Министерство образования и науки Российской Федерации

Федеральное государственное автономное
образовательное учреждение высшего образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

М. Э. Абрамян

Параллельное программирование на основе технологии MPI 2.0

Учебник по курсу
«Параллельное программирование»
для студентов направлений 02.03.02 и 02.04.02
«Фундаментальная информатика
и информационные технологии»
(бакалавриат и магистратура)

Ростов-на-Дону – Таганрог

Издательство Южного федерального университета

2018

УДК 004.42(075.8)

ББК 32.973я73

А 13

Печатается по решению учебно-методической комиссии
Института математики, механики и компьютерных наук им. И.И. Воровича
Южного федерального университета (протокол № 3 от 20 марта 2018 г.)

Рецензенты:

доктор физико-математических наук, профессор,
заведующий кафедрой цифровых технологий факультета компьютерных наук
Воронежского государственного университета
С. Д. Кургалин

доктор технических наук, профессор кафедры «Информатика»
Ростовского государственного университета путей сообщения (РГУПС)
М. А. Бутакова

*Учебник подготовлен и издан при поддержке Стипендиальной программы
Благотворительного фонда В. Потанина в рамках выполнения проекта
«Электронный задачник по параллельному MPI-программированию» —
одного из проектов-победителей грантового конкурса
для преподавателей магистратуры 2016–2017 года.*

Абрамян, М. Э.

Параллельное программирование на основе технологии MPI 2.0 : учебник / М. Э. Абрамян. – Ростов-на-Дону ; Таганрог : Изд-во ЮФУ, 2018. – 357 с.

ISBN

В первом разделе учебника описывается интерфейс передачи сообщений Message Passing Interface (MPI) стандартов 1.1 и 2.0 для языка C, а также рассматриваются алгоритмы параллельного матричного умножения, при реализации которых применяются различные средства технологии MPI. Описание сопровождается решениями типовых задач. Второй раздел содержит 250 учебных заданий, включенных в разработанный автором электронный задачник Programming Taskbook for MPI-2 и охватывающих все темы первого раздела. В третьем разделе описывается конструктор заданий по параллельному MPI-программированию для электронного задачника. В четвертом разделе дается обзор возможностей задачника Programming Taskbook for MPI-2 и приводится набор из 24 вариантов индивидуальных заданий, составленных из задач второго раздела. Учебник снабжен указателем, содержащим все рассмотренные константы, типы и функции интерфейсов MPI-1 и MPI-2.

Для студентов бакалавриата и магистратуры, обучающихся по направлению подготовки «Фундаментальная информатика и информационные технологии».

УДК 004

ББК 32.973

ISBN

© М. Э. Абрамян, 2018

Оглавление

Предисловие 6
Раздел 1. Технология MPI: описание и примеры использования 8
1.1. Введение в технологию MPI 8
1.1.1. Технология MPI и ее изучение с помощью электронного задачника PT for MPI-2 8
1.1.2. Основные понятия MPI-программирования 11
1.1.3. Создание заготовки для параллельной программы 13
1.1.4. Запуск программы в параллельном режиме 18
1.1.5. Выполнение задания MPI1Proc2 24
1.1.6. Использование дополнительной информации в разделе отладки 30
1.2. Базовые возможности интерфейса MPI (стандарт MPI-1) 33
1.2.1. Блокирующая пересылка сообщений между двумя процессами: основные возможности 33
1.2.2. Блокирующая пересылка сообщений между двумя процессами: примеры. Взаимные блокировки процессов 37
1.2.3. Неблокирующая пересылка сообщений между двумя процессами. Отложенные запросы на взаимодействие. Функции замера времени 44
1.2.4. Коллективные операции 48
1.2.5. Операции редукции. Использование составных типов данных 52
1.2.6. Определение пользовательских типов и упаковка данных. Использование динамических массивов и контейнеров vector 58
1.2.7. Создание новых коммуникаторов 73
1.2.8. Декартова топология 81
1.2.9. Топология графа 91
1.3. Дополнительные возможности интерфейса MPI (стандарт MPI-2) 97
1.3.1. Топология распределенного графа 97
1.3.2. Параллельный ввод-вывод. Функции файлового доступа 100
1.3.3. Параллельный ввод-вывод: пример. Настройка вида файловых данных 105
1.3.4. Односторонние коммуникации: общее описание 112
1.3.5. Односторонние коммуникации: пример с использованием простейшего варианта синхронизации 119
1.3.6. Односторонние коммуникации: пример более сложного варианта синхронизации 123

1.3.7. Интеркоммуникаторы ... 130
1.3.8. Динамическое создание процессов ... 139
1.4. Параллельные матричные алгоритмы ... 150
1.4.1. Ленточные и блочные алгоритмы параллельного умножения матриц: общее описание ... 150
1.4.2. Реализация непараллельного алгоритма умножения матриц ... 154
1.4.3. Рассылка исходных данных: пример реализации ... 159
1.4.4. Перераспределение блоков на начальном этапе алгоритма Кэннона ... 165
1.4.5. Этап сборки результатов: пример реализации, основанной на файловом выводе ... 171
1.5. Дополнительные приемы разработки параллельных программ ... 178
1.5.1. Отладка параллельных программ с применением средств электронного задачника ... 178
1.5.2. Разработка и запуск параллельных программ без использования электронного задачника ... 188
1.5.3. Дополнительные возможности, связанные с отладочным выводом. Перенаправление вывода ... 193
Раздел 2. Учебные задания ... 201
2.1. Процессы и их ранги ... 201
2.2. Обмен сообщениями между отдельными процессами ... 202
2.2.1. Блокирующая пересылка данных ... 203
2.2.2. Неблокирующая пересылка данных ... 208
2.3. Коллективные взаимодействия ... 211
2.3.1. Коллективная пересылка данных ... 211
2.3.2. Коллективные операции редукции ... 213
2.4. Производные типы и упаковка данных ... 215
2.4.1. Использование простейших производных типов ... 215
2.4.2. Пересылка упакованных данных ... 216
2.4.3. Более сложные виды производных типов ... 217
2.4.4. Коллективная функция MPI_Alltoallw (MPI-2) ... 221
2.5. Группы процессов и коммуникаторы ... 221
2.5.1. Создание новых коммуникаторов ... 221
2.5.2. Виртуальные топологии ... 224
2.5.3. Топология распределенного графа (MPI-2) ... 230
2.6. Параллельный ввод-вывод файловых данных (MPI-2) ... 232
2.6.1. Локальный файловый ввод-вывод ... 232
2.6.2. Коллективный файловый ввод-вывод ... 234
2.6.3. Настройка вида файловых данных ... 237
2.7. Односторонние коммуникации (MPI-2) ... 241
2.7.1. Односторонние коммуникации с простейшей синхронизацией ... 242

2.7.2. Дополнительные виды синхронизации 247
2.8. Интеркоммуникаторы и динамическое создание процессов (MPI-2).....252
2.8.1. Создание интеркоммуникаторов.....253
2.8.2. Коллективные операции для интеркоммуникаторов.....258
2.8.3. Динамическое создание процессов.....261
2.9. Параллельные матричные алгоритмы 265
2.9.1. Непараллельный алгоритм умножения матриц.....266
2.9.2. Ленточный алгоритм 1 (горизонтальные полосы) 266
2.9.3. Ленточный алгоритм 2 (горизонтальные и вертикальные полосы).....271
2.9.4. Блочный алгоритм Кэннона.....277
2.9.5. Блочный алгоритм Фокса.....283
Раздел 3. Разработка заданий по параллельному программированию 292
3.1. Особенности функционирования задачника в параллельном режиме 292
3.1.1. Варианты запуска откомпилированной программы 292
3.1.2. Действия программы в режиме загрузчика.....293
3.1.3. Действия программы в параллельном режиме и обработка ошибок.....295
3.2. Дополнительные возможности конструктора учебных заданий 298
3.2.1. Новый вариант функции CreateTask.....299
3.2.2. Функция SetProcess.....302
3.3. Примеры разработки заданий по параллельному программированию.....304
3.3.1. Создание новой группы учебных заданий и импортирование в нее имеющегося задания 304
3.3.2. Разработка простого задания.....308
3.3.3. Разработка задания, связанного с пересылкой данных 313
3.3.4. Разработка сложного задания.....317
Раздел 4. Дополнения 324
4.1. Электронный задачник Programming Taskbook for MPI-2.....324
4.1.1. Общее описание.....324
4.1.2. Средства задачника для инициализации заданий и ввода-вывода данных 327
4.1.3. Раздел отладки 329
4.1.4. Функции для вывода и настройки отладочной информации 332
4.2. Варианты индивидуальных заданий.....334
4.2.1. Серии однотипных задач 334
4.2.2. Набор из 24 вариантов индивидуальных заданий.....338
Литература.....353
Указатель 354

Предисловие

Предлагаемый вашему вниманию учебник представляет собой практическое введение в параллельное программирование на основе технологии MPI — *интерфейса передачи сообщений* (Message Passing Interface). В настоящее время данная технология является одной из основных технологий параллельного программирования для кластерных систем и компьютеров с распределенной памятью [5–10]. В учебнике рассматриваются стандарты MPI версий 1.1 [11] и 2.x [12], на которых основано большинство современных программных реализаций. Используется вариант интерфейса MPI для языка C, при организации ввода-вывода применяются средства языка C++.

Учебник состоит из трех основных разделов. В первом разделе дается систематическое описание возможностей интерфейса MPI. Подробно рассматриваются базовые возможности данной технологии, включенные во все версии стандарта, в том числе блокирующий и неблокирующий обмен сообщениями между двумя процессами, коллективные взаимодействия процессов, определение производных типов, работа с группами процессов и коммуникаторами, применение виртуальных топологий. Кроме того, изучаются новые возможности, появившиеся (или существенно расширенные) в стандарте MPI-2: параллельный файловый ввод-вывод, односторонние коммуникации, использование интеркоммуникаторов и динамическое создание процессов. Наряду с различными возможностями интерфейса MPI в первом разделе также рассматривается важный класс параллельных алгоритмов, а именно — алгоритмы параллельного матричного умножения, при реализации которых применяются разнообразные средства технологии MPI. Все обсуждаемые темы сопровождаются примерами программного кода, связанного с решением типовых задач.

Второй раздел содержит 250 учебных заданий по всем темам, рассмотренным в первом разделе. Следует отметить, что для практического изучения основных компонентов MPI достаточно использовать локальный компьютер, имитируя на нем параллельное выполнение процессов. Однако даже в этом простейшем варианте учащийся неизбежно сталкивается с дополнительными трудностями при разработке параллельных программ, обусловленными сложностью организации ввода-вывода данных для различных процессов параллельной программы и невозможностью применения для параллельных программ стандартных отладочных средств, предусмотренных в интегрированных средах. Для того чтобы облегчить освое-

ние технологии MPI, автор разработал специализированную учебную систему — электронный задачник по параллельному программированию на базе MPI-2 **Programming Taskbook for MPI-2** (PT for MPI-2). Все задания, включенные во второй раздел учебника, можно выполнять с использованием задачника PT for MPI-2 в средах Microsoft Visual Studio (версия 2008 и выше) и Code::Blocks (версия 13 и выше). Именно наличие большого числа учебных задач, связанных со всеми аспектами технологии MPI, и возможность использования специализированных программных средств, существенно ускоряющих процесс решения задач, являются отличительными особенностями данной книги и основанного на ней подхода к обучению параллельному программированию.

В третьем разделе описываются средства конструктора учебных заданий для задачника PT for MPI-2, позволяющие разрабатывать новые группы задач по параллельному MPI-программированию. Разработка новых групп задач может оказаться полезным видом учебных заданий повышенной сложности, который можно предложить учащимся, в том числе в рамках выполнения курсовых и выпускных работ.

В четвертом, дополнительном разделе дается общее описание электронного задачника PT for MPI-2. В нем также приводится информация о сериях однотипных задач, приведенных во втором разделе книги. Эта информация может оказаться полезной при составлении различных наборов индивидуальных заданий. В качестве примера в этом разделе представлены 24 варианта индивидуальных заданий, которые охватывают все темы, рассмотренные в учебнике.

Включенный в книгу указатель, содержащий константы, типы и функции интерфейса MPI, позволяет использовать ее как справочник по технологии MPI стандарта 1.1 и 2.x.

Учебник представляет собой существенно переработанный и дополненный вариант книги [1]. В нем более подробно рассматриваются возможности стандарта MPI-1 и освещаются новые темы — средства стандарта MPI-2 и параллельные матричные алгоритмы. К 100 задачам, приведенным в книге [1], добавлены 150 новых; а некоторые прежние задачи снабжены новыми указаниями. Большое число новых возможностей появилось и в электронном задачнике PT for MPI-2, который пришел на смену задачнику PT for MPI, использованному в [1].

Получить дополнительную информацию об электронном задачнике PT for MPI-2 и скачать его дистрибутив можно на сайте универсального электронного задачника по программированию Programming Taskbook http://ptaskbook.com/.

Раздел 1. Технология MPI: описание и примеры использования

1.1. Введение в технологию MPI

1.1.1. Технология MPI и ее изучение с помощью электронного задачника PT for MPI-2

Технология MPI (Message Passing Interface — интерфейс передачи сообщений) предоставляет средства для передачи информации между различными процессами параллельного приложения. Первая версия стандарта MPI (MPI-1) была разработана в 1993–1995 годах [11]; уже в 1997 году появилась вторая версия (MPI-2), дополненная большим количеством новых возможностей [12]. В дальнейшем стандарт MPI-2 пересматривался в 2008 и 2009 годах, а в 2012 году появился стандарт MPI 3.0. В настоящее время наиболее распространена версия MPI 1.1, однако все большее число реализаций начинает поддерживать возможности стандарта MPI-2.

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++). Имеются реализации MPI для других языков (например, Python и C#), но чаще всего 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 изучения возможностей технологии MPI вполне достаточно использовать локальный компьютер, запуская на нем все процессы параллельного приложения. Ожидать в такой ситуации существенного выигрыша в быстродействии параллельных алгоритмов не приходится, но с помощью подобных учебных программ можно познакомиться с механизмами MPI и опробовать их в действии. С этой целью автором настоящей книги был разработан *электронный задачник по параллельному программированию* **Programming Taskbook for MPI-2** (PT for MPI-2). Подробное описание задачника PT for MPI-2 содержится в п. 4.1.

Задачник PT for MPI-2 позволяет разрабатывать параллельные программы на языке C++ с применением технологии MPI для языка C. Дополнительные возможности языка C++ используются, в основном, для более

удобной организации ввода-вывода (с применением потоков и итераторов — см. п. 4.1.2), хотя в некоторых ситуациях оказываются полезными и другие средства C++, например шаблонные функции (см. задания MPI2Send22–MPI2Send25 в п. 2.2.1). 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 (версия 2008 и выше) и Code::Blocks (версия 13 и выше).

Таким образом, для возможности выполнения учебных заданий по параллельному программированию с применением задачника PT for MPI-2 на компьютере необходимо, прежде всего, установить одну 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 1.1. Система MPICH2 1.3 позволяет выполнять все задания, входящие в задачник PT for MPI-2.

Примечание. 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: c:\Program Files (x86).

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 admini-**

стратора 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.

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).

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.

На протяжении всей книги будем считать, что при выполнении заданий используется среда Microsoft Visual Studio 2017, а в качестве активной версии системы MPICH выбрана версия MPICH2 1.3.

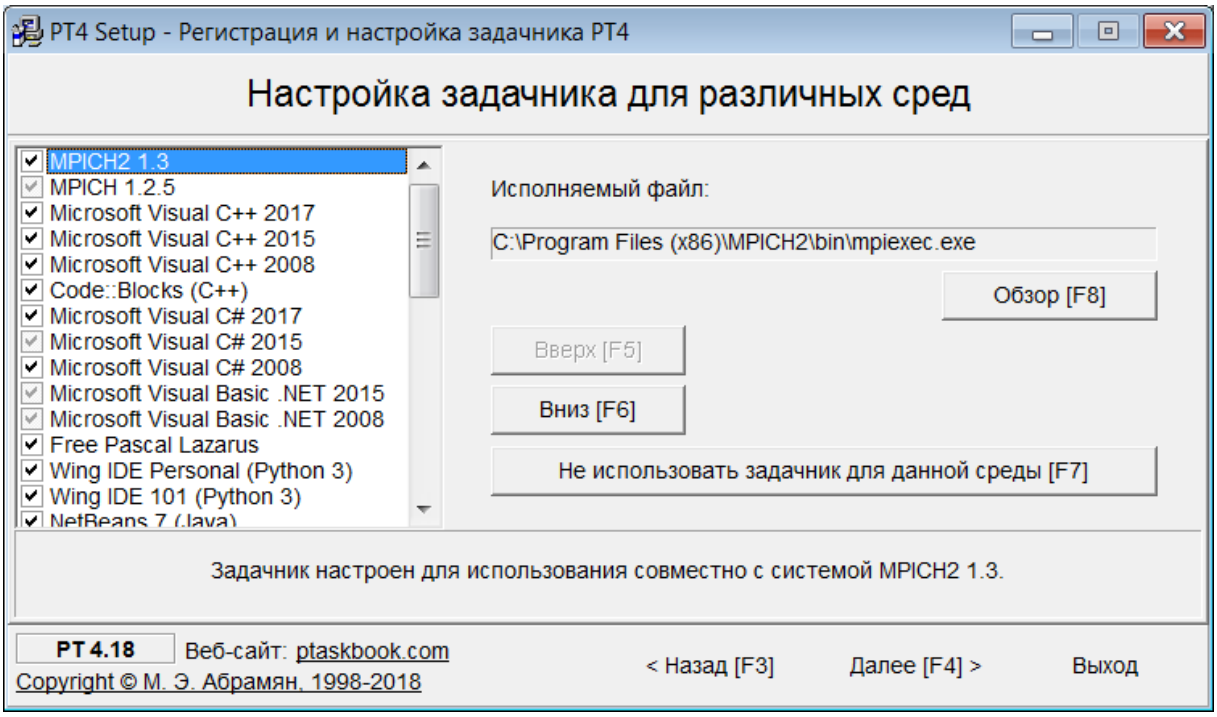

Рис. 1. Окно программы PT4Setup со списков найденных сред разработки

1.1.2. Основные понятия MPI-программирования

Знакомство с параллельным программированием начнем с рассмотрения следующей простой задачи из начальной группы MPI1Proc (см. п. 2.1). Это позволит 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 базовые 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). Функции MPI предоставляют разнообразные средства для реализации такого взаимодействия.

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 (group) 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. Константа MPI_COMM_NULL соответствует «пустому» коммуникатору, который нельзя использовать для пересылки сообщений. Для каждого процесса определен также коммуникатор MPI_COMM_SELF, связанный только с этим процессом. Коммуникатор можно представлять себе как канал, соединяющий между собой процессы, входящие в некоторую группу. Часто 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, информация о которых сохраняется в переменных-дескрипторах типа MPI_Comm. Работа с коммуникаторами рассматривается в группах заданий MPI5Comm и MPI8Inter (см. также п. 1.2.7–1.2.9, 1.3.1 и 1.3.7). 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 группы MPI1Proc).

1.1.3. 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 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 (по умолчанию рабочий каталог имеет имя PT4Work и находится на диске C). После запуска модуля на экране появится его окно (рис. 2).

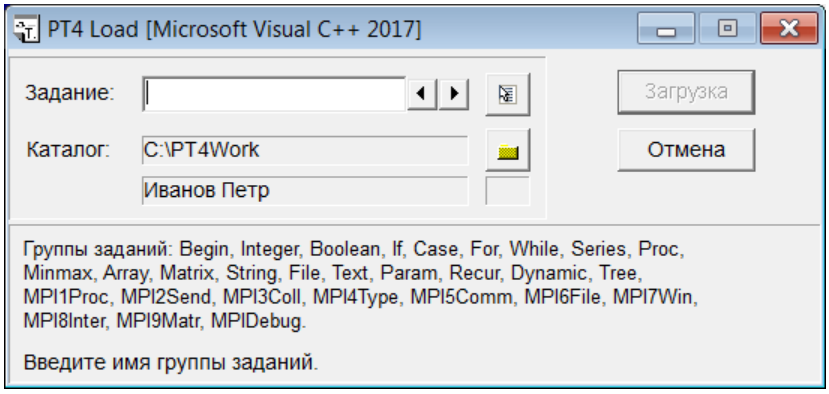

Рис. 2. Окно модуля PT4Load

Так выглядит окно, если текущей программной средой задачника является среда Microsoft Visual Studio 2017 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. 3. 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.

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++.

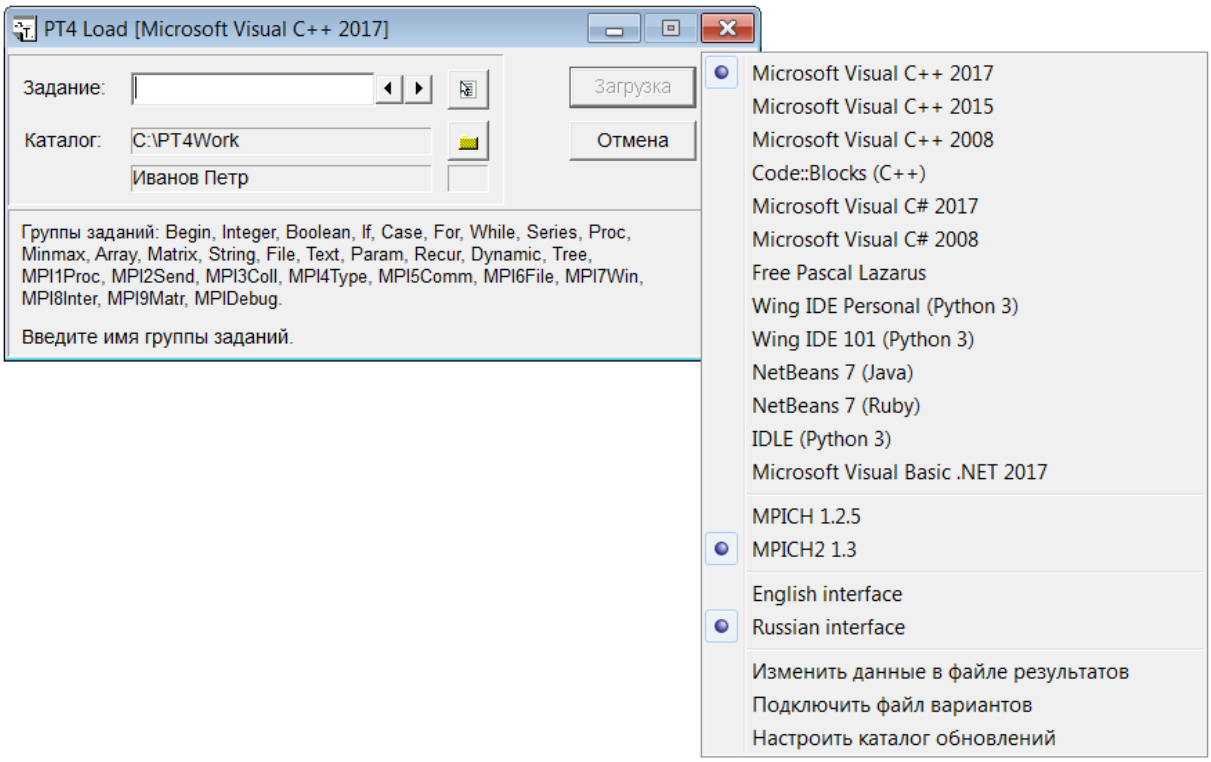

Рис. 3. Модуль PT4Load с развернутым контекстным меню

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 (рис. 4).

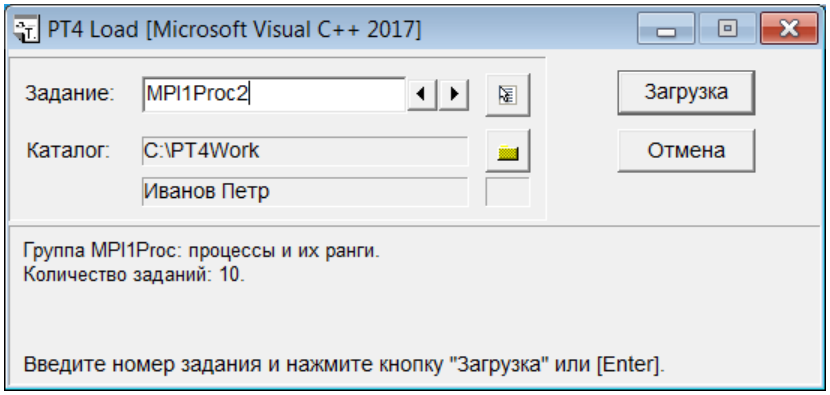

Рис. 4. Вид окна PT4Load после ввода имени задания

Нажав кнопку **Загрузка** или клавишу [Enter], мы создадим заготовку для указанного задания, которая будет немедленно загружена в выбранную среду программирования.

Проект, созданный задачником для языка C++, всегда имеет имя ptprj; это позволяет, в частности, существенно уменьшить количество файлов,

создаваемых в рабочем каталоге при выполнении 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 указаны 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. 4.1.2). Этот оператор имеется в программах-заготовках для всех заданий, в том числе и не связанных с параллельным программи-

рованием. Функция Task реализована в ядре задачника Programming Taskbook (динамической библиотеке) и доступна в учебной программе благодаря подключенному к ней заголовочному файлу pt4.h. Помимо заголовочного файла pt4.h в рабочем каталоге должен находиться файл pt4.cpp, содержащий определения функций, объявленных в файле pt4.h (все эти файлы автоматически добавляются в рабочий каталог при создании проекта-заготовки).

Оставшиеся операторы функции Solve связаны с библиотекой MPI. В п. 1.1.1 отмечалось, 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 необходимо 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 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…** или, для русского интерфейса, **Проект | Свойства <*имя проекта*>…**), перейти в этом окне в раздел **Configuration Properties | Linker | Input** (**Свойства конфигурации | Компоновщик | Ввод**) и указать имя подключаемого файла в поле ввода **Additional Dependencies** (**Дополнительные зависимости**), например mpich.lib; (с завершающей точкой с запятой).

Аналогичные действия надо проделать и в среде Code::Blocks: 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(int* flag) позволяет определить, инициализирован для программы параллельный режим или нет. Если режим инициализирован, то выходной параметр flag принимает значение, отличное

от нуля; в противном случае параметр flag полагается равным нулю. Следует отметить, что инициализация параллельного режима выполняется функцией MPI_Init (см. примечание 3), 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, содержащегося в функции Solve. 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 «?», например Task("MPI1Proc2?")), задачник *не выполняет* инициализацию параллельного режима. В этой ситуации вызов в функции Solve 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 остальные операторы функции Solve, если программа запущена не в параллельном режиме.

Примечание 3. Функция MPI_Init имеет два параметра: (int* argc, char*** argv); первый параметр определяет количество параметров командной строки, а второй содержит сами эти параметры в виде массива типа char*. Параметры передаются по ссылке; это связано с тем, что стандарт MPI предусматривает возможность такой реализации данной функции, при которой передача параметров осуществляется не из параллельной программы в среду MPI, а наоборот: из среды MPI в параллельную программу. Заметим также, что функция MPI_Init должна быть вызвана всеми процессами параллельного приложения.

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(MPI_Comm comm, int* size)) и ранг текущего процесса (функция MPI_Comm_rank(MPI_Comm comm, int* rank)). Текущим считается процесс, вызвавший данную функцию. Требуемая характеристика возвращается во втором параметре соответствующей функции (который является *указателем*); первым параметром является коммуникатор comm, задающий группу процессов. Если текущий процесс не входит в коммуникатор comm, то в параметре rank возвращается значение MPI_UNDEFINED. 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).

Примечание 4. 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* (см. п. 4.1.3). Некоторые возможности MPI, связанные с обработкой ошибок, рассматриваются в заданиях MPI5Comm23–24, более подробное описание средства MPI, связанных с обработкой ошибок, приводится, например, в [8, гл. 8] и [10, гл. 11].

Примечание 5. В библиотеке MPI предусмотрена также функция MPI_Finalize() без параметров, 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).

Как было отмечено выше, функция MPI_Initialized возвращает ненулевой флаг, если в программе была вызвана функция MPI_Init. При этом вызов функции MPI_Finalize никак не влияет на результат работы функции MPI_Initialized. Возможность проверки того, был ли выполнен вызов функции MPI_Finalize, была реализована только в стандарте MPI-2. В нем была добавлена функция MPI_Finalized(int* flag), возвращающая ненулевое значение параметра flag в случае, если программа уже вызвала функцию MPI_Finalize.

1.1.4. 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 этой программы.

Для запуска программы в параллельном режиме необходима *управляющая программа* (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.

В п. 1.1.1 уже отмечалось, 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, 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-2 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 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, затем опять откомпилировать ее и запустить bat-файл, и т. д.

Примечание 1. В среде Microsoft Visual Studio предусмотрен механизм, упрощающий тестирование программ, для запуска которых требуется управляющая программа. В настройках проекта (команда меню **Project | <*имя проекта*> Properties…** или **Проект | Свойства <*имя проекта*>…**) в разделе **Debugging** (**Отладка**) можно указать эту управляющую программу в поле **Command** (**Команда**); в нашем случае это будет MPIRun.exe или mpiexec.exe. Параметры запуска программы указываются в поле **Command Arguments** (**Аргументы команды**); 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 во многих средах 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. 5.

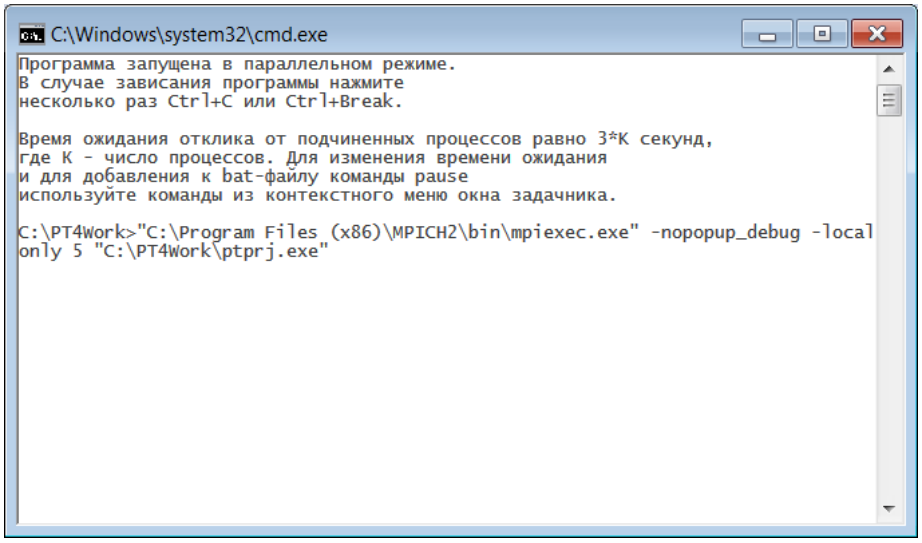

Рис. 5. Консольное окно с информацией о запуске программы в параллельном режиме

После нескольких строк информационного сообщения в этом окне отображается командная строка, которая обеспечивает запуск программы ptprj.exe в параллельном режиме под управлением mpiexec.exe:

```
C:\PT4Work>"C:\Program Files (x86)\MPICH2\bin\mpiexec.exe"
  -nopopup_debug -localonly 5 "C:\PT4Work\ptprj.exe"
```

Число «5», указанное перед полным именем exe-файла (C:\PT4work\ptprj.exe), 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. 6), в котором следует выбрать вариант **Разрешить доступ**.

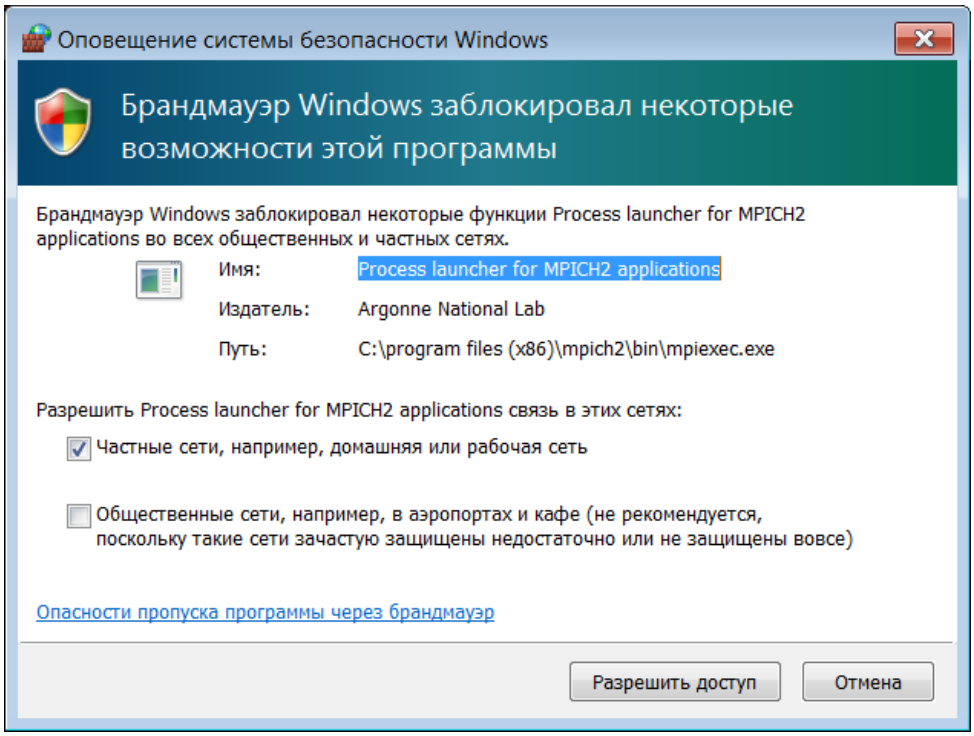

Рис. 6. Окно с запросом о блокировке запущенной параллельной программы

Наконец, на экране появится окно задачника (рис. 7). Внешне это окно ничем не отличается от окна, возникающего при выполнении обычной, «непараллельной» программы. Однако 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. 3.1).

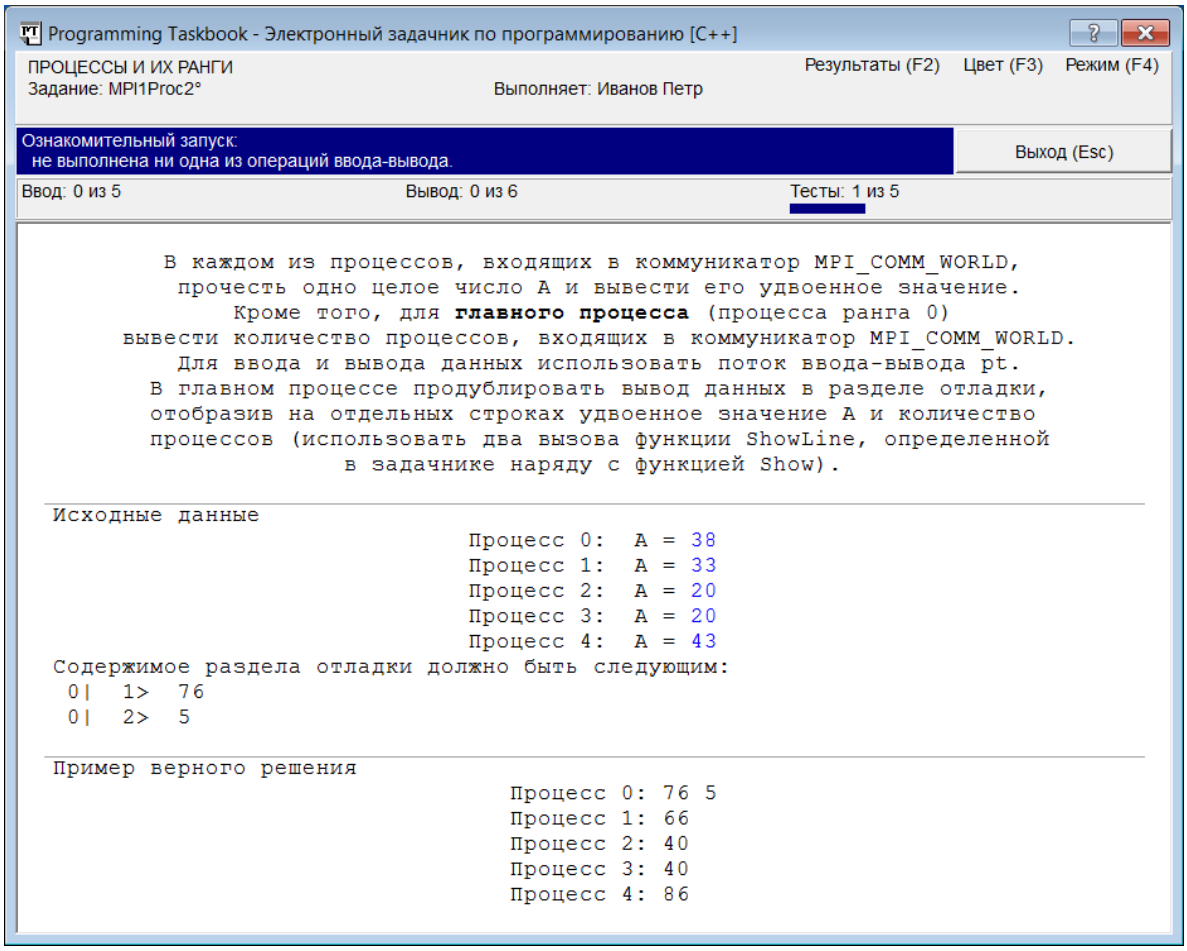

Рис. 7. Ознакомительный запуск задания MPI1Proc2

«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 полученных результатов.

Поскольку в нашем случае ни в одном процессе не была указана ни одна операция ввода-вывода, данный запуск параллельной программы считается ознакомительным, о чем и сообщается в информационном разделе окна задачника. Отметим, что данное окно отображается *главным процессом* параллельной программы, в то время как все подчиненные про-

цессы (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 любой из этих ситуаций 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. Заметим, что пока зависшие процессы остаются в памяти, они не позволяют изменять исполняемый файл программы (в частности, заменять его на новый откомпилированный вариант). В такой ситуации приходится вызывать диспетчер задач Windows (комбинацией [Crtl]+[Alt]+[Del]) и вручную завершать выполнение зависших процессов на вкладке **Процессы**. Автоматическая выгрузка зависших процессов, выполняемая стартовым экземпляром программы, избавляет учащегося от необходимости выполнять подобные действия.

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 отображает 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 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 команду контекстного меню окна задачника **Увеличить время ожидания отклика от подчиненных процессов** (в меню также содержится команда **Уменьшить время ожидания отклика от подчиненных процессов**).

1.1.5. 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 return, 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;
```

Добавленные операторы выделены полужирным шрифтом. Для ввода исходных данных мы использовали специальный поток ввода pt, определенный в задачнике (см. п. 4.1.2). Этот поток позволяет вводить данные любых скалярных типов, в частности, int, double и char* (заметим, что в заданиях, входящих в задачник PT for MPI-2, используются данные только этих типов). Запустив полученную программу, мы увидим на экране окно задачника (рис. 8).

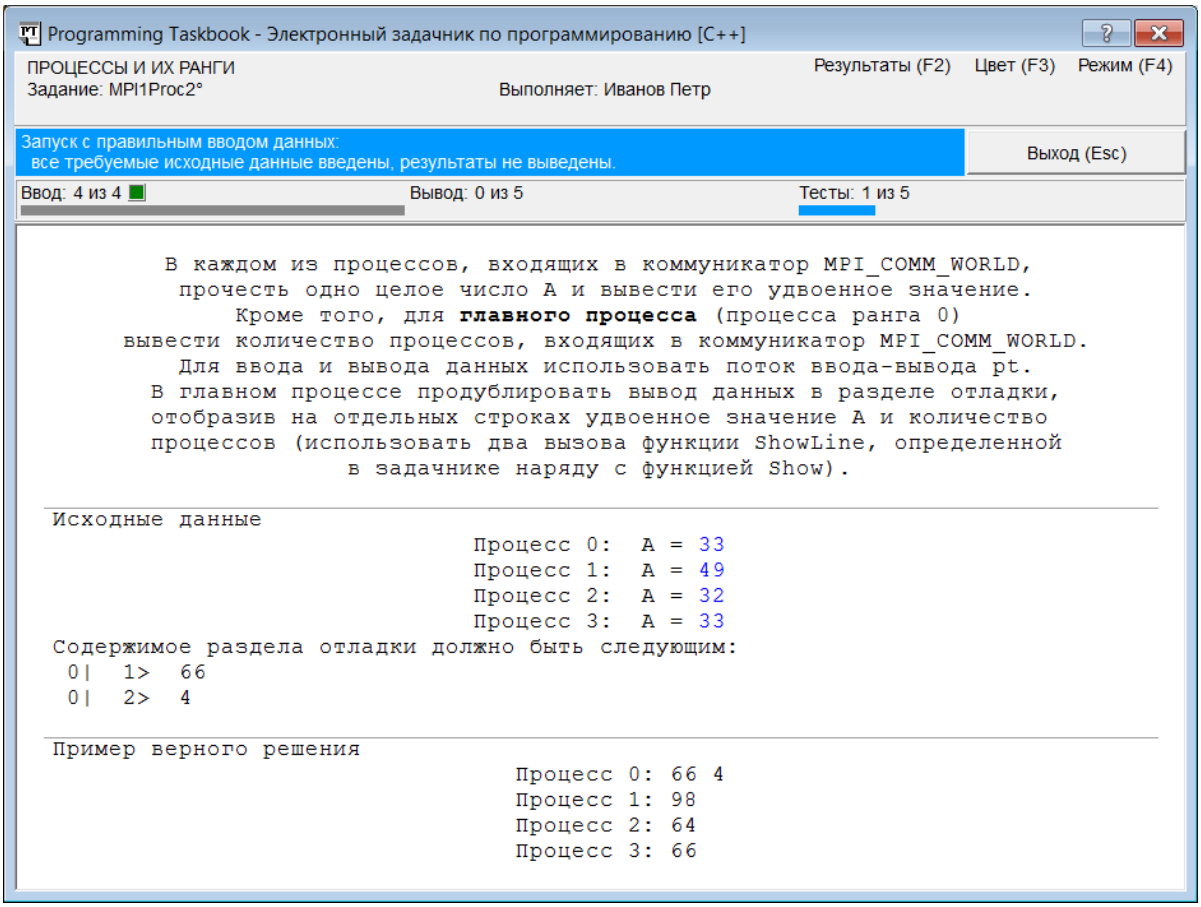

Рис. 8. Окно задачника с информацией о правильном вводе исходных данных

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 сделан: все исходные данные правильно введены. 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 различных запусках учебной число процессов изменяется; это позволяет протестировать решение для различного числа 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. 9).

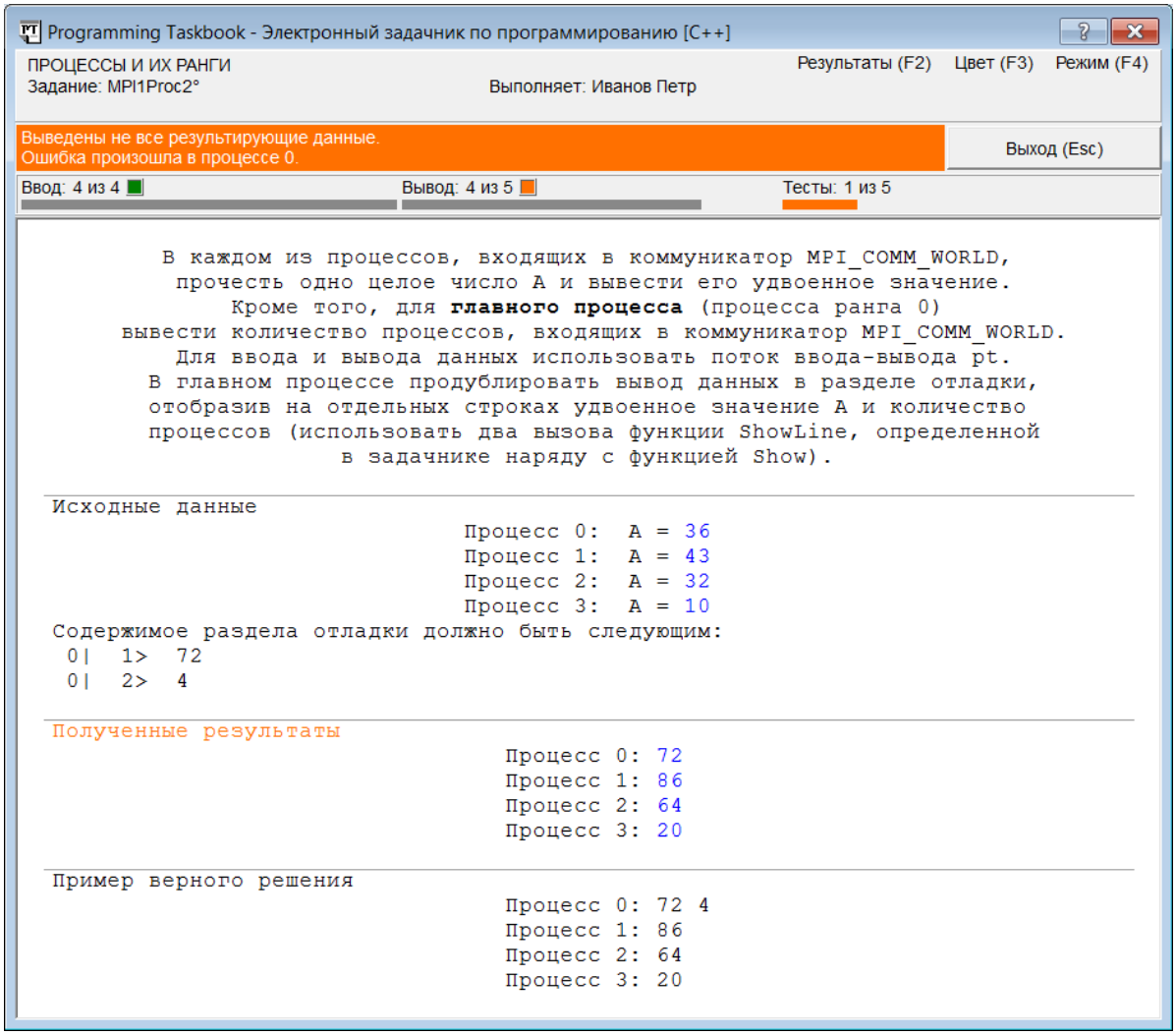

Рис. 9. Окно задачника с информацией об ошибке в главном процессе

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. 10.

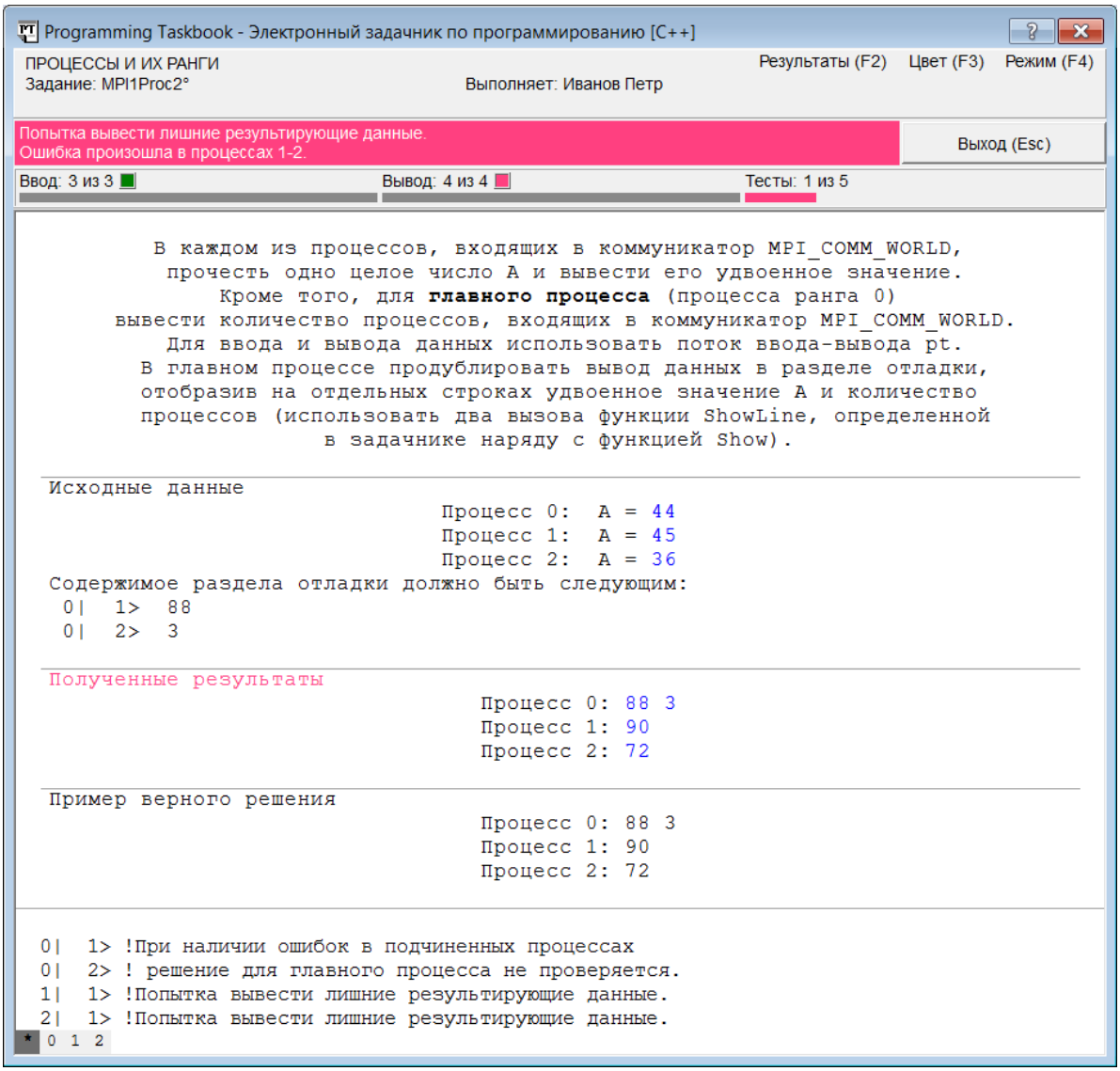

Рис. 10. Окно задачника с информацией о попытке вывода лишних данных

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 указываются 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. Учебная программа может выводить в раздел отладки свои собственные сообщения; об этой возможности будет подробно рассказано далее (см. также п. 4.1.4).

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 — см. рис. 10).

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 нового 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 ошибки, либо при успешном прохождении требуемого количества тестов (для всех заданий, входящих в задачник PT for MPI-2, количество тестовых испытаний 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 о том, что задание выполнено (рис. 11).

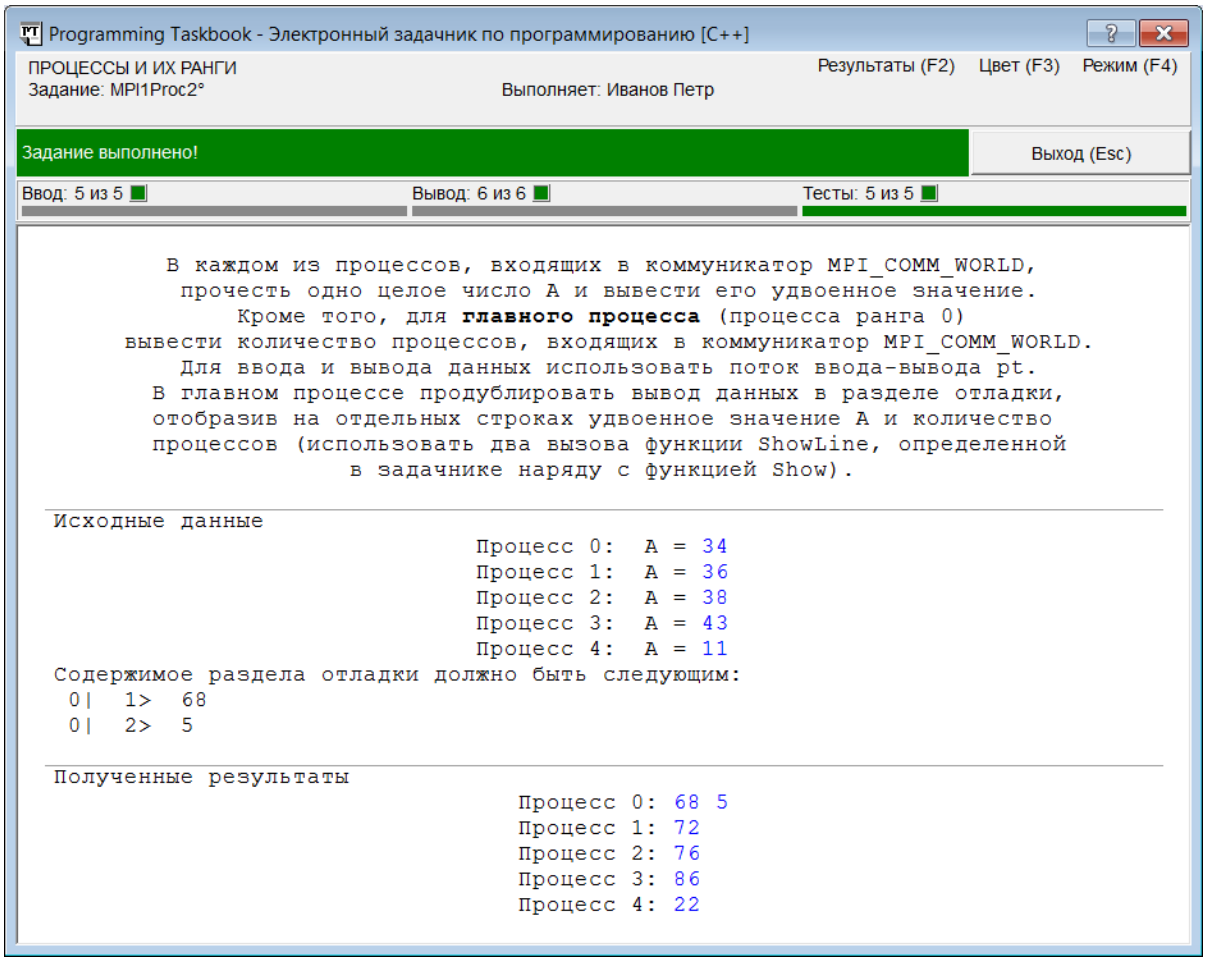

Рис. 11. Окно с сообщением об успешном выполнении задания MPI1Proc2

В данном случае все квадратные маркеры, 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 за-

дачника (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   c14/12 15:44 Ознакомительный запуск.
MPI1Proc2   c14/12 15:49 Запуск с правильным вводом данных.
MPI1Proc2   c14/12 15:54 Выведены не все результирующие данные.
MPI1Proc2   c14/12 15:59 Попытка вывести лишние результирующие данные.
MPI1Proc2   c14/12 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.

1.1.6. Использование дополнительной информации в разделе отладки

Если проанализировать полученное решение, то можно заметить, что оно все еще 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 ходе 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 интегрированной среды, 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, посвященных динамическому созданию процессов.

Напомним заключительную часть формулировки 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 внимание на то, что в окне задачника в разделе исходных данных 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. 4.1.4). 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 автоматический 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 два вызова функции ShowLine. 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);
  ShowLine(size);
}
```

После запуска нового варианта и проверки его на пяти тестовых наборах данных на экране появится окно задачника (рис. 12).

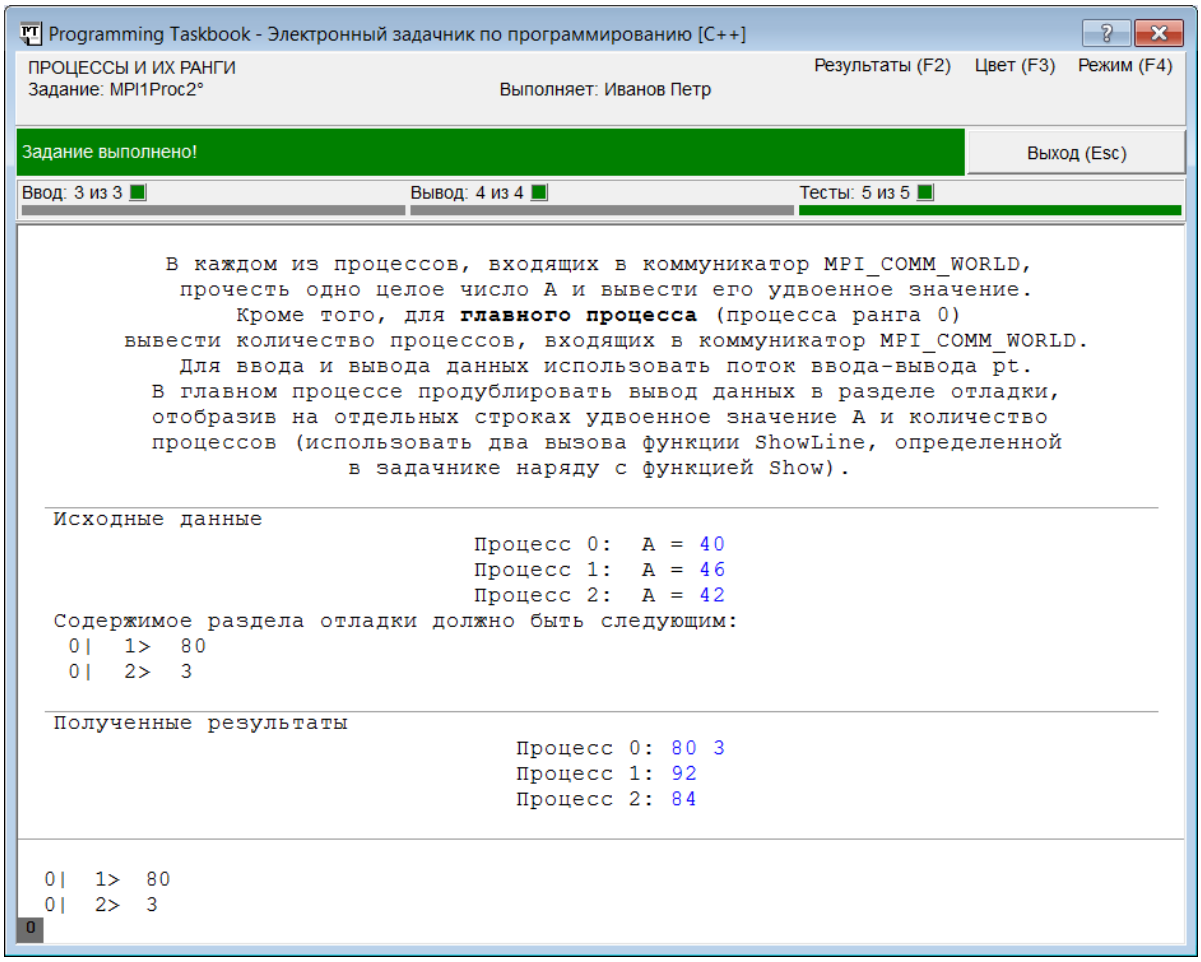

Рис. 12. Окно с полным вариантом решения

Сравнивая содержимое окна отладки с образцом, приведенным в разделе исходных данных, мы можем убедиться, 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 решили задачу 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, вывода результатов и отображения дополнительной информации в разделе отладки. Кроме того, мы увидели, как задачник реагирует на различные виды ошибок. Вся эта информация окажется полезной при решении задач, посвященных различным способам обмена сообщениями между процессами параллельного приложения.

1.2. Базовые возможности интерфейса MPI (стандарт MPI-1)

1.2.1. Блокирующая пересылка сообщений между двумя процессами: основные возможности

Библиотека MPI включает большое число функций, реализующих различные варианты пересылки данных между двумя процессами. Подобное взаимодействие процессов называется *взаимодействием вида «точка-точка»* (point-to-point communication).

Имеется два основных способа пересылки данных между двумя процессами: блокирующий (blocking) и неблокирующий (non-blocking).

При *блокирующей пересылке* выход из любой функции, связанной с операцией посылки или приема сообщения, происходит только после завершения этой операции. Для блокирующей отправки сообщений предусмотрены четыре функции:

MPI_Send – стандартный режим (standard mode);

MPI_Bsend – режим с буферизацией (buffered mode);

MPI_Ssend – синхронный режим (synchronous mode);

MPI_Rsend – режим «по готовности» (ready mode).

Все эти функции имеют одинаковый набор параметров, причем все параметры являются входными:

void* buf – буфер посылки сообщения;

int count – число передаваемых элементов в сообщении;

MPI_Datatype datatype – тип передаваемых элементов;

int dest – ранг процесса-получателя;

int msgtag – идентификатор сообщения (неотрицательное число, не превосходящее константы MPI_TAG_UB);

MPI_Comm comm – коммуникатор.

Заметим, что согласно стандарту MPI константа MPI_TAG_UB не может быть меньше 32767.

Здесь и далее при описании параметров особо не оговаривается тот факт, что параметр является входным; отмечается только ситуация, когда параметр является выходным или одновременно входным и выходным. Выходные параметры всегда передаются в виде *указателей* на переменную, содержимое которой изменяется.

Параметры datatype и comm имеют особые типы, определенные в библиотеке MPI. С типом MPI_Comm мы уже знакомы; этот тип имеют дескрипторы коммуникаторов. Тип MPI_Datatype предназначен для хранения информации о типе пересылаемых данных. Переменные этого типа являются дескрипторами, связанными либо со стандартными типами данных, включенными в библиотеку MPI, либо с пользовательскими типами, определенными с помощью соответствующих функций MPI (см. п. 1.2.6). При вы-

полнении заданий мы будем использовать стандартные типы MPI_INT (соответствует типу signed int языка C), MPI_DOUBLE (double) и MPI_CHAR (signed char). С другими числовыми типами языка C также связаны стандартные типы MPI, например, типу signed long int соответствует тип MPI_LONG, а типу float соответствует тип MPI_FLOAT. Тип MPI_BYTE соответствует *байту* — целому числу в диапазоне от 0 до 255. Имеются также стандартные составные типы, предназначенные для хранения пар чисел, например MPI_2INT и MPI_DOUBLE_INT (тип MPI_DOUBLE_INT будет использован при решении задачи MPI3Coll23 в п. 1.2.5)

Вернемся к режимам блокирующей отправки сообщений и опишем их основные особенности.

В *стандартном режиме* сама среда MPI определяет, будет ли использован специальный системый буфер (который в этом случае создается автоматически). Если используется системный буфер, то операция посылки сообщения завершается после пересылки данных в этот буфер независимо от того, начался или нет процесс приема этого сообщения в процессе-получателе (таким образом, в данном случае стандартный режим работает аналогично режиму с буферизацией). Если системный буфер не используется, то операция посылки сообщения завершается только после того, как процесс-получатель начал действия по приему этого сообщения (в этом случае стандартный режим работает аналогично синхронному режиму). Операция посылки в стандартном режиме является *нелокальной*, т. е. ее завершение может зависеть от действий другого процесса.

Перед использованием *режима с буферизацией* в процессе-отправителе необходимо определить пользовательский буфер достаточного размера (с помощью функции MPI_Buffer_attach). Операция посылки сообщения завершается после пересылки данных в этот буфер независимо от того, начался или нет прием этого сообщения в процессе-получателе, поэтому операция буферизованной посылки данных является локальной.

Операция посылки в *синхронном режиме* может начаться независимо от того, инициирован ли прием этого сообщения в принимающем процессе, однако завершится эта операция только после начала приема этого сообщения процессом-получателем. Данная операция является нелокальной.

В *режиме «по готовности»* операция посылки может начаться только при условии, что процесс-получатель уже инициировал прием данного сообщения (в противном случае операция посылки считается ошибочной и ее результат не определен). Данная операция является нелокальной и используется достаточно редко.

Для блокирующего приема сообщения предназначена функция MPI_Recv со следующими параметрами:

void* buf – буфер приема сообщения (выходной параметр);

int count – максимальное число элементов в принимаемом сообщении (или, иными словами, размер буфера buf в элементах принимаемого сообщения);

MPI_Datatype datatype – тип элементов принимаемого сообщения;

int source – ранг процесса-отправителя;

int msgtag – идентификатор принимаемого сообщения;

MPI_Comm comm – коммуникатор;

MPI_Status* status – дополнительная информация о принятом сообщении (выходной параметр).

Выход из функции MPI_Recv завершается только после заполнения буфера buf. В качестве параметров source и msgtag можно использовать специальные константы MPI_ANY_SOURCE и MPI_ANY_TAG, означающие, соответственно, что сообщение может быть принято от любого процесса или может иметь любой идентификатор.

Параметр status — последний параметр функции MPI_Recv — имеет структурный тип MPI_Status, все поля которого являются целочисленными. С помощью обращения к этим полям можно определить:

- ранг процесса, пославшего сообщение (поле MPI_SOURCE);
- идентификатор сообщения (поле MPI_TAG);
- код ошибки, связанной с данным сообщением (поле MPI_ERROR).

Кроме того, тип MPI_Status содержит дополнительное поле (в реализации MPICH имеющее имя count), позволяющее определить количество элементов в сообщении. Вместо непосредственного обращения к данному полю следует использовать функцию MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int* count) с входными параметрами status и datatype и выходным параметром count — количеством принятых элементов типа datatype.

Примечание. Если в программе не требуется использовать информацию, предоставляемую параметром status, то вместо него можно указывать константу MPI_STATUS_IGNORE, появившуюся в стандарте MPI-2. Отметим, что при использовании системы MPICH 1.2.5, реализующей стандарт MPI-1, *данную константу применять не следует* (несмотря на то, что в библиотеке MPICH 1.2.5 она присутствует).

Иногда бывает желательно получить дополнительную информацию об ожидаемом сообщении до его непосредственного приема функцией MPI_Recv (например, для того чтобы определить размер буфера buf, достаточный для хранения принятого сообщения). Для этого предназначена вспомогательная функция MPI_Probe(int source, int msgtag, MPI_Comm comm, MPI_Status* status), параметры которой имеют тот же смысл, что и одноименные параметры функции MPI_Recv. Данная функция, как и функция MPI_Recv, является блокирующей; выход из нее выполняется только после завершения получения данных от процесса-отправителя.

При организации посылки сообщения в режиме с буферизацией необходимо использовать вспомогательные функции MPI_Buffer_attach и MPI_Buffer_detach.

Функция MPI_Buffer_attach(void* buf, int size) позволяет определить буфер buf, используемый в дальнейшем при посылке сообщений в режиме с буферизацией. Размер буфера size задается в байтах и должен быть достаточным для хранения как пересылаемых сообщений, так и служебной информации. Размер памяти (в байтах), необходимый для размещения служебной информации, определяется константой MPI_BSEND_OVERHEAD. В каждый момент времени процесс может использовать только один буфер, причем *после его определения и до его освобождения к нему вообще не следует обращаться из программы*.

Функция MPI_Buffer_detach(void* buf, int* size) предназначена для освобождения ранее определенного буфера; оба ее параметра являются выходными (в параметре buf возвращается адрес начала освобожденного буфера, а в параметре size — его размер в байтах).

В библиотеке MPI предусмотрены функции MPI_Sendrecv и MPI_Sendrecv_replace для *совмещенных запросов на взаимодействие*, при вызове которых одновременно выполняется и посылка, и прием сообщений (причем не обязательно для одной и той же пары взаимодействующих процессов). Обе функции выполняют одновременную посылку и прием сообщений в стандартном режиме с блокировкой. Отличие состоит в том, что в функции MPI_Sendrecv_replace используется *единственный* буфер, в котором первоначально содержится посылаемое сообщение, а после выхода из данной функции — полученное сообщение (таким образом, этот буфер является входным и выходным параметром).

Приведем список параметров данных функций (фактически он представляет собой объединенный список параметров функций посылки и приема сообщения). Параметры функции MPI_Sendrecv:

void* sbuf – буфер посылки сообщения;

int scount – число передаваемых элементов в сообщении;

MPI_Datatype stype – тип передаваемых элементов;

int dest – ранг процесса-получателя;

int stag – идентификатор посылаемого сообщения;

void* rbuf – буфер приема сообщения (выходной параметр);

int rcount – максимальное число элементов в принимаемом сообщении;

MPI_Datatype rtype – тип принимаемых элементов;

int source – ранг процесса-отправителя;

int rtag – идентификатор принимаемого сообщения;

MPI_Comm comm – коммуникатор;

MPI_Status status – параметры принятого сообщения (выходной параметр).

Следует подчеркнуть, что для функции MPI_Sendrecv *нельзя* применять одинаковые (или даже просто перекрывающиеся) буферы sbuf и rbuf.

Количество параметров в функции MPI_Sendrecv_replace меньше, так как в данном случае буфер посылки и приема сообщения является общим, а следовательно, имеет и общие характеристики (размер и тип элементов):

void* buf – общий буфер посылки и приема сообщения (входной и выходной параметр);

int count – размер буфера посылки и приема сообщения (определяющий число элементов в передаваемом сообщении, а также максимальное число элементов в принимаемом сообщении);

MPI_Datatype datatype – тип передаваемых и принимаемых элементов;

int dest – ранг процесса-получателя;

int stag – идентификатор посылаемого сообщения;

int source – ранг процесса-отправителя;

int rtag – идентификатор принимаемого сообщения;

MPI_Comm comm – коммуникатор;

MPI_Status status – параметры принятого сообщения (выходной параметр).

В ряде случаев совмещенные запросы на взаимодействие позволяют избежать *взаимных блокировок* (см. следующий пункт), которые могли бы возникнуть при использовании отдельных запросов на посылку и прием сообщений.

Все возможности, связанные с блокирующей пересылкой данных, изучаются в первой подгруппе группы MPI2Send (см. п. 2.2.1).

1.2.2. Блокирующая пересылка сообщений между двумя процессами: примеры. Взаимные блокировки процессов

Чтобы познакомиться с особенностями функций, используемых при пересылке сообщений между отдельными процессами, рассмотрим одно из заданий группы MPI2Send.

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 рис. 13.

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 четыре вида функций (см. предыдущий пункт). 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 (описание параметров этих функций см. в предыдущем пункте).

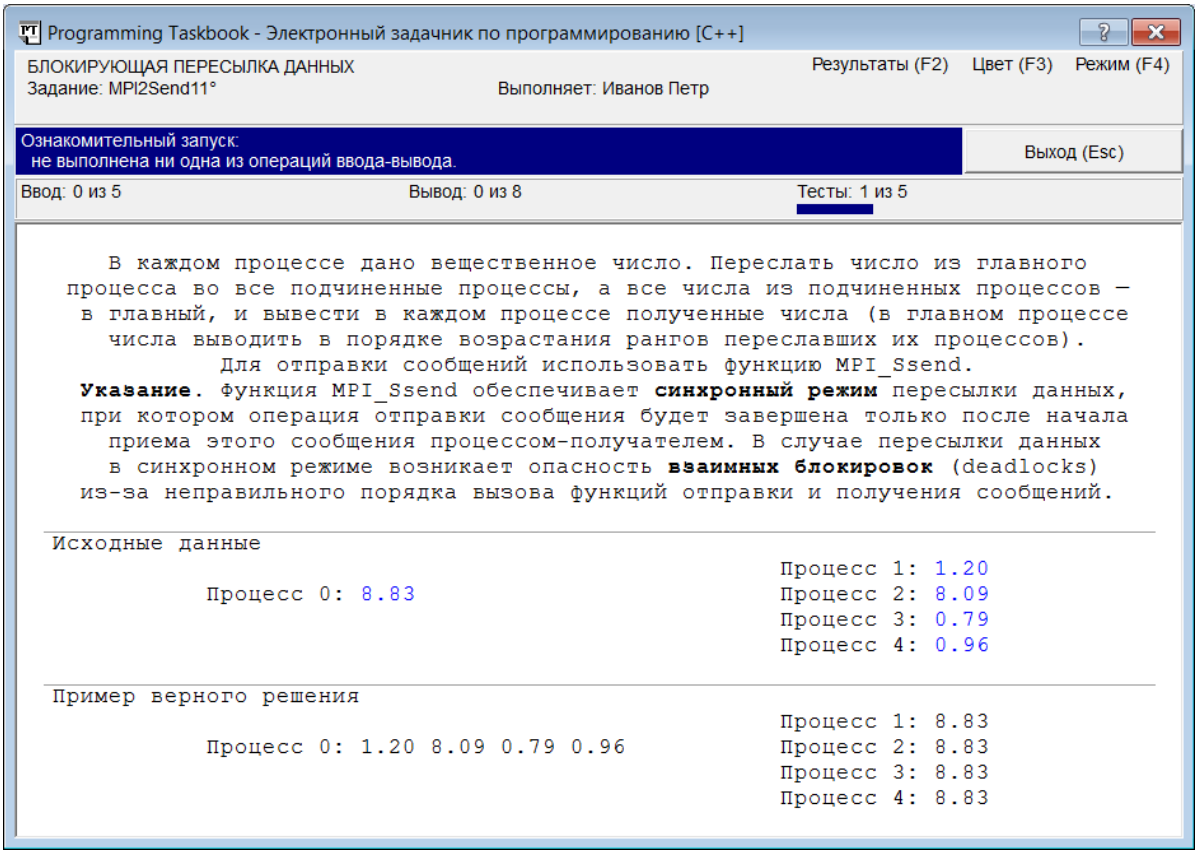

Рис. 13. Ознакомительный запуск задания MPI2Send11

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. 14).

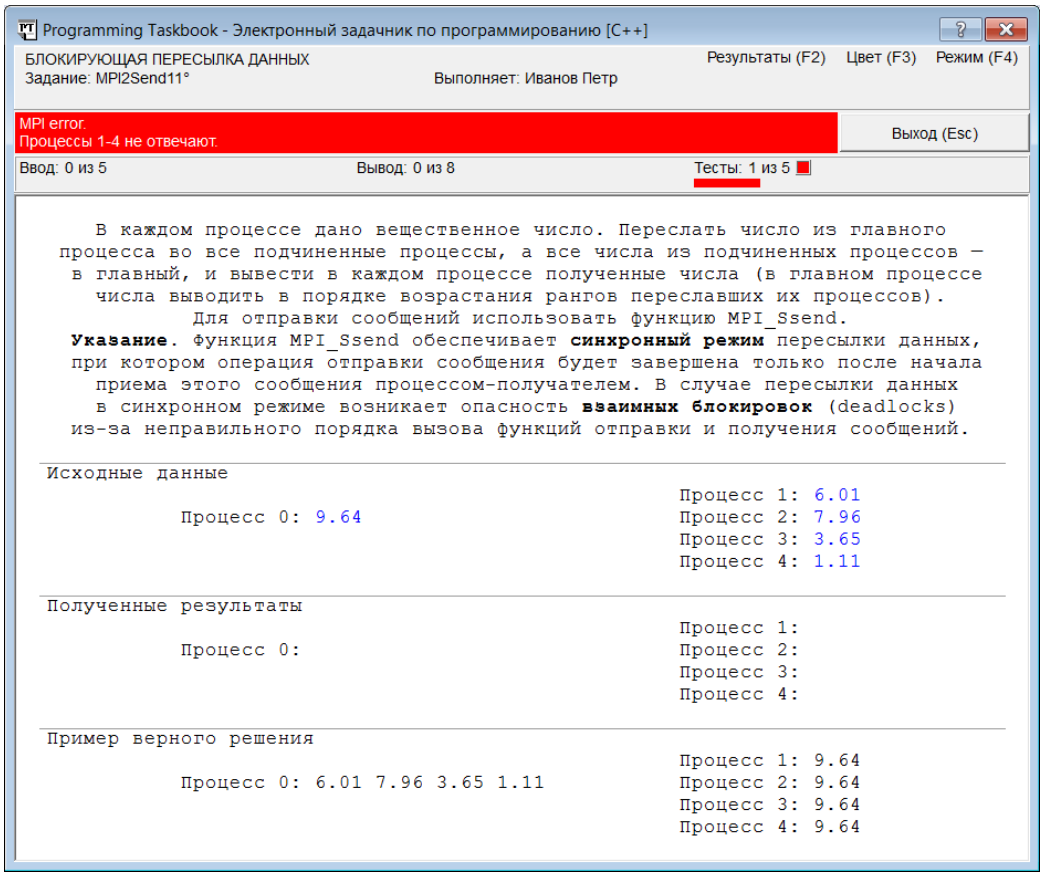

Рис. 14. Окно задачника с информацией о блокировке подчиненных процессов

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. 1.1.4).

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 и завершит свою работу (*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 (*режим пересылки с буферизацией*), 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 (см. примечание 2 в п. 1.1.4).

Итак, мы познакомились с ситуацией, когда один или несколько подчиненных процессов оказываются заблокированными. Подобная «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 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 секунд 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 клавиатурную комбинацию [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 пересылки данных*. Это связано с тем, что в библиотеке MPI системы MPICH стандартный режим, как и режим с буферизацией, использует буфер для хранения отправляемых данных (причем, в отличие от режима с буферизацией, буфер для стандартного режима создается автоматически). После пересылки данных в буфер функция MPI_Send завершает работу, даже если к этому моменту принимающий процесс не вызвал функцию MPI_Recv.

Опишем последовательность действий в этой ситуации, считая. 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 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 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, и на экране появится окно задачника с сообщением о том, что задание выполнено.

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 процес-

сов, *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 (измененный фрагмент кода выделен полужирным шрифтом):

```
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 не возникнет. 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 такой ситуации 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 специальную константу-«заглушку» MPI_STATUS_IGNORE (см. примечание в п. 1.2.1). Значение MPI_STATUS_IGNORE удобно использовать в ситуации, когда в программе не требуется обращаться к информации, предоставляемой параметром типа MPI_Status.

Заметим, что более эффективное решение этого задания можно получить, используя *коллективные операции пересылки данных* (см. п. 1.2.4).

1.2.3. Неблокирующая пересылка сообщений между двумя процессами. Отложенные запросы на взаимодействие. Функции замера времени

В данном пункте описываются возможности интерфейса MPI, связанные с неблокирующими и отложенными запросами. Этим возможностям посвящена вторая подгруппа группы MPI2Send (см. п. 2.2.2).

При неблокирующей пересылке сообщений операции посылки/приема сообщения лишь инициируют соответствующие действия, после чего немедленно завершаются, возвращая особый объект MPI — *запрос обмена* (request) типа MPI_Request, с помощью которого в дальнейшем можно проверить состояние данной операции, используя либо функции группы Wait,

блокирующие выполнение программы до полного завершения операции, либо неблокирующие функции группы Test. При завершении неблокирующей операции пересылки данных связанный с ней запрос обмена «сбрасывается», принимая значение MPI_REQUEST_NULL (это происходит либо при возврате из функции группы Wait, либо при таком вызове функции группы Test, при котором была возвращена информация о завершении операции).

Как и для блокирующей пересылки, имеются четыре функции неблокирующей посылки сообщений с идентичным набором параметров и одна функция неблокирующего приема сообщений. Имена этих функций совпадают с именами соответствующих функций для блокирующей посылки или приема сообщения, к которым добавлен префикс I (от слова immediate — немедленный): MPI_Isend, MPI_Ibsend, MPI_Issend, MPI_Irsend, MPIIrecv.

Функции MPI_Isend, MPI_Ibsend, MPI_Issend, MPI_Irsend инициируют неблокирующую передачу сообщения в одном из четырех возможных режимов (см. п. 1.2.1), возвращая связанный с этой передачей запрос обмена request типа MPI_Request. Выходной параметр request является последним параметром этих функций; все предшествующие параметры совпадают с параметрами функций блокирующей передачи сообщения: buf, count, datatype, dest, msgtag и comm. До сбрасывания запроса обмена request (т. е. до завершения неблокирующей операции) буфер buf *нельзя повторно использовать*.

Функция MPI_Irecv инициирует операцию приема сообщения без блокировки, возвращая связанный с ней запрос обмена request типа MPI_Request. Этот параметр является последним и располагается на месте параметра status функции MPI_Recv; все остальные параметры (buf, count, datatype, source, msgtag и comm) у этих функций совпадают. До сбрасывания запроса обмена request буфер buf *нельзя использовать для чтения полученных данных*.

В блокирующую группу Wait входят четыре функции:

MPI_Wait(MPI_Request* request, MPI_Status* status),

MPI_Waitall(int count, MPI_Request* requests, MPI_Status* statuses),

MPI_Waitany(int count, MPI_Request* requests, int* index, MPI_Status* status),

MPI_Waitsome(int count, MPI_Request* requests, int* outcount, int* indices, MPI_Status* statuses).

Функция MPI_Wait ожидает завершения неблокирующей операции передачи или приема сообщения, связанной с запросом обмена request (входной и выходной параметр), и возвращает выходной параметр status, который обычно используется только при неблокирующем приеме.

Все остальные функции принимают *массив* запросов обмена requests размера count.

Функция MPI_Waitall блокирует выполнение процесса, пока *все* операции обмена, ассоциированные с указанными запросами requests, не будут завершены (в параметре statuses размера count возвращается массив элемен-

тов типа MPI_Status с дополнительной информацией о каждой из завершенных операций).

Функция MPI_Waitany блокирует выполнение процесса, пока *какая-либо* операция обмена, ассоциированная с указанными запросами requests, не будет завершена. При этом в параметре index возвращается индекс завершенной операции, а в параметре status — дополнительная информация об этой операции (все прочие запросы обмена в массиве requests не изменяются).

Наконец, функция MPI_Waitsome блокирует выполнение процесса, пока *хотя бы одна* из операций обмена, ассоциированных с указанными запросами requests, не будет завершена. В отличие от функции MPI_Waitany данная функция может возвращать информацию о *нескольких* завершенных операциях: количество завершенных операций возвращается в параметре outcount, индексы завершенных операций — в первых outcount элементах массива indices, а дополнительная информация о завершенных операциях — в первых outcount элементах массива statuses.

В неблокирующую группу Test также входят четыре функции:

MPI_Test(MPI_Request* request, int* flag, MPI_Status* status),

MPI_Testall(int count, MPI_Request* requests, int* flag, MPI_Status* statuses),

MPI_Testany(int count, MPI_Request* requests, int* index, int* flag, MPI_Status* status),

MPI_Testsome(int count, MPI_Request* requests, int* outcount, int* indices, MPI_Status* statuses).

Функция MPI_Test проверяет завершенность неблокирующей операции передачи или приема, связанной с запросом request, и *немедленно завершает работу*, возвращая результат проверки в выходном параметре flag. Если операция завершена, то параметр flag возвращает ненулевое значение (при этом значение запроса обмена сбрасывается в MPI_REQUEST_NULL, а в параметре status возвращается дополнительная информация о завершенной операции), в противном случае в параметре flag возвращается 0 (в этом случае параметры request и status не изменяются). Остальные функции ведут себя так же как и функция MPI_Test, т. е. проверяют завершенность неблокирующих операций (в данном случае связанных с *массивом* запросов requests) и немедленно завершают работу, возвращая результат в параметре flag (или, для функции MPI_Testsome, в параметре outcount). Назначение остальных параметров этих функций аналогично назначению параметров соответствующих функций группы Wait.

Имеется также неблокирующий вариант функции MPI_Probe:

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

Этот вариант отличается от блокирующего варианта наличием выходного параметра flag. Функция MPI_Iprobe не ожидает завершения операции приема сообщения. Если прием не завершен, то в параметре flag возвращается нулевое значение (в этом случае параметр status использовать не следует). Данная функция, как и ее блокирующий вариант MPI_Probe, обычно

используется в ситуации, когда число элементов в переданном сообщении заранее неизвестно.

Особым видом неблокирующих операций являются отложенные запросы. Данные запросы формируются с помощью функций MPI_Send_init, MPI_Bsend_init, MPI_Ssend_init, MPI_Rsend_init, MPI_Recv_init, имеющих те же параметры, что и рассмотренные ранее неблокирующие функции MPI_Isend, MPI_Ibsend, MPI_Issend, MPI_Irsend, MPI_Irecv. Однако в отличие от «обычных» неблокирующих функций функции группы Init не приводят к немедленному выполнению соответствующей операции; они лишь возвращают *отложенный запрос* — параметр request типа MPI_Request, содержащий все настройки требуемой операции.

Для запуска отложенных запросов, сформированных с помощью функций группы Init, предусмотрены функции MPI_Start(MPI_Request* request) и MPI_Startall(int count, MPI_Request* requests). Первая из них запускает в неблокирующем режиме операцию обмена, связанную с запросом request (входной и выходной параметр), а вторая запускает в неблокирующем режиме все операции обмена, связанные с *массивом* запросов requests размера count. В дальнейшем для проверки завершенности запущенных операций надо использовать ранее рассмотренные функции групп Wait и Test.

Запрос, возвращаемый функциями группы Init, имеет одну важную особенность: он является *устойчивым* (persistent). Это означает, что после завершения соответствующей неблокирующей операции значение связанного с ней запроса не сбрасывается в MPI_REQUEST_NULL, а остается «действительным». Поэтому в дальнейшем отложенный запрос *можно использовать повторно*, еще раз вызвав для него стартовую функцию MPI_Start или MPI_Startall (разумеется, перед этим можно изменить содержимое буфера buf, содержащего посылаемые данные). Для сброса запроса request, сформированного одной из функций группы Init, надо использовать функцию MPI_Request_free(MPI_Request* request).

В некоторых заданиях, связанных с неблокирующими операциями (см. п. 2.2.2), требуется использовать специальную функцию MPI, предназначенную для замера времени: double MPI_Wtime(). Эта функция является одной из двух особых функций MPI, которые возвращают не информацию об успешности своего запуска, а сам результат, в данном случае — время в секундах, прошедшее с некоторого момента в прошлом. Таким образом, для определения продолжительности выполнения некоторого фрагмента программы достаточно вызвать данную функцию в начале и в конце этого фрагмента, после чего найти разность полученных значений. Вторая особая функция MPI тоже связана с измерением времени: это функция double MPI_Wtick(), которая возвращает длительность в секундах между последовательными тактами срабатывания таймера и, таким образом, характеризует *точность* замера времени.

1.2.4. Коллективные операции

Большая группа функций MPI предназначена для организации *коллективного взаимодействия процессов*. «Коллективные» MPI-функции, в отличие от ранее рассмотренных функций MPI_Send, MPI_Recv и т. п., 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.

Использование коллективных функций обменя является более предпочтительным, чем многократный вызов отдельных операций вида «точка-точка», что обусловлено двумя обстоятельствами. Во-первых, при реализации коллективных функций в библиотеке MPI используются эффективные алгоритмы и, во-вторых, в суперкомпьютерных или кластерных системах коллективные операции обмена могут быть реализованы на аппаратном уровне, что может быть учтено при разработке библиотек MPI для этих систем.

Все операции, связанные с коллективным взаимодействием процессов, выполняются в стандартном режиме с блокировкой. Для успешного выполнения коллективной операции необходимо, чтобы соответствующая функция была вызывана во *всех* процессах того коммуникатора, для которого выполняется эта коллективная операция.

Если с коллективной операцией связан процесс, играющий особую роль, то в соответствующей функции присутствует параметр root, содержащий ранг этого процесса (и, кроме того, некоторые параметры этой функции будут использоваться только в процессе ранга root). Если в коллективной функции отсутствует параметр root, то это означает, что все процессы при выполнении коллективной операции являются равноправными.

Простейшей коллективной функцией с равноправными процессами является MPI_Barrier(MPI_Comm comm). Она блокирует работу вызвавших ее процессов до тех пор, пока все оставшиеся процессы коммуникатора comm также не вызовут эту функцию. Таким образом, функция MPI_Barrier позволяет выполнить *синхронизацию* процессов параллельного приложения.

Самой простой коллективной функцией с выделенным процессом root является MPI_Bcast(void* buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm). Данная функция рассылает данные из процесса root всем процессам коммуникатора comm. Параметр buf определяет буфер рассылки/приема сообщения; в процессе root этот параметр является входным, а в остальных процессах выходным. Прочие параметры являются входными во всех процессах: count определяет число пересылаемых элементов, а datatype — их тип.

Пример:

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

Функция MPI_Gather *собирает* данные из всех процессов коммуникатора в буфере процесса-приемника root, причем от каждого процесса принимается одинаковое число элементов данных. Ее параметры:

void* sbuf – буфер рассылки;

int scount – число элементов в посылаемом сообщении;

MPI_Datatype stype – тип элементов посылаемого сообщения;

void* rbuf – буфер сборки данных (это выходной параметр, который используется только в процессе root);

int rcount – число элементов, принимаемых от каждого процесса (этот и следующий параметр также используется только в процессе root; необходимо подчеркнуть, что данный параметр *не равен* размеру буфера rbuf);

MPI_Datatype rtype – тип элементов принимаемого сообщения;

int root – ранг процесса, выполняющего прием данных;

MPI_Comm comm – коммуникатор.

Подчеркнем, что процесс root принимает и свои собственные данные.

Пример (scount = 2, rcount = 2):

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

Более сложным вариантом функции MPI_Gather является функция MPI_Gatherv, позволяющая принимать от каждого процесса различное число элементов данных. Эта функция имеет следующий набор параметров:

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

В данном случае параметры scount могут иметь разное значение в разных процессах, а вместо целочисленного параметра rcount используется массив rcounts, в котором указывается число элементов, принимаемых от каждого процесса. Дополнительную гибкость данной функции придает наличие параметра displs — массива *смещений* (в элементах) от начала буфера приема данных rbuf. Данные, полученные от каждого процесса, записываются в буфер процесса-приемника со смещением, определяемым соответствующим элементом массива displs. Параметр displs учитывается только в процессе root; смещения могут быть как положительными, так и отрицательными.

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

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

«Обратной» по отношению к операции gather является операция scatter, выполняющая *рассылку* данных из выделенного процесса root во все процессы данного коммуникатора. Данная операция также реализована в виде двух функций: MPI_Scatter и MPI_Scatterv. Параметры функции MPI_Scatter:

`void* sbuf` – буфер рассылки (этот и два следующих параметра используются только в процессе root);

`int scount` – число элементов, посылаемых каждому процессу (подчеркнем, что этот параметр *не равен* размеру буфера sbuf);

`MPI_Datatype stype` – тип элементов рассылаемого сообщения;

`void* rbuf` – буфер приема данных (выходной параметр);

`int rcount` – число элементов в буфере приема данных;

`MPI_Datatype rtype` – тип элементов принимаемого сообщения;

`int root` – ранг процесса, выполняющего рассылку данных;

`MPI_Comm comm` – коммуникатор.

Пример (scount = 2, rcount = 2):

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

Функция MPI_Scatterv имеет следующие параметры:

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

В данном случае параметрами-массивами являются scounts (массив, определяющей число элементов, посылаемых каждому процессу) и displs (массив смещений (в элементах) от начала буфера рассылки).

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

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

Операция gather имеет модификацию, в которой результат объединения рассылается во *все* процессы. Данная операция реализована в виде функций MPI_Allgather и MPI_Allgatherv. В этих функциях отсутствует параметр root, поскольку все процессы являются равноправными: они предоставляют

свою часть данных и получают одинаковый объединенный набор. Приведем список параметров этих функций и пример их работы:

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

Пример (scount = 2, rcount = 2):

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

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

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

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

Наиболее сложной коллективной операцией является операция all-to-all, в которой каждый процесс рассылает *различные* данные всем процессам коммуникатора. В функции MPI_Alltoall выполняется пересылка *одинакового* количества данных от каждого процесса всем процессам:

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

Пример (scount = 2, rcount = 2):

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

Функция MPI_Alltoallv обеспечивает рассылку каждым процессом различных данных всем другим процессам, причем каждый процесс может принимать *различное* число элементов данных от разных процессов. Данные, посылаемые каждому процессу, должны размещаться в буфере процессов-источников со смещением, определяемым соответствующим элементом массива sdispls. Данные, полученные от каждого процесса, записываются в буфер процессов-приемников со смещением, определяемым соответствующим элементом массива rdispls:

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

Пример:

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

Описанные выше коллективные функции (за исключением функции MPI_Barrier) рассматриваются в задачах первой подгруппы группы MPI3Coll (см. п. 2.3.1). Кроме того, они активно применяются и в последующих группах заданий. В частности, при решении задачи MPI5Comm3 (см. п. 1.2.7) используется функция MPI_Gather, а при решении задачи MPI5Comm17 (см. п. 1.2.8) используется функция MPI_Scatter. Функция MPI_Barrier используется в завершающих задачах групп MPI7Win и MPI8Inter (п. 2.7.2 и 2.8.3).

Во всех коллективных функциях стандарта MPI-1, содержащих массивы смещений displs, эти смещения задаются в *элементах* пересылаемых данных. Однако в некоторых ситуациях, связанных с пересылкой данных сложных типов, желательно указывать смещения не в элементах, а в *байтах*. Поэтому в стандарте MPI-2 набор коллективных функций был дополнен функцией MPI_Alltoallw, выполняющей то же действие, что и функция MPI_Alltoallv, но позволяющей задавать смещения в байтах. Кроме того, в этом варианте коллективной функции каждый процесс может посылать различным процессам данные *разных типов*:

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

С помощью функции MPI_Alltoallw можно реализовать варианты операций gather и scatter, при которых смещения указываются в байтах, а также собираются или рассылаются данные разных типов. Этой функции посвящена особая подгруппа группы MPI4Type (см. п. 2.4.4). Включение задач на функцию MPI_Alltoallw в раздел, посвященный производным типам, обусловлено тем, что эта функция предназначена, прежде всего, для коллективной пересылки сложных типов данных. В дальнейшем эта функция используется в подгруппе группы MPI9Matr, посвященной блочному алгоритму Фокса перемножения матриц (см. п. 2.9.5).

1.2.5. Операции редукции. Использование составных типов данных

В набор коллективных функций MPI входит группа функций, обеспечивающих выполнение *операций редукции*, т. е. операций, связанных с пересылкой не исходных данных, а результатов их обработки некоторой *групповой операцией* типа MPI_Op. Чаще всего используются такие операции, как нахождение суммы MPI_SUM, произведения MPI_PROD, максимального MPI_MAX или минимального MPI_MIN значения. Предусмотрены также логические операции MPI_LAND, MPI_LOR, MPI_LXOR и их поразрядные аналоги MPI_BAND, MPI_BOR, MPI_BXOR. 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, содержащего этот экстремальный элемент).

Пользователь может определить новую операцию редукции op; для этого предусмотрена функция MPI_Op_create(MPI_User_function* function, int commute, MPI_Op* op). Первый параметр function является указателем на функцию, в которой определяется новая операция. Определяемая операция обязательно должна быть ассоциативной. Если она является также и коммутативной, то параметр-флаг commute должен быть отличен от нуля. Прототип функции function имеет следующий вид:

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

Параметры invec и inoutvec являются указателями на массивы, содержащие len элементов типа datatype. Элементы массивов invec[i] и inoutvec[i], i = 0, …, len – 1, считаются соответственно левым и правым операндом определяемой пользовательской операции; результат применения этой операции к элементам invec[i] и inoutvec[i] должен сохраняться в элементе inoutvec[i]. Таким образом, массив invec является входным, а массив inoutvec — одновременно входным и выходным (что и объясняет выбор их имен).

Если пользовательскую операцию предполагается применять только к данным фиксированного типа, то при ее определении можно считать, что массивы invec и inoutvec имеют требуемый тип, и не анализировать параметр datatype.

В стандарте MPI-1 определены четыре функции, выполняющие операции редукции: MPI_Reduce, MPI_Allreduce, MPI_Reduce_scatter и MPI_Scan.

Функция MPI_Reduce выполняет глобальную операцию с возвратом результатов в указанный процесс-приемник root. Параметры данной функции:

void* sbuf – буфер для аргументов;

void* rbuf – буфер для результата (выходной параметр, который используется только в процессе root);

int count – число аргументов у каждого процесса;

MPI_Datatype – тип аргументов;

MPI_Op op – идентификатор операции;

int root – ранг процесса, выполняющего прием данных;

MPI_Comm comm – коммуникатор.

Пример (count = 3):

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

Функция MPI_Allreduce является вариантом функции MPI_Reduce, в котором результат глобальной операции возвращается во *все* процессы. Поэтому в функции MPI_Allreduce отсутствует параметр root, а выходной параметр rbuf используется во *всех* процессах коммуникатора comm:

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

Пример:

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

Другой вариант функции MPI_Reduce — функция MPI_Reduce_scatter — также не содержит параметр root. В отличие от функции MPI_Allreduce она не пересылает во все процессы полный набор результатов глобальной операции, а выполняет распределение полученных результатов по процессам, причем каждый процесс может принимать различное число результирующих элементов:

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

В данном случае третий параметр rcounts является *массивом*, в котором задается число результирующих элементов, посылаемых каждому процессу (заметим, что сумма значений элементов массива rcounts определяет также размер буфера sbuf в каждом процессе).

Пример (rcounts = {1, 3, 2}):

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

Наконец, функция MPI_Scan выполняет последовательность *частичных* глобальных операций: в *i*-й процесс коммуникатора посылается результат выполнения глобальной операции для процессов от нулевого до *i*-го включительно. Данная функция имеет такой же набор параметров, что и функция MPI_Allreduce.

Пример:

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

Примечание. В стандарте MPI-2 набор функций, связанных с операциями редукции, был расширен. Опишем одну из новых функций — MPI_Reduce_scatter_block, — поскольку она используется в задачах группы MPI8Inter (см. п. 2.8.3). Функция MPI_Reduce_scatter_block(void* sbuf, void* rbuf, int rcount, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm) является упрощенным вариантом функции MPI_Reduce_scatter, в котором каждому процессу посылается блок результирующих данных одного и того же размера rcount (в отличие от массива rcounts, используемого в функ-

ции MPI_Reduce_scatter). Значение rcount определяет размер буфера для результата rbuf. Буфер для аргументов sbuf должен содержать K * rcount элементов, где K — количество процессов в коммуникаторе comm.

Коллективным операциям редукции посвящена вторая подгруппа группы MPI3Coll (см. п. 2.3.2). Рассмотрим одно из входящих в нее заданий, при выполнении которого мы также познакомимся 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.

При запуске программы-заготовки, созданной для выполнения задания MPI3Coll23, мы увидим на экране окно задачника, подобное приведенному на рис. 15.

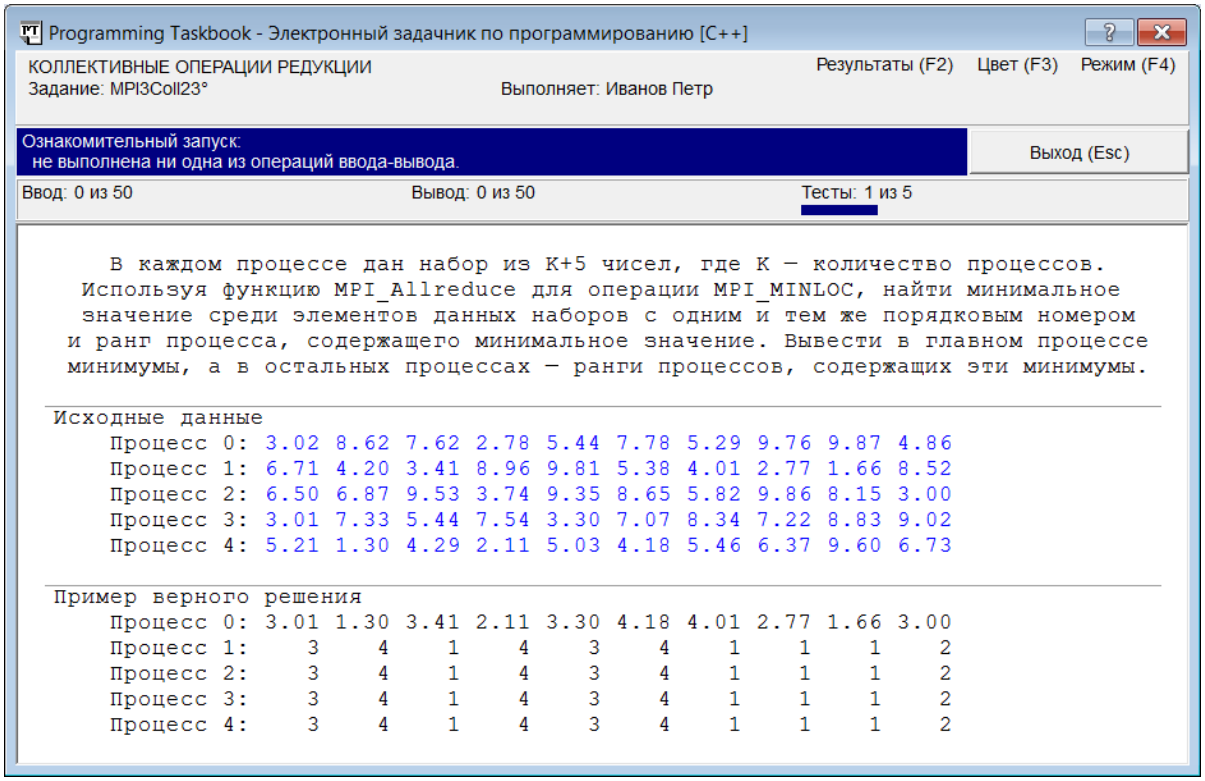

Рис. 15. Ознакомительный запуск задания MPI3Coll23

Каждый процесс предоставляет массив чисел одинакового размера, а операция редукции применяется по отдельности к элементам предоставленных массивов с одним и тем же индексом; в итоге мы получаем массив того же размера, каждый элемент которого является результатом применения операции редукции к элементам исходных массивов с этим же индексом.

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 применять 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, а также контейнеров vector<T> 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;
}
```

Запустив этот вариант программы, мы получим сообщение о том, что все исходные данные успешно введены (рис. 16).

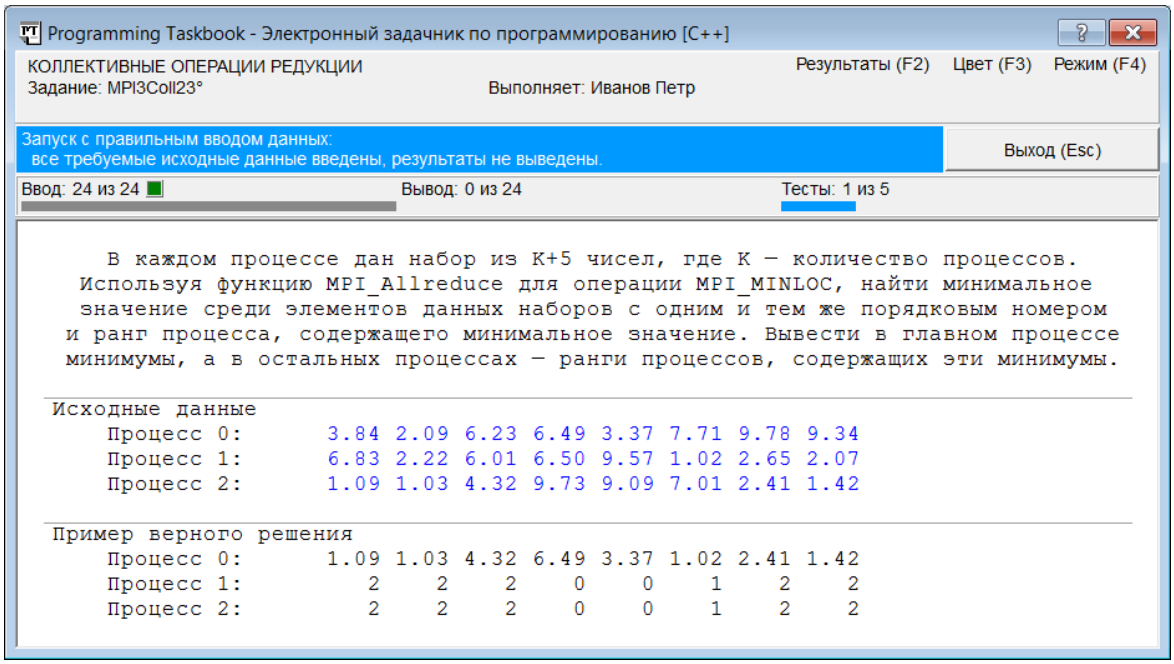

Рис. 16. Окно задачника с информацией об успешном вводе исходных данных

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 функции MPI 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 — вещественного 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;
}
```

1.2.6. Определение пользовательских типов и упаковка данных. Использование динамических массивов и контейнеров vector

В библиотеке MPI предусмотрен большой набор функций для определения новых типов (производных типов, derived datatypes), использование которых позволяет упростить и ускорить действия по пересылке сложных данных. Примерами сложных данных являются структуры, состоящие из *полей разного типа*, а также фрагменты многомерных массивов, *имею-*

щие «пустые» промежутки (например, какой-либо столбец двумерной матрицы). Для того чтобы при определении нового типа можно было учесть обе приведенные особенности, с новым типом связываются два набора характеристик: последовательность базовых типов и последовательность *смещений* (displacements). Таким образом, новый тип может содержать элементы различных базовых типов и, кроме того, эти элементы могут располагаться не подряд, а с некоторыми смещениями друг относительно друга (смещения могут быть как положительными, так и отрицательными). В качестве базовых можно использовать не только стандартные типы MPI (например, MPI_INT или MPI_DOUBLE), но и ранее определенные производные типы.

Наиболее простой из функций MPI, позволяющих определить новые типы, является функция MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype* newtype), которая создает производный тип newtype, состоящий из count последовательно расположенных элементов базового типа oldtype. В этой и всех последующих функциях для определения нового типа единственным выходным параметром является последний параметр — ссылка на созданный тип.

Пример (count = 5):

```
Исходный тип: [T1]
Производный тип: [T1][T1][T1][T1][T1]
```

Типы, созданные с помощью функции MPI_Type_contiguous, обычно используются в качестве «строительных блоков» при определении более сложных типов.

Более полезные возможности предоставляет функция MPI_Type_vector(int count, int blocklen, int stride, MPI_Datatype oldtype, MPI_Datatype* newtype), которая создает производный тип newtype, состоящий из count *блоков*, каждый из которых содержит одинаковое число blocklen элементов базового типа oldtype и находится на одном и том же расстоянии stride от начала предыдущего блока (расстояние указывается в количестве элементов базового типа).

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

```
Исходный тип: [T1]
Участок памяти, равный протяженности исходного типа: [..]
Производный тип: [T1][T1][..][..][..][T1][T1][..][..][..][T1][T1]
```

Тип, приведенный в примере, может интерпретироваться как *два соседних столбца* матрицы размера 5 на 3, причем необязательно два *первых* столбца. Если, например, в качестве начального адреса указать позицию *третьего* элемента в первой строке матрицы, то данный тип будет содержать элементы третьего и четвертого столбца.

Если блоки в новом типе имеют разный размер или между ними должны быть различные расстояния, то следует использовать более слож-

ную функцию MPI_Type_indexed с параметрами-массивами: MPI_Type_indexed(int count, int* blocklens, int* displs, MPI_Datatype oldtype, MPI_Datatype* newtype). Данная функция создает производный тип newtype, состоящий из count блоков, каждый из которых может содержать различное число элементов базового типа oldtype и размещается на указанном расстоянии от начальной позиции определяемого типа. Количество элементов для различных блоков задается в массиве blocklens размера count, а расстояния измеряются в элементах базового типа и содержатся в массиве displs размера count.

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

```
Исходный тип: [T1]
Участок памяти, равный протяженности исходного типа: [..]
Производный тип:
[T1][T1][..][T1][T1][T1][..][..][T1][..][..][..][T1][T1]
```

Обратите внимание на то, что в функции MPI_Type_vector указывается расстояние между началами *соседних* блоков, а в функции MPI_Type_indexed указывается массив расстояний от *начальной позиции* производного типа.

Наиболее гибкой из функций определения новых типов является MPI_Type_struct с параметрами (int count, int* blocklens, MPI_Aint* displs, MPI_Datatype* oldtypes, MPI_Datatype* newtype). В стандарте MPI-2 имя этой функции было изменено на MPI_Type_create_struct. Данная функция имеет два отличия от функции MPI_Type_indexed: во-первых, массив смещений displs от начальной позиции определяемого типа содержит смещения в *байтах*, а во-вторых, для каждого блока указывается *свой базовый тип* (базовые типы задаются в массиве oldtypes). Элементы массива смещений displs имеют тип MPI_Aint; этот тип предназначен для хранения смещений между различными *адресами* в памяти и реализован как знаковый целочисленный тип, размер которого является достаточным для хранения любого возможного смещения в адресном пространстве.

Пример:

```
Исходные типы: [T1], [T2], [T3], [T4]
Участок памяти, равный 1 байту (обозначается точкой): .
Производный тип: [T1][T1].[T2][T2][T2]...[T3].....[T4][T4]
```

Заметим, что в библиотеке MPI предусмотрены варианты функций MPI_Type_vector и MPI_Type_indexed, для которых смещения также задаются не в элементах, а в байтах (и имеют тип MPI_Aint). В стандарте MPI-1 эти функции имеют имена MPI_Type_hvector и MPI_Type_hindexed, а в стандарте MPI-2 — имена MPI_Type_create_hvector и MPI_Type_create_hindexed.

В стандарте MPI-2 набор функций для определения новых типов был расширен. Не описывая все добавленные функции, отметим одну из них, которая занимает промежуточное положение между MPI_Type_vector и MPI_Type_indexed. Это функция MPI_Type_create_indexed_block(int count, int

blocklen, int* displs, MPI_Datatype oldtype, MPI_Datatype* newtype). Она позволяет определить производный тип newtype, состоящий из count блоков, каждый из которых состоит из blocklen элементов базового типа oldtype и находится на указанном расстоянии от начальной позиции определяемого типа (расстояния указываются в количестве элементов базового типа и содержатся в массиве displs размера count). Отличие данной функции от функции MPI_Type_indexed состоит в том, что в определяемом ею типе все блоки имеют *одинаковый размер*, и поэтому для его задания используется не массив, а скалярный параметр blocklen целого типа (как и в функции MPI_Type_vector).

Все описанные функции являются локальными, т. е. могут вызываться только в некоторых процессах параллельного приложения — в тех, которые в дальнейшем применяют новые типы (для определения других новых типов или посылки/приема данных).

Если новый тип требуется использовать при пересылке сообщений между процессами, то его необходимо дополнительно *зарегистрировать*, вызвав для него функцию MPI_Type_commit(MPI_Datatype* datatype). В данном случае параметр datatype является одновременно входным и выходным. Незарегистрированные типы можно использовать при определении новых типов, но при пересылке данных их применять нельзя.

Примечание 1. Отсутствие вызова функции MPI_Type_commit не препятствует выполнению учебных заданий на локальном компьютере при использовании системы MPICH 1.2.5. Однако в случае системы MPICH2 1.3 попытка указать незарегистрированный тип при пересылке данных приводит к ошибке.

Пользовательский тип данных можно уничтожить, освободив связанный с ним дескриптор (типа MPI_Datatype). Для этого предназначена функция MPI_Type_free(MPI_Datatype* datatype), в которой параметр datatype является одновременно входным и выходным. После вызова данной функции в ее параметре возвращается значение MPI_DATATYPE_NULL. Следует подчеркнуть, что производные типы, определенные *с помощью* данного пользовательского типа, сохраняются и после его уничтожения.

Двумя основными характеристиками типа MPI являются *протяженность* (extent) и *размер* (size). Протяженность равна числу байт, которые этот тип занимает в памяти (включая все пустые промежутки между его блоками). Размер равен суммарному размеру (в байтах) всех блоков *без учета промежутков между ними*. Если протяженность характеризует количество оперативной памяти, выделяемой для *хранения* элемента данного типа, то размер определяет количество байт, используемое при *пересылке* элемента данного типа между процессами (поскольку пустые промежутки в формируемое сообщение не включаются). У стандартных типов MPI (MPI_INT, MPI_DOUBLE, MPI_CHAR и т. д.) протяженность и размер совпадают.

Для определения протяженности и размера в стандарте MPI-1 предусмотрены две функции: MPI_Type_extent(MPI_Datatype datatype, MPI_Aint* extent) и MPI_Type_size(MPI_Datatype datatype, int* size). Первая возвращает протяженность extent для типа datatype, а вторая — его размер size.

Иногда при определении нового типа бывает желательно указать для него начальный или конечный пустой промежуток («дыру», hole). В стандарте MPI-1 для этих целей предусмотрены особые базовые типы (*псевдотипы*) MPI_LB и MPI_UB, не связанные ни с какими реальными данными и имеющие нулевой размер и протяженность. Их можно использовать в функции MPI_Type_struct в качестве «меток» начальной и конечной позиции определяемого типа. Например, если в функции MPI_Type_struct при определении типа type1 включить в массив oldtypes три элемента MPI_LB, MPI_INT, MPI_UB, определив массивы blocklens и displs следующим образом: blocklens = {1, 1, 1}, displs = {-3, 0, 6}, то тип type1 будет содержать один целочисленный элемент, перед которым будет располагаться начальный промежуток размером 3 байта, а верхняя граница типа будет располагаться на расстоянии 6 байт от первого байта, занимаемого целочисленным элементом. Таким образом, размер созданного типа будет равен размеру типа MPI_INT (обычно 4 байта), а протяженность будет равна 9 байтам (байт номер 6 в протяженность не включается, так как тип MPI_UB, как и MPI_LB, протяженности не имеет):

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

Если теперь использовать функцию MPI_Type_contiguous с параметрами (2, type1, &type2) для определения нового типа type2, то данный тип будет содержать два целочисленных элемента, а его протяженность будет равна 18 байтам:

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

Заметим, что при объединении нескольких типов с явно указанными метками границ все метки удаляются, кроме самой левой границы MPI_LB и самой правой границы MPI_UB.

Использованный в стандарте MPI-1 способ определения начальных и конечных промежутков, основанный на применении меток, имеет недостаток: явно заданные границы исходного типа *нельзя уменьшить* при определении нового типа; их можно только увеличить путем указания новых меток MPI_LB и MPI_UB. Поэтому в стандарте MPI-2 был предложен новый, более гибкий и удобный способ задания начального и конечного пустого промежутка при определении нового типа, основанный на применении

специальной функции MPI_Type_create_resized(MPI_Datatype oldtype, MPI_Aint lb, MPI_Aint extent, MPI_Datatype* newtype). В ней для определения нового типа newtype указывается базовый тип oldtype, новая позиция левой границы lb и новая протяженность extent. Например, для определения типа type1, описанного выше, достаточно использовать следующий вызов:

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

Если у исходного типа oldtype уже были заданы начальный и конечный пустые промежутки, то функция MPI_Type_create_resized удаляет их и создает новые; таким образом, для нового типа можно как увеличить их, так и уменьшить. Если требуется указать только *конечный* пустой промежуток, то параметр lb следует положить равным 0.

В стандарт MPI-2 была также добавлена новая функция, позволяющая одновременно определить левую границу lb и протяженность extent для типа datatype: MPI_Type_get_extent(MPI_Datatype datatype, MPI_Aint* lb, MPI_Aint* extent). При этом прежняя функция MPI_Type_extent была объявлена устаревшей.

Интерфейс MPI предусматривает еще один способ формирования сообщений, содержащих данные различных типов. Этот способ основан на *упаковке* (pack) данных в процессе-отправителе, их пересылке и последующей *распаковке* (unpack) в процессе-получателе. Преимуществом такого способа является то, что он не требует определения новых типов, а недостатком — необходимость использования дополнительного буфера для хранения упакованных данных.

Для упаковки данных предназначена функция MPI_Pack со следующими параметрами:

void* inbuf – входной буфер с исходными данными;

int incount – количество элементов во входном буфере;

MPI_Datatype datatype – тип элементов во входном буфере;

void* outbuf – выходной буфер с упакованными данными (выходной параметр);

int outsize – размер выходного буфера (в байтах);

int* position – текущая позиция в выходном буфере в байтах (входной и выходной параметр);

MPI_Comm comm – коммуникатор, для которого упаковываются данные.

Функция MPI_Pack упаковывает incount элементов типа datatype в выходном буфере outbuf, начиная с указанной позиции position. После выполнения данной операции значение параметра position увеличивается, определяя новую текущую позицию в выходном буфере. При первом вызове функции для данного выходного буфера параметр position следует положить равным 0. После последнего вызова функции для данного выходного буфера параметр position будет равен размеру его заполненной части (в байтах). Необходимо позаботиться о том, чтобы размер outsize выходного буфера

был достаточным для хранения всех упакованных данных (т. е. чтобы итоговое значение параметра position не превосходило значения outsize).

Обратите внимание на то, что при упаковке (и последующей распаковке) необходимо указывать коммуникатор, используемый для пересылки упакованных данных.

При пересылке упакованных данных указывается специальный тип MPI_PACK, а размер задается в байтах.

Функция MPI_Unpack распаковывает полученное сообщение на стороне процесса-получателя. Ее параметры:

void* inbuf – входной буфер (с упакованными данными);

int insize – размер входного буфера (в байтах);

int* position – текущая позиция во входном буфере в байтах (входной и выходной параметр);

void* outbuf – буфер с распакованными данными (выходной параметр);

int outcount – количество элементов, извлекаемых из входного буфера;

MPI_Datatype datatype – тип элементов, извлекаемых из входного буфера;

MPI_Comm comm – коммуникатор, от которого получены распаковываемые данные.

Распаковка начинается с указанной позиции position входного буфера. После выполнения данной операции значение параметра position увеличивается, определяя новую текущую позицию во входном буфере. При первом вызове функции MPI_Unpack для данного входного буфера параметр position следует положить равным 0.

Имеется также функция MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int* size), позволяющая определить размер памяти size (в байтах), *достаточный* для хранения incount упакованных данных типа datatype. Необходимо, однако, иметь в виду, что возвращенное значение size может оказаться *больше* того, которое реально потребуется для хранения указанного числа упакованных данных.

Задания позволяющие ознакомиться со всеми описанными выше возможностями, собраны в группе MPI4Type (см. п. 2.4). В первой подгруппе данной группы рассматриваются базовые способы определения новых типов, вторая подгруппа посвящена пересылке упакованных данных. В третьей подгруппе представлены более содержательные примеры по определению новых типов, связанных, в основном, с частями двумерных массивов (матриц); в этих примерах, в частности, требуется дополнительно определять типы с завершающими пустыми промежутками.

Рассмотрим первое задание из третьей подгруппы группы MPI4Type.

MPI4Type14. В главном процессе даны два набора целых чисел: A размера $3K$ и N размера K, где K — количество подчиненных процессов. Считая, что элементы наборов нумеруются от 1, переслать и вывести в каждом подчиненном процессе ранга R ($R = 1, 2, \ldots, K$) N_R элементов

из набора A, начиная с элемента A_R и перебирая их через один (например, если N_2 равно 3, то в процесс ранга 2 надо переслать элементы A_2, A_4, A_6). Использовать для пересылки каждого набора элементов по одному вызову функций MPI_Send, MPI_Probe и MPI_Recv, причем функция MPI_Recv должна возвращать массив, содержащий только те элементы, которые требуется вывести. Для этого в главном процессе определить новый тип, содержащий единственный целочисленный элемент и дополнительный конечный *пустой промежуток*, равный размеру элемента целого типа. Использовать в функции MPI_Send исходный массив A с требуемым смещением, указав в качестве второго параметра количество элементов, равное N_R, а в качестве третьего параметра — новый тип. В функции MPI_Recv использовать целочисленный массив размера N_R и тип MPI_INT. Для определения количества N_R переданных элементов использовать в подчиненных процессах функцию MPI_Get_count.

Указание. Для задания завершающего пустого промежутка при определении нового типа в MPI-2 следует использовать функцию MPI_Type_create_resized (в данном случае эту функцию надо применить к типу MPI_INT). В MPI-1 надо использовать *метку нулевого размера* типа MPI_UB совместно с функцией MPI_Type_struct (в стандарте MPI-2 тип MPI_UB объявлен устаревшим).

При запуске программы-заготовки, созданной для этого задания, на экране появится окно с вариантом исходных данных и примером правильных результатов (рис. 17). Для того чтобы уменьшить размер окна, в нем скрыт раздел с формулировкой задания (для скрытия и последующего восстановления раздела с формулировкой достаточно нажать клавишу [Del]).

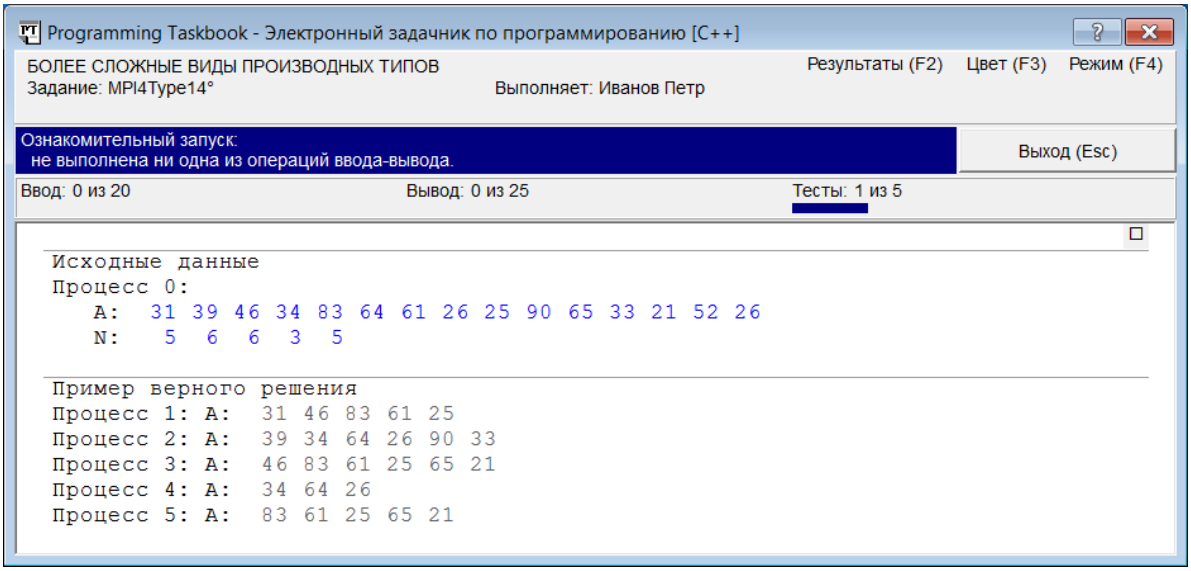

Рис. 17. Ознакомительный запуск задания MPI4Type14

В данном задании надо переслать в подчиненные процессы элементы массива *A* из главного процесса, перебирая их «через один». Если не создавать новые типы, то придется либо пересылать «лишние» данные (что приведет к увеличению размеров пересылаемых сообщений, а также к необходимости выделения дополнительной памяти в процессах-получателях), либо предварительно, до пересылки, копировать требуемые элементы во вспомогательный буфер (что потребует выделения дополнительной памяти в процессе-отправителе, а также дополнительных действий в этом процессе на копирование нужных данных во вспомогательный буфер).

Чтобы реализовать пересылку требуемых данных эффективным образом как на стороне отправителя, так и на стороне получателя, следует определить вспомогательный тип данных, причем только для процесса-отправителя. С помощью этого типа мы сможем сформировать сообщение, содержащее только необходимые элементы массива *A*. Для процесса-получателя новый тип не потребуется, так как полученное этим процессом сообщение не будет содержать «лишних» данных.

На первом этапе решения займемся вводом исходных данных в главном процессе. Поскольку размеры исходных массивов зависят от количества параллельных процессов, используем для них динамическое выделение памяти:

```
if (rank == 0)
{
  int k = size - 1;
  int *a = new int[3 * k];
  int *n = new int[k];
  for (int i = 0; i < 3 * k; ++i)
    pt >> a[i];
  for (int i = 0; i < k; ++i)
    pt >> n[i];
  // определение нового типа и отправка сообщения
  delete[] a;
  delete[] n;
}
```

После завершения работы с созданными динамическими массивами мы освобождаем выделенную для них память с помощью оператора delete[].

При запуске нового варианта программы в окне задачника будет выведено сообщение «*Запуск с правильным вводом данных: все требуемые исходные данные введены, результаты не выведены*».

Теперь займемся определением нового типа (соответствующие операторы надо разместить в позиции, помеченной комментарием). Чтобы про-

иллюстрировать возможности как стандарта MPI-1, так и стандарта MPI-2, опишем два варианта подобного определения.

В первом варианте будем использовать только средства стандарта MPI-1:

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

Вначале с помощью функции MPI_Type_size мы определили размер элемента целого типа MPI_INT. Затем с помощью функции MPI_Type_struct мы создали структуру из двух блоков (каждый длиной 1), причем первый блок содержит одно целое число, а второй — элемент MPI_UB (метку верхней границы) благодаря которому можно задать завершающий пустой промежуток для определяемого типа данных t. Напомним, что смещения для каждого блока (задаваемые в массиве displs) указываются в байтах и отсчитываются от начала первого блока.

Чтобы проверить правильность созданного типа данных, выведем в окне отладки его характеристики (размер и протяженность):

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

При запуске данного варианта программы окно задачника примет вид, приведенный на рис. 18. Мы видим, что протяженность созданного типа действительно в два раза превосходит размер базового типа MPI_INT (равный 4 байтам). Размеры нового типа и типа MPI_INT совпадают, так как созданный тип содержит единственный целочисленный элемент.

Если использовать средства, появившиеся в стандарте MPI-2, то при определении типа t мы сможем обойтись без вспомогательных массивов:

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

Характеристики типа, созданного с помощью функции MPI_Type_create_resized, разумеется, будут совпадать с соответствующими характеристиками типа, созданного с помощью средств стандарта MPI-1.

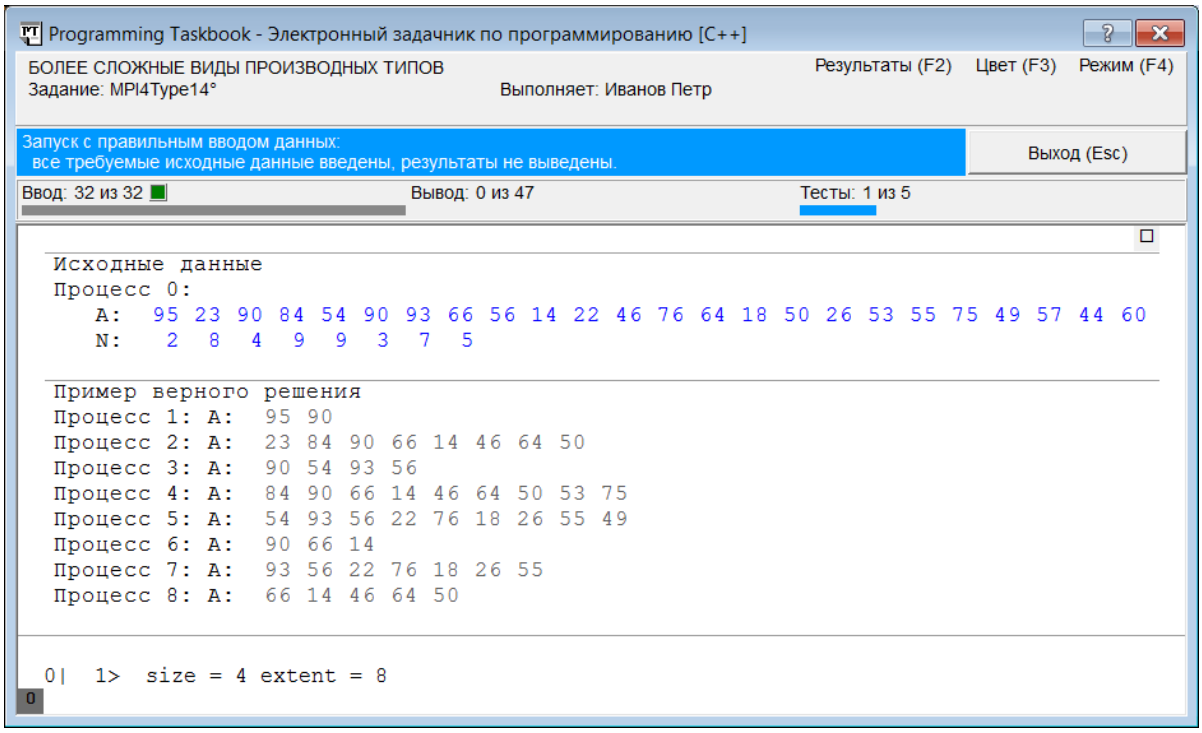

Рис. 18. Окно с информацией о созданном типе данных

Чтобы завершить фрагмент программы, соответствующий главному процессу, нам осталось использовать созданный тип для пересылки в подчиненные процессы требуемых данных, предварительно зарегистрировав его с помощью функции MPI_Type_commit:

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

Примечание 2. Если бы в программе не была вызвана функция MPI_Type_commit, то при использовании системы MPICH2 1.3 в окне задачника было бы выведено сообщение об ошибке MPI: «*Error MPI_ERR_TYPE: Datatype has not been committed*».

Осталось определить фрагмент, соответствующий подчиненным процессам, добавив ветку else к оператору if (rank == 0):

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

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

Для определения количества получаемых данных мы использовали функцию MPI_Probe, после чего создали принимающий буфер нужного размера и заполнили его в функции MPI_Recv.

Заметим, что при решении данной задачи мы впервые встречаемся с ситуацией, когда тип отправляемых данных (t) не совпадает с типом принимаемых (MPI_INT). Кроме того, следует подчеркнуть, что в подчиненных процессах мы не пользовались новым типом и при этом получили именно те данные, которые требовалось переслать из главного процесса, причем в принимающем буфере (в отличие от посылающего буфера) полученные данные располагаются подряд.

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

Приведем полный текст полученного решения, в котором используются возможности, добавленные в стандарт MPI-2:

```
#include "pt4.h"
#include "mpi.h"
void Solve()
{
  Task("MPI4Type14");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0)
  {
    int k = size - 1;
    int *a = new int[3 * k];
    int *n = new int[k];
    for (int i = 0; i < 3 * k; ++i)
      pt >> a[i];
    for (int i = 0; i < k; ++i)
      pt >> n[i];
    MPI_Datatype t;
    int int_sz;
    MPI_Type_size(MPI_INT, &int_sz);
    MPI_Type_create_resized(MPI_INT, 0, 2 * int_sz, &t);
```

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

Вместо массивов (статических или динамических) в программах на C++ можно применять *контейнер «вектор»* std::vector<T> из стандартной библиотеки шаблонов STL [2]. Это позволит использовать для ввода-вывода дополнительные возможности, связанные с *итераторами* потока pt (см. п. 4.1.2). Приведем вариант решения задачи MPI4Type14, в котором вместо массивов применяются векторы (добавленные фрагменты программы выделены полужирным шрифтом, а удаленные фрагменты перечеркнуты):

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

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

Прокомментируем сделанные исправления.

Для возможности работы с векторами к программе необходимо подключить стандартный заголовок <vector>. Кроме того, мы подключили заго-

ловок <algorithm>, позволяющий использовать в программе *алгоритмы* библиотеки STL.

При создании вектора его можно сразу заполнить исходными данными, указав в конструкторе итератор начала и конца потока ввода. В нашем случае используется поток ввода pt, для которого в задачнике определен шаблонный итератор ptin_iterator<T>, позволяющий организовать чтение данных типа T. Конструктор с одним параметром ptin_iterator<T>(int count) создает итератор для чтения из потока pt требуемого числа элементов, конструктор без параметров ptin_iterator<T>() создает итератор конца потока ввода. Если вектор предназначен для хранения набора данных, полученного из другого процесса, то для его создания достаточно использовать конструктор с одним параметром — требуемым размером вектора (такой вариант конструктора мы использовали в ветке else условного оператора). В этом случае вектор заполняется нулевыми значениями типа T.

Следует обратить особое внимание на то, что для создания вектора n мы указали в качестве параметра первого итератора *выражение* 1*k вместо *переменной* k. Это объясняется тем, что описание вида

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

интерпретируется лексическим анализатором языка C++ как объявление прототипа *функции* n с двумя параметрами — указателями на функции. Чтобы данное описание интерпретировалось нужным нам образом (т. е. как описание вектора n, инициализируемого с помощью двух итераторов), достаточно превратить параметр первого итератора в выражение, поскольку в этом случае полученное описание уже не удастся интерпретировать как прототип функции:

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

Имеется и другой способ решить отмеченную проблему — *заключить один из параметров конструктора в круглые скобки*:

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

При передаче вектора в качестве буфера для посылаемых или принимаемых данных необходимо указывать *адрес начального элемента буфера* (в частности, в ветке else нам пришлось изменить первый параметр a функции MPI_Recv на &a[0]).

Для вывода всех элементов вектора достаточно использовать алгоритм copy, указав в качестве первых двух параметров итераторы begin и end начала и конца вектора, а в качестве последнего параметра — итератор ptout_iterator для потока вывода pt.

Итак, использовав средства стандартной библиотеки шаблонов языка C++ (и связанные с ними средства электронного задачника — итераторы ptin_iterator и ptout_iterator), мы смогли более кратко описать действия по вводу и выводу наборов данных. Кроме того, нам не потребовалось выполнять

особые действия, связанные с освобождением памяти, так как память, выделенная для векторов, освобождается в их деструкторах, которые вызываются автоматически.

1.2.7. Создание новых коммуникаторов

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 (см. п. 2.5). Следует заметить, что в заданиях группы MPI5Comm рассматриваются только так называемые *интракоммуникаторы* (intra-communicators), связанные с одной группой процессов. В MPI можно создавать и другой вид коммуникаторов — *интеркоммуникаторы* (inter-communicators), которые связаны не с одной, а с двумя группами процессов. Интеркоммуникаторам посвящена группа заданий MPI8Inter (п. 2.8), большинство из которых можно выполнять только в системе MPICH2 1.3, поддерживающей стандарт MPI-2.

Новые коммуникаторы можно создать тремя способами.

Простейший способ создания нового коммуникатора заключается в создании *копии* имеющегося коммуникатора. Для этого предназначена функция MPI_Comm_dup(MPI_Comm comm, MPI_Comm* newcomm), которую необходимо вызвать во всех процессах исходного коммуникатора comm. Новый коммуникатор newcomm включает ту же группу процессов и имеет те же дополнительные характеристики (связанные, в частности, с наличием виртуальной топологии — см. п. 1.2.8), что и исходный коммуникатор comm. Сообщения, пересылаемые с помощью одного из этих коммуникаторов, никак не влияют на сообщения, пересылаемые с помощью другого; они пересылаются «по разным каналам». Копии коммуникатора MPI_COMM_WORLD часто создаются в дополнительных параллельных библиотеках и используются для пересылки между процессами внутренней информации, необходимой для нормальной работы этих библиотек. Пользователь библиотек не имеет доступа к этим копиям и поэтому не может повлиять на выполняемую с их помощью передачу данных.

Подчеркнем, что обычное присваивание вида

```
MPI_Comm newcomm = comm;
```

не создает копию коммуникатора comm, оно лишь создает *копию дескриптора*, связанного с тем же самым коммуникатором.

Для сравнения коммуникаторов предусмотрена функция MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int* result). При сравнении разных дескрипторов, связанных с одним и тем же коммуникатором, эта функция возвращает в переменной result значение MPI_IDENT. Если сравниваются различные коммуникаторы, содержащие одинаковый набор про-

цессов, и эти процессы упорядочены одинаковым образом, то возвращается значение MPI_CONGRUENT (именно это значение будет возвращено при сравнении исходного коммуникатора и его копии, созданной с помощью функции MPI_Comm_dup). Если два коммуникатора содержат одинаковые наборы процессов, но порядок следования процессов в них различается, то возвращается значение MPI_SIMILAR. Если коммуникаторы содержат различные наборы процессов, то возвращается значение MPI_UNEQUAL.

Второй способ создания нового коммуникатора требует предварительного определения новой *группы* (group) процессов в пределах другого коммуникатора. Имея такую группу group, входящую в исходный коммуникатор comm, можно создать новый коммуникатор newcomm, который будет содержать только процессы из группы group. Для этого предназначена функция MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm* newcomm). Данная функция должна быть вызвана во всех процессах, входящих в коммуникатор comm; для тех процессов, которые не входят в указанную группу group, в параметре newcomm будет возвращено значение MPI_COMM_NULL.

> **Примечание**. В стандарте MPI-2 возможности функции MPI_Comm_create были расширены таким образом, чтобы при ее однократном вызове можно было создавать *несколько* новых коммуникаторов, связанных с *непересекающимися* группами процессов из исходного коммуникатора. Для этого в процессах каждой из таких групп достаточно вызвать функцию MPI_Comm_create с параметром group, равным данной группе (вызов функции MPI_Comm_create по-прежнему надо выполнять во всех процессах исходного коммуникатора comm). Заметим, что новые возможности функции MPI_Comm_create сблизили ее с функцией MPI_Comm_split (см. далее описание третьего способа создания коммуникаторов).

Для работы с группами процессов (объектами типа MPI_Group) в библиотеке MPI предусмотрено много различных функций, а также две константы: MPI_GROUP_EMPTY (соответствует пустой группе, т. е. группе, не содержащей процессов), MPI_GROUP_NULL (значение, используемое для указания ошибочной группы).

Создать группу *всех* процессов коммуникатора comm можно с помощью функции MPI_Comm_group(MPI_Comm comm, MPI_Group* group).

Для групп, как и для коммуникаторов, предусмотрены функции, позволяющие определить размер группы, т. е. количество входящих в нее процессов (функция MPI_Group_size(MPI_Group group, int* size)), а также ранг текущего (вызвавшего данную функцию) процесса в указанной группе (функция MPI_Group_rank(MPI_Group group, int* rank)). Если текущий процесс не входит в указанную группу, то в параметре rank возвращается значение MPI_UNDEFINED.

Имеется также функция MPI_Group_translate_ranks(MPI_Group group1, int n, int* ranks1, MPI_Group group2, int* ranks2), позволяющая определить ранги процессов в группе group2, если известны их ранги в группе group1. При этом известные ранги процессов группы group1 указываются в массиве ranks1 (размера n), а ранги тех же процессов в группе group2 возвращаются в массиве ranks2 того же размера (выходном параметре). Если какой-либо из процессов первой группы не входит во вторую группу, то соответствующему элементу массива ranks2 присваивается значение MPI_UNDEFINED.

Группы, как и коммуникаторы, можно сравнивать. Предназначенная для этого функция MPI_Group_compare(MPI_Group group1, MPI_Group group2, int* result) возвращает в переменной result одно из трех значений:

MPI_IDENT – две группы содержат одинаковые наборы процессов, причем эти наборы одинаково упорядочены;

MPI_SIMILAR – две группы содержат одинаковые наборы процессов, однако порядок расположения процессов в них различается;

MPI_UNEQUAL – две группы содержат различные наборы процессов.

Имея некоторую группу group, можно создать новую группу, содержащую только *часть* процессов исходной группы. Для этого предназначены функции MPI_Group_incl и MPI_Group_excl с одинаковым набором параметров: (MPI_Group group, int n, int* ranks, MPI_Group* newgroup).

При использовании функции MPI_Group_incl в новую группу включаются те процессы исходной группы, ранги которых указаны в массиве ranks размера n; следовательно, новая группа будет содержать n процессов. Порядок процессов в новой группе соответствует порядку следования рангов в массиве ranks; таким образом, процесс новой группы ранга i, i = 0, …, n–1, будет совпадать с процессом ранга ranks[i] исходной группы (массив ranks не может содержать одинаковых элементов). Если параметр n равен 0, то возвращается пустая группа, равная константе MPI_GROUP_EMPTY.

При использовании функции MPI_Group_excl в новую группу включаются те процессы исходной группы, ранги которых *не указаны* в массиве ranks размера n; следовательно, новая группа будет содержать на n процессов меньше, чем исходная группа. Порядок процессов в новой группе соответствует порядку следования процессов в исходной группе; порядок элементов в массиве ranks роли не играет, требуется лишь, чтобы в нем не содержались одинаковые элементы. Если параметр n равен 0, то возвращается группа, равная исходной.

Предусмотрены варианты функций MPI_Group_incl и MPI_Group_excl, которые удобно использовать, если ранги включаемых (или, соответственно, исключаемых) процессов образуют регулярные диапазоны. Эти варианты имеют имена MPI_Group_range_incl и MPI_Group_range_excl и одинаковый набор параметров: (MPI Group group, int n, int ranges[][3], MPI Group *newgroup). Параметр ranges является массивом размера n, причем его элементами являются

тройки — массивы из трех целых чисел. Каждая такая тройка определяет диапазон рангов вида (*first*, *last*, *step*), в который входят ранги от *first* до *last* включительно с шагом *step* (шаг *step* не может быть нулевым, но может быть отрицательным; в этом случае значение *first* должно быть больше значения *last*). Допускаются «вырожденные» диапазоны, состоящие из одного процесса ранга *R* и определяемые тройкой вида (*R*, *R*, 1). Для функции MPI_Group_range_incl массив ranges определяет ранги процессов из группы group, включаемых в группу newgroup (в указанном порядке), а для функции MPI_Group_range_excl — ранги процессов, исключаемых из группы group с целью получения группы newgroup. Диапазоны в массиве ranges должны быть попарно непересекающимися.

Имея две исходные группы group1 и group2, можно применить к ним одну из *теоретико-множественных операций*: объединение (union), пересечение (intersection), разность (difference), получив в результате новую группу newgroup. Для этого предусмотрены функции MPI_Group_union, MPI_Group_intersection, MPI_Group_difference с одинаковым набором параметров: (MPI_Group group1, MPI_Group group2, MPI_Group* newgroup).

Объединение состоит из всех процессов первой группы (взятых в том же порядке), дополненных теми процессами второй группы (в том же порядке), которые отсутствуют в первой группе. *Пересечение* состоит из тех процессов первой группы (взятых в том же порядке), которые входят во вторую группу. *Разность* состоит из тех процессов первой группы (взятых в том же порядке), которые не входят во вторую группу. В результате выполнения операций пересечения и разности может быть получена пустая группа; в этом случае в параметре newgroup возвращается значение MPI_GROUP_EMPTY. Операции объединения и пересечения не являются коммутативными, так как при перемене местами исходных групп может измениться порядок расположения процессов в новой группе.

Заметим, что созданные в программе группы и коммуникаторы можно уничтожить, освободив связанные с ними дескрипторы. Для этого предназначены функции MPI_Group_free(MPI_Group* group) и MPI_Comm_free(MPI_Comm* comm). В результате выполнения данных функций в параметре group возвращается значение MPI_GROUP_NULL, а в параметре comm — значение MPI_COMM_NULL.

Третий способ создания новых коммуникаторов связан с функцией MPI_Comm_split, расщепляющей исходный коммуникатор на набор коммуникаторов с попарно непересекающимися группами процессов. Продемонстрируем использование этой функции на примере выполнения одного из заданий, входящих в первую подгруппу группы MPI5Comm «Группы процессов и коммуникаторы» (см. п. 2.5.1). Задания из двух следующих подгрупп, связанные с виртуальными топологиями, обсуждаются в п. 1.2.8 и 1.2.9.

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, которое отобразилось на экране при ознакомительном запуске программы-заготовки для этого задания (рис. 19).

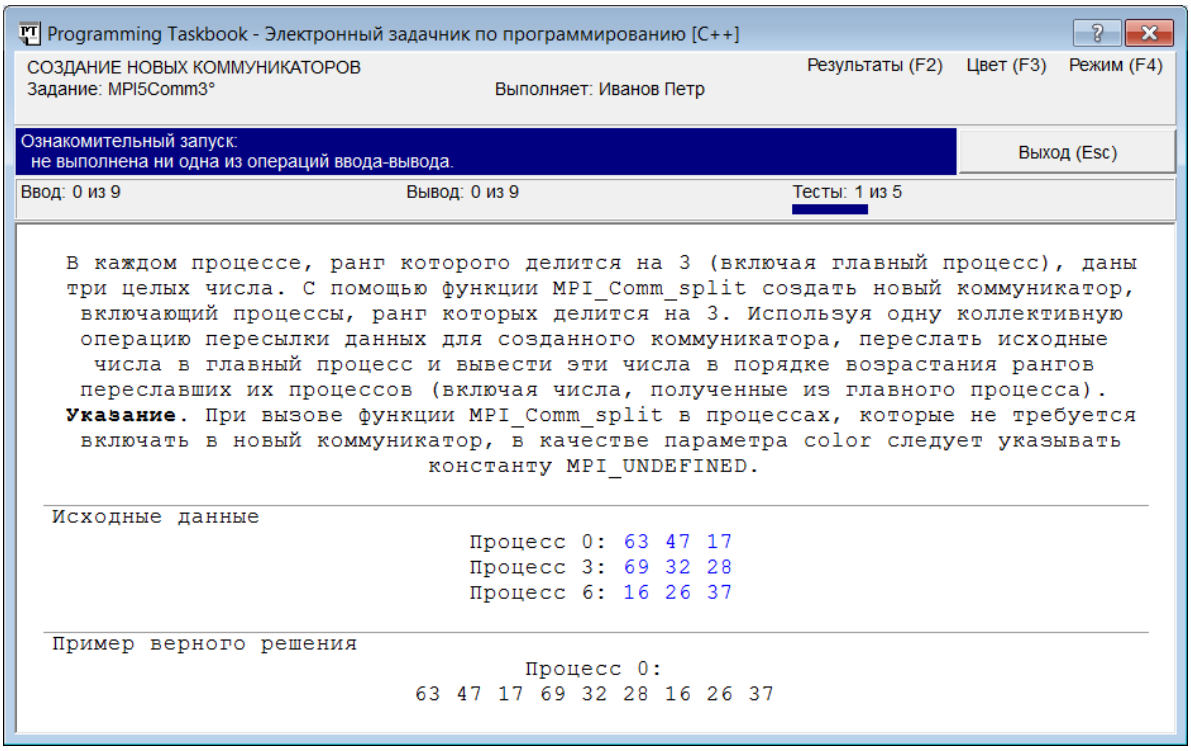

Рис. 19. Ознакомительный запуск задания MPI5Comm3

Заметим, что в консольном окне при этом 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 организовать взаимодействие лишь между некоторыми из имеющихся процессов. Разумеется, мы можем воспользоваться функциями MPI, обеспечивающими обмен данными между двумя процессами (как в решении задачи MPI2Send11, приведенном в п. 1.2.2), но более эффективным будет вариант с подходящей коллектив-

ной операцией пересылки данных. 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 различными способами; мы воспользуемся функцией MPI_Comm_split, упомянутой в формулировке задания.

Функция MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm* newcomm) разбивает множество процессов, входящих в коммуникатор comm, на отдельные коммуникаторы. Данная функция должна быть вызвана во всех процессах, входящих в коммуникатор comm.

В результате выполнения этой функции каждый процесс коммуникатора comm получает именно тот новый коммуникатор newcomm из созданного набора, в который входит данный процесс. Предусмотрена также ситуация, когда некоторые процессы 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, указавшие при вызове функции MPI_Comm_split одинаковое значение параметра color включаются в один и тот же новый коммуникатор. Любой цвет должен задаваться неотрицательным числом. Предусмотрен также «неопределенный цвет» MPI_UNDEFINED; его надо указывать для процессов, которые 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 коммуникатор MPI_COMM_NULL. Разумеется, в нашем случае в условии выхода можно было бы анализировать остаток от деления rank на 3, однако использованный вариант проверки 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 можно использовать массив data из трех элементов. Размер результирующего массива res, который будет получен в главном процессе, зависит от количества процессов параллельного приложения. При обсуждении задания MPI3Coll23 мы отмечали, что в такой ситуации можно использовать либо статический массив достаточно большого размера, либо динамический массив (или вектор std::vector<T>), размер которого будет определен после того, как станет известно количество процессов. В п. 1.2.5, решая задачу MPI3Coll23, мы использовали статический массив. При решении задачи MPI4Type14 мы использовали динамические массивы, а также их альтернативу из стандартной библиотеки шаблонов C++ — векторы std::vector<T>. В данной программе мы еще раз воспользуемся средствами библиотеки STL, описав исходные и результирующие наборы данных data и res как векторы и применив для их ввода и вывода итераторы потока pt:

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

Напомним, что в случае использования в программе векторов и алгоритмов из библиотеки STL к ней надо подключить стандартные заголовки <vector> и <algorithm>.

Для нахождения общего числа полученных элементов мы предварительно определили количество процессов в созданном коммуникаторе comm (с помощью функции MPI_Comm_size), записав это количество в пере-

менную size. Поскольку каждый процесс коммуникатора comm пересылает в главный процесс три элемента, размер вектора res полагается равным 3 * size.

Затем вызывается функция MPI_Gather (см. п. 1.2.4). Напомним, что в функции MPI_Gather в качестве пятого параметра указывается не размер буфера 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.

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

Приведем полный текст полученного решения:

```
#include "pt4.h"
#include "mpi.h"
#include <vector>
#include <algorithm>
void Solve()
{
  Task("MPI5Comm3");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm comm;
  int color = rank % 3 == 0 ? 0 : MPI_UNDEFINED;
  MPI_Comm_split(MPI_COMM_WORLD, color, rank, &comm);
  if (comm == MPI_COMM_NULL)
    return;
  MPI_Comm_size(comm, &size);
  std::vector<int> res(3 * size),
    data(ptin_iterator<int>(3), ptin_iterator<int>()),;
  MPI_Gather(&data[0], 3, MPI_INT, &res[0], 3, MPI_INT, 0, comm);
  if (rank == 0)
    copy(res.begin(), res.end(), ptout_iterator<int>());
}
```

1.2.8. Декартова топология

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 (см. п. 1.2.7). Однако в ряде ситуаций бывает желательно не только выделить требуемую часть процессов (и/или расположить процессы в другом порядке), но и установить между ними дополнительные связи. Для этих целей в библиотеке MPI предусмотрены средства, позволяющие определить для всех или некоторых процессов параллельного приложения *виртуальную топологию* (virtual topology).

Виртуальная топология задает на множестве процессов некоторую структуру, позволяющую упорядочить эти процессы более сложным образом, чем в обычных коммуникаторах (в которых процессы упорядочиваются линейно). Имеются два вида виртуальной топологии: декартова топология и топология графа. В случае *декартовой топологии* (Cartesian topology) все процессы интерпретируются как узлы некоторой n-мерной *решетки* (grid) размера $k_1 \times k_2 \times \ldots \times k_n$ (если $n = 2$, то процессы можно рассматривать как элементы *прямоугольной матрицы* (matrix) 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. В стандарте MPI-2 был добавлен особый вид топологии графа — *топология распределенного графа* (distributed graph topology).

Информация об используемой виртуальной топологии связывается с коммуникатором. Для проверки наличия виртуальной топологии у коммуникатора comm предусмотрена функция MPI_Topo_test(MPI_Comm comm, int* status), которая возвращает обнаруженный тип топологии в выходном параметре status. Параметр status может принимать следующие значения:

MPI_CART – с коммуникатором связана декартова топология;

MPI_GRAPH – с коммуникатором связана топология графа;

MPI_DIST_GRAPH – с коммуникатором связана топология распределенного графа (данная константа появилась в стандарте MPI-2);

MPI_UNDEFINED – с коммуникатором не связана никакая виртуальная топология.

В данном пункте мы расссмотрим функции библиотеки MPI, связанные с декартовой топологией. Их можно разбить на четыре группы:

- определение декартовой топологии и связанного с ней коммуникатора (функция MPI_Cart_create, а также вспомогательная функция MPI_Dims_create);

- определение характеристик имеющейся декартовой топологии (функции MPI_Cartdim_get, MPI_Cart_get, MPI_Cart_rank, MPI_Cart_coords);
- расщепление исходной декартовой решетки на *подрешетки* (subgrids) меньшей размерности (функция MPI_Cart_sub);
- определение рангов источников и приемников при *сдвиге* (shift) данных вдоль некоторой координаты декартовой решетки (функция MPI_Cart_shift).

Некоторые из этих функций (MPI_Cart_create, MPI_Cart_coords, MPI_Cart_rank, MPI_Cart_sub) мы опишем при обсуждении задачи MPI5Comm17, а остальные — в конце пункта, после завершения обсуждения задачи.

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 программы-заготовки для этой задачи окно задачника примет вид, аналогичный приведенному на рис. 20.

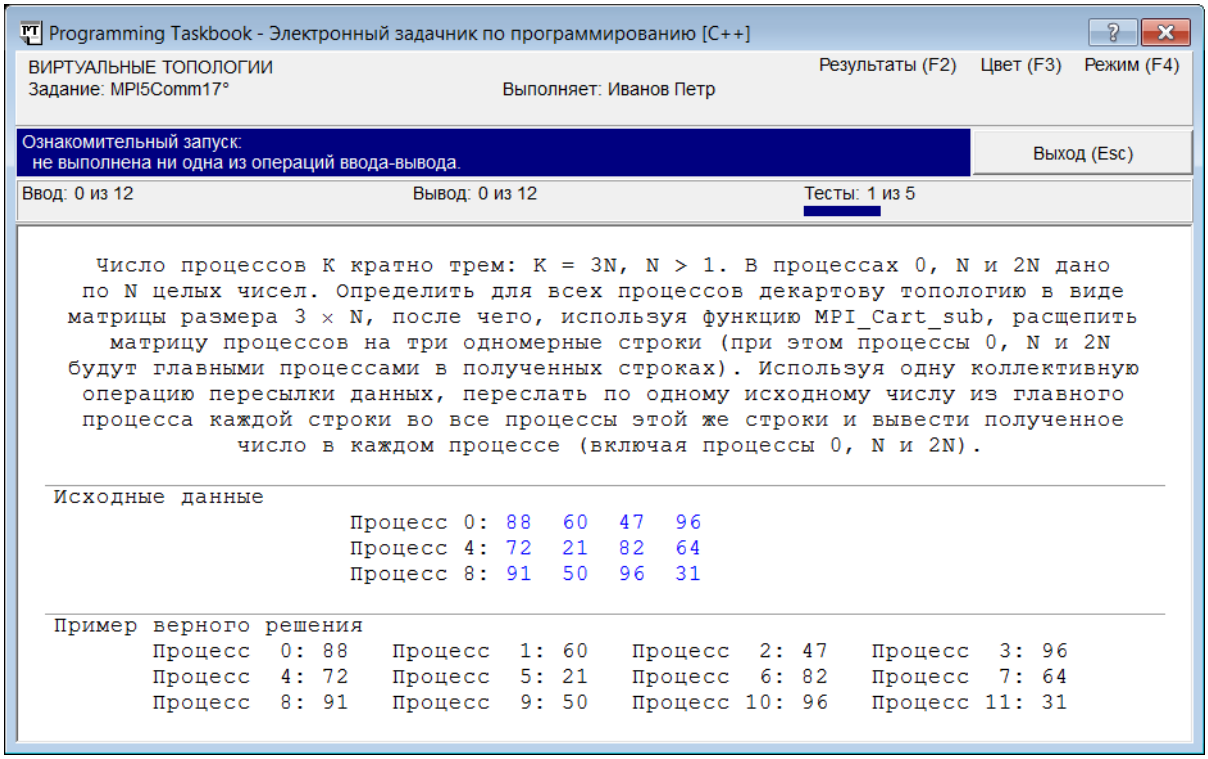

Рис. 20. Ознакомительный запуск задания MPI5Comm17

Данный вариант 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 предназначена функция MPI_Cart_create, которая имеет следующие параметры:

MPI_Comm oldcomm – исходный коммуникатор, для процессов которого определяется декартова топология (в нашем случае MPI_COMM_WORLD);

int ndims – число размерностей создаваемой декартовой решетки (в нашем случае 2);

int* dims – 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);

int* periods – целочисленный массив флагов, определяющих *периодичность* каждого измерения (в нашем случае достаточно использовать массив из двух нулевых элементов);

int reorder – целочисленный флаг, определяющий, можно ли среде MPI автоматически менять порядок нумерации процессов (в нашем случае необходимо положить этот параметр равным 0);

MPI_Comm* cartcomm – результирующий коммуникатор с декартовой топологией (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 конце данного пункта); в этом случае соответствующий элемент в массиве флагов periods 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. Кроме того, с коммуникатором comm дополнительно связана виртуальная топология, а коммуникатор MPI_COMM_WORLD никакой виртуальной топологией не обладает.

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, как и ранг, нумеруются от 0.

Координаты процесса в декартовой топологии можно определить по его рангу с помощью функции MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int* coords), а функция MPI_Cart_rank(MPI_Comm comm, int* coords, int* rank) позволяет решить обратную задачу. Обратите внимание на то, что в функции MPI_Cart_coords необходимо указать дополнительный параметр maxdims — размер выходного массива coords.

Для выполнения нашего 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 применения, 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. 21.

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

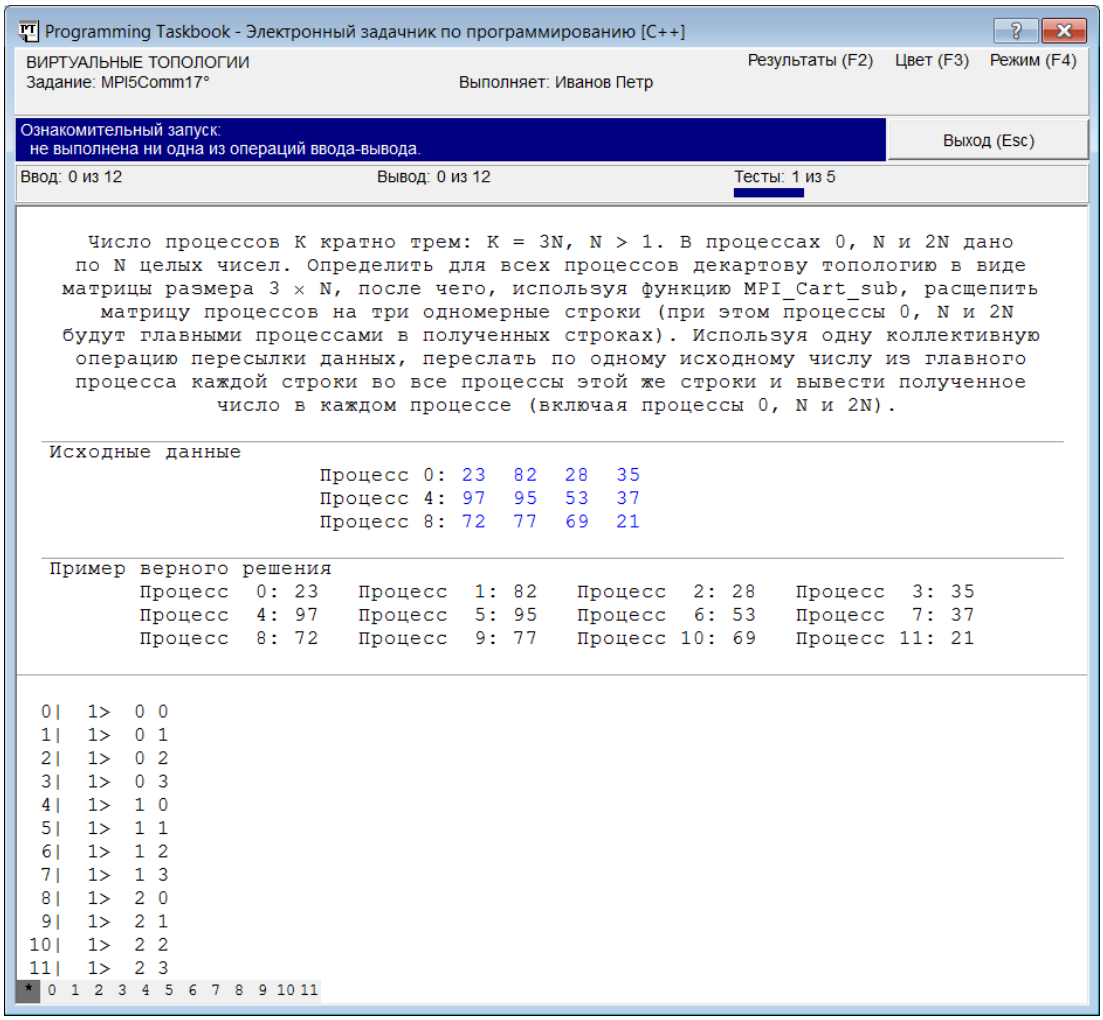

Рис. 21. Вывод декартовых координат процессов в разделе отладки

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 (см. п. 1.2.4), 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(MPI_Comm comm, int* remain_dims, MPI_Comm* newcomm).

В качестве ее первого параметра comm следует указать исходный коммуникатор с декартовой топологией, в качестве второго — массив флагов remain_dims, определяющий номера тех измерений, которые должны остаться в подрешетках: если соответствующее измерение должно *остаться* в каждой подрешетке, то на его месте в массиве указывается *ненулевой* флаг, если же по данному измерению выполняется расщепление исходной решетки (и, следовательно, в полученных подрешетках данное измерения «пропадает»), то значение связанного с этим измерением флага должно быть нулевым.

Функция MPI_Cart_sub должна быть вызвана во всех процессах исходного коммуникатора comm. В результате ее выполнения 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 newcomm) 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 новый вариант программы, 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 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 неверное указание ее второго параметра — массива флагов remain_dims. Если, например, в приведенной 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. 22.

Programming Taskbook - Электронный задачник по программированию [C++]

ВИРТУАЛЬНЫЕ ТОПОЛОГИИ
Задание: MPI5Comm17°
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)

Введены не все требуемые исходные данные.
Ошибка произошла в процессах 4, 8.
Выход (Esc)

Ввод: 3 из 12 Вывод: 9 из 12 Тесты: 1 из 5

```
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 × 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).

Исходные данные
                    Процесс 0: 64  68  13  68
                    Процесс 4: 85  11  79  25
                    Процесс 8: 32  42  36  72

Полученные результаты
      Процесс  0: 64   Процесс  1:      Процесс  2:      Процесс  3:
      Процесс  4: 68   Процесс  5:  0   Процесс  6:  0   Процесс  7:  0
      Процесс  8: 13   Процесс  9:  0   Процесс 10:  0   Процесс 11:  0

Пример верного решения
      Процесс  0: 64   Процесс  1: 68   Процесс  2: 13   Процесс  3: 68
      Процесс  4: 85   Процесс  5: 11   Процесс  6: 79   Процесс  7: 25
      Процесс  8: 32   Процесс  9: 42   Процесс 10: 36   Процесс 11: 72

0|  1> !При наличии ошибок в подчиненных процессах
0|  2> ! решение для главного процесса не проверяется.
1|  1> !Попытка ввести лишние исходные данные.
2|  1> !Попытка ввести лишние исходные данные.
3|  1> !Попытка ввести лишние исходные данные.
4|  1> !Введены не все требуемые исходные данные.
4|  2> ! Количество прочитанных данных: 0 (из 4).
8|  1> !Введены не все требуемые исходные данные.
8|  2> ! Количество прочитанных данных: 0 (из 4).
* 0 1 2 3 4 8
```

Рис. 22. Окно задачника при ошибочном выполнении задания MPI5Comm17

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.

Завершив обсуждение задачи MPI5Comm17, опишем те функции, связанные с декартовой топологией, которые не потребовались при ее решении.

При определении декартовой топологии с помощью функции MPI_Cart_create необходимо указать две основные характеристики: *размерность* декартовой топологии ndims (число измерений) и *количество узлов* (т. е. процессов) по каждому измерению — массив целых чисел dims размера ndims. В библиотеке MPI предусмотрена вспомогательная функция MPI_Dims_create(int nnodes, int ndims, int* dims), позволяющая определить оптимальное количество узлов по каждому измерению декартовой решетки, если известно общее количество узлов nnodes и число измерений ndims. Найденное количество узлов возвращается в выходном массиве dims.

Элементы массива dims, требующие определения, должны иметь нулевые исходные значения; исходные положительные значения элементов массива dims считаются фиксированными и не изменяются. Определенные самой функцией значения элементов массива dims всегда упорядочиваются по убыванию (точнее, по невозрастанию) и выбираются максимально близкими друг к другу (например, из вариантов {6, 1} и {3, 2} выбирается вариант {3, 2}). В случае отрицательных исходных значений или невозможности выбора хотя бы одного варианта требуемой декартовой решетки возникает ошибка (напомним, что в этом случае функция возвращает значение, отличное от MPI_SUCCESS). Приведем несколько примеров (слева от стрелки ==> указываются исходные значения параметров, справа — результирующее содержимое массива dims):

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

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

Функции MPI_Cartdim_get(MPI_Comm comm, int* ndims) и MPI_Cart_get(MPI_Comm comm, int maxdims, int* dims, int* periods, int* coords) позволяют определить характеристики декартовой решетки для существующего коммуникатора comm с декартовой топологией. Первая из них возвращает в параметре ndims размерность декартовой решетки. Вторая функция содержит три выходных параметра:

dims – массив с числом процессов по каждому измерению декартовой решетки;

periods – массив флагов, определяющих периодичность каждого измерения (измерение является периодическим, если соответствующий флаг не равен 0);

coords – массив декартовых координат текущего процесса.

Все эти параметры являются целочисленными массивами размера maxdims.

Нам осталось описать еще одну полезную возможность, которая обеспечивается декартовой топологией: быстрое определение для текущего процесса рангов процесса-источника и процесса-приемника при пересылке данных вдоль указанной координаты (т. е. при обычном или циклическом *сдвиге* данных). Эта возможность обеспечивается функцией MPI_Cart_shift, имеющей следующие параметры:

MPI_Comm comm – коммуникатор с декартовой топологией;

int direction – номер декартовой координаты, вдоль которой выполняется сдвиг (нумерация начинается от 0);

int disp – шаг сдвига вдоль выбранной координаты;

int* rank_source – ранг процесса-источника (выходной параметр);

int* rank_dest – ранг процесса-приемника (выходной параметр).

Возвращаемые данные будут соответствовать *циклическому* сдвигу, если координата, вдоль которой выполняется сдвиг, является *периодической* (это означает, что при определении коммуникатора comm в массиве periods был указан *ненулевой* элемент, соответствующий данной координате). Благодаря параметру disp сдвиг может выполняться с любым шагом, в том числе и отрицательным (в случае отрицательного шага сдвиг выполняется в направлении *убывания* данной координаты).

Если сдвиг не является циклическим, то возможна ситуация, при которой у текущего процесса отсутствует процесс-источник и/или процесс-приемник. Например, при сдвиге с шагом 1 процессы, имеющие координату сдвига, равную 0, не имеют источника, а при сдвиге с шагом –1 у этих процессов отсутствует приемник. В подобной ситуации соответствующий выходной параметр принимает значение MPI_PROC_NULL.

1.2.9. Топология графа

Теперь рассмотрим другой вид виртуальных топологий — топологию графа (данной топологии посвящены заключительные задания группы MPI5Comm — см. п. 2.5.2). 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 ранги по декартовым координатам); кроме того, предусмотрена возможность выделения подрешеток меньшей размерности (причем каждая подрешетка автоматически связывается с новым коммуникатором); также имеется функция MPI_Cart_shift, которая упрощает пересылку сообщений вдоль некоторой координаты декартовой решетки.

Что же касается топологии графа, то после ее определения с помощью функции MPI_Graph_create можно лишь восстанавливать ее характеристики (используя функции MPI_Graphdims_get и MPI_Graph_get), а также получать информацию о количестве и рангах всех процессов-*соседей* (neighbors) некоторого процесса в графе, определяемом данной топологией (для этого предназначены функции MPI_Graph_neighbors_count и MPI_Graph_neighbors).

Для знакомства с возможностями, связанными с топологией графа, выполним 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 — см. рис. 23). 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.

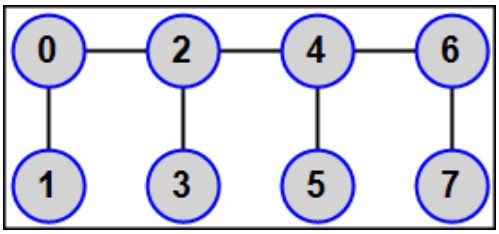

Рис. 23. Пример топологии графа из задания MPI5Comm29

При ознакомительном запуске этого задания окно задачника примет вид, соответствующий приведенному на рис. 24. Одновременно с окном задачника в правом верхнем углу экрана отобразится рисунок из формулировки задания, иллюстрирующий используемую топологию.

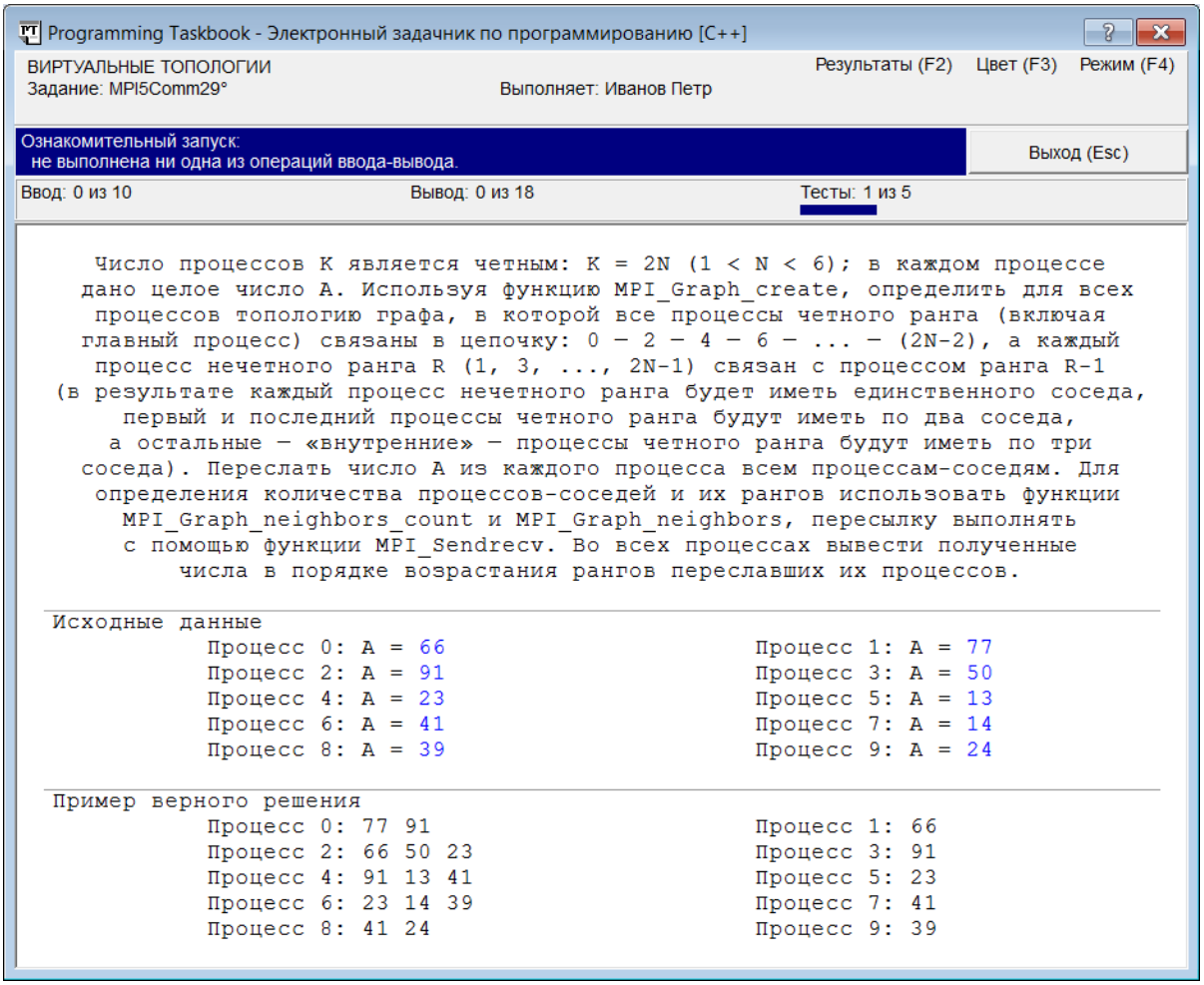

Рис. 24. Ознакомительный запуск задания MPI5Comm29

Для большей наглядности изобразим процессы вместе с их исходными данными в виде графа той структуры, которая описана в формулировке задания (рис. 25).

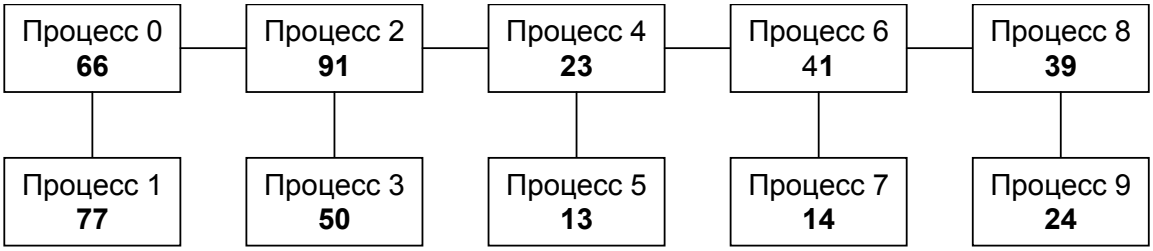

Рис. 25. Вариант исходных данных для задания MPI5Comm29

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:

MPI_Comm oldcomm – исходный коммуникатор, для процессов которого определяется топология графа;

int nnodes – число вершин графа;

int* index – целочисленный массив *степеней вершин*, *i*-й элемент которого равен суммарному количеству соседей для первых *i* вершин графа;

int* edges – целочисленный массив *ребер*, содержащий упорядоченный список ребер для всех вершин (вершины нумеруются от 0);

int reorder – целочисленный флаг, определяющий, можно ли среде MPI автоматически менять порядок нумерации процессов;

MPI_Comm graphcomm – результирующий коммуникатор с топологией графа (выходной параметр).

Как и при выполнении задания на декартову топологию (см. п. 1.2.8), следует запретить перенумерацию процессов, положив флаг reorder равным 0.

Чтобы лучше понять смысл параметров-массивов, задающих характеристики определяемого графа, перечислим их элементы для графа, приведенного на рис. 25. 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 каж-

дой 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 index и edges удобно отдельно обработать две первые (ранга 0 и 1) и две последние (ранга size – 2 и size – 1) вершины графа, а остальные перебирать в цикле, обрабатывая по две вершины (ранга 2 и 3, 4 и 5, …, size – 4 и size – 3) на каждой итерации. При этом будет 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 (поскольку 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 (рис. 26). Чтобы уменьшить размеры окна, на приведенном рисунке скрыт раздел с формулировкой задания.

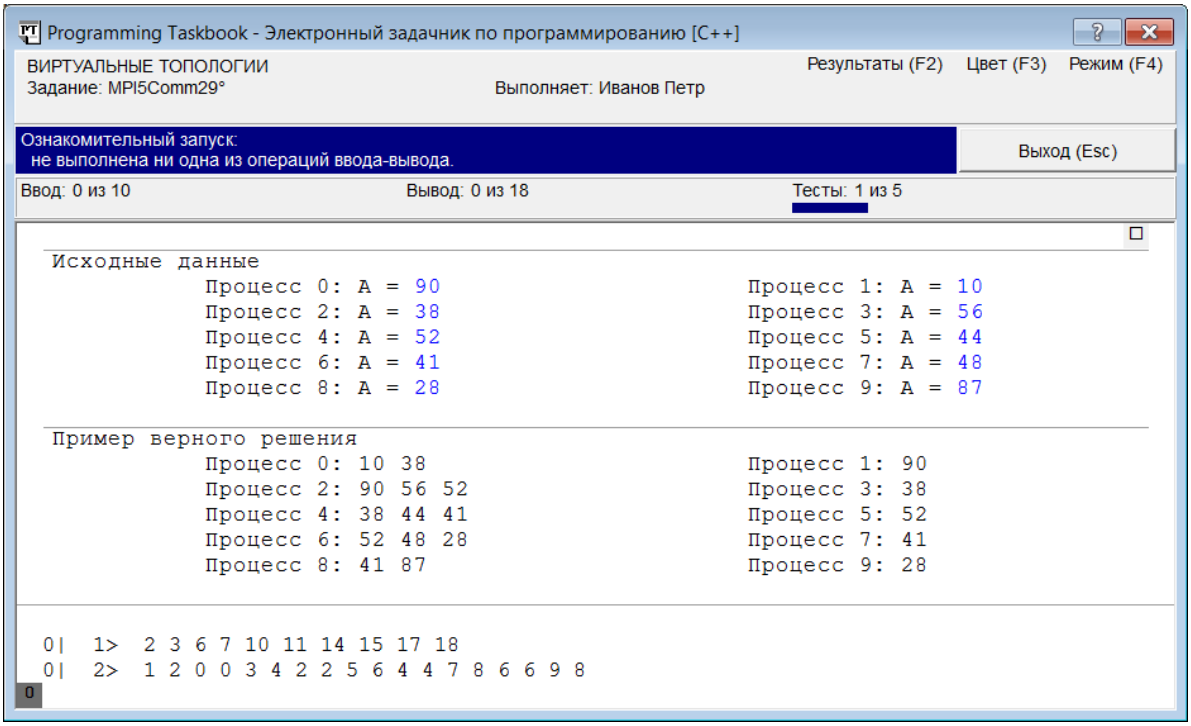

Рис. 26. Отладочный вывод элементов массива степеней вершин и массива ребер

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);
```

Заметим, что для получения характеристик существующего коммуникатора comm, связанного с топологией графа, в MPI предусмотрены две функции: MPI_Graphdims_get(MPI_Comm comm, int* nnodes, int* nedges) и MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int* index, int* edges). Первая из них позволяет определить число вершин nnodes и число ребер nedges графа для коммуникатора comm, а вторая возвращает массив степеней вершин index (размер массива указывается в переменной maxindex) и массив ребер edges (размер этого массива указывается в переменной maxedges). Функции MPI_Graphdims_get и MPI_Graph_get играют для топологии графа такую же роль, как функции MPI_Cartdim_get и MPI_Cart_get для декартовой топологии.

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, после чего организовать обмен данными между текущим процессом и каждым из его соседей.

Для определения количества count соседей процесса ранга rank, входящего в коммуникатор comm с топологией графа предусмотрена функция MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int* count). Зная количество соседей, можно выделить память соответствующего размера для массива neighbors рангов соседей и определить эти ранги с помощью функции MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int* neighbors), где параметр maxneighbors задает размер массива neighbors. Заметим, что с помощью этих функции любой процесс может определить не только своих соседей, но и соседей любого другого процесса из данного коммуникатора.

Для пересылки данных между процессом и его соседями, согласно формулировке задания, следует использовать функцию MPI_Sendrecv, которая обеспечивает как прием сообщения от некоторого процесса, так и отправку ему (или другому процессу) другого сообщения (см. п. 1.2.1).

Таким образом, завершающая часть решения примет следующий вид:

```
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.

1.3. Дополнительные возможности интерфейса MPI (стандарт MPI-2)

1.3.1. Топология распределенного графа

В стандарт MPI-2 был включен новый вид виртуальной топологии — *топология распределенного графа* (distributed graph topology). Ей посвящены задачи, входящие в третью подгруппу группы MPI5Comm (см. п. 2.5.3).

Распределенные графы имеют следующие основные отличия по сравнению с «обычными» графами (рассмотренными в п. 1.2.9):

- в то время как для определения топологии обычного графа необходимо полностью описать его структуру во всех процессах определяемого коммуникатора, распределенный граф можно определить «по частям», указав в каждом процессе только те компоненты структуры распределенного графа, которые «известны» этому процессу (именно по этой причине новый вид графа называется *распределенным* графом);
- ребра распределенного графа, в отличие от ребер обычного графа, являются *ориентированными*; каждое ребро соединяет процесс-источник (source) и процесс-приемник (destination); можно считать, что каждое ребро распределенного графа снабжается «стрелкой», определяющей преимущественное направление передачи данных;
- каждое ребро распределенного графа может снабжаться дополнительной числовой характеристикой — «*весом*» (weight), которая может тем или иным образом учитываться системой MPI при выборе оптимальных маршрутов передачи данных. Веса являются неотрицательными; кроме самих числовых значений с весами можно связывать дополнительную информацию, передаваемую параметром типа MPI_Info.

Для создания коммуникатора, связанного с топологией распределенного графа, в стандарте MPI-2 предусмотрены две функции: MPI_Dist_graph_create_adjacent и MPI_Dist_graph_create. Обе эти функции являются коллективными; они должны вызываться для всех процессов, входящих в исходный коммуникатор oldcomm.

При использовании функции MPI_Dist_graph_create_adjacent(MPI_Comm oldcomm, int indegree, int* sources, int* sourceweights, int outdegree, int* destinations, int* destweights, MPI_Info info, int reorder, MPI_Comm* distcomm) каждый процесс должен определить только то множество ребер графа, которое с ним связано, т. е. те ребра, относительно которых данный процесс является либо приемником, либо источником. Для первого набора ребер указывается их количество indegree и набор sources, содержащий ранги процессов-источников, для второго набора указывается его размер outdegree и набор destinations рангов процессов-приемников; кроме того, для каждого из этих наборов указываются значения весов ребер (наборы sourceweights и destweights соответственно). Если веса задавать не требуется, то в качестве параметров sourceweights и destweights надо указать константу MPI_UNWEIGHTED, причем во *всех* процессах коммуникатора oldcomm.

Разумеется, указываемая информация должна быть согласована для процессов-приемников и процессов-источников; в частности, и приемник, и источник должны указывать одинаковый вес для их общего ребра.

Смысл параметров info и reorder является одинаковым для обеих функций; они будут описаны далее, вместе с остальными параметрами функции MPI_Dist_graph_create.

Функция MPI_Dist_graph_create обеспечивает более гибкие возможности по определению топологии распределенного графа (заметим, что именно эту функцию требуется использовать в двух последних заданиях группы MPI5Comm). С помощью функции MPI_Dist_graph_create каждый процесс может определить *любую* часть создаваемого распределенного графа. Смысл ее параметров следующий:

MPI_Comm oldcomm – исходный коммуникатор, для процессов которого определяется топология графа;

int n – количество процессов-источников, ребра для которых определяются в данном процессе;

int* sources – массив размера n, содержащий ранги определяемых процессов-источников;

int* degrees – массив размера n, задающий количество процессов-приемников для каждого из определяемых процессов-источников;

int* destinations – массив, содержащий ранги всех процессов-приемников для определяемых процессов-источников (его размер равен сумме всех элементов массива degrees);

int* weights – массив, содержащий веса всех определяемых ребер (его размер также равен сумме всех элементов массива degrees), или константа MPI_UNWEIGHTED, если веса задавать не требуется;

MPI_Info info – дополнительная информация, связанная с определяемыми весами; если она не предусматривается, то в качестве этого параметра указывается константа MPI_INFO_NULL;

int reorder – целочисленный флаг, определяющий, можно ли среде MPI автоматически менять порядок нумерации процессов;

MPI_Comm distcomm – результирующий коммуникатор с топологией распределенного графа (выходной параметр).

Если используется константа MPI_UNWEIGHTED, то она должна указываться при вызове функции MPI_Dist_graph_create во *всех* процессах коммуникатора oldcomm.

При определении распределенного графа с помощью функции MPI_Dist_graph_create можно придерживаться различных стратегий. Проиллюстрируем их на примере простого распределенного графа, приведенного на рис. 27 (считая для простоты, что веса в нем не указываются).

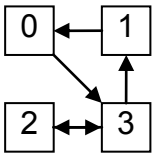

Рис. 27. Пример распределенного графа

Можно, например, для каждого процесса определить только те ребра, для которых он является источником:

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

Другой способ (который требуется применить в заданиях группы MPI5Comm) состоит в том, что граф полностью определяется в каком-либо одном процессе:

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

Возможны и любые промежуточные варианты.

Приведем вариант определения того же распределенного графа с помощью функции MPI_Dist_graph_create_adjacent:

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

Для топологии распределенного графа, как и для топологии обычного графа, предусмотрены две функции, позволяющие восстановить его характеристики. Однако, в отличие от соответствующих функций для топологии обычного графа, которые позволяют *любому* процессу получить полную информацию о соседях *любого* процесса, функции для топологии распределенного графа возвращают информацию о соседях *того процесса, в котором они вызваны*.

Функция MPI_Dist_graph_neighbors_count(MPI_Comm comm, int* indegree, int* outdegree, int* weighted) позволяет определить число процессов-источников и процессов-приемников (выходные параметры indegree и outdegree соответственно). Кроме того, выходной параметр-флаг weighted принимает ненулевое значение, если при определении топологии распределенного графа *не использовалась* константа MPI_UNWEIGHTED.

Функция MPI_Dist_graph_neighbors(MPI_Comm comm, int maxindegree, int* sources, int* sourceweights, int maxoutdegree, int destinations, int* destweights) возвращает информацию о рангах процессов-источников и процессов-приемников (выходные массивы sources и destinations размера maxindegree и maxoutdegree соответственно). Кроме того, если функция MPI_Dist_graph_neighbors_count вернула ненулевой флаг weighted, то в массивах sourceweights и destweights возвращаются веса ребер, связанных с процессами-источниками и процессами-приемниками.

1.3.2. Параллельный ввод-вывод. Функции файлового доступа

Одним из важных нововведений стандарта MPI-2 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 ввода-вывода, 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 — см. п. 1.2.1 и 1.2.3).

Таким образом, комбинируя описанные выше механизмы, можно получить 24 (= 2·2·3·2) 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). Простейшие варианты функций файлового чтения и записи

имеют имена MPI_File_read и MPI_File_write; эти функции являются *блокирующими*, *локальными* и используют *индивидуальные файловые указатели*.

В таблице 1 приведены имена всех функций, связанных с файловым доступом. Функции сгруппированы по описанным выше категориям: «Позиционирование» (явное позиционирование, локальный файловый указатель или общий файловый указатель), «Доступ» (блокирующий или неблокирующий), «Координация» (локальные или коллективные функции).

Таблица 1

Функции MPI, обеспечивающие файловый доступ

Позиционирование	**Доступ**	**Координация (ввод и вывод)**	
		Локальные функции	*Коллективные функции*
Явное	*Блокир.*	MPI_File_read_at MPI_File_write_at	MPI_File_read_at_all MPI_File_write_at_all
	Неблокир.	MPI_File_iread_at MPI_File_iwrite_at	MPI_File_read_at_all_begin MPI_File_read_at_all_end MPI_File_write_at_all_begin MPI_File_write_at_all_end
Локальный файловый указатель	*Блокир.*	MPI_File_read MPI_File_write	MPI_File_read_all MPI_File_write_all
	Неблокир.	MPI_File_iread MPI_File_iwrite	MPI_File_read_all_begin MPI_File_read_all_end MPI_File_write_all_begin MPI_File_write_all_end
Общий файловый указатель	*Блокир.*	MPI_File_read_shared MPI_File_write_shared	MPI_File_read_ordered MPI_File_write_ordered
	Неблокир.	MPI_File_iread_shared MPI_File_iwrite_shared	MPI_File_read_ordered_begin MPI_File_read_ordered_end MPI_File_write_ordered_begin MPI_File_write_ordered_end

Опишем более подробно блокирующие функции, поскольку именно их требуется использовать в учебных заданиях.

Блокирующие функции с явным позиционированием (MPI_File_read_at, MPI_File_write_at, MPI_File_read_at_all, MPI_File_write_at_all) имеют следующий список параметров:

MPI_File f – файловая переменная (дескриптор файла);

MPI_Offset offset – файловая позиция, с которой начинается считывание/запись данных;

void* buf – буфер для хранения прочитанных (для функций чтения) или записываемых данных (для функций записи); для функций чтения является выходным параметром;

int count – размер буфера buf (в элементах типа datatype);

MPI_Datatype datatype – тип элементов буфера buf;

MPI_Status* status – дополнительная информация о выполненной операции чтения/записи (выходной параметр).

Файловый дескриптор f типа MPI_File определяется при открытии файла, которое выполняется функцией MPI_File_open. Данная функция является коллективной и должна вызываться для всех процессов некоторого коммуникатора; в результате все процессы этого коммуникатора получают доступ к файлу. Подробное описание функции MPI_File_open приводится в п. 1.3.3.

Тип MPI_Offset аналогичен ранее рассмотренному типу MPI_Aint (см. п. 1.2.6); он предназначен для хранения смещений между различными файловыми позициями и реализован как знаковый целочисленный тип, размер которого является достаточным для хранения любого возможного смещения в дисковом адресном пространстве.

С помощью выходного параметра status можно определить количество фактически прочитанных/записанных элементов (для этого, как обычно, надо использовать функцию MPI_Get_count — см. п.), а также получить код ошибки (используя поле поле MPI_ERROR). Прочие поля структуры MPI_Status в функциях файлового доступа не используются.

Вызов функций с явным позиционированием никак не влияет на текущую позицию локального и общего файлового указателя.

Все прочие блокирующие функции (MPI_File_read, MPI_File_write, MPI_File_read_all, MPI_File_write_all, MPI_File_read_shared, MPI_File_write_shared, MPI_File_read_ordered, MPI_File_write_ordered) имеют одинаковый набор параметров, отличающийся от приведенного выше лишь отсутствием параметра offset. При использовании этих функций начальная позиция для считывания/записи файловых данных определяется позицией локального или общего файлового указателя.

Позиция используемого в функции файлового указателя автоматически изменяется при выполнении операций чтения/записи. Кроме того, эту позицию можно задать явно с помощью функций MPI_File_seek (для локального указателя) и MPI_File_seek_shared (для общего указателя). Эти функции имеют одинаковый набор параметров:

MPI_File f – файловая переменная;

MPI_Offset offset – требуемое смещение файлового указателя (может быть как положительным, так и отрицательным);

int whence – используемый режим смещения.

Для указания режима смещения whence предусмотрены три константы:

MPI_SEEK_SET – смещение отсчитывается от начальной позиции вида файловых данных;

MPI_SEEK_CUR – смещение отсчитывается от текущей позиции файлового указателя;

MPI_SEEK_END – смещение отсчитывается от метки конца файла.

В режиме MPI_SEEK_SET допускаются только положительные значения параметра offset, в режиме MPI_SEEK_END – только отрицательные.

Для определения текущей позиции локального и общего файлового указателя предусмотрены функции MPI_File_get_position и MPI_File_get_position_shared, также имеющие одинаковый набор параметров:

MPI_File f – файловая переменная;

MPI_Offset* offset – текущая позиция файлового указателя (выходной параметр).

Необходимо учитывать, что при использовании *локальных* функций, связанных с *общим* файловым указателем (MPI_File_read_shared и MPI_File_write_shared) дополнительная синхронизация не выполняется, т. е. порядок чтения данных будет зависеть от момента времени, в который вызывались функции в различных процессах.

Коллективные функции, связанные с *общим* файловым указателем (MPI_File_read_ordered и MPI_File_write_ordered) обеспечивают требуемую синхронизацию. При их вызове (который, как для любых коллективных функций, должен быть выполнен во *всех* процессах коммуникатора, для которого был открыт файл) действия по чтению/записи данных с использованием общего указателя выполняются последовательно, в порядке возрастания рангов процессов в данном коммуникаторе.

Для всех рассмотренных выше функций позиция offset указывается в *элементах базового типа файла* etype и вычисляется с учетом *начального смещения* disp, заданного при определении вида файловых данных (см. описание функции MPI_File_set_view в п. 1.3.3).

Для того чтобы преобразовать позицию offset (в элементах базового типа) в смещение disp в *байтах*, отсчитываемое от начала файла, предусмотрена вспомогательная функция MPI_File_get_byte_offset(MPI_File f, MPI_Offset offset, MPI_Offset* disp).

В некоторых ситуациях оказывается полезной функция MPI_File_get_size (MPI_File f, MPI_Offset* size), возвращающая в параметре size размер файла f в байтах. «Парной» к ней является функция MPI_File_set_size(MPI_File f, MPI_Offset size), позволяющая изменить размер файла f, положив его равным size байтам. Допускается как уменьшать размер файла (при этом удаляется его конечная часть), так и увеличивать его (при этом содержимое добавленной конечной области является неопределенным). Функция MPI_File_set_size является коллективной и должна быть вызвана (с одинаковым значением параметра size) всеми процессами коммуникатора, в котором был открыт файл.

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 на-

бор последовательно расположенных байтов, до весьма сложных, в которых файловый вид может включать группы 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. Эти возможности подробно обсуждаются в следующем пункте.

При выполнении заданий группы MPI6File (см. п. 2.6) 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 общих 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.

1.3.3. Параллельный ввод-вывод: пример. Настройка вида файловых данных

В качестве примера задания, связанного с параллельным файловым вводом-выводом, рассмотрим задание MPI6File26, входящее в третью подгруппу группы MPI6File (см. п. 2.6.3).

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 указанным 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. 28.

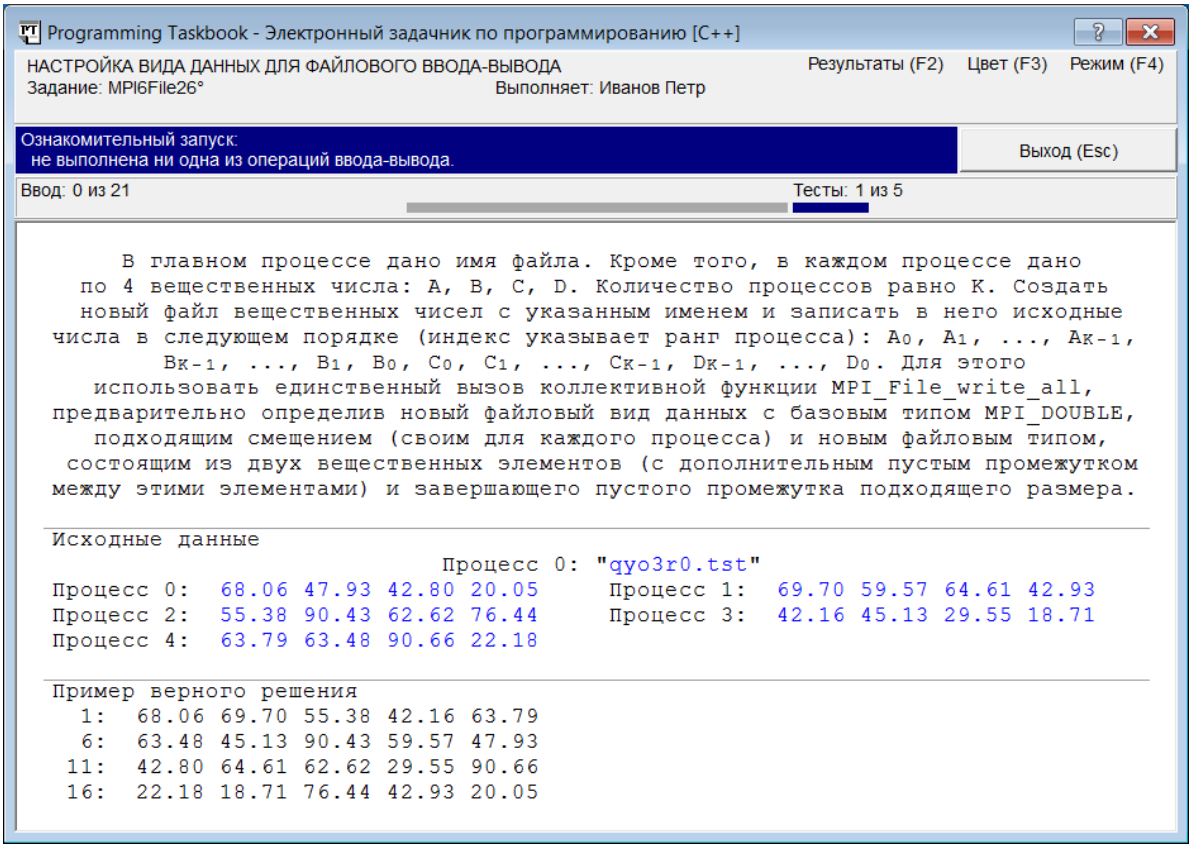

Рис. 28. Ознакомительный запуск задания MPI6File26

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 приложения, проще всего воспользоваться коллективной функцией MPI_Bcast. В преамбуле к группе MPI6File указано, что для хранения имени файла достаточно использовать массив char[12]. При пересылке этого массива следует использовать 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 создать новый файл или преобразовать имеющийся). Файловые данные отображаются в окне задачника на нескольких строках, причем в начале каждой строки указыва-

ется 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* в разделе индикаторов (расположенном над разделом с формулировкой задания). Данный индикатор отображается серым цветом и не содержит сопутствующего текста. Это означает что, результаты, полученные при выполнении задания, не нужно пересылать задачнику (используя поток вывода pt); необходимо лишь записать требуемые данные в 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 нового варианта программы в окне задачника будет выведено сообщение о том, что все исходные данные введены, а результирующий файл не создан (рис. 29).

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

```
MPI_File f;
MPI_File_open(MPI_COMM_WORLD, name,
  MPI_MODE_CREATE | MPI_MODE_WRONLY, MPI_INFO_NULL, &f);
// доступ к содержимому файла f
MPI_File_close(&f);
```

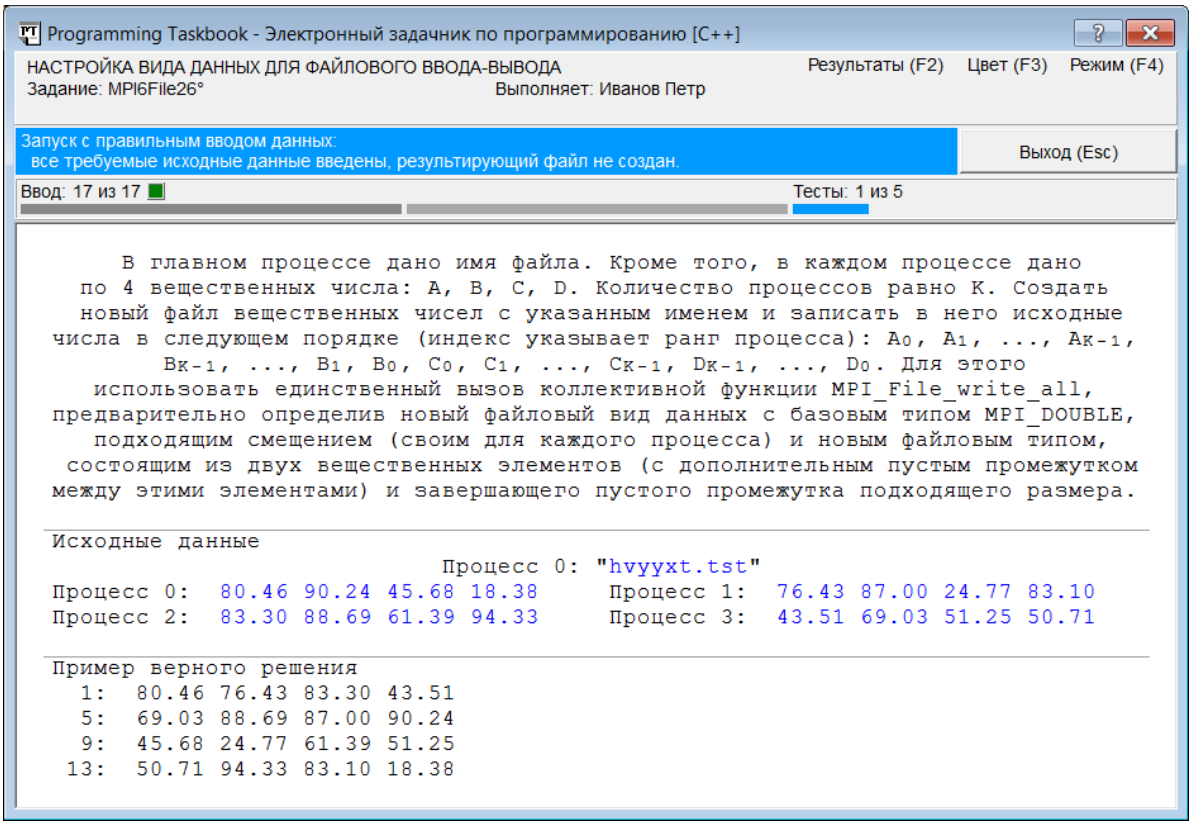

Рис. 29. Первый этап выполнения задания MPI6File26 (ввод исходных данных)

Вначале описывается файловая переменная f типа MPI_File, которая будет использоваться во всех функциях, связанных с файловыми операциями. Затем для открытия файла вызывается функция MPI_File_open(MPI_Comm comm, char* filename, int amode, MPI_Info info, MPI_File* f). Эта функция является коллективной и должна вызываться во всех процессах коммуникатора comm, указанного в качестве ее первого параметра. Имя файла filename указывается во втором параметре, а режим доступа amode — 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 *только для записи* (MPI_MODE_WRONLY). Приведем еще некоторые из имеющихся характеристик режима доступа:

MPI_MODE_RDONLY – открытие только для чтения;

MPI_MODE_RDWR – открытие на чтение и запись;

MPI_MODE_APPEND – автоматическое перемещение всех файловых указателей (как локальных, так и общего) в конец файла при его открытии;

MPI_MODE_DELETE_ON_CLOSE – автоматическое удаление файла при его закрытии.

Параметр info 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.

Последний параметр функции MPI_File_open представляет собой файловую переменную f, связанную с открытым файлом. Этот параметр является выходным, поэтому должен передаваться с помощью указателя.

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

Чтобы максимально упростить действия по заполнению файла требуемыми данными, необходимо предварительно настроить *вид файловых данных*. Для этого предусмотрена функция MPI_File_set_view с большим количеством параметров: (MPI_File f, MPI_Offset disp, MPI_Datatype etype, MPI_Datatype filetype, char* datarep, MPI_Info info).

В качестве первого параметра f 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 элементов 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), можно 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 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*d, где d 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 2: [*](2*K–6)[*](4)
…
процесс K–1: [*][*](2*K–2)

Обратите внимание на то, что *протяженность* файлового типа для всех процессов является одинаковой и равна 2**K* (в элементах типа MPI_DOUBLE); одинаковым является и суммарный размер пустых промежутков, равный 2**K*–2 (также в элементах типа MPI_DOUBLE).

При определении типов данных, содержащих начальные или конечные пустые промежутки, будем использовать новые удобные и гибкие средства, появившиеся в стандарте MPI-2 (другой подход, основанный на применении возможностей стандарта MPI-1, описан в п. 1.2.6).

Напомним, что в нашей программе число процессов *K* хранится в переменной size, а ранг процесса *R* — в переменной rank. Размер типа MPI_DOUBLE найдем с помощью функции MPI_Type_size и сохраним в переменной dbl_sz.

Вначале определим вспомогательный тип t0, содержащий два вещественных числа с необходимым промежутком между ними, а затем дополним этот тип завершающим промежутком с помощью функции MPI_Type_create_resized (см. п. 1.2.6), получив итоговый тип t:

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

После определения типа t мы сможем его использовать в функции MPI_File_set_view:

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

Теперь для записи в файл требуемых данных нам достаточно выполнить *единственный* вызов коллективной функции MPI_File_write_all, передав в нее массив a из 4 введенных вещественных чисел:

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

Действия по настройке вида файловых данных и записи данных в файл надо выполнять между операциями открытия и закрытия файла (данная позиция нашей программы была помечена комментарием «*доступ к содержимому файла f*»).

Приведем полный текст полученного решения:

```
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.

1.3.4. Односторонние коммуникации: общее описание

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* (shared memory), после чего любой другой поток сможет обратиться к этой памяти и получить требуемую часть данных. 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 приложения, называется *окном* (window).

Для **создания окна** предназначена коллективная функция MPI_Win_create(void* base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win* win), которая должна вызываться для всех процессов коммуникатора comm. 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* win типа MPI_Win, который возвращается в последнем параметре функции. Используя данный дескриптор, любой процесс может получить доступ к окну любого другого процесса.

Начальный адрес base окна для данного процесса и его размер size (в байтах) указываются в качестве первых двух параметров функции MPI_Win_create. 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.

Третий параметр функции MPI_Win_create, имеющий имя disp_unit, предназначен для упрощения адресной арифметики, используемой при вызовах функций доступа к окну (описываемых далее): величина смещения от начала окна, указываемая в функциях доступа, автоматически умножается

на значение disp_unit. Если окно предназначено для хранения элементов некоторого массива, то в качестве параметра disp_unit удобно указать размер элемента массива. Это позволит в дальнейшем для обращения к тому или иному элементу массива в функциях доступа указывать его *индекс*. Если положить параметр disp_unit равным 1, то смещение в функциях доступа придется указывать в *байтах*.

Четвертый параметр info позволяет связать с создаваемым окном дополнительную информацию; обычно он не используется и в этом случае полагается равным константе MPI_INFO_NULL.

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.

Большинство параметров всех функций доступа (MPI_Get, MPI_Put, MPI_Accumulate) являются одинаковыми. Первые три параметра определяют данные «на стороне» инициирующего процесса: это void *origin_addr, int origin_count, MPI_Datatype origin_datatype. Первый из них указывает адрес начала буфера данных, второй — количество элементов в буфере, а третий — тип элементов буфера. Следующие четыре параметра определяют данные «на стороне» целевого процесса: int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype 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).

Если типы элементов в буфере инициирующего процесса и в окне целевого процесса совпадают (как чаще всего и происходит), то должны совпадать и указываемые размеры (origin_count и target_count).

Последним параметром во всех функциях доступа указывается дескриптор окна win типа MPI_Win.

Функция MPI_Accumulate имеет те же параметры и один дополнительный параметр op типа MPI_Op, указываемый перед параметром win. Параметр op определяет операцию, используемую для изменения содержимого окна: данная операция применяется к элементу буфера в инициирующем процессе и соответствующему элементу окна в целевом процессе, а результат операции записывается в этот же элемент окна.

В качестве операции op можно использовать любую стандартную операцию редукции, определенную в библиотеке MPI (см. п. 1.2.5). Операции, определенные пользователем, применять нельзя.

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 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).

При вызове функции MPI_Get изменяется содержимое буфера в инициирующем процессе, при вызове функций MPI_Put и MPI_Accumulate изменяется содержимое указанного участка окна в целевом процессе.

Важным 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 называется *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(int assert, MPI_Win win) (английское слово «fence» может быть переведено как «ограда», или «забор»). Это коллективная функция, которая должна вызываться во всех процессах, в которых определено окно win.

Кроме основного параметра win, определяющего окно, для которого устанавливается 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 коммуникациях. Если характер периода доступа не требуется уточнять, то в качестве параметра assert указывается число 0. Заметим, что параметр assert входит и в другие функции синхронизации (описываемые далее).

Первый вызов функции MPI_Win_fence 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.

Примечание 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);
```

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

Выход из функции MPI_Win_wait означает, что все инициирующие процессы завершили период доступа, вызвав функцию MPI_Win_complete.

Примечание 2. Предусмотрен также неблокирующий вариант функции MPI_Win_wait — это функция MPI_Win_test, имеющая дополнительный выходной параметр flag. Функция MPI_Win_test(MPI_Win win, int* flag) возвращает значение flag, не равное 0, если все инициирующие процессы завершили период доступа, что, в свою очередь, означает завершение периода предоставления доступа. Повторно вызывать функцию MPI_Win_test в пределах данного периода предоставления доступа следует только в случае, если ее предыдущий вызов вернул значение flag, равное 0. В учебных заданиях функция MPI_Win_test не используется.

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 предназначены только для начала и окончания периода доступа в инициирующих процессах. Для начала блокирующего периода

доступа инициирующий процесс должен вызвать функцию MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win), для завершения блокирующего периода доступа предназначена функция MPI_Win_unlock(int rank, MPI_Win win). Параметр rank определяет целевой процесс. Блокировка может быть *эксклюзивной* (exclusive lock) или *совместной* (shared lock); вид блокировки задается параметром lock_type, который может принимать значение одной из двух констант: MPI_LOCK_EXCLUSIVE и MPI_LOCK_SHARED. 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 блокировки, 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, п. 2.7.1) 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). 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 (см. п. 2.7.2) 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 задачах MPI7Win28 и MPI7Win30 требуется использовать оба варианта синхронизации, рассматриваемых во второй подгруппе.

Чтобы продемонстрировать различные возможности, связанные с односторонними коммуникациями, в следующих двух пунктах мы рассмотрим по одной задаче из каждой подгруппы группы MPI7Win.

1.3.5. Односторонние коммуникации: пример с использованием простейшего варианта синхронизации

Начнем с задачи MPI7Win13, в которой используется простейшая синхронизация, основанная на функции MPI_Win_fence, и вместе с тем применяется наиболее сложная из функций доступа — MPI_Accumulate.

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. 30). Для уменьшения размеров окна в нем скрыт раздел с формулировкой задания.

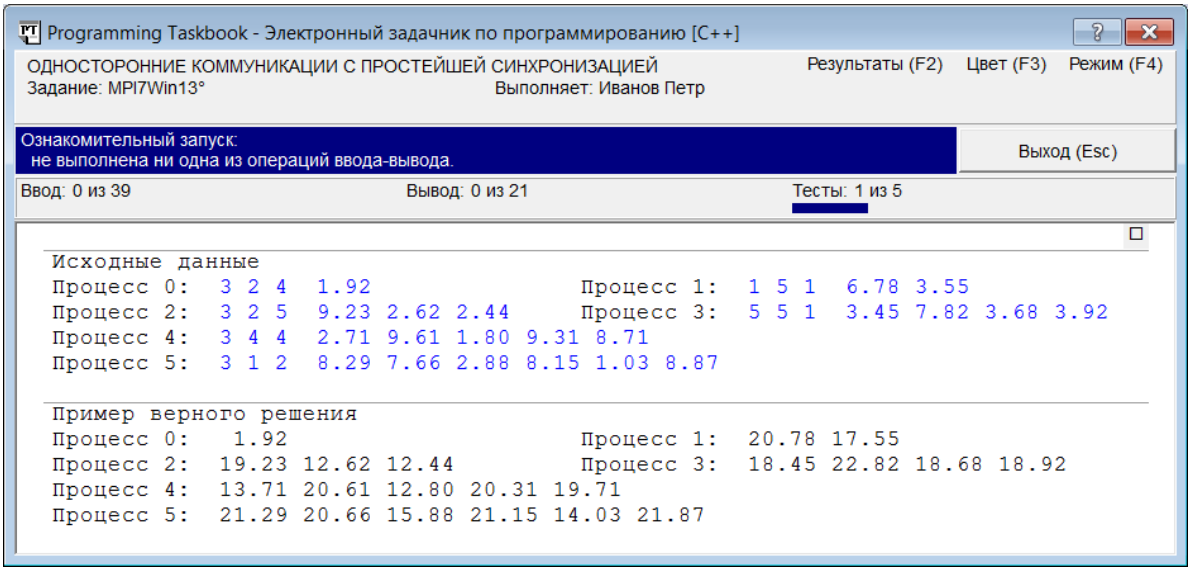

Рис. 30. Ознакомительный запуск задания MPI7Win13

Требуемое в задании преобразование исходных массивов A 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 ме-

ханизма 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. 30, 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 каждый элемент массива из процесса 5 будет увеличен на число 13 = 2 + 3 + + 4 + 4).

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 N_1, N_2 и N_3, 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 процес-

са. Приведем соответствующий фрагмент кода, а затем прокомментируем его:

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

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. В п. 1.3.4 мы 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). В качестве третьего параметра функции MPI_Win_create мы указываем размер (в байтах) элемента массива, хранящегося в окне; это позволит в дальнейшем указывать в функциях доступа к окну *индекс* требуемого элемента массива. Четвертый параметр полагается равным константе MPI_INFO_NULL. В последнем параметре win 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(MPI_Win* win); 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(&win);
delete[] a;
delete[] b;
```

Результат запуска нового варианта программы не будет отличаться от прежнего. Все последующие дополнения к решению надо 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, следует начать такой период, вызвав функцию MPI_Win_fence:

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

Поскольку характер определяемого периода доступа уточнять не требуется, в качестве первого параметра указано число 0.

После начала периода доступа можно вызывать функции доступа к окну. В данном случае надо 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, win);
```

Напомним, что первые три параметра любой функции доступа к окну определяют данные «на стороне» инициирующего процесса: адрес начала буфера данных, количество элементов в буфере и тип элементов буфера. Следующие четыре параметра определяют данные «на стороне» целевого процесса: ранг целевого процесса, смещение от начала окна в этом целевом процессе, количество и тип элементов окна, к которым выполняется доступ. Предпоследний параметр определяет операцию, используемую при изменении элементов окна. Поскольку в нашем случае требуется добавить к исходным значениям элементов окна новые слагаемые, мы использовали операцию MPI_SUM.

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

Примечание. 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.

1.3.6. Односторонние коммуникации: пример более сложного варианта синхронизации

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 ($I = 0, \ldots, 4$) 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. 31 приведено окно задачника для данного задания. Как и для предыдущего задания, в окне скрыт раздел с формулировкой.

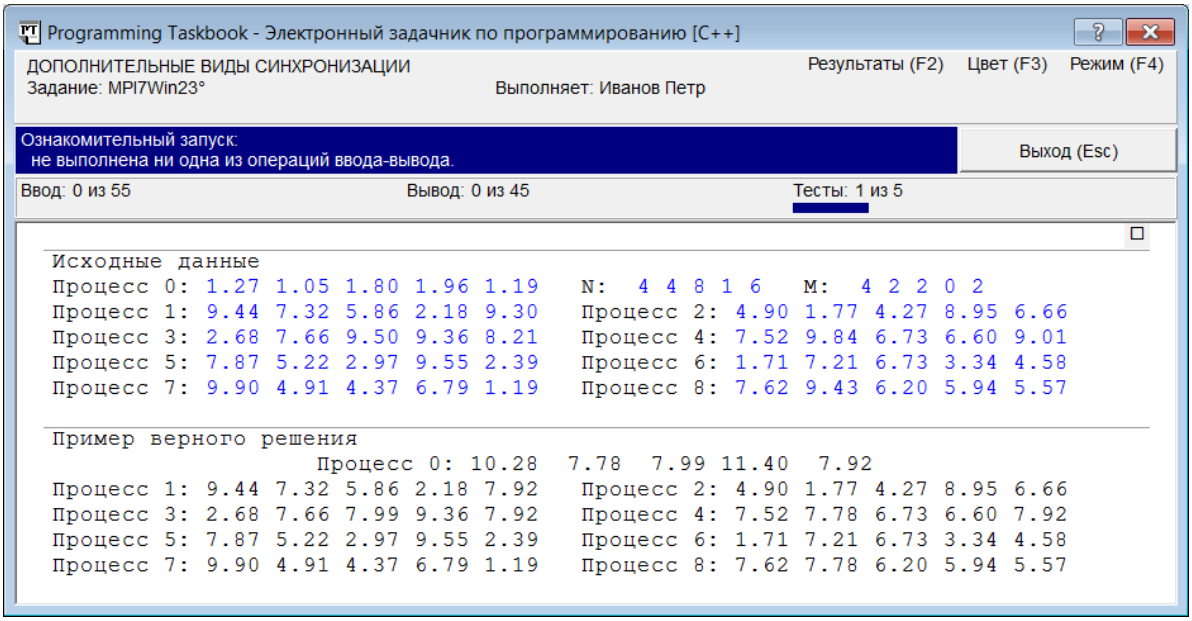

Рис. 31. Ознакомительный запуск задания MPI7Win23

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

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, win);
  // вызов функций доступа
  MPI_Win_complete(win);
  // вывод результатов
}
else
{
  MPI_Group_incl(g0, 1, &b, &g);
  MPI_Win_post(g, 0, win);
  MPI_Win_wait(win);
}
```

Напомним, что 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 получения данных из подчиненных процессов достаточно *перед* оператором MPI_Win_complete(win) (в позицию, помеченную комментарием «*вызов функций доступа*») добавить следующий 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, win);
```

В этом 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].

После оператора MPI_Win_complete(win) (в позицию, помеченную комментарием «*вывод результатов*») надо добавить фрагмент, обеспечивающий изменение и вывод массива a в главном процессе:

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

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

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

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. 32).

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.

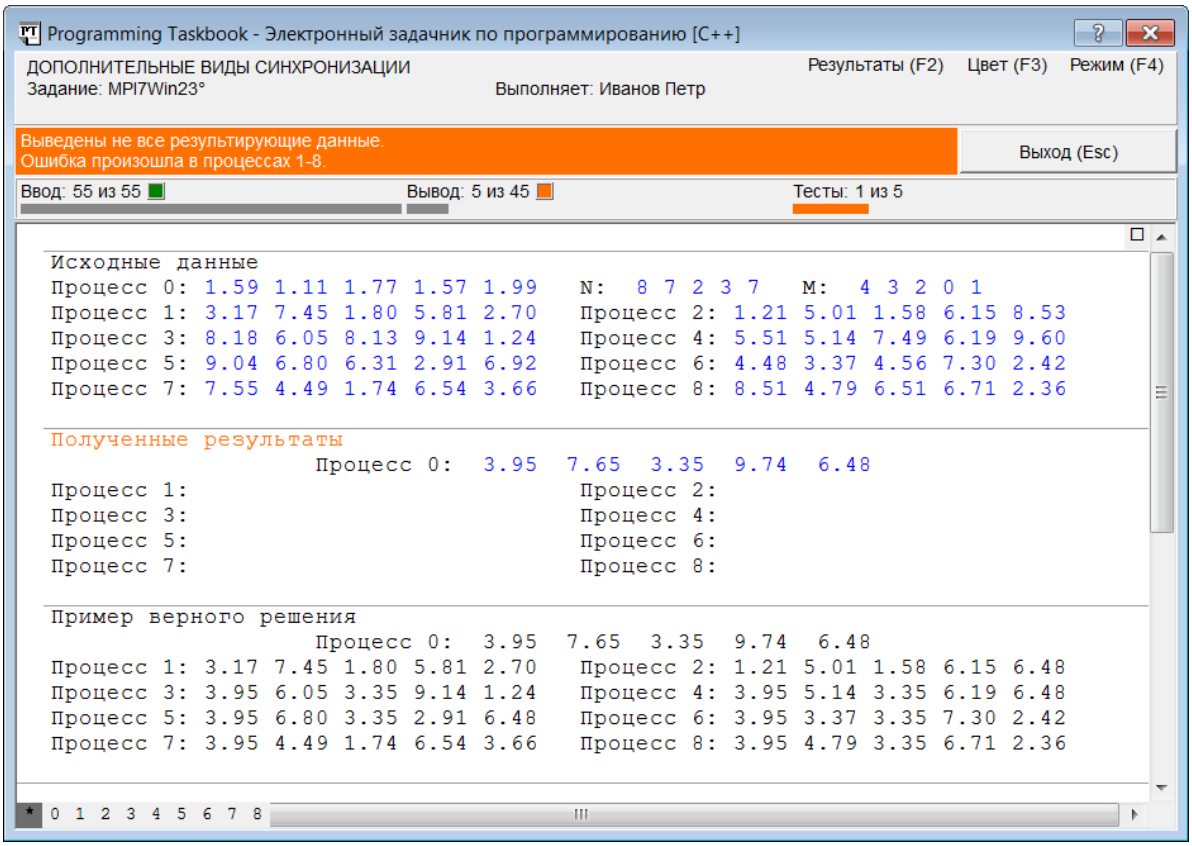

Рис. 32. Вид окна задачника после первого этапа выполнения задания MPI7Win23

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

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

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, win);
MPI_Win_wait(win);
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(&win).

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 win_sz = 5;
  int n[5], m[5];
  double a[5];
  for (int i = 0; i < 5; ++i)
    pt >> a[i];
  if (rank == 0)
  {
    win_sz = 0;
    for (int i = 0; i < 5; ++i)
      pt >> n[i];
    for (int i = 0; i < 5; ++i)
      pt >> m[i];
  }
  int dbl_sz;
  MPI_Type_size(MPI_DOUBLE, &dbl_sz);
  MPI_Win win;
  MPI_Win_create(&a, win_sz * dbl_sz, dbl_sz, MPI_INFO_NULL,
    MPI_COMM_WORLD, &win);
  MPI_Group g0, g;
  MPI_Comm_group(MPI_COMM_WORLD, &g0);
  int b = 0;
  if (rank == 0)
  {
    MPI_Group_excl(g0, 1, &b, &g);
    MPI_Win_start(g, 0, win);
    double a0[5];
```

```
    for (int i = 0; i < 5; ++i)
      MPI_Get(&a0[i], 1, MPI_DOUBLE, n[i], m[i], 1,
        MPI_DOUBLE, win);
    MPI_Win_complete(win);
    for (int i = 0; i < 5; ++i)
      a[i] += a0[i];
    MPI_Win_start(g, 0, win);
    for (int i = 1; i < size; ++i)
      MPI_Accumulate(a, 5, MPI_DOUBLE, i, 0, 5, MPI_DOUBLE,
        MPI_MIN, win);
    MPI_Win_complete(win);
  }
  else
  {
    MPI_Group_incl(g0, 1, &b, &g);
    MPI_Win_post(g, 0, win);
    MPI_Win_wait(win);
    MPI_Win_post(g, 0, win);
    MPI_Win_wait(win);
  }
  for (int i = 0; i < 5; ++i)
    pt << a[i];
  MPI_Win_free(&win);
}
```

1.3.7. Интеркоммуникаторы

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

Функция MPI_Comm_group(MPI_Comm comm, MPI_Group* group) в случае интеркоммуникатора comm возвращает его локальную группу group, а для получения удаленной группы предусмотрена функция MPI_Comm_remote_group с таким же набором параметров.

Проверить, является ли коммуникатор comm интеркоммуникатором, можно с помощью функции MPI_Comm_test_inter(MPI_Comm comm, int *flag), которая возвращает ненулевое значение параметра flag для интеркоммуникаторов и нулевое значение для интракоммуникаторов.

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. 33 (в этом окне скрыт раздел с формулировкой задания).

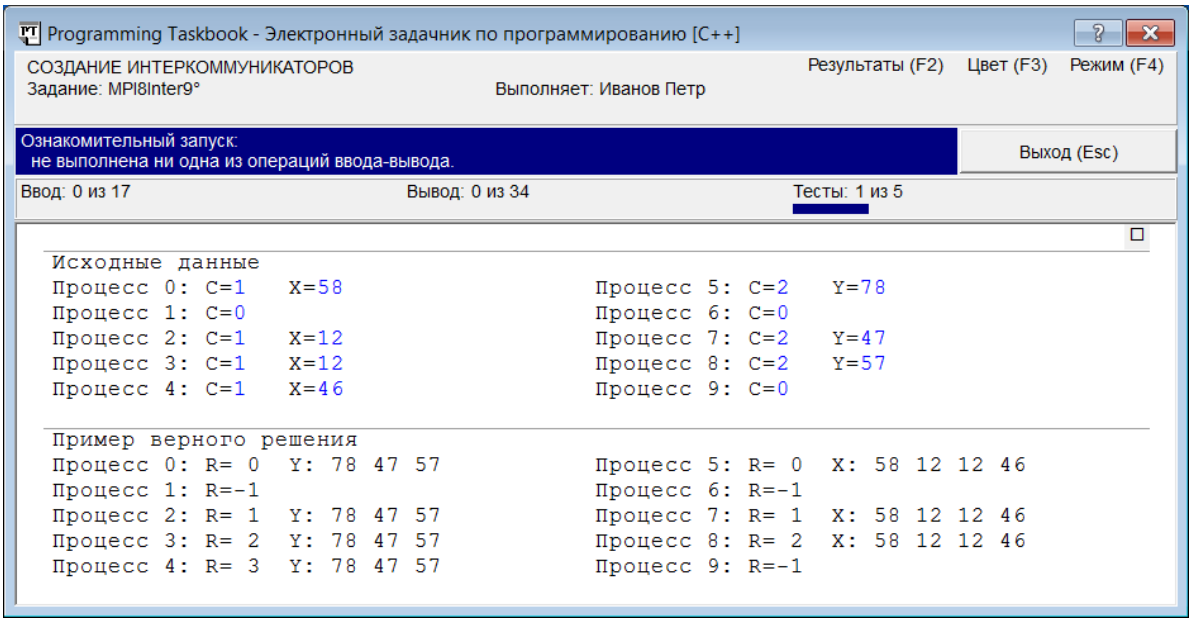

Рис. 33. Ознакомительный запуск задания MPI8Inter9

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. 1.2.7, 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 числа C мы сразу корректируем его значение в случае, если оно равно 0: нулевое значение заменяется на MPI_UNDEFINED, чтобы не создавать коммуникатор для процессов с $C = 0$. Новый коммуникатор, содержащий текущий процесс, связывается с переменной local (имя переменной показывает, что в дальнейшем 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. 34).

Приступим к основному этапу решения: определению 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) каж-

дой 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, желательно использовать *копию* коммуникатора MPI_COMM_WORLD, создав ее с помощью функции MPI_Comm_dup.

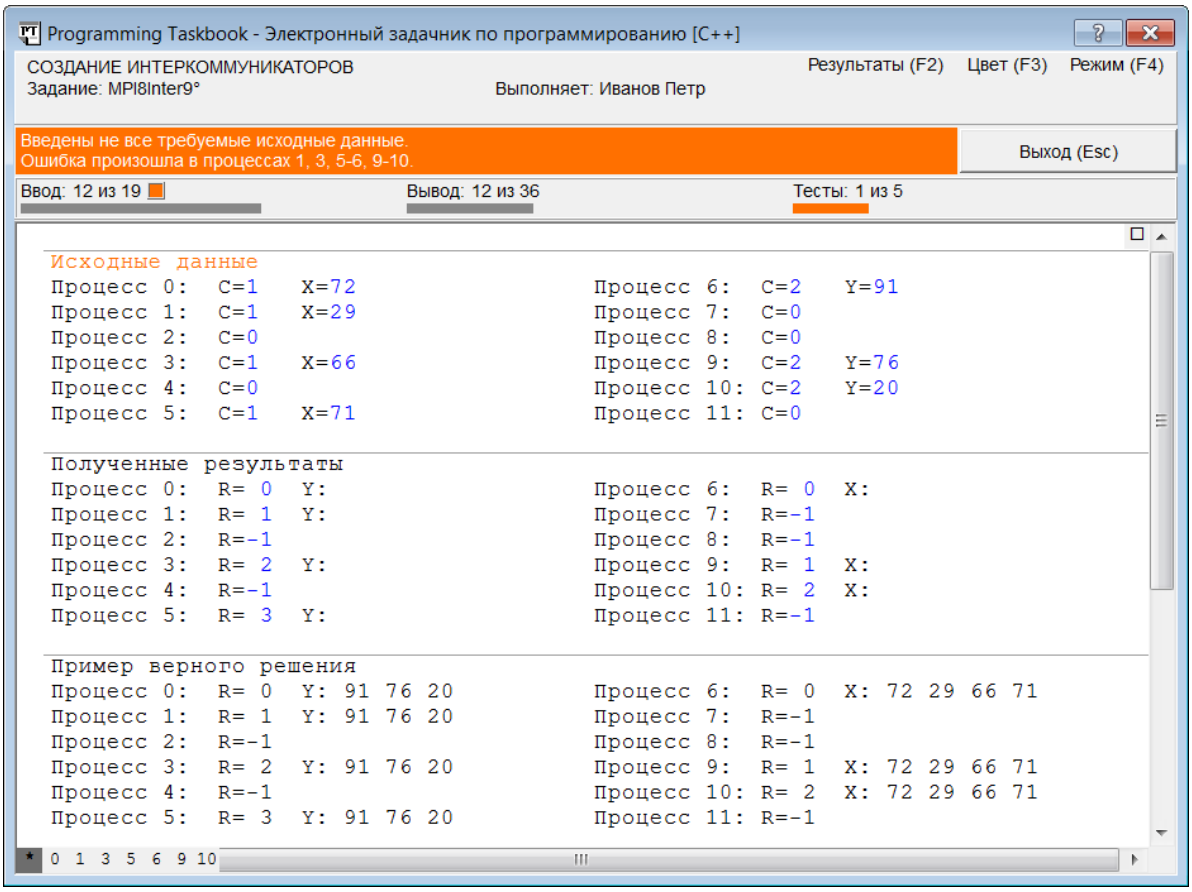

Рис. 34. Вид окна задачника после первого этапа выполнения задания MPI8Inter9

Перечислим параметры функции MPI_Intercomm_create:

MPI_Comm local – коммуникатор, связанный с локальной группой создаваемого интеркоммуникатора;

int local_leader – ранг представителя локальной группы (указывается ранг этого процесса в коммуникаторе local);

MPI_Comm peer_comm – коммуникатор-посредник; этот параметр учитывается только в том процессе, который является представителем локальной группы;

int remote_leader – 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;

int 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);

MPI_Comm* intercomm – указатель на созданный интеркоммуникатор (выходной параметр).

Таким образом, для создания интеркоммуникатора представитель каждой группы должен знать ранг представителя другой 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 вызывающего 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 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.

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 решения 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. 35; в данном случае для процессов с $C = 1$ размер удаленной группы равен 2, а для процессов с $C = 2$ размер равен 3).

Заключительный этап решения задачи является самым простым, поскольку в нем требуется использовать хорошо известные 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-полу-

чателя (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;
}
```

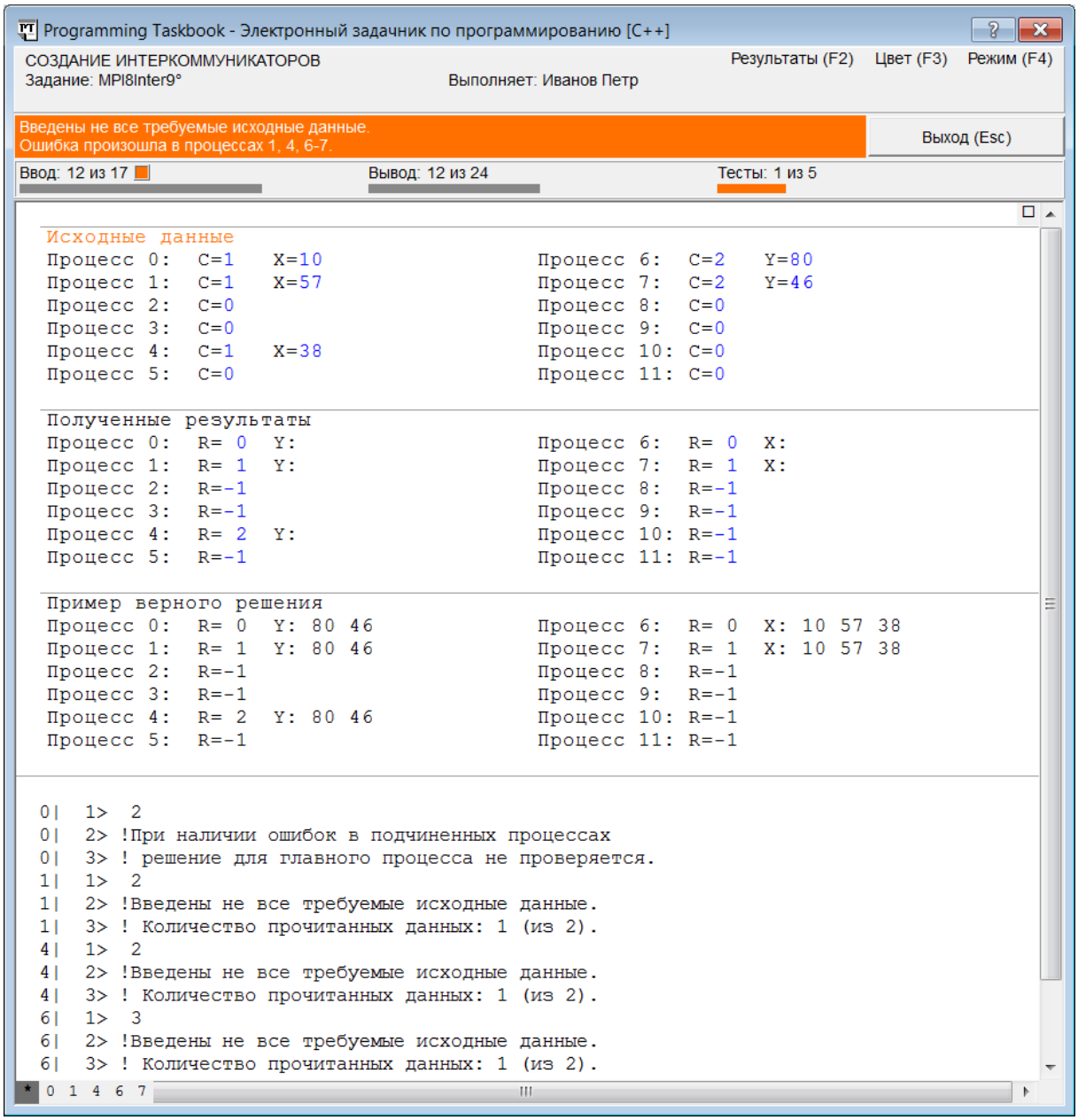

Рис. 35. Вид окна задачника после второго этапа выполнения задания MPI8Inter9

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.

Примечание 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).

В стандарте MPI-2 появился ряд новых возможностей, связанных с **созданием интеркоммуникаторов** (см. задания MPI8Inter5–MPI8Inter8). 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).

Если функция MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm* newcomm) вызывается для процессов интеркоммуникатора comm, то в процессах каждой группы интеркоммуникатора comm необходимо указать один и тот же параметр group, определяющий подмножество процессов из данной группы, которое станет соответствующей группой нового интеркоммуникатора newcomm. Данная функция должна быть вызвана во всех процессах, входящих в коммуникатор comm; при этом для тех процессов каждой группы, которые не входят в указанную группу group, в параметре newcomm будет возвращено значение MPI_COMM_NULL.

Гибкие возможности по созданию целого *семейства* новых интеркоммуникаторов с попарно непересекающимися группами процессов предоставляет функция MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm* newcomm). При ее вызове для интеркоммуникатора comm создается набор новых интеркоммуникаторов, в группы которых входят процессы исходного интеркоммуникатора с одинаковыми значениями параметра color (параметр key, как обычно, используется для определения порядка следования процессов в созданных группах — см. описание этого параметра для данной функции в п. 1.2.7). Если в одной из групп интеркоммуникатора comm отсутствуют процессы со значением color, указанным в каких-либо процессах другой группы, то в процессах с данным значением color возвращается MPI_COMM_NULL (это связано с тем, что интеркоммуникатор обязательно должен содержать две *непустые* группы процессов). Как обычно, значение MPI_COMM_NULL возвращается и для тех процессов, параметр color в которых имеет значение MPI_UNDEFINED.

Примечание 2. Следует 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 (см. п. 2.8.1).

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 (см. п. 1.2.4 и 1.2.5); следует лишь учитывать, что коллективные обмены всегда выполняются между различными группами интеркоммуникатора (иными словами, процессы одной группы обмениваются данными только с процессами другой группы).

Примечание 3. В случае, когда коллективные функции для интеркоммуникаторов используют особый процесс, задаваемый параметром root (таковы, например, функции MPI_Bcast, MPI_Scatter и MPI_Gather), применяются специальные правила. 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 той группы, в которую *не входит* особый процесс. В процессах той группы, которая содержит 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 группы — значение MPI_PROC_NULL. Примеры использования подобных коллективных функций будут приведены в следующем пункте при решении задачи MPI8Inter15.

1.3.8. Динамическое создание процессов

Рассмотренные в предыдущем пункте 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 *различные наборы* параметров командной строки и даже *различные исполняемые файлы*.

Функция MPI_Comm_spawn имеет следующий набор параметров:

char* command – строка, определяющая запускаемый файл (этот параметр учитывается только в процессе root — см. далее описание параметра root);

char** argv – строковый массив, 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);

int maxproc – максимальное количество запускаемых процессов (параметр учитывается только в процессе root);

MPI_Info info – дополнительная информация, связанная с запускаемым файлом (параметр учитывается только в процессе root); если дополнительная информация не используется, то в качестве info достаточно указать константу MPI_INFO_NULL;

int root – ранг того процесса из родительского коммуникатора comm, в котором обязательно должны быть указаны значения четырех предыдущих параметров;

MPI_Comm comm – родительский коммуникатор, обеспечивающий создание новой группы процессов;

MPI_Comm* intercomm – указатель на результирующий интеркоммуникатор (выходной параметр), одной из групп которого является группа всех процессов родительского коммуникатора comm, а другой — группа всех созданных процессов;

int* array_of_errcodes — целочисленный массив, содержащий коды ошибок, связанных с каждым из создаваемых процессов (выходной параметр). Если все процессы успешно созданы, то все элементы массива равны MPI_SUCCESS. 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. В свою очередь, выход из всех функций

MPI_Init в новых процессах произойдет одновременно с выходом из функций MPI_Comm_spawn в родительских процессах.

Функция MPI_Comm_spawn_multiple отличается от функции MPI_Comm_spawn лишь тем, что все настройки параметров запускаемых процессов задаются в *массивах*, каждый из которых имеет размер count (первый параметр функции). Приведем список парамеров данной функции: (int count, char** array_of_commands, char*** array_of_argv, int* array_of_maxprocs, MPI_Info* array_of_info, int root, MPI_Comm comm, MPI_Comm* intercomm, int* array_of_errcodes).

Все описываемые далее свойства функции MPI_Comm_spawn относятся и к функции MPI_Comm_spawn_multiple. В учебных заданиях, включенных в задачник PT for MPI-2, функция MPI_Comm_spawn_multiple не используется.

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 родительским процессам. Как получить доступ к этому интеркоммуникатору для дочерних процессов? Для этого предназначена специальная функция MPI_Comm_get_parent(MPI_Comm* parent). Если эту функцию вызвал один из дочерних процессов, т. е. процессов, созданных уже после запуска параллельного приложения, то в ее единственном параметре parent будет возвращен дескриптор интеркоммуникатора, связывающего родительские и дочерние процессы. Если же функция MPI_Comm_get_parent вызывается одним из исходных процессов параллельного приложения, то в параметре parent возвращается константа MPI_COMM_NULL.

Поскольку часто 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 возможностей рассмотрим первую задачу из той подгруппы группы MPI8Inter, которая посвящена созданию новых процессов (см. п. 2.8.3).

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. 36.

Следует обратить внимание на образец содержимого раздела отладки, приведенный в разделе исходных данных. 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, значения прочих настроек для новых групп процессов определены в преамбуле к группе MPI8Inter (см. п. 2.8).

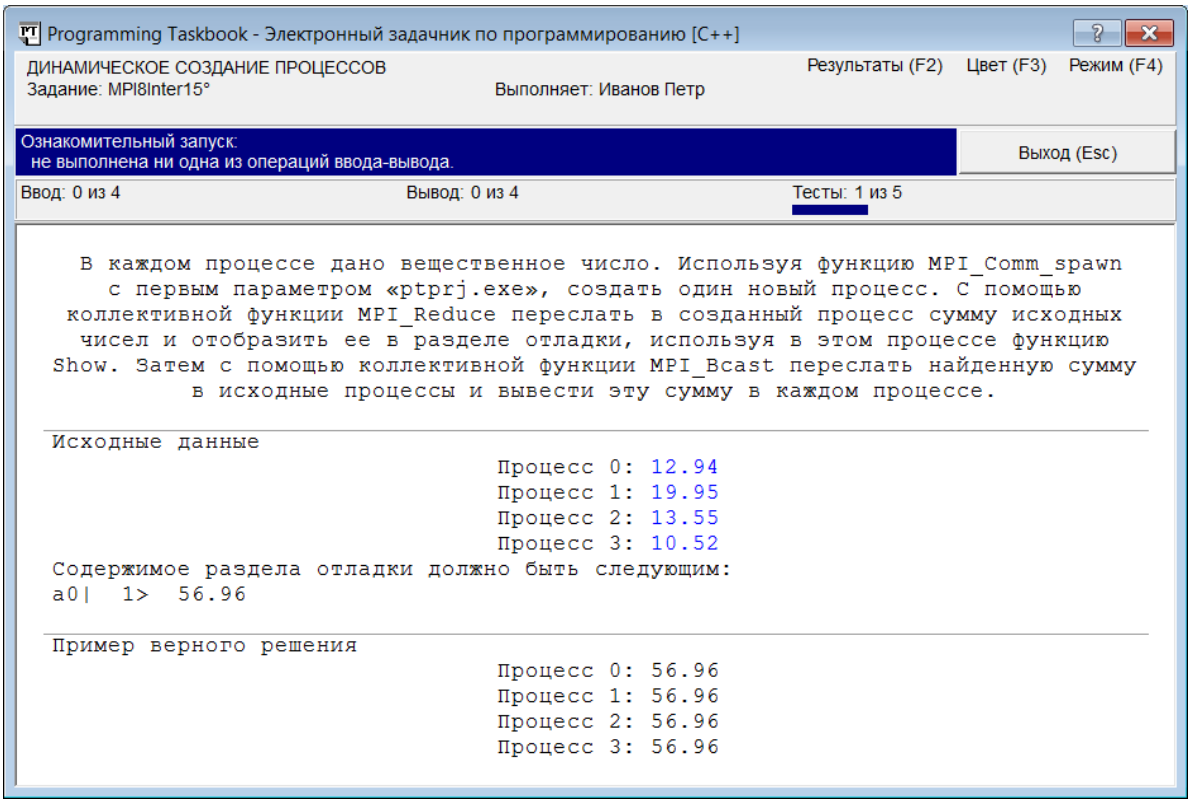

Рис. 36. Ознакомительный запуск задания MPI8Inter15

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. Заметим, что для созданных процессов, как и для исходных, не требуется явным образом вызывать функцию MPI_Init, так как этот вызов автоматически выполняет задачник перед вызовом в каждом процессе функции Solve.

Чтобы вызов функции MPI_Comm_spawn, создающей новый процесс, выполнялся только для исходных процессов приложения, необходимо в начале функции Solve вызвать функцию MPI_Comm_get_parent и проанализировать возвращенный ею результат. 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 проверки того, что новый процесс был успешно создан, выведем в разделе 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. 37.

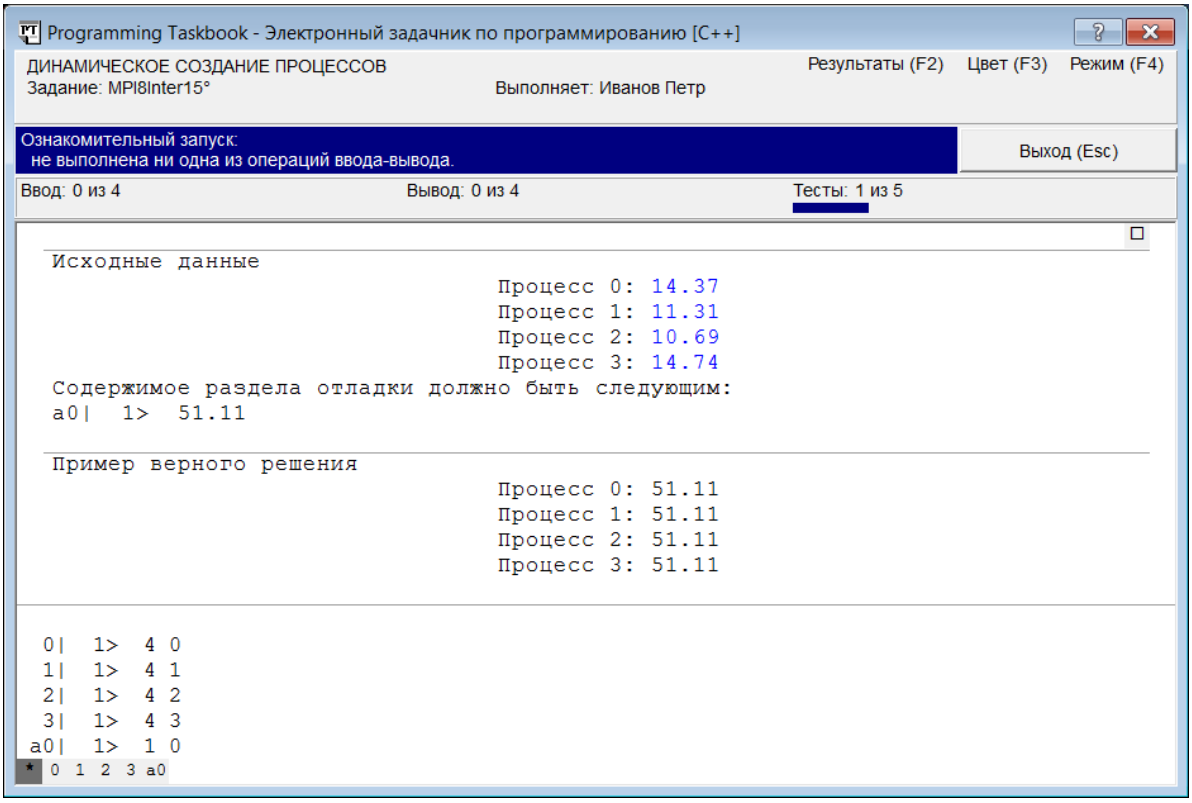

Рис. 37. Вид окна задачника после первого этапа выполнения задания MPI8Inter15

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 способом 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 строке с меткой «a0» 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, заданных в исходных процессах. По условию задачи для этого надо

использовать коллективную функцию MPI_Reduce. Очевидно, ее необходимо применить к интеркоммуникатору inter, который определен и в исходных, и в новом процессе.

При определении параметра root для функции MPI_Reduce необходимо учитывать особенности применения коллективных операций для интеркоммуникаторов (см. примечание 3 в п. 1.3.7): во всех процессах, пересылающих свои данные в другую группу интеркоммуникатора, мы должны указать «настоящий» ранг принимающего процесса root в удаленной группе (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 требуется не только выполнить ввод данных (в исходных процессах), но и определить значение параметра root во всех процессах. После вызова этой функции надо вывести полученный результат в разделе отладки, соответствующем новому процессу.

Приведем новый вариант решения, выделив в нем добавленные 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.

В результате запуска этого варианта программы окно примет вид, приведенный на рис. 38.

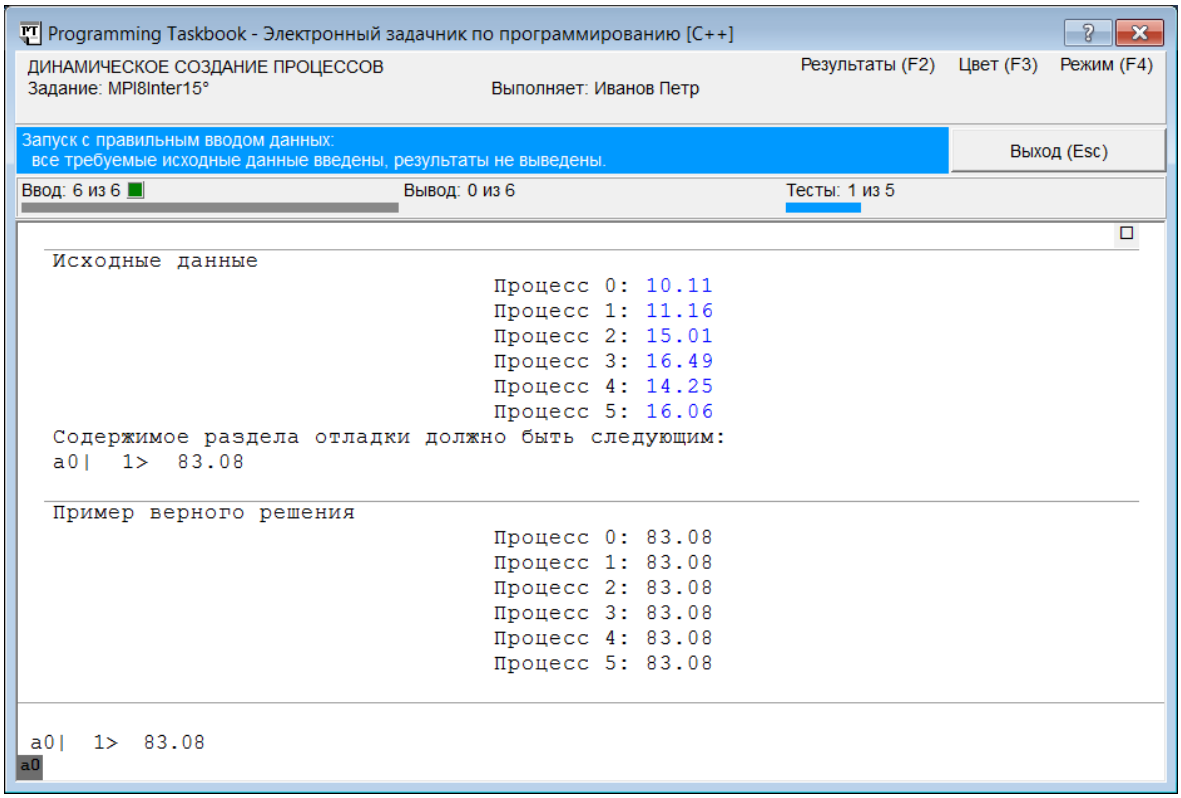

Рис. 38. Вид окна задачника после второго этапа выполнения задания MPI8Inter15

Обратите 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.

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. 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, а все процессы в принимающей группе должны указать, что они ожидают данные от процесса 0 посылающей группы.

Таким образом, после вызова MPI_Reduce нам достаточно добавить вызов новой коллективной функции MPI_Bcast и дополнить последний условный оператор веткой else (выполняемой в исходных процессах), в которой организовать 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 этого варианта программы мы получим сообщение о том, что задание выполнено.

В заключение данного пункта опишем еще две возможности, связанные с интеркоммуникаторами и динамическим созданием процессов. Этим возможностям посвящены четыре завершающих задачи группы MPI8Inter.

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(MPI_Comm comm, int high, MPI_Comm* newcomm) с тремя параметрами: исходным интеркоммуникатором comm, параметром high, определяющим порядок следования процессов в созданном интракоммуникаторе, и выходным параметром — ссылкой на созданный интракоммуникатор newcomm. 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 является одинаковым 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 задачи MPI8Inter19– MPI8Inter20.

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 процессов, между которыми требуется установить связь, играет роль *сервера*, а другая — роль *клиента*. Процессы группы-сервера создают *порт* для связи (функцией MPI_Open_port(MPI_Info info, char* port_name), параметр port_name является выходным) и определяют *публичное имя* этого порта (с помощью функции MPI_Publish_name(char* service_name, MPI_Info info, char* port_name), все параметры которой являются входными), после чего начинают прослушивать этот порт в ожидании подключения клиента функцией MPI_Comm_accept(char* port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm* newcomm). Данная функция является коллективной и должна вызываться для всех процессов коммуникатора comm, однако имя порта port_name достаточно указывать только в процессе ранга root.

Процессы группы-клиента получают порт, созданный группой-сервером, используя публичное имя этого порта service_name (с помощью функции MPI_Lookup_name(char* service_name, MPI_Info info, char* port_name); последний параметр является выходным), после чего подключаются к серверу по этому порту функцией MPI_Comm_connect(char* port_name, MPI_Info info, int root, MPI_Comm comm, MPI_Comm* newcomm). Данная функция, как и функция MPI_Comm_accept, является коллективной и должна вызываться для всех процессов коммуникатора comm, причем, как и для функции MPI_Comm_accept, имя порта port_name достаточно указывать только в процессе ранга root.

При успешном завершении функций MPI_Comm_accept и MPI_Comm_connect они возвращают интеркоммуникатор newcomm, объединяющий группу-клиент и группу-сервер.

Порт (параметр port_name) представляет собой текстовую строку, генерируемую средой MPI при вызове функции MPI_Open_port, причем максимальный размер строки (с завершающим нулевым символом) не превосходит константы MPI_MAX_PORT_NAME. Порт достаточно создать в одном из процессов группы-сервера и получить в одном из процессов группы-клиента. Публичное имя порта (параметр service_name), в отличие от пара-

метра port_name, должно быть известно заранее как процессам группы-сервера, так и процессам группы-клиента; это некоторый «пароль», которым обмениваются эти группы.

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.

Для освобождения ресурсов, выделяемых при клиент-серверном соединении, предназначены три функции:

- MPI_Close_port(char* port_name) освобождает порт port_name, выделенный функцией MPI_Open_port;
- MPI_Unpublish_name(char* service_name, MPI_Info info, char* port_name) освобождает публичное имя порта service_name, ранее связанное с портом port_name (публичное имя должно освобождаться до освобождения самого порта port_name функцией MPI_Close_port);
- MPI_Comm_disconnect(MPI_Comm* comm) разрушает коммуникатор comm, созданный для соединения клиента и сервера и возвращает в переменной comm значение MPI_COMM_NULL. В отличие от стандартной функции разрушения коммуникатора MPI_Comm_free, функция MPI_Comm_disconnect ожидает завершения всех операций обмена, выполняемых с помощью коммуникатора comm, и только после этого разрушает данный коммуникатор.

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.

1.4. Параллельные матричные алгоритмы

1.4.1. Ленточные и блочные алгоритмы параллельного умножения матриц: общее описание

Завершающая группа учебных заданий MPI9Matr, в отличие от предыдущих групп, посвящена не какому-либо определенному разделу библиотеки MPI, а *параллельным матричным алгоритмам*, в которых применяются разнообразные средства этой библиотеки, в том числе различные варианты взаимодействия процессов, новые производные типы, коммуникаторы с виртуальными топологиями и 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.

Обработка матриц является одним из тех видов вычислительных задач, для которых разработаны эффективные методы распараллеливания, в том числе основанные на 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* (bands), включающие наборы смежных строк или столбцов, и *блочные алгоритмы*, в которых применяется разбиение матриц на прямоугольные *блоки* (blocks). Кроме того, в группу включено вводное задание MPI9Matr1 (см. п. 2.9.1), 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, п. 2.9.2) используются только горизонтальные полосы (наборы смежных строк), для пересылки которых не требуется вводить дополнительные типы данных.

В *ленточном алгоритме 2* (задания MPI9Matr11–MPI9Matr20, п. 2.9.3) 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 *столб-*

цы другой), а с другой стороны, требует применения новых типов данных, обеспечивающих более эффективную пересылку вертикальных полос (т. е. наборов смежных столбцов).

В первом варианте блочного алгоритма (*алгоритм Кэннона*, задания MPI9Matr21–MPI9Matr31, п. 2.9.4) перед итерационным вычислением фрагментов итогового матричного произведения выполняется этап начального перераспределения блоков между процессами, благодаря которому упрощаются последующие действия по пересылке данных. Для пересылки блоков на любом из этапов алгоритма используется вспомогательный коммуникатор, снабженный топологией квадратной матрицы процессов.

Во втором варианте блочного алгоритма (*алгоритм Фокса*, задания MPI9Matr32– MPI9Matr44, п. 2.9.5) особый этап начального перераспределения отсутствует и, как следствие, 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.

В каждом из вариантов блочного алгоритма требуется определить новый производный тип данных, 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 (полос или блоков) 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, а также дает возможность реализовывать отдельные этапы алгоритма без предварительной разработки всех предыдущих этапов.

Серии заданий связываются с этапом вычислений, который является наиболее сложным. 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 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), необходимо 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, в которых требуется реализовать соответствующий матричный алгоритм в полном объеме.

В приводимых ниже таблицах 2 и 3 указываются номера заданий, связанных с реализацией различных этапов каждого алгоритма, и, кроме того, приводятся имена функций, которые требуется разработать в этих заданиях. Таблица 2 содержит данные для ленточных алгоритмов (MPI9Matr2–MPI9Matr20), а таблица 3 — для блочных (MPI9Matr21–MPI9Matr44).

Таблица 2

Задания группы MPI9Matr, связанные с ленточными алгоритмами

Этап алгоритма	Ленточный алгоритм 1	Ленточный алгоритм 2
Определение нового производного типа данных	*Отсутствует*	MPI9Matr11 Matr2CreateTypeBand (p, k, q, t)
Этап рассылки исходных данных	MPI9Matr2 Matr1ScatterData()	MPI9Matr12 Matr2ScatterData()

Таблица 2 (продолжение)

Этап алгоритма	Ленточный алгоритм 1	Ленточный алгоритм 2
Этап вычислений	MPI9Matr3 Matr1Calc() MPI9Matr4–MPI9Matr5 Matr1Calc(I)	MPI9Matr13 Matr2Calc() MPI9Matr14–MPI9Matr15 Matr2Calc(I)
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 3

Задания группы MPI9Matr, связанные с блочными алгоритмами

Этап алгоритма	Блочный алгоритм Кэннона	Блочный алгоритм Фокса
Определение нового производного типа данных	MPI9Matr21 Matr3CreateTypeBlock (m0, p0, p, t)	MPI9Matr32 Matr4CreateTypeBlock (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(I), Matr4Calc2()

Таблица 3 (продолжение)

Этап алгоритма	Блочный алгоритм Кэннона	Блочный алгоритм Фокса
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. 2.9).

1.4.2. Реализация непараллельного алгоритма умножения матриц

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, объявленную в этом заголовке. Во-вторых, файл содержит описания набора глобальных переменных, связанных с выполняемым заданием (назначение переменных указывается в комментариях, расположенных рядом с описаниями). В частности, вводятся более краткие имена для количества процессов и ранга текущего процесса: k и r. 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 объясняется тем, что более краткие имена

переменных a, b, c 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. Кроме задания MPI9Matr1 переменные a_, b_, c_ 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. 39 (для уменьшения размеров окна в этом и последующих рисунках скрыт раздел с формулировкой задания).

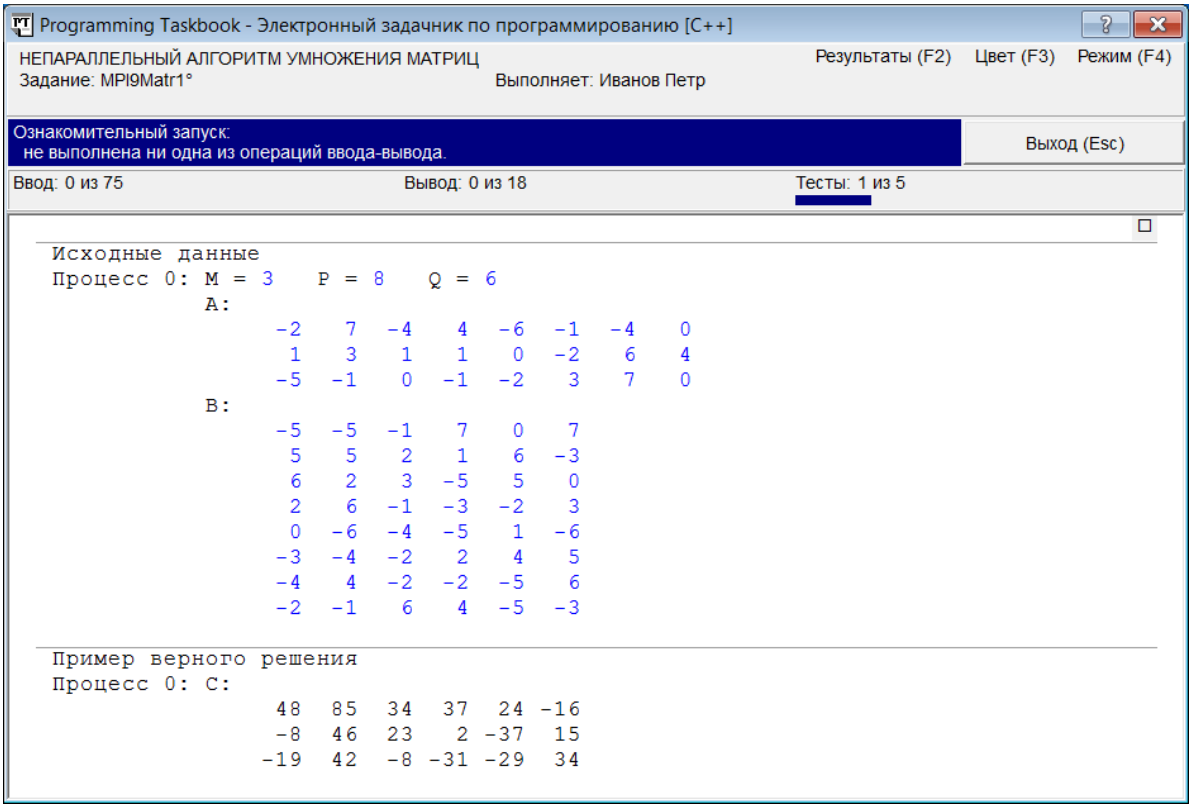

Рис. 39. Ознакомительный запуск задания MPI9Matr1

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 в одномерных динамических массивах (по строкам). Двумерные массивы в данном случае использовать не следует, так как это 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 этого 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 завершающей части решения освободим память, выделенную для массивов a_, b_ и c_:

```
delete[] a_;
delete[] b_;
delete[] c_;
```

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.

Отметим, что вместо динамических массивов при решении задач на реализацию матричных алгоритмов можно использовать контейнеры vector<T> (см. п. 1.2.6 и 1.2.7). Это позволит оформить действия по вводу-выводу матриц более кратким и наглядным образом.

Приведем соответствующий вариант решения задачи MPI9Matr1, выделив полужирным шрифтом измененные или добавленные фрагменты:

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

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

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

std::vector<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;
  if (r != 0)
    return;
  pt >> m >> p >> q;
  a_.assign(ptin_iterator<int>(m*p), ptin_iterator<int>());
  b_.assign(ptin_iterator<int>(p*q), ptin_iterator<int>());
  c_.assign(m*q, 0);
  for (int i = 0; i < m; ++i)
    for (int j = 0; j < q; ++j)
      for (int n = 0; n < p; ++n)
        c_[i*q+j] += a_[i*p+n] * b_[n*q+j];
  copy(c_.begin(), c_.end(), ptout_iterator<int>());
}
```

1.4.3. Рассылка исходных данных: пример реализации

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 $N_A = \mathrm{ceil}(M/K)$, $N_B = \mathrm{ceil}(P/K)$, 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 осуществлять 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. 40 и 41). На первом рисунке приведен раздел с исходными данными, на втором — раздел с примером правильных результатов.

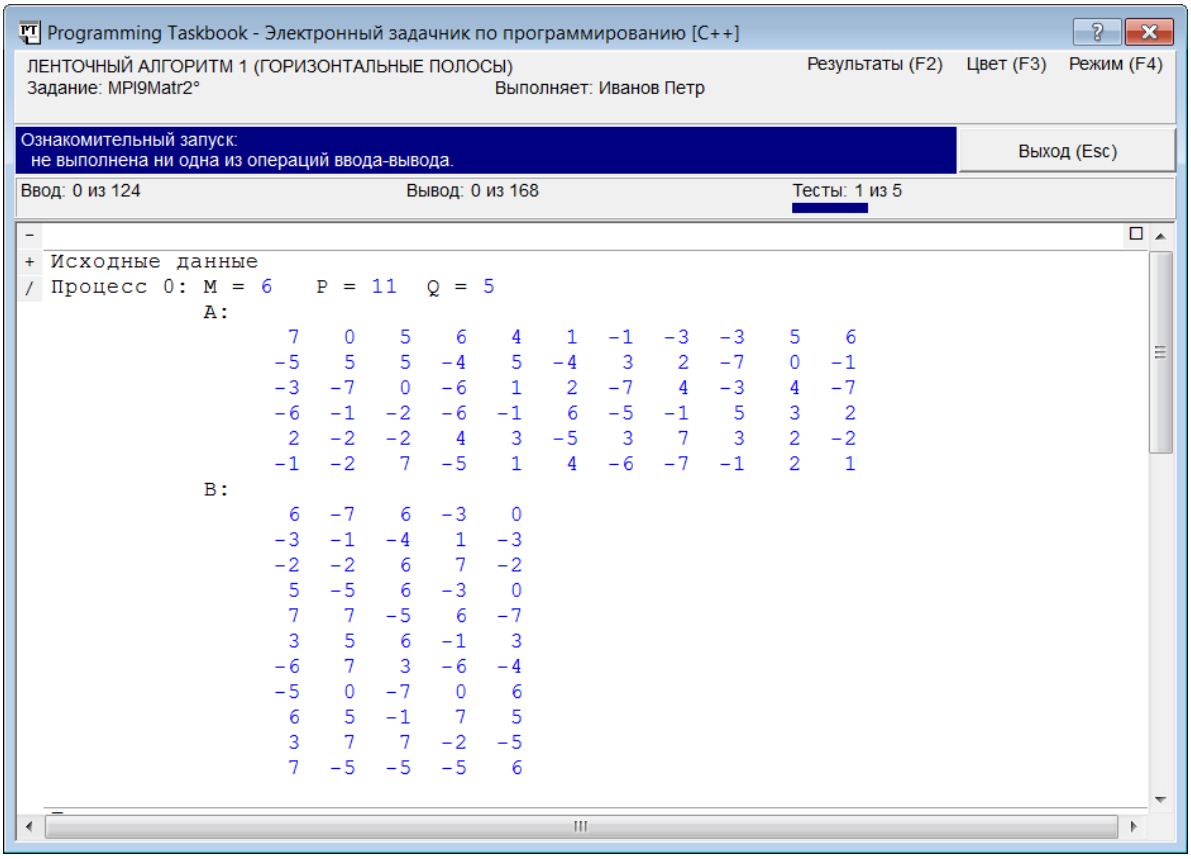

Рис. 40. Ознакомительный запуск задания MPI9Matr2 (раздел с исходными данными)

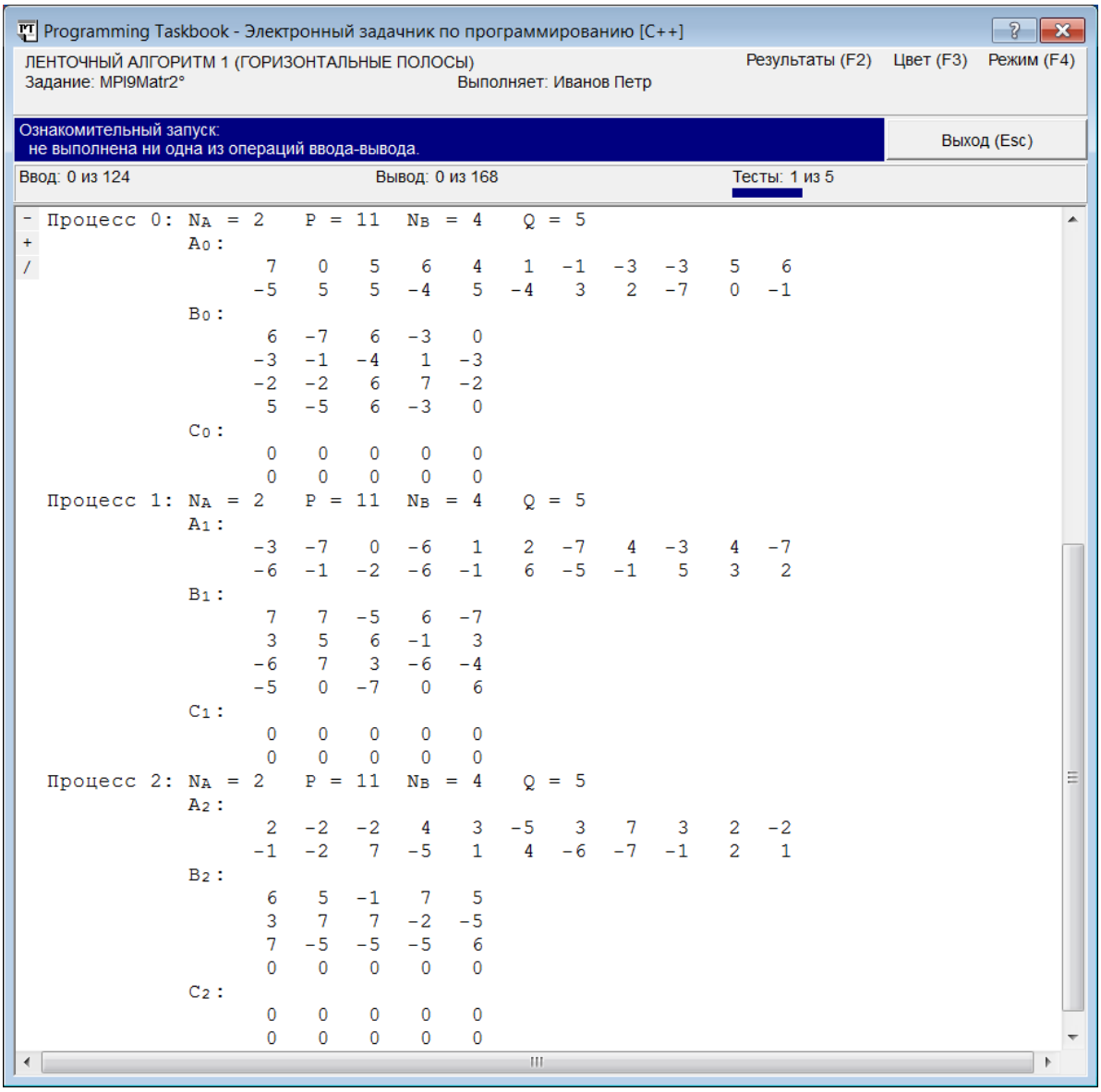

Рис. 41. Ознакомительный запуск задания MPI9Matr2 (пример верного решения)

Как следует из рисунков, в заданиях группы MPI9Matr может использоваться большое число исходных и результирующих данных (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 этого 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 нового варианта не будет отличаться от предыдущего.

Нам осталось вывести 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 увидим сообщение о том, что задание выполнено.

1.4.4. Перераспределение блоков на начальном этапе алгоритма Кэннона

При реализации очередных этапов алгоритмов матричного умножения нам не потребуется выполнять ввод исходных данных в главном процессе и пересылать их в другие процессы приложения, поскольку в соответствующих 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 реализа-

цию 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, подобное приведенному на рис. 42 и 43 (первый рисунок содержит начальную часть раздела с исходными данными, а второй — раздел с примером правильных результатов). Данные распределены по 9 процессам, порядок декартовой решетки K_0 равен 3.

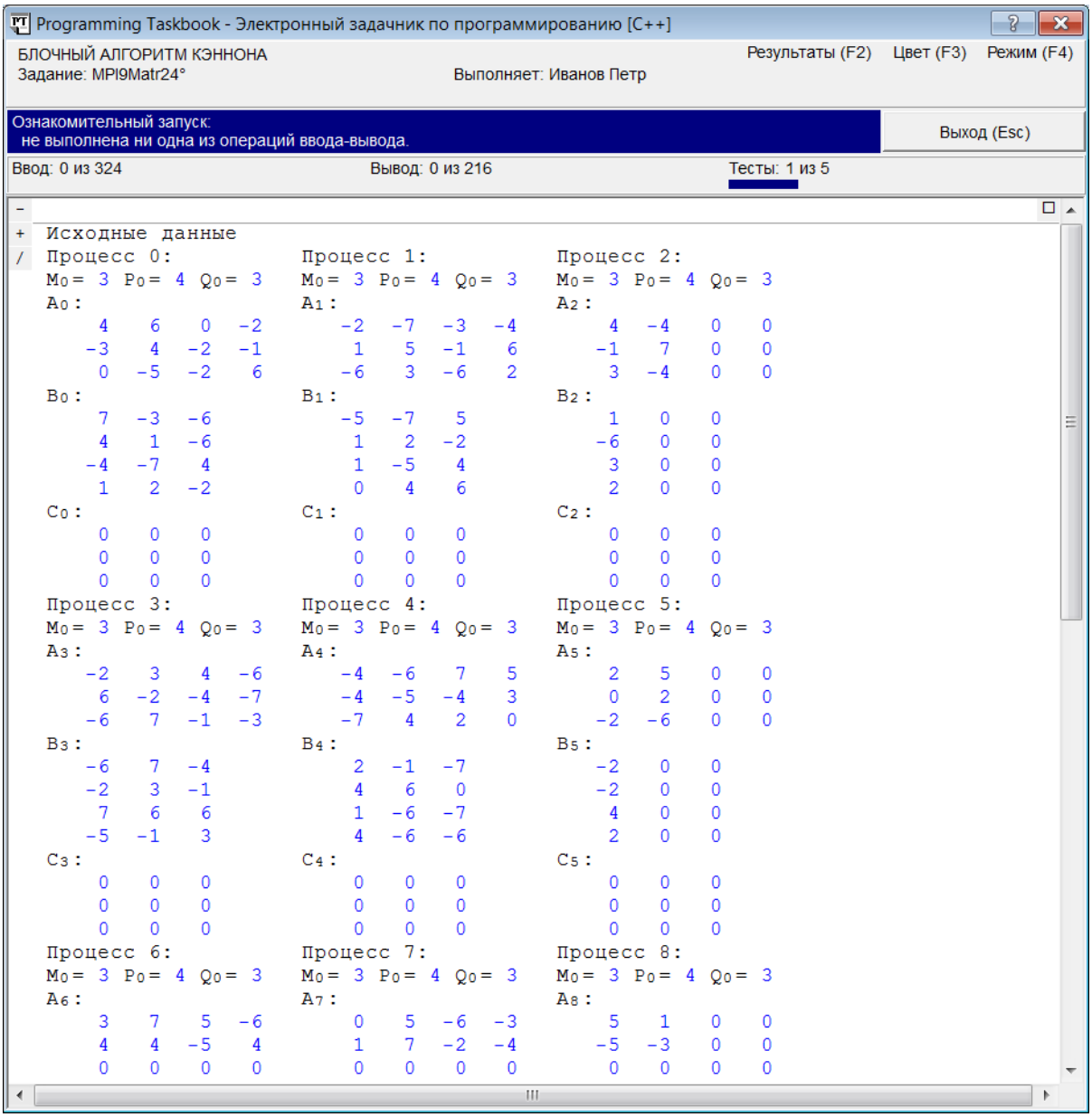

Рис. 42. Ознакомительный запуск задания MPI9Matr24 (раздел с исходными данными)

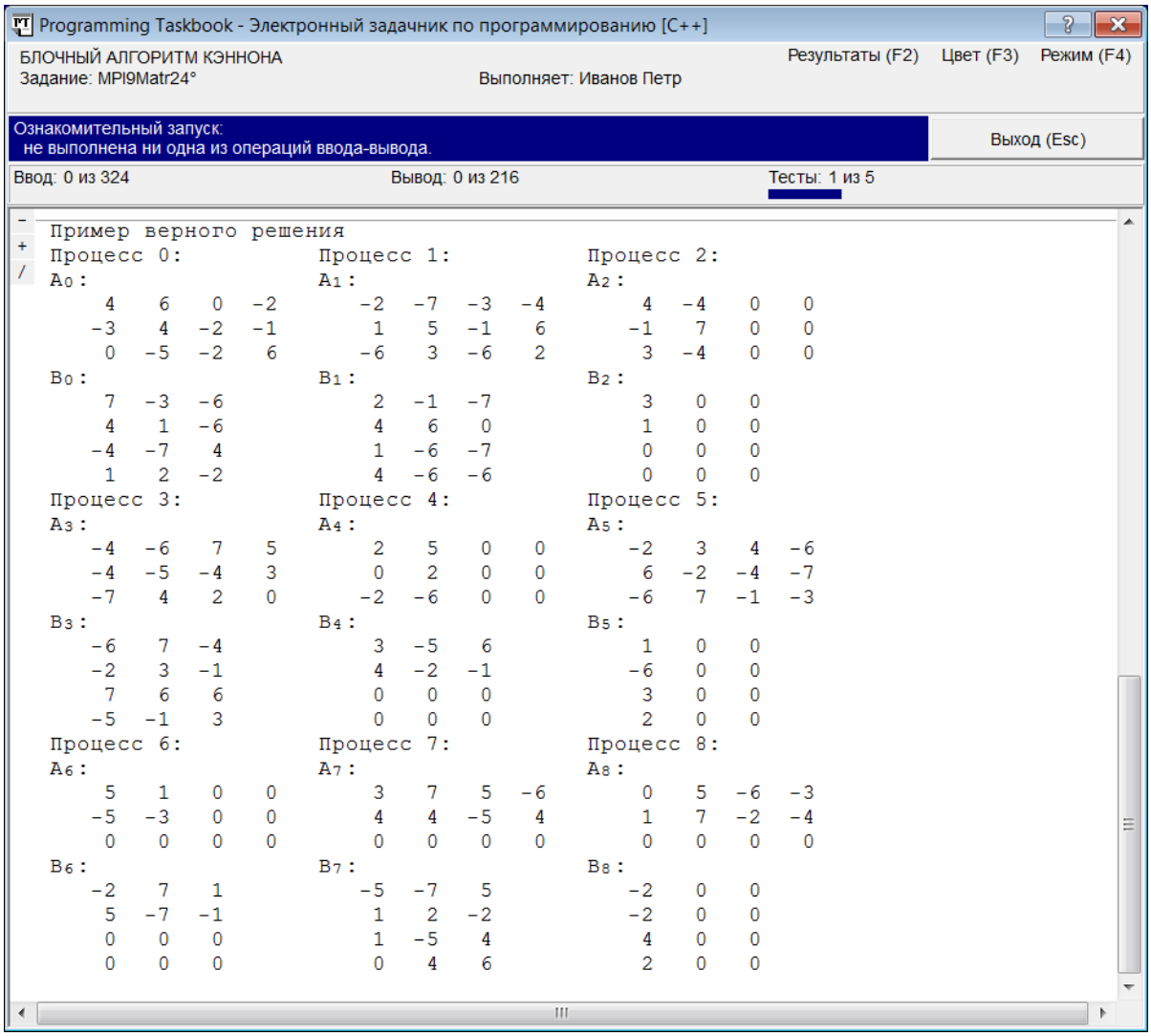

Рис. 43. Ознакомительный запуск задания MPI9Matr24 (пример верного решения)

Действия, связанные с вводом исходных данных, ничем не отличаются от рассмотренных ранее в заданиях MPI9Matr1 и MPI9Matr2, однако теперь они должны выполняться для каждого из процессов параллельного приложения (указанный ниже 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. Этот набор совпадает с результатами, полученными на этапе рассылки исходных данных по всем процессам приложения (ср. с приведенным в п. 1.4.3 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},
    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) с учетом ее цикличности по этой координате, а блоков матрицы *B* — на coord[1] позиций *вверх*, т. е. в направлении убывания *первой* координаты решетки (соответствующей номерам строк), также с учетом ее цикличности. Чтобы выполнить условия задачи, ранги посылающих и принимающих процессов следует определить с использованием функции MPI_Cart_shift, а саму пересылку осуществить с помощью функции

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.

Примечание 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.

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

1.4.5. Этап сборки результатов: пример реализации, основанной на файловом выводе

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 (см. п. 1.3.2–1.3.3). 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 $(N_A \cdot K) \times N_B$), полу-

ченными в результате выполнения K шагов ленточного алгоритма перемножения матриц (см. MPI9Matr15). Кроме того, в главном процессе дано число M — количество строк результирующего матричного произведения 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 в п. 1.4.3 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. 44 приведено окно задачника, которое появится при запуске созданной программы-заготовки.

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 окна задачнка не указывается количество элементов данных, которые надо вывести). Задачник отобразит содержимое созданного файла в своем окне и проверит его правильность. Отметим, 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. Следует также 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 произведение $N_A \cdot K$, где K — количество процессов, в данном случае превышает количество строк M матрицы C).

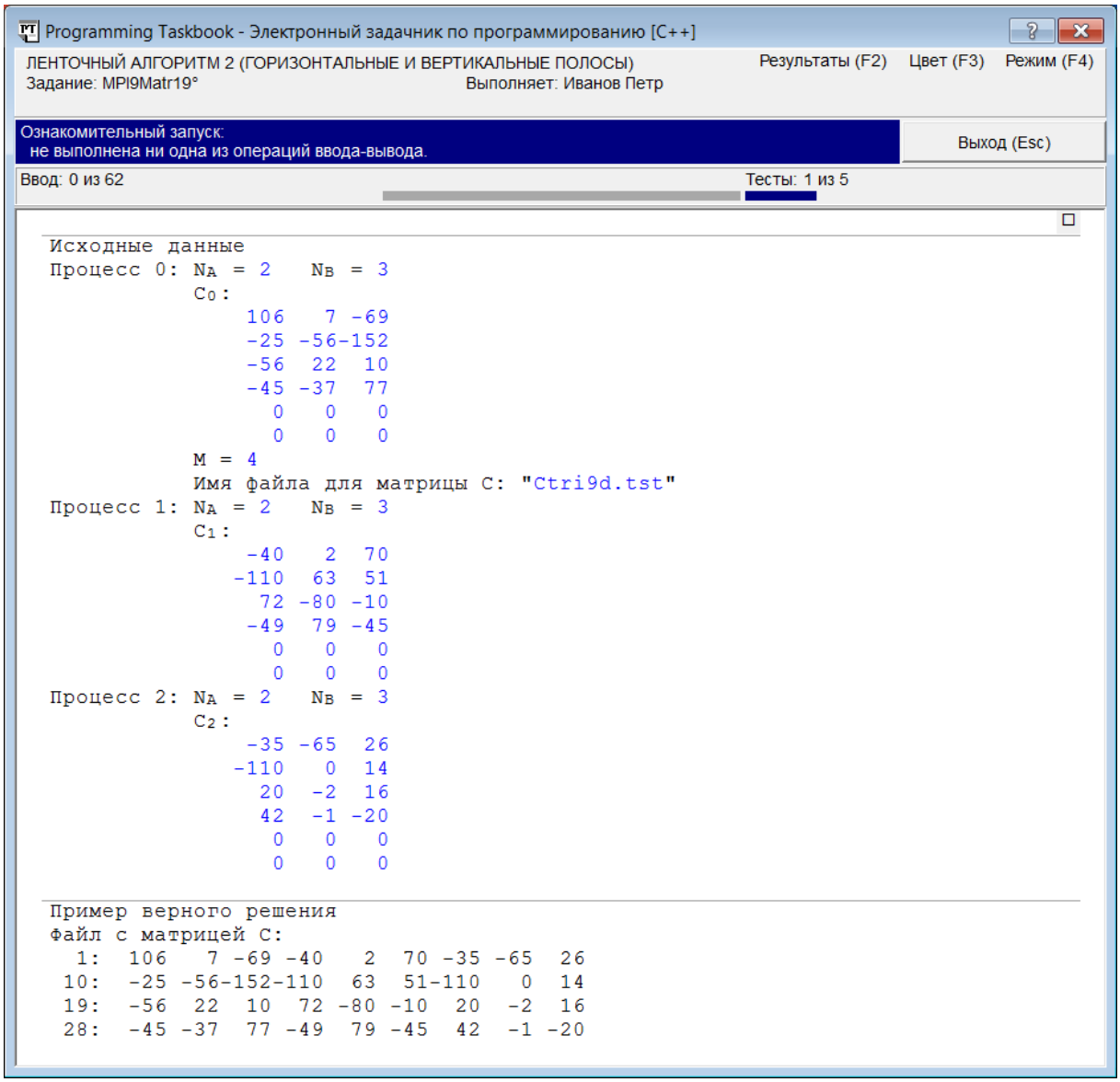

Рис. 44. Ознакомительный запуск задания MPI9Matr19

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 вы-

числены) 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 фрагментом, в котором выполняется ввод всех остальных данных и вызывается функция Matr2GatherFile (вспомогательная функция и добавленные в функцию Solve 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 11 символов (см. преамбулу к группе заданий MPI9Matr — п. 2.9).

Запустив этот 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 f 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 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 пустыми промежутками (каждый фрагмент соответствует элементам отдельной строки, а промежутки 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 полосе 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, такой тип данных 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 (см. п. 2.9.3). Поскольку мы не выполняли задание MPI9Matr11, функцию Matr2CreateTypeBand придется реализовать непосредственно при выполнении нашего задания, используя рекомендации, приведенные в задании MPI9Matr11 (напомним, что с аналогичной ситуацией мы сталкивались ранее в п. 1.4.4 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):

count – число блоков, содержащих последовательные элементы;

blocklen – размер каждого блока (в элементах);

stride – расстояние между начальными позициями соседних блоков (тоже измеряемое в элементах).

Следующий параметр oldtype определяет базовый тип элементов (в нашем случае это MPI_INT), а последний параметр newtype является выходным — это указатель на 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 определяют 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 такую реализацию функции Matr2CreateTypeBand, поскольку она 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. 1.3.3. Для определения файлового вида данных надо указать начальное смещение disp (в байтах), элементарный тип etype (в нашем случае MPI_INT) и собственно файловый тип filetype, 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 — 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 вертикальная полоса с элементами матрицы C (без завершающих нулевых строк) имеет в каждом процессе 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 этот вариант программы, мы получим сообщение о том, что задание выполнено.

1.5. Дополнительные приемы разработки параллельных программ

1.5.1. Отладка параллельных программ с применением средств электронного задачника

Рассмотренные выше примеры выполнения заданий по параллельному программированию показывают, что использование задачника позволяет существенно ускорить разработку параллельных программ. Это достигается, в первую очередь, благодаря специальному механизму, обеспечивающему быстрый запуск параллельной программы непосредственно из интегрированной среды. Кроме того, отладку параллельной программы упрощает использование функции Show, позволяющей вывести в окне задачника любые требуемые данные из различных процессов и впоследствии просмотреть полученную отладочную информацию как для отдельного процесса, так и для всех процессов параллельной программы.

Отмеченные особенности задачника могут оказаться полезными не только при решении учебных задач по параллельному MPI-программированию, но и при разработке параллельных программ общего назначения. Поэтому, наряду с «обычными» группами заданий, позволяющими изучить возможности библиотеки MPI, в задачник Programming Taskbook for MPI-2 включена особая группа MPIDebug, предназначенная не для вы-

полнения определенных заданий, а для отладки *любых* параллельных программ. Данная группа состоит из 36 «заданий», причем задание с номером N обеспечивает запуск параллельной программы с N процессами. Таким образом, если создать программу-заготовку для одного из заданий группы MPIDebug, то при запуске этой программы из интегрированной среды она автоматически будет выполняться в параллельном режиме с требуемым числом процессов.

Воспользуемся группой MPIDebug для разработки программы, реализующей один из параллельных алгоритмов умножения матрицы A на вектор b — так называемый *самопланирующий алгоритм* (см., например, [10, гл. 7]), в котором главный процесс координирует работу подчиненных процессов, пересылая им требуемые наборы данных и получая от них результаты.

Исходные данные вводятся в главном процессе. Затем главный процесс пересылает во все подчиненные процессы вектор b и по одной из строк матрицы A. После этого запускается цикл, в котором главный процесс принимает от подчиненных процессов результаты умножения строки матрицы на вектор и пересылает им оставшиеся строки матрицы A. Цикл завершится, когда главный процесс перешлет подчиненным процессам все строки матрицы A и получит от них все элементы результирующего вектора Ab.

Следует заметить, что при реализации данного алгоритма нельзя заранее определить, какие именно строки матрицы будет обрабатывать тот или иной подчиненный процесс. Если, к примеру, программа использует 5 процессов, то строка матрицы номер 5 будет переслана тому процессу (из числа подчиненных процессов с рангами от 1 до 4), который первым завершит умножение ранее посланной ему строки и вернет результат главному процессу.

Для тестирования алгоритма будем обрабатывать матрицу A порядка $N = 20$, каждая строка которой содержит одинаковые элементы, равные порядковому номеру строки: $a_{ij} = i$ для $i, j = 1, \ldots, N$. Вектор b того же размера N будет содержать одинаковые элементы, равные 0.01. Таким образом, элементы вектора $c = Ab$ будут иметь вид $c_i = 0.2i$, $i = 1, \ldots, N$:

$c = (0.2, 0.4, 0.6, \ldots, 3.8, 4.0)$

Для правильной работы алгоритма необходимо, чтобы количество подчиненных процессов не превосходило порядок матрицы N, так как на начальном этапе алгоритма каждому подчиненному процессу посылается одна из строк данной матрицы. Выберем для определенности число процессов нашей программы равным 10; для того чтобы обеспечить автоматический запуск параллельной программы с этим числом процессов, будем использовать задание MPIDebug10.

При запуске программы-заготовки для задания MPIDebug10 окно задачника будет иметь вид, приведенный на рис. 45. Вместо исходных и результирующих данных в этом окне выводится описание тех средств задачника, которые предназначены для вывода отладочной информации.

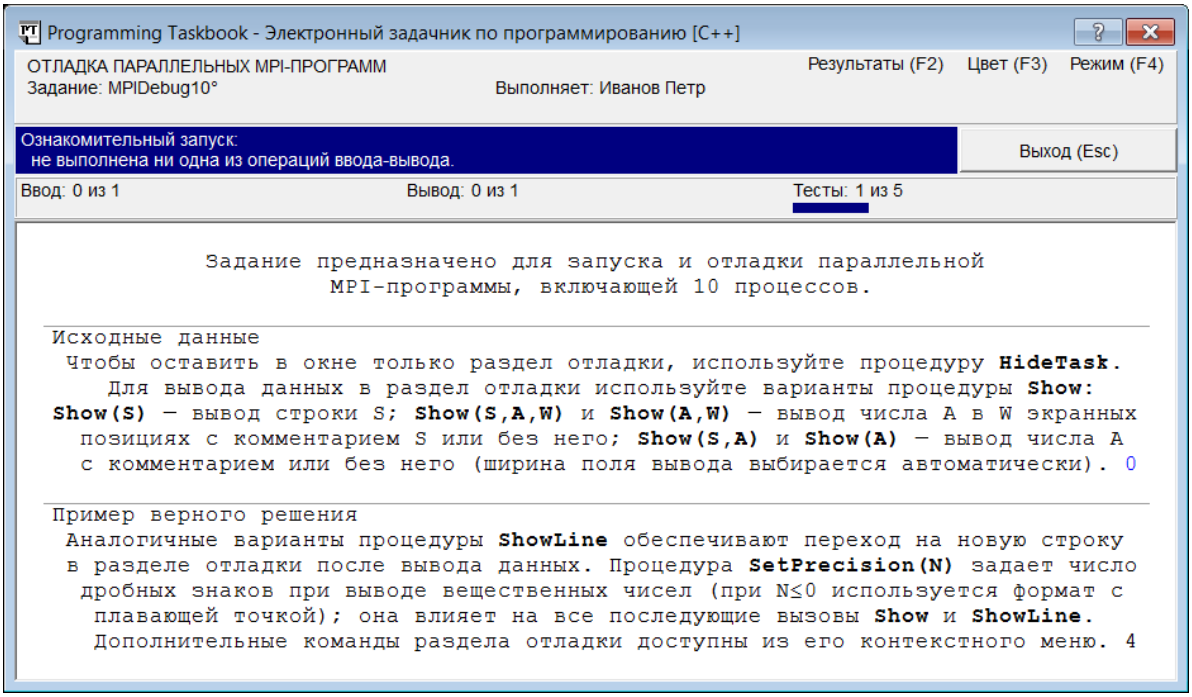

Рис. 45. Ознакомительный запуск задания MPIDebug10

Дополним созданную заготовку фрагментом, в котором описываются все необходимые переменные (в том числе и те, которые потребуются на следующих этапах реализации алгоритма), выполняется формирование в главном процессе исходной матрицы A и вектора b, а также осуществляется пересылка вектора b во все подчиненные процессы (добавленные операторы выделены полужирным шрифтом):

```
void Solve()
{
  Task("MPIDebug10");
  int flag;
  MPI_Initialized(&flag);
  if (flag == 0)
    return;
  int rank, size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  const int n = 20;
  double a[n][n], b[n], c[n], d;
  int cur_row;
```

```
  MPI_Status s;
  // генерация исходных данных в главном процессе:
  if (rank == 0)
    for (int i = 0; i < n; ++i)
    {
      b[i] = 0.01;
      for (int j = 0; j < n; ++j)
        a[i][j] = i + 1;
    }
  // пересылка всем процессам вектора b:
  MPI_Bcast(b, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  ShowLine("Vector b: ..., ", b[n-1]);
}
```

Для хранения матрицы используется двумерный статический массив (массив массивов), так как порядок матрицы $N = 20$ заранее известен.

При запуске нового варианта программы в окне задачника появится раздел отладки (рис. 46).

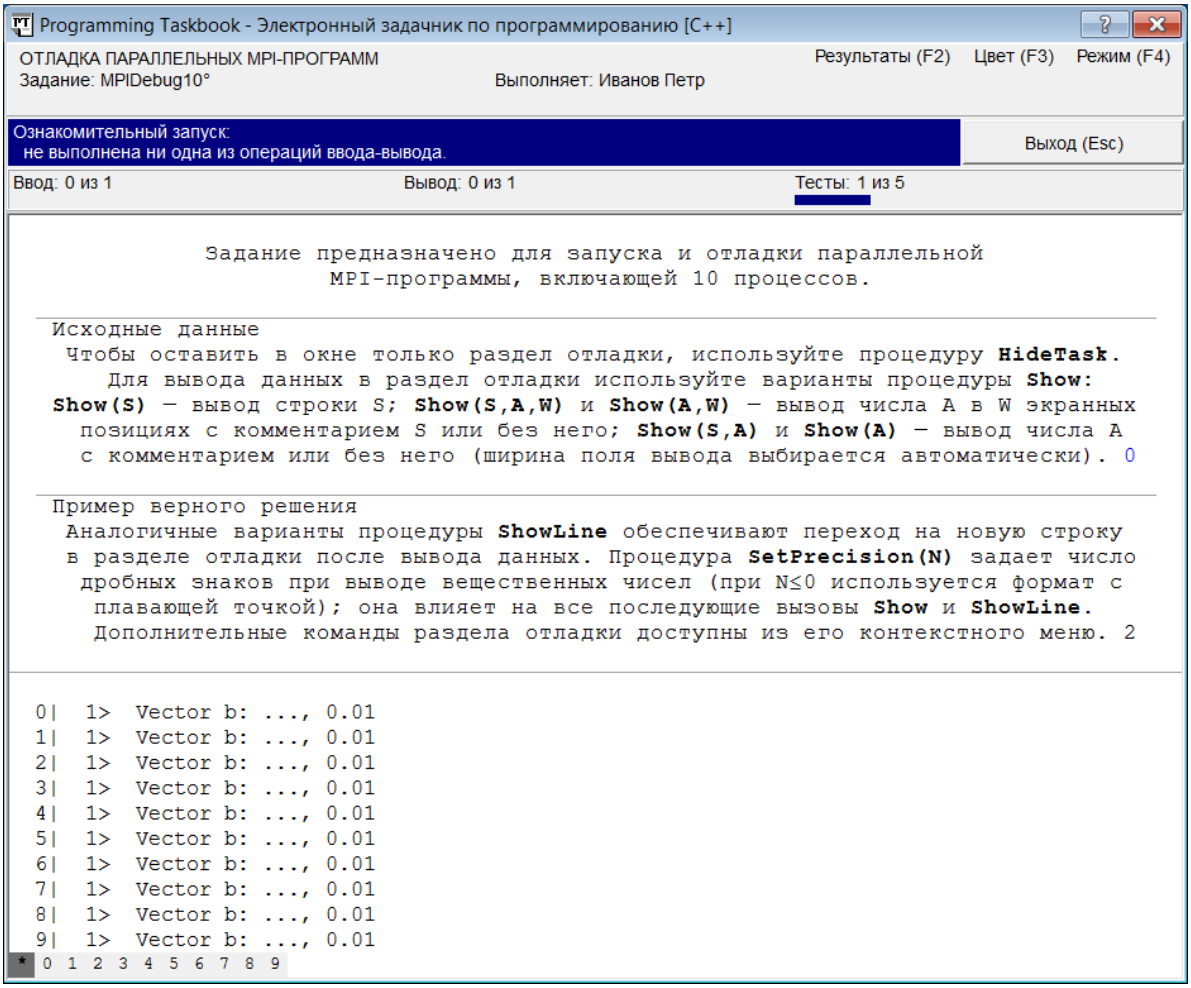

Рис. 46. Вывод отладочных данных

Поскольку в новом варианте программы используются вызовы функции ShowLine, в окне задачника отображается раздел отладки с информацией, которая была выведена с помощью этой функции. В нашем случае каждый из процессов (включая главный) вывел информацию о том, что он успешно получил вектор b. Для краткости выведено только значение *последнего* элемента массива b с индексом n – 1; чтобы подчеркнуть, что выведен последний элемент, перед ним указано многоточие.

> **Примечание**. Отладочная информация выводится на английском языке; это позволит избежать проблем с кодировкой при выводе аналогичной информации в консольном приложении (см. п. 1.5.3).

Так как в основных разделах окна задачника в нашем случае содержится лишь справочная информация, удобно скрыть все разделы окна, кроме раздела отладки. Если мы уже находимся в окне задачника, то для этого достаточно нажать комбинацию [Ctrl]+[пробел]. Однако еще удобнее автоматически скрывать ненужные разделы при выводе окна задачника на экран. Для этого достаточно вызвать в любом месте программы вспомогательную функцию HideTask. Если добавить вызов этой функции в конец нашей программы и повторно запустить ее на выполнение, то окно задачника будет содержать только раздел отладки (рис. 47).

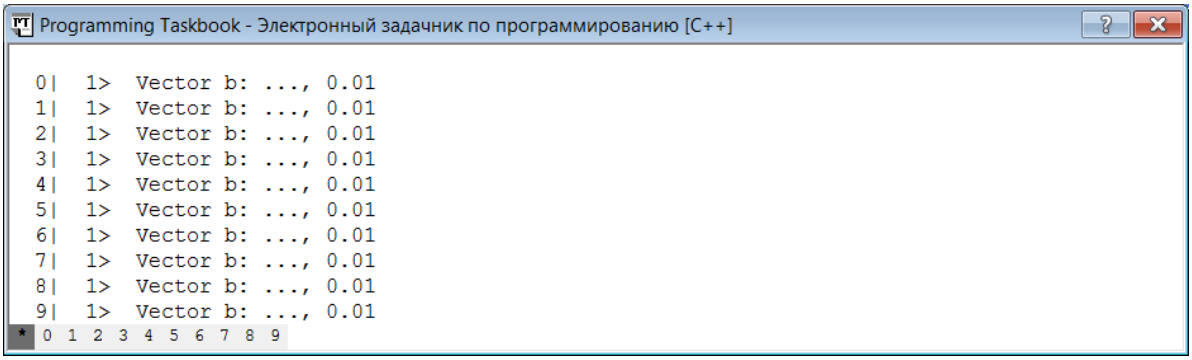

Рис. 47. Скрытие основных разделов окна задачника при выводе отладочных данных

Вернемся к реализации алгоритма умножения матрицы на вектор и приведем его завершающую часть:

```
if (rank == 0)
{
  // пересылка подчиненным процессам начальных строк матрицы A:
  for (int i = 1; i < size; ++i)
  {
    MPI_Send(a[i-1], n, MPI_DOUBLE, i, i - 1, MPI_COMM_WORLD);
    Show("Sending initial data: dest=", i);
    Show("tag=", i - 1);
    ShowLine("-> ..., ", a[i - 1][0]);
  }
```

```
  cur_row = size - 2;
  // получение элементов произведения и пересылка остальных строк:
  for (int i = 0; i < n; ++i)
  {
    MPI_Recv(&d, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
      MPI_COMM_WORLD, &s);
    Show("Receiving result: source=", s.MPI_SOURCE);
    Show("tag=", s.MPI_TAG, 2);
    Show("<- ", d);
    c[s.MPI_TAG] = d;
    ++cur_row;
    if (cur_row < n)
      MPI_Send(a[cur_row], n, MPI_DOUBLE, s.MPI_SOURCE,
        cur_row, MPI_COMM_WORLD);
    else
      MPI_Send(&d, 1, MPI_DOUBLE, s.MPI_SOURCE, cur_row,
        MPI_COMM_WORLD);
    Show("| Sending data: dest=", s.MPI_SOURCE);
    ShowLine("tag=", cur_row, 2);
  }
  // вывод полученного вектора Ab:
  SetPrecision(1);
  ShowLine("Resulting vector Ab:");
  for (int i = 0; i < n; ++i)
    Show(c[i], 3);
}
else
  // обработка данных в подчиненных процессах:
  while (true)
  {
    MPI_Recv(c, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
      &s);
    Show("Receiving data: tag=", s.MPI_TAG, 2);
    if (s.MPI_TAG >= n)
      break;
    d = 0;
    for (int i = 0; i < n; ++i)
      d += c[i] * b[i];
    MPI_Send(&d, 1, MPI_DOUBLE, 0, s.MPI_TAG, MPI_COMM_WORLD);
    Show("| Sending result: tag=", s.MPI_TAG, 2);
    ShowLine("-> ", d);
```

```
}
```

Данная часть состоит из двух больших фрагментов, первый из которых должен выполняться в главном процессе, а второй — в каждом из подчиненных процессов.

Главный процесс вначале пересылает каждому подчиненному процессу по одной из строк исходной матрицы и инициализирует переменную cur_row, в которой содержится индекс последней обработанной, т. е. пересланной в подчиненный процесс, строки матрицы (строки индексируются от нуля). После этого запускается основной цикл, в котором главный процесс получает от подчиненных процессов результирующие элементы произведения и пересылает им очередные строки матрицы.

Поскольку порядок получения данных от подчиненных процессов не определен, в функции MPI_Recv используется параметр MPI_ANY_SOURCE, обеспечивающий получение данных от любого пославшего их процесса. В *метке сообщения* (message tag) подчиненный процесс передает главному процессу дополнительную информацию — индекс вычисленного элемента произведения. Для того чтобы главный процесс мог принимать сообщения с любыми метками, параметр msgtag функции MPI_Recv полагается равным MPI_ANY_TAG. Информацию о ранге процесса, пославшего сообщение, и о метке сообщения главный процесс получает с помощью записи s типа MPI_Status, обращаясь к ее полям MPI_SOURCE и MPI_TAG соответственно.

После получения сообщения от подчиненного процесса главный процесс отправляет ему новое сообщение, которое либо содержит очередную необработанную строку матрицы (если такие строки еще остались), либо специальный признак завершения. Если посылается очередная строка, то метка посылаемого сообщения полагается равной индексу строки cur_row; если же посылается признак завершения, то метка содержит число, которое *больше максимального индекса строки матрицы*.

После завершения цикла for в главном процессе выводятся все элементы найденного произведения.

В подчиненном процессе запускается цикл, в котором вначале выполняется прием очередной строки матрицы из главного процесса (ее индекс передается в метке сообщения), затем полученная строка умножается на ранее полученный вектор, и наконец, результат умножения пересылается главному процессу. Если же при приеме очередного сообщения от главного процесса получена метка, превышающая номер максимального индекса строки матрицы, то происходит немедленный выход из цикла, и подчиненный процесс завершает свою работу.

Все этапы описанного выше алгоритма сопровождаются отладочной печатью. Для этого используется функция Show и ее вариант ShowLine (обеспечивающий дополнительный переход на новую строку в разделе отладки после вывода данных). Перед выводом очередного числового элемента, как

правило, выводится предваряющий его строковый комментарий, а в некоторых случаях после числового элемента дополнительно указывается количество экранных позиций, которые следует отвести для его вывода (это позволяет обеспечить *выравнивание* выводимых данных по вертикали).

Перед выводом в главном процессе элементов результирующего произведения вызывается функция SetPrecision(1), обеспечивающая вывод всех последующих данных вещественного типа с одним знаком после десятичного разделителя. Заметим, что по умолчанию число дробных знаков полагается равным 2, однако при использовании этого значения все результирующие элементы не удалось бы вывести в одной строке, так как ширина области выводимых данных в разделе отладки фиксирована и равна 80 позициям.

При оформлении вывода было учтено, что данные, выводимые функцией Show или ShowLine, отделяются одним пробелом от ранее выведенных данных, находящихся на той же экранной строке.

Приведем отладочные данные, которые были получены при запуске программы (напомним, что перед первым символом «|» указывается ранг процесса, который вывел соответствующую строку отладочной информации, а перед символом «>» — порядковый номер выведенной строки; при этом строки, связанные с различными процессами, нумеруются независимо).

```
0|  1>  Vector b: ..., 0.01
0|  2>  Sending initial data: dest=1 tag=0 -> ..., 1.00
0|  3>  Sending initial data: dest=2 tag=1 -> ..., 2.00
0|  4>  Sending initial data: dest=3 tag=2 -> ..., 3.00
0|  5>  Sending initial data: dest=4 tag=3 -> ..., 4.00
0|  6>  Sending initial data: dest=5 tag=4 -> ..., 5.00
0|  7>  Sending initial data: dest=6 tag=5 -> ..., 6.00
0|  8>  Sending initial data: dest=7 tag=6 -> ..., 7.00
0|  9>  Sending initial data: dest=8 tag=7 -> ..., 8.00
0| 10>  Sending initial data: dest=9 tag=8 -> ..., 9.00
0| 11>  Receiving result: source=1 tag= 0 <- 0.20 | Sending data: dest=1 tag= 9
0| 12>  Receiving result: source=2 tag= 1 <- 0.40 | Sending data: dest=2 tag=10
0| 13>  Receiving result: source=3 tag= 2 <- 0.60 | Sending data: dest=3 tag=11
0| 14>  Receiving result: source=1 tag= 9 <- 2.00 | Sending data: dest=1 tag=12
0| 15>  Receiving result: source=2 tag=10 <- 2.20 | Sending data: dest=2 tag=13
0| 16>  Receiving result: source=3 tag=11 <- 2.40 | Sending data: dest=3 tag=14
0| 17>  Receiving result: source=2 tag=13 <- 2.80 | Sending data: dest=2 tag=15
0| 18>  Receiving result: source=1 tag=12 <- 2.60 | Sending data: dest=1 tag=16
0| 19>  Receiving result: source=3 tag=14 <- 3.00 | Sending data: dest=3 tag=17
0| 20>  Receiving result: source=1 tag=16 <- 3.40 | Sending data: dest=1 tag=18
0| 21>  Receiving result: source=2 tag=15 <- 3.20 | Sending data: dest=2 tag=19
0| 22>  Receiving result: source=3 tag=17 <- 3.60 | Sending data: dest=3 tag=20
0| 23>  Receiving result: source=1 tag=18 <- 3.80 | Sending data: dest=1 tag=21
0| 24>  Receiving result: source=2 tag=19 <- 4.00 | Sending data: dest=2 tag=22
```

```
0| 25>  Receiving result: source=6 tag= 5 <- 1.20 | Sending data: dest=6 tag=23
0| 26>  Receiving result: source=7 tag= 6 <- 1.40 | Sending data: dest=7 tag=24
0| 27>  Receiving result: source=8 tag= 7 <- 1.60 | Sending data: dest=8 tag=25
0| 28>  Receiving result: source=9 tag= 8 <- 1.80 | Sending data: dest=9 tag=26
0| 29>  Receiving result: source=5 tag= 4 <- 1.00 | Sending data: dest=5 tag=27
0| 30>  Receiving result: source=4 tag= 3 <- 0.80 | Sending data: dest=4 tag=28
0| 31>  Resulting vector Ab:
0| 32>  0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0
1|  1>  Vector b: ..., 0.01
1|  2>  Receiving data: tag= 0 | Sending result: tag= 0 -> 0.20
1|  3>  Receiving data: tag= 9 | Sending result: tag= 9 -> 2.00
1|  4>  Receiving data: tag=12 | Sending result: tag=12 -> 2.60
1|  5>  Receiving data: tag=16 | Sending result: tag=16 -> 3.40
1|  6>  Receiving data: tag=18 | Sending result: tag=18 -> 3.80
1|  7>  Receiving data: tag=21
2|  1>  Vector b: ..., 0.01
2|  2>  Receiving data: tag= 1 | Sending result: tag= 1 -> 0.40
2|  3>  Receiving data: tag=10 | Sending result: tag=10 -> 2.20
2|  4>  Receiving data: tag=13 | Sending result: tag=13 -> 2.80
2|  5>  Receiving data: tag=15 | Sending result: tag=15 -> 3.20
2|  6>  Receiving data: tag=19 | Sending result: tag=19 -> 4.00
2|  7>  Receiving data: tag=22
3|  1>  Vector b: ..., 0.01
3|  2>  Receiving data: tag= 2 | Sending result: tag= 2 -> 0.60
3|  3>  Receiving data: tag=11 | Sending result: tag=11 -> 2.40
3|  4>  Receiving data: tag=14 | Sending result: tag=14 -> 3.00
3|  5>  Receiving data: tag=17 | Sending result: tag=17 -> 3.60
3|  6>  Receiving data: tag=20
4|  1>  Vector b: ..., 0.01
4|  2>  Receiving data: tag= 3 | Sending result: tag= 3 -> 0.80
4|  3>  Receiving data: tag=28
5|  1>  Vector b: ..., 0.01
5|  2>  Receiving data: tag= 4 | Sending result: tag= 4 -> 1.00
5|  3>  Receiving data: tag=27
6|  1>  Vector b: ..., 0.01
6|  2>  Receiving data: tag= 5 | Sending result: tag= 5 -> 1.20
6|  3>  Receiving data: tag=23
7|  1>  Vector b: ..., 0.01
7|  2>  Receiving data: tag= 6 | Sending result: tag= 6 -> 1.40
7|  3>  Receiving data: tag=24
8|  1>  Vector b: ..., 0.01
8|  2>  Receiving data: tag= 7 | Sending result: tag= 7 -> 1.60
8|  3>  Receiving data: tag=25
9|  1>  Vector b: ..., 0.01
9|  2>  Receiving data: tag= 8 | Sending result: tag= 8 -> 1.80
9|  3>  Receiving data: tag=26
```

Из приведенного текста видно, что в ходе выполнения алгоритма процессы рангов 1 и 2 обработали по 5 строк исходной матрицы, процесс

ранга 3 обработал 4 строки, а остальные 6 подчиненных процессов — по 1 строке. В последней, 32-й строке, связанной с главным процессом, содержатся элементы найденного произведения.

Заметим, что текст, который отображается в разделе отладки окна задачника, можно скопировать в буфер обмена Windows; для этого достаточно либо нажать стандартную комбинацию клавиш [Ctrl]+[C], либо вызвать контекстное меню окна (нажав правую кнопку мыши) и выполнить его команду **Скопировать данные из раздела отладки в буфер**.

Изменив имя задания в функции Task на «MPIDebug5», мы сможем протестировать разработанный алгоритм в параллельном режиме, использующем 5 процессов. Приведем отладочную информацию, которая была выведена при таком варианте запуска программы:

```
0|  1>  Vector b: ..., 0.01
0|  2>  Sending initial data: dest=1 tag=0 -> ..., 1.00
0|  3>  Sending initial data: dest=2 tag=1 -> ..., 2.00
0|  4>  Sending initial data: dest=3 tag=2 -> ..., 3.00
0|  5>  Sending initial data: dest=4 tag=3 -> ..., 4.00
0|  6>  Receiving result: source=4 tag= 3 <- 0.80 | Sending data: dest=4 tag= 4
0|  7>  Receiving result: source=2 tag= 1 <- 0.40 | Sending data: dest=2 tag= 5
0|  8>  Receiving result: source=3 tag= 2 <- 0.60 | Sending data: dest=3 tag= 6
0|  9>  Receiving result: source=4 tag= 4 <- 1.00 | Sending data: dest=4 tag= 7
0| 10>  Receiving result: source=1 tag= 0 <- 0.20 | Sending data: dest=1 tag= 8
0| 11>  Receiving result: source=2 tag= 5 <- 1.20 | Sending data: dest=2 tag= 9
0| 12>  Receiving result: source=3 tag= 6 <- 1.40 | Sending data: dest=3 tag=10
0| 13>  Receiving result: source=4 tag= 7 <- 1.60 | Sending data: dest=4 tag=11
0| 14>  Receiving result: source=1 tag= 8 <- 1.80 | Sending data: dest=1 tag=12
0| 15>  Receiving result: source=2 tag= 9 <- 2.00 | Sending data: dest=2 tag=13
0| 16>  Receiving result: source=3 tag=10 <- 2.20 | Sending data: dest=3 tag=14
0| 17>  Receiving result: source=4 tag=11 <- 2.40 | Sending data: dest=4 tag=15
0| 18>  Receiving result: source=2 tag=13 <- 2.80 | Sending data: dest=2 tag=16
0| 19>  Receiving result: source=4 tag=15 <- 3.20 | Sending data: dest=4 tag=17
0| 20>  Receiving result: source=2 tag=16 <- 3.40 | Sending data: dest=2 tag=18
0| 21>  Receiving result: source=3 tag=14 <- 3.00 | Sending data: dest=3 tag=19
0| 22>  Receiving result: source=1 tag=12 <- 2.60 | Sending data: dest=1 tag=20
0| 23>  Receiving result: source=4 tag=17 <- 3.60 | Sending data: dest=4 tag=21
0| 24>  Receiving result: source=2 tag=18 <- 3.80 | Sending data: dest=2 tag=22
0| 25>  Receiving result: source=3 tag=19 <- 4.00 | Sending data: dest=3 tag=23
0| 26>  Resulting vector Ab:
0| 27>  0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0
1|  1>  Vector b: ..., 0.01
1|  2>  Receiving data: tag= 0 | Sending result: tag= 0 -> 0.20
1|  3>  Receiving data: tag= 8 | Sending result: tag= 8 -> 1.80
1|  4>  Receiving data: tag=12 | Sending result: tag=12 -> 2.60
1|  5>  Receiving data: tag=20
2|  1>  Vector b: ..., 0.01
2|  2>  Receiving data: tag= 1 | Sending result: tag= 1 -> 0.40
```

```
2|  3>  Receiving data: tag= 5 | Sending result: tag= 5 -> 1.20
2|  4>  Receiving data: tag= 9 | Sending result: tag= 9 -> 2.00
2|  5>  Receiving data: tag=13 | Sending result: tag=13 -> 2.80
2|  6>  Receiving data: tag=16 | Sending result: tag=16 -> 3.40
2|  7>  Receiving data: tag=18 | Sending result: tag=18 -> 3.80
2|  8>  Receiving data: tag=22
3|  1>  Vector b: ..., 0.01
3|  2>  Receiving data: tag= 2 | Sending result: tag= 2 -> 0.60
3|  3>  Receiving data: tag= 6 | Sending result: tag= 6 -> 1.40
3|  4>  Receiving data: tag=10 | Sending result: tag=10 -> 2.20
3|  5>  Receiving data: tag=14 | Sending result: tag=14 -> 3.00
3|  6>  Receiving data: tag=19 | Sending result: tag=19 -> 4.00
3|  7>  Receiving data: tag=23
4|  1>  Vector b: ..., 0.01
4|  2>  Receiving data: tag= 3 | Sending result: tag= 3 -> 0.80
4|  3>  Receiving data: tag= 4 | Sending result: tag= 4 -> 1.00
4|  4>  Receiving data: tag= 7 | Sending result: tag= 7 -> 1.60
4|  5>  Receiving data: tag=11 | Sending result: tag=11 -> 2.40
4|  6>  Receiving data: tag=15 | Sending result: tag=15 -> 3.20
4|  7>  Receiving data: tag=17 | Sending result: tag=17 -> 3.60
4|  8>  Receiving data: tag=21
```

1.5.2. Разработка и запуск параллельных программ без использования электронного задачника

Пример, приведенный в предыдущем пункте, показывает, что применение средств электронного задачника Programming Taskbook for MPI-2 позволяет упростить разработку параллельных программ, не связанных с решением учебных задач.

Однако нашей конечной целью является все же создание параллельных программ, выполнение которых не зависит от наличия на компьютере электронного задачника. В данном пункте мы подробно опишем процесс создания таких программ в среде Microsoft Visual Studio. Аналогичным образом параллельные программы можно разрабатывать и в других средах на локальном компьютере (например, в среде Code::Blocks). Разумеется, параллельные программы обычно предназначаются для запуска на суперкомпьютере или вычислительном кластере, однако при наличии параллельной программы, созданной и протестированной на локальном компьютере, перенос ее на другую вычислительную платформу выполняется достаточно просто и требует лишь знаний о том, как выполнять компиляцию и запуск программы на этой платформе (см., например, главу 4 в книге [4], в которой описаны действия, позволяющие откомпилировать параллельную программу на Unix-сервере с применением компиляторов Intel и GCC и запустить ее на наборе узлов кластера с применением системы управления заданиями PBS).

Как и ранее, будем считать, что программа разрабатывается в среде Microsoft Visual Studio 2017. Создайте в этой среде новое консольное приложение, выполнив для этого следующие стандартные действия:

- начните с команды меню **File | New | Project…** (**Файл | Создать | Проект…**);
- в окне **New Project** (**Создание проекта**) выберите раздел **Visual C++** и в нем вид проекта **Win32 Console Application** (**Консольное приложение Win32**);
- настройте свойства создаваемого проекта: **Name**/**Имя** (будем считать, что указано имя ParallelApplication1) и каталог верхнего уровня **Location**/**Расположение** (будем считать, что выбран рабочий каталог задачника C:\PT4Work), кроме того, снимите выделение с флажка **Create directory for solution** (**Создать каталог для решения**), чтобы не создавать лишний уровень каталогов;
- после настройки свойств проекта нажмите **OK** и настройте дополнительные свойства консольного приложения в окне **Win32 Application Wizard** (**Мастер настройки приложений Win32**): сразу нажмите в этом окне кнопку **Next** (**Далее**), в разделе **Application Settings** (**Параметры приложения**) оставьте выбранным вариант **Console application** (**Консольное приложение**) и *снимите выделение* со всех флажков в группе **Additional options** (**Дополнительные параметры**), в частности, с флажка **Precompiled header** (**Предварительно скомпилированный заголовок**), после чего нажмите кнопку **Finish** (**Готово**).

В результате проект ParallelApplication1 будет создан, и в редактор кода загрузится главный файл этого проекта ParallelApplication1.cpp со следующим содержимым:

```
#include "stdafx.h"

int main()
{
  return 0;
}
```

В других версиях Visual Studio имя стартовой функции и ее параметры могут быть другими. Например, файл, созданный в Visual Studio 2008, будет содержать следующий текст:

```
#include "stdafx.h"

int _tmain(int argc, _TCHAR* argv[])
{
  return 0;
```

```
}
```

Откорректируйте файл ParallelApplication1.cpp, удалив ненужную директиву #include и указав стандартное имя и параметры для стартовой функции:

```
int main(int argc, char* argv[])
{
  return 0;
}
```

Полученная программа должна успешно компилироваться. При ее запуске на экране на долю секунды будет возникать консольное окно.

Теперь добавим к проекту следующие компоненты, позволяющие использовать библиотеку MPI (проще всего скопировать все эти компоненты из рабочего каталога PT4Work в каталог проекта, т. е. в каталог, в котором уже содержится файл ParallelApplication1.cpp):

- mpich.lib (откомпилированная библиотека);
- mpi.h, mpiio.h, mpicxx.h (заголовочные файлы).

Эти файлы можно также скопировать из каталога системы MPICH2: заголовочные файлы содержатся в подкаталоге include, а lib-файл — в подкаталоге lib и имеет имя mpi.lib (ранее уже отмечалось, что имя lib-файла в задачнике было изменено на mpich.lib для согласования с именем lib-файла системы MPICH 1.2.5). В дальнейшем будем считать, что lib-файл имеет имя mpich.lib.

Необходимо подключить lib-файл к проекту. Требуемые для этого действия уже описывались ранее — см. примечание 2 в п. 1.1.3: надо начать с команды меню **Project | <*имя проекта*> Properties…** (**Проект | Свойства <*имя проекта*>…**), перейти в появившемся окне свойств проекта в раздел **Configuration Properties | Linker | Input** (**Свойства конфигурации | Компоновщик | Ввод**) и указать имя подключаемого файла mpich.lib в поле ввода **Additional Dependencies** (**Дополнительные зависимости**), отделив его от последующих имен точкой с запятой. Выполнять особые действия по подключению к проекту заголовочных файлов не требуется.

Добавим в главный файл проекта директивы подключения необходимых заголовков и включим в функцию main операторы, обеспечивающие инициализацию параллельного режима, печать ранга процесса и завершение параллельного режима (добавленный текст выделен полужирным шрифтом):

```
#include <iostream>
#include "mpi.h"

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

```
{
  MPI_Init(&argc, &argv);
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  std::cout << rank << '\n';
  MPI_Finalize();
  return 0;
}
```

Обратите внимание на то, что здесь мы впервые использовали функции MPI_Init и MPI_Finalize (во всех предыдущих программах эти функции вызывались самим задачником).

Если все действия по подключению компонентов MPI были выполнены правильно, то данная программа успешно откомпилируется и запустится на выполнение. При ее первом запуске на экране появится окно системы безопасности Windows, в котором надо нажать кнопку **Разрешить доступ** (рис. 48).

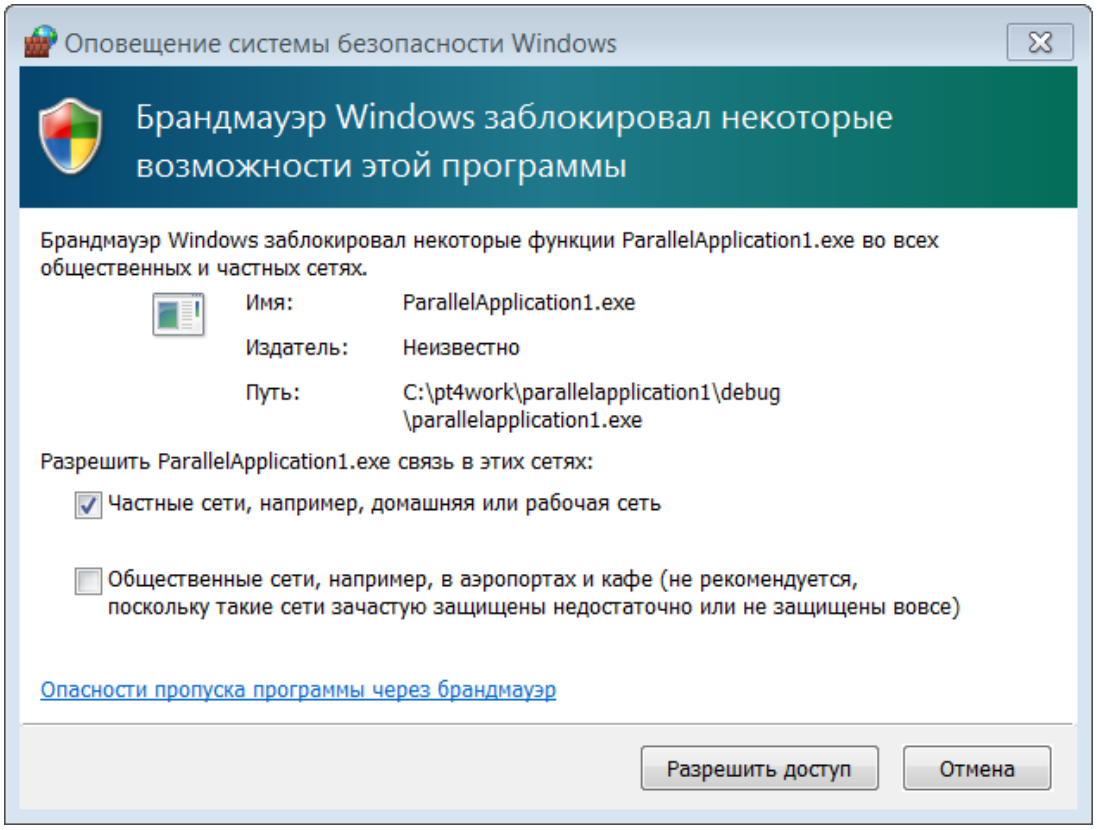

Рис. 48. Окно с запросом о блокировке запущенной параллельной программы

Теперь необходимо подготовить пакетный файл, обеспечивающий запуск полученный программы в параллельном режиме. Назовем этот файл mpi_run.bat и сохраним его в том же каталоге, в котором хранится исполняемый файл ParallelApplication1.exe (это позволит нам не указывать пол-

ный путь к этому файлу). Напомним, что по умолчанию исполняемый файл приложения создается в подкаталоге debug (или bin\debug) каталога приложения (см. путь к запущенному приложению, указанный на рис. 48).

Файл mpi_run.bat должен содержать две команды, каждая команда должна располагаться *на одной строке* (чтобы подчеркнуть это обстоятельство, здесь и далее в аналогичных ситуациях перед продолжением предыдущей строки мы будем указывать специальный символ ↳):

```
"C:\Program Files (x86)\MPICH2\bin\mpiexec.exe" -localonly 10
        ↳ "ParallelApplication1.exe"
pause
```

После запуска этого файла на экране появится консольное окно с результатом работы нашей программы в параллельном режиме (рис. 49).

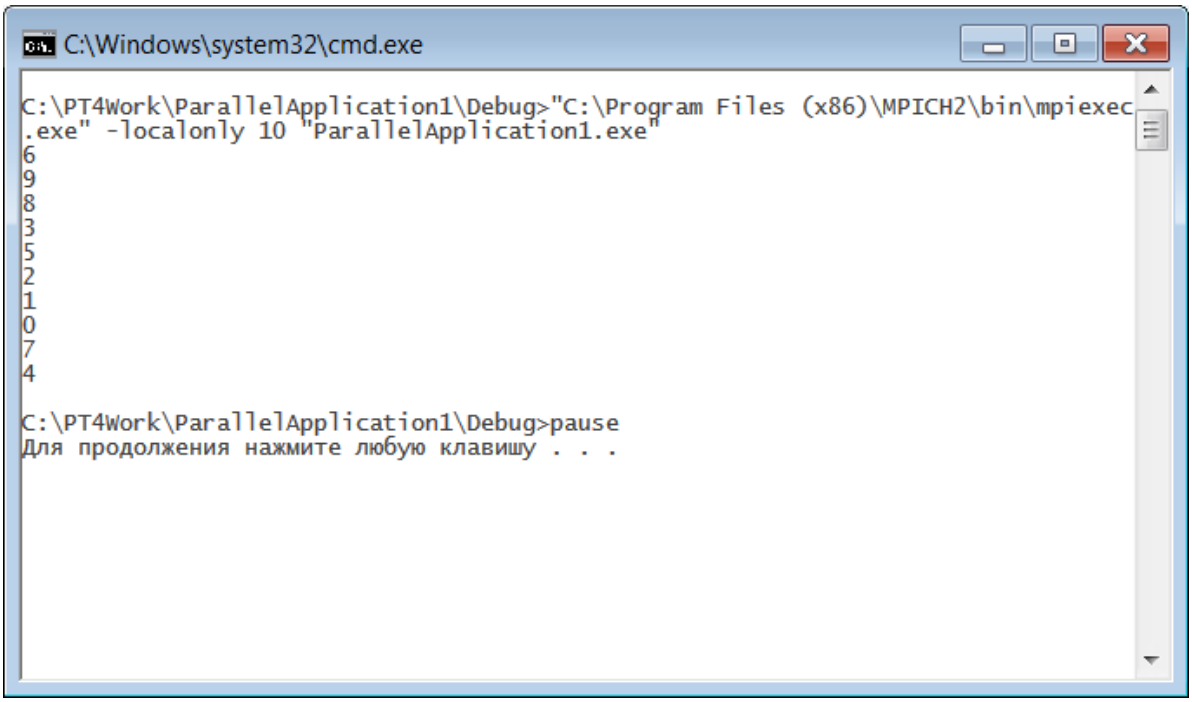

Рис. 49. Результат выполнения параллельной программы ParallelApplication1

Мы видим, что каждый процесс действительно вывел в консольном окне свой ранг. При этом порядок вывода данных из различных процессов является недетерминированным: при новом запуске программы мы будем получать другую последовательность чисел в диапазоне от 0 до 9.

Итак, мы, наконец, разработали параллельную программу, которая может компилироваться и запускаться на выполнение независимо от электронного задачника. Правда, эта программа является очень простой.

> **Примечание**. При разработке параллельных программ в среде Microsoft Visual Studio можно обойтись без вспомогательного bat-файла, если указать управляющую программу mpiexec.exe и ее параметры в настройках проекта (см. примечание 1 в п. 1.1.4, а также

п. 3.3.1, в котором механизм управляющих программ используется для тестирования библиотеки с новой группой заданий). При этом, однако, потребуется добавить в программу фрагмент, обеспечивающий ее приостановку в конце выполнения (можно, например, использовать вызов функции system("pause")).

1.5.3. Дополнительные возможности, связанные с отладочным выводом. Перенаправление вывода

В п. 1.5.1 мы разработали достаточно сложную параллельную программу, реализующую самопланирующий алгоритм умножения матрицы на вектор. При ее разработке и отладке нам помогли возможности задачника, однако теперь мы хотим запустить эту программу независимо от него. Оказывается, сделать это несложно.

Прежде всего, включим в наше консольное приложение файл MPIDebug10.cpp, в котором содержится текст ранее разработанной программы. Для этого скопируем его из каталога PT4Work в каталог консольного приложения (уже содержащий файл ParallelApplication1.cpp) и выполним команду меню **Project | Add Existing Item…** (**Проект | Добавить существующий элемент…**), выбрав в появившемся окне файл MPIDebug10.cpp.

Загрузим файл MPIDebug10.cpp в редактор кода. В настоящий момент он имеет следующий вид (содержимое функции Solve для краткости опускаем):

Файл MPIDebug10.cpp

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

void Solve()
{
  . . .
}
```

При этом функция Solve содержит большое количество вызовов функций, связанных с задачником и взятых из заголовочного файла pt4.h. Разумеется, можно просто удалить директиву подключения файла pt4.h, а также удалить (или закомментировать) все подобные вызовы. Однако в задачнике предусмотрена более удобная возможность: *заглушка* (англ. stub) для заголовочного файла pt4.h. Простейший вариант такой заглушки оформлен в виде файла pt4null.h и содержит «пустые» реализации всех функций, связанных с задачником. Таким образом, можно, не меняя содержимое функции Solve, заменить директиву #include "pt4.h" на следующую директиву:

```
#include "pt4null.h"
```

Необходимо также скопировать файл-заглушку pt4null.h в каталог проекта. Данный файл содержится в каталоге PTforMPI2\stubs системного каталога задачника Programming Taskbook (по умолчанию это каталог PT4, расположенный в каталоге Program Files для 32-разрядных программ — c:\Program Files (x86)).

Выполнив указанные действия, мы обнаружим, что все подчеркивания, связанные с обнаруженными ошибками, в файле MPIDebug10.cpp исчезли. Правда, мы при этом потеряли все возможности нашей программы, связанные с выводом данных на экран.

Мы не будем пытаться (пока) восстановить всю отладочную информацию, которая выводилась в функции Solve, ограничившись печатью элементов итогового произведения — вектора *Ab*. Для этого добавим с начало файла MPIDebug10.cpp директиву #include <iostream>, а после комментария «*вывод полученного вектора Ab*» в функции Solve добавим фрагмент, обеспечивающий вывод массива c. В результате содержимое файла MPIDebug10.cpp станет следующим (все измененные или добавленные в данном пункте операторы выделены полужирным шрифтом):

Файл MPIDebug10.cpp

```
#include <iostream>
#include "pt4null.h"
#include "mpi.h"

void Solve()
{
  . . .
    // вывод полученного вектора Ab:
    for (int i = 0; i < n; ++i)
      std::cout << c[i] << ' ';
    std::cout << '\n';
  . . .
}
```

Нам осталось откорректировать содержимое главного файла приложения ParallelApplication1.cpp таким образом, чтобы в нем вместо печати рангов процессов выполнялся вызов функции Solve (измененные или добавленные фрагменты в данном файле также выделены полужирным шрифтом):

Файл ParallelApplication1.cpp

```
#include <iostream>
#include "mpi.h"

void Solve();
```

```
int main(int argc, char* argv[])
{
  MPI_Init(&argc, &argv);
  Solve();
  MPI_Finalize();
  return 0;
}
```

Теперь осталось откомпилировать и собрать полученный вариант программы, выполнив команду **Build | Build solution** (**Сборка | Собрать решение**) или нажав комбинацию клавиш [Ctrl]+[Shift]+[B], после чего, убедившись, что компиляция и сборка выполнены успешно, запустить файл mpi_run.bat (в который не требуется вносить никаких изменений). Результат работы приведен на рис. 50. Мы видим, что найденные элементы вектора *Ab* совпадают с теми, которые были получены в программе из п. 1.5.1.

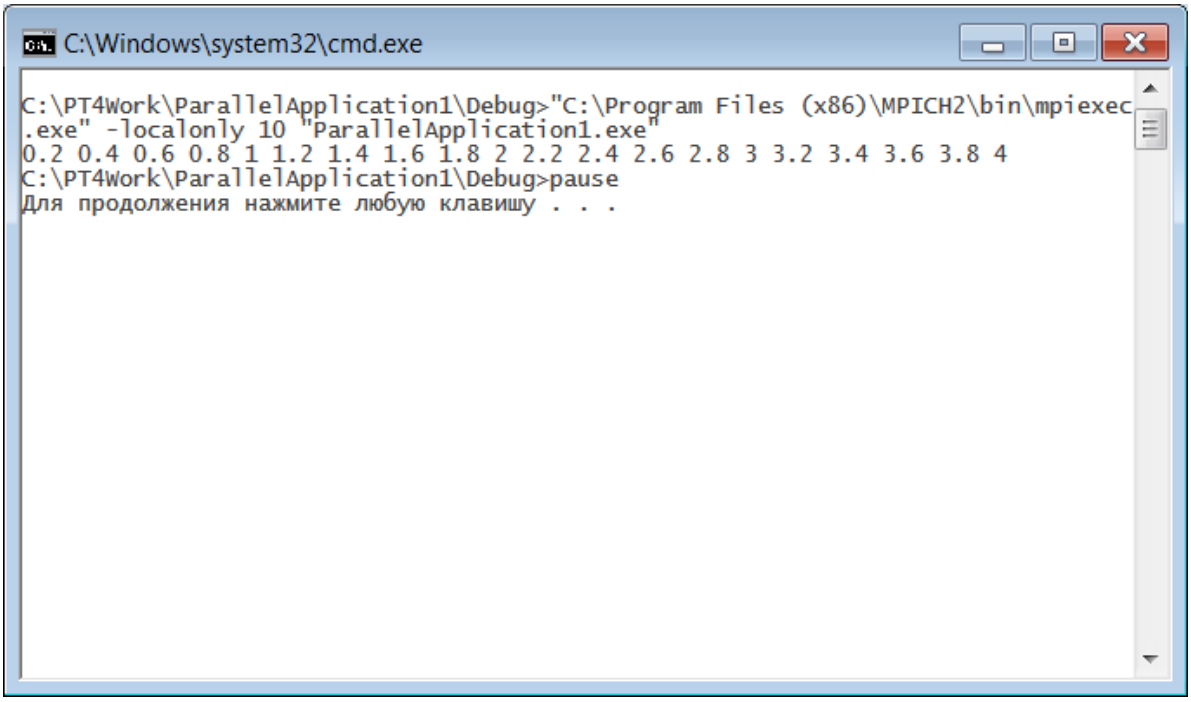

Рис. 50. Результат выполнения программы ParallelApplication1 с заглушкой pt4null.h

В задачнике PT for MPI-2 предусмотрен еще один вариант заглушки для файла pt4.h, который позволяет использовать в «обычной» консольной параллельной программе все возможности по отладочному выводу, предусмотренные в задачнике. Этот вариант заглушки имеет имя pt4console и состоит из двух файлов: pt4console.cpp и pt4console.h. Продемонстрируем его применение.

Прежде всего, скопируем файлы pt4console.cpp и pt4console.h из каталога задачника в каталог консольного приложения (эти файлы располагаются в каталоге задачника там же, где и файл pt4null.h). Затем подключим к консольному приложению файл pt4console.cpp, выполнив для него

команду меню **Project | Add Existing Item…** (**Проект | Добавить существующий элемент…**).

После этого в файле MPIDebug10.cpp заменим директиву #include "pt4null.h" на директиву #include "pt4console.h" и добавим в конец функции Solve вызов функции ShowAll (без параметров):

Файл MPIDebug10.cpp

```
#include <iostream>
#include "pt4console.h"
#include "mpi.h"

void Solve()
{
  . . .
  ShowAll();
}
```

Обратите внимание на то, что функция ShowAll является коллективной и должна вызываться для всех процессов параллельного приложения.

Файл ParallelApplication1.cpp корректировать не требуется.

Откомпилировав и собрав новый вариант программы командой **Build | Build solution** (**Сборка | Собрать решение**) и запустив файл mpi_run.bat, мы получим в консольном окне ту же информацию, которая ранее (см. п. 1.5.1) отображалась в разделе отладки окна задачника (рис. 51).

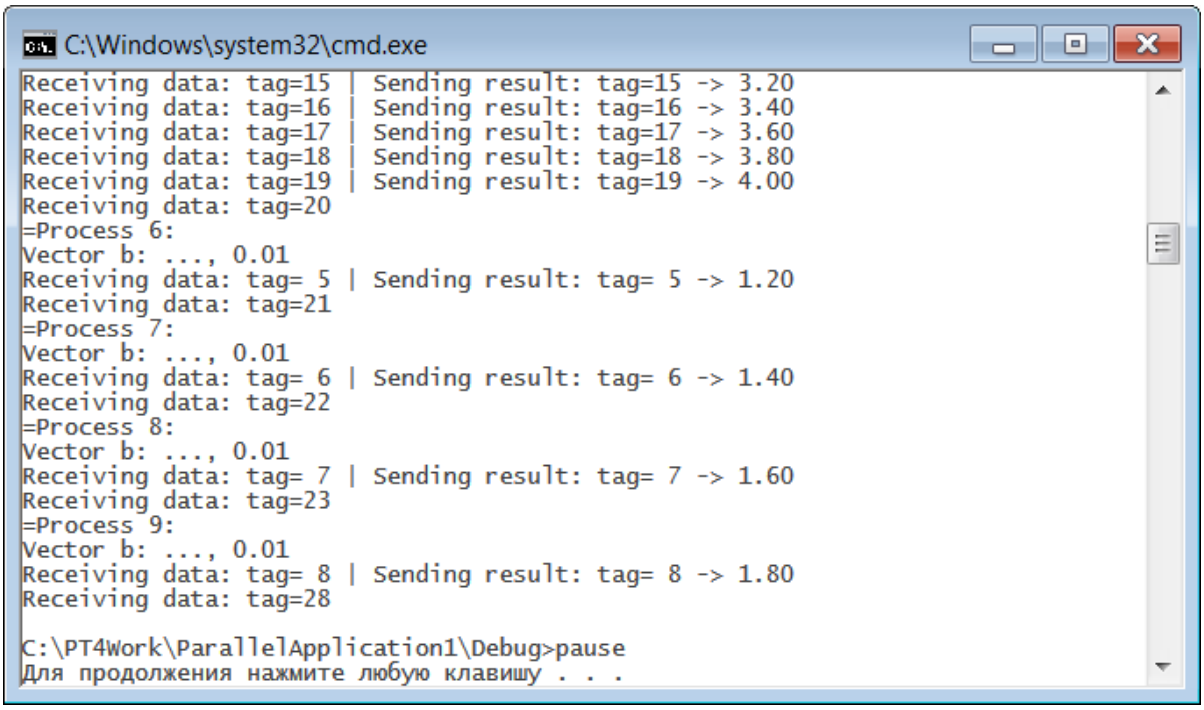

Рис. 51. Результат выполнения программы ParallelApplication1 с заглушкой pt4console.h

Отличие состоит лишь в том, что строки вывода не нумеруются, а ранг процесса указывается в специальной строке, начинающейся со знака равенства «=». Подчеркнем, что данный вариант вывода является *детерминированным*: все данные группируются по процессам, в которых они выведены, а процессы перебираются по возрастанию их рангов.

Если пролистать содержимое консольного окна к началу, то можно убедиться в том, что в главном процессе вначале был выведен полученный вектор *Ab*, а затем начался вывод данных, собранных из функций Show (рис. 52).

```
C:\Windows\system32\cmd.exe

C:\PT4Work\ParallelApplication1\Debug>"C:\Program Files (x86)\MPICH2\bin\mpiexec
.exe" -localonly 10 "ParallelApplication1.exe"
0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2 2.2 2.4 2.6 2.8 3 3.2 3.4 3.6 3.8 4
=Process 0:
Vector b: ..., 0.01
Sending initial data: dest=1 tag=0 -> ..., 1.00
Sending initial data: dest=2 tag=1 -> ..., 2.00
Sending initial data: dest=3 tag=2 -> ..., 3.00
Sending initial data: dest=4 tag=3 -> ..., 4.00
Sending initial data: dest=5 tag=4 -> ..., 5.00
Sending initial data: dest=6 tag=5 -> ..., 6.00
Sending initial data: dest=7 tag=6 -> ..., 7.00
Sending initial data: dest=8 tag=7 -> ..., 8.00
Sending initial data: dest=9 tag=8 -> ..., 9.00
Receiving result: source=1 tag= 0 <- 0.20 | Sending data: dest=1 tag= 9
Receiving result: source=2 tag= 1 <- 0.40 | Sending data: dest=2 tag=10
Receiving result: source=4 tag= 3 <- 0.80 | Sending data: dest=4 tag=11
Receiving result: source=5 tag= 4 <- 1.00 | Sending data: dest=5 tag=12
Receiving result: source=5 tag=12 <- 2.60 | Sending data: dest=5 tag=13
Receiving result: source=5 tag=13 <- 2.80 | Sending data: dest=5 tag=14
Receiving result: source=5 tag=14 <- 3.00 | Sending data: dest=5 tag=15
Receiving result: source=5 tag=15 <- 3.20 | Sending data: dest=5 tag=16
Receiving result: source=5 tag=16 <- 3.40 | Sending data: dest=5 tag=17
Receiving result: source=5 tag=17 <- 3.60 | Sending data: dest=5 tag=18
```

Рис. 52. Начальная часть консольного окна для программы ParallelApplication1 с заглушкой pt4console.h

Читателю рекомендуется проанализировать содержимое файла pt4console.cpp (и прежде всего функции ShowAll), чтобы выяснить, каким образом удалось организовать подобный вывод данных. Вывод на консоль в функции ShowAll выполняет только главный процесс, именно поэтому вывод является полностью детерминированным. Предварительно в этой функции главный процесс собирает от подчиненных процессов все отладочные данные, которые были получены в них с помощью функций Show и ShowLine, после чего выводит их на консоль в требуемом порядке (подчиненные процессы в этой же функции организуют пересылку своих отладочных данных главному процессу).

Завершая обзор примеров, связанных с разработкой параллельных программ без подключения к ним задачника, отметим еще одну полезную возможность.

Часто бывает желательно организовать вывод данных таким образом, чтобы он сохранился после завершения программы. Обычно для этого выполняется сохранение результатов в некотором файле. Для консольных приложений подобное сохранение можно выполнить очень легко, используя **перенаправление потока вывода**. В случае нашей программы следует лишь немного дополнить первую команду из файла mpi_run.bat (добавленный фрагмент выделен полужирным шрифтом и подчеркиванием):

```
"C:\Program Files (x86)\MPICH2\bin\mpiexec.exe" -localonly 10
        ↳ "ParallelApplication1.exe" > results.txt
```

Символ «>» означает, что поток вывода при выполнении данной команды будет перенаправлен из консольного окна в файл, имя которого указано после данного символа (мы выбрали имя results.txt). Если такого файла нет, то он создается, если он уже есть, то его содержимое перезаписывается.

При запуске измененного файла mpi_run.bat консольное окно не будет содержать никаких выведенных данных (рис. 53). В подобной ситуации целесообразно удалить из bat-файла команду pause, поскольку содержимое консольного окна теперь просматривать не требуется.

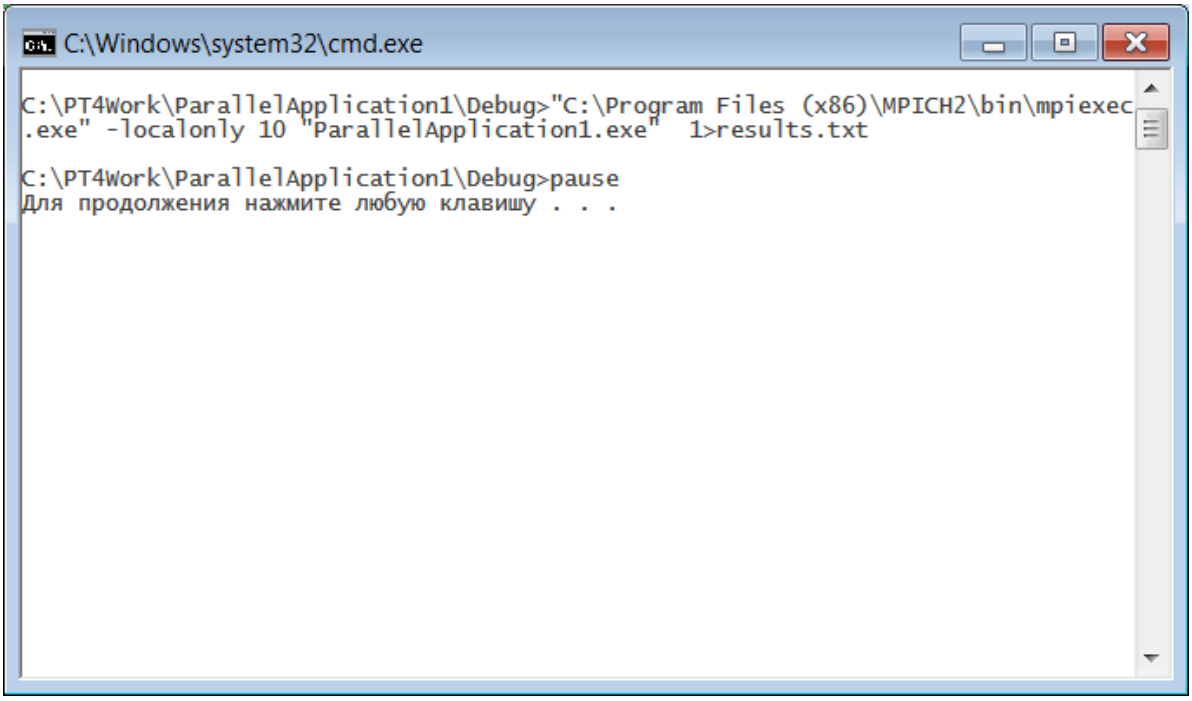

Рис. 53. Результат выполнения программы ParallelApplication1 с перенаправлением потока вывода

Однако теперь на диске в том же каталоге, в котором находится файл mpi_run.bat, появится файл results.txt, содержащий все данные, выведенные нашей программой. Приведем вариант содержимого данного файла (сравните его с вариантами, приведенными в конце п. 1.5.1):

```
0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2 2.2 2.4 2.6 2.8 3 3.2 3.4 3.6 3.8 4
=Process 0:
Vector b: ..., 0.01
Sending initial data: dest=1 tag=0 -> ..., 1.00
Sending initial data: dest=2 tag=1 -> ..., 2.00
Sending initial data: dest=3 tag=2 -> ..., 3.00
Sending initial data: dest=4 tag=3 -> ..., 4.00
Sending initial data: dest=5 tag=4 -> ..., 5.00
Sending initial data: dest=6 tag=5 -> ..., 6.00
Sending initial data: dest=7 tag=6 -> ..., 7.00
Sending initial data: dest=8 tag=7 -> ..., 8.00
Sending initial data: dest=9 tag=8 -> ..., 9.00
Receiving result: source=1 tag= 0 <- 0.20 | Sending data: dest=1 tag= 9
Receiving result: source=2 tag= 1 <- 0.40 | Sending data: dest=2 tag=10
Receiving result: source=4 tag= 3 <- 0.80 | Sending data: dest=4 tag=11
Receiving result: source=5 tag= 4 <- 1.00 | Sending data: dest=5 tag=12
Receiving result: source=6 tag= 5 <- 1.20 | Sending data: dest=6 tag=13
Receiving result: source=7 tag= 6 <- 1.40 | Sending data: dest=7 tag=14
Receiving result: source=7 tag=14 <- 3.00 | Sending data: dest=7 tag=15
Receiving result: source=7 tag=15 <- 3.20 | Sending data: dest=7 tag=16
Receiving result: source=7 tag=16 <- 3.40 | Sending data: dest=7 tag=17
Receiving result: source=7 tag=17 <- 3.60 | Sending data: dest=7 tag=18
Receiving result: source=7 tag=18 <- 3.80 | Sending data: dest=7 tag=19
Receiving result: source=7 tag=19 <- 4.00 | Sending data: dest=7 tag=20
Receiving result: source=8 tag= 7 <- 1.60 | Sending data: dest=8 tag=21
Receiving result: source=9 tag= 8 <- 1.80 | Sending data: dest=9 tag=22
Receiving result: source=1 tag= 9 <- 2.00 | Sending data: dest=1 tag=23
Receiving result: source=2 tag=10 <- 2.20 | Sending data: dest=2 tag=24
Receiving result: source=3 tag= 2 <- 0.60 | Sending data: dest=3 tag=25
Receiving result: source=4 tag=11 <- 2.40 | Sending data: dest=4 tag=26
Receiving result: source=5 tag=12 <- 2.60 | Sending data: dest=5 tag=27
Receiving result: source=6 tag=13 <- 2.80 | Sending data: dest=6 tag=28
Resulting vector Ab:
0.2 0.4 0.6 0.8 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0
=Process 1:
Vector b: ..., 0.01
Receiving data: tag= 0 | Sending result: tag= 0 -> 0.20
Receiving data: tag= 9 | Sending result: tag= 9 -> 2.00
Receiving data: tag=23
=Process 2:
Vector b: ..., 0.01
Receiving data: tag= 1 | Sending result: tag= 1 -> 0.40
Receiving data: tag=10 | Sending result: tag=10 -> 2.20
Receiving data: tag=24
=Process 3:
Vector b: ..., 0.01
Receiving data: tag= 2 | Sending result: tag= 2 -> 0.60
Receiving data: tag=25
=Process 4:
```

```
Vector b: ..., 0.01
Receiving data: tag= 3 | Sending result: tag= 3 -> 0.80
Receiving data: tag=11 | Sending result: tag=11 -> 2.40
Receiving data: tag=26
=Process 5:
Vector b: ..., 0.01
Receiving data: tag= 4 | Sending result: tag= 4 -> 1.00
Receiving data: tag=12 | Sending result: tag=12 -> 2.60
Receiving data: tag=27
=Process 6:
Vector b: ..., 0.01
Receiving data: tag= 5 | Sending result: tag= 5 -> 1.20
Receiving data: tag=13 | Sending result: tag=13 -> 2.80
Receiving data: tag=28
=Process 7:
Vector b: ..., 0.01
Receiving data: tag= 6 | Sending result: tag= 6 -> 1.40
Receiving data: tag=14 | Sending result: tag=14 -> 3.00
Receiving data: tag=15 | Sending result: tag=15 -> 3.20
Receiving data: tag=16 | Sending result: tag=16 -> 3.40
Receiving data: tag=17 | Sending result: tag=17 -> 3.60
Receiving data: tag=18 | Sending result: tag=18 -> 3.80
Receiving data: tag=19 | Sending result: tag=19 -> 4.00
Receiving data: tag=20
=Process 8:
Vector b: ..., 0.01
Receiving data: tag= 7 | Sending result: tag= 7 -> 1.60
Receiving data: tag=21
=Process 9:
Vector b: ..., 0.01
Receiving data: tag= 8 | Sending result: tag= 8 -> 1.80
Receiving data: tag=22
```

Раздел 2. Учебные задания

Если количество процессов в задании не определено, то можно считать, что это количество не превосходит 16. Под *главным процессом* всюду в формулировках заданий понимается процесс ранга 0 для коммуникатора MPI_COMM_WORLD. Для всех процессов ненулевого ранга в заданиях используется общее наименование *подчиненных процессов*.

Если в задании не указан тип обрабатываемых чисел, то числа считаются вещественными. Если в задании не определяется максимальный размер набора чисел, то его следует считать равным 20.

Задачи повышенного уровня сложности помечены символами * и **. Задачи, решения которых приведены в разделе 1, помечены символом °.

2.1. Процессы и их ранги

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи MPI1Proc2, приведенным в п. 1.1.

MPI1Proc1. В каждом из процессов, входящих в коммуникатор MPI_COMM_WORLD, прочесть одно вещественное число и вывести его противоположное значение. Для ввода и вывода данных использовать поток ввода-вывода pt, определенный в задачнике. Кроме того, отобразить найденное значение в разделе отладки, используя функцию Show, также определенную в задачнике.

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).

Примечание. Решение данной задачи приведено в п. 1.1.

MPI1Proc3. В главном процессе прочесть вещественное число X и вывести его противоположное значение, в каждом из остальных процессов (*подчиненных процессов*, ранг которых больше 0) вывести его ранг. Кроме того, продублировать вывод данных в разделе отладки, отобра-

зив значение $-X$ с комментарием «-X=», а значения рангов с комментариями «rank=» (использовать функцию Show с двумя параметрами).

MPI1Proc4. В процессах четного ранга (включая главный) ввести целое число и вывести его удвоенное значение. В процессах нечетного ранга не выполнять никаких действий.

MPI1Proc5. В процессах четного ранга (включая главный) ввести целое число, в процессах нечетного ранга ввести вещественное число. В каждом процессе вывести удвоенное значение введенного числа.

MPI1Proc6. В подчиненных процессах четного ранга ввести целое число, в процессах нечетного ранга ввести вещественное число. В каждом подчиненном процессе вывести удвоенное значение введенного числа. В главном процессе не выполнять никаких действий.

MPI1Proc7. В каждом процессе четного ранга (включая главный) дано целое число N (> 0) и набор из N вещественных чисел. Вывести в каждом из этих процессов сумму чисел из данного набора. В процессах нечетного ранга не выполнять никаких действий.

MPI1Proc8. В каждом процессе дано целое число N (> 0) и набор из N вещественных чисел. В процессах четного ранга (включая главный) вывести сумму чисел из данного набора, в процессах нечетного ранга вывести среднее арифметическое чисел из данного набора.

MPI1Proc9. В каждом процессе дано целое число N (> 0) и набор из N вещественных чисел. В подчиненных процессах четного ранга вывести сумму чисел из данного набора, в процессах нечетного ранга вывести среднее арифметическое чисел из данного набора, в главном процессе вывести произведение чисел из данного набора.

MPI1Proc10. В каждом процессе дано целое число N (> 0) и набор из N чисел, причем в подчиненных процессах нечетного ранга (1, 3, …) набор содержит вещественные числа, в подчиненных процессах четного ранга (2, 4, …) — целые числа, а тип элементов в главном процессе зависит от общего количества процессов: если количество процессов нечетное, то набор содержит целые числа, а если четное, то вещественные. В процессах четного ранга (включая главный) вывести минимальный элемент из данного набора, в процессах нечетного ранга вывести максимальный элемент.

2.2. Обмен сообщениями между отдельными процессами

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи MPI2Send11, приведенным в п. 1.2.2.

2.2.1. Блокирующая пересылка данных

MPI2Send1. В каждом подчиненном процессе дано целое число. Переслать эти числа в главный процесс, используя функции MPI_Send и MPI_Recv (стандартные блокирующие функции для передачи и приема сообщения), и вывести их в главном процессе. Полученные числа выводить в порядке возрастания рангов переславших их процессов.

MPI2Send2. В каждом подчиненном процессе дано вещественное число. Переслать эти числа в главный процесс, используя функции MPI_Bsend (посылка сообщения с буферизацией) и MPI_Recv, и вывести их в главном процессе. Полученные числа выводить в порядке убывания рангов переславших их процессов. Для задания буфера использовать функцию MPI_Buffer_attach.

MPI2Send3. В каждом подчиненном процессе даны четыре целых числа. Переслать эти числа в главный процесс, используя по одному вызову функции MPI_Send для каждого передающего процесса, и вывести их в главном процессе. Полученные числа выводить в порядке возрастания рангов переславших их процессов.

MPI2Send4. В каждом подчиненном процессе дано целое число N ($0 < N < 5$) и набор из N целых чисел. Переслать данные наборы в главный процесс, используя по одному вызову функции MPI_Bsend для каждого передающего процесса, и вывести наборы в главном процессе в порядке возрастания рангов переславших их процессов. Для определения размера пересланного набора использовать функцию MPI_Get_count.

MPI2Send5. В главном процессе дан набор вещественных чисел; количество чисел равно количеству подчиненных процессов. С помощью функции MPI_Send переслать по одному числу в каждый из подчиненных процессов (первое число в процесс 1, второе — в процесс 2, и т. д.) и вывести в подчиненных процессах полученные числа.

MPI2Send6. В главном процессе дан набор вещественных чисел; количество чисел равно количеству подчиненных процессов. С помощью функции MPI_Bsend переслать по одному числу в каждый из подчиненных процессов, перебирая процессы в обратном порядке (первое число в последний процесс, второе — в предпоследний процесс, и т. д.), и вывести в подчиненных процессах полученные числа.

MPI2Send7. В главном процессе дано целое число N и набор из N чисел; $K - 1 \le N < 10$, где K — количество процессов. С помощью функции MPI_Send переслать по одному числу их данного набора в процессы 1, 2, …, $K - 2$, а оставшиеся числа — в процесс $K - 1$, и вывести полученные числа. В процессе $K - 1$ для определения количества полученных чисел использовать функцию MPI_Get_count.

MPI2Send8. В каждом подчиненном процессе дано целое число, причем только для одного процесса это число отлично от нуля. Переслать ненулевое число в главный процесс и вывести в главном процессе полученное число и ранг процесса, переславшего это число. Для приема сообщения в главном процессе использовать функцию MPI_Recv с параметром MPI_ANY_SOURCE.

MPI2Send9. В каждом подчиненном процессе дано целое число N, причем для одного процесса это число больше нуля, а для остальных равно нулю. В процессе с ненулевым N дан также набор из N чисел. Переслать данный набор чисел в главный процесс и вывести в главном процессе полученные числа и ранг процесса, переславшего этот набор. При приеме сообщения использовать параметр MPI_ANY_SOURCE.

MPI2Send10. В каждом подчиненном процессе дано целое число N, в главном процессе дано целое число K (> 0), равное количеству тех подчиненных процессов, в которых даны положительные числа N. Переслать все положительные числа N в главный процесс и вывести в нем сумму полученных чисел. Для приема сообщений в главном процессе использовать функцию MPI_Recv с параметром MPI_ANY_SOURCE.

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.

Примечание. Решение данной задачи приведено в п. 1.2.2.

MPI2Send12. В каждом процессе дано целое число. С помощью функций MPI_Ssend и MPI_Recv осуществить для всех процессов циклический сдвиг данных с шагом 1, переслав число из процесса 0 в процесс 1, из процесса 1 в процесс 2, …, из последнего процесса в процесс 0. В каждом процессе вывести полученное число.

Указание. См. указание к задаче MPI2Send11.

MPI2Send13. В каждом процессе дано целое число. С помощью функций MPI_Ssend и MPI_Recv осуществить для всех процессов циклический сдвиг данных с шагом −1, переслав число из процесса 1 в процесс 0,

из процесса 2 в процесс 1, …, из процесса 0 в последний процесс. В каждом процессе вывести полученное число.

Указание. См. указание к задаче MPI2Send11.

MPI2Send14. В каждом процессе даны два целых числа. С помощью функций MPI_Ssend и MPI_Recv переслать первое число в предыдущий процесс, а второе — в последующий процесс (для процесса 0 считать предыдущим последний процесс, а для последнего процесса считать последующим процесс 0). В каждом процессе вывести числа, полученные от предыдущего и последующего процесса (в указанном порядке).

Указание. См. указание к задаче MPI2Send11.

MPI2Send15. В каждом процессе даны два числа: вещественное A и целое N, причем набор чисел N содержит все значения от 0 до $K-1$, где K — количество процессов. Используя функции MPI_Send и MPI_Recv (с параметром MPI_ANY_SOURCE), выполнить в каждом процессе пересылку числа A в процесс N и вывести полученное число, а также ранг процесса, из которого число было получено.

MPI2Send16. В каждом процессе дано целое число N, причем для одного процесса значение N равно 1, а для остальных равно 0. В процессе с $N = 1$ дан также набор из $K-1$ числа, где K — количество процессов. Переслать из этого процесса по одному из чисел данного набора в остальные процессы, перебирая ранги получателей в возрастающем порядке, и вывести в каждом из них полученное число.

MPI2Send17. В каждом процессе дан набор из $K-1$ целого числа, где K — количество процессов. Для каждого процесса переслать по одному из данных в нем чисел в остальные процессы, перебирая ранги процессов-получателей в возрастающем порядке, и вывести полученные числа в порядке возрастания рангов переславших их процессов.

MPI2Send18. Количество процессов — четное число. В каждом процессе дано целое число N ($0 < N < 5$) и набор из N чисел. С помощью функции MPI_Sendrecv выполнить обмен исходными наборами между парами процессов 0 и 1, 2 и 3, и т. д. В каждом процессе вывести полученный набор чисел.

MPI2Send19. В каждом процессе дано вещественное число. С помощью функции MPI_Sendrecv_replace поменять порядок исходных чисел на обратный (число из процесса 0 должно быть передано в последний процесс, число из процесса 1 — в предпоследний процесс, …, число из последнего процесса — в процесс 0). В каждом процессе вывести полученное число.

MPI2Send20. В каждом подчиненном процессе дано вещественное число A и его порядковый номер N (целое число); набор всех номеров N со-

держит все целые числа от 1 до $K - 1$, где K — количество процессов. Переслать числа A в главный процесс и вывести их в порядке, соответствующем возрастанию их номеров N. Массивы не использовать; для передачи номера N указывать его в качестве параметра msgtag функции MPI_Send.

MPI2Send21. В каждом подчиненном процессе дано целое число L (≥ 0) и набор из L пар чисел (A, N), где A — вещественное число, а N — его порядковый номер. Все числа L в сумме равны $2K$, где K — количество процессов; набор номеров N, данных во всех процессах, содержит все целые числа от 1 до $2K$. Переслать числа A в главный процесс и вывести их в порядке, соответствующем возрастанию их номеров N. Для передачи номера N указывать его в качестве параметра msgtag функции MPI_Send.

MPI2Send22*. В главном процессе дан набор пар чисел (T, A); количество пар равно числу подчиненных процессов. Число T — целое, равное 0 или 1. Число A — целое, если $T = 0$, и вещественное, если $T = 1$. Переслать по одному числу A в каждый из подчиненных процессов (первое число в процесс 1, второе — в процесс 2, и т. д.) и вывести полученные числа. Для передачи информации о типе пересланного числа указывать число T в качестве параметра msgtag функции MPI_Send, для получения этой информации использовать функцию MPI_Probe с параметром MPI_ANY_TAG.

Указание. Чтобы избежать дублирования кода, используйте вспомогательные *шаблонные функции* template<typename T> void send(int t, int dest, MPI_Datatype d) (для отправки данных) и template<typename T> void recv(MPI_Datatype d) (для получения данных). В качестве параметра t указывайте число, равное 0 или 1, в качестве параметра dest — ранг процесса-получателя.

MPI2Send23*. В каждом подчиненном процессе даны два целых числа T, N и набор из N чисел. Число T равно 0 или 1. Набор содержит целые числа, если $T = 0$, и вещественные числа, если $T = 1$. Переслать исходные наборы в главный процесс и вывести полученные числа в порядке возрастания рангов переславших их процессов. Для передачи информации о типе пересланных чисел указывать число T в качестве параметра msgtag функции MPI_Send, для получения этой информации использовать функцию MPI_Probe с параметром MPI_ANY_TAG.

Указание. Чтобы избежать дублирования кода, используйте вспомогательные *шаблонные функции* template<typename T> void send(int t, MPI_Datatype d) (для отправки данных) и template<typename T> void recv(MPI_Datatype d, MPI_Status s) (для получения данных). В качестве па-

раметра t указывайте число, равное 0 или 1, в качестве параметра s — результат, возвращенный функцией MPI_Probe.

MPI2Send24*. Количество процессов K является четным: $K = 2N$. В процессах четного ранга (0, 2, …, $K - 2$) дан набор из N вещественных чисел, в процессах нечетного ранга (1, 3, …, $K - 1$) — набор из N целых чисел. Используя функцию MPI_Sendrecv_replace, выполнить циклический сдвиг всех наборов вещественных чисел в направлении увеличения рангов процессов и циклический сдвиг всех наборов целых чисел в направлении уменьшения рангов (таким образом, вещественные наборы надо переслать из процесса 0 в процесс 2, из процесса 2 — в процесс 4, …, из процесса $K - 2$ — в процесс 0; целочисленные наборы надо переслать из процесса $K - 1$ в процесс $K - 3$, из процесса $K - 3$ в процесс $K - 5$, …, из процесса 1 в процесс $K - 1$). Вывести в каждом процессе полученные числа. Для определения рангов процессов-получателей использовать выражение, содержащее операцию % нахождения остатка от деления, в качестве ранга процесса-отправителя достаточно указывать константу MPI_ANY_SOURCE.

Указание. Чтобы избежать дублирования кода, используйте вспомогательную *шаблонную функцию* template<typename T> void sendrecv(int rank, int size, MPI_Datatype d, int step), где параметр step указывает величину сдвига, равную 2 для наборов вещественных чисел и −2 для целочисленных наборов.

MPI2Send25*. Количество процессов K является четным: $K = 2N$. В первой половине процессов дан набор целых чисел размера $R + 1$, где R — ранг процесса ($R = 0, 1, \ldots, N - 1$), во второй половине процессов дан набор вещественных чисел размера $2N - R$, где R — ранг процесса ($R = N, N + 1, \ldots, 2N - 1$). Используя функцию MPI_Sendrecv, переслать исходные наборы из каждой половины процессов в соответствующий процесс другой половины (в частности, набор из процесса 0 надо переслать в процесс N, из процесса 1 — в процесс $N + 1$, из процесса N — в процесс 0, из процесса $2N - 1$ — в процесс $N - 1$). Вывести в каждом процессе полученные числа.

Указание. Чтобы избежать дублирования кода, используйте вспомогательную *шаблонную функцию* template<typename T1, typename T2> void sendrecv(MPI_Datatype d1, int cnt1, int rank2, MPI_Datatype d2, int cnt2), где параметры d1 и cnt1 определяют характеристики процесса, вызвавшего функцию (тип элементов в наборе и их количество), а параметры rank2, d2, cnt2 — ранг и аналогичные характеристики процесса, с которым выполняется обмен данными.

2.2.2. Неблокирующая пересылка данных

MPI2Send26*. В каждом процессе дано целое число N; во всех процессах, кроме одного, значение N равно 0, в некотором выделенном процессе значение N равно 1. В выделенном процессе также дан набор целых чисел A размера $K-1$, где K — количество процессов. Не сохраняя набор A в массиве, переслать по одному элементу этого набора из выделенного процесса во все остальные процессы, перебирая их в порядке возрастания рангов, и вывести в каждом процессе полученное число. Для пересылки данных использовать требуемое количество вызовов функций MPI_Issend (посылка сообщения в синхронном неблокирующем режиме) и MPI_Wait в выделенном процессе и функцию MPI_Recv в остальных процессах. Дополнительно отобразить в разделе отладки продолжительность каждого вызова функции MPI_Wait в миллисекундах; для этого вызвать функцию MPI_Wtime до и после MPI_Wait и с помощью функции Show вывести разность возвращенных функцией MPI_Wtime значений, умноженную на 1000. Проверить, как изменится отладочная информация, если вместо функции MPI_Issend использовать функцию MPI_Isend (посылка сообщения в стандартном неблокирующем режиме).

MPI2Send27*. В каждом процессе дано целое число N; в некотором выделенном процессе значение N равно -1, а в остальных процессах N является одинаковым и равно рангу R выделенного процесса. Во всех процессах, кроме выделенного, также дано вещественное число A. Переслать данные числа A в выделенный процесс и вывести их в порядке возрастания рангов процессов-отправителей. Для пересылки данных использовать требуемое количество вызовов функции MPI_Recv в выделенном процессе и функции MPI_Issend и MPI_Test в остальных процессах. Вызов функции MPI_Test повторять в цикле до тех пор, пока она не вернет ненулевой флаг, и отобразить в разделе отладки потребовавшееся количество итераций этого цикла, используя функцию Show. Проверить, как изменится отладочная информация, если вместо функции MPI_Issend использовать функцию MPI_Isend.

MPI2Send28*. В каждом процессе дано целое число N; в некотором выделенном процессе значение N равно -1, а в остальных процессах N является одинаковым и равно рангу R выделенного процесса. Во всех процессах, кроме выделенного, также дано вещественное число A. Переслать данные числа A в выделенный процесс и вывести их в порядке убывания рангов процессов-отправителей. Для пересылки данных использовать требуемое количество вызовов функций MPI_Irecv (прием сообщения в неблокирующем режиме) и MPI_Test в выделенном процессе и функцию MPI_Ssend в остальных процессах. После каждого вы-

зова функции MPI_Irecv организовать цикл, в котором вызывать функцию MPI_Test, пока она не вернет ненулевой флаг, и отображать в разделе отладки потребовавшееся количество итераций этого цикла, используя функцию Show. Проверить, как изменится отладочная информация, если вместо функции MPI_Ssend использовать функцию MPI_Send.

MPI2Send29*. В каждом процессе дано целое число N; в некотором выделенном процессе значение N равно -1, а в остальных процессах N является одинаковым и равно рангу R выделенного процесса. Во всех процессах, кроме выделенного, также дано вещественное число A. Переслать данные числа A в выделенный процесс и вывести их сумму S. Для пересылки данных использовать требуемое количество вызовов функций MPI_Irecv и MPI_Waitany в выделенном процессе и функцию MPI_Ssend в остальных процессах. В выделенном процессе описать массив Q типа MPI_Request и организовать вызов функций MPI_Irecv в отдельном цикле, указывая для каждого вызова свой элемент массива Q. После выхода из этого цикла организовать второй цикл, в котором вызывать функцию MPI_Waitany и накапливать сумму S. Дополнительно на каждой итерации второго цикла отображать в разделе отладки следующие данные (используя по одному вызову функций Show и ShowLine): значение A, добавленное к сумме на данной итерации цикла, и ранг процесса, переславшего это значение.

MPI2Send30*. В каждом процессе дано целое число N; во всех процессах, за исключением двух, значение N равно 0; в одном из оставшихся процессов (*процессе-отправителе*) значение N равно 1, в другом (*процессе-получателе*) значение N равно 2. В процессе-отправителе также дано целое число R — ранг процесса получателя и набор A целых чисел размера K, где K — количество процессов. Не сохраняя набор A в массиве, переслать все его элементы процессу-получателю и вывести их в том же порядке. Для пересылки данных использовать единственный вызов функции MPI_Ssend_init и требуемое количество вызовов функций MPI_Start и MPI_Wait в процессе-отправителе и единственный вызов функции MPI_Recv_init и требуемое количество вызовов функций MPI_Start и MPI_Wait в процессе-получателе. Дополнительно отобразить в разделе отладки продолжительность каждого вызова функции MPI_Wait в миллисекундах (как для процесса-отправителя, так и для процесса-получателя); для этого вызвать функцию MPI_Wtime до и после MPI_Wait и с помощью функции Show вывести разность возвращенных функцией MPI_Wtime значений, умноженную на 1000. Проверить, как изменится отладочная информация, если вместо функции MPI_Ssend_init использовать функцию MPI_Send_init.

MPI2Send31*. В каждом процессе дано целое число *N*; в одном из процессов (*процессе-получателе*) значение *N* равно 2, в некоторых процессах (*процессах-отправителях*) значение *N* равно 1, в остальных процессах значение *N* равно 0. В каждом из процессов-отправителей также дано целое число *R* — ранг процесса-получателя и набор *A* целых чисел размера *K*, где *K* — количество процессов. В процессе-получателе дано целое число *C* — количество процессов-отправителей. Переслать все наборы *A* процессу-получателю и вывести набор *S* суммарных значений элементов всех наборов *A* с одинаковыми индексами (в порядке возрастания индексов). Для пересылки данных использовать единственный вызов функции MPI_Ssend в процессах-отправителях. В процессе-получателе описать массив *Q* типа MPI_Request и организовать вызов функций MPI_Recv_init в отдельном цикле, указывая для каждого вызова свой элемент массива *Q*. После выхода из этого цикла выполнить единственный вызов функции MPI_Startall и организовать второй цикл, в котором вызывать функцию MPI_Waitany и накапливать суммарные значения в массиве *S*. Дополнительно на каждой итерации второго цикла отображать в разделе отладки следующие данные (используя два вызова функции Show и один вызов ShowLine): продолжительность каждого вызова функции MPI_Waitany в миллисекундах, значение третьего параметра index, возвращенное функцией MPI_Waitany, и ранг процесса-отправителя, соответствующий параметру index. Для нахождения продолжительности вызвать функцию MPI_Wtime до и после MPI_Waitany и вычислить разность возвращенных функцией MPI_Wtime значений, умножив ее на 1000. Для нахождения ранга процесса-отправителя проанализировать значение последнего параметра (типа MPI_Status), возвращенного функцией MPI_Waitany.

MPI2Send32*. В каждом процессе дано целое число *N*; в одном из процессов (*процессе-отправителе*) значение *N* равно 1, в некоторых процессах (*процессах-получателях*) значение *N* равно 2, в остальных процессах значение *N* равно 0. В процессе-отправителе также дано вещественное число *A*, целое число *C* — количество процессов-получателей и набор целых чисел *R* размера *C*, содержащий ранги процессов-получателей. Переслать число *A* всем процессам-получателям и вывести его в каждом из этих процессов. Для пересылки данных использовать функцию MPI_Recv в процессах-получателях. В процессе-отправителе описать массив *Q* типа MPI_Request и организовать вызов функций MPI_Ssend_init в отдельном цикле, указывая для каждого вызова свой элемент массива *Q*. После выхода из этого цикла выполнить единственный вызов функции MPI_Startall и организовать второй цикл, в котором вызывать функцию MPI_Testany во вложенном цикле, пока она не вернет ненулевой флаг. Дополнительно на каждой итерации

второго цикла отображать в разделе отладки следующие данные (используя по одному вызову функций Show и ShowLine): значение третьего параметра index, возвращенное функцией MPI_Testany (в ситуации, когда она вернула ненулевой флаг), и количество произведенных вызовов функции MPI_Testany (т. е. количество итераций вложенного цикла). Проверить, как изменится отладочная информация, если вместо функции MPI_Ssend_init использовать функцию MPI_Send_init.

2.3. Коллективные взаимодействия

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи MPI3Coll23, приведенным в п. 1.2.5.

2.3.1. Коллективная пересылка данных

MPI3Coll1. В главном процессе дано целое число. Используя функцию MPI_Bcast, переслать это число во все подчиненные процессы и вывести в них полученное число.

MPI3Coll2. В главном процессе дан набор из 5 чисел. Используя функцию MPI_Bcast, переслать этот набор во все подчиненные процессы и вывести в них полученные числа в том же порядке.

MPI3Coll3. В каждом процессе дано вещественное число. Используя функцию MPI_Gather, переслать эти числа в главный процесс и вывести их в порядке возрастания рангов переславших их процессов (первым вывести число, данное в главном процессе).

MPI3Coll4. В каждом процессе дан набор из 5 целых чисел. Используя функцию MPI_Gather, переслать эти наборы в главный процесс и вывести их в порядке возрастания рангов переславших их процессов (первым вывести набор чисел, данный в главном процессе).

MPI3Coll5. В каждом процессе дан набор из $R + 2$ целых чисел, где число R равно рангу процесса (в процессе 0 даны 2 числа, в процессе 1 даны 3 числа, и т. д.). Используя функцию MPI_Gatherv, переслать эти наборы в главный процесс и вывести полученные наборы в порядке возрастания рангов переславших их процессов (первым вывести набор, данный в главном процессе).

MPI3Coll6. В главном процессе дан набор из K чисел, где K — количество процессов. Используя функцию MPI_Scatter, переслать по одному числу в каждый процесс (включая главный) и вывести в каждом процессе полученное число.

MPI3Coll7. В главном процессе дан набор из $3K$ чисел, где K — количество процессов. Используя функцию MPI_Scatter, переслать по 3 числа в каждый процесс (включая главный) и вывести в каждом процессе полученные числа.

MPI3Coll8. В главном процессе дан набор из K чисел, где K — количество процессов. Не меняя порядок расположения чисел в исходном наборе и используя функцию MPI_Scatterv, переслать по одному числу в каждый процесс; при этом первое число надо переслать в процесс $K - 1$, второе число — в процесс $K - 2$, …, последнее число — в процесс 0. Вывести в каждом процессе полученное число.

MPI3Coll9. В главном процессе дан набор из $K(K + 3)/2$ целых чисел, где K — количество процессов. Используя функцию MPI_Scatterv, переслать в каждый процесс часть чисел из данного набора; при этом в процесс ранга R надо переслать $R + 2$ очередных числа (в процесс 0 — первые два числа, в процесс 1 — следующие три числа, и т. д.). В каждом процессе вывести полученные числа.

MPI3Coll10. В главном процессе дан набор из $K + 2$ чисел, где K — количество процессов. Используя функцию MPI_Scatterv, переслать в каждый процесс три числа из данного набора; при этом в процесс ранга R должны быть пересланы числа с номерами от $R + 1$ до $R + 3$ (в процесс 0 — первые три числа, в процесс 1 — числа со второго по четвертое, и т. д.). В каждом процессе вывести полученные числа.

MPI3Coll11. В каждом процессе дано вещественное число. Используя функцию MPI_Allgather, переслать эти числа во все процессы и вывести их в каждом процессе в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

MPI3Coll12. В каждом процессе даны четыре целых числа. Используя функцию MPI_Allgather, переслать эти числа во все процессы и вывести их в каждом процессе в порядке возрастания рангов переславших их процессов (включая числа, полученные из этого же процесса).

MPI3Coll13. В каждом процессе дан набор из $R + 2$ целых чисел, где число R равно рангу процесса (в процессе 0 даны 2 числа, в процессе 1 даны 3 числа, и т. д.). Используя функцию MPI_Allgatherv, переслать эти наборы во все процессы и вывести их в порядке возрастания рангов переславших их процессов (включая числа, полученные из этого же процесса).

MPI3Coll14. В каждом процессе дан набор из K чисел, где K — количество процессов. Используя функцию MPI_Alltoall, переслать в каждый процесс по одному числу из всех наборов: в процесс 0 — первые числа из наборов, в процесс 1 — вторые числа, и т. д. В каждом процессе вывести числа в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

MPI3Coll15. В каждом процессе дан набор из $3K$ целых чисел, где K — количество процессов. Используя функцию MPI_Alltoall, переслать в каждый процесс три очередных числа из каждого набора (в процесс 0 —

первые три числа, в процесс 1 — следующие три числа, и т. д.). В каждом процессе вывести числа в порядке возрастания рангов переславших их процессов (включая числа, полученные из этого же процесса).

MPI3Coll16*. В каждом процессе дан набор из $K(K + 1)/2$ целых чисел, где K — количество процессов. Используя функцию MPI_Alltoallv, переслать в каждый процесс часть чисел из каждого набора; при этом в процесс R должно быть переслано $R + 1$ очередное число (в процесс 0 — первое число каждого набора, в процесс 1 — следующие два числа, и т. д.). В каждом процессе вывести полученные числа.

MPI3Coll17*. В каждом процессе дан набор из $K + 1$ числа, где K — количество процессов. Используя функцию MPI_Alltoallv, переслать в каждый процесс два числа из каждого набора; при этом в процесс 0 надо переслать первые два числа, в процесс 1 — второе и третье число, …, в последний процесс — последние два числа каждого набора. В каждом процессе вывести полученные числа.

MPI3Coll18*. В каждом процессе дан набор из $K + 1$ числа, где K — количество процессов. Используя функцию MPI_Alltoallv, переслать в каждый процесс два числа из каждого набора; при этом в процесс 0 надо переслать последние два числа, в процесс 1 — два числа, предшествующих последнему, …, в последний процесс — первые два числа каждого набора. В каждом процессе вывести полученные числа.

2.3.2. Коллективные операции редукции

MPI3Coll19. В каждом процессе дан набор из $K + 5$ целых чисел, где K — количество процессов. Используя функцию MPI_Reduce для операции MPI_SUM, просуммировать элементы данных наборов с одним и тем же порядковым номером и вывести полученные суммы в главном процессе.

MPI3Coll20. В каждом процессе дан набор из $K + 5$ чисел, где K — количество процессов. Используя функцию MPI_Reduce для операции MPI_MIN, найти минимальное значение среди элементов данных наборов с одним и тем же порядковым номером и вывести полученные минимумы в главном процессе.

MPI3Coll21. В каждом процессе дан набор из $K + 5$ целых чисел, где K — количество процессов. Используя функцию MPI_Reduce для операции MPI_MAXLOC, найти максимальное значение среди элементов данных наборов с одним и тем же порядковым номером и ранг процесса, содержащего это максимальное значение. Вывести в главном процессе вначале все максимумы, а затем — ранги содержащих их процессов.

MPI3Coll22. В каждом процессе дан набор из $K + 5$ чисел, где K — количество процессов. Используя функцию MPI_Allreduce для операции MPI_PROD, перемножить элементы данных наборов с одним и тем же порядковым номером и вывести полученные произведения во всех процессах.

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.

Примечание. Решение данной задачи приведено в п. 1.2.5.

MPI3Coll24. В каждом процессе дан набор из K целых чисел, где K — количество процессов. Используя функцию MPI_Reduce_scatter, просуммировать элементы данных наборов с одним и тем же порядковым номером, переслать по одной из полученных сумм в каждый процесс (первую сумму — в процесс 0, вторую — в процесс 1, и т. д.) и вывести в каждом процессе полученную сумму.

MPI3Coll25. В каждом процессе дан набор из $2K$ чисел, где K — количество процессов. Используя функцию MPI_Reduce_scatter, найти максимумы среди элементов этих наборов с одним и тем же порядковым номером, переслать по два найденных максимума в каждый процесс (первые два максимума — в процесс 0, следующие два — в процесс 1, и т. д.) и вывести в каждом процессе полученные данные.

MPI3Coll26. В каждом процессе дан набор из $K(K + 3)/2$ целых чисел, где K — количество процессов. Используя функцию MPI_Reduce_scatter, найти минимальные значения среди элементов этих наборов с одним и тем же порядковым номером и переслать первые два минимума в процесс 0, следующие три — в процесс 1, …, последние $K + 1$ минимумов — в процесс $K - 1$. Вывести в каждом процессе полученные данные.

MPI3Coll27. В каждом процессе дан набор из $K + 5$ чисел, где K — количество процессов. Используя функцию MPI_Scan, найти в процессе ранга R ($R = 0, 1, \ldots, K - 1$) произведения элементов с одним и тем же порядковым номером для наборов, данных в процессах с рангами от 0 до R, и вывести найденные произведения (при этом в процессе $K - 1$ будут выведены произведения элементов из всех наборов).

MPI3Coll28. В каждом процессе дан набор из $K + 5$ целых чисел, где K — количество процессов. Используя функцию MPI_Scan, найти в процессе ранга R ($R = 0, \ldots, K - 1$) максимальные значения среди элементов

с одним и тем же порядковым номером для наборов, данных в процессах с рангами от 0 до R, и вывести в каждом процессе найденные максимумы.

2.4. Производные типы и упаковка данных

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи MPI4Type14, приведенным в п. 1.2.6.

2.4.1. Использование простейших производных типов

MPI4Type1. В главном процессе дана $K - 1$ тройка целых чисел, где K — количество процессов. Используя производный тип, содержащий три целых числа, и одну коллективную операцию пересылки данных, переслать все данные из главного процесса в подчиненные и вывести их в подчиненных процессах в том же порядке.

MPI4Type2. В главном процессе дана $K - 1$ тройка целых чисел, где K — количество процессов. Используя производный тип, содержащий три целых числа, и одну коллективную операцию пересылки данных, переслать по одной тройке чисел в каждый из подчиненных процессов и вывести их в подчиненных процессах в том же порядке.

MPI4Type3. В каждом подчиненном процессе дана тройка целых чисел. Используя производный тип, содержащий три целых числа, и одну коллективную операцию пересылки данных, переслать числа из подчиненных процессов в главный и вывести полученные числа в порядке возрастания рангов переславших их процессов.

MPI4Type4. В главном процессе дана $K - 1$ тройка чисел, где K — количество процессов, причем первые два числа каждой тройки являются целыми, а третье число — вещественным. Используя производный тип, содержащий три числа (два целых и одно вещественное), переслать числа из главного процесса в подчиненные и вывести их в подчиненных процессах в том же порядке.

MPI4Type5. В главном процессе дана $K - 1$ тройка чисел, где K — количество процессов, причем первое и третье число каждой тройки являются целыми, а второе число — вещественным. Используя производный тип, содержащий три числа (целое, вещественное, целое), переслать по одной тройке чисел в каждый из подчиненных процессов и вывести их в подчиненных процессах в том же порядке.

MPI4Type6. В каждом подчиненном процессе даны три числа: одно вещественное и два целых. Используя производный тип, содержащий три числа (одно вещественное и два целых), переслать числа из подчиненных процессов в главный и вывести полученные числа в порядке возрастания рангов переславших их процессов.

MPI4Type7. В каждом процессе даны три числа: первое и третье являются целыми, а второе — вещественным. Используя производный тип, содержащий три числа (целое, вещественное, целое), переслать данные из каждого процесса во все процессы и вывести в каждом процессе полученные числа в порядке возрастания рангов переславших их процессов (включая числа, полученные из этого же процесса).

MPI4Type8. В каждом подчиненном процессе даны R троек чисел, где R — ранг процесса. Два первых числа в каждой тройке являются целыми, а последнее — вещественным. Используя производный тип, содержащий три числа (два целых и одно вещественное), переслать числа из подчиненных процессов в главный и вывести полученные числа в порядке возрастания рангов переславших их процессов.

2.4.2. Пересылка упакованных данных

MPI4Type9. В главном процессе даны два набора: первый содержит K целых, а второй K вещественных чисел, где K — количество процессов. Используя функции упаковки MPI_Pack и MPI_Unpack и одну коллективную операцию пересылки данных, переслать все данные из главного процесса в подчиненные и вывести их в подчиненных процессах в том же порядке.

MPI4Type10. В главном процессе дана $K - 1$ тройка чисел, где K — количество процессов, причем первое и третье число каждой тройки является целым, а второе число — вещественным. Используя функции упаковки и одну коллективную операцию пересылки данных, переслать по одной тройке чисел в подчиненные процессы и вывести их в подчиненных процессах в том же порядке.

MPI4Type11. В главном процессе дана $K - 1$ тройка чисел, где K — количество процессов, причем первые два числа каждой тройки являются целыми, а третье число — вещественным. Используя функции упаковки и одну коллективную операцию пересылки данных, переслать все данные из главного процесса в подчиненные и вывести их в подчиненных процессах в том же порядке.

MPI4Type12. В каждом подчиненном процессе даны три числа: два целых и одно вещественное. Используя функции упаковки и одну коллективную операцию пересылки данных, переслать числа из подчиненных процессов в главный и вывести полученные числа в порядке возрастания рангов переславших их процессов.

MPI4Type13. В каждом подчиненном процессе дан набор из одного вещественного и R целых чисел, где значение R равно рангу процесса (в процессе 1 дано одно целое число, в процессе 2 — два целых числа, и т. д.). Используя функции упаковки и одну функцию передачи

и приема, переслать все данные в главный процесс и вывести эти данные в порядке возрастания рангов переславших их процессов.

2.4.3. Более сложные виды производных типов

MPI4Type14*°. В главном процессе даны два набора целых чисел: A размера $3K$ и N размера K, где K — количество подчиненных процессов. Считая, что элементы наборов нумеруются от 1, переслать и вывести в каждом подчиненном процессе ранга R ($R = 1, 2, \ldots, K$) N_R элементов из набора A, начиная с элемента A_R и перебирая их через один (например, если N_2 равно 3, то в процесс ранга 2 надо переслать элементы A_2, A_4, A_6). Использовать для пересылки каждого набора элементов по одному вызову функций MPI_Send, MPI_Probe и MPI_Recv, причем функция MPI_Recv должна возвращать массив, содержащий только те элементы, которые требуется вывести. Для этого в главном процессе определить новый тип, содержащий единственный целочисленный элемент и дополнительный конечный *пустой промежуток*, равный размеру элемента целого типа. Использовать в функции MPI_Send исходный массив A с требуемым смещением, указав в качестве второго параметра количество элементов, равное N_R, а в качестве третьего параметра — новый тип. В функции MPI_Recv использовать целочисленный массив размера N_R и тип MPI_INT. Для определения количества N_R переданных элементов использовать в подчиненных процессах функцию MPI_Get_count.

Указание. Для задания завершающего пустого промежутка при определении нового типа в MPI-2 следует использовать функцию MPI_Type_create_resized (в данном случае эту функцию надо применить к типу MPI_INT). В MPI-1 надо использовать *метку нулевого размера* типа MPI_UB совместно с функцией MPI_Type_struct (в стандарте MPI-2 тип MPI_UB объявлен устаревшим).

Примечание. Решение данной задачи приведено в п. 1.2.6.

MPI4Type15*. В главном процессе дана вещественная квадратная матрица порядка K, где K — количество подчиненных процессов (матрица должна храниться в одномерном массиве A, в котором элементы матрицы располагаются по строкам). Считая, что столбцы матрицы нумеруются от 1, переслать и вывести в каждом подчиненном процессе ранга R ($R = 1, 2, \ldots, K$) элементы столбца матрицы с номером R. Использовать по одному вызову функций MPI_Send и MPI_Recv, причем функция MPI_Recv должна возвращать массив, содержащий только те элементы, которые требуется вывести. Для этого в главном процессе определить новый тип, содержащий единственный вещественный элемент и дополнительный конечный пустой промежуток требуемого размера. Использовать в функции MPI_Send массив A с требуемым

смещением, указав в качестве второго параметра число K (т. е. количество элементов в каждом столбце), а в качестве третьего параметра — новый тип. В функции MPI_Recv использовать вещественный массив размера K и тип MPI_DOUBLE.

Указание. См. указание к задаче MPI4Type14.

MPI4Type16*. В каждом из подчиненных процессов дан столбец вещественной квадратной матрицы порядка K, где K — количество подчиненных процессов, причем в процессе ранга R ($R = 1, \ldots, K$) содержится столбец с номером R, если считать, что столбцы нумеруются от 1. Переслать все столбцы матрицы в главный процесс, разместив их в одномерном массиве A, в котором элементы матрицы хранятся по строкам, и вывести полученный массив A. Для пересылки каждого столбца использовать по одному вызову функций MPI_Send и MPI_Recv, причем функция MPI_Recv должна в качестве первого параметра содержать массив A (с требуемым смещением), а в качестве второго параметра — число 1. Для этого в главном процессе определить новый тип, содержащий K вещественных элементов, после каждого из которых следует пустой промежуток требуемого размера. Определение нового типа провести в два этапа: на первом создать вспомогательный тип, состоящий из одного вещественного числа и требуемого конечного пустого промежутка (см. указание к MPI4Type14), на втором — создать на его основе итоговый тип с помощью функции MPI_Type_contiguous, который использовать в качестве третьего параметра функции MPI_Recv (функцию MPI_Type_commit достаточно вызвать только для итогового типа). В функции MPI_Send использовать вещественный массив размера K и тип MPI_DOUBLE.

MPI4Type17*. Количество подчиненных процессов K кратно 3 и не превосходит 9. В каждом процессе дано целое число N; все числа N одинаковы и лежат в диапазоне от 3 до 5. Кроме того, в каждом подчиненном процессе дана целочисленная квадратная матрица порядка N (*блок*), которую следует прочесть в одномерный массив B по строкам. Переслать все массивы B в главный процесс и составить из них *блочную матрицу* размера $(K/3) \times 3$ (размер указан в блоках), располагая блоки по строкам (первая строка блоков должна содержать блоки, полученные из процессов ранга 1, 2, 3, вторая — из процессов ранга 4, 5, 6, и т. д.). Блочная матрица должна храниться по строкам в одномерном массиве A. Использовать для пересылки каждого блока B по одному вызову функций MPI_Send и MPI_Recv, причем функция MPI_Recv должна в качестве первого параметра содержать массив A (с требуемым смещением), а в качестве второго параметра — число 1. Для этого в главном процессе определить новый тип, содержащий N наборов элементов, причем каждый набор содержит N целых чисел, а между

наборами располагается пустой промежуток требуемого размера (при определении нового типа использовать функцию MPI_Type_vector). Указать созданный тип в качестве третьего параметра функции MPI_Recv. В функции MPI_Send использовать целочисленный массив *B* (размера $N \cdot N$) и тип MPI_INT.

MPI4Type18*. Количество подчиненных процессов *K* кратно 3 и не превосходит 9. В главном процессе дано целое число *N*, лежащее в диапазоне от 3 до 5, и целочисленная блочная матрица порядка $(K/3) \times 3$ (размер указан в блоках). Каждый блок представляет собой *нижнетреугольную* квадратную матрицу порядка *N*. Блочную матрицу следует прочесть в одномерный массив *A* по строкам. Переслать в каждый подчиненный процесс ненулевую часть соответствующего блока исходной матрицы, перебирая блоки по строкам (блоки из первой строки должны пересылаться в процессы ранга 1, 2, 3, блоки из второй строки — в процессы ранга 4, 5, 6, и т. д.), и вывести в каждом подчиненном процессе полученные данные (по строкам). Использовать для пересылки каждого блока по одному вызову функций MPI_Send, MPI_Probe и MPI_Recv, причем функция MPI_Send должна в качестве первого параметра содержать массив *A* (с требуемым смещением), а в качестве второго параметра — число 1. Для этого в главном процессе определить новый тип, содержащий *N* наборов элементов, причем каждый набор содержит ненулевую часть очередной строки блока (первый набор состоит из одного элемента, второй из двух, и т. д.), а между наборами располагается пустой промежуток требуемого размера (при определении нового типа использовать функцию MPI_Type_indexed). Указать созданный тип в качестве третьего параметра функции MPI_Send. В функции MPI_Recv использовать целочисленный массив *B* (содержащий ненулевую часть полученного блока) и тип MPI_INT. Для определения количества переданных элементов использовать в подчиненных процессах функцию MPI_Get_count.

MPI4Type19*. Количество подчиненных процессов *K* кратно 3 и не превосходит 9. В каждом процессе дано целое число *N*; все числа *N* одинаковы и лежат в диапазоне от 3 до 5. Кроме того, в каждом подчиненном процессе дано целое число *P* и ненулевые элементы целочисленной квадратной матрицы порядка *N* (*Z-блока*), которую следует прочесть в одномерный массив *B* по строкам. Ненулевые элементы располагаются в *Z*-блоке в форме символа «Z», т. е. занимают первую и последнюю строку, а также побочную диагональ. Определить в главном процессе нулевую целочисленную прямоугольную матрицу размера $N \cdot (K/3) \times 3N$, создав для этого одномерный массив *A* и обнулив его элементы (которые должны храниться в массиве *A* по строкам). Перебирая подчиненные процессы в порядке возрастания их

рангов, переслать из каждого подчиненного процесса данную в нем ненулевую часть *Z*-блока и записать полученный блок в массив *A*, начиная с элемента массива *A* с индексом *P* (позиции *Z*-блоков могут перекрываться; в этом случае элементы блоков, полученных из процессов большего ранга, будут заменять некоторые из элементов ранее записанных блоков). Использовать для пересылки ненулевой части каждого *Z*-блока по одному вызову функций MPI_Send и MPI_Recv, причем функция MPI_Recv должна в качестве первого параметра содержать массив *A* (с требуемым смещением), а в качестве второго параметра — число 1. Для этого в главном процессе определить новый тип, содержащий *N* наборов элементов, причем первый и последний набор содержат по *N* целых чисел, остальные наборы — по одному числу, а между наборами располагается пустой промежуток требуемого размера (при определении нового типа использовать функцию MPI_Type_indexed). Указать созданный тип в качестве третьего параметра функции MPI_Recv. В функции MPI_Send использовать целочисленный массив *B*, содержащий ненулевую часть *Z*-блока (по строкам), и тип MPI_INT.

Указание. Для передачи в главный процесс позиции *P* вставки *Z*-блока используйте параметр msgtag; для этого в подчиненных процессах задавайте значение *P* в качестве параметра msgtag функции MPI_Send, а в главном процессе до вызова функции MPI_Recv вызывайте функцию MPI_Probe с параметром MPI_ANY_TAG и анализируйте возвращенный ею параметр типа MPI_Status.

MPI4Type20*. Количество подчиненных процессов *K* кратно 3 и не превосходит 9. В каждом процессе дано целое число *N*; все числа *N* одинаковы и лежат в диапазоне от 3 до 5. Кроме того, в каждом подчиненном процессе дано целое число *P* и ненулевые элементы целочисленной квадратной матрицы порядка *N* (*U-блока*), которую следует прочесть в одномерный массив *B* по строкам. Ненулевые элементы располагаются в *U*-блоке в форме символа «U», т. е. занимают первый и последний столбец, а также последнюю строку. Определить в главном процессе нулевую целочисленную прямоугольную матрицу размера $N\cdot(K/3) \times 3N$, создав для этого одномерный массив *A* и обнулив его элементы (которые должны храниться в массиве *A* по строкам). Перебирая подчиненные процессы в порядке возрастания их рангов, переслать из каждого подчиненного процесса данную в нем ненулевую часть *U*-блока и записать полученный блок в массив *A*, начиная с элемента массива *A* с индексом *P* (позиции *U*-блоков могут перекрываться; в этом случае элементы блоков, полученных из процессов большего ранга, будут заменять некоторые из элементов ранее записанных блоков). Использовать для пересылки ненулевой части каждо-

го U-блока по одному вызову функций MPI_Send и MPI_Recv, причем функция MPI_Recv должна в качестве первого параметра содержать массив A (с требуемым смещением), а в качестве второго параметра — число 1. Для этого в главном процессе определить новый тип, содержащий требуемое количество наборов элементов с пустыми промежутками между ними (при определении нового типа использовать функцию MPI_Type_indexed). Указать созданный тип в качестве третьего параметра функции MPI_Recv. В функции MPI_Send использовать целочисленный массив B, содержащий ненулевую часть U-блока (по строкам), и тип MPI_INT.

Указание. См. указание к задаче MPI4Type19.

2.4.4. Коллективная функция MPI_Alltoallw (MPI-2)

MPI4Type21**. Решить задачу MPI4Type15, используя для пересылки данных вместо функций MPI_Send и MPI_Recv одну коллективную операцию.

Указание. Использовать функции группы Scatter не удастся, так как значения смещений для рассылаемых элементов (столбцов матрицы) требуется указывать не в элементах, а в *байтах*. Поэтому подходящим вариантом будет введенная в MPI-2 функция MPI_Alltoallw, позволяющая наиболее гибким образом настраивать вариант коллективной пересылки. В данном случае с ее помощью следует реализовать вариант рассылки вида Scatter (при этом большинство параметров-массивов, используемых в этой функции, необходимо по-разному определять в главном и подчиненных процессах).

MPI4Type22**. Решить задачу MPI4Type16, используя для пересылки данных вместо функций MPI_Send и MPI_Recv одну коллективную операцию.

Указание. См. указание к задаче MPI4Type21. В данном случае с помощью функции MPI_Alltoallw следует реализовать вариант рассылки вида Gather.

2.5. Группы процессов и коммуникаторы

Перед выполнением заданий из данного пункта следует ознакомиться с примерами решения задач MPI5Comm3, MPI5Comm17, MPI5Comm29, приведенными в п. 1.2.7–1.2.9.

2.5.1. Создание новых коммуникаторов

MPI5Comm1. В главном процессе дан набор из K целых чисел, где K — количество процессов четного ранга (0, 2, …). С помощью функций MPI_Comm_group, MPI_Group_incl и MPI_Comm_create создать новый коммуникатор, включающий процессы четного ранга. Используя одну кол-

лективную операцию пересылки данных для созданного коммуникатора, переслать по одному исходному числу в каждый процесс четного ранга (включая главный) и вывести полученные числа.

MPI5Comm2. В каждом процессе нечетного ранга (1, 3, …) даны два вещественных числа. С помощью функций MPI_Comm_group, MPI_Group_excl и MPI_Comm_create создать новый коммуникатор, включающий процессы нечетного ранга. Используя одну коллективную операцию пересылки данных для созданного коммуникатора, переслать исходные числа во все процессы нечетного ранга и вывести эти числа в порядке возрастания рангов переславших их процессов (включая числа, полученные из этого же процесса).

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.

Примечание. Решение данной задачи приведено в п. 1.2.7.

MPI5Comm4. В каждом процессе четного ранга (включая главный процесс) дан набор из трех элементов — вещественных чисел. Используя новый коммуникатор и одну коллективную операцию редукции, найти минимальные значения среди элементов исходных наборов с одним и тем же порядковым номером и вывести найденные минимумы в главном процессе. Новый коммуникатор создать с помощью функции MPI_Comm_split.

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm5. В каждом процессе дано вещественное число. Используя функцию MPI_Comm_split и одну коллективную операцию редукции, найти максимальное из чисел, данных в процессах с четным рангом (включая главный процесс), и минимальное из чисел, данных в процессах с нечетным рангом. Найденный максимум вывести в процессе 0, а найденный минимум — в процессе 1.

Указание. Программа должна содержать единственный вызов функции MPI_Comm_split, создающий оба требуемых коммуникатора (каждый в соответствующей группе процессов).

MPI5Comm6. В главном процессе дано целое число K и набор из K вещественных чисел, в каждом подчиненном процессе дано целое число N, которое может принимать два значения: 0 и 1 (количество подчиненных процессов с $N = 1$ равно K). Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать по одному вещественному числу из главного процесса в каждый подчиненный процесс с $N = 1$ и вывести в этих подчиненных процессах полученные числа.

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm7. В каждом процессе дано целое число N, которое может принимать два значения: 0 и 1 (имеется хотя бы один процесс с $N = 1$). Кроме того, в каждом процессе с $N = 1$ дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать числа A в первый из процессов с $N = 1$ и вывести их в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm8. В каждом процессе дано целое число N, которое может принимать два значения: 0 и 1 (имеется хотя бы один процесс с $N = 1$). Кроме того, в каждом процессе с $N = 1$ дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать числа A в последний из процессов с $N = 1$ и вывести их в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm9. В каждом процессе дано целое число N, которое может принимать два значения: 0 и 1 (имеется хотя бы один процесс с $N = 1$). Кроме того, в каждом процессе с $N = 1$ дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать числа A во все процессы с $N = 1$ и вывести их в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm10. В каждом процессе дано целое число N, которое может принимать два значения: 1 и 2 (имеется хотя бы один процесс с каждым из возможных значений). Кроме того, в каждом процессе дано целое число A. Используя функцию MPI_Comm_split и одну коллективную операцию пересылки данных, переслать числа A, данные в процессах с $N = 1$, во все процессы с $N = 1$, а числа A, данные в процессах с $N = 2$, во все процессы с $N = 2$. Во всех процессах вывести получен-

ные числа в порядке возрастания рангов переславших их процессов (включая число, полученное из этого же процесса).

Указание. См. указание к задаче MPI5Comm5.

MPI5Comm11. В каждом процессе дано целое число N, которое может принимать два значения: 0 и 1 (имеется хотя бы один процесс с $N = 1$). Кроме того, в каждом процессе с $N = 1$ дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию редукции, найти сумму всех исходных чисел A и вывести ее во всех процессах с $N = 1$.

Указание. См. указание к задаче MPI5Comm3.

MPI5Comm12. В каждом процессе дано целое число N, которое может принимать два значения: 1 и 2 (имеется хотя бы один процесс с каждым из возможных значений). Кроме того, в каждом процессе дано вещественное число A. Используя функцию MPI_Comm_split и одну коллективную операцию редукции, найти минимальное значение среди чисел A, которые даны в процессах с $N = 1$, и максимальное значение среди чисел A, которые даны в процессах с $N = 2$. Найденный минимум вывести в процессах с $N = 1$, а найденный максимум — в процессах с $N = 2$.

Указание. См. указание к задаче MPI5Comm5.

2.5.2. Виртуальные топологии

MPI5Comm13. В главном процессе дано целое число N (> 1), причем известно, что количество процессов K делится на N. Переслать число N во все процессы, после чего, используя функцию MPI_Cart_create, определить для всех процессов декартову топологию в виде двумерной решетки — матрицы размера $N \times K/N$ (порядок нумерации процессов оставить прежним). Используя функцию MPI_Cart_coords, вывести для каждого процесса его координаты в созданной топологии.

MPI5Comm14. В главном процессе дано целое число N (> 1), не превосходящее количества процессов K. Переслать число N во все процессы, после чего определить декартову топологию для начальной части процессов в виде двумерной решетки — матрицы размера $N \times K/N$ (символ «/» обозначает операцию деления нацело, порядок нумерации процессов следует оставить прежним). Для каждого процесса, включенного в декартову топологию, вывести его координаты в этой топологии.

MPI5Comm15. Число процессов K является четным: $K = 2N$, $N > 1$. В процессах 0 и N дано по одному вещественному числу A. Определить для всех процессов декартову топологию в виде матрицы размера $2 \times N$, после чего, используя функцию MPI_Cart_sub, расщепить матрицу про-

цессов на две одномерные строки (при этом процессы 0 и N будут главными процессами в полученных строках). Используя одну коллективную операцию пересылки данных, переслать число A из главного процесса каждой строки во все процессы этой же строки и вывести полученное число в каждом процессе (включая процессы 0 и N).

MPI5Comm16. Число процессов K является четным: $K = 2N$, $N > 1$. В процессах 0 и 1 дано по одному вещественному числу A. Определить для всех процессов декартову топологию в виде матрицы размера $N \times 2$, после чего, используя функцию MPI_Cart_sub, расщепить матрицу процессов на два одномерных столбца (при этом процессы 0 и 1 будут главными процессами в полученных столбцах). Используя одну коллективную операцию пересылки данных, переслать число A из главного процесса каждого столбца во все процессы этого же столбца и вывести полученное число в каждом процессе (включая процессы 0 и 1).

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 $2N$).

Примечание. Решение данной задачи приведено в п. 1.2.8.

MPI5Comm18. Число процессов K кратно трем: $K = 3N$, $N > 1$. В процессах 0, 1 и 2 дано по N целых чисел. Определить для всех процессов декартову топологию в виде матрицы размера $N \times 3$, после чего, используя функцию MPI_Cart_sub, расщепить матрицу процессов на три одномерных столбца (при этом процессы 0, 1 и 2 будут главными процессами в полученных столбцах). Используя одну коллективную операцию пересылки данных, переслать по одному исходному числу из главного процесса каждого столбца во все процессы этого же столбца и вывести полученное число в каждом процессе (включая процессы 0, 1 и 2).

MPI5Comm19. Количество процессов K равно 8 или 12, в каждом процессе дано целое число. Определить для всех процессов декартову топологию в виде трехмерной решетки размера $2 \times 2 \times K/4$ (порядок нумерации процессов оставить прежним). Интерпретируя эту решетку как две матрицы размера $2 \times K/4$ (в одну матрицу входят процессы с оди-

наковой первой координатой), расщепить каждую матрицу процессов на две одномерные строки. Используя одну коллективную операцию пересылки данных, переслать в главный процесс каждой строки исходные числа из процессов этой же строки и вывести полученные наборы чисел (включая числа, полученные из главных процессов строк).

MPI5Comm20. Количество процессов K равно 8 или 12, в каждом процессе дано целое число. Определить для всех процессов декартову топологию в виде трехмерной решетки размера $2 \times 2 \times K/4$ (порядок нумерации процессов оставить прежним). Интерпретируя полученную решетку как $K/4$ матриц размера 2×2 (в одну матрицу входят процессы с одинаковой третьей координатой), расщепить эту решетку на $K/4$ указанных матриц. Используя одну коллективную операцию пересылки данных, переслать в главный процесс каждой из полученных матриц исходные числа из процессов этой же матрицы и вывести полученные наборы чисел (включая числа, полученные из главных процессов матриц).

MPI5Comm21. Количество процессов K равно 8 или 12, в каждом процессе дано вещественное число. Определить для всех процессов декартову топологию в виде трехмерной решетки размера $2 \times 2 \times K/4$ (порядок нумерации процессов оставить прежним). Интерпретируя эту решетку как две матрицы размера $2 \times K/4$ (в одну матрицу входят процессы с одинаковой первой координатой), расщепить каждую матрицу процессов на $K/4$ одномерных столбцов. Используя одну коллективную операцию редукции, для каждого столбца процессов найти произведение исходных чисел и вывести найденные произведения в главных процессах каждого столбца.

MPI5Comm22. Количество процессов K равно 8 или 12, в каждом процессе дано вещественное число. Определить для всех процессов декартову топологию в виде трехмерной решетки размера $2 \times 2 \times K/4$ (порядок нумерации процессов оставить прежним). Интерпретируя полученную решетку как $K/4$ матриц размера 2×2 (в одну матрицу входят процессы с одинаковой третьей координатой), расщепить эту решетку на $K/4$ указанных матриц. Используя одну коллективную операцию редукции, для каждой из полученных матриц найти сумму исходных чисел и вывести найденные суммы в каждом процессе соответствующей матрицы.

MPI5Comm23. В главном процессе даны положительные целые числа M и N, произведение которых не превосходит количества процессов; кроме того, в процессах с рангами от 0 до $M \cdot N - 1$ даны целые числа X и Y. Переслать числа M и N во все процессы, после чего определить для начальных $M \cdot N$ процессов декартову топологию в виде двумерной

решетки размера $M \times N$, являющейся периодической по первому измерению (порядок нумерации процессов оставить прежним). В каждом процессе, входящем в созданную топологию, вывести ранг процесса с координатами X, Y (с учетом периодичности), используя для этого функцию MPI_Cart_rank. В случае недопустимых координат вывести −1.

Примечание. Если при вызове функции MPI_Cart_rank указаны недопустимые координаты (например, отрицательные координаты для измерений, по которым декартова решетка не является периодической), то сама функция возвращает ненулевое значение (являющееся признаком ошибки), а возвращаемое значение параметра rank является неопределенным. Таким образом, в данном задании число −1 следует выводить, если функция MPI_Cart_rank вернула ненулевое значение. Чтобы подавить возникающие при этом сообщения об ошибках, отображаемые в разделе отладки окна задачника, достаточно перед вызовом функции, который может привести к ошибке, установить с помощью функции MPI_Comm_set_errhandler (MPI_Errhandler_set в MPI-1) специальный вариант *обработчика ошибок* MPI_ERROR_RETURN, который при возникновении ошибки не выполняет никаких действий, кроме установки ненулевого возвращаемого значения для этой функции. Следует заметить, что в версии MPICH 1.2.5 в случае указания недопустимых координат функция MPI_Cart_rank возвращает значение параметра rank, равное −1, что позволяет упростить решение, избавившись от проверки возвращаемого значения функции. Однако и в этом случае желательно подавить вывод сообщений об ошибках описанным выше способом.

MPI5Comm24. В главном процессе даны положительные целые числа M и N, произведение которых не превосходит количества процессов; кроме того, в процессах с рангами от 0 до $M \cdot N - 1$ даны целые числа X и Y. Переслать числа M и N во все процессы, после чего определить для начальных $M \cdot N$ процессов декартову топологию в виде двумерной решетки размера $M \times N$, являющейся периодической по второму измерению (порядок нумерации процессов оставить прежним). В каждом процессе, входящем в созданную топологию, вывести ранг процесса с координатами X, Y (с учетом периодичности), используя для этого функцию MPI_Cart_rank. В случае недопустимых координат вывести −1.

Примечание. См. примечание к заданию MPI5Comm23.

MPI5Comm25. В каждом подчиненном процессе дано вещественное число. Определить для всех процессов декартову топологию в виде одномерной решетки и осуществить простой сдвиг исходных данных с шагом −1 (число из каждого *подчиненного* процесса пересылается в пре-

дыдущий процесс). Для определения рангов посылающих и принимающих процессов использовать функцию MPI_Cart_shift, пересылку выполнять с помощью функций MPI_Send и MPI_Recv. Во всех процессах, получивших данные, вывести эти данные.

MPI5Comm26. Число процессов K является четным: $K = 2N$, $N > 1$; в каждом процессе дано вещественное число A. Определить для всех процессов декартову топологию в виде матрицы размера $2 \times N$ (порядок нумерации процессов оставить прежним) и для каждой строки матрицы осуществить циклический сдвиг исходных данных с шагом 1 (число A из каждого процесса, кроме последнего в строке, пересылается в следующий процесс этой же строки, а из последнего процесса — в главный процесс этой строки). Для определения рангов посылающих и принимающих процессов использовать функцию MPI_Cart_shift, пересылку выполнять с помощью функции MPI_Sendrecv. Во всех процессах вывести полученные данные.

MPI5Comm27. Количество процессов K равно 8 или 12, в каждом процессе дано вещественное число. Определить для всех процессов декартову топологию в виде трехмерной решетки размера $2 \times 2 \times K/4$ (порядок нумерации процессов оставить прежним). Интерпретируя полученную решетку как $K/4$ матриц размера 2×2 (в одну матрицу входят процессы с одинаковой третьей координатой, матрицы упорядочены по возрастанию третьей координаты), осуществить циклический сдвиг исходных данных из процессов каждой матрицы в соответствующие процессы следующей матрицы (из процессов последней матрицы данные перемещаются в первую матрицу). Для определения рангов посылающих и принимающих процессов использовать функцию MPI_Cart_shift, пересылку выполнять с помощью функции MPI_Sendrecv_replace. Во всех процессах вывести полученные данные.

MPI5Comm28**. Число процессов K является нечетным: $K = 2N + 1$ ($1 < N < 5$); в каждом процессе дано целое число A. Используя функцию MPI_Graph_create, определить для всех процессов топологию графа, в которой главный процесс связан ребрами со всеми процессами нечетного ранга (1, 3, …, $2N - 1$), а каждый процесс четного положительного ранга R (2, 4, …, $2N$) связан ребром с процессом ранга $R - 1$ (в результате получается N-лучевая звезда, центром которой является главный процесс, а каждый луч состоит из двух подчиненных процессов R и $R + 1$, причем ближайшим к центру является процесс нечетного ранга R — рис. 54).

Переслать число A из каждого процесса всем процессам, связанным с ним ребрами (*процессам-соседям*). Для определения количества процессов-соседей и их рангов использовать функции

MPI_Graph_neighbors_count и MPI_Graph_neighbors, пересылку выполнять с помощью функций MPI_Send и MPI_Recv. Во всех процессах вывести полученные числа в порядке возрастания рангов переславших их процессов.

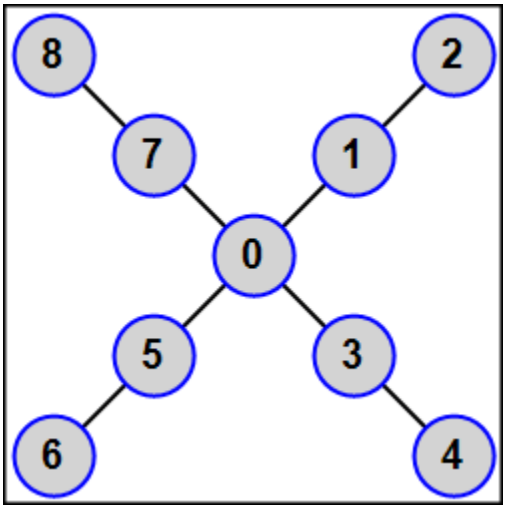

Рис. 54. Пример топологии графа из задания MPI5Comm28

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 — рис. 55).

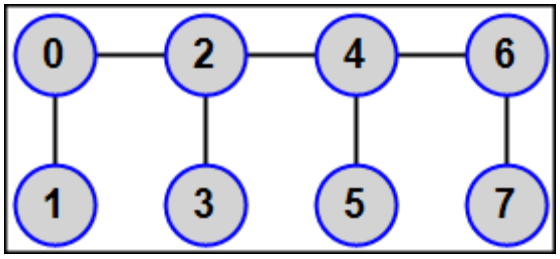

Рис. 55. Пример топологии графа из задания MPI5Comm29

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.

Примечание. Решение данной задачи приведено в п. 1.2.9.

MPI5Comm30**. Количество процессов K равно $3N + 1$ ($1 < N < 5$); в каждом процессе дано целое число A. Используя функцию MPI_Graph_create, определить для всех процессов топологию графа, в которой процессы R, $R + 1$, $R + 2$, где $R = 1, 4, 7, \ldots$, связаны между собой ребрами, и, кроме того, каждый процесс положительного ранга, кратного трем (3, 6, …), связан ребром с главным процессом (в результате получается N-лучевая звезда, центром которой является главный процесс, а каждый луч состоит из трех связанных между собой процессов, причем с центром связан процесс ранга, кратного трем — рис. 56).

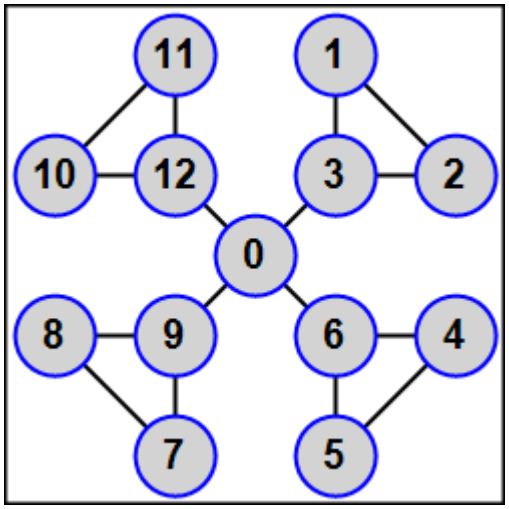

Рис. 56. Пример топологии графа из задания MPI5Comm30

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.

2.5.3. Топология распределенного графа (MPI-2)

MPI5Comm31**. Количество процессов K кратно 3; в каждом процессе дано целое число A. Используя функцию MPI_Dist_graph_create, определить для всех процессов топологию распределенного графа, в которой все процессы рангов, кратных 3 (0, 3, …, $K - 3$) образуют кольцо, причем каждый из процессов, входящих в кольцо, является источником для последующего процесса (процесс 0 является источником для процесса 3, процесс 3 — для процесса 6, …, процесс $K - 3$ — для процесса 0) и, кроме того, процесс ранга $3N$ ($N = 0, 1, \ldots, K/3 - 1$) является источником для процессов ранга $3N + 1$ и $3N + 2$, а процесс ранга $3N + 1$ является источником для процесса ранга $3N + 2$ (рис. 57).

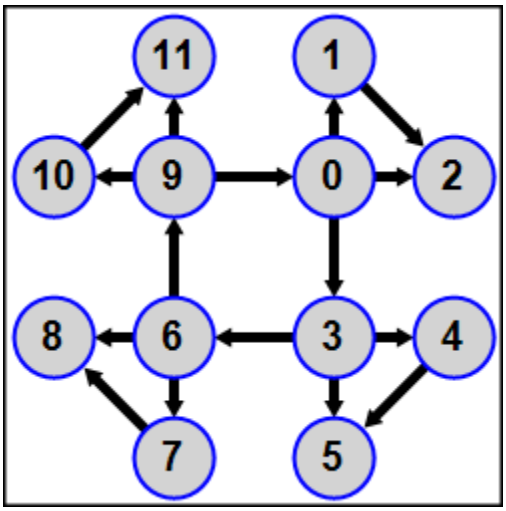

Рис. 57. Пример топологии графа из задания MPI5Comm31

Полное определение топологии дать в главном процессе (в подчиненных процессах второй параметр функции MPI_Dist_graph_create должен быть равен 0); параметр weights положить равным MPI_UNWEIGHTED, параметр info — равным MPI_INFO_NULL, порядок нумерации процессов не изменять. Переслать числа *A* от процессов-источников процессам-приемникам и вывести в каждом процессе сумму исходного числа *A* и всех чисел, полученных от процессов-источников. Для определения количества источников и приемников, а также их рангов использовать функции MPI_Dist_graph_neighbors_count и MPI_Dist_graph_neighbors, пересылку выполнять с помощью функций MPI_Send и MPI_Recv.

MPI5Comm32**. Количество процессов лежит в диапазоне от 4 до 15, в каждом процессе дано целое число *A*. Используя функцию MPI_Dist_graph_create, определить для всех процессов топологию распределенного графа, которая представляет собой бинарное дерево с корнем в главном процессе 0, вершинами первого уровня в процессах рангов 1 и 2, вершинами второго уровня в процессах рангов 3-6 (причем процессы ранга 3 и 4 являются дочерними вершинами процесса 1, а процессы ранга 5 и 6 — дочерними вершинами процесса 2), и т. д. Каждый процесс является источником для своих дочерних вершин; таким образом, каждый процесс имеет от 0 до 2 процессов-приемников (рис. 58).

Полное определение топологии дать в главном процессе (в подчиненных процессах второй параметр функции MPI_Dist_graph_create должен быть равен 0); параметр weights положить равным MPI_UNWEIGHTED, параметр info — равным MPI_INFO_NULL, порядок нумерации процессов не изменять. Найти и вывести в каждом процессе сумму исходного числа *A* и чисел, данных в процессах-предках *всех уровней* — от корня (главного процесса) до ближайшего предка (родительского процесса-

источника). Для определения количества источников и приемников, а также их рангов использовать функции MPI_Dist_graph_neighbors_count и MPI_Dist_graph_neighbors, пересылку данных выполнять с помощью функций MPI_Send и MPI_Recv.

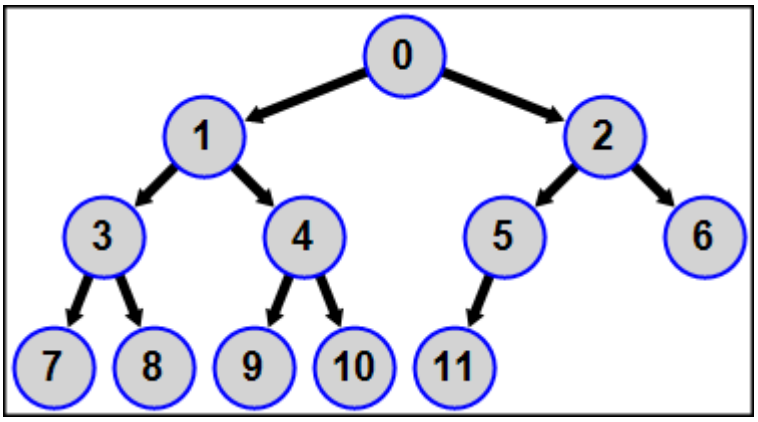

Рис. 58. Пример топологии графа из задания MPI5Comm32

2.6. Параллельный ввод-вывод файловых данных (MPI-2)

Для хранения имени файла достаточно использовать массив char[12], а для его пересылки из главного процесса в подчиненные — функцию MPI_Bcast с параметром типа MPI_CHAR.

В первых двух подгруппах (задания MPI6File1–MPI6File16) не требуется настраивать вид файловых данных с помощью функции MPI_File_set_view; достаточно использовать вид по умолчанию, при котором базовые и файловые элементы имеют тип MPI_BYTE, начальное смещение для всех процессов равно 0 и используется представление «native». Это же представление надо указывать и при явной настройке вида файловых данных в заданиях третьей подгруппы (MPI6File17–MPI6File30).

Для определения размера типов MPI_INT и MPI_DOUBLE следует применять функцию MPI_Type_size.

Для указания завершающего пустого промежутка (в заданиях, связанных с настройкой вида файловых данных) следует использовать функцию MPI_Type_create_resized. Альтернативным вариантом является использование специального типа MPI_UB (*метки нулевого размера*), однако в стандарте MPI-2 этот тип объявлен устаревшим.

Перед выполнением заданий из данного пункта следует ознакомиться с примером решения задачи MPI6File26, приведенным в п. 1.3.3.

2.6.1. Локальный файловый ввод-вывод

MPI6File1. В главном процессе дано имя существующего файла целых чисел. В каждом из подчиненных процессов дано количество файловых элементов, предназначенных для считывания в этом процессе, и порядковые номера этих элементов (элементы нумеруются от 1). Эле-

менты с некоторыми номерами в исходном файле могут отсутствовать. Используя в каждом процессе требуемое количество вызовов локальной функции MPI_File_read_at, прочесть из файла существующие элементы с указанными номерами и вывести их в том же порядке. Для проверки наличия элемента с указанным порядковым номером можно либо использовать функцию MPI_File_get_size, либо анализировать информацию, возвращенную в параметре типа MPI_Status функции MPI_File_read_at.

MPI6File2. В главном процессе дано имя файла. В каждом из подчиненных процессов дано количество пар целых чисел и сами пары, в которых первый член равен порядковому номеру файлового элемента, а второй — значению этого файлового элемента (элементы нумеруются от 1, все номера различны и принимают все значения от 1 до некоторого целого числа). Создать новый файл целых чисел с указанным именем и записать в него данные элементы, используя в каждом подчиненном процессе требуемое количество вызовов локальной функции MPI_File_write_at.

MPI6File3. В главном процессе дано имя существующего файла вещественных чисел, содержащего элементы прямоугольной матрицы размера $K \times N$, где K — число подчиненных процессов. В каждом подчиненном процессе прочесть и вывести элементы R-й строки матрицы, где R — ранг подчиненного процесса (строки нумеруются от 1), используя один вызов локальной функции MPI_File_read_at. Для определения размера файла использовать функцию MPI_File_get_size.

MPI6File4. В главном процессе дано имя файла, в каждом из подчиненных процессов дано R вещественных чисел, где R — ранг процесса. Создать новый файл вещественных чисел с указанным именем и записать в него данные элементы в порядке возрастания рангов содержащих их процессов. Использовать в каждом подчиненном процессе один вызов локальной функции MPI_File_write_at.

MPI6File5. В главном процессе дано имя существующего файла целых чисел. Файл обязательно содержит все целые числа в диапазоне от 1 до K, где K — максимальный ранг процесса. В каждом подчиненном процессе прочесть и вывести два фрагмента файловых данных: первый содержит *начальную* часть элементов до первого элемента, равного рангу процесса (включая этот элемент), второй содержит *завершающую* часть файловых элементов и имеет тот же размер, что и первый фрагмент. Элементы в каждом фрагменте выводятся в порядке их следования в файле. Использовать в каждом подчиненном процессе необходимое количество вызовов локальной функции MPI_File_read (без применения массивов), а также вызовы функций MPI_File_get_position для

определения размера первого фрагмента и MPI_File_seek с параметром MPI_SEEK_END для перемещения к началу второго фрагмента.

MPI6File6. В главном процессе дано имя файла, в каждом из подчиненных процессов дано целое число. Создать новый файл целых чисел с указанным именем и записать в него K подряд идущих копий каждого числа, где K равно количеству подчиненных процессов. Порядок следования чисел в файле должен быть обратным порядку подчиненных процессов (K копий числа из процесса 1 должны находиться в конце файла, перед ними должны располагаться K копий числа из процесса 2, и т. д.). Использовать в каждом подчиненном процессе один вызов локальной функции MPI_File_write, а также функцию MPI_File_seek с параметром MPI_SEEK_SET.

MPI6File7. В главном процессе дано имя существующего файла вещественных чисел. Известно, что сумма значений всех файловых элементов превосходит K, где K — количество подчиненных процессов. В каждом подчиненном процессе считывать начальные элементы файла, пока сумма их значений не превысит ранг процесса, после чего вывести найденную сумму S и количество N прочитанных чисел. После этого дополнительно прочесть и вывести значения N *последних* файловых элементов (в порядке их следования в файле). Использовать в каждом подчиненном процессе необходимое количество вызовов локальной функции MPI_File_read (без применения массивов), а также вызовы функций MPI_File_get_position для определения N и MPI_File_seek с параметром MPI_SEEK_END для перемещения к началу фрагмента из N последних файловых элементов.

MPI6File8. В главном процессе дано имя файла, в каждом из подчиненных процессов дано вещественное число. Создать новый файл вещественных чисел с указанным именем и записать в него R подряд идущих копий каждого числа, где R равно рангу процесса, в котором дано это число. Порядок следования чисел в файле должен быть *обратным* порядку подчиненных процессов (в конце файла должно находиться единственное число из процесса 1, перед ним должны располагаться две копии числа из процесса 2, и т. д.). Использовать в каждом подчиненном процессе один вызов локальной функции MPI_File_write, а также функцию MPI_File_seek с параметром MPI_SEEK_SET.

2.6.2. Коллективный файловый ввод-вывод

MPI6File9. В главном процессе дано имя существующего файла целых чисел. В каждом процессе прочесть и вывести $R + 1$ элемент файла, начиная с элемента с номером $R + 1$, где число R равно рангу процесса (0, 1, …). Файловые элементы нумеруются от 1; таким образом, в про-

цессе ранга 0 требуется прочесть и вывести только начальный файловый элемент, в процессе ранга 1 — два следующих файловых элемента, в процессе ранга 2 — три элемента, начиная с третьего, и т. д. Если файл содержит недостаточное количество элементов, то в некоторых процессах число выведенных элементов может быть меньше требуемого. Использовать один вызов коллективной функции MPI_File_read_all, а также функцию MPI_File_seek с параметром MPI_SEEK_SET и функцию MPI_File_get_size для определения размера исходного файла.

Примечание. В реализации MPICH2 версии 1.3 функция MPI_File_read_all не позволяет определить количество фактически считанных файловых элементов на основе информации, содержащейся в параметре типа MPI_Status: в этом параметре количество прочитанных элементов всегда равно требуемому, а при недостаточном количестве файловых элементов в завершающую часть выходного массива записываются нулевые значения.

MPI6File10. В главном процессе дано имя файла, в каждом из подчиненных процессов дан набор из R целых чисел, где R — ранг процесса (1, 2, …). Создать новый файл целых чисел с указанным именем и записать в него все данные числа в порядке их следования (процессы перебираются в порядке возрастания их рангов). Использовать один вызов коллективной функции MPI_File_write_all (для всех процессов, в том числе и для процесса ранга 0), а также функцию MPI_File_seek с параметром MPI_SEEK_SET.

MPI6File11*. В главном процессе дано имя существующего файла вещественных чисел. Кроме того, в каждом процессе дано целое число, равное либо 0, либо порядковому номеру одного из существующих файловых элементов (элементы нумеруются от 1). Используя функцию MPI_Comm_split, создать новый коммуникатор, содержащий только те процессы, в которых дано число, отличное от 0, и с помощью коллективной функции MPI_File_read_at_all прочесть и вывести в каждом процессе из этого коммуникатора элемент, расположенный в позиции с указанным порядковым номером.

MPI6File12*. В главном процессе дано имя существующего файла вещественных чисел. Кроме того, в каждом процессе дано целое число, равное либо 0, либо порядковому номеру одного из существующих файловых элементов (элементы нумеруются от 1). Используя функцию MPI_Comm_split, создать новый коммуникатор, содержащий только те процессы, в которых дано число, отличное от 0, и с помощью коллективной функции MPI_File_write_at_all заменить исходное значение файлового элемента в позиции с указанным порядковым номером на значе-

ние ранга процесса в новом коммуникаторе (преобразовав ранг в вещественное число).

MPI6File13. В главном процессе дано имя существующего файла целых чисел. Кроме того, в каждом процессе дано целое число, равное либо 0, либо 1. Используя функцию MPI_Comm_split, создать новый коммуникатор, содержащий только те процессы, в которых дано число 1, и с помощью коллективной функции MPI_File_read_ordered прочесть и вывести в каждом процессе из этого коммуникатора $R + 1$ элемент исходного файла, где R — ранг процесса в новом коммуникаторе (элементы считываются последовательно: один элемент в процессе 0, два следующих элемента — в процессе 1, три следующих — в процессе 2, и т. д.). Если файл содержит недостаточное количество элементов, то в некоторых процессах число выведенных элементов N может быть меньше требуемого или даже равно нулю. В каждом процессе, входящем в новый коммуникатор, дополнительно вывести количество N фактически прочитанных элементов и новое значение P коллективного файлового указателя. Для определения количества прочитанных элементов N использовать информацию, возвращенную в параметре типа MPI_Status функции MPI_File_read_ordered, для нахождения значения P использовать функцию MPI_File_get_position_shared (во всех процессах значение P должно быть одинаковым).

MPI6File14. В главном процессе дано имя файла. Кроме того, в каждом процессе дано целое число N. Создать новый файл целых чисел с указанным именем. Используя функцию MPI_Comm_split, создать новый коммуникатор, содержащий только те процессы, в которых число N не равно 0, и с помощью коллективной функции MPI_File_write_ordered записать в файл K подряд идущих копий каждого из чисел N, где K равно количеству процессов в *новом* коммуникаторе, а числа располагаются в порядке возрастания рангов содержащих их процессов.

MPI6File15*. В главном процессе дано имя существующего файла целых чисел, содержащего не менее K элементов, где K — количество процессов. С помощью функции MPI_Comm_split создать новый коммуникатор, содержащий только процессы с нечетным рангом (1, 3, …). Используя по одному вызову коллективных функций MPI_File_seek_shared и MPI_File_read_ordered, прочесть и вывести в каждом процессе из нового коммуникатора по 2 элемента исходного файла, причем предпоследний и последний файловые элементы должны считываться и выводиться (в указанном порядке) в процессе с рангом 1 в исходном коммуникаторе MPI_COMM_WORLD, четвертый и третий с конца элементы — в процессе с рангом 3, и т. д.

Указание. Для того чтобы обеспечить требуемый порядок считывания данных в функции MPI_File_read_ordered, необходимо изменить в созданном коммуникаторе порядок следования процессов на противоположный (по сравнению с исходным порядком в коммуникаторе MPI_COMM_WORLD).

MPI6File16*. В главном процессе дано имя файла. Кроме того, в каждом процессе дано целое число N (≥ 0) и N вещественных чисел. Создать новый файл целых чисел с указанным именем. Используя функцию MPI_Comm_split, создать новый коммуникатор, содержащий только те процессы, в которых число N не равно 0, и с помощью единственного вызова коллективной функции MPI_File_write_ordered записать в файл все вещественные числа в порядке, *обратном* их следованию в наборе исходных данных: вначале записываются (в обратном порядке) все числа из процесса с наибольшим рангом в коммуникаторе MPI_COMM_WORLD, затем все числа из процесса с предыдущим рангом, и т. д.

Указание. Для того чтобы обеспечить требуемый порядок записи данных в функции MPI_File_write_ordered, необходимо изменить в созданном коммуникаторе порядок следования процессов на противоположный (по сравнению с исходным порядком в коммуникаторе MPI_COMM_WORLD).

2.6.3. Настройка вида файловых данных

MPI6File17. В главном процессе дано имя существующего файла целых чисел, содержащего $2K$ элементов, где K — количество процессов. Используя единственный вызов коллективной функции MPI_File_read_all (и не применяя функцию MPI_File_seek), прочесть и вывести по 2 элемента в каждом процессе, перебирая элементы в порядке их следования в файле. Для этого предварительно с помощью функции MPI_File_set_view определить новый вид данных с базовым типом MPI_INT, таким же файловым типом и подходящим смещением (своим для каждого процесса).

MPI6File18. В главном процессе дано имя существующего файла целых чисел, содержащего элементы прямоугольной матрицы размера $K \times 5$, где K — количество процессов. Кроме того, в каждом процессе дано целое число N ($1 \leq N \leq 5$) — порядковый номер *выделенного* элемента в некоторой строке матрицы (элементы в строке нумеруются от 1), причем с процессом ранга 0 связывается первая строка матрицы, с процессом ранга 1 — вторая строка, и т. д. Используя единственный вызов коллективной функции MPI_File_write_at_all со вторым параметром, равным $N - 1$, изменить в каждой строке исходной матрицы выделенный элемент, заменив его значение на ранг связанного с ним

процесса (выделенный элемент в первой строке следует заменить на 0, во второй строке — на 1, и т. д.). Для этого предварительно с помощью функции MPI_File_set_view определить новый вид данных с базовым типом MPI_INT, таким же файловым типом и подходящим смещением (своим для каждого процесса).

MPI6File19. В главном процессе дано имя существующего файла вещественных чисел, содержащего элементы прямоугольной матрицы размера $K \times 6$, где K — количество процессов. Кроме того, в каждом процессе дано целое число N ($1 \le N \le 6$) — порядковый номер *выделенного* элемента в некоторой строке матрицы (элементы в строке нумеруются от 1), причем с процессом ранга 0 связывается последняя строка матрицы, с процессом ранга 1 — предпоследняя строка, и т. д. Используя единственный вызов коллективной функции MPI_File_read_at_all со вторым параметром, равным $N - 1$, прочесть в каждой строке исходной матрицы выделенный элемент и вывести его значение в соответствующем процессе (выделенный элемент в первой строке следует вывести в последнем процессе, во второй строке — в предпоследнем процессе, и т. д.). Для этого предварительно с помощью функции MPI_File_set_view определить новый вид данных с базовым типом MPI_DOUBLE, таким же файловым типом и подходящим смещением (своим для каждого процесса).

MPI6File20. В главном процессе дано имя файла. Кроме того, в каждом процессе дан набор из $R + 1$ вещественного числа, где R — ранг процесса (0, 1, …). Создать новый файл целых чисел с указанным именем. Используя единственный вызов коллективной функции MPI_File_write_all (и не применяя функцию MPI_File_seek), записать в файл все данные числа в порядке, обратном их следованию в исходном наборе: вначале записываются (в обратном порядке) все числа из процесса с наибольшим рангом, затем все числа из процесса с предыдущим рангом, и т. д. Для этого предварительно с помощью функции MPI_File_set_view определить новый вид данных с базовым типом MPI_DOUBLE, таким же файловым типом и подходящим смещением (своим для каждого процесса).

MPI6File21*. В главном процессе дано имя существующего файла целых чисел, содержащего $3K$ элементов, где K — количество процессов. В каждом процессе прочесть и вывести три элемента A, B, C, расположенных в исходном файле в следующем порядке (индекс указывает ранг процесса): $A_0, A_1, \ldots, A_{K-1}, B_0, B_1, \ldots, B_{K-1}, C_0, C_1, \ldots, C_{K-1}$. Для этого использовать единственный вызов коллективной функции MPI_File_read_all, предварительно определив новый файловый вид данных с базовым типом MPI_INT, подходящим смещением (своим для ка-

ждого процесса) и новым файловым типом, состоящим из одного целочисленного элемента и завершающего пустого промежутка, размер которого равен протяженности набора из $K - 1$ целого числа.

MPI6File22*. В главном процессе дано имя файла. Кроме того, в каждом процессе дано по 3 целых числа: A, B, C. Количество процессов равно K. Создать новый файл целых чисел с указанным именем и записать в него исходные числа в следующем порядке (индекс указывает ранг процесса): A_{K-1}, A_{K-2}, …, A_0, B_{K-1}, B_{K-2}, …, B_0, C_{K-1}, C_{K-2}, …, C_0. Для этого использовать единственный вызов коллективной функции MPI_File_write_all, предварительно определив новый файловый вид данных с базовым типом MPI_INT, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из одного целочисленного элемента и завершающего пустого промежутка, размер которого равен протяженности набора из $K - 1$ целого числа.

MPI6File23**. В главном процессе дано имя существующего файла вещественных чисел, содержащего $6K$ элементов, где K — количество процессов. В каждом процессе прочесть и вывести шесть элементов A, B, C, D, E, F, расположенных в исходном файле в следующем порядке (индекс указывает ранг процесса): A_0, B_0, C_0, A_1, B_1, C_1, …, A_{K-1}, B_{K-1}, C_{K-1}, D_0, E_0, F_0, D_1, E_1, F_1, …, D_{K-1}, E_{K-1}, F_{K-1}. Для этого использовать единственный вызов коллективной функции MPI_File_read_all, предварительно определив новый файловый вид данных с базовым типом MPI_DOUBLE, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из трех вещественных элементов и завершающего пустого промежутка подходящего размера.

MPI6File24**. 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 указанным именем и записать в него исходные числа в следующем порядке (индекс указывает ранг процесса): A_{K-1}, B_{K-1}, A_{K-2}, B_{K-2}, …, A_0, B_0, C_{K-1}, D_{K-1}, C_{K-2}, D_{K-2}, …, C_0, 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 и завершающего пустого промежутка подходящего размера.

MPI6File25**. В главном процессе дано имя файла. Кроме того, в процессе ранга R ($R = 0$, 1, …, $K - 1$, где K — количество процессов) дано $3 \cdot (R + 1)$ целых чисел: в процессе 0 даны 3 числа A, B, C, в процессе 1 — 6 чисел A, A', B, B', C, C', в процессе 2 — 9 чисел A, A', A'', B, B', B'', C, C', C'', и т. д. Создать новый файл целых чисел с указанным

именем и записать в него исходные числа в следующем порядке (индекс указывает ранг процесса): A_0, A_1, A'_1, A_2, A'_2, A''_2, …, B_0, B_1, B'_1, B_2, B'_2, B''_2, … . Для этого использовать единственный вызов коллективной функции MPI_File_write_all, предварительно определив новый файловый вид данных с базовым типом MPI_INT, подходящим смещением (своим для каждого процесса) и новым файловым типом (своим для каждого процесса), состоящим из $R + 1$ целого элемента и завершающего пустого промежутка подходящего размера.

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 указанным 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.

Примечание. Решение данной задачи приведено в п. 1.3.3.

MPI6File27*. В главном процессе дано имя существующего файла вещественных чисел, содержащего элементы прямоугольной матрицы размера $(K/2) \times K$, где K — количество процессов (четное число). В процессе ранга R, $R = 0, \ldots, K - 1$, прочесть и вывести элементы столбца исходной матрицы с номером $R + 1$ (столбцы нумеруются от 1). Для этого использовать единственный вызов функции MPI_File_read_all, предварительно определив новый файловый вид данных с базовым типом MPI_DOUBLE, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из вещественного элемента и завершающего пустого промежутка подходящего размера.

MPI6File28*. В главном процессе дано имя файла. Кроме того, в каждом процессе дано целое число N и $K/2$ вещественных чисел, где K — количество процессов (четное число). Числа N для всех процессов различны и лежат в диапазоне от 1 до K. Создать новый файл вещественных чисел с указанным именем и записать в него матрицу размера $(K/2) \times K$, причем каждый процесс должен записать свой набор вещественных чисел в столбец матрицы с порядковым номером N (столбцы нумеруются от 1). 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 вещественного элемента и завершающего пустого промежутка подходящего размера.

MPI6File29**. В главном процессе дано имя существующего файла целых чисел, содержащего элементы блочной прямоугольной матрицы. Размер матрицы в блоках равен $(K/3) \times 3$, где K — количество процессов (число K кратно 3). Каждый блок представляет собой квадратную матрицу порядка N (значение N может лежать в диапазоне от 2 до 5). В каждом процессе требуется прочесть и вывести один блок исходной матрицы, перебирая блоки по строкам. Для этого использовать единственный вызов функции MPI_File_read_all, предварительно определив новый файловый вид данных с базовым типом MPI_INT, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из N целочисленных элементов и завершающего пустого промежутка подходящего размера. Для определения числа N использовать размер исходного файла, получив его с помощью функции MPI_File_get_size.

MPI6File30**. В главном процессе дано имя файла и число N, которое может лежать в диапазоне от 2 до 5. Кроме того, в каждом процессе даны два целых числа I и J, определяющие позицию (номер строки и столбца) некоторого квадратного блока блочной прямоугольной матрицы размера $(K/3) \times 3$ блоков, где K — количество процессов (число K кратно 3). Значения I лежат в диапазоне от 1 до $K/3$, значения J лежат в диапазоне от 1 до 3; все процессы содержат различные позиции блоков. Каждый блок представляет собой квадратную матрицу целых чисел порядка N. Создать новый файл целых чисел с указанным именем и записать в него блочную матрицу размера $(K/3) \times 3$, причем каждый процесс должен записать в матрицу блок в позиции (I, J), а все элементы блока, записанного процессом ранга R ($R = 0, 1, \ldots, K - 1$), должны быть равны числу R. Для этого использовать единственный вызов функции MPI_File_write_all, предварительно определив новый файловый вид данных с базовым типом MPI_INT, подходящим смещением (своим для каждого процесса) и новым файловым типом, состоящим из K целочисленных элементов и завершающего пустого промежутка подходящего размера. Для передачи значения N во все процессы использовать коллективную функцию MPI_Bcast.

2.7. Односторонние коммуникации (MPI-2)

При определении окна доступа с помощью функции MPI_Win_create рекомендуется указывать смещение disp_unit (третий параметр) равным протяженности элемента данных соответствующего типа (в заданиях это все-

гда либо тип MPI_INT, либо тип MPI_DOUBLE; для данных типов протяженность равна размеру, и ее можно определять с помощью функции MPI_Type_size). В этом случае при последующих вызовах функций одностороннего доступа MPI_Get, MPI_Put, MPI_Accumulate смещение target_disp (пятый параметр любой из этих функций) будет определяться начальным индексом используемого фрагмента массива (а не числом байтов от начала массива до требуемого фрагмента, как в случае параметра disp_unit, равного 1).

В качестве параметра info (четвертый параметр функции MPI_Win_create) достаточно указывать константу MPI_INFO_NULL.

Если в некоторых процессах окно доступа определять не требуется, то параметр size (второй параметр функции MPI_Win_create) в этих процессах должен быть равен 0.

Параметр assert во всех синхронизирующих функциях (MPI_Win_fence, MPI_Win_start, MPI_Win_post, MPI_Win_lock) достаточно полагать равным 0 (данный параметр является предпоследним параметром во всех перечисленных функциях).

В первой подгруппе (задания MPI7Win1–MPI7Win17) в качестве синхронизирующих функций следует использовать *коллективную* функцию MPI_Win_fence, которая должна вызываться как перед действиями, связанными с односторонней пересылкой данных, так и после этих действий (но перед действиями, связанными с доступом к пересланным данным).

Задания второй подгруппы (MPI7Win18–MPI7Win30) требуют применения *локальных* вариантов синхронизации: функций MPI_Win_start, MPI_Win_complete, MPI_Win_post, MPI_Win_wait или пары функций MPI_Win_lock, MPI_Win_unlock. В заданиях этой подгруппы всегда указывается, какой вариант синхронизации требуется использовать.

Перед выполнением заданий из данного пункта следует ознакомиться с примерами решения задач MPI7Win13 и MPI7Win23, приведенными в п. 1.3.5 и 1.3.6.

2.7.1. Односторонние коммуникации с простейшей синхронизацией

MPI7Win1. В каждом из подчиненных процессов дано одно целое число. В главном процессе определить окно доступа размера K целых чисел (K — количество подчиненных процессов) и, используя функцию MPI_Put в подчиненных процессах, записать в главный процесс все исходные числа, после чего вывести эти числа в порядке возрастания рангов переславших их процессов.

MPI7Win2. В каждом из подчиненных процессов дано R вещественных чисел, где R — ранг процесса (1, 2, …). В главном процессе определить окно доступа подходящего размера и, используя функцию MPI_Put в подчиненных процессах, записать в главный процесс все исходные

числа, после чего вывести эти числа в порядке возрастания рангов переславших их процессов.

MPI7Win3. В главном процессе дан массив A из K целых чисел, где K — количество подчиненных процессов. Определить в главном процессе окно доступа, содержащее массив A, и, используя функцию MPI_Get в подчиненных процессах, получить и вывести в них по одному элементу массива A, перебирая элементы с конца (элемент A_0 получить в последнем процессе, A_1 в предпоследнем, и т. д.).

MPI7Win4. В главном процессе дан массив A из $K + 4$ вещественных чисел, где K — количество подчиненных процессов. Определить в главном процессе окно доступа, содержащее массив A, и, используя функцию MPI_Get в подчиненных процессах, получить и вывести в них по пять элементов массива A, начиная с элемента с индексом $R - 1$, где R — ранг подчиненного процесса ($R = 1, 2, \ldots, K - 1$).

MPI7Win5. В главном процессе дан массив A из K целых чисел, где K — количество подчиненных процессов. В каждом подчиненном процессе дан индекс N (число от 0 до $K - 1$) и целое число B. В главном процессе определить окно доступа, содержащее массив A, и, используя функцию MPI_Accumulate в каждом подчиненном процессе, умножить элемент A_N на число B, после чего вывести в главном процессе измененный массив A.

Примечание. Некоторые подчиненные процессы могут содержать совпадающие значения N; в этом случае элемент A_N должен умножаться на несколько чисел. Данное обстоятельство не требует дополнительных действий по синхронизации в силу особенностей реализации функции MPI_Accumulate.

MPI7Win6. В главном процессе дан вещественный массив A размера $2K - 1$ (K — количество подчиненных процессов), в каждом подчиненном процессе дан вещественный массив B размера R, где R — ранг процесса ($1, 2, \ldots, K - 1$). В главном процессе определить окно доступа, содержащее массив A, и, используя функцию MPI_Accumulate в каждом подчиненном процессе, прибавить к элементам массива A, начиная с индекса $R - 1$, значения всех элементов массива B из процесса ранга R, после чего вывести в главном процессе измененный массив A (единственный элемент B_0 из процесса 1 прибавляется к элементу A_0, элементы B_0 и B_1 из процесса 2 прибавляются соответственно к элементам A_1 и A_2, элементы B_0, B_1 и B_2 из процесса 3 прибавляются соответственно к элементам A_2, A_3 и A_4, и т. д.).

Примечание. К элементам массива A, начиная с индекса 2, потребуется прибавлять несколько значений из разных подчиненных процессов. Данное обстоятельство не требует дополнительных действий

по синхронизации в силу особенностей реализации функции MPI_Accumulate.

MPI7Win7*. В главном процессе дан массив A из $2K$ целых чисел, где K — количество подчиненных процессов. Во всех подчиненных процессах определить окно доступа из двух целых чисел и, используя несколько вызовов функции MPI_Put в главном процессе, записать и вывести в каждом подчиненном процессе по два элемента из массива A, перебирая их в исходном порядке (элементы A_0 и A_1 надо вывести в процессе 1, элементы A_2 и A_3 — в процессе 2, и т. д.).

MPI7Win8*. В каждом процессе дано целое число R и вещественное число B. Все числа R различны и лежат в диапазоне от 0 до $K - 1$, где K — количество процессов. Во всех процессах определить окно доступа из одного вещественного числа и, используя функцию MPI_Put в каждом процессе, переслать число B из этого процесса в процесс R, после чего вывести полученные числа во всех процессах.

MPI7Win9*. В каждом процессе дан массив A из K целых чисел, где K — количество процессов. Во всех процессах определить окно доступа, содержащее массив A, и, используя несколько вызовов функции MPI_Get в каждом процессе R ($R = 0, \ldots, K - 1$), получить и вывести элементы всех массивов A с индексом R, перебирая эти элементы в порядке убывания рангов содержащих их процессов (вначале выводится элемент, полученный из процесса ранга $K - 1$, затем элемент из процесса ранга $K - 2$, и т. д.).

Примечание. Функцию MPI_Get, как и другие функции, обеспечивающие односторонние коммуникации, можно использовать и для доступа к окну, определенному в вызывающем процессе.

MPI7Win10*. В каждом процессе дан массив A из 3 вещественных чисел и целые числа N_1 и N_2, каждое из которых лежит в диапазоне от 0 до 2. Во всех процессах определить окно доступа, содержащее массив A, и, используя по два вызова функции MPI_Get в каждом процессе, получить из предыдущего процесса элемент его массива A с индексом N_1, а из последующего процесса — элемент с индексом N_2 (числа N_1 и N_2 берутся из вызывающего процесса, процессы перебираются циклически). Вывести в каждом процессе полученные числа в указанном порядке.

MPI7Win11*. Количество процессов K — четное число. В каждом процессе дан массив A из $K/2$ целых чисел. Во всех процессах нечетного ранга ($1, 3, \ldots, K - 1$) определить окно доступа, содержащее массив A, и, используя требуемое число вызовов функции MPI_Accumulate в каждом процессе четного ранга, добавить элемент $A[I]$ из процесса ран-

га $2J$ к элементу $A[J]$ из процесса ранга $2I + 1$ (I, $J = 0$, …, $K/2 - 1$) и вывести измененные массивы A во всех процессах нечетного ранга.

Указание. Требуемое преобразование можно описать по-другому: если через B обозначить матрицу порядка $K/2$, строки которой совпадают с массивами A из процессов четного ранга, а через C — матрицу того же порядка, строки которой совпадают с массивами A из процессов нечетного ранга, то преобразование состоит в прибавлении к I-му столбцу матрицы C соответствующих элементов I-й строки матрицы B.

MPI7Win12*. Решить задачу MPI7Win11, определив окна доступа в процессах четного ранга и используя вместо функций MPI_Accumulate функции MPI_Get в процессах нечетного ранга.

Указание. Поскольку числа, полученные из процессов четного ранга, необходимо добавлять к элементам массива A после вызова функции синхронизации MPI_Win_fence, удобно использовать вспомогательный массив для хранения полученных чисел.

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.

Примечание. Решение данной задачи приведено в п. 1.3.5.

MPI7Win14*. В каждом процессе дан массив вещественных чисел размера K (K — количество процессов), содержащий строку верхнетреугольной матрицы A, включая ее начальную нулевую часть (процесс ранга R содержит R-ю строку матрицы в предположении, что строки нумеруются от 0). Во всех процессах определить окно доступа, содержащее исходный массив, и, используя требуемое количество вызовов функции MPI_Get в каждом процессе, записать в эти массивы (и затем вывести) строки матрицы, транспонированной к исходной матри-

це A, включая ее завершающую нулевую часть. Вспомогательные массивы не использовать.

Указания. (1) Строки транспонированной матрицы совпадают со столбцами исходной матрицы, таким образом, полученная матрица будет нижнетреугольной. (2) Обнулять требуемые элементы массивов перед их выводом необходимо после второго вызова функции MPI_Win_fence. (3) Окно доступа для последнего процесса можно не создавать.

MPI7Win15*. Решить задачу MPI7Win14, используя вместо вызовов функций MPI_Get вызовы функций MPI_Put.

Указание. В данном случае можно не создавать окно доступа для главного процесса.

MPI7Win16**. В каждом процессе дана одна строка вещественной квадратной матрицы A порядка K, где K — количество процессов (процесс ранга R содержит R-ю строку матрицы в предположении, что строки нумеруются от 0). Кроме того, в каждом процессе дано вещественное число B. Во всех процессах определить окно доступа, содержащее строку матрицы A, и, используя требуемое число вызовов функции MPI_Accumulate в каждом процессе ранга R ($R = 0, \ldots, K - 1$), заменить в строке матрицы из следующего процесса все элементы, меньшие числа B из процесса R, на это число (процессы перебираются циклически). Затем, используя K вызовов функции MPI_Get в каждом процессе, получить и вывести столбец преобразованной матрицы A с номером, совпадающим с рангом процесса (столбцы также нумеруются от 0).

Указание. При выполнении этого задания в каждом процессе необходимо *трижды* вызывать функцию синхронизации MPI_Win_fence.

MPI7Win17**. В каждом процессе дана одна строка вещественной квадратной матрицы A порядка K, где K — количество процессов (процесс ранга R содержит R-ю строку матрицы в предположении, что строки нумеруются от 0). Кроме того, в каждом процессе дано вещественное число B. Во всех процессах определить окно доступа, содержащее строку матрицы A, и, используя требуемое число вызовов функции MPI_Accumulate в каждом процессе ранга R ($R = 0, \ldots, K - 1$), заменить в строке матрицы из предыдущего процесса все элементы, бóльшие числа B из процесса R, на это число (процессы перебираются циклически). Затем, используя K вызовов функции MPI_Accumulate в каждом подчиненном процессе, добавить начальный элемент строки из каждого подчиненного процесса ранга R ($1, \ldots, K - 1$) ко всем элементам столбца преобразованной матрицы A с номером R (столбцы также нумеруются от 0). После всех описанных преобразований вывести

в каждом процессе новое содержимое соответствующей строки матрицы A.

Указание. При выполнении этого задания в каждом процессе необходимо *трижды* вызывать функцию синхронизации MPI_Win_fence.

2.7.2. Дополнительные виды синхронизации

MPI7Win18. Количество процессов K — четное число. В каждом процессе четного ранга (0, 2, …, $K-2$) дано целое число A. Во всех процессах нечетного ранга (1, 3, …, $K-1$) определить окно доступа из одного целого числа и, используя функцию MPI_Put в каждом процессе четного ранга $2N$, переслать число A в процесс ранга $2N+1$ и вывести его в этом процессе. Для синхронизации использовать функции MPI_Win_start и MPI_Win_complete в процессах четного ранга и функции MPI_Win_post и MPI_Win_wait в процессах нечетного ранга; для создания группы процессов, указываемой в качестве первого параметра функций MPI_Win_start и MPI_Win_post, использовать функцию MPI_Group_incl, применив ее к группе, полученной из коммуникатора MPI_COMM_WORLD (с помощью функции MPI_Comm_group).

Примечание. В отличие от *коллективной* функции MPI_Win_fence, использовавшейся в предыдущих заданиях данной группы, синхронизирующие функции, используемые в этом и последующих заданиях, являются *локальными* и, кроме того, позволяют явным образом определить группы инициирующих (origin) и целевых (target) процессов при односторонних коммуникациях.

MPI7Win19. В главном процессе дан массив A из K вещественных чисел, где K — количество подчиненных процессов. В главном процессе определить окно доступа, содержащее массив A, и, используя функцию MPI_Get в каждом подчиненном процессе, получить и вывести один из элементов массива A, перебирая элементы в обратном порядке (в процессе ранга 1 надо получить элемент массива с индексом $K-1$, в процессе ранга 2 — элемент с индексом $K-2$, и т. д.). Для синхронизации использовать функции MPI_Win_start и MPI_Win_complete в подчиненных процессах и функции MPI_Win_post и MPI_Win_wait в главном процессе; для создания группы процессов, указываемой в качестве первого параметра функции MPI_Win_start, использовать функцию MPI_Group_incl, для создания группы, указываемой в функции MPI_Win_post, использовать функцию MPI_Group_excl (применив обе эти функции к группе, полученной из коммуникатора MPI_COMM_WORLD).

MPI7Win20. Число процессов K кратно 3. В процессах ранга $3N$ ($N = 0$, …, $K/3-1$) дан массив A из трех вещественных чисел. Во всех процессах, в которых дан массив, определить окно доступа, содержащее этот массив, и, используя по одному вызову функции MPI_Get в процессах

ранга $3N + 1$ и $3N + 2$ ($N = 0, \ldots, K/3 - 1$), прочесть и вывести соответственно один элемент A_0 и два элемента A_1 и A_2 из процесса ранга $3N$ (процесс 1 должен вывести элемент A_0, полученный из процесса 0, процесс 2 — элементы A_1 и A_2, полученные из процесса 0, процесс 4 — элемент A_0, полученный из процесса 3, и т. д.). Для синхронизации использовать функции MPI_Win_post и MPI_Win_wait в процессах ранга $3N$ и функции MPI_Win_start и MPI_Win_complete в остальных процессах.

MPI7Win21*. Количество процессов K — четное число. В главном процессе дан массив A из $K/2$ вещественных чисел и массив N целых чисел того же размера. Все элементы массива N различны и лежат в диапазоне от 1 до $K - 1$. В каждом подчиненном процессе определить окно доступа из одного вещественного числа и, используя требуемое количество вызовов функции MPI_Put в главном процессе, переслать в каждый из подчиненных процессов ранга N_I ($I = 0, \ldots, K/2 - 1$) число A_I и вывести полученное число (в остальных подчиненных процессах вывести вещественное число 0.0). Для синхронизации использовать функции MPI_Win_post и MPI_Win_wait в подчиненных процессах и функции MPI_Win_start и MPI_Win_complete в главном процессе.

MPI7Win22*. В главном процессе дан массив A вещественных чисел размера K (K — количество подчиненных процессов) и массив N целых чисел размера 8. Все элементы массива N лежат в диапазоне от 1 до K; некоторые элементы данного массива могут совпадать. В каждом подчиненном процессе дан вещественный массив B размера R, где R — ранг процесса ($R = 1, \ldots, K$). Определить в каждом подчиненном процессе окно доступа, содержащее массив B, и, используя требуемое количество вызовов функции MPI_Accumulate в главном процессе, добавить ко всем элементам массивов B из процессов ранга N_I ($I = 0, \ldots, 7$) элементы массива A с теми же индексами (к элементу B_0 добавляется элемент A_0, к элементу B_1 — элемент A_1, и т. д.); для некоторых массивов B элементы из массива A могут добавляться несколько раз. В каждом подчиненном процессе вывести массив B (который либо изменится, либо сохранит исходные значения элементов). Для синхронизации использовать функции MPI_Win_post и MPI_Win_wait в подчиненных процессах и функции MPI_Win_start и MPI_Win_complete в главном процессе.

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 ($I = 0, \ldots, 4$) 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.

Примечание. Решение данной задачи приведено в п. 1.3.6.

MPI7Win24. В каждом подчиненном процессе дано целое число N; все числа N различны и лежат в диапазоне от 0 до $K - 1$, где K — количество подчиненных процессов. В главном процессе определить окно доступа, содержащее целочисленный массив A размера K. Не выполняя никаких вызовов синхронизирующих функций в главном процессе (кроме вызова функции MPI_Barrier) и используя в подчиненных процессах последовательность вызовов синхронизирующих функции MPI_Win_lock, MPI_Win_unlock, MPI_Barrier, MPI_Win_lock, MPI_Win_unlock, записать в элементы массива A с индексом N ранг подчиненного процесса, содержащего данное значение N (с помощью функции MPI_Put), после чего получить и вывести в каждом подчиненном процессе все элементы преобразованного массива A (с помощью функции MPI_Get). В качестве первого параметра функции MPI_Win_lock указывать MPI_LOCK_SHARED.

Примечание. Синхронизирующие функции MPI_Win_lock, MPI_Win_unlock используются преимущественно при односторонних коммуникациях с *пассивными целевыми процессами* (passive targets), при которых целевой процесс не обрабатывает переданные ему данные, а выступает в роли их хранилища, доступного для других процессов.

MPI7Win25. Количество процессов K кратно 3. В процессах ранга $3N$ ($N = 0, \ldots, K/3 - 1$) дан вещественный массив A размера 5, в процессах ранга $3N + 1$ дано целое число M, лежащее в диапазоне от 0 до 4, и вещественное число B. В каждом процессе, содержащем исходный массив A, определить окно доступа с этим массивом. Используя функцию MPI_Accumulate в каждом процессе ранга $3N + 1$ ($N = 0, \ldots,$

$K/3 - 1$), преобразовать массив A из процесса ранга $3N$ следующим образом: если элемент массива с индексом M больше числа B, то он заменяется на число B (числа M и B берутся из процесса ранга $3N + 1$). После этого, используя функцию MPI_Get в каждом процессе ранга $3N + 2$, получить и вывести в нем все элементы преобразованного массива A из процесса ранга $3N$. Использовать синхронизирующие функции MPI_Win_Lock, MPI_Win_unlock, MPI_Barrier в процессах ранга $3N + 1$, функции MPI_Barrier, MPI_Win_Lock, MPI_Win_unlock в процессах ранга $3N + 2$ и функцию MPI_Barrier в процессах ранга $3N$. В качестве первого параметра функции MPI_Win_lock указывать MPI_LOCK_EXCLUSIVE.

MPI7Win26. В каждом подчиненном процессе дан вещественный массив A размера 5 с положительными элементами. В главном процессе определить окно доступа, содержащее вещественный массив B размера 5 с нулевыми элементами. Не выполняя никаких вызовов синхронизирующих функций в главном процессе (кроме вызова функции MPI_Barrier) и используя в подчиненных процессах последовательность вызовов синхронизирующих функции MPI_Win_lock, MPI_Win_unlock, MPI_Barrier, MPI_Win_lock, MPI_Win_unlock, записать в каждый из элементов массива B максимальный из элементов массивов A с тем же индексом (с помощью функции MPI_Accumulate), после чего получить и вывести в каждом подчиненном процессе все элементы преобразованного массива B (с помощью функции MPI_Get). В качестве первого параметра функции MPI_Win_lock указывать MPI_LOCK_SHARED.

MPI7Win27*. В каждом подчиненном процессе даны два вещественных числа X, Y — координаты точки на плоскости. Используя вызовы функции MPI_Get в главном процессе, получить в нем числа X_0, Y_0, равные координатам той точки (X, Y) среди точек, данных в подчиненных процессах, которая является наиболее удаленной от начала координат. После этого, используя вызовы функции MPI_Get в подчиненных процессах, получить и вывести в каждом из них числа X_0, Y_0, найденные в главном процессе. В каждом процессе определить окно доступа, содержащее два вещественных числа (X, Y для подчиненных процессов, X_0, Y_0 для главного процесса). Использовать синхронизирующие функции MPI_Win_lock, MPI_Win_unlock, MPI_Barrier в главном процессе и функции MPI_Barrier, MPI_Win_lock, MPI_Win_unlock в подчиненных процессах.

Примечание. Данную задачу невозможно решить, используя односторонние коммуникации только на стороне подчиненных процессов и при этом выполняя синхронизацию с помощью блокировок lock/unlock.

MPI7Win28*. Решить задачу MPI7Win27, используя единственное окно доступа в главном процессе, содержащее числа X_0, Y_0. Для нахождения чисел X_0, Y_0 использовать функцию MPI_Get и функцию MPI_Put в подчиненных процессах (для некоторых процессов вызывать функцию MPI_Put не потребуется), для пересылки найденных чисел X_0, Y_0 во все подчиненные процессы использовать в них функцию MPI_Get (как и в задаче MPI7Win27). Для синхронизации обменов на этапе нахождения чисел X_0, Y_0 использовать два вызова пары функций MPI_Win_start и MPI_Win_complete в подчиненных процессах и вызовы *в цикле* двух пар функций MPI_Win_post и MPI_Win_wait в главном процессе (при этом на каждой итерации цикла необходимо определять *новую группу процессов*, используемую при вызове функций MPI_Win_post). Для синхронизации действий при пересылке чисел X_0, Y_0 в подчиненные процессы использовать, как и в задаче MPI7Win27, функцию MPI_Barrier в главном процессе и функции MPI_Barrier, MPI_Win_Lock, MPI_Win_unlock в подчиненных процессах.

Примечание. Описанный вариант решения (в отличие от варианта, приведенного в задаче MPI7Win27) позволяет использовать односторонние коммуникации только на стороне подчиненных процессов, однако при этом на первом этапе решения требуется применять вариант синхронизации, отличный от блокировок lock/unlock.

MPI7Win29*. В каждом процессе дана одна строка целочисленной квадратной матрицы порядка K, где K — количество процессов (в процессе ранга R дана строка матрицы с номером R; строки нумеруются от 0). Используя вызовы функции MPI_Get в главном процессе, получить в нем строку матрицы с минимальной суммой элементов S и, кроме того, найти количество N строк с минимальной суммой (если $N > 1$, то в главном процессе надо сохранить последнюю из таких строк, т. е. строку с наибольшим номером). После этого, используя вызовы функции MPI_Get в подчиненных процессах, получить и вывести в каждом из них строку с минимальной суммой S, найденную в главном процессе, значение суммы S и количество N строк с минимальной суммой. В каждом процессе определить окно доступа, содержащее $K + 2$ целых числа; в первых K элементах окна содержатся элементы строки матрицы, в следующем элементе — сумма их значений S, а последний элемент предназначен для хранения числа N. Использовать синхронизирующие функции MPI_Win_lock, MPI_Win_unlock, MPI_Barrier в главном процессе и функции MPI_Barrier, MPI_Win_lock, MPI_Win_unlock в подчиненных процессах.

Примечание. Данную задачу невозможно решить, используя односторонние коммуникации только на стороне подчиненных процессов и выполняя синхронизацию с помощью блокировок lock/unlock.

MPI7Win30*. Решить задачу MPI7Win29, используя единственное окно доступа в главном процессе, содержащее строку матрицы и числа *S* и *N*. Для нахождения строки с минимальной суммой элементов и связанных с ней характеристик использовать функцию MPI_Get и функцию MPI_Put в подчиненных процессах (для некоторых процессов вызывать функцию MPI_Put не потребуется), для пересылки найденных данных во все подчиненные процессы использовать в них функцию MPI_Get (как и в задаче MPI7Win29). На этапе нахождения строки с минимальной суммой функцию MPI_Get следует использовать только для получения значений *S* и *N*. Для синхронизации обменов на этапе нахождения строки с минимальной суммой использовать два вызова пары функций MPI_Win_start и MPI_Win_complete в подчиненных процессах и вызовы *в цикле* двух пар функций MPI_Win_post и MPI_Win_wait в главном процессе (при этом на каждой итерации цикла необходимо определять *новую группу процессов*, используемую при вызове функций MPI_Win_post). Для синхронизации действий при пересылке найденных данных в подчиненные процессы использовать, как и в задаче MPI7Win29, функцию MPI_Barrier в главном процессе и функции MPI_Barrier, MPI_Win_Lock, MPI_Win_unlock в подчиненных процессах.

Примечание. Описанный вариант решения (в отличие от варианта, приведенного в задаче MPI7Win29) позволяет использовать односторонние коммуникации только на стороне подчиненных процессов, однако при этом на первом этапе решения требуется применять вариант синхронизации, отличный от блокировок lock/unlock.

2.8. Интеркоммуникаторы и динамическое создание процессов (MPI-2)

Базовые возможности, связанные с созданием интеркоммуникаторов и их использованием для обмена сообщениями между отдельными процессами, определены в стандарте MPI-1. Поэтому 5 заданий данной группы можно выполнять с применением системы MPICH 1.2.5 (это задания MPI8Inter1–MPI8Inter4 и MPI8Inter9). Прочие задания посвящены новым возможностям по созданию интеркоммуникаторов (MPI8Inter5–MPI8Inter8), средствам коллективного обмена для интеркоммуникаторов (MPI8Inter10–MPI8Inter14) и использованию интеркоммуникаторов для динамического создания процессов (MPI8Inter15–MPI8Inter22). Все эти возможности появились в стандарте MPI-2, поэтому для выполнения заданий необходимо подключать к программе систему MPICH2 1.3.

В качестве коммуникатора-*посредника* (третьего параметра peer функции MPI_Intercomm_create) следует указывать *копию* коммуникатора MPI_COMM_WORLD, созданную с помощью функции MPI_Comm_dup.

При динамическом создании процессов с помощью функции MPI_Comm_spawn в заданиях MPI8Inter15–MPI8Inter22 первый параметр command должен быть равен имени исполняемого файла ptprj.exe, второй параметр argv — нулевому указателю NULL, четвертый параметр info — константе MPI_NULL_INFO, а последний параметр array_of_errcodes — константе MPI_ERRCODES_IGNORE. Если в задании не указано, какой коммуникатор должен использоваться в качестве исходного при создании новых процессов, то предполагается, что этим коммуникатором является MPI_COMM_WORLD.

Вместо строки "ptprj.exe" можно использовать реализованную в задачнике функцию char* GetExename(), которая возвращает полное имя исполняемого файла.

Перед выполнением заданий из данного пункта следует ознакомиться с примерами решения задач MPI8Inter9 и MPI8Inter15, приведенными в п. 1.3.7 и 1.3.8.

2.8.1. Создание интеркоммуникаторов

MPI8Inter1. Количество процессов K — четное число. В каждом процессе дано целое число X. Используя функции MPI_Comm_group, MPI_Group_range_incl и MPI_Comm_create, создать коммуникаторы, первый из которых содержит процессы четного ранга в том же порядке (0, 2, …, $K/2 - 2$), а второй — процессы нечетного ранга в том же порядке (1, 3, …, $K/2 - 1$). Вывести ранги процессов R в созданных коммуникаторах. Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. Используя функции MPI_Send и MPI_Recv для созданного интеркоммуникатора, получить для каждого процесса число X из процесса того же ранга, входящего в другую группу этого же интеркоммуникатора, и вывести полученные числа.

MPI8Inter2. Количество процессов K — четное число. В каждом процессе дано целое число C и вещественное число X. Числа C равны либо 0, либо 1, количество значений 1 равно количеству значений 0, причем в процессе ранга 0 дано C, равное 0, а в процессе ранга $K - 1$ дано C, равное 1. Используя один вызов функции MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы со значениями $C = 0$ в том же порядке, а второй — процессы со значениями $C = 1$ в обратном порядке. Вывести ранги процессов R в созданных коммуникаторах (при этом значение $R = 0$ получат первый и последний процессы исходного коммуникатора MPI_COMM_WORLD). Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. Используя функции MPI_Send и MPI_Recv для соз-

данного интеркоммуникатора, получить для каждого процесса число X из процесса того же ранга, входящего в другую группу этого интеркоммуникатора, и вывести полученные числа.

MPI8Inter3*. Количество процессов K кратно 3. В процессах ранга $3N$ ($N = 0, \ldots, 3K - 3$) дано вещественное число X, в процессах ранга $3N + 1$ даны вещественные числа X и Y, в процессах ранга $3N + 2$ дано вещественное число Y. Используя функции MPI_Comm_group, MPI_Group_range_incl и MPI_Comm_create, создать коммуникаторы, первый из которых содержит группу процессов ранга $3N$ в том же порядке ($0, 3, \ldots, K - 3$), второй — группу процессов ранга $3N + 1$ в обратном порядке ($K - 2, K - 5, \ldots, 1$), а третий — группу процессов ранга $3N + 2$ в том же порядке ($2, 5, \ldots, K - 1$). Вывести ранги процессов R в созданных коммуникаторах. Затем, используя функцию MPI_Intercomm_create, объединить созданные коммуникаторы в два интеркоммуникатора: первый должен содержать первую и вторую группы процессов, а второй — вторую и третью группы. Используя функции MPI_Send и MPI_Recv для созданных интеркоммуникаторов, поменять числа X для процессов одинакового ранга, входящих в первую и вторую группу, и числа Y для процессов одинакового ранга, входящих во вторую и третью группу. Вывести в каждом процессе полученные числа.

Указание. Следует обратить внимание на то, что функция MPI_Intercomm_create должна вызываться один раз для процессов, входящих в первую и третью группы, и два раза для процессов, входящих во вторую группу, и такое же количество вызовов должно быть для функций MPI_Send и MPI_Recv.

MPI8Inter4*. Количество процессов K кратно 3. В каждом процессе даны три целых числа. Первое число (обозначаемое буквой C) лежит в диапазоне от 0 до 2, причем каждое из значений 0, 1, 2 встречается одинаковое число раз (равное $K/3$) и при этом в процессах ранга 0, 1, 2 значения C совпадают с рангами этих процессов. Используя один вызов функции MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы со значениями $C = 0$ в том же порядке, второй — процессы со значениями $C = 1$ в том же порядке, а третий — процессы со значениями $C = 2$ в том же порядке. Вывести ранги процессов R в созданных коммуникаторах (при этом значение $R = 0$ получат первые три процесса исходного коммуникатора MPI_COMM_WORLD). Затем, используя по два вызова функции MPI_Intercomm_create в каждом процессе, объединить созданные коммуникаторы в три интеркоммуникатора: первый должен содержать группы процессов со значениями C, равными 0 и 1, второй — группы со значениями 1 и 2, третий — группы со значениями 0 и 2 (таким образом, созданные интеркомму-

никаторы будут образовывать кольцо, связывающее все три ранее созданные группы). Считая, что в первой группе процессов два следующих исходных числа обозначаются буквами X и Y, во второй группе — Y и Z, а в третьей — Z и X (в указанном порядке) и используя в каждом процессе по два вызова функций MPI_Send и MPI_Recv для созданных интеркоммуникаторов, поменять числа X для процессов одинакового ранга, входящих в первую и вторую группу, числа Y — для процессов одинакового ранга, входящих во вторую и третью группу, и числа Z — для процессов одинакового ранга, входящих в первую и третью группу. Вывести в каждом процессе полученные числа.

MPI8Inter5*. Количество процессов K кратно 4. В каждом процессе дано целое число X. Используя функции MPI_Comm_group, MPI_Group_range_incl и MPI_Comm_create, создать коммуникаторы, первый из которых содержит первую половину процессов (с рангами 0, 1, …, $K/2 - 1$ в указанном порядке), а второй — вторую половину (с рангами $K/2$, $K/2 + 1$, …, $K - 1$ в указанном порядке). Вывести ранги процессов R_1 в созданных коммуникаторах. Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. После этого, используя функцию MPI_Comm_create для созданного интеркоммуникатора, получить новый интеркоммуникатор, первая группа которого содержит процессы из первой группы исходного интеркоммуникатора с четными рангами (в том же порядке), а вторая группа — процессы из второй группы исходного интеркоммуникатора с нечетными рангами в обратном порядке (таким образом, в первую группу нового интеркоммуникатора будут входить процессы исходного коммуникатора MPI_COMM_WORLD с рангами 0, 2, …, $K/2 - 2$, а во вторую — с рангами $K - 1$, $K - 3$, …, $K/2 + 1$). Вывести ранги процессов R_2, входящих в новый интеркоммуникатор. Используя функции MPI_Send и MPI_Recv для нового интеркоммуникатора, получить для каждого процесса число X из процесса того же ранга, входящего в другую группу этого интеркоммуникатора, и вывести полученные числа.

MPI8Inter6*. Количество процессов K кратно 4. В каждом процессе дано вещественное число X. Используя функции MPI_Comm_group, MPI_Group_range_incl и MPI_Comm_create, создать коммуникаторы, первый из которых содержит первую половину процессов (с рангами 0, 1, …, $K/2 - 1$ в указанном порядке), а второй — вторую половину (с рангами $K/2$, $K/2 + 1$, …, $K - 1$ в указанном порядке). Вывести ранги процессов R_1 в созданных коммуникаторах. Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. После этого, используя один вызов функции

MPI_Comm_split для созданного интеркоммуникатора, получить два новых интеркоммуникатора. Первый из новых интеркоммуникаторов содержит процессы исходного интеркоммуникатора с четными рангами, а второй — с нечетными, причем процессы во второй группе каждого из новых интеркоммуникаторов должны располагаться в обратном порядке (таким образом, если использовать ранги процессов для исходного коммуникатора MPI_COMM_WORLD, то первый интеркоммуникатор содержит группы процессов ранга 0, 2, …, $K/2 - 2$ и $K - 2$, $K - 4$, …, $K/2$, а второй — группы процессов ранга 1, 3, …, $K/2 - 1$ и $K - 1$, $K - 3$, …, $K/2 + 1$). Вывести ранги процессов R_2, входящих в новые интеркоммуникаторы. Используя функции MPI_Send и MPI_Recv для новых интеркоммуникаторов, получить для каждого процесса число X из процесса того же ранга, входящего в другую группу этого интеркоммуникатора, и вывести полученные числа.

MPI8Inter7**. Количество процессов K — четное число. В каждом процессе дано целое число C, равное либо 0, либо 1, причем известно, что в первой половине процессов дано единственное значение $C = 1$, а во второй половине количество значений $C = 1$ больше одного и, кроме того, имеется хотя бы одно значение $C = 0$. Используя функцию MPI_Comm_split, создать коммуникаторы, первый из которых содержит первую половину процессов (с рангами 0, 1, …, $K/2 - 1$ в указанном порядке), а второй — вторую половину (с рангами $K/2$, $K/2 + 1$, …, $K - 1$ в указанном порядке). Вывести ранги процессов R_1 в созданных коммуникаторах. Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. После этого, используя функцию MPI_Comm_split для созданного интеркоммуникатора, получить новый интеркоммуникатор, группы которого содержат процессы из соответствующих групп исходного интеркоммуникатора со значениями $C = 1$, взятые в обратном порядке (таким образом, в первую группу нового интеркоммуникатора будет входить единственный процесс, а количество процессов во второй группе будет лежать в диапазоне от 2 до $K/2 - 1$). Вывести ранги процессов R_2, входящих в ту группу нового интеркоммуникатора, которая содержит более одного процесса. В единственном процессе первой группы нового интеркоммуникатора ввести массив Y из K_2 целых чисел, где K_2 — количество процессов во второй группе; в каждом процессе второй группы этого же интеркоммуникатора ввести по одному целому числу X. Используя требуемое количество вызовов функций MPI_Send и MPI_Recv для всех процессов нового интеркоммуникатора, получить в процессе первой группы все числа X из процессов второй группы (в порядке возрастания рангов этих процессов), а в каждом процессе второй группы ранга R_2 (0, 1, …, $K_2 - 1$) получить элемент

с индексом R_2 из массива Y, данного в процессе первой группы. Вывести полученные числа.

Примечание. В реализации MPICH 2 версии 1.3, используемой в задачнике, функция MPI_Comm_split приводит к ошибочной программе, если она применяется к интеркоммуникатору и при этом какие-либо из значений ее параметра color равны MPI_UNDEFINED. Таким образом, для корректной работы программ необходимо использовать только *неотрицательные* значения color. Кроме того, программа в дальнейшем может вести себя некорректно, если в результате применения функции MPI_Comm_split к интеркоммуникатору создаются *пустые группы* (это возможно, если для некоторых процессов одной из групп исходного интеркоммуникатора указываются значения color, отличные от всех значений color для процессов другой группы).

MPI8Inter8**. В каждом процессе дано целое число C, лежащее в диапазоне от 0 до 2, причем известно, что и у процессов четного ранга, и у процессов нечетного ранга имеется хотя бы по одному из значений 0, 1, 2. Используя один вызов функции MPI_Comm_split, создать коммуникаторы, первый из которых содержит процессы с четными рангами (в порядке возрастания рангов), а второй — процессы с нечетными рангами (также в порядке возрастания рангов). Вывести ранги процессов R_1 в созданных коммуникаторах. Затем объединить созданные коммуникаторы в интеркоммуникатор с помощью функции MPI_Intercomm_create. После этого, используя один вызов функции MPI_Comm_split для созданного интеркоммуникатора, получить три новых интеркоммуникатора, группы которых содержат процессы из соответствующих групп исходного интеркоммуникатора с одинаковыми значениями C, взятые в том же порядке (таким образом, в первую группу первого интеркоммуникатора будут входить процессы четного ранга со значениями $C = 0$, а, например, во вторую группу третьего интеркоммуникатора будут входить процессы нечетного ранга со значениями $C = 2$). Вывести ранги процессов R_2 в новых интеркоммуникаторах. В процессах из первых групп созданных интеркоммуникаторов ввести по одному целому числу X, а в процессах из вторых групп — по одному целому числу Y. Используя требуемое количество вызовов функций MPI_Send и MPI_Recv для всех процессов новых интеркоммуникаторов, переслать все числа X каждому из процессов второй группы этого же коммуникатора, а все числа Y — каждому из процессов первой группы и вывести полученные числа в порядке возрастания рангов переславших их процессов.

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.

Примечание. Решение данной задачи приведено в п. 1.3.7.

2.8.2. Коллективные операции для интеркоммуникаторов

MPI8Inter10*. 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 (группа, содержащая процессы со значениями $C = 1$, считается первой группой созданного интеркоммуникатора, а группа процессов со значениями $C = 2$ — второй). После этого ввести в процессах обеих групп созданного интеркоммуникатора целые числа R_1 и R_2. Значения чисел R_1 во всех процессах совпадают и указывают ранг выделенного процесса первой группы; значения чисел R_2 во всех процессах также совпадают и указывают ранг выделенного процесса второй группы. В выделенном процессе первой группы дан набор из трех целых чисел X, в выделенном процессе второй группы — набор из трех целых чисел Y. Используя по два вызова коллективной функции MPI_Bcast в каждом процессе созданного интеркоммуникатора, переслать набор чисел X во все про-

цессы второй группы, а набор чисел Y во все процессы первой группы, после чего вывести полученные числа.

MPI8Inter11*. 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 (группа, содержащая процессы со значениями $C = 1$, считается первой группой созданного интеркоммуникатора, а группа процессов со значениями $C = 2$ — второй). После этого ввести в процессах обеих групп созданного интеркоммуникатора целое число R_1, которое совпадает во всех процессах и указывает ранг выделенного процесса первой группы. В выделенном процессе первой группы дан набор из K_2 целых чисел X, где K_2 — количество процессов во второй группе. Используя один вызов коллективной функции MPI_Scatter в каждом процессе созданного интеркоммуникатора, переслать по одному числу из набора X во все процессы второй группы (в порядке возрастания рангов процессов), после чего вывести полученные числа.

MPI8Inter12*. 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 (группа, содержащая процессы со значениями $C = 1$, считается первой группой созданного интеркоммуникатора, а группа процессов со значениями $C = 2$ — второй). После этого ввести в процессах обеих групп созданного интеркоммуникатора целое число R_2, которое совпадает во всех процессах и указывает ранг выделенного процесса второй группы. В каждом процессе первой группы дано по одному целому числу X. Используя один вызов коллективной функции MPI_Gather в каждом процессе созданного

интеркоммуникатора, переслать исходные числа X (в порядке возрастания рангов содержащих их процессов) в выделенный процесс второй группы, после чего вывести в этом процессе полученные числа.

MPI8Inter13*. 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 (группа, содержащая процессы со значениями $C = 1$, считается первой группой созданного интеркоммуникатора, а группа процессов со значениями $C = 2$ — второй). В каждом процессе первой группы дано по одному целому числу X, в каждом процессе второй группы дано по одному целому числу Y. Используя один вызов коллективной функции MPI_Allreduce в каждом процессе созданного интеркоммуникатора, получить в каждом процессе первой группы число Y_{min} — минимальное из исходных чисел Y, а в каждом процессе второй группы число X_{max} — максимальное из исходных чисел X, после чего вывести полученные числа.

MPI8Inter14*. 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 $K - 1$. 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 обратном 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 (группа, содержащая процессы со значениями $C = 1$, считается первой группой созданного интеркоммуникатора, а группа процессов со значениями $C = 2$ — второй). В каждом процессе первой группы дан массив целых чисел X размера K_2, где K_2 — количество процессов во второй группе, в каждом процессе второй группы дан массив целых чисел Y размера K_1, где K_1 — количество процессов в первой группе. Используя один вызов коллективной функции MPI_Alltoall в каждом процессе созданного интеркоммуникатора, переслать в процесс ранга R_1 первой группы

($R_1 = 0, \ldots, K_1 - 1$) элементы с индексом R_1 из всех массивов Y, а в процесс ранга R_2 второй группы ($R_2 = 0, \ldots, K_2 - 1$) элементы с индексом R_2 из всех массивов X. Вывести полученные числа в порядке возрастания рангов переславших их процессов.

2.8.3. Динамическое создание 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.

Примечание. Решение данной задачи приведено в п. 1.3.8.

MPI8Inter16. В каждом процессе дан массив из K вещественных чисел, где K — количество процессов. Используя один вызов функции MPI_Comm_spawn с первым параметром "ptprj.exe", создать K новых процессов. С помощью коллективной функции MPI_Reduce_scatter_block переслать в созданный процесс ранга R ($R = 0, \ldots, K - 1$) максимальный из элементов исходных массивов с индексом R и отобразить полученные максимальные элементы в разделе отладки, используя в каждом новом процессе функцию Show. Затем с помощью функций MPI_Send и MPI_Recv переслать найденный максимальный элемент из нового процесса ранга R ($R = 0, \ldots, K - 1$) в исходный процесс того же ранга и вывести полученные элементы в исходных процессах.

MPI8Inter17*. Количество процессов K — четное число. В процессах ранга 0 и 1 даны массивы вещественных чисел размера $K/2$. Используя один вызов функции MPI_Comm_spawn с первым параметром "ptprj.exe", создать два новых процесса. Используя один вызов функции MPI_Comm_split для интеркоммуникатора, связанного с созданными процессами, создать два новых интеркоммуникатора: первый содержит группу исходных процессов четного ранга ($0, \ldots, K - 2$), а также, в качестве второй группы, первый из новых процессов (ранга 0), второй содержит группу исходных процессов нечетного ранга ($1, \ldots, K - 1$), а также, в качестве второй группы, второй из новых процессов (ранга 1). Используя функции MPI_Send в исходных процессах и функции MPI_Recv в новых процессах, переслать все исходные числа из первого процесса первой группы каждого интеркоммуникатора в единственный процесс, входящий в его вторую группу. Отобразить полученные числа в разделе отладки, используя в новых процессах функцию Show. Затем с помощью коллективной функции MPI_Scatter для интеркоммуникаторов переслать по одному числу из новых про-

цессов во все процессы первой группы соответствующего интеркоммуникатора (в порядке возрастания рангов процессов) и вывести полученные числа.

MPI8Inter18*. Количество процессов K — четное число. В каждом процессе дан массив вещественных чисел размера $K/2$. Используя один вызов функции MPI_Comm_spawn с первым параметром "ptprj.exe", создать K новых процессов. Используя один вызов функции MPI_Comm_split для интеркоммуникатора, связанного с созданными процессами, создать два новых интеркоммуникатора: первый содержит группу исходных процессов четного ранга (0, …, $K - 2$), а также, в качестве второй группы, созданные процессы четного ранга, второй содержит группу исходных процессов нечетного ранга (1, …, $K - 1$), а также, в качестве второй группы, созданные процессы нечетного ранга. Для каждого из созданных интеркоммуникаторов выполнить следующие действия: среди элементов исходных массивов с индексом R ($R = 0, \ldots, K/2 - 1$), входящих в первую группу каждого интеркоммуникатора, найти минимальный (для первого интеркоммуникатора) или максимальный (для второго) и переслать этот экстремальный элемент в тот из новых процессов, который имеет ранг R во второй группе соответствующего интеркоммуникатора (например, минимальный из первых элементов массивов, которые даны в исходных процессах четного ранга, следует переслать в первый из созданных процессов, а максимальный из первых элементов массивов, которые даны в исходных процессах нечетного ранга, следует переслать во второй из созданных процессов, поскольку этот процесс имеет ранг 0 в соответствующем интеркоммуникаторе). Для выполнения этих действий использовать коллективную функцию MPI_Reduce_scatter_block. Полученные экстремальные элементы отобразить в разделе отладки, используя в каждом новом процессе функцию Show. Затем с помощью коллективной функции MPI_Reduce найти минимальный элемент из тех минимумов, которые были получены во второй группе первого интеркоммуникатора, переслать и вывести его в первом процессе первой группы этого интеркоммуникатора (т. е. в процессе ранга 0 в исходном коммуникаторе MPI_COMM_WORLD), а также найти максимальный элемент из тех максимумов, которые были получены во второй группе второго интеркоммуникатора, переслать и вывести его в первом процессе первой группы этого интеркоммуникатора (т. е. в процессе ранга 1 в исходном коммуникаторе MPI_COMM_WORLD).

MPI8Inter19*. В главном процессе дан массив A из $2K$ целых чисел, где K — количество процессов. Используя один вызов функции MPI_Comm_spawn с первым параметром "ptprj.exe", создать K новых процессов. С помощью коллективной функции MPI_Intercomm_merge для ин-

теркоммуникатора, связанного с созданными процессами, создать новый интракоммуникатор, содержащий как исходные, так и новые процессы (параметр high функции MPI_Intercomm_merge необходимо задать таким образом, чтобы в созданном интракоммуникаторе вначале располагались исходные, а затем созданные процессы). Используя коллективную функцию MPI_Scatter для нового интракоммуникатора, переслать по одному элементу массива A из главного процесса во все процессы приложения (как исходные, так и созданные) в порядке возрастания их рангов в новом интракоммуникаторе. В исходных процессах вывести полученные числа, в новых процессах отобразить полученные числа в разделе отладки, используя в этих процессах функцию Show. Затем, используя коллективную функцию MPI_Reduce в этом же интракоммуникаторе, переслать в исходный процесс ранга 1 сумму всех чисел и вывести ее в этом процессе.

MPI8Inter20*. Количество процессов K не делится на 4. В каждом процессе дано целое число A. Используя один вызов функции MPI_Comm_spawn с первым параметром "ptprj.exe", создать такое количество новых процессов (1, 2 или 3), чтобы общее число процессов K_0 в приложении стало кратным 4. В каждом из созданных процессов задать значение числа A, равное $-R-1$, где R — ранг созданного процесса. Используя функцию MPI_Intercomm_merge для интеркоммуникатора, связанного с созданными процессами, создать новый интракоммуникатор, содержащий как исходные, так и новые процессы (параметр high функции MPI_Intercomm_merge необходимо задать таким образом, чтобы в созданном интракоммуникаторе вначале располагались исходные, а затем новые процессы). Используя функцию MPI_Cart_create для созданного интракоммуникатора, определить для всех процессов декартову топологию в виде двумерной решетки размера $(K_0/4) \times 4$, являющейся периодической по второму измерению (порядок нумерации процессов оставить прежним). Используя функцию MPI_Cart_coords, определить координаты X, Y каждого процесса в данной решетке; в исходных процессах вывести эти координаты, в новых процессах отобразить координаты в разделе отладки с помощью функций Show, снабдив их комментариями «X = » и «Y = ». Выполнить циклический сдвиг значений A в каждом столбце созданной решетки с шагом -1 (т. е. в направлении убывания рангов процессов), используя для этого функции MPI_Cart_shift и MPI_Sendrecv_replace. В исходных процессах вывести полученные значения A, в новых процессах отобразить их в разделе отладки с помощью функции Show, снабдив комментарием «A = ».

MPI8Inter21**. В каждом процессе дано вещественное число: в главном процессе это число обозначается буквой A, в подчиненных процес-

сах — буквой *B*. Используя два вызова функции MPI_Comm_spawn с первым параметром "ptprj.exe", создать две группы новых процессов: первая группа (*группа-сервер*) должна содержать один процесс, вторая группа (*группа-клиент*) должна содержать $K - 1$ процесс, где K — количество исходных процессов. Переслать в процесс группы-сервера число *A* из главного процесса, а в процессы группы-клиента — числа *B* из подчиненных процессов (в порядке возрастания рангов процессов) и отобразить полученные числа в разделе отладки, используя в каждом новом процессе функцию Show. С помощью функций MPI_Open_port, MPI_Publish_name и MPI_Comm_accept на стороне сервера и функций MPI_Lookup_name и MPI_Comm_connect на стороне клиента установить связь между двумя новыми группами процессов, создав для них новый интеркоммуникатор. Используя функции MPI_Send и MPI_Recv для этого интеркоммуникатора, получить в каждом процессе группы-клиента число *A* из процесса группы-сервера и найти сумму этого числа и числа *B*, полученного ранее из исходных подчиненных процессов. Отобразить найденную сумму $A + B$ в разделе отладки, используя в каждом процессе группы-клиента функцию Show, а также переслать эту сумму в соответствующий исходный подчиненный процесс и вывести полученную сумму в этом процессе (сумма, найденная в процессе ранга *R* группы-клиента, должна быть переслана в исходный процесс ранга $R + 1$).

Примечание. В программе необходимо обеспечить такую последовательность действий, чтобы вызов функции MPI_Lookup_name в процессах клиента гарантированно выполнялся *после* вызова функции MPI_Publish_name в процессе сервера. Можно, например, использовать функцию MPI_Barrier для исходных процессов и процесса сервера, вызвав ее в процессе сервера только после завершения вызова функции MPI_Publish_name, и вызвать функцию MPI_Comm_spawn для создания группы-клиента уже после выхода из функции MPI_Barrier в исходных процессах.

MPI8Inter22**. В каждом процессе дано целое число *N*, которое может принимать три значения: 0, 1 и *K* ($K > 1$). Известно, что имеется ровно один процесс со значением $N = 1$ и ровно *K* процессов со значением $N = K$. В тех процессах, в которых число *N* не равно 0, также дано целое число *A*. Используя функцию MPI_Comm_split, разбить исходный коммуникатор MPI_COMM_WORLD на два, в каждый из которых входят процессы с одинаковым ненулевым значением *N*. Используя один вызов функции MPI_Comm_spawn для каждого из созданных коммуникаторов, создать два набора новых процессов; число новых процессов должно совпадать с количеством процессов в соответствующем коммуникаторе (таким образом, в одной новой группе будет содержать-

ся 1 процесс, в другой — *K* процессов). Переслать в каждый из новых процессов число *A* из того исходного процесса, ранг которого в созданном коммуникаторе совпадает с рангом нового процесса. Отобразить полученные числа в разделе отладки, используя в каждом новом процессе функцию Show. Считая новую группу, состоящую из одного процесса, *группой-сервером*, а новую группу из *K* процессов — *группой-клиентом*, установить между ними связь с помощью функций MPI_Open_port, MPI_Publish_name и MPI_Comm_accept на стороне сервера и MPI_Lookup_name и MPI_Comm_connect на стороне клиента. Используя коллективную функцию MPI_Gather для интеркоммуникатора, связывающего группу-клиент и группу-сервер, переслать все числа *A* из процессов группы-клиента в процесс группы-сервера и отобразить полученные числа в разделе отладки, используя несколько вызовов функции Show в процессе группы-сервера. Кроме того, переслать эти числа с помощью функций MPI_Send и MPI_Recv из процесса группы-сервера в тот исходный процесс, который создал группу-сервер, и вывести полученные числа в этом исходном процессе.

Примечание. В программе необходимо обеспечить такую последовательность действий, чтобы вызов функции MPI_Lookup_name в процессах клиента гарантированно выполнялся *после* вызова функции MPI_Publish_name в процессе сервера. Можно, например, поступить следующим образом. Для пересылки числа *A* в процесс группы-сервера применить функцию MPI_Ssend, вызвав после нее функцию MPI_Barrier для исходного коммуникатора MPI_COMM_WORLD и организовав прием этого числа в процессе группы-сервера только после выполнения функции MPI_Publish_name. В остальных процессах исходного коммуникатора MPI_COMM_WORLD следует вначале вызвать функцию MPI_Barrier, а затем организовать пересылку чисел *A* в созданные процессы группы-клиента. Таким образом, любой из процессов группы-клиента получит число *A* только тогда, когда процесс группы-сервера уже выполнит вызов функции MPI_Publish_name.

2.9. Параллельные матричные алгоритмы

Все числа в заданиях являются целыми. Матрицы должны вводиться и выводиться по строкам. Файлы с элементами матриц также содержат их по строкам.

Количество процессов в заданиях, связанных с ленточными алгоритмами (MPI9Matr2–MPI9Matr20), не превосходит 5, а в заданиях, связанных с блочными алгоритмами (MPI9Matr21–MPI9Matr44), не превосходит 16.

Для хранения имени файла достаточно использовать массив char[12], а для его пересылки из главного процесса в подчиненные — функцию MPI_Bcast с параметром типа MPI_CHAR.

В заготовках для каждого задания уже содержатся описания целочисленных переменных для хранения числовых данных, упоминаемых в заданиях (в частности, размеров матриц), указателей на массивы для хранения самих матриц, а также переменных типа MPI_Datatype и MPI_Comm. Эти переменные следует использовать во всех функциях, которые требуется реализовать при выполнении заданий. Все имена переменных соответствуют обозначениям, применяемым в формулировках заданий. Для массивов, связанных с полосами или блоками матриц, используются имена a, b, c, t; для массивов, связанных с исходными матрицами A, B и их результирующим произведением C, используются имена с подчеркиванием: a_, b_, c_.

Задания, в которых используется файловый ввод-вывод (MPI9Matr8–MPI9Matr10, MPI9Matr18–MPI9Matr20, MPI9Matr29–MPI9Matr31, MPI9Matr42–MPI9Matr44), требуют подключения библиотеки MPI-2 (система MPICH2 1.3). Для выполнения остальных заданий данной группы можно использовать любую версию MPI (MPICH 1.3.5 или MPICH2 1.3).

Перед выполнением заданий из данного пункта следует ознакомиться с примерами решения задач MPI9Matr1, MPI9Matr2, MPI9Matr24, MPI9Matr19, приведенными в п. 1.4.2–1.4.5.

2.9.1. Непараллельный алгоритм умножения матриц

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.

Примечание. Решение данной задачи приведено в п. 1.4.2.

2.9.2. Ленточный алгоритм 1 (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 осуществлять 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.

Примечание. Решение данной задачи приведено в п. 1.4.3.

MPI9Matr3. В каждом процессе даны числа N_A, P, N_B, Q, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr2). Реализовать первый шаг ленточного алгоритма перемножения матриц, выполнив перемножение элементов, содержащихся в полосах A_R и B_R каждого процесса, после чего организовать циклическую пересылку каждой полосы B_R в процесс предыдущего ранга (из процесса 1 в процесс 0, из процесса 2 в про-

цесс 1, …, из процесса 0 в процесс $K - 1$, где K — количество процессов).

Циклическую пересылку полос B_R выполнять с помощью функции MPI_Sendrecv_replace, используя для определения рангов процесса-отправителя и процесса-получателя операцию % взятия остатка от деления.

Оформить эти действия в виде функции Matr1Calc (без параметров). Вывести новое содержимое полос C_R и B_R в каждом процессе (ввод и вывод данных выполнять во внешней функции Solve).

Указание. В результате перемножения полос A_R и B_R каждый элемент полосы C_R будет содержать *часть* слагаемых, входящих в итоговое произведение AB; при этом будут использованы все элементы полосы B_R и часть элементов полосы A_R (в частности, на первом шаге в процессе 0 будут использованы N_B первых столбцов полосы A_0, а в процессе $K - 1$ — N_B последних столбцов полосы A_{K-1}).

MPI9Matr4. В каждом процессе даны числа N_A, P, N_B, Q, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr2). Модифицировать функцию Matr1Calc, реализованную в предыдущем задании, таким образом, чтобы она обеспечивала выполнение любого шага алгоритма ленточного умножения.

Для этого добавить к ней параметр step, определяющий номер шага (изменяется от 0 до $K - 1$, где K — количество процессов), и использовать значение этого параметра в той части алгоритма, которая связана с пересчетом элементов полосы C_R (действия, связанные с циклической пересылкой полос B_R, от значения параметра step не зависят).

Используя два вызова модифицированной функции Matr1Calc с параметрами 0 и 1, выполнить два начальных шага ленточного алгоритма и вывести в каждом процессе новое содержимое полос C_R и B_R (ввод и вывод данных выполнять во внешней функции Solve).

Указание. Номер шага step определяет, какая часть элементов полосы A_R будет использована при очередном пересчете элементов полосы C_R (следует обратить внимание на то, что эти части перебираются циклически).

MPI9Matr5*. В каждом процессе даны числа N_A, P, N_B, Q, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr2). Кроме того, в каждом процессе дано одно и то же число L, лежащее в диапазоне от 3 до K (K — количество процессов) и определяющее требуемое число шагов ленточного алгоритма.

Вызывая в цикле по параметру I ($I = 0, \ldots, L - 1$) функцию Matr1Calc(I), разработанную в предыдущем задании, выполнить L начальных шагов ленточного алгоритма и вывести в каждом процессе новое содержимое полос C_R и B_R (ввод и вывод данных выполнять во внешней функции Solve).

Примечание. Если значение L равно K, то полосы C_R будут содержать соответствующие фрагменты итогового матричного произведения AB.

MPI9Matr6*. В главном процессе дано число M — количество строк результирующего матричного произведения. Кроме того, в каждом процессе даны числа N_A, Q, а также одномерные массивы, заполненные полосами матрицы C (размера $N_A \times Q$), которые были получены в результате выполнения K шагов ленточного алгоритма перемножения матриц (см. MPI9Matr5). Переслать все полосы C_R в главный процесс и вывести в нем полученную матрицу C (размера $M \times Q$).

Для хранения результирующей матрицы C в главном процессе использовать одномерный массив, достаточный для хранения матрицы размера $(N_A \cdot K) \times Q$; для пересылки данных в этот массив использовать коллективную функцию MPI_Gather.

Оформить эти действия в виде функции Matr1GatherData (без параметров). Ввод данных выполнять во внешней функции Solve, вывод полученной матрицы включить в функцию Matr1GatherData.

MPI9Matr7**. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно (таким образом, исходные данные совпадают с исходными данными для задания MPI9Matr2).

Последовательно вызывая разработанные в заданиях MPI9Matr2–MPI9Matr6 функции Matr1ScatterData, Matr1Calc (в цикле) и Matr1GatherData, получить и вывести в главном процессе матрицу C, равную произведению исходных матриц A и B.

После каждого вызова функции Matr1Calc дополнительно выводить в каждом процессе текущее содержимое полосы C_R. Перед использованием в данном задании следует модифицировать разработанную в MPI9Matr4 функцию Matr1Calc таким образом, чтобы при значении параметра step, равном $K - 1$, не выполнялась пересылка полос B_R.

MPI9Matr8*. В главном процессе даны числа M, P, Q, а также имена двух файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно. Модифицировать начальный этап ленточного алгоритма перемножения матриц (см. MPI9Matr2) таким образом, чтобы каждый процесс считывал соответствующие полосы матриц A и B непосредственно из исходных файлов, используя коллективные функ-

ции MPI_File_seek и MPI_File_read_all (новый вид файловых данных создавать не требуется).

Для пересылки размеров матриц и имен файлов использовать коллективную функцию MPI_Bcast.

Оформить все действия в виде функции Matr1ScatterFile (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 Matr1ScatterFile вывести в каждом процессе полученные данные (числа N_A, P, N_B, Q и полосы матриц A, B, C). Ввод исходных данных осуществлять в функции Matr1ScatterFile, вывод результатов выполнять во внешней функции Solve.

Примечание. Для некоторых полос часть элементов (последние строки) или даже вся полоса не должна считываться из исходных файлов и будет оставаться нулевой. Однако эта ситуация не требует специальной обработки, так как при достижении конца файла функция MPI_File_read_all автоматически прекращает считывание данных, не генерируя никакого сообщения об ошибке.

MPI9Matr9*. В каждом процессе даны числа N_A, Q, а также одномерные массивы, заполненные полосами C_R (размера $N_A \times Q$), полученными в результате выполнения K шагов ленточного алгоритма перемножения матриц (см. MPI9Matr5). Кроме того, в главном процессе дано число M — количество строк результирующего матричного произведения и имя файла для хранения этого произведения.

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 использовать коллективные функции MPI_File_seek и 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 Matr1GatherFile (считывание всех исходных данных, кроме имени файла, должно осуществляться во внешней функции Solve).

Указание. При записи данных в результирующий файл необходимо учитывать, что некоторые 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).

MPI9Matr10**. В главном процессе даны числа M, P, Q, а также имена трех файлов: вначале даются имена двух существующих файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответст-

венно, а затем имя файла для хранения результирующего матричного произведения $C = AB$.

Последовательно вызывая разработанные в заданиях MPI9Matr8, MPI9Matr5 и MPI9Matr9 функции Matr1ScatterFile, Matr1Calc (в цикле) и Matr1GatherFile, получить в результирующем файле произведение исходных матриц A и B, найденное с помощью первого варианта ленточного алгоритма.

После каждого вызова функции Matr1Calc дополнительно выводить в каждом процессе текущее значение элемента c[step], где c — одномерный массив, содержащий полосу C_R, а step — номер шага алгоритма (0, 1, …, $K - 1$); таким образом, на первом шаге алгоритма следует вывести элемент c[0], на втором шаге — элемент c[1], и т. д.

2.9.3. Ленточный алгоритм 2 (горизонтальные и вертикальные полосы)

MPI9Matr11. В каждом процессе даны числа P и Q; кроме того, в главном процессе дана матрица B размера $P \times Q$. Известно, что число Q кратно количеству процессов K. Прочесть в главном процессе матрицу B в одномерный массив размера $P \cdot Q$ и определить новый тип MPI_BAND_B, содержащий вертикальную полосу матрицы B шириной $N_B = Q/K$ столбцов.

При определении типа MPI_BAND_B использовать функции MPI_Type_vector и MPI_Type_commit, оформив это определение в виде функции Matr2CreateTypeBand(p, n, q, t), где целочисленные параметры p, n, q являются входными, а параметр t типа MPI_Datatype является выходным; при этом параметры p и n определяют размер вертикальной полосы (число ее строк и столбцов), а параметр q — число столбцов матрицы, из которой извлекается эта полоса.

Используя тип MPI_BAND_B, переслать в каждый процесс (включая главный) соответствующую полосу матрицы B, перебирая полосы в порядке возрастания рангов процессов-получателей. Пересылку выполнять с помощью функций MPI_Send и MPI_Recv; полосы хранить в одномерных массивах размера $P \cdot N_B$. Вывести в каждом процессе полученную полосу.

Примечание. В реализации MPICH2 версии 1.3 с помощью функции MPI_Send нельзя выполнить пересылку данных в тот же процесс, из которого данные посылаются (происходит зависание программы). Для пересылки полосы в главный процесс можно использовать функцию MPI_Sendrecv; можно также заполнить соответствующую полосу в главном процессе, не прибегая к средствам библиотеки MPI.

MPI9Matr12*. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно. Во втором варианте ленточно-

го алгоритма перемножения матриц первая матрица (A) разбивается на K горизонтальных полос, а вторая (B) — на K вертикальных 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 столбцов; числа N_A и N_B вычисляются по формулам $N_A = \text{ceil}(M/K)$, $N_B = \text{ceil}(Q/K)$, где операция «/» означает вещественное деление, а функция ceil выполняет округление с избытком. Если матрица содержит недостаточно строк (или столбцов) для заполнения последней полосы, то полоса дополняется нулевыми строками (столбцами).

Сохранить исходные матрицы, дополненные при необходимости нулевыми строками или столбцами, 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 $P \times N_B$. Кроме того, создать в каждом процессе полосу C_R для хранения фрагмента матричного произведения $C = AB$, который будет вычисляться в этом процессе; каждая полоса C_R имеет размер $(N_A \cdot K) \times N_B$ 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 матрицы A использовать коллективную функцию MPI_Scatter, для пересылки полос матрицы B использовать функции MPI_Send и MPI_Recv, а также вспомогательный тип MPI_BAND_B, созданный с помощью функции Matr2CreateTypeBand (см. предыдущее задание и примечание к нему).

Оформить все описанные действия в виде функции Matr2ScatterData (без параметров), в результате вызова которой каждый процесс получает значения N_A, P, N_B, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C. После вызова функции Matr2ScatterData вывести в каждом процессе полученные данные (числа N_A, P, N_B и полосы матриц A, B, C). Ввод исходных данных осуществлять в функции Matr2ScatterData, вывод результатов выполнять во внешней функции Solve.

Указания. (1) При считывании матрицы B в главном процессе следует учитывать, что предназначенный для ее хранения массив может содержать элементы, соответствующие дополнительным нулевым столбцам.

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

MPI9Matr13. В каждом процессе даны числа N_A, P, N_B, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr12). Реализовать первый шаг ленточного алгоритма перемножения матриц, выполнив перемножение элементов, содержащихся в полосах A_R и B_R каждого процесса, после чего организовать циклическую пересылку каждой полосы A_R в процесс предыдущего ранга (из процесса 1 в процесс 0, из процесса 2 в процесс 1, …, из процесса 0 в процесс $K - 1$, где K — количество процессов).

Циклическую пересылку полос A_R выполнять с помощью функции MPI_Sendrecv_replace, используя для определения рангов процесса-отправителя и процесса-получателя операцию % взятия остатка от деления.

Оформить эти действия в виде функции Matr2Calc (без параметров). Вывести новое содержимое полос C_R и A_R в каждом процессе (ввод и вывод данных выполнять во внешней функции Solve).

Указание. Поскольку в данном варианте ленточного алгоритма полосы A_R содержат полные строки матрицы A, а полосы B_R — полные столбцы матрицы B, в результате их перемножения уже на первом шаге алгоритма полоса C_R будет содержать часть элементов итогового матричного произведения (прочие элементы полосы останутся нулевыми). Расположение найденных элементов в полосе C_R зависит от ранга процесса (в частности, на первом шаге в процессе 0 будут заполнены N_A первых строк полосы C_0, а в процессе $K - 1$ — N_A последних строк полосы C_{K-1}).

MPI9Matr14. В каждом процессе даны числа N_A, P, N_B, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr12). Модифицировать функцию Matr2Calc, реализованную в предыдущем задании, таким образом, чтобы она обеспечивала выполнение любого шага алгоритма ленточного умножения.

Для этого добавить к ней параметр step, определяющий номер шага (изменяется от 0 до $K - 1$, где K — количество процессов) и использовать значение этого параметра в той части алгоритма, которая связана с пересчетом элементов полосы C_R (действия, связанные с циклической пересылкой полос A_R, от значения параметра step не зависят).

Используя два вызова модифицированной функции Matr2Calc с параметрами 0 и 1, выполнить два начальных шага ленточного алгоритма

и вывести в каждом процессе новое содержимое полос C_R и A_R (ввод и вывод данных выполнять во внешней функции Solve).

Указание. Номер шага step определяет, какие наборы строк полосы C_R будут вычислены на данном этапе алгоритма (следует обратить внимание на то, что эти наборы строк перебираются циклически).

MPI9Matr15*. В каждом процессе даны числа N_A, P, N_B, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr12). Кроме того, в каждом процессе дано одно и то же число L, лежащее в диапазоне от 3 до K (K — количество процессов) и определяющее требуемое число шагов ленточного алгоритма.

Вызывая в цикле по параметру I ($I = 0, \ldots, L - 1$) функцию Matr2Calc(I), разработанную в предыдущем задании, выполнить L начальных шагов ленточного алгоритма и вывести в каждом процессе новое содержимое полос C_R и A_R (ввод и вывод данных выполнять во внешней функции Solve).

Примечание. Если значение L равно K, то полосы C_R будут содержать соответствующие фрагменты итогового матричного произведения AB.

MPI9Matr16*. В главном процессе даны числа M и Q — количество строк и столбцов результирующего матричного произведения. Кроме того, в каждом процессе даны числа N_A, N_B, а также одномерные массивы, заполненные полосами матрицы C (размера $(N_A \cdot K) \times N_B$), которые были получены в результате выполнения K шагов ленточного алгоритма перемножения матриц (см. MPI9Matr15). Переслать все полосы C_R в главный процесс и вывести в нем полученную матрицу C (размера $M \times Q$).

Для хранения результирующей матрицы C в главном процессе использовать одномерный массив, достаточный для хранения матрицы размера $(N_A \cdot K) \times (N_B \cdot K)$. Для пересылки полос C_R использовать функции MPI_Send и MPI_Recv, а также вспомогательный тип MPI_BAND_C, созданный с помощью функции Matr2CreateTypeBand (см. задание MPI9Matr11 и примечание к нему).

Оформить эти действия в виде функции Matr2GatherData (без параметров). Ввод данных выполнять во внешней функции Solve, вывод полученной матрицы включить в функцию Matr2GatherData.

Указание. При выводе матрицы C в главном процессе следует учитывать, что предназначенный для ее хранения массив может содержать элементы, соответствующие дополнительным нулевым столбцам.

MPI9Matr17**. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно (таким образом, исходные данные совпадают с исходными данными для задания MPI9Matr12).

Последовательно вызывая разработанные в заданиях MPI9Matr12–MPI9Matr16 функции Matr2ScatterData, Matr2Calc (в цикле) и Matr2GatherData, получить и вывести в главном процессе матрицу C, равную произведению исходных матриц A и B.

После каждого вызова функции Matr2Calc дополнительно выводить в каждом процессе текущее содержимое полосы C_R.

Перед использованием в данном задании следует модифицировать разработанную в MPI9Matr14 функцию Matr2Calc таким образом, чтобы при значении параметра step, равном $K - 1$, не выполнялась пересылка полос A_R.

MPI9Matr18*. В главном процессе даны числа M, P, Q, а также имена двух файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно. Известно, что число Q кратно количеству процессов K. Модифицировать начальный этап ленточного алгоритма перемножения матриц (см. MPI9Matr12) таким образом, чтобы каждый процесс считывал соответствующие полосы матриц A и B непосредственно из исходных файлов.

Для пересылки размеров матриц и имен файлов использовать коллективную функцию MPI_Bcast. Для считывания горизонтальных полос матрицы A использовать коллективные функции MPI_File_seek и MPI_File_read_all; для считывания вертикальных полос матрицы B задать соответствующий вид данных, используя функцию MPI_File_set_view и новый файловый тип MPI_BAND_B, определенный с помощью функции Matr2CreateTypeBand (см. MPI9Matr11), после чего также использовать функцию MPI_File_read_all.

Оформить все действия в виде функции Matr2ScatterFile (без параметров), в результате вызова которой каждый процесс получает значения N_A, P, N_B, а также одномерные массивы, заполненные соответствующими полосами матриц A, B, C. После вызова функции Matr2ScatterFile вывести в каждом процессе полученные данные (числа N_A, P, N_B и полосы матриц A, B, C). Ввод исходных данных осуществлять в функции Matr2ScatterFile, вывод результатов выполнять во внешней функции Solve.

Примечание. Дополнительное условие о кратности значения Q числу K позволяет организовать считывание полос B_R с использованием одинакового файлового типа во всех процессах.

При отсутствии этого условия потребовалось бы применять специальные типы, обеспечивающие корректное считывание из файла и запись в массив «укороченных» полос матрицы B в последних процессах

(кроме того, в этом случае потребовалось бы переслать каждому процессу значение Q, необходимое для правильного определения типов для «укороченных» полос).

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 произведения 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).

Примечание. Решение данной задачи приведено в п. 1.4.5.

MPI9Matr20**. В главном процессе даны числа M, P, Q, а также имена трех файлов: вначале даются имена двух существующих файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно, а затем имя файла для хранения результирующего матричного произведения $C = AB$. Дополнительно известно, что число Q кратно количеству процессов K.

Последовательно вызывая разработанные в заданиях MPI9Matr18, MPI9Matr15 и MPI9Matr19 функции Matr2ScatterFile, Matr2Calc (в цикле) и Matr2GatherFile, получить в результирующем файле произведение

исходных матриц A и B, найденное с помощью второго варианта ленточного алгоритма.

После каждого вызова функции Matr2Calc дополнительно выводить в каждом процессе текущее значение элемента c[step], где c — одномерный массив, содержащий полосу C_R, а step — номер шага алгоритма (0, 1, …, $K - 1$); таким образом, на первом шаге алгоритма следует вывести элемент c[0], на втором шаге — элемент c[1], и т. д.

2.9.4. Блочный алгоритм Кэннона

MPI9Matr21. В каждом процессе даны числа M и P; кроме того, в главном процессе дана матрица A размера $M \times P$. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$, а числа M и P кратны числу K_0. Прочесть в главном процессе матрицу A в одномерный массив размера $M \cdot P$ и определить новый тип MPI_BLOCK_A, содержащий блок матрицы A размера $M_0 \times P_0$, где $M_0 = M/K_0$, $P_0 = P/K_0$.

При определении типа MPI_BLOCK_A использовать функции MPI_Type_vector и MPI_Type_commit, оформив это определение в виде функции Matr3CreateTypeBlock(m0, p0, p, t), где целочисленные параметры m0, p0, p являются входными, а параметр t типа MPI_Datatype является выходным; при этом параметры m0 и p0 определяют размеры блока, а параметр p — число столбцов матрицы, из которой извлекается этот блок.

Используя тип MPI_BLOCK_A, переслать в каждый процесс (включая главный) соответствующий блок матрицы A, перебирая блоки по строкам и пересылая их в процессы в порядке возрастания их рангов (первый блок пересылается в процесс 0, следующий за ним блок в этой же строке пересылается в процесс 1, и т. д.). Пересылку выполнять с помощью функций MPI_Send и MPI_Recv; блоки хранить в одномерных массивах размера $M_0 \cdot P_0$. Вывести в каждом процессе полученный блок.

Примечание. В реализации MPICH2 версии 1.3 с помощью функции MPI_Send нельзя выполнить пересылку данных в тот же процесс, из которого данные посылаются (происходит зависание программы). Для пересылки блока в главный процесс можно использовать функцию MPI_Sendrecv; можно также заполнить соответствующий блок в главном процессе, не прибегая к средствам библиотеки MPI.

MPI9Matr22. В каждом процессе даны числа M_0 и P_0, а также матрица A размера $M_0 \times P_0$. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$. Прочесть в каждом процессе матрицу A в одномерный массив размера $M_0 \cdot P_0$ и определить на множестве исходных процессов коммуникатор MPI_COMM_GRID, задающий декар-

тову топологию двумерной квадратной циклической решетки порядка K_0 (исходный порядок нумерации процессов сохраняется).

При определении коммуникатора MPI_COMM_GRID использовать функцию MPI_Cart_create, оформив это определение в виде функции Matr3CreateCommGrid(comm) с выходным параметром comm типа MPI_Comm.

Вывести в каждом процессе координаты (I_0, J_0) этого процесса в созданной топологии, используя функцию MPI_Cart_coords.

Для каждой строки I_0 полученной решетки осуществить циклический сдвиг матриц A на I_0 позиций влево (т. е. в направлении убывания рангов процессов), используя функции MPI_Cart_shift и MPI_Sendrecv_replace. Вывести в каждом процессе матрицу, полученную в результате сдвига.

MPI9Matr23*. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$. В блочных алгоритмах перемножения матриц исходные матрицы разбиваются на K блоков, образуя квадратные блочные матрицы порядка K_0 (в дальнейшем блоки распределяются по процессам и используются для вычисления в каждом процессе части итогового матричного произведения).

Блок для матрицы A имеет размер $M_0 \times P_0$, блок для матрицы B имеет размер $P_0 \times Q_0$; числа M_0, P_0, Q_0 вычисляются по формулам $M_0 = \text{ceil}(M/K_0)$, $P_0 = \text{ceil}(P/K_0)$, $Q_0 = \text{ceil}(Q/K_0)$, где операция «/» означает вещественное деление, а функция ceil выполняет округление с избытком. Если матрица содержит недостаточно строк (или столбцов) для заполнения последних блоков, то блоки дополняются нулевыми строками (столбцами).

Сохранить исходные матрицы A и B, дополненные при необходимости нулевыми строками и столбцами до размеров $(M_0 \cdot K_0) \times (P_0 \cdot K_0)$ и $(P_0 \cdot K_0) \times (Q_0 \cdot K_0)$, в одномерных массивах в главном процессе, после чего организовать пересылку блоков из этих массивов во все процессы, перебирая блоки по строкам и пересылая их в процессы в порядке возрастания их рангов (процесс ранга R получит блоки A_R и B_R, $R = 0, \ldots, K - 1$). Кроме того, создать в каждом процессе блок C_R для хранения фрагмента матричного произведения $C = AB$, которое будет вычисляться в этом процессе; каждый блок C_R имеет размер $M_0 \times Q_0$ и заполняется нулевыми элементами.

Блоки, как и исходные матрицы, должны храниться по строкам в одномерных массивах соответствующего размера. Для пересылки размеров матриц использовать коллективную функцию MPI_Bcast, для пересылки блоков матриц A и B использовать функции MPI_Send и MPI_Recv, а также вспомогательные типы MPI_BLOCK_A и MPI_BLOCK_B,

созданные с помощью функции Matr3CreateTypeBlock (см. задание MPI9Matr21 и примечание к нему).

Оформить все описанные действия в виде функции Matr3ScatterData (без параметров), в результате вызова которой каждый процесс получает значения M_0, P_0, Q_0, а также одномерные массивы, заполненные соответствующими блоками матриц A, B, C. После вызова функции Matr3ScatterData вывести в каждом процессе полученные данные (числа M_0, P_0, Q_0 и блоки матриц A, B, C). Ввод исходных данных осуществлять в функции Matr3ScatterData, вывод результатов выполнять во внешней функции Solve.

Указания. (1) При считывании матриц A и B в главном процессе следует учитывать, что предназначенные для ее хранения массивы могут содержать элементы, соответствующие дополнительным нулевым столбцам.

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

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).

Примечание. Решение данной задачи приведено в п. 1.4.4.

MPI9Matr25. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, заполненные соответствующими блоками матриц A, B и C, причем известно, что блоки C_R являются нулевыми, а для блоков A_R и B_R уже выполнено их начальное перераспределение в соответствии с алгоритмом Кэннона (см. предыдущее задание). Реализовать один шаг алгоритма Кэннона перемножения матриц, выполнив перемножение элементов, содержащихся в блоках A_R и B_R каждого процесса, после чего организовать для каждой строки декартовой решетки циклический сдвиг блоков A_R на 1 позицию влево (т. е. в направлении убывания рангов процессов), а для каждого столбца решетки циклический сдвиг блоков B_R на 1 позицию вверх (т. е. также в направлении убывания рангов процессов).

Для циклической пересылки данных использовать коммуникатор MPI_COMM_GRID, создав его с помощью функции Matr3CreateCommGrid (см. задание MPI9Matr22), и функции MPI_Cart_shift и MPI_Sendrecv_replace (ср. с MPI9Matr22).

Оформить эти действия в виде функции Matr3Calc (без параметров). Вывести новое содержимое блоков C_R, A_R и B_R в каждом процессе (ввод и вывод данных выполнять во внешней функции Solve).

Примечание. Особенность алгоритма Кэннона состоит в том, что действия на каждом его шаге не зависят от номера шага.

MPI9Matr26*. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, заполненные соответствующими блоками матриц A, B и C, причем известно, что блоки C_R являются нулевыми, а для блоков A_R и B_R уже выполнено их начальное перераспределение в соответствии с алгоритмом Кэннона (см. задание MPI9Matr24). Кроме того, в каждом процессе дано одно и то же число L, лежащее в диапазоне от 2 до K_0 (где $K_0 \cdot K_0$ равно количеству процессов) и определяющее требуемое число шагов алгоритма Кэннона.

Вызывая в цикле функцию Matr3Calc, разработанную в предыдущем задании, выполнить L начальных шагов алгоритма Кэннона и вывести в каждом процессе новое содержимое блоков C_R, A_R и B_R (ввод и вывод данных выполнять во внешней функции Solve).

Примечание. Если значение L равно K_0, то блоки C_R будут содержать соответствующие фрагменты итогового матричного произведения AB.

MPI9Matr27*. В главном процессе даны числа M и Q — число строк и столбцов результирующего матричного произведения. Кроме того, в каждом процессе даны числа M_0, Q_0, а также одномерные массивы, заполненные блоками матрицы C (размера $M_0 \times Q_0$), которые были получены в результате выполнения K_0 шагов блочного алгоритма Кэннона перемножения матриц (см. MPI9Matr26). Переслать все бло-

ки C_R в главный процесс и вывести в нем полученную матрицу C (размера $M \times Q$).

Для хранения результирующей матрицы C в главном процессе использовать одномерный массив, достаточный для хранения матрицы размера $(M_0 \cdot K_0) \times (Q_0 \cdot K_0)$. Для пересылки блоков C_R использовать функции MPI_Send и MPI_Recv, а также вспомогательный тип MPI_BLOCK_C, созданный с помощью функции Matr3CreateTypeBlock (см. задание MPI9Matr21 и примечание к нему).

Оформить эти действия в виде функции Matr3GatherData (без параметров). Ввод данных выполнять во внешней функции Solve, вывод полученной матрицы включить в функцию Matr3GatherData.

Указание. При выводе матрицы C в главном процессе следует учитывать, что предназначенный для ее хранения массив может содержать элементы, соответствующие дополнительным нулевым столбцам.

MPI9Matr28**. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно (таким образом, исходные данные совпадают с исходными данными для задания MPI9Matr23).

Последовательно вызывая разработанные в заданиях MPI9Matr23–MPI9Matr27 функции Matr3ScatterData, Matr3Init, Matr3Calc (в цикле) и Matr3GatherData, получить и вывести в главном процессе матрицу C, равную произведению исходных матриц A и B.

После каждого вызова функции Matr3Calc дополнительно выводить в каждом процессе текущее содержимое блока C_R.

Для того чтобы коммуникатор MPI_COMM_GRID, используемый в функциях Matr3Init и Matr3Calc, не создавался несколько раз, модифицировать функцию Matr3CreateCommGrid таким образом, чтобы при ее вызове для уже определенного коммуникатора (не равного MPI_COMM_NULL) она не выполняла никаких действий.

Перед использованием в данном задании следует модифицировать разработанную в MPI9Matr25 функцию Matr3Calc, добавив в нее параметр step, равный номеру шага (step = 0, …, $K_0 - 1$), и изменив ее таким образом, чтобы при значении параметра step, равном $K_0 - 1$, не выполнялась пересылка блоков A_R и B_R.

MPI9Matr29*. В главном процессе даны числа M, P, Q, а также имена двух файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно. Модифицировать этап получения блоков для блочного алгоритма Кэннона перемножения матриц (см. MPI9Matr23) таким образом, чтобы каждый процесс считывал соответствующие блоки матриц A и B непосредственно из исходных файлов. В данном случае всем процессам требуется переслать не только размеры блоков M_0, P_0, Q_0, но и размеры исходных матриц M, P, Q, которые требуются для правильного определения позиций блоков в исходных файлах.

Для пересылки размеров матриц и имен файлов использовать коллективную функцию MPI_Bcast. Для считывания блоков использовать локальные функции MPI_File_read_at, вызывая отдельную функцию для считывания каждой строки блока (новый вид файловых данных создавать не требуется).

Оформить все действия в виде функции Matr3ScatterFile (без параметров), в результате вызова которой каждый процесс получает значения M, P, Q, M_0, P_0, Q_0, а также одномерные массивы, заполненные соответствующими блоками матриц A, B, C. После вызова функции Matr3ScatterFile вывести в каждом процессе полученные данные (числа M, P, Q, M_0, P_0, Q_0 и блоки матриц A, B, C). Ввод исходных данных осуществлять в функции Matr3ScatterFile, вывод результатов выполнять во внешней функции Solve.

Указание. При чтении файловых данных следует учитывать, что для некоторых блоков часть элементов (последние строки и/или столбцы) не должна считываться из исходных файлов и будет оставаться нулевой. Для определения фактического размера считываемого блока (числа строк и числа столбцов) потребуется использовать размеры исходных матриц и координаты блока (I_0, J_0) в квадратной декартовой решетке порядка K_0, которые легко определить по рангу процесса R: $I_0 = R/K_0$, $J_0 = R\%K_0$.

Примечание. В то время как значения P и Q необходимы для обеспечения правильного считывания файловых блоков, значение M можно не использовать, поскольку попытка чтения данных за концом файла просто игнорируется, не приводя к ошибке. Однако значение M потребуется на завершающем этапе алгоритма (см. следующее задание), поэтому его тоже необходимо переслать всем процессам.

MPI9Matr30*. В каждом процессе даны числа M, Q, M_0, Q_0, а также одномерные массивы, заполненные блоками C_R (размера $M_0 \times Q_0$), полученными в результате выполнения K_0 шагов блочного алгоритма Кэннона перемножения матриц (см. MPI9Matr25). Кроме того, в главном процессе дано имя файла для хранения результирующего матричного произведения.

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

Для записи данных в файл использовать локальные функции MPI_File_write_at, вызывая отдельную функцию для записи каждой строки блока (новый вид файловых данных создавать не требуется).

Оформить считывание имени файла, его пересылку, а также все действия по записи данных в файл в виде функции Matr3GatherFile (считы-

вание всех исходных данных, кроме имени файла, должно осуществляться во внешней функции Solve).

Указание. При записи файловых данных следует учитывать, что для некоторых блоков C_R часть элементов (последние строки и/или столбцы, заполненные нулями) не должна записываться в результирующий файл. См. также указание и примечание к предыдущему заданию.

MPI9Matr31**. В главном процессе даны числа M, P, Q, а также имена трех файлов: вначале даются имена двух существующих файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно, а затем имя файла для хранения результирующего матричного произведения $C = AB$.

Последовательно вызывая разработанные в заданиях MPI9Matr29, MPI9Matr24, MPI9Matr25 и MPI9Matr30 функции Matr3ScatterFile, Matr3Init, Matr3Calc (в цикле) и Matr3GatherFile, получить в результирующем файле произведение исходных матриц A и B, найденное с помощью блочного алгоритма Кэннона.

После каждого вызова функции Matr3Calc дополнительно выводить в каждом процессе текущее значение элемента c[step], где c — одномерный массив, содержащий блок C_R, а step — номер шага алгоритма (0, 1, …, $K_0 - 1$); таким образом, на первом шаге алгоритма следует вывести элемент c[0], на втором шаге — элемент c[1], и т. д.

2.9.5. Блочный алгоритм Фокса

MPI9Matr32. В каждом процессе даны числа M и P; кроме того, в главном процессе дана матрица A размера $M \times P$. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$, а числа M и P кратны числу K_0. Прочесть в главном процессе матрицу A в одномерный массив размера $M \cdot P$ и определить новый тип MPI_BLOCK_A, содержащий блок матрицы A размера $M_0 \times P_0$, где $M_0 = M/K_0$, $P_0 = P/K_0$.

При определении типа MPI_BLOCK_A использовать функции MPI_Type_vector и MPI_Type_commit, оформив это определение в виде функции Matr4CreateTypeBlock(m0, p0, p, t), где целочисленные параметры m0, p0, p являются входными, а параметр t типа MPI_Datatype является выходным; при этом параметры m0 и p0 определяют размеры блока, а параметр p — число столбцов матрицы, из которой извлекается этот блок.

Используя тип MPI_BLOCK_A, переслать в каждый процесс (включая главный) соответствующий блок матрицы A, перебирая блоки по строкам и пересылая их в процессы в порядке возрастания их рангов (первый блок пересылается в процесс 0, следующий за ним блок в этой же строке пересылается в процесс 1, и т. д.). Пересылку выполнять с помощью коллективной функции MPI_Alltoallw; блоки хранить

в одномерных массивах размера $M_0 \cdot P_0$. Вывести в каждом процессе полученный блок.

Указания. (1) При выполнении задания с применением библиотеки MPI-1 вместо функции MPI_Alltoallw следует использовать функции MPI_Send и MPI_Recv.

(2) Функция MPI_Alltoallw, введенная в MPI-2, является единственной коллективной функцией, которая позволяет указывать смещения для пересылаемых данных в *байтах*, а не в элементах. Это дает возможность использовать ее совместно со сложными типами данных для реализации любых вариантов коллективных обменов (в данном случае требуется реализовать вариант вида Scatter).

Следует учитывать, что при подобном варианте рассылки все параметры-массивы функции MPI_Alltoallw, связанные с посылаемыми данными, необходимо по-разному определять в главном и подчиненных процессах. В частности, массив scounts (определяющий количество посылаемых данных) должен содержать значения 0 во всех подчиненных процессах и значения 1 в главном процессе (посылаемые элементы имеют тип MPI_BLOCK_A).

В то же время, массивы, связанные с принимаемыми данными, будут определяться одинаковым образом во всех процессах; в частности, в массиве rcounts (определяющем количество принимаемых данных) элемент с индексом 0 должен быть равен $M_0 \cdot P_0$, а все остальные элементы должны быть равны 0 (принимаемые элементы имеют тип MPI_INT).

Необходимо обратить особое внимание на правильное определение элементов в массиве sdispls смещений для посылаемых данных в главном процессе (в подчиненных процессах этот массив достаточно обнулить).

MPI9Matr33. В каждом процессе даны числа M_0 и P_0, а также матрица A размера $M_0 \times P_0$. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$. Прочесть в каждом процессе матрицу A в одномерный массив размера $M_0 \cdot P_0$ и определить на множестве исходных процессов коммуникатор MPI_COMM_GRID, задающий декартову топологию двумерной квадратной циклической решетки порядка K_0 (исходный порядок нумерации процессов сохраняется).

При определении коммуникатора MPI_COMM_GRID использовать функцию MPI_Cart_create, оформив это определение в виде функции Matr4CreateCommGrid(comm) с выходным параметром comm типа MPI_Comm. Вывести в каждом процессе координаты (I_0, J_0) этого процесса в созданной топологии, используя функцию MPI_Cart_coords.

На основе коммуникатора MPI_COMM_GRID создать набор коммуникаторов MPI_COMM_ROW, связанных со строками исходной двумерной

решетки. Для определения коммуникаторов MPI_COMM_ROW использовать функцию MPI_Cart_sub, оформив это определение в виде вспомогательной функции Matr4CreateCommRow(grid, row) с входным параметром grid (коммуникатором, связанным с исходной двумерной решеткой) и выходным параметром row (оба параметра типа MPI_Comm). Вывести в каждом процессе его ранг R_0 для коммуникатора MPI_COMM_ROW (этот ранг должен совпадать со значением J_0).

Кроме того, для каждой строки I_0 полученной решетки осуществить пересылку матрицы A из столбца I_0 во все процессы этой же строки, используя коллективную функцию MPI_Bcast для коммуникатора MPI_COMM_ROW и сохранив результат во вспомогательной матрице T того же размера, что и матрица A (перед пересылкой необходимо скопировать в матрицу T рассылающего процесса элементы пересылаемой матрицы A). Вывести в каждом процессе полученную матрицу T.

MPI9Matr34. В каждом процессе даны числа P_0 и Q_0, а также матрица B размера $P_0 \times Q_0$. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$. Прочесть в каждом процессе матрицу B в одномерный массив размера $P_0 \cdot Q_0$ и определить на множестве исходных процессов коммуникатор MPI_COMM_GRID, задающий декартову топологию двумерной квадратной циклической решетки порядка K_0.

Для определения коммуникатора MPI_COMM_GRID использовать функцию Matr4CreateCommGrid (см. задание MPI9Matr33). Вывести в каждом процессе его координаты (I_0, J_0) в созданной топологии, используя функцию MPI_Cart_coords.

На основе коммуникатора MPI_COMM_GRID создать набор коммуникаторов MPI_COMM_COL, связанных со столбцами исходной двумерной решетки. Для определения коммуникаторов MPI_COMM_COL использовать функцию MPI_Cart_sub, оформив это определение в виде функции Matr4CreateCommCol(grid, col) с входным параметром grid (коммуникатором, связанным с исходной двумерной решеткой) и выходным параметром col (оба параметра типа MPI_Comm). Вывести в каждом процессе его ранг R_0 для коммуникатора MPI_COMM_COL (этот ранг должен совпадать со значением I_0).

Кроме того, для каждого столбца J_0 полученной решетки осуществить циклический сдвиг матриц B на 1 позицию вверх (т. е. в направлении убывания рангов процессов), используя функции MPI_Sendrecv_replace для коммуникатора MPI_COMM_COL (при определении рангов процесса-отправителя и процесса-получателя использовать операцию % взятия остатка от деления). Вывести в каждом процессе матрицу, полученную в результате сдвига.

MPI9Matr35*. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно. Известно, что количество процессов K является полным квадратом: $K = K_0 \cdot K_0$. В блочных алгоритмах перемножения матриц исходные матрицы разбиваются на K блоков, образуя квадратные блочные матрицы порядка K_0 (в дальнейшем блоки распределяются по процессам и используются для вычисления в каждом процессе части итогового матричного произведения).

Блок для матрицы A имеет размер $M_0 \times P_0$, блок для матрицы B имеет размер $P_0 \times Q_0$; числа M_0, P_0, Q_0 вычисляются по формулам $M_0 = \text{ceil}(M/K_0)$, $P_0 = \text{ceil}(P/K_0)$, $Q_0 = \text{ceil}(Q/K_0)$, где операция «/» означает вещественное деление, а функция ceil выполняет округление с избытком. Если матрица содержит недостаточно строк (или столбцов) для заполнения последних блоков, то блоки дополняются нулевыми строками (столбцами).

Сохранить исходные матрицы A и B, дополненные при необходимости нулевыми строками и столбцами до размеров $(M_0 \cdot K_0) \times (P_0 \cdot K_0)$ и $(P_0 \cdot K_0) \times (Q_0 \cdot K_0)$, в одномерных массивах в главном процессе, после чего организовать пересылку блоков из этих массивов во все процессы, перебирая блоки по строкам и пересылая их в процессы в порядке возрастания их рангов (процесс ранга R получит блоки A_R и B_R, $R = 0, \ldots, K - 1$). Кроме того, создать в каждом процессе два блока, заполненных нулевыми элементами: блок C_R размера $M_0 \times Q_0$ для хранения фрагмента матричного произведения $C = AB$, которое будет вычисляться в этом процессе, и вспомогательный блок T_R того же размера $M_0 \times P_0$, что и блок A_R.

Блоки, как и исходные матрицы, должны храниться по строкам в одномерных массивах соответствующего размера. Для пересылки размеров матриц использовать коллективную функцию MPI_Bcast, для пересылки блоков матриц A и B использовать коллективную функцию MPI_Alltoallw, а также вспомогательные типы MPI_BLOCK_A и MPI_BLOCK_B, созданные с помощью функции Matr4CreateTypeBlock (см. задание MPI9Matr32, а также указания к нему).

Оформить все описанные действия в виде функции Matr4ScatterData (без параметров), в результате вызова которой каждый процесс получает значения M_0, P_0, Q_0, а также одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R. После вызова функции Matr4ScatterData вывести в каждом процессе полученные данные (числа M_0, P_0, Q_0 и блоки A_R, B_R, C_R, T_R). Ввод исходных данных осуществлять в функции Matr4ScatterData, вывод данных выполнять во внешней функции Solve.

Указания. (1) При считывании матриц A и B в главном процессе следует учитывать, что предназначенные для ее хранения массивы могут

содержать элементы, соответствующие дополнительным нулевым столбцам.

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

MPI9Matr36. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr35). Каждый шаг блочного алгоритма Фокса перемножения матриц состоит из двух этапов.

На первом этапе первого шага для каждой строки I_0 квадратной решетки процессов порядка K_0 ($I_0 = 0, \ldots, K_0 - 1$, где $K_0 \cdot K_0$ равно количеству процессов) выполняется пересылка блока A_R из процесса, расположенного в строке I_0 и столбце с тем же номером I_0, во все процессы этой же строки (с сохранением пересланного блока в блоке T_R), после чего полученный в результате этой пересылки блок T_R умножается на блок B_R из этого процесса, и результат добавляется к блоку C_R.

Реализовать первый этап первого шага алгоритма Фокса. Для пересылки блоков A_R использовать функцию MPI_Bcast для коммуникатора MPI_COMM_ROW, создав этот коммуникатор с помощью функции Matr4CreateCommRow (см. задание MPI9Matr33, в котором описывается аналогичная пересылка данных).

Оформить все описанные действия в виде функции Matr4Calc1 (без параметров). Вывести новое содержимое блоков T_R и C_R в каждом процессе (ввод и вывод данных выполнять во внешней функции Solve).

MPI9Matr37. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr35).

Реализовать второй этап первого шага алгоритма Фокса, который состоит в циклическом сдвиге блоков B_R для каждого столбца декартовой решетки на 1 позицию вверх (т. е. в направлении убывания рангов процессов).

Для циклической пересылки блоков B_R использовать функцию MPI_Sendrecv_replace для коммуникатора MPI_COMM_COL, создав этот коммуникатор с помощью функции Matr4CreateCommCol (см. задание MPI9Matr34, в котором описывается аналогичная пересылка данных).

Оформить эти действия в виде функции Matr4Calc2 (без параметров). Вывести новое содержимое блока B_R в каждом процессе (ввод и вывод данных выполнять во внешней функции Solve).

MPI9Matr38. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R (таким

образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr35).

Модифицировать функцию Matr4Calc1, реализованную в задании MPI9Matr36, таким образом, чтобы она обеспечивала выполнение первого этапа на любом шаге алгоритма Фокса. Для этого добавить к ней параметр step, определяющий номер шага (изменяется от 0 до $K_0 - 1$, где K_0 — порядок декартовой решетки процессов), и учесть значение этого шага при рассылке блоков A_R: на шаге step для каждой строки I_0 декартовой решетки должна выполняться рассылка блока A_R из процесса, расположенного в столбце с номером $(I_0 + \text{step})\%K_0$ (действия, связанные с пересчетом блоков C_R, от номера шага не зависят).

Выполнить два начальных шага алгоритма Фокса, последовательно вызвав функции Matr4Calc1(0), Matr4Calc2() (обеспечивающую второй этап шага алгоритма — см. MPI9Matr37) и Matr4Calc1(1), и вывести в каждом процессе новое содержимое блоков T_R, B_R и C_R (ввод и вывод данных выполнять во внешней функции Solve).

MPI9Matr39*. В каждом процессе даны числа M_0, P_0, Q_0, а также одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R (таким образом, исходные данные совпадают с результатами, полученными в задании MPI9Matr35). Кроме того, в каждом процессе дано одно и то же число L, лежащее в диапазоне от 3 до K_0 и определяющее требуемое число шагов алгоритма Фокса.

Выполнить L начальных шагов алгоритма Фокса, вызвав функции, разработанные в заданиях MPI9Matr38 и MPI9Matr37, в следующей последовательности: Matr4Calc1(0), Matr4Calc2(), Matr4Calc1(1), Matr4Calc2(), …, Matr4Calc1($L - 1$). Вывести в каждом процессе новое содержимое блоков T_R, B_R и C_R (ввод и вывод данных выполнять во внешней функции Solve).

Примечание. Если значение L равно K_0, то блоки C_R будут содержать соответствующие фрагменты матричного произведения AB. Обратите внимание на то, что второй этап (связанный с вызовом функции Matr4Calc2) на последнем шаге алгоритма выполнять не требуется.

MPI9Matr40*. В главном процессе даны числа M и Q — число строк и столбцов результирующего матричного произведения. Кроме того, в каждом процессе даны числа M_0, Q_0, а также одномерные массивы, заполненные блоками матрицы C (размера $M_0 \times Q_0$), которые были получены в результате выполнения K_0 шагов блочного алгоритма Фокса перемножения матриц (см. MPI9Matr39).

Переслать все блоки C_R в главный процесс и вывести в нем полученную матрицу C (размера $M \times Q$). Для хранения результирующей матрицы C в главном процессе использовать одномерный массив, достаточный для хранения матрицы размера $(M_0 \cdot K_0) \times (Q_0 \cdot K_0)$. Для пере-

сылки блоков C_R использовать коллективную функцию MPI_Alltoallw, а также вспомогательный тип MPI_BLOCK_C, созданный с помощью функции Matr4CreateTypeBlock (см. задание MPI9Matr32, а также указания к нему).

Оформить эти действия в виде функции Matr4GatherData (без параметров). Ввод данных выполнять во внешней функции Solve, вывод полученной матрицы включить в функцию Matr4GatherData.

Указание. При выводе матрицы C в главном процессе следует учитывать, что предназначенный для ее хранения массив может содержать элементы, соответствующие дополнительным нулевым столбцам.

MPI9Matr41**. В главном процессе даны числа M, P, Q и матрицы A и B размера $M \times P$ и $P \times Q$ соответственно (таким образом, исходные данные совпадают с исходными данными для задания MPI9Matr35).

Последовательно вызывая разработанные в заданиях MPI9Matr35–MPI9Matr40 функции Matr4ScatterData, Matr4Calc1, Matr4Calc2 и Matr4GatherData, получить и вывести в главном процессе матрицу C, равную произведению исходных матриц A и B. Функции Matr4Calc1 и Matr4Calc2 должны вызываться в цикле, причем количество вызовов функции Matr4Calc2 должно быть на 1 меньше количества вызовов функции Matr4Calc1.

После каждого вызова функции Matr4Calc1 дополнительно выводить в каждом процессе текущее содержимое блока C_R.

Для того чтобы вспомогательные коммуникаторы MPI_COMM_GRID, MPI_COMM_ROW и MPI_COMM_COL, используемые в функциях Matr4Calc1 и Matr4Calc2, не создавались несколько раз, модифицировать функции Matr4CreateCommGrid, Matr4CreateCommRow, Matr4CreateCommCol (см. MPI9Matr33 и MPI9Matr34) таким образом, чтобы при их вызове для уже определенного коммуникатора (не равного MPI_COMM_NULL) они не выполняли никаких действий.

MPI9Matr42*. В главном процессе даны числа M, P, Q, а также имена двух файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно. Дополнительно известно, что числа M, P и Q кратны порядку K_0 квадратной решетки процессов.

Модифицировать этап формирования блоков для блочного алгоритма Фокса перемножения матриц (см. MPI9Matr35) таким образом, чтобы каждый процесс считывал соответствующие блоки матриц A и B непосредственно из исходных файлов.

Для пересылки размеров матриц и имен файлов использовать коллективную функцию MPI_Bcast. Для считывания блоков задать соответствующий вид данных, используя функцию MPI_File_set_view и типы MPI_BLOCK_A и MPI_BLOCK_B, определенные с помощью функции

Matr4CreateTypeBlock (см. MPI9Matr32) после чего использовать коллективную функцию MPI_File_read_all.

Оформить все действия в виде функции Matr4ScatterFile (без параметров), в результате вызова которой каждый процесс получает значения M_0, P_0, Q_0, одномерные массивы, содержащие элементы блоков A_R, B_R, C_R и T_R (блоки C_R и T_R должны быть нулевыми). После вызова функции Matr4ScatterFile вывести в каждом процессе полученные данные (числа M_0, P_0, Q_0 и блоки A_R, B_R, C_R, T_R). Ввод исходных данных осуществлять в функции Matr4ScatterFile, вывод результатов выполнять во внешней функции Solve.

Примечание. Дополнительное условие о кратности чисел M, P, Q числу K_0 означает, что блоки, полученные из матриц A и B, не требуется дополнять нулевыми строками и/или столбцами, и поэтому для чтения из файлов блоков матриц A и B в любые процессы можно использовать одинаковые файловые типы MPI_BLOCK_A и MPI_BLOCK_B. При отсутствии этого условия потребовалось бы применять специальные типы, обеспечивающие корректное считывание из файла и запись в массив «укороченных» блоков матриц A и B (кроме того, в этом случае потребовалось бы переслать каждому процессу значения P и Q, необходимые для правильного определения типов для «укороченных» блоков).

MPI9Matr43*. В каждом процессе даны числа M_0, Q_0, а также одномерные массивы, заполненные блоками C_R (размера $M_0 \times Q_0$), полученными в результате выполнения K_0 шагов блочного алгоритма Фокса перемножения матриц (см. MPI9Matr39). Кроме того, в главном процессе дано имя файла для хранения результирующего матричного произведения. Дополнительно известно, что число строк M и число столбцов Q матричного произведения кратны порядку K_0 квадратной решетки процессов (таким образом, $M = M_0 \cdot K_0$, $Q = Q_0 \cdot K_0$).

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

Для записи блоков задать соответствующий вид данных, используя функцию MPI_File_set_view и файловый тип MPI_BLOCK_C, определенный с помощью функции Matr4CreateTypeBlock (см. MPI9Matr32), после чего использовать коллективную функцию MPI_File_write_all.

Оформить считывание имени файла, его пересылку, а также все действия по записи данных в файл в виде функции Matr4GatherFile (считывание всех исходных данных, кроме имени файла, должно осуществляться во внешней функции Solve).

Примечание. Дополнительное условие о кратности чисел M и Q числу K_0 означает, что блоки C_R не содержат «лишних» нулевых строк и/или столбцов, и поэтому для их записи в файл из любых процессов можно использовать одинаковые файловые типы MPI_BLOCK_C.

MPI9Matr44**. В главном процессе даны числа M, P, Q, а также имена трех файлов: вначале даются имена двух существующих файлов, содержащих элементы матриц A и B размера $M \times P$ и $P \times Q$ соответственно, а затем имя файла для хранения результирующего матричного произведения $C = AB$. Дополнительно известно, что числа M, P и Q кратны порядку K_0 квадратной решетки процессов.

Последовательно вызывая разработанные в заданиях MPI9Matr42, MPI9Matr38, MPI9Matr37 и MPI9Matr43 функции Matr4ScatterFile, Matr4Calc1, Matr4Calc2 и Matr4GatherFile, получить в результирующем файле произведение исходных матриц A и B, найденное с помощью блочного алгоритма Фокса. Функции Matr4Calc1 и Matr4Calc2 должны вызываться в цикле, причем количество вызовов функции Matr4Calc2 должно быть на 1 меньше количества вызовов функции Matr4Calc1.

После каждого вызова функции Matr4Calc1 дополнительно выводить в каждом процессе текущее значение элемента c[step], где c — одномерный массив, содержащий блок C_R, а step — номер шага алгоритма (0, 1, …, $K_0 - 1$); таким образом, на первом шаге алгоритма следует вывести элемент c[0], на втором шаге — элемент c[1], и т. д.

Раздел 3. Разработка заданий по параллельному программированию

3.1. Особенности функционирования задачника в параллельном режиме

3.1.1. Варианты запуска откомпилированной программы

Для успешной разработки новых заданий по параллельному программированию необходимо четкое понимание механизма функционирования задачника в параллельном режиме.

Прежде всего, следует учитывать, что откомпилированная учебная программа (exe-файл), предназначенная для выполнения задания по параллельному программированию, может выступать в различных качествах.

Программа, запускаемая непосредственно из среды программирования, всегда выполняется как обычная непараллельная программа. В частности, в непараллельном режиме всегда выполняется демонстрационный запуск программы (напомним, что для этого в конце имени задания, указываемого в качестве параметра функции Task, надо добавить символ «?» или «#»). При демонстрационном запуске не требуется подключения системы MPICH2, поэтому ознакомиться с содержанием задач по параллельному программированию можно и на тех компьютерах, на которых система MPICH2 не установлена.

Если запуск не является демонстрационным, то программа, запущенная из среды программирования, выступает в роли «загрузчика» своего параллельного варианта. Это позволяет максимально упростить действия учащегося при тестировании параллельной программы, поскольку требует от него только запуска разработанной программы непосредственно из среды программирования. Все прочие действия, связанные с нахождением файла mpiexec.exe системы MPICH2 и его запуском с требуемыми параметрами командной строки, выполняются задачником автоматически. Для ускорения процесса тестирования разработанного алгоритма программа-загрузчик последовательно выполняет *несколько* запусков параллельного варианта. Кроме того, при каждом запуске параллельного варианта программа-загрузчик обеспечивает случайный выбор количества процессов.

3.1.2. Действия программы в режиме загрузчика

При своем запуске программа-загрузчик проверяет наличие программы mpiexec.exe системы MPICH2, создает в рабочем каталоге учащегося пакетный файл pt_run.bat, содержащий команду для запуска программы mpiexec.exe, и запускает созданный пакетный файл. Если программа mpiexec не найдена или пакетный файл нельзя создать или запустить, то выводится сообщение об ошибке, и выполнение программы-загрузчика немедленно завершается. Если же пакетный файл успешно запущен, то программа-загрузчик ожидает его завершения, после чего выполняет необходимые завершающие действия и заканчивает работу.

Разумеется, программу mpiexec.exe с требуемыми параметрами можно было бы запускать и непосредственно из программы-загрузчика, однако вариант с использованием пакетного файла обладает рядом преимуществ. Прежде всего, запуск пакетного файла приводит к отображению на экране стандартного консольного окна, в котором можно вывести информацию о том, что выполнен запуск программы в параллельном режиме, а также отобразить ту командную строку, которая обеспечивает запуск программы mpiexec.exe. Это делает более наглядным для учащегося способ запуска параллельной программы; кроме того, из параметров командной строки учащийся может легко определить, сколько процессов параллельной программы запущено. Дополнительно появляется возможность стандартным образом (нажатием [Ctrl]+[C] или [Ctrl]+[Break]) *прервать* выполнение пакетного файла в случае зависания параллельной программы, что при выполнении заданий может происходить достаточно часто. Описание действий, требующихся для прерывания выполнения пакетного файла, и действий, связанных с настройкой параметров выполнения учебных программ в параллельным режиме, также приводится в его консольном окне (рис. 59).

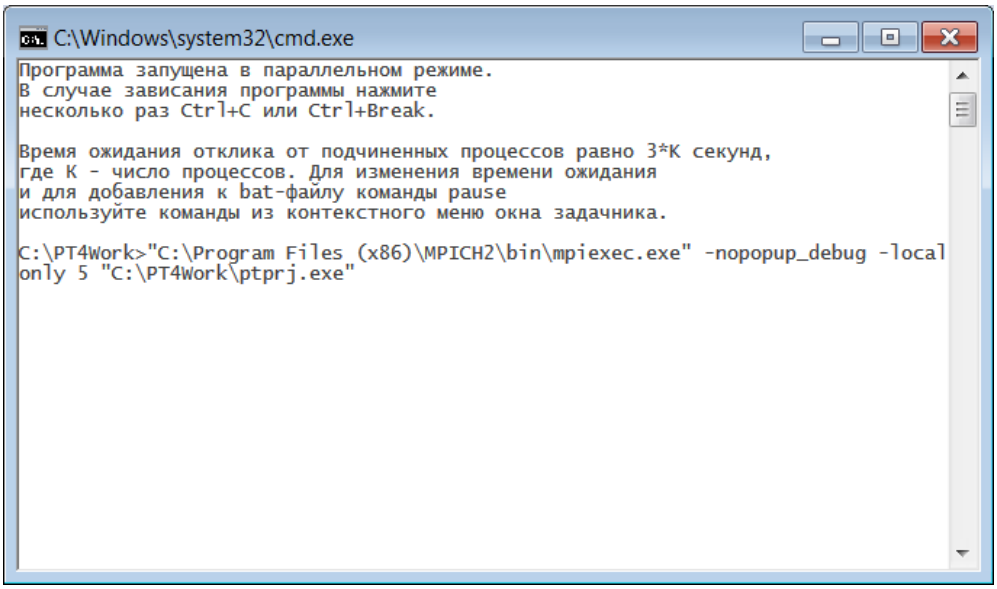

Рис. 59. Консольное окно с информацией о запуске параллельной программы

Перед завершением своей работы программа-загрузчик вначале удаляет из каталога учащегося вспомогательный пакетный файл, а затем выгружает из памяти все процессы, связанные с задачником (за исключением своего собственного процесса). Действия по выгрузке процессов необходимы в ситуации, когда параллельная программа зависла и выполнение пакетного файла было прервано явным образом. Если не удалить зависшие процессы из памяти, то станет невозможной перекомпиляция исправленной программы, так как exe-файл программы нельзя будет перезаписать из-за его блокировки зависшими процессами.

Осталось обсудить следующий вопрос: как запущенная программа определяет, что она является загрузчиком? Казалось бы, для этого достаточно с помощью средств MPI определить количество процессов, связанных с запущенным экземпляром программы: если число процессов равно 1, значит, программа запущена не в параллельном режиме и, следовательно, является загрузчиком, тогда как в противном случае программа выступает в роли одного из процессов параллельной программы и не должна выполнять действия загрузчика. Однако при таком способе проверки появляется возможность запустить программу в параллельном режиме *без предварительного запуска программы-загрузчика*. Действительно, ничто не мешает учащемуся создать явным образом пакетный файл с командой запуска программы mpiexec.exe и, запустив этот пакетный файл, выполнить программу, решающую задачу, в параллельном режиме. Такой способ запуска является нежелательным прежде всего потому, что в этой ситуации количество процессов параллельной программы задает сам учащийся и, таким образом, задачник лишается возможности протестировать предложенное решение при различном числе процессов. Кроме того, в задании обычно предполагается, что число параллельных процессов лежит в определенном диапазоне, выход за границы которого может привести к ошибке при инициализации этого задания.

Следовало реализовать такой способ проверки, который делал бы невозможным выполнение параллельного варианта программы без предварительного запуска программы-загрузчика. Для этого проще всего было сделать так, чтобы экземпляр программы, выступающий в роли загрузчика, при своем запуске оставлял какую-либо *метку* в каталоге учащегося, а любой экземпляр запущенной программы проверял наличие этой метки: если метка обнаружена, то этот экземпляр считается одним из процессов параллельной программы, в противном случае он считается загрузчиком. При завершении работы параллельного варианта программы метка удаляется. Ясно, что метка должна быть защищена от «подделки», поэтому в качестве такой метки нельзя использовать, например, факт наличия в каталоге пакетного файла pt_run.bat.

В задачнике реализован достаточно надежный способ установки метки программой-загрузчиком. Благодаря ему учащийся *не сможет* запустить параллельный вариант своей программы с помощью «стороннего» пакетного файла: при таком запуске первый из запущенных параллельных процессов не обнаружит в каталоге метки загрузчика, поэтому он «решит», что сам является загрузчиком, поместит свою метку в каталог, создаст свой пакетный файл и запустит его на выполнение. При этом остальные параллельные процессы из «стороннего» пакетного файла даже не будут запущены, так как для их запуска программой mpiexec.exe необходимо, чтобы первый запущенный ею процесс в течение определенного времени после запуска перешел в параллельный режим, а в случае программы, выполняющей роль загрузчика, этого не произойдет.

Заканчивая обсуждение вопросов, связанных с программой-загрузчиком, отметим, что механизм отслеживания зависших процессов, связанных с задачником, реализован таким образом, что удаляются только те процессы, которые запущены непосредственно для данного параллельного варианта программы. Таким образом, при запуске параллельного приложения допустимо оставлять открытым окно задачника, связанное с модулем PT4Demo, или окна, относящиеся к заданиям, выполняемым в данное время в других программных средах.

3.1.3. Действия программы в параллельном режиме и обработка ошибок

Теперь опишем действия учебной программы, запущенной в параллельном режиме. При работе программы в параллельном режиме выполняется несколько экземпляров этой программы (процессов), причем процесс ранга 0 играет особую роль. Как и ранее, будем называть этот процесс главным, а остальные процессы подчиненными. Именно в главном процессе происходит инициализация задания, т. е. генерируются наборы исходных и контрольных данных для всех процессов параллельной программы. После генерации этих наборов выполняется пересылка в каждый подчиненный процесс тех исходных и контрольных данных, которые определены для этого процесса (для связывания элементов данных с конкретным процессом при инициализации задания необходимо использовать функцию SetProcess — см. ее описание в п.).

После завершения пересылки данных каждый процесс выполняет свою часть алгоритма решения задачи, разработанного учащимся, в частности, вводит исходные данные, подготовленные задачником, и выводит результирующие данные (вывод данных осуществляется в специальный выходной буфер в оперативной памяти). Затем все подчиненные процессы пересылают главному процессу сведения о результатах своей работы, включающие все данные из выходного буфера и некоторую дополнитель-

ную информацию, в том числе сообщения об ошибках, возникших в данном подчиненном процессе, а также сведения о количестве введенных и выведенных данных. Пересылка этих сведений выполняется с использованием особого коммуникатора, определенного в задачнике и недоступного учебной программе. Главный процесс анализирует полученную информацию и отображает ее в окне задачника. После закрытия окна задачника выполнение параллельной программы завершается.

Следует отметить, что главный процесс не только принимает от подчиненных процессов сигналы об ошибках, но и самостоятельно выявляет те ошибки, которые сами подчиненные процессы выявить не могут. В частности, он распознает ошибки, связанные с полным отсутствием операций ввода-вывода в некотором подчиненном процессе (в ситуации, когда в других процессах ввод или вывод выполнялся). Сам подчиненный процесс, в котором не выполнялись действия по вводу-выводу, «предполагает», что производится ознакомительный запуск программы, и поэтому в подобной ситуации не посылает главному процессу сигнал об ошибке.

При выполнении подчиненных процессов могут возникнуть ошибки, приводящие к их зависанию или аварийному завершению. В этом случае главный процесс не сможет получить от этих процессов информацию о результатах их работы. Если в течение определенного времени (зависящего от общего числа подчиненных процессов) главный процесс не получит информацию от некоторых подчиненных процессов, то он выведет в окне задачника сообщение «*MPI error. Процессы ... не отвечают*» (где на месте многоточия указываются ранги зависших процессов). В этой ситуации после закрытия окна задачника, как правило, требуется явным образом прервать выполнение параллельного приложения, нажав несколько раз комбинацию клавиш [Ctrl]+[C] или [Ctrl]+[Break]. Время ожидания отклика от подчиненных процессов можно настраивать с помощью команд контекстного меню окна задачника; эта возможность добавлена для того, чтобы при выполнении сложных заданий на компьютерах с низким быстродействием время ожидания было достаточным для завершения в подчиненных процессах всех необходимых операций.

Прочие ошибки, связанные с функциями MPI, не приводят к выводу особых сообщений в информационном разделе окна задачника, однако сведения о них выводятся в разделе отладки, который в этом случае автоматически отображается на экране (для перехвата ошибок системы MPI и отмены их стандартной обработки, приводящей по умолчанию к немедленному завершению всех процессов параллельного приложения, в задачнике определяется специальный обработчик ошибок).

Все «обычные» ошибки времени выполнения (не связанные с библиотекой MPI) подчиненные процессы также перехватывают, после чего пересылают информацию о них главному процессу, который в этом случае вы-

водит сообщение «*Run-time error*». Кроме того, подчиненные процессы могут распознавать различные ошибки, связанные с неверным вводом-выводом: ввод недостаточного числа исходных данных, попытка ввода лишних исходных данных или исходных данных неверного типа; вывод недостаточного числа результирующих данных, попытка вывода лишних результирующих данных или результирующих данных неверного типа.

Сведения, полученные от подчиненных процессов, анализируются в главном процессе в порядке убывания рангов (от процесса $K-1$ к процессу 1, где K — общее число процессов). Если в подчиненных процессах возникли ошибки различного типа, то в информационном разделе окна задачника выводится информация о той ошибке, которая была обнаружена первой. При этом не только выводится описание ошибки, но и перечисляются ранги подчиненных процессов, в которых обнаружена данная ошибка.

Дополнительная информация о каждой ошибке, обнаруженной в подчиненных процессах, приводится в разделе отладки окна задачника, причем рядом с описанием ошибки указывается ранг процесса, в котором она произошла.

В случае обнаружения ошибок в подчиненных процессах главный процесс (процесс ранга 0) не проверяется на наличие ошибок; об этом тоже выводится сообщение в разделе отладки.

Если ошибки в подчиненных процессах не обнаружены, то выполняется завершающая проверка правильности решения в главном процессе. При наличии ошибок ввода-вывода, ошибок времени выполнения или в случае несоответствия результирующих данных контрольным данным выводится соответствующее сообщение; если же ошибки отсутствуют, а результаты совпадают с требуемыми контрольными значениями, то данное испытание программы считается успешным. Задание считается выполненным после определенного числа успешных испытаний, проведенных подряд (для всех заданий, входящих в задачник PT for MPI-2, количество тестовых испытаний равно пяти). Как уже отмечалось ранее, все тестовые испытания выполняются при однократном запуске учебной программы (это еще один вид действий, которые берет на себя экземпляр программы, выступающий в роли загрузчика).

Возможна ситуация, когда из-за ошибок в реализации параллельного алгоритма произойдет зависание *главного процесса*. Главный процесс можно считать зависшим, если в течение 20–30 с после отображения консольного окна с информацией о запуске программы в параллельном режиме на экране не появится окно задачника. В этом случае необходимо явным образом прервать выполнение параллельной программы, нажав несколько раз [Ctrl]+[C] или [Ctrl]+[Break]. В файл результатов в такой ситуации заносится информация о том, что выполнение задания было прервано.

Завершая данный пункт, приведем схему, иллюстрирующую этапы выполнения учебного задания по параллельному программированию (рис. 60):

- **этап 1** – запуск программы из интегрированной среды
- **этап 2** (выполняется несколько раз) – определение числа процессов *K* для очередного тестового испытания, создание и запуск пакетного файла;
- **этап 3** – запуск программы в параллельном режиме на локальном компьютере;
- **этап 4** – взаимодействие параллельных процессов при выполнении задания;
- **этап 5** – отображение результатов (в случае успешного прохождения всех испытаний или при обнаружении ошибки);
- **этап 6** (выполняется для каждого тестового испытания) – завершающие действия, в частности, выгрузка зависших процессов.

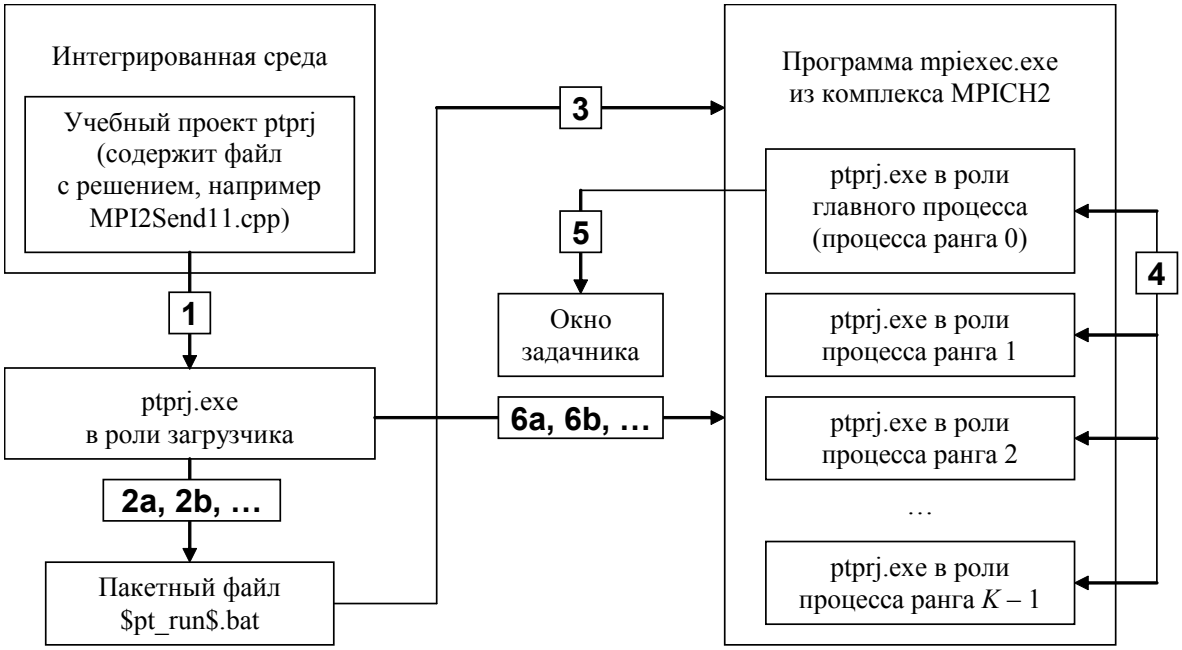

Рис. 60. Схема выполнения задания по параллельному программированию с применением электронного задачника Programming Taskbook for MPI-2

3.2. Дополнительные возможности конструктора учебных заданий

В состав задачника PT for MPI-2 включен вариант конструктора учебных заданий PT4TaskMaker, позволяющий разрабатывать новые группы заданий на языке C++ в среде программирования Microsoft Visual Studio. Следует заметить, что имеются варианты конструктора учебных заданий

и для других языков программирования (Delphi Pascal, PascalABC.NET и C#); все они входят в состав программного комплекса Teacher Pack for Programming Taskbook и позволяют разрабатывать новые задания, которые можно выполнять на любом из языков, поддерживаемых задачником. Однако, поскольку сам задачник PT for MPI-2 ориентирован на использование языка C++, представляется естественным разрабатывать задания на этом же языке.

Компоненты конструктора PT4TaskMaker содержатся в подкаталоге taskmaker каталога PTforMPI2, который, в свою очередь, находится в системном каталоге электронного задачника Programming Taskbook (обычно это каталог c:\Program Files (x86)\PT4).

Все средства конструктора для языка C++ реализованы в виде набора функций, определенных в файле pt4taskmaker.cpp; описания этих функций содержатся в заголовочном файле pt4taskmaker.h.

В п. 3.2.1 и 3.2.2 описываются те возможности конструктора PT4TaskMaker, которые связаны с разработкой заданий по параллельному программированию. Подробное описание конструктора PT4TaskMaker содержится в посвященном ему разделе сайта задачника Programming Taskbook (http://ptaskbook.com/ru/teacherpack/tmaker_main.php), а также в книге [3]. Изучить многие из возможностей конструктора можно по приводимым далее примерам их применения (см. п. 3.3).

При разработке заданий по параллельному программированию не требуется использовать систему MPICH2.

3.2.1. Новый вариант функции CreateTask

```
void CreateTask(const char* SubgroupName, int* ProcessCount);
void CreateTask(std::string SubgroupName, int* ProcessCount);
void CreateTask(int* ProcessCount);
```

Функция CreateTask предназначена для инициализации задания. Приведенные выше перегруженные варианты этой функции предназначены для инициализации задания по параллельному программированию. Параметр SubgroupName имеет тот же смысл, что и для исходных вариантов функции: он определяет *заголовок подгруппы*, в которую включается задание, если разрабатываемую группу заданий целесообразно разбить на подгруппы. Если параметр SubgroupName является пустой строкой или отсутствует, то задание не связывается с какой-либо подгруппой.

В отличие от исходных вариантов функции CreateTask, новые варианты содержат дополнительный параметр ProcessCount. Этот параметр определяет *количество процессов*, которое будет использовано при запуске программы с заданием в параллельном режиме.

Если параметр ProcessCount меньше или равен 1, то для инициализации задания используется соответствующий вариант функции CreateTask без данного параметра (при этом выходное значение параметра ProcessCount будет равно 1). Далее при описании функции CreateTask будем предполагать, что параметр ProcessCount имеет значение, большее 1.

Если параметр ProcessCount превосходит 36, то в окне задачника выводится сообщение об ошибке. Указанное максимально допустимое число процессов представляется достаточным для реализации любого учебного задания по параллельному программированию. К примеру, в заданиях, включенных в задачник PT for MPI-2, возможное число процессов не превосходит 16. Следует заметить, что при использовании в параллельном приложении существенно большего числа процессов заметно возрастает время начальной загрузки приложения.

> **Примечание**. Выбор числа 36 в качестве максимально возможного количества процессов объясняется тем, что для обозначения любого из 36 процессов можно использовать единственный символ — цифру (0–9) для первых 10 процессов или латинскую букву (A–Z) для оставшихся 26 процессов, и это позволяет реализовать быстрый переход к требуемому процессу в разделе отладки по нажатию цифровой или буквенной клавиши, соответствующей этому процессу.

Для того чтобы в ходе тестирования параллельной программы с решением задачи использовалось различное количество процессов, рекомендуется при определении значения параметра ProcessCount применять датчик случайных чисел. В конструкторе для этих целей предусмотрена функция RandomN(int M, int N), возвращающая случайное целое число, лежащее в диапазоне от M до N (включительно). Выполнять особые действия по инициализации датчика случайных чисел не требуется, так как подобная инициализация выполняется автоматически в функции CreateGroup, которая инициализирует группу заданий. Если в задании налагается какое-либо ограничение на количество процессов (например, количество процессов всегда должно быть четным), то это ограничение также следует учитывать при генерации параметра ProcessCount.

Обратите внимание на то, что параметр ProcessCount передается как указатель, т. е. его значение может измениться в результате выполнения данной функции. Это связано с тем, что при выполнении задания по параллельному программированию запускаются несколько экземпляров учебной программы, причем каждый из них выполняет определенную роль в ходе инициализации параллельного задания. Напомним, что первой запускается непараллельная программа, выполняющая роль загрузчика. То значение параметра ProcessCount, которое будет в ней получено, используется при запуске программы в параллельном режиме. Экземпляры программы, запущенные в параллельном режиме, тоже вызывают функцию

CreateTask, однако входное значение параметра ProcessCount им не требуется, так как параллельный режим *уже запущен*, и число процессов определено. Поэтому для *процессов параллельного приложения* параметр ProcessCount используется как *выходной параметр*, позволяющий определить *фактическое* количество запущенных процессов.

Следует также учитывать «неравноправие» процессов параллельного приложения при инициализации задания. Все необходимые исходные и контрольные данные определяются в главном процессе (процессе ранга 0), который затем автоматически пересылает каждому подчиненному процессу предназначенные для него данные. Таким образом, действия, связанные с инициализацией всех данных, входящих в задание, достаточно выполнять только в главном процессе параллельного приложения, тогда как в подчиненных процессах инициализация задания может (и должна) состоять лишь в вызове функции CreateTask. Как, однако, отличить главный процесс от подчиненного в функции, определяющей учебное задание? Для этого также используется возвращаемое значение параметра ProcessCount: для главного процесса он возвращает действительное количество процессов параллельного приложения, тогда как для подчиненных процессов всегда возвращается значение 0. Нулевое значение параметра ProcessCount означает, что никакие дополнительные действия, связанные с инициализацией учебного задания, в данном процессе выполнять не требуется.

Отметим, что нулевое значение параметра ProcessCount возвращается также в экземпляре непараллельной программы, выполняющей роль загрузчика. В самом деле, единственная «обязанность» этой программы состоит в определении числа процессов и использовании этого числа в качестве параметра приложения mpiexec.exe, запускающего параллельный вариант программы. Поэтому в программе-загрузчике, как и в программе, выполняемой в качестве подчиненного процесса параллельного приложения, действия по инициализации задания должны состоять только в вызове функции CreateTask.

Наконец, следует упомянуть о демонстрационном варианте программы с учебным заданием, который всегда выполняется в непараллельном режиме. При демонстрационном запуске программа выступает не в роли загрузчика, а в роли обычной непараллельной программы, в которой задание инициализируется и отображается в окне задачника. В этой ситуации выходное значение параметра ProcessCount всегда совпадает с его входным значением.

Все приведенные выше сведения, связанные с параметром ProcessCount, представлены в таблице 4.

Таблица 4

Особенности использования параметра ProcessCount

«Роль» программы	Входное значение параметра ProcessCount	Выходное значение параметра ProcessCount
Непараллельная программа-загрузчик, обеспечивающая запуск параллельного варианта программы	Используется: определяет число процессов при запуске параллельного варианта программы	Всегда равно 0
Главный процесс параллельной программы (процесс ранга 0)	Не используется	Равно числу процессов в параллельной программе; используется при формировании входных и выходных данных
Подчиненный процесс параллельной программы	Не используется	Всегда равно 0
Непараллельная программа, обеспечивающая демонстрационный запуск учебного задания	Используется	Всегда равно входному значению; используется при формировании входных и выходных данных

3.2.2. Функция SetProcess

```
void SetProcess(int ProcessRank);
```

Данная функция устанавливает в качестве *текущего процесса* параллельного приложения процесс ранга ProcessRank. Все *числовые* исходные и контрольные данные связываются с текущим процессом. До первого вызова данной функции текущим процессом считается процесс ранга 0. Функцию SetProcess можно вызывать несколько раз с одним и тем же параметром (например, первый раз процесс делается текущим при определении связанных с ним исходных данных, а второй раз — при определении его контрольных данных).

Если какой-либо элемент исходных или контрольных данных связан с процессом ранга ProcessRank, то ввести или, соответственно, вывести этот элемент при выполнении задания можно будет только в этом процессе. Порядок ввода и вывода элементов в каждом процессе определяется, как обычно, порядком вызова соответствующих функций групп Data и Result, определяющих наборы исходных и контрольных данных, включаемых в задание. Для повышения наглядности следует всегда указывать в окне

задачника комментарий вида «Процесс *R*:» перед теми исходными или результирующими данными, которые связаны с процессом ранга *R*.

Параметр ProcessRank должен принимать значения в диапазоне от 0 до $K-1$, где K — количество процессов, возвращаемое параметром ProcessCount функции CreateTask. При нарушении этого условия выводится сообщение об ошибке «*Параметр функции SetProcess находится вне диапазона 0..K–1, где K — количество процессов*». Особое сообщение об ошибке будет выведено при попытке вызвать функцию SetProcess при выполнении подчиненного процесса параллельного приложения или непараллельной программы-загрузчика (напомним, что в этих режимах работы программы функция CreateTask возвращает в параметре ProcessCount значение 0, свидетельствующее о том, что выполнять дальнейшие действия по инициализации задания не требуется). В этом случае текст сообщения об ошибке будет следующим: «*В ситуации, когда параметр ProcessCount функции CreateTask вернул значение 0, не выполнен немедленный выход из функции инициализации задания*».

Функцию SetProcess можно вызывать и при формировании задания, не связанного с параллельным программированием. В подобной ситуации количество процессов всегда равно 1, допустимым значением параметра ProcessRank является только 0, и вызов функции SetProcess(0) не выполняет никаких действий.

Следует подчеркнуть, что с текущим процессом связываются только *числовые* данные. Данные прочих простых типов (логические, символьные, строковые, данные-указатели), а также все «внешние» данные (файлы и динамические структуры) остаются связанными с главным процессом параллельного приложения. Это ограничение объясняется тем, что, во-первых, основной областью применения параллельного программирования являются численные методы, и, во-вторых, для освоения возможностей библиотеки MPI вполне достаточно использования числовых данных (включая числовые массивы и структуры). Единственным исключением является появившаяся в MPI-2 возможность *параллельного файлового ввода-вывода*, для изучения которой необходимо включать в задание «внешние» файлы, а также их имена. Однако все подобные данные можно определять в главном процессе, при этом потребуется лишь организовать в начале выполнения учебной программы пересылку информации об имени файла во все подчиненные процессы. Именно по такому принципу реализованы все задания из задачника PT for MPI-2, связанные с файловым вводом-выводом (см. п. 2.6).

3.3. Примеры разработки заданий по параллельному программированию

3.3.1. Создание новой группы учебных заданий и импортирование в нее имеющегося задания

Любая группа заданий должна оформляться в виде динамической библиотеки — dll-файла. Будем разрабатывать библиотеку с новой группой заданий в среде Visual Studio 2017. Назовем новую группу MPIDemo (в качестве префикса имени группы, связанной с MPI-программированием, необходимо указать текст «MPI», поскольку при наличии такого префикса в именах заданий модуль PT4Load будет создавать для этих заданий специализированные проекты-заготовки, предназначенные для выполнения задания в параллельном режиме — см. п. 1.1.3). Все динамические библиотеки, содержащие группы заданий для электронного задачника Programming Taskbook, должны иметь имя, которое начинается с префикса «PT4», за которым следует имя группы. Таким образом, наш проект должен иметь имя PT4MPIDemo.

Создадим заготовку проекта в подкаталоге рабочего каталога PT4Work, выполнив следующие действия:

- начнем с вызова команды меню **File | New | Project…** (**Файл | Создать | Проект…**);
- в появившемся окне выберем раздел **Visual C++** и в нем вариант **Win32 Project** (**Проект Win32**);
- в качестве имени проекта укажем PT4MPIDemo, в качестве каталога размещения проекта укажем c:\PT4Work, снимем выделение с флажка **Create directory for solution** (**Создать каталог для решения**), если этот флажок установлен;
- после нажатия кнопки **OK** на экране появится окно **Win32 Application Wizard** (**Мастер настройки приложений Win32**), в котором следует сразу нажать кнопку **Next** (**Далее**), установить в появившемся списке настроек вариант **DLL** (**Библиотека DLL**) и *снять все флажки в разделе **Additional options** (**Дополнительные параметры**)*, после чего нажать кнопку **Finish** (**Готово**).

Созданный проект будет включать файл PT4MPIDemo.cpp, в котором мы и будем разрабатывать новые задания. Кроме того, в этот проект надо добавить файлы pt4taskmaker.cpp и pt4taskmaker.h, скопировав их из подкаталога PT4forMPI2\taskmaker системного каталога задачника (c:\Program Files (x86)\PT4). Для файла pt4taskmaker.cpp необходимо также выполнить команду меню **Project | Add Existing Item…** (**Проект | Добавить существующий элемент…**).

Определяемая нами в файле PT4MPIDemo.cpp группа заданий должна включать ряд стандартных элементов. Перечислим эти элементы:

- директива подключения заголовочного файла pt4taskmaker.h;
- *основная функция группы заданий*, определяющая задание по его номеру (обычно она имеет имя InitTask); данная функция имеет возвращаемый тип void, один параметр целого типа num и должна снабжаться модификатором _stdcall;
- *функция инициализации группы заданий*, которая должна иметь фиксированное имя inittaskgroup (все буквы строчные); эта функция имеет возвращаемый тип void, не имеет параметров и также должна снабжаться модификатором _stdcall.

Для подключения заголовочного файла pt4taskmaker.h добавим в начало файла PT4MPIDemo.cpp следующую директиву:

```
#include "pt4taskmaker.h"
```

Функция InitTask(int num) обеспечивает инициализацию задания с номером num. Обычно в ней используется оператор switch, в котором происходит выбор инициализирующей функции по значению параметра num.

Простейшим способом добавить задание в новую группу является импортирование задания из уже существующей группы. Для этого в конструкторе предусмотрена функция UseTask с двумя параметрами: первый параметр (типа const char* или std::string) задает имя существующей группы, из которой импортируется задание, второй параметр (типа int) — номер импортируемого задания. Никаких особых действий, связанных с загрузкой указанной группы заданий, выполнять не требуется, поскольку задачник выполняет эту загрузку автоматически. Воспользуемся этой функцией и импортируем в нашу группу второе задание из первой группы MPI1Proc, входящей в задачник PT for MPI-2 (ранее мы обсуждали это задание в п. 1.1). В результате основная функция группы заданий должна принять следующий вид:

```
void _stdcall InitTask(int num)
{
  switch (num)
  {
    case 1:
      UseTask("MPI1Proc", 2);
      break;
  }
}
```

Осталось реализовать функцию инициализации группы заданий inittaskgroup. В этой функции должна вызываться функция CreateGroup, в которой задаются настройки создаваемой группы: имя группы, ее описание,

сведения об авторе, строковый ключ (последовательность произвольных символов), число заданий и имя основной функции группы.

По умолчанию группа считается доступной для всех языков программирования, поддерживаемых универсальным задачником Programming Taskbook. Поскольку задачник PT for MPI-2 ориентирован только на язык C++, необходимо добавить в функцию inittaskgroup фрагмент, позволяющий сделать группу доступной только для указанного языка (для этого следует проанализировать возвращаемое значение функции CurrentLanguage, позволяющей определить выбранный для задачника язык программирования). Приведем определение функции inittaskgroup для нашей группы заданий:

```
void _stdcall inittaskgroup()
{
  if ((CurrentLanguage() & lgCPP) == 0)
    return;
  CreateGroup("MPIDemo",
    "Примеры задач по параллельному программированию",
    "М. Э. Абрамян, 2017", "sddwertfghklbfdgfgd", 1, InitTask);
}
```

Условный оператор в функции inittaskgroup обеспечивает немедленный выход из этой функции (без выполнения инициализации группы заданий) в случае, если текущий язык не является языком C++.

Для возможности использования созданной динамической библиотеки на любом компьютере, необходимо выполнять ее компиляцию в режиме Release, поэтому сразу изменим режим Debug на Release, выбрав его из соответствующего выпадающего списка на панели инструментов.

Поскольку наш проект является библиотекой, а не исполняемым приложением, запустить его на выполнение нельзя. Для тестирования динамических библиотек удобно использовать *управляющее приложение* (host application), которое при запуске подключает библиотеку и тем самым позволяет проверить правильность ее работы. При тестировании библиотеки с учебными заданиями наиболее подходящим вариантом управляющего приложения является программный модуль PT4Demo.exe, входящий в состав задачника Programming Taskbook и позволяющий отобразить на экране любое задание группы в демонстрационном режиме. С помощью параметров командной строки программу PT4Demo можно настроить на немедленное отображение определенного задания требуемой группы. Не приводя здесь все возможные варианты этих параметров, укажем те, которые следует использовать для тестирования нашей библиотеки:

```
-gMPIDemo -n999
```

Первый параметр определяет группу заданий (MPIDemo), а второй обеспечивает отображение на экране *последнего* из заданий этой группы.

Для определения управляющего приложения и его параметров в среде Visual Studio надо выполнить следующие действия:

- начните с команды меню **Project | PT4MPIDemo Properties... (Проект | Свойства PT4MPIDemo…)**;
- выберите в появившемся окне вариант **Release** в качестве значения настройки **Configuration (Конфигурация)**;
- перейдите в раздел **Debugging (Отладка)** и заполните значения трех первых настроек следующим образом (мы предполагаем, что задачник Programming Taskbook размещен в каталоге c:\Program files (x86)\PT4):

  Command (Команда): c:\Program files (x86)\PT4\PT4Demo.exe
  Command Arguments (Аргументы команды): -gMPIDemo -n999
  Working Directory (Рабочий каталог): .\Release

После этого закройте окно свойств, нажав кнопку **ОК**.

Теперь мы можем протестировать заготовку нашей библиотеки. При запуске программы (например, по нажатию клавиши [F5]) библиотека должна успешно откомпилироваться, после чего на экране появится окно модуля PT4Demo с выбранной группой MPIDemo, а также окно задачника с заданием MPIDemo1 в демонстрационном режиме (рис. 61).

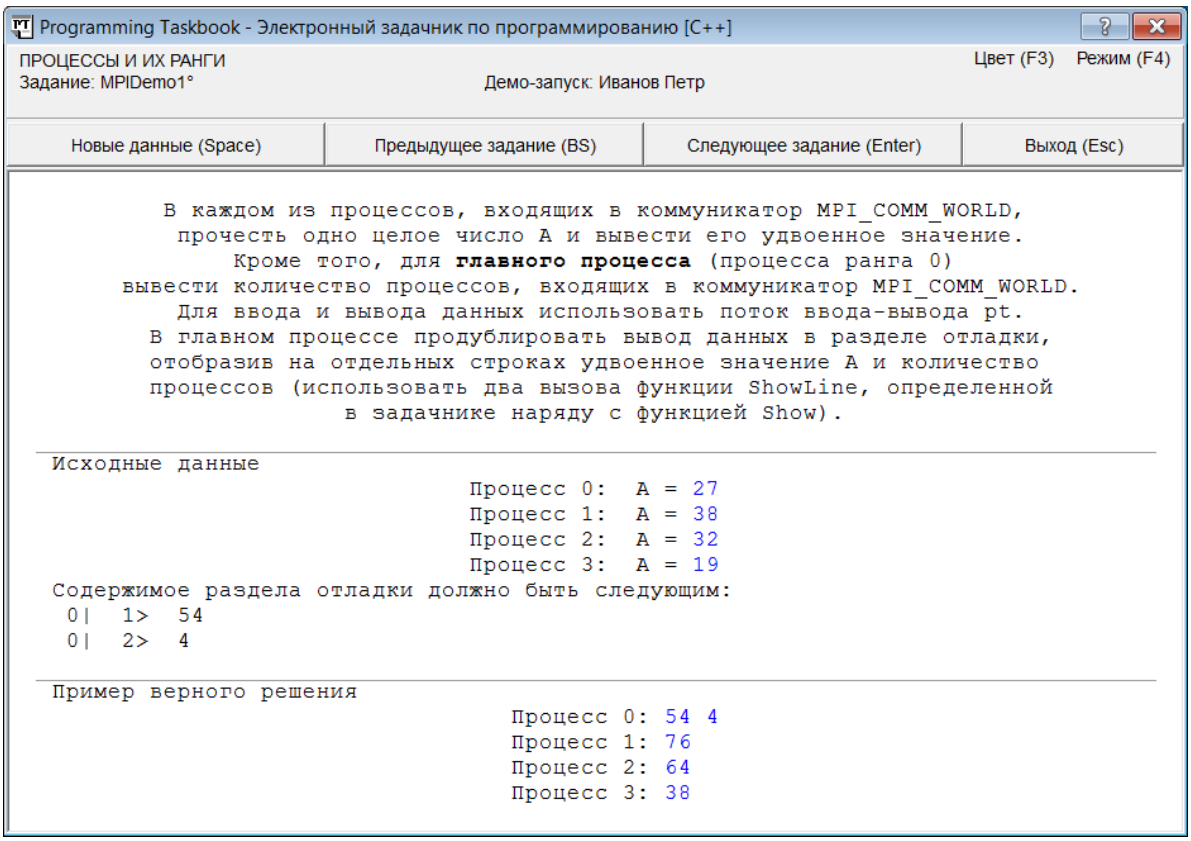

Рис. 61. Демонстрационный запуск задания MPIDemo1

Задание MPIDemo1 будет совпадать с заданием MPI1Proc2. Возможность подобного импортирования заданий полезна, если требуется исключить некоторые задания (слишком простые или слишком сложные), изменить порядок их следования или объединить в одной группе задания из нескольких имеющихся групп (например, для создания группы, используемой при проведении проверочной работы).

3.3.2. Разработка простого задания

Первое задание, которое мы реализуем явным образом, относится к той же начальной теме «Процессы и их ранги», что и задание MPI1Proc2, уже импортированное в разрабатываемую группу.

MPIDemo2. В главном процессе коммуникатора MPI_COMM_WORLD прочесть целое число и вывести его противоположное значение. В каждом подчиненном процессе того же коммуникатора прочесть вещественное число и вывести значение этого числа, умноженное на ранг процесса.

При выполнения этого задания не требуется выполнять пересылку данных. Тем не менее при реализации этого задания нам придется использовать все новые средства конструктора заданий, связанные с параллельным программированием («параллельный» вариант функции CreateTask и функцию SetProcess). Кроме того, мы будем применять базовые средства конструктора, связанные с определением формулировки задания и входящих в него исходных и контрольных данных (причем как целого, так и вещественного типа).

Предварительно определим вспомогательную функцию, которая упростит действия по добавлению к заданию комментариев вида «Процесс <*номер процесса*>:»:

```
#include <string>
std::string prc(int n)
{
  return "Процесс " + std::to_string(n) + ": ";
}
```

Реализуем новое задание в виде функции MPIDemo2:

```
void MPIDemo2()
{
  int count = RandomN(3, 5);
  CreateTask("Процессы и их ранги", &count);
  if (count == 0)
    return;
  TaskText(
    "В главном процессе коммуникатора MPI\\_COMM\\_WORLD
```

```
          ↳ прочесть целое число\n"
    "и вывести его противоположное значение.
          ↳ В каждом подчиненном процессе\n"
    "того же коммуникатора прочесть
          ↳ вещественное число и вывести\n"
    "значение этого числа, умноженное на ранг процесса."
  );
  SetProcess(0);
  int n = RandomN(-99, 99);
  DataN(prc(0), n, 0, 1, 7);
  ResultN(prc(0), -n, 0, 1, 7);
  for (int i = 1; i < count; ++i)
  {
    SetProcess(i);
    double d = RandomR(-99, 99);
    DataR(prc(i), d, 0, i + 1, 7);
    ResultR(prc(i), d * i, 0, i + 1, 7);
  }
}
```

Необходимо также откорректировать функцию InitTask, включив в нее вызов функции MPIDemo2 (в дальнейшем функция InitTask будет дополняться вызовами функций, реализующих другие задания нашей группы):

```
void _stdcall InitTask(int num)
{
  switch (num)
  {
    case 1:
      UseTask("MPI1Proc", 2);
      break;
    case 2:
      MPIDemo2();
      break;
  }
}
```

Наконец, надо откорректировать предпоследний параметр функции CreateGroup, вызываемой в функции inittaskgroup, изменив его значение с 1 на 2:

```
  CreateGroup("MPIDemo",
    "Примеры задач по параллельному программированию",
    "М. Э. Абрамян, 2017", "sddwertfghklbfdgfgd", 2, InitTask);
```

При запуске нашего проекта на экране появится окно задачника, подобное приведенному на рис. 62 (количество процессов, как обычно, может быть другим). Используя кнопки окна или клавиши [Пробел], [Enter] и [Backspace], можно просматривать различные варианты исходных и контрольных данных для этого задания или переходить к другим заданиям группы MPIDemo.

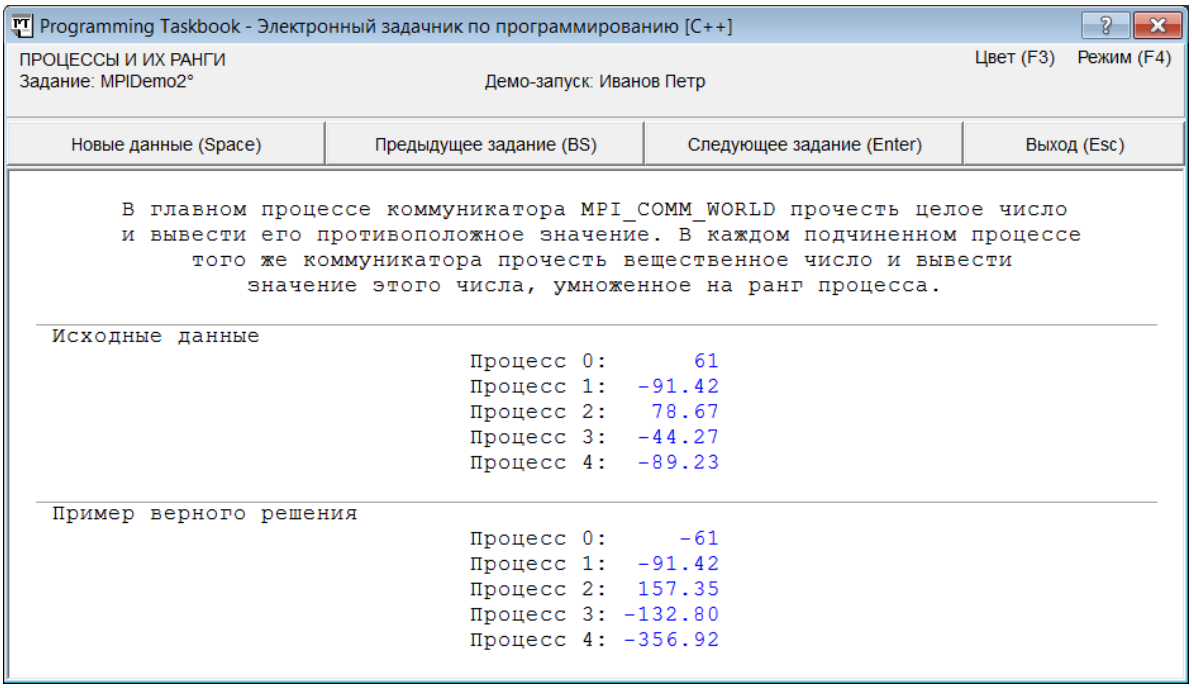

Рис. 62. Демонстрационный запуск задания MPIDemo2

Прокомментируем функцию MPIDemo2.

Первый оператор определяет (с помощью функции RandomN, входящей в конструктор PT4TaskMaker) количество процессов, используемое при текущем запуске задания. Возможное число процессов будет лежать в диапазоне от 3 до 5; для подобного простого задания такого количества процессов вполне достаточно — главное, чтобы при различных тестовых испытаниях программы количество процессов изменялось, пусть и в небольшом диапазоне.

Следующий оператор инициализирует задание, вызывая функцию CreateTask. Благодаря указанию количества процессов count в качестве второго параметра функции, задание будет выполняться в параллельном режиме.

Особое внимание надо обратить на следующий условный оператор:

```
if (count == 0)
  return;
```

Для того чтобы понять его смысл, необходимо вспомнить механизм выполнения задания в параллельном режиме (см. п. 3.1). Экземпляр программы, запущенной непосредственно из интегрированной среды, является

непараллельной программой-загрузчиком, которая определяет требуемое количество процессов и запускает параллельное приложение. После этого никакие действия, связанные с настройкой задания, ей выполнять не требуется. При выполнении параллельного приложения запускаются несколько экземпляров исходной программы (по одному экземпляру для каждого процесса), однако только в главном процессе (процессе ранга 0) определяются все данные, связанные с заданием; прочие процессы эти данные не формируют, а лишь получают от главного процесса. Таким образом, часть функции MPIDemo2, следующую за вызовом функции CreateTask, требуется выполнить *только* для главного процесса параллельного приложения, причем именно возвращаемое значение параметра count позволяет проверить, что данный экземпляр программы представляет собой главный процесс: в этом (и только в этом) случае параметр count отличен от нуля и, кроме того, равен числу запущенных процессов параллельного приложения. Для непараллельной программы-загрузчика и программ, запущенных в качестве подчиненных процессов параллельного приложения, параметр count вернет значение 0, означающее, что дальнейшие действия, определенные в функции MPIDemo2, выполнять не требуется. *Входное* значение параметра count в процессах параллельного приложения не используется; оно учитывается только в программе-загрузчике при запуске программы в параллельном режиме, а также при демо-запуске задания.

Программа, выполняющая демо-запуск задания, является непараллельной, как и программа-загрузчик (именно в таком непараллельном режиме работает, в частности, программный модуль PT4Demo, используемый нами для предварительного тестирования создаваемых заданий). Только при демо-запуске задания можно гарантировать, что входное значение параметра count будет равно его выходному значению.

Следует заметить, что отсутствие оператора if (count == 0) return; не скажется на предварительном тестировании задания, так как при запуске задания в демо-режиме возвращаемое значение параметра count никогда не будет равно 0. Однако при попытке запуска этого задания в параллельном режиме в случае отсутствия данного оператора будет выведено сообщение об ошибке: «*В ситуации, когда параметр ProcessCount функции CreateTask вернул значение 0, не выполнен немедленный выход из функции инициализации задания*».

Рассмотрим оставшиеся операторы функции MPIDemo2.

С помощью функции TaskText задается формулировка задания (обратите внимание на то, что перед символом подчеркивания указывается экранированный символ «\» (двойная обратная косая черта); это связано с тем, что «обычный» символ подчеркивания в формулировках заданий означает переход в режим нижних индексов). В строке, указанной в функции TaskText, с помощью символов '\n' явным образом задаются позиции, по ко-

торым она разбивается на отдельные строки при выводе в разделе с формулировкой задания.

С помощью функций DataN, DataR и ResultN, ResultR определяются исходные и результирующие данные (данные для подчиненных процессов определяются в цикле). Перед указанием данных для каждого процесса вызывается функция SetProcess, определяющая текущий процесс (в качестве параметра функции указывается ранг этого процесса). Все последующие данные, определяемые в задании, связываются с текущим процессом. Напомним, что до первого вызова функции SetProcess текущим считается процесс ранга 0 (таким образом, оператор SetProcess(0); можно было не указывать). В нашем случае с каждым процессом связывается один элемент исходных данных (целое или вещественное число) и один элемент результирующих данных (соответственно, целое число, противоположное исходному, или вещественное число, умноженное на ранг подчиненного процесса). Элементы исходных данных генерируются с помощью функций RandomN и RandomR, возвращающих соответственно целое и вещественное число, лежащее в указанном диапазоне. Опишем параметры функций DataN, DataR, ResultN, ResultR:

const char* cmt – комментарий к определяемому элементу данных (этот параметр может отсутствовать),

int value или double value – значение элемента данных,

int X – горизонтальная координата, определяющая начальную позицию вывода элемента вместе с его комментарием в соответствующем разделе окна задачника — разделе исходных данных для функций DataN и DataR и разделе результатов и примера верного решения для функций ResultN и ResultR; параметр X может лежать в диапазоне от 1 до 78; кроме того, для этого параметра предусмотрены три особых значения: 0, означающее центрирование данного элемента по всей ширине раздела, 100 — центрирование элемента по левой половине раздела, и 200 — центрирование по правой половине; можно также использовать константы xCenter = 0, xLeft = 100 и xRight = 200;

int Y – вертикальная координата, определяющая номер строки соответствующего раздела, в которой будет выведен элемент (строки нумеруются от 1);

int W – ширина области вывода элемента (комментарий располагается слева от области вывода; элементы выравниваются по правой границе области вывода).

При определении ширины вывода W следует учитывать, что вещественные числа по умолчанию отображаются с двумя дробными знаками (это значение можно изменить с помощью функции SetPrecision).

В конце функции, определяющей задание, можно настроить количество успешных тестовых испытаний (от 3 до 9), необходимое для того, чтобы данное задание считалось выполненным (успешные тестовые испытания должны быть проведены подряд). Для этого предназначена функция SetTestCount(int count). Если данная функция не вызвана (как в нашем случае), то количество тестовых испытаний полагается равным 5.

Заканчивая обзор функции MPIDemo2, заметим, что при отсутствии функций SetProcess вид окна задачника в демонстрационном режиме не изменится и никакого сообщения об ошибке выведено не будет. Однако в этом случае все исходные и результирующие данные будут связаны с процессом ранга 0, что приведет к противоречию с формулировкой задания. Чтобы выявить ошибки такого рода, необходимо попытаться *выполнить* разработанное задание, запустив его в параллельном режиме.

Библиотеки с дополнительными группами заданий задачник ищет в рабочем каталоге учащегося и в специальном подкаталоге LIB своего системного каталога (обычно системным каталогом задачника является каталог C:\Program Files (x86)\PT4). Чтобы не выполнять дополнительных действий по копированию файла PT4MPIDemo.dll в подкаталог LIB, можно настроить каталог Release, в котором создается данный файл, в качестве еще одного рабочего каталога задачника (используя, например, программу PT4Setup). После такой настройки в каталоге Release появится ярлык Load.lnk, позволяющий создать заготовки для учебных заданий. Заметим, что благодаря префиксу «MPI» в имени MPIDemo созданные проекты-заготовки для заданий группы MPIDemo будут учитывать особенности заданий по параллельному MPI-программированию (в частности, к проектам будут подключены необходимые файлы библиотеки MPI). Мы не будем приводить программу, решающую задачу MPIDemo2, поскольку она аналогична решению задачи MPI1Proc2, описанному в п. 1.1.5.

3.3.3. Разработка задания, связанного с пересылкой данных

Теперь реализуем задание, в котором требуется осуществлять пересылку данных между процессами. Сформулируем его следующим образом.

MPIDemo3. Количество процессов K является четным, в каждом процессе дано целое число. Переслать числа из всех процессов четного ранга $(0, 2, \ldots, K-2)$ в процесс 0, а числа из всех процессов нечетного ранга $(1, 3, \ldots, K-1)$ — в процесс 1. В процессах 0 и 1 вывести полученные числа в порядке возрастания рангов переславших их процессов.

По сравнению с предыдущим заданием здесь желательно использовать большее количество процессов, например, 6, 8 или 10. В такой ситуации целесообразно расположить данные о процессах в два столбца. При этом группировка процессов с четными и нечетными рангами по разным экранным столбцам придаст заданию дополнительную наглядность.

В разделе результатов требуется отобразить данные только для двух процессов, поэтому можно, как и в предыдущем задании, использовать размещение процессов в один столбец, обеспечив центрирование результирующих наборов данных, связанных с каждым процессом.

Приведем вид окна задачника для задания MPIDemo3 (рис. 63) и одноименную функцию, реализующую это задание.

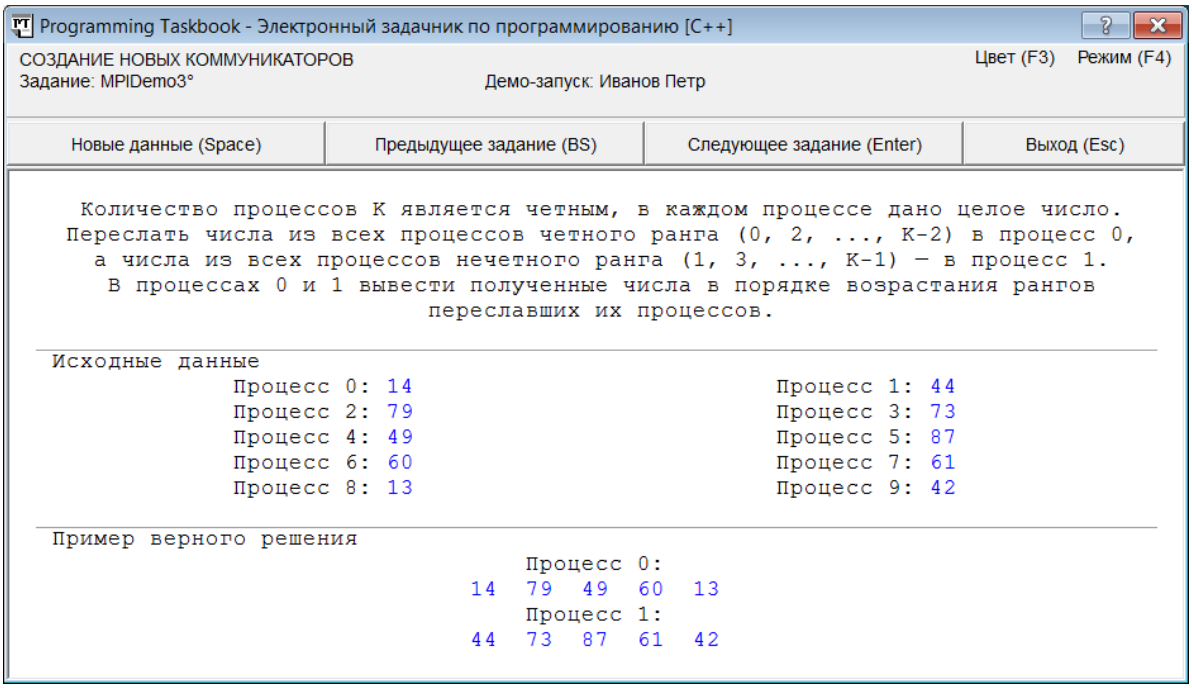

Рис. 63. Демонстрационный запуск задания MPIDemo3

```
void MPIDemo3()
{
  int count = 2 * RandomN(3, 5);
  CreateTask("Создание новых коммуникаторов", &count);
  if (count == 0)
    return;
  TaskText(
    "Количество процессов~{K} является четным, в каждом процессе
        ↳ дано целое число.\n"
    "Переслать числа из всех процессов четного ранга (0, 2,~\\.,
        ↳ {K}\\,\\-\\,2) в процесс~0,\n"
    "а числа из всех процессов нечетного ранга (1, 3,~\\.,
        ↳ {K}\\,\\-\\,1)~\\= в процесс~1.\n"
    "В процессах 0 и 1 вывести полученные числа в порядке
        ↳ возрастания рангов\n"
    "переславших их процессов."
  );
  int k = count / 2;
```

```
  ResultComment("Процесс 0:", 0, 1);
  ResultComment("Процесс 1:", 0, 3);
  for (int i = 0; i < k; ++i)
  {
    int d = RandomN(10, 99);
    SetProcess(2 * i);
    DataN(prc(2 * i), d, xLeft, i + 1, 2);
    SetProcess(0);
    ResultN(d, Center(i + 1, k, 2, 2), 2, 2);
    d = RandomN(10, 99);
    SetProcess(2 * i + 1);
    DataN(prc(2 * i + 1), d, xRight, i + 1, 2);
    SetProcess(1);
    ResultN(d, Center(i + 1, k, 2, 2), 4, 2);
  }
}
```

Прокомментируем приведенную функцию.

Задание отнесено к подгруппе «Создание новых коммуникаторов» (это имя указано в качестве первого параметра функции CreateTask), так как наиболее эффективный вариант его решения связан с применением вспомогательных коммуникаторов.

В тексте формулировки задания используется ряд *управляющих последовательностей* (команд): фигурные скобки выделяют имена переменных, символ «~» обозначает стандартный неразрывный пробел, команда «\,» — малый неразрывный пробел, команда «\=» обозначает тире (—), команда «\-» — знак «минус» (−), команда «\.» — многоточие (…). Некоторые из этих последовательностей учитываются только при выводе формулировки задания в виде html-документа в веб-браузере (см. примечание в п. 3.3.4). В частности, в html-документе все переменные выделяются *курсивом*. Если в управляющую последовательность входит символ «\», то его необходимо экранировать.

Для вывода комментариев, не связанных с конкретным элементом данных, предусмотрены функции DataComment и ResultComment; последнюю из этих функций мы использовали в нашем задании.

В функциях DataN мы применили горизонтальное центрирование данных относительно левой или правой половины раздела, указав в качестве параметра X константы xLeft и xRight соответственно. Напомним, что для центрирования относительно всего раздела можно использовать константу xCenter, но проще явно указать ее значение, равное 0 (мы использовали это значение в функциях ResultComment). В функциях ResultN применяется вспомогательная функция Center(I, N, W, B), обеспечивающая центрирование

по горизонтали *набора* элементов: она возвращает позицию, с которой надо вывести I-й элемент набора из N элементов (I меняется от 1 до N) при условии, что ширина каждого элемента равна W позициям, а между соседними элементами надо указывать B пробелов.

Обратите внимание на то, что функцию SetProcess можно многократно вызывать с одним и тем же параметром.

После добавления в библиотеку PT4MPIDemo функции MPIDemo3 необходимо включить ее вызов в функцию InitTask:

```
void _stdcall InitTask(int num)
{
  switch (num)
  {
    case 1:
      UseTask("MPI1Proc", 2);
      break;
    case 2:
      MPIDemo2();
      break;
    case 3:
      MPIDemo3();
      break;
  }
}
```

Кроме того, в вызове функции CreateGroup следует положить равным 3 значение предпоследнего параметра, определяющего количество заданий в группе.

Для проверки правильности разработанного задания следует выполнить его. Как было отмечено выше, при решении целесообразно использовать вспомогательные коммуникаторы. Приведем только заключительную часть функции Solve (эта часть должна располагаться после операторов, автоматически добавляемых в проект-заготовку):

```
MPI_Comm c;
MPI_Comm_split(MPI_COMM_WORLD, rank % 2, rank, &c);
int n, res[5];
pt >> n;
MPI_Gather(&n, 1, MPI_INT, res, 1, MPI_INT, 0, c);
if (rank < 2)
  for (int i = 0; i < size / 2; ++i)
    pt << res[i];
```

Вначале с помощью функции MPI_Comm_split исходный коммуникатор расщепляется на два, каждый из которых связывается с процессами одина-

ковой четности. Затем с помощью функции MPI_Gather данные из всех процессов, входящих в новые коммуникаторы, пересылаются в начальный процесс каждого коммуникатора. В исходном коммуникаторе MPI_COMM_WORLD начальные процессы двух созданных коммуникаторов имеют ранг 0 и 1, поэтому для определения процессов, в которых надо вывести полученные данные, достаточно использовать условие rank < 2.

3.3.4. Разработка сложного задания

В качестве последнего примера рассмотрим достаточно сложное задание, требующее применения различных средств библиотеки MPI. Данное задание можно отнести к подгруппе «Параллельное умножение матрицы на вектор».

MPIDemo4. В процессе 0 дано число N и вектор b размера N с вещественными элементами. В остальных процессах даны строки вещественной квадратной матрицы A порядка N, причем в каждом процессе вначале указывается число строк K, затем для каждой строки указывается ее порядковый номер в матрице и элементы этой строки. Найти произведение Ab и вывести его элементы в процессе 0.

Главной особенностью этого задания по сравнению с ранее рассмотренными является большое число исходных данных, которые надо определить для каждого процесса. Для повышения наглядности следует разбить данные для каждого процесса на несколько экранных строк, отведя одну строку для вывода комментария «Процесс *<номер процесса>*» и значения N (для главного процесса) или K (для подчиненных процессов), а последующие строки — для вывода данных, связанных с вектором b (для главного процесса) или с элементами соответствующих строк матрицы A (для подчиненных процессов).

Приведем вид окна задачника для задания MPIDemo4 (рис. 64) и одноименную функцию, реализующую это задание, вместе со вспомогательной функцией Swap. Учитывая сложность функции MPIDemo4, снабдим ее комментариями, поясняющими назначение каждого этапа инициализации задания.

```
void Swap(int &a, int &b)
{
  int x = a;
  a = b;
  b = x;
}
```

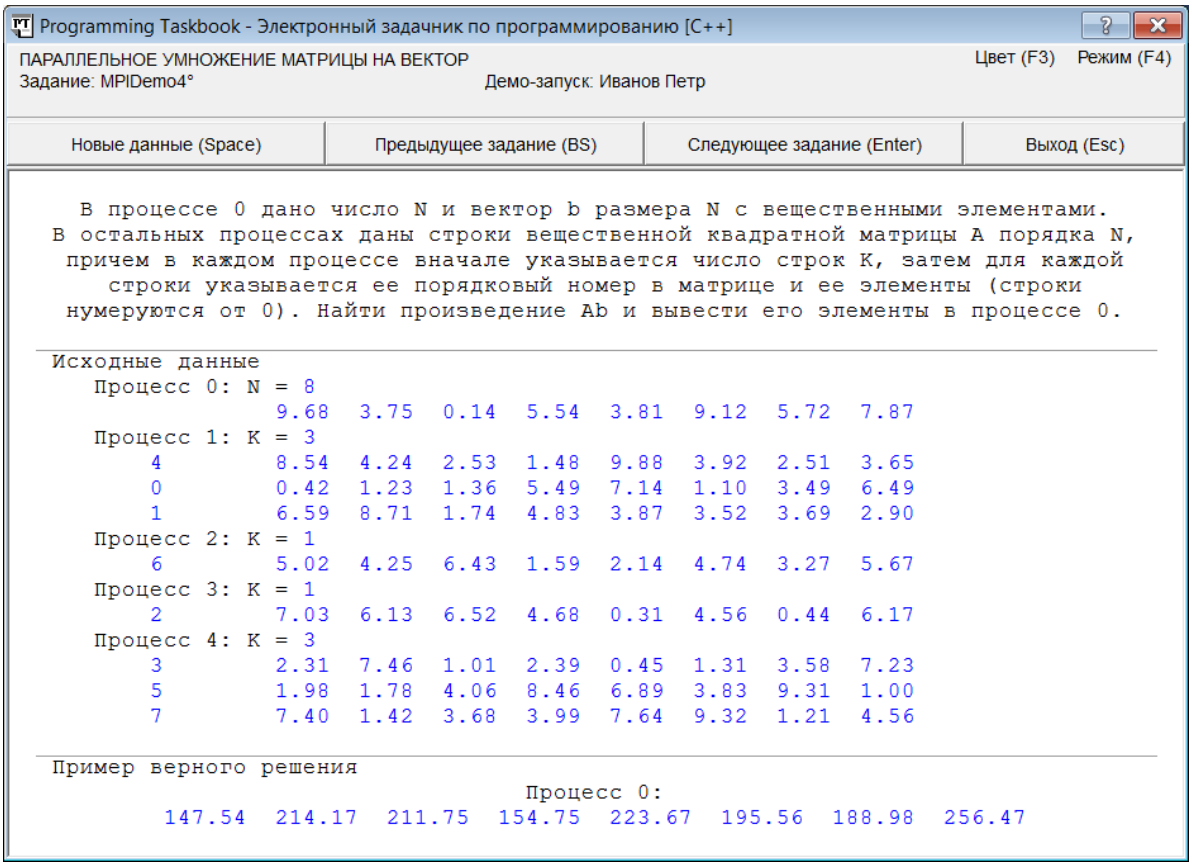

Рис. 64. Демонстрационный запуск задания MPIDemo4

```
void MPIDemo4()
{
  int count = RandomN(3, 6);
  CreateTask("Параллельное умножение матрицы на вектор", &count);
  if (count == 0)
    return;
  TaskText(
    "В процессе 0 дано число {N} и вектор {b} размера {N}
          ↳ с вещественными элементами.\n"
    "В остальных процессах даны строки вещественной квадратной
          ↳ матрицы {A} порядка {N},\n"
    "причем в каждом процессе вначале указывается число
          ↳ строк {K}, затем для каждой\n"
    "строки указывается ее порядковый номер в матрице
          ↳ и ее элементы (строки\n"
    "нумеруются от~0). Найти произведение {Ab} и вывести
          ↳ его элементы в процессе~0."
  );

  // Формирование матрицы a и вектора b; n - порядок матрицы:
```

```
double a[10][10], b[10];
int num[10];
int n = RandomN(6, 9);
for (int i = 0; i < n; ++i)
{
  b[i] = RandomR(0, 9.9);
  for (int j = 0; j < n; ++j)
    a[i][j] = RandomR(0, 9.9);
  num[i] = i;
}

// Вычисление произведения Ab в массиве c:
double c[10];
for (int i = 0; i < n; ++i)
{
  c[i] = 0;
  for (int j = 0; j < n; ++j)
    c[i] += a[i][j] * b[j];
}

// Получение в массиве num случайного порядка
// номеров строк матрицы:
for (int i = 0; i < 2 * n; ++i)
  Swap(num[RandomN(0, n - 1)], num[RandomN(0, n - 1)]);

// Определение числа строк матрицы,
// передаваемых каждому подчиненному процессу:
int k[10];
do
{
  k[count - 2] = n;
  for (int i = 0; i < count - 2; ++i)
  {
    k[i] = RandomN(1, 3);
    k[count - 2] -= k[i];
  }
} while (k[count - 2] <= 0);

// Вывод исходных и контрольных данных для процесса 0:
SetProcess(0);
DataN(prc(0) + "N = ", n, 4, 1, 1);
```

```
  for (int i = 0; i < n; ++i)
    DataR(b[i], Center(i + 1, n, 4, 2), 2, 4);
  ResultComment(prc(0), 0, 1);
  for (int i = 0; i < n; ++i)
    ResultR(c[i], Center(i + 1, n, 6, 2), 2, 6);

  // Вывод исходных данных для остальных процессов:
  int y = 2;
  int cnt = -1;
  for (int i = 1; i < count; ++i)
  {
    SetProcess(i);
    DataN(prc(i) + "K = ", k[i - 1], 4, ++y, 1);
    for (int m = 1; m <= k[i - 1]; ++m)
    {
      ++y;
      ++cnt;
      DataN(num[cnt], 8, y, 1);
      for (int j = 0; j < n; ++j)
        DataR(a[num[cnt]][j], Center(j + 1, n, 4, 2), y, 4);
    }
  }
}
```

После добавления в файл PT4MPIDemo функции MPIDemo4 необходимо изменить на 4 значение предпоследнего параметра функции CreateGroup и откорректировать функцию InitTask, добавив новый вариант в оператор switch:

```
    case 4:
      MPIDemo4();
      break;
```

Приведем решение данного задания, не указывая ту часть функции Solve, которая создается при генерации проекта-заготовки:

```
int n;
double b[10], c;
if (rank == 0)
{
  pt >> n;
  for (int i = 0; i < n; ++i)
    pt >> b[i];
}
```

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(b, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (rank != 0)
{
  int k;
  pt >> k;
  for (int k0 = 0; k0 < k; ++k0)
  {
    int m;
    pt >> m;
    c = 0;
    for (int i = 0; i < n; ++i)
    {
      double a;
      pt >> a;
      c += b[i]*a;
    }
    MPI_Send(&c, 1, MPI_DOUBLE, 0, m, MPI_COMM_WORLD);
  }
}
else
{
  for (int i = 0; i < n; ++i)
  {
    MPI_Status s;
    MPI_Recv(&c, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
      MPI_COMM_WORLD, &s);
    b[s.MPI_TAG] = c;
  }
  for (int i = 0; i < n; ++i)
    pt << b[i];
}
```

В этом решении используется как групповая пересылка данных (из процесса 0 в подчиненные процессы с применением функции MPI_Bcast), так и пересылка данных между двумя процессами (реализуемая функциями MPI_Send и MPI_Recv).

Следует обратить внимание на две особенности использования функций MPI_Send и MPI_Recv: во-первых, при пересылке найденного элемента результирующего вектора связанная с этим элементом информация (порядковый номер элемента) пересылается в виде параметра msgtag (и поэтому в принимающей функции MPI_Recv в качестве этого параметра указана

константа MPI_ANY_TAG); во-вторых, принимающий процесс ранга 0 «не знает», сколько сообщений будет ему послано каждым из подчиненных процессов; ему известно лишь общее число посланных ему сообщений (равное размеру результирующего вектора), поэтому в принимающей функции он использует константу MPI_ANY_SOURCE в качестве «заменителя» ранга процесса-отправителя.

Заметим, что задание MPIDemo4 представляет собой центральный фрагмент самопланирующего алгоритма умножения матрицы на вектор, описанного в п. 1.5.1. Исключен лишь «недетерминированный» этап, связанный с пересылкой матричных строк от главного процесса очередным подчиненным процессам; вместо этого матричные строки сразу распределяются задачником между подчиненными процессами, в которых они и должны вводиться.

Примечание. В задачнике предусмотрена возможность генерации html-страницы с текстом всех формулировок любой группы заданий. Чтобы получить такую страницу для разработанной нами группы, достаточно внести небольшое изменение в настройку **Command Arguments** (**Аргументы команды**) нашего проекта, дополнив параметр –g символом # (в результате настройка **Command Arguments** примет вид -gMPIDemo# -n999). Теперь при запуске проекта PT4MPIDemo на экране вместо окна задачника с последним заданием из группы MPIDemo будет отображаться html-браузер с описанием созданной группы (рис. 65).

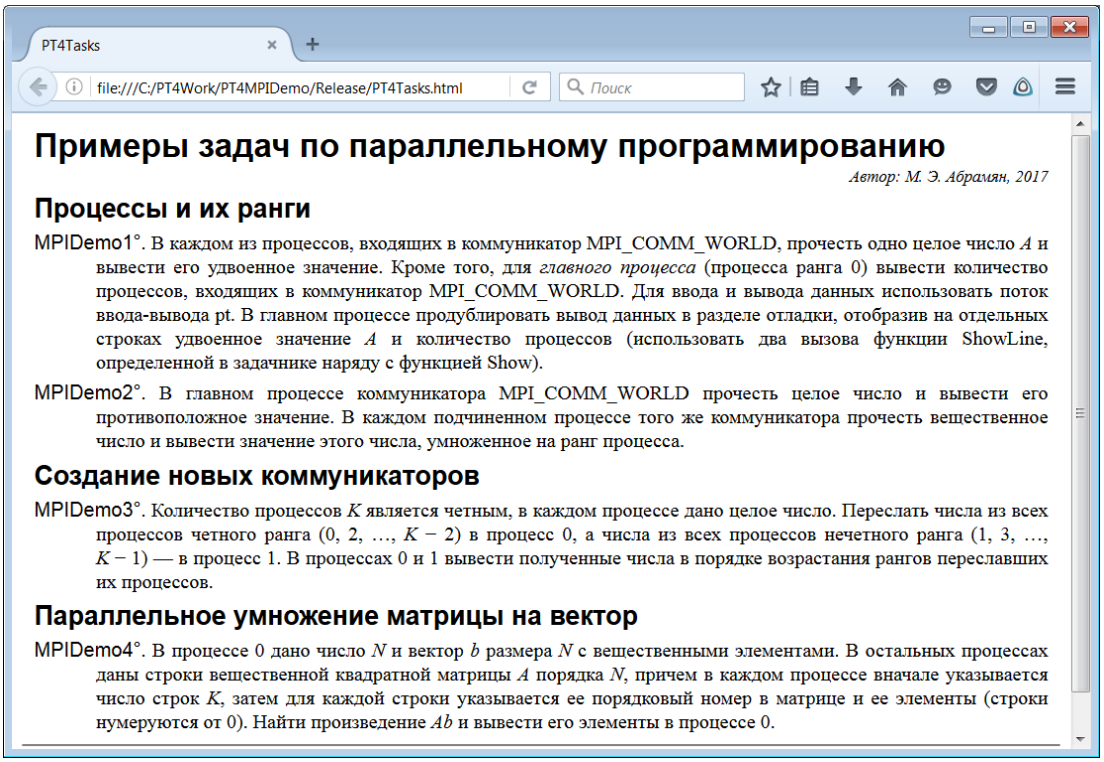

Рис. 65. Страница веб-браузера с описанием группы заданий MPIDemo

Для генерации html-описания группы заданий с помощью программного модуля PT4Demo не обязательно использовать параметры командной строки. Достаточно запустить программу PT4Demo.exe с помощью ярлыка Demo.lnk, содержащегося в рабочем каталоге, выбрать из списка групп нужную группу и нажать клавишу [F2] или кнопку в окне данной программы. Вывести html-описание группы можно также с помощью программы-заготовки, созданной для выполнения задания. Для этого достаточно изменить параметр в функции Task, удалив в нем номер и добавив символ #, например Task("MPIDemo#"). Если указать в параметре символ #, не удаляя номер задания (например Task("MPIDemo4#")), то в html-описание будет включено только задание с указанным номером.

Раздел 4. Дополнения

4.1. Электронный задачник Programming Taskbook for MPI-2

4.1.1. Общее описание

Комплекс «**Электронный задачник по параллельному программированию на базе MPI-2 Programming Taskbook for MPI-2**» (PT for MPI-2) является расширением универсального электронного задачника Programming Taskbook, которое позволяет выполнять задания на разработку параллельных программ с применением технологии MPI стандарта 1.1 и 2.x.

Для возможности использования данного комплекса его следует установить в системный каталог базового варианта электронного задачника Programming Taskbook версии не ниже 4.17 (обычно системным каталогом задачника является каталог C:\Program Files (x86)\PT4). На компьютере надо также установить систему MPICH для Windows, которая обеспечивает запуск программ в параллельном режиме. Задачник может использоваться совместно со следующими версиями системы MPICH:

- 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.

По поводу особенностей установки систем MPICH см. примечание в п. 1.1.1. Если на компьютере установлены обе версии MPICH, то с помощью модулей PT4Setup и PT4Load можно выбирать любую из них для использования с учебными программами.

Комплекс PT for MPI-2 является свободно распространяемым программным продуктом (freeware); он может использоваться как с полным вариантом универсального задачника PT4Complete–1100, так и со свободно распространяемым мини-вариантом PT4Mini–270.

Задания, включенные в задачник PT for MPI-2, могут выполняться на языке C++ во всех программных средах для этого языка, поддерживаемых универсальным задачником Programming Taskbook:

- Microsoft Visual Studio (версия 2008 или выше);
- Code::Blocks (версия 13 или выше).

К учебным программам должна подключаться динамическая библиотека, входящая в состав комплекса MPICH. Доступ к библиотеке обеспечи-

вается с помощью lib-файла и набора заголовочных файлов (также входящих в комплекс MPICH). Все действия по копированию в рабочий каталог дополнительных файлов и их подключению к учебному проекту задачник выполняет автоматически.

Задачник PT for MPI-2 предоставляет при выполнении заданий те же возможности, что и базовый задачник Programming Taskbook; в частности, он передает программе учащегося исходные данные, проверяет правильность результатов, полученных программой, выполняя серию тестовых запусков учебной программы, и сохраняет сведения о каждом тестовом испытании программы в специальном файле. Кроме того, в задачнике PT for MPI-2 предусмотрены дополнительные возможности, связанные со спецификой выполнения заданий по параллельному программированию (более подробно данные возможности описываются в п. 1.1 и 3.1):

- демонстрационный просмотр заданий, не требующий использования параллельного режима;
- создание для выбранного задания проекта-заготовки с подключенными к нему модулями библиотеки MPI;
- особый механизм, обеспечивающий выполнение программы учащегося в параллельном режиме при ее обычном запуске из среды разработки: запущенная программа выполняет запуск приложения MPIRun.exe из комплекса MPICH, которое, в свою очередь, запускает программу в параллельном режиме (все процессы выполняются на локальном компьютере);
- передача каждому процессу параллельной программы его собственного набора исходных данных;
- получение от каждого процесса требуемых результатов и их автоматическая пересылка в главный процесс для проверки и отображения в окне задачника;
- вывод информации об ошибках времени выполнения (в том числе ошибках, возникших при выполнении функций MPI) и ошибках ввода-вывода с указанием рангов процессов, в которых эти ошибки произошли;
- возможность вывода отладочной информации для каждого процесса в специальном разделе окна задачника;
- автоматическая выгрузка из памяти всех запущенных процессов даже в случае зависания параллельной программы.

Перечисленные выше возможности избавляют учащегося от необходимости выполнения дополнительных действий, связанных с запуском его программы в параллельном режиме, и упрощают выявление и исправление стандартных ошибок, возникающих в параллельных программах.

Использование исходных данных, подготовленных задачником для каждого процесса параллельной программы, наглядный вывод в одном окне всех результатов, полученных каждым процессом, и их автоматическая проверка, а также дополнительные средства отладки позволяют учащемуся сосредоточиться на реализации алгоритма решения задачи и обеспечивают надежное тестирование разработанного алгоритма.

Задачник PT for MPI-2 является расширенной версией задачника Programming Taskbook for MPI (PT for MPI), ориентированного на стандарт MPI-1 и разработанного в 2009 г. [1]. Помимо 100 учебных заданий, перенесенных из задачника PT for MPI, задачник PT for MPI-2 включает 150 новых заданий. Часть новых заданий дополняет прежние темы, часть связана с новыми возможностями стандарта MPI-2.

Наряду с группами, посвященными конкретным разделам библиотеки MPI, задачник включает группу заданий на разработку *параллельных алгоритмов* (а именно ленточных и блочных алгоритмов умножения матриц), в которых используются различные средства MPI, изученные в предыдущих группах. Формулировки всех учебных заданий, входящих в задачник PT for MPI-2, приведены в разделе 2.

Реализованные в задачнике PT for MPI-2 средства автоматического запуска и отладки параллельных приложений позволяют использовать его для разработки и тестирования параллельных программ, не связанных с конкретными учебными заданиями. С этой целью в задачник PT for MPI-2 (как и ранее в задачник PT for MPI) включена вспомогательная группа MPIDebug из 36 «заданий», каждое из которых обеспечивает автоматический запуск параллельной программы, причем количество процессов определяется порядковым номером задания (пример использования этой группы приведен в п. 1.5.1). Таким образом, «задания» группы MPIDebug позволяют запускать любые параллельные программы с требуемым количеством процессов непосредственно из интегрированной среды и предоставляют для их отладки средства, имеющиеся в задачнике.

Предусмотренные в задачнике возможности, связанные с отладочной печатью, можно использовать и при разработке консольных параллельных программ, не требующих подключения ядра задачника. Для этих целей достаточно подключить к программе файлы pt4console.h и pt4console.cpp, размещенные в подкаталоге PTforMPI2\stubs системного каталога задачника Programming Taskbook. Пример использования файлов pt4console.h и pt4console.cpp приведен в п. 1.5.3.

С помощью конструктора учебных заданий PT4TaskMaker преподаватель может разрабатывать дополнительные группы заданий, связанные с параллельным MPI-программированием (см. п. 3.2–3.3). Вариант конструктора учебных заданий для языка C++ включен с дистрибутив задачника PT for MPI-2; его компоненты pt4taskmaker.h и pt4taskmaker.cpp со-

держатся в подкаталоге PTforMPI2\taskmaker системного каталога задачника Programming Taskbook.

В состав задачника PT for MPI-2 входит гипертекстовая справочная система PTforMPI2.chm, содержащая ту же информацию, что и раздел сайта ptaskbook.com, посвященный задачнику PT for MPI-2.

Программный комплекс «Электронный задачник по параллельному программированию на базе MPI-2 Programming Taskbook for MPI-2» зарегистрирован в Реестре программ для ЭВМ 2 февраля 2018 г. (свидетельство о государственной регистрации программы для ЭВМ № 2018611548).

Подробная информация о возможностях задачника PT for MPI-2 содержится на сайте ptaskbook.com. С этого сайта можно загрузить последние версии задачника Programming Taskbook и его расширения PT for MPI-2.

4.1.2. Средства задачника для инициализации заданий и ввода-вывода данных

Все описываемые ниже средства задачника реализованы в файлах pt4.h и pt4.cpp. При создании проекта-заготовки с использованием программного модуля PT4Load эти файлы подключаются к проекту автоматически.

Для **инициализации задания** с именем name предназначена функция void Task(const char* name). Функция Task должна вызываться в начале программы, выполняющей это задание (до выполнения операций ввода-вывода). Если в программе, выполняющей задание, не указана функция Task, то при запуске программы будет выведено окно с сообщением «*Не вызвана процедура Task с именем задания*».

Если функция Task вызывается в программе несколько раз, то все последующие ее вызовы игнорируются.

Имя задания должно включать имя группы заданий и порядковый номер в пределах группы (например "MPI1Proc2"). Регистр букв в имени группы может быть произвольным. Если указана неверная группа, то программа выведет сообщение «*Указана неверная тема задания*». Если указан недопустимый номер задания, то программа выведет сообщение, в котором будет указан диапазон допустимых номеров для данной группы. Если после имени задания в параметре name указан символ «?» (например "MPI1Proc2?"), то программа будет работать в демонстрационном режиме, имеющем следующие особенности:

- даже если программа содержит решение задания, это решение не анализируется и информация в файл результатов не заносится;
- при одном запуске программы можно просмотреть несколько вариантов исходных и контрольных данных; для смены набора данных требуется нажать кнопку **Новые данные** или клавишу пробела (см. рис. 61 в п. 3.3.1);

- при одном запуске программы можно последовательно просмотреть все задания данной группы; для перехода к заданию с бóльшим номером требуется нажать кнопку **Следующее задание** или клавишу [Enter], а для перехода к заданию с меньшим номером требуется нажать кнопку **Предыдущее задание** или клавишу [Backspace] (задания перебираются циклически).

Демонстрационный запуск задания по параллельному программированию выполняется в непараллельном режиме, без обращения к программным средствам комплекса MPICH.

Функция Task может также использоваться для генерации и вывода на экран html-страницы с текстом задания или группы заданий (см. рис. 65 в п. 3.3.4). Для этого необходимо указать в качестве параметра name имя конкретного задания или группы заданий и символ «#», например "MPI1Proc2#" или "MPI1Proc#". Для включения в html-страницу нескольких заданий или групп заданий достаточно для каждого задания или группы вызвать функцию Task с параметром, оканчивающимся символом #.

Основным средством **ввода-вывода данных** при выполнении заданий на языке C++ с применением задачника Programming Taskbook является *поток ввода-вывода* pt, определенный в заголовочном файле pt4.h. С его помощью можно выполнять ввод и вывод данных всех стандартных типов, в том числе int, double, char*, используемых в задачнике PT for MPI-2. В заголовочном файле pt4.h определены также функции, предназначенные для ввода и вывода отдельных элементов, однако использование потока pt является более удобным способом организации ввода-вывода. Для ввода данных из потока pt предназначена операция >> (например pt >> a >> b;), для вывода результатов в этот поток предназначена операция << (например pt << 2 * a << 3 + b;).

Ввод и вывод данных должен выполняться после вызова функции Task, в противном случае будет выведено сообщение об ошибке «*В начале программы не вызвана процедура Task с именем задания*».

Для ввода-вывода *последовательностей* с элементами типа T предусмотрены *итераторы* ptin_iterator<T> и ptout_iterator<T>, связанные с потоком pt. Эти итераторы обладают теми же свойствами, что и стандартные потоковые итераторы istream_iterator<T> и ostream_iterator<T> (см. [2]).

У итератора ptin_iterator<T> имеются два конструктора. Конструктор без параметров создает итератор конца последовательности. Конструктор с параметром count типа unsigned int создает итератор чтения, который позволяет прочесть count элементов типа T из потока pt, после чего переходит в состояние конца последовательности (т. е. становится равным итератору ptin_iterator<T>()).

В случае особого значения параметра count, равного 0, размер последовательности считывается из самого потока pt непосредственно перед счи-

тыванием первого элемента последовательности. Если при считывании размера оказывается, что прочитанный элемент данных не является целым числом или это число не является положительным, то итератор сразу переходит в состояние конца последовательности.

Подобная организация итераторов чтения ptin_iterator<T> позволяет легко реализовать считывание из потока ввода pt нескольких последовательностей, если заранее известен их размер или если размер указывается непосредственно перед началом последовательности.

Итератор ptout_iterator<T> создается с помощью конструктора без параметров и позволяет записать в поток pt произвольное количество данных типа T.

4.1.3. Раздел отладки

Задачник Programming Taskbook включает средства, позволяющие выводить отладочную информацию непосредственно в его окно (в специальный *раздел отладки*). Необходимость в подобных дополнительных средствах возникает, прежде всего, при отладке параллельных программ, поскольку для них нельзя использовать такие стандартные средства отладки, как точки останова, пошаговое выполнение программы и окна просмотра значений переменных. Следует также отметить, что возможность вывода информации в раздел отладки позволяет использовать задачник для написания и отладки параллельных программ, не связанных с выполнением конкретных учебных заданий (см. п. 1.5.1).

Отладочные средства задачника могут оказаться полезными и для обычных, непараллельных программ. В этом случае их можно применять в качестве дополнения к средствам встроенного отладчика.

Раздел отладки представляет собой одну или несколько многострочных текстовых областей вывода. Он располагается под основными разделами задачника и выводится на экран только в случае, если в нем содержится какой-либо текст (рис. 66).

При демонстрационном запуске программы все функции, связанные с разделом отладки, игнорируются, поэтому раздел отладки на экране не отображается.

Имеется возможность скрыть в окне задачника все его разделы, кроме раздела отладки; для этого достаточно нажать клавиатурную комбинацию [Ctrl]+[пробел]. Повторное нажатие этой комбинации восстанавливает в окне задачника ранее скрытые разделы. Для скрытия/отображения основных разделов окна задачника можно также использовать соответствующую команду контекстного меню. Скрыть все разделы окна задачника, кроме раздела отладки, можно и программным способом, вызвав функцию HideTask (см. п. 4.1.4).

Programming Taskbook - Электронный задачник по программированию [C++]

ПРОЦЕССЫ И ИХ РАНГИ
Задание: MPI1Proc2°
Выполняет: Иванов Петр
Результаты (F2) Цвет (F3) Режим (F4)

Ознакомительный запуск:
не выполнена ни одна из операций ввода-вывода.
Выход (Esc)

Ввод: 0 из 4 Вывод: 0 из 5 Тесты: 1 из 5

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).

Исходные данные

Процесс 0: A = 30
Процесс 1: A = 17
Процесс 2: A = 41
Процесс 3: A = 28

Содержимое раздела отладки должно быть следующим:

0| 1> 60
0| 2> 4

Пример верного решения

Процесс 0: 60 4
Процесс 1: 34
Процесс 2: 82
Процесс 3: 56

0| 1> Количество процессов: 4
0| 2> Ранг процесса: 0
1| 1> Количество процессов: 4
1| 2> Ранг процесса: 1
2| 1> Количество процессов: 4
2| 2> Ранг процесса: 2
3| 1> Количество процессов: 4
3| 2> Ранг процесса: 3

* 0 1 2 3

Рис. 66. Окно задачника с разделом отладки

Для заданий, не связанных с параллельным программированием, раздел отладки содержит единственную область вывода. Для заданий по параллельному программированию число областей вывода определяется количеством процессов параллельного приложения, которые вывели данные в раздел отладки. В каждый момент времени в разделе отладки отображается одна из областей. Если областей вывода больше одной, то в нижней части раздела отладки выводится набор *маркеров*, позволяющих переключиться на любую из имеющихся областей вывода.

Маркеры с номерами позволяют просмотреть содержимое области вывода, связанной с процессом соответствующего ранга (рис. 67); маркер с символом «*» позволяет просмотреть область вывода, содержащую объединенный текст всех других областей (см. рис. 66).

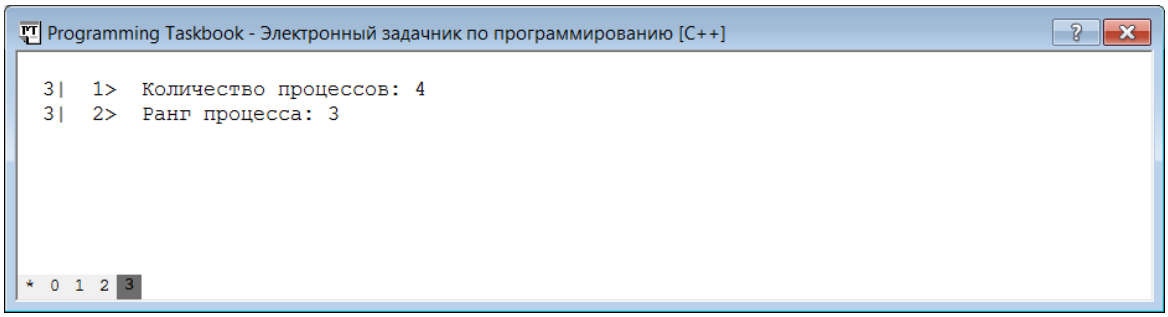

Рис. 67. Отладочная информация для процесса ранга 3

Каждая экранная строка в разделе отладки начинается с ранга процесса, после которого указывается символ «|». Затем указывается порядковый номер строки, который завершается символом «>» (строки для каждого процесса нумеруются независимо).

Для процессов из исходного коммуникатора MPI_COMM_WORLD указываются их ранги в этом коммуникаторе. Для динамически созданных групп процессов (см. п. 1.3.8) указываются их ранги в коммуникаторе MPI_COMM_WORLD, связанном с этими группами, причем перед значением ранга указываются буквенные префиксы «a», «b» и т. д. Все процессы, входящие в одну группу, имеют одинаковый префикс. Наличие префиксов позволяет различать исходные и новые процессы, а также новые процессы, входящие в различные группы. Пример раздела отладки с информацией, выведенной в динамически созданных процессах, приведен в п. 1.3.8 (см. рис. 37 и 38).

Количество отладочных строк для каждого процесса не должно превышать 999; если некоторый процесс пытается вывести данные в строку с номером, превышающим 999, то в связанной с этим процессом области выводится сообщение об ошибке и последующий вывод отладочных данных для этого процесса блокируется. Указанное ограничение позволяет избежать проблем, возникающих при бесконечном выводе отладочной информации из какого-либо зациклившегося подчиненного процесса.

Для переключения на нужную область вывода достаточно щелкнуть мышью на соответствующем маркере. Кроме того, для последовательного перебора маркеров слева направо или справа налево можно использовать клавиши со стрелками [→] и [←] (перебор осуществляется циклически). Можно также сразу перейти к нужной области вывода, нажав соответствующую клавишу: для области «*» — клавишу [*], для областей «0»–«9» — цифровые клавиши [0]–[9], для областей «10»–«35» — буквенные клавиши от [A] до [Z] (для областей, связанных с динамически созданными процессами, клавиши не определены).

Отладочная информация, получаемая из подчиненных процессов, предварительно сохраняется в специальных временных файлах в каталоге учащегося, поэтому она будет доступна для просмотра даже если на каком-

либо этапе выполнения параллельной программы произойдет зависание некоторых подчиненных процессов. Отладочная информация, получаемая из главного процесса, выводится непосредственно в раздел отладки.

Содержимое области вывода, отображаемой в разделе отладки, можно копировать в буфер Windows; для этого предназначена стандартная клавиатурная комбинация [Ctrl]+[C] и соответствующая команда контекстного меню окна задачника.

4.1.4. Функции для вывода и настройки отладочной информации

Для вывода данных в раздел отладки предназначена функция Show. Ее простейший вариант содержит единственный строковый параметр s типа std::string или char*:

```
void Show(const char* s);
void Show(std::string s);
```

Эта функция выводит строку s в область данных раздела отладки.

Если текущая экранная строка уже содержит некоторый текст, то строка s снабжается начальным пробелом и приписывается к этому тексту, за исключением случая, когда при таком приписывании размер полученного текста превысит ширину области данных (равную 80 символам). В последнем случае вывод строки s осуществляется с начала следующей экранной строки; если же и в этой ситуации строка s превысит ширину области данных, то строка s будет выведена на нескольких экранных строках, причем разрывы текста будут выполняться по пробельным символам строки s, а при отсутствии пробелов — при достижении очередного фрагмента строки длины, равной 80.

Строка s может содержать явные команды перехода на новую экранную строку. В качестве таких команд можно использовать или символ '\r' («возврат каретки»), или символ '\n' («переход на новую строку»), или их комбинацию в указанном порядке "\r\n".

Имеется модификация функции Show — функция ShowLine, которая после вывода строки s осуществляет автоматический переход на следующую экранную строку:

```
void ShowLine(const char* s);
void ShowLine(std::string s);
```

Функцию ShowLine можно вызывать без строкового параметра; в этом случае в разделе отладки будет просто выполнен переход на новую экранную строку:

```
void ShowLine();
```

Помимо функции Show с единственным строковым параметром предусмотрены ее перегруженные варианты, предназначенные для вывода числовых отладочных данных. Использование этих вариантов позволяет мак-

симально упростить действия, связанные с выводом числовых данных, поскольку избавляет от необходимости применять стандартные средства языка C++, предназначенные для преобразования чисел в их строковые представления.

Имеется несколько перегруженных вариантов функции Show, предназначенных для вывода числовых данных. Ниже приведены варианты, содержащие наибольшее количество параметров (остальные варианты получаются из приведенных вариантов удалением параметра s, параметра w или обоих этих параметров):

```
void Show(const char* s, int a, int w);
void Show(const char* s, double a, int w);
void Show(std::string s, int a, int w);
void Show(std::string s, double a, int w);
```

Строковый параметр s определяет необязательный *комментарий*, который указывается перед выводимым числом; если параметр s отсутствует, то комментарий полагается равным пустой строке.

Числовой параметр a определяет выводимое число.

Необязательный целочисленный параметр w определяет *ширину поля вывода* (т. е. количество экранных позиций, отводимое для вывода числа). Если указанной ширины w поля вывода недостаточно, то значение параметра w игнорируется; в этом случае (а также в случае, если параметр w отсутствует) используется ширина поля вывода, минимально необходимая для отображения данного числа. Если число не занимает всего поля вывода, то оно дополняется слева пробелами (т. е. выравнивается по *правой* границе поля вывода). В качестве десятичного разделителя для чисел с дробной частью используется точка. Вещественные числа по умолчанию выводятся в формате с фиксированной точкой и двумя дробными знаками. Изменить формат вывода вещественных чисел можно с помощью вспомогательной функции SetPrecision, описываемой далее.

Аналогичные перегруженные варианты предусмотрены и для функции ShowLine. В этих вариантах после вывода отладочных данных автоматически выполняется переход на следующую экранную строку.

> **Примечание**. Имеются также перегруженные *шаблонные варианты* функций Show и ShowLine с параметрами-итераторами first и last, предназначенные для вывода элементов *стандартных контейнеров библиотеки STL* (см. [2]). Итераторы first и last, как обычно, определяют начальный выводимый элемент и позицию за конечным выводимым элементом. Данные варианты могут содержать необязательный третий параметр s типа std::string, определяющий комментарий, который указывается перед выводимой последовательностью элементов. Шаблонный вариант функции Show выводит элементы последовательности

на одной строке, разделяя их пробелами; после завершения вывода автоматически выполняется переход на новую строку. Шаблонный вариант ShowLine выводит каждый элемент последовательности на новой строке; строковый комментарий выводится в той же строке, что и первый элемент.

С выводом отладочной информации связаны также две вспомогательные функции: HideTask и SetPrecision.

Вызов функции void HideTask() обеспечивает скрытие всех разделов окна задачника, кроме раздела отладки. Если раздел отладки в окне задачника не отображается (в частности, если программа запущена в демонстрационном режиме), то вызов функции HideTask игнорируется.

Скрыть/восстановить основные разделы окна задачника после его отображения на экране можно с помощью комбинации [Ctrl]+[пробел] или соответствующей команды контекстного меню.

Функция void SetPrecision(int n) предназначена для настройки формата вывода вещественных отладочных данных. Если параметр n положителен, то он определяет количество выводимых дробных знаков; при этом число выводится в формате с фиксированной точкой. Если параметр n равен нулю или является отрицательным, то число выводится в формате с плавающей точкой (экспоненциальном формате). Число дробных знаков полагается равным *модулю* параметра n; если же параметр n равен нулю, то устанавливается количество дробных знаков, равное 6.

Действие текущей настройки числового формата, определенной функцией SetPrecision, продолжается до очередного вызова этой функции. До первого вызова функции SetPrecision вещественные числа выводятся в формате с фиксированной точкой и двумя дробными знаками.

4.2. Варианты индивидуальных заданий

4.2.1. Серии однотипных задач

Задачи, включенные в задачник PT for MPI-2 и представленные в разделе 2, можно разбить на серии из нескольких (от 2 до 4) однотипных задач. Ниже приводится список таких серий, содержащий все задачи из раздела 2, кроме тех, решения которых приведены в разделе 1. Дополнительно указывается сравнительная сложность каждой серии. Сложность оценивается в баллах (от 1 до 3), которые приводятся после имени группы задач и отделяются от него символом «-» (заметим, что задачи, оцениваемые в 2 и 3 балла, помечены в разделе 2 символами «*» и «**» соответственно).

```
-Процессы и их ранги (задача 2 решена в п.1.1)
MPI1Proc-1     1  3
MPI1Proc-1     4  5  6  7
MPI1Proc-1     8  9 10
```

```
-Блокирующая пересылка данных (задача 11 решена в п.1.2.2)
MPI2Send-1     1  2  3  4
MPI2Send-1     5  6  7
MPI2Send-1     8  9 10
MPI2Send-1    12 13 14
MPI2Send-1    15 16 17
MPI2Send-1    18 19
MPI2Send-1    20 21
MPI2Send-2    22 23
MPI2Send-2    24 25
-Неблокирующая пересылка данных
MPI2Send-2    26 27
MPI2Send-2    28 29
MPI2Send-2    30 31 32
-Коллективная пересылка данных
MPI3Coll-1     1  2  3
MPI3Coll-1     4  5  6  7
MPI3Coll-1     8  9 10
MPI3Coll-1    11 12 13
MPI3Coll-1    14 15
MPI3Coll-2    16 17 18
-Коллективные операции редукции (задача 23 решена в п.1.2.5)
MPI3Coll-1    19 20 21 22
MPI3Coll-1    24 25 26
MPI3Coll-1    27 28
-Использование простейших производных типов
MPI4Type-1     1  2  3
MPI4Type-1     4  5  6
MPI4Type-1     7  8
-Пересылка упакованных данных
MPI4Type-1     9 10 11
MPI4Type-1    12 13
-Более сложные виды производных типов (задача 14 решена в п.1.2.6)
MPI4Type-2    15 16
MPI4Type-2    17 18
MPI4Type-2    19 20
-Коллективная функция MPI_Alltoallw (MPI-2)
MPI4Type-3    21 22
-Создание новых коммуникаторов (задача 3 решена в п.1.2.7)
MPI5Comm-1     1  2
MPI5Comm-1     4  5
```

```
MPI5Comm-1     6  7
MPI5Comm-1     8  9
MPI5Comm-1    10 11 12
-Виртуальные топологии (задачи 17 и 29 решены в п.1.2.8 и 1.2.9)
MPI5Comm-1    13 14
MPI5Comm-1    15 16 18
MPI5Comm-1    19 20 21 22
MPI5Comm-1    23 24
MPI5Comm-1    25 26 27
MPI5Comm-3    28 30
-Топология распределенного графа (MPI-2)
MPI5Comm-3    31 32
-Локальный файловый ввод-вывод
MPI6File-1     1  2  3  4
MPI6File-1     5  6  7  8
-Коллективный файловый ввод-вывод
MPI6File-1     9 10
MPI6File-2    11 12
MPI6File-1    13 14
MPI6File-2    15 16
-Настройка вида файловых данных (задача 26 решена в п.1.3.3)
MPI6File-1    17 20
MPI6File-1    18 19
MPI6File-2    21 22
MPI6File-3    23 24 25
MPI6File-2    27 28
MPI6File-3    29 30
-Односторонние коммуникации (задача 13 решена в п.1.3.5)
MPI7Win-1      1  2
MPI7Win-1      3  4
MPI7Win-1      5  6
MPI7Win-2      7  8  9 10
MPI7Win-2     11 12
MPI7Win-2     14 15
MPI7Win-3     16 17
-Дополнительные виды синхронизации (задача 23 решена в п.1.3.6)
MPI7Win-1     18 19 20
MPI7Win-2     21 22
MPI7Win-1     24 25 26
MPI7Win-2     27 29
MPI7Win-2     28 30
```

```
-Создание интеркоммуникаторов (задача 9 решена в п.1.3.7)
MPI8Inter-1    1  2
MPI8Inter-2    3  4
MPI8Inter-2    5  6
MPI8Inter-3    7  8
-Коллективные операции для интеркоммуникаторов
MPI8Inter-2   10 11 12
MPI8Inter-2   13 14
-Динамическое создание процессов (задача 15 решена в п.1.3.8)
MPI8Inter-1   16
MPI8Inter-2   17 18
MPI8Inter-2   19 20
MPI8Inter-3   21 22
-Параллельные матричные алгоритмы (задача 1 решена в п.1.4.2)
-Определение вспомогательных типов и топологий
MPI9Matr-1    11 21 32
MPI9Matr-1    22 33 34
-Рассылка исходных данных (задача 2 решена в п.1.4.3)
MPI9Matr-2    12 23 35
-Реализация параллельных вычислений (задача 24 решена в п.1.4.4)
MPI9Matr-1     3 13 36 37
MPI9Matr-1     4 14 25 38
MPI9Matr-2     5 15 26 39
-Сборка результатов
MPI9Matr-2     6 16 27 40
-Реализация алгоритма в полном объеме
MPI9Matr-3     7 17 28 41
-Файловый ввод-вывод (MPI-2) (задача 19 решена в п.1.4.5)
MPI9Matr-2     8 18 29 42
MPI9Matr-2     9 30 43
MPI9Matr-3    10 20 31 44
```

Чтобы сформировать набор индивидуальных заданий, охватывающих все рассмотренные в книге возможности технологии MPI, достаточно выбрать по одной задаче из каждой серии однотипных задач. Сводная информация о полученных при этом наборах индивидуальных заданий приведена в таблице 5.

Каждый набор можно условно разделить на 4 части. В первую часть входят задачи из групп MPI1Proc, MPI2Send и MPI3Coll, которые посвящены базовым возможностям технологии MPI, связанным с пересылкой данных между процессами. Во вторую часть входят задачи, посвященные дополнительным возможностям технологии MPI (в основном, стандар-

та 1.1): определению новых типов данных и новых коммуникаторов, в том числе связанных с виртуальными топологиями. Третья часть содержит задачи, связанные с новыми возможностями стандарта MPI-2, а четвертая часть посвящена матричным алгоритмам.

Таблица 5

Сводная информация о наборах индивидуальных заданий

Группа задач	Количество задач	Общее число баллов
Часть 1	**24**	**30**
MPI1Proc	3	3
MPI2Send	12	17
MPI3Coll	9	10
Часть 2	**21**	**30**
MPI4Type	9	14
MPI5Comm	12	16
Часть 3	**34**	**60**
MPI6File	12	20
MPI7Win	12	20
MPI8Inter	10	20
Часть 4	**11**	**20**
MPI9Matr	11	20
Всего	**90**	**140**

В зависимости от количества часов, выделяемых на лабораторные занятия, и изучаемых тем, связанных с технологией MPI, в наборы индивидуальных заданий можно включать не все, а лишь некоторые из указанных частей.

4.2.2. Набор из 24 вариантов индивидуальных заданий

В данном пункте приводится набор из 24 вариантов индивидуальных заданий, сгенерированный на основе серий однотипных задач (см. п. 4.2.1). Для генерации вариантов использовалась программа PTVarMaker, входящая в состав комплекса для преподавателя программирования Teacher Pack. Данный комплекс является одним из расширений электронного задачника Programming Taskbook и включает программы-утилиты, предназначенные для использования при организации и проведении групповых лабораторных занятий. Описание комплекса Teacher Pack и, в частности, конструктора вариантов PTVarMaker приводится на сайте задачника ptaskbook.com (раздел «Teacher Pack»), а также в книге [3].

Если за выполнение задания дается более одного балла, то количество баллов указывается после номера задания в квадратных скобках. Если количество баллов после номера задания не указано, то считается, что оно равно 1.

Таблица 6

24 варианта индивидуальных заданий

ВАРИАНТ 1 MPI1Proc Процессы и их ранги: 1, 4, 9 MPI2Send(1) Блокирующая пересылка данных: 3, 6, 8, 12, 17, 18, 20, 22[2], 25[2] MPI2Send(2) Неблокирующая пересылка данных: 26[2], 28[2], 30[2] MPI3Coll(1) Коллективная пересылка данных: 2, 7, 9, 11, 14, 17[2] MPI3Coll(2) Коллективные операции редукции: 20, 24, 27 MPI4Type(1) Использование простейших производных типов: 2, 5, 7 MPI4Type(2) Пересылка упакованных данных: 9, 12 MPI4Type(3) Более сложные виды производных типов: 16[2], 18[2], 19[2] MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3] MPI5Comm(1) Создание новых коммуникаторов: 1, 4, 7, 9, 11 MPI5Comm(2) Виртуальные топологии: 13, 15, 20, 23, 27, 30[3] MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3] MPI6File(1) Локальный файловый ввод-вывод: 3, 6 MPI6File(2) Коллективный файловый ввод-вывод: 9, 12[2], 13, 16[2] MPI6File(3) Настройка вида файловых данных: 17, 18, 22[2], 24[3], 27[2], 29[3] MPI7Win(1) Односторонние коммуникации: 1, 4, 5, 7[2], 11[2], 15[2], 17[3] MPI7Win(2) Дополнительные виды синхронизации: 20, 21[2], 25, 29[2], 28[2] MPI8Inter(1) Создание интеркоммуникаторов: 2, 4[2], 6[2], 7[3] MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 12[2], 14[2] MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 19[2], 22[3] MPI9Matr(1) Определение вспомогательных типов и топологий: 32, 22 MPI9Matr(2) Рассылка исходных данных: 12[2] MPI9Matr(3) Реализация параллельных вычислений: 13, 14, 15[2] MPI9Matr(4) Сборка результатов: 27[2] MPI9Matr(5) Реализация алгоритма в полном объеме: 41[3] MPI9Matr(6) Файловый ввод-вывод (MPI-2): 8[2], 9[2], 10[3]
ВАРИАНТ 2 MPI1Proc Процессы и их ранги: 1, 7, 8 MPI2Send(1) Блокирующая пересылка данных: 3, 5, 9, 14, 15, 19, 20, 22[2], 25[2] MPI2Send(2) Неблокирующая пересылка данных: 27[2], 28[2], 32[2] MPI3Coll(1) Коллективная пересылка данных: 3, 4, 8, 12, 15, 18[2] MPI3Coll(2) Коллективные операции редукции: 20, 24, 27 MPI4Type(1) Использование простейших производных типов: 1, 6, 7 MPI4Type(2) Пересылка упакованных данных: 10, 12 MPI4Type(3) Более сложные виды производных типов: 15[2], 18[2], 20[2] MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3] MPI5Comm(1) Создание новых коммуникаторов: 2, 4, 7, 9, 11 MPI5Comm(2) Виртуальные топологии: 13, 16, 19, 24, 26, 30[3] MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3] MPI6File(1) Локальный файловый ввод-вывод: 4, 7 MPI6File(2) Коллективный файловый ввод-вывод: 10, 11[2], 13, 16[2] MPI6File(3) Настройка вида файловых данных: 17, 18, 22[2], 23[3], 28[2], 29[3] MPI7Win(1) Односторонние коммуникации: 2, 3, 6, 9[2], 12[2], 14[2], 16[3] MPI7Win(2) Дополнительные виды синхронизации: 20, 22[2], 26, 27[2], 30[2] MPI8Inter(1) Создание интеркоммуникаторов: 2, 4[2], 6[2], 7[3]

MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 12[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 19[2], 22[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 32, 34
MPI9Matr(2) Рассылка исходных данных: 35[2]
MPI9Matr(3) Реализация параллельных вычислений: 37, 25, 26[2]
MPI9Matr(4) Сборка результатов: 16[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 28[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 29[2], 43[2], 20[3]

ВАРИАНТ 3
MPI1Proc Процессы и их ранги: 1, 5, 10
MPI2Send(1) Блокирующая пересылка данных: 2, 6, 8, 14, 15, 18, 20, 22[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 27[2], 29[2], 32[2]
MPI3Coll(1) Коллективная пересылка данных: 1, 6, 9, 13, 15, 17[2]
MPI3Coll(2) Коллективные операции редукции: 19, 26, 27
MPI4Type(1) Использование простейших производных типов: 3, 5, 8
MPI4Type(2) Пересылка упакованных данных: 11, 13
MPI4Type(3) Более сложные виды производных типов: 16[2], 18[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 4, 6, 9, 10
MPI5Comm(2) Виртуальные топологии: 13, 18, 22, 23, 25, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 1, 5
MPI6File(2) Коллективный файловый ввод-вывод: 10, 11[2], 13, 15[2]
MPI6File(3) Настройка вида файловых данных: 20, 19, 21[2], 24[3], 27[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 2, 4, 5, 9[2], 12[2], 15[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 19, 21[2], 25, 27[2], 30[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 3[2], 5[2], 7[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 11[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 19[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 21, 22
MPI9Matr(2) Рассылка исходных данных: 35[2]
MPI9Matr(3) Реализация параллельных вычислений: 3, 25, 39[2]
MPI9Matr(4) Сборка результатов: 6[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 28[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 8[2], 9[2], 44[3]

ВАРИАНТ 4
MPI1Proc Процессы и их ранги: 1, 5, 9
MPI2Send(1) Блокирующая пересылка данных: 1, 5, 8, 12, 16, 19, 20, 22[2], 25[2]
MPI2Send(2) Неблокирующая пересылка данных: 27[2], 29[2], 30[2]
MPI3Coll(1) Коллективная пересылка данных: 2, 6, 8, 12, 15, 16[2]
MPI3Coll(2) Коллективные операции редукции: 22, 25, 28
MPI4Type(1) Использование простейших производных типов: 2, 6, 8
MPI4Type(2) Пересылка упакованных данных: 9, 12
MPI4Type(3) Более сложные виды производных типов: 15[2], 17[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 5, 7, 8, 11
MPI5Comm(2) Виртуальные топологии: 13, 15, 21, 24, 25, 30[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]

MPI6File(1) Локальный файловый ввод-вывод: 3, 6
MPI6File(2) Коллективный файловый ввод-вывод: 9, 11[2], 14, 16[2]
MPI6File(3) Настройка вида файловых данных: 17, 19, 22[2], 25[3], 27[2], 29[3]
MPI7Win(1) Односторонние коммуникации: 2, 3, 6, 10[2], 11[2], 15[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 18, 22[2], 24, 29[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 3[2], 5[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 11[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 20[2], 22[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 21, 33
MPI9Matr(2) Рассылка исходных данных: 23[2]
MPI9Matr(3) Реализация параллельных вычислений: 36, 4, 15[2]
MPI9Matr(4) Сборка результатов: 16[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 7[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 8[2], 43[2], 44[3]

ВАРИАНТ 5
MPI1Proc Процессы и их ранги: 1, 4, 8
MPI2Send(1) Блокирующая пересылка данных: 4, 7, 10, 13, 17, 19, 21, 22[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 29[2], 32[2]
MPI3Coll(1) Коллективная пересылка данных: 1, 5, 8, 12, 15, 17[2]
MPI3Coll(2) Коллективные операции редукции: 19, 24, 27
MPI4Type(1) Использование простейших производных типов: 1, 4, 7
MPI4Type(2) Пересылка упакованных данных: 10, 12
MPI4Type(3) Более сложные виды производных типов: 15[2], 18[2], 20[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 4, 7, 9, 12
MPI5Comm(2) Виртуальные топологии: 14, 16, 22, 24, 26, 30[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 2, 8
MPI6File(2) Коллективный файловый ввод-вывод: 9, 11[2], 14, 16[2]
MPI6File(3) Настройка вида файловых данных: 17, 19, 21[2], 24[3], 27[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 1, 3, 5, 7[2], 12[2], 14[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 18, 21[2], 26, 27[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 2, 3[2], 6[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 12[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 20[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 32, 34
MPI9Matr(2) Рассылка исходных данных: 12[2]
MPI9Matr(3) Реализация параллельных вычислений: 36, 4, 39[2]
MPI9Matr(4) Сборка результатов: 6[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 17[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 42[2], 30[2], 44[3]

ВАРИАНТ 6
MPI1Proc Процессы и их ранги: 3, 7, 8
MPI2Send(1) Блокирующая пересылка данных: 4, 7, 10, 13, 17, 18, 21, 23[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 27[2], 28[2], 31[2]
MPI3Coll(1) Коллективная пересылка данных: 1, 7, 9, 11, 14, 17[2]
MPI3Coll(2) Коллективные операции редукции: 22, 26, 28
MPI4Type(1) Использование простейших производных типов: 2, 4, 8

MPI4Type(2) Пересылка упакованных данных: 10, 13
MPI4Type(3) Более сложные виды производных типов: 15[2], 18[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 5, 6, 9, 10
MPI5Comm(2) Виртуальные топологии: 14, 15, 22, 24, 27, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 4, 6
MPI6File(2) Коллективный файловый ввод-вывод: 10, 12[2], 13, 16[2]
MPI6File(3) Настройка вида файловых данных: 20, 18, 21[2], 23[3], 28[2], 29[3]
MPI7Win(1) Односторонние коммуникации: 1, 4, 5, 10[2], 11[2], 14[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 19, 21[2], 25, 29[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 3[2], 5[2], 7[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 10[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 20[2], 22[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 21, 33
MPI9Matr(2) Рассылка исходных данных: 35[2]
MPI9Matr(3) Реализация параллельных вычислений: 36, 14, 15[2]
MPI9Matr(4) Сборка результатов: 40[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 28[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 18[2], 30[2], 20[3]

ВАРИАНТ 7

MPI1Proc Процессы и их ранги: 3, 6, 9
MPI2Send(1) Блокирующая пересылка данных: 3, 6, 9, 13, 15, 19, 21, 23[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 27[2], 28[2], 31[2]
MPI3Coll(1) Коллективная пересылка данных: 1, 4, 10, 11, 15, 16[2]
MPI3Coll(2) Коллективные операции редукции: 19, 25, 28
MPI4Type(1) Использование простейших производных типов: 1, 4, 7
MPI4Type(2) Пересылка упакованных данных: 11, 13
MPI4Type(3) Более сложные виды производных типов: 15[2], 18[2], 20[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 4, 7, 8, 12
MPI5Comm(2) Виртуальные топологии: 13, 16, 21, 23, 27, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 3, 8
MPI6File(2) Коллективный файловый ввод-вывод: 9, 11[2], 14, 15[2]
MPI6File(3) Настройка вида файловых данных: 20, 18, 21[2], 25[3], 28[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 1, 4, 5, 9[2], 11[2], 14[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 19, 22[2], 24, 29[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 2, 3[2], 5[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 10[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 19[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 11, 34
MPI9Matr(2) Рассылка исходных данных: 23[2]
MPI9Matr(3) Реализация параллельных вычислений: 37, 25, 26[2]
MPI9Matr(4) Сборка результатов: 16[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 7[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 29[2], 43[2], 10[3]

ВАРИАНТ 8
MPI1Proc Процессы и их ранги: 1, 6, 10
MPI2Send(1) Блокирующая пересылка данных: 1, 6, 10, 12, 15, 18, 20, 23[2], 25[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 29[2], 31[2]
MPI3Coll(1) Коллективная пересылка данных: 2, 4, 10, 13, 14, 18[2]
MPI3Coll(2) Коллективные операции редукции: 21, 25, 27
MPI4Type(1) Использование простейших производных типов: 1, 4, 8
MPI4Type(2) Пересылка упакованных данных: 11, 12
MPI4Type(3) Более сложные виды производных типов: 16[2], 17[2], 20[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 5, 6, 8, 12
MPI5Comm(2) Виртуальные топологии: 14, 15, 21, 24, 26, 30[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3]
MPI6File(1) Локальный файловый ввод-вывод: 1, 5
MPI6File(2) Коллективный файловый ввод-вывод: 9, 12[2], 13, 15[2]
MPI6File(3) Настройка вида файловых данных: 20, 18, 21[2], 23[3], 28[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 2, 3, 5, 8[2], 12[2], 14[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 18, 22[2], 25, 29[2], 30[2]
MPI8Inter(1) Создание интеркоммуникаторов: 2, 3[2], 5[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 11[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 19[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 11, 33
MPI9Matr(2) Рассылка исходных данных: 12[2]
MPI9Matr(3) Реализация параллельных вычислений: 13, 38, 5[2]
MPI9Matr(4) Сборка результатов: 40[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 41[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 18[2], 9[2], 20[3]

ВАРИАНТ 9
MPI1Proc Процессы и их ранги: 3, 6, 10
MPI2Send(1) Блокирующая пересылка данных: 2, 5, 10, 14, 16, 18, 21, 22[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 27[2], 29[2], 31[2]
MPI3Coll(1) Коллективная пересылка данных: 2, 5, 9, 13, 14, 16[2]
MPI3Coll(2) Коллективные операции редукции: 20, 25, 27
MPI4Type(1) Использование простейших производных типов: 3, 6, 7
MPI4Type(2) Пересылка упакованных данных: 10, 12
MPI4Type(3) Более сложные виды производных типов: 16[2], 17[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 5, 7, 8, 11
MPI5Comm(2) Виртуальные топологии: 14, 18, 20, 24, 25, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3]
MPI6File(1) Локальный файловый ввод-вывод: 2, 7
MPI6File(2) Коллективный файловый ввод-вывод: 10, 12[2], 13, 15[2]
MPI6File(3) Настройка вида файловых данных: 20, 19, 22[2], 23[3], 28[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 2, 3, 6, 8[2], 11[2], 15[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 18, 21[2], 24, 29[2], 30[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 4[2], 6[2], 7[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 10[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 20[2], 22[3]

MPI9Matr(1) Определение вспомогательных типов и топологий: 11, 34
MPI9Matr(2) Рассылка исходных данных: 23[2]
MPI9Matr(3) Реализация параллельных вычислений: 13, 4, 5[2]
MPI9Matr(4) Сборка результатов: 27[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 41[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 29[2], 9[2], 10[3]

ВАРИАНТ 10
MPI1Proc Процессы и их ранги: 3, 7, 8
MPI2Send(1) Блокирующая пересылка данных: 4, 7, 8, 12, 16, 19, 21, 23[2], 25[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 28[2], 32[2]
MPI3Coll(1) Коллективная пересылка данных: 3, 7, 8, 13, 15, 18[2]
MPI3Coll(2) Коллективные операции редукции: 21, 26, 28
MPI4Type(1) Использование простейших производных типов: 3, 5, 8
MPI4Type(2) Пересылка упакованных данных: 9, 13
MPI4Type(3) Более сложные виды производных типов: 16[2], 17[2], 20[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 5, 6, 8, 10
MPI5Comm(2) Виртуальные топологии: 14, 18, 20, 23, 27, 30[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3]
MPI6File(1) Локальный файловый ввод-вывод: 4, 5
MPI6File(2) Коллективный файловый ввод-вывод: 9, 12[2], 14, 15[2]
MPI6File(3) Настройка вида файловых данных: 17, 19, 22[2], 25[3], 28[2], 29[3]
MPI7Win(1) Односторонние коммуникации: 1, 3, 6, 8[2], 12[2], 15[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 20, 22[2], 24, 27[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 4[2], 5[2], 7[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 12[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 20[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 21, 33
MPI9Matr(2) Рассылка исходных данных: 23[2]
MPI9Matr(3) Реализация параллельных вычислений: 37, 38, 5[2]
MPI9Matr(4) Сборка результатов: 6[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 17[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 42[2], 30[2], 31[3]

ВАРИАНТ 11
MPI1Proc Процессы и их ранги: 3, 4, 9
MPI2Send(1) Блокирующая пересылка данных: 1, 7, 9, 13, 16, 18, 21, 23[2], 25[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 29[2], 30[2]
MPI3Coll(1) Коллективная пересылка данных: 3, 5, 10, 11, 14, 16[2]
MPI3Coll(2) Коллективные операции редукции: 22, 26, 28
MPI4Type(1) Использование простейших производных типов: 2, 5, 7
MPI4Type(2) Пересылка упакованных данных: 9, 13
MPI4Type(3) Более сложные виды производных типов: 16[2], 17[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 4, 6, 8, 12
MPI5Comm(2) Виртуальные топологии: 14, 18, 19, 23, 25, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 2, 8
MPI6File(2) Коллективный файловый ввод-вывод: 10, 12[2], 14, 16[2]

MPI6File(3) Настройка вида файловых данных: 17, 19, 21[2], 24[3], 27[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 2, 4, 6, 7[2], 12[2], 14[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 19, 22[2], 26, 27[2], 30[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 4[2], 6[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 10[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 19[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 32, 22
MPI9Matr(2) Рассылка исходных данных: 35[2]
MPI9Matr(3) Реализация параллельных вычислений: 3, 38, 39[2]
MPI9Matr(4) Сборка результатов: 40[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 17[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 42[2], 43[2], 31[3]

ВАРИАНТ 12
MPI1Proc Процессы и их ранги: 3, 5, 10
MPI2Send(1) Блокирующая пересылка данных: 2, 5, 9, 14, 17, 19, 20, 23[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 28[2], 30[2]
MPI3Coll(1) Коллективная пересылка данных: 3, 6, 10, 12, 14, 18[2]
MPI3Coll(2) Коллективные операции редукции: 21, 24, 28
MPI4Type(1) Использование простейших производных типов: 3, 6, 8
MPI4Type(2) Пересылка упакованных данных: 11, 13
MPI4Type(3) Более сложные виды производных типов: 15[2], 17[2], 20[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 5, 6, 9, 10
MPI5Comm(2) Виртуальные топологии: 13, 16, 19, 23, 26, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3]
MPI6File(1) Локальный файловый ввод-вывод: 1, 7
MPI6File(2) Коллективный файловый ввод-вывод: 10, 11[2], 14, 15[2]
MPI6File(3) Настройка вида файловых данных: 20, 18, 22[2], 25[3], 27[2], 29[3]
MPI7Win(1) Односторонние коммуникации: 1, 4, 6, 10[2], 11[2], 15[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 20, 21[2], 26, 27[2], 30[2]
MPI8Inter(1) Создание интеркоммуникаторов: 2, 4[2], 6[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 11[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 20[2], 22[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 11, 22
MPI9Matr(2) Рассылка исходных данных: 12[2]
MPI9Matr(3) Реализация параллельных вычислений: 3, 14, 26[2]
MPI9Matr(4) Сборка результатов: 27[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 7[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 18[2], 30[2], 31[3]

ВАРИАНТ 13
MPI1Proc Процессы и их ранги: 3, 6, 10
MPI2Send(1) Блокирующая пересылка данных: 1, 5, 8, 12, 15, 18, 20, 22[2], 25[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 28[2], 31[2]
MPI3Coll(1) Коллективная пересылка данных: 3, 7, 8, 13, 15, 16[2]
MPI3Coll(2) Коллективные операции редукции: 21, 25, 27
MPI4Type(1) Использование простейших производных типов: 2, 5, 8
MPI4Type(2) Пересылка упакованных данных: 10, 12
MPI4Type(3) Более сложные виды производных типов: 15[2], 17[2], 20[2]

MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 4, 7, 8, 12
MPI5Comm(2) Виртуальные топологии: 13, 16, 22, 24, 27, 30[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 3, 5
MPI6File(2) Коллективный файловый ввод-вывод: 9, 11[2], 13, 16[2]
MPI6File(3) Настройка вида файловых данных: 20, 18, 22[2], 24[3], 27[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 1, 4, 5, 9[2], 11[2], 15[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 18, 21[2], 26, 27[2], 30[2]
MPI8Inter(1) Создание интеркоммуникаторов: 2, 3[2], 6[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 11[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 20[2], 22[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 11, 33
MPI9Matr(2) Рассылка исходных данных: 12[2]
MPI9Matr(3) Реализация параллельных вычислений: 37, 14, 5[2]
MPI9Matr(4) Сборка результатов: 40[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 7[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 42[2], 30[2], 20[3]

ВАРИАНТ 14
MPI1Proc Процессы и их ранги: 3, 5, 8
MPI2Send(1) Блокирующая пересылка данных: 4, 7, 10, 14, 17, 19, 20, 23[2], 25[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 29[2], 32[2]
MPI3Coll(1) Коллективная пересылка данных: 1, 7, 10, 11, 15, 18[2]
MPI3Coll(2) Коллективные операции редукции: 22, 26, 28
MPI4Type(1) Использование простейших производных типов: 1, 4, 7
MPI4Type(2) Пересылка упакованных данных: 11, 12
MPI4Type(3) Более сложные виды производных типов: 16[2], 18[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 4, 6, 9, 12
MPI5Comm(2) Виртуальные топологии: 14, 16, 21, 24, 27, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3]
MPI6File(1) Локальный файловый ввод-вывод: 4, 8
MPI6File(2) Коллективный файловый ввод-вывод: 9, 11[2], 13, 16[2]
MPI6File(3) Настройка вида файловых данных: 17, 18, 21[2], 24[3], 27[2], 29[3]
MPI7Win(1) Односторонние коммуникации: 1, 4, 6, 8[2], 12[2], 14[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 20, 21[2], 25, 27[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 3[2], 5[2], 7[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 10[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 19[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 32, 34
MPI9Matr(2) Рассылка исходных данных: 35[2]
MPI9Matr(3) Реализация параллельных вычислений: 36, 14, 15[2]
MPI9Matr(4) Сборка результатов: 27[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 28[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 8[2], 30[2], 44[3]

ВАРИАНТ 15
MPI1Proc Процессы и их ранги: 1, 7, 9
MPI2Send(1) Блокирующая пересылка данных: 4, 7, 9, 12, 16, 18, 21, 23[2], 25[2]

MPI2Send(2) Неблокирующая пересылка данных: 27[2], 29[2], 32[2]
MPI3Coll(1) Коллективная пересылка данных: 2, 4, 9, 13, 15, 17[2]
MPI3Coll(2) Коллективные операции редукции: 19, 24, 27
MPI4Type(1) Использование простейших производных типов: 1, 6, 8
MPI4Type(2) Пересылка упакованных данных: 11, 12
MPI4Type(3) Более сложные виды производных типов: 16[2], 18[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 4, 6, 8, 11
MPI5Comm(2) Виртуальные топологии: 14, 16, 20, 23, 26, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3]
MPI6File(1) Локальный файловый ввод-вывод: 1, 6
MPI6File(2) Коллективный файловый ввод-вывод: 9, 11[2], 14, 16[2]
MPI6File(3) Настройка вида файловых данных: 17, 18, 21[2], 25[3], 28[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 1, 3, 5, 9[2], 11[2], 14[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 18, 21[2], 26, 27[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 2, 3[2], 5[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 11[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 19[2], 22[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 21, 34
MPI9Matr(2) Рассылка исходных данных: 12[2]
MPI9Matr(3) Реализация параллельных вычислений: 3, 25, 39[2]
MPI9Matr(4) Сборка результатов: 27[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 7[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 42[2], 9[2], 10[3]

ВАРИАНТ 16

MPI1Proc Процессы и их ранги: 3, 4, 10
MPI2Send(1) Блокирующая пересылка данных: 1, 7, 10, 14, 16, 19, 21, 22[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 27[2], 28[2], 30[2]
MPI3Coll(1) Коллективная пересылка данных: 2, 5, 9, 13, 14, 18[2]
MPI3Coll(2) Коллективные операции редукции: 20, 24, 28
MPI4Type(1) Использование простейших производных типов: 2, 5, 8
MPI4Type(2) Пересылка упакованных данных: 10, 12
MPI4Type(3) Более сложные виды производных типов: 15[2], 17[2], 20[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 4, 7, 8, 12
MPI5Comm(2) Виртуальные топологии: 14, 18, 22, 24, 27, 30[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3]
MPI6File(1) Локальный файловый ввод-вывод: 4, 5
MPI6File(2) Коллективный файловый ввод-вывод: 9, 11[2], 14, 15[2]
MPI6File(3) Настройка вида файловых данных: 20, 18, 21[2], 23[3], 27[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 2, 3, 6, 8[2], 12[2], 14[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 19, 22[2], 24, 29[2], 30[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 4[2], 6[2], 7[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 12[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 19[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 32, 22
MPI9Matr(2) Рассылка исходных данных: 35[2]
MPI9Matr(3) Реализация параллельных вычислений: 3, 25, 39[2]

MPI9Matr(4) Сборка результатов: 40[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 17[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 8[2], 43[2], 10[3]

ВАРИАНТ 17
MPI1Proc Процессы и их ранги: 1, 4, 10
MPI2Send(1) Блокирующая пересылка данных: 2, 5, 8, 13, 17, 19, 20, 22[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 27[2], 28[2], 32[2]
MPI3Coll(1) Коллективная пересылка данных: 1, 7, 8, 12, 15, 16[2]
MPI3Coll(2) Коллективные операции редукции: 21, 25, 28
MPI4Type(1) Использование простейших производных типов: 2, 6, 7
MPI4Type(2) Пересылка упакованных данных: 9, 12
MPI4Type(3) Более сложные виды производных типов: 15[2], 17[2], 20[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 5, 7, 9, 11
MPI5Comm(2) Виртуальные топологии: 13, 18, 19, 24, 26, 30[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3]
MPI6File(1) Локальный файловый ввод-вывод: 2, 6
MPI6File(2) Коллективный файловый ввод-вывод: 10, 12[2], 13, 16[2]
MPI6File(3) Настройка вида файловых данных: 17, 19, 22[2], 23[3], 28[2], 29[3]
MPI7Win(1) Односторонние коммуникации: 2, 3, 6, 10[2], 11[2], 15[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 20, 21[2], 26, 27[2], 30[2]
MPI8Inter(1) Создание интеркоммуникаторов: 2, 3[2], 5[2], 7[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 10[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 19[2], 22[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 11, 34
MPI9Matr(2) Рассылка исходных данных: 23[2]
MPI9Matr(3) Реализация параллельных вычислений: 36, 4, 26[2]
MPI9Matr(4) Сборка результатов: 6[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 17[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 29[2], 30[2], 31[3]

ВАРИАНТ 18
MPI1Proc Процессы и их ранги: 1, 7, 8
MPI2Send(1) Блокирующая пересылка данных: 3, 5, 10, 12, 16, 18, 21, 22[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 28[2], 30[2]
MPI3Coll(1) Коллективная пересылка данных: 3, 6, 8, 13, 15, 16[2]
MPI3Coll(2) Коллективные операции редукции: 19, 24, 27
MPI4Type(1) Использование простейших производных типов: 1, 6, 8
MPI4Type(2) Пересылка упакованных данных: 11, 13
MPI4Type(3) Более сложные виды производных типов: 16[2], 18[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 5, 7, 9, 10
MPI5Comm(2) Виртуальные топологии: 14, 18, 19, 23, 27, 30[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3]
MPI6File(1) Локальный файловый ввод-вывод: 2, 7
MPI6File(2) Коллективный файловый ввод-вывод: 10, 12[2], 13, 15[2]
MPI6File(3) Настройка вида файловых данных: 17, 19, 21[2], 25[3], 27[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 2, 4, 6, 7[2], 11[2], 14[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 19, 21[2], 24, 29[2], 30[2]

MPI8Inter(1) Создание интеркоммуникаторов: 2, 4[2], 5[2], 7[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 12[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 20[2], 22[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 32, 22
MPI9Matr(2) Рассылка исходных данных: 12[2]
MPI9Matr(3) Реализация параллельных вычислений: 13, 4, 5[2]
MPI9Matr(4) Сборка результатов: 16[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 28[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 18[2], 43[2], 20[3]

ВАРИАНТ 19
MPI1Proc Процессы и их ранги: 1, 6, 10
MPI2Send(1) Блокирующая пересылка данных: 2, 6, 9, 12, 17, 18, 21, 23[2], 25[2]
MPI2Send(2) Неблокирующая пересылка данных: 27[2], 28[2], 31[2]
MPI3Coll(1) Коллективная пересылка данных: 1, 6, 9, 11, 14, 17[2]
MPI3Coll(2) Коллективные операции редукции: 19, 25, 28
MPI4Type(1) Использование простейших производных типов: 3, 6, 8
MPI4Type(2) Пересылка упакованных данных: 10, 13
MPI4Type(3) Более сложные виды производных типов: 16[2], 18[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 4, 6, 8, 11
MPI5Comm(2) Виртуальные топологии: 14, 15, 20, 24, 25, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 2, 7
MPI6File(2) Коллективный файловый ввод-вывод: 10, 11[2], 13, 15[2]
MPI6File(3) Настройка вида файловых данных: 17, 19, 22[2], 24[3], 27[2], 29[3]
MPI7Win(1) Односторонние коммуникации: 2, 4, 6, 10[2], 12[2], 15[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 19, 22[2], 24, 29[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 3[2], 6[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 10[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 20[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 32, 22
MPI9Matr(2) Рассылка исходных данных: 23[2]
MPI9Matr(3) Реализация параллельных вычислений: 13, 14, 5[2]
MPI9Matr(4) Сборка результатов: 6[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 41[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 29[2], 30[2], 44[3]

ВАРИАНТ 20
MPI1Proc Процессы и их ранги: 1, 7, 9
MPI2Send(1) Блокирующая пересылка данных: 4, 6, 9, 13, 16, 18, 20, 23[2], 25[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 28[2], 31[2]
MPI3Coll(1) Коллективная пересылка данных: 3, 5, 8, 11, 14, 16[2]
MPI3Coll(2) Коллективные операции редукции: 21, 26, 27
MPI4Type(1) Использование простейших производных типов: 3, 5, 8
MPI4Type(2) Пересылка упакованных данных: 9, 13
MPI4Type(3) Более сложные виды производных типов: 15[2], 18[2], 20[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 4, 7, 9, 10
MPI5Comm(2) Виртуальные топологии: 13, 15, 21, 23, 25, 30[3]

MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 1, 8
MPI6File(2) Коллективный файловый ввод-вывод: 9, 12[2], 14, 15[2]
MPI6File(3) Настройка вида файловых данных: 20, 18, 22[2], 25[3], 28[2], 29[3]
MPI7Win(1) Односторонние коммуникации: 1, 4, 5, 8[2], 12[2], 14[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 19, 21[2], 25, 29[2], 30[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 4[2], 5[2], 7[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 11[2], 14[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 20[2], 22[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 21, 33
MPI9Matr(2) Рассылка исходных данных: 23[2]
MPI9Matr(3) Реализация параллельных вычислений: 3, 38, 39[2]
MPI9Matr(4) Сборка результатов: 40[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 41[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 8[2], 9[2], 31[3]

ВАРИАНТ 21
MPI1Proc Процессы и их ранги: 3, 4, 8
MPI2Send(1) Блокирующая пересылка данных: 2, 6, 9, 13, 15, 18, 21, 22[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 29[2], 31[2]
MPI3Coll(1) Коллективная пересылка данных: 2, 5, 10, 12, 14, 17[2]
MPI3Coll(2) Коллективные операции редукции: 22, 24, 27
MPI4Type(1) Использование простейших производных типов: 3, 4, 7
MPI4Type(2) Пересылка упакованных данных: 10, 12
MPI4Type(3) Более сложные виды производных типов: 15[2], 17[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 5, 6, 8, 11
MPI5Comm(2) Виртуальные топологии: 13, 15, 21, 23, 26, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 3, 5
MPI6File(2) Коллективный файловый ввод-вывод: 9, 12[2], 13, 16[2]
MPI6File(3) Настройка вида файловых данных: 20, 18, 21[2], 24[3], 27[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 2, 3, 5, 7[2], 11[2], 15[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 18, 22[2], 24, 27[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 3[2], 5[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 12[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 19[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 11, 33
MPI9Matr(2) Рассылка исходных данных: 23[2]
MPI9Matr(3) Реализация параллельных вычислений: 13, 25, 26[2]
MPI9Matr(4) Сборка результатов: 16[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 28[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 29[2], 43[2], 31[3]

ВАРИАНТ 22
MPI1Proc Процессы и их ранги: 3, 5, 9
MPI2Send(1) Блокирующая пересылка данных: 3, 7, 10, 14, 15, 19, 21, 23[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 27[2], 29[2], 32[2]
MPI3Coll(1) Коллективная пересылка данных: 2, 6, 10, 12, 14, 18[2]
MPI3Coll(2) Коллективные операции редукции: 22, 25, 28

MPI4Type(1) Использование простейших производных типов: 3, 4, 7
MPI4Type(2) Пересылка упакованных данных: 11, 13
MPI4Type(3) Более сложные виды производных типов: 15[2], 18[2], 19[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 5, 6, 8, 12
MPI5Comm(2) Виртуальные топологии: 13, 16, 22, 24, 26, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 4, 6
MPI6File(2) Коллективный файловый ввод-вывод: 10, 12[2], 14, 16[2]
MPI6File(3) Настройка вида файловых данных: 20, 19, 21[2], 25[3], 28[2], 30[3]
MPI7Win(1) Односторонние коммуникации: 2, 3, 5, 7[2], 12[2], 15[2], 17[3]
MPI7Win(2) Дополнительные виды синхронизации: 20, 22[2], 25, 29[2], 30[2]
MPI8Inter(1) Создание интеркоммуникаторов: 1, 4[2], 6[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 12[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 18[2], 20[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 11, 33
MPI9Matr(2) Рассылка исходных данных: 35[2]
MPI9Matr(3) Реализация параллельных вычислений: 36, 38, 26[2]
MPI9Matr(4) Сборка результатов: 16[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 41[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 42[2], 43[2], 44[3]

ВАРИАНТ 23
MPI1Proc Процессы и их ранги: 1, 5, 8
MPI2Send(1) Блокирующая пересылка данных: 1, 5, 8, 13, 15, 19, 20, 22[2], 24[2]
MPI2Send(2) Неблокирующая пересылка данных: 26[2], 29[2], 30[2]
MPI3Coll(1) Коллективная пересылка данных: 1, 4, 9, 11, 14, 17[2]
MPI3Coll(2) Коллективные операции редукции: 20, 26, 27
MPI4Type(1) Использование простейших производных типов: 1, 4, 7
MPI4Type(2) Пересылка упакованных данных: 9, 13
MPI4Type(3) Более сложные виды производных типов: 16[2], 17[2], 20[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 21[3]
MPI5Comm(1) Создание новых коммуникаторов: 1, 5, 6, 9, 10
MPI5Comm(2) Виртуальные топологии: 13, 15, 19, 23, 25, 28[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 31[3]
MPI6File(1) Локальный файловый ввод-вывод: 1, 7
MPI6File(2) Коллективный файловый ввод-вывод: 10, 11[2], 14, 15[2]
MPI6File(3) Настройка вида файловых данных: 17, 19, 22[2], 23[3], 28[2], 29[3]
MPI7Win(1) Односторонние коммуникации: 1, 3, 5, 9[2], 11[2], 14[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 18, 22[2], 25, 27[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 2, 4[2], 6[2], 7[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 10[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 19[2], 21[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 21, 22
MPI9Matr(2) Рассылка исходных данных: 12[2]
MPI9Matr(3) Реализация параллельных вычислений: 37, 4, 15[2]
MPI9Matr(4) Сборка результатов: 6[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 7[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 18[2], 9[2], 10[3]

ВАРИАНТ 24
MPI1Proc Процессы и их ранги: 3, 6, 9
MPI2Send(1) Блокирующая пересылка данных: 3, 6, 8, 14, 17, 19, 20, 23[2], 25[2]
MPI2Send(2) Неблокирующая пересылка данных: 27[2], 29[2], 30[2]
MPI3Coll(1) Коллективная пересылка данных: 3, 4, 10, 12, 15, 18[2]
MPI3Coll(2) Коллективные операции редукции: 20, 26, 28
MPI4Type(1) Использование простейших производных типов: 2, 5, 7
MPI4Type(2) Пересылка упакованных данных: 9, 13
MPI4Type(3) Более сложные виды производных типов: 16[2], 17[2], 20[2]
MPI4Type(4) Коллективная функция MPI_Alltoallw (MPI-2): 22[3]
MPI5Comm(1) Создание новых коммуникаторов: 2, 5, 7, 9, 10
MPI5Comm(2) Виртуальные топологии: 14, 18, 20, 23, 25, 30[3]
MPI5Comm(3) Топология распределенного графа (MPI-2): 32[3]
MPI6File(1) Локальный файловый ввод-вывод: 3, 8
MPI6File(2) Коллективный файловый ввод-вывод: 10, 12[2], 14, 15[2]
MPI6File(3) Настройка вида файловых данных: 20, 19, 22[2], 23[3], 28[2], 29[3]
MPI7Win(1) Односторонние коммуникации: 1, 4, 6, 10[2], 12[2], 15[2], 16[3]
MPI7Win(2) Дополнительные виды синхронизации: 20, 22[2], 26, 29[2], 28[2]
MPI8Inter(1) Создание интеркоммуникаторов: 2, 4[2], 6[2], 8[3]
MPI8Inter(2) Коллективные операции для интеркоммуникаторов: 11[2], 13[2]
MPI8Inter(3) Динамическое создание процессов: 16, 17[2], 20[2], 22[3]
MPI9Matr(1) Определение вспомогательных типов и топологий: 21, 34
MPI9Matr(2) Рассылка исходных данных: 35[2]
MPI9Matr(3) Реализация параллельных вычислений: 37, 38, 15[2]
MPI9Matr(4) Сборка результатов: 27[2]
MPI9Matr(5) Реализация алгоритма в полном объеме: 17[3]
MPI9Matr(6) Файловый ввод-вывод (MPI-2): 18[2], 9[2], 20[3]

Литература

1. *Абрамян М. Э.* Практикум по параллельному программированию с использованием электронного задачника Programming Taskbook for MPI. — Ростов н/Д: Изд-во ЮФУ, 2010. — 172 с.
2. *Абрамян М. Э.* Введение в стандартную библиотеку шаблонов C++. Описание, примеры использования, учебные задачи. — Ростов н/Д; Таганрог: Изд-во ЮФУ, 2017. — 178 с.
3. *Абрамян М. Э.* Инструменты и методы разработки электронных образовательных ресурсов по компьютерным наукам. — Ростов-н/Д; Таганрог: Изд-во ЮФУ, 2018. — 259 с.
4. *Абрамян М. Э., Штейнберг Б. Я., Штейнберг О. Б.* Технология распараллеливания программ. MPI и OpenMP, векторизация циклов, оптимизация использования памяти. — Ростов н/Д: Изд-во ЮФУ, 2014. — 148 с.
5. *Антонов А. С.* Технологии параллельного программирования MPI и OpenMP. — М.: Изд-во МГУ, 2012. — 344 с.
6. *Борзунов С. В., Кургалин С. Д., Флегель А. В.* Практикум по параллельному программированию. — СПб.: БХВ, 2017. — 236 с.
7. *Букатов А. А., Дацюк В. Н., Жегуло А. Н.* Программирование для многопроцессорных вычислительных систем. — Ростов н/Д: ЦВВР, 2003. — 208 с.
8. *Корнеев В. Д.* Параллельное программирование в MPI. — Новосибирск: Изд-во ИВМиМГ СО РАН, 2002. — 215 с.
9. *Немнюгин С. А., Стесик О. Л.* Параллельное программирование для многопроцессорных вычислительных систем. — СПб: БХВ–Петербург, 2002. — 396 с.
10. *Шпаковский Г. И., Серикова Н. В.* Программирование для многопроцессорных систем в стандарте MPI. — Минск: БГУ, 2002. — 323 с.
11. *MPI: A Message-Passing Interface Standard*. Version 1.1: June, 1995 / Message Passing Interface Forum, 2003. 238 pp. [Электронный ресурс] URL: http://mpi-forum.org/docs/mpi-1.1/mpi1-report.pdf (дата обращения: 10.02.2018).
12. *MPI: A Message-Passing Interface Standard*. Version 2.2 / Message Passing Interface Forum, 2009. 647 pp. [Электронный ресурс] URL: http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf (дата обращения: 10.02.2018).

Указатель

Указатель содержит все функции, типы и константы библиотеки MPI, описанные в данном учебнике, а также использованные в нем средства электронного задачника Programming Taskbook for MPI-2, конструктора учебных заданий PT4TaskMaker и стандартной библиотеки STL. В группе «Решения задач» перечислены все задачи из задачника Programming Taskbook for MPI-2, решения которых приведены в разделе 1.

mpi.h, 15, 16, 190
mpi.lib, 16, 190
MPI_2INT, 34
MPI_Accumulate, 114, 122, 128
MPI_Aint, 60
MPI_Allgather, 50
MPI_Allgatherv, 50
MPI_Allreduce, 53
MPI_Alltoall, 51
MPI_Alltoallv, 51
MPI_Alltoallw, 52
MPI_ANY_SOURCE, 35, 184
MPI_ANY_TAG, 35, 184
MPI_BAND, 52
MPI_Barrier, 48
MPI_Bcast, 48, 139, 146
MPI_BOR, 52
MPI_Bsend, 33
MPI_Bsend_init, 47
MPI_BSEND_OVERHEAD, 36
MPI_Buffer_attach, 36
MPI_Buffer_detach, 36
MPI_BXOR, 52
MPI_BYTE, 34, 110
MPI_CART, 81
MPI_Cart_coords, 84
MPI_Cart_create, 83
MPI_Cart_get, 90
MPI_Cart_rank, 84, 227
MPI_Cart_shift, 90
MPI_Cart_sub, 85, 87
MPI_Cartdim_get, 90
MPI_CHAR, 34
MPI_Close_port, 149
MPI_Comm, 12
MPI_Comm_accept, 148
MPI_Comm_compare, 73
MPI_Comm_connect, 148
MPI_Comm_create, 74, 138
MPI_Comm_disconnect, 149
MPI_Comm_dup, 73, 134
MPI_Comm_free, 76, 149
MPI_Comm_get_parent, 141
MPI_Comm_group, 74, 131
MPI_COMM_NULL, 12, 74, 76, 138, 141
MPI_Comm_rank, 17, 131
MPI_Comm_remote_group, 131
MPI_Comm_remote_size, 131, 135
MPI_COMM_SELF, 12
MPI_Comm_set_errhandler, 227
MPI_Comm_size, 17, 131
MPI_Comm_spawn, 139
MPI_Comm_spawn_multiple, 139, 141
MPI_Comm_split, 78, 132, 138
MPI_Comm_test_inter, 131
MPI_COMM_WORLD, 12, 141
MPI_CONGRUENT, 74
MPI_Datatype, 33
MPI_DATATYPE_NULL, 61
MPI_Dims_create, 89
MPI_DIST_GRAPH, 81
MPI_Dist_graph_create, 98
MPI_Dist_graph_create_adjacent, 98
MPI_Dist_graph_neighbors, 100
MPI_Dist_graph_neighbors_count, 100
MPI_DOUBLE, 34
MPI_DOUBLE_INT, 34
MPI_ERRCODES_IGNORE, 140
MPI_Errhandler_set, 227
MPI_ERROR, 35, 103
MPI_ERROR_RETURN, 227
MPI_File, 103, 108

MPI_File_close, 109
MPI_File_get_byte_offset, 104
MPI_File_get_position, 104
MPI_File_get_position_shared, 104
MPI_File_get_size, 104
MPI_File_iread, 102
MPI_File_iread_at, 102
MPI_File_iread_shared, 102
MPI_File_iwrite, 102
MPI_File_iwrite_at, 102
MPI_File_iwrite_shared, 102
MPI_FILE_NULL, 109
MPI_File_open, 108
MPI_File_read, 102, 103
MPI_File_read_all, 102, 103
MPI_File_read_all_begin, 102
MPI_File_read_all_end, 102
MPI_File_read_at, 102
MPI_File_read_at_all, 102
MPI_File_read_at_all_begin, 102
MPI_File_read_at_all_end, 102
MPI_File_read_ordered, 102, 103, 104
MPI_File_read_ordered_begin, 102
MPI_File_read_ordered_end, 102
MPI_File_read_shared, 102, 103, 104
MPI_File_seek, 103
MPI_File_seek_shared, 103
MPI_File_set_size, 104
MPI_File_set_view, 109
MPI_File_write, 102, 103
MPI_File_write_all, 102, 103, 111
MPI_File_write_all_begin, 102
MPI_File_write_all_end, 102
MPI_File_write_at, 102
MPI_File_write_at_all, 102
MPI_File_write_at_all_begin, 102
MPI_File_write_at_all_end, 102
MPI_File_write_ordered, 102, 103, 104
MPI_File_write_ordered_begin, 102
MPI_File_write_ordered_end, 102
MPI_File_write_shared, 102, 103, 104
MPI_Finalize, 18, 191
MPI_Finalized, 18
MPI_FLOAT, 34
MPI_Gather, 49, 79, 139
MPI_Gatherv, 49
MPI_Get, 114, 127
MPI_Get_count, 35, 103
MPI_GRAPH, 81
MPI_Graph_create, 93
MPI_Graph_get, 96
MPI_Graph_neighbors, 96
MPI_Graph_neighbors_count, 96
MPI_Graphdims_get, 96
MPI_Group, 74
MPI_Group_compare, 75
MPI_Group_difference, 76
MPI_GROUP_EMPTY, 74, 76
MPI_Group_excl, 75
MPI_Group_free, 76
MPI_Group_incl, 75
MPI_Group_intersection, 76
MPI_GROUP_NULL, 74, 76
MPI_Group_range_excl, 75
MPI_Group_range_incl, 75
MPI_Group_rank, 74
MPI_Group_size, 74
MPI_Group_translate_ranks, 75
MPI_Group_union, 76
MPI_Ibsend, 45
MPI_IDENT, 73, 75
MPI_Info, 109
MPI_INFO_NULL, 98, 109, 114, 149
MPI_Init, 17, 140, 191
MPI_Initialized, 16
MPI_INT, 34
MPI_Intercomm_create, 133
MPI_Intercomm_merge, 147
MPI_Iprobe, 46
MPI_Irecv, 45
MPI_Irsend, 45
MPI_Isend, 45
MPI_Issend, 45
MPI_LAND, 52
MPI_LB, 62
MPI_LOCK_EXCLUSIVE, 118
MPI_LOCK_SHARED, 118
MPI_LONG, 34
MPI_Lookup_name, 148
MPI_LOR, 52
MPI_LXOR, 52
MPI_MAX, 52
MPI_MAXLOC, 52, 56
MPI_MIN, 52, 128
MPI_MINLOC, 52, 56
MPI_MODE_APPEND, 108
MPI_MODE_CREATE, 108
MPI_MODE_DELETE_ON_CLOSE, 108

MPI_MODE_NOPUT, 116
MPI_MODE_NOSUCCEED, 116
MPI_MODE_RDONLY, 108
MPI_MODE_RDWR, 108
MPI_MODE_WRONLY, 108
MPI_Offset, 103
MPI_Op, 52, 115
MPI_Op_create, 53
MPI_Open_port, 148
MPI_Pack, 63
MPI_PACK, 64
MPI_Pack_size, 64
MPI_Probe, 35
MPI_PROC_NULL, 90, 139, 145
MPI_PROD, 52
MPI_Publish_name, 148
MPI_Put, 114
MPI_Recv, 34, 136
MPI_Recv_init, 47
MPI_Reduce, 53, 145
MPI_Reduce_scatter, 54
MPI_Reduce_scatter_block, 54
MPI_Request, 44
MPI_Request_free, 47
MPI_REQUEST_NULL, 45
MPI_ROOT, 139, 145, 146
MPI_Rsend, 33
MPI_Rsend_init, 47
MPI_Scan, 54
MPI_Scatter, 50, 86, 139
MPI_Scatterv, 50
MPI_SEEK_CUR, 103
MPI_SEEK_END, 103
MPI_SEEK_SET, 103
MPI_Send, 33, 136
MPI_Send_init, 47
MPI_Sendrecv, 36, 96
MPI_Sendrecv_replace, 36
MPI_SIMILAR, 74, 75
MPI_SOURCE, 35
MPI_Ssend, 33
MPI_Ssend_init, 47
MPI_Start, 47
MPI_Startall, 47
MPI_Status, 35, 103, 184
MPI_STATUS_IGNORE, 35
MPI_SUCCESS, 17
MPI_SUM, 52, 122
MPI_TAG, 35
MPI_TAG_UB, 33
MPI_Test, 46
MPI_Testall, 46
MPI_Testany, 46
MPI_Testsome, 46
MPI_Topo_test, 81
MPI_Type_commit, 61
MPI_Type_contiguous, 59
MPI_Type_create_hindexed, 60
MPI_Type_create_hvector, 60
MPI_Type_create_indexed_block, 60
MPI_Type_create_resized, 63
MPI_Type_create_struct, 60
MPI_Type_extent, 62
MPI_Type_free, 61
MPI_Type_get_extent, 63
MPI_Type_hindexed, 60
MPI_Type_hvector, 60
MPI_Type_indexed, 60
MPI_Type_size, 62
MPI_Type_struct, 60
MPI_Type_vector, 59
MPI_UB, 62
MPI_UNDEFINED, 17, 74, 81, 133
MPI_UNEQUAL, 74, 75
MPI_Unpack, 64
MPI_Unpublish_name, 149
MPI_UNWEIGHTED, 98
MPI_User_function, 53
MPI_Wait, 45
MPI_Waitall, 45
MPI_Waitany, 45
MPI_Waitsome, 45
MPI_Win, 113
MPI_Win_complete, 117, 126
MPI_Win_create, 113, 121, 126
MPI_Win_fence, 116, 122
MPI_Win_free, 121, 129
MPI_Win_lock, 118
MPI_Win_post, 117, 126
MPI_Win_start, 117, 126
MPI_Win_test, 117
MPI_Win_unlock, 118
MPI_Win_wait, 117, 126
MPI_Wtick, 47
MPI_Wtime, 47
mpich.lib, 16, 190
mpicxx.h, 190
mpiio.h, 190

Библиотека STL
<algorithm>, 72
<vector>, 71
assign, 159
begin, 72, 159
copy, 72, 79, 159
end, 72, 79, 159
vector<T>, 70, 79, 158
Задачник PT for MPI-2
GetExename, 253
HideTask, 182, 334
pt, 24, 328
pt4.h, 15, 16, 193, 327, 328
pt4console.h, 195, 326
pt4null.h, 193
ptin_iterator<T>, 72, 328
ptout_iterator<T>, 72, 328
SetPrecision, 334
Show, 31, 332
ShowAll, 196
ShowLine, 31, 332
Task, 15, 327
Конструктор учебных заданий
Center, 315
CreateGroup, 305
CreateTask, 299, 310
CurrentLanguage, 306
DataComment, 315
DataN, 312
DataR, 312
InitTask, 305
inittaskgroup, 305
lgCPP, 306
pt4taskmaker.h, 299, 304, 326
RandomN, 300, 310, 312
RandomR, 312
ResultComment, 315
ResultN, 312
ResultR, 312
SetPrecision, 312
SetProcess, 302
SetTestCount, 313
TaskText, 311
UseTask, 305
xCenter, 312
xLeft, 312, 315
xRight, 312, 315
Решения задач
MPI1Proc2, 11
MPI2Send11, 37
MPI3Coll23, 55
MPI4Type14, 64
MPI5Comm17, 82
MPI5Comm29, 91
MPI5Comm3, 77
MPI6File26, 105
MPI7Win13, 119
MPI7Win23, 124
MPI8Inter15, 142
MPI8Inter9, 131
MPI9Matr1, 154
MPI9Matr19, 171
MPI9Matr2, 159
MPI9Matr24, 165