Часть III
Разработка учебных заданий

1. Особенности функционирования задачника в параллельном режиме

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 программы программа-загрузчик обеспечивает случайный выбор количества выполняемых процессов.

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 позволяет сделать для учащегося более наглядным способ запуска параллельной программы; кроме того, из параметров командной строки учащийся может легко определить, сколько процессов параллельной программы запущено. Еще более важной является 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. Заметим, что описание действий, требующихся для прерывания выполнения пакетного файла (а также действий, связанных с настройкой некоторых параметров выполнения учебных программ в параллельным режиме), также приводится в его консольном окне (рис. 33).

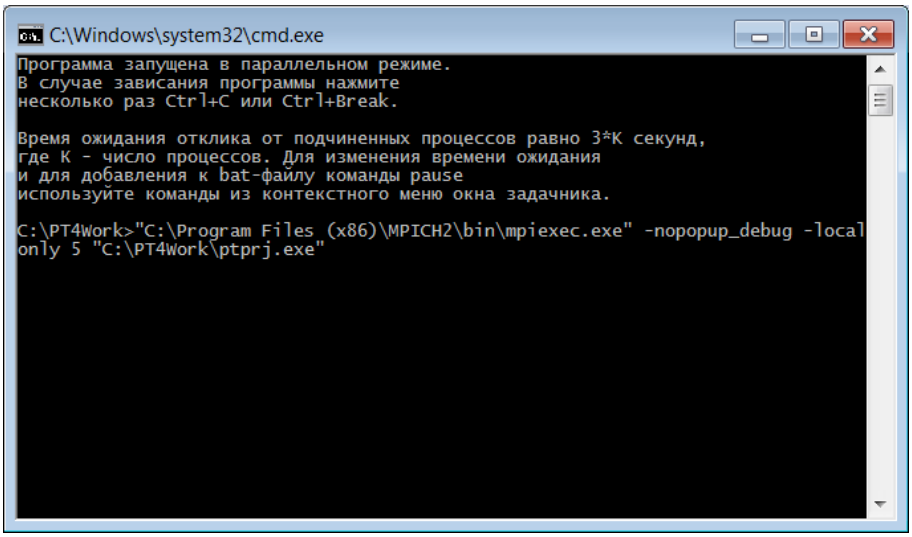

Рис. 1. Консольное окно с информацией о запуске параллельной программы

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). Последнее действие необходимо 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 этими 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, связанное 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 (по умолчанию 10 с) 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.

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. 13.2).

После завершения пересылки данных каждый процесс выполняет свою часть алгоритма решения задания, 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 программе учащегося. 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 потребуется явным образом прервать выполнение зависших процессов 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 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 $K-1$ до процесса 1, где K — общее число процессов). Если в различных 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 начальной части раздела 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. Как и для обычных, «непараллельных» заданий, входящих в базовый вариант задачника Programming Taskbook, 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 одна часть 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 не-

сколько 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. 34):

- **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.

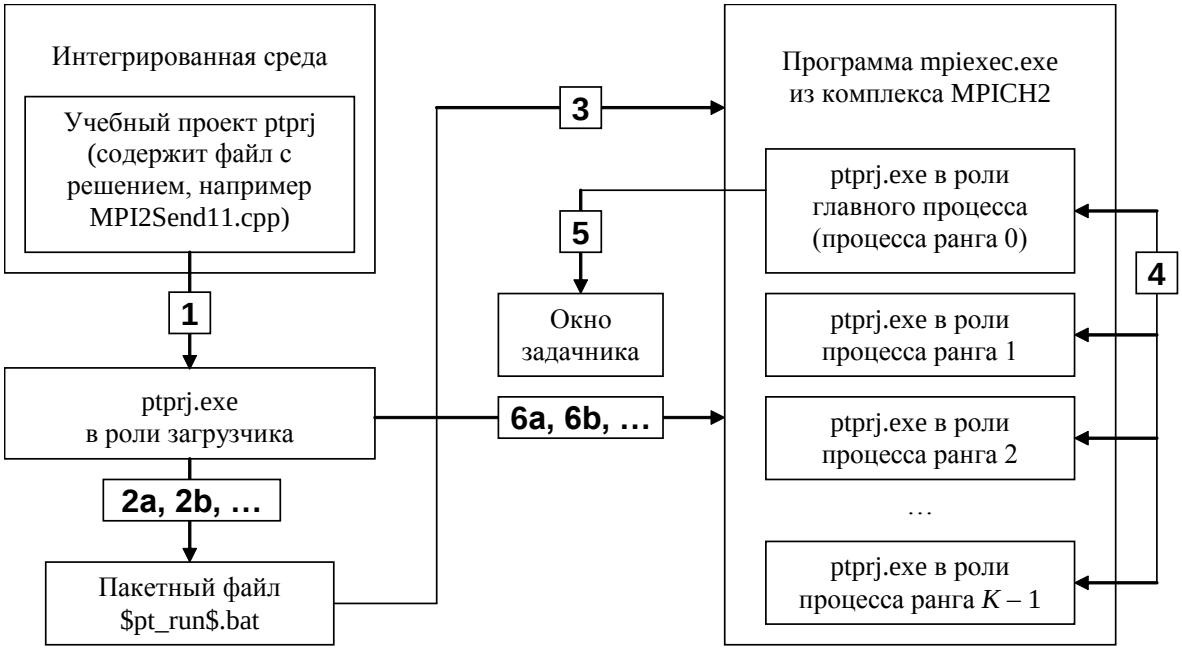

Рис. 2. Схема выполнения задания по параллельному программированию с применением электронного задачника Programming Taskbook for MPI-2

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 PT4 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, определенного в файле pt4taskmaker.cpp; описания этих функций содержатся в заголовочном файле pt4taskmaker.h.

В следующих двух пунктах 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). Изучить многие из возможностей конструктора можно по приводимым далее примерам их применения (п. 14).

Следует отметить, что при разработке заданий по параллельному программированию не требуется использовать систему MPICH2.

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, которые будут использованы при инициализации задания 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 1.

Таблица 1

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

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 *N*:» перед теми исходными или результирующими данными, которые связаны с процессом ранга *N*.

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. 6).

3. Примеры разработки заданий по параллельному программированию

В данном разделе приводятся примеры разработки заданий по параллельному программированию с использованием конструктора учебных заданий PT4TaskMaker.

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 2015. 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. 10.3). 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…»;
- в появившемся окне выберем раздел «Visual C++» и в нем вариант «Win32 Project»;
- в качестве имени проекта укажем PT4MPIDemo, в качестве каталога размещения проекта укажем c:\PT4Work, снимем выделение с флажка «Create directory for solution», если этот флажок установлен;
- после нажатия кнопки «ОК» на экране появится окно «Win32 Application Wizard», в котором следует сразу нажать кнопку «Next», установить в появившемся списке настроек вариант DLL и *снять все флажки в разделе «Additional options»*, после чего нажать кнопку «Finish».

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…».

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 один параметр целого типа num, не возвращает результаты и должна снабжаться модификатором _stdcall;
- *функция инициализации группы заданий*, которая должна иметь фиксированное имя inittaskgroup (все буквы строчные); эта функция не имеет параметров, не возвращает результаты 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(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. 10). 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 описа-

ние, 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 задачником Programming Taskbook. Поскольку задачник PT4 for MPI-2 ориентирован только на язык C++, необходимо добавить в функцию inittaskgroup фрагмент, позволяющий сделать группу доступной только для указанных языков (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 указать для тестирования нашей библиотеки:

```
-gMPIDemo -n999
```

Первый из параметров определяет используемую 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...»;
- выберите в появившемся окне вариант «Release» в качестве значения настройки «Configuration»;
- перейдите в раздел «Debugging» и заполните значения трех первых настроек следующим образом (мы предполагаем, что задачник Programming Taskbook размещен в каталоге c:\Program files (x86)\PT4):

Command	c:\Program files (x86)\PT4\PT4Demo.exe
Command Arguments	-gMPIDemo -n999
Working Directory	.\Release

После этого закройте окно свойств, нажав кнопку «ОК».

Теперь мы можем протестировать заготовку нашей библиотеки. При запуске нашей программы (например, по нажатию клавиши F5) она должна успешно откомпилироваться, после чего на экране должно появиться окно модуля PT4Demo с выбранной группой MPIDemo, а также окно задачника с заданием MPIDemo1 в демонстрационном режиме (рис. 35).

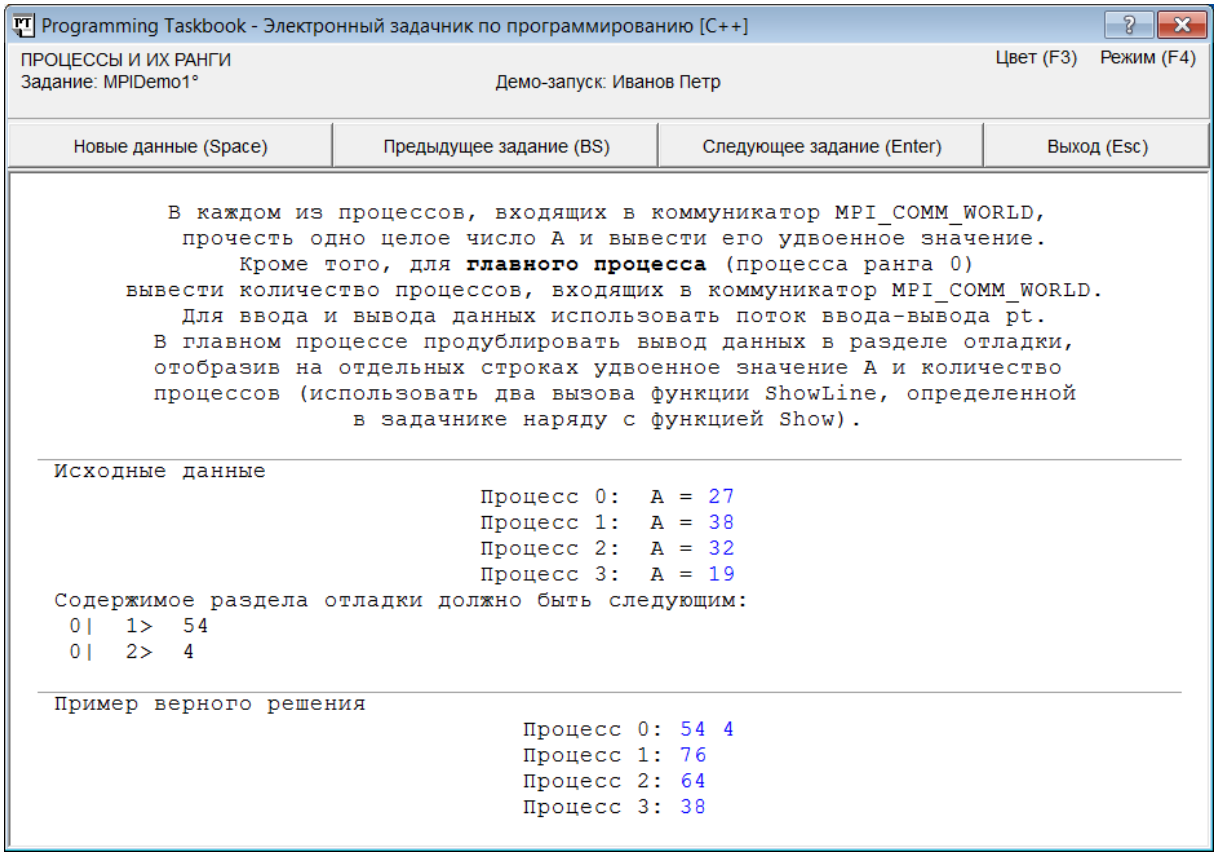

Рис. 3. Демонстрационный запуск задания MPIDemo1

Задание MPIDemo1 будет совпадать с заданием MPI1Proc2. Возможность подобного импортирования заданий полезна, если требуется исклю-

чить 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).

3.2. Разработка простого задания: MPIDemo2

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), и 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 = RandomR(-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. 36 (количество процессов, как обычно, мо-

жет быть другим). Используя кнопки или клавиши «Пробел», Enter и Backspace, можно просматривать различные варианты исходных и контрольных данных для этого задания или переходить к другим заданиям группы MPIDemo.

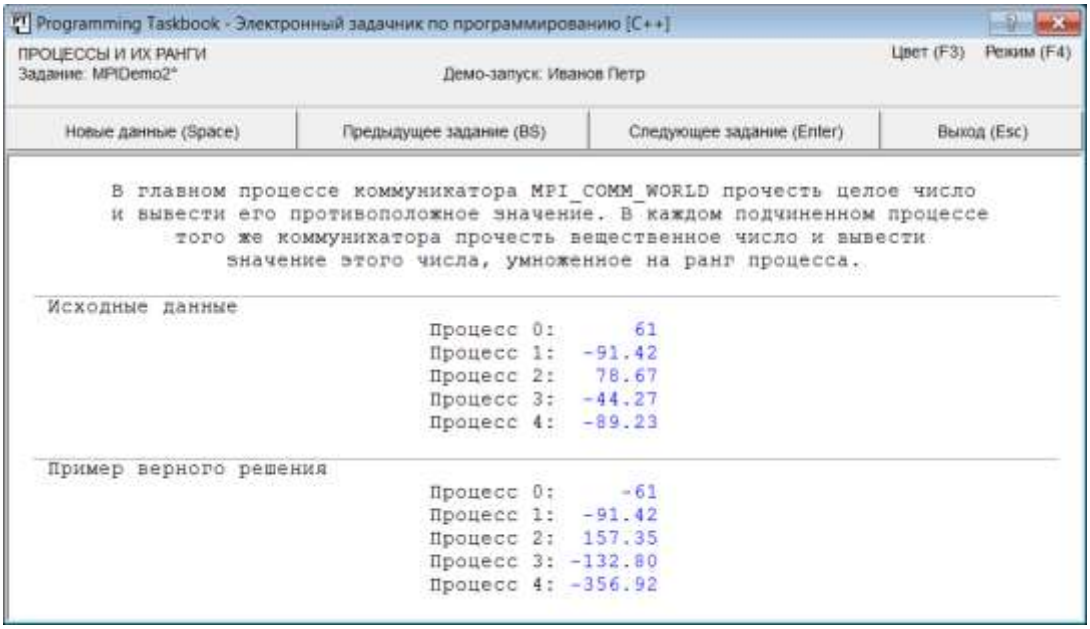

Рис. 4. Демонстрационный запуск задания MPIDemo2

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. 12). 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 тре-

буется. 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 выполняется в главном процессе: 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 «\»; 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 указываются позиции, по которым 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 определением данных для каждого процесса вызывается функция SetProcess, определяющая текущий процесс задания (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:

- комментарий к определяемому элементу данных (этот параметр может отсутствовать),
- значение элемента данных,
- горизонтальная координата X, определяющая позицию вывода элемента 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); 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;
- вертикальная координата Y, определяющая номер строки соответствующего раздела, в которой будет выведен элемент (строки нумеруются от 1);
- ширина области вывода элемента (комментарий располагается слева от области вывода; элементы выравниваются по правой границе области вывода).

При определении ширины вывода 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. 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 останавливаться на описании решения задания MPIDemo2, поскольку оно очень похоже на решение задания MPI1Proc2, описанное в п. 10.

3.3. Разработка задания, связанного с пересылкой данных: MPIDemo3

Теперь разработаем 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, …, $K-2$) в процесс 0, а числа из всех процессов нечетного ранга (1, 3, …, $K-1$) — 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. 37) и одноименную функцию, реализующую это задание.

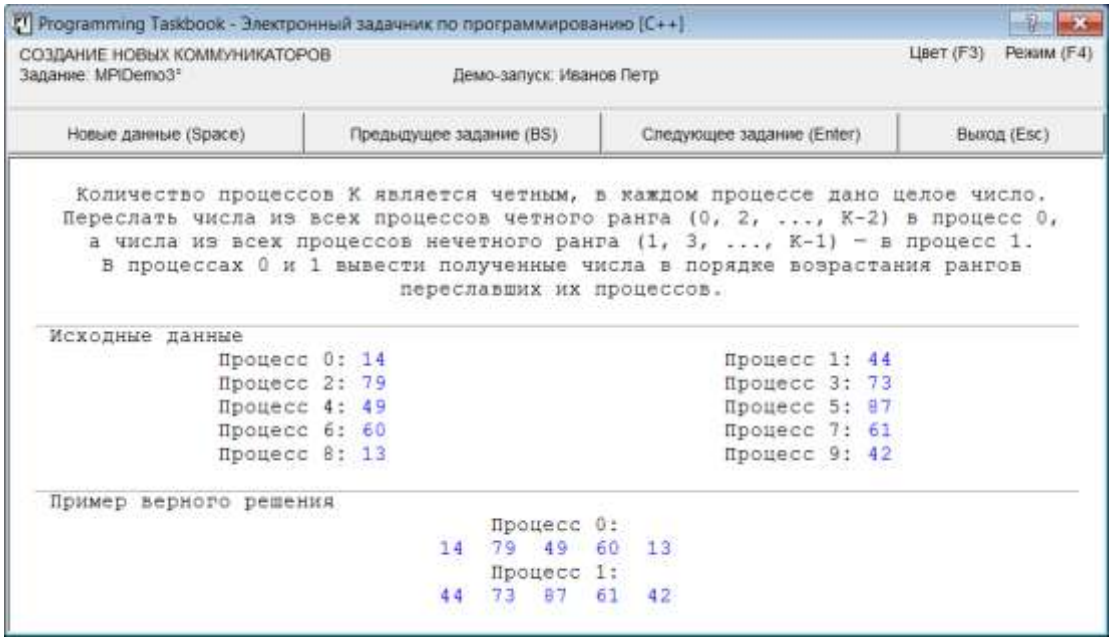

Рис. 5. Демонстрационный запуск задания MPIDemo3

```
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 эффективное его решение связано 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 html-документе (см. п. 14.4). 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 уже было отмечено выше, при решении целесообразно использовать вспомогательные коммуникаторы (приведем только заключительную часть функции Solve, которая 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 одинаковой 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.

3.4. Разработка сложного задания: MPIDemo4

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).

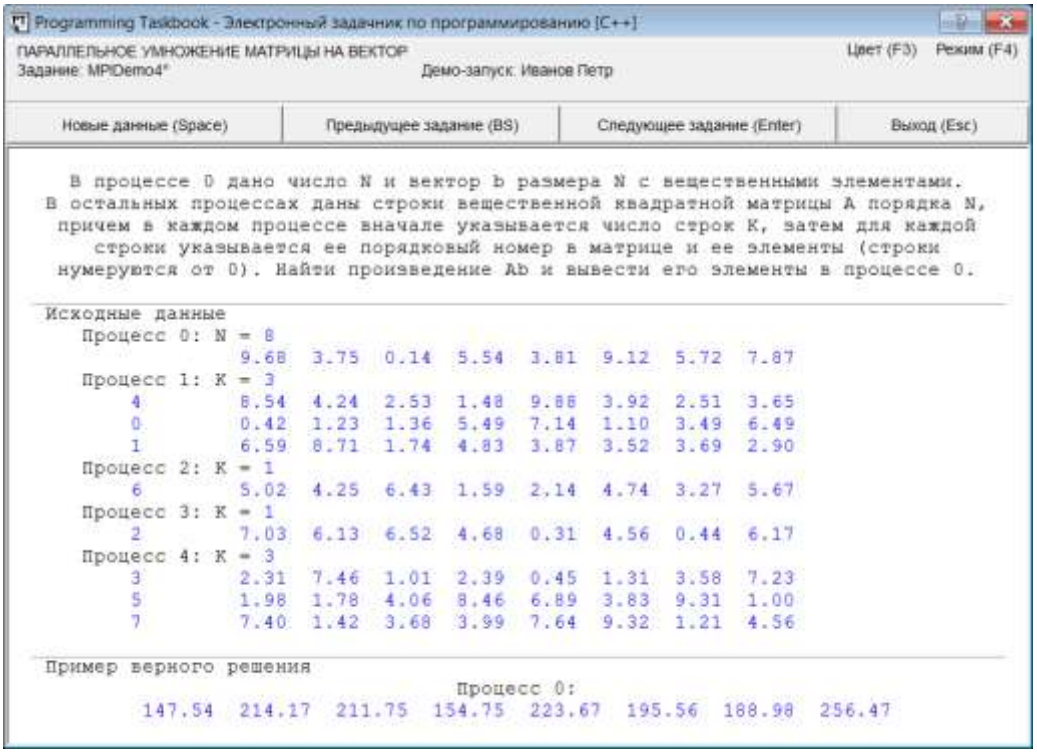

Рис. 6. Демонстрационный запуск задания MPIDemo4

Приведем вид окна задачника для задания MPIDemo4 (рис. 38) и одноименную функцию, реализующую это задание, вместе со вспомогатель-

ной 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;
}
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
// процессу (в процесс i передаются k[i] строк матрицы)
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 результирующих данных для процесса 0
SetProcess(0);
DataN("Процесс 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("Процесс 0:", 0, 2);
for (int i = 0; i < n; ++i)
  ResultR(c[i], Center(i + 1, n, 6, 2), 3, 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-получателя.

Заметим, что задание MPIDemo4 представляет собой центральный фрагмент самопланирующего алгоритма умножения матрицы на вектор, описанного в п. 11.5. 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.

Завершая описание процесса разработки новой группы заданий, отметим, что 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» 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. 39).

Примечание. 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#")`. Заметим, что если указать в параметре символ #, не удаляя номер задания (например `Task("MPIDemo4#")`), то в html-описание будет включено только задание с указанным номером.

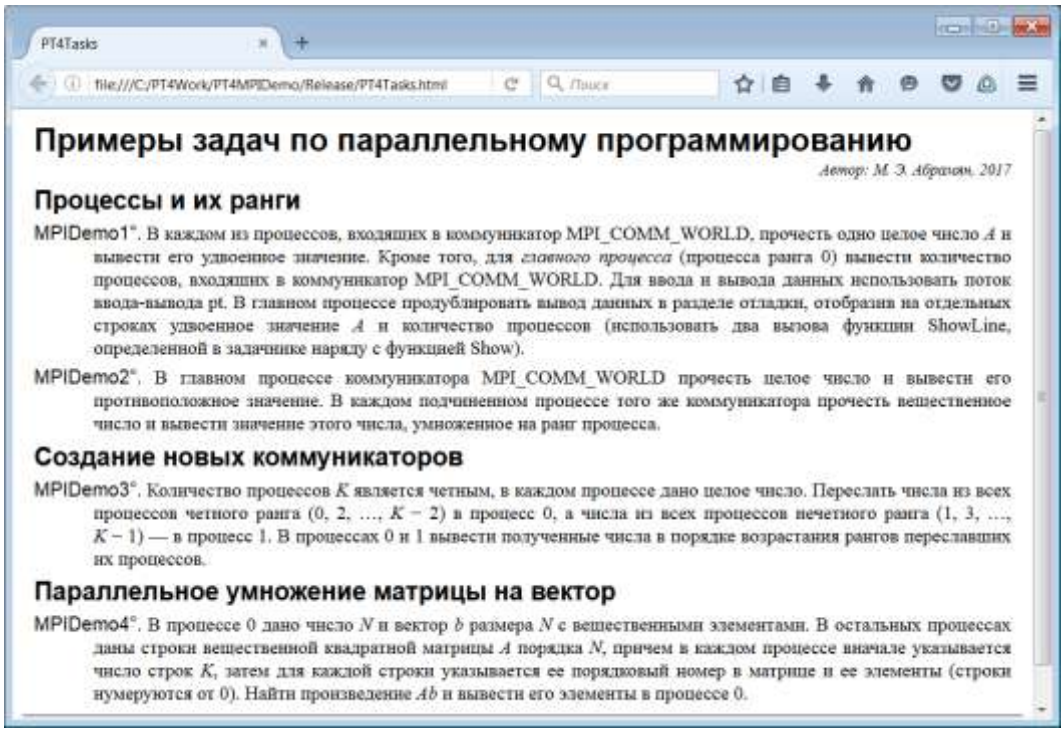

Рис. 7. Страница html-браузера с описанием группы заданий MPIDemo