переславших их процессов). Для отправки сообщений использовать функцию MPI_Ssend.

Указание. Функция MPI_Ssend обеспечивает *синхронный режим* пересылки данных, при котором операция отправки сообщения будет завершена только после начала приема этого сообщения процессом-получателем. В случае пересылки данных в синхронном режиме возникает опасность *взаимных блокировок* (deadlocks) из-за неправильного порядка вызова функций отправки и получения сообщений.

Создадим проект-заготовку для выполнения этого задания и запустим полученную программу. Появившееся на экране окно задачника будет иметь вид, приведенный на рисунке 9.

Для чтения исходных данных нам будет достаточно использовать единственную переменную вещественного типа, поскольку в каждом процессе дано только одно вещественное число.

Исходные данные надо переслать в другие процессы параллельной программы. Для этого требуется использовать пару функций библиотеки MPI: одну для отправки сообщения, другую для его приема. Поскольку в данной подгруппе группы MPI2Send изучаются блокирующие варианты пересылки сообщений, для приема необходимо использовать функцию MPI_Recv. Для того чтобы отправить сообщение в блокирующем режиме, предусмотрено несколько видов функций; чаще всего используется функция MPI_Send, однако в нашем случае надо использовать функцию MPI_Ssend, поскольку об этом явным образом сказано в задании.

Функция MPI_Ssend (как и другие функции для отправки сообщения, например MPI_Send) вызывается передающим процессом и определяет, какому процессу и какие данные он собирается переслать. Функция MPI_Recv вызывается принимающим процессом; в ней указываются процесс-отправитель и переменная-буфер, в которую будут записаны полученные от него данные (подробное описание параметров этих функций см. в п. 7.3).

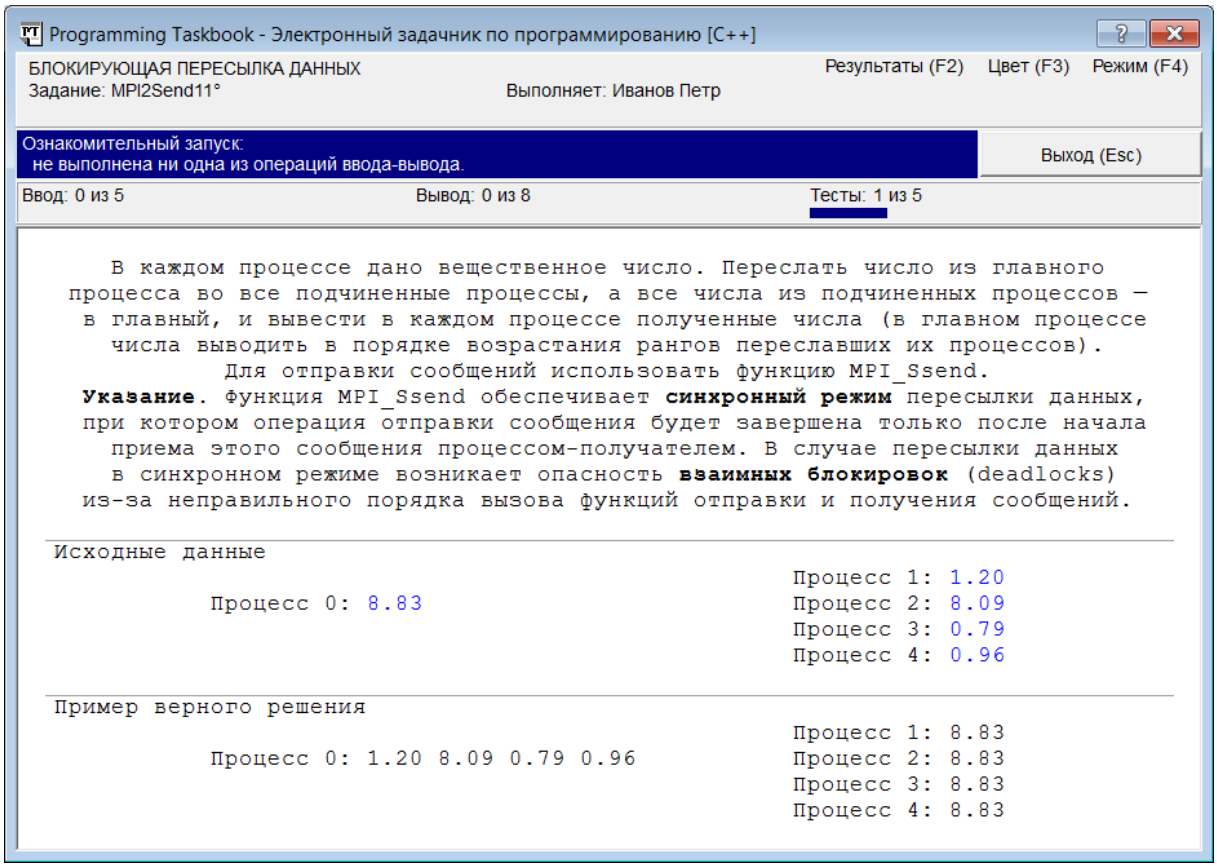


Рис. 13. Ознакомительный запуск задания MPI2Send11

Вначале займемся приемом и пересылкой данных для подчиненных процессов, не реализуя пока действия, которые надо выполнить в главном процессе.

Добавим в конец функции Solve следующий фрагмент кода:

```
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;
}
```

Обратите внимание на то, что в качестве первого параметра обеих функций указывается *адрес* той переменной, которая содержит (или должна принять) пересылаемые данные.

Запустим нашу программу. Через 20–30 с после появления консольного окна с информацией о том, что программа запущена в параллельном режиме, на экране появится окно задачника с сообщением об ошибке в подчиненных процессах (рис.).

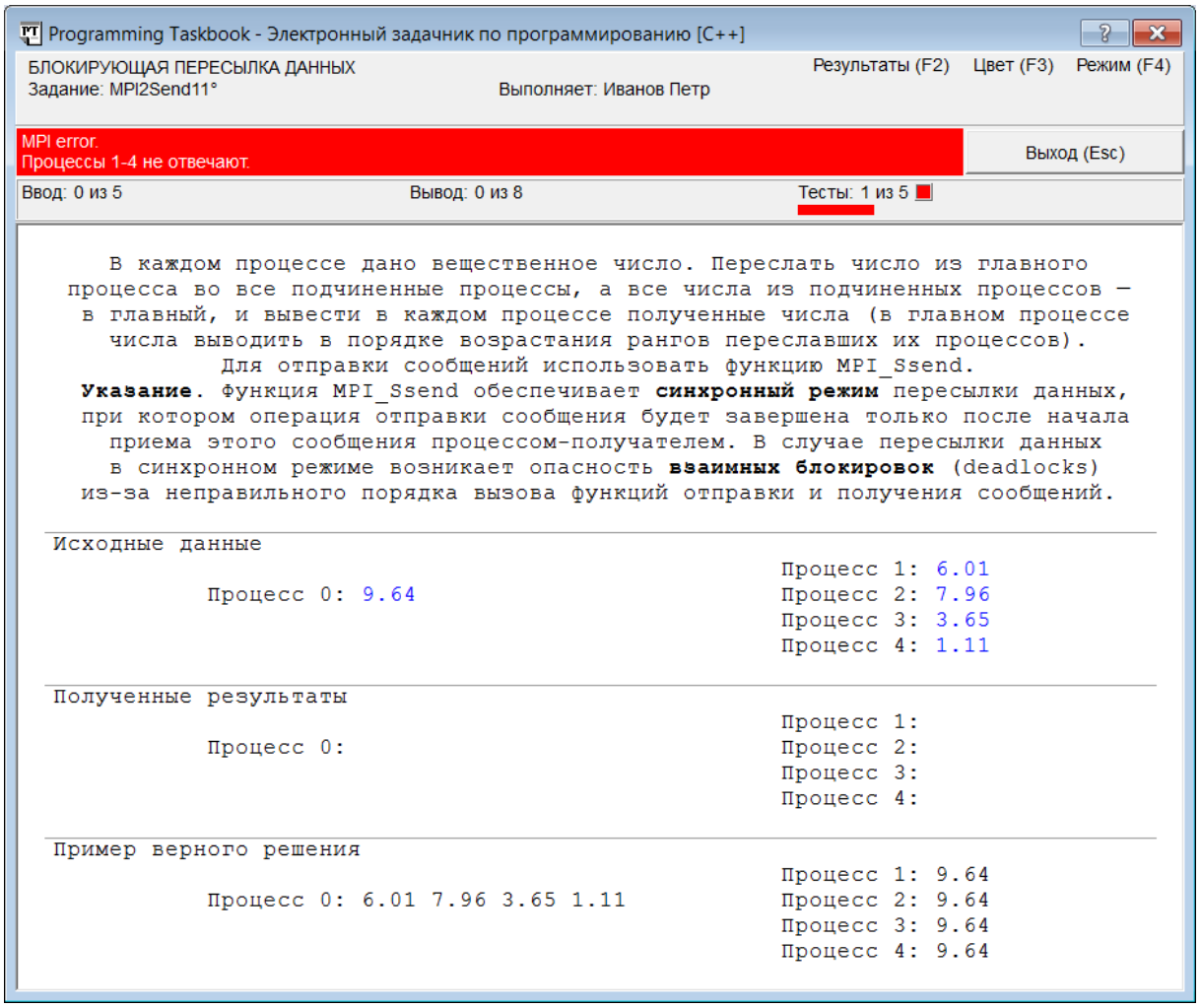


Рис. 14. Окно задачника с информацией о блокировке подчиненных процессов

Сообщение об ошибке вида «*MPI error. Процессы 1–4 не отвечают*» означает, что главный процесс нашей параллельной программы не смог в течение определенного времени «связаться» с подчиненными процессами с целью получить от них информацию о введенных и выведенных в них данных (о том, как определить и изменить время ожидания отклика от подчиненных процессов, см. примечание 3 в п. 10.4).

Как следует из второй строки сообщения, ошибка возникла при попытке связаться со всеми подчиненными процессами (которых при данном запуске программы было четыре).

Причина ошибки состоит в том, что функция MPI_Ssend отправки сообщения будет ожидать, пока принимающий (в данном случае главный) процесс вызовет соответствующую функцию приема (MPI_Recv), и только после этого выполнит пересылку данных с завершит свою работу (говорят, что функция MPI_Ssend обеспечивает *синхронный* режим пересылки). Но в нашей программе пока не содержится вызова функции MPI_Recv в главном процессе. Поэтому ожидание функции MPI_Ssend будет длиться вечно (точнее, пока выполнение подчиненных процессов не будет прекращено

«насильственным образом»). Это пример *зависания* параллельной программы, возникающего обычно из-за того, что один или несколько процессов блокируются в ожидании информации, которая им не послана (в данном случае функция MPI_Ssend ожидает информацию о том, что главный процесс приступил к действиям по приему данных).

Заметим, что если бы мы использовали для отправки сообщения другую функцию, например MPI_Bsend, которая не дожидается информации от принимающего процесса, а просто пересылает данные в специальный буфер отправки и сразу после этого завершает работу (так называемая *пересылка в буферизованном режиме*), то программа все равно бы зависла, но уже по другой причине: теперь функция MPI_Recv вечно ждала бы тех данных, которые должен был отправить ей главный процесс.

Обратите внимание на то, что при закрытии окна задачника консольное окно останется на экране. Причина понятна: консольное окно управляется программой mpiexec, которая завершает работу только при завершении *всех* процессов запущенной параллельной программы, а в данном случае завершился только главный процесс (подчиненные процессы остаются заблокированными). Для завершения программы mpiexec и закрытия консольного окна необходимо нажать несколько раз комбинацию клавиш Ctrl+C или Ctrl+Break (как сказано в комментарии, выведенном в консольном окне).

> **Примечание 1**. При «аварийном» завершении программы mpiexec в памяти могут остаться запущенные (и зависшие) процессы параллельной программы. Это в дальнейшем будет препятствовать перекомпиляции нашей программы, поскольку, пока процесс находится в памяти, связанный в ним exe-файл недоступен для изменения. Однако при выполнении заданий с использованием задачника PT for MPI-2 такой проблемы не возникает: вспомним о том, что программу mpiexec запустила наша программа (которая сама была запущена из интегрированной среды). Этот «непараллельный» экземпляр нашей программы остается в памяти, пока программа mpiexec не завершит работу, после чего он выгружает из памяти все зависшие процессы. Если бы эта полезная работа не выполнялась задачником, то выгружать каждый из зависших процессов пришлось бы вручную, используя диспетчер задач Windows.

Итак, мы познакомились с ситуацией, когда один или несколько подчиненных процессов оказываются заблокированными. Такая же «неприятность» может произойти и с главным процессом. Дополним нашу программу фрагментом, связанным с главным процессом, причем в этом фрагменте также организуем вызов MPI-функций в том же самом («естественном») порядке — вначале отправка данных, затем прием:

```
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;
  }
}
```

Если запустить эту программу, то после появления консольного окна можно ожидать сколько угодно, однако окно задачника на экране не появится. Это связано с тем, что заблокированным оказался главный процесс нашей параллельной программы: перед отображением окна задачника главный процесс должен выполнить тот фрагмент программы, который разработан для него учащимся, а в нашем случае этот фрагмент привел к блокировке. Поэтому главный процесс просто не дошел до того места программы, в котором выполняется вывод окна задачника на экран.

Почему опять возникает блокировка? Ведь теперь, казалось бы, каждый процесс готов и отправить, и принять сообщение. Однако для того чтобы завершилась функция MPI_Ssend, реализующая *синхронный* режим отправки, необходимо, чтобы в процессе-получателе *уже была вызвана* функция приема MPI_Recv, а дойти до этой функции процесс-получатель не может, так как в нем тоже вызвана функция MPI_Ssend. Такое явление носит название *взаимной блокировки* (англ. deadlock).

Если в течение 20–30 с окно задачника не появилось, то можно считать, что произошло зависание главного процесса. В такой ситуации, как и в ситуации, описанной ранее, необходимо явным образом прервать выполнение параллельной программы, нажав несколько раз Ctrl+C или Ctrl+Break.

> **Примечание 2**. Любой из описанных выше «аварийных» способов завершения программы фиксируется задачником в файле результатов. Однако в случае если произошло зависание только подчиненных процессов (и на экране появилось окно задачника), в файл результатов будет записан текст «*MPI error*», тогда как в случае зависания главного процесса текст будет другим: «*Выполнение задания прервано*».

Простейший способ исправить нашу программу — это стереть вторую букву «s» в имени *хотя бы одной* функции MPI_Ssend, т. е. заменить либо в подчиненных, либо в главном процессе вызов функции MPI_Ssend на вызов функции MPI_Send, реализующей не синхронный, а *стандартный* режим отправки данных. Это связано с тем, что в библиотеке MPI системы MPICH стандартный режим, как и режим с буферизацией, использует бу-

фер для хранения отправляемых данных (только в отличие от режима с буферизацией этот стандартный буфер создается автоматически). После пересылки данных в стандартный буфер функция MPI_Send завершает работу, даже если к этому моменту принимающий процесс не вызвал функцию MPI_Recv. Получается такая последовательность действий (будем считать, что мы изменили функцию MPI_Ssend на MPI_Send в подчиненных процессах). В каждом из подчиненных процессов вызывается функция MPI_Send; она обеспечивает копирование пересылаемых данных в стандартный буфер, после чего немедленно завершает работу; после этого вызывается функция MPI_Recv, которая ожидает приема данных от главного процесса. В это же время в главном процессе в цикле вызывается функция MPI_Ssend, которая приостанавливает выполнение, пока в подчиненных процессах не будет вызвана функция MPI_Recv. Но функция MPI_Recv в подчиненных процессах обязательно будет вызвана, в этот момент функция MPI_Ssend в главном процессе осуществляет отправку данных и завершает работу. Таким образом, все функции MPI_Ssend в цикле успешно проработают, после чего в главном процессе во втором цикле будут вызваны функции MPI_Recv, которые примут те данные от подчиненных процессов, которые ранее были помещены в стандартный буфер. Наконец, после завершения работы функций MPI_Ssend в главном процессе успешно выполнятся и ожидающие получения данных функции MPI_Recv в подчиненных процессах. Итак, никакой взаимной блокировки не произойдет.

При запуске исправленной программы она будет успешно протестирована на пяти наборах исходных данных, после чего будет выведено окно задачника с сообщением о том, что задание выполнено (см. рис.).

Однако описанное выше исправление не вполне соответствует условию задачи, поскольку в условии требовалось использовать только функции MPI_Ssend. Вариант исправления с сохранением функции MPI_Ssend состоит в том, что *либо* во фрагменте программы для подчиненных процессов, *либо* во фрагменте программы для главного процесса надо изменить *порядок вызова* функций отправки и приема сообщения. Например, можно изменить порядок вызова функций в главном процессе:

```
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);
}
```

В данной ситуации взаимная блокировка также будет отсутствовать. Действительно, в главном процессе сразу вызываются функции MPI_Recv, поэтому в подчиненном процессе соответствующие функции MPI_Ssend успешно проработают и передадут данные в главный процесс, которые в нем будут приняты. Затем в подчиненных процессах, в свою очередь, будут вызваны функции MPI_Recv, которые позволят успешно проработать функциям MPI_Ssend в главном процессе.

> **Примечание 3**. Описанный вариант исправления имеет еще одно преимущество. Дело в том, что в стандарте MPI *не гарантируется*, что функция MPI_Send *обязательно* будет использовать буфер для промежуточного хранения пересылаемых данных. Это определяется самой исполняющей средой MPI, поэтому возможна ситуация, когда функция MPI_Send будет применять не буферизованный, а синхронный режим пересылки; в этом случае по-прежнему будет возникать взаимная блокировка.

Полученную программу можно упростить, если в разделе else воспользоваться вспомогательной вещественной переменной b для получения данных из подчиненных процессов. Это позволит разместить оператор ввода `pt >> a` *перед* условным оператором, а также даст возможность выполнить все действия в разделе else в единственном цикле. Приведем соответствующий вариант решения:

```
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);
    }
  }
```

Еще одного небольшого упрощения можно добиться, удалив описание переменной s типа MPI_Status и заменив параметр &s в функциях MPI_Recv на специальный указатель-«заглушку» MPI_STATUS_IGNORE. Значение MPI_STATUS_IGNORE удобно использовать в ситуации, когда в программе не требуется обращаться к информации, предоставляемой параметром типа MPI_Status.

> **Примечание 4**. Константа MPI_STATUS_IGNORE появилась в стандарте MPI-2, поэтому при использовании системы MPICH 1.2.5, реализующей стандарт MPI-1, ее применять не следует.

Заметим, что более эффективное решение этого задания можно получить, используя коллективные операции пересылки данных (см. п. 7.6).

2.2. Операции редукции и составные типы данных: MPI3Coll23

При выполнении очередного задания мы познакомимся с коллективными операциями редукции и с примером использования в MPI-программах составных типов данных (структур).

MPI3Coll23. В каждом процессе дан набор из $K + 5$ чисел, где K — количество процессов. Используя функцию MPI_Allreduce для операции MPI_MINLOC, найти минимальное значение среди элементов данных наборов с одним и тем же порядковым номером и ранг процесса, содержащего минимальное значение. Вывести в главном процессе минимумы, а в остальных процессах — ранги процессов, содержащих эти минимумы.

Большая группа функций MPI предназначена для организации *коллективного взаимодействия процессов* (см. п. 7.6). «Коллективные» MPI-функции, в отличие от функций MPI_Send, MPI_Ssend и MPI_Recv (рассмотренных в предыдущем пункте), позволяют организовать обмен сообщениями не между двумя отдельными процессами (отправителем и получателем), а между всеми процессами, входящими в некоторый коммуникатор. В частности, при использовании коммуникатора MPI_COMM_WORLD можно организовать коллективный обмен сообщениями между всеми запущенными процессами параллельной программы.

Среди коллективных MPI-функций выделяют группу функций, обеспечивающих выполнение *операций редукции*, т. е. операций, связанных с пересылкой не исходных данных, а результатов их обработки некоторой *групповой операцией*: нахождением суммы MPI_SUM, произведения MPI_PROD, максимального MPI_MAX или минимального MPI_MIN значения и т. д. — всего в стандарте MPI предусмотрено 12 операций редукции (см. п. 7.1), кроме того, программист может определять и свои собственные операции. Среди операций редукции особое место занимают операции MPI_MAXLOC и MPI_MINLOC, позволяющие найти не только максимальный или минимальный элемент среди элементов, предоставленных каждым процессом, но и его номер (в качестве номера обычно используется ранг процесса, предоставившего этот экстремальный элемент).

При запуске программы-заготовки, созданной для выполнения задания MPI3Coll23, мы увидим на экране окно задачника, подобное приведенному на рис. .

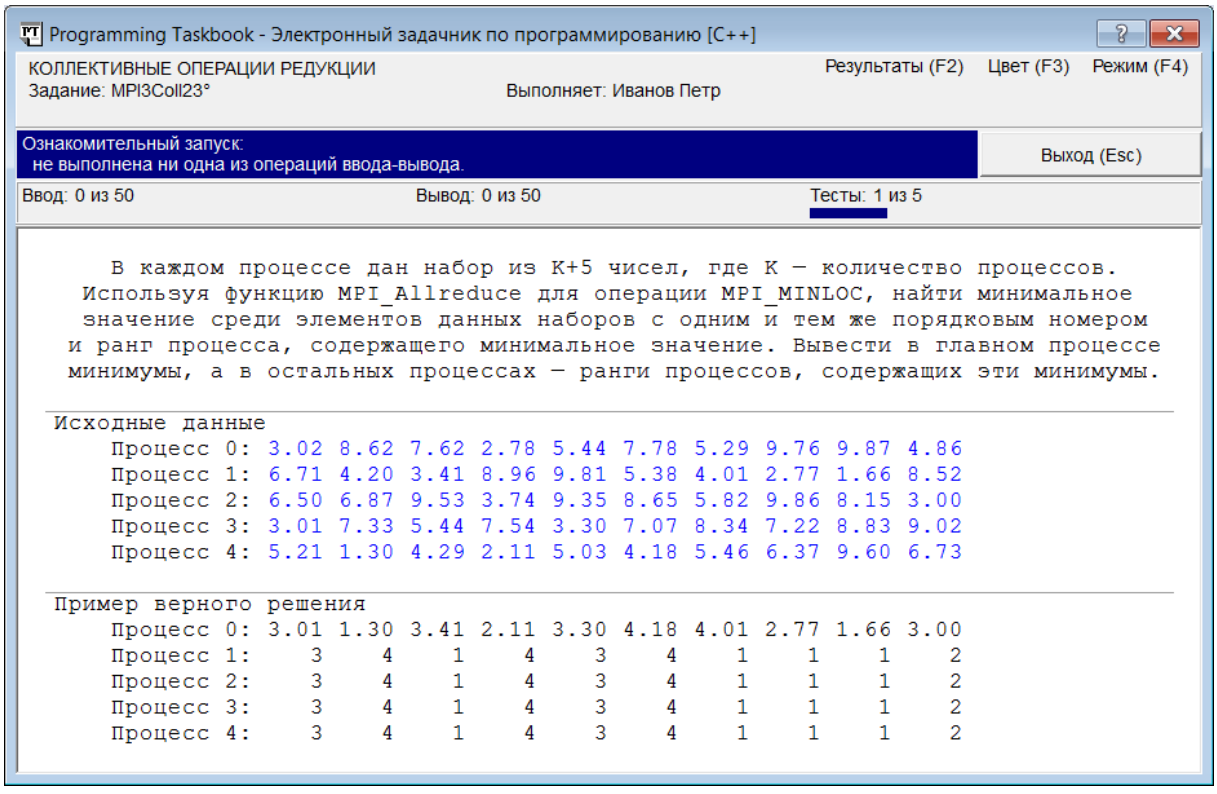


Рис. 15. Ознакомительный запуск задания MPI3Coll23

Коллективная операция редукции может применяться одновременно к нескольким наборам данных. Если каждый процесс предоставляет массив чисел (одного и того же размера), то операция редукции применяется по отдельности к элементам предоставленных массивов с одним и тем же индексом; в результате будет получен массив того же размера, каждый элемент которого будет являться результатом применения операции редукции к элементам исходных массивов с этим же индексом. В зависимости от ис-

пользуемой MPI-функции полученный массив может быть переслан какому-либо конкретному процессу (функция MPI_Reduce) или всем процессам данного коммуникатора (функция MPI_Allreduce).

При использовании операций MPI_MAXLOC и MPI_MINLOC исходные наборы данных должны содержать *пары* чисел: собственно число, которое надо обработать, и его номер. Поэтому в программе необходимо определить вспомогательную структуру для хранения таких пар. В нашем случае должны обрабатываться вещественные числа, поэтому первый элемент пары будет вещественным, а второй — целым:

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

Для хранения исходных данных в каждом процессе должен быть выделен массив элементов типа MINLOC_Data, и такой же массив должен использоваться для хранения результатов выполнения операции редукции. Размер набора данных, который придется хранить в этих массивах, заранее неизвестен, так как он связан с количеством процессов параллельной программы. Поэтому можно либо выделять память для массивов динамически (после того как программе станет известно число процессов size), либо использовать статические массивы, размер которых окажется достаточным для любых наборов исходных данных. При выполнении задания MPI3Coll23 мы будем использовать статические массивы (особенности, связанные с использованием динамических массивов, будут рассмотрены в следующем пункте). Запустив созданную программу-заготовку несколько раз, мы можем убедиться в том, что для данного задания количество процессов может меняться в диапазоне от 3 до 5. Таким образом, учитывая, что размер наборов исходных данных равен $K + 5$, где K — количество процессов, нам достаточно описать в функции Solve массивы размера 10:

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

Инициализация исходного массива d должна выполняться в каждом процессе параллельной программы:

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

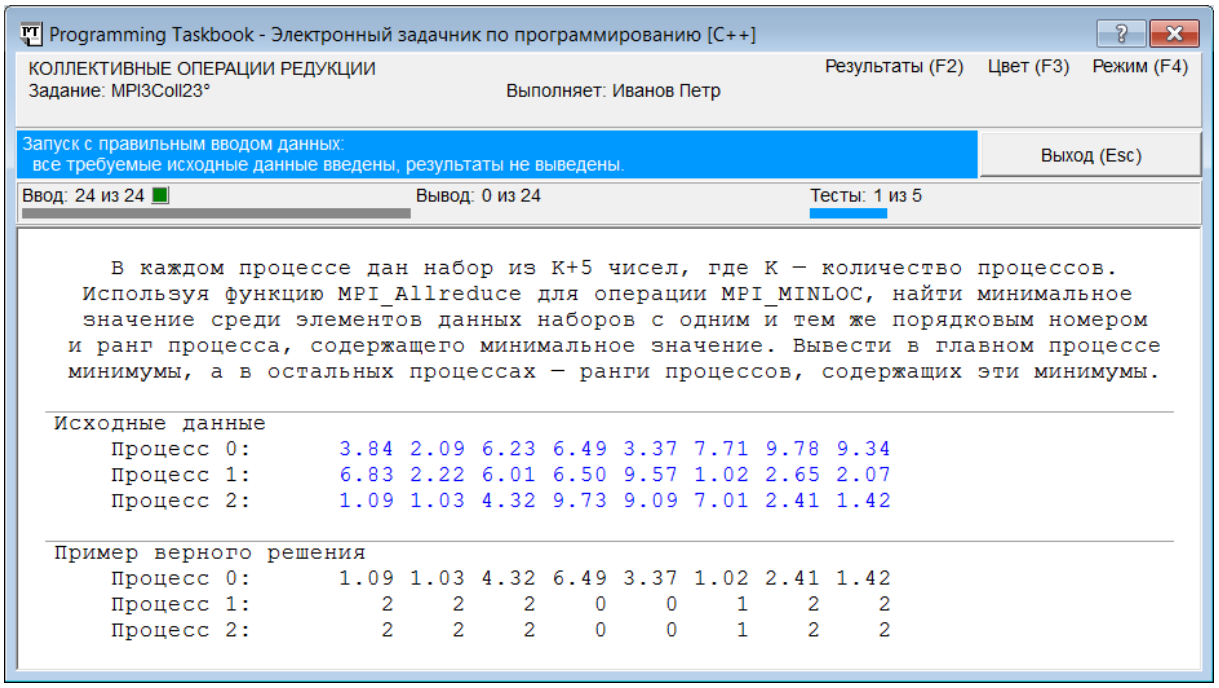


Рис. 16. Окно задачника с информацией об ошибках ввода-вывода

Запустив этот вариант программы, мы получим сообщение о том, что все исходные данные успешно введены (рис.).

Перед выводом результатов необходимо выполнить соответствующую коллективную операцию редукции. Она должна быть выполнена во всех процессах, после чего в главном процессе (ранга 0) надо вывести поле a каждого элемента результирующего массива res (т. е. минимальное значение, выбранное из всех элементов исходных массивов с данным индексом), а в остальных (подчиненных) процессах — поле n (т. е. ранг процесса с этим минимальным значением):

```
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;
```

Обратите внимание на два важных момента. Во-первых, массивы исходных и результирующих данных передаются в MPI-функции как *указатели на их начальный элемент*, поэтому в качестве первых двух параметров функции MPI_Allreduce просто указаны идентификаторы массивов d и res. Во-вторых, имя типа, указываемое в качестве четвертого параметра, должно соответствовать типу элементов обрабатываемых массивов (в данном случае надо указать один из стандартных типов MPI: MPI_DOUBLE_INT, соответствующий структуре из двух полей — вещест-

венного и целого). В ситуациях, когда стандартных типов данных, предоставляемых библиотекой MPI, оказывается недостаточно, приходится определять новые типы MPI (этой теме посвящена группа заданий MPI4Type).

При запуске полученной программы будет выведено сообщение о том, что задание выполнено.

В заключение приведем полный текст решения задачи MPI3Coll23:

```
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;
}
```

2.3. Коллективные операции и создание новых коммуникаторов: MPI5Comm3

Часто для эффективной реализации пересылки данных бывает удобно воспользоваться вспомогательными коммуникаторами, которые включают

не все процессы параллельного приложения, а только требуемую их часть (*группу* процессов). Задания на применение вспомогательных коммуникаторов собраны в трех подгруппах группы MPI5Comm. Следует заметить, что в заданиях группе MPI5Comm рассматриваются только так называемые *интракоммуникаторы*, связанные с одной группой процессов. В MPI можно создавать и другой вид коммуникаторов — *интеркоммуникаторы*, которые связаны не с одной, а с двумя группами процессов. Интеркоммуникаторам посвящена группа задания MPI8Inter (п.), большинство из которых можно выполнять только в системе MPICH2 1.3, поддерживающей стандарт MPI-2.

Рассмотрим одно из заданий, входящих в первую подгруппу группы MPI5Comm: «Группы процессов и коммуникаторы» (задания из двух следующих подгрупп, связанные с виртуальными топологиями, обсуждаются в следующем пункте).

MPI5Comm3. В каждом процессе, ранг которого делится на 3 (включая главный процесс), даны три целых числа. С помощью функции MPI_Comm_split создать новый коммуникатор, включающий процессы, ранг которых делится на 3. Используя одну коллективную операцию пересылки данных для созданного коммуникатора, переслать исходные числа в главный процесс и вывести эти числа в порядке возрастания рангов переславших их процессов (включая числа, полученные из главного процесса).

Указание. При вызове функции MPI_Comm_split в процессах, которые не требуется включать в новый коммуникатор, в качестве параметра color следует указывать константу MPI_UNDEFINED.

Приведем окно задачника, появившееся на экране при ознакомительном запуске программы-заготовки для этого задания (рис.).

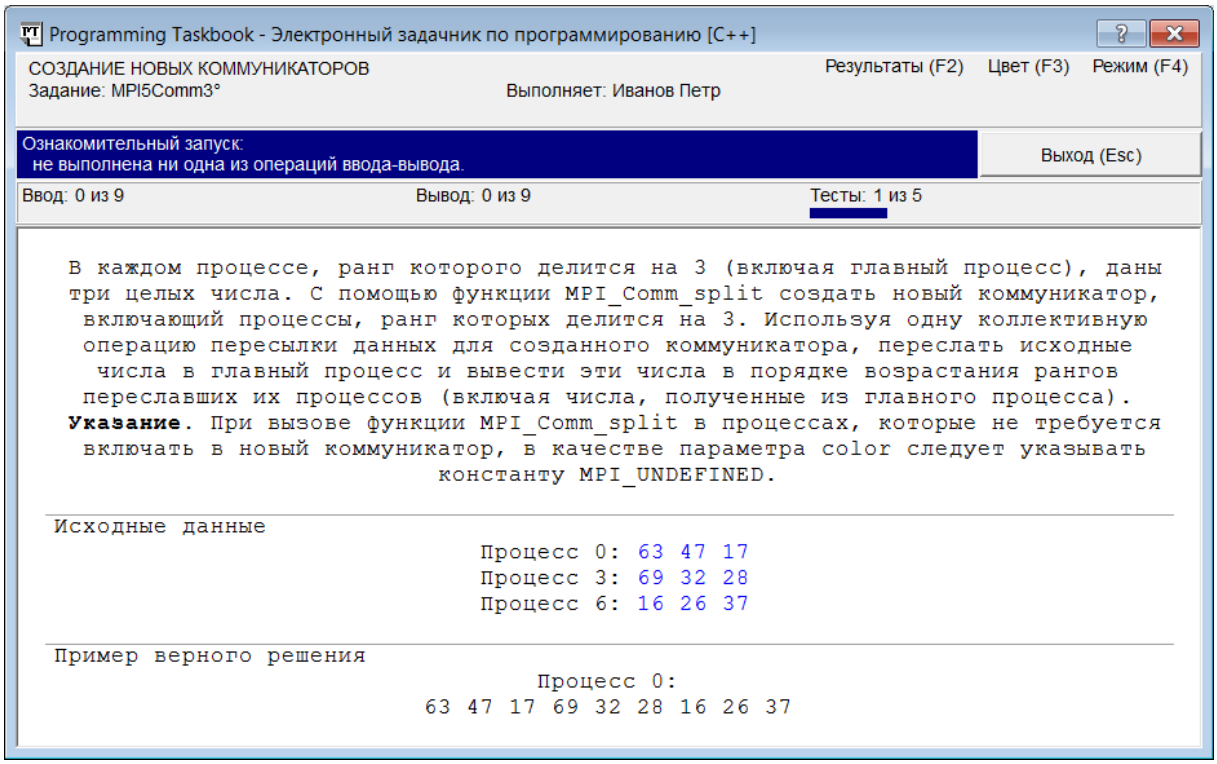


Рис. 17. Ознакомительный запуск задания MPI5Comm3

При этом в консольном окне был выведен текст, показывающий, что в параллельной программе было запущено восемь процессов:

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

Таким образом, в задании надо использовать лишь часть имеющихся процессов. Разумеется, мы можем воспользоваться функциями MPI, обеспечивающими обмен данными между двумя процессами (как в решении задачи MPI2Send11, приведенном в п. 3.1), но более эффективным будет вариант, использующий подходящую коллективную операцию пересылки данных (заметим, что именно такой вариант решения указан в формулировке задания). Однако коллективные операции выполняются для *всех* процессов, входящих в некоторый коммуникатор, поэтому в программе необходимо предварительно создать коммуникатор, включающий только процессы, ранг которых делится на 3. Сделать это можно несколькими способами; один из них связан с использованием функции MPI_Comm_split, упомянутой в формулировке задания.

Функция MPI_Comm_split позволяет «расщепить» исходный коммуникатор на набор коммуникаторов, каждый из которых связан с некоторой частью процессов, входящих в исходный коммуникатор. Следует учитывать, что массив новых коммуникаторов в программе не возникает; вместо этого каждому из процессов программы функция MPI_Comm_split предоставляет именно тот коммуникатор из созданного набора, в который входит данный процесс. Предусмотрена также ситуация, когда некоторые процес-

сы не будут включены ни в один из созданных коммуникаторов; для таких процессов функция MPI_Comm_split возвращает «пустой» коммуникатор MPI_COMM_NULL.

Для разбиения процессов на новые группы в функции MPI_Comm_split используется параметр color («цвет»). Все процессы одного цвета включаются в один и тот же новый коммуникатор; при этом любой цвет представляет собой обычное целое число. Предусмотрен также «неопределенный цвет» MPI_UNDEFINED; его надо указывать для процесса, который не следует включать ни в один из новых коммуникаторов. Второй характеристикой, используемой в функции MPI_Comm_split при создании нового набора коммуникаторов, является параметр key («ключ»). Он определяет порядок, в котором будут располагаться процессы в каждом из новых коммуникаторов: процессы в каждом коммуникаторе упорядочиваются по возрастанию их ключей (если некоторые процессы имеют одинаковые ключи, то их порядок определяется средой MPI, которая управляет параллельной программой). Для сохранения в каждом из вновь созданных коммуникаторов исходного порядка следования процессов достаточно в качестве параметра key для каждого процесса указать ранг этого процесса в исходном коммуникаторе.

Предусмотренная для функции MPI_Comm_split возможность использования константы MPI_UNDEFINED позволяет создавать новые коммуникаторы только для некоторых из имеющихся процессов. Ввиду важности этой возможности о ней сказано в указании к данному заданию.

Учитывая перечисленные особенности функции MPI_Comm_split, создадим с ее помощью коммуникатор, в который будут входить только процессы ранга, кратного трем:

```
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;
```

Можно запустить данный вариант программы, чтобы убедиться, что при создании нового коммуникатора мы не допустили ошибок (задачник по-прежнему будет считать запуск программы ознакомительным, так как ввод и вывод данных в ней не выполняется).

Последний условный оператор обеспечивает немедленный выход из процесса, если с ним оказался связан «нулевой» коммуникатор comm (разумеется, в нашем случае в условии выхода можно было бы анализировать остаток от деления rank на 3, однако использованный вариант является более универсальным).

Во всех остальных процессах осталось ввести три целых числа, переслать все введенные числа в главный процесс, используя коллективную

функцию MPI_Gather, и вывести полученные числа. Для ввода исходных чисел в каждом процессе достаточно описать массив data из трех элементов. Размер результирующего массива res, который будет получен в главном процессе, зависит от количества процессов параллельного приложения. При обсуждении задания MPI3Call23 мы уже отмечали, что в такой ситуации можно использовать либо статический массив достаточно большого размера, либо динамический массив, размер которого будет определен после того, как станет известно количество процессов. В п. 3.2, выполняя задание MPI3Call23, мы использовали статический массив. Теперь в качестве результирующего массива res будем использовать динамический массив:

```
int data[3];
for (int i = 0; i < 3 ; ++i)
  pt >> data[i];
MPI_Comm_size(comm, &size);
int *res = new int[3 * size];
MPI_Gather(data, 3, MPI_INT, res, 3, MPI_INT, 0, comm);
if (rank == 0)
  for (int i = 0; i < 3 * size; ++i)
    pt << res[i];
delete[] res;
```

Для нахождения общего числа полученных элементов мы предварительно определили количество процессов в коммуникаторе comm (с помощью функции MPI_Comm_size), записав его в переменную size. Поскольку каждый процесс коммуникатора comm пересылает в главный процесс три элемента, размер динамического массива res полагается равным size * 3. Для выделения памяти мы использовали оператор new языка C++.

Затем вызывается функция MPI_Gather (см. п. 7.6). Обратите внимание на то, что в функции MPI_Gather в качестве пятого параметра указывается не размер массива res, а количество элементов, принимаемых от *каждого* процесса. Заметим также, что функция MPI_Gather получает данные от всех процессов коммуникатора comm, в том числе и от того процесса, который служит получателем всех данных. При указании процесса-получателя мы учли, что процесс ранга 0 в коммуникаторе MPI_COMM_WORLD является одновременно и процессом ранга 0 в коммуникаторе comm.

В конце программы мы с помощью оператора delete освобождаем динамическую память, ранее выделенную оператором new.

Запустив полученную программу, мы получим сообщение о том, что задание выполнено.

Приведем полный текст полученного решения:

```
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;
  int data[3];
  for (int i = 0; i < 3 ; ++i)
    pt >> data[i];
  MPI_Comm_size(comm, &size);
  int* res = new int[3 * size];
  MPI_Gather(data, 3, MPI_INT, res, 3, MPI_INT, 0, comm);
  if (rank == 0)
    for (int i = 0; i < 3 * size; ++i)
      pt << res[i];
  delete[] res;
}
```

2.4. *Использование виртуальных топологий: MPI5Comm17, MPI5Comm29*

При выполнении параллельной программы каждый процесс может обмениваться данными с любым другим процессом посредством стандартного коммуникатора MPI_COMM_WORLD. Если требуется выделить какую-либо *часть* имеющихся процессов для организации взаимодействия между ними (например, для коллективного обмена данными в пределах только этой части процессов), то необходимо определить для нужных процессов новый коммуникатор. В предыдущем пункте для создания новых коммуникаторов мы использовали функцию MPI_Comm_split. В библиотеке MPI предусмотрен еще один способ, позволяющий объединить нужные процессы; он состоит в определении для процессов параллельного приложения *виртуальной топологии* (virtual topology).

Виртуальная топология задает на множестве процессов некоторую структуру, определенным образом упорядочивающую эти процессы. Име-

ются два вида виртуальной топологии: декартова топология и топология графа. В случае *декартовой топологии* (Cartesian topology) все процессы интерпретируются как узлы некоторой n-мерной решетки размера $k_1 \times k_2 \times \ldots \times k_n$ (если $n = 2$, то процессы можно рассматривать как элементы прямоугольной матрицы размера $k_1 \times k_2$). В случае *топологии графа* (graph topology) процессы интерпретируются как вершины некоторого графа; при этом связи между процессами определяются посредством задания набора ребер (дуг) для этого графа.

Если коммуникатор снабжен виртуальной топологией, то из входящих в него процессов можно выделять различные подмножества регулярной структуры. В частности, из n-мерных решеток, задаваемых с помощью декартовой топологии, можно выделять различные *сечения* размерности m, где m может меняться от 1 до $n - 1$ (например, из двумерных матриц можно выделять одномерные строки или столбцы).

Воспользуемся механизмом виртуальных топологий для выполнения следующего задания.

MPI5Comm17. Число процессов K кратно трем: $K = 3N$, $N > 1$. В процессах 0, N и $2N$ дано по N целых чисел. Определить для всех процессов декартову топологию в виде матрицы размера $3 \times N$, после чего, используя функцию MPI_Cart_sub, расщепить матрицу процессов на три одномерные строки (при этом процессы 0, N и $2N$ будут главными процессами в полученных строках). Используя одну коллективную операцию пересылки данных, переслать по одному исходному числу из главного процесса каждой строки во все процессы этой же строки и вывести полученное число в каждом процессе (включая процессы 0, N и $2N$).

При ознакомительном запуске этого задания окно задачника примет вид, примерно соответствующий приведенному на рисунке 17.

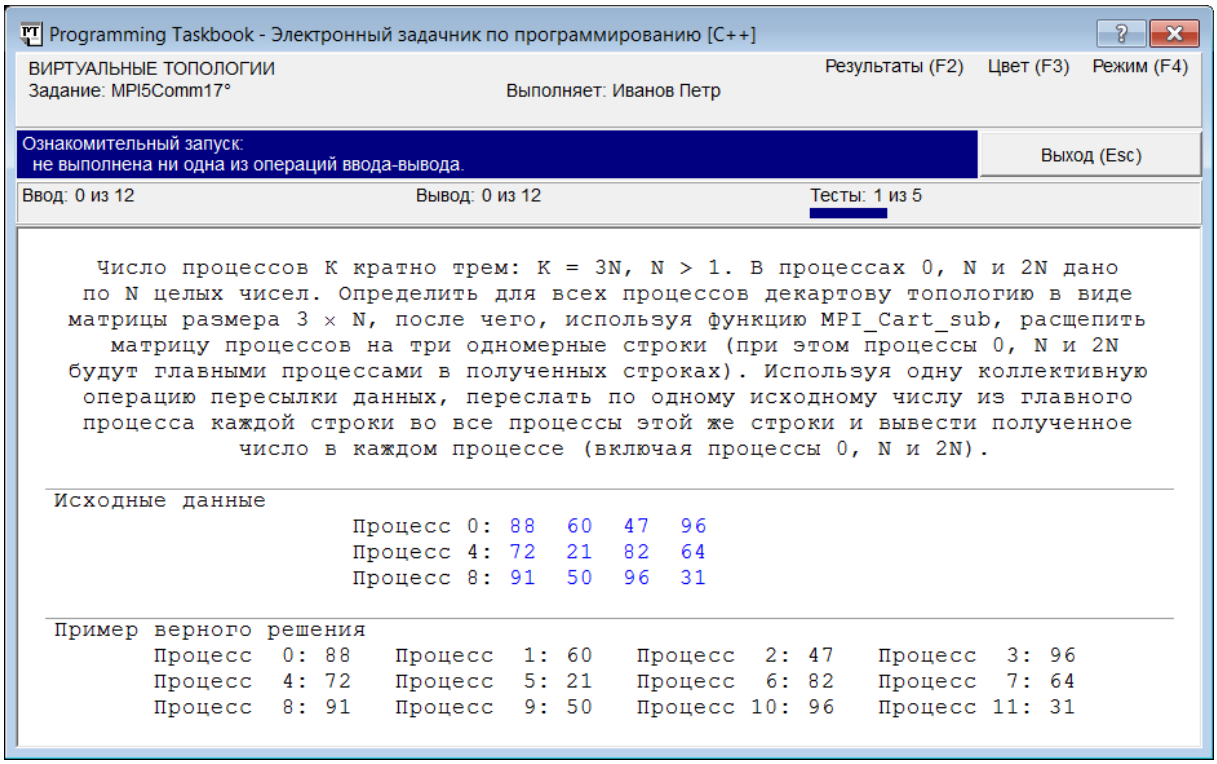


Рис. 18. Ознакомительный запуск задания MPI5Comm17

Вариант, приведенный в окне, соответствует случаю $N = 4$: имеется 12 процессов, которые следует интерпретировать как элементы матрицы размера 3×4. При этом в процессах, являющихся начальными элементами строк (иначе говоря, в процессах, входящих в первый столбец матрицы), дано по четыре числа, каждое из которых надо переслать в соответствующий процесс *этой же строки* матрицы процессов.

На первом этапе решения задачи необходимо определить требуемую декартову топологию. Для этого надо использовать функцию MPI_Cart_create, которая имеет следующие параметры:

- исходный коммуникатор, для процессов которого определяется декартова топология (в нашем случае MPI_COMM_WORLD);
- число размерностей создаваемой декартовой решетки (в нашем случае 2);
- целочисленный массив, каждый элемент которого определяет размер по каждому измерению (в нашем случае массив должен состоять из двух элементов со значениями 3 и size/3);
- целочисленный массив флагов, определяющих периодичность каждого измерения (в нашем случае достаточно использовать массив из двух нулевых элементов);
- целочисленный флаг, определяющий, можно ли среде MPI автоматически менять порядок нумерации процессов (в нашем случае необходимо положить этот параметр равным 0);

- результирующий коммуникатор с декартовой топологией (единственный выходной параметр).

Периодичность для некоторых измерений декартовой решетки удобно использовать, например, при выполнении циклического переноса данных между процессами, входящими в эти измерения (см. описание функции MPI_Cart_shift); в этом случае соответствующий элемент в массиве флагов надо задать отличным от 0.

Автоматическая перенумерация процессов при определении декартовой топологии позволяет учесть физическую конфигурацию компьютерной системы, на которой выполняется параллельная программа, и тем самым повысить эффективность ее выполнения. Однако в учебных программах, выполняемых под управлением задачника PT for MPI-2, порядок процессов в созданных декартовых топологиях должен оставаться неизменным, поэтому перенумерацию процессов следует запретить.

Приведем фрагмент программы, определяющий декартову топологию и связывающий ее с новым коммуникатором comm (этот фрагмент надо поместить в конце функции Solve):

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

Коммуникатор comm, созданный в результате выполнения функции MPI_Cart_create, будет содержать те же процессы, что и исходный коммуникатор MPI_COMM_WORLD, причем в том же порядке. Однако эти коммуникаторы являются *различными*. Это проявляется прежде всего в том, что операции пересылки данных, осуществляемые с помощью коммуникаторов MPI_COMM_WORLD и comm, выполняются независимо и не влияют друг на друга (различные коммуникаторы, содержащие один и тот же одинаково упорядоченный набор процессов, называются *конгруэнтными*; при их сравнении функцией MPI_Comm_compare возвращается результат, равный значению константы MPI_CONGRUENT). Кроме того, с коммуникатором comm дополнительно связана виртуальная топология, а коммуникатор MPI_COMM_WORLD никакой виртуальной топологией не обладает (проверить, связана ли с коммуникатором виртуальная топология определенного типа, можно с помощью функции MPI_Topo_test).

Благодаря наличию декартовой топологии, с каждым процессом коммуникатора comm связывается не только порядковый номер (ранг процесса), но и набор целых чисел, определяющих *координаты* этого процесса в соответствующей декартовой решетке. Координаты нумеруются от 0.

Координаты процесса в декартовой топологии можно определить по его рангу с помощью функции MPI_Cart_coords (заметим, что функция MPI_Cart_rank позволяет решить обратную задачу). Для выполнения наше-

го задания использовать функцию MPI_Cart_coords не требуется, однако в ряде случаев (в частности, при отладке параллельных программ) она может оказаться полезной. Поэтому приведем пример ее использования, выведя в раздел отладки окна задачника координаты всех процессов, включенных в декартову решетку. Для этого дополним текст программы следующими операторами:

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

При запуске дополненной программы окно задачника примет вид, соответствующий приведенному на рис. 18.

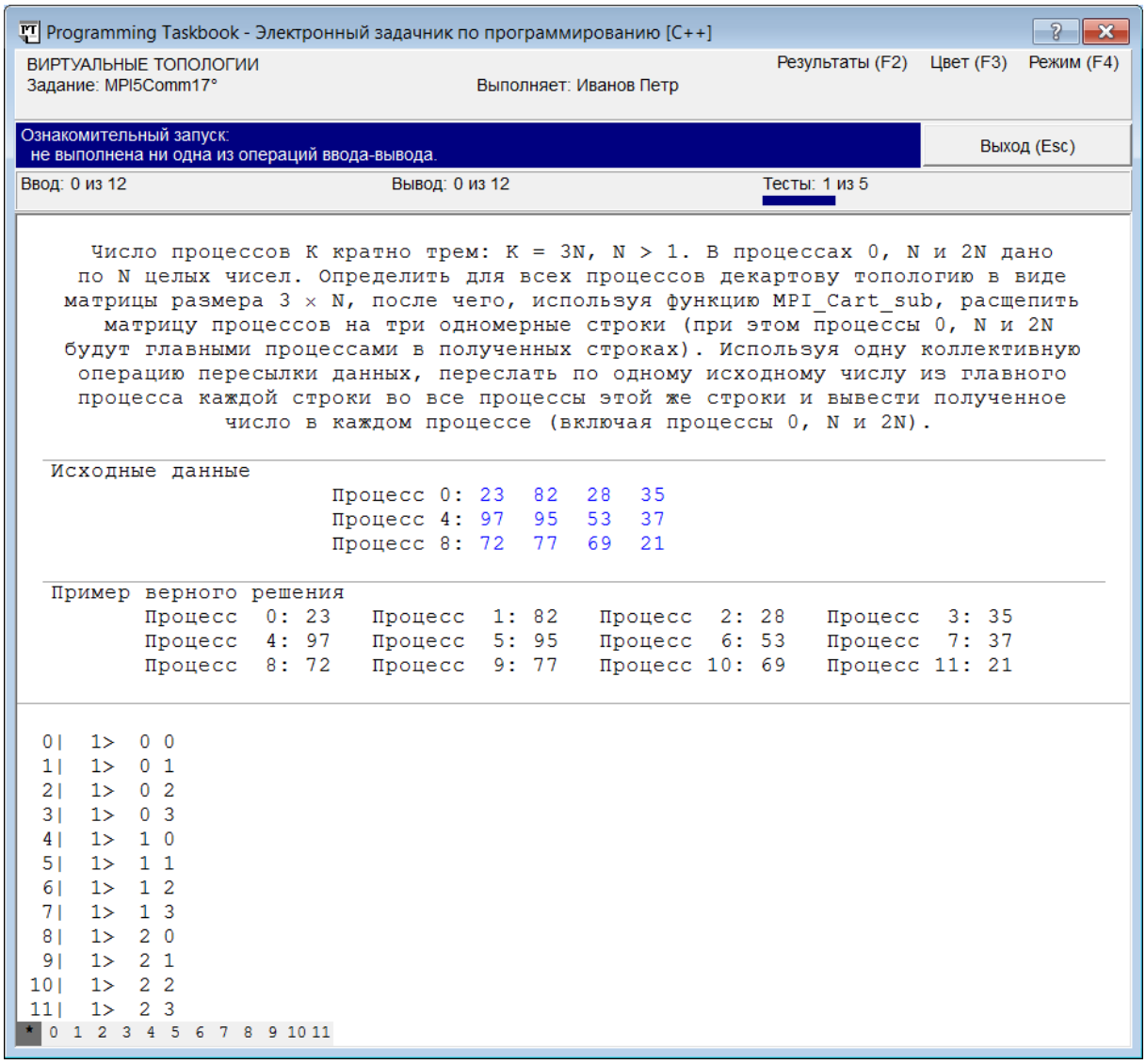


Рис. 19. Вывод декартовых координат процессов в разделе отладки

Напомним, что первое число в каждой строке раздела отладки (перед символом «|») обозначает ранг процесса, в котором были выведены дан-

ные, указанные в этой строке. Второе число (после которого указывается символ «>») обозначает порядковый номер строки вывода для данного процесса. В нашем случае каждый процесс вывел по одной строке, содержащей два числа: свои координаты в декартовой топологии.

Мы видим, что процесс ранга 0 имеет координаты (0, 0), т. е. является первым элементом первой строки матрицы, а процесс ранга 11 имеет координаты (2, 3), т. е. является последним (четвертым по счету) элементом последней (третьей по счету) строки. Кроме того, в данном случае в первую строку матрицы входят процессы рангов 0, 1, 2, 3, а в первый столбец — процессы рангов 0, 4 и 8.

Вернемся к нашей задаче. Для ее решения необходимо вначале разбить полученную матрицу процессов на отдельные строки, связав с каждой строкой новый коммуникатор. После этого следует выполнить для всех процессов, входящих в одну строку, коллективную операцию MPI_Scatter, рассылающую фрагменты набора данных из одного процесса во все процессы, входящие в коммуникатор.

Расщепление декартовой решетки на набор подрешеток меньшей размерности (в частности, разбиение матрицы на набор строк или столбцов) и связывание с каждой полученной подрешеткой нового коммуникатора выполняется с помощью функции MPI_Cart_sub. В качестве ее первого параметра следует указать исходный коммуникатор с декартовой топологией, в качестве второго — массив флагов, определяющий номера тех измерений, которые должны остаться в подрешетках: если соответствующее измерение должно входить в каждую подрешетку, то на его месте в массиве указывается ненулевой флаг, если же по данному измерению выполняется расщепление исходной решетки, то значение связанного с этим измерением флага должно быть нулевым.

В результате выполнения функции MPI_Cart_sub создается набор новых коммуникаторов, каждый из которых связывается с одной из полученных подрешеток (все созданные коммуникаторы автоматически снабжаются декартовой топологией). Однако возвращается данной функцией (в качестве третьего, выходного параметра) только *один* из созданных коммуникаторов — тот, в который входит процесс, вызывавший эту функцию. Заметим, что аналогичным образом ведет себя и функция MPI_Comm_split, рассмотренная в предыдущем пункте.

Для разбиения исходной матрицы процессов на набор *строк* надо в качестве второго параметра функции MPI_Cart_sub указать массив из двух целочисленных элементов, первый из которых равен 0, а второй является ненулевым (например, равен 1). В этом случае все элементы матрицы с одинаковым значением *первой* (удаляемой) координаты будут объединены в новом коммуникаторе (назовем его comm_sub).

Первый процесс каждой строки (тот, который должен по условию задачи переслать свои данные во все остальные процессы этой же строки) будет иметь в полученном коммуникаторе comm_sub ранг, равный 0. Для определения ранга следует воспользоваться функцией MPI_Comm_rank. После этого, если ранг равен 0, надо прочесть исходные данные и переслать по одному элементу данных каждому процессу этого же коммуникатора, используя функцию MPI_Scatter. В конце останется вывести элемент, полученный каждым процессом.

Приведем завершающую часть решения:

```
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;
```

Запустив полученную программу, мы получим сообщение о том, что задание выполнено. Удалять фрагмент, обеспечивающий отладочную печать координат процессов, не требуется, так как вывод отладочных данных никак не влияет на проверку правильности решения.

Приведем полный текст полученного решения (без отладочной печати координат):

```
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;
}
```

Примечание. Распространенной ошибкой, связанной с использованием функции MPI_Cart_sub, является неправильное указание ее второго параметра — массива флагов. Если, например, в приведенной выше программе поменять местами элементы со значениями 0 и 1 в массиве remain_dims, то при запуске программы в окне задачника будут выведены сообщения об ошибках, подобные приведенным на рис. 19.

Проанализируем эти сообщения. Из-за неверного указания массива флагов функция MPI_Cart_sub выполнила разбиение исходной матрицы не на строки, а на *столбцы*; в результате были созданы 4 новых коммуникатора, каждый из которых содержит 3 процесса, входящих в один и тот же столбец матрицы. При этом процесс, являющийся первым в столбце, считается процессом ранга 0 для соответствующего коммуникатора. Поэтому условие в последнем операторе if будет истинным для процессов 0, 1, 2 и 3, и именно для них будут выполняться операторы ввода исходных данных. Однако в процессах 1, 2 и 3 исходных данных не предусмотрено, поэтому при выполнении программы для этих процессов выводится сообщение об ошибке «*Попытка ввести лишние исходные данные*». С другой стороны, процессы 4 и 8 (являющиеся начальными процессами во второй и третьей строке матрицы) в новых коммуникаторах имеют ненулевой ранг, и поэтому для них ввод данных не выполняется, что и отмечено в сообщении об ошибке для этих процессов: «*Введены не все требуемые исходные данные. Количество прочитанных данных: 0 (из 4)*». Обратите также внимание на то, что процесс 0 переслал свои исходные данные не в процессы, входящие в первую строку матрицы (как требовалось по условию задачи), а в процессы, входящие в первый столбец. Так как в процессах 1, 2 и 3 исходные данные отсутствовали, в остальные процессы соответствующих столбцов были пересланы нули. Заметим, что в самих процессах 1, 2 и 3 полученные нули не были выведены,

так как ранее в каждом из этих процессов задачник выявил ошибку ввода и поэтому заблокировал все последующие операции ввода-вывода для этих процессов. Таким образом, приведенной в окне задачника информации вполне достаточно, чтобы распознать причину ошибки и внести в программу необходимые исправления.

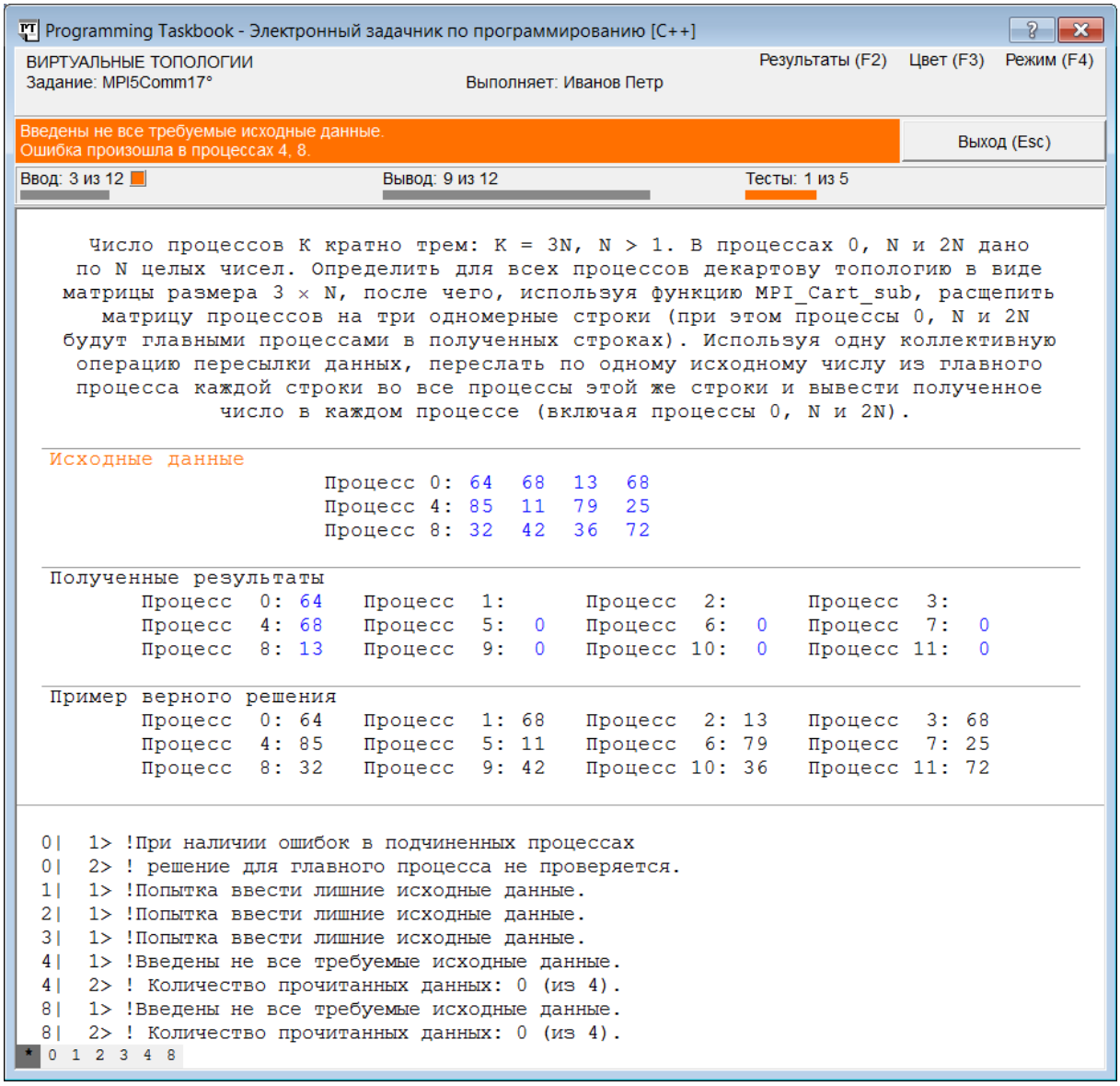


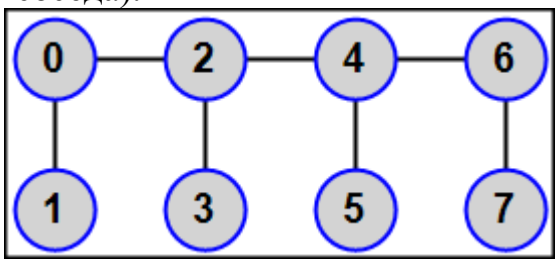
Рис. 20. Окно задачника при ошибочном выполнении задания MPI5Comm17

Теперь рассмотрим другой вид виртуальных топологий — топологию графа.

Следует отметить, что для работы с топологиями графа библиотека MPI предоставляет существенно меньше средств, чем для работы с декартовыми топологиями. Напомним, что для процессов, входящих в декартову топологию, можно определить декартовы координаты по их рангам (и ран-

ги по декартовым координатам); кроме того, предусмотрена возможность выделения подрешеток меньшей размерности (причем каждая подрешетка автоматически связывается с новым коммуникатором). Также имеется специальная функция MPI_Cart_shift, позволяющая определить «соседей» процесса вдоль одной из декартовых координат (использование этой функции упрощает пересылку сообщений вдоль указанной координаты). Что же касается топологии графа, то для нее предусмотрена лишь возможность, аналогичная возможности функции MPI_Cart_Shift, а именно определение количества и рангов всех процессов-*соседей* (neighbors) некоторого процесса в графе, определяемом данной топологией (соседями считаются процессы, связанные ребрами с данным процессом). Познакомимся с этой возможностью на практике, выполнив следующее задание.

MPI5Comm29. Число процессов K является четным: $K = 2N$ ($1 < N < 6$); в каждом процессе дано целое число A. Используя функцию MPI_Graph_create, определить для всех процессов топологию графа, в которой все процессы четного ранга (включая главный процесс) связаны в цепочку: 0 — 2 — 4 — 6 — … — ($2N - 2$), а каждый процесс нечетного ранга R (1, 3, …, $2N - 1$) связан с процессом ранга $R - 1$ (в результате каждый процесс нечетного ранга будет иметь единственного соседа, первый и последний процессы четного ранга будут иметь по два соседа, а остальные — «внутренние» — процессы четного ранга будут иметь по три соседа).



Переслать число A из каждого процесса всем процессам-соседям. Для определения количества процессов-соседей и их рангов использовать функции MPI_Graph_neighbors_count и MPI_Graph_neighbors, пересылку выполнять с помощью функции MPI_Sendrecv. Во всех процессах вывести полученные числа в порядке возрастания рангов переславших их процессов.

При ознакомительном запуске этого задания окно задачника примет вид, примерно соответствующий приведенному на рис. 20. Заметим, что одновременно с окном задачника в правом верхнем углу экрана отобразится рисунок из формулировки задания, иллюстрирующий используемую топологию.

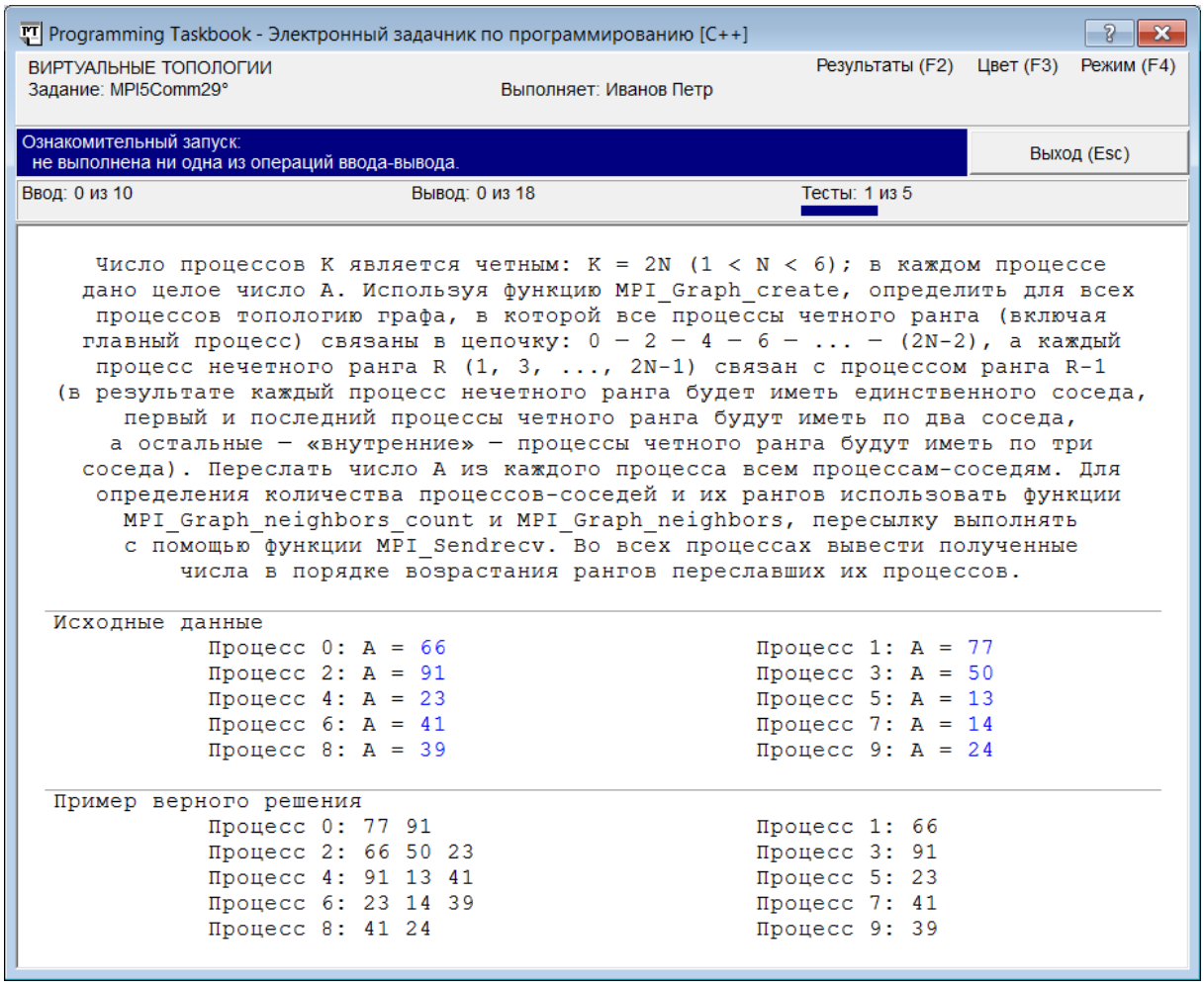


Рис. 21. Ознакомительный запуск задания MPI5Comm29

Для большей наглядности изобразим процессы вместе с их исходными данными в виде графа той структуры, которая описана в формулировке задания (рис. 21).

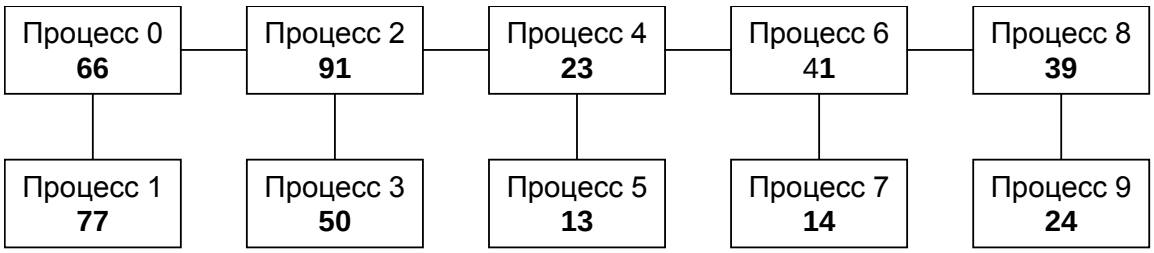


Рис. 22. Топология графа, описанная в задании MPI5Comm29

Поскольку процесс ранга 0 имеет двух соседей (процессы ранга 1 и 2), он должен отправить им число 66 и получить от них числа 77 и 91. Процесс ранга 1 имеет только одного соседа (процесс ранга 0), поэтому он должен отправить ему число 77 и получить от него число 66. Процесс ранга 2, имеющий три соседа, должен отправить им число 91 и получить от них числа 66, 50 и 23, и т. д.

Если бы каждый процесс имел информацию о количестве своих соседей, а также об их рангах (в порядке возрастания), то это позволило бы единообразно запрограммировать действия по пересылке данных для любого процесса, независимо от того, сколько соседей он имеет. Требуемая информация о соседях может быть легко получена, если на множестве всех процессов будет определена соответствующая топология графа, а для этого необходимо воспользоваться функцией MPI_Graph_create. Данная функция имеет следующие параметры:

- исходный коммуникатор, для процессов которого определяется топология графа;
- число вершин графа;
- целочисленный *массив степеней вершин*, i-й элемент которого равен суммарному количеству соседей для первых i вершин графа;
- целочисленный *массив ребер*, содержащий упорядоченный список ребер для всех вершин (вершины нумеруются от 0);
- целочисленный флаг, определяющий, можно ли среде MPI автоматически менять порядок нумерации процессов;
- результирующий коммуникатор с топологией графа (единственный выходной параметр).

Как и при выполнении предыдущего задания, следует запретить перенумерацию процессов, положив соответствующей флаг равным 0.

Чтобы лучше понять смысл параметров-массивов, задающих характеристики определяемого графа, перечислим их элементы для графа, приведенного на рисунке 21. Первая вершина графа (процесс ранга 0) имеет двух соседей, поэтому первый элемент массива степеней вершин будет равен 2. Вторая вершина графа (процесс ранга 1) имеет одного соседа, поэтому второй элемент массива степеней вершин будет равен 3 (к значению предыдущего элемента прибавляется 1). Третья вершина (процесс ранга 2) имеет трех соседей, поэтому третий элемент массива степеней вершин будет равен 6, и т. д. Получаем следующий набор значений: 2, 3, 6, 7, 10, 11, 14, 15, 17, 18 (предпоследний элемент массива равен 17, так как процесс ранга 8, как и процесс ранга 0, имеет двух соседей). Заметим, что значение последнего элемента массива степеней вершин всегда будет в два раза больше общего количества ребер графа.

В массиве ребер необходимо указать ранги всех соседей для каждой вершины (для большей наглядности будем выделять группы соседей каждой вершины дополнительными пробелами, а сверху в скобках указывать ранг вершины, соседи которой перечислены ниже):

```
(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
```

Размер полученного массива ребер равен значению последнего элемента массива степеней вершин.

Если число процессов равно size, то массив степеней вершин должен содержать size элементов. Размер массива ребер зависит от структуры графа; в нашем случае размер массива ребер равен 2·(size – 1), где size — число процессов.

При заполнении массивов удобно отдельно обработать фрагменты, соответствующие двум первым (ранга 0 и 1) и двум последним (ранга size – 2 и size – 1) вершинам графа, а остальные вершины перебирать в цикле, обрабатывая по две вершины (ранга 2 и 3, 4 и 5, …, size – 4 и size – 3) на каждой итерации.

Следуя описанию функции MPI_Graph_create в стандарте MPI [7], будем использовать для массива степеней вершин и массива ребер названия index и edges соответственно. При заполнении этих массивов удобно использовать вспомогательную переменную n, равную половине количества процессов. Приведем фрагмент программы, заполняющей массивы index и 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;
```

Для проверки правильности данной части алгоритма выведем в раздел отладки окна задачника значения элементов полученных массивов (по-

скольку эти массивы формируются одинаково во всех процессах, достаточно вывести их значения только в главном процессе):

```
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]);
}
```

Если при очередном запуске программы число процессов будет равно 10, то в разделе отладки мы увидим наборы значений, совпадающие с теми, которые были нами получены ранее (см. рис. 22). Чтобы уменьшить размеры окна, на приведенном рисунке был скрыт раздел с формулировкой задания. Для этого достаточно нажать клавишу Delete или щелкнуть на маркере, расположенном в правом верхнем углу этого раздела. Повторное нажатие клавиши Delete или щелчок на этом маркере восстанавливает раздел с формулировкой.

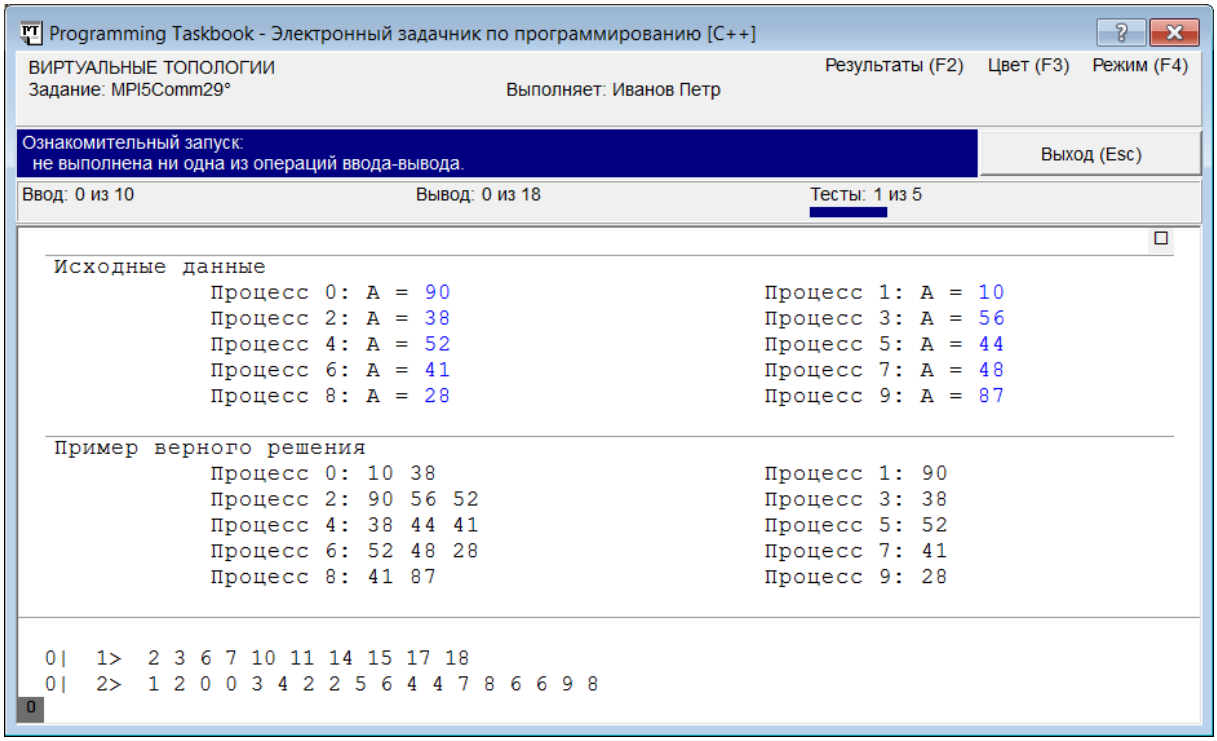


Рис. 23. Отладочный вывод элементов массива степеней вершин и массива ребер

Убедившись, что массивы сформированы правильно, создадим топологию графа, вызвав функцию MPI_Graph_create в каждом процессе параллельного приложения:

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

Осталось реализовать завершающую часть алгоритма, непосредственно связанную с пересылкой данных. В этой части для текущего процесса (процесса ранга rank) следует определить количество count его соседей и массив neighbors их рангов в текущей топологии графа (используя функции MPI_Graph_neighbors_count и MPI_Graph_neighbors), после чего организовать обмен данными между текущим процессом и каждым из его соседей (согласно формулировке задания для этого надо использовать функцию MPI_Sendrecv, которая обеспечивает как прием сообщения от некоторого процесса, так и отправку ему другого сообщения).

Приведем завершающую часть решения:

```
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;
```

Запустив программу, мы получим сообщение о том, что задание выполнено.

В заключение данного пункта отметим, что в стандарте MPI-2 появился новый вид топологии графов: *топология распределенного графа* (distributed graph topology). Ей посвящены два задания, входящие в третью подгруппу группы MPI5Comm (см. п.).

2.5. Параллельный ввод-вывод (MPI-2): MPI6File26

Одним из важных нововведений стандарта MPI 2.0 является поддержка *параллельного файлового ввода-вывода*.

В параллельных программах исходные данные, как правило, считываются из файлов, а результаты, соответственно, в файлы записываются. Из-за отсутствия средств параллельного файлового доступа в MPI-1 приходилось организовывать чтение данных в каком-либо выделенном (обычно главном) процессе, после чего выполнять их пересылку во все остальные процессы параллельного приложения. Аналогично, для сохранения

полученных результатов требовалось предварительно пересылать эти результаты в некоторый процесс и уже в этом процессе организовывать их запись в файл. Если же в параллельном приложении требовалось обеспечить доступ к одному и тому же файлу для нескольких процессов, то для корректной организации подобного доступа было необходимо выполнять особые синхронизирующие действия.

С появлением стандарта MPI-2 появилась возможность считывать или записывать файловые данные в каждом процессе параллельного приложения, не предпринимая специальных усилий по синхронизации доступа к файлу.

Механизмы файлового чтения-записи, определенные в стандарте MPI-2, являются чрезвычайно гибкими. Они включают как *локальные*, так и *коллективные* функции доступа к файлу, причем для каждого вида доступа предусмотрены три варианта *позиционирования*: либо явное, с помощью указания файловой позиции в специальном параметре функции, либо неявное, использующее текущее значение индивидуального (своего для каждого процесса) или общего (разделяемого всеми процессами) *файлового указателя* (при этом оба вида файловых указателей могут применяться как для локальных, так и для коллективных функций доступа). Кроме того, все описанные выше виды параллельного файлового доступа реализованы в двух вариантах: *блокирующем* и *неблокирующем* (различия между которыми подобны различиям между блокирующими и неблокирующими операциями пересылки данных).

Таким образом, комбинируя описанные выше механизмы, можно указать 24 (= 2·2·3·2) различных варианта параллельного файлового доступа:

- чтение *или* запись (2 варианта); в имени функций чтения используется слово read, в имени функций записи — write;
- локальный *или* коллективный доступ (2 варианта); в имени функций коллективного доступа добавляется слово all (за исключением функций, использующих общие файловые указатели, для которых в случае локального доступа используется слово shared, а в случае коллективного доступа — ordered);
- явное позиционирование, *или* индивидуальный файловый указатель, *или* общий файловый указатель (3 варианта); в имени функций для явного позиционирования добавляется слово at, в имени функций, использующих общий указатель, добавляется слово shared или ordered;
- блокирующий *или* неблокирующий доступ (2 варианта); для локальных функций в случае неблокирующего доступа к словам read и write добавляется префикс i (iread и iwrite), коллективные неблокирующие функции являются парными, имя первой из них завершается словом begin, а имя второй — словом end.

С каждым вариантом файлового доступа связана своя функция MPI (или пара функций для вариантов коллективного неблокирующего доступа), поэтому общее количество функций равно 30.

По имени функции легко определить связанный с ней вариант файлового доступа. Например, функция MPI_File_read_at_all обеспечивает *блокирующий* вариант *чтения* (read), основанный на *явном позиционировании* (at), и является *коллективной* (all), а функция MPI_iwrite_shared обеспечивает *неблокирующий* вариант *записи* (iwrite), использует *общий файловый указатель* и является *локальной* (обе последние характеристики определяются словом shared). Простейший вариант функции файлового чтения (записи) имеет имя MPI_File_read (соответственно MPI_File_write), эти функции являются *блокирующими*, *локальными* и используют *индивидуальные файловые указатели*.

Помимо различных вариантов организации чтения-записи, в MPI-2 предусмотрен гибкий способ настройки *вида файловых данных* (file view; в русскоязычной литературе также используется понятие «образ файловых данных») — от простейшего, при котором файл интерпретируется как набор последовательно расположенных байтов, до весьма сложных, в которых файловый вид может включать последовательности элементов, причем не обязательно непрерывные (и в начале, и в конце, и между некоторыми элементами допускаются пустые промежутки). Кроме того, для каждого процесса можно определить свой вид файловых данных.

При выполнении заданий группы MPI6File можно познакомиться с большинством возможностей параллельного файлового ввода-вывода. В первой подгруппе этой группы (задания MPI6File1–MPI6File8) изучаются *локальные* файловые операции, во второй (MPI6File9–MPI6File16) — *коллективные* файловые операции, а завершающая подгруппа (MPI6File17–MPI6File30) посвящена различным способам определения *сложных видов файловых данных*. В заданиях каждой из двух первых подгрупп используются все три механизма *позиционирования* (основанные на явном указании позиции или на применении локальных или разделяемых файловых указателей); в третьей группе применяются коллективные файловые операции, использующие, в основном, локальные файловые указатели. За рамками группы MPI6File остались лишь дополнительные возможности, связанные с неблокирующим файловым доступом, необходимость в котором возникает сравнительно редко.

В качестве примера задания, связанного с параллельным файловым вводом-выводом, рассмотрим задание MPI6File26, входящее в третью подгруппу.

MPI6File26. В главном процессе дано имя файла. Кроме того, в каждом процессе дано по 4 вещественных числа: *A*, *B*, *C*, *D*. Количество процессов равно *K*. Создать новый файл вещественных чисел с указан-

ным именем и записать в него исходные числа в следующем порядке (индекс указывает ранг процесса): A_0, A_1, …, A_{K-1}, B_{K-1}, …, B_1, B_0, C_0, C_1, …, C_{K-1}, D_{K-1}, …, D_0. Для этого использовать единственный вызов коллективной функции MPI_File_write_all, предварительно определив новый файловый вид данных с базовым типом MPI_DOUBLE, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из двух вещественных элементов (с дополнительным пустым промежутком между этими элементами) и завершающего пустого промежутка подходящего размера.

При выполнении этого задания мы познакомимся и с действиями, требующимися для открытия файла в параллельном режиме, и с одной из наиболее распространенных коллективных файловых операций MPI_File_write_all, и с приемами определения сложного вида файловых данных.

При ознакомительном запуске этого задания окно задачника примет вид, соответствующий приведенному на рис. 24.

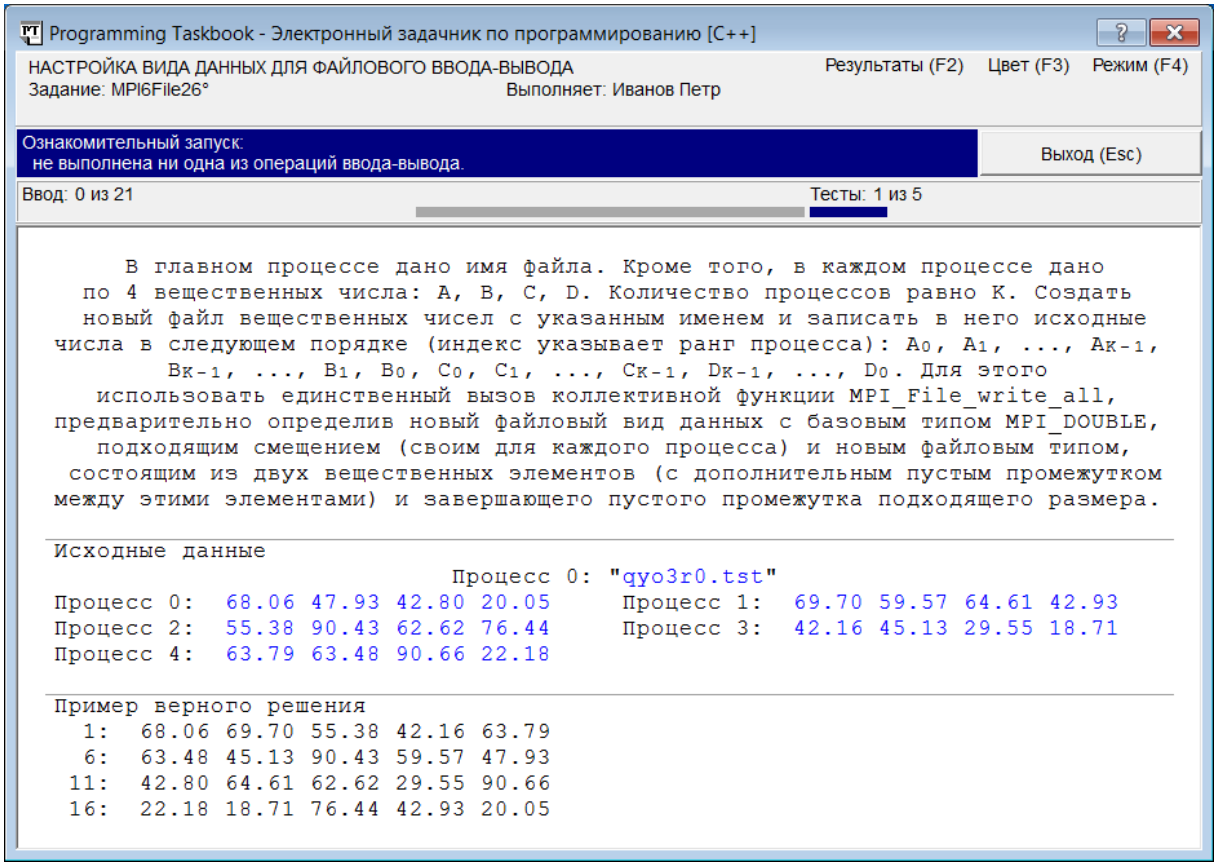


Рис. 24. Ознакомительный запуск задания MPI6File26

Задания на параллельный файловый ввод-вывод имеют ряд особенностей. Во-первых, в этих заданиях впервые появляются строковые исходные данные, содержащие имя файла. Это имя всегда дается в главном процессе; для того чтобы переслать его в другие процессы параллельного приложе-

ния, проще всего воспользоваться коллективной функцией MPI_Bcast. В преамбуле к группе MPI6File указано, что имя файла содержит не более 12 символов, поэтому для его хранения достаточно использовать массив char[12], а при пересылке указывать тип MPI_CHAR. Заметим, что для ввода строки s (типа char* или char[]) в задачнике достаточно использовать *единственное* обращение к потоку ввода pt: pt >> s.

Второй особенностью задания является включение в него *файловых данных*. В некоторых заданиях файловые данные указаны в разделе исходных данных (если в задании требуется обработать существующий файл), в других — в разделе результатов (если требуется преобразовать имеющийся файл или, как в нашем случае, создать новый). Файловые данные отображаются в окне задачника на нескольких строках, причем в начале каждой строки указывается текущий номер файлового элемента (элементы нумеруются от 1). Во всех заданиях используются *типизированные двоичные файлы*, состоящие либо из целых, либо из вещественных чисел. Заметим, что содержимое таких файлов нельзя просмотреть в обычных текстовых редакторах, поэтому возможность их отображения в окне задачника оказывается особенно полезной. Способ разбиения файловых элементов на строки в окне задачника зависит от особенностей задания. В нашем случае требуется записать в файл вначале первые элементы, данные в каждом процессе, затем — вторые элементы (в обратном порядке), затем — третьи элементы и, наконец, четвертые элементы (также в обратном порядке). Поэтому удобнее всего представить содержимое файла в виде *четырех* строк, каждая из которых содержит элементы всех процессов, имеющие один и тот же порядковый номер.

Следует также обратить внимание на особый вид *индикатора вывода*, который в данном случае отображается серым цветом и не содержит сопутствующего текста. Это означает что, результаты, полученные при выполнении задания, не следует пересылать задачнику (используя поток вывода pt); необходимо лишь требуемым образом заполнить файл с указанным именем, после чего задачник сам проверит правильность созданного файла.

Приступим к выполнению задания. На первом этапе организуем ввод исходных данных и пересылку имени файла во все процессы:

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

После запуска данного варианта в окне задачника будет выведено сообщение о том, что все исходные данные введены, а результирующий файл не создан (рис. 25).

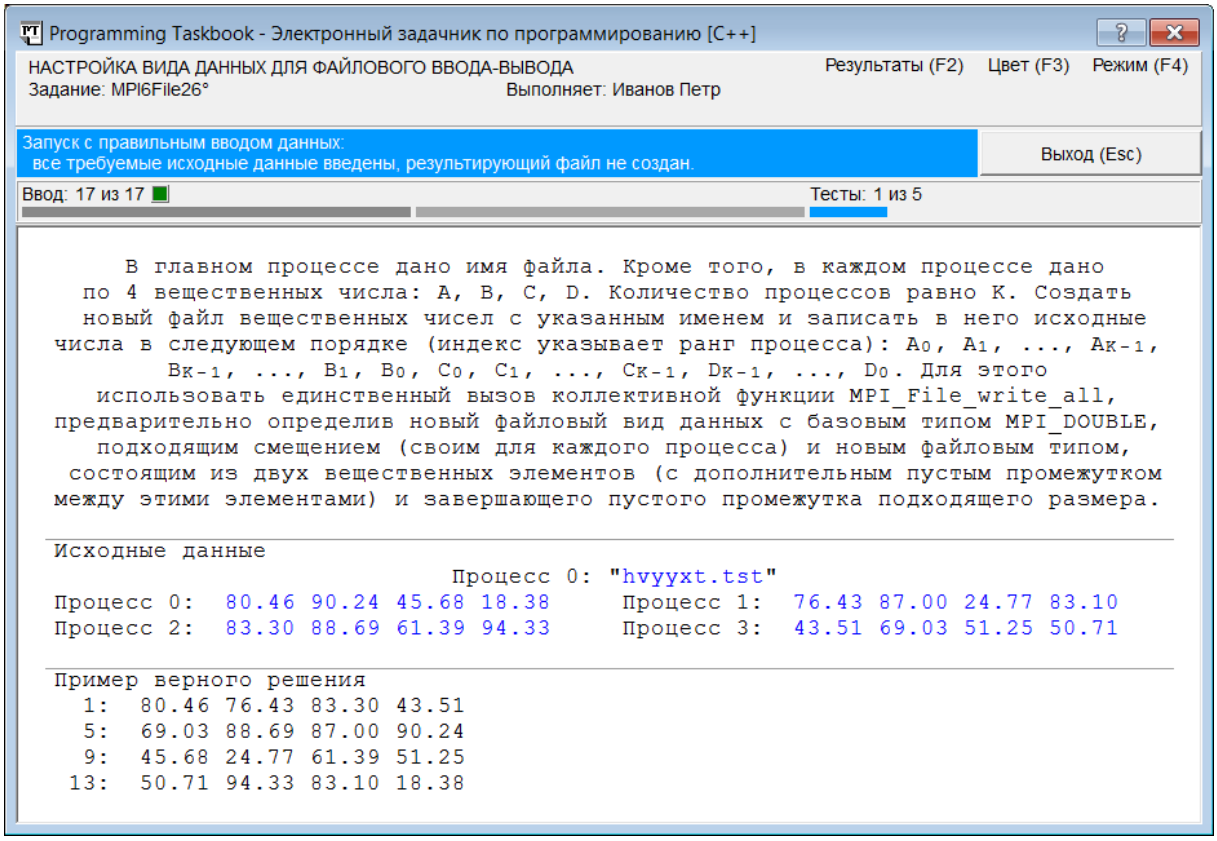


Рис. 25. Первый этап выполнения задания MPI6File26 (ввод исходных данных)

На следующем этапе выполним действия, связанные с открытием и закрытием файла:

```
MPI_File f;
MPI_File_open(MPI_COMM_WORLD, name,
  MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);
MPI_File_close(&f);
```

Вначале описывается файловая переменная f, которая будет использоваться во всех функциях, связанных с файловыми операциями. Затем для открытия файла вызывается функция MPI_File_open. Эта функция является коллективной и должна вызываться во всех процессах коммуникатора, указанного в качестве ее первого параметра. Имя файла указывается во втором параметре, а режим доступа — в третьем; оба эти параметра должны иметь одинаковые значения во всех процессах. Для комбинирования характеристик режима доступа они должны объединяться операцией |. Например, в нашем случае необходимо создать файл (MPI_MODE_CREATE), после чего открыть его на запись (MPI_MODE_WRONLY). Из числа других характеристик можно отметить MPI_MODE_RDONLY — открытие только для чтения и MPI_MODE_RDWR — открытие на чтение и запись.

Следующий параметр типа MPI_Info позволяет задать дополнительные характеристики файла; если в этом нет необходимости (как обычно и бывает), то в качестве него указывается константа MPI_INFO_NULL.

Функция MPI_File_open возвращает переменную, связанную с открытым файлом. Эта переменная является выходным параметром, поэтому должна передаваться с помощью указателя.

После завершения работы с файлом его надо закрыть функцией MPI_File_close, которая, как и MPI_File_open, является коллективной и должна вызываться во всех процессах, в которых файл был открыт. Файловая переменная передается в эту функцию также в виде указателя; в результате выполнения функции MPI_File_close значение файловой переменной становится равным MPI_FILE_NULL.

При запуске нового варианта программы сообщение на информационной панели немного изменится: «*Запуск с правильным вводом данных: все требуемые исходные данные введены, результирующий файл является пустым*». Это сообщение показывает, что файл действительно был создан.

Заметим, что после завершения программы, выполняющей задание, задачник автоматически удаляет все файлы, созданные в ходе ее работы. Убедиться, что требуемый файл действительно был создан, можно, если просмотреть содержимое рабочего каталога *перед* закрытием окна задачника.

Следующий этап решения состоит в настройке вида файловых данных. Для этого предусмотрена функция MPI_File_set_view с большим количеством параметров:

```
int MPI_File_set_view(MPI_File f, MPI_Offset disp, MPI_Datatype
  etype, MPI_Datatype filetype, char* datarep, MPI_Info info)
```

В качестве первого параметра указывается файловая переменная, связанная с открытым файлом. Три следующих параметра определяют основные характеристики вида файловых данных:

disp — начальное смещение (в байтах), которое будет выполнено в данном процессе перед считыванием первого файлового элемента;

etype — базовый (элементарный) тип данных, на основе которого определяется файловый тип (во всех процессах, использующих данный файл, необходимо указывать один и тот же базовый тип данных);

filetype — файловый тип, используемый при непосредственном чтении или записи файловых данных; представляет собой набор данных базового типа, а также пустых промежутков (размер которых также должен быть кратным размеру базового типа).

Позиционирование в файле всегда выполняется в элементах базового (т. е. элементарного) типа, причем пустые промежутки, входящие в файловый тип, не учитываются. Заметим, что по умолчанию (если функция

MPI_File_set_view не вызвана) начальное смещение disp полагается равным 0, а типы etype и filetype полагаются равными MPI_BYTE.

Параметр datarep определяет, каким образом следует интерпретировать файловые данные. В простейшем случае, когда все процессы выполняются на одном и том же компьютере (или на различных компьютерах, имеющих одинаковую архитектуру), достаточно использовать вариант «native», при котором файловое представление в точности совпадает с представлением таких же данных в оперативной памяти. В более сложных ситуациях (в частности, когда программа выполняется на гетерогенном оборудовании) применяются другие варианты представления данных, при которых данные, хранящиеся в памяти, перед записью в файл подвергаются дополнительному преобразованию (и аналогичное преобразование выполняется при считывании данных из файла в память). В нашем случае достаточно использовать вариант «native».

Наконец, параметр info позволяет указать дополнительную информацию, связанную с определяемым видом данных. Обычно в качестве этого параметра указывается MPI_INFO_NULL.

Определим вид файловых данных, который будет наиболее удобен для решения нашей задачи. Если обозначить через [R] файловые элементы (типа MPI_DOUBLE), которые требуется записать в процессе R (R = 0, 1, …, K–1), то распределение этих элементов в файле будет следующим:

[0][1][2]…[K-1][K-1]…[2][1][0][0][1] [2]…[K-1][K-1]…[2][1][0]

Таким образом, файл будет содержать два фрагмента одинаковой структуры (первый фрагмент выделен полужирным шрифтом). Нам достаточно определить файловый вид, позволяющий правильно прочесть данные из первого фрагмента и перейти ко второму фрагменту.

В качестве базового типа естественно выбрать тип MPI_DOUBLE. В этом случае файловый тип для любого процесса должен иметь размер 2*K вещественных элементов (т. е. размер фрагмента, выделенного полужирным шрифтом). При этом в этот файловый тип будут входить два вещественных числа, между которыми располагается пустой промежуток. Размер промежутка зависит от ранга процесса, для процесса 0 он наибольший и равен размеру 2*K–2 вещественных чисел, для процесса K–1 промежуток между элементами отсутствует.

Таким образом, если для процесса ранга R задать начальное смещение disp равным R*dbl_sz, где dbl_sz равен размеру типа MPI_DOUBLE в байтах, то файловые типы для различных процессов можно будет представить следующим образом (элемент типа MPI_DOUBLE обозначается через [*], а в круглых скобках указывается размер пустого промежутка в элементах типа MPI_DOUBLE):

процесс 0: [*](2*K–2)[*]

процесс 1: [*](2*K–4)[*](2)

процесс 1: [*](2*K–6)[*](4)

…

процесс K–1: [*][*](2*K–2)

Обратите внимание на то, что размер файлового типа для всех процессов является одинаковым и равен 2*K (в элементах типа MPI_DOUBLE); одинаковым также является и суммарный пустой промежуток, равный 2*K–2.

Опишем два варианта определения требуемого файлового типа, предполагая, что число процессов K хранится в переменной size, ранг процесса R хранится в переменной rank, а размер типа MPI_DOUBLE хранится в переменной dbl_sz. В первом варианте будем использовать только средства стандарта MPI-1:

```
MPI_Datatype t;
int block_cnt[] = {1, 1, 1};
MPI_Datatype block_type[] = {MPI_DOUBLE, MPI_DOUBLE, MPI_UB};
int block_displ[] = {0, (2*(size - rank) - 1)*dbl_sz,
  2*size*dbl_sz};
MPI_Type_struct(3, block_cnt, block_displ, block_type, &t);
```

В данном случае мы создали структуру из трех блоков размера 1, причем два первых блока содержат по одному вещественному числу, а последний блок содержит «фиктивный» элемент MPI_UB — *метку нулевого размера*, благодаря которой можно задать завершающий пустой промежуток для определяемого типа данных. Напомним, что смещения для каждого блока (задаваемые в массиве block_displ) указываются в байтах и отсчитываются от начала первого блока.

В стандарте MPI-2 появились новые средства, позволяющие определять начальные и конечные пустые промежутки для производных типов более удобным способом (по этой причине в этом стандарте тип MPI_UB объявлен устаревшим и не рекомендован к использованию). Используя подход, рекомендованный в MPI-2, мы должны вначале определить тип, содержащий два вещественных числа с необходимым промежутком между ними, а затем дополнить этот тип завершающим промежутком:

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

В этом варианте нам не потребовалось использовать массивы, так как для создания типа из двух вещественных чисел с промежутком между ними достаточно воспользоваться функцией MPI_Type_vector (напомним, что третий параметр этой функции определяет расстояние между началом соседних блоков, которое измеряется не в байтах, а в элементах). После создания этого вспомогательного типа t0 мы применили к нему функцию MPI_Type_create_resized, которая появилась в стандарте MPI-2 и позволяет

дополнить любой тип некоторым начальным и конечным пустым промежутком: второй параметр определяет новую «нижнюю границу» типа (если добавлять начальный промежуток не требуется, то второй параметр полагается равным 0), а третий определяет суммарную протяженность нового типа (протяженность увеличивается за счет добавления завершающего пустого промежутка); оба параметра указываются в байтах.

Для определения размера типа MPI_DOUBLE достаточно использовать функцию MPI_Type_size:

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

После определения типа t нам останется настроить вид файловых данных, вызвав функцию MPI_File_set_view, и вызвать коллективную функцию записи MPI_File_write_all, передав в нее массив a из 4 введенных вещественных чисел:

```
MPI_File_set_view(f, rank * dbl_ext, MPI_DOUBLE, t, "native",
  MPI_INFO_NULL);
MPI_File_write_all(f, a, 4, MPI_DOUBLE, MPI_STATUS_IGNORE);
```

Разумеется, все эти действия надо выполнять между операциями открытия и закрытия файла.

Приведем полный текст решения, в котором использован второй из описанных вариантов определения файлового типа:

```
void Solve()
{
  Task("MPI6File26");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  char name[20];
  if (rank == 0)
    pt >> name;
  MPI_Bcast(name, 20, MPI_CHAR, 0, MPI_COMM_WORLD);
  double a[4];
  for (int i = 0; i < 4; ++i)
    pt >> a[i];
  MPI_File f;
  MPI_File_open(MPI_COMM_WORLD, name,
    MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);
```

```
  int dbl_sz;
  MPI_Type_size(MPI_DOUBLE, &dbl_sz);
  MPI_Datatype t0, t;
  MPI_Type_vector(2, 1, 2 * (size - rank) - 1, MPI_DOUBLE, &t0);
  MPI_Type_create_resized(t0, 0, 2 * size * dbl_sz, &t);
  MPI_File_set_view(f, rank * dbl_sz, MPI_DOUBLE, t, "native",
    MPI_INFO_NULL);
  MPI_File_write_all(f, a, 4, MPI_DOUBLE, MPI_STATUS_IGNORE);
  MPI_File_close(&f);
}
```

Запустив программу, мы получим сообщение о том, что задание выполнено.

2.6. Односторонние коммуникации (MPI-2): MPI7Win13, MPI7Win23

Односторонние коммуникации, появившиеся в стандарте MPI-2, позволяют организовать пересылку данных между процессами, не выполняя специальных действий на стороне обоих участников взаимодействия. Напомним, что в традиционной схеме взаимодействия двух процессов требуется вызов одной из функций посылки (MPI_Send или ее вариантов, как блокирующего, так и неблокирующего) на стороне процесса-*источника* (source process, или sender), а также одной из функций приема (MPI_Recv или ее неблокирующего варианта) на стороне процесса-*приемника* (destination process, или receiver). Однако возможна ситуация, когда процесс-источник «не знает», какая часть из его данных потребуется другим процессам, или, наоборот, процесс-приемник «не знает», какие данные будут посланы ему другими процессами. В последнем случае, правда, имеется возможность использовать параметры MPI_ANY_SOURCE и MPI_ANY_TAG, позволяющие получать данные от произвольных источников, а также определять характер полученных данных на основе переданной с ними метки msgtag. Но подобная возможность требует, как правило, предварительного анализа полученного сообщения (путем вызова функции MPI_Probe или ее неблокирующего варианта) и, тем самым, усложняет алгоритм действий на стороне процесса-приемника. Если же процесс-источник «не знает», кому могут потребоваться его данные, то применение традиционного механизма MPI «отправка–получение» становится принципиально невозможным.

Между тем, в технологии многопоточного программирования именно односторонние взаимодействия являются стандартным способом обмена информацией: поток может записать данные в некоторую область *разделяемой памяти*, после чего любой другой поток сможет обратиться к этой памяти и получить требуемую часть данных. Таким образом, односторон-

ние взаимодействия основаны на концепции *разделяемой памяти*. Следовательно, для реализации такого вида взаимодействий в технологии MPI необходимо, чтобы процессы параллельного MPI-приложения имели возможность определять «разделяемые участки» своей памяти, к которым напрямую могли бы обращаться другие процессы этого же приложения. По этой причине механизм односторонних коммуникаций (one-sided communications) в MPI также называется *удаленным доступом к памяти* (remote memory access, RMA).

Участок памяти некоторого процесса, к которому может обратиться любой процесс параллельного MPI-приложения, называется *окном* (window). Для создания окна предназначена коллективная функция MPI_Win_create, которая должна вызываться для всех процессов некоторого коммуникатора. Один вызов этой функции позволяет создавать разделяемые участки памяти (т. е. окна) во *всех* процессах, входящих в указанный коммуникатор; при этом все эти участки будут связаны с одним и тем же *дескриптором окна* типа MPI_Win. Используя данный дескриптор, любой процесс может получить доступ к окну любого другого процесса. Окна, определенные в разных процессах, могут иметь различные размеры; если для некоторых процессов создавать окно не требуется, то при вызове функции MPI_Win_create в этом процессе достаточно положить равным нулю параметр size, отвечающий за размер окна.

Для доступа к окну в библиотеке MPI предусмотрены три функции: MPI_Get (доступ на чтение), MPI_Put (доступ на запись) и MPI_Accumulate (доступ на изменение). Процесс, вызывающий эти функции, называется *инициирующим процессом* (origin process), процесс, содержащий то окно, к которому выполняется доступ, называется *целевым процессом* (target process). Именно инициирующий процесс обеспечивает одностороннюю коммуникацию, в то время как целевой процесс играет пассивную роль, не выполняя никаких специальных действий. При этом и инициирующий, и целевой процессы могут выступать как в роли источника, так и в роли приемника данных. Если используется операция MPI_Get, то инициирующий процесс является приемником данных, а целевой процесс — источником; если используется операция MPI_Put или MPI_Accumulate, то инициирующий процесс является источником, а целевой процесс — приемником данных.

Очень важным (и наиболее сложным) аспектом механизма односторонних коммуникаций является *синхронизация* доступа к окну. Поскольку, в отличие от стандартной схемы двустороннего обмена «отправка–получение», целевой процесс не выполняет специальных действий при односторонних обменах, необходимо предпринимать дополнительные усилия по согласованию доступа к данным, размещенным в окне. В частности, если целевой процесс играет роль приемника данных (в этом случае он на-

зывается *активным целевым процессом* — active target), то он должен знать, когда можно обратиться к окну для чтения полученных данных, если же приемником данных выступает инициирующий процесс, то он должен знать, когда можно обратиться к данным, полученным из целевого процесса. В обоих случаях требуется синхронизация. Единственным исключением, при котором на стороне целевого процесса синхронизация не требуется, является вариант одностороннего взаимодействия с так называемым *пассивным целевым процессом* (passive target), при котором целевой процесс вовсе не обращается к своему окну. Окно пассивного целевого процесса используется в роли *хранилища данных*, к которому обращаются другие процессы параллельного приложения (подобный вариант односторонних коммуникаций наиболее близок модели с разделяемой памятью, применяемой в многопоточном программировании).

Функции синхронизации позволяют задавать так называемые *периоды доступа* (access epoch) на стороне инициирующих процессов и *периоды предоставления доступа* (exposure epoch) на стороне активных целевых процессов. Все результаты односторонних взаимодействий, выполненных в течение периода доступа (и согласованного с ним периода предоставления доступа) будут доступны процессу *только при завершении данного периода*. Иными словами, пока не завершен текущий период доступа, инициирующий процесс не должен обращаться к данным, полученным с помощью функций MPI_Get, и пока не завершен период предоставления доступа, активный целевой процесс не должен обращаться к своему окну для чтения данных, переданных в это окно с помощью функций MPI_Put или MPI_Accumulate.

Простейший вариант синхронизации предоставляет функция MPI_Win_fence (английское слово «fence» может быть переведено как «ограда», или «забор»). Это коллективная функция, которая должна вызываться во всех процессах, в которых определено окно. Первый вызов этой функции начинает первый период доступа (и связанный с ним период предоставления доступа) для данного окна. Каждый последующий вызов этой функции завершает предыдущий период доступа (и связанный с ним период предоставления доступа) и одновременно начинает новый период доступа (и связанный с ним новый период предоставления доступа). Таким образом, при использовании данной функции синхронизации в каждом процессе с окном необходимо выполнить по крайней мере *два* вызова этой функции. Подобный вариант синхронизации используется, когда в течение периода доступа многие процессы выступают в роли активных целевых процессов. Его ограничением является глобальный характер: единая синхронизация устанавливается для всех процессов, для которых определено данное окно.

Другой вариант синхронизации позволяет выполнять ее *локально*, определяя период доступа (и связанный с ним период предоставления доступа) только для некоторых из процессов, в которых определено окно. За подобную гибкость приходится платить более сложным способом настройки периодов доступа, при котором для начала и окончания как периода доступа, так и периода предоставления доступа предусматриваются особые функции:

- функция MPI_Win_start начинает период доступа для всех процессов, в которых она вызвана, причем в ней указывается группа процессов, которые могут выступать в роли активных целевых процессов для этого периода;
- функция MPI_Win_complete завершает период доступа, начатый функцией MPI_Win_start;
- функция MPI_Win_post начинает период предоставления доступа для всех процессов, в которых она вызвана, причем в ней указывается группа процессов, которые могут выступать в роли инициирующих процессов для этого периода;
- функция MPI_Win_wait завершает период предоставления доступа, начатый функцией MPI_Win_post.

Оба рассмотренных варианта синхронизации предполагают, что целевые процессы являются активными. Для варианта односторонних коммуникаций, в которых целевые процессы являются пассивными, т. е. не обращаются к своим окнам, предусмотрен третий вариант синхронизации, основанный на *блокировке*. Основной особенностью блокирующего варианта синхронизации является то, что в этом варианте целевой процесс не должен вызывать никаких функций синхронизации (и, следовательно, для него не задается период предоставления доступа); специальные функции синхронизации в этом варианте предназначены только для начала и окончания периода доступа в инициирующих процессах. Для начала блокирующего периода доступа инициирующий процесс должен вызвать функцию MPI_Win_lock, для завершения блокирующего периода доступа предназначена функция MPI_Win_unlock. На протяжении данного периода блокирующего доступа инициирующий процесс может обращаться только к окну указанного целевого процесса. Блокировка может быть *эксклюзивной* (exclusive lock) или *совместной* (shared lock). Если какой-либо процесс пытается организовать доступ с эксклюзивной блокировкой к целевому процессу, для которого в данный момент эксклюзивная блокировка уже установлена (другим процессом), то предоставление доступа откладывается до того момента, когда ранее установленная эксклюзивная блокировка будет снята (режим с эксклюзивной блокировкой обычно применяется, если несколько инициирующих процессов организуют доступ к окну целевого процесса на запись или изменение). В отличие от эксклюзивной блоки-

ровки, несколько процессов могут одновременно организовать доступ с совместной блокировкой к одному и тому же целевому процессу (режим с совместной блокировкой применяется, если инициирующие процессы обращаются к окну целевого процесса только для чтения).

Задания группы MPI7Win позволяют познакомиться со всеми аспектами механизма односторонних коммуникаций. В первой подгруппе этой группы (MPI7Win1– MPI7Win17) используется простейшая синхронизация, основанная на использовании коллективной функции MPI_Win_fence; при этом рассматриваются различные варианты применения всех трех видов односторонних коммуникаций (с доступом на чтение, запись и изменение); при этом в начальных задачах этой подгруппы (MPI7Win1–MPI7Win6) разделяемая память в рамках окна доступа создается только в одном процессе, а в остальных задачах разделяемые участки памяти создаются в группах процессов или во всех процессах приложения. Во второй подгруппе изучаются более сложные виды синхронизации: локальная синхронизация, основанная на применении четырех функций MPI_Win_start, MPI_Win_complete, MPI_Win_post, MPI_Win_wait (MPI7Win18–MPI7Win23), и синхронизация с блокировкой, основанная на применении функций MPI_Win_lock и MPI_Win_unlock (MPI7Win24–MPI7Win27, MPI7Win29); в двух задачах (MPI7Win28, MPI7Win30) требуется использовать оба варианта синхронизации, рассматриваемых во второй подгруппе.

Чтобы продемонстрировать различные возможности, связанные с односторонними коммуникациями, рассмотрим две задачи — по одной из каждой подгруппы группы MPI7Win.

Начнем с задачи MPI7Win13, в которой используется простейшая синхронизация, основанная на функции MPI_Win_fence и, кроме того, применяется наиболее сложная из функций доступа MPI_Accumulate, обеспечивающая доступ к окну на преобразование.

MPI7Win13. В каждом процессе даны три целых числа N_1, N_2, N_3, каждое из которых лежит в диапазоне от 0 до $K - 1$, где K — количество процессов (значения некоторых из этих чисел в каждом процессе могут совпадать). Кроме того, в каждом процессе дан массив A вещественных чисел размера $R + 1$, где R — ранг процесса (0, …, $K - 1$). Во всех процессах определить окно доступа, содержащее массив A, и, используя по три вызова функции MPI_Accumulate в каждом процессе, добавить ко всем элементам массива A в процессах рангов N_1, N_2 и N_3 вещественное число, равное $R + 1$, где R — ранг процесса, вызвавшего функции MPI_Accumulate (например, если число N_1 в процессе ранга 3 равно 2, то ко всем элементам массива A из процесса 2 надо добавить вещественное число 4.0). Если некоторые из чисел N_1, N_2, N_3 в процессе R совпадают,

то числа $R + 1$ надо добавлять к элементам соответствующих массивов несколько раз. Вывести измененные массивы A в каждом процессе.

Приведем окно задачника при ознакомительном запуске программы с заготовкой для данного задания (рис. 26). Для уменьшения размеров окна в нем скрыт раздел с формулировкой задания.

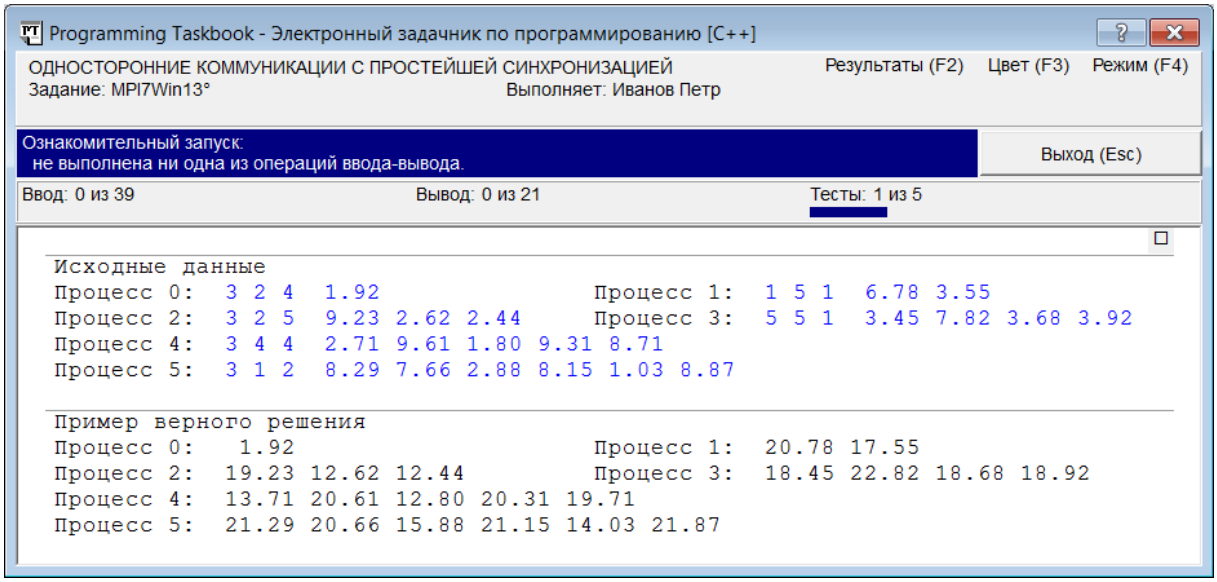


Рис. 26. Ознакомительный запуск задания MPI7Win13

Требуемое в задании преобразование исходных массивов A достаточно сложно (хотя и возможно) реализовать с помощью традиционных средств MPI. Основной проблемой является то обстоятельство, что процесс-приемник не знает, от каких процессов-источников он получит данные, которые надо добавить к элементам своего массива A. В то же время, с применением механизма односторонних коммуникаций требуемое преобразование реализуется достаточно просто. Поскольку в данном случае все процессы выступают в роли инициирующих процессов и, кроме того, большинство процессов одновременно являются также и целевыми процессами, коллективный вариант синхронизации является наиболее оправданным.

В примере, приведенном на рис. 26, только один процесс не является целевым: это процесс 0, для которого содержимое массива A не изменится (данное обстоятельство связано с тем, что среди целых чисел, данных в процессах, нет ни одного числа, равного 0). Следует также обратить внимание на то, что в некоторых случаях один и тот же процесс выступает и в роли инициирующего, и в роле целевого (например, процесс 2 должен увеличить элементы своего массива, так как среди целых чисел, данных в этом процессе, имеется число 2). Кроме того, для большинства процессов содержащиеся в них массивы будут изменены несколькими инициирующими процессами (например, массив из процесса 5 будет изменен процессами 1, 2 и 3, причем процесс 3 изменит этот массив дважды; в результате

элементы массива из процесса 5 будут увеличены на величину 2 + 3 + 4 + 4 = 13).

На начальном этапе решения опишем все необходимые массивы и обеспечим их заполнение исходными данными. Помимо массива a, данного в условии задачи, опишем массив n из трех элементов, содержащий данные целые числа N1, N2, N3, а также массив b, который будет содержать значения, добавляемые инициирующим процессом к массивам a целевых процессов; в качестве размера массива b достаточно задать максимальный из размеров массивов a, равный максимальному рангу плюс 1, или, иными словами, равный количеству процессов size.

Приведем первую часть решения:

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

При запуске этого варианта программы мы получим сообщение о том, что все требуемые исходные данные введены, а результаты не выведены.

Теперь необходимо определить окно доступа, содержащее те участки памяти, к которым могут обращаться другие процессы приложения. В нашем случае в окно доступа должны входить массивы a из каждого процесса. Приведем соответствующий фрагмент кода, а затем прокомментируем его:

```
int dbl_sz;
MPI_Type_size(MPI_DOUBLE, &dbl_sz);
MPI_Win w;
MPI_Win_create(a, (rank + 1) * dbl_sz, dbl_sz, MPI_INFO_NULL,
  MPI_COMM_WORLD, &w);
```

Вначале мы определяем в переменной dbl_sz размер типа MPI_DOUBLE в байтах (используя отладочный вывод, можно убедиться, что в нашем случае он равен 8). Затем мы создаем окно w, используя функцию MPI_Win_create. Выше мы уже отмечали, что данная функция является коллективной и поэтому должна вызываться во всех процессах того коммуникатора, для которого создается окно (мы создаем окно для коммуникатора MPI_COMM_WORLD, который указывается в качестве предпоследнего параметра данной функции).

Первый параметр функции задает адрес начала участка памяти, связываемого с окном в данном процессе; в нашем случае это всегда начало

массива a. Затем указывается размер этого участка памяти в байтах (напомним, что допустимо указывать размер, равный 0; это означает, что в данном процессе с окном не связывается никакой участок памяти). Третий параметр функции MPI_Win_create имеет имя disp_unit; он предназначен для упрощения адресной арифметики, используемой при вызовах функций доступа: величина смещения он начала окна, указываемая в функциях доступа, автоматически умножается на значение disp_unit. Таким образом, если в качестве параметра disp_unit указать размер элемента массива (как в нашем случае), то в дальнейшем для обращения к тому или иному элементу массива в функциях доступа будет достаточно указать его *индекс* (если в качестве параметра disp_unit указать 1, то смещение в функциях доступа придется указывать в *байтах*, что менее удобно). Четвертый параметр info типа MPI_Info позволяет связать с создаваемым окном дополнительную информацию; обычно он не используется и в этом случае полагается равным константе MPI_INFO_NULL. В последнем параметре возвращается дескриптор созданного окна, который необходимо указывать во всех функциях, используемых при работе с данным окном.

После завершения работы с окном доступа его необходимо разрушить, вызвав функцию MPI_Win_free; дескриптор разрушенного окна получает значение MPI_WIN_NULL, означающее, что с данным окном нельзя выполнять никакие операции. Кроме того, в нашей программе необходимо освободить память, выделенную для динамических массивов a и b. Поэтому добавим в функцию Solve следующие завершающие операторы:

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

Результат запуска нового варианта программы не будет отличаться от прежнего. Все последующие дополнения к решению необходимо указывать *перед* фрагментом, приводящим к разрушению окна доступа.

Теперь нам необходимо организовать доступ к созданному окну из различных процессов. Поскольку такой доступ возможен только в пределах периода доступа, надо начать такой период, вызвав функцию MPI_Win_fence:

```
MPI_Win_fence(0, w);
```

Кроме основного параметра w, определяющего окно, для которого начинается период доступа, данная функция включает параметр assert, который может содержать набор констант, уточняющих характер определяемого периода доступа (например, константа MPI_MODE_NOPUT означает, что в течение данного периода доступа не будут выполняться действия, связанные с *изменением* содержимого окна функциями MPI_Put или MPI_Accumulate). Подобные константы позволяют управляющей среде MPI оптимизировать действия, выполняемые при односторонних комму-

никациях. Если характер периода доступа не требуется уточнять, то в качестве параметра assert указывается число 0.

Начав период доступа, мы можем вызывать функции доступа к окну. В данном случае нам требуется трижды вызвать функцию MPI_Accumulate в каждом процессе. Для этого организуем цикл:

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

Большинство параметров всех функций доступа (MPI_Get, MPI_Put, MPI_Accumulate) являются одинаковыми. Первые три параметра определяют данные «на стороне» инициирующего процесса (и поэтому имеют имена origin_addr, origin_count и origin_datatype). Первый из них указывает адрес начало буфера данных, второй — количество элементов в буфере, а третий — тип элементов буфера. Следующие четыре параметра определяют данные «на стороне» целевого процесса (и поэтому имеют имена target_rank, target_disp, target_count и target_datatype). Первый параметр задает ранг целевого процесса, второй — смещение от начала окна в этом целевом процессе (в единицах disp_unit, указанных при определении окна — см. выше описание функции MPI_Win_create), третий и четвертый определяют количество и тип элементов окна, к которым выполняется доступ (подчеркнем, что количество измеряется в элементах указанного типа и никак не связано с единицами disp_unit). Последним параметром во всех функциях доступа указывается дескриптор окна. Если типы элементов в буфере инициирующего процесса и в окне целевого процесса совпадают (как в нашем случае), то должны совпадать и указываемые размеры (origin_count и target_count).

Единственной особенностью функции MPI_Accumulate является указание дополнительного (предпоследнего) параметра op типа MPI_Op, который определяет характер изменения элементов окна. Поскольку в нашем случае требует добавить к исходным значениям элементов окна новые слагаемые, мы использовали операцию MPI_SUM.

Следует заметить, что функция MPI_Accumulate реализована таким образом, что ее можно безопасно использовать в случае, когда окно одного и того же целевого процесса изменяется несколькими инициирующими процессами (функция MPI_Put, очевидно, таким свойством не обладает: если два инициирующих процесса в ходе одного периода доступа попытаются записать свои данные в окно одного и того же целевого процесса, то в этом окне сохранятся только данные одного из инициирующих процессов, причем заранее нельзя сказать, какого именно).

Для того чтобы можно было получить доступ к измененным элементам окна доступа, *необходимо завершить период доступа* (и связанный с ним период предоставления доступа на стороне целевых процессов), в те-

чение которого выполнялись действия по изменению окна. Поэтому перед выводом измененного массива a *необходимо* еще один раз вызвать функцию синхронизации MPI_Win_fence:

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

После запуска последнего варианта программы мы получим сообщение о том, что задание выполнено.

Приведем полный текст решения:

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

```
  delete[] a;
  delete[] b;
}
```

Примечания. 1. Если функция MPI_Win_fence вызывается только для того, чтобы завершить последний период доступа, то этот факт можно отметить, указав в качестве параметра assert особое значение MPI_MODE_NOSUCCEED:

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

2. Оба вызова функции MPI_Win_fence в приведенной программе являются *обязательными*. Если закомментировать хотя бы один из них, то программа завершит работу без сообщений об ошибках MPI, однако полученные результаты будут отличаться от требуемых.

Теперь обратимся к заданию MPI7Win23 из второй подгруппы, в котором требуется использовать другой способ синхронизации.

MPI7Win23. Во всех процессах даны вещественные массивы A размера 5. Кроме того, в главном процессе даны целочисленные массивы N и M размера 5 каждый. Все элементы массива N лежат в диапазоне от 1 до K, где K — количество подчиненных процессов, все элементы массива M лежат в диапазоне от 0 до 4; некоторые элементы как в массиве N, так и в массиве M могут совпадать. В каждом подчиненном процессе определить окно доступа, содержащее массив A, и, используя требуемое количество вызовов функции MPI_Get в главном процессе, получить из процесса ранга N_I (I = 0, …, 4) элемент массива A с индексом M_I и добавить его значение к элементу массива A главного процесса с индексом I. После изменения массива A в главном процессе выполнить следующую корректировку массивов A всех подчиненных процессов: заменить в них те элементы, которые больше элемента массива A с тем же индексом из главного процесса, на этот элемент, используя требуемое количество вызовов функции MPI_Accumulate в главном процессе. В каждом процессе вывести преобразованные массивы A. Для синхронизации использовать два вызова пары функции MPI_Win_post и MPI_Win_wait в подчиненных процессах и два вызова пары функции MPI_Win_start и MPI_Win_complete в главном процессе.

На рис. 27 приведено окно задачника для данного задания. Как и для предыдущего задания, в окне скрыт раздел с формулировкой.

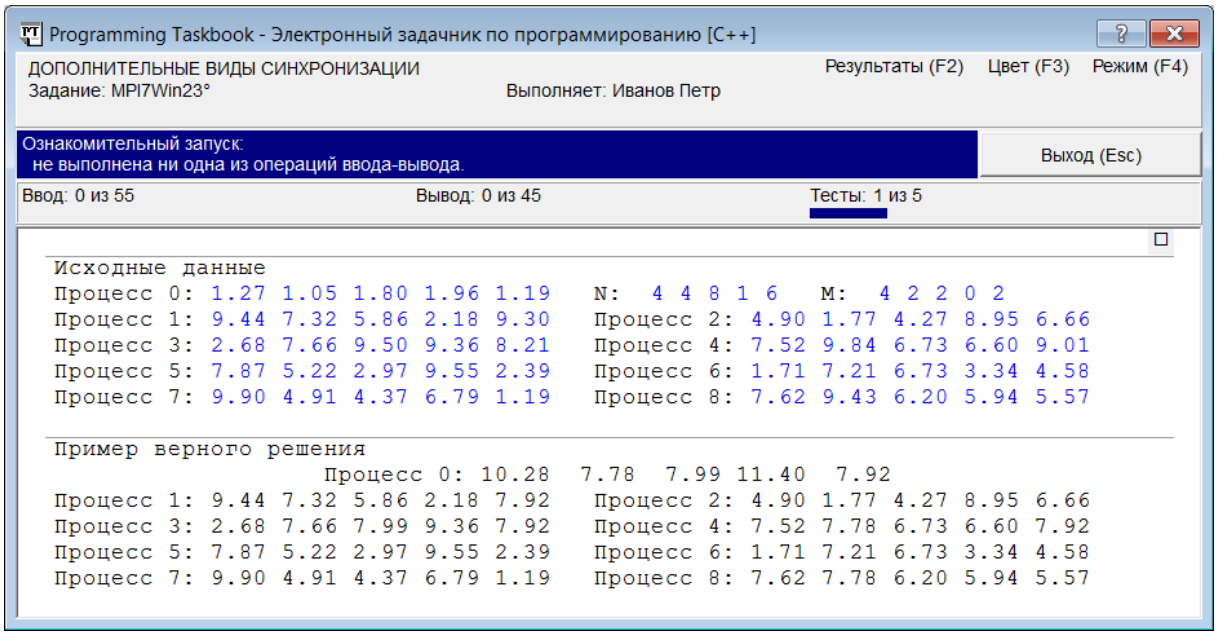


Рис. 27. Ознакомительный запуск задания MPI7Win23

В данном случае требуется создать окно для доступа к массивам A в подчиненных процессах; таким образом, в роли инициирующего процесса выступает главный процесс, а в роли целевых процессов — подчиненные процессы. В подобной ситуации имеет смысл использовать синхронизацию, учитывающую указанную специфику операций обмена.

Следует обратить внимание еще на одну особенность этого задания: в нем требуется реализовать два набора последовательно выполняемых односторонних обменов. Вначале необходимо изменить массив A в главном процессе; поскольку сам этот процесс выступает в роли инициирующего, для выполнения подобного действия следует использовать функцию MPI_Get. Затем измененный массив A из главного процесса следует использовать для корректировки некоторых элементов массива A в подчиненных процессах; поскольку в роли инициирующего процесса по-прежнему выступает главный процесс, в данном случае требуется применить функцию MPI_Accumulate. Так как начать корректировку массивов в подчиненных процессах необходимо только после изменения массива в главном процессе, в программе требуется использовать два периода доступа: в первом периоде выполняется корректировка массива в главном процессе, а во втором — корректировка массивов в подчиненных процессах.

На начальном этапе организуем ввод всех исходных данных и определение окна доступа:

```
int w_sz = 5;
int n[5], m[5];
double a[5];
for (int i = 0; i < 5; ++i)
  pt >> a[i];
```

```
if (rank == 0)
{
  w_sz = 0;
  for (int i = 0; i < 5; ++i)
    pt >> n[i];
  for (int i = 0; i < 5; ++i)
    pt >> m[i];
}
int dbl_sz;
MPI_Type_size(MPI_DOUBLE, &dbl_sz);
MPI_Win w;
MPI_Win_create(&a, w_sz * dbl_sz, dbl_sz, MPI_INFO_NULL,
  MPI_COMM_WORLD, &w);
// Выполнение односторонних обменов и вывод результата
MPI_Win_free(&w);
```

Запуск данного варианта программы приведет к сообщению о том, что все исходные данные введены, а результаты не выведены.

Следует обратить внимание на то, что некоторые исходные данные (массивы n и m) вводятся только в главном процессе, а также на то, что в главном процессе не создается разделяемый участок памяти (второй параметр size функции MPI_Win_create в главном окне имеет значение 0).

Оставшаяся часть решения должна быть помещена в позицию, помеченную комментарием «Выполнение односторонних обменов и вывод результата».

В функциях синхронизации MPI_Win_start и MPI_Win_post, которые требуется использовать в данном задании, необходимо указать группу целевых процессов и группу инициирующих процессов соответственно. Подобные группы удобнее всего получить из группы процессов, связанной с коммуникатором MPI_COMM_WORLD:

```
MPI_Group g0, g;
MPI_Comm_group(MPI_COMM_WORLD, &g0);
```

Имея группу g0 всех процессов, мы можем для получения группы целевых процессов просто *удалить* из группы g0 процесс ранга 0, а для получения группы инициирующих процессов взять первый элемент группы g0 (т. е. процесс ранга 0).

Таким образом, для реализации первого периода доступа в главном процессе (и связанного с ним периода предоставления доступа в подчиненных процессах) достаточно выполнить следующие действия:

```
 int b = 0;
 if (rank == 0)
 {
   MPI_Group_excl(g0, 1, &b, &g);
```

```
  MPI_Win_start(g, 0, w);
  MPI_Win_complete(w);
}
else
{
  MPI_Group_incl(g0, 1, &b, &g);
  MPI_Win_post(g, 0, w);
  MPI_Win_wait(w);
}
```

Первым параметром функций MPI_Win_start и MPI_Win_post является группа процессов, вторым — параметр assert, имеющий тот же смысл, что и одноименный параметр функции MPI_Win_fence (его достаточно положить равным 0), третий параметр определяет используемое окно доступа. Функции MPI_Win_complete и MPI_Win_wait, завершающие текущий период доступа, являются более простыми: в них указывается лишь окно доступа.

Между вызовами функций MPI_Win_start и MPI_Win_complete можно указывать функции доступа, в данном случае — функции MPI_Get, позволяющие получить элементы из подчиненных процессов, которые следует добавить к элементам массива a главного процесса. Для хранения данных, полученных из подчиненных процессов, выделим вспомогательный буфер вещественных чисел a0 размера 5 (размер буфера соответствует количеству получаемых из подчиненных процессов чисел). Обращаться к содержимому буфера можно только после завершения периода доступа (т. е. после вызова функции MPI_Win_complete), таким образом, для изменения и вывода массива a в главном процессе достаточно перед оператором MPI_Win_complete(w) добавить фрагмент

```
double a0[5];
for (int i = 0; i < 5; ++i)
  MPI_Get(&a0[i], 1, MPI_DOUBLE, n[i], m[i], 1, MPI_DOUBLE, w);
```

а после оператора MPI_Win_complete(w) — следующий фрагмент:

```
for (int i = 0; i < 5; ++i)
  a[i] += a0[i];
for (int i = 0; i < 5; ++i)
  pt << a[i];
```

В первом фрагменте в цикле заполняются элементы вспомогательного массива a0: значение элемента с индексом i полагается равным значению элемента массива a с индексом m[i], расположенного в процессе ранга n[i].

Во втором фрагменте полученные элементы массива a0 добавляются к соответствующим элементам массива a главного процесса, после чего выполняется вывод измененного массива.

При запуске данного варианта программы в окне задачника появится сообщение о том, что в подчиненных процессах не выведены результирующие данные, однако содержимое массива, выведенного в главном процессе, будет правильным (рис. 28).

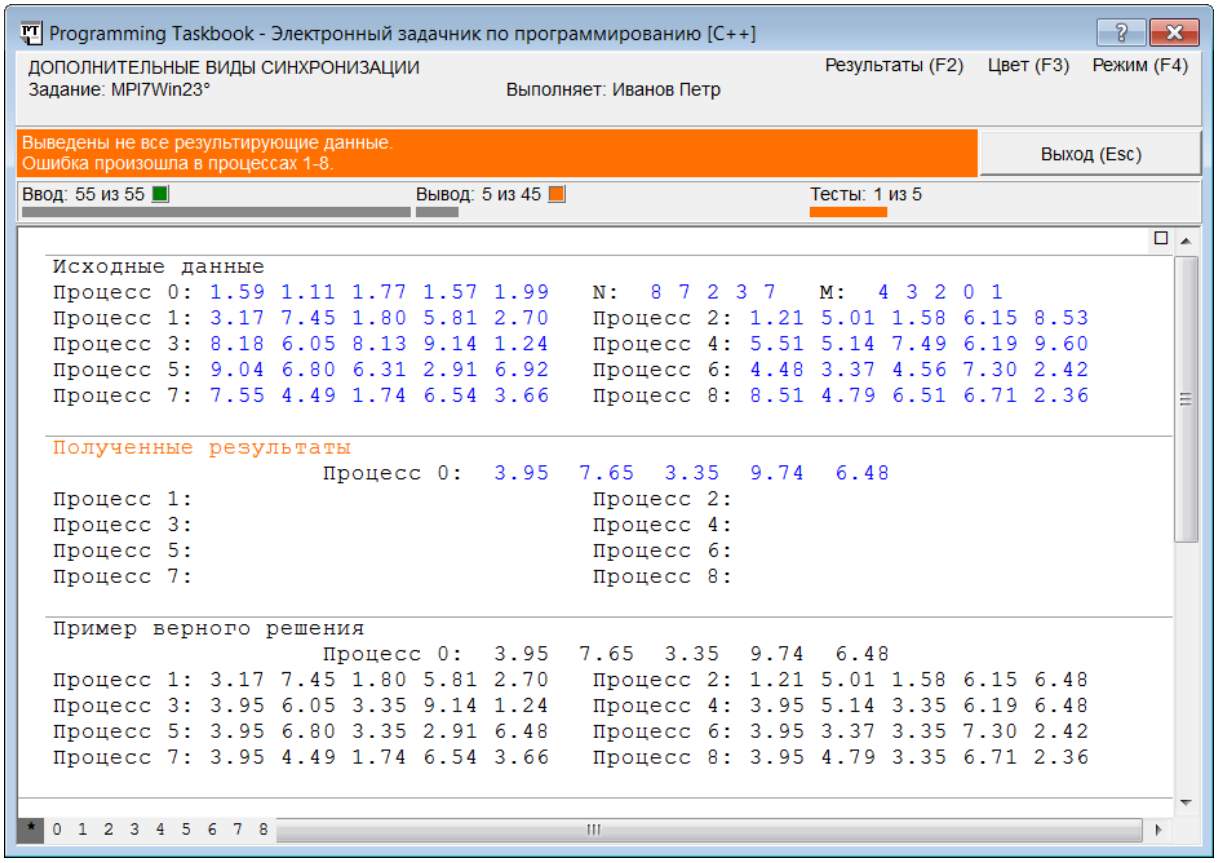


Рис. 28. Вид окна задачника после первого этапа выполнения задания MPI7Win23

Заметим, что между вызовами функций MPI_Win_post и MPI_Win_wait в подчиненных процессах нам не потребовалось выполнять никаких действий.

Осталось выполнить вторую часть односторонних обменов: добавить измененное содержимое массива a из главного процесса ко всем массивам в подчиненных процессах. Для этого надо начать новый период доступа в главном процессе и соответствующий ему период предоставления доступа в подчиненных процессах.

В пределах нового периода доступа требуется выполнить функцию MPI_Accumulate, передав данные из массива a главного процесса в окна всех подчиненных процессов:

```
MPI_Win_start(g, 0, w);
for (int i = 1; i < size; ++i)
  MPI_Accumulate(a, 5, MPI_DOUBLE, i, 0, 5, MPI_DOUBLE,
    MPI_MIN, w);
MPI_Win_complete(w);
```

В соответствии с условием задачи мы использовали в функции MPI_Accumulate операцию MPI_MIN.

В пределах нового периода предоставления доступа (в подчиненных процессах), как и для первого периода предоставления доступа, не требуется выполнять никаких действий, однако после завершения этого периода мы можем вывести измененное содержимое массива a:

```
MPI_Win_post(g, 0, w);
MPI_Win_wait(w);
for (int i = 0; i < 5; ++i)
  pt << a[i];
```

Запустив данный вариант программы, мы получим сообщение о том, что задание выполнено.

Вывод результатов в главном процессе и в подчиненных процессах можно объединить, если вынести соответствующий цикл из двух частей условного оператора и разместить его непосредственно перед оператором разрушения окна MPI_Win_free(&w).

Приведем окончательный вариант решения:

```
void Solve()
{
  Task("MPI7Win23");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int w_sz = 5;
  int n[5], m[5];
  double a[5];
  for (int i = 0; i < 5; ++i)
    pt >> a[i];
  if (rank == 0)
  {
    w_sz = 0;
    for (int i = 0; i < 5; ++i)
      pt >> n[i];
    for (int i = 0; i < 5; ++i)
      pt >> m[i];
```

```
  }
  int dbl_sz;
  MPI_Type_size(MPI_DOUBLE, &dbl_sz);
  MPI_Win w;
  MPI_Win_create(&a, w_sz * dbl_sz, dbl_sz, MPI_INFO_NULL,
    MPI_COMM_WORLD, &w);
  MPI_Group g0, g;
  MPI_Comm_group(MPI_COMM_WORLD, &g0);
  int b = 0;
  if (rank == 0)
  {
    MPI_Group_excl(g0, 1, &b, &g);
    MPI_Win_start(g, 0, w);
    double a0[5];
    for (int i = 0; i < 5; ++i)
      MPI_Get(&a0[i], 1, MPI_DOUBLE, n[i], m[i], 1,
        MPI_DOUBLE, w);
    MPI_Win_complete(w);
    for (int i = 0; i < 5; ++i)
      a[i] += a0[i];
    MPI_Win_start(g, 0, w);
    for (int i = 1; i < size; ++i)
      MPI_Accumulate(a, 5, MPI_DOUBLE, i, 0, 5, MPI_DOUBLE,
        MPI_MIN, w);
    MPI_Win_complete(w);
  }
  else
  {
    MPI_Group_incl(g0, 1, &b, &g);
    MPI_Win_post(g, 0, w);
    MPI_Win_wait(w);
    MPI_Win_post(g, 0, w);
    MPI_Win_wait(w);
  }
  for (int i = 0; i < 5; ++i)
    pt << a[i];
  MPI_Win_free(&w);
}
```

2.7. Интеркоммуникаторы и динамическое создание процессов (MPI-2): MPI8Inter9, MPI8Inter15

Интеркоммуникаторы (inter-communicators) появились уже в стандарте MPI-1. В отличие об «обычного» коммуникатора, называемого также интракоммуникатором (intra-communicator), который связывается с некоторой группой процессов и обеспечивает различные виды взаимодействия между любыми процессами, входящими в эту группу, интеркоммуникатор связывается с *двумя* группами процессов и предназначен для обеспечения взаимодействия процессов из *различных* групп; при этом используются ранги процессов в этих группах. Такой способ взаимодействия оказывается удобным, если параллельный алгоритм предполагает распределение действий между несколькими группами процессов и при этом требует обмена информацией между процессами, входящими в различные группы.

В стандарте MPI-2 концепция интеркоммуникаторов получила дальнейшее развитие: были расширены возможности по *созданию* интеркоммуникаторов, стали возможны *коллективные* взаимодействия процессов в рамках интеркоммуникаторов и, наконец, интеркоммуникаторы стали тем инструментом, который был положен в основу механизма *динамического создания процессов*.

С любым интеркоммуникатором связываются две равноправные группы процессов. Процесс из любой группы может инициировать обмен информацией с процессом из другой группы в рамках интеркоммуникатора, связывающего эти группы. При этом та группа, к которой относится процесс, вызывающий функцию для посылки или получения сообщения, называется *локальной группой* (local group), а группа, содержащие процессы, с которыми устанавливается связь, называется *удаленной группой* (remote group). Таким образом, для процесса-отправителя удаленной группой является та, в которой находится процесс-получатель, а для процесса-получателя удаленной группой является та, в которой находится процесс-отправитель. В качестве ранга *целевого процесса* (т. е. процесса, относящегося к удаленной группе) указывается ранг процесса в удаленной группе.

Стандартная функция MPI_Comm_size для определения количества процессов, входящих в коммуникатор, может использоваться и для интеркоммуникатора; в этом случае она возвращает размер локальной группы, т. е. той группы интеркоммуникатора, к которой относится процесс, вызвавший эту функцию. Функция MPI_Comm_rank для интеркоммуникатора возвращает ранг процесса в локальной группе. Имеется также дополнительная функция MPI_Comm_remote_size, доступная только для интеркоммуникаторов; она возвращает размер удаленной группы, т. е. той группы интеркоммуникатора, к которой *не относится* процесс, вызвавший эту функцию.

Для знакомства с основным способом создания интеркоммуникатора и простейшими приемами организации взаимодействия между его группами рассмотрим следующее задание.

MPI8Inter9. Количество процессов K — четное число. В каждом процессе дано целое число C, лежащее в диапазоне от 0 до 2, причем известно, что в процессе ранга 0 дано число $C = 1$, а первое из значений $C = 2$ имеется у процесса ранга $K/2$. Используя функцию MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы со значениями $C = 1$ в том же порядке, а второй — процессы со значениями $C = 2$ в том же порядке. Вывести ранги процессов R в созданных коммуникаторах (если процесс не входит ни в один из созданных коммуникаторов, то вывести для него число −1). Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. После этого ввести в процессах первой группы созданного интеркоммуникатора (соответствующей значениям $C = 1$) по одному целому числу X, а в процессах второй группы — по одному целому числу Y. Используя требуемое количество вызовов функций MPI_Send и MPI_Recv для всех процессов созданного интеркоммуникатора, переслать все числа X каждому из процессов второй группы этого коммуникатора, а все числа Y — каждому из процессов первой группы и вывести полученные числа в порядке возрастания рангов переславших их процессов.

При ознакомительном запуске программы с заготовкой для данного задания на экране появится окно, подобное приведенному на рис. 29 (в этом окне скрыт раздел с формулировкой задания).

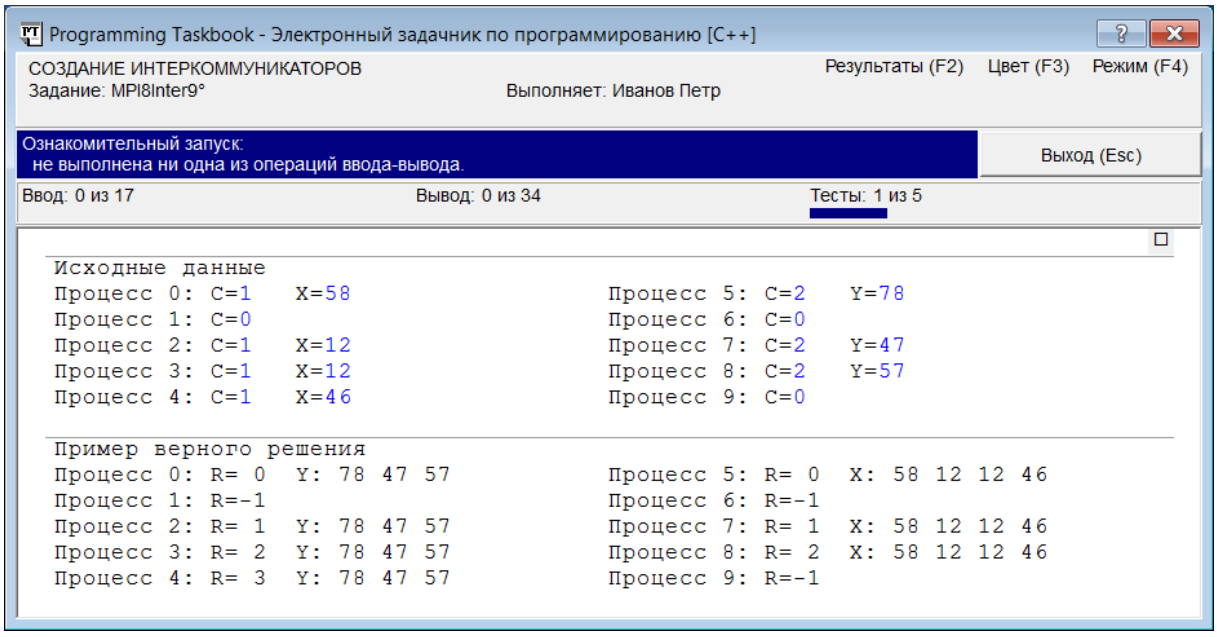


Рис. 299. Ознакомительный запуск задания MPI8Inter9

На первом этапе решения задачи надо создать два новых коммуникатора, содержащих процессы с одинаковыми ненулевыми значениями C. Признаком успешного завершения этого этапа будет вывод правильных значений R рангов процессов в новых коммуникаторах.

Поскольку на данном этапе не требуется использовать новые средства библиотеки MPI, сразу приведем соответствующий фрагмент программы (особенности применения функции MPI_Comm_split ранее подробно обсуждались в п. 11.3, посвященном созданию новых коммуникаторов):

```
int c;
pt >> c;
if (c == 0)
  c = MPI_UNDEFINED;
MPI_Comm local;
MPI_Comm_split(MPI_COMM_WORLD, c, rank, &local);
if (local == MPI_COMM_NULL)
{
  pt << -1;
  return;
}
int local_rank;
MPI_Comm_rank(local, &local_rank);
pt << local_rank;
```

После ввода C мы сразу корректируем ее значение, если оно равно 0, заменяя его на MPI_UNDEFINED, чтобы не создавать коммуникатор для процессов с C = 0. Новый коммуникатор, содержащий текущий процесс, связывается с переменной local, чтобы подчеркнуть, что в дальнейшем именно группа этого коммуникатора станет локальной группой интеркоммуникатора. Если с процессом не связывается новый коммуникатор, то в нем выводится значение –1 и выполняется выход из программы; в противном случае определяется и выводится ранг local_rank процесса в новом коммуникаторе.

При запуске программы будет выведено сообщение о том, что в некоторых процессах введены не все результирующие данные (поскольку в нашей программе еще не организован ввод чисел X), однако значения R для всех процессов будут найдены правильно (рис. 30).

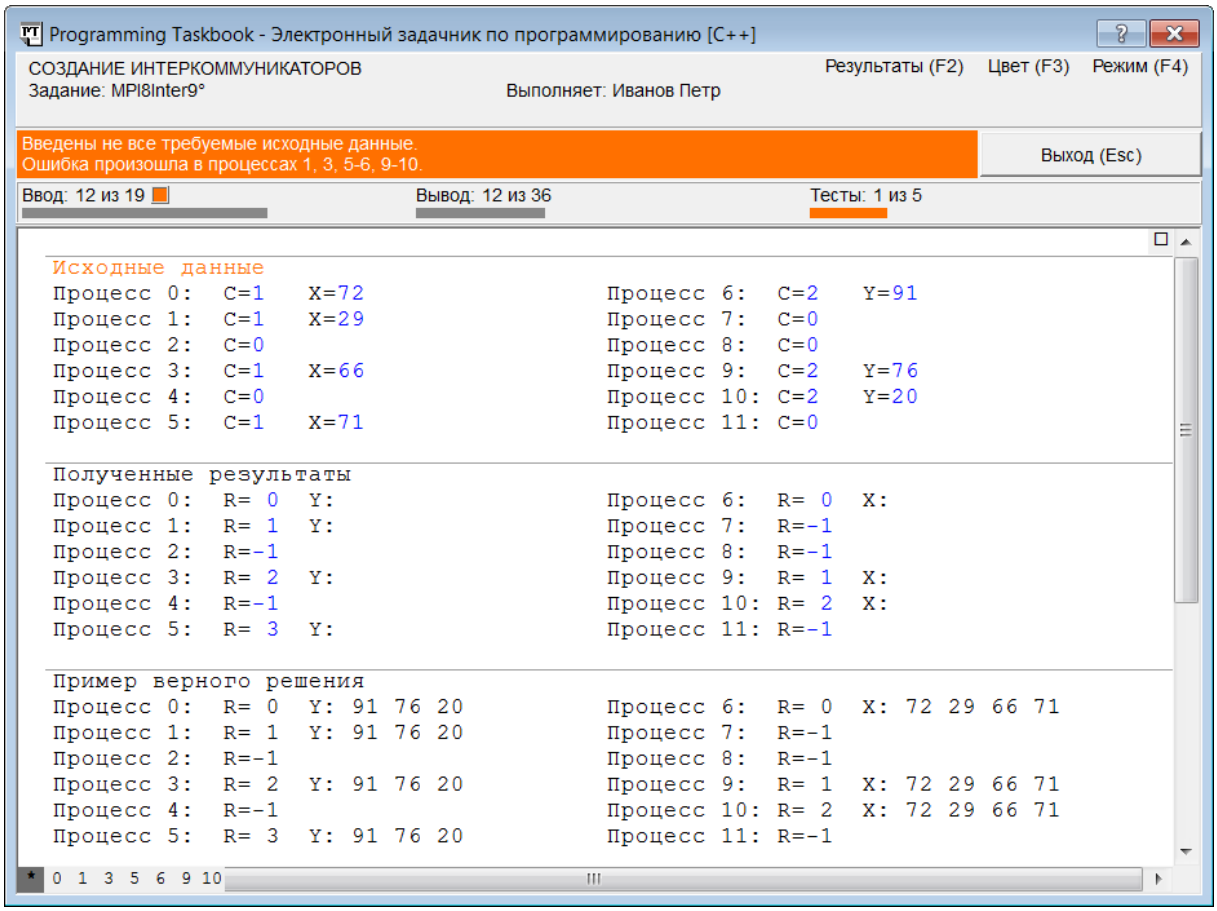


Рис. 30. Вид окна задачника после первого этапа выполнения задания MPI8Inter9

Приступим к основному этапу решения: созданию интеркоммуникатора, содержащего обе созданные ранее группы процессов. Основным средством для создания интеркоммуникаторов является функция MPI_Intercomm_create. Это коллективная функция, которая должна вызываться во всех процессах, которые требуется включить в создаваемый интеркоммуникатор. Иными словами, ее надо вызвать во всех процессах тех двух «обычных» коммуникаторов, которые содержат группы процессов, включаемые в интеркоммуникатор.

Основной проблемой при создании интеркоммуникатора является определение удаленной группы. В качестве локальной группы достаточно указать соответствующий коммуникатор, в который входит процесс, вызывающий функцию MPI_Intercomm_create, однако коммуникатор, созданный для процессов другой группы, в данном процессе недоступен. Проблема решается с помощью задания процессов-«представителей» (leaders) каждой из двух групп. При этом необходимо, чтобы оба выбранных представителя входили в какой-либо общий коммуникатор-посредник (peer). Естественным кандидатом на роль посредника является универсальный коммуникатор MPI_COMM_WORLD, однако для того чтобы избежать возможных конфликтов при пересылке данных, следует использовать ко-

пию коммуникатора MPI_COMM_WORLD, создав ее с помощью функции MPI_Comm_dup. Перечислим параметры функции MPI_Intercomm_create (все они, кроме последнего, являются входными):

- local — коммуникатор (типа MPI_Comm), связанный с локальной группой создаваемого интеркоммуникатора;
- local_leader — целочисленный ранг представителя локальной группы (указывается ранг этого процесса в коммуникаторе local);
- peer_comm — коммуникатор-посредник (типа MPI_Comm); этот параметр учитывается только в том процессе, который является представителем локальной группы;
- remote_leader — целочисленный ранг представителя удаленной группы (указывается ранг этого процесса в коммуникаторе peer_comm); этот параметр, как и предыдущий, учитывается только в том процессе, который является представителем локальной группы;
- tag — целочисленная «метка безопасности», которая должна быть одинаковой во всех процессах, вызвавших функцию MPI_Intercomm_create для создания данного интеркоммуникатора (при создании других интеркоммуникаторов следует использовать другие значения для данной метки);
- intercomm — указатель на созданный интеркоммуникатор (типа MPI_Comm*).

Таким образом, для создания интеркоммуникатора каждый представитель группы должен знать ранг представителя другой (удаленной) группы в коммуникаторе-посреднике. В нашем случае мы можем воспользоваться следующей частью условия задачи: «В процессе ранга 0 дано число $C = 1$, а первое из значений $C = 2$ имеется у процесса ранга $K/2$» (K обозначает общее количество процессов). Это значит, что в первую из созданных нами групп (для процессов с C = 1) гарантированно входит процесс ранга 0 коммуникатора MPI_COMM_WORLD, причем это процесс является первым процессом созданной группы (т. е. имеет ранг 0 в соответствующем коммуникаторе), а во вторую из созданных групп (для процессов с C = 2) гарантированно входит процесс ранга $K/2$ коммуникатора MPI_COMM_WORLD, причем это процесс также является первым процессом созданной группы.

Следовательно, при вызове функции MPI_Intercomm_create во всех процессах нам достаточно положить параметр local_leader равным 0. Что касается параметра remote_leader, то его значение можно определить по рангу вызывающего процесса в коммуникаторе MPI_COMM_WORLD: если этот ранг равен 0 (это означает, что процесс является представителем первой группы), то параметр remote_leader надо положить равным $K/2$, а если ранг процесса в коммуникаторе MPI_COMM_WORLD равен $K/2$, то

параметр remote_leader надо положить равным 0 (в остальных процессах значение параметра remote_leader может быть произвольным, например, также равным 0). Можно поступить и по-другому: анализировать не ранг процесса в коммуникаторе MPI_COMM_WORLD, а значение C; в этом случае для процессов с C = 1 параметр remote_leader надо положить равным *K*/2, а для остальных процессов (с C = 2) — равным 0.

Чтобы убедиться в том, что требуемый интеркоммуникатор создан правильно, можно воспользоваться следующей простой проверкой: определить для каждого процесса интеркоммуникатора размер удаленной группы, вызвав функцию MPI_Comm_remote_size (и выведя полученное ею значение в разделе отладки). Заметим, что данный размер впоследствии нам потребуется при организации пересылки и приема данных.

При реализации второго этапа решения задачи легко допустить серьезную ошибку, попытавшись создать копию коммуникатора MPI_COMM_WORLD *после того*, как некоторые процессы выйдут из функции Solve. Это неизбежно приведет к зависанию параллельного приложения. Таким образом, определить вспомогательный коммуникатор-посредник необходимо до того условного оператора, в которым выполняется оператор return (например, в самом начале решения).

Приведем дополненный вариант решения, в котором новые фрагменты выделены полужирным шрифтом:

```
MPI_Comm peer;
MPI_Comm_dup(MPI_COMM_WORLD, &peer);
int c;
MPI_Comm local;
pt >> c;
if (c == 0)
  c = MPI_UNDEFINED;
MPI_Comm_split(MPI_COMM_WORLD, c, rank, &local);
if (local == MPI_COMM_NULL)
{
  pt << -1;
  return;
}
int local_rank;
MPI_Comm_rank(local, &local_rank);
pt << local_rank;
MPI_Comm inter;
int lead = 0;
if (rank == 0)
  lead = size / 2;
```

```
MPI_Intercomm_create(local, 0, peer, lead, 100, &inter);
int remote_size;
MPI_Comm_remote_size(inter, &remote_size);
Show(remote_size);
```

При запуске этого варианта решение сообщение в информационном разделе не изменится, однако в разделе отладки, наряду с сообщениями об ошибках, для каждого процесса с ненулевым значением C будет выведен правильный размер соответствующей удаленной группы в созданном коммуникаторе (рис. 31; в данном случае для процессов с C = 1 размер удаленной группы равен 2, а для процессов с C = 2 размер равен 3).

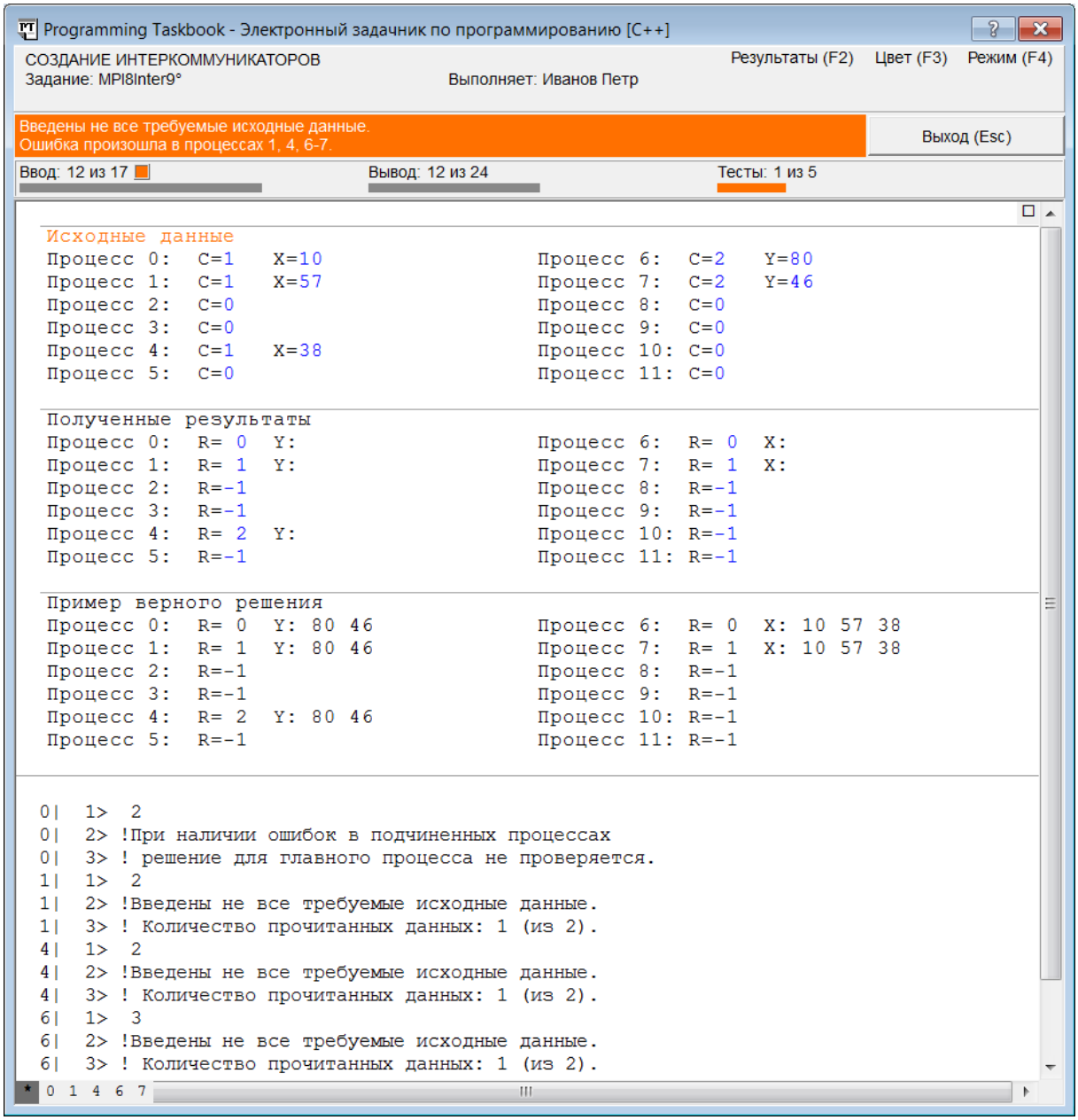


Рис. 31. Вид окна задачника после второго этапа выполнения задания MPI8Inter9

Заключительный этап решения задачи является самым простым, поскольку в нем требуется использовать хорошо знакомые функции MPI_Send и MPI_Recv для обмена сообщениями между двумя процессами. Единственной особенностью является то, что в данном случае эти функции вызываются для интеркоммуникатора, и поэтому в качестве ранга процесса-получателя (в функции MPI_Send) и ранга процесса-отправителя (в функции MPI_Recv) следует указывать ранг процесса в *удаленной* группе:

```
MPI_Status s;
int a, b;
pt >> a;
for (int i = 0; i < remote_size; ++i)
{
  MPI_Send(&a, 1, MPI_INT, i, 0, inter);
  MPI_Recv(&b, 1, MPI_INT, i, 0, inter, &s);
  pt << b;
}
```

Обратите внимание на то, что количество итераций в цикле равно размеру удаленной группы.

После запуска данного варианта программы мы получим сообщение о том, что задание выполнено.

Примечания. 1. В решении использовались только те средства библиотеки MPI, которые уже имеются в стандарте MPI-1, поэтому данная программа будет успешно работать и под управлением системы MPICH версии 1.2.5. Как отмечено в преамбуле к группе заданий MPI8Inter, в версии MPI-1 можно выполнять пять заданий данной группы (с номерами 1–4 и 9).

2. В заданиях MPI8Inter5–MPI8Inter8 рассматриваются новые возможности, связанные с созданием интеркоммуникаторов, появившиеся в стандарте MPI-2. В этом стандарте функции MPI_Comm_create и MPI_Comm_split могут использоваться не только для создания новых интракоммуникаторов на базе существующих, но и для создания новых интеркоммуникаторов (также на базе существующих). Особенно гибкие возможности предоставляет в этом отношении функция MPI_Comm_split, которая позволяет создавать на базе имеющегося интеркоммуникатора целое *семейство* новых интеркоммуникаторов с попарно непересекающимися группами процессов.

Следует, однако, иметь в виду, что в реализации MPICH2 версии 1.3, используемой в задачнике, функция MPI_Comm_split ведет себя некорректно в некоторых ситуациях, связанных с созданием новых интеркоммуникаторов. Эти ситуации подробно описаны в примечании к задаче MPI8Inter7.

3. Еще одним важным нововведением стандарта MPI-2 является возможность использования для интеркоммуникаторов коллективных операций. Этим возможностям посвящены задачи MPI8Inter10–MPI8Inter15. При выполнении коллективных обменов для интеркоммуникаторов используются те же функции, что и для интракоммуникаторов; следует лишь учитывать, что коллективные обмены всегда выполняются между различными группами интеркоммуникатора (иными словами, процессы одной группы обмениваются данными с процессами другой группы).

Особые правила предусмотрены в случае, когда коллективные функции для интеркоммуникаторов используют особый процесс, задаваемый параметром root (таковы, например, функции MPI_Bcast, MPI_Scatter и MPI_Gather). Напомним, что для интракоммуникаторов параметр root во всех процессах должен принимать одно и то же значение, равное рангу особого процесса. Для интеркоммуникаторов ранг особого процесса указывается только в процессах *удаленной* группы, тогда как в процессах локальной группы (т. е. той группы, в которую входит особый процесс) в качестве параметра root указывается одно из двух предопределенных значений: в самом особом процессе параметр root должен иметь значение MPI_ROOT, а в остальных процессах этой же группы — значение MPI_PROC_NULL. Примеры использования подобных коллективных функций будут приведены ниже, при решении задачи MPI8Inter15.

Интеркоммуникаторы используются также при *динамическом порождении новых процессов* в ходе выполнения параллельного приложения. Возможность создания новых процессов появилась в стандарте MPI-2. Подобная возможность позволяет реализовывать параллельные алгоритмы, для которых количество используемых процессов может быть увеличено непосредственно в ходе работы приложения. Кроме того, включение этой возможности в стандарт MPI упрощает переход на технологию MPI тех разработчиков, которые ранее применяли другие параллельные технологии, допускающие порождение процессов (например, PVM — Parallel Virtual Machine).

Для создания новых процессов предусмотрены две функции: MPI_Comm_spawn и MPI_Comm_spawn_multiple (англ. to spawn — *порождать*). Первая из этих функций позволяет создавать требуемое число новых процессов, запуская для каждого из них *один и тот же* исполняемый файл с *одинаковыми* параметрами командной строки (таким образом, она действует аналогично среде MPI, выполняющей начальный запуск параллельного приложения). Вторая функция является более гибкой: она позволяет использовать для различных процессов создаваемой группы *различ-*

ные наборы параметров командной строки и даже *различные* исполняемые файлы. Мы ограничимся рассмотрением первой функции.

Функция MPI_Comm_spawn имеет следующий набор параметров (все параметры, кроме двух последних, являются входными):

- command — строка (типа char*), определяющая запускаемый файл (этот параметр учитывается только в процессе root — см. далее описание параметра root);
- argv — строковый массив (типа char**), содержащий параметры командной строки (этот параметр также учитывается только в процессе root); если параметры не требуются, то в качестве argv достаточно указать нулевой указатель (NULL или nullptr);
- maxproc — максимальное количество запускаемых процессов (целое число; параметр учитывается только в процессе root);
- info — параметр типа MPI_Info, позволяющий задать дополнительную информацию, связанную с запускаемым файлом (параметр учитывается только в процессе root); если дополнительная информация не используется, то в качестве info указывается константа MPI_INFO_NULL;
- root — целочисленный ранг того процесса из родительского коммуникатора comm, в котором обязательно должны быть указаны значения четырех предыдущих параметров;
- comm — родительский коммуникатор (типа MPI_Comm), обеспечивающий создание новой группы процессов;
- intercomm — указатель на результирующий интеркоммуникатор (выходной параметр типа MPI_Comm*), одной из групп которого является группа процессов родительского коммуникатора, а другой — группа созданных процессов;
- array_of_errcodes — целочисленный массив, содержащий коды ошибок, связанных с каждым из создаваемых процессов (если все процессы успешно созданы, то все элементы массива равны MPI_SUCCESS); в случае, если в программе не предполагается использовать данные, возвращаемые этим параметром, вместо него достаточно указать константу MPI_ERRCODES_IGNORE.

Функция MPI_Comm_spawn является коллективной; она должна быть вызвана всеми процессами родительского коммуникатора comm (однако значения параметров, определяющих свойства создаваемых процессов, достаточно указать лишь в одном процессе этого коммуникатора, имеющем ранг root). Успешный выход из этой функции произойдет только в случае, когда будут запущены все требуемые новые процессы и при этом каждый из этих новых процессов вызовет функцию MPI_Init, инициализирующую параллельный режим.

По умолчанию запуск функции MPI_Comm_spawn считается успешным, если запущено требуемое число процессов maxproc. Имеется возможность изменить подобное поведение за счет указания дополнительных настроек в параметре info; в этом случае при успешном завершении функции MPI_Comm_spawn может быть создано меньше процессов, чем указано в параметре maxproc (мы не будем более подробно обсуждать эту возможность).

Связь между исходными (родительскими, parent) и новыми (дочерними, child) процессами устанавливается с помощью нового интеркоммуникатора intercomm, возвращаемого функцией MPI_Comm_spawn. Именно благодаря данному интеркоммуникатора оказывается возможным обмен информацией между родительскими и дочерними процессами.

Впрочем, в результате вызова функции MPI_Comm_spawn данный интеркоммуникатор будет доступен только родительских процессам. Как получить доступ к этому интеркоммуникатору для дочерних процессов? Для этого предназначена специальная функция MPI_Comm_get_parent с единственным выходным параметром parent типа MPI_Comm*, в котором возвращается дескриптор данного интеркоммуникатора, если функцию вызвал один из дочерних процессов, т. е. процессов, созданных уже после запуска параллельного приложения.

Если же функция MPI_Comm_get_parent вызывается одним из исходных процессов параллельного приложения, то она возвращает «пустой» коммуникатор MPI_COMM_NULL. Поскольку обычно один и тот же исполняемый файл используется и для запуска исходных, и для запуска новых (дочерних) процессов, именно функция MPI_Comm_get_parent позволяет «распознать» исходные процессы и, тем самым, обеспечить выполнение различных фрагментов кода для исходных и новых процессов.

Заметим также, что для новых процессов также определен коммуникатор MPI_COMM_WORLD, который включает все процессы, созданные при очередном запуске функции MPI_Comm_spawn. Можно считать, что интеркоммуникатор, возвращаемый функцией MPI_Comm_spawn, объединяет два «обычных» интракоммуникатора MPI_COMM_WORLD, один из которых содержит все родительские процессы, а другой — все дочерние.

Если в родительском коммуникаторе выполнить два вызова функции MPI_Comm_spawn, то в параллельном приложении будут созданы две новые группы процессов, каждая из которых будет связана с родительской группой посредством своего собственного интеркоммуникатора. Возможна и другая схема, при которой дочерняя группа процессов child1 сама вызывает функцию MPI_Comm_spawn; в результате создается новая группа процессов child2, для которой родительской группой будет группа child1.

Итак, для создания новых процессов и обеспечения их последующего взаимодействия с родительскими процессами достаточно использовать две

функции MPI — MPI_Comm_spawn и MPI_Comm_get_parent — и применять те средства пересылки данных, которые предусмотрены для интеркоммуникаторов.

В качестве иллюстрации описанных выше возможностей рассмотрим первую задачу из той подгруппы группы MPI8Inter, которая посвящена созданию новых процессов.

MPI8Inter15. В каждом процессе дано вещественное число. Используя функцию MPI_Comm_spawn с первым параметром «ptprj.exe», создать один новый процесс. С помощью коллективной функции MPI_Reduce переслать в созданный процесс сумму исходных чисел и отобразить ее в разделе отладки, используя в этом процессе функцию Show. Затем с помощью коллективной функции MPI_Bcast переслать найденную сумму в исходные процессы и вывести эту сумму в каждом процессе.

При запуске заготовки для данного задания мы увидим на экране окно, подобное приведенному на рис.

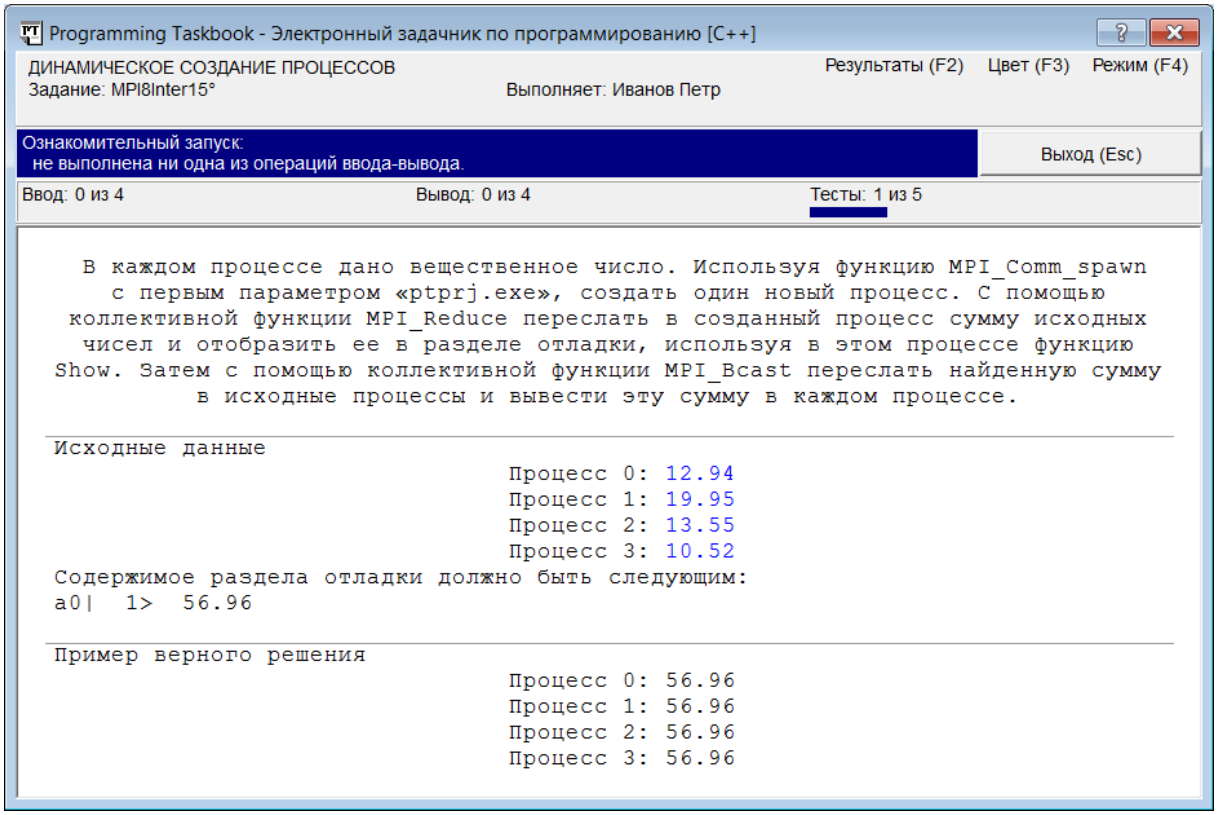


Рис. 32. Ознакомительный запуск задания MPI8Inter15

Следует обратить внимание на образец содержимого раздела отладки, приведенный в разделе «Исходные данные». Указанный в нем ранг процесса содержит, помимо числового значения «0», префикс «a». Этот префикс означает, что данный процесс был создан уже во время выполнения параллельного приложения (если в параллельном приложении создается

несколько групп новых процессов, то в разделе отладки с ними связываются различные буквенные префиксы: «a», «b», «c» и т. д.).

Приступим к выполнению задания. Прежде всего, надо создать новый процесс. Во всех заданиях данной подгруппы в качестве родительского коммуникатора следует использовать стандартный коммуникатор MPI_COMM_WORLD. Кроме того, во всех заданиях надо указывать одно и то же имя исполняемого файла ptprj.exe, поскольку такое имя имеет любой проект, создаваемый программой PT4Load. Количество создаваемых процессов определяется в формулировке задания, а прочие настройки устанавливаются по умолчанию (см. преамбулу к группе MPI8Inter — п.).

При выполнении заданий на создание процессов необходимо учитывать, что код функции Solve будет выполняться не только в исходных процессах параллельного приложения, но и в созданных процессах. При этом, как в случае исходных процессов, не требуется явным образом вызывать функцию MPI_Init, так как этот вызов автоматически выполняет задачник перед вызовом функции Solve с решением задачи.

Для того чтобы вызов функции MPI_Comm_spawn, создающей новый процесс, выполнялся только для исходных процессов приложения, необходимо проанализировать результат, возвращенный функцией MPI_Comm_get_parent. Если она возвращает значение MPI_COMM_NULL, значит, данный процесс входит в число исходных, и для него надо вызвать функцию MPI_Comm_spawn; если же возвращается непустой коммуникатор, то это означает, что процесс является дочерним, и этот коммуникатор можно использовать для связи с родительскими процессами.

Для того чтобы убедиться в том, что новый процесс был действительно создан, выведем в раздел отладки значения size и rank для каждого процесса параллельного приложения. Получаем следующий фрагмент кода:

```
MPI_Comm inter;
MPI_Comm_get_parent(&inter);
if (inter == MPI_COMM_NULL)
{
  MPI_Comm_spawn("ptprj.exe", NULL, 1, MPI_INFO_NULL, 0,
    MPI_COMM_WORLD, &inter, MPI_ERRCODES_IGNORE);
}
Show(size);
Show(rank);
```

При запуске данного варианта программы окно задачника примет вид, подобный приведенному на рис.

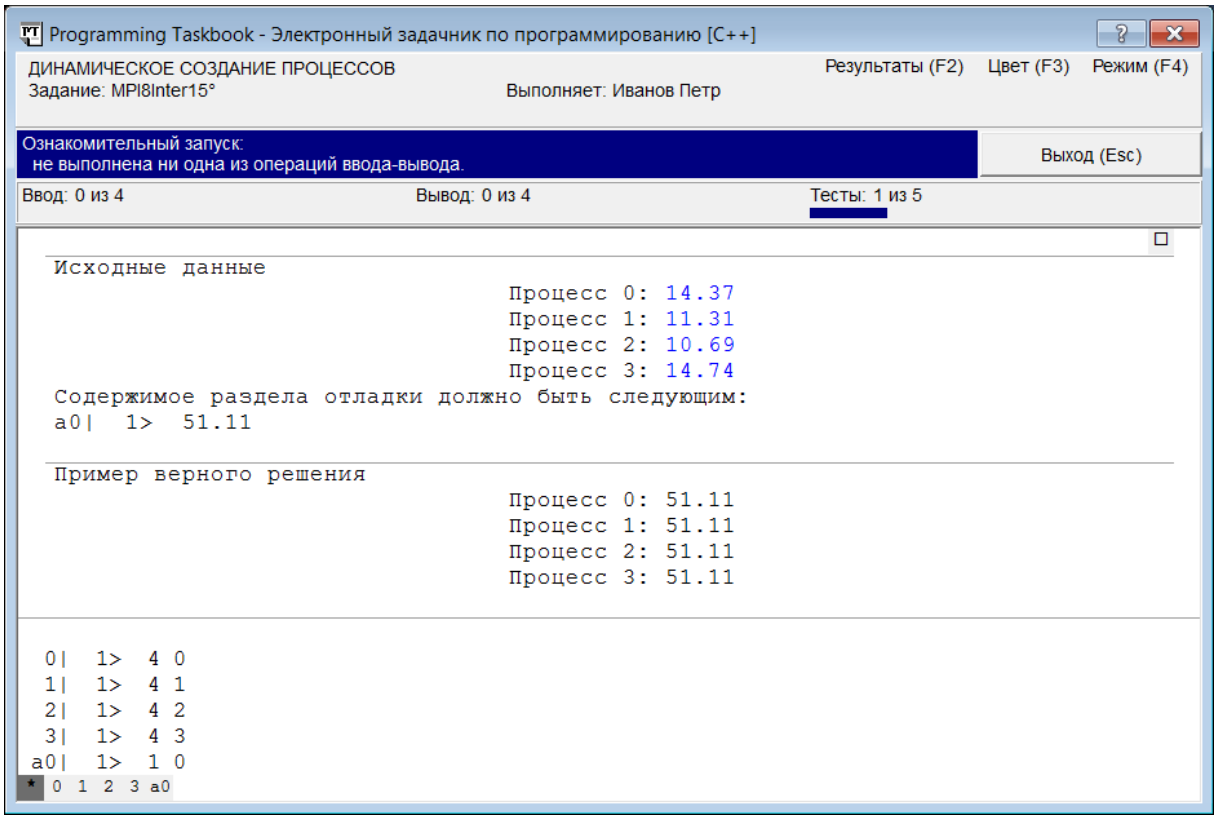


Рис. 33. Вид окна задачника после первого этапа выполнения задания MPI8Inter15

Напомним, что в начале каждой строки в разделе отладки указывается ранг процесса, в котором была выведена соответствующая информация. Наряду с «обычными» рангами 0–4 в разделе присутствует ранг, начинающийся с префикса «a». Как было сказано выше, таким образом помечаются динамически созданные процессы. Итак, в нашем случае в программу входят 4 исходных процесса (для них size = 4, а ранги изменяются от 0 до 3) и один новый процесс. С этим процессом тоже связывается стандартный коммуникатор MPI_COMM_WORLD, однако в него входит единственный процесс (поскольку при вызове функции MPI_Comm_spawn в качестве параметра maxproc мы указали значение 1), поэтому в данной строки выведены числа 1 (количество процессов в коммуникаторе) и 0 (ранг процесса). Итак, наша программа правильно создает новый процесс.

На втором этапе решения получим в новом процессе сумму чисел, данных в исходном процессе. По условию для этого надо использовать коллективную функцию MPI_Reduce. Очевидно, ее необходимо применить к интеркоммуникатору inter, который определен и в исходных, и в новом процессе.

Вспомним об особенности применения для интеркоммуникаторов коллективных операций с выделенным процессом root (см. выше примечание 3): во всех исходных процессах (пересылающих свои данные в другую группу интеркоммуникатора) необходимо указать «настоящий» ранг при-

нимающего процесса root в удаленной группы (в данном случае 0), а в принимающем процессе требуется указать особое значение MPI_ROOT (если бы группа новых процессов содержала более одного элемента, то в остальных элементах в качестве параметра root надо было бы указать значение MPI_PROC_NULL).

Таким образом, до вызова функции MPI_Reduce нам необходимо выполнить ввод данных (в исходных процессах) и правильно указать параметр root. После вызова этой функции выведем полученный результат в разделе отладки, соответствующем новому процессу. Приведем новый вариант решения, выделив в нем новые фрагменты полужирным шрифтом:

```
double a, sum;
int root;
MPI_Comm inter;
MPI_Comm_get_parent(&inter);
if (inter == MPI_COMM_NULL)
{
  MPI_Comm_spawn("ptprj.exe", NULL, 1, MPI_INFO_NULL, 0,
    MPI_COMM_WORLD, &inter, MPI_ERRCODES_IGNORE);
  pt >> a;
  root = 0;
}
else
{
  root = MPI_ROOT;
}
MPI_Reduce(&a, &sum, 1, MPI_DOUBLE, MPI_SUM, root, inter);
if (root == MPI_ROOT)
  Show(sum);
```

Прежние операторы отладочной печати мы заменили единственным оператором Show, выводящим в новом процессе полученную сумму исходных чисел. Заметим, что мы использовали значение root также для того, чтобы «отличить» новый процесс от исходных (разумеется, можно было бы ввести для этой цели и особую переменную).

В результате запуска этого варианта программы окно примет вид, приведенный на рис.

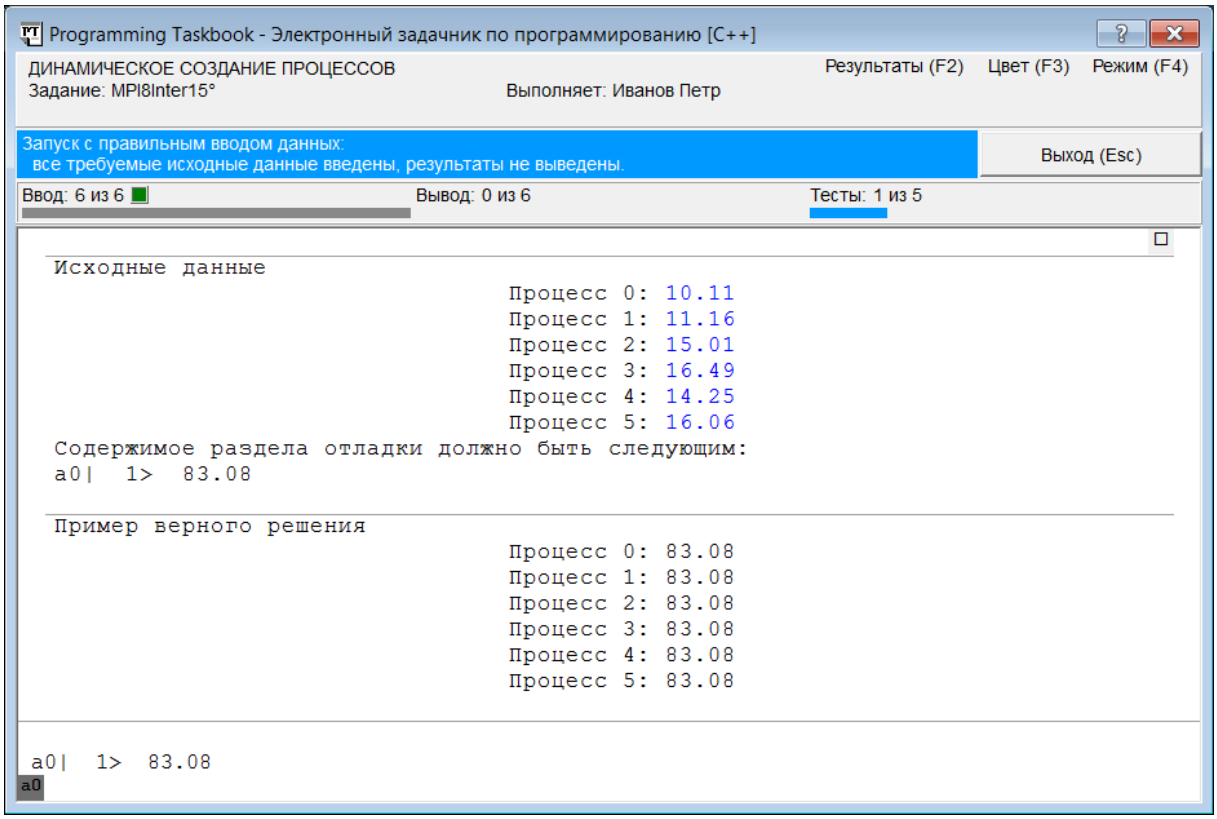


Рис. 34. Вид окна задачника после второго этапа выполнения задания MPI8Inter15

Следует обратить внимание на то, что содержимое раздела отладки в точности соответствует образцу, приведенному в разделе исходных данных. Сообщение в информационном разделе имеет вид «Запуск с правильным вводом данных», поскольку в программе введены все требуемые исходные данные, но никакие данные не выведены ни в одном из исходных процессов.

Нам осталось выполнить последнюю часть задания: переслать найденную сумму из нового процесса во все исходные процессы и вывести полученные данные. Для этого также надо использовать коллективную функцию (в данном случае MPI_Bcast), однако теперь посылающим процессом будет новый (дочерний) процесс, а принимающими — исходные (родительские) процессы. Интересно, что для функции MPI_Bcast требуется задать те же значения параметра root, что и для ранее вызванной функции MPI_Reduce: действительно, в данном случае выделенным процессом (посылающим данные) является новый процесс, поэтому в нем надо использовать параметр root, равный MPI_ROOT; все же процессы в принимающей группе должны указать, что они ожидают данные от процесса 0 посылающей группы.

Таким образом, достаточно после вызова MPI_Reduce добавить вызов новой коллективной функции MPI_Bcast и дополнить последний условный оператор веткой else (выполняемой в исходных процессах), в которой ор-

ганизовать вывод полученной суммы sum. Приведем окончательный вариант решения, выделив полужирным шрифтом новые операторы:

```
double a, sum;
int root;
MPI_Comm inter;
MPI_Comm_get_parent(&inter);
if (inter == MPI_COMM_NULL)
{
  MPI_Comm_spawn("ptprj.exe", NULL, 1, MPI_INFO_NULL, 0,
    MPI_COMM_WORLD, &inter, MPI_ERRCODES_IGNORE);
  pt >> a;
  root = 0;
}
else
  root = MPI_ROOT;
MPI_Reduce(&a, &sum, 1, MPI_DOUBLE, MPI_SUM, root, inter);
MPI_Bcast(&sum, 1, MPI_DOUBLE, root, inter);
if (root == MPI_ROOT)
  Show(sum);
else
  pt << sum;
```

После запуска данного варианта программы мы получим сообщение о том, что задание выполнено.

Примечания. 1. Коммуникатор, который создается при порождении новых процессов, является интеркоммуникатором, одна группа которого включает исходные, а другая — новые процессы. Иногда после создания новых процессов бывает удобно *объединить* исходные и новые процессы в один общий интракоммуникатор. Для этой цели в MPI предусмотрена специальная функция MPI_Intercomm_merge с тремя параметрами: исходным интеркоммуникатором, параметром high, определяющим порядок следования процессов в созданном интракоммуникаторе, и выходным параметром — ссылкой на созданный интракоммуникатор. Функция MPI_Intercomm_merge должна вызываться во всех процессах исходного интеркоммуникатора.

Параметр high является целочисленным *флагом*; все процессы в каждой группе исходного интеркоммуникатора должны указывать *одинаковое* значение параметра high. Если в одной из групп значение параметра high равно 0, а в другой равно 1, то в созданном интракоммуникаторе вначале располагаются процессы первой группы (с high = 0) в порядке их следования в этой группе, а затем — процессы второй группы (с high = 1) также в порядке их следования в этой группе. Если

значение параметра high одинаково во всех процессах интеркоммуникатора, то порядок следования групп является неопределенным, однако и в этом случае порядок следования процессов из каждой группы совпадает с порядком процессов в этой группе.

Использованию функции MPI_Intercomm_merge посвящены задания MPI8Inter19– MPI8Inter20.

2. Возможна ситуация, когда у двух групп процессов отсутствует общий интеркоммуникатор (например, если каждая из этих групп была создана с помощью отдельного вызова функции MPI_Comm_spawn). Для установки связи между такими группами можно использовать специальный механизм *клиент-серверного взаимодействия*, реализованный в MPI-2. Одна из групп, между которыми требуется установить связь, играет роль *сервера*, а другая — роль *клиента*. Процессы группы-сервера создают порт для связи (функцией MPI_Open_port) и публикуют имя этого порта (с помощью функции MPI_Publish_name), после чего начинают прослушивать этот порт в ожидании подключения клиента (функцией MPI_Comm_accept). Процессы группы-клиента получают порт, созданный группой-сервером, используя публичное имя этого порта (с помощью функции MPI_Lookup_name), после чего подключаются к серверу по этому порту (функцией MPI_Comm_connect). При успешном завершении функций MPI_Comm_accept и MPI_Comm_connect они возвращают интеркоммуникатор intercomm, объединяющий группу-клиент и группу-сервер.

Порт (параметр port) представляет собой текстовую строку, генерируемую средой MPI при вызове функции MPI_Open_port; порт достаточно создать в одном из процессов группы-сервера и получить в одном из процессов группы-клиента. Публичное имя порта (параметр public_name), в отличие от параметра port, должно быть известно заранее как процессам группы-сервера, так и процессам группы-клиента; это некоторый «пароль», которым обмениваются эти группы.

В каждой из функций MPI_Comm_accept и MPI_Comm_connect дополнительно указывается параметр root — ранг процесса, в котором известен порт, — и параметр comm — коммуникатор, содержащий все процессы группы-клиента или, соответственно, группы-сервера.

Во всех перечисленных функциях предусматривается дополнительный параметр info, который достаточно положить равным MPI_INFO_NULL.

Приведем стандартную последовательность действий на стороне группы-сервера:

```
char port[MPI_MAX_PORT_NAME];
MPI_Open_port(MPI_INFO_NULL, port);
MPI_Publish_name("password", MPI_INFO_NULL, port);
```

```
MPI_Comm_accept(port, MPI_INFO_NULL, root, comm, &inter);
```
Последовательность действий на стороне группы-клиента выглядит следующим образом:
```
char port[MPI_MAX_PORT_NAME];
MPI_Lookup_name("password", MPI_INFO_NULL, port);
MPI_Comm_connect(port, MPI_INFO_NULL, root, comm, &inter);
```
Важно согласовать вызовы функций таким образом, чтобы функция MPI_Publish_name в процессе root группы-сервера была вызвана по времени раньше, чем функция MPI_Lookup_name в процессе root группы-клиента.

Механизму клиент-серверного взаимодействия посвящены задания MPI8Inter21–MPI8Inter22. В примечаниях к этим заданиям описаны варианты согласования порядка вызова функций MPI_Publish_name и MPI_Lookup_name, основанные на применении функции MPI_Barrier.

2.8. Параллельные матричные алгоритмы: MPI9Matr1, MPI9Matr2, MPI9Matr24, MPI9Matr19

В завершающей группе учебных заданий MPI9Matr, в отличие от предыдущих групп, изучаются не тот или иной раздел библиотеки MPI, а *параллельные алгоритмы*, в которых применяются разнообразные средства этой библиотеки, в том числе различные варианты взаимодействия процессов, новые производные типы, коммуникаторы с виртуальными топологиями, параллельный файловый ввод-вывод. Таким образом, группу MPI9Matr можно рассматривать как итоговую группу, позволяющую повторить и закрепить большинство из ранее изученных тем, связанных с технологией MPI.

Группа MPI9Matr посвящена параллельным *матричным алгоритмам* — одному из наиболее известных видов параллельных алгоритмов, реализуемых с применением технологии MPI. Следует заметить, что библиотека MPI содержит ряд средств, специально предназначенных для работы с матрицами; к таким средствам можно отнести производные типы, связанные с различными фрагментами матриц (столбцами, наборами столбцов или блоками), а также дополнительные возможности коммуникаторов с декартовой топологией.

Характерным представителем матричных алгоритмов является алгоритм *матричного умножения*. В группе MPI9Matr рассмотрены два основных вида параллельных распределенных алгоритмов матричного умножения: *ленточные алгоритмы*, в которых распределение вычислений между процессами достигается за счет разбиения матриц на *полосы*, включающие наборы смежных строк или столбцов, и *блочные алгоритмы*, в которых применяется разбиение матриц на прямоугольные *блоки*. Кроме того, в

группу включено вводное задание MPI9Matr1, в котором описывается формат хранения матричных данных, приводятся необходимые формулы и требуется реализовать простейший непараллельный алгоритм матричного умножения.

Для каждого из видов параллельных алгоритмов матричного умножения рассматриваются два варианта, отличающиеся деталями реализации.

В *ленточном алгоритме 1* (задания MPI9Matr2– MPI9Matr10) используются только горизонтальные полосы (наборы смежных строк), для пересылки которых не требуется вводить дополнительные типы данных. В *ленточном алгоритме 2* (задания MPI9Matr11– MPI9Matr20) применяются как горизонтальные, так и вертикальные полосы, что, с одной стороны, несколько упрощает реализацию самого алгоритма умножения (поскольку он основан на умножении *строк* одной матрицы на *столбцы* другой), а с другой стороны, требует применения новых типов, обеспечивающих более эффективную пересылку вертикальных полос (т. е. наборов смежных столбцов).

В первом варианте блочного алгоритма (*алгоритм Кэннона*, задания MPI9Matr21–MPI9Matr31) перед итерационным вычислением фрагментов итогового матричного произведения выполняется этап начального перераспределения блоков между процессами, благодаря которому упрощаются последующие действия по пересылке данных; при этом для упрощения действий, связанных с пересылкой блоков (как на начальном этапе, так и на этапе вычисления произведения), используется вспомогательный коммуникатор, снабженный топологией квадратной матрицы процессов. Во втором варианте блочного алгоритма (*алгоритм Фокса*, задания MPI9Matr32– MPI9Matr44) особый этап начального перераспределения отсутствует, однако каждый шаг вычисления итогового матричного произведения требует более сложных действий по пересылке блоков; при этой пересылке предлагается использовать не только коммуникатор с топологией квадратной матрицы процессов, но и порожденные на его основе коммуникаторы, связанные с отдельными строками и столбцами этой матрицы. Дополнительно в каждом из вариантов блочного алгоритма требуется определить новый производный тип, упрощающий пересылку блоков матриц; в алгоритме Кэннона для пересылки блоков предлагается использовать функции MPI_Send и MPI_Recv, а в алгоритме Фокса — коллективную функцию MPI_Alltoallw (при условии использования библиотеки MPI-2).

В каждом из рассмотренных алгоритмов матричного умножения можно выделить три основных этапа:

- *этап рассылки исходных данных*: начальная рассылка фрагментов исходных матриц во все процессы;

- *этап вычислений*: последовательное вычисление фрагментов итогового матричного произведения, каждый шаг которого сопровождается пересылкой фрагментов исходных матриц между процессами (для алгоритма Кэннона этапу вычислений предшествует *этап инициализации*, связанный в начальным перераспределением блоков между процессами);
- *этап сбора результатов*: пересылка вычисленных фрагментов матричного произведения в главный процесс с целью получения итоговой матрицы.

С каждым из этих этапов связывается отдельное задание (или серия заданий); при этом в качестве исходных данных в каждом задании предлагается набор данных, которые должны быть сформированы в результате выполнения предыдущего этапа. Это упрощает разработку и тестирование каждого этапа, позволяет разрабатывать различные этапы алгоритма в произвольном порядке, а также дает возможность реализовывать только отдельные этапы алгоритма без необходимости предварительной разработки всех предыдущих этапов).

Серии заданий связываются с этапом вычислений, который является наиболее сложным; при этом в начальном задании каждой серии требуется разработать простейший вариант вычисления, используемый на первом шаге алгоритма, а в последующих заданиях данный вариант модифицируется для возможности применения на каждом шаге этапа вычислений (исключение составляет алгоритм Кэннона, для которого действия на каждом шаге не зависят от номера шага — см. примечание к заданию MPI9Matr25).

Предусмотрены также задания, в которых начальный и завершающий этапы каждого алгоритма требуется модифицировать таким образом, чтобы в них использовался параллельный файловый ввод-вывод:

- *этап чтения файловых данных*: каждый процесс получает фрагменты исходных матриц непосредственно из файлов, содержащих эти матрицы;
- *этап записи в итоговый файл*: каждый процесс записывает полученные фрагменты итогового произведения в соответствующую часть результирующего файла.

Кроме того, для тех алгоритмов, в которых требуется использовать новые типы данных MPI или коммуникаторы с декартовой топологией, предусмотрены дополнительные задания, связанные с созданием соответствующих объектов (типа MPI_Datatype или MPI_Comm).

Во всех заданиях, посвященных реализации отдельных этапов матричных алгоритмов, а также созданию вспомогательных объектов (новых производных типов или коммуникаторов), требуется оформить соответствующие действия в виде *вспомогательной функции*. Функции, создающие новые объекты, используются в дальнейшем при реализации различных

этапов алгоритма, а функции, связанные с самими этапами, применяются в итоговых заданиях, требующих реализации соответствующего матричного алгоритма в полном объеме.

В приводимых ниже таблицах 1 и 2 указываются номера заданий, связанных с реализацией различных этапов каждого алгоритма, и, кроме того, приводятся имена функций, которые требуется разработать в этих заданиях. Таблица 1 содержит данные ленточных алгоритмов (задания MPI9Matr2–MPI9Matr20), а таблица 2 — для блочных (задания MPI9Matr21–MPI9Matr44).

Таблица 1

Задания группы MPI9Matr, связанные с ленточными алгоритмами

Этап алгоритма	**Ленточный алгоритм 1 (горизонтальные полосы)**	**Ленточный алгоритм 2 (горизонтальные и вертикальные полосы)**
Определение нового производного типа данных	*Отсутствует*	MPI9Matr11 Matr2CreateTypeBand (p, k, q, t)
Этап рассылки исходных данных	MPI9Matr2 Matr1ScatterData()	MPI9Matr12 Matr2ScatterData()
Этап вычислений	MPI9Matr3 Matr1Calc() MPI9Matr4–MPI9Matr5 Matr1Calc(l)	MPI9Matr13 Matr2Calc() MPI9Matr14–MPI9Matr15 Matr2Calc(l)
Этап сбора результатов	MPI9Matr6 Matr1GatherData()	MPI9Matr16 Matr2GatherData()
Полная реализация алгоритма	MPI9Matr7	MPI9Matr17
Этап чтения файловых данных	MPI9Matr8 Matr1ScatterFile()	MPI9Matr18 Matr2ScatterFile()
Этап записи в итоговый файл	MPI9Matr9 Matr1GatherFile()	MPI9Matr19 Matr2GatherFile()
Полная реализация алгоритма с применением файлового ввода-вывода	MPI9Matr10	MPI9Matr20

Таблица 2

Задания группы MPI9Matr, связанные с блочными алгоритмами

Этап алгоритма	**Блочный алгоритм Кэннона**	**Блочный алгоритм Фокса**
Определение нового производ-	MPI9Matr21 Matr3CreateTypeBlock	MPI9Matr32 Matr4CreateTypeBlock

ного типа данных	(m0, p0, p, t)	(m0, p0, p, t)
Определение новых коммуникаторов с декартовой топологией	MPI9Matr22 Matr3CreateCommGrid (comm)	MPI9Matr33 Matr4CreateCommGrid (comm), Matr4CreateCommRow (grid, row) MPI9Matr34 Matr4CreateCommCol (grid, col)
Этап рассылки исходных данных	MPI9Matr23 Matr3ScatterData()	MPI9Matr35 Matr4ScatterData()
Этап инициализации	MPI9Matr24 Matr3Init()	*Отсутствует*
Этап вычислений	MPI9Matr25–MPI9Matr26 Matr3Calc()	MPI9Matr36 Matr4Calc1() MPI9Matr37 Matr4Calc2() MPI9Matr38–MPI9Matr39 Matr4Calc1(l), Matr4Calc2()
Этап сбора результатов	MPI9Matr27 Matr3GatherData()	MPI9Matr40 Matr4GatherData()
Полная реализация алгоритма	MPI9Matr28	MPI9Matr41
Этап чтения файловых данных	MPI9Matr29 Matr3ScatterFile()	MPI9Matr42 Matr4ScatterFile()
Этап записи в итоговый файл	MPI9Matr30 Matr3GatherFile()	MPI9Matr43 Matr4GatherFile()
Полная реализация алгоритма с применением файлового ввода-вывода	MPI9Matr31	MPI9Matr44

Еще одной особенностью заданий группы MPI9Matr является создание *специализированных проектов-заготовок*, в которых уже содержатся описания глобальных переменных для хранения различных объектов, используемых при выполнении задания (эта особенность отмечена в преамбуле к данной группе заданий — см. п.).

Начнем с рассмотрения вводного задания группы MPI9Matr, которое предназначено для ознакомления с приемами работы с матрицами, необходимыми при выполнении любых заданий данной группы.

MPI9Matr1. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно. Найти и вывести в главном процессе матрицу C размера $M \times Q$, являющуюся произведением матриц A и B.

Формула для вычисления элементов матрицы C в предположении, что строки и столбцы всех матриц нумеруются от 0, имеет вид: $C_{I,J} = A_{I,0} \cdot B_{0,J} + A_{I,1} \cdot B_{1,J} + \ldots + A_{I,P-1} \cdot B_{P-1,J}$, где $I = 0, \ldots, M - 1$, $J = 0, \ldots, Q - 1$.

Для хранения матриц A, B, C использовать одномерные массивы размера $M \cdot P$, $P \cdot Q$ и $M \cdot Q$, размещая в них элементы матриц по строкам (при этом элемент матрицы с индексами I и J будет храниться в элементе соответствующего массива с индексом $I \cdot N + J$, где N — количество столбцов матрицы). При выполнении данного задания подчиненные процессы не используются.

Файл MPI9Matr1.cpp, созданный в качестве заготовки для выполнения данного задания, содержит следующий код:

```
#include "pt4.h"
#include "mpi.h"
#include <cmath>

int k;              // количество процессов
int r;              // ранг текущего процесса

int m, p, q;        // размеры исходных матриц

int *a_, *b_, *c_;
  // массивы для хранения исходных матриц в главном процессе

void Solve()
{
  Task("MPI9Matr1");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  k = size;
  r = rank;

}
```

Перечислим особенности созданной заготовки. Во-первых, в ней подключается стандартный заголовок cmath, поскольку во многих заданиях группы MPI9Matr требуется использовать функцию округления ceil, объ-

явленную в этом заголовке. Во-вторых, файл содержит описания набора глобальных переменных, связанных с выполняемым заданием (назначение переменных приводится в комментариях, расположенных рядом с описаниями). В частности, вводятся более краткие имена для количества процессов и ранга текущего процесса: k и r соответственно. Переменные k и r, в отличие от переменных size и rank, добавляемых в заготовку любой программы, связанной с задачником PT for MPI-2, можно использовать не только в функции Solve, но и во вспомогательных функциях, которые требуется разработать при выполнении большинства заданий группы MPI9Matr.

Следует обратить внимание на то, что имена переменных-указателей a_, b_, c_, которые должны связываться с массивами, содержащими исходные матрицы *A* и *B*, а также результат их умножения *C* = *AB*, снабжены символом подчеркивания. Это связано с тем, что более краткие имена переменных a, b, c во всех заданиях (кроме вводного задания MPI9Matr1) связываются с *фрагментами матриц* (полосами или блоками), которые обрабатываются в каждом процессе. Переменные a_, b_, c_, в отличие от переменных a, b, c, должны использоваться только в главном процессе и требуются лишь в тех заданиях, в которых выполняется пересылка фрагментов исходных матриц во все процессы, а также пересылка фрагментов полученного матричного произведения в главный процесс.

При запуске данной заготовки на экране появится окно задачника, вид которого приведен на рис. (для уменьшения размеров окна в этом и последующих рисунках в данном пункте скрыт раздел с формулировкой задания).

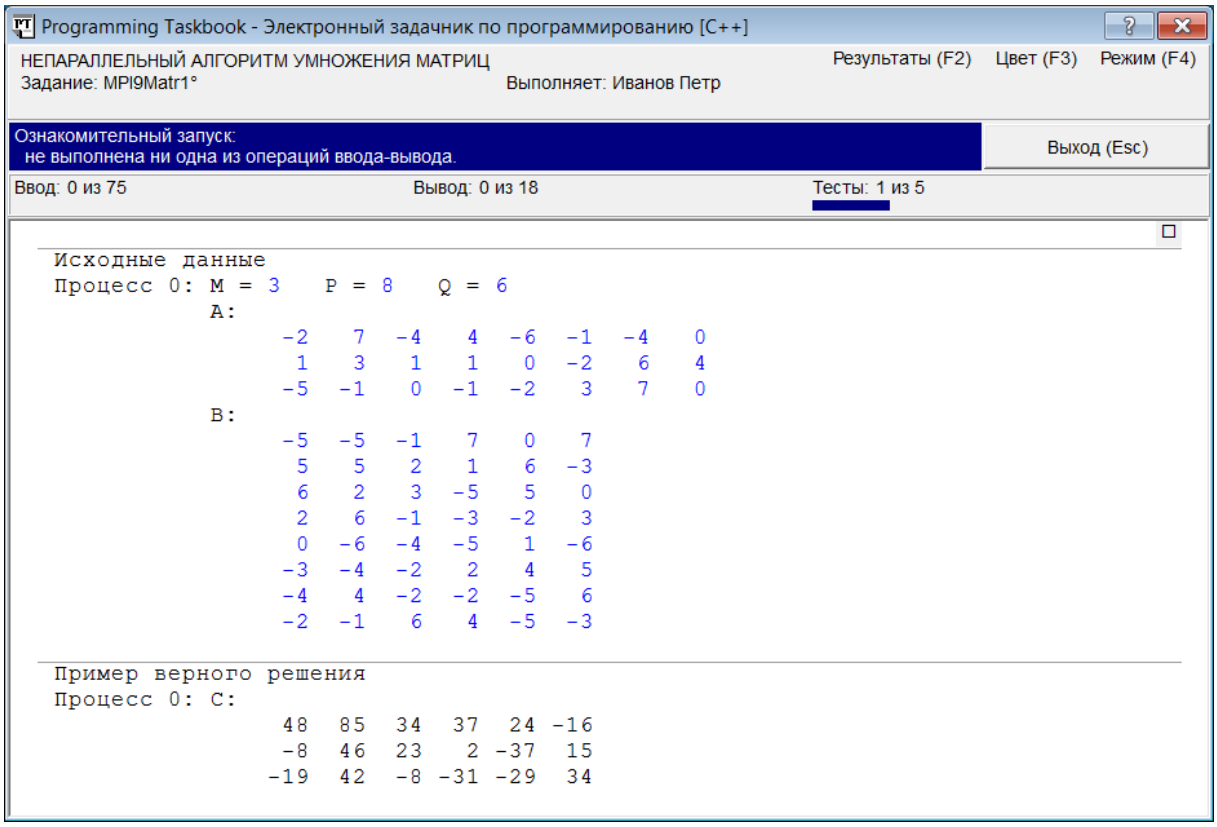


Рис. 35. Ознакомительный запуск задания MPI9Matr1

Хотя созданный проект, как и все остальные проекты для заданий, входящих в задачник PT for MPI-2, запускается в виде параллельного приложения, в нем не требуется использовать подчиненные процессы: ввод данных, их обработку и вывод результатов необходимо выполнять только в главном процессе.

Во всех заданиях группы MPI9Matr матрицы должны храниться с одномерных динамических массивах (по строкам). Двумерные массивы в данном случае использовать не следует, так как это существенно затруднит пересылку фрагментов матриц между различными процессами.

На первом этапе решения организуем ввод исходных данных. Для этого дополним функцию Solve следующими операторами:

```
if (r != 0)
  return;
pt >> m >> p >> q;
a_ = new int[m*p];
b_ = new int[p*q];
for (int i = 0; i < m*p; ++i)
  pt >> a_[i];
for (int i = 0; i < p*q; ++i)
  pt >> b_[i];
```

В начале данного фрагмента мы проверяем, в каком процессе находимся, и немедленно выходим из функции Solve в случае, если процесс не является главным процессом (т. е. процессом ранга 0). Затем вводятся размеры матриц и выделяется память для исходных массивов a_ и b_, после чего в эти массивы считываются элементы матриц *A* и *B*. Так как задачник всегда передает элементы матрицы по строкам, для ввода этих элементов в каждый из одномерных массивов достаточно использовать *единственный* цикл.

При запуске данного варианта программы в окне задачника появится сообщение о том, что все исходные данные успешно введены.

На втором этапе решения выделим память для массива c_, предназначенного для хранения результирующего матричного произведения *C*, обнулим его элементы и выполним перемножение матриц, используя формулу, приведенную в формулировке задания (элемент матрицы *C* с индексами *I* и *J* получается в результате попарного перемножения элементов *I*-й строки матрицы *A* и *J*-го столбца матрицы *B* и суммирования полученных произведений):

$C_{I,J} = A_{I,0} \cdot B_{0,J} + A_{I,1} \cdot B_{1,J} + \ldots + A_{I,P-1} \cdot B_{P-1,J}$

Поскольку все матрицы хранятся в одномерных массивах (по строкам), а индексирование как элементов матриц, так и элементов массивов начинается от 0, для доступа к элементу матрицы с индексами *I* и *J* следует обратиться к элементу массива с индексом $I \cdot N + J$, где символом *N* обозначается *количество столбцов* матрицы.

После нахождения матричного произведения надо вывести все его элементы. Таким образом, вторая часть решения будет иметь следующий вид:

```
c_ = new int[m*q];
for (int i = 0; i < m*q; ++i)
  c_[i] = 0;
for (int i = 0; i < m; ++i)
  for (int j = 0; j < q; ++j)
    for (int n = 0; n < p; ++n)
      c_[i*q+j] += a_[i*p+n] * b_[n*q+j];
for (int i = 0; i < m*q; ++i)
  pt << c_[i];
```

Обратите внимание на то, что для вывода полученных результатов (как и для ввода элементов исходных матриц) достаточно использовать единственный цикл; при этом задачник сам обеспечивает наглядное отображение результирующей матрицы в своем окне.

В результате запуска данного варианта программы в окне задачника будет выведено сообщение о том, что задание выполнено.

Итак, рассмотренное задание позволило познакомиться с приемами ввода и вывода матричных данных, а также продемонстрировало стандартный алгоритм матричного умножения.

Теперь обратимся к заданию MPI9Matr2, связанному с реализацией первого этапа алгоритма — вводом исходных данных в главном процессе и пересылкой их в другие процессы параллельного приложения. Это задание является первым в подгруппе, посвященной первому варианту ленточного алгоритма (в котором обе матрицы разбиваются на горизонтальные полосы).

MPI9Matr2. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно. В первом варианте ленточного алгоритма перемножения матриц каждая матрица-сомножитель разбивается на K горизонтальных полос, где K — количество процессов (в дальнейшем полосы распределяются по процессам и используются для вычисления в каждом процессе части итогового матричного произведения).

Полоса для матрицы A содержит N_A строк, полоса для матрицы B содержит N_B строк; числа N_A и N_B вычисляются по формулам $N_A = \text{ceil}(M/K)$, $N_B = \text{ceil}(P/K)$, где операция «/» означает вещественное деление, а функция ceil выполняет округление с избытком. Если матрица содержит недостаточно строк для заполнения последней полосы, то полоса дополняется нулевыми строками.

Сохранить исходные матрицы, дополненные при необходимости нулевыми строками, в одномерных массивах в главном процессе, после чего организовать пересылку полос из этих массивов во все процессы: в процесс ранга R ($R = 0, 1, \ldots, K - 1$) пересылается полоса с индексом R, все полосы A_R имеют размер $N_A \times P$, все полосы B_R имеют размер $N_B \times Q$. Кроме того, создать в каждом процессе полосу C_R для хранения фрагмента матричного произведения $C = AB$, которое будет вычисляться в этом процессе; каждая полоса C_R имеет размер $N_A \times Q$ и заполняется нулевыми элементами.

Полосы, как и исходные матрицы, должны храниться по строкам в одномерных массивах соответствующего размера. Для пересылки размеров матриц использовать коллективную функцию MPI_Bcast, для пересылки полос матриц A и B использовать коллективную функцию MPI_Scatter.

Оформить все описанные действия в виде функции Matr1ScatterData (без параметров), в результате вызова которой каждый процесс получает значения N_A, P, N_B, Q, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C. После вызова функции Matr1ScatterData вывести в каждом процессе полученные данные (числа N_A, P, N_B, Q и полосы матриц A, B, C). Ввод исходных данных осуществ-

лять в функции Matr1ScatterData, вывод результатов выполнять во внешней функции Solve.

Указание. Для уменьшения числа вызовов функции MPI_Bcast все пересылаемые размеры матриц можно поместить во вспомогательный массив.

Заготовка для этого задания имеет следующий вид:

```
#include "pt4.h"
#include "mpi.h"
#include <cmath>

int k;              // количество процессов
int r;              // ранг текущего процесса

int m, p, q;        // размеры исходных матриц
int na, nb;         // размеры полос матриц

int *a_, *b_, *c_;
  // массивы для хранения исходных матриц в главном процессе
int *a, *b, *c;
  // массивы для хранения полос матриц в каждом процессе

void Solve()
{
  Task("MPI9Matr2");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  k = size;
  r = rank;

}
```

В данном случае в число глобальных переменных дополнительно входят массивы a, b, c для хранения полос матриц в каждом процессе, а также переменные na и nb, определяющие размеры этих полос.

При запуске созданной заготовки на экране появится окно задачника (рис. 36 и 37). На первом рисунке приведен раздел с исходными данными, на втором — раздел с примером правильных результатов.

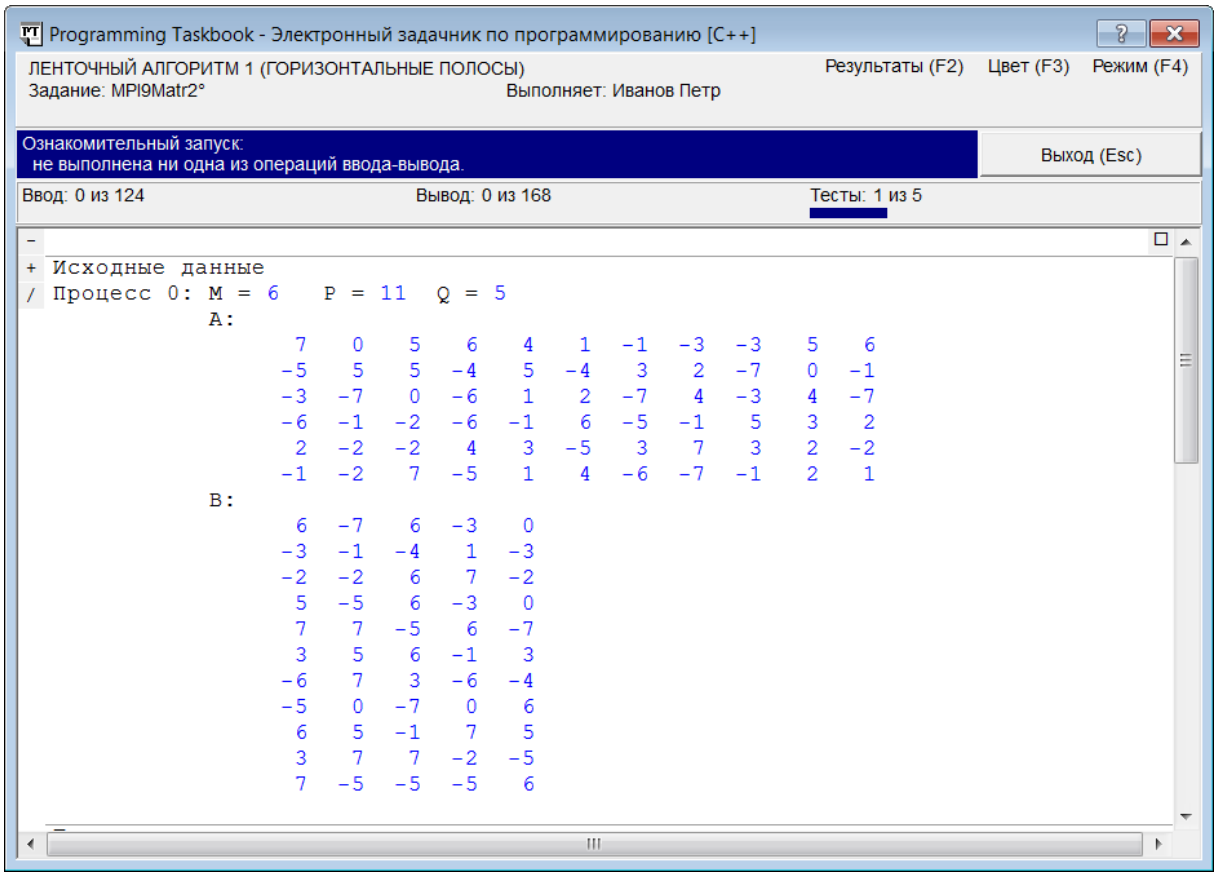


Рис. 36. Ознакомительный запуск задания MPI9Matr2 (раздел с исходными данными)

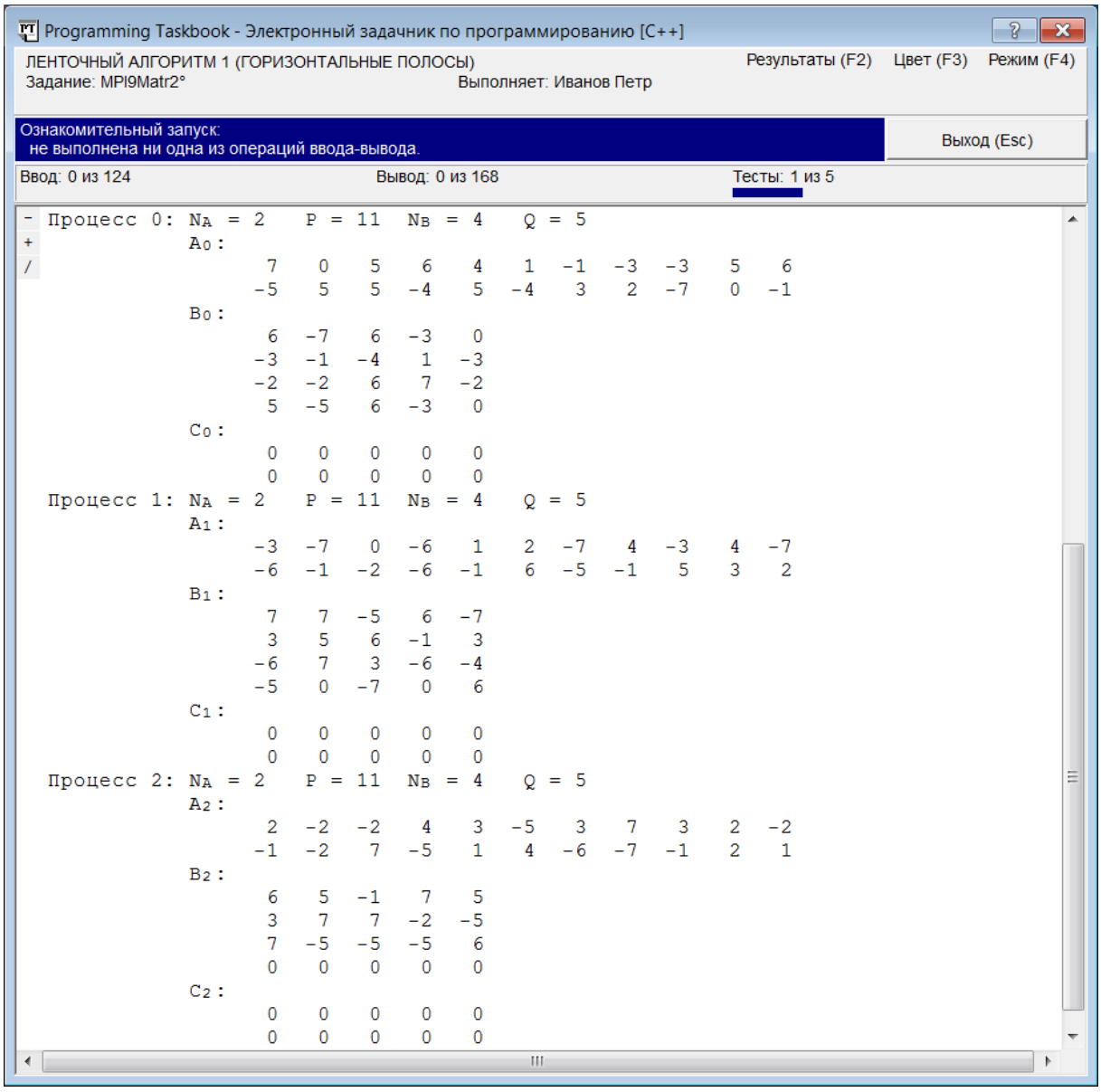


Рис. 37. Ознакомительный запуск задания MPI9Matr2 (пример верного решения)

Рисунки показывают, что в заданиях группы MPI9Matr может использоваться большое число как исходных данных, так и результатов (для варианта, приведенного на рисунках, количество исходных чисел равно 124, а количество результирующих данных равно 168). Однако, несмотря на большое количество данных, все они отображаются в окне задачника в наглядном виде благодаря специальному форматированию и комментариям.

Все исходные данные требуется ввести в главном процессе. В качестве результатов для каждого процесса надо вывести размеры полос (эти значения являются одинаковыми для всех процессов) и сами полосы; при этом полосы для итогового произведения *C* должны быть нулевыми. В некоторых процессах могут оказаться нулевыми и завершающие строки полос, связанных с исходными матрицами *A* и *B*. В нашем случае нулевую

строку содержит полоса матрицы *B*, связанная с последним процессом (ранга 2).

Согласно условию задачи, все действия, связанные с вводом исходных данных и их рассылкой, необходимо оформить в виде функции Matr1ScatterData без параметров (в этой функции будут использоваться глобальные переменные, уже описанные в заготовке программы). На первом этапе выполним ввод данных в главном процессе:

```
void Matr1ScatterData()
{
  if (r == 0)
    {
      int m;
      pt >> m >> p >> q;
      na = (int)ceil(m / (k*1.0));
      nb = (int)ceil(p / (k*1.0));
      a_ = new int[na*k*p];
      b_ = new int[nb*k*q];
      for (int i = 0; i < m*p; ++i)
        pt >> a_[i];
      for (int i = m*p; i < na*k*p; ++i)
        a_[i] = 0;
      for (int i = 0; i < p*q; ++i)
        pt >> b_[i];
      for (int i = p*q; i < nb*k*q; ++i)
        b_[i] = 0;
    }
}
```

Обратите внимание на то, что в массивы a_ и b_ необходимо занести не только элементы исходных матриц, но и завершающие нулевые строки, которые позволят получить в каждом процессе полосы одинакового размера.

Функцию Solve надо дополнить вызовом функции Matr1ScatterData.

При запуске данного варианта программы будет выведено сообщение о том, что все исходные данные успешно введены.

Теперь дополним функцию Matr1ScatterData завершающим фрагментом, обеспечивающим пересылку данных:

```
MPI_Bcast(&na, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&p, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&nb, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&q, 1, MPI_INT, 0, MPI_COMM_WORLD);
a = new int[na*p];
```

```
b = new int[nb*q];
c = new int[na*q];
MPI_Scatter(a_, p*na, MPI_INT, a, p*na, MPI_INT, 0,
  MPI_COMM_WORLD);
MPI_Scatter(b_, q*nb, MPI_INT, b, q*nb, MPI_INT, 0,
  MPI_COMM_WORLD);
for (int i = 0; i < na*q; ++i)
  c[i] = 0;
```

В данном фрагменте мы не воспользовались указанием к заданию и не стали помещать пересылаемые размеры во вспомогательный массив (читателю рекомендуется самостоятельно выполнить подобную модификацию алгоритма). Кроме пересылки полос матриц *A* и *B* мы создаем в каждом процессе полосу для итогового произведения *C* и обнуляем ее элементы.

Результат запуска этого варианта не будет отличаться от предыдущего.

Нам осталось вывести полученные результаты. Это действие не следует включать в функцию Matr1ScatterData, поскольку в дальнейшем данная функция будет использоваться в итоговом задании MPI9Matr7, в котором не требуется выводить результаты, полученные на первом этапе алгоритма. Поэтому поместим операторы вывода в конец функции Solve (добавленные операторы выделены полужирным шрифтом):

```
void Solve()
{
  Task("MPI9Matr2");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  k = size;
  r = rank;
  Matr1ScatterData();
  pt << na << p << nb << q;
  for (int i = 0; i < na*p; ++i)
    pt << a[i];
  for (int i = 0; i < nb*q; ++i)
    pt << b[i];
  for (int i = 0; i < na*q; ++i)
    pt << c[i];
```

```
}
```

Запустив программу, мы получим сообщение о том, что задание выполнено.

Заметим, что при реализации следующих этапов алгоритма нам не потребуется выполнять ввод исходных данных в главном процессе и пересылать их в другие процессы приложения, поскольку в последующих заданиях в каждом процессе сразу будут даны связанные с ним исходные данные.

В качестве подобного примера рассмотрим задание MPI9Matr24, посвященное этапу, который требуется выполнить сразу после начальной рассылки исходных данных (в данном случае выбрано задание из подгруппы, связанной с одним из блочных алгоритмов, а именно алгоритмом Кэннона).

MPI9Matr24. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, заполненные соответствующими блоками матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr23). Реализовать начальное перераспределение блоков, используемое в алгоритме Кэннона блочного перемножения матриц.

Для этого задать на множестве исходных процессов декартову топологию двумерной квадратной циклической решетки порядка K_0 (где $K_0 \cdot K_0$ равно количеству процессов), сохранив исходный порядок нумерации процессов, и выполнить для каждой строки I_0 полученной решетки ($I_0 = 0, \ldots, K_0 - 1$) циклический сдвиг блоков A_R на I_0 позиций влево (т. е. в направлении убывания рангов процессов), а для каждого столбца J_0 решетки ($J_0 = 0, \ldots, K_0 - 1$) циклический сдвиг блоков B_R на J_0 позиций вверх (т. е. также в направлении убывания рангов процессов).

Для определения коммуникатора MPI_COMM_GRID, связанного с декартовой топологией, использовать функцию Matr3CreateCommGrid, реализованную в задании MPI9Matr22. При выполнении циклического сдвига использовать функции MPI_Cart_coords, MPI_Cart_shift, MPI_Sendrecv_replace (ср. с MPI9Matr22).

Оформить описанные действия в виде функции Matr3Init (без параметров). Вывести в каждом процессе блоки A_R и B_R, полученные в результате сдвига (ввод и вывод данных выполнять во внешней функции Solve).

Заготовка для этого задания имеет следующий вид:

```
#include "mpi.h"
#include "pt4.h"
#include <cmath>

int k;              // количество процессов
int r;              // ранг текущего процесса
```

```
int m, p, q;        // размеры исходных матриц
int m0, p0, q0;     // размеры блоков матриц
int k0;             // порядок декартовой решетки, равный sqrt(k)

int *a_, *b_, *c_;
  // массивы для хранения исходных матриц в главном процессе
int *a, *b, *c;
  // массивы для хранения блоков матриц в каждом процессе

MPI_Datatype MPI_BLOCK_A; // тип данных для блока матрицы A
MPI_Datatype MPI_BLOCK_B; // тип данных для блока матрицы B
MPI_Datatype MPI_BLOCK_C; // тип данных для блока матрицы C

MPI_Comm MPI_COMM_GRID = MPI_COMM_NULL;
  // коммуникатор, связанный с двумерной декартовой решеткой

void Solve()
{
  Task("MPI9Matr24");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  k = size;
  r = rank;
  k0 = (int)floor(sqrt((double)k) + 0.1);

}
```

В число глобальных переменных, уже описанных в заготовке, входят не только массивы a, b, c для хранения блоков матриц в каждом процессе и переменные m0, p0, q0, определяющие размеры этих блоков, но также и объекты, связанные с новыми *типами данных* и *коммуникаторами*, используемыми в алгоритме Кэннона. Заметим, что в заданиях на реализацию блочных алгоритмов количество процессов K является *полным квадратом*: $K = K_0 \cdot K_0$; при этом для хранения значения K_0 (порядка декартовой решетки процессов) предусмотрена специальная переменная k0, уже содержащая требуемое значение (см. последний оператор в функции Solve).

При запуске созданной заготовки на экране появится окно задачника, приведенное на рис. и (первый рисунок содержит начальную часть раздела с исходными данными, а второй — раздел с примером правильных результатов).

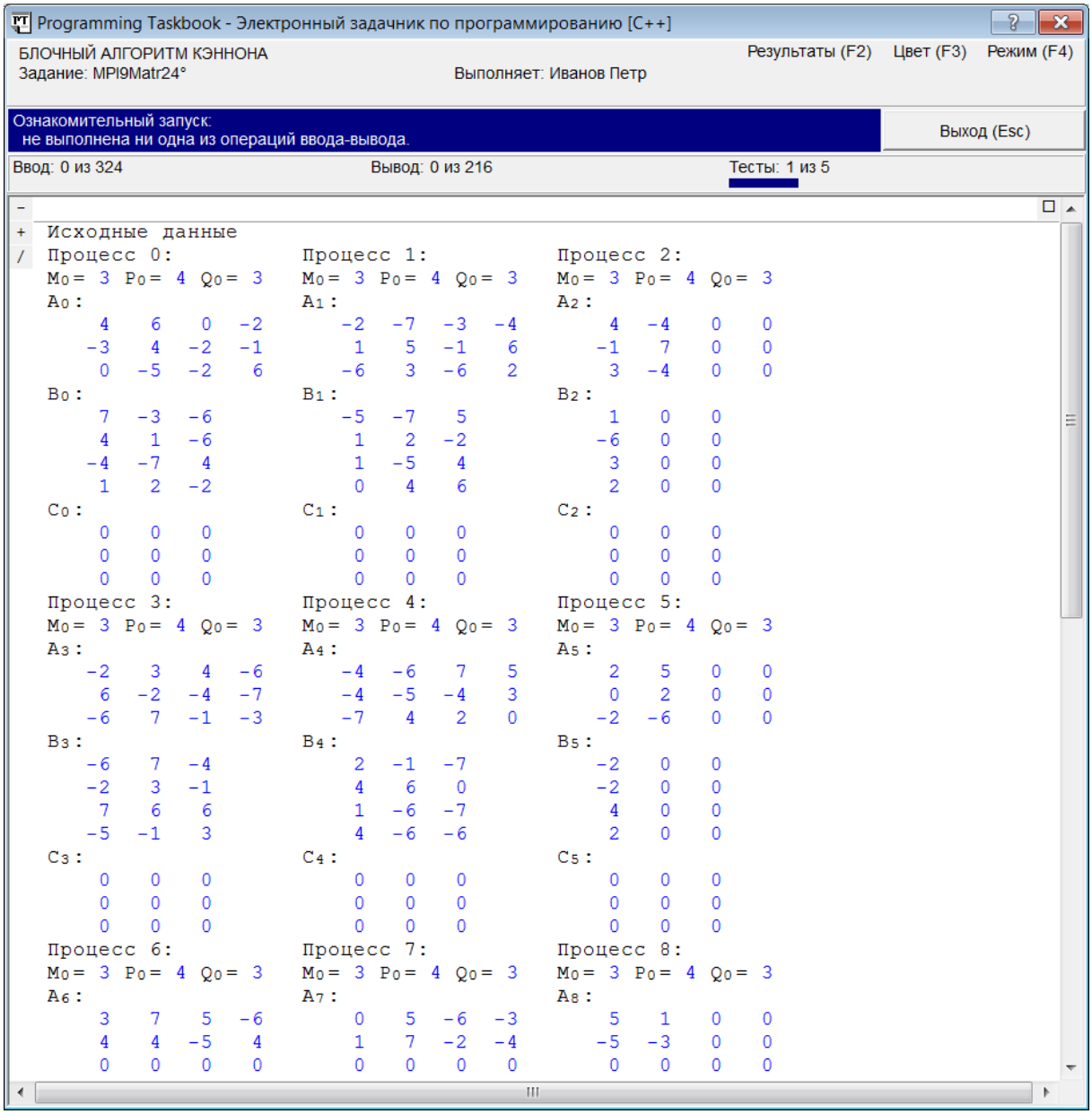


Рис. 38. Ознакомительный запуск задания MPI9Matr24 (раздел с исходными данными)

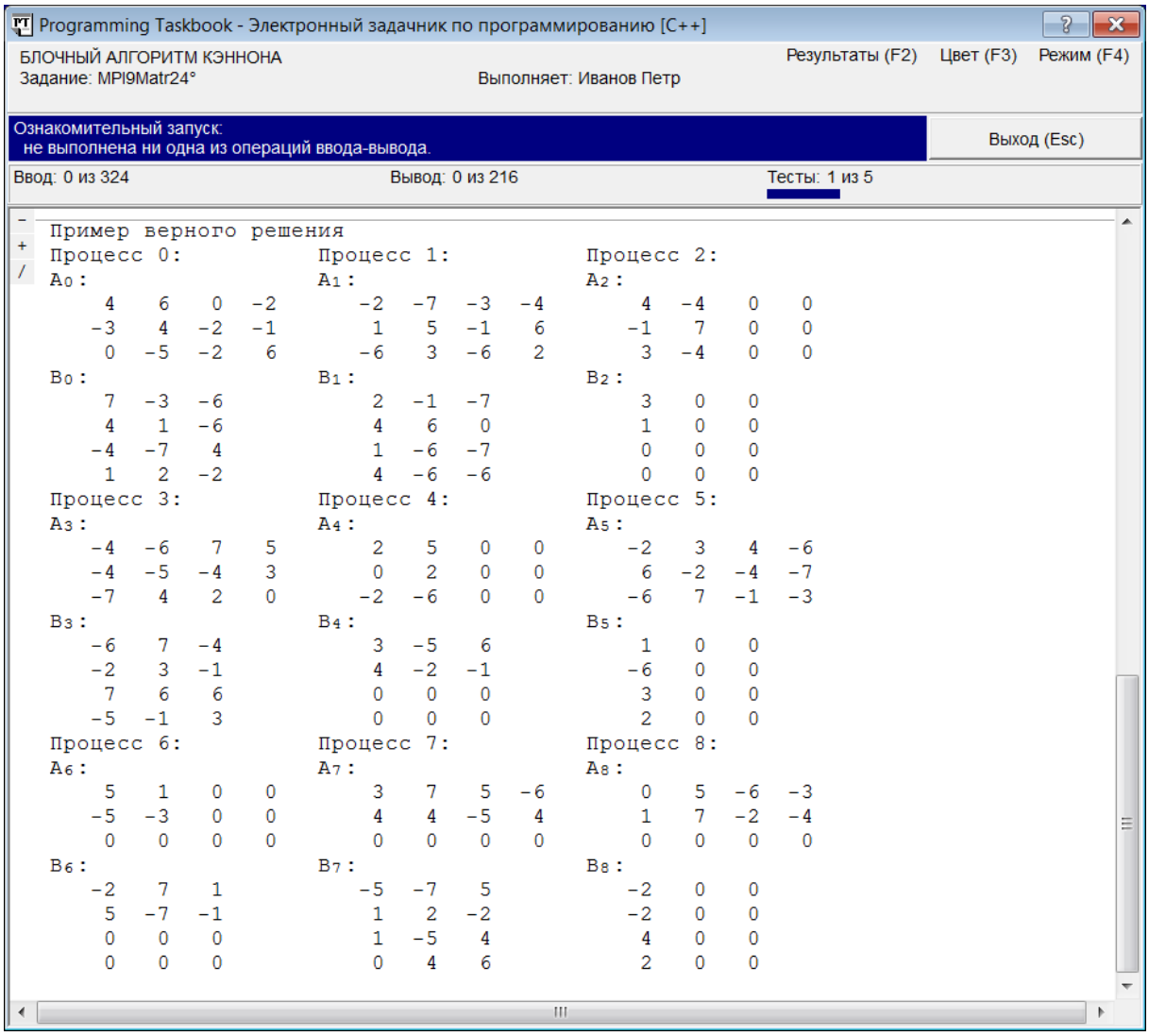


Рис. 39. Ознакомительный запуск задания MPI9Matr24 (пример верного решения)

В данном случае число процессов K равно 9, а порядок декартовой решетки K_0, соответственно, равен 3.

Действия, связанные с вводом исходных данных, ничем не отличаются от рассмотренных в заданиях MPI9Matr1 и MPI9Matr2, однако теперь они должны выполняться для каждого из процессов параллельного приложения (данный фрагмент надо добавить в конец функции Solve):

```
pt >> m0 >> p0 >> q0;
a = new int[m0*p0];
b = new int[p0*q0];
c = new int[m0*q0];
for (int i = 0; i < m0*p0; i++)
  pt >> a[i];
for (int i = 0; i < p0*q0; i++)
  pt >> b[i];
for (int i = 0; i < m0*q0; i++)
```

```
  pt >> c[i];
```

Следует заметить, что содержимое блоков матрицы *C* в данном задании не используется, однако его требуется ввести, поскольку во всех заданиях, связанных с этапом вычисления матричного произведения, предлагается одинаковый набор исходных данных, совпадающий с результатами, полученными на этапе рассылки исходных данных по всем процессам приложения (ср. с приведенным выше решением задачи MPI9Matr2).

При запуске нового варианта программы будет выведено сообщение о том, что все исходные данные успешно введены.

Обратимся к реализации начального перераспределения блоков.

Прежде всего, необходимо описать вспомогательную функцию Matr3CreateCommGrid, предназначенную для создания коммуникатора с топологией двумерной квадратной циклической решетки порядка k0. С данной функцией связано специальное задание MPI9Matr22, представляющее собой, по существу, более простой вариант задания MPI9Matr24. Поскольку мы не выполняли задание MPI9Matr22, функцию Matr3CreateCommGrid придется реализовать непосредственно при выполнении нашего задания, используя рекомендации, приведенные в задании MPI9Matr22:

```
void Matr3CreateCommGrid(MPI_Comm &comm)
{
  int dims[] = {k0, k0};
  int periods[] = {1, 1};
  MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, 0, &comm);
}
```

В описании функции Matr3CreateCommGrid мы использовали глобальную переменную k0, содержащую порядок декартовой решетки.

Теперь опишем основную функцию Matr3Init. Вначале в ней необходимо создать коммуникатор с требуемой декартовой топологией (заметим, что связанная с этим коммуникатором переменная MPI_COMM_GRID уже описана в программе-заготовке). Затем необходимо определить координаты текущего процесса в этом коммуникаторе, используя функцию MPI_Cart_coords. Координаты будем хранить во вспомогательном массиве coord с двумя элементами: (coord[0], coord[1]). Используя эти координаты, следует выполнить пересылку блоков матрицы *A* на coord[0] позиций *влево*, т. е. в направлении убывания *второй* координаты решетки (соответствующей номерам столбцов) и учитывая ее цикличность по этой координате, а блоков матрицы *B* — на coord[1] позиций *вверх*, т. е. в направлении убывания *первой* координаты решетки (соответствующей номерам строк) с учетом ее цикличности. Чтобы выполнить условия задания, ранги посылающих и принимающих процессов следует определить с использованием функции MPI_Cart_shift, а саму пересылку осуществить с помощью функ-

ции MPI_Sendrecv_replace (благодаря применению этой функции, принимаемый блок будет размещен на месте прежнего блока, пересланного в другой процесс):

```
void Matr3Init()
{
  Matr3CreateCommGrid(MPI_COMM_GRID);
  int coord[2];
  MPI_Cart_coords(MPI_COMM_GRID, r, 2, coord);
  int src, dst;
  MPI_Cart_shift(MPI_COMM_GRID, 1, -coord[0], &src, &dst);
  MPI_Sendrecv_replace(a, m0*p0, MPI_INT, dst, 0, src, 0,
    MPI_COMM_GRID, MPI_STATUS_IGNORE);
  MPI_Cart_shift(MPI_COMM_GRID, 0, -coord[1], &src, &dst);
  MPI_Sendrecv_replace(b, p0*q0, MPI_INT, dst, 0, src, 0,
    MPI_COMM_GRID, MPI_STATUS_IGNORE);
}
```

В функции Matr3Init мы использовали, наряду с локальными переменными coord, src и dst, глобальные переменные MPI_COMM_GRID, r (ранг текущего процесса), a, b (массивы с блоками матриц *A* и *B*) и m0, p0, q0 (размеры данных блоков).

Осталось вызвать эту функцию во внешней функции Solve и вывести новое содержимое блоков матриц *A* и *B* в каждом процессе. Приведем итоговое содержимое функции Solve (добавленные операторы выделены полужирным шрифтом):

```
void Solve()
{
  Task("MPI9Matr24");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  k = size;
  r = rank;
  k0 = (int)floor(sqrt((double)k) + 0.1);
  pt >> m0 >> p0 >> q0;
  a = new int[m0*p0];
  b = new int[p0*q0];
  c = new int[m0*q0];
```

```
  for (int i = 0; i < m0*p0; ++i)
    pt >> a[i];
  for (int i = 0; i < p0*q0; ++i)
    pt >> b[i];
  for (int i = 0; i < m0*q0; ++i)
    pt >> c[i];
  Matr3Init();
  for (int i = 0; i < m0*p0; ++i)
    pt << a[i];
  for (int i = 0; i < p0*q0; ++i)
    pt << b[i];
}
```

После запуска данного варианта программы в окне задачника будет выведено сообщение о том, что задание выполнено.

Примечания. 1. Смысл инициализирующей пересылки блоков матриц *A* и *B* заключается в том, что в результате этой пересылки каждый процесс получит пару блоков, перемножение которых позволит определить часть слагаемых для элементов соответствующего блока матричного произведения *C*.

2. Реализованная в данном задании функция Matr3Init будет использоваться далее в заданиях MPI9Matr28 и MPI9Matr31, которые посвящены реализации алгоритма Кэннона в полном объеме.

На завершающем этапе любого матричного алгоритма требуется объединить фрагменты итогового матричного произведения *C*, полученные в различных процессах. Результатом подобного объединения может являться, например, массив, полученный в главном процессе и содержащий все элементы матрицы *C* (которые можно вывести на экран или сохранить в файле). При реализации данного этапа с применением библиотеки MPI стандарта 2.0 можно поступить по-другому, сразу записав отдельные фрагменты матрицы *C* в двоичный файл, используя появившиеся в этом стандарте средства параллельного файлового ввода-вывода. Это позволит избежать дополнительных действий, связанных с пересылкой результирующих данных в главный процесс.

В качестве примера рассмотрим задание MPI9Matr19, в котором требуется сохранить в результирующем файле матричное произведение, полученное с помощью второго варианта ленточного алгоритма.

MPI9Matr19. В каждом процессе даны числа N_A, N_B, а также одномерные массивы, заполненные полосами C_R (размера $(N_A \cdot K) \times N_B$), полученными в результате выполнения K шагов ленточного алгоритма перемножения матриц (см. MPI9Matr15). Кроме того, в главном процессе дано число M — количество строк результирующего матричного произве-

дения и имя файла для хранения этого произведения. Дополнительно известно, что количество столбцов Q результирующего матричного произведения кратно числу процессов (и, следовательно, равно $N_B \cdot K$).

Переслать число M и имя файла во все процессы (используя функцию MPI_Bcast) и записать все фрагменты матричного произведения, содержащиеся в полосах C_R, в результирующий файл, который в итоге будет содержать матрицу C размера $M \times Q$.

Для записи полос в файл задать соответствующий вид данных, используя функцию MPI_File_set_view и новый файловый тип MPI_BAND_C, определенный с помощью функции Matr2CreateTypeBand (см. MPI9Matr11), после чего использовать коллективную функцию MPI_File_write_all.

Оформить считывание имени файла, пересылку значения M и имени файла, а также все действия по записи полос в файл в виде функции Matr2GatherFile (считывание всех исходных данных, кроме имени файла, должно осуществляться во внешней функции Solve).

Указание. При записи данных в результирующий файл необходимо учитывать, что полосы C_R могут содержать завершающие нулевые строки, не связанные с полученным произведением (именно для проверки этой ситуации требуется пересылать во все процессы значение M).

Заготовка для данного задания будет содержать те же глобальные переменные, что и приведенная выше заготовка для задания MPI9Matr2, в частности, k (количество процессов), r (ранг текущего процесса), m, p, q, na, nb (размеры матриц A, B, C и их полос), a, b, c (массивы для хранения полос матриц A, B, C в каждом процессе). Кроме того, в число глобальных переменных будут включены переменные MPI_BAND_B и MPI_BAND_C, предназначенные для хранения новых типов данных, связанных с вертикальными полосами матриц B и C.

На рис. 40 приведено окно задачника, которое появится при запуске созданной программы-заготовки.

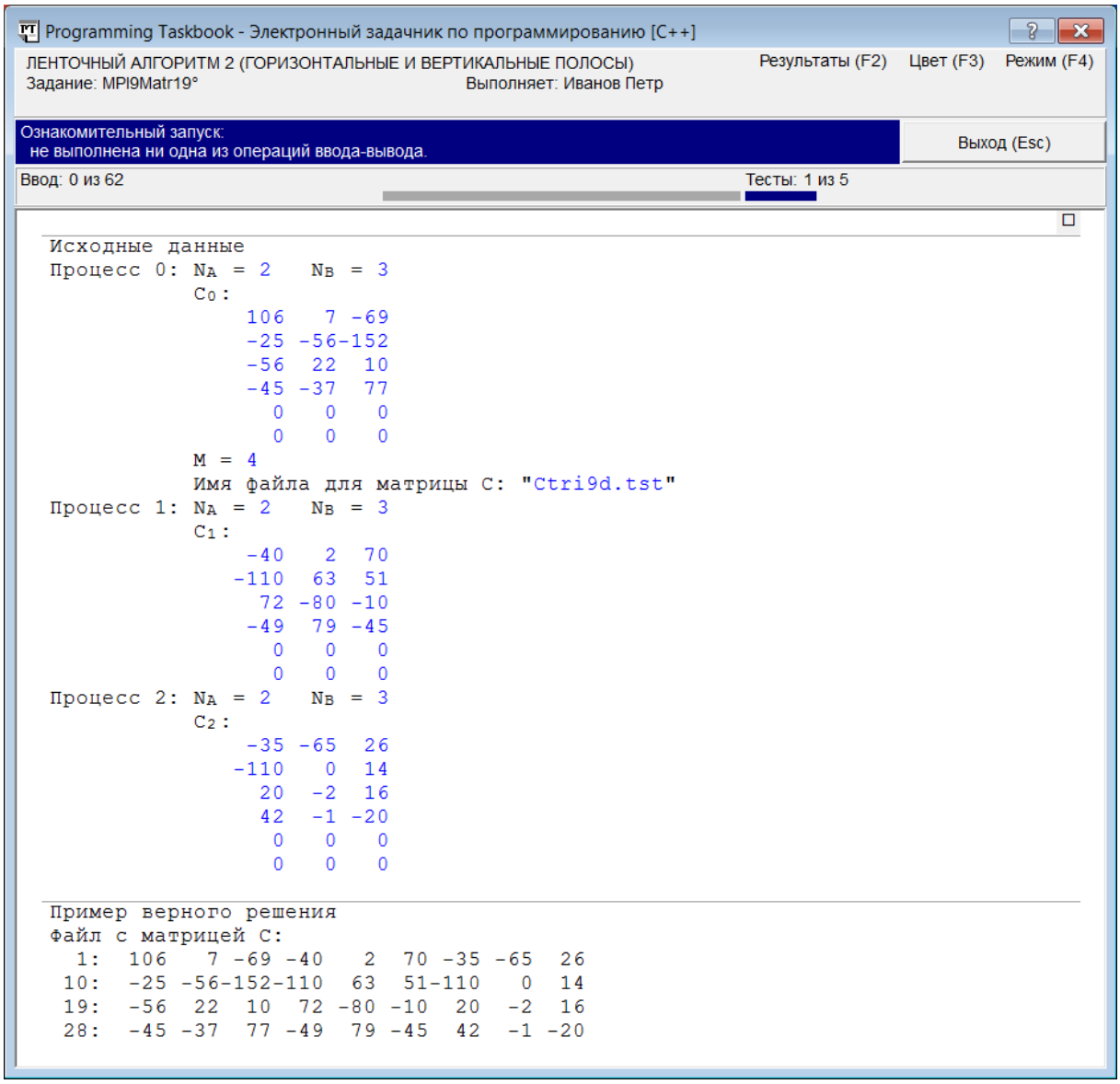


Рис. 40. Ознакомительный запуск задания MPI9Matr19

При выполнении данного задания не требуется выводить результаты, используя поток вывода pt; необходимо лишь записать элементы полученной матрицы *C* в двоичный файл целых чисел (обратите внимание на то, что в разделе индикаторов не указывается количество элементов данных, которые надо вывести). Задачник отобразит содержимое созданного файла в своем окне и проверит его правильность. Отметим также, что в главном процессе указаны дополнительные исходные данные, отсутствующие в подчиненных процессах: это число строк *M* результирующей матрицы *C* и имя файла для ее хранения. Кроме того, следует обратить внимание на то, что полосы матрицы *C*, данные в каждом процессе, содержат не только ее элементы, но и «лишние» нулевые строки, которые не следует сохранять в результирующем файле (наличие этих строк объясняется тем, что произве-

дение $N_A \cdot K$, где K — количество процессов, в данном случае превышает количество строк M матрицы C).

На первом этапе решения надо, как обычно, организовать ввод исходных данных. В условии сказано, что все исходные данные, кроме имени файла (которое дается в главном процессе), надо ввести в функции Solve. Следует также учесть, что число строк M результирующей матрицы C также дано только в главном процессе. Имя файла следует ввести в начале вспомогательной функции Matr2GatherFile. Требование вводить все данные, кроме имени файла, во внешней функции Solve связано с тем, что при реализации алгоритма матричного умножения в полном объеме все эти данные к моменту вызова функции Matr2GatherFile уже будут введены (или вычислены) в соответствующих процессах, поэтому их повторный ввод в функции Matr2GatherFile привел бы к неправильной работе программы.

Итак, опишем вспомогательную функцию Matr2GatherFile, в которой выполним ввод имени файла в главном процессе, и дополним функцию Solve фрагментов, в котором выполняется ввод всех остальных данных и вызывается функция Matr2GatherFile (добавленные операторы выделены полужирным шрифтом):

```
void Matr2GatherFile()
{
  char cname[12];
  if (r == 0)
    pt >> cname;
}

void Solve()
{
  Task("MPI9Matr19");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  k = size;
  r = rank;
  pt >> na >> nb;
  c = new int[na*k*nb];
  for (int i = 0; i < na*k*nb; ++i)
```

```
    pt >> c[i];
  if (r == 0)
    pt >> m;
  Matr2GatherFile();
}
```

При вводе имени файла cname мы учли, что это имя состоит не более чем из 12 символов (см. преамбулу к группе заданий MPI9Matr).

Запустив данный вариант программы, мы получим сообщение о том, что все исходные данные успешно введены, а результирующий файл не создан.

Остальные действия запрограммируем в функции Matr2GatherFile. Прежде всего, в ней необходимо выполнить пересылку числа *M* и имени файла во все процессы:

```
MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(cname, 12, MPI_CHAR, 0, MPI_COMM_WORLD);
```

Затем надо описать файловую переменную и связать с ней создаваемый файл, открыв его на запись с помощью функции MPI_File_open. Целесообразно сразу добавить в программу и функцию закрытия данного файла:

```
MPI_File f;
MPI_File_open(MPI_COMM_WORLD, cname,
  MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);

MPI_File_close(&f);
```

Все другие действия с файлом надо выполнять между вызовами функций MPI_File_open и MPI_File_close.

Запуск данного варианта программы приведет к новому сообщению, в котором, кроме информации о том, что все исходные данные введены, будет отмечено, что результирующий файл является пустым. Это показывает, что пересылка имени файла во все процессы и действия по его созданию и закрытию в каждом процессе были успешно выполнены.

Осталось обеспечить правильную запись полос результирующей матрицы в созданный файл. Для этого необходимо определить специальный вид данных, поскольку каждый процесс должен записать в файл не набор последовательно расположенных элементов (как было бы, если бы в процессе содержались *строки* матрицы), а несколько фрагментов с «пропусками» (каждый фрагмент соответствует элементам отдельной строки, а пропуски обеспечивают переход к соответствующим элементам следующей строки). Каждая полоса C_R имеет размер $(N_A \cdot K) \times N_B$; таким образом, она содержит N_B соседних столбцов. Это означает, что фрагмент для каждой строки должен содержать N_B соседних элементов. Каково количество таких фрагментов? Казалось бы, оно должно быть равно числу строк в по-

лосе C_R, т. е. значению $N_A \cdot K$, однако не следует забывать, что число строк в полосе может быть больше, чем число строк M в результирующей матрице («лишние» строки являются нулевыми, и их не следует записывать в результирующий файл). Таким образом, число записываемых в файл фрагментов должно быть равно значению M.

Итак, каждый процесс должен записывать в файл M фрагментов по N_B последовательных чисел в каждом фрагменте, причем расстояние между начальными позициями соседних фрагментов должно быть равно Q, т. е. *числу столбцов* результирующей матрицы (это приведет к тому, что фрагменты, записываемые в файл каждым процессом, будут образовывать *столбцы* итоговой матрицы).

Подобные составные типы данных (связанные с набором соседних *столбцов* матрицы) требуются и при реализации других этапов ленточного алгоритма, использующего вертикальные полосы. В частности, он упрощает пересылку исходных полос матрицы B (которые также представляют собой наборы столбцов) и объединение полученных полос матрицы C в главном процессе (если в алгоритме не используется файловый ввод-вывод). Поэтому было бы целесообразно описать вспомогательную функцию Matr2CreateTypeBand, которая упростила бы создание типов, связанных с вертикальными полосами.

Реализации функции Matr2CreateTypeBand посвящено особое задание MPI9Matr11, являющееся первым в серии заданий, связанных со вторым вариантом ленточного алгоритма. Поскольку мы не выполняли задание MPI9Matr22, функцию Matr2CreateTypeBand придется реализовать непосредственно при выполнении нашего задания, используя рекомендации, приведенные в задании MPI9Matr11 (напомним, что с аналогичной ситуацией мы сталкивались ранее при решении задачи MPI9Matr24, в которой нам пришлось реализовать вспомогательную функцию Matr3CreateCommGrid, определяющую новый коммуникатор с декартовой топологией).

Как сказано в задании MPI9Matr11, для реализации производного типа данных, связанного со столбцами матрицы, следует использовать функцию MPI_Type_vector. Напомним смысл первых трех параметров этой функции (все они имеют целый тип):

- число фрагментов, содержащих последовательные элементы;
- размер каждого фрагмента (в элементах);
- расстояние между начальной позицией соседних фрагментов (тоже в элементах).

Следующий параметр определяет тип элементов, входящих в фрагмент (в нашем случае это MPI_INT), а последний параметр является выходным — это созданный тип данных.

При описании вспомогательной функции Matr2CreateTypeBand удобнее использовать параметры, связанные с характеристиками вертикальных полос. В задании MPI9Matr11 предлагается использовать параметры p, n, q, где p и n определяет число строк и столбцов полосы, а q — число столбцов матрицы, из которой эта полоса извлекается. Нетрудно заметить, что эти параметры в точности соответствуют по смыслу трем первым параметрам функции MPI_Type_vector. Получаем следующую реализацию функции Matr2CreateTypeBand:

```
void Matr2CreateTypeBand(int p, int n, int q, MPI_Datatype &t)
{
  MPI_Type_vector(p, n, q, MPI_INT, &t);
  MPI_Type_commit(&t);
}
```

Включенный в данную функцию вызов функции MPI_Type_commit необходим, если новый тип будет использоваться при пересылке данных. При выполнении задания MPI9Matr19 вызов MPI_Type_commit не требуется, однако мы привели именно такой вариант реализации функции Matr2CreateTypeBand, поскольку он может использоваться во всех заданиях, связанных со вторым вариантом ленточного алгоритма.

Описав функцию Matr2CreateTypeBand, вернемся к определению файлового вида данных. Для этого определения предназначена функция MPI_File_set_view, которая ранее подробно рассматривалась в п. , посвященном заданиям на файловый ввод-вывод. Напомним, что для определения файлового вида данных надо указать начальное смещение (в байтах), элементарный тип (в нашем случае MPI_INT) и собственно файловый тип, для определения которого мы воспользуемся функцией Matr2CreateTypeBand. Что касается начального смещения, то для процесса ранга r оно должно быть равно int_sz*nb*r, где int_sz — это размер базового типа MPI_INT в байтах, а nb — число столбцов каждой полосы. Для определения значения int_sz проще всего воспользоваться функцией MPI_Type_size. Таким образом, фрагмент функции Matr2GatherFile, связанный с определением файлового вида, будет выглядеть следующим образом:

```
Matr2CreateTypeBand(m, nb, nb*k, MPI_BAND_C);
int int_sz;
MPI_Type_size(MPI_INT, &int_sz);
MPI_File_set_view(f, int_sz*nb*r, MPI_INT, MPI_BAND_C,
  "native", MPI_INFO_NULL);
```

Напомним, что переменную MPI_BAND_C описывать не требуется, так как она уже описана в нашей программе как глобальная переменная. При определении типа MPI_BAND_C мы учли, что в каждом процессе вертикальная полоса с элементами матрицы *C* (без завершающих нулевых

строк) имеет размер $M \times N_B$, а число столбцов Q матрицы C равно произведению $N_B \cdot K$, где K — количество процессов.

После определения файлового вида мы можем реализовать запись в файл всех элементов каждой полосы с помощью *единственного* вызова коллективной функции MPI_File_write_all:

```
MPI_File_write_all(f, c, m*nb, MPI_INT, MPI_STATUS_IGNORE);
```

В этом вызове мы указываем массив, содержащий полосу, и количество тех его элементов, которые надо записать в файл (напомним, что массив может также содержать завершающие нулевые элементы, которые записывать в файл не требуется).

Приведем окончательный вид функции Matr2GatherFile, выделив в нем операторы, связанные с определением файлового вида и записью данных в файл:

```
void Matr2GatherFile()
{
  char cname[12];
  if (r == 0)
    pt >> cname;
  MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(cname, 12, MPI_CHAR, 0, MPI_COMM_WORLD);
  MPI_File f;
  MPI_File_open(MPI_COMM_WORLD, cname,
    MPI_MODE_WRONLY | MPI_MODE_CREATE, MPI_INFO_NULL, &f);
  Matr2CreateTypeBand(m, nb, nb*k, MPI_BAND_C);
  int int_sz;
  MPI_Type_size(MPI_INT, &int_sz);
  MPI_File_set_view(f, int_sz*nb*r, MPI_INT, MPI_BAND_C,
    "native", MPI_INFO_NULL);
  MPI_File_write_all(f, c, m*nb, MPI_INT, MPI_STATUS_IGNORE);
  MPI_File_close(&f);
}
```

Запустив данный вариант программы, мы получим сообщение о том, что задание выполнено.