

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«КЕМЕРОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ» (КемГУ)

С. В. Стуколов

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ
практикум

Издатель:
Кемеровский государственный университет

© С. В. Стуколов, 2020
© Кемеровский государственный
университет, 2020

ISBN 978-?-????- ????-?

Об издании – 1, 2, 3

Кемерово 2020

УДК 004.032.24 (076.5)

С 88

*Издается по решению научно-методического совета
Кемеровского государственного университета*

Рецензенты:

Федосенко Б.А. – доктор техн. наук, профессор кафедры «Информационные и автоматизированные производственные системы» Кузбасского государственного технического университета им. Т.Ф. Горбачева (КузГТУ).

Макарчук Р.С. – канд. физ.-мат. наук, доцент кафедры «Математики, физики и информационных технологий» Кузбасской государственной сельскохозяйственной академии.

Автор:

Стуколов Сергей Владимирович – канд. физ.-мат. наук, доцент кафедры ЮНЕСКО по информационным вычислительным технологиям КемГУ.

Стуколов, С. В.

С 88 Параллельное программирование: практикум [Электронный ресурс] / С. В. Стуколов; Кемеровский государственный университет. – Электрон. дан. (объем 4 Мб). – Кемерово: КемГУ, 2020. – 1 электрон. опт. диск (CD-ROM). – Систем. требования: Intel Pentium (или аналогичный процессор других производителей), 1,2 ГГц; 512 Мб оперативной памяти; видеокарта SVGA, 1280x1024 HighColor (32 bit); 5 Мб свободного дискового пространства; операц. система версии не ниже Windows 7 или Linux; AdobeReader. – Загл. с экрана.

ISBN 978-?-????-????-?

Учебное издание разработано по дисциплине «Технологии параллельного программирования». В нем представлены сведения по наиболее распространенной библиотеке параллельного программирования – Message Passing Interface (MPI). Изложение материала сопровождается примерами использования функций MPI с подробным разбором параллельных программ, а также заданиями для самостоятельного выполнения к каждой теме. Для проверки навыков использования библиотеки MPI предложены варианты контрольных и индивидуальных заданий.

Учебное издание предназначено для обучающихся высших учебных заведений по направлениям подготовки 01.03.02 Прикладная математика и информатика, 02.03.02 Фундаментальная информатика и информационные технологии, 02.03.03 Математическое обеспечение и администрирование информационных систем, а также может быть интересно студентам других направлений подготовки, аспирантам, научным сотрудникам и преподавателям вузов, занимающихся численным моделированием с применением многопроцессорной (многоядерной) вычислительной техники.

ISBN 978-?-????-????-?

УДК 004.032.24 (076.5)

ББК 3973.2-018я73

© С. В. Стуколов, 2020

© Кемеровский государственный
университет, 2020

Текстовое электронное издание

Минимальные системные требования:

Компьютер: Intel Pentium (или аналогичный процессор других производителей), 1,2 ГГц; 512 Мб оперативной памяти; видеокарта SVGA, 1280x1024 HighColor (32 bit); 5 Мб свободного дискового пространства; привод CD-ROM.

Операционная система: Windows 7 и выше, Linux.

Программное обеспечение: Adobe Reader

© С. В. Стуколов, 2020

© Кемеровский государственный
университет, 2020

ОГЛАВЛЕНИЕ

Введение	8
1. Знакомство с технологией Message Passing Interface: обрамляющие функции	13
1.1. Компиляция и запуск параллельных программ в UNIX-подобных операционных системах	13
1.2. Установка и настройка библиотеки MPI в операционной системе Windows	16
1.3. Обрамляющие функции MPI	19
1.4. Примеры.....	21
Задания	23
2. Передача данных с помощью блокирующих коммуникационных функций типа “Точка-Точка”	25
2.1. Функции парного обмена сообщениями	25
2.2. Проблемы использования блокирующих коммуникационных функций типа “точка-точка”	27
2.3. Примеры, трассировка выполнения параллельной программы.....	28
Задания	36
3. Передача одномерных массивов с помощью блокирующих функций “Send-Recv”	38
3.1. Примеры.....	38
Задания	42
4. Способ оценки эффективности распараллеливания алгоритмов	44
4.1. Оценка эффективности распараллеливания алгоритма суммирования ряда чисел	44
4.2. Оценка времени выполнения одной арифметической операции.....	45
4.3. Оценка времени передачи данных	47
Задания	48
5. Передача двумерных массивов с помощью блокирующих функций “Send-Recv”	50
5.1. Особенности использования динамических массивов	50
5.2. Примеры.....	51

Задания	54
6. Эффективное использование статусной информации при приеме сообщений	57
6.1. Прием сообщения от произвольного процесса-отравителя	57
6.2. Определение процесса-отправителя до приема сообщения	60
6.3. Определение количества элементов в приемном буфере до приема сообщения	62
Задания	64
7. Дополнительные коммуникации парного обмена	66
7.1. Неблокирующие коммуникационные операции	67
7.2. Одновременная передача данных	78
Задания	82
8. Коллективные операции: синхронизация процессов, широковещательная рассылка	83
8.1. Обзор коллективных операций	83
8.2. Принцип работы коллективных коммуникационных операций	84
8.3. Синхронизация процессов	86
8.4. Широковещательная рассылка	87
Задания	89
9. Коллективные операции: функции сбора блоков данных от всех процессов	91
9.1. Функции сбора одинаковых блоков данных от всех процессов	91
9.2. Функции сбора неодинаковых блоков данных от всех процессов	95
9.3. Примеры работы функций сборки блоков данных с двумерными массивами	98
Задания	101
10. Коллективные операции: функции распределения блоков данных по всем процессам	102
10.1. Функция распределения одинаковых блоков данных по всем процессам	102
10.2. Функция распределения неодинаковых блоков данных по всем процессам	104
10.3. Примеры работы распределения блоков данных с двумерными массивами	106
Задания	112
11. Коллективные вычислительные и совмещенные операции	114
11.1. Глобальные вычислительные операции	114
11.2. Совмещенные коллективные операции	121
Задания	124

12. Производные типы данных: простейшие конструкторы	126
12.1. Карты размещения данных и сценарий их использования	126
12.2. Конструктор описания карты непрерывного расположения блоков.....	128
12.3. Конструктор описания карты расположения одинаковых блоков с равными промежутками	130
Задания	137
13. Производные типы данных: работа с двумерными массивами.....	139
13.1. Конструктор описания карты расположения одинаковых блоков с байтовым смещением между началами блоков	139
13.2. Конструктор описания карты расположения различных блоков с разными промежутками.....	145
Задания	152
14. Производные типы данных: универсальный конструктор. Передача упакованных данных.....	155
14.1. Передача упакованных данных	155
14.2. Конструктор описания карты расположения различных блоков разных типов данных с разными промежутками	159
Задания	162
15. Группы и коммутаторы	164
15.1. Функции работы с группами процессов.....	164
15.2. Функции работы с коммутаторами	169
Задания	177
16. Логические топологии процессов	178
16.1. Декартова топология процессов.....	178
16.2. Примеры.....	182
Задания	188
Заключение.....	189
Литература	191
Приложение 1. Варианты контрольных работ.....	193
Вариант 1	193
Вариант 2	194
Вариант 3	195
Вариант 4	196
Вариант 5	197
Вариант 6	198
Вариант 7	199
Вариант 8	200
Вариант 9	201

Вариант 10	202
Приложение 2. Варианты индивидуальных заданий	203
Требования к выполнению индивидуальной работы.....	203
Вариант 1: Задача об “обедающих философах”	203
Вариант 2: Задача о восьми ферзях.....	205
Вариант 3: Задача о пяти ферзях	206
Вариант 4: Задача о коммивояжере.....	206
Вариант 5: Задача “Игра в жизнь”	206
Вариант 6: Сортировка последовательности чисел.....	208
Вариант 7: Задача о многопроцессорном расписании (задача о кучах)..	208
Вариант 8: Определение частоты события.....	208
Вариант 9: Размен денег	209
Вариант 10: Алгоритм Фокса перемножения матриц	209

Введение

Широкое применение информационных технологий во всех сферах деятельности человека требует от специалиста знаний, умений и навыков параллельного программирования для решения профессиональных задач, будь то создание информационных систем или проведение исследований с применением высокопроизводительных вычислительных систем с параллельной архитектурой.

Для работы с параллельными или распределенными базами данных требуются дополнения к существующим средствам работы с данными, обеспечивающих преимущество, в первую очередь, по времени в сравнении с обычной архитектурой. Для разработки корпоративных информационных систем также используются стандартные средства, дополненные библиотеками, поддерживающими параллельный режим работы системы. Примеров таких систем множество: банковские системы – одновременно совершается много транзакций, поступающих с терминалов операторов, банкоматов, пользователей Интернет-банков и т.д.; социальные сети – одновременно тысячи пользователей просматривают информацию, размещают свои фотографии, музыку, видео, работают с предоставляемыми сервисами. Без использования параллельно функционирующих серверов баз данных, web-серверов, серверов приложений одновременная работа многих пользователей таких информационных систем была бы невозможной. В декабре 2019 года Сбербанк создал самый мощный в России суперкомпьютер, который, как рассчитывает банк, ускорит разработку сервисов и процессов, основанных на искусственном интеллекте [14].

Другое направление использования вычислительных систем с параллельной архитектурой – проведение вычислений для решения сложных ресурсоемких задач. Примерами таких задач являются задачи прогнозирования погоды, изменения климата в долгосрочной перспективе, исследование новых материалов, создание новых лекарств и методов лечения, транспортные задачи, задачи геномики, астрономии, космонавтики, экологии, науки о мировом океане, синтез и распознавание речи, распознавание изображений, робототехника и искусственный интеллект и многое другое.

В ноябре 2018 года введен в эксплуатацию высокопроизводительный вычислительный кластер Росгидромета России с целью обеспечения вычислительных потребностей Росгидромета в вычислительных мощностях для реализации прогностических технологий современного уровня [5]. На момент ввода в эксплуатацию по производительности данный кластер занимал второе место в России.

Крупнейшие автоконцерны используют численное моделирование для проектирования новых автомобилей, а также для проведения краш-тестов на виртуальных стендах. Это значительно удешевляет процесс создания новинок автомобилестроения.

Суперкомпьютеры используются при прогнозировании чрезвычайных ситуаций. Например, цунами зарождается вследствие землетрясения, причем амплитуда порождаемой волны небольшая, скорость распространения, зависящая от амплитуды, также невелика. При приближении к береговой зоне длинная волна собирается в волновой сгусток, возрастает амплитуда и скорость, что может привести к катастрофическим последствиям. Факт землетрясения фиксируется мгновенно и есть некоторое время, за которое нужно определить будет ли порождаемая волна опасна или нет. В случае катастрофического прогноза необходимо принять меры по эвакуации населения из потенциально опасных береговых зон. Фактор времени в такой ситуации играет существенную роль и без суперкомпьютерных технологий требуемый прогноз с требуемой достоверностью получить невозможно.

Прогресс в суперкомпьютерных технологиях последних лет и быстрое вторжение этих технологий во все новые сферы человеческой деятельности говорит о том, что суперкомпьютерные технологии – это мощный инструмент, позволяющий сделать качественный рывок в познании многих актуальных задач современной науки, технологий и техники.

В математические модели исследований включают все большее количество факторов, что приближает их к реальным исследуемым объектам, но приводит к тому, что получить решение можно только с помощью численного эксперимента, который можно провести только на высокопроизводительной вычислительной системе с параллельной архитектурой. Поэтому изучение основных технологий параллельного программирования становится очень важным при подготовке специалистов, которым возможно придется заниматься разработкой программного обеспечения для многопроцессорных (многоядерных) вычислительных систем.

Важность подготовки кадров по указанному направлению подчеркивает и член-корр. РАН Вл. В. Воеводин “Через несколько лет отсутствие навыков работы с параллельными компьютерами будет равносильно компьютерной безграмотности” [7].

Самый мощный из кластеров, расположенных в российских вузах, – суперкомпьютер “Ломоносов-2” принадлежит Московскому государственному университету [18]. Сотрудники многих институтов РАН и его отделений, а также российских вузов имеют возможность проведения исследований, используя вычислительные мощности МГУ.

Кемеровский государственный университет наряду с 12-ю другими вузами России в 2009 году в рамках инновационной программы «Университетский кластер» [13] получил высокопроизводительный

вычислительный кластер. Программа «Университетский кластер» была инициирована компанией НР совместно с Учреждением Российской академии наук Институтом системного программирования (ИСП РАН) и Учреждением Российской академии наук Межведомственным суперкомпьютерным центром (МСЦ РАН), при участии национального оператора связи ЗАО «Синтерра» и при поддержке Министерства связи и массовых коммуникаций Российской Федерации и Агентства по делам молодежи Российской Федерации.

Основной целью программы «Университетский кластер» являлось повышение уровня использования параллельных и распределенных вычислений в образовательной и научно-исследовательской деятельности российских вузов. Это заложило основу для развития инновационных и научно-исследовательских проектов в вузах за счет активного использования технологий параллельных и распределенных вычислений и позволило повысить качество подготовки специалистов в области суперкомпьютерных технологий.

Для эффективного использования имеющихся вычислительных комплексов создан центр коллективного пользования «Высокопроизводительные параллельные вычисления» (ЦКП КемГУ по ВПВ) [22].

С 2009 года имеющийся высокопроизводительный вычислительный кластер (НР ВLc3000 производительностью 0.3 TFlops) дважды подвергался модернизации путем дооснащения современным коммуникационным оборудованием и мощными вычислительными узлами. В итоге к настоящему времени вычислительный комплекс обладает значительной мощностью (около 5 TFlops), однако для его эффективного использования требуется применение соответствующих технологий программирования.

В Кемеровском государственном университете издана серия учебных пособий «Основы высокопроизводительных вычислений».

Первый том данной серии – "Высокопроизводительные вычислительные системы" [1] посвящен рассмотрению вопросов, касающихся современных высокопроизводительных вычислительных систем (ВВС). Представлены архитектуры и классификация ВВС, принципы работы, современные направления развития, методы оценки производительности, общие принципы программирования, операционные системы и прикладное программное обеспечение ВВС. Приводится обзор и анализ самых мощных суперкомпьютеров мира.

Второй том – «Технологии параллельного программирования» [2] посвящен описанию наиболее распространенных технологий параллельного программирования. В данном учебном пособии представлены наиболее распространенные технологии параллельного программирования: OpenMP (Open Multi-Processing) – для систем с общей оперативной памятью, MPI (Message Passing Interface) – для систем с распределенной оперативной

памятью, UPC (Unified Parallel C) – для систем с общей и/или распределенной оперативной памятью.

Третий том – “Параллельные вычислительные алгоритмы” [3] посвящен вопросам проектирования, реализации и анализа параллельных алгоритмов обработки данных.

С 1993 года ведется рейтинг ТОП500 самых мощных компьютерных систем мира, который обновляется дважды в год и публикуется на сайте [15]. Анализ последних редакций списка ТОП500 представлен на сайте научно-исследовательского вычислительного центра (НИВЦ) МГУ [12]. В России по инициативе Межведомственного суперкомпьютерного центр РАН и НИВЦ МГУ с 2004 года ведется рейтинг ТОП50 [16]. Данные рейтинги позволяют проанализировать вычислительные системы, занимающие топовые позиции, и выявить основные тенденции развития суперкомпьютерных технологий.

В последних рейтингах все представленные системы кластерного типа, многие из систем дополнительно оснащаются специальными видеоускорителями, позволяющими увеличить вычислительную мощность за счет GPU-вычислений (вычислений на видеокартах), а также сопроцессорами, позволяющими значительно ускорить работу с вещественной арифметикой. Для организации связи между узлами кластерной системы используется специализированное коммуникационное оборудование, скорость передачи по которому сопоставима со скоростью передачи по шине данных.

Эффективность реализации параллельного алгоритма зависит от нескольких основных параметров:

- архитектуры многопроцессорного вычислительного комплекса, на котором будет выполняться параллельная программа;
- навыков выявления параллелизма в численных алгоритмах и выражения данного параллелизма в рамках той или иной технологии параллельного программирования;
- используемой технологии параллельного программирования, которая во многом зависит от архитектуры вычислительного ресурса.

Основной технологией параллельного программирования на системах кластерного типа является технология передачи сообщений (Message Passing Interface, MPI). Данная технология считается “ассемблером” параллельного программирования и традиционно вызывает трудности у студентов в ее освоении.

Данный практикум направлен на исправление сложившегося мнения о сложности использования MPI – на многочисленных, но простых, примерах подробно рассмотрены основные приемы параллельного программирования, достаточные для написания большинства высокоэффективных параллельных программ.

При написании практикума использовались материалы второго тома “Технологии параллельного программирования” [2] серии учебных пособий

“Основы высокопроизводительных вычислений” в части описания функций библиотеки Message Passing Interface (MPI), а также личный опыт автора ведения данной дисциплины на протяжении многих лет.

Объем практикума соответствует односеместровому курсу по дисциплинам “Технологии параллельного программирования”, “Технологии параллельных вычислений”, проводимых для обучающихся направлений 01.03.02 Прикладная математика и информатика, 02.03.02 Фундаментальная информатика и информационные технологии, 02.03.03 Математическое обеспечение и администрирование информационных систем.

Весь материал тематически разбит на 16 пунктов, в каждом из которых содержатся необходимые для выполнения заданий теоретические сведения (назначение и синтаксис функций MPI), примеры параллельных программ, демонстрирующие использование изученных функций, а также задания для закрепления материала, предлагаемые студентам для самостоятельного выполнения.

В первом приложении приведены варианты контрольных работ для проверки базовых знаний по использованию функций библиотеки MPI для передачи сообщений между процессами, а также простое упражнение на определение теоретической оценки ускорения алгоритма.

Во втором приложении студентам предлагаются индивидуальные задания, причем задания подобраны таким образом, чтобы большинство из них можно было решить методом полного перебора, который при последовательной реализации программы будет занимать много времени, а при параллельной реализации дает существенное (почти идеальное) ускорение.

В результате освоения материала практикума студенты будут:

знать: современное состояние и основные технологии и модели параллельного программирования;

уметь: проводить тестирование параллельных программ на вычислительных комплексах с параллельной архитектурой; определять степень параллелизма алгоритма, его ускорение и эффективность по сравнению с последовательным;

владеть: технологией создания параллельных аналогов программ для вычислительных систем с распределенной и общей оперативной памятью.

1. Знакомство с технологией Message Passing Interface: обрамляющие функции

Наиболее распространенной технологией параллельного программирования для вычислительных систем с распределенной оперативной памятью является MPI (Message Passing Interface – интерфейс передачи сообщений). Для создания MPI-программ используются стандартные языки программирования Си или FORTRAN с вызовом функций библиотеки MPI. Вместе с библиотекой MPI устанавливается загрузчик параллельного приложения, который порождает набор независимых процессов запускаемой MPI-программы. Процессы MPI-программы взаимодействуют друг с другом посредством вызова коммуникационных функций. Каждый процесс выполняется в своем собственном адресном пространстве, при одновременной работе нескольких процессов над одной общей вычислительной задачей требуется обмен сообщениями.

В стандарте MPI нет ограничений на то, как процессы будут запущены по узлам кластерной системы. Например, возможен запуск параллельного приложения несколькими процессами на обычной однопроцессорной или даже одноядерной системе.

1.1. Компиляция и запуск параллельных программ в UNIX-подобных операционных системах

Для обеспечения ведения дисциплины "Технологии параллельного программирования" в Кемеровском государственном университете создан учебный кластер-полигон, представляющий собой несколько двухядерных компьютеров, объединенных сетью (рис. 1).

На кластере установлена UNIX-подобная операционная система. Схема работы на учебном кластере следующая:

1. Пользователи подключаются к головному узлу кластера с помощью ssh-клиента (например, putty [26], рис. 2).
2. В поле "Host Name (or IP address)" необходимо ввести hpchead.kemsu.ru (доменное имя головного узла учебного кластера), выбрать порт соединения 22.
3. Выбрав категорию настроек "Translation", укажите кодировку UTF-8 для корректного отображения символов.
4. Для того, чтобы не указывать перечисленное каждый раз при подключении к кластеру, удобно сохранить настройки текущей сессии, указав ее имя в поле "Saved Sessions".

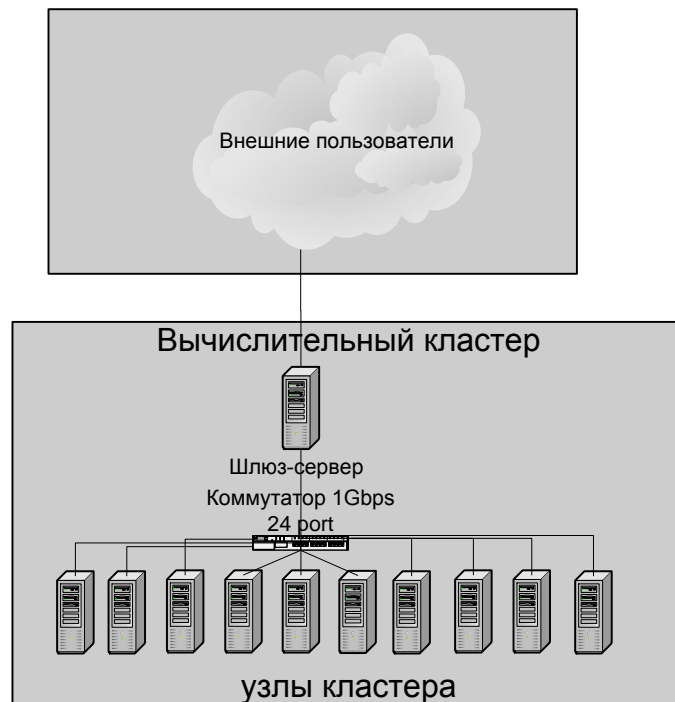


Рис. 1. Схема организации кластера

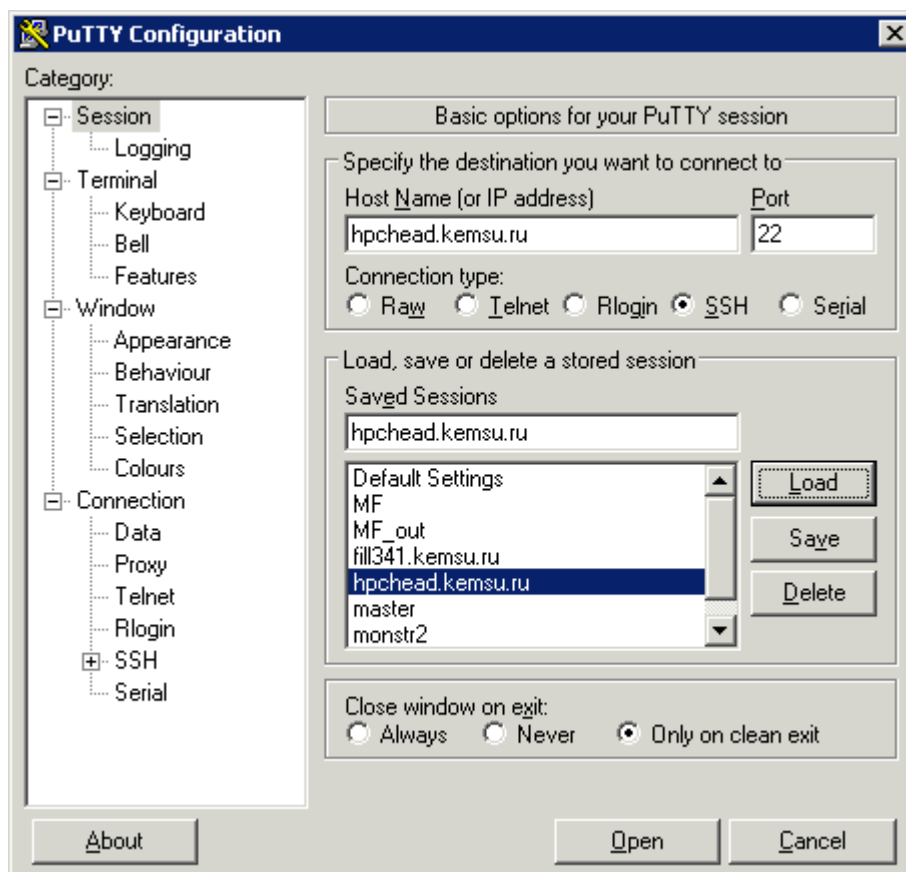


Рис. 2. Окно ssh-клиента putty

5. Далее, для соединения с хостом нужно нажать кнопку "Open", введя логин и пароль. На рис. 3 показан успешный вход в систему.

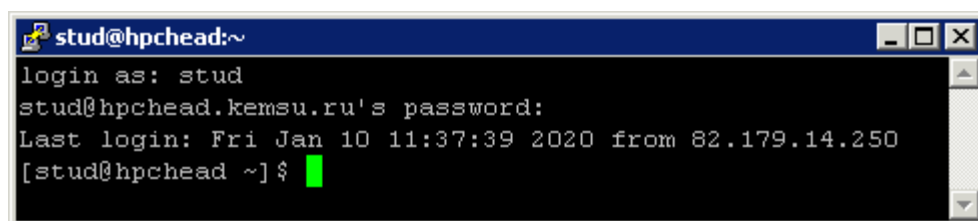


Рис. 3. Окно ssh-клиента putty после ввода логина и пароля

6. Большинство действий удобнее выполнять из файлового менеджера Midnight Commander, вызов которого осуществляется командой “mc”. Для выхода из Midnight Commander следует нажать клавишу “F10” (рис. 4).

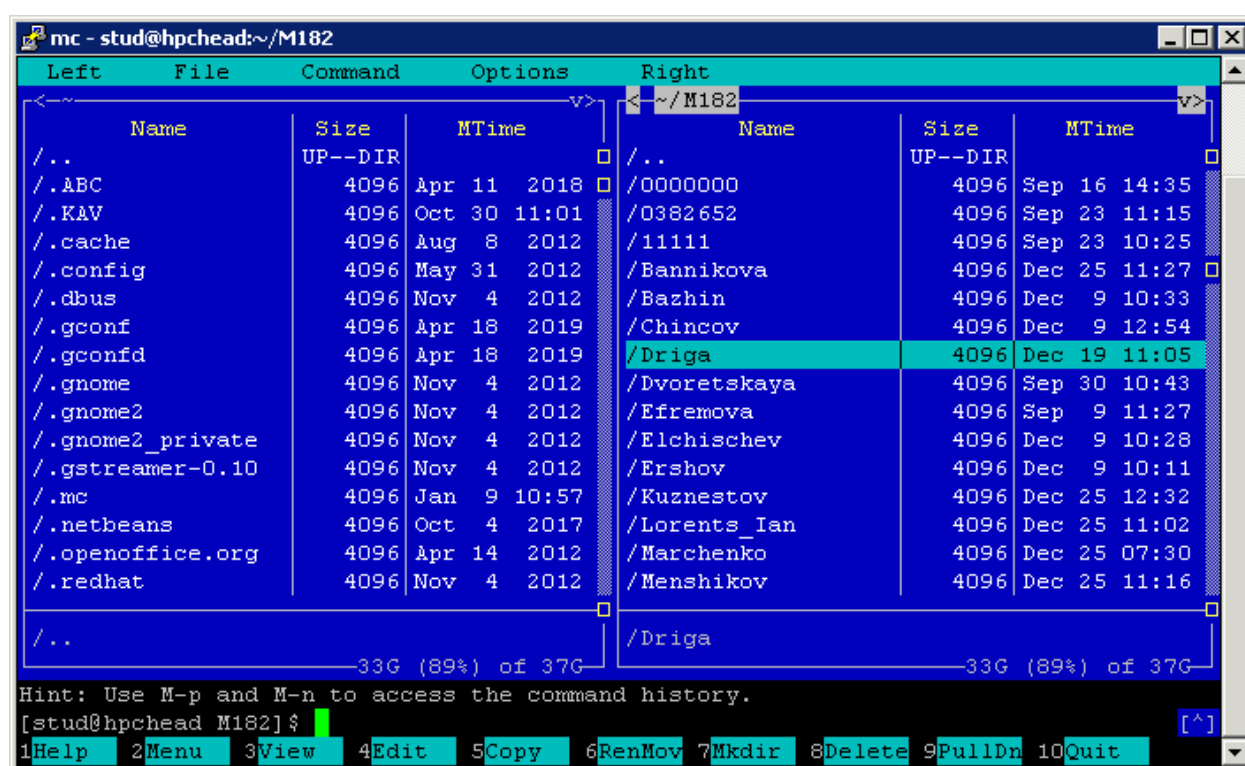


Рис. 4. Окно ssh-клиента putty после запуска Midnight Commander

7. Для создания директории следует пользоваться клавишей “F7” (подсказки о назначении функциональных клавиш расположены внизу экрана, рис. 4). Для создания нового файла нужно нажать сочетание клавиш “Shift”+”F4”. Для редактирования существующего файла – просто клавиша ”F4”.

8. Компиляция параллельных MPI-программ на языке C++ осуществляется при помощи команды “mpic++” (для “чистого” C - mpicc):

mpic++ first.c (будет откомпилирован файл first.c и создан исполняемый файл a.out).

mpic++ first.c -o program.out (будет откомпилирован файл first.c и создан исполняемый файл programm.out).

9. Запуск параллельной программы осуществляется с помощью загрузчика параллельного приложения (`mpiexec`, входит в состав библиотеки `mpich`, установленной на кластере).

`mpiexec -n 5 -f ./machines program.out`

`-n` – параметр для задания количества запускаемых копий исполняемого файла `program.out`.

`-f` – параметр для указания файла конфигурации кластера (`./machines`) распределения запускаемых копий на узлах кластера. На рис. 5 показано содержимое данного файла, соответствующее запуску первых двух процессов на узле с именем `hpchead`, следующих двух – на `hpcnode1`, следующих двух – на `hpcnode2`, затем следующие снова будут запущены на `hpchead` и т.д.

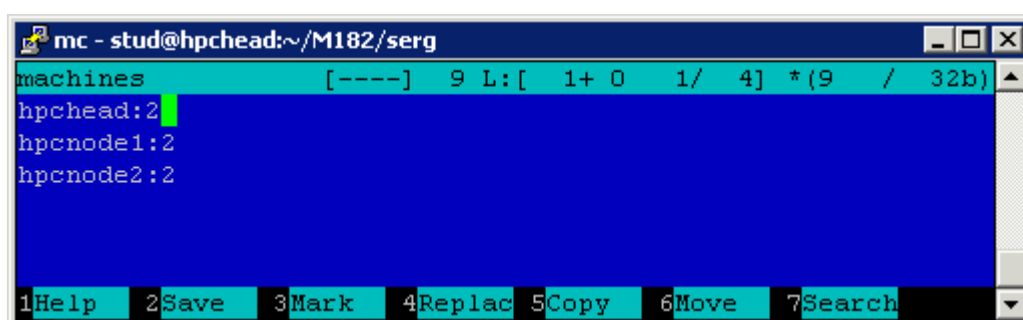


Рис. 5. Содержимое файла конфигурации кластера (`./machines`)

1.2. Установка и настройка библиотеки MPI в операционной системе Windows

Для выполнения домашних заданий можно дополнительно установить библиотеку `mpich` на своем персональном компьютере. При этом MPI-программы будут представлены множеством процессов, работающих на одном узле, передача данных между процессами будет выполняться посредством шины данных.

Разберем пример установки библиотеки `mpich` и ее использования совместно с CodeBlocks. Скачать библиотеку `mpich` можно с сайта разработчиков [23]. Следует обратить внимание на битность библиотеки – CodeBlocks содержит компилятор 32bit C++ с GNU-лицензией, соответственно и библиотеку `mpich` необходимо устанавливать 32bit.

Установщик библиотеки (`msi`-файл) нужно запустить из консоли с правами администратора – это необходимо по причине записи настроек в процессе установки в папки `windows`, защищенные от записи.

Во время установки будет запрошен пароль для запуска `mpi`-процессов (пароль может быть любым, не совпадающим с паролем Windows). Данный пароль (вернее – секретное слово, рис. 6) необходим для обеспечения

безопасности запуска процессов загрузчиком параллельного приложения с использованием протокола smpd.

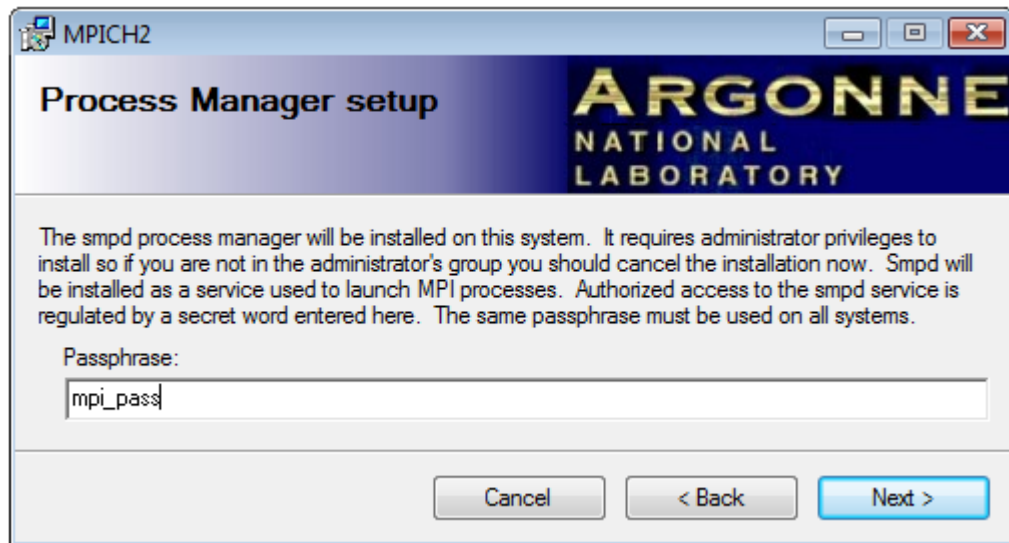


Рис. 6. Ввод пароля для запуска mpi-процессов

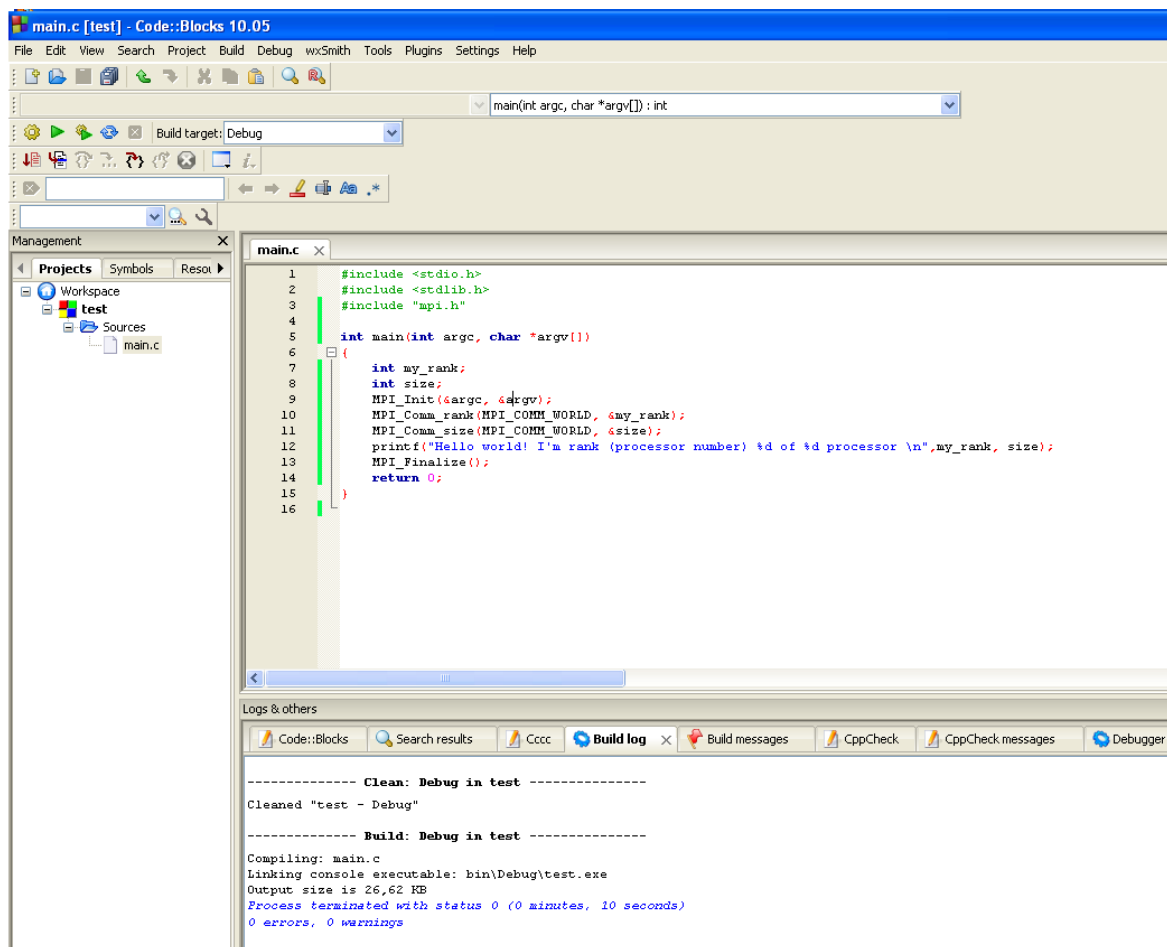


Рис. 7. Компиляция тестовой MPI-программы

Настройка CodeBlocks осуществляется по следующему алгоритму:

1. Запускаем CodeBlocks.
2. Выбираем меню "Settings", далее – "Compiler and debugger".
3. Выбираем вкладку "Linker settings" и добавляем библиотеку `mpi.lib`
Например:
`D:\Program Files\MPICH2\lib\mpi.lib`
4. Выбираем вкладку "Search Directories" и добавляем путь `D:\Program Files\MPICH2\include`.
5. Все – CodeBlocks настроен, можно проводить компиляцию программ! На рис. 7 приведен скрин удачной компиляции тестовой MPI-программы.
6. Для запуска MPI-программ в комплект MPICH2 входит программа с графическим интерфейсом `Wmpiexec`, которая представляет собой оболочку вокруг соответствующей утилиты командной строки `Mpiexec`. Окно программы `Wmpiexec` показано на рис. 8 (обратите внимание, что включён флажок «more options»).

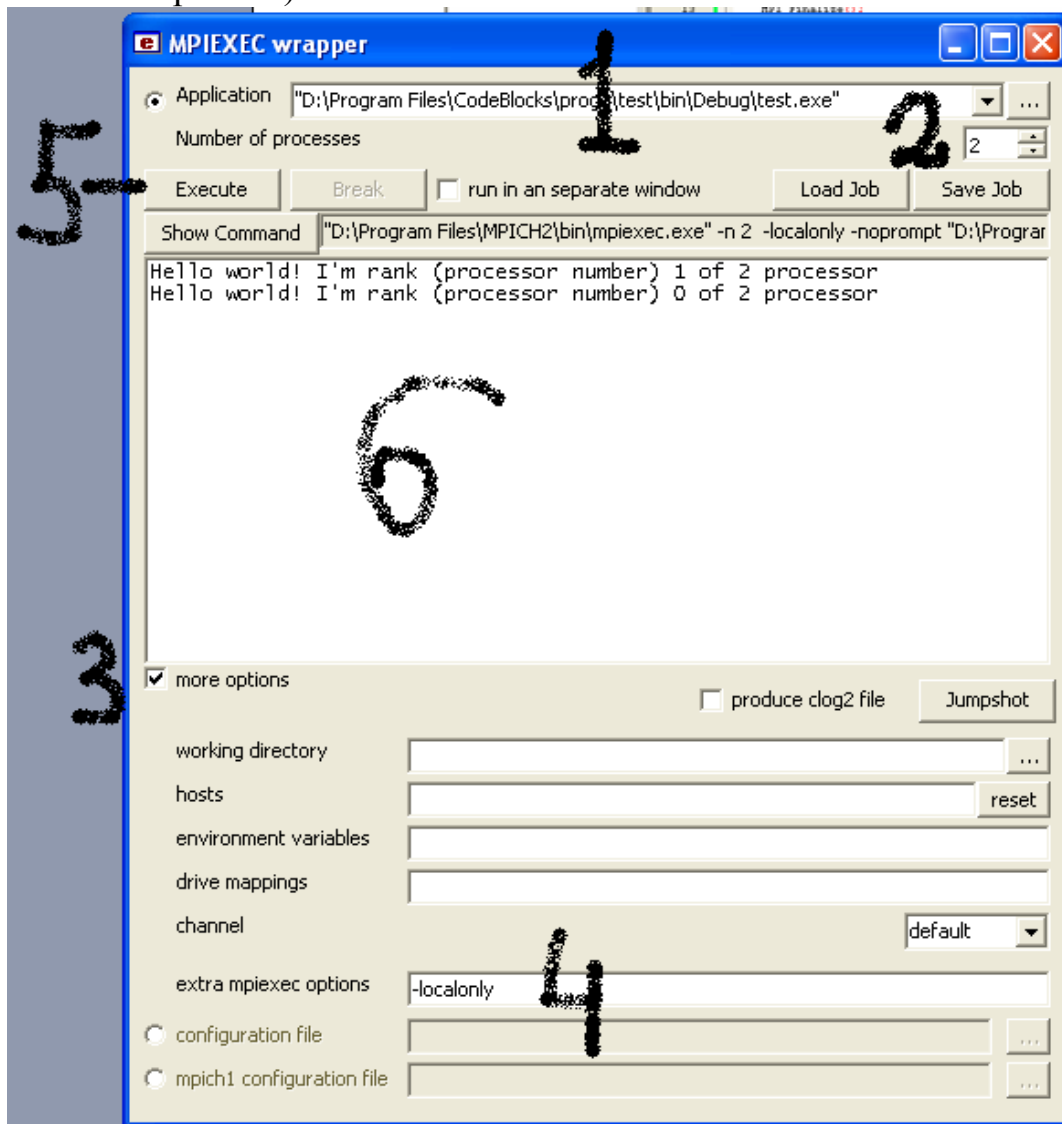


Рис. 8. Графическая оболочка загрузчика параллельных программ

Так как запуск всех процессов параллельного приложения будет осуществляться на локальной машине, то в поле “Extra mpiexec options” необходимо задать параметр “-localonly”.

Обозначения на рис. 8:

- 1 – адресная строка запускаемой программы;
- 2 – количество процессов параллельного приложения;
- 3 – опция для отображения дополнительных опций;
- 4 – опция запуска на локальном компьютере всех копий параллельного приложения;
- 5 – кнопка запуска параллельного приложения;
- 6 – окно вывода результатов выполнения параллельной программы.

Подробное руководство по установке и настройке библиотеки `mpich` можно посмотреть по следующей ссылке [21].

1.3. Обрамляющие функции MPI

Теперь, после рассмотрения компиляции и запуска параллельных приложений, начнем знакомство с технологией использования библиотеки передачи сообщений.

Все объекты библиотеки `mpi`, к которым можно отнести функции, константы, предопределенные типы данных и другие, имеют префикс **MPI_**. В связи с этим рекомендация не использовать имен с таким префиксом для избегания конфликтов именования с объектами MPI. Описание всех объектов библиотеки MPI собраны в файле **mpi.h**, поэтому в начале MPI-программы должна стоять директива:

#include <mpi.h>.

Каждая функция библиотеки MPI при своем вызове возвращает целочисленное значение (0 – удачное завершение функции, любая другая цифра – код ошибки). Также библиотека содержит функции обработки ошибок, позволяющие по коду ошибки получить ее словесное описание.

Загрузчиком параллельного приложения MPI-программа запускается несколькими копиями, представляющими множество параллельно работающих процессов. Во время работы параллельного приложения порождение дополнительных или уничтожение существующих процессов не допускается. Каждый процесс работает в своем адресном пространстве и для организации одновременной работы многих процессов над общей вычислительной задачей требуется явная пересылка сообщений между процессами.

Для пересылок между процессами используется канал связи – коммуникатор. При старте параллельной программы все порожденные процессы по умолчанию работают в рамках одного предопределенного коммуникатора, имеющего имя – **MPI_COMM_WORLD**.

Каждый процесс параллельного приложения имеет уникальный идентификатор – номер процесса. Если параллельное приложение образовано N процессами, то каждый процесс будет иметь уникальный номер от 0 до N-1. Данный номер процесса можно использовать в программе для деления вычислений между процессами. Например, 0-й процесс будет считать сумму четных элементов ряда, а 1-й процесс – сумму нечетных. Также номера процессов активно используются для пересылки данных – необходимо явно указывать адресатов и отправителей в коммуникационных функциях.

Подводя итог, MPI – это библиотека функций для коммуникаций между параллельными процессами с помощью механизма передачи сообщений. Данная библиотека содержит более 200 функций, которые можно классифицировать следующим образом:

- функции инициализации и закрытия MPI процессов;
- функции, реализующие коммуникационные операции типа точка-точка;
- функции, реализующие коллективные операции;
- функции для работы с группами процессов и коммуникаторами;
- функции для работы со структурами данных;
- функции формирования топологии процессов.

На первый взгляд, такое обилие функций заведомо усложняет процесс изучения библиотеки MPI и ее использования для создания параллельных приложений. Однако это не так. Любую параллельную программу можно написать с использованием всего 6 MPI функций, однако для получения большей эффективности распараллеливания требуется знание дополнительных функций, открывающих более полно потенциал библиотеки. Практикующие программисты ограничиваются довольно скромным набором функций – всего около 30, и этого вполне достаточно для распараллеливания множества задач.

Каждая MPI-программа должна начинаться с вызова функции инициализации MPI (функция `MPI_Init`). В результате выполнения этой функции создается группа процессов, в которую помещаются все процессы приложения, и создается область связи, описываемая предопределенным коммуникатором `MPI_COMM_WORLD`. Данный коммуникатор необходим для организации взаимодействия между процессами и входит в число обязательных аргументов практически каждой из MPI-функций.

Синтаксис функции инициализации параллельного приложения:

```
int MPI_Init(int *argc, char ***argv)
```

Функция `MPI_Init` содержит два аргумента, которые дублируют аргументы функции `main`. Это требуется для неявной передачи аргументов командной строки при инициализации параллельного приложения.

В конце параллельной программы должен быть вызов функции `MPI_Finalize`, которая уничтожает описатель канала связи (коммуникатор).

После данной функции вызов любой другой MPI-функции приведет к возникновению ошибки.

Синтаксис функции уничтожения коммуникатора MPI_Finalize:

int MPI_Finalize(void)

Следующая функция возвращает количество процессов в области связи коммуникатора.

int MPI_Comm_size(MPI_Comm comm, int *size)

Входные аргументы:

comm - коммуникатор.

Выходные аргументы:

size - число процессов в области связи коммуникатора comm.

Функция определения номера процесса MPI_Comm_rank имеет следующий синтаксис:

int MPI_Comm_rank(MPI_Comm comm, int *rank)

Входные аргументы:

comm - коммуникатор.

Выходные аргументы:

rank - номер процесса, вызвавшего функцию.

1.4. Примеры

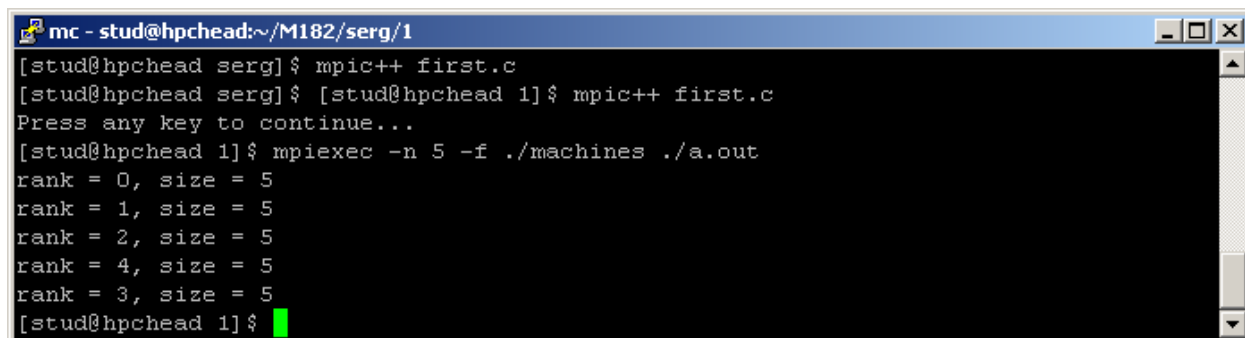
Пример 1.

На рис. 9 приведен пример программы, использующей обрамляющие функции MPI: строка 2 – подключение заголовочного файла библиотеки MPI, строка 7 – инициализация библиотеки MPI (получение канала связи – коммуникатора MPI_COMM_WORLD), строка 9 – получение общего числа процессоров параллельного приложения, 11 – получение идентификатора (номера) процесса, индивидуального для каждого процесса, 14 – завершение всех MPI-процессов (уничтожение коммуникатора).

На рис. 10 приведен скрин компиляции и запуска данной программы на 5-и процессах.

```
1.  #include <stdio.h>
2.  #include "mpi.h"
3.  int main(int argc, char* argv[])
4.  {
5.      int rank, size;
6.      // инициализация библиотеки MPI
7.      MPI_Init(&argc, &argv);
8.      // получение количества процессов в программе
9.      MPI_Comm_size(MPI_COMM_WORLD, &size);
10.     // получение ранга процесса
11.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12.     printf ("rank = %d, size = %d \n", rank, size);
13.     // завершение библиотеки MPI
14.     MPI_Finalize();
15.     return 0;
16. }
```

Рис. 9. Пример первой MPI-программы использования обрамляющих функций MPI



```
mc - stud@hpchead:~/M182/serg/1
[stud@hpchead serg]$ mpic++ first.c
[stud@hpchead serg]$ [stud@hpchead 1]$ mpic++ first.c
Press any key to continue...
[stud@hpchead 1]$ mpiexec -n 5 -f ./machines ./a.out
rank = 0, size = 5
rank = 1, size = 5
rank = 2, size = 5
rank = 4, size = 5
rank = 3, size = 5
[stud@hpchead 1]$
```

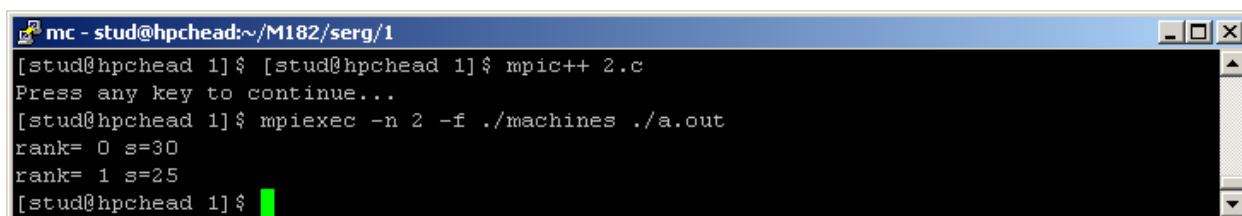
Рис. 10. Скрин компиляции и запуска программы на 5 процессах

Пример 2.

При создании параллельных программ требуется разделение вычислений между процессами, для этого в программе активно используется номер процесса (ранг): в условии, в цикле и т. д. Рассмотрим пример программы (рис. 11), вычисляющей сумму четных элементов ряда на 0-м процессе, а сумму нечетных – на 1-м. На рис. 12 приведен скрин запуска программы на 2-х процессах при пределе ряда равным 10. Строка 10 задает выполнение последующего цикла на 0-м процессе, а строка 13 – на 1-м.

```
1.  #include <stdio.h>
2.  #include "mpi.h"
3.  int main(int argc, char *argv[])
4.  {
5.    int rank, size;
6.    int n=10, s=0, i=0;
7.    MPI_Init(&argc, &argv);
8.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.    MPI_Comm_size(MPI_COMM_WORLD, &size);
10.   if(rank==0)
11.       for(i=0; i<=n; i+=2)
12.           s+=i;
13.   if(rank==1)
14.       for(i=1; i<=n; i+=2)
15.           s+=i;
16.   printf("rank= %d s=%d \n", rank, s);
17.   MPI_Finalize();
18.   return 0;
19. }
```

Рис. 11. Листинг программы для вычисления частичных сумм ряда



```
mc - stud@hpchead:~/M182/serg/1
[stud@hpchead 1]$ [stud@hpchead 1]$ mpic++ 2.c
Press any key to continue...
[stud@hpchead 1]$ mpiexec -n 2 -f ./machines ./a.out
rank= 0 s=30
rank= 1 s=25
[stud@hpchead 1]$
```

Рис. 12. Скрин компиляции и запуска программы на 2 процессах

Пример 3.

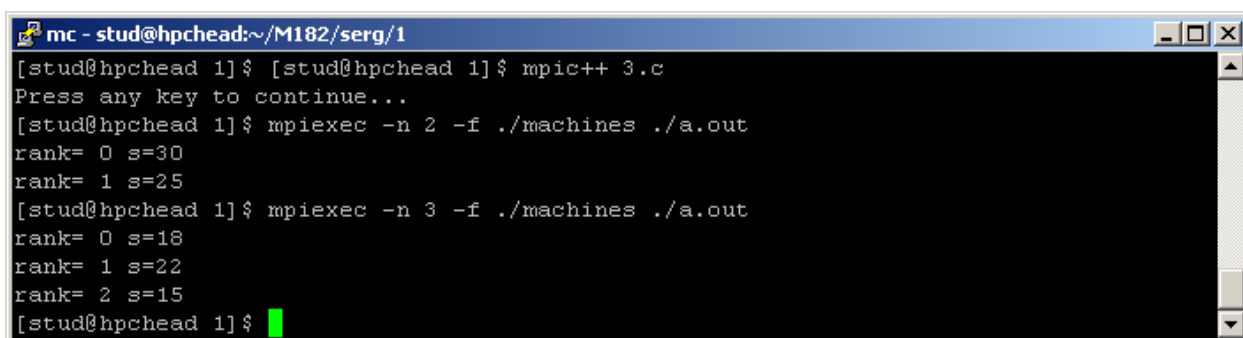
Следует отметить несовершенство программы из 2-го примера. Например, если потребуется посчитать частичные суммы на 3-х процессах или большем количестве, то потребуется 10-12 строки с небольшими изменениями продублировать для каждого процесса. Изменения коснутся начала и шага цикла для каждого процесса: 0-й процесс начнет цикл с 0, 1-й процесс – с 1, 2-й процесс – с 3 и т.д., а шаг цикла у всех будет одинаков и равен количеству процессов – size.

На рис. 13 приведен листинг программы, лишенный недостатков 2-го примера.

На рис. 14 приведен скрин компиляции и запуска параллельной программы на 2-х процессах – результат аналогичен предыдущему примеру, а также запуск на 3-х процессах – общая сумма осталась прежней.

```
20. #include <stdio.h>
21. #include "mpi.h"
22. int main(int argc, char *argv[])
23. {
24.     int rank, size;
25.     int n=10, s=0, i=0;
26.     MPI_Init(&argc, &argv);
27.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
28.     MPI_Comm_size(MPI_COMM_WORLD, &size);
29.     for(i=rank; i<=n; i+=size)
30.         s+=i;
31.     printf("rank= %d s=%d \n", rank, s);
32.     MPI_Finalize();
33.     return 0;
34. }
```

Рис. 13. Листинг программы для вычисления частичных сумм ряда



```
mc - stud@hpchead:~/M182/serg/1
[stud@hpchead 1]$ [stud@hpchead 1]$ mpic++ 3.c
Press any key to continue...
[stud@hpchead 1]$ mpirun -n 2 -f ./machines ./a.out
rank= 0 s=30
rank= 1 s=25
[stud@hpchead 1]$ mpirun -n 3 -f ./machines ./a.out
rank= 0 s=18
rank= 1 s=22
rank= 2 s=15
[stud@hpchead 1]$
```

Рис. 14. Скрин компиляции и запуска программы на 2-х и 3-х процессах

Задания

1. Скачайте, установите и настройте библиотеку `mpich` на домашнем компьютере. В файле с описанием выполнения задания приведите скрины, демонстрирующие шаги подключения библиотеки, компиляции и запуска

- тестовой MPI-программы. Укажите характеристики компьютера, операционную систему, используемое программное обеспечение.
2. Дайте описание функции `MPI_Get_processor_name` и приведите пример ее использования.
 3. Дайте описание и продемонстрируйте выполнение функции `MPI_Wtick()`.
 4. Напишите последовательную программу суммирования ряда чисел $(1/(1+i))$. Проведите исследование времени выполнения программы в зависимости от предела ряда. Предел ряда взять $10^6, 10^7 \dots 10^9$. Для определения времени выполнения программы используйте функцию `MPI_Wtime()`, описание которой также приведите в отчете. Полученные данные приведите в табличном виде (предел ряда, результат суммирования, время выполнения).
 5. Напишите программу, которая выводит на каждом процессе номер процесса (`rank`) и переменную `n`, которая вычисляется следующим образом: на четных – $n=2*\text{rank}$, на нечетных – $n=3*\text{rank}$. Запуск параллельной программы осуществите на различном количестве процессов (например, `size=10,15,20`).
 6. Напишите программу, которая определяет среднее арифметическое 100 чисел, заданных случайным образом от 0 до 100 на каждом процессе. Вывод должен содержать номер процесса и вычисленное среднее значение случайных чисел для данного процесса.
 7. Напишите программу, в которой на каждом процессе создается одномерный массив `a[n]`, инициализация элементов которого происходит на каждом процессоре одинаково $a[i]=i$. Затем, каждый процесс производит вычисление сумм первых `k` элементов данного массива, где `k` – номер процесса, увеличенный на 1. Сделайте контрольный вывод результатов: номер процессора, результат вычислений.

2. Передача данных с помощью блокирующих коммуникационных функций типа "Точка-Точка"

Для создания простейших параллельных программ, помимо изученных в предыдущем пункте обрамляющих функций, следует использовать парные функции приема-передачи сообщений. Данный набор (обрамляющие функции плюс функции передачи данных) из 6-и функций позволяет создавать простейшие параллельные программы.

2.1. Функции парного обмена сообщениями

Синтаксис функции отправки сообщения:

```
int MPI_Send(void* buf, int count, MPI_Datatype  
datatype, int dest, int tag, MPI_Comm comm)
```

Входные аргументы:

buf - адрес начала расположения пересылаемых данных;
count - число пересылаемых элементов;
datatype - тип посылаемых элементов;
dest - номер процесса-получателя в группе, связанной с коммуникатором comm;
tag - идентификатор сообщения;
comm - коммуникатор области связи.

Функция выполняет посылку count элементов типа datatype сообщения с идентификатором tag процессу с номером dest в области связи коммуникатора comm. Аргумент buf – это то, что требуется отправлять: массив, структура или скалярная переменная. В случае отправки переменной значение count = 1. Следует отметить, что первый аргумент должен быть адресом отправляемых данных. Например, требуется отправить значение переменной a: первым аргументом требуется указать адрес переменной - &a.

Типы данных определены в заголовочном файле библиотеки MPI – mpi.h. На самом деле, это не в полном смысле типы данных, а описатели типов данных. Введение собственных типов в MPI было продиктовано тем обстоятельством, что стандартные типы языков на разных платформах имеют различное представление. MPI допускает возможность запуска процессов параллельной программы на компьютерах различных платформ, обеспечивая при этом автоматическое преобразование данных при пересылках. В таблице 1 приведено соответствие предопределенных в MPI типов стандартным типам языка C.

В таблице 1 перечислен обязательный минимум поддерживаемых стандартных типов, однако, если в базовой системе представлены и другие типы, то их поддержку будет осуществлять и MPI. Типы MPI_BYTE и MPI_PACKED используются для передачи двоичной информации без какого-либо преобразования. Кроме того, программисту предоставляются средства создания собственных типов на базе стандартных.

Таблица 1: Соответствие между MPI-типами и типами языка C

Тип MPI	Тип языка C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

Блокировка выполнения программы функцией MPI_Send означает, что после возврата из функции можно использовать любые присутствующие в аргументах переменные без опасения испортить передаваемое сообщение. При этом возврат из данной функции не означает ни того, что сообщение покинуло процесс-отправитель, ни того, что сообщение принято процессом-получателем. Предоставляется только гарантия безопасного изменения переменных, использованных в вызове функции.

Синтаксис функции приема сообщения:

```
int MPI_Recv(void* buf, int count, MPI_Datatype  
datatype, int source, int tag, MPI_Comm comm,  
MPI_Status *status)
```

Входные аргументы:

count - максимальное число принимаемых элементов;
datatype - тип элементов принимаемого сообщения;
source - номер процесса-отправителя;
tag - идентификатор сообщения;
comm - коммуникатор области связи.

Выходные аргументы:

buf - адрес начала расположения принимаемого сообщения;
status - атрибуты принятого сообщения.

Функция выполняет прием count элементов типа datatype сообщения с идентификатором tag от процесса source в области связи коммуникатора comm.

Идентификатор сообщения – это целочисленное значение, одинаковое для пары MPI_Send и MPI_Recv.

Атрибуты полученного сообщения доступны в переменных status. Переменные status должны быть явно объявлены в MPI программе. В языке C status – это структура типа MPI_Status с тремя полями: MPI_SOURCE, MPI_TAG, MPI_ERROR.

Функция приема сообщения блокирует выполнение программы, пока все элементы сообщения не будут размещены в приемном буфере buf.

2.2. Проблемы использования блокирующих коммуникационных функций типа “точка-точка”

Проблема взаимных блокировок

Например, 0-й процессор организует с помощью блокирующей функции MPI_Send посылку переменной 1-му процессору, в тоже время 1-й процессор организует пересылку переменной 0-му процессору, затем каждый из них запускает MPI_Recv для приема ожидаемой пересылки. В некоторых реализациях MPI данная конструкция может привести к тупиковой ситуации, когда оба процесса ждут друг друга.

Следствие: нельзя организовать передачу данных с помощью MPI_Send и MPI_Recv отдельно взятого процесса самому себе.

Если же при взаимной пересылке выставить на обоих процессах функцию MPI_Recv прежде MPI_Send, то тупиковая ситуация возникнет в любой реализации MPI.

Единственно корректным способом организовать такой обмен информацией является следующий: на одном из процессов вызвать функции передачи и приема сообщений в последовательности MPI_Send, MPI_Recv, а на другом в обратной последовательности – MPI_Recv, MPI_Send (последовательная передача данных: сначала будет осуществлена передача данных от одного процесса другому, а затем – в обратную сторону). Другим, более эффективным способом решения является использование специальной функции одновременной передачи данных – MPI_Sendrecv (ее изучение будет представлено далее), которая не приводит к блокировке процессов.

Неэффективность применения для многоточечной рассылки данных

В случае передачи одних и тех же данных от одного процессора всем остальным в группе использование функций MPI_Send и MPI_Recv малоэффективно, наиболее привлекательным является применение коллективных операций, например, широковещательная рассылка данных – MPI_Bcast, использование которых приводит к уменьшению программного кода и увеличению эффективности распараллеливания за счет более быстрого обмена данными.

2.3. Примеры, трассировка выполнения параллельной программы

Пример 1.

На рис. 15 приведен листинг программы, реализующей следующий алгоритм:

0-й процесс	1-й процесс
Считывание значения переменной n	
Отправка переменной n 1-му процессу	Прием переменной n от 0-го процесса
Вывод на экран номера процесса и значения переменной n	

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    if(rank==0)
13.    {
14.        scanf("%d",&n);
15.        MPI_Send(&n,1,MPI_INT,1,777,MPI_COMM_WORLD);
16.    }
17.    if(rank==1)
18.    {
19.        MPI_Recv(&n,1,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
20.    }
21.    printf("processor number %d  n=%d \n",rank, n);
22.    MPI_Finalize();
23.    return 0;
24. }
```

Рис. 15. Листинг программы передачи данных

Аргументы MPI_Send:

&n – адрес расположения данных для отправки.

1 – один элемент, т.к. отправляется переменная.

MPI_INT – тип пересылаемых данных.

1 – номер процесса-получателя.

777 – идентификатор сообщения (одинаков для функции отправки и приема сообщения).

MPI_COMM_WORLD – коммуникатор (канал связи).

Аргументы MPI_Recv:

&n – адрес расположения данных для приема.

1 – один элемент, т.к. принимается переменная.

MPI_INT – тип получаемых данных.

0 – номер процесса-отправителя.

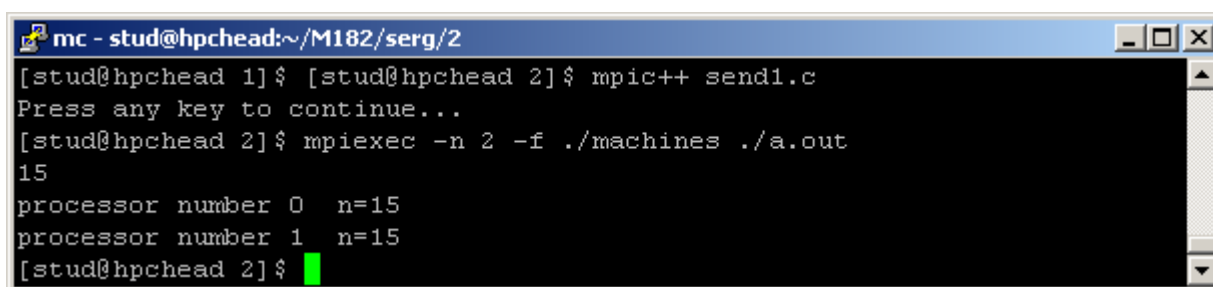
777 – идентификатор сообщения (одинаков для функции отправки и приема сообщения).

MPI_COMM_WORLD – коммуникатор (канал связи).

&stat – структура с типом данных MPI_Status, содержащая служебную информацию о полученном сообщении.

На рис. 16 приведен скрин компиляции и запуска данной программы на 2-х процессах. Изначально значение переменной n доступно после ввода 0-му процессу, после передачи данных оба процесса выводят на экран одинаковое значение.

Особое указание: устройство ввода-вывода доступно только на 0-й консоли, в связи с этим ввод значения переменной n обязательно надо выполнить только на 0-м процессе, в противном случае ненулевые процессы будут заблокированы ожиданием ввода, которого никогда не произойдет.



```
mc - stud@hpchead:~/M182/serg/2
[stud@hpchead 1]$ [stud@hpchead 2]$ mpic++ send1.c
Press any key to continue...
[stud@hpchead 2]$ mpiexec -n 2 -f ./machines ./a.out
15
processor number 0 n=15
processor number 1 n=15
[stud@hpchead 2]$
```

Рис. 16. Скрин компиляции и запуска программы на 2 процессах

Пример 2. Трассировка выполнения параллельной программы

На рис. 17 приведен листинг программы, реализующей пересылку переменной от 0-го процесса всем остальным по следующему алгоритму:

0-й процесс	ненулевые процессы
Считывание значения переменной n	
Отправка переменной n i-му процессу в цикле по i от 1 до size-1.	Прием переменной n от 0-го процесса
Вывод на экран номера процесса и значения переменной n	

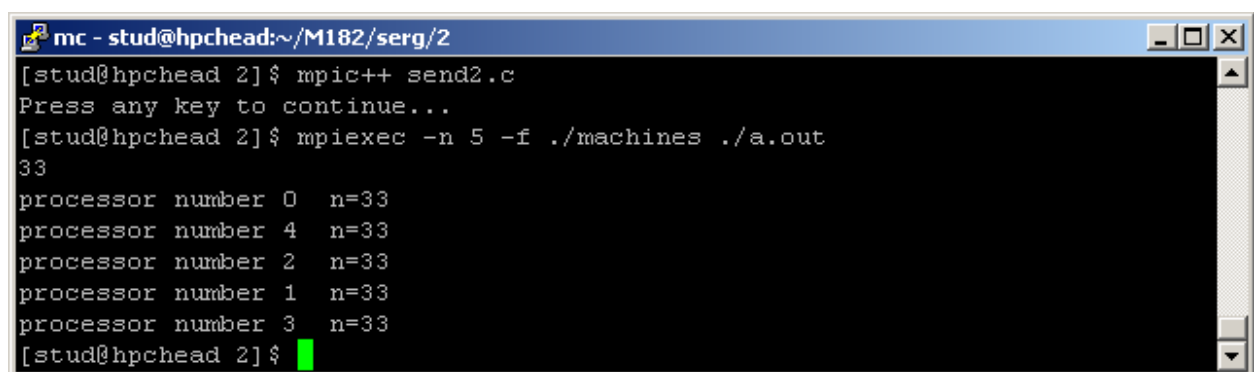
На рис. 18 приведен скрин компиляции и запуска данной программы на 5-х процессах при вводе числа 33. Изначально значение переменной n доступно после ввода 0-му процессу, после передачи данных все процессы выводят на экран одинаковое значение.

```

1.  #include <stdio.h>
2.  #include "mpi.h"
3.
4.  int main(int argc, char *argv[])
5.  {
6.      int rank;
7.      int size;
8.      int n=0,i;
9.      MPI_Status stat;
10.     MPI_Init(&argc, &argv);
11.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12.     MPI_Comm_size(MPI_COMM_WORLD, &size);
13.     if(rank==0)
14.     {
15.         scanf("%d",&n);
16.         for(i=1;i<size;i++)
17.             MPI_Send(&n,1,MPI_INT,i,777,MPI_COMM_WORLD);
18.     }
19.     else
20.     {
21.         MPI_Recv(&n,1,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
22.     }
23.     printf("processor number %d  n=%d \n",rank, n);
24.     MPI_Finalize();
25.     return 0;
26. }

```

Рис. 17. Листинг программы передачи данных



```

mc - stud@hpchead:~/M182/serg/2
[stud@hpchead 2]$ mpic++ send2.c
Press any key to continue...
[stud@hpchead 2]$ mpiexec -n 5 -f ./machines ./a.out
33
processor number 0  n=33
processor number 4  n=33
processor number 2  n=33
processor number 1  n=33
processor number 3  n=33
[stud@hpchead 2]$

```

Рис. 18. Скрин компиляции и запуска программы на 5 процессах

Визуализация трассы выполнения параллельной программы состоит из двух этапов – сборка лог-файла при выполнении параллельной программы и дальнейшая ее визуализация при помощи утилиты Jumpshot (требуется установка **jawa**).

Разберем пошагово процесс сборки лога и визуализации трассы.

1) перед выполнением программы необходимо указать флаг “produce clog2 file” (рис. 19).

2) Запускаем программу – на рис. 20 видно, что лог-файл записан в папку расположения параллельного приложения.

Конец строки для ввода данных в загрузчике параллельного приложения можно ввести при нажатии комбинации клавиш: <Ctrl>+<Enter>.

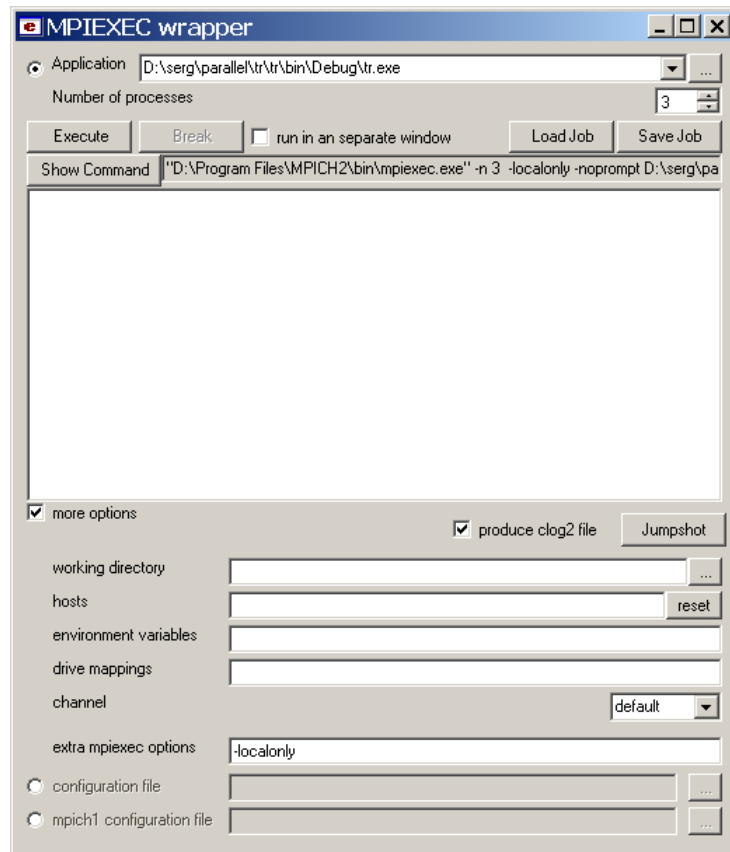


Рис. 19. Скрин загрузчика параллельного приложения

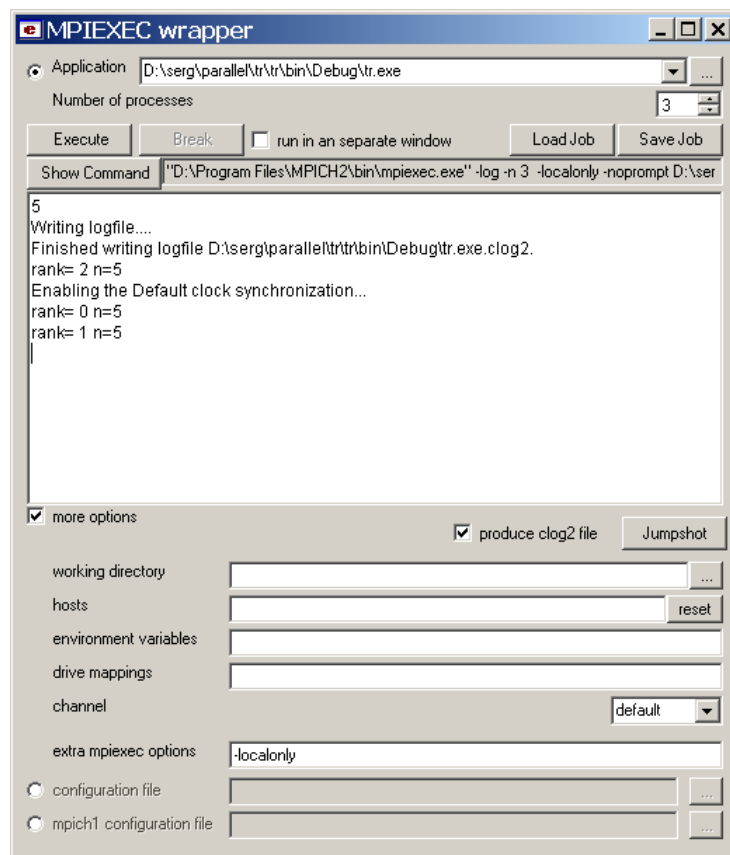


Рис. 20. Демонстрация работы программы на 3-х процессах и вводе числа 5

3) Далее запускаем утилиту визуализации лога кнопкой "Jumpshot".

Если java установлена позже MPICH, то может и не запуститься ничего. В этом случае запускайте Jumpshot из семейства приложений MPICH (start->программы->MPICH-> Jumpshot)

В приведенном примере при нажатии на кнопку "Jumpshot" вывелась ошибка, что java не найдена. Пришлось запускать Jumpshot отдельно (рис. 21).

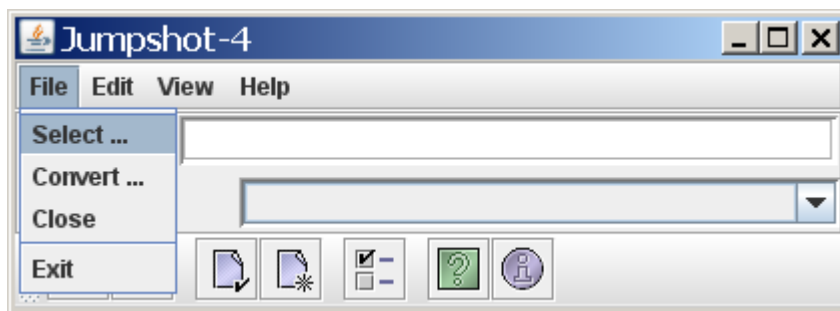


Рис. 21. Демонстрация работы "Jumpshot"

4) Выбираем файл с трассой (рис. 22)

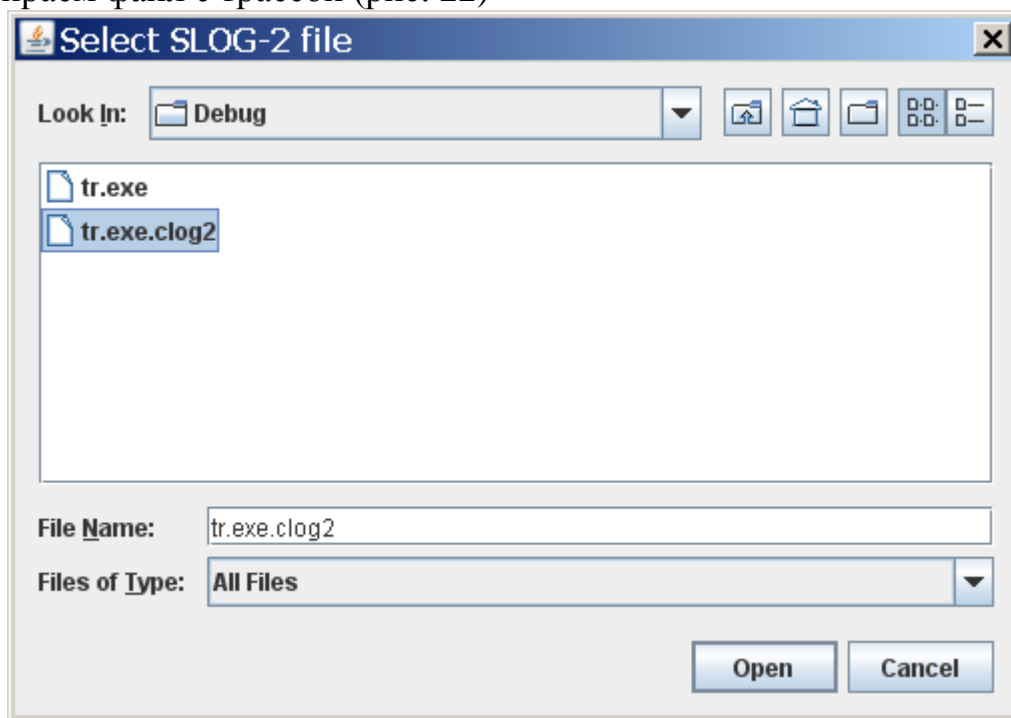


Рис. 22. Демонстрация работы "Jumpshot"

5) Осуществляем конвертирование лог-файла (рис. 23-24).

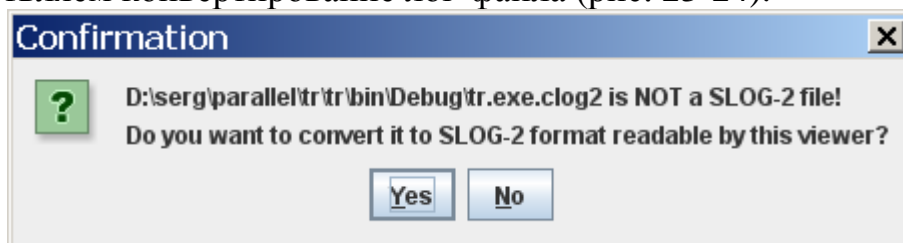


Рис. 23. Демонстрация работы "Jumpshot"

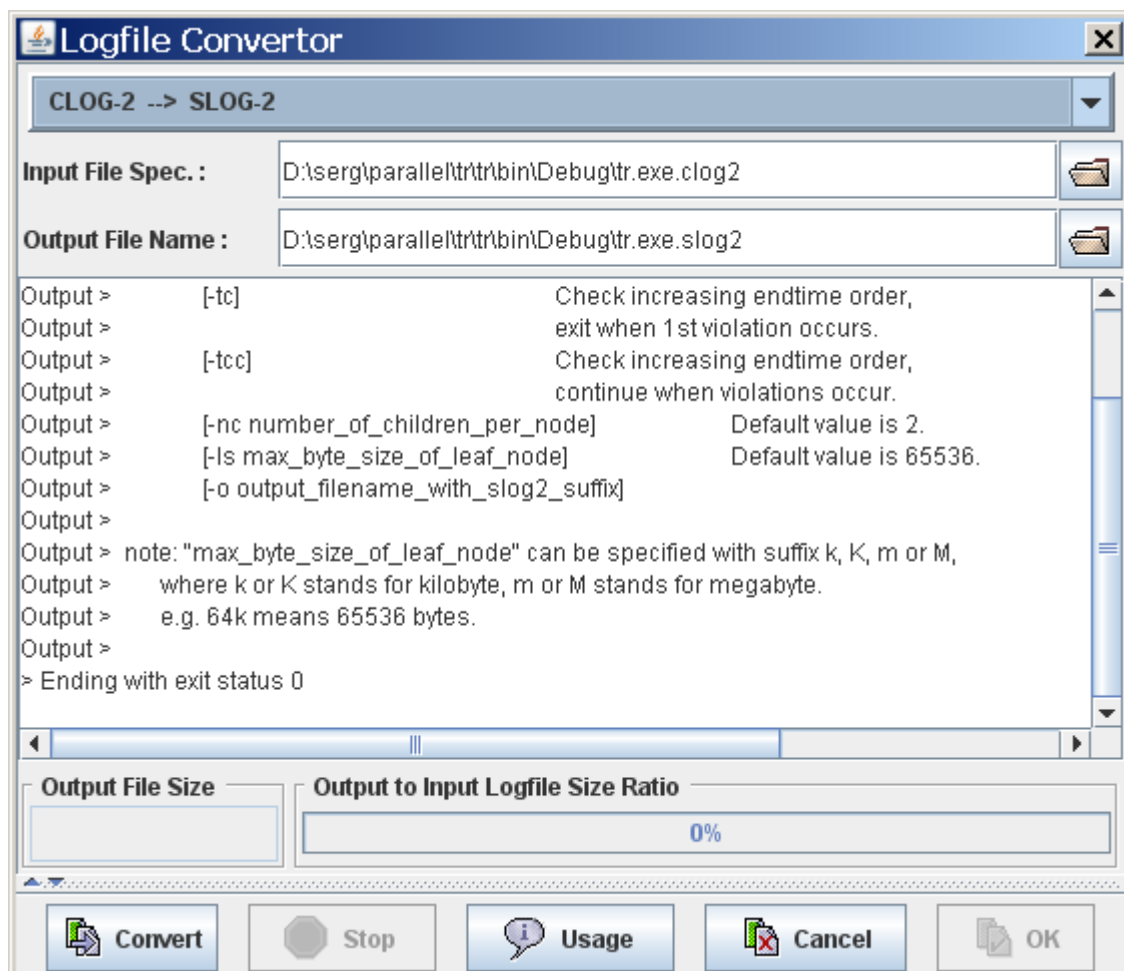


Рис. 24. Демонстрация работы "Jumpshot"

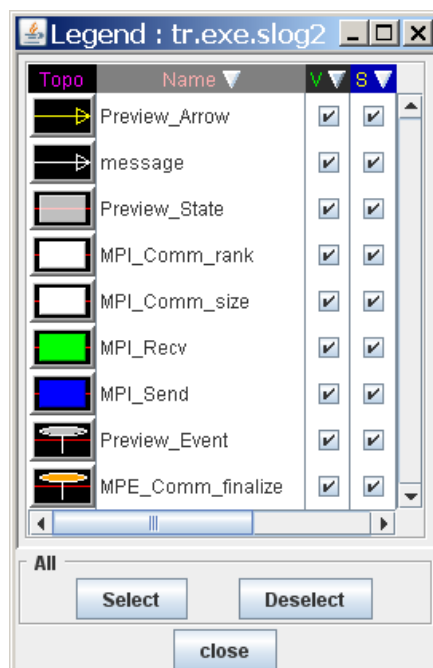


Рис. 25. Демонстрация работы "Jumpshot"

6) Нажимаем кнопку "ok" и получаем два окна – легенда (рис. 25, нас интересуют две функции Send-Recv) и саму трассу параллельного приложения (рис. 26). На трассе видно, что программа запущена на 3-х процессах, 0-й процесс (синим цветом помечено время работы функции отправки сообщений) отправляет данные на 1-й и 2-й процессы, время приема на которых помечено зеленым цветом.

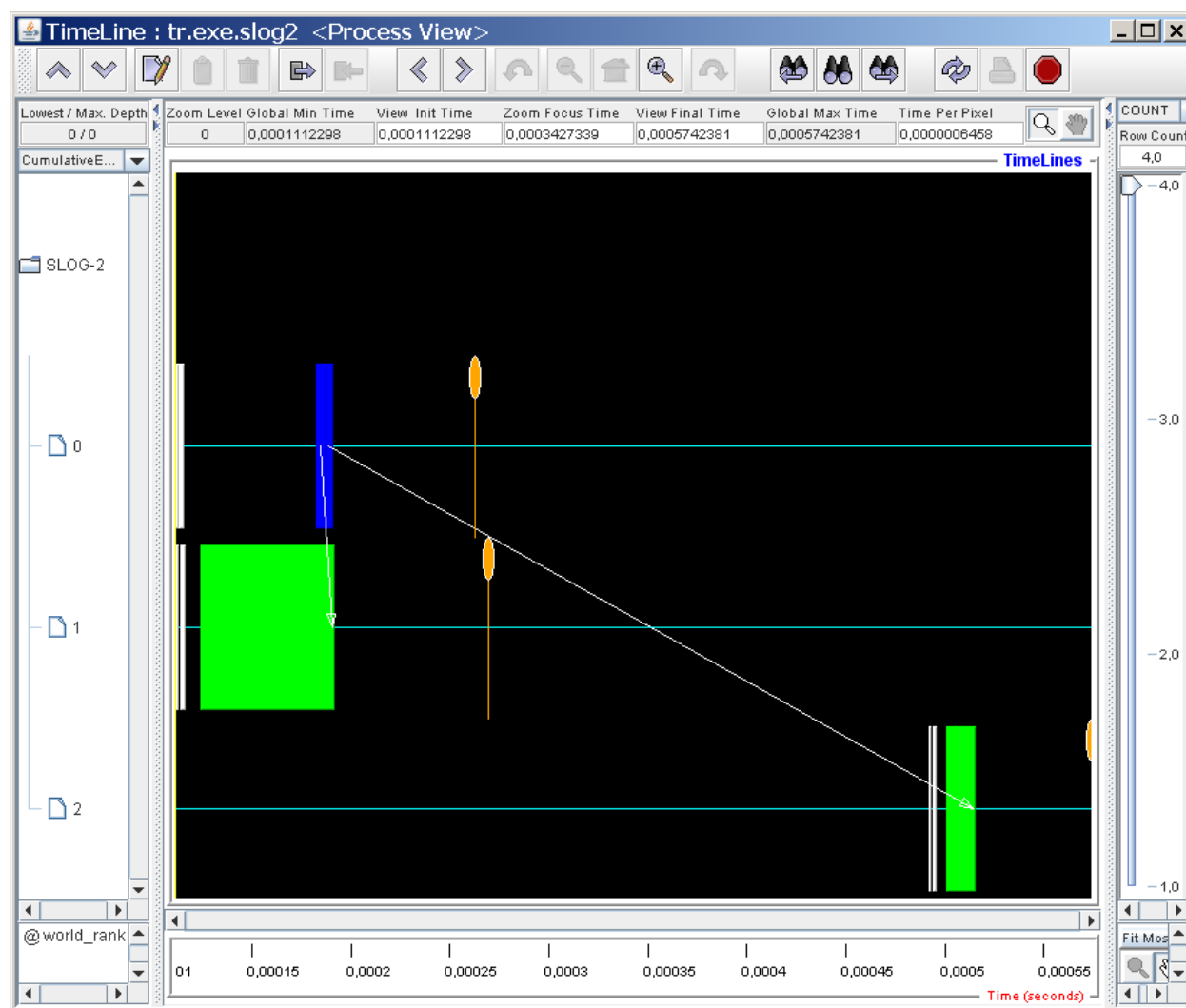


Рис. 26. Демонстрация работы "Jumpshot"

Пример 3.

На рис. 27 приведен листинг программы, реализующей алгоритм суммирования ряда чисел по следующему алгоритму:

0-й процесс	ненулевые процессы
Считывание значения переменной n	
Отправка переменной n i-му процессу в цикле по i от 1 до size-1.	Прием переменной n от 0-го процесса
Вычисление частичной суммы	
Прием в цикле от i-го процесса	Отправка частичной суммы на 0-й

присланного сообщения и процесс суммирование в глобальную сумму.	
Вывод на экран итоговой суммы	

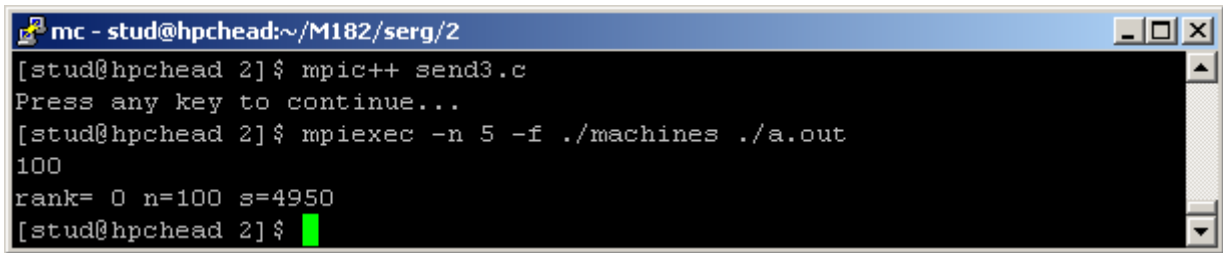
На рис. 28 приведен скрин компиляции и запуска данной программы на 5-х процессах и предела ряда равного 100.

```

1.  #include <stdio.h>
2.  #include "mpi.h"
3.
4.  int main(int argc, char *argv[])
5.  {
6.      int rank;
7.      int size;
8.      int n, s=0, s1;
9.      MPI_Status stat;
10. MPI_Init(&argc, &argv);
11. MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12. MPI_Comm_size(MPI_COMM_WORLD, &size);
13.
14. if(rank==0)
15. {
16.     scanf("%d",&n);
17.     for(int i=1;i<size;i++)
18.         MPI_Send(&n,1,MPI_INT,i,777,MPI_COMM_WORLD);
19. }
20. else
21. {
22.     MPI_Recv(&n,1,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
23. }
24.
25. for(int i=rank;i<n;i+=size)
26.     s+=i;
27.
28. if(rank==0)
29. {
30.     for (int i=1;i<size;i++)
31.     {
32.         MPI_Recv(&s1,1,MPI_INT,i,777,MPI_COMM_WORLD,&stat);
33.         s+=s1;
34.     }
35.     printf("rank= %d n=%d s=%d \n",rank, n, s);
36. }
37. else
38. {
39.     MPI_Send(&s,1,MPI_INT,0,777,MPI_COMM_WORLD);
40. }
41.
42. MPI_Finalize();
43. return 0;
44. }

```

Рис. 27. Листинг программы суммирования ряда чисел



```
mc - stud@hpchead:~/M182/serg/2
[stud@hpchead 2]$ mpicc send3.c
Press any key to continue...
[stud@hpchead 2]$ mpiexec -n 5 -f ./machines ./a.out
100
rank= 0 n=100 s=4950
[stud@hpchead 2]$
```

Рис. 28. Скрин компиляции и запуска программы на 5 процессах при n=100

Задания

Указание: для всех заданий во время выполнения программы соберите трассу параллельного приложения и в отчете приведите скрин визуализации трассы параллельного приложения.

1. Напишите программу, в которой все ненулевые процессы генерируют случайное число и отправляют его на 0-й процесс. 0-й процесс в цикле принимает сообщения и выводит на экран номер процесса-отправителя и присланное им число.
2. Напишите программу, в которой все ненулевые процессы отправляют свой идентификатор на 0-й процесс. 0-й процесс в цикле принимает сообщения и производит следующие вычисления: при приеме четных элементов их просто суммирует в некоторую переменную S1, а при приеме нечетных умножает на 2 перед суммированием в переменную S2. После приема выводит на экран обе суммы.
3. Напишите программу, используя блокирующие коммуникационные функции (MPI_Send, MPI_Recv), реализующую следующий алгоритм:
 - на 0 процессе инициализируется переменная (double a);
 - 0 процесс пересылает переменную a первому процессу;
 - Первый, получив значение переменной a, добавляет к нему единицу и отправляет на нулевой процесс.

Вывод результатов должен быть оформлен следующим образом: номер процесса, посылаемое значение, полученное значение.

4. Напишите программу, используя блокирующие коммуникационные функции (MPI_Send, MPI_Recv), реализующую следующий алгоритм:
 - на 0 процессе инициализируется переменная (int a);
 - 0 процессор рассылает переменную a всем процессам;
 - после получения переменной a, все процессы умножают a на свой идентификатор и передают на 0 процесс;
 - 0 процесс получает от всех процессов данные и выводит на экран номер процесса-отправителя и полученные от него данные.
5. Напишите программу, используя блокирующие коммуникационные функции (MPI_Send, MPI_Recv), реализующую следующий алгоритм:

- на 0 процессе инициализируется переменная (int a);
 - 0 процессор рассылает переменную a всем процессам;
 - после получения переменной a, все процессы прибавляют к ней свой индивидуальный номер и передают на 0 процесс;
 - 0 процесс получает от всех процессов данные и суммирует полученные данные.
6. Напишите программу, используя блокирующие коммуникационные функции (MPI_Send, MPI_Recv), реализующую алгоритм передачи данных по кольцу. Алгоритм программы можно представить следующим образом:
- на 0 процессе инициализируется переменная (int a);
 - 0 процесс пересылает переменную a первому процессу;
 - первый, получив значение переменной a, добавляет к нему единицу и отправляет на 2-й процесс.
 - ...
 - (size-1) процесс, получив значение переменной a от (size-2) процесса, прибавляет к значению a единицу и отправляет на 0-й процесс.
7. Напишите программу, используя блокирующие коммуникационные функции (MPI_Send, MPI_Recv), реализующую алгоритм передачи данных по двум кольцам: нечетные процессы образуют 1 кольцо, четные – второе.
8. Создайте последовательную программу (она уже должна быть выполнена в 1-й работе), реализующую алгоритм суммирования ряда чисел для предела суммы 10^6 , $10^7 \dots 10^9$. Член ряда равен $(1/(1+i))$. Создайте параллельную программу, реализующую данный алгоритм. Определите время выполнения последовательной и параллельной программ в зависимости от предела суммы (запустите несколько раз и возьмите наилучшие результаты). Параллельную программу запустите на 2-х и 4-х процессах. Определите ускорение параллельной программы по сравнению с последовательной. Требуется представить полученные значения в табличном виде, привести скрин компиляции и запуска параллельной программы. Сделайте выводы о том, при каком пределе ряда уже наблюдается ускорение параллельной программы по сравнению с последовательной. Укажите параметры домашнего компьютера (сколько ядер, поддерживается ли многопоточность?).

3. Передача одномерных массивов с помощью блокирующих функций "Send-Recv"

В рамках данного пункта рассматривается передача одномерных массивов с использованием уже изученных коммуникационных функций парного обмена сообщениями.

3.1. Примеры

Пример 1.

На рис. 29 приведен листинг программы, реализующей следующий алгоритм:

0-й процесс	Ненулевые процессы
Считывание значения переменной n	
Отправка переменной n всем остальным процессам	Прием переменной n от 0-го процесса
Выделение памяти под массив a[n]	
Присваивание a[i]=i	
Отправка массива всем остальным процессам	Прием массива a с 0-го процесса
Вывод на экран номера процесса и массива a	

С 15 по 22 строки – рассылка переменной n, уже рассмотренная ранее.

23 строка – выделение памяти под массив a.

26-27 – присваивание значений элементам массива a на 0-м процессе.

28-29 – отправка в цикле массива a каждому ненулевому процессу.

33 строка – прием массива на ненулевых процессах.

35-36 строки – "пустой" цикл с увеличивающимся пределом для каждого последующего процесса. Этот цикл необходим для того, чтобы разнести по времени последующий вывод массива всеми процессами (37-40 строки).

0-й адрес массива (первый аргумент в функциях MPI_Send и MPI_Recv) можно было указать иначе - &a[0], просто используемая в примере запись короче. Для пересылки нескольких элементов одной посылкой достаточно указать вторым параметром их количество.

На рис. 30 приведен скрин компиляции и запуска данной программы на 5-и процессах.

```

1.  #include <stdio.h>
2.  #include "mpi.h"
3.  #include "iostream"
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int rank;
8.      int size;
9.      int n=0,i,s=0;
10.     MPI_Status stat;
11.     MPI_Init(&argc, &argv);
12.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13.     MPI_Comm_size(MPI_COMM_WORLD, &size);
14.
15.     if(rank==0)
16.     {
17.         scanf("%d",&n);
18.         for(i=1; i<size; i++)
19.             MPI_Send(&n,1,MPI_INT,i,777,MPI_COMM_WORLD);
20.     }
21.     else
22.         MPI_Recv(&n,1,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
23.     int *a=new int[n];
24.     if(rank==0)
25.     {
26.         for(i=0; i<n; i++)
27.             a[i]=i;
28.         for(i=1; i<size; i++)
29.             MPI_Send(a,n,MPI_INT,i,777,MPI_COMM_WORLD);
30.     }
31.     else
32.     {
33.         MPI_Recv(a,n,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
34.     }
35.     for(i=0; i<1000000*rank; i++)
36.         s+=1;
37.     printf("rank= %d  a: ",rank);
38.     for(i=0; i<n; i++)
39.         printf(" %d ",a[i]);
40.     printf("\n ");
41.     MPI_Finalize();
42.     return 0;
43. }

```

Рис. 29. Листинг программы передачи данных

```

mc - stud@hpchead:~/M182/serg/3
[stud@hpchead 3]$ mpic++ send_mas1.c
Press any key to continue...
[stud@hpchead 3]$ mpiexec -n 5 -f ./machines ./a.out
10
rank= 0  a:  0  1  2  3  4  5  6  7  8  9
rank= 1  a:  0  1  2  3  4  5  6  7  8  9
rank= 4  a:  0  1  2  3  4  5  6  7  8  9
rank= 2  a:  0  1  2  3  4  5  6  7  8  9
rank= 3  a:  0  1  2  3  4  5  6  7  8  9
[stud@hpchead 3]$

```

Рис. 30. Скрин компиляции и запуска программы на 5 процессах с вводом n=10

Пример 2.

На рис. 31 приведен листинг программы, реализующей следующий алгоритм:

0-й процесс	1-й процесс
Выделение памяти под массив a[10]	
Присваивание a[i]=i	
Отправка 5 элементов массива a, начиная со 2-го элемента, на 1-й процесс	Прием элементов массива a с 0-го процесса с размещением элементов в начале массива a
Вывод на экран номера процесса и массива a	

Для отправки с заданного адреса массива нужного количества элементов нужно указать 1-м аргументом требуемый адрес (&a[2], строка 20) и вторым аргументом – требуемое количество элементов.

На рис. 32 приведен скрин компиляции и запуска данной программы на 2-х процессах. Можно увидеть, что элементы массива переданы правильно с корректным размещением на заданных местах.

```
1.  #include <stdio.h>
2.  #include "mpi.h"
3.  #include "iostream"
4.  int main(int argc, char *argv[])
5.  {
6.      int rank;
7.      int size;
8.      int n=10,i,s=0;
9.      MPI_Status stat;
10.     MPI_Init(&argc, &argv);
11.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12.     MPI_Comm_size(MPI_COMM_WORLD, &size);
13.     int *a=new int[n];
14.     for(i=0; i<n; i++)
15.         a[i]=-1;
16.     if(rank==0)
17.     {
18.         for(i=0; i<n; i++)
19.             a[i]=i;
20.         MPI_Send(&a[2],5,MPI_INT,1,777,MPI_COMM_WORLD);
21.     }
22.     else
23.     {
24.         MPI_Recv(&a[0],5,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
25.     }
26.     for(i=0; i<1000000*rank; i++)
27.         s+=1;
28.     printf("rank= %d a: ",rank);
29.     for(i=0; i<n; i++)
30.         printf(" %d ",a[i]);
31.     printf("\n ");
32.     MPI_Finalize();
33.     return 0;
34. }
```

Рис. 31. Листинг программы передачи данных


```

mc - stud@hpchead:~/M182/serg/3
[stud@hpchead 3]$ mpic++ send2.c
Press any key to continue...
[stud@hpchead 3]$ mpiexec -n 2 -f ./machines ./a.out
rank= 0  a:  0  1  2  3  4  5  6  7  8  9
rank= 1  a:  2  3  4  5  6 -1 -1 -1 -1 -1
[stud@hpchead 3]$

```

Рис. 32. Скрин компиляции и запуска программы на 2 процессах

```

1.  #include <stdio.h>
2.  #include "mpi.h"
3.  #include "iostream"
4.  int main(int argc, char *argv[])
5.  {
6.      int rank;
7.      int size;
8.      int n,i,s=0;
9.      MPI_Status stat;
10.     MPI_Init(&argc, &argv);
11.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12.     MPI_Comm_size(MPI_COMM_WORLD, &size);
13.     if(rank==0)
14.     {
15.         n=5*size;
16.         int *a=new int[n];
17.         for(i=0; i<n; i++)
18.             a[i]=i;
19.         for(i=1; i<size; i++)
20.             MPI_Send(&a[(i-1)*5],5,MPI_INT,i,777,MPI_COMM_WORLD);
21.     }
22.     else
23.     {
24.         int *b=new int[5];
25.         MPI_Recv(&b[0],5,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
26.         for(i=0; i<1000000*rank; i++)
27.             s+=1;
28.         printf("\trank= %d b: ",rank);
29.         for(i=0; i<5; i++)
30.             printf(" %d ",b[i]);
31.         printf("\n ");
32.     }
33.     MPI_Finalize();
34.     return 0;
35. }

```

Рис. 33. Листинг программы передачи данных

Пример 3.

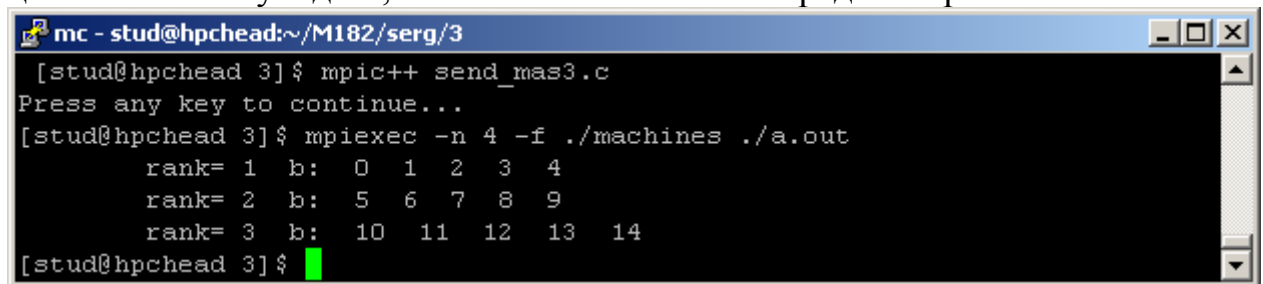
На рис. 33 приведен листинг программы, реализующей рассылку массива с 0-го процесса по 5 элементов остальным процессам по следующему алгоритму:

0-й процесс	Ненулевые процессы
Выделение памяти под массив a[5*size]	

Присваивание $a[i]=i$	Выделение памяти под массив $b[5]$
Отправка в цикле по 5 элементов массива a каждому ненулевому процессу	Прием присланных элементов массива с 0-го процесса с размещением элементов в массив b
	Вывод на экран номера процесса и массива b

Для отправки нужных элементов с заданного адреса массива необходимо указать 1-м аргументом требуемый адрес с привязкой к итератору цикла ($\&a[(i-1)*5]$, строка 20).

На рис. 34 приведен скрин компиляции и запуска данной программы на 4-х процессах. Можно увидеть, что элементы массива переданы правильно.



```

mc - stud@hpchead:~/M182/serg/3
[stud@hpchead 3]$ mpic++ send_mas3.c
Press any key to continue...
[stud@hpchead 3]$ mpiexec -n 4 -f ./machines ./a.out
rank= 1 b: 0 1 2 3 4
rank= 2 b: 5 6 7 8 9
rank= 3 b: 10 11 12 13 14
[stud@hpchead 3]$

```

Рис. 34. Скрин компиляции и запуска программы на 4 процессах

Задания

Указание: для всех заданий во время выполнения программы соберите трассу параллельного приложения и в отчете приведите скрин визуализации трассы параллельного приложения.

1. Напишите программу, реализующую рассылку с 0-го процесса по 3 элемента каждому ненулевому (см. пример 3).
2. Используя блокирующие коммуникационные функции типа (Точка-Точка), создайте следующую параллельную программу: на всех процессах задается одномерный массив ($a[i]=rank$, $i=0...2$), который отправляется на 0-й процесс; 0-й процесс объявляет необходимый по размеру одномерный массив и принимает от всех остальных пересылаемые данные. Например, Вы запускаете программу на 3-х процессах, 1-й процесс отправляет 0-му следующий массив (1,1,1), 2-й процесс отправляет 0-му - (2,2,2), 0-й процесс получает данные и, сохраняя в одномерный массив, выводит на экран следующее: 1,1,1,2,2,2.
3. Напишите программу, которая реализует следующий алгоритм: на 0-м процессе задается одномерный массив ($a[i]=rank$, $i=0...3*size$), который по одному элементу рассылается всем ненулевым процессам по кругу. Например, Вы запускаете программу на 3-х процессах, 1-й процесс в итоге должен получить 1, 4, 7, 2-й процесс – 2, 5, 8. Не забудьте сделать контрольный вывод пересланных данных.

4. Создайте следующую параллельную программу: на всех процессах задается одномерный массив ($a[i]=rank$, $i=0...rank+1$), который отправляется на 0-й процесс; 0-й процесс объявляет необходимый по размеру одномерный массив и принимает от всех остальных пересылаемые данные. Например, Вы запускаете программу на 4-х процессах, 1-й процесс отправляет 0-му следующий массив (1,1), 2-й процесс отправляет 0-му - (2,2,2), 3-й процесс отправляет 0-му (3,3,3,3), 0-й процесс получает данные и, сохраняя в одномерный массив, выводит на экран следующее: 1,1,2,2,2,3,3,3,3.

4. Способ оценки эффективности распараллеливания алгоритмов

4.1. Оценка эффективности распараллеливания алгоритма суммирования ряда чисел

Рассмотрим идею оценивания эффективности распараллеливания алгоритмов на примере распараллеливания алгоритма суммирования ряда чисел.

Для подсчета теоретического ускорения будем использовать формулу:

$$S_p = \frac{T_1}{T_p} \quad (\text{отношение времени выполнения на одном процессе к времени}$$

выполнения на системе из p процессов). Для определения ускорения нужно оценить время выполнения последовательной программы и параллельной.

Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Дополнительно упростим задачу, предположив, что вычислительная система будет состоять из 2-х узлов. Такое предположение вполне оправдано: если ускорения не наблюдается на 2-х процессах, то добавление дополнительных процессов для реализации параллельной программы только еще больше усугубит ситуацию. Доля вычислений на каждый процесс будет уменьшаться, а объем передаваемых данных – увеличиваться, что негативно скажется на эффективности параллельной реализации алгоритма.

Время выполнения последовательной программы составит:

$T_1 = (n-1)t$ (количество операций, затрачиваемых на суммирование n элементов ряда чисел).

Время выполнения параллельной программы оценим, исходя из следующего алгоритма: пусть оба процесса изначально имеют необходимые элементы в памяти для суммирования, оба процесса одновременно выполняют суммирование половины ряда чисел (например, 0-й процесс – суммирует четные, 1-й процесс – нечетные), затем 1-й процесс отправляет результат суммирования на 0-й процесс, который после получения прибавляет его к своей сумме для получения конечного результата. В итоге время реализации параллельной программы составит:

$$T_2 = \left(\frac{n}{2} - 1 \right) t + \alpha t + t = \frac{n}{2} t + \alpha t.$$

Ускорение выполнения параллельного алгоритма на системе из двух процессов в сравнении со временем выполнения алгоритма на одном процессе получилось следующее:

$$S_2 = \frac{T_1}{T_2} = \frac{(n-1)t}{\frac{n}{2}t + \alpha t} = \frac{2(n-1)}{n+2\alpha}.$$

Ускорение будет наблюдаться, если S_2 будет находиться в промежутке от 1 до 2. Если $S_2 < 1$, то реализация параллельной программы на двух процессорах займет больше времени, чем на одном. В приведенной формуле ускорения $S_2 > 1$ при $n > 2(\alpha + 1)$.

Так как ускорение и эффективность параллельного алгоритма зависят от параллельной вычислительной системы, на которой он будет реализован, то необходимо получить оценку параметра α . Для этого нужно определить время t для выполнения одной арифметической операции, а также время αt для пересылки значения переменной от одного узла другому. Параметр α задает соотношение времени выполнения одной арифметической операции ко времени выполнения пересылки одной переменной длиной 8 байт (удвоенной точности).

Приведем несколько вспомогательных программ, позволяющих получить оценку параметра α .

4.2. Оценка времени выполнения одной арифметической операции

Для получения оценки производительности отдельного узла кластерной системы можно воспользоваться тестом LINPACK [11].

LINPACK – это пакет фортран-программ для решения систем линейных алгебраических уравнений. Целью создания LINPACK отнюдь не было измерение производительности. Алгоритмы линейной алгебры весьма широко используются в самых разных задачах, и поэтому измерение производительности на LINPACK представляет интерес для многих пользователей.

В основе алгоритмов действующего варианта LINPACK лежит метод декомпозиции. Исходная матрица сначала представляется в виде произведения двух матриц стандартной структуры, над которыми затем выполняется собственно алгоритм нахождения решения. Подпрограммы, входящие в LINPACK, структурированы. В стандартном варианте LINPACK выделен внутренний уровень базовых подпрограмм, каждая из которых выполняет элементарную операцию над векторами. Набор базовых подпрограмм называется BLAS (Basic Linear Algebra Subprograms). Например, в BLAS входят две простые подпрограммы SAXPY (умножение вектора на скаляр и сложение векторов) и SDOT (скалярное произведение

векторов). Все операции выполняются над числами с плавающей точкой, представленными с двойной точностью. Результат измеряется в MFLOPS (миллионах чисел-результатов вычислений с плавающей точкой в секунду, или миллионах элементарных арифметических операций над числами с плавающей точкой, выполненных в секунду).

Использование результатов работы тестового пакета LINPACK с двойной точностью как основы для демонстрации рейтинга MFLOPS стало общепринятой практикой в компьютерной промышленности.

В исходном тесте LINPACK [11] матрица генерируется размером 100x100, однако при таком малом размере матрицы есть риск получить завышенную оценку производительности вычислительного узла из-за влияния кэша и малого времени, затрачиваемого на реализацию теста. В связи с этим тестовая программа была модифицирована: размер матрицы был увеличен до 1000.

В результате тестирования узла имеющейся кластерной системы получена оценка в 350 MFLOPS при матрице размера 1000x1000. Если взять обратную величину от найденной, то получим оценку для времени выполнения одной арифметической операции: $t \approx 2,85 \cdot 10^{-9} \text{ с}$.

```
#include <mpi.h>
#include <stdio.h>
int main( int argc, char *argv[ ] ){
    int rank, size, j;
    long int n=1000000000;
    double t1,t2,s;
    MPI_Init( &argc, &argv );

    t1=MPI_Wtime();
    for(j=0;j<n;j++)
        s=s+0.00000000001+0.00000000001;
    t2=MPI_Wtime();
    printf("n=%d  MF=%f  t= %f s=%f\n",2*n,2*n/(1000000*(t2-t1)),t2-t1,s);

    MPI_Finalize();
    return(0);
}
```

Рис. 35. Программа для оценки производительности узла кластерной системы

Получить “грубую” оценку времени выполнения одной арифметической операции можно и с помощью простой программы, представленной на рис. 35.

На каждой итерации цикла выполняются две операции сложения с целью компенсировать влияние инкремента счетчика цикла на общее время выполнения алгоритма суммирования. Производительность узла при выполнении этой программы составила 380 MFLOPS. Данный подход для определения производительности узла был апробирован на различных системах и, в сравнении с тестом LINPACK, показал схожие результаты.

4.3. Оценка времени передачи данных

Для оценки параметра α необходимо замерить скорость передачи данных между двумя узлами кластерной системы. При этом следует учитывать, что при передаче данных малого размера существенную роль играет латентность. На рис. 36 приведена программа, которая осуществляет замер времени и характеристик при передаче одномерного массива с одного узла на другой. Для получения достоверных результатов осуществлялись 100 передач данных, после этого замерялось совокупное время на передачу данных. Это требуется, чтобы существенно превысить доступное разрешение системного таймера, которое можно получить с помощью функции `MPI_Wtick()`. Для используемой кластерной системы разрешение системного таймера составило 10^{-6} . Для каждого размера передаваемой посылки определялись следующие характеристики: количество передаваемых байт, затраченное время на одну пересылку (с), скорость передачи данных (Мбайт/с).

```
#include <mpi.h>
#include <stdio.h>
#define n 32768
int main( int argc, char *argv[ ] ){
    int rank, size;
    int i, j;
    double t1,t2;
    double *a;
    a= new double[n];
    MPI_Status stat;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    for (i=1;i<=32768;i=i*2){
        if (rank==0){
            t1=MPI_Wtime();
            for (j=0;j<100;j++){
                MPI_Send(a,i,MPI_DOUBLE,1,10,MPI_COMM_WORLD);
                MPI_Recv(a,i,MPI_DOUBLE,1,10,MPI_COMM_WORLD,&stat);
            }
            t2=MPI_Wtime();
            printf( "Count= %d Bite= %d Time=%f Mb=%f \n",
                i,i*8, (t2-t1)/200,i*8/ (t2-t1)/1024/1024*200);
        }
        else if(rank==1){
            for (j=0;j<100;j++){
                MPI_Recv(a,i,MPI_DOUBLE,0,10,MPI_COMM_WORLD,&stat);
                MPI_Send(a,i,MPI_DOUBLE,0,10,MPI_COMM_WORLD);
            }
        }
    }
    MPI_Finalize();
    delete (a);
    return(0);
};
```

Рис. 36. Программа для определения скорости передачи данных между узлами

Результаты тестирования с использованием сети Gigabit Ethernet (1Гбит/с) приведены в таблице 2.

Ранее было получено, что ускорение при использовании 2-х узлов кластерной системы будет больше 1 ($S_2 > 1$) при $n > 2(\alpha + 1)$. Анализируя данные в первой строке и последнем столбце таблицы 2, можно предположить, что ускорение больше единицы будет наблюдаться при суммировании более чем 20000 элементов, при меньшем количестве суммируемых элементов последовательная программа будет работать быстрее.

Таблица 2. Результаты тестирования скорости передачи данных

Количество передаваемых элементов	Размер посылки, байт	Время, 10^{-6} сек	Скорость передачи данных, Мбайт/с	Оценка параметра α
1	8	27	0,28	9473
2	16	25	0,61	4385
4	32	26	1,19	2280
8	64	26	2,35	1140
16	128	28	4,33	614
32	256	31	7,82	339
64	512	36	13,71	197
128	1024	51	19,30	139
256	2048	166	11,79	227
512	4096	167	23,45	114
1024	8192	238	32,77	81
2048	16384	255	61,27	43
4096	32768	419	75	35
8192	65536	693	90	29
16384	131072	1263	99	27
32768	262144	2417	104	25
65536	524288	4744	105	25
131072	1048576	9206	109	24
262144	2097152	18185	110	24
524288	4194304	36015	111	24
1048576	8388608	71577	112	23

Задания

1. Определите примерное время t , затрачиваемое на выполнение одной арифметической операции на Вашем персональном компьютере. Определите производительность в MFlops.
2. Скачайте утилиту Linx от компании Интел, которая используется для тестирования стабильности системы. По сути, данная утилита является графической оболочкой для теста Linkpack, способного заставить

процессор работать на пределе. Проведите тестирование домашнего компьютера, указав характеристики домашнего компьютера, откуда скачан тест и какие параметры его запуска. Проведите сравнение с результатами, полученными с помощью программы суммирования ряда малых чисел, подготовленной на занятии. Объясните причины расхождения полученных результатов тестирования.

3. Требуется выполнить обзор и анализ рейтингов ТОП500, ТОП50, дать описание и анализ первых 10 систем из этих рейтингов с указанием систем, вошедших в ТОП500. Укажите критерии анализа и подведите итоги проведенного анализа. Обязательные критерии анализа: архитектура вычислительного комплекса, тип сети (бренд), тип процессора, есть ли графические ускорители или сопроцессоры, ОС, инструменты параллельного программирования (коммуникационные библиотеки, компиляторы, пакеты прикладных программ), область использования вычислительного комплекса, страна размещения и сфера деятельности владельца. Обязательно сопоставьте производительность топовой системы с производительностью своего компьютера!
4. Определите латентность – задержку на подготовку канала для передачи данных. Для этого необходимо замерить время на передачу посылки 0-й длины.
5. Определите параметр α для своего домашнего компьютера. Сравните теоретическую оценку эффективности распараллеливания алгоритма суммирования ряда чисел с полученными ранее при написании программ (определите, наблюдается ли ускорение для предсказанного предела ряда).
6. Создайте последовательную программу, реализующую алгоритм скалярного произведения векторов. Размер векторов возьмите $n=(10^3, 10^4 \dots 10^9)$. Элемент 1-го вектора задайте $(1/(1+i))$, второго – $(i/(1+i))$. Создайте параллельную программу, реализующую данный алгоритм. Для реализации используйте изученные коммуникационные функции. Определите время выполнения последовательной и параллельной программ в зависимости от размера массивов (запустите несколько раз и возьмите наилучшие результаты). Параллельную программу запустите на 2-х и 4-х процессах. Требуется представить полученные значения в табличном виде, привести скрин компиляции и запуска параллельной программы. Получите теоретическую оценку ускорения параллельного алгоритма (выписать время работы последовательной программы, параллельной для 2-х процессов, взять отношение - это и будет ускорение, определить когда оно будет больше 1). Сделайте выводы о целесообразности распараллеливания данного алгоритма.

5. Передача двумерных массивов с помощью блокирующих функций "Send-Recv"

В рамках данного пункта рассматривается передача двумерных массивов с использованием уже изученных коммуникационных функций парного обмена сообщениями.

5.1. Особенности использования динамических массивов

При объявлении на языке C двумерных статических массивов размещение их в памяти осуществляется непрерывно строчным образом – сначала размещается 1 строка, следом – 2-я и т.д. Например, в случае передачи одновременно 2-х строк (предположим первых), первые три аргумента при вызове функции передачи сообщения будут следующие:

MPI_Send(&a[0][0], 2*n, MPI_INT, ...)

Первый аргумент – адрес размещения данных в оперативной памяти, с которого требуется отправить элементы, второй аргумент – сколько элементов, третий – тип данных пересылаемых элементов.

Так как строки массива расположены непрерывно (одна за другой), то данная передача будет корректной.

Использование статических массивов допустимо при малых размерах массивов и такие программы будут работать приемлемо быстро и без распараллеливания. На деле же приходится иметь дело с динамическими массивами большого размера. Вот тут и надо разобраться с тонкостями выделения памяти под динамический массив для последующей корректной передачи их между процессами.

Рассмотрим способ объявления динамического массива, используемого чаще всего на практике (см. рис. 37).

В первой строке `**a` – выделение памяти под указатель на массив указателей, `new int*[n]` – выделение памяти под массив указателей, указывающих на произвольные ячейки памяти.

Третья строка – выделение памяти под элементы строк, при этом строки будут размещены в оперативной памяти произвольно (фрагментами).

```
1. int **a=new int*[n];  
2. for(i=0; i<n; i++)  
3.   a[i]=new int[n];
```

Рис. 37. Код для выделения памяти под двумерных массив фрагментами

При попытке отправить сразу несколько строк, допустим две первых, первая строка отправится корректно, а далее в оперативной памяти будет взят фрагмент, идущий за первой строкой, но не содержащий вторую. При этом, если данный участок оперативной памяти занят другой программой, то

запускаемая Вами параллельная программа будет завершена аварийно с появлением ошибки памяти. В противном случае – будет передана информация, не соответствующая второй строке.

Рассмотрим альтернативный способ выделения памяти под двумерный динамический массив (см. рис. 38).

Первая строка – выделение памяти под массив указателей на начала строк осталась без изменений. Указатели по-прежнему указывают на произвольные ячейки памяти.

Вторая строка – выделение памяти непрерывно (линейно) одним фрагментом сразу под весь двумерный массив. При этом $a[0]$ – указатель на 0-ю строку (начало всего массива).

Осталось только правильно расставить указатели на начала оставшихся строк, что и сделано в цикле в 3-4 строках.

```
1. int **a=new int*[n];
2. a[0]=new int [n*n];
3. for(i=1; i<n; i++)
4. a[i]=a[i-1]+n;
```

Рис. 38. Фрагмент кода выделения памяти под двумерный массив непрерывным куском

5.2. Примеры

Пример 1.

Рассмотрим пример отправки двух первых строк двумерного массива с 0-го процесса на 1-й.

Итак, на рис. 39 приведен листинг программы, реализующей следующий алгоритм:

0-й процесс	1-й процесс
Выделение памяти под двумерный массив a из n строк и n столбцов	
Зануление элементов массива	
Присваивание значений элементам массива для последующей проверки корректности передачи данных	
Отправка двух первых строк 1-му процессу	Прием $2*n$ элементов от 0-го процесса и размещение их вначале массива
	Вывод на экран номера процесса и массива a

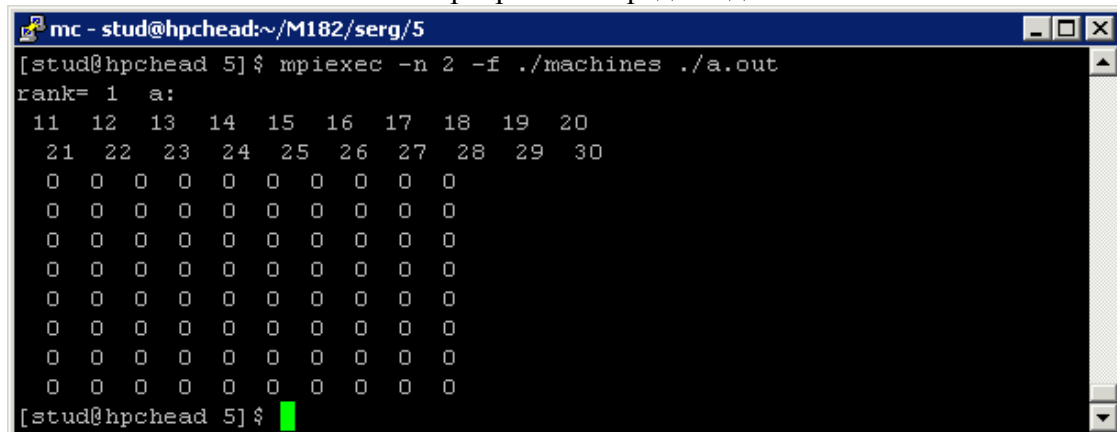
Для обозначения адреса, с которого будет осуществлять отправка элементов, можно использовать следующие варианты (1-й аргумент в функциях `MPI_Send`, `MPI_Recv`):

- $*a$;
- $a[0]$;
- $\&a[0][0]$.

На рис. 40 приведен скрин запуска данной программы на 2-х процессах.

```
1.  #include <stdio.h>
2.  #include "mpi.h"
3.  #include "iostream"
4.  int main(int argc, char *argv[])
5.  {
6.      int rank;
7.      int size;
8.      int n=10,i,s=0, j;
9.      MPI_Status stat;
10.     MPI_Init(&argc, &argv);
11.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12.     MPI_Comm_size(MPI_COMM_WORLD, &size);
13.     int **a=new int *[n];
14.     a[0]=new int [n*n];
15.     for(i=1; i<n; i++)
16.         a[i]=a[i-1]+n;
17.     for(i=0; i<n; i++)
18.         for(j=0; j<n; j++)
19.             a[i][j]=0;
20.     if(rank==0)
21.     {
22.         for(i=0; i<n; i++)
23.             for(j=0; j<n; j++)
24.                 a[i][j]=10*(i+1)+j+1;
25.     MPI_Send(a[0],2*n,MPI_INT,1,777,MPI_COMM_WORLD);
26.     }
27.     if(rank==1)
28.     {
29.         MPI_Recv(a[0],2*n,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
30.         printf("rank= %d  a: \n",rank);
31.         for(i=0; i<n; i++)
32.         {
33.             for(j=0; j<n; j++)
34.                 printf(" %d ",a[i][j]);
35.             printf("\n ");
36.         }
37.     }
38.     MPI_Finalize();
39.     return 0;
40. }
```

Рис. 39. Листинг программы передачи данных



```
mc - stud@hpchead:~/M182/serg/5
[stud@hpchead 5]$ mpirun -n 2 -f ./machines ./a.out
rank= 1  a:
 11 12 13 14 15 16 17 18 19 20
 21 22 23 24 25 26 27 28 29 30
 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0
[stud@hpchead 5]$
```

Рис. 40. Скрин запуска программы на 2 процессах

Пример 2.

Рассмотрим пример распределения двумерного массива с 0-го процесса на все остальные с пересылкой по две строки каждому.

```
1.  #include <stdio.h>
2.  #include "mpi.h"
3.  #include "iostream"
4.  int main(int argc, char *argv[])
5.  {
6.      int rank;
7.      int size;
8.      int n=10,i,s=0, j;
9.      MPI_Status stat;
10.     MPI_Init(&argc, &argv);
11.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12.     MPI_Comm_size(MPI_COMM_WORLD, &size);
13.     if(rank==0)
14.     {
15.         int **a=new int *[2*size];
16.         a[0]=new int [2*size*n];
17.         for(i=1; i<2*size; i++)
18.             a[i]=a[i-1]+n;
19.         for(i=0; i<2*size; i++)
20.             for(j=0; j<n; j++)
21.                 a[i][j]=10*(i+1)+j+1;
22.         for(i=1; i<size; i++)
23.             MPI_Send(a[2*i],2*n,MPI_INT,i,777,MPI_COMM_WORLD);
24.     }
25.     else
26.     {
27.         int **b=new int *[2];
28.         b[0]=new int [2*n];
29.         for(i=1; i<2; i++)
30.             b[i]=b[i-1]+n;
31.         MPI_Recv(b[0],2*n,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
32.
33.         printf("rank= %d b: \n",rank);
34.         for(i=0; i<2; i++)
35.         {
36.             for(j=0; j<n; j++)
37.                 printf(" %d ",b[i][j]);
38.             printf("\n ");
39.         }
40.     }
41.     MPI_Finalize();
42.     return 0;
43. }
```

Рис. 41. Листинг программы передачи данных

Итак, на рис. 41 приведен листинг программы, реализующей следующий алгоритм:

0-й процесс	Ненулевые процессы
Выделение памяти под двумерный массив a из 2*size строк и n столбцов	Выделение памяти под двумерный массив b из 2 строк и n столбцов
Присваивание значений элементам	

массива для последующей проверки корректности передачи данных	
Отправка в цикле по две строки каждому ненулевому процессу	Прием $2*n$ элементов от 0-го процесса и размещение их вначале массива
	Вывод на экран номера процесса и массива b

На рис. 42 приведен скрин компиляции и запуска данной программы на 4-х процессах.

```

mc - stud@hpchead:~/M182/serg/5
[stud@hpchead 5]$ mpic++ send2.c
Press any key to continue...
[stud@hpchead 5]$ mpiexec -n 4 -f ./machines ./a.out
rank= 1  b:
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
rank= 3  b:
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
rank= 2  b:
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
[stud@hpchead 5]$

```

Рис. 42. Скрин компиляции и запуска программы на 4 процессах

Задания

1. Напишите программу, используя блокирующие коммуникационные функции (MPI_Send, MPI_Recv), реализующую построчное распределение двумерного массива, расположенного в памяти 0-го процессора. Все остальные процессы выводят номер процесса и полученную строку от 0-го процесса. В момент запуска параллельного приложения соберите трассу выполнения программы и приведите рисунок трассы в отчете.
2. Напишите программу, используя блокирующие коммуникационные функции (MPI_Send, MPI_Recv), реализующую построчную сборку двумерного массива: на каждом процессе задается одномерный массив, который передается на 0-й, 0-й принимает одномерный массив, сразу сохраняя в соответствующую строку двумерного массива.
3. Напишите программу для распределения по три строки на каждый ненулевой процесс двумерного массива, расположенного в памяти 0-го процесса. На ненулевых процессах для приема объявите трехстрочный массив.

4. Напишите программу для сборки по три строки с каждого процесса на 0-й, который осуществляет прием сразу в двумерный массив.
5. Напишите программу, которая собирает на 0-м процессе в двумерный массив следующие данные, присылаемые с ненулевых: по (rank+1) строк с каждого.
6. Напишите программу, которая осуществляет рассылку строк двумерного массива с 0-го процесса все остальным по следующему правилу: на 1-й процесс отправляются 1-2 строки, на 2-й процесс – 3-5 строки, на 3-й – 5-9 строки и т.д.
7. Напишите программу, которая реализует следующий алгоритм: на 0-м процессе задается двумерный массив $N \times N$ по формуле: $a(i,j)=10*(i+1)+(j+1)$, который по одной строке рассылается всем ненулевым процессам по кругу. Например, Вы запускаете программу на 3-х процессах, 1-й процесс в итоге должен получить 1, 4, 7 и т.д., 2-й процесс – 2, 5, 8 и т.д. Не забудьте сделать контрольный вывод пересланных данных.
8. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ и заполняется единицами. 2) на 1-м процессе создается такая же матрица и заполняется нулями. 3) требуется передать с 0-го на 1-й процесс элементы главной и побочной диагоналей (должен получиться крестик).
9. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$. 2) первый и последний столбец, первая и последняя строка передаются с 0-го процесса на 1-й. 3) 1-й процесс сохраняет полученные элементы в точно такие же места в матрицу (должна получиться рамка по периметру).
10. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ по формуле: $a(i,j)=10*(i+1)+(j+1)$ 2) на 1-м процессе создается такая же матрица и заполняется нулями. 3) требуется передать с 0-го на 1-й процесс четные строки.
11. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ по формуле: $a(i,j)=10*(i+1)+(j+1)$ 2) на 1-м процессе создается такая же матрица и заполняется нулями. 3) требуется передать с 0-го на 1-й процесс нечетные строки.
12. Напишите последовательный строчный алгоритм умножения квадратной матрицы на вектор. Определите время выполнения в зависимости от размера матрицы (матрицу возьмите размером от 1000 до возможного на Вашем компьютере) с шагом 1000. Для возможности задания больших матриц необходимо их динамическое объявление. Напишите параллельную программу: каждый процесс производит умножение строк на вектор ($i=\text{rank}, n, \text{size}$) или блоками строк, получившиеся результаты

собираются на 0-м процессе. Для реализации используйте изученные коммуникационные функции Send-Recv. Получите время выполнения параллельного алгоритма в зависимости от размера матрицы. Определите ускорение и эффективность параллельного алгоритма. Количество процессов в параллельном исполнении задайте 2 и 4. Все данные приведите в табличном и графическом виде.

6. Эффективное использование статусной информации при приеме сообщений

В рамках данного пункта подробно разберем ситуации, при которых можно существенно экономить время при передаче данных, используя все те же функции точечного обмена информацией.

6.1. Прием сообщения от произвольного процесса-отправителя

Существует множество алгоритмов, при которых окончательный результат собирается на 0-м процессе и при этом порядок, в котором должен осуществляться прием сообщений, не важен. Например, в ранее реализованном параллельном алгоритме суммирования ряда чисел все ненулевые процессы шлют результат своих вычислений (частичные суммы) на 0-й процесс, который после приема сообщения от очередного процесса прибавляет полученный промежуточный результат к своей сумме. Итоговая сумма ряда в данном случае не зависит от порядка приема сообщений от ненулевых процессов.

При приеме вместо параметра `source` можно использовать предопределенный параметр `MPI_ANY_SOURCE` – читать сообщение от любого отправителя, вместо параметра идентификатора сообщения `tag` – `MPI_ANY_TAG` – читать сообщение с любым идентификатором.

В случае необходимости, можно узнать от какого процесса пришло сообщение по статусной информации – атрибуты полученного сообщения доступны в полях структуры `status` после приема сообщения. Эта структура типа `MPI_Status` с тремя полями `MPI_SOURCE`, `MPI_TAG`, `MPI_ERROR`.

Пример 1.

Рассмотрим пример, демонстрирующий использование `MPI_ANY_SOURCE` вместо указания процесса-отправителя при приеме сообщения. Также в этом примере покажем, что сообщения от множества процессов приходят произвольно. Для этого напомним программу, реализующую следующий алгоритм:

0-й процесс	Ненулевые процессы
В цикле прием поступающих сообщений и их вывод на экран.	Отправка своего номера (<code>rank</code>) на 0-й процесс.

Так как процессы отправляют свой номер, то 0-й процесс, выводя на экран полученные сообщения, распечатает порядок получения посылок от процессов!

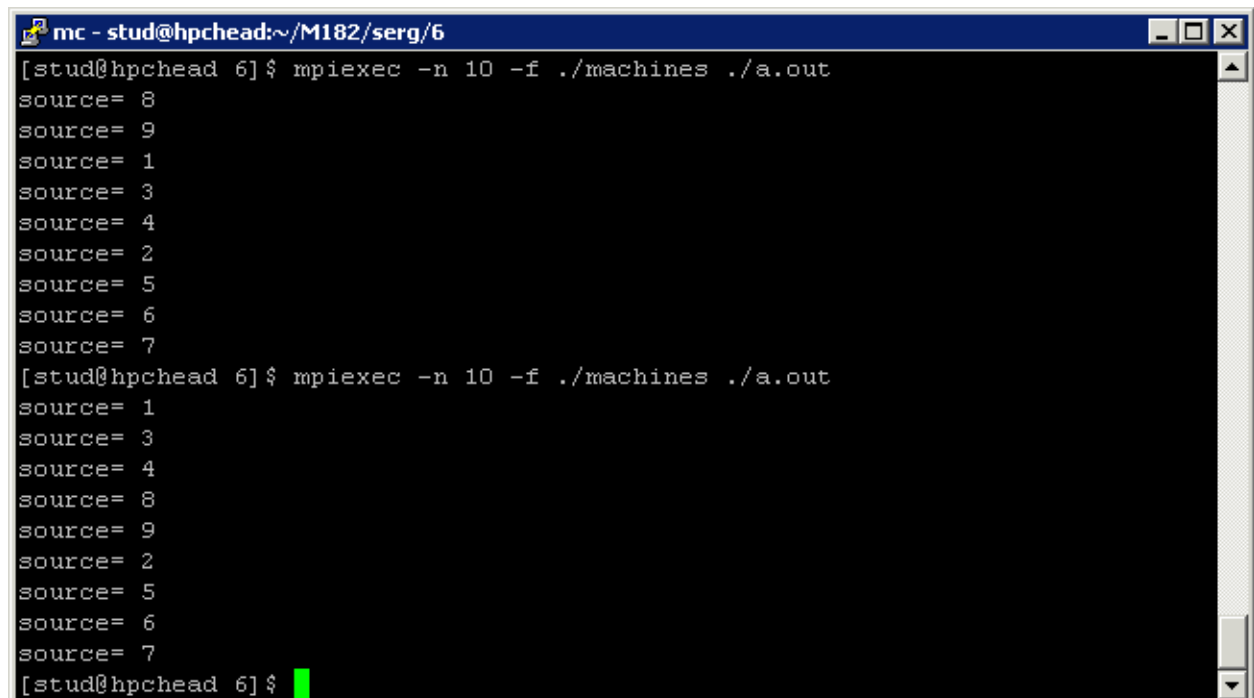
На рис. 43 приведен листинг, реализующий заданный алгоритм, а на рис. 44 – скрин запуска данной программы на 10-и процессах два раза. Видим, что от запуска к запуску порядок приема меняется.

```

1. #include <stdio.h>
2. #include "mpi.h"
3.
4. int main(int argc, char *argv[])
5. {
6.     int rank;
7.     int size;
8.     int n=0;
9.     MPI_Status stat;
10.    MPI_Init(&argc, &argv);
11.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
12.    MPI_Comm_size(MPI_COMM_WORLD, &size);
13.    if(rank==0)
14.    {
15.        for(int i=1;i<size;i++)
16.        {
17.            MPI_Recv(&n,1,MPI_INT,MPI_ANY_SOURCE,777,MPI_COMM_WORLD,&stat);
18.            printf("source= %d \n",n);
19.        }
20.    }
21.    else
22.    {
23.        MPI_Send(&rank,1,MPI_INT,0,777,MPI_COMM_WORLD);
24.    }
25.    MPI_Finalize();
26.    return 0;
27. }

```

Рис. 43. Листинг программы передачи данных



```

mc - stud@hpchead:~/M182/serg/6
[stud@hpchead 6]$ mpiexec -n 10 -f ./machines ./a.out
source= 8
source= 9
source= 1
source= 3
source= 4
source= 2
source= 5
source= 6
source= 7
[stud@hpchead 6]$ mpiexec -n 10 -f ./machines ./a.out
source= 1
source= 3
source= 4
source= 8
source= 9
source= 2
source= 5
source= 6
source= 7
[stud@hpchead 6]$

```

Рис. 44. Скрин запуска программы на 10 процессах

Пример 2.

В предыдущем примере мы определяли от кого пришло сообщение по содержимому самого сообщения – оно содержало номер процесса-отправителя. В данном примере будем отправлять сообщение, не позволяющего по нему определить номер процесса-отправителя. Например, отправим с каждого процесса значение MPI_Wtime. В этом случае узнать информацию от кого пришло сообщение поможет статус принятого сообщения.

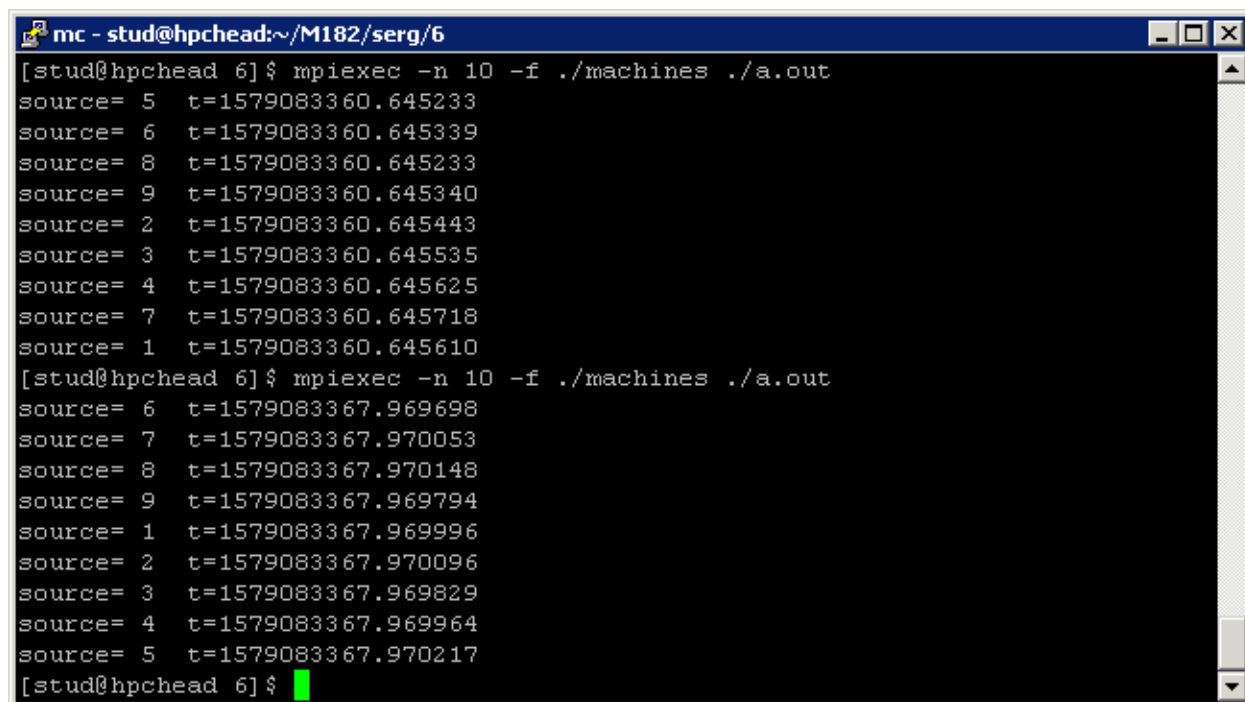
```
1. #include <stdio.h>
2. #include "mpi.h"
3.
4. int main(int argc, char *argv[])
5. {
6.     int rank;
7.     int size;
8.     int n=0;
9.     double t;
10.    MPI_Status stat;
11.    MPI_Init(&argc, &argv);
12.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13.    MPI_Comm_size(MPI_COMM_WORLD, &size);
14.
15.    if(rank==0)
16.    {
17.        for(int i=1;i<size;i++)
18.        {
19.            MPI_Recv(&t,1,MPI_DOUBLE,MPI_ANY_SOURCE,777,MPI_COMM_WORLD,&stat);
20.            printf("source= %d t=%f \n",stat.MPI_SOURCE,t);
21.        }
22.    }
23.    else
24.    {
25.        t=MPI_Wtime();
26.        MPI_Send(&t,1,MPI_DOUBLE,0,777,MPI_COMM_WORLD);
27.    }
28.    MPI_Finalize();
29.    return 0;
30. }
```

Рис. 45. Листинг программы передачи данных

Напишем программу, реализующую следующий алгоритм:

0-й процесс	Ненулевые процессы
	Запись в переменную типа double значения, возвращаемого функцией MPI_Wtime.
В цикле: - прием поступающих сообщений, - вывод процесса отправителя, используя статус сообщения, и принятого сообщения.	Отправка значения данной переменной на 0-й процесс.

На рис. 45 приведен листинг программы, реализующий заданный алгоритм, а на рис. 46 – скрин запуска данной программы на 10-и процессах два раза. Видим, что от запуска к запуску порядок приема меняется, а благодаря статусной информации, мы имеем возможность узнать от кого получено сообщение.



```
mc - stud@hpchead:~/M182/serg/6
[stud@hpchead 6]$ mpirun -n 10 -f ./machines ./a.out
source= 5 t=1579083360.645233
source= 6 t=1579083360.645339
source= 8 t=1579083360.645233
source= 9 t=1579083360.645340
source= 2 t=1579083360.645443
source= 3 t=1579083360.645535
source= 4 t=1579083360.645625
source= 7 t=1579083360.645718
source= 1 t=1579083360.645610
[stud@hpchead 6]$ mpirun -n 10 -f ./machines ./a.out
source= 6 t=1579083367.969698
source= 7 t=1579083367.970053
source= 8 t=1579083367.970148
source= 9 t=1579083367.969794
source= 1 t=1579083367.969996
source= 2 t=1579083367.970096
source= 3 t=1579083367.969829
source= 4 t=1579083367.969964
source= 5 t=1579083367.970217
[stud@hpchead 6]$
```

Рис. 46. Скрин запуска программы на 10 процессах

6.2. Определение процесса-отправителя до приема сообщения

Пример 3.

Усложним задачу – допустим при реализации алгоритма нам необходимо еще до приема сообщения узнать от кого оно пришло. Другими словами, нам необходимо узнать статус сообщения в приемном буфере без его получения. Именно для этой цели предназначена функция `MPI_Probe`:

```
int MPI_Probe (int source, int tag, MPI_Comm comm,  
MPI_Status *status)
```

Входные аргументы:

source - номер процесса-отправителя;

tag - идентификатор сообщения;

comm - коммуникатор.

Выходные аргументы:

status - атрибуты опрошенного сообщения.

Вместо `source` и `tag` можно поставить предопределенные переменные `MPI_ANY_SOURCE` и `MPI_ANY_TAG`.

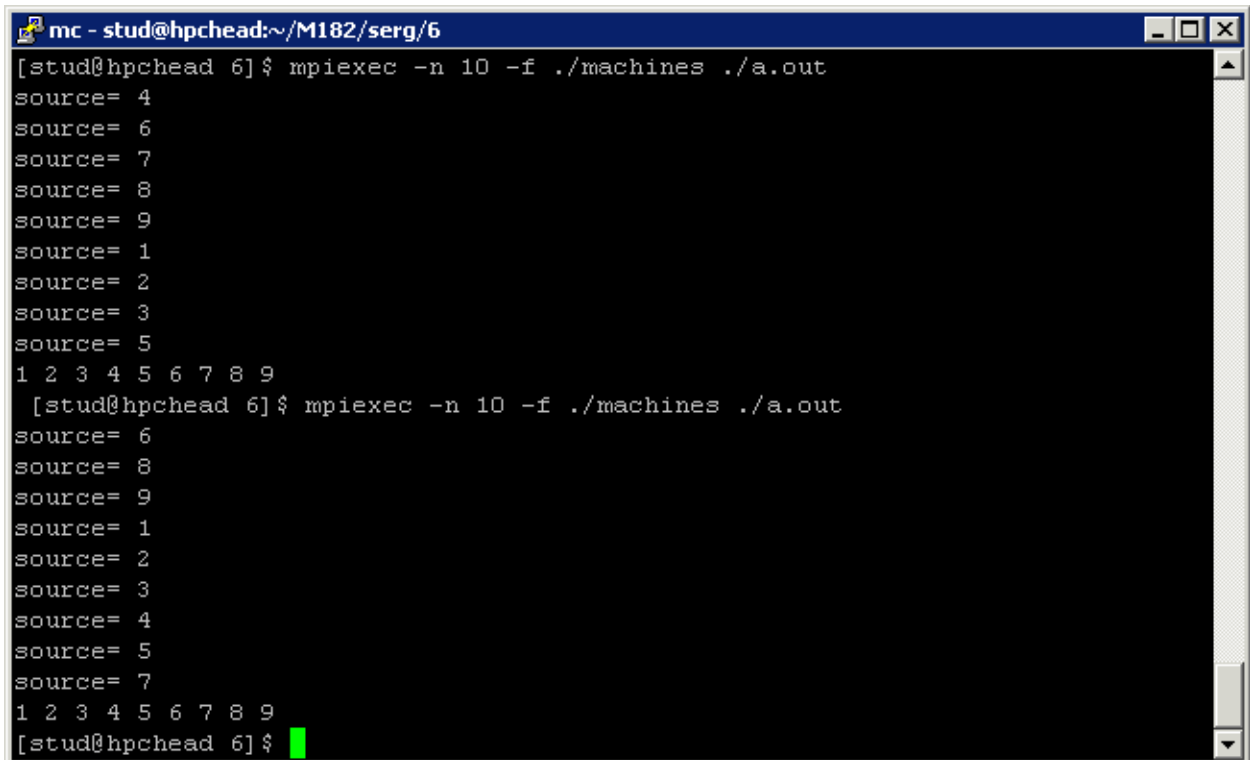
Для демонстрации использования новой функции реализуем следующий алгоритм – каждый процесс отправляет на 0-й свой номер, 0-й процесс

должен перед получением определить от кого пришло сообщение и при приеме сообщения сразу записать в ячейку одномерного массива, соответствующую номеру процесса-отправителя. В итоге получим следующее – сообщения принимались от того процесса, от которого пришли раньше, а итоговый одномерный массив будет содержать все цифры по порядку!

На рис. 47 приведен листинг программы, реализующий заданный алгоритм, а на рис. 48 – скрин запуска данной программы на 10-и процессах два раза. Видим, что от запуска к запуску порядок приема меняется, а благодаря статусной информации, полученной с помощью функции `MPI_Probe`, мы имеем возможность записать полученные элементы на верные позиции в массив. Следует обратить внимание на то, что при приеме сообщения теперь необходимо указывать номер процесса-отправителя для избегания ситуации, когда функция `MPI_Probe` получила статус одного сообщения, а функция `MPI_Recv` получила другое сообщение!

```
1. #include <stdio.h>
2. #include "mpi.h"
3.
4. int main(int argc, char *argv[])
5. {
6.     int rank;
7.     int size;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.
13.    if(rank==0)
14.    {
15.        int *a = new int[size];
16.        for(int i=1;i<size;i++)
17.        {
18.            MPI_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&stat);
19.            MPI_Recv(&a[stat.MPI_SOURCE],1,MPI_INT,stat.MPI_SOURCE,777,MPI_COMM_WORLD,&stat);
20.            printf("source= %d \n",stat.MPI_SOURCE);
21.        }
22.        for(int i=1; i<size; i++)
23.            printf("%d ", a[i]);
24.        printf("\n");
25.    }
26.    else
27.    {
28.        MPI_Send(&rank,1,MPI_INT,0,777,MPI_COMM_WORLD);
29.    }
30.
31.
32.    MPI_Finalize();
33.    return 0;
34. }
```

Рис. 47. Листинг программы передачи данных



```
mc - stud@hpchead:~/M182/serg/6
[stud@hpchead 6]$ mpiexec -n 10 -f ./machines ./a.out
source= 4
source= 6
source= 7
source= 8
source= 9
source= 1
source= 2
source= 3
source= 5
1 2 3 4 5 6 7 8 9
[stud@hpchead 6]$ mpiexec -n 10 -f ./machines ./a.out
source= 6
source= 8
source= 9
source= 1
source= 2
source= 3
source= 4
source= 5
source= 7
1 2 3 4 5 6 7 8 9
[stud@hpchead 6]$
```

Рис. 48. Скрин запуска программы на 10 процессах

6.3. Определение количества элементов в приемном буфере до приема сообщения

Пример 4.

Дополнительно усложним задачу – допустим при реализации алгоритма нам необходимо еще до приема сообщения узнать не только от кого оно пришло, но и сколько элементов в нем содержится. В этом случае кроме функции `MPI_Probe` нам понадобится следующая:

```
int MPI_Get_count (MPI_Status *status, MPI_Datatype  
datatype,  
int *count)
```

Входные аргументы:

Status – атрибуты принятого сообщения;

Datatype – тип элементов принятого сообщения.

Выходные аргументы:

count – число полученных элементов.

Для демонстрации использования функции `MPI_Get_count` реализуем следующий алгоритм – каждый процесс отправляет на 0-й (rank) элементов, 0-й процесс должен перед получением определить от кого пришло сообщение и сколько в нем элементов, затем при приеме сообщения сразу записать в строку двумерного массива, соответствующую номеру процесса-

отправителя. В итоге получим следующее – сообщения принимались от того процесса, от которого пришли раньше, а итоговый двумерный массив будет содержать 1 элемент в первой строке, 2 – во второй и т.д.

На рис. 49 приведен листинг программы, реализующий заданный алгоритм, а на рис. 50 – скрин запуска данной программы на 10-и процессах два раза. Видим, что от запуска к запуску порядок приема меняется, а благодаря статусной информации, полученной с помощью функции MPI_Probe, мы имеем возможность записать полученные элементы на верные позиции в массив.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=0, count, i, j;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    if(rank==0)
13.    {
14.        n=size;
15.        int **a=new int *[n];
16.        a[0]=new int [n*n];
17.        for(i=1; i<n; i++)
18.            a[i]=a[i-1]+n;
19.        for(i=0; i<n; i++)
20.            for(j=0; j<n; j++)
21.                a[i][j]=0;
22.        for (i=1;i<size;i++)
23.        {
24.            MPI_Probe(MPI_ANY_SOURCE,MPI_ANY_TAG,MPI_COMM_WORLD,&stat);
25.            MPI_Get_count(&stat,MPI_INT,&count);
26.            MPI_Recv(a[stat.MPI_SOURCE],count,MPI_INT,stat.MPI_SOURCE,777,MPI_COMM_WORLD,&stat);
27.            printf("source= %d \n",stat.MPI_SOURCE);
28.        }
29.        for(i=0; i<n; i++)
30.        {
31.            for(j=0; j<n; j++)
32.                printf(" %d ",a[i][j]);
33.            printf("\n ");
34.        }
35.    }
36.    else
37.    {
38.        int *a = new int[rank];
39.        for (i=0;i<rank;i++)
40.            a[i]=rank;
41.        MPI_Send(a,rank,MPI_INT,0,777,MPI_COMM_WORLD);
42.    }
43.    MPI_Finalize();
44.    return 0;
45. }
```

Рис. 49. Листинг программы передачи данных



```
mc - stud@hpchead:~/M182/serg/6
[stud@hpchead 6]$ mpic++ send_source3.c
Press any key to continue...
[stud@hpchead 6]$ mpiexec -n 10 -f ./machines ./a.out
source= 1
source= 2
source= 5
source= 7
source= 8
source= 9
source= 3
source= 4
source= 6
0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0
2 2 0 0 0 0 0 0 0 0
3 3 3 0 0 0 0 0 0 0
4 4 4 4 0 0 0 0 0 0
5 5 5 5 5 0 0 0 0 0
6 6 6 6 6 6 0 0 0 0
7 7 7 7 7 7 7 0 0 0
8 8 8 8 8 8 8 8 0 0
9 9 9 9 9 9 9 9 9 0
[stud@hpchead 6]$
```

Рис. 50. Скрин запуска программы на 10 процессах

Задания

1. Напишите программу, аналогичную из примера 4, но на каждом ненулевом процессе сформируйте одномерный массив длины, полученной датчиком случайных чисел от 1 до size. Для проверки корректности работы программы выведите отправляемые массивы на ненулевых процессах и полученный результат на 0-м процессе.
2. Создайте последовательную программу (она уже должна быть выполнена в 1-й домашке), реализующую алгоритм суммирования ряда чисел для предела суммы $n=(10^3, 10^4 \dots 10^9)$. Член ряда равен $(1/(1+i))$. Создайте параллельную программу, реализующую данный алгоритм. Для реализации используйте изученные коммуникационные функции, а также возможности приема данных от неизвестного отправителя (MPI_ANY_SOURCE). Определите время выполнения последовательной и параллельной программ в зависимости от предела суммы (запустите несколько раз и возьмите наилучшие результаты). Параллельную программу запустите на 2-х и 4-х процессах. Требуется представить полученные значения в табличном виде, привести скрин компиляции и запуска параллельной программы. Сравните время выполнения параллельной программы на большом количестве процессов с использованием предопределенной переменной MPI_ANY_SOURCE и без нее.

3. Модифицируйте ранее созданную параллельную программу умножения квадратной матрицы на вектор путем использования приема получения результатов от того процесса, который пришлет свои данные раньше. Сравните время работы параллельных программ: до модификации и после.

7. Дополнительные коммуникации парного обмена

К операциям типа “Точка-Точка” (парного обмена) относятся две уже рассмотренные коммуникационные функции – MPI_Send и MPI_Recv. В коммуникационных операциях типа точка-точка всегда участвуют не более двух процессов: передающий и принимающий. В MPI имеется множество функций, реализующих такой тип обменов. Многообразие объясняется возможностью организации таких обменов множеством способов. Уже изученные функции реализуют *стандартный режим с блокировкой*.

Блокирующие функции подразумевают полное окончание операции после выхода из процедуры, то есть вызывающий процесс блокируется, пока операция не будет завершена. Для функции отправки сообщения это означает, что все пересылаемые данные помещены в буфер (для разных реализаций MPI это может быть либо какой-то промежуточный системный буфер, либо непосредственно буфер получателя). Для функции приема сообщения блокируется выполнение других операций, пока все данные из буфера не будут помещены в адресное пространство принимающего процесса.

Неблокирующие функции подразумевают совмещение операций обмена с другими операциями, поэтому неблокирующие функции передачи и приема по сути дела являются функциями инициализации соответствующих операций. Для опроса завершенности операции (и завершения) вводятся дополнительные функции.

Как для блокирующих, так и неблокирующих операций MPI поддерживает четыре режима выполнения. Эти режимы касаются только функций передачи данных, поэтому для блокирующих и неблокирующих операций имеется по четыре функции отправки сообщения. В таблице 3 перечислены имена базовых коммуникационных функций типа точка-точка, имеющих в библиотеке MPI.

Таблица 3: Перечень функций двухточечного обмена

Способ связи	С блокировкой	Без блокировки
Стандартная посылка	MPI_Send	MPI_Isend
Синхронная посылка	MPI_Ssend	MPI_Issend
Буферизованная посылка	MPI_Bsend	MPI_Ibsend
Согласованная посылка	MPI_Rsend	MPI_Irsend
Прием информации	MPI_Recv	MPI_Irecv

Из таблицы хорошо виден принцип формирования имен функций. К именам базовых функций Send/Recv добавляются различные префиксы.

Префикс S (synchronous) – означает синхронный режим передачи данных.

Операция передачи данных заканчивается только тогда, когда

- заканчивается прием данных. Функция нелокальная.
- Префикс B (buffered) – означает буферизованный режим передачи данных. В адресном пространстве передающего процесса с помощью специальной функции создается буфер обмена, который используется в операциях обмена. Операция отправки заканчивается, когда данные помещены в этот буфер. Функция имеет локальный характер.
- Префикс R (ready) – согласованный или подготовленный режим передачи данных. Операция передачи данных начинается только тогда, когда принимающий процессор выставил признак готовности приема данных. Функция нелокальная.
- Префикс I (immediate) – относится к неблокирующим операциям.

Все функции передачи и приема сообщений могут использоваться в любой комбинации друг с другом. Функции передачи, находящиеся в одном столбце, имеют совершенно одинаковый синтаксис и отличаются только внутренней реализацией. Поэтому в дальнейшем будем рассматривать только стандартный режим, который в обязательном порядке поддерживают все реализации MPI.

В рамках данного пункта подробно разберем функции неблокирующей передачи данных, а также функции, реализующие одновременную передачу данных. Использование данных функций позволяет значительно сократить время на передачу данных, а значит повысить эффективность распараллеливания программ.

7.1. Неблокирующие коммуникационные операции

Использование неблокирующих коммуникационных операций более безопасно с точки зрения возникновения тупиковых ситуаций, а также может увеличить скорость работы программы за счет совмещения выполнения вычислительных и коммуникационных операций. Эти задачи решаются разделением коммуникационных операций на две стадии: инициирование операции и проверку завершения операции, которые могут быть разнесены по времени.

Приведем пример, демонстрирующий экономию времени при использовании неблокирующих функций передачи данных. Рассмотрим следующую ситуацию: требуется помыть парты в двух компьютерных классах, расположенных друг от друга на значительном расстоянии. Данную работу поручили выполнить двум девушкам – Маше и Насте, а помогать им поручили Борису. Для выполнения работы выделено два ведра, две тряпочки и один кусок мыла. Протокол работы при этом следующий: для того, чтобы вымыть парту, сначала надо взять мыло и намылить тряпку, затем намылить стол, прополоскать тряпку в ведре и протереть стол несколько раз. Эту работу будут выполнять девушки, помощь же Бориса заключается в доставке мыла из одного компьютерного класса в другой. Алгоритм работы с

использованием блокирующих функций можно представить следующим образом:

Маша	Настя
взять мыло и намылить тряпку	Ожидание (MPI_Recv мыла)
намылить стол	Ожидание
прополоскать тряпку в ведре и протереть стол несколько раз	Ожидание
Отправить Бориса с мылом в класс, который моет Настя (MPI_Send)	Ожидание и, спустя некоторое время, необходимое для перехода из одного класса в другой, получение мыла
Ожидание (MPI_Recv мыла), чтобы помыть следующую парту	взять мыло и намылить тряпку
Ожидание	намылить стол
Ожидание	прополоскать тряпку в ведре и протереть стол несколько раз
Ожидание и, спустя некоторое время, необходимое для перехода из одного класса в другой, получение мыла	Отправить Бориса с мылом в класс, который моет Маша (MPI_Send)
взять мыло и намылить тряпку	Ожидание (MPI_Recv мыла), чтобы помыть следующую парту
И т.д.	И т.д.

Что мы видим на этом примере? Мыло – разделяемый ресурс, который нужен каждому для работы. В данном алгоритме четко прослеживается поочередный простой в работе из-за отсутствия мыла (в данный период работает только один человек), дополнительно тратится время на доставку мыла из одного класса в другой (в данный период времени никто не работает).

Изменим алгоритм работы с использованием неблокирующих функций передачи данных:

Маша	Настя
взять мыло и намылить тряпку	Ожидание (MPI_Recv мыла)
Дать мыло Борису, чтобы он отнес его Нaste (MPI_Isend)	Ожидание и, спустя некоторое время, необходимое для перехода из одного класса в другой, получение мыла
Приготовить мыльницу, чтобы Борис положил в нее мыло, когда вернется от Насти (MPI_Irecv)	взять мыло и намылить тряпку
намылить стол	Дать мыло Борису, чтобы он отнес его Маше (MPI_Isend)
прополоскать тряпку в ведре и протереть стол несколько раз	Приготовить мыльницу, чтобы Борис положил в нее мыло, когда вернется

	от Маши (MPI_Irecv)
Проверить мыльницу и, если мыла в ней еще нет, то ждать его доставки (MPI_Wait)	намылить стол
взять мыло и намылить тряпку	прополоскать тряпку в ведре и протереть стол несколько раз
Дать мыло Борису, чтобы он отнес его Насе (MPI_Isend)	Проверить мыльницу и, если мыла в ней еще нет, то ждать его доставки (MPI_Wait)
Приготовить мыльницу, чтобы Борис положил в нее мыло, когда вернется от Наси (MPI_Irecv)	взять мыло и намылить тряпку
намылить стол	Дать мыло Борису, чтобы он отнес его Насе (MPI_Isend)
прополоскать тряпку в ведре и протереть стол несколько раз	Приготовить мыльницу, чтобы Борис положил в нее мыло, когда вернется от Наси (MPI_Irecv)
И т.д.	И т.д.

Что изменилось? Даже на этом алгоритме мы видим, что работа ведется значительно эффективнее предыдущего способа организации уборки классов. Основные моменты: прежде чем мыть очередную парту, девушки освобождают мыло (разделяемый ресурс) и отправляют Бориса с посылкой соседке, а только потом, приготовив мыльницу, выполняют свой очередной блок работы, не тратя времени на ожидание мыла. Вполне возможно, что пока моется очередная парта, Борис успеет уже донести мыло. В этом случае простоя вообще не будет.

Данный прием – сначала выполни все вычисления, результат которых нужен другим процессам и отправь их им, а только потом производи остальные вычисления, называется ”опережающее вычисление и рассылка”. В результате применения данного приема можно добиться ускорения, близкого к идеальному. Например, при реализации семестровой работы ”Игра в жизнь” с использованием данного приема и неблокирующих коммуникационных функций можно получить эффективность распараллеливания на 2-х процессах около 100%, а с использованием стандартных блокирующих функций – около 60%, то есть чуть быстрее, чем на 1 процессе!

Рассмотрим далее синтаксис функций, используемых при неблокирующих передачах данных, а также ряд примеров.

Неблокирующие операции используют специальный объект "запрос обмена" (request) для связи между функциями обмена и функциями опроса их завершения. В параллельной программе доступ к этому объекту возможен только через вызовы специальных MPI-функций. Если операция обмена

завершена, функция проверки обнуляет "запрос обмена", устанавливая его в значение MPI_REQUEST_NULL. Снять запрос без ожидания завершения операции можно подпрограммой MPI_Request_free.

Функция передачи сообщения без блокировки MPI_Isend.

```
int MPI_Isend(void* buf, int count, MPI_Datatype  
datatype, int dest, int tag, MPI_Comm comm, MPI_Request  
*request)
```

Входные аргументы:

buf - адрес начала расположения передаваемых данных;
count - число посылаемых элементов;
datatype - тип посылаемых элементов;
dest - номер процесса-получателя;
tag - идентификатор сообщения;
comm - коммуникатор.

Выходные аргументы:

request - "запрос обмена".

Возврат из подпрограммы происходит немедленно (immediate), без ожидания окончания передачи данных. Этим объясняется префикс I в именах функций. Поэтому переменную buf повторно использовать нельзя до тех пор, пока не будет погашен "запрос обмена". Это можно сделать с помощью подпрограмм MPI_Wait или MPI_Test, передав им параметр request. При этом можно производить вычисления, которые не используют передаваемую информацию.

Функция приема сообщения без блокировки MPI_Irecv.

```
int MPI_Irecv(void* buf, int count, MPI_Datatype  
datatype, int source, int tag, MPI_Comm comm,  
MPI_Request *request)
```

Входные аргументы:

count - максимальное число принимаемых элементов;
datatype - тип элементов принимаемого сообщения;
source - номер процесса-отправителя;
tag - идентификатор сообщения;
comm - коммуникатор.

Выходные аргументы:

buf - адрес для принимаемых данных;
request - "запрос обмена".

Возврат из подпрограммы происходит немедленно, без ожидания окончания приема данных. Определить момент окончания приема можно с

помощью подпрограмм MPI_Wait или MPI_Test с соответствующим параметром request.

Как и в блокирующих операциях, часто возникает необходимость опроса параметров полученного сообщения без его фактического чтения. Это делается с помощью функции MPI_Iprobe.

Неблокирующая функция чтения параметров полученного сообщения MPI_Iprobe.

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

Входные аргументы:

source - номер процесса-отправителя;

tag - идентификатор сообщения;

comm - коммуникатор.

Выходные аргументы:

flag - признак завершения операции;

status - атрибуты опрошенного сообщения.

Если flag=true, то операция завершилась, и в переменной status находятся атрибуты этого сообщения.

Воспользоваться результатом неблокирующей коммуникационной операции или повторно использовать ее аргументы можно только после ее полного завершения.

Имеется два типа функций завершения неблокирующих операций:

- операции ожидания завершения семейства Wait блокируют работу процесса до полного завершения операции;
- операции проверки завершения семейства Test возвращают значения TRUE или FALSE в зависимости от того, завершилась операция или нет.

Функции семейства MPI_Test не блокируют работу процесса и полезны для предварительного определения факта завершения операции.

Функция ожидания завершения неблокирующей операции MPI_Wait.

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

Входные аргументы:

request - запрос связи.

Выходные аргументы:

request - запрос связи;

status - атрибуты сообщения.

Это нелокальная блокирующая операция. Возврат происходит после завершения операции, связанной с запросом request. В параметре status возвращается информация о законченной операции.

Функция проверки завершения неблокирующей операции MPI_Test.

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

Входные аргументы:

request - запрос связи.

Выходные аргументы:

request - запрос связи;

flag - признак завершенности проверяемой операции;

status - атрибуты сообщения, если операция завершилась.

Это локальная неблокирующая операция. Если связанная с запросом request операция завершена, возвращается flag = true, а status содержит информацию о завершенной операции. Если проверяемая операция не завершена, возвращается flag = false, а значение status в этом случае не определено.

Функция снятия запроса без ожидания завершения неблокирующей операции MPI_Request_free.

```
int MPI_Request_free(MPI_Request *request)
```

Входные аргументы:

request - запрос связи.

Выходные аргументы:

request - запрос связи.

Параметр request устанавливается в значение MPI_REQUEST_NULL. Связанная с этим запросом операция не прерывается, однако проверить ее завершение с помощью MPI_Wait или MPI_Test уже нельзя.

Пример 1.

Рассмотрим пример, демонстрирующий использование неблокирующего приема сообщения. Для этого напишем программу (рис. 51), реализующую следующий алгоритм:

- нулевой процессор выполняет функцию sleep(1) (“спит” 1 секунду) и после этого посылает первому процессору значение переменной a при помощи коммуникационной функции MPI_Send;
- первый процессор засекает время t1, выполняет блокирующую функцию MPI_Recv для получения присланного значения, засекает время t2, выводит присланное значение и затраченное время на ожидание и прием посылки.

0-й процесс	1-й процесс
a=5	a=0
Замер времени	Замер времени

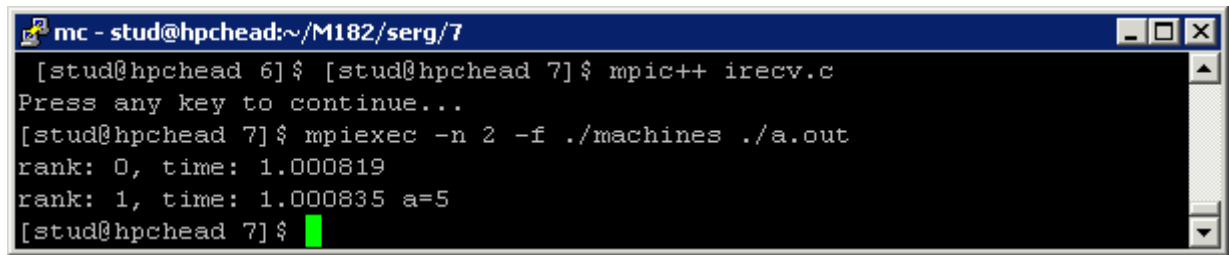
sleep(1) – “спит” 1 секунду	MPI_Recv(&a... - принимает блокирующей функцией сообщение
MPI_Send(&a... отправляет на 1-й процесс сообщение	Замер времени
Замер времени	
Вывод номера процесса и времени между замерами	Вывод номера процесса, времени между замерами, значения переменной a

Так как используются блокирующие функции, то 0-й процесс ожидаемо выведет время, равное 1 секунде (время сна) + добавочное время на отправку сообщения, 1-й же процесс, хотя и не ”спал”, но все равно выведет время, равное 1 секунде + добавочное время на пересылку и получение сообщения (рис. 52).

```

1.  #include <stdio.h>
2.  #include "mpi.h"
3.  #include "iostream"
4.
5.  int main(int argc, char *argv[])
6.  {
7.      int rank;
8.      int size;
9.      int a;
10.     double t1, t2;
11.     MPI_Status stat;
12.     MPI_Init(&argc, &argv);
13.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14.     MPI_Comm_size(MPI_COMM_WORLD, &size);
15.
16.     if(rank==0)
17.     {
18.         a = 5;
19.         t1 = MPI_Wtime();
20.         sleep(1);
21.         MPI_Send(&a, 1, MPI_INT, 1, 777, MPI_COMM_WORLD);
22.         t2 = MPI_Wtime();
23.         printf("rank: %d, time: %f\n", rank, t2 - t1);
24.
25.     }
26.     if(rank==1)
27.     {
28.         t1 = MPI_Wtime();
29.         MPI_Recv(&a, 1, MPI_INT, 0, 777, MPI_COMM_WORLD, &stat);
30.         t2 = MPI_Wtime();
31.         printf("rank: %d, time: %f a=%d\n",rank,t2-t1,a);
32.     }
33.     MPI_Finalize();
34.     return 0;
35. }
```

Рис. 51. Листинг программы передачи данных



```
mc - stud@hpchead:~/M182/serg/7
[stud@hpchead 6]$ [stud@hpchead 7]$ mpic++ irecv.c
Press any key to continue...
[stud@hpchead 7]$ mpiexec -n 2 -f ./machines ./a.out
rank: 0, time: 1.000819
rank: 1, time: 1.000835 a=5
[stud@hpchead 7]$
```

Рис. 52. Скрин запуска программы на 2-х процессах

```
1. #include <stdio.h>
2. #include "mpi.h"
3. #include "iostream"
4.
5. int main(int argc, char *argv[])
6. {
7.     int rank;
8.     int size;
9.     int a;
10.    double t1, t2;
11.    MPI_Status stat;
12.    MPI_Request req;
13.    MPI_Init(&argc, &argv);
14.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15.    MPI_Comm_size(MPI_COMM_WORLD, &size);
16.
17.    if(rank==0)
18.    {
19.        a = 5;
20.        t1 = MPI_Wtime();
21.        sleep(1);
22.        MPI_Send(&a, 1, MPI_INT, 1, 777, MPI_COMM_WORLD);
23.        t2 = MPI_Wtime();
24.        printf("rank: %d, time: %f\n", rank, t2 - t1);
25.
26.    }
27.    if(rank==1)
28.    {
29.        t1 = MPI_Wtime();
30.        MPI_Irecv(&a, 1, MPI_INT, 0, 777, MPI_COMM_WORLD, &req);
31.        t2 = MPI_Wtime();
32.        printf("rank: %d, time: %f a=%d\n", rank, t2-t1, a);
33.    }
34.    MPI_Finalize();
35.    return 0;
36. }
```

Рис. 53. Листинг программы передачи данных

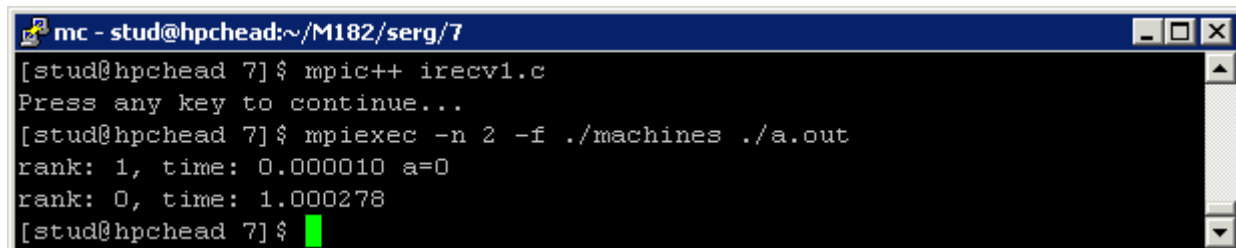
Пример 2.

Модифицируем программу – при приеме будем использовать неблокирующую функцию `MPI_Irecv` (рис. 53-54):

- нулевой процессор выполняет функцию `sleep(1)` (“спит” 1 секунду) и после этого посылает первому процессору значение переменной `a` при помощи коммуникационной функции `MPI_Send`;

- первый процессор засекает время t_1 , выполняет неблокирующую функцию `MPI_Irecv` для получения присланного значения, засекает время t_2 , выводит полученное значение и затраченное время на выполнение неблокирующей операции приема.

Каков результат и почему?



```
mc - stud@hpchead:~/M182/serg/7
[stud@hpchead 7]$ mpic++ irecv1.c
Press any key to continue...
[stud@hpchead 7]$ mpiexec -n 2 -f ./machines ./a.out
rank: 1, time: 0.000010 a=0
rank: 0, time: 1.000278
[stud@hpchead 7]$
```

Рис. 54. Скрин запуска программы

Что мы видим? 1-й процесс на вызов функции `MPI_Irecv` практически не затратил времени, но и значение переменной не получил. Нужно запомнить – функция `MPI_Irecv` не получает сообщение, а только специфицирует куда будет сохранена посылка после ее получения.

Получить же можно сообщения двумя способами – используя функцию `MPI_Wait` или `MPI_Test`.

Пример 3.

Модифицируем программу – после вызова функции `MPI_Irecv` добавим функцию ожидания сообщения `MPI_Wait` (рис. 55-56):

- нулевой процессор выполняет функцию `sleep(1)` (“спит” 1 секунду) и после этого посылает первому процессору значение переменной `a` при помощи коммуникационной функции `MPI_Send`;
- первый процессор засекает время t_1 , выполняет неблокирующую функцию `MPI_Irecv` для получения присланного значения, засекает время t_2 , выполняет операцию `MPI_Wait`, засекает время t_3 , выводит полученное значение и затраченное время на выполнение неблокирующей операции приема и ожидание получения посылки.

Каков результат и почему?

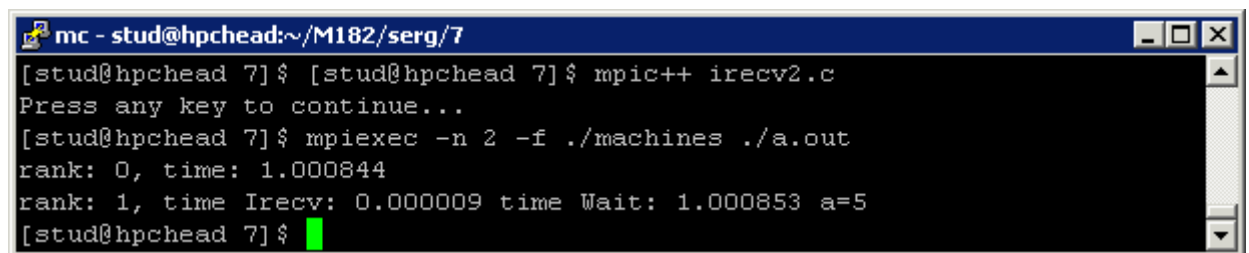
Аналогично предыдущему примеру, `MPI_Irecv` практически не затратил времени, а вот `MPI_Wait` ожидал сообщения секунду плюс пересылка и получение сообщения от 0-го процесса. Следует заметить, что идущие подряд вызовы функций `MPI_Irecv` и `MPI_Wait` эквивалентны вызову блокирующего приема сообщений `MPI_Recv`. В реальных программах между их вызовами располагается блок строк кода, производящих вычисления, для которых пересылаемая информация еще не нужна.

```

1. #include <stdio.h>
2. #include "mpi.h"
3. #include "iostream"
4.
5. int main(int argc, char *argv[])
6. {
7.     int rank;
8.     int size;
9.     int a;
10.    double t1, t2, t3;
11.    MPI_Status stat;
12.    MPI_Request req;
13.    MPI_Init(&argc, &argv);
14.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15.    MPI_Comm_size(MPI_COMM_WORLD, &size);
16.
17.    if(rank==0)
18.    {
19.        a = 5;
20.        t1 = MPI_Wtime();
21.        sleep(1);
22.        MPI_Send(&a, 1, MPI_INT, 1, 777, MPI_COMM_WORLD);
23.        t2 = MPI_Wtime();
24.        printf("rank: %d, time: %f\n", rank, t2 - t1);
25.    }
26. }
27. if(rank==1)
28. {
29.     t1 = MPI_Wtime();
30.     MPI_Irecv(&a, 1, MPI_INT, 0, 777, MPI_COMM_WORLD, &req);
31.     t2 = MPI_Wtime();
32.     MPI_Wait(&req, &stat);
33.     t3 = MPI_Wtime();
34.     printf("rank: %d, time Irecv: %f time Wait: %f a=%d\n", rank, t2-t1, t3-t2, a);
35. }
36. MPI_Finalize();
37. return 0;
38. }

```

Рис. 55. Листинг программы передачи данных



```

mc - stud@hpchead:~/M182/serg/7
[stud@hpchead 7]$ [stud@hpchead 7]$ mpic++ irecv2.c
Press any key to continue...
[stud@hpchead 7]$ mpiexec -n 2 -f ./machines ./a.out
rank: 0, time: 1.000844
rank: 1, time Irecv: 0.000009 time Wait: 1.000853 a=5
[stud@hpchead 7]$

```

Рис. 56. Скрин запуска программы

Пример 4.

Модифицируем программу – после вызова функции `MPI_Irecv` вместо `MPI_Wait` поставим цикл с проверкой получения сообщения `MPI_Test` (рис. 57-58), дополнительно определим сколько раз данный тест проводился:

- нулевой процессор выполняет функцию `sleep(1)` (“спит” 1 секунду) и после этого посылает первому процессору значение переменной `a` при помощи коммуникационной функции `MPI_Send`;
- первый процессор засекает время `t1`, выполняет неблокирующую функцию `MPI_Irecv` для получения присланного значения, засекает время `t2`, запускает условный цикл (условие – сообщение получено, проверка с помощью функции `MPI_Test`) со счетчиком итераций, после выхода из цикла, засекает время `t3`, выводит полученное значение, количество итераций для теста приема значения и затраченное время на выполнение неблокирующей операции приема и ожидание получения посылки.

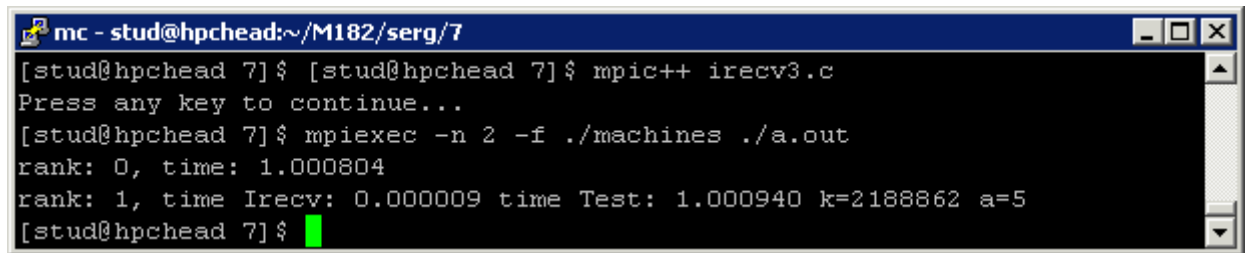
Каков результат и почему?

```

1. #include <stdio.h>
2. #include "mpi.h"
3. #include "iostream"
4. int main(int argc, char *argv[])
5. {
6.     int rank;
7.     int size;
8.     int a,k=0,fl=0;
9.     double t1, t2,t3;
10.    MPI_Status stat;
11.    MPI_Request req;
12.    MPI_Init(&argc, &argv);
13.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14.    MPI_Comm_size(MPI_COMM_WORLD, &size);
15.
16.    if(rank==0)
17.    {
18.        a = 5;
19.        t1 = MPI_Wtime();
20.        sleep(1);
21.        MPI_Send(&a, 1, MPI_INT, 1, 777, MPI_COMM_WORLD);
22.        t2 = MPI_Wtime();
23.        printf("rank: %d, time: %f\n", rank, t2 - t1);
24.    }
25.    if(rank==1)
26.    {
27.        t1 = MPI_Wtime();
28.        MPI_Irecv(&a, 1, MPI_INT, 0, 777, MPI_COMM_WORLD, &req);
29.        t2 = MPI_Wtime();
30.        while(fl==0)
31.        {
32.            k++;
33.            MPI_Test(&req,&fl,&stat);
34.        }
35.        t3 = MPI_Wtime();
36.        printf("rank: %d, time Irecv: %f time Test: %f k=%d a=%d\n",rank,t2-t1,t3-t2,k,a);
37.    }
38.    MPI_Finalize();
39.    return 0;
40. }
```

Рис. 57. Листинг программы передачи данных

Аналогично предыдущему примеру, MPI_Irecv практически не затратил времени, а вот функция MPI_Test за время, пока не было получено сообщение, была вызвана более 2 млн. раз! Данную функцию имеет смысл использовать, когда есть несколько порций вычислений, которые можно проводить без ожидаемой информации, а необходимость возникает после очередной порции вычислений осуществить проверку доступности ожидаемого сообщения.



```
mc - stud@hpchead:~/M182/serg/7
[stud@hpchead 7]$ [stud@hpchead 7]$ mpic++ irecv3.c
Press any key to continue...
[stud@hpchead 7]$ mpiexec -n 2 -f ./machines ./a.out
rank: 0, time: 1.000804
rank: 1, time Irecv: 0.000009 time Test: 1.000940 k=2188862 a=5
[stud@hpchead 7]$
```

Рис. 58. Скрин запуска программы

7.2. Одновременная передача данных

При использовании блокирующего режима передачи сообщений существует потенциальная опасность возникновения тупиковых ситуаций, в которых операции обмена данными блокируют друг друга, то есть когда процессы переходят в режим взаимного ожидания.

В ситуациях, когда требуется выполнить *взаимный обмен данными между процессами*, безопаснее использовать *совмещенную операцию MPI_Sendrecv*.

```
int MPI_Sendrecv(void *sendbuf, int sendcount,
MPI_Datatype sendtype, int dest, int sendtag, void
*recvbuf, int recvcount, MPI_Datatype recvtype, int
source, MPI_Datatype recvtag, MPI_Comm comm,
MPI_Status *status)
```

Входные аргументы:

sendbuf	- адрес начала расположения посылаемого сообщения;
sendcount	- число посылаемых элементов;
sendtype	- тип посылаемых элементов;
dest	- номер процесса-получателя;
sendtag	- идентификатор посылаемого сообщения;
recvcount	- максимальное число принимаемых элементов;
recvtype	- тип элементов принимаемого сообщения;
source	- номер процесса-отправителя;
recvtag	- идентификатор принимаемого сообщения;
comm	- коммуникатор области связи.

Выходные аргументы:

recvbuf - адрес начала расположения принимаемого сообщения;
status - атрибуты принятого сообщения.

Пример 5.

В качестве примера использования изучаемой функции рассмотрим задачу обмена информацией между двумя процессами:

0-й процесс	1-й процесс
a=5	b=7
Отправка a на 1-й процесс, прием от 1-го процесса значения переменной b	Отправка b на 0-й процесс, прием от 0-го процесса значения переменной a
Вывод номера процесса и значений переменных	

Программа, реализующая обмен данными совмещенной операцией MPI_Sendrecv, приведена на рис. 59, скрин ее запуска – на рис. 60. Видно, что в результате оба процесса вывели одинаковые значения обеих переменных.

```

1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int a=0,b=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    if(rank==0)
13.    {
14.        a=5;
15.        MPI_Sendrecv(&a,1,MPI_INT,1,777,&b,1,MPI_INT,1,777,MPI_COMM_WORLD,&stat);
16.    }
17.    if(rank==1)
18.    {
19.        b=7;
20.        MPI_Sendrecv(&b,1,MPI_INT,0,777,&a,1,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
21.    }
22.    printf("rank= %d, a= %d, b=%d \n ", rank,a,b);
23.    MPI_Finalize();
24.    return 0;
25. }
```

Рис. 59. Листинг программы передачи данных

```

mc - stud@hpchead:~/M182/serg/7
Press any key to continue...
[stud@hpchead 7]$ mpiexec -n 2 -f ./machines ./a.out
rank= 0, a= 5, b=7
rank= 1, a= 5, b=7
[stud@hpchead 7]$
```

Рис. 60. Скрин запуска программы

Функция `MPI_Sendrecv` совмещает выполнение операций передачи и приема. Обе операции используют один и тот же коммуникатор, но идентификаторы сообщений могут различаться. Расположение в адресном пространстве процесса принимаемых и передаваемых данных не должно пересекаться. Пересылаемые данные могут быть различного типа и иметь разную длину. В тех случаях, когда необходим обмен данными одного типа с замещением посылаемых данных на принимаемые, удобнее пользоваться функцией `MPI_Sendrecv_replace`.

```
MPI_Sendrecv_replace(void* buf, int count,  
MPI_Datatype datatype, int dest, int sendtag, int  
source,  
int recvtag, MPI_Comm comm, MPI_Status *status)
```

Входные аргументы:

`buf` - адрес начала расположения посылаемого и принимаемого сообщения;
`count` - число передаваемых элементов;
`datatype` - тип передаваемых элементов;
`dest` - номер процесса-получателя;
`sendtag` - идентификатор посылаемого сообщения;
`source` - номер процесса-отправителя;
`recvtag` - идентификатор принимаемого сообщения;
`comm` - коммуникатор области связи.

Выходные аргументы:

`buf` - адрес начала расположения посылаемого и принимаемого сообщения;
`status` - атрибуты принятого сообщения.

В данной операции посылаемые данные из массива `buf` замещаются принимаемыми данными. В качестве адресатов `source` и `dest` в операциях пересылки данных можно использовать специальный адрес `MPI_PROC_NULL`. Коммуникационные операции с таким адресом ничего не делают. Применение этого адреса бывает удобным вместо использования логических конструкций для анализа условий посылать/читать сообщение или нет.

Пример 6.

В качестве примера использования изучаемой функции рассмотрим задачу обмена информацией между двумя процессами:

0-й процесс	1-й процесс
a=5	b=7
Отправка a на 1-й процесс, прием от 1-го процесса значения его переменной b с присваиванием в	Отправка b на 0-й процесс, прием от 0-го процесса значения переменной a с присваиванием в переменную b

переменную a	
Вывод номера процесса и значений переменных	

Программа, реализующая обмен данными совмещенной операцией `MPI_Sendrecv_replace`, приведена на рис. 61, скрин ее запуска – на рис. 62. Видно, что в результате оба процесса вывели значения, полученные в результате обмена.

```
1. #include <stdio.h>
2. #include "mpi.h"
3.
4. int main(int argc, char *argv[])
5. {
6.     int rank;
7.     int size;
8.     int a=0,b=0;
9.     double t1,t2,t3;
10.
11.     MPI_Status stat;
12.     MPI_Request req;
13.     MPI_Init(&argc, &argv);
14.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
15.     MPI_Comm_size(MPI_COMM_WORLD, &size);
16.
17.     if(rank==0)
18.     {
19.         a=5;
20.         MPI_Sendrecv_replace(&a,1,MPI_INT,1,777,1,777,MPI_COMM_WORLD,&stat);
21.         printf("rank= %d, a= %d \n ", rank,a);
22.     }
23.
24.     if(rank==1)
25.     {
26.         b=7;
27.         MPI_Sendrecv_replace(&b,1,MPI_INT,0,777,0,777,MPI_COMM_WORLD,&stat);
28.         printf("rank= %d, b= %d \n ", rank,b);
29.     }
30.
31.
32.     MPI_Finalize();
33.     return 0;
34. }
```

Рис. 61. Листинг программы передачи данных

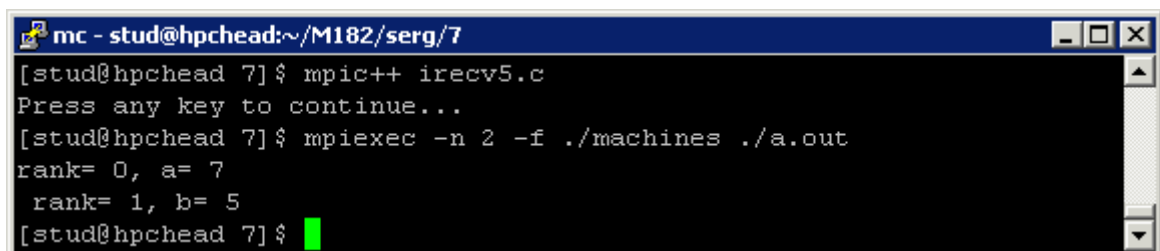


Рис. 62. Скрин запуска программы

Задания

1. Используя неблокирующие функции и прием опережающего вычисления и отправки сообщений, реализуйте следующий алгоритм:
 - на 0-м и 1-м процессах инициализируется двумерный массив A и вектор B при помощи датчика случайных чисел;
 - на каждом процессе матрица A умножается на вектор B , результат сохраняется в первую строку матрицы C ;
 - на каждом процессе первая строка матрицы A покомпонентно умножается на вектор B , полученный в результате вектор сохраняется в B ;
 - процессы обмениваются векторами B ;
 - на каждом процессе матрица A умножается на вектор B , результат сохраняется во вторую строку матрицы C ;
 - на каждом процессе вторая строка матрицы A покомпонентно умножается на вектор B , полученный в результате вектор сохраняется в B ;
 - процессы обмениваются векторами B , и т.д.

Проведите анализ времени выполнения алгоритма с использованием обычных блокирующих и неблокирующих функций в зависимости от размера матрицы (возьмите от 1000 до 10000 с шагом 1000).

2. Создайте и выполните программу, используя коммуникационную функцию (`MPI_Sendrecv`), реализующую алгоритм передачи данных по кольцу.
3. Создайте и выполните программу, используя коммуникационную функцию (`MPI_Sendrecv`), реализующую алгоритм передачи данных по двум кольцам: нечетные процессора образуют первое кольцо, четные – второе.
4. Напишите программу, которая будет выполняться на двух процессах и реализовывать следующий алгоритм:
 - На 0-м процессе создается динамический массив $a[n]$, на 1-м – $b[n]$;
 - На обоих процессах засекается время;
 - Осуществляется обмен массивами;
 - На обоих процессах замеряется время и производится вывод на экран номера процесса и затраченное время на передачу данных.

Реализуйте два варианта данной программы: 1) для передачи данных используйте стандартные функции `MPI_Send`, `MPI_Recv`; 2) для передачи данных используйте совмещенную функцию `MPI_Sendrecv`. Выполните сравнение времени передачи данных в зависимости от способа передачи и различных размеров передаваемых массивов.

8. Коллективные операции: синхронизация процессов, широковещательная рассылка

8.1. Обзор коллективных операций

Набор операций типа “точка-точка” является достаточным для программирования любых алгоритмов, однако MPI вряд ли бы завоевал такую популярность, если бы ограничивался только этим набором коммуникационных операций. Одной из наиболее привлекательных сторон MPI является наличие широкого набора коллективных операций, которые берут на себя выполнение наиболее часто встречающихся при программировании действий. Например, часто возникает потребность разослать некоторую переменную или массив из одного процесса всем остальным. Каждый программист может написать такую процедуру с использованием операций Send/Recv, однако гораздо удобнее воспользоваться коллективной операцией MPI_Bcast. Причем гарантировано, что эта операция будет выполняться гораздо эффективнее, поскольку MPI-функция реализована с использованием внутренних возможностей коммуникационной среды. Главное отличие коллективных операций от операций типа “точка-точка” состоит в том, что в них всегда участвуют все процессы, связанные с некоторым коммуникатором. Несоблюдение этого правила приводит либо к аварийному завершению задачи, либо к еще более неприятному зависанию задачи.

Набор коллективных операций включает:

- синхронизацию всех процессов с помощью барьеров (MPI_Barrier);
- коллективные коммуникационные операции;
- глобальные вычислительные операции.

Коллективные коммуникационные операции реализуют следующее:

- рассылку информации от одного процесса всем остальным членам некоторой области связи (MPI_Bcast);
- сборку (gather) распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (root) процесса (MPI_Gather, MPI_Gatherv);
- сборку (gather) распределенного массива в один массив с рассылкой его всем процессам некоторой области связи (MPI_Allgather, MPI_Allgatherv);
- разбиение массива и рассылку его фрагментов (scatter) всем процессам области связи (MPI_Scatter, MPI_Scatterv);
- совмещенную операцию Scatter/Gather (All-to-All), каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами в свой буфер приема (MPI_Alltoall, MPI_Alltoallv).

Глобальные вычислительные операции (sum, min, max и др.) реализуются над данными, расположенными в адресных пространствах различных процессов:

- с сохранением результата в адресном пространстве одного процесса (MPI_Reduce);
- с рассылкой результата всем процессам (MPI_Allreduce);
- совмещенная операция Reduce/Scatter (MPI_Reduce_scatter);
- префиксная редукция (MPI_Scan).

Все коммуникационные подпрограммы, за исключением MPI_Bcast, представлены в двух вариантах:

- простой вариант, когда все части передаваемого сообщения имеют одинаковую длину и занимают смежные области в адресном пространстве процессов;
- "векторный" вариант предоставляет более широкие возможности по организации коллективных коммуникаций, снимая ограничения, присущие простому варианту как в части длин блоков, так и в части размещения данных в адресном пространстве процессов. Векторные варианты обозначаются символом "v" в конце имени функции.

Отличительные особенности коллективных операций:

- коллективные коммуникации не взаимодействуют с коммуникациями типа "точка-точка";
- коллективные коммуникации выполняются в режиме с блокировкой. Возврат из подпрограммы в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы завершили операцию;
- количество получаемых данных должно быть равно количеству посланных данных;
- типы элементов посылаемых и получаемых сообщений должны совпадать;
- сообщения не имеют идентификаторов.

8.2. Принцип работы коллективных коммуникационных операций

Разберем принцип выполнения рассылки на следующем примере: допустим, стоит задача рассылки значения переменной от 0-го процесса всем остальным. Подобный пример нами был уже ранее разобран при изучении функций MPI_Send-MPI_Recv. Алгоритм был следующим: 0-й процесс в цикле шлет поочередно всем ненулевым процессам, а ненулевые, соответственно, получают данное сообщение от 0-го процесса (схема рассылки приведена на рис. 63, а). Данный подход к рассылке требует (size-1) итерацию цикла для отправки сообщений с 0-го процесса и в итоге затратится времени (size-1) * (время 1-й пересылки). Давайте задумаемся, как выполнить данную рассылку за меньшее время. На первой итерации 0-й

процесс отправляет сообщение 1-му процессу (рис. 63, б), на второй итерации уже два процесса могут осуществлять отправку элементов: 0-й процесс – 2-му процессу, а 1-й процесс – 3-у! На каждой итерации количество одновременных передач возрастает вдвое!

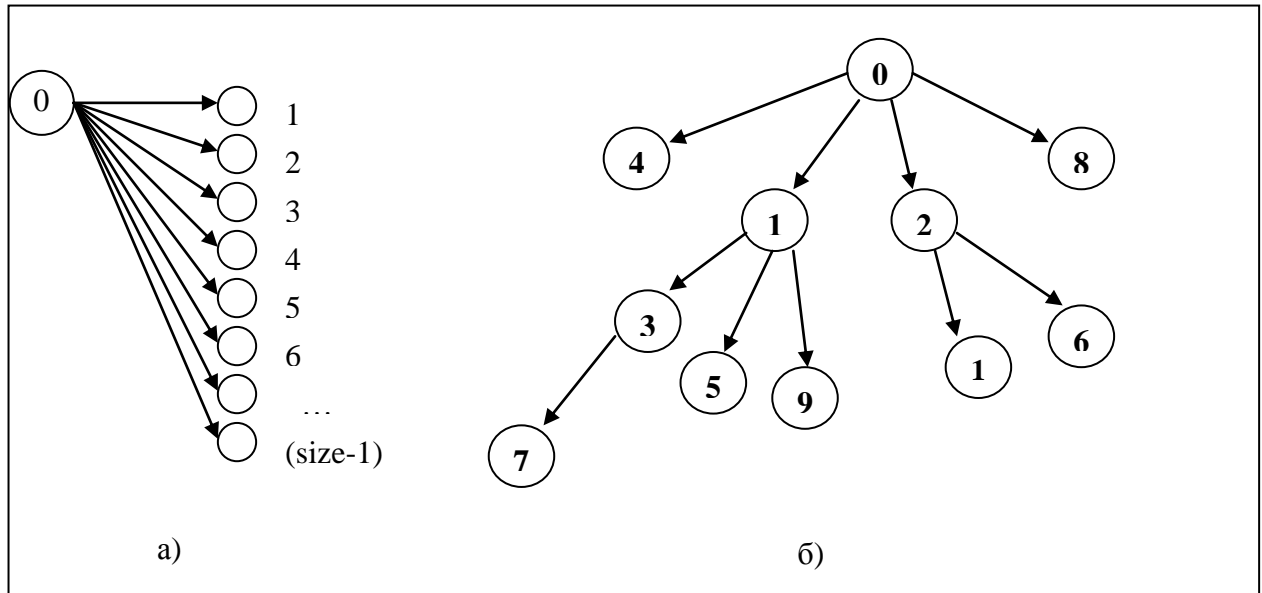


Рис. 63. Схемы рассылок сообщения от 0-го процесса все остальным

Первым заданием к данному пункту Вам будет предложено реализовать собственную функцию широковещательной рассылки, использующей только что рассмотренный алгоритм.

Для облегчения выполнения задания приведем ряд “подсказок”.

Первый вопрос: сколько требуется итераций для рассылки информации по данному алгоритму? Ответ: так как на каждой итерации количество отправляющих процессов удваивается, то потребуется $\log_2(size)$ с округлением в большую сторону итераций. Например, если процессов 8, то потребуется всего 3 итерации:

$0 \rightarrow 1$;

$0 \rightarrow 2, 1 \rightarrow 3$;

$0 \rightarrow 4, 1 \rightarrow 5, 2 \rightarrow 6, 3 \rightarrow 7$.

Однако, если процессов больше 8, но меньше 16, то итераций уже потребуется 4.

Второй вопрос: какие процессы производят отправку на очередной итерации? Ответ: те процессы, номер которых удовлетворяет условию $rank < 2^{it}$, где it – номер итерации, изменяющийся от 0 и до требуемого значения. Например, на 0-й итерации отправляет только 0 процесс: $0 < 2^0$, на второй итерации – 0 и 1-й процессы: $0 < 2^1, 1 < 2^1$, т.д.

Третий вопрос: кому шлют отправляющие процессы? Ответ: номер процесса-адресата можно определить по формуле: $2^{it} + rank$. Действительно,

на 0-й итерации 0-й процесс шлет ($2^0 + 0 = 1$) первому процессу; на первой итерации 0-й процесс шлет ($2^1 + 0 = 2$) второму процессу, а 1-й - ($2^1 + 1 = 3$) третьему процессу, и т.д.

Четвертый вопрос: сколько раз каждый некорневой процесс получает сообщение и от кого? Ответ: один раз и совсем нет никакой разницы от кого сообщение будет получено – при приеме надо использовать предопределенную переменную `MPI_ANY_SOURCE`.

Пятая подсказка: библиотека `mpich` – это библиотека с открытым исходным кодом!

Все коллективные операции используют аналогичную схему коммуникаций (хоть для рассылки, хоть для сборки информации), используя во внутренней реализации уже известные нам функции точечного обмена `MPI_Send-MPI_Recv`.

8.3. Синхронизация процессов

Функция синхронизации процессов `MPI_Barrier` блокирует работу вызвавшего ее процесса до тех пор, пока все другие процессы группы также не вызовут эту функцию. Завершение работы этой функции возможно только всеми процессами одновременно (все процессы "преодолевают барьер" одновременно).

`int MPI_Barrier(MPI_Comm comm)`

Входные аргументы:

`comm` - коммуникатор.

Синхронизация с помощью барьеров используется, например, для завершения всеми процессами некоторого этапа решения задачи, результаты которого будут использоваться на следующем этапе. Использование барьера гарантирует, что ни один из процессов не приступит раньше времени к выполнению следующего этапа, пока результат работы предыдущего не будет окончательно сформирован. Неявную синхронизацию процессов выполняет любая коллективная функция.

Следует понимать, что, хотя данная функция никакой полезной информации не передает, обмен служебной информацией для определения синхронизации процессов присутствует. Это требует время и негативно сказывается на эффективности распараллеливания алгоритма в целом. На практике данную функцию следует использовать только лишь в исключительных случаях, так как было уже сказано, что неявно синхронизацию процессов выполняет любая коллективная функция. Где имеет смысл производить синхронизацию процессов?

Первый пример: требуется определить время выполнения параллельной программы. После инициализации процессов имеет смысл провести их синхронизацию, так как процессы запускаются асинхронно и замер времени без синхронизации будет неточен. Аналогия – синхронизация спортсменов на старте.

Второй пример: вывод на экран отладочной информации каждым процессом в конце программы. Перед вызовом функции `MPI_Finalize` имеет смысл выполнить синхронизацию, чтобы все процессы успели произвести вывод, иначе может появиться сообщение об ошибке "socket error ...", так как устройство ввода-вывода доступно только на 0-й консоле (процессе), а, если 0-й процесс раньше других завершает свою работу, то остальные процессы остаются без устройства вывода.

8.4. Широковещательная рассылка

Широковещательная рассылка данных выполняется с помощью функции `MPI_Bcast`. Процесс с номером `root` рассылает сообщение из своего буфера передачи всем процессам области связи коммуникатора `comm`.

`int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`

Входные аргументы:

`buffer` - адрес начала расположения в памяти рассылаемых данных;
`count` - число посылаемых элементов;
`datatype` - тип посылаемых элементов;
`root` - номер процесса-отправителя;
`comm` - коммуникатор.

Выходные аргументы:

`buffer` - адрес начала расположения в памяти рассылаемых данных.

После завершения подпрограммы каждый процесс в области связи коммуникатора `comm`, включая и самого отправителя, получит копию сообщения от процесса-отправителя `root`.

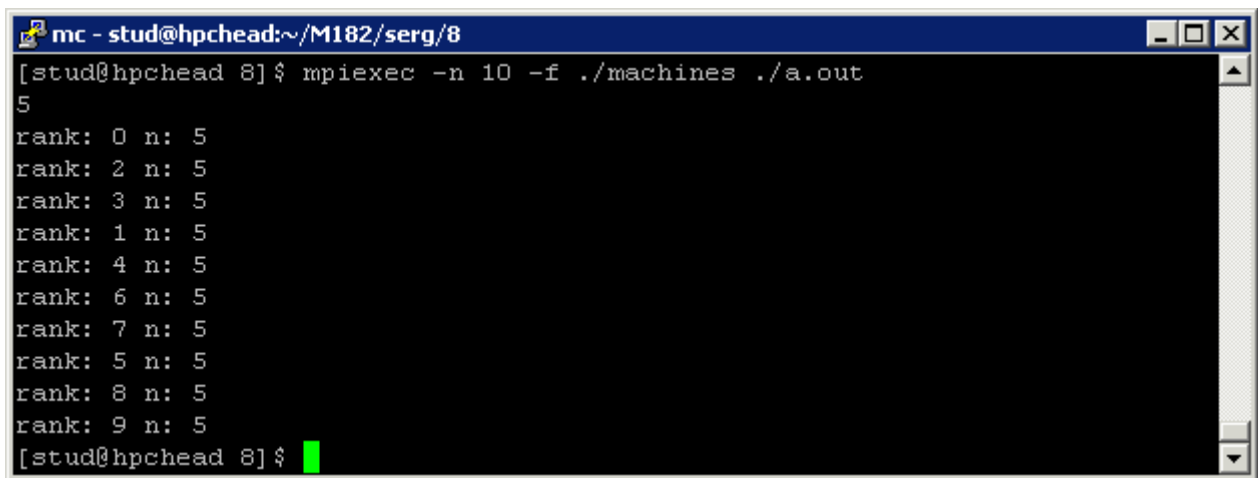
```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    if(rank==0) scanf("%d", &n);
12.    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
13.    printf("rank: %d n: %d\n", rank, n);
14.    MPI_Finalize();
15.    return 0;
16. }
```

Рис. 64. Листинг программы широковещательной рассылки данных

Пример 1.

В качестве примера использования изучаемой функции рассмотрим задачу рассылки значения переменной с 0-го процесса всем остальным.

Листинг программы приведен на рис. 64, скрин ее запуска – на рис. 65. В строке 11 происходит считывание значения переменной со стандартного устройства вывода (не забываем, что ввод доступен только на 0-м процессе). В 12 строке произведен вызов функции MPI_Bcast всеми процессами в области связи MPI_COMM_WORLD. Ранее данный алгоритм был реализован функциями парного обмена MPI_Send-MPI_Recv, следует отметить, что использование коллективной операции позволило существенно сократить количество строк кода, а также сделало программу более ”читабельной”.



```
mc - stud@hpchead:~/M182/serg/8
[stud@hpchead 8]$ mpirun -n 10 -f ./machines ./a.out
5
rank: 0 n: 5
rank: 2 n: 5
rank: 3 n: 5
rank: 1 n: 5
rank: 4 n: 5
rank: 6 n: 5
rank: 7 n: 5
rank: 5 n: 5
rank: 8 n: 5
rank: 9 n: 5
[stud@hpchead 8]$
```

Рис. 65. Скрин запуска программы

Пример 2.

В данном примере напомним программу, реализующую следующий алгоритм:

0-процесс	Все процессы, включая 0-й
Считать n	
	Вызвать функцию MPI_Bcast для рассылки n с 0-го процесса всем
	Выделить место в памяти под массив a[n]
Присвоить элементам массива a[i]=i	
	Вызвать функцию MPI_Bcast для рассылки массива с 0-го процесса всем
	Вывести номер процесса и массив

Листинг программы приведен на рис.66, скрин ее запуска – на рис.67.


```

1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=0,i,s=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    if(rank==0) scanf("%d",&n);
13.    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
14.    int *a=new int[n];
15.    if(rank==0)
16.        for(i=0;i<n;i++)
17.            a[i]=i;
18.
19.    MPI_Bcast(a,n,MPI_INT,0,MPI_COMM_WORLD);
20.    for(i=0; i<100000000*rank; i++)
21.        s+=1;
22.    printf("rank= %d a: ",rank);
23.    for(i=0; i<n; i++)
24.        printf(" %d ",a[i]);
25.    printf("\n");
26.    MPI_Finalize();
27.    return 0;
28. }

```

Рис. 66. Листинг программы рассылки переменной и массива

```

mc - stud@hpchead:~/M182/serg/8
[stud@hpchead 8]$ mpirun -n 5 -f ./machines ./a.out
10
rank= 0 a: 0 1 2 3 4 5 6 7 8 9
rank= 1 a: 0 1 2 3 4 5 6 7 8 9
rank= 2 a: 0 1 2 3 4 5 6 7 8 9
rank= 3 a: 0 1 2 3 4 5 6 7 8 9
rank= 4 a: 0 1 2 3 4 5 6 7 8 9
[stud@hpchead 8] $

```

Рис. 67. Скрин запуска программы

Задания

1. Создайте собственную функцию широковещательной рассылки (аналог `MPI_Bcast`), но построенную с использованием блокирующих функций парного обмена сообщениями (`MPI_Send`, `MPI_Recv`). Рассылку сообщения реализовать по рассмотренному ранее алгоритму: сначала 0-й процесс отправляет сообщение 1-му, затем 0-й и 1-й осуществляют отправки сообщений 2-му и 3-му, и т.д. Написать универсальную функцию (для любого количества процессов, для любого источника рассылки, для любых типов данных и любого количества пересылаемых элементов). В момент запуска параллельного приложения соберите трассу выполнения программы и приведите рисунок трассы в отчете. Проведите тестирование

созданной функции с демонстрацией всех указанных вариантов. Проведите сравнение времени выполнения созданной функции с функцией `MPI_Bcast`.

2. Напишите программу, которая реализует следующий алгоритм: на 0-м процессе считывается значение переменной `n` и рассылается каждому; на каждом процессе выделяется место в памяти под одномерный массив `a[n]`; на 2-м процессе данный массив заполняется с помощью датчика случайных чисел и рассылается каждому; для проверки корректности работы программы распечатайте изначальный массив на 2-м процессе и полученные остальными процессами массивы.
3. Напишите программу, которая реализует следующий алгоритм: на 0-м процессе считывается значение переменной `n` и рассылается каждому; на каждом процессе выделяется место в памяти под двумерный массив `a[n][n]`; на 0-м процессе данный массив заполняется по формуле $a[i][j] = 10 * (i + 1) + (j + 1)$; далее 2-я строка рассылается каждому процессу; для проверки корректности работы программы распечатайте изначальный массив на 0-м процессе и полученные остальными процессами массивы.

9. Коллективные операции: функции сбора блоков данных от всех процессов

Семейство функций сбора блоков данных от всех процессов группы состоит из четырех подпрограмм: `MPI_Gather`, `MPI_Allgather`, `MPI_Gatherv`, `MPI_Allgatherv`. Каждая из указанных функций расширяет функциональные возможности предыдущих.

9.1. Функции сбора одинаковых блоков данных от всех процессов

Функция `MPI_Gather` производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером `root`.

Графическая интерпретация операции `Gather` представлена на рис. 68.

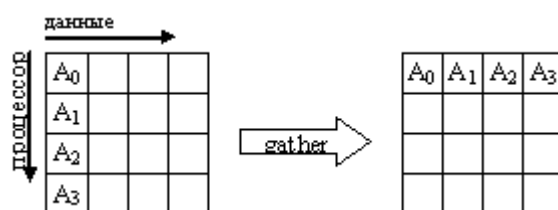


Рис. 68. Графическая интерпретация операции `Gather`

Длина блоков предполагается одинаковой. Объединение происходит в порядке увеличения номеров процессов-отправителей. То есть данные, посланные процессом i из своего буфера `sendbuf`, помещаются в i -ю порцию буфера `recvbuf` процесса `root`. Длина массива, в который собираются данные, должна быть достаточной для их размещения.

```
int MPI_Gather(void* sendbuf, int sendcount,  
MPI_Datatype sendtype, void* recvbuf, int recvcount,  
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Входные аргументы:

- `sendbuf` - адрес начала размещения посылаемых данных;
- `sendcount` - число посылаемых элементов;
- `sendtype` - тип посылаемых элементов;
- `recvcount` - число элементов, получаемых от каждого процесса (используется только в процессе-получателе `root`);
- `recvtype` - тип получаемых элементов;
- `root` - номер процесса-получателя;
- `comm` - коммуникатор.

Выходные аргументы:

- `recvbuf` - адрес начала буфера приема (используется только в процессе-получателе `root`).

Тип посылаемых элементов `sendtype` должен совпадать с типом `recvtype` получаемых элементов, а число `sendcount` должно равняться числу `recvcount`. То есть `recvcount` в вызове из процесса `root` – это число собираемых от каждого процесса элементов, а не общее количество собранных элементов.

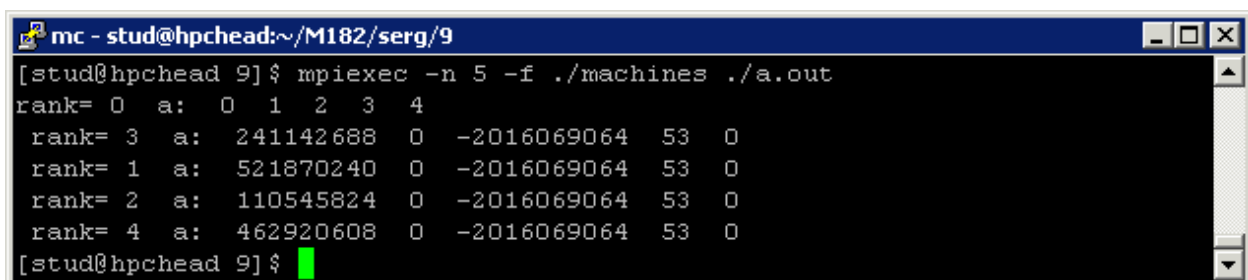
Пример 1.

В качестве примера использования изучаемой функции рассмотрим задачу сборки значения переменной с каждого процесса в одномерный массив на 0-м процессе.

Листинг программы приведен на рис. 69, скрин ее запуска – на рис. 70. На скрине можно заметить, что массив на 0-м процессе получен верным, а на ненулевых – вывод произвольных данных, расположенных в оперативной памяти. Следует отметить, что хотя данная функция производит сборку данных в массив на 0-м процессе, массив должен быть объявлен и на всех остальных процессах (он используется для сборки данных по ранее рассмотренному ”двоичному дереву”).

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=0,i,s=0;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    int *a=new int[size];
12.    MPI_Gather(&rank,1,MPI_INT,a,1,MPI_INT,0,MPI_COMM_WORLD);
13.    for(i=0; i<100000000*rank; i++)
14.        s+=1;
15.    printf("rank= %d a: ",rank);
16.    for(i=0; i<size; i++)
17.        printf(" %d ",a[i]);
18.    printf("\n ");
19.    MPI_Finalize();
20.    return 0;
21. }
```

Рис. 69. Листинг программы сборки данных



```
mc - stud@hpchead:~/M182/serg/9
[stud@hpchead 9]$ mpirun -n 5 -f ./machines ./a.out
rank= 0  a:  0  1  2  3  4
rank= 3  a: 241142688 0 -2016069064 53 0
rank= 1  a: 521870240 0 -2016069064 53 0
rank= 2  a: 110545824 0 -2016069064 53 0
rank= 4  a: 462920608 0 -2016069064 53 0
[stud@hpchead 9]$
```

Рис. 70. Скрин запуска программы

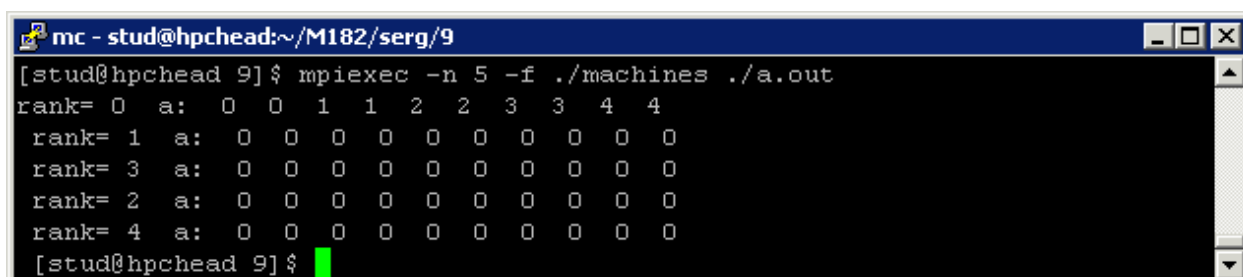
Пример 2.

Усложним задачу – теперь требуется объявить одномерный массив из двух элементов на каждом процессе и собрать его в подходящий по размеру массив на 0-м процессе.

Листинг программы приведен на рис. 71, скрин ее запуска – на рис. 72. На скрине можно заметить, что массив на 0-м процессе получен верным, а на ненулевых – вывод произвольных данных, расположенных в оперативной памяти. Для различения массивов, пересылаемых с кадного процесса, его элементам присвоили значение номера процесса (строка 13); для сборки по 2 элемента с каждого процесса на 0-й создали массив размером 2*size (строка 14), строки 16-17 уже ранее обсуждались – напомним, что они необходимы для разнесения по времени вывода данных с каждого процесса.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int i,s=0;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    int *b=new int [2];
12.    for(i=0;i<2;i++)
13.        b[i]=rank;
14.    int *a=new int[2*size];
15.    MPI_Gather(b,2,MPI_INT,a,2,MPI_INT,0,MPI_COMM_WORLD);
16.    for(i=0; i<100000000*rank; i++)
17.        s+=1;
18.    printf("rank= %d a: ",rank);
19.    for(i=0; i<2*size; i++)
20.        printf(" %d ",a[i]);
21.    printf("\n ");
22.    MPI_Finalize();
23.    return 0;
24. }
```

Рис. 71. Листинг программы



```
mc - stud@hpchead:~/M182/serg/9
[stud@hpchead 9]$ mpirun -n 5 -f ./machines ./a.out
rank= 0  a:  0  0  1  1  2  2  3  3  4  4
rank= 1  a:  0  0  0  0  0  0  0  0  0  0
rank= 3  a:  0  0  0  0  0  0  0  0  0  0
rank= 2  a:  0  0  0  0  0  0  0  0  0  0
rank= 4  a:  0  0  0  0  0  0  0  0  0  0
[stud@hpchead 9]$
```

Рис. 72. Скрин запуска программы

Функция MPI_Allgather выполняется так же, как MPI_Gather, но получателями являются все процессы группы. Данные, посланные процессом

i из своего буфера `sendbuf`, помещаются в *i*-ю порцию буфера `recvbuf` каждого процесса. После завершения операции содержимое буферов приема `recvbuf` у всех процессов одинаково.

Схема выполнения операции `Allgather` представлена на рис. 73.

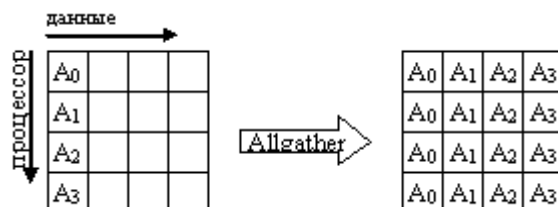


Рис. 73. Схема выполнения операции `Allgather`

```
int MPI_Allgather(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm)
```

Входные аргументы:

`sendbuf` - адрес начала буфера отправки;
`sendcount` - число посылаемых элементов;
`sendtype` - тип посылаемых элементов;
`recvcount` - число элементов, получаемых от каждого процесса;
`recvtype` - тип получаемых элементов;
`comm` - коммуникатор.

Входные аргументы:

`Recvbuf` - адрес начала буфера приема.

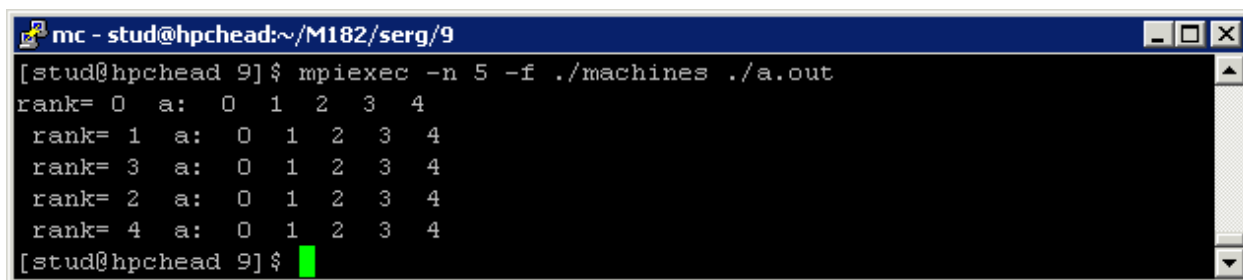
```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=0,i,s=0;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    int *a=new int[size];
12.    MPI_Allgather(&rank,1,MPI_INT,a,1,MPI_INT,MPI_COMM_WORLD);
13.    for(i=0; i<100000000*rank; i++)
14.        s+=1;
15.    printf("rank= %d a: ",rank);
16.    for(i=0; i<size; i++)
17.        printf(" %d ",a[i]);
18.    printf("\n ");
19.    MPI_Finalize();
20.    return 0;
21. }
```

Рис. 74. Листинг программы

Пример 3.

Повторим программу из первого примера, но для сборки данных в массив применим функцию `MPI_Allgather`.

Листинг программы приведен на рис. 74 (изменена только 12 строка), скрин ее запуска – на рис. 75. На скрине можно заметить, что массив после вызова коллективной функции сборки доступен на каждом процессе.



```
mc - stud@hpchead:~/M182/serg/9
[stud@hpchead 9]$ mpirun -n 5 -f ./machines ./a.out
rank= 0  a:  0  1  2  3  4
rank= 1  a:  0  1  2  3  4
rank= 3  a:  0  1  2  3  4
rank= 2  a:  0  1  2  3  4
rank= 4  a:  0  1  2  3  4
[stud@hpchead 9]$
```

Рис. 75. Скрин запуска программы

9.2. Функции сбора неодинаковых блоков данных от всех процессов

Функция `MPI_Gatherv` позволяет собирать блоки с разным числом элементов от каждого процесса, так как количество элементов, принимаемых от каждого процесса, задается индивидуально с помощью массива `recvcounts`. Эта функция обеспечивает также большую гибкость при размещении данных в процессе-получателе, благодаря введению в качестве параметра массива смещений `displs`.

```
int MPI_Gatherv(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* rbuf, int *recvcounts, int
*displs, MPI_Datatype recvtype, int root, MPI_Comm
comm)
```

Входные аргументы:

- `sendbuf` - адрес начала буфера передачи;
- `sendcount` - число посылаемых элементов;
- `sendtype` - тип посылаемых элементов;
- `recvcounts` - целочисленный массив (размер равен числу процессов в группе), *i*-й элемент которого определяет число элементов, которое должно быть получено от процесса *i*;
- `displs` - целочисленный массив (размер равен числу процессов в группе), *i*-ое значение определяет смещение *i*-го блока данных относительно начала `rbuf`;
- `recvtype` - тип получаемых элементов;
- `root` - номер процесса-получателя;
- `comm` - коммуникатор.

Выходные аргументы:

- `rbuf` - адрес начала буфера приема.

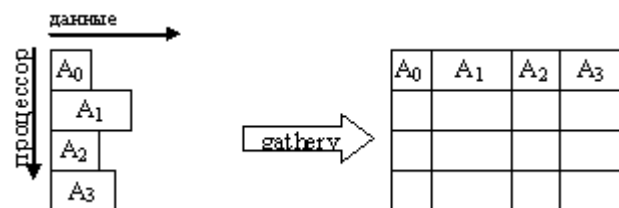


Рис. 76. Графическая интерпретация операции Gather

Сообщения помещаются в буфер приема процесса root в соответствии с номерами посылающих процессов, а именно, данные, посланные процессом i , размещаются в адресном пространстве процесса root, начиная с адреса $rbuf + displs[i]$. Графическая интерпретация операции Gather представлена на рис. 76.

Функция *MPI_Allgather* является аналогом функции *MPI_Gather*, но сборка выполняется всеми процессами группы. Поэтому в списке параметров отсутствует параметр root.

```
int MPI_Allgather(void* sendbuf, int sendcount,
MPI_Datatype
sendtype, void* rbuf, int *recvcounts, int *displs,
MPI_Datatype recvtype, MPI_Comm comm)
```

Входные аргументы:

- sendbuf - адрес начала буфера передачи;
- sendcount - число посылаемых элементов;
- sendtype - тип посылаемых элементов;
- recvcounts - целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, которое должно быть получено от каждого процесса;
- displs - целочисленный массив (размер равен числу процессов в группе), i -ое значение определяет смещение относительно начала $rbuf$ i -го блока данных;
- recvtype - тип получаемых элементов;
- comm - коммунникатор.

Выходные аргументы:

- rbuf - адрес начала буфера приема.

Пример 4.

Усложним задачу – теперь требуется собрать с каждого процесса массивы разной длины. Реализуем следующий алгоритм (рис. 77):

- Каждый процесс объявляет массив $b[\text{rank}+1]$ (строка 12).
- Каждый процесс присваивает $b[i]=\text{rank}$ (строки 13-14).

В итоге на 0-м процессе это будет массив из одного элемента $\{0\}$, на 1-м – $\{1,1\}$, на втором – $\{2,2,2\}$ и т.д. Для сборки распределенных массивов в один требуется определить его размер, который будет равен сумме элементов

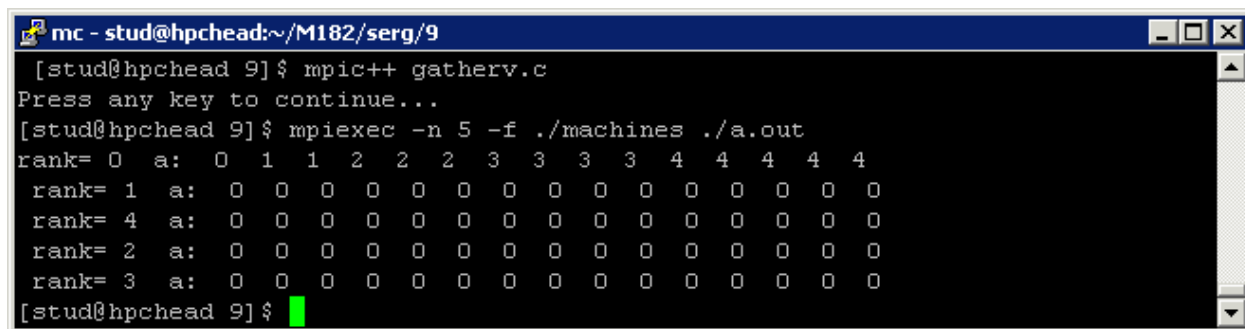
массива *b* каждого процесса. По сути надо найти сумму членов арифметической прогрессии $1+2+3+\dots+(size)=size*(size+1)/2$.

- Каждый процесс объявляет массив $a[size*(size+1)/2]$.
- Объявляются вспомогательные массивы *RC* (для указания количества принимаемых элементов с каждого процесса), *ds* (для указания адресов размещения посылок в итоговый массив) (строки 16-17).
- Вспомогательным массивам присваиваются соответствующие задаче значения (строки 18-22).
- Выполняется коллективная операция *MPI_Gatherv* (результат будет доступен только одному корневому процессу) или *MPI_Allgatherv* (результат будет доступен каждому процессу).
- Каждый процесс выводит на экран свой номер и массив *a*.

Листинг программы приведен на рис. 77, скрин ее запуска – на рис. 78.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int i,s=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int *b=new int [rank+1];
13.    for(i=0;i<rank+1;i++)
14.        b[i]=rank;
15.    int *a=new int[(size*(size+1))/2];
16.    int *RC = new int[size];
17.    int *ds = new int [size];
18.    for (int i=0;i<size;i++)
19.    {
20.        RC[i]=i+1;
21.        ds[i]=(i*(i+1))/2;
22.    }
23.    MPI_Gatherv(b,rank+1,MPI_INT,a,RC,ds,MPI_INT,0,MPI_COMM_WORLD);
24.    for(i=0; i<100000000*rank; i++)
25.        s+=1;
26.    printf("rank= %d a: ",rank);
27.    for(i=0; i<(size*(size+1))/2; i++)
28.        printf(" %d ",a[i]);
29.    printf("\n ");
30.    MPI_Finalize();
31.    return 0;
32. }
```

Рис. 77. Листинг программы для сборки массивов разной длины



```
mc - stud@hpchead:~/M182/serg/9
[stud@hpchead 9]$ mpic++ gatherv.c
Press any key to continue...
[stud@hpchead 9]$ mpiexec -n 5 -f ./machines ./a.out
rank= 0 a:  0  1  1  2  2  2  3  3  3  3  4  4  4  4
rank= 1 a:  0  0  0  0  0  0  0  0  0  0  0  0  0  0
rank= 4 a:  0  0  0  0  0  0  0  0  0  0  0  0  0  0
rank= 2 a:  0  0  0  0  0  0  0  0  0  0  0  0  0  0
rank= 3 a:  0  0  0  0  0  0  0  0  0  0  0  0  0  0
[stud@hpchead 9]$
```

Рис. 78. Скрин запуска программы

9.3. Примеры работы функций сборки блоков данных с двумерными массивами

Работа с двумерными массивами ничем не отличается от работы с одномерными, однако традиционно вызывает затруднения в освоении. В связи с этим рассмотрим ряд примеров применения разновидностей функций сборки блоков двумерных массивов.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n,i,s=0, j;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    if(rank==0) scanf("%d",&n);
12.    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
13.    int *b=new int [n];
14.    for(i=0; i<n; i++)
15.        b[i]=rank;
16.    int **a=new int *[size];
17.        a[0]=new int [n*size];
18.    for(i=1; i<size; i++)
19.        a[i]=a[i-1]+n;
20.    MPI_Gather(b,n,MPI_INT,*a,n,MPI_INT,0,MPI_COMM_WORLD);
21.    if(rank==0)
22.    {
23.        printf("rank= %d a: \n",rank);
24.        for(i=0; i<size; i++)
25.        {
26.            for(j=0; j<n; j++)
27.                printf(" %d ",a[i][j]);
28.            printf("\n ");
29.        }
30.    }
31.    MPI_Finalize();
32.    return 0;
33. }
```

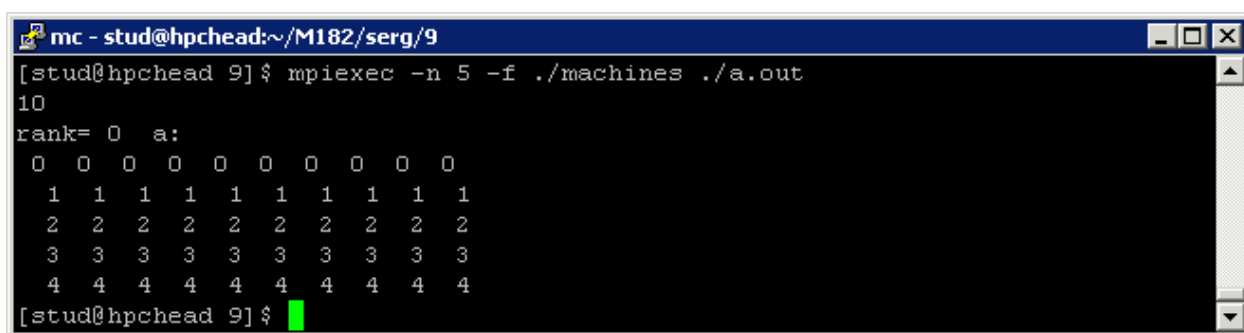
Рис. 79. Листинг программы сборки массивов

Пример 5.

Реализуем алгоритм построчной сборки двумерного массива на 0-м процессе. Алгоритм следующий:

- На 0-м процессе вводится значение переменной n (строка 11).
- С помощью функции широковещательной рассылки эта переменная передается каждому процессу (строка 12).
- Каждый процесс объявляет массив $b[n]$ (строка 13).
- Каждый процесс присваивает $b[i]=rank$ (строки 14-15).
- Каждый процесс объявляет двумерный массив $a[size][n]$ (строки 16-19).
- Выполняется коллективная операция `MPI_Gather` со сборкой одномерных массивов в строки матрицы массива a на 0-м процессе (строка 20).
- На 0-м процессе выводится собранный массив a (строки 21-30).

Листинг программы приведен на рис. 79, скрин ее запуска – на рис. 80.



```
mc - stud@hpchead:~/M182/serg/9
[stud@hpchead 9]$ mpirun -n 5 -f ./machines ./a.out
10
rank= 0  a:
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
[stud@hpchead 9]$
```

Рис. 80. Скрин запуска программы

Пример 6.

Усложним задачу и реализуем алгоритм построчной сборки двумерного массива на 0-м процессе из собираемых строк разной длины. Алгоритм следующий:

- Каждый процесс объявляет массив $b[rank+1]$ (строка 11).
- Каждый процесс присваивает $b[i]=rank$ (строки 12-13).
- Каждый процесс объявляет двумерный массив $a[size][size]$ (строки 14-17).
- Объявляются вспомогательные массивы RC (для указания количества принимаемых элементов с каждого процесса), ds (для указания адресов размещения посылок в итоговый массив) (строки 18, 21).
- Вспомогательным массивам присваиваются соответствующие задаче значения (строки 20, 23).
- Выполняется коллективная операция `MPI_Gatherv` со сборкой одномерных массивов в строки матрицы массива a на 0-м процессе (строка 24).
- На 0-м процессе выводится собранный массив a (строки 25-34).

Листинг программы приведен на рис. 81, скрин ее запуска – на рис.82.

```

1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int i,s=0, j;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    int *b=new int [rank+1];
12.    for(i=0; i<rank+1; i++)
13.        b[i]=rank;
14.    int **a=new int *[size];
15.        a[0]=new int [size*size];
16.    for(i=1; i<size; i++)
17.        a[i]=a[i-1]+size;
18.    int *RC = new int[size];
19.    for (int i=0;i<size;i++)
20.        RC[i]=i+1;
21.    int *ds = new int [size];
22.    for (int i=0;i<size;i++)
23.        ds[i]=i*size;
24.    MPI_Gatherv(b,rank+1,MPI_INT,*a,RC,ds,MPI_INT,0,MPI_COMM_WORLD);
25.    if(rank==0)
26.    {
27.        printf("rank= %d a: \n",rank);
28.        for(i=0; i<size; i++)
29.        {
30.            for(j=0; j<size; j++)
31.                printf(" %d ",a[i][j]);
32.            printf("\n ");
33.        }
34.    }
35.    MPI_Finalize();
36.    return 0;
37. }

```

Рис. 81. Листинг программы

```

mc - stud@hpchead:~/M182/serg/9
[stud@hpchead 9]$ mpiexec -n 10 -f ./machines ./a.out
rank= 0 a:
0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 0
2 2 2 0 0 0 0 0 0 0
3 3 3 3 0 0 0 0 0 0
4 4 4 4 4 0 0 0 0 0
5 5 5 5 5 5 0 0 0 0
6 6 6 6 6 6 6 0 0 0
7 7 7 7 7 7 7 7 0 0
8 8 8 8 8 8 8 8 8 0
9 9 9 9 9 9 9 9 9 9
[stud@hpchead 9]$

```

Рис. 82. Скрин запуска программы

Задания

1. Используя коллективные коммуникационные функции, создайте следующую параллельную программу: на всех процессах задается одномерный массив ($a[i]=rank$, $i=0\dots2$), который отправляется на 0-й процесс; 0-й процесс принимает от всех остальных пересылаемые данные в одномерный массив. Например, Вы запускаете программу на 3-х процессах, 1-й процесс отправляет 0-му следующий массив (1,1,1), 2-й процесс отправляет 0-му - (2,2,2), 0-й процесс получает данные и, сохраняя в одномерный массив, выводит на экран следующее: 1,1,1,2,2,2.
2. Создайте следующую параллельную программу: на всех процессах задается одномерный массив ($a[i]=rank$, $i=0\dots size-rank$), который отправляется на 0-й процесс; 0-й процесс принимает от всех остальных пересылаемые данные. Например, Вы запускаете программу на 4-х процессах, 0-й процесс будет отправлять следующий массив (0,0,0...0), состоящий из $size$ элементов, 1-й процесс отправляет 0-му следующий массив (1,1...1), состоящий из $size-1$ элемента, ..., $(size-1)$ -й процесс отправляет 0-му - $(size-1)$, 0-й процесс получает данные и, сохраняя в одномерный массив, выводит на экран следующее: 0,...,0,1...1,...,(size-1).
3. Напишите программу, которая реализует сборку по две строки с каждого процесса в двумерный массив на 0-м процессе по следующему алгоритму: на 0-м процессе считывается значение переменной n и рассылается каждому; на каждом процессе выделяется место в памяти под двумерный массив $a[2][n]$, который заполняется номером процесса; на всех процессах выделяется место в памяти под двумерный массив $b[2*size][n]$, после чего производится сборка массивов a в массив b на 0-м процессе.
4. На основе примера 6 напишите программу, которая собирает с каждого процесса по две строки длиной $(rank+1)$ элементов.
5. На основе примера 6 напишите программу, которая собирает с каждого процесса по одной строке, но с 0-го процесса – строку длины $size$ элементов, с 1-го процесса – $(size-1)$ элемент и т.д. Например, запустив программу на 5-и процессах, в итоге Вы должны получить следующую матрицу:

```
0 0 0 0 0
1 1 1 1 0
2 2 2 0 0
3 3 0 0 0
4 0 0 0 0
```

6. Модифицируйте предыдущий пример, осуществляя сохранение каждой строки, начиная с элемента на главной диагонали. Например, запустив программу на 5-и процессах, в итоге Вы должны получить следующую матрицу:

```
0 0 0 0 0
0 1 1 1 1
0 0 2 2 2
0 0 0 3 3
0 0 0 0 4
```

10. Коллективные операции: функции распределения блоков данных по всем процессам

10.1. Функция распределения одинаковых блоков данных по всем процессам

Семейство функций распределения блоков данных по всем процессам группы состоит из двух: `MPI_Scatter` и `MPI_Scatterv`.

Функция `MPI_Scatter` разбивает сообщение из буфера отправки процесса `root` на равные части размером `sendcount` и посылает i -ю часть в буфер приема процесса с номером i (в том числе и самому себе).

На рис. 83 представлена графическая интерпретация операции `Scatter`.

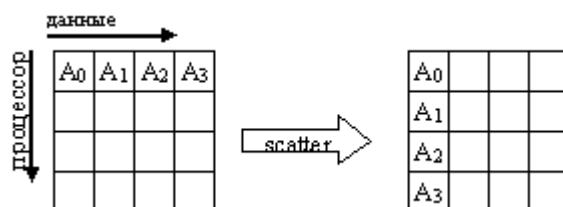


Рис. 83. Графическая интерпретация операции `Scatter`

Процесс `root` использует оба буфера (отправки и приема), поэтому в вызываемой им функции все аргументы являются существенными. Остальные процессы группы с коммуникатором `comm` являются только получателями, поэтому для них аргументы, специфицирующие буфер отправки, не существенны.

```
int MPI_Scatter(void* sendbuf, int sendcount,  
MPI_Datatype sendtype, void* recvbuf, int recvcount,  
MPI_Datatype recvtype, int root, MPI_Comm comm)
```

Входные аргументы:

- `sendbuf` - адрес начала размещения блоков распределяемых данных (используется только в процессе-отправителе `root`);
- `sendcount` - число элементов, посылаемых каждому процессу;
- `Sendtype` - тип посылаемых элементов;
- `recvcount` - число получаемых элементов;
- `recvtype` - тип получаемых элементов;
- `root` - номер процесса-отправителя;
- `comm` - коммуникатор.

Выходные аргументы:

- `recvbuf` - адрес начала буфера приема.

Тип посылаемых элементов `sendtype` должен совпадать с типом `recvtype` получаемых элементов, а число посылаемых элементов `sendcount` должно равняться числу принимаемых `recvcount`. Следует обратить внимание, что значение `sendcount` в вызове из процесса `root` – это число посылаемых каждому процессу элементов, а не общее их количество. Операция `Scatter` является обратной по отношению к `Gather`.

Пример 1.

В качестве примера использования изучаемой функции рассмотрим задачу распределения одномерного массива, расположенного в памяти 0-го процесса, по три элемента на каждый процесс.

Листинг программы приведен на рис. 84, скрин ее запуска – на рис. 85. В строке 11 выделено место в памяти под массив `a[3*size]`, чтобы в дальнейшем каждому процессу раздать по 3 элемента. Строки 12-16 – присваивание элементам рассылаемого массива значений на 0-м процессе. Строка 17 – выделение места в памяти под массив, в который будет осуществлена рассылка. Строка 18 – вызов функции `MPI_Scatter` для выполнения требуемой рассылки. Строки 19-24 – контрольный вывод полученных массивов.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int i,j,s=0;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    int *a=new int [3*size];
12.    if(rank==0)
13.    {
14.        for(i=0;i<3*size;i++)
15.            a[i]=i;
16.    }
17.    int *b=new int[3];
18.    MPI_Scatter(a,3,MPI_INT,b,3,MPI_INT,0,MPI_COMM_WORLD);
19.    for(i=0; i<100000000*rank; i++)
20.        s+=1;
21.    printf("rank= %d a: ",rank);
22.    for(i=0; i<3; i++)
23.        printf(" %d ",b[i]);
24.    printf("\n");
25.    MPI_Finalize();
26.    return 0;
27. }
```

Рис. 84. Листинг программы рассылки массива

```

mc - stud@hpchead:~/M182/serg/10
[stud@hpchead 10]$ mpiexec -n 5 -f ./machines ./a.out
rank= 0  a:  0  1  2
rank= 3  a:  9 10 11
rank= 1  a:  3  4  5
rank= 2  a:  6  7  8
rank= 4  a: 12 13 14
[stud@hpchead 10]$

```

Рис. 85. Скрин запуска программы

10.2. Функция распределения неодинаковых блоков данных по всем процессам

Функция *MPI_Scatterv* является векторным вариантом функции *MPI_Scatter*, позволяющим посылать каждому процессу различное количество элементов.

На рис. 86 представлена графическая интерпретация операции *Scatterv*.

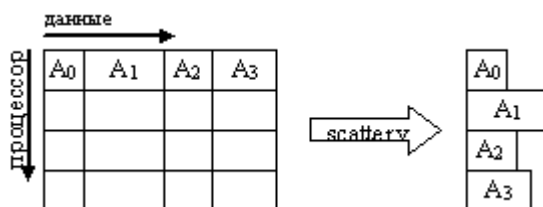


Рис. 86. Графическая интерпретация операции *Scatterv*

Начало расположения элементов блока, посылаемого *i*-му процессу, задается в массиве смещений *displs*, а число посылаемых элементов – в массиве *sendcounts*. Эта функция является обратной по отношению к функции *MPI_Gatherv*.

```

int MPI_Scatterv(void* sendbuf, int *sendcounts, int
*displs, MPI_Datatype sendtype, void* recvbuf, int
recvcount, MPI_Datatype recvtype, int root, MPI_Comm
comm)

```

Входные аргументы:

- sendbuf* - адрес начала буфера отправки (используется только в процессе-отправителе *root*);
- sendcounts* - целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, посылаемых каждому процессу;
- displs* - целочисленный массив (размер равен числу процессов в группе), *i*-ое значение определяет смещение относительно начала *sendbuf* для данных, посылаемых процессу *i*;
- sendtype* - тип посылаемых элементов;
- recvcount* - число получаемых элементов;
- recvtype* - тип получаемых элементов;

root - номер процесса-отправителя;

comm - коммуникатор.

Входные аргументы:

recvbuf - адрес начала буфера приема.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int i,s=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int *a=new int[(size*(size+1))/2];
13.    if(rank==0)
14.        for(i=0; i<((size*(size+1))/2); i++)
15.            a[i]=i;
16.    int *b=new int [rank+1];
17.    int *RC = new int[size];
18.    int *ds = new int [size];
19.    for(int i=0;i<size;i++)
20.    {
21.        RC[i]=i+1;
22.        ds[i]=(i*(i+1))/2;
23.    }
24.    MPI_Scatterv(a,RC,ds,MPI_INT,b,rank+1,MPI_INT,0,MPI_COMM_WORLD);
25.    for(i=0; i<100000000*rank; i++)
26.        s+=1;
27.    printf("rank= %d b: ",rank);
28.    for(i=0; i<rank+1; i++)
29.        printf(" %d ",b[i]);
30.    printf("\n ");
31.    MPI_Finalize();
32.    return 0;
33. }
```

Рис. 87. Листинг программы

Пример 2.

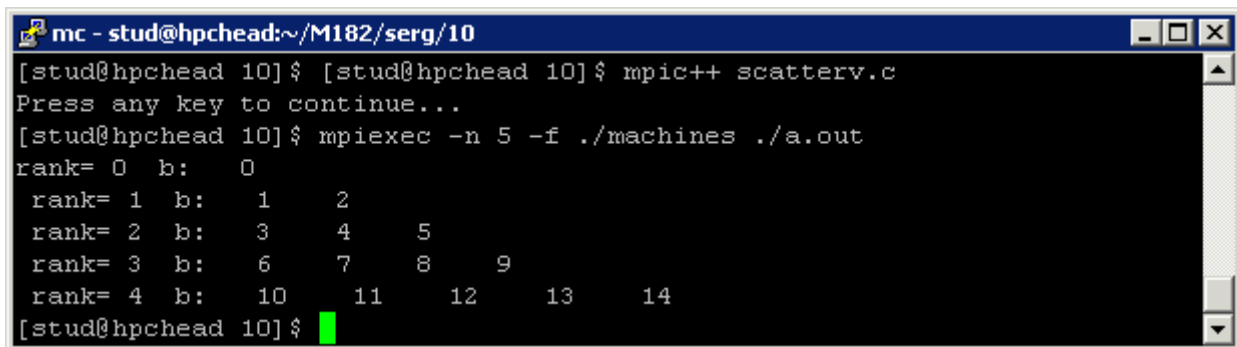
Усложним задачу – теперь требуется разослать одномерный массив а по процессам разными по размеру частями. Реализуем следующий алгоритм (рис. 87):

- Каждый процесс объявляет массив $a[\text{size}*(\text{size}+1)/2]$ (строка 12), но только 0-й процесс присваивает элементам этого массива значения (строки 13-15) и будет осуществлять его рассылку.
- Каждый процесс объявляет массив $b[\text{rank}+1]$ (строка 16), в который будут сохранены присланные элементы.
- Объявляются вспомогательные массивы RC (для указания количества отсылаемых элементов каждому процессу), ds (для указания адресов, с

которых будут отправляться элементы очередному процессу) (строки 17-18).

- Вспомогательным массивам присваиваются соответствующие задаче значения (строки 19-23).
- Выполняется коллективная операция MPI_Scatterv (строка 24).
- Контрольный вывод полученных массивов (строки 25-30).

Листинг программы приведен на рис. 87, скрин ее запуска – на рис. 88. В итоге на 0-м процессе будет получен массив из одного элемента {0}, на 1-м – {1,2}, на втором – {3,4,5} и т.д.



```
mc - stud@hpchead:~/M182/serg/10
[stud@hpchead 10]$ [stud@hpchead 10]$ mpic++ scatterv.c
Press any key to continue...
[stud@hpchead 10]$ mpiexec -n 5 -f ./machines ./a.out
rank= 0  b:   0
rank= 1  b:   1   2
rank= 2  b:   3   4   5
rank= 3  b:   6   7   8   9
rank= 4  b:  10  11  12  13  14
[stud@hpchead 10]$
```

Рис. 88. Скрин запуска программы

10.3. Примеры работы распределения блоков данных с двумерными массивами

Также как и в предыдущем пункте рассмотрим ряд примеров применения изученных функций для рассылки двумерных массивов.

Пример 3.

Реализуем алгоритм построчной рассылки двумерного массива, расположенного на 0-м процессе, в одномерные на все процессы в области связи. Алгоритм следующий:

- На 0-м процессе вводится значение переменной n (строка 11).
- С помощью функции широковещательной рассылки эта переменная передается каждому процессу (строка 12).
- Каждый процесс объявляет двумерный массив a[size][n] (строки 13-16).
- На 0-м процессе массиву a присваиваются значения (строки 17-22).
- Каждый процесс объявляет массив b[n] (строка 23).
- Выполняется коллективная операция MPI_Scatter для рассылки n элементов массива a в одномерные массивы b на каждый процесс (строка 24).
- Производится контрольный вывод полученных массивов (строки 25-30).

Листинг программы приведен на рис. 89, скрин ее запуска – на рис. 90.

```

1.  #include <stdio.h>
2.  #include "mpi.h"
3.  int main(int argc, char *argv[])
4.  {
5.      int rank;
6.      int size;
7.      int n=0,i,j,s=0;
8.      MPI_Init(&argc, &argv);
9.      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.     MPI_Comm_size(MPI_COMM_WORLD, &size);
11.     if(rank==0) scanf("%d",&n);
12.     MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
13.     int **a=new int *[size];
14.         a[0]=new int [n*size];
15.     for(i=1; i<size; i++)
16.         a[i]=a[i-1]+n;
17.     if(rank==0)
18.     {
19.         for(i=0;i<size;i++)
20.         for(j=0;j<n;j++)
21.             a[i][j]=i;
22.     }
23.     int *b=new int [n];
24.     MPI_Scatter(*a,n,MPI_INT,b,n,MPI_INT,0,MPI_COMM_WORLD);
25.     for(i=0; i<100000000*rank; i++)
26.         s+=1;
27.     printf("rank= %d b: \n",rank);
28.     for(i=0;i<n;i++)
29.         printf(" %d ",b[i]);
30.     printf("\n");
31.     MPI_Finalize();
32.     return 0;
33. }

```

Рис. 89. Листинг программы

```

mc - stud@hpchead:~/M182/serg/10
[stud@hpchead 10]$ mpirun -n 5 -f ./machines ./a.out
10
rank= 0 b:
0 0 0 0 0 0 0 0 0 0
rank= 1 b:
1 1 1 1 1 1 1 1 1 1
rank= 2 b:
2 2 2 2 2 2 2 2 2 2
rank= 3 b:
3 3 3 3 3 3 3 3 3 3
rank= 4 b:
4 4 4 4 4 4 4 4 4 4
[stud@hpchead 10]$

```

Рис. 90. Скрин запуска программы

Пример 4.

Усложним задачу – реализуем алгоритм рассылки двумерного массива, расположенного на 0-м процессе, в двумерные на все процессы по три строки каждому.

Листинг программы приведен на рис. 91, скрин ее запуска – на рис. 92. Попробуйте самостоятельно разобраться какие изменения внесены в программу по сравнению с предыдущим примером.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=0,i,j,s=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    if(rank==0) scanf("%d",&n);
13.    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
14.    int **a=new int *[3*size];
15.        a[0]=new int [3*n*size];
16.    for(i=1; i<3*size; i++)
17.        a[i]=a[i-1]+n;
18.    if(rank==0)
19.    {
20.        for(i=0;i<3*size;i++)
21.        for(j=0;j<n;j++)
22.            a[i][j]=i;
23.    }
24.    int **b=new int *[3];
25.        b[0]=new int [3*n];
26.    for(i=1; i<3; i++)
27.        b[i]=b[i-1]+n;
28.    MPI_Scatter(*a,3*n,MPI_INT,*b,3*n,MPI_INT,0,MPI_COMM_WORLD);
29.    for(i=0; i<100000000*rank; i++)
30.        s+=1;
31.    printf("rank= %d b: \n",rank);
32.    for(i=0; i<3; i++)
33.    {
34.        for(j=0;j<n;j++)
35.            printf(" %d ",b[i][j]);
36.        printf("\n");
37.    }
38.    MPI_Finalize();
39.    return 0;
40. }
```

Рис. 91. Листинг программы

```
mc - stud@hpchead:~/M182/serg/10
[stud@hpchead 10]$ mpic++ scatter2.c
Press any key to continue...
[stud@hpchead 10]$ mpiexec -n 5 -f ./machines ./a.out
10
rank= 0  b:
0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
rank= 1  b:
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
rank= 2  b:
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
rank= 3  b:
9 9 9 9 9 9 9 9 9 9
10 10 10 10 10 10 10 10 10 10
11 11 11 11 11 11 11 11 11 11
rank= 4  b:
12 12 12 12 12 12 12 12 12 12
13 13 13 13 13 13 13 13 13 13
14 14 14 14 14 14 14 14 14 14
[stud@hpchead 10]$
```

Рис. 92. Скрин запуска программы

Пример 5.

Теперь выполним рассылку блоков разной длины из двумерного массива в одномерные. Алгоритм следующий:

- Каждый процесс объявляет двумерный массив `a[size][size]` (строки 11-14).
- На 0-м процессе массиву `a` присваиваются значения (строки 15-20).
- Каждый процесс объявляет массив `b[rank+1]` (строка 28), в который будут сохранены присланные элементы.
- Объявляются вспомогательные массивы `RC` (для указания количества отсылаемых элементов каждому процессу), `ds` (для указания адресов, с которых будут отправляться элементы очередному процессу) (строки 21-22).
- Вспомогательным массивам присваиваются соответствующие задаче значения (строки 23-27).
- Выполняется коллективная операция `MPI_Scatterv` (строка 29).
- Контрольный вывод полученных массивов (строки 30-35).

Листинг программы приведен на рис. 93, скрин ее запуска – на рис. 94.

```

1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int i,s=0, j;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    int **a=new int *[size];
12.        a[0]=new int [size*size];
13.        for(i=1; i<size; i++)
14.            a[i]=a[i-1]+size;
15.    if(rank==0)
16.    {
17.        for(i=0;i<size;i++)
18.        for(j=0;j<size;j++)
19.            a[i][j]=i;
20.    }
21.    int *RC = new int[size];
22.    int *ds = new int [size];
23.    for (int i=0;i<size;i++)
24.    {
25.        RC[i]=i+1;
26.        ds[i]=i*size;
27.    }
28.    int *b=new int [rank+1];
29.    MPI_Scatterv(*a,RC,ds,MPI_INT,b,rank+1,MPI_INT,0,MPI_COMM_WORLD);
30.    for(i=0; i<100000000*rank; i++)
31.        s+=1;
32.    printf("rank= %d b: ",rank);
33.    for(i=0; i<rank+1; i++)
34.        printf(" %d ",b[i]);
35.    printf("\n ");
36.    MPI_Finalize();
37.    return 0;
38. }

```

Рис. 93. Листинг программы

```

mc - stud@hpchead:~/M182/serg/10
[stud@hpchead 10]$ mpiexec -n 5 -f ./machines ./a.out
rank= 0 b:  0
rank= 1 b:  1  1
rank= 2 b:  2  2  2
rank= 3 b:  3  3  3  3
rank= 4 b:  4  4  4  4  4
[stud@hpchead 10]$

```

Рис. 94. Скрин запуска программы

Пример 6.

Усложним задачу, соединив примеры 4-5: выполним рассылку двумерного массива из 0-го процесса по всем процессам с разным количеством строк: 0-му процессу отправим 1 строку, 1-му – 2 строки и т.д.

Сравните программы из примеров 4-5 с данной программой и попробуйте самостоятельно разобраться какие внесены изменения.

Листинг программы приведен на рис. 95, скрин ее запуска – на рис. 96.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n,i,s=0, j;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    if(rank==0) scanf("%d",&n);
12.    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
13.    int **a=new int *[(size*(size+1))/2];
14.        a[0]=new int [(size*(size+1))/2*n];
15.    for(i=1; i<((size*(size+1))/2); i++)
16.        a[i]=a[i-1]+n;
17.    if(rank==0)
18.    {
19.        for(i=0;i<((size*(size+1))/2;i++)
20.        for(j=0;j<n;j++)
21.            a[i][j]=i;
22.    }
23.    int **b=new int *[rank+1];
24.        b[0]=new int [(rank+1)*n];
25.    for(i=1; i<rank+1; i++)
26.        b[i]=b[i-1]+n;
27.    int *RC = new int[size];
28.    int *ds = new int [size];
29.    for (int i=0;i<size;i++)
30.    {
31.        RC[i]=(i+1)*n;
32.        ds[i]=i*(i+1)/2*n;
33.    }
34.    MPI_Scatterv(*a,RC,ds,MPI_INT,*b,(rank+1)*n,MPI_INT,0,MPI_COMM_WORLD);
35.    for(i=0; i<100000000*rank; i++)
36.        s+=1;
37.    printf("rank= %d b: \n",rank);
38.    for(i=0; i<rank+1; i++)
39.    {
40.        for(j=0;j<n;j++)
41.            printf(" %d ",b[i][j]);
42.        printf("\n");
43.    }
44.    MPI_Finalize();
45.    return 0;
46. }
```

Рис. 95. Листинг программы

```
mc - stud@hpchead:~/M182/serg/10
[stud@hpchead 10]$ mpiexec -n 5 -f ./machines ./a.out
10
rank= 0  b:
0 0 0 0 0 0 0 0 0 0
rank= 1  b:
1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2
rank= 3  b:
6 6 6 6 6 6 6 6 6 6
7 7 7 7 7 7 7 7 7 7
8 8 8 8 8 8 8 8 8 8
9 9 9 9 9 9 9 9 9 9
rank= 2  b:
3 3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4 4
5 5 5 5 5 5 5 5 5 5
rank= 4  b:
10 10 10 10 10 10 10 10 10 10
11 11 11 11 11 11 11 11 11 11
12 12 12 12 12 12 12 12 12 12
13 13 13 13 13 13 13 13 13 13
14 14 14 14 14 14 14 14 14 14
[stud@hpchead 10]$
```

Рис. 96. Скрин запуска программы

Задания

1. Используя функцию `MPI_Scatter`, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a[i]=i$, $i=0\dots 2*\text{size}$), который распределяется по процессам по 2 элемента каждому.
2. Используя функцию `MPI_Scatterv`, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a[i]=i$, $i=0\dots \text{size}*(\text{size}+1)$), который распределяется по процессам следующим образом: на 0-й процесс будет отправлено 2 элемента (0,1), на 1-й процесс – 4 элемента (2,3,4,5), на 2-й процесс – 6 элементов и т.д..
3. Напишите программу, которая реализует рассылку по две строки каждому процессу из двумерного массива, расположенного в памяти 0-го процесса.
4. Используя функцию `MPI_Scatterv`, напишите программу, которая распределяет двумерный массив (квадратную матрицу произвольного размера), расположенный в памяти 0-го процесса, как можно одинаковыми по размеру блоками строк по процессам. Возможный алгоритм следующий:
 - На 0-м процессе считывается значение переменной n (размер массива) и рассылается всем (`MPI_Bcast`).
 - Выделяется место под рассылаемый массив $a[n][n]$.
 - На 0-м процессе элементам массива присваиваются значения.

- Вычисляется оптимальное распределение строк в блоках для рассылки каждому процессу (разница в количестве рассылаемых строк не более 1).
 - Выделяется место в памяти под двумерные массивы на каждом процессе для сохранения рассылаемых блоков строк.
 - Выделяется место в памяти под вспомогательные массивы и им присваиваются соответствующие задаче значения.
 - Выполняется распределение массива по процессам.
 - Производится контрольный вывод полученных массивов.
5. На основе примера 6 напишите программу, которая рассылает с 0-го процесса двумерный массив так, что в итоге каждому процессу достанется по две строки длиной $(rank+1)$ элементов.
 6. На основе примера 5 напишите программу, которая осуществляет рассылку двумерного массива с 0-го процесса следующим образом (в случае запуска на 5-и процессах пример изначального массива и что каждый процесс получит):
 0 0 0 0 0 (0-й процесс получит строку целиком)
 1 1 1 1 1 (1-й процесс получит 1 1 1 1)
 2 2 2 2 2 (2-й процесс получит 2 2 2)
 3 3 3 3 3 (3-й процесс получит 3 3)
 4 4 4 4 4 (4-й процесс получит 4).

11. Коллективные вычислительные и совмещенные операции

11.1. Глобальные вычислительные операции

В параллельном программировании математические операции над блоками данных, распределенных по процессорам, называют глобальными операциями редукции. В общем случае операцией редукции называется операция, аргументом которой является вектор, а результатом – скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора. В частности, если компоненты вектора расположены в адресных пространствах процессов, выполняющихся на различных процессорах, то в этом случае говорят о глобальной (параллельной) редукции. Например, пусть в адресном пространстве всех процессов некоторой группы процессов имеются копии переменной *var* (необязательно имеющие одно и то же значение), тогда применение к ней операции вычисления глобальной суммы или, другими словами, операции редукции SUM возвратит одно значение, которое будет содержать сумму всех локальных значений этой переменной. Использование этих операций является одним из основных средств организации распределенных вычислений.

В MPI глобальные операции редукции представлены в нескольких вариантах:

- с сохранением результата в адресном пространстве одного процесса (MPI_Reduce);
- с сохранением результата в адресном пространстве всех процессов (MPI_Allreduce);
- префиксная операция редукции, которая в качестве результата операции возвращает вектор. *i*-я компонента этого вектора является результатом редукции первых *i* компонент распределенного вектора (MPI_Scan);
- совмещенная операция Reduce/Scatter (MPI_Reduce_scatter).

Функция *MPI_Reduce* выполняется следующим образом. Операция глобальной редукции, указанная параметром (OP), выполняется над первыми элементами входного буфера, и результат посылается в первый элемент буфера приема процесса root. Затем то же самое делается для вторых элементов буфера и т. д.

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm  
comm)
```

Входные аргументы:

sendbuf - адрес начала входного буфера;
count - число элементов во входном буфере;
datatype - тип элементов во входном буфере;
op - операция, по которой выполняется редукция;
root - номер процесса-получателя результата операции;
comm - коммуникатор.

Выходные аргументы:

recvbuf - адрес начала буфера результатов (используется только в процессе-получателе root).

На рис. 97 представлена графическая интерпретация операции Reduce. На данной схеме операция "+" означает любую допустимую операцию редукции.

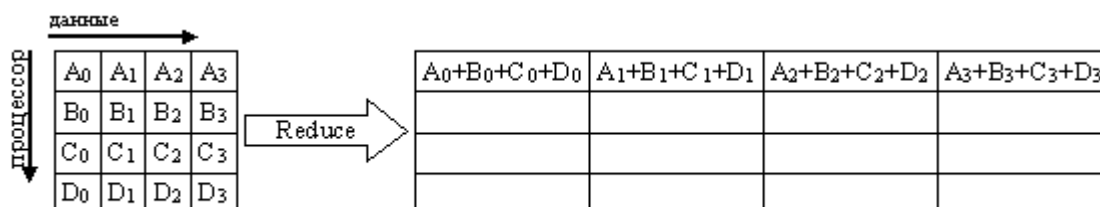


Рис. 97. Графическая интерпретация операции Reduce

В качестве операции (OP) можно использовать либо одну из predefined операций, либо операцию, сконструированную пользователем. Все predefined операции являются ассоциативными и коммутативными. Сконструированная пользователем операция, по крайней мере, должна быть ассоциативной. Порядок редукции определяется номерами процессов в группе. Тип datatype элементов должен быть совместим с операцией (OP). В таблице 4 представлен перечень predefined операций, которые могут быть использованы в функциях редукции MPI.

Таблица 4: Перечень predefined операций

Название	Операция	Разрешенные типы
MPI_MAX	Максимум	C integer, Floating point
MPI_MIN	Минимум	
MPI_SUM	Сумма	C integer, Floating point
MPI_PROD	Произведение	
MPI_LAND	Логическое AND	C integer, Logical
MPI_LOR	Логическое OR	
MPI_LXOR	Логическое исключающее OR	
MPI_BAND	Поразрядное AND	C integer, Byte
MPI BOR	Поразрядное OR	
MPI_BXOR	Поразрядное исключающее OR	
MPI_MAXLOC	Максимальное значение и его индекс	Специальные типы для этих функций
MPI_MINLOC	Минимальное значение и его индекс	

В таблице 4 используются следующие обозначения:

C integer: MPI_INT, MPI_LONG, MPI_SHORT,
MPI_UNSIGNED_SHORT, MPI_UNSIGNED,
MPI_UNSIGNED_LONG
Floating point: MPI_FLOAT, MPI_DOUBLE, MPI_REAL,
MPI_DOUBLE_PRECISION, MPI_LONG_DOUBLE
Logical: MPI_LOGICAL
Byte: MPI_BYTE

Операции MAXLOC и MINLOC выполняются над специальными парными типами, каждый элемент которых хранит две величины: значение, по которому ищется максимум или минимум, и индекс элемента. В MPI имеется 6 таких predefined типов.

C:

MPI_FLOAT_INT	float	and	int
MPI_DOUBLE_INT	double	and	int
MPI_LONG_INT	long	and	int
MPI_2INT	int	and	int
MPI_SHORT_INT	short	and	int
MPI_LONG_DOUBLE_INT	long double	and	int

Функция *MPI_Allreduce* сохраняет результат редукции в адресном пространстве всех процессов, поэтому в списке параметров функции отсутствует идентификатор корневого процесса root. В остальном, набор параметров такой же, как и в предыдущей функции.

int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Входные аргументы:

sendbuf - адрес начала входного буфера;
count - число элементов во входном буфере;
datatype - тип элементов во входном буфере;
op - операция, по которой выполняется редукция;
comm - коммуникатор.

Выходные аргументы:

recvbuf - адрес начала буфера приема.

На рис. 98 представлена графическая интерпретация операции Allreduce.

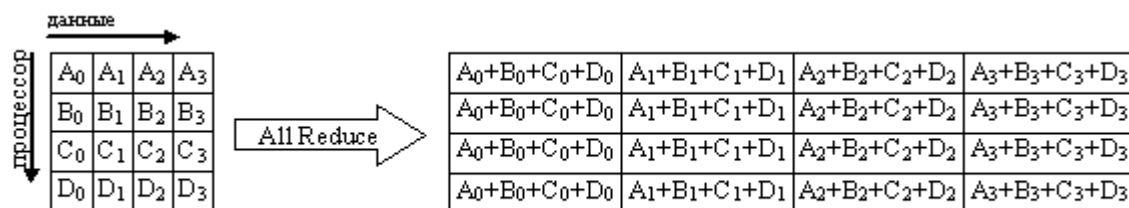


Рис. 98. Графическая интерпретация операции Allreduce

Пример 1.

Продemonстрируем работу данной функции на примере суммирования номеров процессов с сохранением результата на 0-м процессе.

Листинг программы приведен на рис. 99, скрин ее запуска – на рис. 100.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int s=0;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    MPI_Reduce(&rank,&s,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
12.    printf("rank=%d s=%d \n",rank, s);
13.    MPI_Finalize();
14.    return 0;
15. }
```

Рис. 99. Листинг программы суммирования распределенных данных

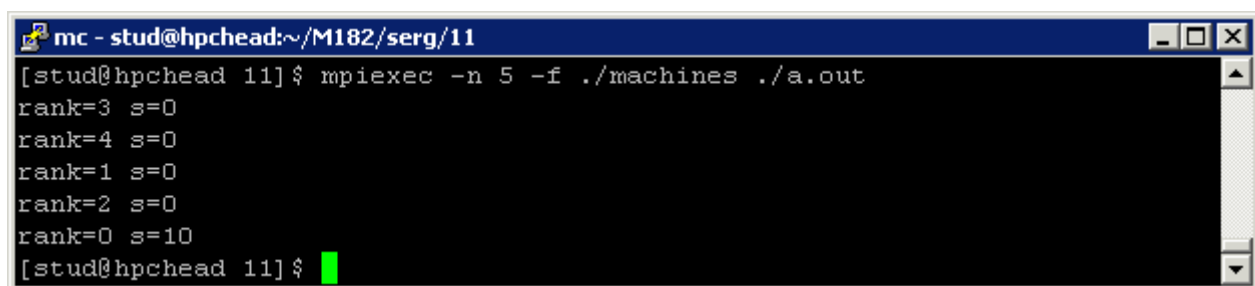


Рис. 100. Скрин запуска программы

```
16. #include <stdio.h>
17. #include "mpi.h"
18. int main(int argc, char *argv[])
19. {
20.     int rank;
21.     int size;
22.     int s=0;
23.     MPI_Init(&argc, &argv);
24.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
25.     MPI_Comm_size(MPI_COMM_WORLD, &size);
26.     MPI_Allreduce(&rank,&s,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
27.     printf("rank=%d s=%d \n",rank, s);
28.     MPI_Finalize();
29.     return 0;
30. }
```

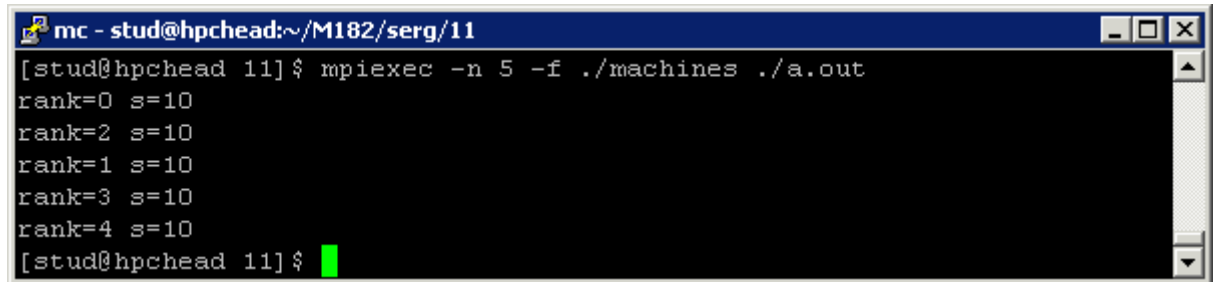
Рис. 101. Листинг программы

Пример 2.

Модифицируем предыдущую программу – будем использовать функцию MPI_Allreduce. Если в предыдущем примере результат суммирования

сохранялся только в памяти 0-го процесса, то теперь он доступен каждому процессу.

Листинг программы приведен на рис. 101, скрин ее запуска – на рис. 102.



```
mc - stud@hpchead:~/M182/serg/11
[stud@hpchead 11]$ mpiexec -n 5 -f ./machines ./a.out
rank=0 s=10
rank=2 s=10
rank=1 s=10
rank=3 s=10
rank=4 s=10
[stud@hpchead 11]$
```

Рис. 102. Скрин запуска программы

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n,i,s=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    if(rank==0) scanf("%d",&n);
13.    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
14.    int *a=new int[n];
15.    for(i=0;i<n;i++)
16.        a[i]=rank;
17.    int *b=new int[n];
18.    MPI_Reduce(a,b,n,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
19.    for(i=0; i<100000000*rank; i++)
20.        s+=1;
21.    printf("rank= %d b: ",rank);
22.    for(i=0;i<n;i++)
23.        printf(" %d ", b[i]);
24.    printf("\n ");
25.    MPI_Finalize();
26.    return 0;
27. }
```

Рис. 103. Листинг программы суммирования векторов

Пример 3.

Теперь выполним суммирование распределенных по процессам векторов с сохранением вектора-результата в памяти 0-го процесса. Алгоритм следующий:

- На 0-м процессе считывается значение переменной n (строка 12).
- С помощью функции широковещательной рассылки значение считанной переменной рассылается каждому процессу (строка 13).

- Каждый процесс объявляет массив `a[n]` (строка 14) и присваивает элементам массива свой номер (строки 15-16).
 - В строке 17 объявляется массив `b[n]` для сборки результата суммирования.
 - В строке 18 вызывается функция `MPI_Reduce` для распределенного суммирования векторов `a` с сохранением результата в вектор `b` на 0-м процессе.
 - Выполняется контрольный вывод полученного массива (строки 19-24).
- Листинг программы приведен на рис. 103, скрин ее запуска – на рис. 104.

```
mc - stud@hpchead:~/M182/serg/11
[stud@hpchead 11]$ mpirun -n 5 -f ./machines ./a.out
11
rank= 0 b: 10 10 10 10 10 10 10 10 10 10 10
rank= 1 b: 0 0 0 0 0 0 0 0 0 0 0
rank= 3 b: 0 0 0 0 0 0 0 0 0 0 0
rank= 2 b: 0 0 0 0 0 0 0 0 0 0 0
rank= 4 b: 0 0 0 0 0 0 0 0 0 0 0
[stud@hpchead 11]$
```

Рис. 104. Скрин запуска программы

```

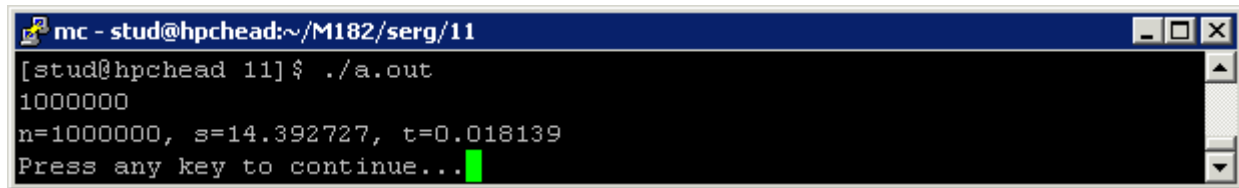
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10.    double t1, t2, s=0;
11.    int n;
12.    scanf("%d",&n);
13.    t1=MPI_Wtime();
14.    for(int i=0; i<n; i++)
15.        s+=1./(1.+i);
16.    t2=MPI_Wtime();
17.    printf("n=%d, s=%f, t=%f\n",n,s,t2-t1);
18.    MPI_Finalize();
19.    return 0;
20. }
```

Рис. 105. Листинг последовательной программы

Пример 4.

Проведем распараллеливание алгоритма суммирования ряда чисел. На рис. 105 приведен листинг последовательной программы, в которой считывается предел ряда, засекается время, производится суммирование ряда чисел, замеряется время окончания суммирования и результаты выводятся на экран. На рис. 106 приведен скрин запуска программы для предела ряда

равного 1000000. Время работы цикла для суммирования ряда составило 0,018139 сек.



```
mc - stud@hpchead:~/M182/serg/11
[stud@hpchead 11]$ ./a.out
1000000
n=1000000, s=14.392727, t=0.018139
Press any key to continue...
```

Рис. 106. Скрин запуска программы

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10.    double t1, t2, s=0, s1;
11.    int n;
12.    if(rank==0) scanf("%d",&n);
13.    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
14.    if(rank==0) t1=MPI_Wtime();
15.    for(int i=rank; i<n; i+=size)
16.        s+=1./(1.+i);
17.    MPI_Reduce(&s,&s1,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
18.    if(rank==0)
19.    {
20.        t2=MPI_Wtime();
21.        printf("n=%d, s=%f, t=%f \n",n,s1,t2-t1);
22.    }
23.    MPI_Finalize();
24.    return 0;
25. }
```

Рис. 107. Листинг параллельной программы

Реализуем параллельную программу суммирования ряда чисел. Давайте проанализируем каким изменениям подверглась последовательная программа при трансформации в параллельную (рис. 107).

- Считывание значения переменной *n* допустимо только на 0-м процессе (строка 12).
- Значение считанной переменной необходимо разослать всем остальным процессам (строка 13, реализовано при помощи функции широковещательной рассылки).
- 14 строка – засекли момент времени перед выполнение суммирования на 0-м процессе.
- Для параллельного суммирования ”развернули” цикл в строке 15.
- В строке 17 при помощи функции `MPI_Reduce` собрали окончательный результат суммирования частичных сумм в переменную *s1*.
- 18-22 строки – замер времени окончания блока кода и вывод результатов.

На рис. 108 приведен скрин запуска программы на 2-х процессах с вводом предела ряда такого же, как и в последовательной программе. Время работы программы сократилось в два раза!

```
mc - stud@hpchead:~/M182/serg/11
[stud@hpchead 11]$ mpiexec -n 2 -f ./machines ./a.out
1000000
n=1000000, s=14.392727, t=0.009026
[stud@hpchead 11]$
```

Рис. 108. Скрин запуска программы

11.2. Совмещенные коллективные операции

Функция *MPI_Reduce_scatter* совмещает в себе операции редукции и распределения результата по процессам.

MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Входные аргументы:

- sendbuf - адрес начала входного буфера;
- recvcount - массив, в котором задаются размеры блоков, посылаемых процессам;
- datatype - тип элементов во входном буфере;
- op - операция, по которой выполняется редукция;
- comm - коммунитор.

Выходные аргументы:

- recvbuf - адрес начала буфера приема.

Функция *MPI_Reduce_scatter* отличается от *MPI_Allreduce* тем, что результат операции разрезается на непересекающиеся части по числу процессов в группе, *i*-ая часть посылается *i*-ому процессу в его буфер приема. Длины этих частей задает третий параметр, являющийся массивом. На рис. 109 представлена графическая интерпретация операции *Reduce_scatter*.

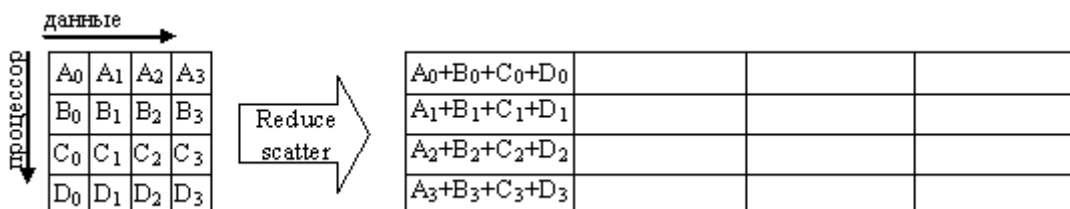


Рис. 109. Графическая интерпретация операции *Reduce_scatter*

Пример 5.

Рассмотрим применение данной функции при реализации следующего алгоритма:

- На 0-м процессе считывается значение переменной n (строка 12).
- С помощью функции широковещательной рассылки значение считанной переменной рассылается каждому процессу (строка 13).
- Каждый процесс объявляет массив $a[n*size]$ (строка 14) и выполняет присваивание элементам массива (строки 15-16).
- В строке 17 объявляется массив $b[n]$ для получения итоговых массивов.
- В строке 18 объявляется вспомогательный массив $rc[size]$.
- В строках 19-20 элементам вспомогательного массива присваивается n (для равномерного деления на блоки результирующего массива).
- В строке 21 вызывается функция `MPI_Reduce_scatter` для распределенного суммирования векторов a с последующим распределением равными блоками между процессами результата суммирования и сохранением полученных блоков в вектор b .
- Выполняется контрольный вывод полученного массива (строки 22-27).

Листинг программы приведен на рис. 110, скрин ее запуска – на рис.

111.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n,i,s=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    if(rank==0) scanf("%d",&n);
13.    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
14.    int *a=new int[n*size];
15.    for(i=0;i<n*size;i++)
16.        a[i]=rank+i;
17.    int *b=new int[n];
18.    int *rc=new int[size];
19.    for(i=0;i<size;i++)
20.        rc[i]=n;
21.    MPI_Reduce_scatter(a,b,rc,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
22.    for(i=0; i<100000000*rank; i++)
23.        s+=1;
24.    printf("rank= %d b: ",rank);
25.    for(i=0;i<n;i++)
26.        printf(" %d ", b[i]);
27.    printf("\n ");
28.    MPI_Finalize();
29.    return 0;
30. }
```

Рис. 110. Листинг программы

```

mc - stud@hpchead:~/M182/serg/11
[stud@hpchead 11]$ mpiexec -n 5 -f ./machines ./a.out
10
rank= 0  b:  10  15  20  25  30  35  40  45  50  55
rank= 1  b:  60  65  70  75  80  85  90  95  100 105
rank= 2  b: 110 115 120 125 130 135 140 145 150 155
rank= 3  b: 160 165 170 175 180 185 190 195 200 205
rank= 4  b: 210 215 220 225 230 235 240 245 250 255
[stud@hpchead 11]$

```

Рис. 111. Скрин запуска программы

Функция *MPI_Scan* выполняет префиксную редукцию. Аргументы такие же, как в *MPI_Allreduce*, но получаемые каждым процессом результаты отличаются друг от друга. Операция пересылает в буфер приема *i*-го процесса редукцию значений из входных буферов процессов с номерами 0, ... *i* включительно.

int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

Входные аргументы:

- sendbuf - адрес начала входного буфера;
- count - число элементов во входном буфере;
- datatype - тип элементов во входном буфере;
- op - операция, по которой выполняется редукция;
- comm - Коммуникатор.

Выходные аргументы:

- recvbuf - адрес начала буфера приема.

На рис. 112 представлена графическая интерпретация операции Scan.

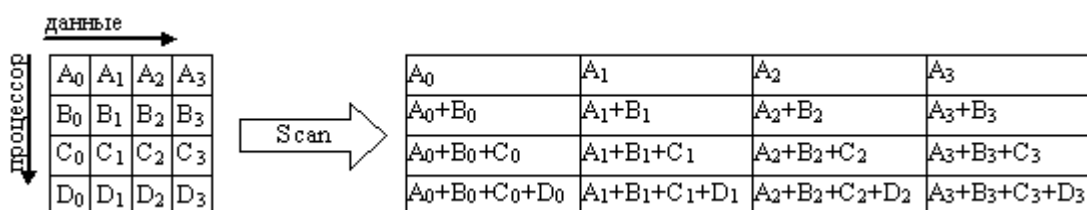


Рис. 112. Графическая интерпретация операции Scan

Пример 6.

Рассмотрим пример, аналогичный примеру 3, но используем вместо функции *MPI_Reduce* функцию *MPI_Scan*.

Листинг программы приведен на рис. 113, скрин ее запуска – на рис. 114. На 0-м процессе мы получили его изначальный вектор, на 1-м процессе – результат суммирования векторов 0-го и 1-го процессов, на 2-м процессе – результат суммирования векторов 0-го, 1-го и 2-го процессов и т.д.

```

1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n,i,s=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    if(rank==0) scanf("%d",&n);
13.    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
14.    int *a=new int[n];
15.    for(i=0;i<n;i++)
16.        a[i]=rank;
17.    int *b=new int[n];
18.    MPI_Scan(a,b,n,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
19.    for(i=0; i<100000000*rank; i++)
20.        s+=1;
21.    printf("rank= %d b: ",rank);
22.    for(i=0;i<n;i++)
23.        printf(" %d ", b[i]);
24.    printf("\n ");
25.    MPI_Finalize();
26.    return 0;
27. }

```

Рис. 113. Листинг программы

```

mc - stud@hpchead:~/M182/serg/11
[stud@hpchead 11]$ mpirun -n 5 -f ./machines ./a.out
10
rank= 0 b: 0 0 0 0 0 0 0 0 0 0
rank= 1 b: 1 1 1 1 1 1 1 1 1 1
rank= 3 b: 6 6 6 6 6 6 6 6 6 6
rank= 2 b: 3 3 3 3 3 3 3 3 3 3
rank= 4 b: 10 10 10 10 10 10 10 10 10 10
[stud@hpchead 11]$

```

Рис. 114. Скрин запуска программы

Задания

1. Создайте и выполните программу, используя распределенную вычислительную операцию (`MPI_Reduce`), реализующую следующий алгоритм: на каждом процессоре с помощью датчика случайных чисел генерируется целочисленное значение от 0 до 100, из этих значений выбирается максимальное и сохраняется в памяти нулевого процессора.
2. Используя функцию `MPI_Reduce_scatter`, напишите следующую программу: на 0-м процессе считывается значение переменной `n` и рассылается каждому; каждый процесс объявляет массив `a[2*n*size]` и присваивает его элементам значения; каждый процесс объявляет массив `b[2*n]`; применяя функцию `MPI_Reduce_scatter`, результат суммирования

распределяется по процессам равными блоками $2 \cdot n$ элементов в каждом с сохранением в массив b .

3. Модифицируйте предыдущую программу таким образом, чтобы результат был распределен следующим образом: на 0-м процессе – $b[0]$, на 1-м процессе – $b[1]$, $b[2]$, на 2-м процессе – три следующих элемента результирующего вектора и т.д.
4. Создайте последовательную программу, реализующую алгоритм скалярного произведения векторов. Размер векторов возьмите $n=(10^3 \dots 10^9)$. Элемент 1-го вектора задайте $(1/(1+i))$, второго – $(i/(1+i))$. Создайте параллельную программу, реализующую данный алгоритм. Для реализации используйте изученные коллективные коммуникационные функции. Определите время выполнения последовательной и параллельной программ в зависимости от размера массивов (запустите несколько раз и возьмите наилучшие результаты). Параллельную программу запустите на 2-х и 4-х процессах. Требуется представить полученные значения в табличном виде, привести скрин компиляции и запуска параллельной программы. Получите теоретическую оценку ускорения параллельного алгоритма и сравните с полученными результатами программ. Сделайте выводы о целесообразности распараллеливания данного алгоритма.
5. Напишите программу, используя изученные функции, реализующую алгоритм умножения матрицы на вектор. Размерность массива вводится при выполнении программы, элементы массива формируются на 0-ом процессоре, затем распределяются по процессорам, каждый процессор получает промежуточный результат и пересылает его на нулевой процессор, нулевой процессор производит окончательное формирование результата и выводит его. При выполнении данного упражнения можно использовать строчное или столбцовое распределение двумерного массива по процессорам. Определите время выполнения последовательной программы и параллельной в зависимости от размерности массива.

12. Производные типы данных: простейшие конструкторы

В процессе создания параллельных приложений часто возникает необходимость передачи данных, имеющих нестандартную структуру. Например, передача в одной посылке данных различных типов или данных, расположенных в несмежных областях оперативной памяти. В качестве первого способа решения указанной задачи в стандарте MPI предлагается использовать механизм передачи упакованных данных, однако существенным недостатком такого подхода являются временные затраты, необходимые для выполнения операции копирования данных из буфера в буфер на принимающей и передающей стороне, а также дополнительная утилизация оперативной памяти для размещения результата упаковки. Вместо этого MPI обеспечивает механизм для описания более общих буферов оперативной памяти (в том числе несмежных) путем создания карт размещения данных в оперативной памяти (в литературе часто именуют производными типами данных). В данном пункте рассматриваются простейшие конструкторы типов данных, а также сценарий их использования для передачи данных.

12.1. Карты размещения данных и сценарий их использования

Производные типы MPI не являются в полном смысле типами данных, как это понимается в языках программирования. Они могут использоваться только в коммуникационных операциях. Производные типы MPI следует понимать как описатели расположения в памяти элементов базовых типов. **Общий тип** данных, принятый в MPI, определяется двумя характеристиками:

- последовательностью базовых типов данных;
- последовательностью смещений.

Последовательность таких пар называется картой типов (*type map*), а последовательность базовых типов ($type_0, type_1, \dots, type_{n-1}$) – сигнатурой типа.

Карта типа обозначается в виде:

$$Type_{map} = \{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}.$$

Значения смещений не обязательно должны быть неотрицательными, различными и упорядоченными по возрастанию.

Карта типа вместе с адресом *buf* описывает коммуникационный буфер, который состоит из *n* элементов, где *i*-ый элемент расположен по адресу $buf + disp_i$ и имеет тип $type_i$. Сообщение, полученное из такого коммуникационного буфера, будет состоять из *n* значений с типами, определенными в сигнатуре. Базовые типы данных являются частным случаем общего типа, например, тип MPI_INT описывается картой $\{(int, 0)\}$.

Экстент (extent) типа данных определяется как адресное пространство, от первого байта до последнего байта, занятое элементами в типе данных, округленное вверх с учетом требований выравнивания данных. Для вычисления протяженности производного типа используется понятие верхней *ub* и нижней *lb* границы карты типа, определяемые следующим образом:

$$lb(\text{Typemap}) = \min_j(\text{disp}_j),$$

$$ub(\text{Typemap}) = \max_j(\text{disp}_j + \text{sizeof}(\text{type}_j)) + \varepsilon,$$

где ε – наименьшее неотрицательное число, необходимое для выравнивания типа данных в памяти. Тогда экстент равен значению $\text{extent} = ub - lb$. Использование производного типа в функциях обмена сообщениями можно рассматривать как трафарет, наложенный на область памяти, которая содержит передаваемое или принятое сообщение.

Стандартный сценарий определения и использования производных типов включает следующие шаги:

- Производный тип строится из предопределенных типов MPI и ранее определенных производных типов с помощью специальных функций-конструкторов.
- Новый производный тип регистрируется вызовом функции ***MPI_Type_commit***. Только после этого новый производный тип может использоваться в коммуникационных функциях и при конструировании других типов.
- Когда производный тип становится ненужным, он уничтожается вызовом функции ***MPI_Type_free***.

Для регистрации производного типа данных и возможности его дальнейшего использования в программе необходимо вызвать функцию ***MPI_Type_commit***, передав в нее имя нового типа.

int MPI_Type_commit (MPI_Datatype *datatype)

Входные аргументы:

datatype – новый производный тип данных.

Выходные аргументы:

datatype – новый производный тип данных.

Функция ***MPI_Type_free*** помечает объекты, связанные с производным типом данных, на уничтожение и присваивает описателю типа значение MPI_TYPE_NULL. Все коммуникационные операции, использующие производный тип, завершатся корректно, а все типы, определенные с помощью освобождаемого производного типа, останутся нетронутыми.

int MPI_Type_free (MPI_Datatype *datatype)

Входные аргументы:

Datatype – уничтожаемый производный тип данных.

Выходные аргументы:

Datatype - уничтожаемый производный тип данных.

Функция *MPI_Type_extent* определяет протяженность элемента некоторого типа.

```
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
```

Входные аргументы:

datatype - тип данных.

Выходные аргументы:

extent - протяженность элемента заданного типа.

Функция *MPI_Type_size* определяет "чистый" размер элемента некоторого типа (за вычетом пустых промежутков).

```
int MPI_Type_size(MPI_Datatype datatype, int *size)
```

Входные аргументы:

datatype - тип данных.

Выходные аргументы:

size - размер элемента заданного типа.

12.2. Конструктор описания карты непрерывного расположения блоков

Самый простой конструктор производного типа *MPI_Type_contiguous* создает новый тип, элементы которого состоят из указанного числа элементов базового типа, занимающих смежные области памяти.

```
int MPI_Type_contiguous (int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Входные аргументы:

count - число элементов базового типа;

oldtype - базовый тип данных;

Выходные аргументы:

newtype - новый производный тип данных.

Базовый тип может быть зарегистрированным ранее производным типом данных.

Графическая интерпретация работы конструктора *MPI_Type_contiguous* приведена на рисунке 115.

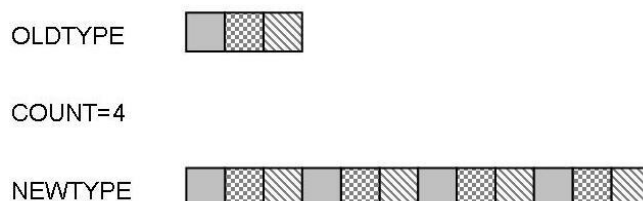


Рис. 115. Графическая интерпретация работы конструктора *MPI_Type_contiguous*

Пример 1.

Рассмотрим применение конструктора `MPI_Type_contiguous` на примере передачи массива с 0-го процесса на 1-й:

- Строка 12 – выделение памяти под одномерный массив `a[n]`.
- Строка 15 – объявление нового описателя типа `mt`.
- Строка 16 – конструирование нового типа, состоящего из `n` элементов типа `MPI_INT`.
- Строка 17 – регистрация созданного типа.
- Строки 20-21 – присвоение значения элементам массива на 0-м процессе.
- Строка 22 – отправка массива `a` 1-му процессу. Обратите внимание на 2-3 аргументы функции: отправляется 1 элемент типа `mt`, в котором содержится `n` элементов типа `MPI_INT`.
- Строка 26 прием массива по той же карте на 1-м процессе.
- Выполняется контрольный вывод полученного массива (строки 27-31).

Листинг программы приведен на рис. 116, скрин ее запуска – на рис. 117. В данном примере можно было отправить 1 элемент типа `mt`, а принять `n` элементов типа `MPI_INT`, так как они эквивалентны!

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=20,i;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int *a = new int [n];
13.    for(i=0;i<n;i++)
14.        a[i]=-1;
15.    MPI_Datatype mt;
16.    MPI_Type_contiguous(n,MPI_INT,&mt);
17.    MPI_Type_commit(&mt);
18.    if(rank==0)
19.    {
20.        for(i=0;i<n;i++)
21.            a[i]=i;
22.        MPI_Send(a,1,mt,1,777,MPI_COMM_WORLD);
23.    }
24.    if(rank==1)
25.    {
26.        MPI_Recv(a,1,mt,0,777,MPI_COMM_WORLD,&stat);
27.        printf("rank= %d a: ",rank);
28.        for(i=0;i<n;i++)
29.            printf("%d ",a[i]);
30.        printf("\n");
31.    }
32.    MPI_Finalize();
33.    return 0;
34. }
```

Рис. 116. Листинг программы передачи данных

```

mc - stud@hpchead:~/M182/serg/12
[stud@hpchead 12]$ mpiexec -n 2 -f ./machines ./a.out
rank= 1 a: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
[stud@hpchead 12]$

```

Рис. 117. Скрин запуска программы

12.3. Конструктор описания карты расположения одинаковых блоков с равными промежутками

Конструктор `MPI_Type_vector` создает производный тип, элементами которого являются блоки, удаленные между собой на расстояние, кратное размеру экстенда (протяженности) базового типа. Блоки состоят из одинакового числа смежных элементов базового типа. Как и в предыдущем конструкторе, базовый тип может быть ранее зарегистрированным производным типом.

int MPI_Type_vector (int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype)

Входные аргументы:

- count - количество блоков;
- blocklength - число элементов базового типа в каждом блоке;
- stride - число элементов между началами соседних блоков;
- oldtype - базовый тип данных;

Выходные аргументы:

newtype - новый производный тип данных.

Например, пусть тип `oldtype` имеет карту $\{(double,0),(char,8)\}$ с размером экстенда 16. Тогда вызов функции `MPI_Type_vector` (2, 3, 4, `oldtype`, `newtype`) создаст новый тип `newtype` с картой следующего вида:

$\{(double,0),(char,8),(double,16),(char,24),(double,32),(char,40),$
 $(double,64),(char,72),(double,80),(char,88),(double,96),(char,104)\}.$

В результате создается новый тип данных из двух блоков, каждый блок состоит из трех элементов базового типа, и расстояние между блоками равно $4 \cdot 16$ байт. Данный пример проиллюстрирован на рисунке 118.

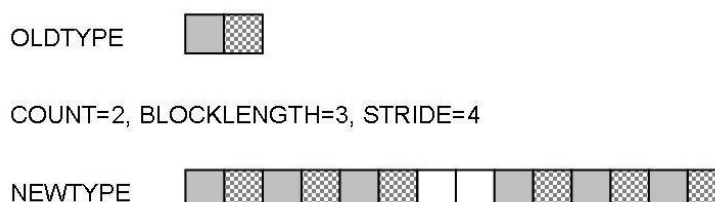


Рис. 118. Графическая интерпретация работы конструктора `MPI_Type_vector`

Пример 2.

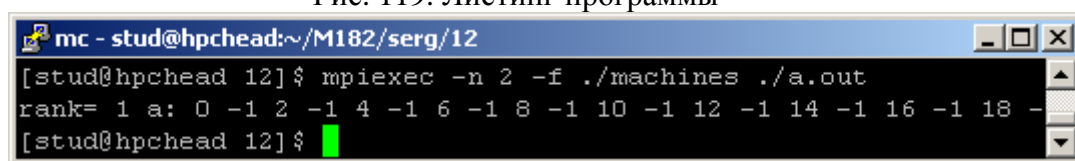
Рассмотрим применение конструктора `MPI_Type_vector` на примере передачи четных элементов массива с 0-го процесса на 1-й:

- Строка 16 – определение нового типа `mt`, в котором $n/2$ блоков, в каждом блоке 1 актуальный элемент, длина блока 2 элемента, базовый типа данных – `MPI_INT`.
- Строка 17 – регистрация созданного типа.
- Строка 22 – отправка массива `a` 1-му процессу. Обратите внимание на 2-3 аргументы функции: отправляется 1 элемент типа `mt`, в котором содержится $n/2$ элементов типа `MPI_INT`.

Листинг программы приведен на рис. 119, скрин ее запуска – на рис. 120. На скрине выведен массив на 1-м процессе после получения четных элементов, нечетные элементы равны (-1) (строка 14).

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=20,i;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int *a = new int [n];
13.    for(i=0;i<n;i++)
14.        a[i]=-1;
15.    MPI_Datatype mt;
16.    MPI_Type_vector(n/2,1,2,MPI_INT,&mt);
17.    MPI_Type_commit(&mt);
18.    if(rank==0)
19.    {
20.        for(i=0;i<n;i++)
21.            a[i]=i;
22.        MPI_Send(a,1,mt,1,777,MPI_COMM_WORLD);
23.    }
24.    if(rank==1)
25.    {
26.        MPI_Recv(a,1,mt,0,777,MPI_COMM_WORLD,&stat);
27.        printf("rank= %d a: ",rank);
28.        for(i=0;i<n;i++)
29.            printf("%d ",a[i]);
30.        printf("\n");
31.    }
32.    MPI_Finalize();
33.    return 0;
34. }
```

Рис. 119. Листинг программы



```
mc - stud@hpchead:~/M182/serg/12
[stud@hpchead 12]$ mpirun -n 2 -f ./machines ./a.out
rank= 1 a: 0 -1 2 -1 4 -1 6 -1 8 -1 10 -1 12 -1 14 -1 16 -1 18 -
[stud@hpchead 12]$
```

Рис. 120. Скрин запуска программы

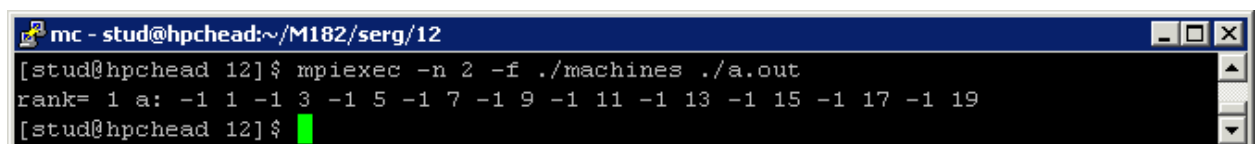
Пример 3.

Что требуется изменить в предыдущем примере, чтобы передать нечетные элементы? Карта размещения данных в оперативной памяти никак не изменится – нужно передавать один элемент с шагом (длиной блока) 2 элемента. Достаточно изменить первый аргумент в функциях передачи данных (строки 22, 26). В примере целенаправленно использованы два разных обозначения требуемого адреса.

Листинг программы приведен на рис. 121, скрин ее запуска – на рис. 122. На скрине выведен массив на 1-м процессе после получения нечетных элементов, четные элементы равны (-1) (строка 14).

```
35. #include <stdio.h>
36. #include "mpi.h"
37. int main(int argc, char *argv[])
38. {
39.     int rank;
40.     int size;
41.     int n=20,i;
42.     MPI_Status stat;
43.     MPI_Init(&argc, &argv);
44.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
45.     MPI_Comm_size(MPI_COMM_WORLD, &size);
46.     int *a = new int [n];
47.     for(i=0;i<n;i++)
48.         a[i]=-1;
49.     MPI_Datatype mt;
50.     MPI_Type_vector(n/2,1,2,MPI_INT,&mt);
51.     MPI_Type_commit(&mt);
52.     if(rank==0)
53.     {
54.         for(i=0;i<n;i++)
55.             a[i]=i;
56.         MPI_Send(a+1,1,mt,1,777,MPI_COMM_WORLD);
57.     }
58.     if(rank==1)
59.     {
60.         MPI_Recv(&a[1],1,mt,0,777,MPI_COMM_WORLD,&stat);
61.         printf("rank= %d a: ",rank);
62.         for(i=0;i<n;i++)
63.             printf("%d ",a[i]);
64.         printf("\n");
65.     }
66.     MPI_Finalize();
67.     return 0;
68. }
```

Рис. 121. Листинг программы



```
mc - stud@hpchead:~/M182/serg/12
[stud@hpchead 12]$ mpiexec -n 2 -f ./machines ./a.out
rank= 1 a: -1 1 -1 3 -1 5 -1 7 -1 9 -1 11 -1 13 -1 15 -1 17 -1 19
[stud@hpchead 12]$
```

Рис. 122. Скрин запуска программы

Пример 4.

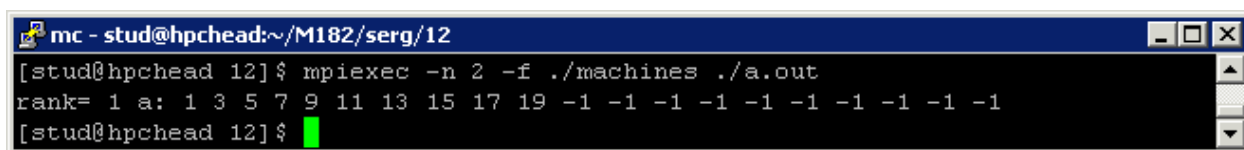
Теперь попытаемся ответить на следующие вопросы: сколько элементов передается и что произойдет, если принять их, не используя карту, которая применялась при отправке?

Очевидно, что передается $(n/2)$ элементов типа `MPI_INT`, которые мы можем принять как угодно – используя карту, содержащую $(n/2)$ элементов, или сразу, записывая в массив $(n/2)$ элементов подряд.

На рис. 123 приведен листинг программы, которая реализует отправку нечетных элементов с 0-го процесса, а прием на 1-м осуществляется подряд всех присланных элементов. На скрине выведен массив на 1-м процессе после получения нечетных элементов с записью подряд в начало массива, остальные элементы равны (-1).

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=20,i;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int *a = new int [n];
13.    for(i=0;i<n;i++)
14.        a[i]=-1;
15.    MPI_Datatype mt;
16.    MPI_Type_vector(n/2,1,2,MPI_INT,&mt);
17.    MPI_Type_commit(&mt);
18.    if(rank==0)
19.    {
20.        for(i=0;i<n;i++)
21.            a[i]=i;
22.        MPI_Send(a+1,1,mt,1,777,MPI_COMM_WORLD);
23.    }
24.    if(rank==1)
25.    {
26.        MPI_Recv(a,n/2,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
27.        printf("rank= %d a: ",rank);
28.        for(i=0;i<n;i++)
29.            printf("%d ",a[i]);
30.        printf("\n");
31.    }
32.    MPI_Finalize();
33.    return 0;
34. }
```

Рис. 123. Листинг программы



```
mc - stud@hpchead:~/M182/serg/12
[stud@hpchead 12]$ mpirun -n 2 -f ./machines ./a.out
rank= 1 a: 1 3 5 7 9 11 13 15 17 19 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
[stud@hpchead 12]$
```

Рис. 124. Скрин запуска программы

Пример 5.

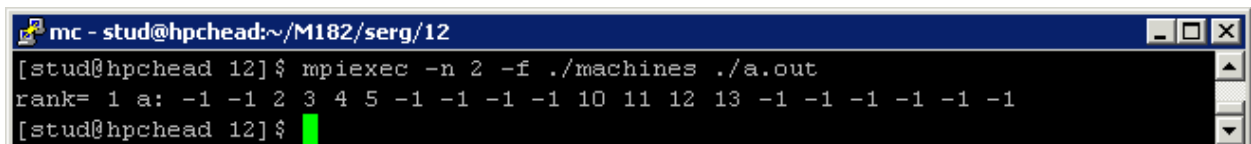
Выполним пересылку элементов массива, отмеченных на следующей схеме:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

Изменились аргументы конструктора (строка 16), а также адреса для отправки (MPI_Send, строка 22) и приема сообщения (MPI_Recv, строка 26).

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=20,i;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int *a = new int [n];
13.    for(i=0;i<n;i++)
14.        a[i]=-1;
15.    MPI_Datatype mt;
16.    MPI_Type_vector(2,4,8,MPI_INT,&mt);
17.    MPI_Type_commit(&mt);
18.    if(rank==0)
19.    {
20.        for(i=0;i<n;i++)
21.            a[i]=i;
22.        MPI_Send(a+2,1,mt,1,777,MPI_COMM_WORLD);
23.    }
24.    if(rank==1)
25.    {
26.        MPI_Recv(a+2,1,mt,0,777,MPI_COMM_WORLD,&stat);
27.        printf("rank= %d a: ",rank);
28.        for(i=0;i<n;i++)
29.            printf("%d ",a[i]);
30.        printf("\n");
31.    }
32.    MPI_Finalize();
33.    return 0;
34. }
```

Рис. 125. Листинг программы



```
mc - stud@hpchead:~/M182/serg/12
[stud@hpchead 12]$ mpirun -n 2 -f ./machines ./a.out
rank= 1 a: -1 -1 2 3 4 5 -1 -1 -1 -1 10 11 12 13 -1 -1 -1 -1 -1 -1
[stud@hpchead 12]$
```

Рис. 126. Скрин запуска программы

Пример 6.

Выполним пересылку элементов массива, отмеченных на следующей схеме:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Для начала построим карту, включающую два элемента массива – 0 и 2 (строка 16: 2 блока, по 1 актуальному элементу в каждом блоке, длина блока – 2 элемента). Затем, при использовании функций MPI_Send-MPI_Recv, укажем отправку двух элементов типа mt. Почему же при таких аргументах не отправились элементы 0, 2, 4, 6? Это происходит потому, что верхняя граница типа определена последним актуальным элементом. В нашем случае 1 элемент mt заканчивается на 2-м элементе массива, а следующий элемент mt начинается с 3-го элемента массива.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=20,i;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int *a = new int [n];
13.    for(i=0;i<n;i++)
14.        a[i]=-1;
15.    MPI_Datatype mt;
16.    MPI_Type_vector(2,1,2,MPI_INT,&mt);
17.    MPI_Type_commit(&mt);
18.    if(rank==0)
19.    {
20.        for(i=0;i<n;i++)
21.            a[i]=i;
22.        MPI_Send(a,2,mt,1,777,MPI_COMM_WORLD);
23.    }
24.    if(rank==1)
25.    {
26.        MPI_Recv(a,2,mt,0,777,MPI_COMM_WORLD,&stat);
27.        printf("rank= %d a: ",rank);
28.        for(i=0;i<n;i++)
29.            printf("%d ",a[i]);
30.        printf("\n");
31.    }
32.    MPI_Finalize();
33.    return 0;
34. }
```

Рис. 127. Листинг программы

Пример 7.

В данном примере продемонстрируем использование уже созданного типа при конструировании следующего.

Выполним пересылку элементов массива, отмеченных на следующей схеме:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=20,i;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int *a = new int [n];
13.    for(i=0;i<n;i++)
14.        a[i]=-1;
15.    MPI_Datatype mt,mt1,mt2;
16.    MPI_Type_vector(2,1,2,MPI_INT,&mt);
17.    MPI_Type_commit(&mt);
18.    MPI_Type_contiguous(2,mt,&mt1);
19.    MPI_Type_commit(&mt1);
20.    MPI_Type_vector(2,1,2,mt1,&mt2);
21.    MPI_Type_commit(&mt2);
22.    if(rank==0)
23.    {
24.        for(i=0;i<n;i++)
25.            a[i]=i;
26.        MPI_Send(a,1,mt2,1,777,MPI_COMM_WORLD);
27.    }
28.    if(rank==1)
29.    {
30.        MPI_Recv(a,1,mt2,0,777,MPI_COMM_WORLD,&stat);
31.        printf("rank= %d a: ",rank);
32.        for(i=0;i<n;i++)
33.            printf("%d ",a[i]);
34.        printf("\n");
35.    }
36.    MPI_Finalize();
37.    return 0;
38. }
```

Рис. 128. Листинг программы

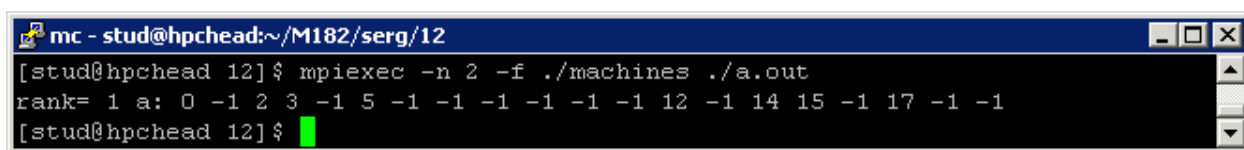
Для начала построим карту, включающую два элемента массива – 0 и 2 (строка 16: 2 блока, по 1 актуальному элементу в каждом блоке, длина блока – 2 элемента, базовый тип MPI_INT).

Затем, используя конструктор непрерывного расположения данных в оперативной памяти, соединим два блока `mt` в один `mt1` (строка 18). Получим блок, содержащий 6 элементов `MPI_INT`: с 0-го по 5.

Далее все просто (см. схему выше): нужно создать карту, в которую войдут два четных одинаковых блока (помечены желтым) с пропуском такого же по размеру красного блока (строка 20, в качестве базового типа указан `mt1`).

Осталось лишь при использовании функций `MPI_Send-MPI_Recv` указать отправку-получение элемента с типом `mt2`. На скрине (рис. 129) видно, что передача запланированных элементов прошла успешно.

Следует отметить, что на практике вряд ли кто будет использовать такой подход для создания карты. Для сложного расположения данных используются более универсальные конструкторы, которые будут рассмотрены позже. Этот пример мы выполним, изучая конструктор `MPI_Type_indexed`, и Вы убедитесь, что им пользоваться для такого нетривиального случая гораздо удобнее. Здесь же приведен такой пример с одной только целью – продемонстрировать использование уже созданного и зарегистрированного типа при конструировании другого.



```
mc - stud@hpchead:~/M182/serg/12
[stud@hpchead 12]$ mpirun -n 2 -f ./machines ./a.out
rank= 1 a: 0 -1 2 3 -1 5 -1 -1 -1 -1 -1 12 -1 14 15 -1 17 -1 -1
[stud@hpchead 12]$
```

Рис. 129. Скрин запуска программы

Задания

1. Требуется провести исследование по способам передачи данных четных элементов одномерного массива. Для этого напишите несколько программ передачи данных с 0-го процесса на 1-й и обратно с замером времени на 0-м процессе перед отправкой и после получения. Размер передаваемого массива должен быть значительным ($10^3 \dots 10^9$), чтобы можно было пренебречь погрешностью измерения времени. 1-я программа: массив передается целиком одной посылкой. 2-я программа: передаются в цикле по одному элементу только четные элементы. 3-я программа: перед отправкой четные элементы переписываются в промежуточный массив половинной длины, он передается и после получения снова переписывается в исходный массив. 4-я программа: при помощи конструктора `MPI_Type_vector` создается новый описатель расположения данных в оперативной памяти, передача осуществляется с его помощью. Результат измерения времени при разных вариантах передачи данных приведите в таблице. Укажите плюсы и минусы каждого из рассмотренных подходов.
2. На основе рассмотренных примеров выполните пересылку элементов массива, отмеченных на схеме:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

3. Выполните пересылку элементов массива, отмеченных на схеме:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

4. Выполните пересылку элементов массива, отмеченных на схеме:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

5. Выполните пересылку элементов массива, отмеченных на схеме:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

6. Выполните пересылку элементов массива, отмеченных на схеме:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

7. Напишите программу, которая осуществляет рассылку одномерного массива а слоистым образом по процессам. Например, если программу запустить на 3-х процессах, то 0-й процесс должен отправить 1-му процессу следующие элементы: 1, 4, 7 и т.д., а 2-му 2, 5, 8 и т.д. Все элементы, отправляемые очередному процессу, необходимо собрать в единую посылку с использованием конструктора `MPI_Type_vector`. Программа должна быть универсальной относительно количества процессов и размера массива для рассылки.

13. Производные типы данных: работа с двумерными массивами

13.1. Конструктор описания карты расположения одинаковых блоков с байтовым смещением между началами блоков

Пример 1. Передача столбца двумерного массива

Начнем с рассмотрения примера, в котором продемонстрируем использование уже изученного конструктора `MPI_Type_vector` для передачи столбца элементов двумерного массива. Для передачи строки ничего дополнительно делать не требуется, так как элементы строки расположены непрерывно в оперативной памяти и могут быть отправлены с использованием стандартных описателей типа. Например, для отправки 1-й строки двумерного массива `b` первые три аргумента коммуникационной функции будут следующие: `(&b[0][0], n, MPI_INT, ...`

Элементы столбца расположены в оперативной памяти через равные промежутки, составляющие длину строки. В связи с этим работа с конструктором `MPI_Type_vector` аналогична при его применении к одномерным массивам.

Разберем программу, приведенную на рис. 130.

В строке 13 строим карту, состоящую из (n) блоков с 1 актуальным элементом в каждом блоке и длиной блока (n) элементов типа `MPI_INT`. Под блоком здесь подразумевается строка массива `b[n][n]`.

Строки 15-18 – выделение памяти под двумерный динамический массив, строчным образом расположенный в оперативной памяти.

Строки 19-21 – обнуление элементов массива.

Строки 24-26 – присваивание значений элементам массива на 0-м процессе.

Строка 27 – отправка 1 элемента типа `mt` (описывает столбец), начиная с начала массива.

Строка 31 – прием в начало массива присланных элементов также по карте `mt`.

Далее – контрольный вывод изначального массива на 0-м процессе и получившегося после приема на 1-м процессе.

На рис. 131 приведен скрин запуска данной программы, демонстрирующий успешную передачу 1-го столбца массива.

Давайте ответим на вопрос: что требуется изменить в программе, чтобы отправлять с 0-го процесса столбец, а принимать на 1-м процессе его в строку? Для этого достаточно всего лишь изменить аргументы функции приема сообщения (строка 31): `MPI_Recv(*b,n,MPI_INT...`

Это допустимо, так как по карте mt с 0-го процесса отправляются одной посылкой (n) элементов столбца, а при приеме мы просто их разместили подряд в оперативной памяти, тем самым, заполнив строку. Результат запуска измененной программы приведен на рис. 132.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=9,i,j,s=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    MPI_Datatype mt;
13.    MPI_Type_vector(n,1,n,MPI_INT,&mt);
14.    MPI_Type_commit(&mt);
15.    int **b=new int *[n];
16.    b[0]=new int[n*n];
17.    for(i=1;i<n;i++)
18.        b[i]=b[i-1]+n;
19.    for(i=0;i<n;i++)
20.        for(j=0;j<n;j++)
21.            b[i][j]=0;
22.    if(rank==0)
23.    {
24.        for(i=0;i<n;i++)
25.            for(j=0;j<n;j++)
26.                b[i][j]=10*(i+1)+j+1;
27.        MPI_Send(*b,1,mt,1,777,MPI_COMM_WORLD);
28.    }
29.    if(rank==1)
30.    {
31.        MPI_Recv(*b,1,mt,0,777,MPI_COMM_WORLD,&stat);
32.    }
33.    for(i=0; i<100000000*rank; i++)
34.        s+=1;
35.    printf("rank= %d b: \n",rank);
36.    for(i=0; i<n; i++)
37.    {
38.        for(j=0;j<n;j++)
39.            printf(" %d ",b[i][j]);
40.        printf("\n");
41.    }
42.    MPI_Finalize();
43.    return 0;
44. }
```

Рис. 130. Листинг программы

```
mc - stud@hpchead:~/M182/serg/13
[stud@hpchead 13]$ mpiexec -n 2 -f ./machines ./a.out
rank= 0  b:
 11 12 13 14 15 16 17 18 19
 21 22 23 24 25 26 27 28 29
 31 32 33 34 35 36 37 38 39
 41 42 43 44 45 46 47 48 49
 51 52 53 54 55 56 57 58 59
 61 62 63 64 65 66 67 68 69
 71 72 73 74 75 76 77 78 79
 81 82 83 84 85 86 87 88 89
 91 92 93 94 95 96 97 98 99
rank= 1  b:
 11 0 0 0 0 0 0 0 0
 21 0 0 0 0 0 0 0 0
 31 0 0 0 0 0 0 0 0
 41 0 0 0 0 0 0 0 0
 51 0 0 0 0 0 0 0 0
 61 0 0 0 0 0 0 0 0
 71 0 0 0 0 0 0 0 0
 81 0 0 0 0 0 0 0 0
 91 0 0 0 0 0 0 0 0
[stud@hpchead 13]$
```

Рис. 131. Скрин запуска программы

```
mc - stud@hpchead:~/M182/serg/13
[stud@hpchead 13]$ mpiexec -n 2 -f ./machines ./a.out
rank= 0  b:
 11 12 13 14 15 16 17 18 19
 21 22 23 24 25 26 27 28 29
 31 32 33 34 35 36 37 38 39
 41 42 43 44 45 46 47 48 49
 51 52 53 54 55 56 57 58 59
 61 62 63 64 65 66 67 68 69
 71 72 73 74 75 76 77 78 79
 81 82 83 84 85 86 87 88 89
 91 92 93 94 95 96 97 98 99
rank= 1  b:
 11 21 31 41 51 61 71 81 91
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0
[stud@hpchead 13]$
```

Рис. 132. Скрин запуска программы

Теперь усложним задачу: требуется передать матрицу целиком и при передаче данных сохранить ее в транспонированном виде.

На первый взгляд нужно для отправки просто собрать (n) столбцов *mt* при помощи конструктора непрерывного расположения данных или с

помощью `MPI_Type_vector`, а при приеме указать прием $(n*n)$ элементов с типом `MPI_INT`. Например, на основе ранее созданного описателя столбца вызвать `MPI_Type_vector(n,1,???,mt,&mt1)`, где третий аргумент – шаг между началами соседних блоков (в нашем случае шаг между столбцами) должен быть (1) `MPI_INT`, но в данной функции мы можем оперировать только типом `mt`. Именно это обстоятельство препятствует использованию данного конструктора для решения поставленной задачи.

Конструктор *типа* `MPI_Type_hvector` расширяет возможности конструктора `MPI_Type_vector`, позволяя задавать произвольный шаг между началами блоков в байтах.

```
int MPI_Type_hvector(int count, int blocklength,  
MPI_Aint stride, MPI_Datatype oldtype, MPI_Datatype  
*newtype)
```

Входные аргументы:

<code>count</code>	- число блоков;
<code>blocklength</code>	- число элементов базового типа в каждом блоке;
<code>stride</code>	- шаг между началами соседних блоков в байтах;
<code>oldtype</code>	- базовый тип данных.

Выходные аргументы:

<code>newtype</code>	- новый производный тип данных.
----------------------	---------------------------------

Определенный в стандарте MPI-1 конструктор типов `MPI_Type_hvector` (приставка `h` означает гетерогенный, неоднородный), расширяющий возможности `MPI_Type_vector`, в стандарте MPI-2 не рекомендован для использования так же, как и функции `MPI_Type_hindexed`, `MPI_Type_struct`, `MPI_Type_extent` и некоторые другие [25]. Сделано это по двум основным причинам:

- Функции стандарта MPI-1 используют целые значения (типа `integer`) для адресов и байтовых смещений. Это приводит к возникновению проблем в системах, которые используют 64-битную адресацию памяти. К тому же 32-разрядное значение для целых чисел не позволяет адресовать более 2 Гбайт прикладного адресного пространства. Даже если приложение использует менее 2 Гбайт памяти, то эта память состоит обычно из нескольких несмежных сегментов в виртуальной памяти.
- В MPI-1 поддерживается явное управление границами типов данных и экстенциями, обеспечиваемое двумя предопределенными типами данных `MPI_LB` и `MPI_UB`. В MPI-1 нет механизма для стирания маркеров верхней и нижней границ.

Ниже дается список устаревших функций и констант для работы с производными типами MPI-1 и приводится их замена. Как обычно, устаревшие функции продолжают оставаться частью стандарта MPI-2, однако пользователю настоятельно рекомендуется использовать новые функции везде, где это возможно.

Таблица 5: Заменяемые функции

Нерекомендуемые функции	Замена в MPI-2
MPI_Address	MPI_Get_addresses
MPI_Type_hindexed	MPI_Type_create_hindexed
MPI_Type_hvector	MPI_Type_create_hvector
MPI_Type_struct	MPI_Type_create_struct
MPI_Type_extnt	MPI_Type_get_extnt

При этом изменения касаются только названий конструкторов, аргументы остаются прежними.

Например, конструктор *MPI_Type_create_hvector* имеет следующий синтаксис (видно, что они такие же, как и у MPI_Type_hvector):

```
int MPI_Type_create_hvector (int count, int  
blocklength, MPI_Aint stride, MPI_Datatype oldtype,  
MPI_Datatype *newtype)
```

Входные аргументы:

count - количество блоков;

blocklength - число элементов базового типа в каждом блоке;

stride - количество байт между началами соседних блоков;

oldtype - базовый тип данных;

Выходные аргументы:

newtype - новый производный тип данных.

```
mc - stud@hpchead:~/M182/serg/13
[stud@hpchead 13]$ mpirun -n 2 -f ./machines ./a.out
rank= 0  b:
 11 12 13 14 15 16 17 18 19
 21 22 23 24 25 26 27 28 29
 31 32 33 34 35 36 37 38 39
 41 42 43 44 45 46 47 48 49
 51 52 53 54 55 56 57 58 59
 61 62 63 64 65 66 67 68 69
 71 72 73 74 75 76 77 78 79
 81 82 83 84 85 86 87 88 89
 91 92 93 94 95 96 97 98 99
rank= 1  b:
 11 21 31 41 51 61 71 81 91
 12 22 32 42 52 62 72 82 92
 13 23 33 43 53 63 73 83 93
 14 24 34 44 54 64 74 84 94
 15 25 35 45 55 65 75 85 95
 16 26 36 46 56 66 76 86 96
 17 27 37 47 57 67 77 87 97
 18 28 38 48 58 68 78 88 98
 19 29 39 49 59 69 79 89 99
[stud@hpchead 13]$
```

Рис. 133. Скрин запуска программы

Пример 2. Передача матрицы с транспонированием

Разберем, какие изменения в программе из примера 1 пришлось выполнить для решения поставленной задачи (рис. 134).

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=9,i,j,s=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    MPI_Datatype mt,mt1;
13.    MPI_Type_vector(n,1,n,MPI_INT,&mt);
14.    MPI_Type_commit(&mt);
15.    MPI_Type_create_hvector(n,1,4,mt,&mt1);
16.    MPI_Type_commit(&mt1);
17.    int **b=new int *[n];
18.    b[0]=new int[n*n];
19.    for(i=1;i<n;i++)
20.        b[i]=b[i-1]+n;
21.    for(i=0;i<n;i++)
22.        for(j=0;j<n;j++)
23.            b[i][j]=0;
24.    if(rank==0)
25.    {
26.        for(i=0;i<n;i++)
27.            for(j=0;j<n;j++)
28.                b[i][j]=10*(i+1)+j+1;
29.    MPI_Send(*b,1,mt1,1,777,MPI_COMM_WORLD);
30.    }
31.    if(rank==1)
32.    {
33.    MPI_Recv(*b,n*n,MPI_INT,0,777,MPI_COMM_WORLD,&stat);
34.    }
35.    for(i=0; i<100000000*rank; i++)
36.        s+=1;
37.    printf("rank= %d b: \n",rank);
38.    for(i=0; i<n; i++)
39.    {
40.        for(j=0;j<n;j++)
41.            printf(" %d ",b[i][j]);
42.        printf("\n");
43.    }
44.    MPI_Finalize();
45.    return 0;
46. }
```

Рис. 134. Листинг программы

Строка 15 – на основе ранее созданной карты mt, описывающей столбец двумерного массива, создаем карту mt1, описывающую всю матрицу, составленную из столбцов mt.

Строка 27 – отправка 1 элемента типа mt1, начиная с начала массива.

Строка 31 – прием в начало массива присланных ($n*n$) элементов с типом `MPI_INT`.

Далее – контрольный вывод изначального массива на 0-м процессе и получившегося после приема на 1-м процессе.

На рис. 133 приведен скрин запуска данной программы, демонстрирующий успешную передачу двумерного массива с его транспонированием.

13.2. Конструктор описания карты расположения различных блоков с разными промежутками

Конструктор ***MPI_Type_indexed*** позволяет создавать тип данных, элементами которого являются произвольные по длине блоки на основе базового типа. Смещения между блоками могут быть произвольными, однако их значения должны быть кратны размеру экстенда базового типа.

```
int MPI_Type_indexed (int count, int  
array_of_blocklengths, int *array_of_displacements,  
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Входные аргументы:

`count` - количество блоков;

`array_of_blocklengths` - массив, хранящий число элементов базового типа в каждом блоке;

`array_of_displacements` - массив смещений каждого блока от начала размещения элемента нового типа, смещения кратны размеру экстенда базового типа;

`oldtype` - базовый тип данных.

Выходные аргументы:

`newtype` - новый производный тип данных.

Эта функция создает тип `newtype`, каждый элемент которого состоит из `count` блоков, где i -ый блок содержит `array_of_blocklengths[i]` элементов базового типа и смещен от начала размещения элемента нового типа на `array_of_displacements[i]*extent(oldtype)` байт. Например, пусть тип `oldtype` имеет карту $\{(double, 0), (char, 8)\}$ с размером экстенда 16. Пусть массивы $B = (3, 1)$ и $D = (4, 0)$. Тогда вызов функции `MPI_Type_indexed (2, B, D, oldtype, newtype)` создаст новый тип `newtype` с картой следующего вида:

$\{(double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104),$
 $(double, 0), (char, 8)\}.$

Конструктор типа ***MPI_Type_create_hindexed*** идентичен конструктору `MPI_Type_indexed` за исключением того, что смещения измеряются в байтах. На рисунке 135 приведена графическая интерпретация работы конструктора.

```
int MPI_Type_create_hindexed (int count, int
array_of_blocklengths[], MPI_Aint array_of_displacements[],
MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Входные аргументы:

count - количество блоков;
array_of_blocklengths - массив, хранящий число элементов базового типа в каждом блоке;
array_of_displacements - массив смещений каждого блока от начала размещения элемента нового типа, выраженный в байтах;
oldtype - базовый тип данных;

Выходные аргументы:

newtype - новый производный тип данных.

Например, тип данных oldtype имеет карту $\{(type_0, disp_0), \dots, (type_{n-1}, disp_{n-1})\}$. Пусть B – массив array_of_blocklengths, а D – массив смещений array_of_displacements. Тогда новый тип из n блоков будет иметь следующий вид:

$\{(type_0, disp_0 + D(0)), \dots, (type_{n-1}, disp_{n-1} + D(0))\},$
 $(type_0, disp_0 + D(0) + (B(0) - 1) \cdot ex), \dots, (type_{n-1}, disp_{n-1} + D(0) + (B(0) - 1) \cdot ex), \dots,$
 $(type_{n-1}, disp_{n-1} + D(n-1) + (B(n-1) - 1) \cdot ex)\}.$

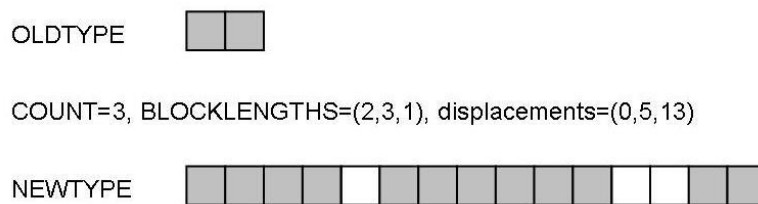


Рис. 135. Графическая интерпретация работы конструктора MPI_Type_hindexed

Пример 3. Передача элементов одномерного массива

Выполним пример 7 из предыдущего пункта с применением MPI_Type_indexed. Напомним условия: требовалось выполнить пересылку элементов массива, отмеченных на следующей схеме:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

Можно заметить, что в данной схеме 6 блоков, в которых содержится (1, 2, 1, 1, 2, 1) элементов, адреса блоков следующие – (0, 2, 5, 12, 14, 17). Этого достаточно для описания аргументов функции MPI_Type_indexed.

Разберем, какие изменения в программе пришлось выполнить для решения поставленной задачи (рис. 136).

Строки 16-17 – подготовка вспомогательных массивов для описания карты размещения данных в оперативной памяти.

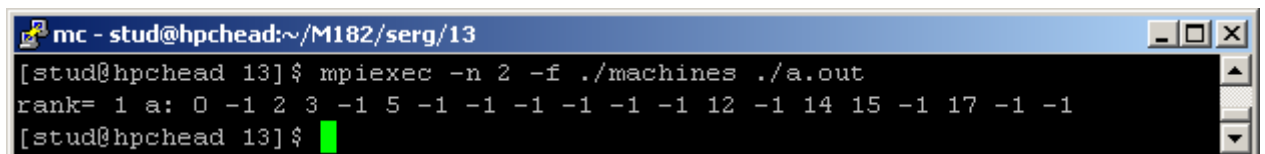
Строка 18 – создание карты, описывающей все необходимые для передачи данных элементы.

Остальное осталось без изменений.

На рис. 137 приведен скрин запуска данной программы, демонстрирующий успешную передачу требуемых элементов. Логика построения карты с более универсальным конструктором значительно упростилась!

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=20,i;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int *a = new int [n];
13.    for(i=0;i<n;i++)
14.        a[i]=-1;
15.    MPI_Datatype mt;
16.    int bl[] = {1,2,1,1,2,1};
17.    int ds[] = {0,2,5,12,14,17};
18.    MPI_Type_indexed(6,bl,ds,MPI_INT,&mt);
19.    MPI_Type_commit(&mt);
20.    if(rank==0)
21.    {
22.        for(i=0;i<n;i++)
23.            a[i]=i;
24.        MPI_Send(a,1,mt,1,777,MPI_COMM_WORLD);
25.    }
26.    if(rank==1)
27.    {
28.        MPI_Recv(a,1,mt,0,777,MPI_COMM_WORLD,&stat);
29.        printf("rank= %d a: ",rank);
30.        for(i=0;i<n;i++)
31.            printf("%d ",a[i]);
32.        printf("\n");
33.    }
34.    MPI_Finalize();
35.    return 0;
36. }
```

Рис. 136. Листинг программы



```
mc - stud@hpchead:~/M182/serg/13
[stud@hpchead 13]$ mpiexec -n 2 -f ./machines ./a.out
rank= 1 a: 0 -1 2 3 -1 5 -1 -1 -1 -1 -1 -1 12 -1 14 15 -1 17 -1 -1
[stud@hpchead 13]$
```

Рис. 137. Скрин запуска программы

Пример 4. Передача элементов одномерного массива

Выполним пересылку элементов массива, отмеченных на следующей схеме:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

Можно заметить, что в данной схеме 4 блока, в которых содержится (1, 2, 3, 4) элементов, адреса блоков следующие – (0, 2, 6, 12). Этого достаточно для описания аргументов функции `MPI_Type_indexed`.

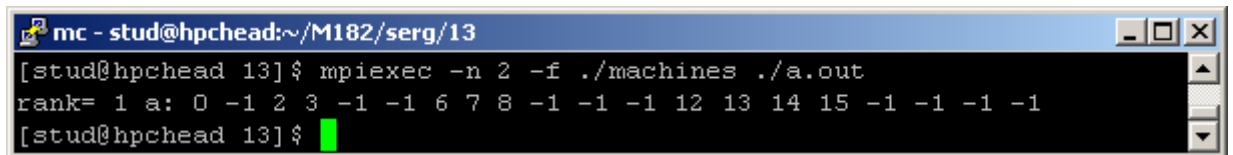
В программе, приведенной на рис. 138, возможно непонятна строка 21, в которой определена формула для определения адресов отправляемых блоков данных. Данная формула получается применением формулы удвоенной суммы членов арифметической прогрессии: 1 элемент берем, 1 пропускаем, 2 берем, 2 пропускаем и т.д.

Остальных изменений по сравнению с предыдущим примером нет.

На рис. 139 приведен скрин запуска данной программы, демонстрирующий успешную передачу требуемых элементов.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=20,i;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int *a = new int [n];
13.    for(i=0;i<n;i++)
14.        a[i]=-1;
15.    MPI_Datatype mt;
16.    int *bl=new int[5];
17.    int *ds=new int[5];
18.    for(i=0;i<5;i++)
19.    {
20.        bl[i]=i+1;
21.        ds[i]=i*(i+1);
22.    }
23.    MPI_Type_indexed(5,bl,ds,MPI_INT,&mt);
24.    MPI_Type_commit(&mt);
25.    if(rank==0)
26.    {
27.        for(i=0;i<n;i++)
28.            a[i]=i;
29.        MPI_Send(a,1,mt,1,777,MPI_COMM_WORLD);
30.    }
31.    if(rank==1)
32.    {
33.        MPI_Recv(a,1,mt,0,777,MPI_COMM_WORLD,&stat);
34.        printf("rank= %d a: ",rank);
35.        for(i=0;i<n;i++)
36.            printf("%d ",a[i]);
37.        printf("\n");
38.    }
39.    MPI_Finalize();
40.    return 0;
41. }
```

Рис. 138. Листинг программы



```
mc - stud@hpchead:~/M182/serg/13
[stud@hpchead 13]$ mpirun -n 2 -f ./machines ./a.out
rank= 1 a: 0 -1 2 3 -1 -1 6 7 8 -1 -1 -1 12 13 14 15 -1 -1 -1 -1
[stud@hpchead 13]$
```

Рис. 139. Скрин запуска программы

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     int n=9,i,j,s=0;
8.     MPI_Status stat;
9.     MPI_Init(&argc, &argv);
10.    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11.    MPI_Comm_size(MPI_COMM_WORLD, &size);
12.    int* bl=new int[n];
13.    int* ds=new int[n];
14.    for(i=0;i<n;i++)
15.    {
16.        bl[i]=i+1;
17.        ds[i]=i*n;
18.    }
19.    MPI_Datatype mt;
20.    MPI_Type_indexed(n,bl,ds,MPI_INT,&mt);
21.    MPI_Type_commit(&mt);
22.    int **b=new int *[n];
23.    b[0]=new int[n*n];
24.    for(i=1;i<n;i++)
25.        b[i]=b[i-1]+n;
26.    for(i=0;i<n;i++)
27.        for(j=0;j<n;j++)
28.            b[i][j]=0;
29.    if(rank==0)
30.    {
31.        for(i=0;i<n;i++)
32.            for(j=0;j<n;j++)
33.                b[i][j]=10*(i+1)+j+1;
34.        MPI_Send(*b,1,mt,1,777,MPI_COMM_WORLD);
35.    }
36.    if(rank==1)
37.    {
38.        MPI_Recv(*b,1,mt,0,777,MPI_COMM_WORLD,&stat);
39.    }
40.    for(i=0; i<100000000*rank; i++)
41.        s+=1;
42.    printf("rank= %d b: \n",rank);
43.    for(i=0; i<n; i++)
44.    {
45.        for(j=0;j<n;j++)
46.            printf(" %d ",b[i][j]);
47.        printf("\n");
48.    }
49.    MPI_Finalize();
50.    return 0;
51. }
```

Рис. 140. Листинг программы

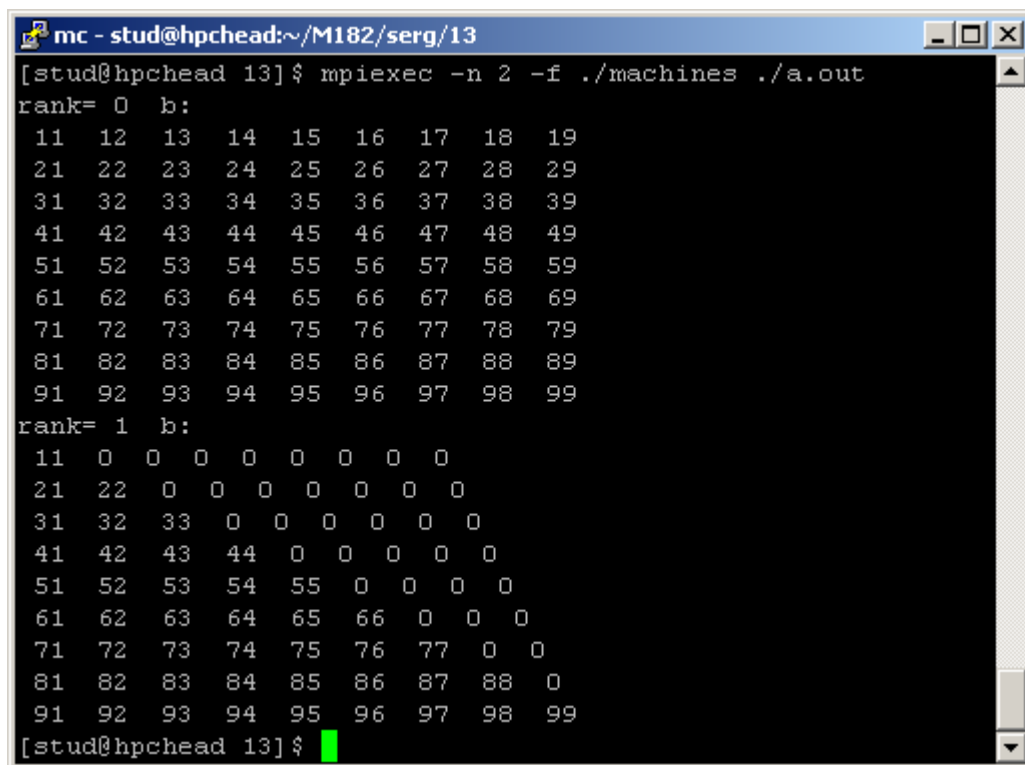
Пример 5. Передача нижнетреугольной матрицы

Выполним пересылку элементов двумерного массива, отмеченных на следующей схеме:

11								
21	22							
31	32	33						
41	42	43	44					
51	52	53	54	55				
61	62	63	64	65	66			
71	72	73	74	75	76	77		
81	82	83	84	85	86	87	88	
91	92	93	94	95	96	97	98	99

Если предположить, что размер матрицы $b[n][n]$, то можно заметить, что в данной схеме n блоков (строк), в которых содержится $(1, 2, 3, 4, \dots, i, \dots)$ элементов, адреса блоков следующие – $(0, n, 2n, 3n, \dots, i*n, \dots)$ Этого достаточно для описания аргументов функции `MPI_Type_indexed`.

На рис. 140 приведен листинг программы, а на рис. 141 – скрин ее запуска, демонстрирующий успешную передачу нижнетреугольной матрицы.



```
mc - stud@hpchead:~/M182/serg/13
[stud@hpchead 13]$ mpirun -n 2 -f ./machines ./a.out
rank= 0  b:
 11 12 13 14 15 16 17 18 19
 21 22 23 24 25 26 27 28 29
 31 32 33 34 35 36 37 38 39
 41 42 43 44 45 46 47 48 49
 51 52 53 54 55 56 57 58 59
 61 62 63 64 65 66 67 68 69
 71 72 73 74 75 76 77 78 79
 81 82 83 84 85 86 87 88 89
 91 92 93 94 95 96 97 98 99
rank= 1  b:
 11 0 0 0 0 0 0 0 0
 21 22 0 0 0 0 0 0 0
 31 32 33 0 0 0 0 0 0
 41 42 43 44 0 0 0 0 0
 51 52 53 54 55 0 0 0 0
 61 62 63 64 65 66 0 0 0
 71 72 73 74 75 76 77 0 0
 81 82 83 84 85 86 87 88 0
 91 92 93 94 95 96 97 98 99
[stud@hpchead 13]$
```

Рис. 141. Скрин запуска программы

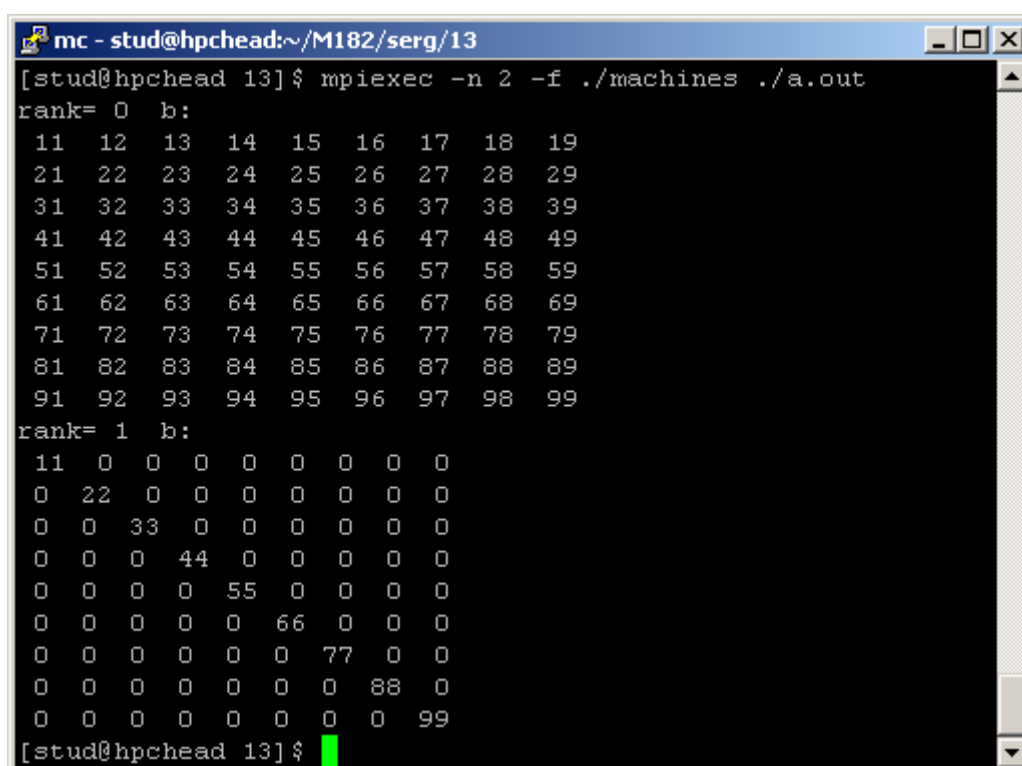
Пример 6. Передача главной диагонали матрицы

В отличие от предыдущего случая в каждом блоке содержится по 1-му элементу, адреса блоков следующие – $(0, n+1, 2(n+1), 3(n+1), \dots, i*(n+1), \dots)$ Этого достаточно для описания аргументов функции `MPI_Type_indexed`.

На рис. 142 приведены строки из предыдущей программы, которые были изменены (строки 5-6), а на рис. 143 – скрин ее запуска, демонстрирующий успешную передачу элементов главной диагонали матрицы.

```
1. int* bl=new int[n];
2. int* ds=new int[n];
3.   for(i=0;i<n;i++)
4.   {
5.     bl[i]=1;
6.     ds[i]=i*(n+1);
7.   }
8. MPI_Datatype mt;
9. MPI_Type_indexed(n,bl,ds,MPI_INT,&mt);
10. MPI_Type_commit(&mt);
```

Рис. 142. Листинг фрагмента программы



```
mc - stud@hpchead:~/M182/serg/13
[stud@hpchead 13]$ mpiexec -n 2 -f ./machines ./a.out
rank= 0  b:
11 12 13 14 15 16 17 18 19
21 22 23 24 25 26 27 28 29
31 32 33 34 35 36 37 38 39
41 42 43 44 45 46 47 48 49
51 52 53 54 55 56 57 58 59
61 62 63 64 65 66 67 68 69
71 72 73 74 75 76 77 78 79
81 82 83 84 85 86 87 88 89
91 92 93 94 95 96 97 98 99
rank= 1  b:
11 0 0 0 0 0 0 0 0
0 22 0 0 0 0 0 0 0
0 0 33 0 0 0 0 0 0
0 0 0 44 0 0 0 0 0
0 0 0 0 55 0 0 0 0
0 0 0 0 0 66 0 0 0
0 0 0 0 0 0 77 0 0
0 0 0 0 0 0 0 88 0
0 0 0 0 0 0 0 0 99
[stud@hpchead 13]$
```

Рис. 143. Скрин запуска программы

Пример 7. Передача главной и побочной диагоналей матрицы одной посылкой

В отличие от предыдущего случая блоков уже будет в два раза больше – $(2n)$, в каждом блоке по прежнему содержится по 1-му элементу, адреса первых (n) блоков следующие – $(0, n+1, 2(n+1), 3(n+1), \dots, i*(n+1), \dots)$, следующих (n) блоков – $(n-1, 2(n-1), 3(n-1), \dots, (i+1)(n-1), \dots)$.

На рис. 144 приведены строки из предыдущей программы, которые были изменены, а на рис. 145 – скрин ее запуска, демонстрирующий успешную передачу элементов главной и побочной диагоналей матрицы.

```

1. int* bl=new int[2*n];
2.   for(i=0;i<2*n;i++)
3.     bl[i]=1;
4. int* ds=new int[2*n];
5.   for(i=0;i<n;i++)
6.     ds[i]=i*(n+1);
7.   for(i=n;i<2*n;i++)
8.     ds[i]=(i-n+1)*(n-1);
9. MPI_Datatype mt;
10. MPI_Type_indexed(2*n,bl,ds,MPI_INT,&mt);
11. MPI_Type_commit(&mt);

```

Рис. 144. Листинг фрагмента программы

```

mc - stud@hpchead:~/M182/serg/13
[stud@hpchead 13]$ mpiexec -n 2 -f ./machines ./a.out
rank= 0  b:
 11 12 13 14 15 16 17 18 19
 21 22 23 24 25 26 27 28 29
 31 32 33 34 35 36 37 38 39
 41 42 43 44 45 46 47 48 49
 51 52 53 54 55 56 57 58 59
 61 62 63 64 65 66 67 68 69
 71 72 73 74 75 76 77 78 79
 81 82 83 84 85 86 87 88 89
 91 92 93 94 95 96 97 98 99
rank= 1  b:
 11 0 0 0 0 0 0 0 19
 0 22 0 0 0 0 0 28 0
 0 0 33 0 0 0 37 0 0
 0 0 0 44 0 46 0 0 0
 0 0 0 0 55 0 0 0 0
 0 0 0 64 0 66 0 0 0
 0 0 73 0 0 0 77 0 0
 0 82 0 0 0 0 0 88 0
 91 0 0 0 0 0 0 0 99
[stud@hpchead 13]$

```

Рис. 145. Скрин запуска программы

Задания

1. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ по формуле: $a(i,j)=10*(i+1)+(j+1)$. 2) нижнетреугольная матрица передается с 0-го процесса на 1-й. 3) 1-й процесс сохраняет полученную матрицу в транспонированном виде (должна получиться верхнетреугольная матрица).
2. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ по формуле: $a(i,j)=10*(i+1)+(j+1)$. 2) матрица целиком передается с 0-го процесса на 1-й. 3) 1-й процесс сохраняет полученную матрицу, расставляя столбцы в обратном порядке.
3. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ и заполняется единицами. 2)

первый и последний столбец, первая и последняя строка передаются одной посылкой с 0-го процесса на 1-й. 3) 1-й процесс сохраняет полученные элементы в точно такие же места в матрицу (должна получиться рамка по периметру).

4. Выполните пересылку с 0-го процесса на 1-й процесс элементов двумерного массива, отмеченных на следующей схеме:

11								
	22	23						
		33	34	35				
			44	45	46	47		
				55	56	57	58	59
					66	67	68	69
						77	78	79
							88	89
								99

5. Выполните пересылку с 0-го процесса на 1-й процесс элементов двумерного массива, отмеченных на следующей схеме:

11								
							28	29
31	32	33						
					46	47	48	49
51	52	53	54	55				
			64	65	66	67	68	69
71	72	73	74	75	76	77		
	82	83	84	85	86	87	88	89
91	92	93	94	95	96	97	98	99

6. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ по формуле: $a(i,j)=10*(i+1)+(j+1)$ 2) на 1-м процессе создается такая же матрица и заполняется нулями. 3) требуется передать с 0-го на 1-й процесс четные строки одной посылкой.
7. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ по формуле: $a(i,j)=10*(i+1)+(j+1)$ 2) на 1-м процессе создается такая же матрица и заполняется нулями. 3) требуется передать с 0-го на 1-й процесс четные столбцы одной посылкой.
8. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ по формуле: $a(i,j)=10*(i+1)+(j+1)$ 2) на 1-м процессе создается такая же матрица и заполняется нулями. 3) требуется передать с 0-го на 1-й процесс нечетные строки одной посылкой.
9. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ по формуле: $a(i,j)=10*(i+1)+(j+1)$ 2) на 1-м процессе создается такая же матрица и

заполняется нулями. 3) требуется передать с 0-го на 1-й процесс нечетные столбцы одной посылкой.

10. Напишите программу, которая осуществляет рассылку строк двумерного массива с 0-го процесса все остальным по следующему правилу: на 1-й процесс отправляются 1-2 столбцы, на 2-й процесс – 3-5 столбцы, на 3-й – 5-9 столбцы и т.д. Для рассылки используйте функцию `MPI_Scatter`.
11. Напишите программу, которая реализует следующий алгоритм: на 0-м процессе задается двумерный массив $N \times N$ по формуле: $a(i,j) = 10 \cdot (i+1) + (j+1)$, который по одной строке рассылается всем ненулевым процессам по кругу. Например, Вы запускаете программу на 3-х процессах, 1-й процесс в итоге должен получить 1, 4, 7 и т.д., 2-й процесс – 2, 5, 8 и т.д. Для рассылки используйте функцию `MPI_Scatter`.

14. Производные типы данных: универсальный конструктор. Передача упакованных данных

В процессе создания параллельных приложений часто возникает необходимость передачи разнотипных данных, а также данных, имеющих нестандартную структуру. В стандарте MPI предлагается два способа решения данной проблемы: использование механизма передачи упакованных данных или использование универсального конструктора для описания карт размещения данных в оперативной памяти. В данном пункте рассматриваются оба подхода с демонстрацией на примере передачи разнотипных данных одной посылкой.

14.1. Передача упакованных данных

Функция *MPI_Pack* упаковывает элементы предопределенного или производного типа MPI, помещая их побайтное представление в выходной буфер.

```
int MPI_Pack(void* inbuf, int incount, MPI_Datatype  
datatype, void *outbuf, int outsize, int *position,  
MPI_Comm comm)
```

Входные аргументы:

- inbuf - адрес начала области памяти с элементами, которые требуется упаковать;
- incount - число упаковываемых элементов;
- datatype - тип упаковываемых элементов;
- outsize - размер выходного буфера в байтах;
- comm - коммуникатор.

Выходные аргументы:

- outbuf - адрес начала выходного буфера для упакованных данных;
- position - текущая позиция в выходном буфере в байтах.

Функция *MPI_Pack* упаковывает incount элементов типа datatype из области памяти с начальным адресом inbuf. Результат упаковки помещается в выходной буфер с начальным адресом outbuf и размером outsize байт. Параметр position указывает текущую позицию в байтах, начиная с которой будут размещаться упакованные данные. На выходе из функции значение position увеличивается на число упакованных байт, указывая на первый свободный байт для упаковки следующей порции данных. Параметр comm - коммуникатор.

Функция *MPI_Unpack* извлекает заданное число элементов некоторого типа из побайтного представления элементов во входном массиве.

```
int MPI_Unpack(void* inbuf, int insize, int *position,
void *outbuf, int outcount, MPI_Datatype datatype,
MPI_Comm comm)
```

Входные аргументы:

inbuf - адрес начала входного буфера с упакованными данными;
 insize - размер входного буфера в байтах;
 position - текущая позиция во входном буфере в байтах;
 outcount - число извлекаемых элементов;
 datatype - тип извлекаемых элементов;
 comm - коммуниторатор.

Выходные аргументы:

outbuf - адрес начала области памяти для размещения распакованных элементов;
 position - текущая позиция в входном буфере в байтах.

Функция *MPI_Unpack* извлекает outcount элементов типа datatype из побайтного представления элементов в массиве inbuf, начиная с адреса position. После возврата из функции параметр position увеличивается на размер распакованного сообщения. Результат распаковки помещается в область памяти с начальным адресом outbuf.

Сценарий использования механизма упаковки-распаковки данных для передачи разнотипных данных можно описать следующим образом.

- Для посылки элементов разного типа из нескольких областей памяти их следует предварительно запаковать в один массив, последовательно обращаясь к функции упаковки *MPI_Pack*. При первом вызове функции упаковки параметр position, как правило, устанавливается в 0, чтобы упакованное представление размещалось с начала буфера. Для непрерывного заполнения буфера необходимо в каждом последующем вызове использовать значение параметра position, полученное из предыдущего вызова.
- Упакованный буфер пересылается любыми коммуникационными операциями с указанием типа MPI_PACKED и коммуниторатора comm, который использовался при обращениях к функции *MPI_Pack*.
- Полученное упакованное сообщение распаковывается в различные массивы или переменные. Это реализуется последовательными вызовами функции распаковки *MPI_Unpack* с указанием числа элементов, которое следует извлечь при каждом вызове, и с передачей значения position, возвращенного предыдущим вызовом. При первом вызове функции параметр position следует установить в 0. В общем случае, при первом обращении должно быть установлено то значение параметра position, которое было использовано при первом обращении к функции упаковки данных. Очевидно, что для правильной распаковки данных очередность извлечения данных должна быть той же самой, как и при упаковке.

Для упаковки данных и последующей передачи требуется выделить место в памяти под непрерывный буфер, который и пересылается при помощи коммуникационных функций. Удобнее всего для выделения требуемого количества байт в памяти объявлять символьный массив, так как один элемент типа `char` занимает в памяти один байт.

Функция `MPI_Pack_size` помогает определить размер буфера, необходимый для упаковки некоторого количества данных типа `datatype`.

```
int MPI_Pack_size(int incount, MPI_Datatype datatype,  
MPI_Comm comm, int *size)
```

Входные аргументы:

`incount` - число элементов, подлежащих упаковке;
`datatype` - тип элементов, подлежащих упаковке;
`comm` - коммуникатор.

Выходные аргументы:

`size` - размер сообщения в байтах после его упаковки.

Первые три параметра функции `MPI_Pack_size` такие же, как у функции `MPI_Pack`. После обращения к функции параметр `size` будет содержать размер сообщения в байтах после его упаковки.

В Си есть штатная функция – `sizeof(datatype)`, которая возвращает количество байт, занимаемое типом данных (`datatype`). В примере передачи разнотипных данных именно она и использована для определения требуемого количества байт для буфера пересылки.

Пример 1. Передача разнотипных данных одной посылкой

Рассмотрим пример передачи данных с 0-го процесса на 1-й переменной типа `int`, переменной типа `double` и массива типа `char`. Пересылку обычным образом можно выполнить тремя парами `MPI_Send-MPI_Recv`. Однако при такой пересылке основное время уйдет на подготовку канала для передачи данных (3 раза), а не на собственно передачу малого количества байт информации от одного процесса другому. Гораздо эффективнее собрать все необходимые данные в один буфер и осуществить передачу одной отправкой данных. Однако существенным недостатком такого подхода являются временные затраты, необходимые для выполнения операции копирования данных из буфера в буфер на принимающей и передающей стороне, а также дополнительная утилизация оперативной памяти для размещения буфера под упаковываемые данные.

Разберем программу на рис. 146, осуществляющую передачу целочисленной переменной (`a=7`), переменной типа `double` (`b=0.8`) и массива типа `char` (“hello world!”) с 0-го процесса на 1-й.

Строки 7-9 – выделение памяти под передаваемые данные.

Строка 14 – определение размера буфера для упаковки передаваемых данных.

Строка 15 – выделение места под буфер упаковки размера (`n`) байт.

16 строка – выделение места под переменную, отвечающую за номер байтовой позиции при упаковке-распаковке данных.

Строки 19-21 – присваивание значений переменным и массиву на 0-м процессе.

Строки 22-24 – поочередная упаковка данных в буфер для отправки.

Строка 25 – отправка буфера с типом MPI_PACKED. На кластере, состоящем из однотипных узлов, можно тип данных указывать MPI_CHAR, MPI_BYTE, данные будут переданы корректно.

29 строка – прием буфера.

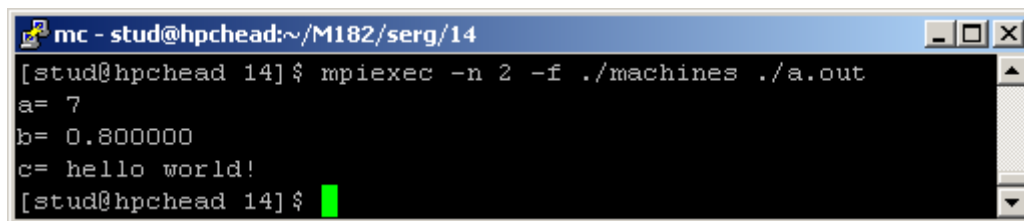
30-32 строки – распаковка данных из присланного буфера.

33-35 строки – контрольный вывод.

На рис. 147 приведен скрин запуска программы, демонстрирующий успешную передачу разнотипных данных.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. #include "string.h"
4. int main(int argc, char *argv[])
5. {
6.     int rank,size;
7.     int a;
8.     double b;
9.     char c[12];
10. MPI_Status stat;
11. MPI_Init(&argc, &argv);
12. MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13. MPI_Comm_size(MPI_COMM_WORLD, &size);
14. int n=sizeof(int)+sizeof(double)+12*sizeof(char);
15. char *buf=new char[n];
16. int pos=0;
17. if(rank==0)
18. {
19.     a=7;
20.     b=0.8;
21. strcpy(c,"hello world!");
22. MPI_Pack(&a,1,MPI_INT,buf,n,&pos,MPI_COMM_WORLD);
23. MPI_Pack(&b,1,MPI_DOUBLE,buf,n,&pos,MPI_COMM_WORLD);
24. MPI_Pack(&c,12,MPI_CHAR,buf,n,&pos,MPI_COMM_WORLD);
25. MPI_Send(buf,n,MPI_PACKED,1,777,MPI_COMM_WORLD);
26. }
27. if(rank==1)
28. {
29.     MPI_Recv(buf,n,MPI_PACKED,0,777,MPI_COMM_WORLD,&stat);
30.     MPI_Unpack(buf,n,&pos,&a,1,MPI_INT,MPI_COMM_WORLD);
31.     MPI_Unpack(buf,n,&pos,&b,1,MPI_DOUBLE,MPI_COMM_WORLD);
32.     MPI_Unpack(buf,n,&pos,&c,12,MPI_CHAR,MPI_COMM_WORLD);
33.     printf("a= %d \n",a);
34.     printf("b= %f \n",b);
35.     printf("c= %s \n",c);
36. }
37. MPI_Finalize();
38. return 0;
39. }
```

Рис. 146. Листинг программы



```
mc - stud@hpchead:~/M182/serg/14
[stud@hpchead 14]$ mpiexec -n 2 -f ./machines ./a.out
a= 7
b= 0.800000
c= hello world!
[stud@hpchead 14]$
```

Рис. 147. Скрин запуска программы

14.2. Конструктор описания карты расположения различных блоков разных типов данных с разными промежутками

Конструктор типа ***MPI_Type_create_struct*** – самый универсальный из всех конструкторов типа. Создаваемый им тип является структурой, состоящей из произвольного числа блоков, каждый из которых может содержать произвольное число элементов одного из зарегистрированных типов и может быть смещен на произвольное число байтов от начала размещения структуры.

```
int MPI_Type_create_struct (int count, int  
array_of_blocklengths[], MPI_Aint  
array_of_displacements[], MPI_Datatype array_of_types[],  
MPI_Datatype *newtype)
```

Входные аргументы:

- count - количество блоков;
- array_of_blocklengths - массив, хранящий число элементов базового типа в каждом блоке;
- array_of_displacements - массив смещений каждого блока от начала размещения элемента нового типа, выраженный в байтах;
- array_of_types - массив, содержащий тип элементов в каждом блоке;
- oldtype - базовый тип данных;

Выходные аргументы:

- newtype - новый производный тип данных.

Функция создает тип newtype, элемент которого состоит из count блоков, где i-ый блок содержит array_of_blocklengths[i] элементов типа array_of_types[i]. Смещение i-ого блока от начала размещения элемента нового типа измеряется в байтах и задается в array_of_displacements[i]. Графическая интерпретация работы конструктора MPI_Type_create_struct приведена на рисунке 148.

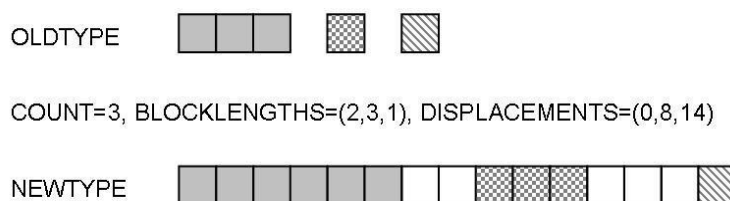


Рис. 148. Графическая интерпретация работы конструктора MPI_Type_create_struct

Смещения в общем типе данных задаются относительно некоторого начального буферного адреса. *Абсолютный адрес* может быть определен через эти смещения как смещение относительно «адресного нуля» (начала адресного пространства). Этот начальный «нулевой адрес» помечается константой **MPI_BOTTOM**. Поэтому тип данных может описывать абсолютный адрес элементов в коммуникационном буфере, в этом случае аргумент **buf** получает значение MPI_BOTTOM. Адрес ячейки памяти может быть найден вызовом функции **MPI_Get_address**.

int MPI_Get_address (void *location, MPI_Aint *address)

Входные аргументы:

location - ячейка в памяти вызывающего процесса (например, некоторая переменная).

Выходные аргументы:

address - адрес ячейки.

Пример 2.

Задача для выполнения аналогична примеру 1, но для ее решения применим изученный конструктор описания карты расположения данных в оперативной памяти.

Данный способ лишен недостатков механизма упаковки-распаковки, так как для его реализации не требуется дополнительной утилизации оперативной памяти, а также дополнительного копирования данных в пересылаемый буфер перед отправкой и обратного копирования после приема.

Разберем программу на рис. 149.

Строки 7-9 – выделение памяти под передаваемые данные. При этом данные расположены в оперативной памяти по некоторым физическим адресам.

Строка 14 – определение вспомогательного массива, содержащего информацию о количестве элементов в каждом блоке. В нашем случае разнотипных блоков будет 3: 1 элемент целочисленной переменной (a=7), 1 элемент переменной типа double (b=0.8) и 12 элементов массива типа char (“hello world!”).

Строка 16 – определение вспомогательного массива, содержащего описание типов каждого из трех блоков.


```

1. #include <stdio.h>
2. #include "mpi.h"
3. #include "string.h"
4. int main(int argc, char *argv[])
5. {
6.     int rank,size;
7.     int a;
8.     double b;
9.     char *c=new char[12];
10. MPI_Status stat;
11. MPI_Init(&argc, &argv);
12. MPI_Comm_rank(MPI_COMM_WORLD, &rank);
13. MPI_Comm_size(MPI_COMM_WORLD, &size);
14. int bl[]={ 1,1,12};
15. MPI_Datatype mt;
16. MPI_Datatype dt[]={MPI_INT,MPI_DOUBLE,MPI_CHAR};
17. MPI_Aint ds[3],aa,ab,ac;
18. MPI_Get_address(&a,&aa);
19. MPI_Get_address(&b,&ab);
20. MPI_Get_address(c,&ac);
21. ds[0]=0;
22. ds[1]=ab-aa;
23. ds[2]=ac-aa;
24. MPI_Type_struct(3,bl,ds,dt,&mt);
25. MPI_Type_commit(&mt);
26. if(rank==0)
27. {
28.     a=7;
29.     b=0.8;
30.     strcpy(c,"hello world!");
31. MPI_Send(&a,1,mt,1,777,MPI_COMM_WORLD);
32. }
33. if(rank==1)
34. {
35.     MPI_Recv(&a,1,mt,0,777,MPI_COMM_WORLD,&stat);
36.     printf("a= %d \n",a);
37.     printf("b= %f \n",b);
38.     printf("c= %s \n",c);
39. }
40. MPI_Finalize();
41. return 0;
42. }

```

Рис. 149. Листинг программы

17 строка – выделение места под переменные и массив, отвечающие за байтовые адреса блоков.

Строки 18-20 – определение байтовых адресов блоков по их физическим адресам.

Строки 21-23 – определение смещений каждого блока относительно адреса, указанного первым аргументом в функциях MPI_Send-MPI_Recv. Так как первым аргументом в этих функциях указан адрес переменной a, то ds[0], описывающее смещение первого блока относительно этого адреса будет равно нулю. Следующие смещения определены разницей соответствующих байтовых адресов блоков.

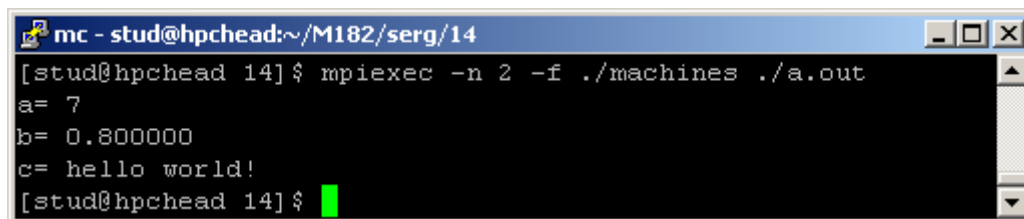
Строка 24-25 – создание и регистрация нового описателя карты размещения данных в оперативной памяти.

31 строка - отправка данных по карте, состоящей из трех разнотипных блоков.

35 строка – прием.

36-38 строки – контрольный вывод.

На рис. 150 приведен скрин запуска программы, демонстрирующий успешную передачу разнотипных данных.



```
mc - stud@hpchead:~/M182/serg/14
[stud@hpchead 14]$ mpiexec -n 2 -f ./machines ./a.out
a= 7
b= 0.800000
c= hello world!
[stud@hpchead 14]$
```

Рис. 150. Скрин запуска программы

Задания

1. Требуется одной пересылкой с 0-го процесса на 1-й отправить целочисленный массив $a(i)=i, i=0..n$, массив $\text{double } b(i)=i/(i+1), i=0..n$, массив $\text{char } h$ =(набор из n символов). Для передачи данных одной посылкой необходимо использовать упаковку-распаковку данных и конструктор `MPI_Type_struct`. Проведите исследование времени на передачу данных в зависимости от способа передачи данных и размера передаваемых массивов. Полученные данные сведите в таблицу.
2. Требуется написать следующую параллельную программу: 1) на 0-м процессе задается квадратная матрица $N \times N$ по формуле: $a(i,j)=10*(i+1)+(j+1)$. 2) нижнетреугольная матрица передается с 0-го процесса на 1-й. 3) 1-й процесс сохраняет полученную матрицу в транспонированном виде (должна получиться верхнетреугольная матрица). На процессе-отправителе и получателе нужно использовать конструктор `MPI_Type_struct` для сборки подготовленных карт столбцов и строк. В отчете нужно дать графическое и словесное описание карты.
3. Напишите программу для пересылки массива структур от 0-го процесса 1-му. Определите время пересылки в зависимости от размера передаваемого массива. Структура массива следующая:

```
struct Partstruct
{ int type; /* тип точки (0 – граничная, 1 - внутренняя) */
double d[3]; /* координаты частицы */
double value; /* значение функции в точке */
};
struct Partstruct particle[n];
```

4. Используя изученные конструкторы описания карт расположения данных в оперативной памяти, а также функцию `MPI_Scatter` (`MPI_Scatterv`), напишите программу для слоисто-строчного распределения двумерного массива по процессам. Например, Вы запускаете программу на 3-х процессах, 0-й процесс в итоге должен получить 0, 3, 6 и т.д. строки, 1-й процесс - 1, 4, 7 и т.д., 2-й процесс – 2, 5, 8 и т.д.
5. Используя изученные конструкторы описания карт расположения данных в оперативной памяти, а также функцию `MPI_Scatter` (`MPI_Scatterv`), напишите программу для слоисто-столбцового распределения двумерного массива по процессам.

15. Группы и коммутаторы

Часто в приложениях возникает потребность ограничить область коммуникаций некоторым набором процессов, которые составляют подмножество исходного набора. Для выполнения каких-либо коллективных операций внутри этого подмножества из них должна быть сформирована своя область связи, описываемая своим коммутатором. Для решения таких задач MPI поддерживает два взаимосвязанных механизма. Во-первых, имеется набор функций для работы с группами процессов как упорядоченными множествами, и, во-вторых, набор функций для работы с коммутаторами для создания новых коммутаторов как описателей новых областей связи.

15.1. Функции работы с группами процессов

Группа представляет собой упорядоченное множество процессов. Каждый процесс идентифицируется переменной целого типа. Идентификаторы процессов образуют непрерывный ряд, начинающийся с 0. В MPI вводится специальный тип данных `MPI_Group` и набор функций для работы с переменными и константами этого типа. Существует две предопределенных группы:

- `MPI_GROUP_EMPTY` – группа, не содержащая ни одного процесса;
- `MPI_GROUP_NULL` – значение, возвращаемое в случае, когда группа не может быть создана.

Созданная группа не может быть модифицирована (расширена или усечена), может быть только создана новая группа. Интересно отметить, что при инициализации MPI не создается группы, соответствующей коммутатору `MPI_COMM_WORLD`. Она должна создаваться специальной функцией явным образом.

Функция определения числа процессов в группе `MPI_Group_size`

`MPI_Group_size(MPI_Group group, int *size)`

Входные аргументы:

`group` – группа;

Выходные аргументы:

`size` – число процессов в группе.

Функция возвращает число процессов в группе. Если `group = MPI_GROUP_EMPTY`, тогда `size = 0`.

Функция определения номера процесса в группе `MPI_Group_rank`

`MPI_Group_rank(MPI_Group group, int *rank)`

Входные аргументы:

`group` – группа;

Выходные аргументы:

rank - номер процесса в группе.

Функция `MPI_Group_rank` возвращает номер в группе процесса, вызвавшего функцию. Если процесс не является членом группы, то возвращается значение `MPI_UNDEFINED`.

Функция установки соответствия между номерами процессов в двух группах `MPI_Group_translate_ranks`

`MPI_Group_translate_ranks(MPI_Group group1, int n, int *ranks1, MPI_Group group2, int *ranks2)`

Входные аргументы:

group1 - группа1;

n - число процессов, для которых устанавливается соответствие;

ranks1 - массив номеров процессов из 1-й группы;

group2 - группа2.

Выходные аргументы:

ranks2 - номера тех же процессов во второй группе.

Функция определяет относительные номера одних и тех же процессов в двух разных группах. Если процесс во второй группе отсутствует, то для него устанавливается значение `MPI_UNDEFINED`.

Для создания новых групп в MPI имеется 8 функций. Группа может быть создана либо с помощью коммуникатора, либо с помощью операций над множествами процессов других групп.

Функция создания группы с помощью коммуникатора `MPI_Comm_group`

`MPI_Comm_group(MPI_Comm comm, MPI_Group *group)`

Входные аргументы:

comm - коммуникатор;

Выходные аргументы:

group - группа.

Функция создает группу group для множества процессов, входящих в область связи коммуникатора comm.

Следующие три функции имеют одинаковый синтаксис и создают новую группу как результат операции над множествами процессов двух групп.

`MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)`

`MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)`

`MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)`

Входные аргументы:

group1 - первая группа;

group2 - вторая группа.

Выходные аргументы:

newgroup - новая группа.

Операции определяются следующим образом:

- Union – формирует новую группу из элементов 1-й группы и из элементов 2-й группы, не входящих в 1-ю (объединение множеств).
- Intersection – новая группа формируется из элементов 1-й группы, которые входят также и во 2-ю. Упорядочивание, как в 1-й группе (пересечение множеств).
- Difference – новую группу образуют все элементы 1-й группы, которые не входят во 2-ю. Упорядочивание, как в 1-й группе (дополнение множеств).

Созданная группа может быть пустой, что эквивалентно MPI_GROUP_EMPTY.

Новые группы могут быть созданы с помощью различных выборок из существующей группы. Следующие две функции имеют одинаковый синтаксис, но являются дополнительными по отношению друг к другу.

```
MPI_Group_incl(MPI_Group group, int n, int *ranks,  
MPI_Group *newgroup)
```

```
MPI_Group_excl(MPI_Group group, int n, int *ranks,  
MPI_Group *newgroup)
```

Входные аргументы:

group - существующая группа;

n - число элементов в массиве ranks;

ranks - массив номеров процессов.

Выходные аргументы:

newgroup - новая группа.

Функция MPI_Group_incl создает новую группу, которая состоит из процессов существующей группы, перечисленных в массиве ranks. Процесс с номером i в новой группе есть процесс с номером ranks[i] в существующей группе. Каждый элемент в массиве ranks должен иметь корректный номер в группе group, и среди этих элементов не должно быть совпадающих.

Функция MPI_Group_excl создает новую группу из тех процессов group, которые не перечислены в массиве ranks. Процессы упорядочиваются, как в группе group. Каждый элемент в массиве ranks должен иметь корректный номер в группе group, и среди них не должно быть совпадающих.

Две следующие функции по смыслу совпадают с предыдущими, но используют более сложное формирование выборки. Массив ranks заменяется двумерным массивом ranges, представляющим собой набор триплетов для задания диапазонов процессов.

```
MPI_Group_range_incl(MPI_Group group, int n, int  
ranges[][3], MPI_Group *newgroup)
```

```
MPI_Group_range_excl(MPI_Group group, int n, int  
ranges[][3], MPI_Group *newgroup)
```

Каждый триплет имеет вид: нижняя граница, верхняя граница, шаг.

Уничтожение созданных групп выполняется функцией `MPI_Group_free`.

`MPI_Group_free(MPI_Group *group)`

Входные аргументы:

`group` - уничтожаемая группа.

Пример 1. Создание групп процессов.

Разберем программу создания групп процессов, соответствующих следующей схеме:

rank	0	1	2	3	4	5	6	7	8	9
R1	0	1	2	3	4	5	-	-	-	-
R2	-	-	-	-	-	0	1	2	3	4

Первая строка в таблице – перечислены номера процессов в коммуникаторе `MPI_COMM_WORLD`.

Вторая строка – номера процессов для включения в первую группу.

Третья строка – номера процессов для включения во вторую группу.

Назначение программы (рис. 151) – создание двух групп и определение номеров процессов в рамках данных групп.

Строка 10 – выделение памяти под описатели родительской группы и двух создаваемых новых групп.

Так как при инициализации `MPI` создается коммуникатор `MPI_COMM_WORLD`, а никакой группы не порождается, то на первом этапе необходимо создать родительскую группу при помощи функции `MPI_Comm_group` (строка 11).

Данная группа будет иметь в своем составе все процессы коммуникатора `MPI_COMM_WORLD`. Более того – номера процессов в группе будут такими же, как и в коммуникаторе (можете проверить функцией `MPI_Group_rank`).

Строка 12 – определение переменных `n1`, `n2`, содержащих количество процессов в каждой группе, а также выделение памяти под переменные `R1`, `R2`, которые в последствии будут содержать номера процессов в группах.

Строки 13-14 – выделение памяти под вспомогательные массивы, содержащие номера процессов для создания групп. Присвоение значений элементам вспомогательных массивов производится в строках 15-18.

Строка 19 – создание группы `g1`, в которую вошли процессы с номерами от 0 до 5.

Строка 20 – определение номера процесса в первой группе. Следует понимать, что вызов данной функции будет произведен на всех процессах, но только на процессах с 0 по 5 функция вернет номера процессов, на остальных – некоторое числовое значение, отвечающее за то, что номер процесса не определен.

Строка 21 – создание группы g2, в которую вошли процессы с номерами от 5 до 9.

Строка 22 – определение номера процесса во второй группе. Здесь наоборот – процессы с 0-го по 4-й будут в данной группе неопределенными.

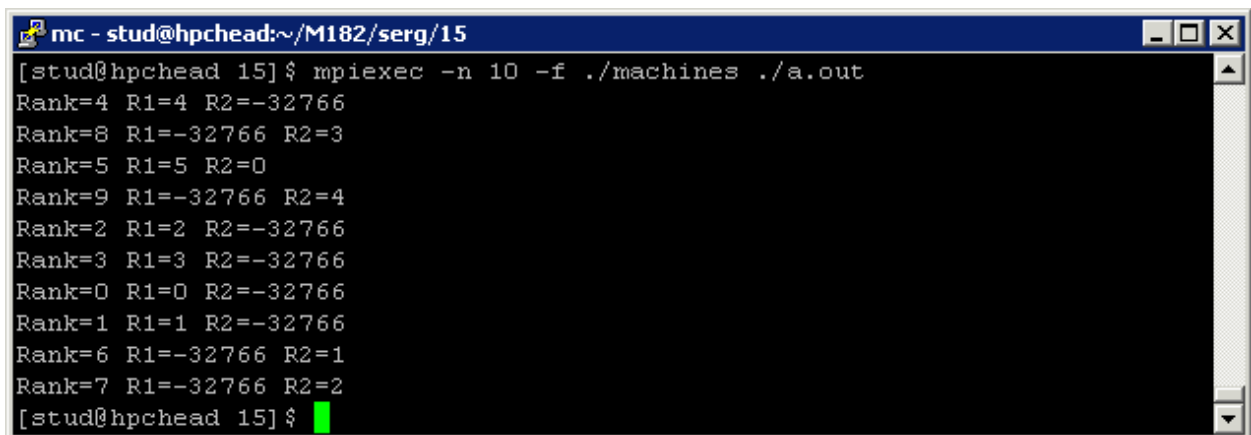
Строка 23-25 – вывод номера процесса в рамках коммуникатора MPI_COMM_WORLD, номера процесса в первой и во второй группах.

Строки 24-26 – уничтожение описателей групп.

На рис. 152 приведен скрин запуска программы на 10-и процессах.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10.    MPI_Group g,g1,g2;
11.    MPI_Comm_group(MPI_COMM_WORLD,&g);
12.    int n1 = 6, n2=5,R1, R2;
13.    int *m1 = new int[n1];
14.    int *m2 = new int[n2];
15.    for (int i = 0; i < n1; i++)
16.        m1[i] = i;
17.    for (int i = 0; i < n2; i++)
18.        m2[i] = i+5;
19.    MPI_Group_incl(g,n1,m1,&g1);
20.    MPI_Group_rank(g1,&R1);
21.    MPI_Group_incl(g,n2,m2,&g2);
22.    MPI_Group_rank(g2,&R2);
23.    printf("Rank=%d R1=%d R2=%d\n",rank,R1,R2);
24.    MPI_Group_free(&g1);
25.    MPI_Group_free(&g2);
26.    MPI_Group_free(&g);
27.    MPI_Finalize();
28.    return 0;
29. }
```

Рис. 151. Листинг программы



```
mc - stud@hpchead:~/M182/serg/15
[stud@hpchead 15]$ mpirun -n 10 -f ./machines ./a.out
Rank=4 R1=4 R2=-32766
Rank=8 R1=-32766 R2=3
Rank=5 R1=5 R2=0
Rank=9 R1=-32766 R2=4
Rank=2 R1=2 R2=-32766
Rank=3 R1=3 R2=-32766
Rank=0 R1=0 R2=-32766
Rank=1 R1=1 R2=-32766
Rank=6 R1=-32766 R2=1
Rank=7 R1=-32766 R2=2
[stud@hpchead 15]$
```

Рис. 152. Скрин запуска программы

15.2. Функции работы с коммутаторами

Коммутатор представляет собой скрытый объект с некоторым набором атрибутов, а также правилами его создания, использования и уничтожения. Коммутатор описывает некоторую область связи. Одной и той же области связи может соответствовать несколько коммутаторов, но даже в этом случае они не являются тождественными и не могут участвовать во взаимном обмене сообщениями. Если данные посылаются через один коммутатор, процесс-получатель может получить их только через тот же самый коммутатор.

При инициализации MPI создается два предопределенных коммутатора:

- `MPI_COMM_WORLD` – описывает область связи, содержащую все процессы;
- `MPI_COMM_SELF` – описывает область связи, состоящую из одного процесса.

В данном разделе рассматриваются функции работы с коммутаторами, которые разделяются на функции доступа к коммутаторам и функции создания коммутаторов. Функции доступа являются локальными и не требуют коммуникаций в отличие от функций создания, которые являются коллективными и могут потребовать межпроцессорных коммуникаций.

Две основные функции доступа к коммутатору (`MPI_Comm_size` – опрос числа процессов в области связи и `MPI_Comm_rank` – опрос идентификатора, или номера, процесса в области связи) были рассмотрены среди базовых функций MPI. Кроме них, имеется функция сравнения двух коммутаторов `MPI_Comm_compare`.

`MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)`

Входные аргументы:

- `comm1` – первый коммутатор;
- `comm2` – второй коммутатор.

Выходные аргументы:

- `result` – результат сравнения.

Возможные значения результата сравнения:

- `MPI_IDENT` – коммутаторы идентичны, представляют один и тот же объект;
- `MPI_CONGRUENT` – коммутаторы конгруэнтны, две области связи с одними и теми же атрибутами группы;
- `MPI_SIMILAR` – коммутаторы подобны, группы содержат одни и те же процессы, но другое упорядочивание;
- `MPI_UNEQUAL` – во всех других случаях.

Создание нового коммуникатора возможно с помощью одной из трех функций: `MPI_Comm_dup`, `MPI_Comm_create`, `MPI_Comm_split`.

Функция дублирования коммуникатора `MPI_Comm_dup`

`MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)`

Входные аргументы:

`comm` - коммуникатор.

Выходные аргументы:

`newcomm` - копия коммуникатора.

Данную функцию удобно для сокращения написания `MPI_COMM_WORLD`.

Функция создания коммуникатора `MPI_Comm_create`

`MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm)`

Входные аргументы:

`group` - группа, для которой создается коммуникатор.

Выходные аргументы:

`newcomm` - новый коммуникатор.

Эта функция создает коммуникатор для группы `group`. Для процессов, которые не являются членами группы, возвращается значение `MPI_COMM_NULL`. Функция возвращает код ошибки, если группа `group` не является подгруппой родительского коммуникатора.

Функция расщепления коммуникатора `MPI_Comm_split`

`MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)`

Входные аргументы:

`comm` - родительский коммуникатор;

`color` - признак подгруппы;

`key` - управление упорядочиванием.

Выходные аргументы:

`newcomm` - новый коммуникатор.

Данная функция удобна тем, что для создания коммуникатора нет необходимости создавать группу.

Функция расщепляет группу, связанную с родительским коммуникатором, на непересекающиеся подгруппы по одной на каждое значение признака подгруппы `color`. Значение `color` должно быть неотрицательным. Каждая подгруппа содержит процессы с одним и тем же значением `color`. Параметр `key` управляет упорядочиванием внутри новых групп: меньшему значению `key` соответствует меньшее значение идентификатора процесса. В случае равенства параметра `key` для нескольких

процессов упорядочивание выполняется в соответствии с порядком в родительской группе.

Функция уничтожения коммуникатора `MPI_Comm_free`

`MPI_Comm_free(MPI_Comm *comm)`

Входные аргументы:

`comm` - уничтожаемый коммуникатор.

Пример 2. Создание коммуникаторов функцией расщепления

Приведем алгоритм расщепления группы из восьми процессов на три подгруппы и его графическую интерпретацию (рис. 153).

```
MPI_comm comm, newcomm;
```

```
int rank;
```

```
.....
```

```
MPI_Comm_rank(comm, &rank);
```

```
color = myid%3;
```

```
MPI_Comm_split(comm, color, myid, &newcomm);
```

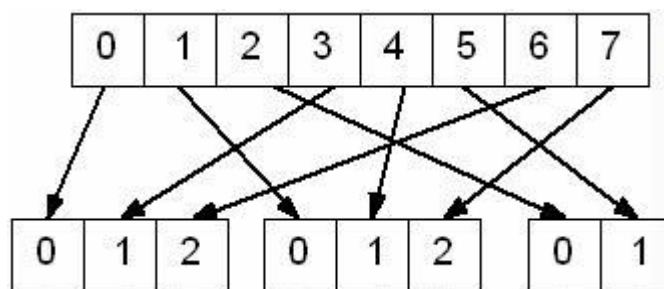


Рис. 153. Разбиение группы из восьми процессов на три подгруппы

В данном примере первую подгруппу образовали процессы, номера которых делятся на 3 без остатка, вторую, для которых остаток равен 1, и третью, для которых остаток равен 2. Отметим, что после выполнения функции `MPI_Comm_split` значения коммуникатора `newcomm` в процессах разных подгрупп будут отличаться. Первая строка в таблице – перечислены номера процессов в коммуникаторе `MPI_COMM_WORLD`.

Разберем программу создания коммуникаторов с помощью функции `MPI_Comm_split` (рис. 154).

Строка 5 – выделение места в памяти под переменную `rank_comm` для определения номеров процессов в рамках полученных коммуникаторов.

Строка 10 – выделение места в памяти под описатель новых коммуникаторов.

Строка 11 – создание трех новых коммуникаторов расщеплением на непересекающиеся множества процессов родительского коммуникатора.

Строка 12 – определение номеров процессов в рамках нового коммуникатора.

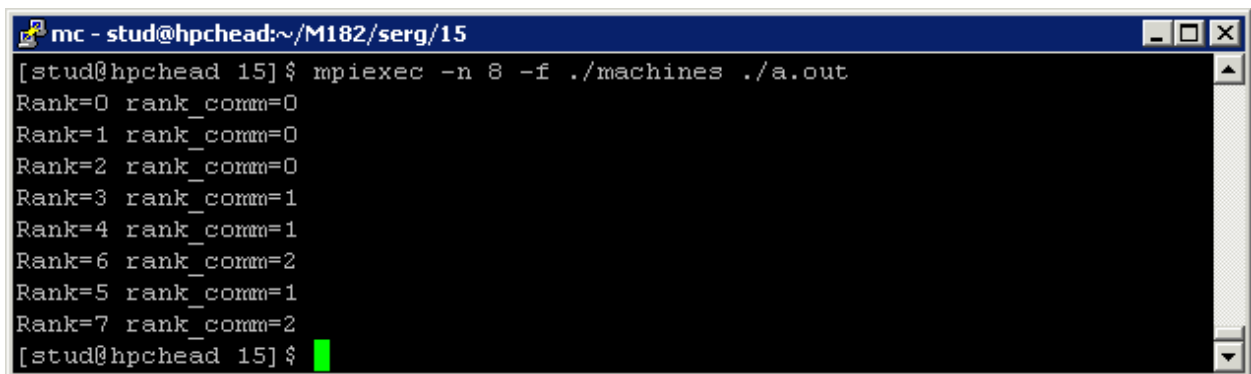
Строка 13 – вывод номеров процесса в рамках коммуникатора MPI_COMM_WORLD и вновь созданного.

Строка 14 – уничтожение описателя созданного коммуникатора.

На рис. 155 приведен скрин запуска программы на 8-и процессах. Вывод номеров процессов соответствует схеме, приведенной на рис. 153.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank,rank_comm;
6.     int size;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10.    MPI_Comm comm;
11.    MPI_Comm_split(MPI_COMM_WORLD,rank%3,0,&comm);
12.    MPI_Comm_rank(comm, &rank_comm);
13.    printf("Rank=%d rank_comm=%d \n",rank,rank_comm);
14.    MPI_Comm_free(&comm);
15.    MPI_Finalize();
16.    return 0;
17. }
```

Рис. 154. Листинг программы



```
mc - stud@hpchead:~/M182/serg/15
[stud@hpchead 15]$ mpirun -n 8 -f ./machines ./a.out
Rank=0 rank_comm=0
Rank=1 rank_comm=0
Rank=2 rank_comm=0
Rank=3 rank_comm=1
Rank=4 rank_comm=1
Rank=5 rank_comm=2
Rank=6 rank_comm=2
Rank=7 rank_comm=2
[stud@hpchead 15]$
```

Рис. 155. Скрин запуска программы

Пример 3.

Приведем пример использования расщепленного коммуникатора в коллективных коммуникационных функциях.

На рис. 156 приведена программа, созданная на основе предыдущего примера: создаются три новых коммуникатора, в каждом выполняется коллективная функция MPI_Gather для сборки номеров процессов в массив на 0-х процессах в рамках каждого коммуникатора, затем массивы выводятся на экран. Элементы массива содержат номера процессов, составляющих каждый из коммуникаторов. Наглядно демонстрирует разбиение на 3-и части родительского коммуникатора.

Итак, разберем программу, приведенную на рис. 156. Комментировать будем строки кода, которые добавились в этой программе по сравнению с предыдущей.

Строка 14 – определение количества процессов в рамках каждого из трех созданных коммунитаторов.

Строка 15 – выделение места в памяти под массив для сборки распределенных по процессам данных. С каждого процесса будут собираться в массив номера процессов (1 переменная), соответственно размер массива равен количеству процессов в рамках каждого из коммунитаторов.

Строка 16 – вызов коллективной функции для сборки массива а на 0-х процессах (их будет 3) в каждом из коммунитаторов.

Строка 17 задает условие на последующий вывод массива а на 0-х процессах – вывод будет осуществлен тремя процессами, так как коммунитаторов три.

Строки 19-20 – искусственная задержка по времени для избегания хаотичности вывода, когда несколько процессов одновременно выводят массив а.

Строки 21-24 – вывод номера процесса в рамках коммунитатора MPI_COMM_WORLD, номера в рамках созданного коммунитатора, полученного массива.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank,rank_comm;
6.     int size,size_comm;
7.     int s=0,i;
8.     MPI_Init(&argc, &argv);
9.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
10.    MPI_Comm_size(MPI_COMM_WORLD, &size);
11.    MPI_Comm comm;
12.    MPI_Comm_split(MPI_COMM_WORLD,rank%3,0,&comm);
13.    MPI_Comm_rank(comm, &rank_comm);
14.    MPI_Comm_size(comm, &size_comm);
15.    int *a=new int[size_comm];
16.    MPI_Gather(&rank_comm,1,MPI_INT,a,1,MPI_INT,0,comm);
17.    if(rank_comm==0)
18.    {
19.        for(i=0; i<100000000*rank; i++)
20.            s+=1;
21.        printf("Rank=%d rank_comm=%d a: ",rank,rank_comm);
22.        for(i=0; i<size_comm; i++)
23.            printf(" %d ",a[i]);
24.            printf("\n");
25.    }
26.    MPI_Comm_free(&comm);
27.    MPI_Finalize();
28.    return 0;
29. }
```

Рис. 156. Листинг программы

На рис. 157 приведен скрин запуска программы на 8-и процессах. Вывод номеров процессов (состав собранного массива а) соответствует схеме, приведенной на рис. 153. Также для примера на этом же скрине приведен запуск программы на 14 процессов.

```
mc - stud@hpchead:~/M182/serg/15
[stud@hpchead 15]$ mpiexec -n 8 -f ./machines ./a.out
Rank=0 rank_comm=0 a: 0 1 2
Rank=1 rank_comm=0 a: 0 1 2
Rank=2 rank_comm=0 a: 0 1
[stud@hpchead 15]$ mpiexec -n 14 -f ./machines ./a.out
Rank=0 rank_comm=0 a: 0 1 2 3 4
Rank=1 rank_comm=0 a: 0 1 2 3 4
Rank=2 rank_comm=0 a: 0 1 2 3
[stud@hpchead 15]$
```

Рис. 157. Скрин запуска программы

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10.    MPI_Group g1,g2;
11.    MPI_Comm_group(MPI_COMM_WORLD,&g);
12.    int n1=6,n2=5, R1=-1, R2=-1;
13.    int *m1 = new int[n1];
14.    int *m2 = new int[n2];
15.    for (int i = 0; i < n1; i++)
16.        m1[i] = i;
17.    for (int i = 0; i < n2; i++)
18.        m2[i] = i+5;
19.    MPI_Group_incl(g,n1,m1,&g1);
20.    MPI_Group_incl(g,n2,m2,&g2);
21.    MPI_Comm c1,c2;
22.    MPI_Comm_create(MPI_COMM_WORLD,g1,&c1);
23.    MPI_Comm_create(MPI_COMM_WORLD,g2,&c2);
24.    MPI_Group_free(&g1);
25.    MPI_Group_free(&g2);
26.    if(c1!=MPI_COMM_NULL) MPI_Comm_rank(c1,&R1);
27.    if(c2!=MPI_COMM_NULL) MPI_Comm_rank(c2,&R2);
28.    printf("Rank=%d R1=%d R2=%d\n",rank,R1,R2);
29.    if(c1!=MPI_COMM_NULL) MPI_Comm_free(&c1);
30.    if(c2!=MPI_COMM_NULL) MPI_Comm_free(&c2);
31.    MPI_Finalize();
32.    return 0;
33. }
```

Рис. 158. Листинг программы

Пример 4. Создание коммуникатор на основе групп процессов.

В данном примере дополним программу из примера 1 созданием коммуникаторов, а также их использованием в коммуникационных функциях.

rank	0	1	2	3	4	5	6	7	8	9
R1	0	1	2	3	4	5	-	-	-	-
R2	-	-	-	-	-	0	1	2	3	4

Алгоритм следующий (рис. 158):

Создание двух групп с пересечением на 5-м процессе (строки 19-20).

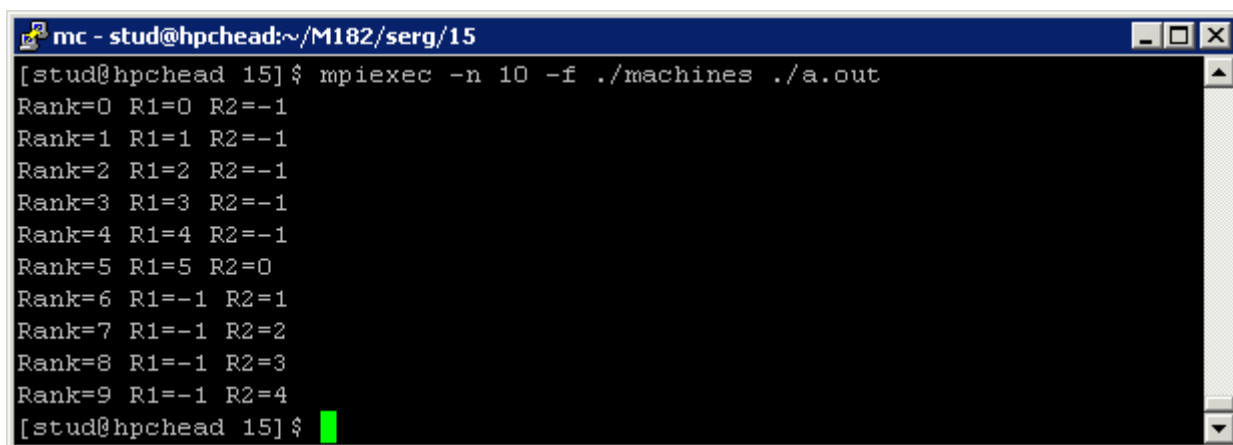
Создание двух коммуникаторов на основе созданных групп (строки 22-23).

Определение номеров процессов в рамках созданных коммуникаторов (строки 26-27).

Вывод номера процесса в рамках коммуникатора MPI_COMM_WORLD, номера процесса в рамках созданных коммуникаторов.

На рис. 159 приведен скрин запуска программы на 10-и процессах.

Следует обратить внимание на то, что если обратиться к коммуникатору на процессе, на котором он не определен, то это приведет к ошибке. Именно по этой причине любое обращение к коммуникатору должно предваряться проверкой его существования. Например, в строке 26 определяется номер процесса в рамках 1-го коммуникатора, соответственно надо эту строку программы выполнить только на процессах, принадлежащих именно 1-му коммуникатору (if(c1!=MPI_COMM_NULL)).



```
mc - stud@hpchead:~/M182/serg/15
[stud@hpchead 15] $ mpiexec -n 10 -f ./machines ./a.out
Rank=0 R1=0 R2=-1
Rank=1 R1=1 R2=-1
Rank=2 R1=2 R2=-1
Rank=3 R1=3 R2=-1
Rank=4 R1=4 R2=-1
Rank=5 R1=5 R2=0
Rank=6 R1=-1 R2=1
Rank=7 R1=-1 R2=2
Rank=8 R1=-1 R2=3
Rank=9 R1=-1 R2=4
[stud@hpchead 15] $
```

Рис. 159. Скрин запуска программы

Пример 5.

Дополним предыдущий пример использованием новых коммуникаторов в коммуникационных функциях.

Для этого реализуем следующий алгоритм (рис. 160):

Считывание значения переменной на 0-м процессе (строка 28).

Широковещательная рассылка данной переменной в рамках первого коммуникатора (строка 29).

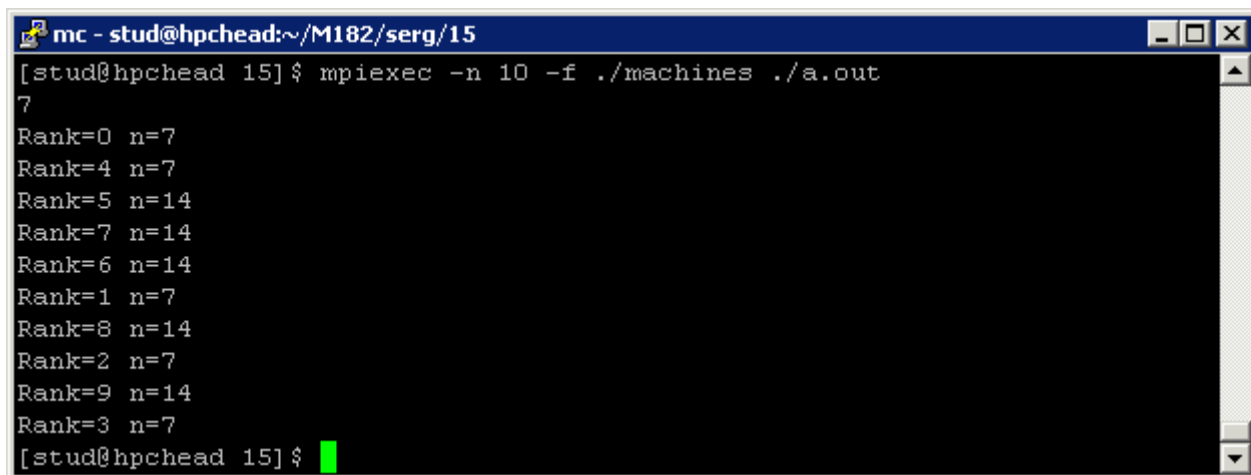
Удвоение значения переменной на 5-м процессе, принадлежащем одновременно обоим коммуникаторам (строка 30).

Широковещательная рассылка данной переменной в рамках второго коммуникатора (строка 31).

Вывод номера процесса в рамках коммуникатора MPI_COMM_WORLD и значения переменной n. С 0-го процесса по 4-й значение переменной должно быть равным введенному, на остальных – удвоенному. Это и демонстрирует скрин запуска программы, приведенной на рис. 161.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10.    MPI_Group g1,g2;
11.    MPI_Comm_group(MPI_COMM_WORLD,&g);
12.    int n1=6,n2=5, R1=-1, R2=-1,n;
13.    int *m1 = new int[n1];
14.    int *m2 = new int[n2];
15.    for (int i = 0; i < n1; i++)
16.        m1[i] = i;
17.    for (int i = 0; i < n2; i++)
18.        m2[i] = i+5;
19.    MPI_Group_incl(g,n1,m1,&g1);
20.    MPI_Group_incl(g,n2,m2,&g2);
21.    MPI_Comm c1,c2;
22.    MPI_Comm_create(MPI_COMM_WORLD,g1,&c1);
23.    MPI_Comm_create(MPI_COMM_WORLD,g2,&c2);
24.    MPI_Group_free(&g1);
25.    MPI_Group_free(&g2);
26.    if(c1!=MPI_COMM_NULL) MPI_Comm_rank(c1,&R1);
27.    if(c2!=MPI_COMM_NULL) MPI_Comm_rank(c2,&R2);
28.    if(rank==0) scanf("%d",&n);
29.    if(c1!=MPI_COMM_NULL) MPI_Bcast(&n,1,MPI_INT,0,c1);
30.    if(rank==5) n*=2;
31.    if(c2!=MPI_COMM_NULL) MPI_Bcast(&n,1,MPI_INT,0,c2);
32.    printf("Rank=%d n=%d \n",rank,n);
33.    if(c1!=MPI_COMM_NULL) MPI_Comm_free(&c1);
34.    if(c2!=MPI_COMM_NULL) MPI_Comm_free(&c2);
35.    MPI_Finalize();
36.    return 0;
37. }
```

Рис. 160. Листинг программы



```
mc - stud@hpchead:~/M182/serg/15
[stud@hpchead 15]$ mpirun -n 10 -f ./machines ./a.out
7
Rank=0 n=7
Rank=4 n=7
Rank=5 n=14
Rank=7 n=14
Rank=6 n=14
Rank=1 n=7
Rank=8 n=14
Rank=2 n=7
Rank=9 n=14
Rank=3 n=7
[stud@hpchead 15]$
```

Рис. 161. Скрин запуска программы

Задания

1. Используя функцию `MPI_Comm_dup`, создайте дубликат коммуникатора `MPI_COMM_WORLD` и продемонстрируйте его работоспособность на использовании функции широковещательной рассылки.
2. С помощью функций `MPI_Group_incl`, `MPI_Group_excl`, `MPI_Comm_create` создайте два коммуникатора, содержащих четные и нечетные процессы. Задайте на 0-х процессах каждого коммуникатора значение переменной `n` и разошлите его по процессам при помощи функции широковещательной рассылки.
3. С помощью функций `MPI_Group_incl`, `MPI_Group_excl`, `MPI_Comm_create` создайте два коммуникатора, содержащих четные и нечетные процессы. Сберите номера процессов каждого из коммуникаторов в массив и выведите данный массив (см. пример 3).
4. С помощью функций `MPI_Group_incl`, `MPI_Group_excl`, `MPI_Comm_create` создайте два коммуникатора, содержащих четные и нечетные процессы. Осуществите передачу данных по кольцу внутри каждого коммуникатора. В отчете приведите скрин трассы параллельного приложения.
5. Выполните предыдущее задания, используя функцию `MPI_Comm_split` вместо ранее задействованных трех.
6. Используя функцию `MPI_Comm_split`, создайте новый коммуникатор из процессов, ранг которых кратен 5. На каждом процессе с помощью датчика случайных чисел определите вещественную переменную. Определите при помощи функции редукции (`MPI_Reduce`) минимум и максимум сгенерированных вещественных чисел в рамках каждого из пяти коммуникаторов.

16. Логические топологии процессов

Топология процессов является одним из необязательных атрибутов коммуникатора. По умолчанию предполагается линейная топология, в которой процессы пронумерованы в диапазоне от 0 до $\text{size}-1$, где size – число процессов в группе. Однако для многих задач линейная топология неадекватно отражает логику коммуникационных связей между процессами. Например, при численном моделировании физических пространственных процессов чаще всего для распараллеливания программного комплекса используется координатное разбиение расчетной области по процессам (параллелизм по данным). Вся расчетная область разбивается на кубы (регулярная расчетная сетка), каждый из которых обрабатывается одним из процессов. Во время счета необходим интенсивный обмен между процессами информацией о значениях исследуемых функций приграничных расчетных ячеек. Для организации обмена потребуется знание номеров процессов, обрабатывающих расчетные кубы, соседствующие с кубом, обрабатываемым текущим процессом. Другой пример – расчетная область разбивается на тетраэдры Делоне (нерегулярная сетка), для описания связности узлов данного разбиения используется графовая структура. В данном случае распараллеливать расчетную программу проще, используя логику деления графа на подграфы для обработки разными процессами. Соответственно и обмен данными происходит между процессами, которые обрабатывают соседствующие графы.

MPI предоставляет средства для создания достаточно сложных "виртуальных" топологий в виде графов, где узлы являются процессами, а грани – каналами связи между процессами. Конечно же, следует различать логическую топологию процессов, которую позволяет формировать MPI, и физическую топологию процессоров.

16.1. Декартова топология процессов

В стандарте MPI предусмотрены два вида топологий: декартова топология и топология в виде графа. Наиболее используемой является декартова топология, на ее рассмотрении мы и остановимся.

В идеале логическая топология процессов должна учитывать как алгоритм решения задачи, так и физическую топологию процессоров. Для очень широкого круга задач наиболее адекватной топологией процессов является двумерная или трехмерная сетка. Такие структуры полностью определяются числом измерений и количеством процессов вдоль каждого координатного направления, а также способом раскладки процессов на координатную сетку. В MPI, как правило, используется row-major нумерация процессов, то есть используется нумерация вдоль строки. На рис. 162

представлено соответствие между нумерациями 12 процессов в одномерной и двумерной (3x4) топологиях.

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

Рис. 162. Соотношение между идентификатором процесса (верхнее число) и координатами в двумерной сетке 3x4 (нижняя пара чисел)

Обобщением линейной и матричной топологий на произвольное число измерений является декартова топология. Для создания коммуникатора с декартовой топологией используется функция `MPI_Cart_create`. С помощью этой функции можно создавать топологии с произвольным числом измерений, причем по каждому измерению в отдельности можно накладывать периодические граничные условия. Таким образом, для одномерной топологии мы можем получить или линейную структуру, или кольцо – в зависимости от того, какие граничные условия будут наложены. Для двумерной топологии соответственно – либо прямоугольник, либо цилиндр, либо тор. Заметим, что не требуется специальной поддержки гиперкубовой структуры, поскольку она представляет собой n-мерный тор с двумя процессами вдоль каждого координатного направления.

Функция создания коммуникатора с декартовой топологией

`MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)`

Входные аргументы:

`comm_old` - родительский коммуникатор;
`ndims` - число измерений;
`dims` - массив размера `ndims`, в котором задается число процессов вдоль каждого измерения;
`periods` - логический массив размера `ndims` для задания граничных условий (`true` - периодические, `false` - непериодические);
`reorder` - логическая переменная указывает, производить перенумерацию процессов (`true`) или нет (`false`).

Выходные аргументы:

`comm_cart` - новый коммуникатор.

Функция является коллективной, то есть должна запускаться на всех процессах, входящих в группу коммуникатора `comm_old`. При этом, если какие-то процессы не попадают в новую группу, то для них возвращается результат `MPI_COMM_NULL`. В случае, когда размеры заказываемой сетки больше имеющегося в группе числа процессов, функция завершается

аварийно. Значение параметра `reorder=true` означает, что MPI будет пытаться перенумеровать идентификаторы процессов с целью оптимизации коммуникаций. Это означает, что в процессе вызова функции будут осуществлены массивованные служебные пересылки для определения физической близости процессов.

Остальные функции, которые будут рассмотрены в этом разделе, имеют вспомогательный или информационный характер.

Функция определения оптимальной конфигурации сетки

MPI_Dims_create(int nnodes, int ndims, int *dims)

Входные аргументы:

`nnodes` - общее число узлов в сетке;
`ndims` - число измерений;
`dims` - массив целого типа размерности `ndims`, в который помещается рекомендуемое число процессов вдоль каждого измерения.

Выходные аргументы:

`dims` - массив целого типа размерности `ndims`, в который помещается рекомендуемое число процессов вдоль каждого измерения.

На входе в процедуру в массив `dims` должны быть занесены целые неотрицательные числа. Если элементу массива `dims[i]` присвоено положительное число, то для этой размерности вычисление не производится (число процессов вдоль этого направления считается заданным). Вычисляются только те компоненты `dims[i]`, для которых перед обращением к процедуре были присвоены значения 0. Функция стремится создать максимально равномерное распределение процессов вдоль направлений, выстраивая их по убыванию, то есть для 12 процессов она построит трехмерную сетку 4 x 3 x 1. Результат работы этой процедуры может использоваться в качестве входного параметра для процедуры `MPI_Cart_create`.

Функции опроса характеристик декартовой топологии

MPI_Cartdim_get(MPI_Comm comm, int *ndims)

Входные аргументы:

`comm` - коммуникатор с декартовой топологией.

Выходные аргументы:

`ndim` - число измерений в декартовой топологии.

Функция возвращает число измерений в декартовой топологии `ndims` для коммуникатора `comm`. Результат может быть использован в качестве параметра для вызова функции `MPI_Cart_get`, которая служит для получения более детальной информации.

MPI_Cart_get(MPI_Comm comm, int ndims, int *dims, int *periods, int *coords)

Входные аргументы:

`comm` - коммуникатор с декартовой топологией;
`ndims` - число измерений.

Выходные аргументы:

dims - массив размера **ndims**, в котором возвращается число процессов вдоль каждого измерения;

periods - логический массив размера **ndims**, в котором возвращаются наложенные граничные условия; (**true** - периодические, **false** – непериодические);

coords - координаты в декартовой сетке вызывающего процесса.

Две следующие функции устанавливают соответствие между идентификатором процесса и его координатами в декартовой сетке. Под идентификатором процесса понимается его номер в исходной области связи, из которой была создана декартова топология.

Функция получения идентификатора процесса по его координатам
MPI_Cart_rank.

FORTRAN:

INTEGER COMM, COORDS(*), RANK, IERR

MPI_CART_RANK(COMM, COORDS, RANK, IERR)

C:

MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

Входные аргументы:

comm - коммуникатор с декартовой топологией;

coords - координаты в декартовой системе.

Выходные аргументы:

rank - идентификатор процесса.

Для измерений с периодическими граничными условиями будет выполняться приведение к основной области определения $0 \leq \text{coords}(i) < \text{dims}(i)$.

Функция определения координат процесса по его идентификатору
MPI_Cart_coords.

MPI_Cart_coords(MPI_Comm comm, int rank, int ndims, int *coords)

Входные аргументы:

comm - коммуникатор с декартовой топологией;

rank - идентификатор процесса;

ndim - число измерений.

Выходные аргументы:

coords - координаты процесса в декартовой топологии.

Во многих численных алгоритмах используется операция сдвига данных вдоль каких-то направлений декартовой решетки. В MPI существует специальная функция **MPI_Cart_shift**, реализующая эту операцию. Точнее говоря, сдвиг данных осуществляется с помощью функции **MPI_Sendrecv**, а функция **MPI_Cart_shift** вычисляет для каждого процесса параметры для функции **MPI_Sendrecv** (**source** и **dest**).

Функция определения номеров процессов отправителей и получателей сообщений

MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)

Входные аргументы:

comm - коммуникатор с декартовой топологией;
direction - номер измерения, вдоль которого выполняется сдвиг;
disp - величина сдвига (может быть как положительной, так и отрицательной).

Выходные аргументы:

rank_source - номер процесса, от которого должны быть получены данные;
rank_dest - номер процесса, которому должны быть посланы данные.

Номер измерения и величина сдвига не обязаны быть одинаковыми для всех процессов. В зависимости от граничных условий сдвиг может быть либо циклический, либо с учетом граничных процессов. В последнем случае для граничных процессов возвращается MPI_PROC_NULL либо для переменной rank_source, либо для rank_dest. Это значение также может быть использовано при обращении к функции MPI_sendrecv.

Другая, часто используемая операция - выделение в декартовой топологии подпространств меньшей размерности и связывание с ними отдельных коммуникаторов.

Функция выделения подпространства в декартовой топологии

MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)

Входные аргументы:

comm - коммуникатор с декартовой топологией;
remain_dims - логический массив размера ndims, указывающий, входит ли i-е измерение в новую подрешетку (remain_dims[i] = true).

Выходные аргументы:

newcomm - новый коммуникатор, описывающий подрешетку, содержащую вызывающий процесс.

Функция является коллективной. Действие функции проиллюстрируем следующим примером. Предположим, что имеется декартова решетка 2 x 3 x 4, тогда обращение к функции MPI_Cart_sub с массивом remain_dims (true, false, true) создаст три коммуникатора с топологией 2 x 4. Каждый из коммуникаторов будет описывать область связи, состоящую из 1/3 процессов, входивших в исходную область связи.

16.2. Примеры

Пример 1. Создание двумерной декартовой топологии процессов.

В данном примере создадим коммуникатор, соответствующий следующей двумерной декартовой топологии:

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

На рис. 163 приведен листинг программы:

Строка 10 – выделение памяти под описатель коммуникатора декартовой топологии.

Строки 11-13 – выделение памяти под вспомогательный массив, элементы которого задают количество процессов вдоль каждого направления в решетке процессов декартовой топологии.

Строки 14-16 – выделение памяти под вспомогательный массив, элементы которого определяют будет ли направление в решетке цикличным.

Строка 17 – создание декартовой топологии.

Строка 18 – выделение памяти под массив координат процессов в решетке декартовой топологии.

Строка 19 – определение координат процессов по номеру процесса.

Строка 20 – вывод номера процесса в коммуникаторе MPI_COMM_WORLD, а также его координат в декартовой топологии.

На рис. 164 приведен скрин запуска программы на 12-и процессах, запуск на меньшем количестве процессов приведет к аварийному останову программы.

```

1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10. MPI_Comm dec;
11. int *m=new int[2];
12. m[0]=3;
13. m[1]=4;
14. int *k=new int[2];
15. k[0]=1;
16. k[1]=1;
17. MPI_Cart_create(MPI_COMM_WORLD,2,m,k,0,&dec);
18. int *coords=new int[2];
19. MPI_Cart_coords(dec,rank,2,coords);
20. printf("rank=%d -> (%d,%d) \n",rank,coords[0],coords[1]);
21. MPI_Finalize();
22. return 0;
23. }
```

Рис. 163. Листинг программы

```
mc - stud@hpchead:~/M182/serg/16
[stud@hpchead 16]$ mpiexec -n 12 -f ./machines ./a.out
rank=0 -> (0,0)
rank=8 -> (2,0)
rank=9 -> (2,1)
rank=10 -> (2,2)
rank=11 -> (2,3)
rank=1 -> (0,1)
rank=2 -> (0,2)
rank=3 -> (0,3)
rank=4 -> (1,0)
rank=5 -> (1,1)
rank=6 -> (1,2)
rank=7 -> (1,3)
[stud@hpchead 16]$
```

Рис. 164. Скрин запуска программы

```

1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10.    MPI_Comm dec;
11.    int *m=new int[2];
12.    m[0]=3;
13.    m[1]=4;
14.    int *k=new int[2];
15.    k[0]=1;
16.    k[1]=1;
17.    MPI_Cart_create(MPI_COMM_WORLD,2,m,k,0,&dec);
18.    int *coords=new int[2];
19.    MPI_Cart_coords(dec,rank,2,coords);
20.    int rs,rd;
21.    MPI_Cart_shift(dec,1,1,&rs,&rd);
22.    if(rank==0)
23.        {printf("-----\n");
24.         printf("rank_source\t|trank\t|   rank_dest\n");
25.         printf("-----\n");
26.         }
27.    printf("\t%d\t|\t%d\t|\t%d\n",rs,rank,rd);
28.    MPI_Finalize();
29.    return 0;
30. }

```

Рис. 165. Листинг программы

Пример 2. Определение соседних процессов в декартовой топологии.

Добавим в предыдущую программу вызов функции MPI_Cart_shift для определения номеров соседних процессов: второй аргумент – направление, в котором определить соседей (0 – вертикальное, 1 - горизонтальной), третий аргумент – расстояние до соседей (1 – сосед слева, сосед справа).

Строки 22-27 (рис. 165) – вывод номера процесса в коммуникаторе MPI_COMM_WORLD, а также номера его соседей слева и справа.

На рис. 166 приведен скрин запуска программы на 12-и процессах, сопоставьте вывод программы с ранее приведенной схемой создания декартовой топологии.

```

mc - stud@hpchead:~/M182/serg/16
[stud@hpchead 16]$ mpiexec -n 12 -f ./machines ./a.out
-----
rank_source |      rank      |   rank_dest   |
-----
          3 |          0      |           1     |
          0 |          1      |           2     |
          1 |          2      |           3     |
          2 |          3      |           0     |
          7 |          4      |           5     |
         11 |          8      |           9     |
          8 |          9      |          10     |
          4 |          5      |           6     |
          5 |          6      |           7     |
          6 |          7      |           4     |
          9 |         10      |          11     |
         10 |         11      |           8     |
[stud@hpchead 16]$

```

Рис. 166. Скрин запуска программы

Пример 3. Передача данных по 3-м кольцам.

Добавим в предыдущую программу пересылку данных сразу по трем кольцам процессов, образованных из строк процессов декартовой топологии.

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

Строка 25 (рис. 167) – задает условие на 0-й столбец процессов в декартовой топологии. В нашем случае это будут процессы с номерами 0, 4, 8. Строка 28 – отправка $a = \text{rank}$ соседям справа ($0 \rightarrow 1$, $4 \rightarrow 5$, $8 \rightarrow 9$), затем в 29 строке прием сообщения от соседей слева. Так как декартова топология создана с цикличным замыканием, то соседом слева для 0-го процесса будет процесс с номером 3, для 4-го – 7, а для 8-го – 11.

Строки 33-35 – на всех остальных процессах прием сообщений от соседей слева, добавление к пересылаемой переменной значения своего ранга и отправка соседу справа.

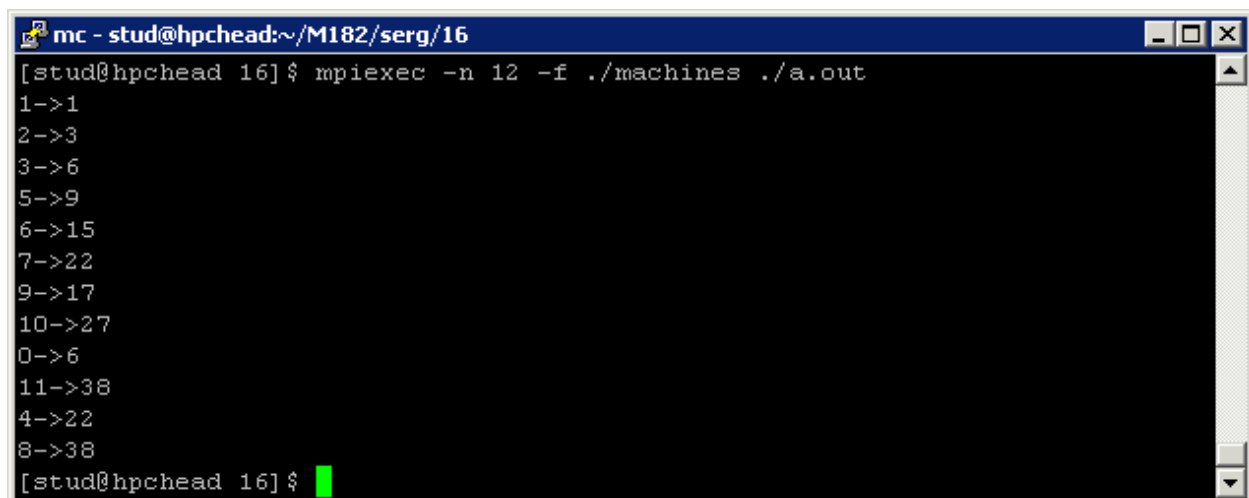
На рис. 168 приведен скрин запуска программы на 12-и процессах. 0-й процесс в итоге должен получить сумму номеров процессов первой строки, 4 – второй, 8 – третьей.

```

1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10.    MPI_Status stat;
11.    MPI_Comm dec;
12.    int *m=new int[2];
13.    m[0]=3;
14.    m[1]=4;
15.    int *k=new int[2];
16.    k[0]=1;
17.    k[1]=1;
18.    MPI_Cart_create(MPI_COMM_WORLD,2,m,k,0,&dec);
19.    int *coords=new int[2];
20.    MPI_Cart_coords(dec,rank,2,coords);
21.    //printf("rank=%d -> (%d,%d) \n",rank,coords[0],coords[1]);
22.    int rs,rd;
23.    MPI_Cart_shift(dec,1,1,&rs,&rd);
24.    int a;
25.    if (coords[1]==0)
26.    {
27.        a=rank;
28.        MPI_Send(&a,1,MPI_INT,rd,000,MPI_COMM_WORLD);
29.        MPI_Recv(&a,1,MPI_INT,rs,000,MPI_COMM_WORLD,&stat);
30.    }
31.    else
32.    {
33.        MPI_Recv(&a,1,MPI_INT,rs,000,MPI_COMM_WORLD,&stat);
34.        a+=rank;
35.        MPI_Send(&a,1,MPI_INT,rd,000,MPI_COMM_WORLD);
36.    }
37.    printf("%d->%d\n",rank,a);
38.    MPI_Finalize();
39.    return 0;
40. }

```

Рис. 167. Листинг программы



```

mc - stud@hpchead:~/M182/serg/16
[stud@hpchead 16]$ mpirun -n 12 -f ./machines ./a.out
1->1
2->3
3->6
5->9
6->15
7->22
9->17
10->27
0->6
11->38
4->22
8->38
[stud@hpchead 16]$

```

Рис. 168. Скрин запуска программы

Пример 4. Передача данных с использованием подрешеток декартовой топологии.

Изменим предыдущую программу – создадим подрешетки процессов, соответствующие строкам процессов в декартовой топологии. Затем, в каждой подрешетке вычислим сумму номеров процессов – результат на 1-м столбце процессов будет аналогичен результатам предыдущей программы.

Строка 19 (рис. 169) – создание вспомогательного массива, элементы которого указывают направление в декартовой топологии для создания подрешетки процессов.

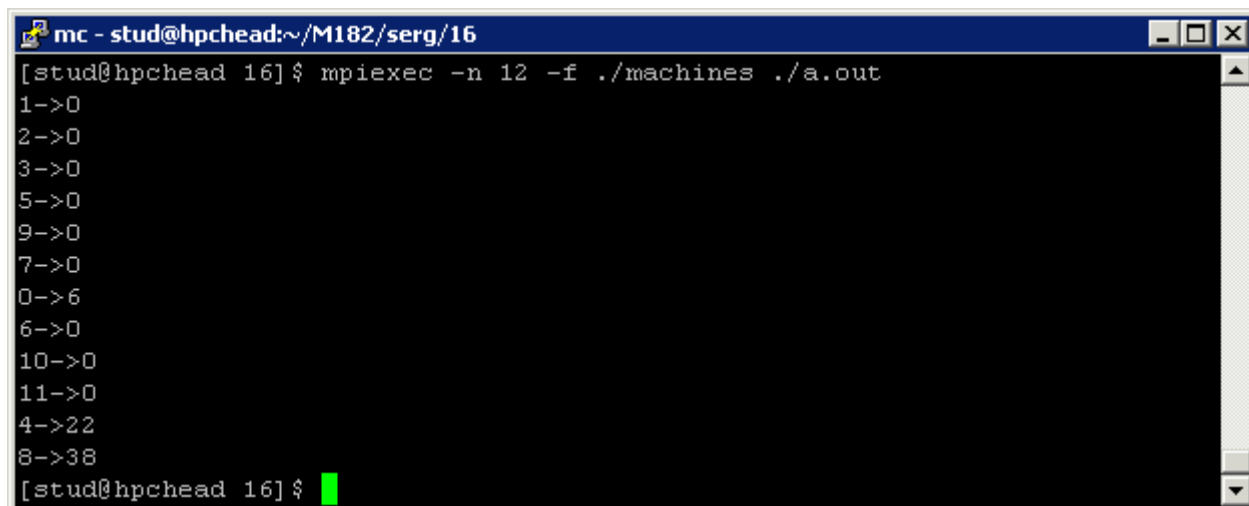
Строка 20 – создание коммуникаторов, объединяющих процессы строк декартовой топологии.

Строка 22 – выполнение коллективной операции редукции для суммирования номеров процессов в созданных коммуникаторах подрешеток. В нашем случае будет три подрешетки для каждой строки процессов.

На рис. 170 приведен скрин запуска программы на 12-и процессах. 0-й процесс в итоге должен получить сумму номеров процессов первой строки, 4 – второй, 8 – третьей.

```
1. #include <stdio.h>
2. #include "mpi.h"
3. int main(int argc, char *argv[])
4. {
5.     int rank;
6.     int size;
7.     MPI_Init(&argc, &argv);
8.     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
9.     MPI_Comm_size(MPI_COMM_WORLD, &size);
10.    MPI_Status stat;
11.    MPI_Comm dec, sub_cart;
12.    int *m=new int[2];
13.    m[0]=3;
14.    m[1]=4;
15.    int *k=new int[2];
16.    k[0]=1;
17.    k[1]=1;
18.    MPI_Cart_create(MPI_COMM_WORLD,2,m,k,0,&dec);
19.    int mass[2]={0,1};
20.    MPI_Cart_sub(dec,mass,&sub_cart);
21.    int a;
22.    MPI_Reduce(&rank,&a,1,MPI_INT,MPI_SUM,0,sub_cart);
23.    printf("%d->%d\n",rank,a);
24.    MPI_Finalize();
25.    return 0;
26. }
```

Рис. 169. Листинг программы



```
mc - stud@hpchead:~/M182/serg/16
[stud@hpchead 16]$ mpiexec -n 12 -f ./machines ./a.out
1->0
2->0
3->0
5->0
9->0
7->0
0->6
6->0
10->0
11->0
4->22
8->38
[stud@hpchead 16]$
```

Рис. 170. Скрин запуска программы

Задания

1. С помощью функции `MPI_Cart_create` создайте логическую декартовую топологию из 25 процессоров: 5×5 . Определите координаты процессов и номера соседних процессов по каждому направлению. Вывод сделайте следующий: номер процесса в коммуникаторе `MPI_COMM_WORLD`, координаты процесса в декартовой топологии, номера процессов слева-справа, сверху-снизу.
2. Добавьте в программу объявление двумерного массива на 0-м процессе `a[5][5]`. При помощи функций `MPI_Send`-`MPI_Recv` и `MPI_Cart_shift` разошлите строки матрицы по первому столбцу процессов в декартовой топологии. Например, 1-ю строку матрицы 0-й процесс отправляет процессу с координатами (1, 0), 2-ю строку – (2,0), ..., 4-ю строку – (4,0). Затем, каждый процесс 0-го столбца процессов декартовой топологии рассылает строку по одному элементу каждому процессу вдоль строки процессов в декартовой топологии. Например, на 0-й процесс должен послать процессу с координатами (0, 1) 1-й элемент массива, процессу (0, 2) – 2-й элемент массива и т.д. В результате такой рассылки каждый процесс получит 1 элемент начального массива, индексы которого совпадут с координатами процесса в декартовой топологии.
3. Выполните вторую задачу, создав коммуникаторы подрешеток процессов строк и столбцов декартовой топологии, а также коллективную коммуникационную функцию `MPI_Scatter`. Сначала рассылку выполнить матрицы построчно по столбцу процессов, затем – рассылку строки по 1-му элементу каждому по строке процессов.

Заключение

Наиболее популярной технологией параллельного программирования на кластерных системах является технология передачи сообщений, реализованная в библиотеке MPI. Вместе с тем, MPI считается ассемблером параллельного программирования ввиду необходимости в программе явно распределять вычисления по процессам, а также осуществлять пересылки между ними для организации совместной работы над единой вычислительной задачей. При этом, одни и те же пересылки возможно осуществить различными функциями библиотеки MPI – их выбор остается за программистом. Зачастую последовательная программа подвергается существенной модификации в процессе распараллеливания.

Вполне вероятно, что в будущем эпоха ”кремниевого чипа” завершится и на смену кластерным системам, занимающим огромные площади, придут новые, не уступающие по производительности, компактные вычислительные машины, для написания программ на которых можно будет обойтись без параллельного программирования. Однако, пока не наступило это замечательное время, нам придется довольствоваться существующими технологиями.

Материалы практикума прошли апробацию на протяжении ряда лет при ведении дисциплин ”Технологии параллельного программирования”, ”Технологии параллельных вычислений”, проводимых для обучающихся направлений 01.03.02 Прикладная математика и информатика, 02.03.02 Фундаментальная информатика и информационные технологии, 02.03.03 Математическое обеспечение и администрирование информационных систем.

Проблеме подготовки специалистов в области параллельных вычислений в России в последние годы уделяется особое внимание. В рамках реализации Проекта комиссии Президента РФ по модернизации и технологическому развитию экономики России «Создание системы подготовки высококвалифицированных кадров в области суперкомпьютерных технологий и специализированного программного обеспечения» создана сеть научно-образовательных центров, расположенных в федеральных округах РФ [19]. Научно-образовательные центры созданы на базе структурных подразделений учреждений высшего профессионального образования, входящих в Суперкомпьютерный консорциум университетов России, которые обладают значительным опытом выполнения научно-исследовательских работ и ведения образовательной деятельности в области суперкомпьютерных технологий (СКТ). Основными задачами системы НОЦ СКТ являются: подготовка, переподготовка и повышение квалификации специалистов по приоритетным и перспективным направлениям

суперкомпьютерных технологий и специализированного программного обеспечения; повышение эффективности научных исследований путем объединения усилий и ресурсов; осуществление инновационной деятельности в научной и образовательной сферах совместно с организациями науки, промышленности и бизнеса. В рамках реализации данного проекта была подготовлена серия учебников “Суперкомпьютерное образование”, выпускаемая издательством Московского университета. К настоящему моменту выпущено более 20 учебников, подготовленных известными в области СКТ учеными России [4, 6, 9, 10, 17 и др.]. Посмотреть полный перечень серии учебников можно на сайте издательства МГУ [20].

Автор надеется, что данный практикум, содержащий множество примеров параллельных программ с подробным их разбором, сделает процесс изучения технологии параллельного программирования более легким, а само издание послужит вкладом в общее дело по подготовке специалистов в области СКТ.

Литература

1. Афанасьев, К.Е. Основы высокопроизводительных вычислений. Том 1: Высокопроизводительные вычислительные системы: учебное пособие [Текст] / К.Е. Афанасьев, С.Ю. Завозкин, С.Н. Трофимов, А.Ю. Власенко. – Кемерово: КемГУ, – 2011. – 228 с.
2. Афанасьев, К.Е. Основы высокопроизводительных вычислений. Том II: Технологии параллельного программирования: учебное пособие [Текст] / К.Е. Афанасьев, С.В. Стуколов, В.В. Малышенко, С.Н. Карабцев. – Кемерово: КемГУ, – 2012. – 412 с.
3. Афанасьев, К.Е. Основы высокопроизводительных вычислений. Т. III: Параллельные вычислительные алгоритмы: учебное пособие [Текст] / К.Е. Афанасьев, И.В. Григорьева, Т.С. Рейн. – Кемерово: КемГУ, 2012. – 185с.
4. Воеводин, В. В. Вычислительная математика и структура алгоритмов [Текст] / В. В. Воеводин. – М.: МГУ, 2010. – 168 с.
5. Вычислительный кластер Гидрометцентра России [Электронный ресурс] // <https://meteoinfo.ru/novosti/15610-rosgidromet-predstavil-vysokoproizvoditelnyj-vychislitelnyj-klaster>. – 31.01.2020.
6. Гергель, В. П. Высокопроизводительные вычисления для многоядерных многопроцессорных систем [Текст] / В. П. Гергель. – М.: МГУ, 2010. – 544 с.
7. Интернет университет суперкомпьютерных технологий [Электронный ресурс] // <http://www.hpcu.ru>. – 23.11.2019.
8. Интернет-центр системы образовательных ресурсов в области СКТ [Электронный ресурс] // <http://hpc-education.ru>. – 23.05.2019.
9. Корняков, К. В. Инструменты параллельного программирования в системах с общей памятью [Текст] / К. В. Корняков, В. Д. Кустикова, И. Б. Мееров и др. – М.: МГУ, 2010. – 272 с.
10. Линев, А. В. Технологии параллельного программирования для процессоров новых архитектур [Текст] / А. В. Линев, Д. К. Боголепов, С. И. Бастраков. – М.: МГУ, 2010. – 160 с.
11. Набор тестов производительности вычислительных систем [Электронный ресурс] // <http://www.netlib.org/benchmark>. – 23.01.2020.
12. Некоторая статистика 54-ой редакции списка мощнейших компьютеров мира [Электронный ресурс] // https://parallel.ru/news/top500_54edition.html. - 21.01.2020
13. О программе «Университетский кластер» [Электронный ресурс] // <http://www.unicluster.ru/about.html>. - 21.01.2020
14. РБК. Технологии и медиа: Сбербанк объявил о создании самого мощного в России суперкомпьютера [Электронный ресурс] //

https://www.rbc.ru/technology_and_media/08/11/2019/5dc45b3d9a79475ce6cda125. – 31.01.2020.

15. Рейтинг самых мощных суперкомпьютерных систем мира [Электронный ресурс] // <http://www.top500.org>. – 23.11.2019.
16. Рейтинг самых мощных суперкомпьютерных систем СНГ [Электронный ресурс] // <http://supercomputers.ru>. – 23.11.2019.
17. Старченко, А. В. Практикум по методам параллельных вычислений [Текст] / А. В. Старченко, Е. А. Данилкин, В. И. Лаева, С. А. Проханов. – М.: МГУ, 2010. – 200 с.
18. Суперкомпьютер “Ломоносов-2” [Электронный ресурс] // <https://parallel.ru/cluster/lomonosov2.html>. – 31.01.2020.
19. Суперкомпьютерное образование: Интернет-центр системы образовательных ресурсов в области СКТ [Электронный ресурс] // <http://hpc-education.ru>. – 22.12.2019.
20. Суперкомпьютерное образование: Издательский дом МГУ. [Электронный ресурс] // <https://msupress.com/catalogue/books/knizhnye-serii/superkompyuternoe-obrazovanie/>. – 15.01.2020.
21. Установка MPICH2 в CodeBlocks. [Электронный ресурс] / http://scarabis.ucoz.ru/publ/ustanovka_mpich2_v_codeblocks/1-1-0-2?id=738. 20.01.2020.
22. Центр коллективного пользования «Высокопроизводительные параллельные вычисления» Кемеровского государственного университета (ЦКП по ВПВ) [Электронный ресурс] / <http://icp2.kemsu.ru>. 20.01.2020.
23. High-Performance Portable MPI. [Электронный ресурс] / <http://www.mpich.org>. – 26.11.2019.
24. MPICH-2 [Электронный ресурс] / Argone National Laboratory // <http://www.mcs.anl.gov/research/projects/mpich2>. – 23.11.2019.
25. MPI Forum. MPI-2: a message-passing interface standard [Текст] // International Journal of High Performance Computing Applications, 1998. – Vol. 12, – P. 1-299.
26. ssh-клиент для терминального доступа на кластер [Электронный ресурс] / <http://the.earth.li/~sgtatham/putty/latest/x86/putty.exe>. – 23.01.2011.

Приложение 1. Варианты контрольных работ

Вариант 1

1. Используя функции Send-Recv, осуществите передачу значения переменной с 0-го процесса на 1-й и обратно. Определите имена узлов, задействованных в передаче данных.
2. Используя блокирующие коммуникационные функции типа “Точка-Точка”, создайте следующую параллельную программу: на 0-м процессе считывается значение переменной n , которое передается всем остальным процессам.
3. Используя коллективные коммуникационные или вычислительные операции, создайте следующую параллельную программу: на всех процессах задается одномерный массив ($a_i = \text{rank}$, $i = 0 \dots 10$), который передается на 0-й процесс, 0-й процесс принимает посылку и накапливает сумму в вектор s .
4. Используя конструкторы карт размещения данных в оперативной памяти, создайте следующую параллельную программу передачи данных: на 0-м процессе задается одномерный массив ($a_i = i$, $i = 0 \dots 20$), с 0-го на 1-й процесс требуется передать одной посылкой 0, 1, 2, 6, 7, 8, 12, 13, 14 элементы массива.
5. Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Предположим, что последовательный алгоритм обладает максимальной степенью параллелизма и состоит из N^3 арифметических операций, а для сборки результата требуется N^2 пересылок данных.
 - 5.1. Подсчитайте время $T_1(n)$ на реализацию алгоритма на однопроцессорной машине.
 - 5.2. Определите время, затраченное на реализацию параллельного алгоритма $T_p(n)$, если число процессоров p равно 2.
 - 5.3. Вычислите ускорение параллельного алгоритма по сравнению с последовательным.

Вариант 2

1. Используя функции Send-Recv, осуществите передачу значения переменной с 0-го процесса на 1-й и обратно. Определите время, затраченное на передачу данных.
2. Используя блокирующие коммуникационные функции типа “Точка-Точка”, создайте следующую параллельную программу: на всех процессах задается значение переменной n , которое затем передается на 0-й процесс, 0-й процесс принимает посылку и накапливает сумму в переменную s .
3. Используя коллективные коммуникационные или вычислительные операции, создайте следующую параллельную программу: на 0-м процессе задается значение переменной n , которое передается всем процессам.
4. Используя конструкторы карт размещения данных в оперативной памяти, создайте следующую параллельную программу передачи данных: на 0-м процессе задается одномерный массив ($a_i=i, i=0\dots20$), с 0-го на 1-й процесс требуется передать одной посылкой 1, 5, 9, 13, 17 элементы массива.
5. Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Алгоритм суммирования ряда чисел.
 - 5.1. Подсчитайте время $T_1(n)$ на реализацию алгоритма на однопроцессорной машине.
 - 5.2. Определите степень параллелизма алгоритма.
 - 5.3. Определите время, затраченное на реализацию параллельного алгоритма $T_p(n)$, если число процессоров p равно 2.
 - 5.4. Вычислите ускорение параллельного алгоритма по сравнению с последовательным.

Вариант 3

1. Используя функции Send-Recv, осуществите передачу значения переменной с 0-го процесса на 2-й и обратно. Определите имена узлов, задействованных в передаче данных.
2. Используя блокирующие коммуникационные функции типа “Точка-Точка”, создайте следующую параллельную программу: на всех процессах задается значение переменной n , которое затем передается на 0-й процесс, 0-й процесс принимает посылки и сохраняет полученные значения в вектор s .
3. Используя коллективные коммуникационные или вычислительные операции, создайте следующую параллельную программу: на всех процессах задается одномерный массив ($a_i = \text{rank}$, $i = 0 \dots 10$), который передается на 0-й процесс, 0-й процесс принимает посылки и сохраняет полученные вектора построчно в 2-мерный массив s .
4. Используя конструкторы карт размещения данных в оперативной памяти, создайте следующую параллельную программу передачи данных: на 0-м процессе задается одномерный массив ($a_i = i$, $i = 0 \dots 20$), с 0-го на 1-й процесс требуется передать одной посылкой 0, 4, 8, 12, 16 элементы массива.
5. Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Алгоритм скалярного произведения векторов.
 - 5.1. Подсчитайте время $T_1(n)$ на реализацию алгоритма на однопроцессорной машине.
 - 5.2. Определите степень параллелизма алгоритма.
 - 5.3. Определите время, затраченное на реализацию параллельного алгоритма $T_p(n)$, если число процессоров p равно 2.
 - 5.4. Вычислите ускорение параллельного алгоритма по сравнению с последовательным.

Вариант 4

1. Используя функции Send-Recv, осуществите передачу значения переменной с 0-го процесса на 2-й и обратно. Определите время, затраченное на передачу данных.
2. Используя блокирующие коммуникационные функции типа “Точка-Точка”, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots10$), который передается всем остальным процессам.
3. Используя коллективные коммуникационные или вычислительные операции, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots10$), который передается всем процессам.
4. Используя конструкторы карт размещения данных в оперативной памяти, создайте следующую параллельную программу передачи данных: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots20$), с 0-го на 1-й процесс требуется передать одной посылкой 1, 5, 9, 13, 17 элементы массива.
5. Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Алгоритм суммирования векторов.
 - 5.1. Подсчитайте время $T_1(n)$ на реализацию алгоритма на однопроцессорной машине.
 - 5.2. Определите степень параллелизма алгоритма.
 - 5.3. Определите время, затраченное на реализацию параллельного алгоритма $T_p(n)$, если число процессоров p равно 2.
 - 5.4. Вычислите ускорение параллельного алгоритма по сравнению с последовательным.

Вариант 5

1. Используя функции Send-Recv, осуществите передачу значения переменной с 0-го процесса на 3-й и обратно. Определите имена узлов, задействованных в передаче данных.
2. Используя блокирующие коммуникационные функции типа “Точка-Точка”, создайте следующую параллельную программу: на всех процессах задается одномерный массив ($a_i = \text{rank}$, $i = 0 \dots 10$), который передается на 0-й процесс, 0-й процесс принимает посылку и накапливает сумму в вектор s .
3. Используя коллективные коммуникационные или вычислительные операции, создайте следующую параллельную программу: на всех процессах задается значение переменной n , которое затем передается на 0-й процесс, 0-й процесс принимает посылки и сохраняет полученные значения в вектор s .
4. Используя конструкторы карт размещения данных в оперативной памяти, создайте следующую параллельную программу передачи данных: на 0-м процессе задается одномерный массив ($a_i = i$, $i = 0 \dots 20$), с 0-го на 1-й процесс требуется передать одной посылкой 0, 3, 6, 9, 12, 15, 18 элементы массива.
5. Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Алгоритм суммирования квадратных матриц.
 - 5.1. Подсчитайте время $T_1(n)$ на реализацию алгоритма на однопроцессорной машине.
 - 5.2. Определите степень параллелизма алгоритма.
 - 5.3. Определите время, затраченное на реализацию параллельного алгоритма $T_p(n)$, если число процессоров p равно 2.
 - 5.4. Вычислите ускорение параллельного алгоритма по сравнению с последовательным.

Вариант 6

1. Используя функции Send-Recv, осуществите передачу значения переменной с 0-го процесса на 3-й и обратно. Определите время, затраченное на передачу данных.
2. Используя блокирующие коммуникационные функции типа “Точка-Точка”, создайте следующую параллельную программу: на всех процессах задается одномерный массив ($a_i = i$, $i = 0 \dots 10$), который передается на 0-й процесс, 0-й процесс принимает посылки и сохраняет полученные вектора построчно в 2-мерный массив s .
3. Используя коллективные коммуникационные или вычислительные операции, создайте следующую параллельную программу: на всех процессах задается значение переменной n , которое затем передается на 0-й процесс, 0-й процесс принимает посылку и накапливает сумму в переменную s .
4. Используя конструкторы карт размещения данных в оперативной памяти, создайте следующую параллельную программу передачи данных: на 0-м процессе задается одномерный массив ($a_i = i$, $i = 0 \dots 20$), с 0-го на 1-й процесс требуется передать одной посылкой 1, 4, 7, 10, 13, 16, 19 элементы массива.
5. Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Алгоритм умножения квадратной матрицы на вектор.
 - 5.1. Подсчитайте время $T_1(n)$ на реализацию алгоритма на однопроцессорной машине.
 - 5.2. Определите степень параллелизма алгоритма.
 - 5.3. Определите время, затраченное на реализацию параллельного алгоритма $T_p(n)$, если число процессоров p равно 2.
 - 5.4. Вычислите ускорение параллельного алгоритма по сравнению с последовательным.

Вариант 7

1. Используя функции Send-Recv, осуществите передачу значения переменной с 0-го процесса на 4-й и обратно. Определите имена узлов, задействованных в передаче данных.
2. Используя блокирующие коммуникационные функции типа “Точка-Точка”, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots10$), начиная с 5-го элемента, одной посылкой отправляются 5 элементов на 1-й процесс, который сохраняет полученную посылку, начиная со 2-го элемента.
3. Используя коллективные коммуникационные или вычислительные операции, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots20$), который по 2-а элемента рассылается всем остальным процессам.
4. Используя конструкторы карт размещения данных в оперативной памяти, создайте следующую параллельную программу передачи данных: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots20$), с 0-го на 1-й процесс требуется передать одной посылкой 0, 8, 16 элементы массива.
5. Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Предположим, что последовательный алгоритм обладает максимальной степенью параллелизма и состоит из N арифметических операций, а для сборки результата требуется $N/2$ пересылок данных.
 - 5.1. Подсчитайте время $T_1(n)$ на реализацию алгоритма на однопроцессорной машине.
 - 5.2. Определите время, затраченное на реализацию параллельного алгоритма $T_p(n)$, если число процессоров p равно 2.
 - 5.3. Вычислите ускорение параллельного алгоритма по сравнению с последовательным.

Вариант 8

1. Используя функции Send-Recv, осуществите передачу значения переменной с 0-го процесса на 4-й и обратно. Определите время, затраченное на передачу данных.
2. Используя блокирующие коммуникационные функции типа “Точка-Точка”, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots10$), начиная с 5-го элемента, одной посылкой отправляются 5 элементов на 1-й процесс, который сохраняет полученную посылку, начиная со 0-го элемента.
3. Используя коллективные коммуникационные или вычислительные операции, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots20$), который по 4-е элемента рассылается всем остальным процессам.
4. Используя конструкторы карт размещения данных в оперативной памяти, создайте следующую параллельную программу передачи данных: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots20$), с 0-го на 1-й процесс требуется передать одной посылкой 0, 1, 4, 5, 8, 9, 12, 13, 16, 17 элементы массива.
5. Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Предположим, что последовательный алгоритм обладает максимальной степенью параллелизма и состоит из N^2 арифметических операций, а для сборки результата требуется $N/2$ пересылок данных.
 - 5.1. Подсчитайте время $T_1(n)$ на реализацию алгоритма на однопроцессорной машине.
 - 5.2. Определите время, затраченное на реализацию параллельного алгоритма $T_p(n)$, если число процессоров p равно 2.
 - 5.3. Вычислите ускорение параллельного алгоритма по сравнению с последовательным.

Вариант 9

1. Используя функции Send-Recv, осуществите передачу значения переменной с 0-го процесса на 5-й и обратно. Определите имена узлов, задействованных в передаче данных.
2. Используя блокирующие коммуникационные функции типа “Точка-Точка”, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots20$), который по 2-а элемента рассылается всем остальным процессам.
3. Используя коллективные коммуникационные или вычислительные операции, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots20$), который по 5 элементов рассылается всем остальным процессам.
4. Используя конструкторы карт размещения данных в оперативной памяти, создайте следующую параллельную программу передачи данных: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots20$), с 0-го на 1-й процесс требуется передать одной посылкой 1, 2, 5, 6, 9, 10, 13, 14, 17, 18 элементы массива.
5. Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Предположим, что последовательный алгоритм обладает максимальной степенью параллелизма и состоит из N^2 арифметических операций, а для сборки результата требуется N пересылок данных.
 - 5.1. Подсчитайте время $T_1(n)$ на реализацию алгоритма на однопроцессорной машине.
 - 5.2. Определите время, затраченное на реализацию параллельного алгоритма $T_p(n)$, если число процессоров p равно 2.
 - 5.3. Вычислите ускорение параллельного алгоритма по сравнению с последовательным.

Вариант 10

1. Используя функции Send-Recv, осуществите передачу значения переменной с 0-го процесса на 5-й и обратно. Определите время, затраченное на передачу данных.
2. Используя блокирующие коммуникационные функции типа “Точка-Точка”, создайте следующую параллельную программу: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots 20$), который по 4-е элемента рассылается всем остальным процессам.
3. Используя коллективные коммуникационные или вычислительные операции, создайте следующую параллельную программу: на 0-м процессе задается двумерный массив ($a_{ij}=10*(i+1)+(j+1)$, $i,j=0\dots 10$), который построчно рассылается всем остальным процессам.
4. Используя конструкторы карт размещения данных в оперативной памяти, создайте следующую параллельную программу передачи данных: на 0-м процессе задается одномерный массив ($a_i=i$, $i=0\dots 20$), с 0-го на 1-й процесс требуется передать одной посылкой 0, 1, 2, 3, 8, 9, 10, 11, 16, 17, 18, 19 элементы массива.
5. Предположим, что параллельная вычислительная система состоит из однородных узлов и для выполнения одной арифметической операции требуется время t , для пересылки значения переменной от одного узла другому требуется время αt . Предположим, что последовательный алгоритм обладает максимальной степенью параллелизма и состоит из N^3 арифметических операций, а для сборки результата требуется N пересылок данных.
 - 5.1. Подсчитайте время $T_1(n)$ на реализацию алгоритма на однопроцессорной машине.
 - 5.2. Определите время, затраченное на реализацию параллельного алгоритма $T_p(n)$, если число процессоров p равно 2.
 - 5.3. Вычислите ускорение параллельного алгоритма по сравнению с последовательным.

Приложение 2. Варианты индивидуальных заданий

Требования к выполнению индивидуальной работы

Индивидуальная работа направлена на самостоятельную работу по созданию параллельных программ и исследования эффективности проведенного распараллеливания.

При реализации алгоритмов индивидуального задания на параллельных компьютерах для анализа эффективности варианта распараллеливания необходимо проводить исследование по следующей схеме:

1. Написать последовательную программу, реализующую алгоритм решения задачи. Провести тестирование корректности работы программы на заранее известном решении.
2. Определить время, затраченное на реализацию последовательного алгоритма $T_1(n)$.
3. Написать параллельную программу и провести тестирование путем сравнения с результатами, полученными последовательной реализацией программы.
4. Определить время, затраченное на реализацию параллельной программы $T_p(n)$ (p – количество процессов).
5. Вычислить ускорение параллельного алгоритма по сравнению с последовательным $S_p(n) = \frac{T_1(n)}{T_p(n)}$.
6. Вычислить эффективность параллельного алгоритма $E_p(n) = \frac{S_p(n)}{p}$.
7. Подготовить отчет, в котором привести описание последовательного и параллельного алгоритмов; листинги программ, реализующих заданный алгоритм; таблицы и графики, демонстрирующие основные характеристики результатов выполнения программ на вычислительном кластере; трассировки параллельных программ и результаты анализа эффективности распараллеливания программы.

Вариант 1: Задача об “обедающих философах”

Данный вариант предполагает создание только параллельной реализации программы, соответственно проводить исследование эффективности распараллеливания не требуется.



Рис. 171.

Запрограммируйте параллельную программу, реализующую задачу “об обедающих философах”. Для реализации потребуется пять процессов.

Суть задачи следующая: пять философов сидят за круглым столом (рис. 171). Они проводят жизнь, чередуя приемы пищи и размышления. В центре стола находится большое блюдо спагетти. Философам, чтобы съесть порцию спагетти, требуется две вилки. Вилоч всего пять: между каждой парой философов лежит по одной вилке. Каждому философу позволительно пользоваться только вилками, которые лежат рядом с ним (слева и справа).

Задача – написать программу, моделирующую поведение философов. Программа должна избегать ситуации, в которой все философы голодны, то есть ни один из них не может взять себе две вилки (например, когда каждый философ держит по одной вилке и не хочет отдавать ее). Раз вилоч всего пять, то одновременно могут есть не более, чем двое философов. Два сидящих рядом философа не могут есть одновременно. Предположим, что периоды раздумий и приемов пищи различны – для их имитации в программе можно использовать генератор случайных чисел. Имитация поведения каждого философа может быть разбита на следующие блоки: поразмыслить, взять вилки, поесть, отдать вилки. Вилки являются разделяемым ресурсом. Запрограммируйте остановку алгоритма по достижении контрольного времени. Выведите в файл результатов общее время реализации параллельной программы, количество приемов пищи для каждого философа.

Реализацию программы допускается выполнить одним из двух вариантов: более сложным – когда процессы, имитирующие поведение философов, взаимодействуют только с соседними процессами, а также простой вариант реализации – когда есть 6-й процесс, с которым

взаимодействует каждый из 5-и процессов, имитирующих поведение философов, между собой никаких пересылок нет.

Для первого случая при реализации понадобятся неблокирующие функции передачи сообщений (MPI_Isend, MPI_Irecv), а также функции MPI_Test, MPI_Wait.

Для реализации программы, соответствующей второму случаю, достаточно использовать обычные блокирующие функции передачи сообщений.

В программе необходимо выделить четко выраженные периоды, за один из которых происходит всего одно событие на каждом из 5-и процессов. Другими словами, необходимо хоть как то синхронизировать работу алгоритма, в асинхронном варианте реализации довольно сложна.

Продумайте вывод, производимый каждым процессом по окончании каждого периода, позволяющий определить какое событие произошло за прошедший период. В качестве событий можно порекомендовать следующие: размышление, решение поест, решение взять правую (левую) вилку, отправка уведомления соседу о том, что в следующий период времени будет предпринята попытка взять вилку, проверка приемного буфера на предмет наличия сообщения от соседа на то, что он претендует взять вилку в следующий период времени и т.д.

Вариант 2: Задача о восьми ферзях

Запрограммируйте параллельную программу, реализующую задачу “о восьми ферзях”. Нужно расставить на шахматной доске восемь ферзей так, чтобы они не атаквали друг друга. Разработайте программу, которая строит все 92 решения этой задачи.

Данная задача довольно известна и на просторах Интернета можно найти много эффективных реализаций, которые даже без всякого распараллеливания получают ее решение за доли секунды. Например, очевидно, что никакие два ферзя не могут находиться на одной горизонтали. Написав программу, в которой каждый ферзь перемещается по своей горизонтали, Вы получите уже существенное ускорение по сравнению с полным перебором расстановок ферзей.

Вам же при реализации надо построить алгоритм полного перебора всех вариантов расстановки ферзей, исключая лишь случаи расстановки ферзей в одну клетку доски.

Возможно Вам будет проще реализовать данную программу, представив шахматную доску в виде одномерного массива в 64 клетки.

Параллельная программа должна найти все решения, передать с каждого процесса решения на 0-й процесс, который произведет их запись в файл для последующего анализа корректности работы программы.

Вариант 3: Задача о пяти ферзях

Принцип реализации аналогичен предыдущему варианту работы, а суть задачи заключается в следующем: требуется найти все расстановки пяти ферзей на шахматной доске, при которых каждое поле будет находиться под ударом одного из них.

Вариант 4: Задача о коммивояжере

Запрограммируйте параллельную программу, реализующую задачу “о коммивояжере”. Эта классическая задача имеет практическое применение, например, при планировании обслуживания населения городским общественным транспортом. Дано n городов и симметричная матрица $A(n \times n)$. Значением элемента $A(i, j)$ является расстояние между городами i и j . Коммивояжер начинает путь в городе 1 и должен посетить по одному разу каждый город, закончив свой путь снова в городе 1. Требуется найти путь, минимизирующий расстояние, которое придется проехать коммивояжеру, а результат сохранить в векторе $b(n)$. Значением вектора b должна быть перестановка целых чисел от 1 до n , соответствующая порядку посещения городов коммивояжером. В программе требуется перебрать все возможные варианты очередности посещения городов коммивояжером. Именно поиск всех перестановок составляет основную сложность при реализации данного варианта. Рекомендация использовать лексикографический алгоритм для нахождения перестановок. При параллельной реализации каждый процесс будет осуществлять поиск оптимального пути среди своей порции перестановок. В результате необходимо определить процесс, который обладает наилучшим найденным решением. Если этот процесс ненулевой, то передать решение на 0-й процесс. Так как количество вариантов перестановок представляет собой факториальную зависимость от количества городов, то максимальное количество городов для экспериментов надо подобрать таким образом, чтобы Ваша программа работала несколько минут.

Вариант 5: Задача “Игра в жизнь”

Запрограммируйте параллельную программу, реализующую задачу клеточный автомат “игра в жизнь”. Многие биологические и физические системы можно смоделировать в виде набора объектов, которые с течением времени циклически взаимодействуют и развиваются. Некоторые простейшие системы можно моделировать с помощью клеточных автоматов. Основная идея – разделить пространство физической или биологической задачи на отдельные клетки. Каждая клетка – это конечный автомат. После инициализации все клетки сначала совершают один переход в новое

состояние, затем второй переход и т.д. Результат каждого перехода зависит от текущего состояния клетки и ее соседей. Дано двумерное поле клеток. Каждая клетка либо содержит организм (жива), либо пуста (мертва). Каждая клетка имеет восемь соседей, которые расположены сверху, снизу, слева, справа и по четырем диагоналям от нее. Игра “жизнь” происходит следующим образом. Сначала поле инициализируется: определяются мертвые и живые клетки (для этой цели в программе можно использовать генератор случайных чисел). Затем каждая клетка проверяет состояние свое и своих соседей и изменяет свое состояние в соответствии со следующими правилами:

- живая клетка, возле которой меньше двух живых клеток, умирает от одиночества;
- живая клетка, возле которой есть две или три живые клетки, выживает еще на одно поколение;
- живая клетка, возле которой находится больше трех живых клеток, умирает от перенаселения;
- мертвая клетка, рядом с которой есть ровно три живых соседа, оживает.

Этот процесс повторяется заданное число шагов (поколений).

Игровое поле возьмите в виде квадрата $n \times n$. Для замеров времени размер поля поставьте от 100 до 1000. Количество поколений – 10000. В программе на каждом поколении определите размер популяции (количество живых клеток) и запишите в одномерный массив.

Следует обратить внимание, что даже у приграничных клеток также восемь соседей. Например, для клетки под номером 1 на приведенной схеме соседи выделены зеленым цветом, для клетки под номером 2 – синим.

2							
1							

Для параллельной реализации поле требуется разделить на полосы, при этом каждая полоса будет обрабатываться отдельным процессом. Для реализации алгоритма потребуется организовать передачу данных о состоянии пограничных клеток. При блокирующем обмене данной информации ускорение параллельной программы по сравнению с последовательной будет незначительным. Рекомендация использовать прием опережающего вычисления и опережающей рассылки с применением функций неблокирующего обмена.

Вариант 6: Сортировка последовательности чисел

Дан ряд случайных чисел, написать параллельную программу сортировки данных чисел. Алгоритм сортировки выберите любой – полный перебор, сортировка "пузырьком", Шелла. Для параллельной реализации сгенерированную последовательность необходимо равными блоками разослать по процессам, каждый из которых произведет сортировку своего блока чисел. Затем, из отсортированных частей массива необходимо собрать единый отсортированный массив. Можно использовать алгоритм сдвигания для сборки результирующего массива, можно отсылать 0-му процессу, который будет производить окончательную сборку. Выбор остается за Вами. При верной реализации параллельной программы ее ускорение имеет квадратичную зависимость от числа задействованных процессов. Объясните данный факт.

Вариант 7: Задача о многопроцессорном расписании (задача о кучах)

В литературе распространены оба названия одной и той же задачи. Программа на входе получает одномерный массив $a[n]$, элементы которого характеризуют вес камня. Требуется распределить эти камни по m кучам таким образом, чтобы вес куч был максимально одинаков (разница весов куч была минимальна). Данную задачу также необходимо выполнить, используя полный перебор всех вариантов расположения камней по кучам. Эта задача относится к NP полным задачам и при реализации рекомендуется задавать количество камней от 100 до 1000, а количество куч – от 2 до 5.

Также данная задача имеет более простое решение – сначала упорядочить массив весов камней по убыванию, а затем разложить камни по кучам по следующему правилу: очередной камень добавлять в самую легкую кучу. При такой реализации данная задача сводится к распараллеливанию алгоритма сортировки чисел.

Реализацию программы допускается выполнить одним из двух вариантов: более сложным с полным перебором всех вариантов расположения камней по кучам или простым – с использованием сортировки по убыванию весов камней.

Вариант 8: Определение частоты события

Дан текстовый файл, определить частоту встречи слов в тексте. В такой интерпретации на выходе требуется получить текстовый файл, содержащий словарь слов (упорядочен по алфавиту) с указанием количества сколько раз каждое встретилось в тексте.

Допустима реализация данного варианта в следующей трактовке: пусть 0 – означает пробел, цифры от 1 до k – буквы алфавита. В программе генерируется случайным образом n чисел от 0 до k – это будет текст. Вот его и надо разобрать на слова (цифры), разделенные пробелами (нулем или

нулями) и составить словарь с упорядочением во возрастанию чисел (слов) с указанием количества встреч в изначальной последовательности (тексте).

Вариант 9: Размен денег

Дана сумма денег, а также определенный набор монет. Определить сколько существует способов собрать необходимую сумму денег, используя доступные номиналы монет. Считается, что количество монет неограниченно. Например, есть монеты достоинством 1, 2, 3, 5, 10 рублей, а требуется определить все варианты наборов этих монет для получения в сумме 100 рублей.

В программе задается сумма, а также количество и номиналы монет. Программа должна определить не только количество способов, но и все решения записать в файл. Реализацию также предлагается выполнить простым перебором всех вариантов.

Вариант 10: Алгоритм Фокса перемножения матриц

Пусть перемножаемые матрицы A и B размера $n \times n$, в результате получается матрица C такого же размера. Будем также считать, что процессов $p = q \times q$, причем n кратно q . В алгоритме Фокса матрицы разделяются по процессам блоками в виде клеток шахматной доски. При этом все процессы представляются в виде виртуальной двумерной сетки $q \times q$, где каждому процессу принадлежит соответствующие блоки исходных и результирующей матриц.

Для параллельной реализации необходимо применение функций создания виртуальных декартовых топологий, а также подрешеток для организации пересылок блоков матриц между процессами.

Учебное электронное издание

СТУКОЛОВ Сергей Владимирович

ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ
практикум

16+

Редактор
Технический редактор И.О. Фамилия

Заказ № ??.

Подписано к использованию 2020.
Объем 4 Мб.

Кемеровский государственный университет,
650043, Кемерово, ул. Красная, 6.