Г. И. ШПАКОВСКИЙ
Н. В. СЕРИКОВА

ПРОГРАММИРОВАНИЕ

ДЛЯ МНОГОПРОЦЕССОРНЫХ

СИСТЕМ
В СТАНДАРТЕ

MPI

МИНСК
БГУ
2002

Настоящее пособие предназначено для практического обучения параллельному программированию в стандарте MPI (The Message Passing Interface). В пособии содержатся: общие сведения по параллельным системам и их программированию; полные справочные данные по библиотеке функций MPI; примеры программирования приложений (матричные задачи, решение ДУЧП, СЛАУ, криптоанализ); сведения по организации вычислений в различных исполнительных средах. Имеется большой объем приложений, включающий справочные материалы и примеры MPI программ.

Издание предназначено студентам естественнонаучных направлений, специалистам и научным работникам, заинтересованным в решении прикладных задач с большим объемом вычислений.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Настоящее пособие предназначено для практического обучения параллельному программированию в стандарте MPI (The Message Passing Interface). В пособии содержатся: общие сведения по параллельным системам и их программированию; полные справочные данные по библиотеке функций MPI; примеры программирования приложений, важных для многих областей науки и техники; сведения по организации вычислений в различных исполнительных средах.

MPI является библиотекой функций обмена данными между процессами, реализованная для языков C и Fortran. Головной организацией проекта MPI является Аргоннская национальная лаборатория США [1]. После появления первой версии стандарта MPI в мае 1994 года MPI получил широкое распространение. В настоящее время стандарт MPI адаптирован для большинства суперЭВМ и кластеров, в том числе и в России [2]. Благодаря простоте технической реализации кластеров на базе локальных сетей сотни университетов используют MPI для учебных и научных целей.

Стандарт MPI-1 использует статическое размещение процессов и данных по процессорам, а стандарт MPI-2 предназначен для динамического распределения работ. В настоящее время стандарт MPI-2 полностью не реализован и в основном используется стандарт MPI-1. Настоящее пособие построено на версии стандарта MPI-1.2 с добавлением привязок для языка C++. Все ошибки предыдущих версий исключены.

При создании пособия использовались следующие источники информации:

- Материалы сайта Аргоннской национальной лаборатории [1], где размещены: различные версии стандартов MPI-1 и MPI-2, документация и дистрибутивы для различных версий MPICH, параллельные библиотеки и много других материалов справочного и учебного характера.
- Материалы сайта Научно-исследовательского вычислительного центра МГУ [3].
- Книги и руководства, авторами которых являются основные разработчики библиотеки интерфейса MPI и его реализаций [4,5,6,7,8].
- Переводы, выполненные в 2001 году, в рамках программы Союзного государства СКИФ по разработке кластерных систем [9,10,11].

Все разделы пособия сопровождаются большим количеством примеров, контрольными вопросами, заданиями для самостоятельной работы. Поскольку MPI используется для программирования на языках C и Fortran, то и примеры даются попеременно на обоих языках. Для большинства заданий в приложении приведены готовые и протестированные программы-ответы на языке C. В приложении содержится только часть программ-ответов, это вызвано ограниченным объемом книги. Остальные ответы и многие другие материалы (дистрибутивы MPICH, параллельных библиотек, руководства по инсталляции и много другой документации) содержатся на сайте Белорусского государственного университета [12] или могут быть получены по электронной почте, адрес которой указан ниже.

Предполагается, что на начальном уровне обучение будет проводится на сетях персональных ЭВМ под управлением операционной системы Windows NT с использованием языка C, поэтому в приложении рассмотрены вопросы создания и настройки локальной сети на базе Windows NT (настройка сети, MPICH, среды языка C) для написания приложений.

Настоящее пособие увидело свет благодаря помощи многих людей. Постоянную поддержку в издании пособия оказывали А. Н. Курбацкий и С. Г. Мулярчик, много советов по содержанию книги дал М. К. Буза. Практическая проверка некоторых примеров выполнена студентами А. Е. Верхотуровым, А. Н. Гришановичем и А. В. Орловым. Глава 10 написана с участием Г. Ф. Астапенко. В обсуждении работы принимали участие А. С. Липницкий, В. Ф. Ранчинский, Г. К. Афанасьев и многие другие сотрудники кафедры информатики.

Авторы будут признательны всем, кто поделится своими соображениями по совершенствованию данного пособия. Русскоязычная терминология по MPI еще не устоялась, поэтому некоторые термины, использованные в пособии, могут оказаться не совсем удачными. Возможные предложения и замечания по этим и другим вопросам просим присылать по адресу:

Республика Беларусь
220050, Минск, проспект Франциска Скорины, 4
Белорусский государственный университет
Факультет радиофизики и электроники, кафедра информатики
E-mail: Serikova@bsu.by, Shpakovski@bsu.by

Р А З Д Е Л 1. **ВВЕДЕНИЕ**

Глава 1. ОРГАНИЗАЦИЯ ВЫЧИСЛЕНИЙ В МНОГОПРОЦЕССОРНЫХ СИСТЕМАХ

В главе 1 приведен обзор методов организации вычислений в современных многопроцессорных системах, получивших в последние годы широкое распространение, рассматривается классификация систем, эффективность параллельных вычислений, техническая реализация многопроцессорных систем и систем передачи данных, методы программирования и, наконец, делается переход к основному объекту настоящего издания – системе программирования в стандарте MPI.

1.1. КЛАССИФИКАЦИЯ МНОГОПРОЦЕССОРНЫХ СИСТЕМ

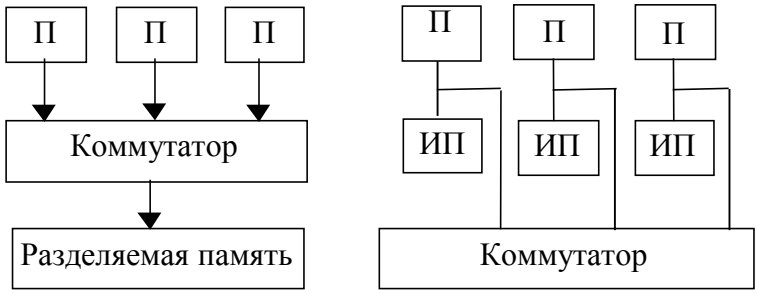
Наиболее известная классификация параллельных ЭВМ предложена Флинном [13] и отражает форму реализуемого ЭВМ параллелизма. Основными понятиями классификации являются "поток команд" и "поток данных". Под потоком команд упрощенно понимают последовательность команд одной программы. Поток данных – это последовательность данных, обрабатываемых одной программой. Согласно этой классификации имеется четыре больших класса ЭВМ:

1) ОКОД (одиночный поток команд – одиночный поток данных) или SISD (Single Instruction – Single Data). Это последовательные ЭВМ, в которых выполняется единственная программа, т. е. имеется только один счетчик команд.
2) ОКМД (одиночный поток команд – множественный поток данных) или SIMD (Single Instruction – Multiple Data). В таких ЭВМ выполняется единственная программа, но каждая команда обрабатывает массив данных. Это соответствует векторной форме параллелизма.
3) МКОД (множественный поток команд– одиночный поток данных) или MISD (Multiple Instruction – Single Data). Подразумевается, что в данном классе несколько команд одновременно работает с одним элементом данных, однако эта позиция классификации Флинна на практике не нашла применения.
4) МКМД (множественный поток команд – множественный поток данных) или MIMD (Multiple Instruction – Multiple Data). В таких ЭВМ одновременно и независимо друг от друга выполняется несколько программных ветвей, в определенные промежутки времени обменивающихся данными. Такие системы обычно называют

многопроцессорными. Далее будут рассматриваться только многопроцессорные системы.

Классы многопроцессорных систем. В основе МКМД-ЭВМ лежит традиционная последовательная организация программы, расширенная добавлением специальных средств для указания независимых фрагментов, которые можно выполнять параллельно. Параллельно-последовательная программа привычна для пользователя и позволяет относительно просто собирать параллельную программу из обычных последовательных программ.

МКМД-ЭВМ имеет две разновидности: ЭВМ с разделяемой (общей) и распределенной (индивидуальной) памятью. Структура этих ЭВМ представлена на рис. 1.1.



а *б*

Рис. 1.1. Структура ЭВМ с разделяемой (*а*)
и индивидуальной (*б*) памятью.

Здесь: П – процессор, ИП – индивидуальная память.

Главное различие между МКМД-ЭВМ с общей и индивидуальной памятью состоит в характере адресной системы. В машинах с разделяемой памятью адресное пространство всех процессоров является единым, следовательно, если в программах нескольких процессоров встречается одна и та же переменная *X*, то эти процессоры будут обращаться в одну и ту же физическую ячейку общей памяти. Это вызывает как положительные, так и отрицательные последствия.

Наличие общей памяти не требует физического перемещения данных между взаимодействующими программами, которые параллельно выполняются в разных процессорах. Это упрощает программирование и исключает затраты времени на межпроцессорный обмен.

Однако одновременное обращение нескольких процессоров к общим данным может привести к получению неверных результатов. Рассмотрим следующий пример. Пусть имеется система с разделяе-

мой памятью и двумя процессорами, каждый с одним регистром. Пусть в первом процессоре выполняется процесс $L1$, во втором – $L2$:

$$L\,1\colon \ldots X\colon = X + 1; \ldots$$
$$L\,2\colon \ldots X\colon = X + 1; \ldots$$

Процессы выполняются асинхронно, используя общую переменную X. При выполнении процессов возможно различное их взаиморасположение во времени, например, возможны следующие две ситуации:

$$\begin{array}{llllll} L1 & R1\colon = X; & R1\colon = R1 + 1; & X\colon = R1; \ldots & & \\ L2 & & R2\colon = X; & R2\colon = R2 + 1; & X\colon = R2; & \end{array} \qquad (1.1)$$

$$\begin{array}{llllll} L1 & R1\colon = X; & R1\colon = R1 + 1; & X\colon = R1; & & \\ L2 & & & & R2\colon = X; & R2\colon = R2 + 1; \quad X\colon = R2; \end{array} \qquad (1.2)$$

Пусть в начальный момент $X = V$. Тогда в случае (1.1) второй процессор производит чтение X до завершения всех операций в первом процессоре, поэтому $X = V + 1$.

В случае (1.2) второй процессор читает X после завершения всех операций первым процессором, поэтому $X = V + 2$. Таким образом, результат зависит от взаиморасположения процессов во времени, что для асинхронных процессов определяется случайным образом. Чтобы исключить такие ситуации, необходимо ввести систему синхронизации параллельных процессов (например, семафоры), что усложняет механизмы операционной системы.

Поскольку при выполнении каждой команды процессорам необходимо обращаться в общую память, то требования к пропускной способности коммутатора этой памяти чрезвычайно высоки, что ограничивает число процессоров в системах величиной 10...20.

В системах с индивидуальной памятью (с обменом сообщениями) каждый процессор имеет независимое адресное пространство, и наличие одной и той же переменной X в программах разных процессоров приводит к обращению в физически разные ячейки памяти. Это вызывает физическое перемещение данных между взаимодействующими программами в разных процессорах, однако, поскольку основная часть обращений производится каждым процессором в собственную память, то требования к коммутатору ослабляются, и число процессоров в системах с распределенной памятью и коммутатором типа гиперкуб может достигать нескольких сотен и даже тысяч.

1.2. СЕТЕВОЙ ЗАКОН АМДАЛА

Закон Амдала. Одной из главных характеристик параллельных систем является ускорение R параллельной системы, которое определяется выражением:

$$R = T_1 / T_n ,$$

где T_1 – время решения задачи на однопроцессорной системе, а T_n – время решения той же задачи на n – процессорной системе.

Пусть $W = W_{ск} + W_{пр}$, где W – общее число операций в задаче, $W_{пр}$ – число операций, которые можно выполнять параллельно, а $W_{ск}$ – число скалярных (нераспараллеливаемых) операций.

Обозначим также через t время выполнения одной операции. Тогда получаем известный *закон Амдала* [13]:

$$R = \frac{W \cdot t}{(W_{ск} + \frac{W_{пр}}{n}) \cdot t} = \frac{1}{a + \frac{1-a}{n}} \xrightarrow{n \to \infty} \frac{1}{a} . \qquad (1.3)$$

Здесь $a = W_{ск} / W$ – удельный вес скалярных операций.

Закон Амдала определяет принципиально важные для параллельных вычислений положения:

1. Ускорение зависит от потенциального параллелизма задачи (величина $1 - a$) и параметров аппаратуры (числа процессоров n).
2. Предельное ускорение определяется свойствами задачи.

Пусть, например, $a = 0{,}2$ (что является реальным значением), тогда ускорение не может превосходить 5 при любом числе процессоров, то есть максимальное ускорение определяется потенциальным параллелизмом задачи. Очевидной является чрезвычайно высокая чувствительность ускорения к изменению величины a.

Сетевой закон Амдала. Основной вариант закона Амдала не отражает потерь времени на межпроцессорный обмен сообщениями. Эти потери могут не только снизить ускорение вычислений, но и замедлить вычисления по сравнению с однопроцессорным вариантом. Поэтому необходима некоторая модернизация выражения (1.3).

Перепишем (1.3) следующим образом:

$$R_c = \frac{W \cdot t}{(W_{ск} + \frac{W_{пр}}{n}) \cdot t + W_c \cdot t_c} = \frac{1}{a + \frac{1-a}{n} + \frac{W_c \cdot t_c}{W \cdot t}} = \frac{1}{a + \frac{1-a}{n} + c} . \qquad (1.4)$$

Здесь W_c – количество передач данных, t_c – время одной передачи данных. Выражение

$$R_C = \frac{1}{a + \frac{1-a}{n} + c} \tag{1.5}$$

и является сетевым законом Амдала. Этот закон определяет следующие две особенности многопроцессорных вычислений:

1. Коэффициент сетевой деградации вычислений *c:*

$$c = \frac{W_c \cdot t_c}{W \cdot t} = c_A \cdot c_T, \tag{1.6}$$

определяет объем вычислений, приходящийся на одну передачу данных (по затратам времени). При этом c_A определяет алгоритмическую составляющую коэффициента деградации, обусловленную свойствами алгоритма, а c_T – техническую составляющую, которая зависит от соотношения технического быстродействия процессора и аппаратуры сети. Таким образом, для повышения скорости вычислений следует воздействовать на обе составляющие коэффициента деградации. Для многих задач и сетей коэффициенты c_A и c_T могут быть вычислены аналитически и заранее, хотя они определяются множеством факторов: алгоритмом задачи [14,15], размером данных, реализацией функций обмена библиотеки MPI, использованием разделяемой памяти и, конечно, техническими характеристиками коммуникационных сред и их протоколов.

2. Даже если задача обладает идеальным параллелизмом, сетевое ускорение определяется величиной

$$R_c = \frac{1}{\frac{1}{n} + c} = \frac{n}{1 + c \cdot n} \xrightarrow[c \to 0]{} n \tag{1.7}$$

и увеличивается при уменьшении с. Следовательно, сетевой закон Амдала должен быть основой оптимальной разработки алгоритма и программирования задач, предназначенных для решения на многопроцессорных ЭВМ.

В некоторых случаях используется еще один параметр для измерения эффективности вычислений – коэффициент утилизации *z:*

$$z = \frac{R_c}{n} = \frac{1}{1 + c \cdot n} \xrightarrow[c \to 0]{} 1 \ . \tag{1.8}$$

Более подробно вопросы эффективности вычислений изложены в главе 12.

1.3. ТЕХНИЧЕСКАЯ РЕАЛИЗАЦИЯ МНОГОПРОЦЕССОРНЫХ СИСТЕМ

Существующие параллельные вычислительные средства класса MIMD образуют три технических подкласса: симметричные мультипроцессоры (SMP), системы с массовым параллелизмом (MPP) и кластеры. В основе этой классификации лежит структурно-функциональный подход.

Симметричные мультипроцессоры используют принцип разделяемой памяти. В этом случае система состоит из нескольких однородных процессоров и массива общей памяти (обычно из нескольких независимых блоков). Все процессоры имеют доступ к любой точке памяти с одинаковой скоростью. Процессоры подключены к памяти с помощью общей шины или коммутатора. Аппаратно поддерживается когерентность кэшей.

Наличие общей памяти сильно упрощает взаимодействие процессоров между собой, однако накладывает сильные ограничения на их число – не более 32 в реальных системах. Вся система работает под управлением единой ОС (обычно UNIX-подобной, но для Intel-платформ поддерживается Windows NT).

Системы с массовым параллелизмом содержат множество процессоров (обычно RISC) с индивидуальной памятью в каждом из них (прямой доступ к памяти других узлов невозможен), коммуникационный процессор или сетевой адаптер, иногда – жесткие диски и/или другие устройства ввода–вывода. Узлы связаны через некоторую коммуникационную среду (высокоскоростная сеть, коммутатор и т.п.). Общее число процессоров в реальных системах достигает нескольких тысяч (ASCI Red, Blue Mountain).

Полноценная операционная система может располагаться только на управляющей машине (на других узлах работает сильно урезанный вариант ОС), либо на каждом узле работает полноценная UNIX-подобная ОС (вариант, близкий к кластерному подходу). Программирование осуществляется в рамках модели передачи сообщений (библиотеки параллельных функций MPI, PVM, BSPlib).

Кластерные системы – более дешевый вариант MPP–систем, поскольку они также используют принцип передачи сообщений, но строятся из компонентов высокой степени готовности [16,2].

Вычислительный кластер – это совокупность компьютеров, объединенных в рамках некоторой сети для решения одной задачи. В качестве вычислительных узлов обычно используются доступные на рынке однопроцессорные компьютеры, двух– или четырехпроцессорные SMP-серверы. Каждый узел работает под управлением своей копии операционной системы, в качестве которой чаще всего используются стандартные операционные системы: Linux, NT, Solaris и т.п. Состав и мощность узлов может меняться даже в рамках одного кластера, давая возможность создавать неоднородные системы.

Для кластерных систем в соответствии с сетевым законом Амдала характеристики коммуникационных сетей имеют принципиальное значение.

Коммуникационные сети [16] имеют две основные характеристики: латентность – время начальной задержки при посылке сообщений и пропускную способность сети, определяющую скорость передачи информации по каналам связи. После вызова пользователем функции посылки сообщения Send() сообщение последовательно проходит через целый набор слоев, определяемых особенностями организации программного обеспечения и аппаратуры, прежде чем покинуть процессор. Наличие латентности определяет и тот факт, что максимальная скорость передачи по сети не может быть достигнута на сообщениях с небольшой длиной.

Чаще всего используется сеть Fast Ethernet, основное достоинство которой – низкая стоимость оборудования. Однако большие накладные расходы на передачу сообщений в рамках Fast Ethernet приводят к серьезным ограничениям на спектр задач, которые можно эффективно решать на таком кластере. Если от кластера требуется большая универсальность, то нужно переходить на более производительные коммуникационные сети, например, SCI, Myrinet, некоторые другие. Характеристики некоторых сетей представлены в приложении 4.

1.4. ПРОГРАММИРОВАНИЕ ДЛЯ СИСТЕМ С РАЗДЕЛЯЕМОЙ ПАМЯТЬЮ

Процессы. В операционной системе UNIX поддерживается возможность параллельного выполнения нескольких пользовательских программ. Каждому такому выполнению соответствует процесс операционной системы [17]. Каждый процесс обладает собственными ресурсами, то есть выполняется в собственной виртуальной памяти, и тем самым процессы защищены один от другого, т.е. один процесс не

в состоянии неконтролируемым образом прочитать что-либо из памяти другого процесса или записать в нее. Однако контролируемые взаимодействия процессов допускаются системой, в том числе за счет возможности разделения одного сегмента памяти между виртуальной памятью нескольких процессов. Каждый процесс может образовать полностью идентичный подчиненный процесс выполнением системного вызова **FORK()** и дожидаться окончания выполнения своих подчиненных процессов с помощью системного вызова **WAIT**.

После создания процесса предок и потомок могут произвольным образом изменять свой контекст. В частности, и предок, и потомок могут выполнить какой-либо из вариантов системного вызова exec, приводящего к полному изменению контекста процесса.

Нити. Понятие нити (thread, light-weight process – легковесный процесс, поток управления) давно известно в области операционных систем. В одной виртуальной памяти может выполняться не один поток управления. Если несколько процессов совместно пользуются некоторыми ресурсами (общим адресным пространством, общими переменными, аппаратными ресурсами и др.), то при доступе к этим ресурсам они должны синхронизовать свои действия. Многолетний опыт программирования с использованием явных примитивов синхронизации показал, что этот стиль "параллельного" программирования порождает серьезные проблемы при написании, отладке и сопровождении программ (наиболее трудно обнаруживаемые ошибки в программах обычно связаны с синхронизацией). Это было главной причиной того, что в традиционных вариантах ОС UNIX понятие процесса жестко связывалось с понятием отдельной и недоступной для других процессов виртуальной памяти.

При появлении систем SMP отношение к процессам претерпело существенные изменения. В таких компьютерах физически присутствуют несколько процессоров, которые имеют одинаковые по скорости возможности доступа к совместно используемой основной памяти. Появление подобных машин на мировом рынке поставило проблему их эффективного использования. При применении традиционного подхода ОС UNIX к организации процессов наличие общей памяти не давало эффекта. К моменту появления SMP выяснилось, что технология программирования еще не может предложить эффективного и безопасного способа реального параллельного программирования. Поэтому пришлось вернуться к явному параллельному программиро-

ванию с использованием параллельных процессов в общей виртуальной (а тем самым и основной) памяти с явной синхронизацией.

Нить – это независимый поток управления с собственным счетчиком команд, выполняемый в контексте некоторого процесса. Фактически, понятие контекста процесса изменяется следующим образом. Все, что не относится к потоку управления (виртуальная память, дескрипторы открытых файлов и т.д.), остается в общем контексте процесса. Вещи, которые характерны для потока управления (регистровый контекст, стеки разного уровня и т.д.), переходят из контекста процесса в контекст нити. При этом контексты всех нитей остаются вложенными в контекст породившего их процесса.

Поскольку нити одного процесса выполняются в общей виртуальной памяти, стек любой нити процесса в принципе не защищен от произвольного (например, по причине ошибки) доступа со стороны других нитей.

Семафоры. Чтобы исключить упомянутую выше ситуацию, необходимо ввести систему синхронизации параллельных процессов.

Выход заключается в разрешении входить в критическую секцию (КС) только одному из нескольких асинхронных процессов. Под критической секцией понимается участок процесса, в котором процесс нуждается в ресурсе. Решение проблемы критической секции предложил Дейкстра [17] в виде семафоров. Семафором называется переменная S, связанная, например, с некоторым ресурсом и принимающая два состояния: 0 (запрещено обращение) и 1 (разрешено обращение). Над S определены две операции: V и P. Операция V изменяет значение S семафора на значение $S + 1$. Действие операции P таково:

1. если $S \neq 0$, то P уменьшает значение на единицу;
2. если $S = 0$, то P не изменяет значения S и не завершается до тех пор, пока некоторый другой процесс не изменит значение S с помощью операции V.

Операции V и P считаются неделимыми, т. е. не могут исполняться одновременно.

Приведем пример синхронизации двух процессов, в котором process 1 и process 2 могут выполняться параллельно.

Процесс может захватить ресурс только тогда, когда S:=1. После захвата процесс закрывает семафор операции $P(S)$ и открывает его вновь после прохождения критической секции $V(S)$. Таким образом, семафор S обеспечивает неделимость процессов Li и, значит, их по-

следовательное выполнение. Это и есть решение задачи взаимного исключения для процессов *Li.*

```
begin
semaphore S;
S:=1;
process 1:
  begin
     L1:P(S);
     Критический участок 1;
     V(S);
     Остаток цикла, go to L1
  end
process 2:
   begin
     L2:P(S);
     Критический участок 2;
     V(S);
     Остаток цикла, go to L2
   end
end
```

OpenMP. Интерфейс OpenMP [18] является стандартом для программирования на масштабируемых SMP-системах с разделяемой памятью. В стандарт OpenMP входят описания набора директив компилятора, переменных среды и процедур. За счет идеи "инкрементального распараллеливания" OpenMP идеально подходит для разработчиков, желающих быстро распараллелить свои вычислительные программы с большими параллельными циклами. Разработчик не создает новую параллельную программу, а просто добавляет в текст последовательной программы OpenMP-директивы.

Предполагается, что OpenMP-программа на однопроцессорной платформе может быть использована в качестве последовательной программы, т.е. нет необходимости поддерживать последовательную и параллельную версии. Директивы OpenMP просто игнорируются последовательным компилятором, а для вызова процедур OpenMP могут быть подставлены заглушки, текст которых приведен в спецификациях. В OpenMP любой процесс состоит из нескольких *нитей управления*, которые имеют общее адресное пространство, но разные потоки команд и раздельные стеки. В простейшем случае, процесс состоит из одной нити.

Обычно для демонстрации параллельных вычислений используют простую программу вычисления числа π. Рассмотрим, как можно на-

писать такую программу в OpenMP. Число π можно определить следующим образом:

$$\int_0^1 \frac{1}{1+x^2} dx = \text{arctg}(1)\text{-arctg}(0)= \pi/4.$$

Вычисление интеграла затем заменяют вычислением суммы :

$$\int_0^1 \frac{4}{1+x^2} dx = \frac{1}{n} \sum_{i=1}^{n} \frac{4}{1+x_i^2},$$

где: $x_i = (i\text{-}0.5)/n$.

В последовательную программу вставлены две строки, и она становится параллельной.

```
program compute_pi
  parameter (n = 1000)
  integer i
  double precision w,x,sum,pi,f,a
  f(a) = 4.d0/(1.d0+a*a)
  w = 1.0d0/n
  sum = 0.0d0;
!$OMP PARALLEL DO PRIVATE(x), SHARED(w)
!$OMP& REDUCTION(+:sum)
  do i=1,n
     x = w*(i-0.5d0)
     sum = sum + f(x)
  enddo
  pi = w*sum
  print *,'pi = ',pi
  stop
  end
```

Программа начинается как единственный процесс на головном процессоре. Он исполняет все операторы вплоть до первой конструкции типа PARALLEL. В рассматриваемом примере это оператор PARALLEL DO, при исполнении которого порождается множество процессов с соответствующим каждому процессу окружением. В рассматриваемом примере окружение состоит из локальной (PRIVATE) переменной x, переменной sum редукции (REDUCTION) и одной разделяемой (SHARED) переменной w. Переменные x и sum локальны в каждом процессе без разделения между несколькими процессами. Переменная w располагается в головном процессе. Оператор редукции

REDUCTION имеет в качестве атрибута операцию, которая применяется к локальным копиям параллельных процессов в конце каждого процесса для вычисления значения переменной в головном процессе. Переменная цикла i является локальной в каждом процессе по своей сути, так как именно с уникальным значением этой переменной порождается каждый процесс. Параллельные процессы завершаются оператором END DO, выступающим как синхронизирующий барьер для порожденных процессов. После завершения всех процессов продолжается только головной процесс.

Директивы. Директивы OpenMP с точки зрения Фортрана являются комментариями и начинаются с комбинации символов "!$OMP". Директивы можно разделить на 3 категории: определение параллельной секции, разделение работы, синхронизация. Каждая директива может иметь несколько дополнительных атрибутов – клауз. Отдельно специфицируются клаузы для назначения классов переменных, которые могут быть атрибутами различных директив.

Директивы порождения нитей предназначены для генерации и распределения работы между ними, в том числе и для явного распределения. **PARALLEL ... END PARALLEL** определяет параллельную область программы. При входе в эту область порождается (*N–1*) новых процессов, образуется "команда" из *N* нитей, а порождающая нить получает номер 0 и становится основной нитью команды (т.н. "master thread"). При выходе из параллельной области основная нить дожидается завершения остальных нитей и продолжает выполнение в одном экземпляре. Предполагается, что в SMP-системе нити будут распределены по различным процессорам (однако это, как правило, находится в ведении операционной системы).

Директивы разделение работ. Работа распределяется директивами **DO**, **SECTIONS** и **SINGLE**. Возможно также явное управление распределением работы с помощью функций, возвращающих номер текущей нити и общее число нитей. По умолчанию код внутри **PARALLEL** исполняется всеми нитями одинаково. Директива **DO ... [ENDDO]** определяет параллельный цикл. Директива **SECTIONS ... END SECTIONS** определяет набор независимых секций кода. Секции отделяются друг от друга директивой **SECTION**. Директива **SINGLE ... END SINGLE** определяет блок кода, который будет исполнен только одной нитью (первой, которая дойдет до этого блока).

Директивы синхронизации. Директива **MASTER ... END MASTER** определяет блок кода, который будет выполнен только master-ом (нулевой нитью). Директива **CRITICAL ... END CRITICAL** определяет критическую секцию, то есть блок кода, который не должен выполняться одновременно двумя или более нитями. Директива **BARRIER** определяет точку барьерной синхронизации, в которой каждая нить дожидается всех остальных. Существуют и другие директивы синхронизации.

Классы переменных. OpenMP–переменные в параллельных областях программы разделяются на два основных класса: **SHARED** (общие – под именем A все нити видят одну переменную) и **PRIVATE** (приватные – под именем A каждая нить видит свою переменную).

1.5. ПРОГРАММИРОВАНИЕ ДЛЯ СИСТЕМ С ПЕРЕДАЧЕЙ СООБЩЕНИЙ

Система программирования **MPI** относится к классу МКМД ЭВМ с индивидуальной памятью, то есть к многопроцессорным системам с обменом сообщениями. MPI имеет следующие особенности:

- MPI – библиотека, а не язык. Она определяет имена, вызовы процедур и результаты их работы. Программы, которые пишутся на FORTRAN, C, и C++ компилируются обычными компиляторами и связаны с MPI–библиотекой.
- MPI – описание, а не реализация. Все поставщики параллельных компьютерных систем предлагают реализации MPI для своих машин как бесплатные, и они могут быть получены из Интернета. Правильная MPI-программа должна выполняться на всех реализациях без изменения.
- MPI соответствует модели многопроцессорной ЭВМ с передачей сообщений.

В модели передачи сообщений процессы, выполняющиеся параллельно, имеют раздельные адресные пространства. Связь происходит, когда часть адресного пространства одного процесса скопирована в адресное пространство другого процесса. Эта операция совместная и возможна только когда первый процесс выполняет операцию передачи сообщения, а второй процесс – операцию его получения.

Процессы в MPI принадлежат группам. Если группа содержит **n** процессов, то процессы нумеруются внутри группы номерами, кото-

рые являются целыми числами от 0 до **n-l**. Имеется начальная группа, которой принадлежат все процессы в реализации MPI.

Понятия контекста и группы объединены в едином объекте, называемом коммуникатором. Таким образом, отправитель или получатель, определенные в операции посылки или получения, всегда обращается к номеру процесса в группе, идентифицированной данным коммуникатором.

В MPI базисной операцией посылки является операция:

MPI_Send (**address, count, datatype, destination, tag, comm**),

где (**address, count, datatype**) – количество (**count)** объектов типа **datatype**, начинающихся с адреса **address** в буфере посылки; **destination** – номер получателя в группе, определяемой коммуникатором **comm**; **tag** – целое число, используемое для описания сообщения; **comm** – идентификатор группы процессов и коммуникационный контекст.

Базисной операцией приема является операция:

MPI_Recv (**address, maxcount, datatype, source, tag, comm, status**),

где (**address, count, datatype**) описывают буфер приемника, как в случае **MPI_Send**; **sourse** – номер процесса-отправителя сообщения в группе, определяемой коммуникатором **comm**; **status** – содержит информацию относительно фактического размера сообщения, источника и тэга. **Sourse, tag, count** фактически полученного сообщения восстанавливаются на основе **status**.

В MPI используются коллективные операции, которые можно разделить на два вида:

- операции перемещения данных между процессами. Самый простой из них – широковещание (broadcasting), MPI имеет много и более сложных коллективных операций передачи и сбора сообщений;
- операции коллективного вычисления (минимум, максимум, сумма и другие, в том числе и определяемые пользователем операции).

В обоих случаях библиотеки функций коллективных операций строятся с использованием знания о преимуществах структуры машины, чтобы увеличить параллелизм выполнения этих операций.

Часто предпочтительно описывать процессы в проблемно-ориентированной топологии. В MPI используется описание процессов в топологии графовых структур и решеток.

В MPI введены средства, позволяющие определять состояние процесса вычислений, которые позволяют отлаживать программы и улучшать их характеристики.

В MPI имеются как блокирующие операции **send** и **receive**, так и неблокирующий их вариант, благодаря чему окончание этих операций может быть определено явно. MPI также имеет несколько коммуникационных режимов. Стандартный режим соответствует общей практике в системах передачи сообщений. Синхронный режим требует блокировать **send** на время приема сообщения в противоположность стандартному режиму, при котором **send** блокируется до момента захвата буфера. Режим по готовности (для **send**) – способ, предоставленный программисту, чтобы сообщить системе, что этот прием был зафиксирован, следовательно, низлежащая система может использовать более быстрый протокол, если он доступен. Буферизованный режим позволяет пользователю управлять буферизацией.

Программы MPI могут выполняться на сетях машин, которые имеют различные длины и форматы для одного и того же типа **datatype**, так что каждая коммуникационная операция определяет структуру и все компоненты **datatype**. Следовательно, реализация всегда имеет достаточную информацию, чтобы делать преобразования формата данных, если они необходимы. MPI не определяет, как эти преобразования должны выполняться, разрешая реализации производить оптимизацию для конкретных условий.

Процесс есть программная единица, у которой имеется собственное адресное пространство и одна или несколько нитей. **Процессор** – фрагмент аппаратных средств, способный к выполнению программы. Некоторые реализации MPI устанавливают, что в программе MPI всегда одному процессу соответствует один процессор; другие – позволяют размещать много процессов на каждом процессоре.

Если в кластере используются SMP–узлы (симметричная многопроцессорная система с множественными процессорами), то для организации вычислений возможны два варианта.

1. Для каждого процессора в SMP-узле порождается отдельный MPI-процесс. MPI-процессы внутри этого узла обмениваются сообщениями через разделяемую память (необходимо настроить MPICH соответствующим образом).
2. На каждой узле запускается только один MPI-процесс. Внутри каждого MPI-процесса производится распараллеливание в модели "общей памяти", например с помощью директив OpenMP.

Чем больше функций содержит библиотека MPI, тем больше возможностей представляется пользователю для написания эффективных программ. Однако для написания подавляющего числа программ принципиально достаточно следующих шести функций:

MPI_Init	Инициализация MPI
MPI_Comm_size	Определение числа процессов
MPI_Comm_rank	Определение процессом собственного номера
MPI_Send	Посылка сообщения
MPI_Recv	Получение сообщения
MPI_Finalize	Завершение программы MPI

В качестве примера параллельной программы, написанной в стандарте MPI для языка C, рассмотрим программу вычисления числа π. Алгоритм вычисления π уже описывался в параграфе 1.4.

```
#include "mpi.h"
#include <math.h>
int main ( int argc, char *argv[ ] )
{   int n, myid, numprocs, i;
    double mypi, pi, h, sum, x, t1, t2, PI25DT = 3.14159265358979323846264 3;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (1)
    {
      if (myid == 0)
      {       printf ("Enter the number of intervals: (0 quits) ");
              scanf ("%d", &n);
              t1 = MPI_Wtime();
      }
      MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      if (n == 0)  break;
      else
      {  h  = 1.0/ (double) n;
         sum = 0.0;
         for (i = myid +1; i <= n; i+= numprocs)
         {    x = h * ( (double)i - 0.5);
              sum += (4.0 / (1.0 + x*x));
          }
          mypi = h * sum;
          MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                           MPI_COMM_WORLD);
        if (myid == 0)
        {    t2 = MPI_Wtime();
```

```
            printf ("pi is approximately %.16f. Error is %.16f\n",pi, fabs(pi - PI25DT));
            printf ("'time is  %f  seconds \n", t2-t1);
        }
       }
    }
    MPI_Finalize();
    return 0;
}
```

В программе после нескольких строк определения переменных следуют три строки, которые есть в каждой MPI–программе:

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

Обращение к **MPI_Init** должно быть первым обращением в MPI–программе, оно устанавливает "среду" MPI. В каждом выполнении программы может выполняться только один вызов **MPI_Init**.

Коммуникатор **MPI_COMM_WORLD** описывает состав процессов и связи между ними. Вызов **MPI_Comm_size** возвращает в **numprocs** число процессов, которые пользователь запустил в этой программе. Значение **numprocs** – размер группы процессов, связанной с коммуникатором **MPI_COMM_WORLD**. Процессы в любой группе нумеруются последовательными целыми числами, начиная с 0. Вызывая **MPI_ Comm_rank,** каждый процесс выясняет свой номер (**rank)** в группе, связанной с коммуникатором. Затем главный процесс (который имеет **myid**=0) получает от пользователя значение числа прямоугольников **n**:

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Первые три параметра соответственно обозначают адрес, количество и тип данных. Четвертый параметр указывает номер источника данных (головной процесс), пятый параметр – название коммуникатора группы. Таким образом, после обращения к **MPI_Bcast** все процессы имеют значение **n** и собственные идентификаторы, что является достаточным для каждого процесса, чтобы вычислить **mypi** – свой вклад в вычисление π. Для этого каждый процесс вычисляет область каждого прямоугольника, начинающегося с **myid + l**.

Затем все значения **mypi**, вычисленные индивидуальными процессами, суммируются с помощью вызова **Reduce**:

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);

Первые два параметра описывают источник и адрес результата. Третий и четвертый параметр описывают число данных (1) и их тип, пятый параметр – тип арифметико-логической операции, шестой – номер процесса для размещения результата.

Затем по метке 10 управление передается на начало цикла. Этим пользователю предоставляется возможность задать новое **n** и повысить точность вычислений. Когда пользователь печатает нуль в ответ на запрос о новом **n**, цикл завершается, и все процессы выполняют:

MPI_Finalize();

после которого любые операции MPI выполняться не будут.

Функция MPI_Wtime() используется для измерения времени исполнения участка программы, расположенного между двумя включениями в программу этой функции.

Ранее говорилось, что для написания большинства программ достаточно 6 функции, среди которых основными являются функции обмена сообщениями типа “точка-точка” (в дальнейшем – функции парного обмена). Программу вычисления можно написать с помощью функций парного обмена, но функции, относящиеся к классу коллективных обменов, как правило, будут эффективнее. Коллективные функции **Bcast** и **Reduce** можно выразить через парные операции **Send** и **Recv**. Например, для той же программы вычисления числа π операция **Bcast** для рассылки числа интервалов выражается через цикл следующим образом:

```
for (i=0; i<numprocs; i++)
    MPI_Send(&n, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
```

Параллельная MPI программа может содержать различные исполняемые файлы. Этот стиль параллельного программирования часто называется MPMD (множество программ при множестве данных) в отличие от программ SPMD (одна программа при множестве данных). SPMD не следует путать с SIMD (один поток команд при множестве данных). Вышеприведенная программа вычисления числа π относится к классу программ SPMD. Такие программы значительно легче писать и отлаживать, чем программы MPMD. В общем случае системы SPM и MPI могут имитировать друг друга:

1. Посредством организации единого адресного пространства для физически разделенной по разным процессорам памяти.
2. На SMP–машинах вырожденным каналом связи для передачи сообщений служит разделяемая память.
3. Путем использования компьютеров с разделяемой виртуальной памятью. Общая память как таковая отсутствует. Каждый процессор имеет собственную локальную память и может обращаться к локальной памяти других процессоров, используя "глобальный адрес". Если "глобальный адрес" указывает не на локальную память, то доступ к памяти реализуется с помощью сообщений, пересылаемых по коммуникационной сети.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ К ГЛАВЕ 1

Контрольные вопросы к 1.1

1. Какие понятия положены в основу классификации Флинна?
2. Назовите и опишите классы параллельных ЭВМ по Флинну.
3. Что такое многопроцессорные ЭВМ с разделяемой памятью?
4. Что вызывает некорректность вычислений в ЭВМ с разделяемой памятью?
5. Каковы достоинства и недостатки ЭВМ с передачей сообщений?

Контрольные вопросы к 1.2

1. Что такое ускорение вычислений?
2. Что определяет закон Амдала?
3. Какую характеристику определяет сетевой закон Амдала?
4. Какие факторы влияют на эффективность сетевых вычислений?

Контрольные вопросы к 1.3

1. Определите три класса технической реализации многопроцессорных ЭВМ.
2. Что такое симметричные мультипроцессоры (SMP)?
3. Каковы особенности систем с массовым параллелизмом (MPP)?
4. Дайте определение вычислительного кластера.
5. Опишите виды кластеров, их особенности, дайте примеры кластеров.
6. Что такое коммуникационная сеть, каковы ее основные параметры?

Контрольные вопросы к 1.4

1. Определите понятие процесса и нити, в чем их различие?
2. Как в Unix создаются процессы, нити?
3. Что такое семафоры, для чего они необходимы?
4. Что такое стандарт OpenMP?
5. Опишите как выполняется в языке OpenMP программа вычисления числа π.
6. Назовите типы директив стандарта OpenMP.
7. Какие классы переменных используются в OpenMP?

Контрольные вопросы к 1.5

1. Что такое стандарт MPI?
2. Назовите основные операции передачи и приема в MPI.
3. Назовите и опишите состав и назначение параметров обменных функций MPI.
4. Что такое процесс и процессор в MPI?
5. Перечислите минимально возможный состав MPI функций.
6. Расскажите, как выполняется программа MPI для вычисления числа π.
7. Какой коммуникатор определен после выполнения функции MPI_Init?
8. Можно ли использовать функции MPI до вызова MPI_Init?
9. Как в MPI определить номер процесса?
10. Как узнать число запущенных процессов приложения?
11. Возможна ли замена в MPI коллективных операций на парные обмены?

Глава 2. РЕАЛИЗАЦИИ ИНТЕРФЕЙСА ПРОГРАММИРОВАНИЯ MPI

2.1. MPICH – ОСНОВНАЯ РЕАЛИЗАЦИЯ MPI

MPI – это описание библиотеки функций, которые обеспечивают в первую очередь обмен данными между процессами. Следовательно, чтобы такая библиотека работала в некоторой исполнительной среде, необходимо между описанием библиотеки и исполнительной средой иметь промежуточный слой, который называется реализацией MPI для данной исполнительной среды.

Под реализацией обычно понимают программные средства, обеспечивающие выполнение всех функций MPI в исполнительной среде. Эти процедуры обеспечивают парные и коллективные обмены, реализуют коммуникаторы, топологию, средства общения с исполнительной средой, средства профилирования и множество вспомогательных операций, связанных, например, с запуском вычислений, оценкой эффективности параллельных вычислений и многое другое.

Возможны два способа построения реализаций: прямая реализация для конкретной ЭВМ и реализация через ADI (Abstract Device Interface – интерфейс для абстрактного прибора). Поскольку имеется большое количество типов реальных параллельных систем, то количество реализаций в первом случае будет слишком велико. Во втором случае строится реализация только для одного ADI, а затем архитектура ADI поставщиками параллельного оборудования реализуется в конкретной системе (как правило, программно). Такая двухступенчатая реализация MPI уменьшает число вариантов реализаций и обеспечивает переносимость реализации. Поскольку аппаратно зависимая

часть этого описания невелика, то использование метода ADI позволяет создать пакет программ, который разработчики реальных систем могут использовать практически без переделок.

Основной объем работ по разработке стандарта MPI и построению его реализаций выполняется в Аргоннской национальной лаборатории США [1]. Здесь подготовлены и получили широкое распространение реализации MPI, получившие название MPICH (добавка CH взята из названия пакета Chameleon, который ранее использовался для систем с передачей сообщений, многое из этого пакета вошло в MPICH).

Имеется три поколения MPICH, связанных с развитием ADI. Первое поколение ADI-1 было спроектировано для компьютеров с массовым параллелизмом, где механизм обмена между процессами принадлежал системе. Этот ADI обеспечивал хорошие характеристики с малыми накладными расходами, такая версия MPICH была установлена на параллельных компьютерах: Intel iPSC/860, Delta, Paragon, nCUBE.

Второе поколение – ADI-2 – было введено, чтобы получить большую гибкость в реализациях и эффективно поддерживать коммуникационные механизмы с большим объемом функций.

Третье поколение – ADI-3 – было спроектировано, чтобы обеспечить большее соответствие появляющимся сетям с удаленным доступом, многопоточной исполнительной среде и поддержке операций MPI-2, таких как удаленный доступ к памяти и динамическое управление процессами. ADI-3 есть первая версия MPICH, в которой при проектировании не ставилась задача близкого соответствия другим библиотекам с обменом сообщениями (в частности, PVM [16]), поскольку MPI вытеснил большинство систем с обменом сообщениями в научных вычислениях. ADI-3 подобно предыдущим ADI спроектирован так, чтобы содействовать переносу MPICH на новые платформы и коммуникационные методы.

ADI для обмена сообщениями должен обеспечивать четыре набора функций:

- для описания передаваемых и получаемых сообщений;
- для перемещения данных между ADI и передающей аппаратурой;
- для управления списком зависших сообщений (как посланных, так и принимаемых);
- для получения основной информации об исполнительной среде и ее состоянии (например, как много задач выполняется).

MPICH ADI выполняет все эти функции; однако многие аппаратные средства для передачи сообщений не могут обеспечить, например,

списковое управление или возможности сложной передачи данных. Эти функции эмулируются путем использования вспомогательных процедур.

Следовательно, ADI – это совокупность определений функций (которые могут быть реализованы как функции С или макроопределения) из пользовательского набора MPI. Если так, то это создает протоколы, которые отличают MPICH от других реализаций MPI. В частности, уровень ADI содержит процедуры для упаковки сообщений и подключения заголовочной информации, управления политикой буферизации, для установления соответствия запущенных приемов приходящих сообщений и др.

Для того чтобы понять, как работает MPICH, рассмотрим в качестве примера реализацию простых функций посылки и приема **MPI_Send** и **MPI_Recv**. Для этой цели могут использоваться два протокола: Eager и Rendezvous.

Eager. При посылке данных MPI вместе с адресом буфера должен включить информацию пользователя о тэге, коммуникаторе, длине, источнике и получателе сообщения. Эту дополнительную информацию называют оболочкой (envelope). Посылаемое сообщение состоит из оболочки, которая следует за данными. Метод посылки данных вместе с оболочкой называется eager («жадным») протоколом.

Когда сообщение прибывает, возможны два случая: либо соответствующая приемная процедура запущена, либо нет. В первом случае предоставляется место для приходящих данных. Во втором случае ситуация много сложнее. Принимающий процесс должен помнить, что сообщение прибыло, и где-то его сохранить. Первое требование выполнить относительно легко, отслеживая очередь поступивших сообщений. Когда программа выполняет **MPI_Recv**, она прежде всего проверяет эту очередь. Если сообщение прибыло, операция выполняется и завершается. Но с данными может быть проблема. Что, например, будет, если множество подчиненных процессов почти одновременно пошлют главному процессу свои длинные сообщения (например, по 100 MB каждое) и места в памяти главного процесса для их размещения не хватит? Похожая ситуация возникает, например, при умножении матриц. Стандарт MPI требует, чтобы в этой ситуации прием данных выполнялся, а не выдавался отказ. Это и приводит к буферизации.

Rendezvous. Чтобы решить проблему доставки большого объема данных по назначению, нужно контролировать, как много и когда эти

данные прибывают на процесс-получатель. Одно простое решение состоит в том, чтобы послать процессу-получателю только оболочку. Когда процесс-получатель потребует данные (и имеет место для их размещения), он посылает отправителю сообщение, в котором говорится: “теперь посылай данные”. Отправитель затем пошлет данные, будучи уверен, что они будут приняты. Этот метод называется протоколом “рандеву”. В этом случае получатель должен отводить память только для хранения оболочек, имеющих очень небольшой размер, а не для хранения самих данных.

Причина для разработки многих вариаций режимов передачи теперь довольно ясны. Каждый режим может быть реализован с помощью комбинации “жадного” и “рандеву” протоколов. Некоторые варианты представлены в таб. 2.1.

Таблица 2.1

Режимы send с протоколами Eager и Rendezvous

Вызов MPI	Размер сообщения	Протокол
MPI_Ssend	any	Rendezvous всегда
MPI_Rsend	any	Eager всегда
MPI_Send	< =16 KB	Eager
MPI_Send	>16 KB	Rendezvous

Главное преимущество протокола «рандеву» состоит в том, что он позволяет принимать произвольно большие сообщения в любом количестве. Но этот метод невыгодно использовать для всех сообщений, поскольку, например, «жадный» протокол быстрее, в частности, для коротких сообщений. Возможно большое количество других протоколов и их модификаций.

Канальный интерфейс является одним из наиболее важных уровней иерархии ADI и может иметь множественные реализации.

На рис. 2.1. имеется два ADI: p4 – для систем с передачей сообщений, и p2 – для систем с разделяемой памятью. Реализация Chameleon создана давно, построена на базе интерфейса p4, многие ее элементы частично использовались на начальной стадии реализации MPICH. Поэтому в MPICH также используется интерфейс p4, который перенесен поставщиками аппаратуры на ряд машин, что и показано на рисунке. Интерфейс p2 также адаптирован для ряда систем.

Однако на рисунке представлены и примеры прямой реализации MPI без промежуточного ADI. Так, ряд машин напрямую использует макросы, из которых состоит сама реализация Chameleon.

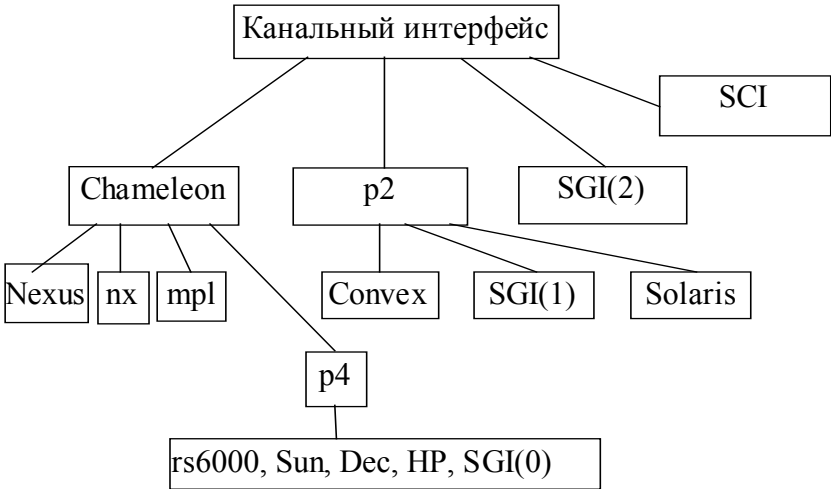


Рис. 2.1. Нижние слои MPICH

Другим примером непосредственной реализации канального интерфейса являются машины, использующие коммуникационные сети SCI. Рис. 2.1. характеризует гибкость в построении канального интерфейса. Это, в частности, относится к машинам SGI. MPICH стал использоваться на машинах SGI с самого начала. Это отмечено блоком SGI(0). Затем стала использоваться усовершенствованная версия SGI(1), использующая интерфейс разделяемой памяти. SGI (2) является прямой реализацией канального интерфейса, использующей ряд новых механизмов. Затем появились еще более совершенные варианты SGI(3) и SGI(4), которые обходят канальный интерфейс и ADI.

MPI в зависимости от версии содержит 150 – 200 функций, и все эти функции должны быть реализованы с помощью MPICH. Часто реализация полного набора функций растянута во времени и занимает достаточно длительный период.

MPICH является переносимой реализацией полного описания MPI для широкого круга параллельных вычислительных сред, включая кластеры рабочих станций и массово-параллельные системы. MPICH содержит кроме самой библиотеки функций MPI программную среду для работы с программами MPI. Программная среда включает мобильный механизм запуска, несколько профилирующих библиотек для изучения характеристик MPI–программ и интерфейс для всех других средств.

2.2. СПОСОБЫ ЗАПУСКА ПРИЛОЖЕНИЙ В MPICH

Запуск приложений выполняется с помощью MPICH. В этом разделе описаны некоторые наиболее распространенных способы запуска приложений. Подробная информация о способах запуска находится в папке www/nt/ любой версии MPICH [1].

2.2.1. Запуск с помощью MPIRun. exe

Это наиболее распространенный способ запуска приложений. Команда MPIRun.exe находится в [MPICH Launcher Home]\bin directory. Использование команды:

1. MPIRun configfile [-logon] [args ...]
2. MPIRun -np #processes [-logon][-env "var1=val1|var2=val2..."] executable [args ...]
3. MPIRun -localonly #processes [-env "var1=val1|var2=val2..."]executable [args ...]

Аргументы в скобках являются опциями.

Формат файла конфигурации config следующий:

```
exe c:\somepath\myapp.exe    или   \\host\share\somepath\myapp.exe
[args arg1 arg2 arg3 ...]
[env VAR1=VAL1|VAR2=VAL2|...|VARn=VALn]
hosts
hostA #procs [path\myapp.exe]
hostB #procs [\\host\share\somepath\myapp2.exe]
hostC #procs
```

Можно описать путь к исполняемому коду отдельной строкой для каждого хоста, тем самым вводя MPMD–программирование. Если путь не описывается, тогда используется по умолчанию путь из строки exe. Приведем пример простого файла конфигурации:

```
exe c:\temp\slave.exe
env MINX=0|MAXX=2|MINY=0|MAXY=2
args -i c:\temp\cool.points
hosts
fry 1 c:\temp\master.exe
fry 1
#light 1
jazz 2
```

Во втором случае запускается количество процессов, равное #processes, начиная с текущей машины и затем по одному процессу на

каждую следующую машину, описанную на этапе инсталляции, пока все процессы не исчерпаны. Если процессов больше, чем машин, то распределение повторяется по циклу.

В третьем случае команда запускает выполнение нескольких процессов на локальной машине, использующей устройство разделяемой памяти.

-env "var1=val1|var2=val2|var3=val3|...varn=valn"

Эта опция устанавливает переменные среды, описанные в строке, перед запуском каждого процесса. Следует помнить, что надо взять в кавычки строку, чтобы приглашение команды не интерпретировало вертикальные линии как конвейерную команду.

-logon

Эта опция mpirun приглашает установить имя пользователя (account) и пароль (password). Если использовать эту опцию, можно описать исполняемую программу, которая размещена в разделяемой памяти. Если не применять –logon, то исполняемая программа должна находиться на каждом хосте локально. Необходимо использовать mpiregister.exe, чтобы закодировать имя и пароль в регистре и избежать приглашения.

2.2.2. Процедура MPIConfig. exe

Чтобы выполнять приложение на различных хостах без описания их в конфигурационном файле, процедура запуска должна знать, какие хосты уже инсталлированы. MPIConfig.exe – это простая программа, которая находит хосты, где процедура запуска уже установлена, и записывает этот список хостов в регистр, который представлен в специальном окне “MPI Configuration Tool”. По этой информации MPIRun может выбрать из списка в окне хосты, где следует запустить процессы. Операции с окном следующие:

Refresh опрашивает сеть для обновления списка имен хостов.

Find подключается к регистру на каждом хосте и определяет по списку, успешно ли уже установлена процедура запуска. Когда он заканчивается, то избранные хосты такие, где эта процедура установлена.

Verify приводит к тому, что mpiconfig подключается к каждому из избранных хостов и убеждается, что DCOM–сервер достижим. Эта особенность еще не реализована.

Set приводит к тому, что вызывается окно – “MPICH Registry settings” – и выполняется следующий диалог:

- Если выбрать "set HOSTS", mpiconfig создаст группу из всех избранных хостов и запишет этот список имен в регистр на каждом хосте. Когда MPIRun выполняется из любого хоста группы с опцией –np, хосты будут выбираться из этого списка.
- Если выбрать "set TEMP", mpiconfig запишет этот директорий в регистр каждого хоста. remote shell server должен создать временный файл для связи с первым запущенным процессом, и этот файл должен располагаться в ячейке, которая пригодна для read/write как для remote shell service, так и для процедуры запуска mpich приложения. remote shell server использует этот вход, чтобы определить, где записать этот файл. По умолчанию устанавливается C:\
- Позиция “launch timeout” указывает, как долго MPIRun будет ждать, пока не убедится, что процесс запустить нельзя. Время задается в миллисекундах.

2.2.3. Процедура MPIRegister. exe

Процедура MPIRegister.exe используется для того, чтобы закодировать имя и пароль в регистр для текущего пользователя. Он находится в [MPICH Launcher Home]\bin directory. Эта информация используется командой MPIRun.exe для запуска приложений в контексте этого пользователя. Если mpiregister не используется, то mpirun будет постоянно приглашать ввести имя и пароль.

Использование:

- MPIRegister
- MPIRegister -remove

Сначала команда MPIRegister попросит ввести имя пользователя. Введите имя в форме [Domain\]Account, где domain name есть опция (например, mcs\ashton or ashton). Затем дважды выполнится приглашение для ввода пароля. Затем последует вопрос, желаете ли Вы хранить эти параметры постоянно. Если Вы говорите “да”, то эти данные будут сохранены на жестком диске. Если “нет “, данные останутся только в памяти. Это означает, что Вы можете запускать mpirun много раз и при этом не потребуется вводить имя и пароль. Однако при перезагрузке машины и использовании mpirun опять возникнет приглашение для ввода имени и пароля. remove приводит к удалению информации из регистра.

2.3. БИБЛИОТЕКА MPE И ЛОГФАЙЛЫ

Библиотека MPE (Multi-Processing Environment) содержит процедуры, которые находятся "под рукой" и облегчают написание, отладку и оценку эффективности MPI–программ. MPE–процедуры делятся на несколько категорий [19].

Параллельная графика (Parallel X graphics**).** Эти процедуры обеспечивают доступ всем процессам к разделяемому X–дисплею. Они создают удобный вывод для параллельных программ, позволяют чертить текст, прямоугольники, круги, линии и так далее.

Регистрация (Logging**).** Одним из наиболее распространенных средств для анализа характеристик параллельных программ является файл трассы отмеченных во времени событий – **логфайл** (logfile). Библиотека MPE создает возможность легко получить такой файл в каждом процессе и собрать их вместе по окончанию работы. Она также автоматически обрабатывает рассогласование и дрейф часов на множественных процессорах, если система не обеспечивает синхронизацию часов. Это библиотека для пользователя, который желает создать свои собственные события и программные состояния.

Последовательные секции (Sequential Sections**).** Иногда секция кода, которая выполняется на ряде процессов, должна быть выполнена только по одному процессу за раз в порядке номеров этих процессов. MPE имеет функции для такой работы.

Обработка ошибок (Error Handling). MPI имеет механизм, который позволяет пользователю управлять реакцией реализации на ошибки времени исполнения, включая возможность создать свой собственный обработчик ошибок.

Далее будут преимущественно рассмотрены средства регистрации (logging) для анализа эффективности параллельных программ. Анализ результатов регистрации производится после выполнения вычислений. Средства регистрации и анализа включают ряд профилирующих библиотек, утилитных программ и ряд графических средств.

Первая группа средств – профилирование. Библиотечные ключи обеспечивают собрание процедур, которые создают логфайлы. Эти логфайлы могут быть созданы вручную путем размещения в MPI–программе обращений к MPE, или автоматически при установлении связи с соответствующими MPE–библиотеками, или комбинацией этих двух методов. В настоящее время MPE предлагает следующие три профилирующие библиотеки:

1. **Tracing Library** (библиотека трассирования) – трассирует все MPI–вызовы. Каждый вызов предваряется строкой, которая содержит номер вызывающего процесса в **MPI_COMM_WORLD** и сопровождается другой строкой, указывающей, что вызов завершился. Большинство процедур **send** и **receive** также указывают значение **count**, **tag** и имена процессов, которые посылают или принимают данные.
2. **Animation Libraries (**анимационная библиотека**)** – простая форма программной анимации в реальном времени, которая требует процедур X–окна.
3. **Logging Libraries** (библиотека регистрации) – самые полезные и широко используемые профилирующие библиотеки в MPE. Они формируют базис для генерации логфайлов из пользовательских программ. Сейчас имеется три различных формата логфайлов, допустимых в MPE. По умолчанию используется формат CLOG. Он содержит совокупность событий с единым отметчиком времени. Формат ALOG больше не развивается и поддерживается для обеспечения совместимости с ранними программами. И наиболее мощным является формат – SLOG (для Scalable Logfile), который может быть конвертирован из уже имеющегося CLOG–файла или получен прямо из выполняемой программы (для этого необходимо установить пременную среды MPE_LOG_FORMAT в SLOG).

Набор утилитных программ в MPE включает конверторы логформатов (например, clog2slog), печать логфайлов (slog_print), оболочки средств визуализации логфайлов, которые выбирают корректные графические средства для представления логфайлов в соответствии с их расширениями.

Далее будут рассмотрены только библиотеки регистрации **Logging Libraries**. Результаты расчета времени дают некоторое представление об эффективности программы. Но в большинстве случаев нужно подробно узнать, какова была последовательность событий, сколько времени было потрачено на каждую стадию вычисления и сколько времени занимает каждая отдельная операция передачи. Чтобы облегчить их восприятие, нужно представить их в графической форме. Но для этого сначала нужно создать файлы событий со связанными временными отметками, затем исследовать их после окончания программы и только затем интерпретировать их графически на рабочей станции. Такие файлы ранее уже названы логфайлами. Способность автомати-

чески генерировать логфайлы является важной компонентой всех средств для анализа эффективности параллельных программ.

Далее в этой главе будут описаны некоторые простые инструментальные средства для создания логфайлов и их просмотра. Библиотека для создания логфайлов отделена от библиотеки обмена сообщениями MPI. Просмотр логфайлов независим от их создания, и поэтому могут использоваться различные инструментальные средства. Библиотека для создания логфайлов MPE разработана таким образом, чтобы сосуществовать с любой MPI–реализацией и распространяется наряду с модельной версией MPI.

Чтобы создать файл регистрации, необходимо вызвать процедуру **MPE_Log_event**. Кроме того, каждый процесс должен вызвать процедуру **MPE_Init_log**, чтобы приготовиться к регистрации, и **MPE_Finish_log**, чтобы объединить файлы, сохраняемые локально при каждом процессе в единый логфайл. **MPE_Stop_log** используется, чтобы приостановить регистрацию, хотя таймер продолжает работать. **MPE_Start_log** возобновляет регистрацию.

Программист выбирает любые неотрицательные целые числа, желательные для типов событий; система не придает никаких частных значений типам событий. События рассматриваются как не имеющие продолжительность. Чтобы измерить продолжительность состояния программы, необходимо, чтобы пара событий отметила начало и окончание состояния. Состояние определяется процедурой **MPE_Describe_state**, которая описывает начало и окончание типов событий. Процедура **MPE_Describe_state** также добавляет название состояния и его цвет на графическом представлении. Соответствующая процедура **MPE_Describe_event** обеспечивает описание события каждого типа. Используя эти процедуры, приведем пример вычисления числа π. Для этого оснастим программу вычисления числа π соответствующими операторами. Важно, чтобы регистрация события не создавала больших накладных расходов. **MPE_Log_event** хранит небольшое количество информации в быстрой памяти. Во время выполнения **MPE_Log_event** эти буфера объединяются параллельно и конечный буфер, отсортированный по временным меткам, записывается процессом 0.

```
#include "mpi.h"
#include "mpe.h"
#include <math.h>
#include <stdio.h>
```

```
int main(int argc, char *argv[ ])
{  int  n, myid, numprocs;
   double PI25DT = 3.14159265358979323846264643;
   double mypi, pi, h, sum, x, startwtime = 0.0, endwtime;
   int  event1a, event1b, event2a, event2b,event3a, event3b, event4a, event4b;
   char processor_name[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   MPE_Init_log();
/*  Пользователь не дает имена событиям, он получает их из MPE */
/*   определяем  8 событий  для 4 состояний Bcast",”Compute",”Reduce" ,”Sync" */
   event1a = MPE_Log_get_event_number();
   event1b = MPE_Log_get_event_number();
   event2a = MPE_Log_get_event_number();
   event2b = MPE_Log_get_event_number();
   event3a = MPE_Log_get_event_number();
   event3b = MPE_Log_get_event_number();
   event4a = MPE_Log_get_event_number();
   event4b = MPE_Log_get_event_number();
   if (myid == 0) {
/* задаем  состояние "Bcast" как время между событиями event1a и event1b. */
        MPE_Describe_state(event1a, event1b, "Broadcast", "red");
/* задаем  состояние "Compute" как время между событиями event2a и vent2b. */
        MPE_Describe_state(event2a, event2b, "Compute",   "blue");
/* задаем состояние "Reduce" как время между событиями event3a и event3b. */
        MPE_Describe_state(event3a, event3b, "Reduce",    "green");
/* задаем состояние "Sync" как время между событиями event4a и event4b. */
        MPE_Describe_state(event4a, event4b, "Sync",   "orange");
   }
   if (myid == 0)
   {     n = 1000000;
         startwtime = MPI_Wtime();
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPE_Start_log();
                    /* регистрируем событие event1a */
    MPE_Log_event(event1a, 0, "start broadcast");
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
                   /* регистрируем событие event1b */
    MPE_Log_event(event1b, 0, "end broadcast");
                    /* регистрируем событие event4a */
    MPE_Log_event(event4a,0,"Start Sync");
    MPI_Barrier(MPI_COMM_WORLD);
                  /* регистрируем событие event4b */
```

```
    MPE_Log_event(event4b,0,"End Sync");
                    /* регистрируем событие event2a */
    MPE_Log_event(event2a, 0, "start compute");
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs)
    {  x = h * ((double)i - 0.5);
       sum += (4.0 / (1.0 + x*x));
     }
     mypi = h * sum;
                     /* регистрируем событие event2b */
     MPE_Log_event(event2b, 0, "end compute");
                      /* регистрируем событие event3a */
     MPE_Log_event(event3a, 0, "start reduce");
     MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
                      /* регистрируем событие event3b */
     MPE_Log_event(event3b, 0, "end reduce");
     MPE_Finish_log("cpilog");
     if (myid == 0)
     {    endwtime = MPI_Wtime();
          printf("pi is approximately %.16f, Error is %.16f\n",
                pi, fabs(pi - PI25DT));
          printf("wall clock time = %f\n", endwtime-startwtime);
     }
    MPI_Finalize();
    return(0);
}
```

Важный вопрос – доверие к локальным часам. На некоторых параллельных компьютерах часы синхронизированы, но на других – только приблизительно. В локальных сетях рабочих станций ситуация намного хуже, и генераторы тактовых импульсов в разных компьютерах иногда «плывут» один относительно другого.

Анализ логфайлов. После выполнения программы **MPI**, которая содержит процедуры MPE для регистрации событий, директорий, где она выполнялась, содержит файл событий, отсортированных по времени, причем время скорректировано с учетом «плавания» частоты генераторов. Можно написать много программ для анализа этого файла и представления информации.

Например, одна из реализаций MPE содержит короткую программу, называемую **states**. Если мы выполняем ее с логфайлом, который описали выше, мы получим:

Состояние:	**Время:**
Broadcast	0.000184
Compute	4.980584
Reduce	0.000248
Sync	0.000095
Сумма:	**4.981111**

Такая итоговая информация является довольно распространенной, но грубой формой профилирования; она сообщает только, где программа тратит время. Значительно информативнее графическое представление, обеспечиваемое специализированными программами, например, **upshot** и **jampshot**.

Входные процедуры MPE используются, чтобы создать логфайлы (отчеты) о событиях, которые имели место при выполнении параллельной программы. Форматы этих процедур представлены ниже.

Форматы для C

int **MPE_Init_log** (void)
int **MPE_Start_log** (void)
int **MPE_Stop_log** (void)
int **MPE_Finish_log** (char *logfilename)
int **MPE_Describe_state** (int start, int end, char *name, char *color)
int **MPE_Describe_event** (int event, char *name)
int **MPE_Log_event** (int event, int intdata char *chardata)

Форматы для Fortran

MPE_INIT_LOG( )
MPE_FINISH_LOG (LOGFILENAME)
CHARACTER*(*) LOGFILENAME
MPE_START_LOG ( )
MPE_STOP_LOG ( )
MPE-DESCRIBE_STATE(START**,** END, NAME, COLOR)
INTEGER START, END
CHARACTER*(*) NAME, COLOR
MPE_DESCRIBE_EVENT(EVENT, NAME)
INTEGER EVENT
CHARACTER*(*) NAME
MPE_LOG_EVENT(EVENT, INTDATA, CHARDATA)
INTEGER EVENT,
INTDATA CHARACTER*(*) CHARDATA

Эти процедуры позволяют пользователю включать только те события, которые ему интересны в данной программе. Базовые процедуры – **MPE_Init_log**, **MPE_Log_event** и **MPE_Finish_log**.

MPE_Init_log должна вызываться всеми процессами, чтобы инициализировать необходимые структуры данных. **MPE_Finish_log** собирает отчеты из всех процессов, объединяет их, выравнивает по общей шкале времени. Затем процесс с номером 0 в коммуникаторе **MPI_COMM_WORLD** записывает отчет в файл, имя которого указано в аргументе. Одиночное событие устанавливается процедурой **MPE_Log_event**, которая задает тип события (выбирает пользователь), целое число и строку для данных. Чтобы разместить логфайл, который будет нужен для анализа или для программы визуализации (подобной upshot), процедура **MPE_Describe_state** позволяет добавить события и описываемые состояния, указать точку старта и окончания для каждого состояния. При желании для визуализации отчета можно использовать цвет. **MPE_Stop_log** и **MPE_Start_log** предназначены для того, чтобы динамически включать и выключать создание отчета.

Эти процедуры используются в одной из профилирующих библиотек, поставляемых с дистрибутивом для автоматического запуска событий библиотечных вызовов MPI.

Две переменные среды **TMPDIR** и **MPE LOG FORMAT** нужны пользователю для установки некоторых параметров перед генерацией логфайлов.

MPE LOG FORMAT – определяет формат логфайла, полученного после исполнения приложения, связанного с MPE–библиотекой. **MPE LOG FORMAT** может принимать только значения **CLOG**, **SLOG** и **ALOG**. Когда **MPE LOG FORMAT** установлен в **NOT**, предполагается формат **CLOG**.

TMPDIR – описывает директорий, который используется как временная память для каждого процесса. По умолчанию, когда **TMPDIR** есть **NOT**, будет использоваться “**/tmp**”. Когда пользователю нужно получить очень длинный логфайл для очень длинной MPI–работы, пользователь должен убедиться, что **TMPDIR** достаточно велик, чтобы хранить временный логфайл, который будет удален, если объединенный файл будет создан успешно.

2.4. СРЕДСТВА ПРОСМОТРА ЛОГФАЙЛОВ

Существует четыре графических средства визуализации, распространяемых вместе с MPE: upshot, nupshot, Jumpshot-2 и Jumpshot-3. Из этих четырех просмотрщиков логфайлов только три построены с помощью MPE. Это upshot, Jumpshot-2 и Jumpshot-3.

Upshot и Nupshot. Один из используемых инструментов называется upshot. Самый простой вид экрана Upshot показан на рис. 2.2. Результат представлен в виде полос, по одной полосе на каждый процесс. Каждая полоса состоит из участков разного цвета (для черно-белых мониторов цвет заменяется различными штриховками). Каждому цвету соответствует состояние процесса определенного типа. Внизу приведены отметки времени, начинающиеся от нуля. Такое представление позволяет визуально определить состояние любого процесса в любой момент времени.

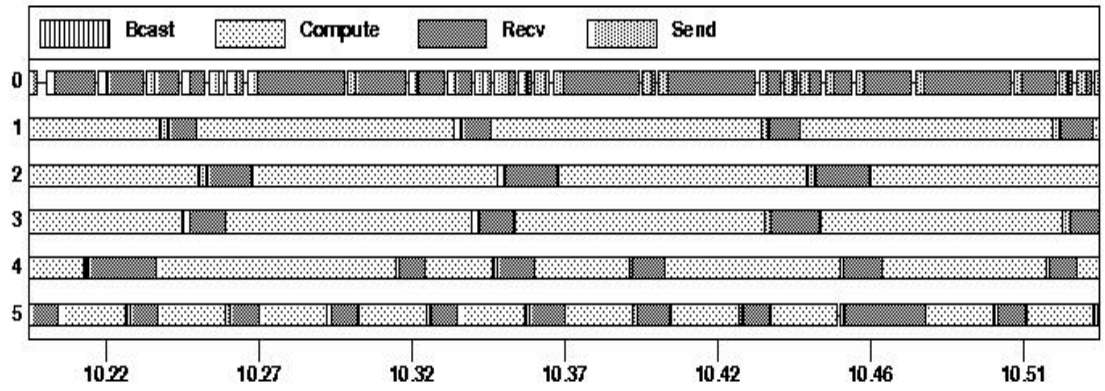


Рис. 2.2. Представление результатов профилирования с помощью upshot

Существуют и более сложные изображения окон, которые позволяют менять размер изображения по горизонтали или вертикали, центрировать на любой точке дисплея, выбранной мышью.

Jumpshot-2 и Jumpshot-3. Существует еще две версии, поставляемых вместе с MPE. Это Jumpshot-2 и Jumpshot-3, развившиеся из Upshot и Nupshot. Обе написаны на Java и являются графическими средствами визуализации для интерпретации двоичных файлов трасс, которые показывают их на дисплее.

Конкретные сведения по средствам просмотра обычно представлены в соответствующей реализации MPICH.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ К ГЛАВЕ 2

Контрольные вопросы к 2.1

1. Что такое реализация MPI?
2. Что такое MPICH?
3. Укажите основные функции MPICH.
4. Дайте определение интерфейса абстрактного прибора ADI.

5. Какие типы ADI известны?
6. Что такое канальный интерфейс?
7. Что такое прямая реализация MPICH?
8. Какие протоколы используются для обмена сообщениями между процессами?

Контрольные вопросы к 2.2

1. Как производится запуск с помощью команды MPIRun?
2. Как задается число процессов при запуске с MPIRun?
3. Что выполняет опция –localonly?
4. Для чего используется процедура MPIConfig?
5. Для чего используется команда MPIRegister?

Контрольные вопросы к 2.3

1. Что такое библиотека MPE, ее назначение?
2. Основные функции MPE?
3. Что такое профилирование?
4. Какова методика оценки эффективности вычислений?
5. Что такое регистрация, логфайлы?
6. Как создается файл регистрации?
7. Какие процедуры используются при создании логфайлов?
8. Опишите процесс создания логфайлов на примере программы вычисления числа π.

Контрольные вопросы к 2.4

1. Какие способы анализа логфайлов Вы знаете?
2. Какое различие между форматами логфайлов ALOG, CLOG и SLOG?
3. Какие средства просмотра логфайлов графического типа Вы знаете?
4. Объясните структуру изображения, создаваемого инструментом upshot?

Р А З Д Е Л 2. **БИБЛИОТЕКА ФУНКЦИЙ MPI**

Глава 3. ПАРНЫЕ МЕЖПРОЦЕССНЫЕ ОБМЕНЫ

3.1. ВВЕДЕНИЕ

Главы 3 – 6 написаны в соответствии со стандартом MPI-1.2 [20, 11], также использовались книги [4, 5, 6,7] и другие документы.

Передача и прием сообщений процессами – это базовый коммуникационный механизм MPI. Основными операциями парного обмена являются операции **send** (послать) и **receive** (получить). Их использование иллюстрируется следующим примером:

```
#include "mpi.h"
main( argc, argv ) 7
int argc;
char **argv;
{   char message[20];
    int myrank;
    MPI_Status status;

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );

   if (myrank == 0)                          /* код для процесса 0 */
   {    strcpy(message,"Hello, there");
        MPI_Send(message,strlen(message),MPI_CHAR,1,99,
                    MPI_COMM_WORLD);
   }
   else                                      /* код для процесса 1 */
   {   MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD,
                    &status);
      printf("received :%s:\n", message);
   }
   MPI_Finalize();
}
```

В этом примере процесс с номером 0 (**myrank = 0**) посылает сообщение процессу с номером 1, используя операцию посылки **MPI_Send**. Эта операция описывает буфер посылающего процесса, из которого извлекаются посылаемые данные. В приведенном примере посылающий буфер состоит из накопителя в памяти процесса 0, содержащего переменную **message**. Размещение, размер и тип буфера

посылающего процесса описываются первыми тремя параметрами операции **send.** Посланное сообщение будет содержать 13 символов этой переменной. Операция посылки также связывает с сообщением его атрибуты. Атрибуты определяют номер процесса-получателя сообщения и содержат различную информацию, которая может быть использована операцией **receive**, чтобы выбрать определенное сообщение среди других. Последние три параметра операции посылки описывают атрибуты посланного сообщения. Процесс 1 (**myrank = 1**) получает это сообщение, используя операцию приема **MPI_Recv,** и данные сообщения записываются в буфер процесса-получателя. В приведенном примере буфер получателя состоит из накопителя в памяти процесса один, содержащего строку **message**. Первые три параметра операции приема описывают размещение, размер и тип буфера приема. Следующие три параметра необходимы для выбора входного сообщения. Последний параметр необходим для возврата информации о только что полученном сообщении.

3.2. ОПЕРАЦИИ БЛОКИРУЮЩЕЙ ПЕРЕДАЧИ И БЛОКИРУЮЩЕГО ПРИЕМА

3.2.1. Блокирующая передача

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	**buf**	начальный адрес буфера посылки сообщения (альтернатива)
IN	**count**	число элементов в буфере посылки (неотрицательное целое)
IN	**datatype**	тип данных каждого элемента в буфере передачи (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**tag**	тэг сообщения (целое)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM,  IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

void MPI::Comm::Send (const void* buf, int count,  const MPI::Datatype& datatype,
                      int dest, int tag) const
```

Семантика этого блокирующего вызова описана в 3.4.

3.2.2. Данные в сообщении

Буфер посылки описывается операцией **MPI_SEND**, в которой указано количество последовательных элементов, тип которых указан в поле **datatype,** начиная с элемента по адресу **buf.** Длина сообщения задается числом *элементов*, а не числом *байт*.

Число данных **count** в сообщении может быть равно нулю, это означает, что область данных в сообщении пуста. Базисные типы данных в сообщении соответствуют базисным типам данных используемого языка программирования. Список возможного соответствия этих типов данных для языка Fortran и MPI представлен ниже.

MPI datatype	**Fortran datatype**
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

Список соответствия типов данных для языка C и MPI дан ниже.

MPI datatype	**C datatype**
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	

Типы **MPI_BYTE** и **MPI_PACKED** не имеют соответствия в языках C или Fortran. Значением типа **MPI_BYTE** является байт. Байт не интерпретируется и отличен от символа. Различные машины могут иметь различное представление для символов или могут использовать

для представления символов более одного байта. С другой стороны, байт имеет то же самое двоичное значение на всех машинах.

3.2.3. Атрибуты сообщения

В дополнение к описанию данных сообщение несет информацию, которая используется, чтобы различать и выбирать сообщения. Эта информация состоит из фиксированного количества полей, которые в совокупности называются **атрибутами** сообщения. Эти поля таковы: **source, destination, tag, communicator** (номер процесса-отправителя сообщения, номер процесса-получателя, тэг, коммуникатор).

Целочисленный аргумент тэг используется, чтобы различать типы сообщений. Диапазон значений тэга находится в пределах 0,…,UB, где верхнее значение UB зависит от реализации. MPI требует, чтобы UB было не менее 32767.

Аргумент **comm** описывает коммуникатор, который используется в операции обмена. Коммуникатор описывает коммуникационный контекст коммуникационной операции. Сообщение всегда принимается внутри контекста, в котором оно было послано; сообщения, посланные в различных контекстах, не взаимодействуют.

Коммуникатор также описывает ряд процессов, которые разделяют этот коммуникационный контекст. Эта группа процессов упорядочена, и процессы определяются их номером внутри этой группы: диапазон значений для **dest** есть 0,...,n-1, где n есть число процессов в группе

В MPI предопределен коммуникатор **MPI_COMM_WORLD**. Он разрешает обмен для всех процессов, которые доступны после инициализации MPI, и процессы идентифицируются их номерами в группе **MPI_COMM_WORLD**.

3.2.4. Блокирующий прием

MPI_RECV (buf, count, datatype, source, tag, comm, status)

OUT	**buf**	начальный адрес буфера процесса-получателя (альтернатива)
IN	**count**	число элементов в принимаемом сообщении (целое)
IN	**datatype**	тип данных каждого элемента сообщения (дескриптор)
IN	**source**	номер процесса-отправителя (целое)
IN	**tag**	тэг сообщения (целое)
IN	**comm**	коммуникатор (дескриптор)
OUT	**status**	параметры принятого сообщения (статус)

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

```
MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM,  STATUS,
              IERROR)

<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM,
STATUS(MPI_STATUS_SIZE), IERROR

void MPI::Comm::Recv (void* buf, int count, const MPI::Datatype& datatype,
                      int source, int tag) const
```

Буфер получения состоит из накопителя, содержащего последовательность элементов, размещенных по адресу **buf**.

Тип элементов указан в поле **datatype**. Длина получаемого сообщения должна быть равна или меньше длины буфера получения, в противном случае будет иметь место ошибка переполнения. Если сообщение меньше размера буфера получения, то в нем модифицируются только ячейки, соответствующие длине сообщения.

Прием сообщения осуществляется, если его атрибуты соответствуют значениям источника, тэга и коммуникатора, которые указаны в операции приема. Процесс-получатель может задавать значение **MPI_ANY_SOURCE** для отправителя и/или значение **MPI_ANY_TAG** для тэга, определяя, что любой отправитель и/или тэг разрешен. Нельзя задать произвольное значение для **comm**. Следовательно, сообщение может быть принято, если оно адресовано данному получателю и имеет соответствующий коммуникатор.

Тэг сообщения задается аргументом **tag** операции приема. Аргумент отправителя, если он отличен от **MPI_ANY_SOURCE**, задается как номер внутри группы процессов, связанной с тем же самым коммуникатором. Следовательно, диапазон значений для аргумента отправителя есть **{0,...,n-1}**$\cup${**MPI_ANY_SOURCE**}, где **n** есть количество процессов в этой группе.

Отметим ассиметрию между операциями посылки и приема. Операция приема допускает получение сообщения от произвольного отправителя, в то время как в операции посылки должен быть указан уникальный получатель.

Допускается ситуация, когда имена источника и получателя совпадают, то есть процесс может посылать сообщение самому себе (это небезопасно, поскольку может привести к дедлоку (**deadlock**)).

3.2.5. Возвращаемая статусная информация

Источник или тэг принимаемого сообщения могут быть неизвестны, если в операции приема были использованы значения типа **ANY**.

Иногда может потребоваться возвратить различные коды ошибок для каждого запроса. Эта информация возвращается с помощью аргумента **status** операции **MPI_RECV.**

Тип аргумента **status** определяется MPI. Статусные переменные размещаются пользователем явно, то есть они не являются системными объектами.

В языке С **status** есть структура, которая содержит три поля, называемые **MPI_SOURCE, MPI_TAG** и **MPI_ERROR**. Следовательно, **status.MPI_SOURCE**, **status.MPI_TAG** и **status.MPI_ERROR** содержат источник, тэг и код ошибки принятого сообщения.

В языке Fortran **status** есть массив целых значений размера **MPI_STATUS_SIZE**. Константы **MPI_SOURCE, MPI_TAG** и **MPI_ERROR** определяют объекты, которые хранят поля источника, тэга и ошибки. Следовательно, **status(MPI_SOURCE), status(MPI_TAG)** и **status(MPI_ERROR)** содержат соответственно источник, тэг и код ошибки принимаемого сообщения.

Вызовы передачи сообщений не модифицируют значения полей кода ошибки статусных переменных. Статусный аргумент также возвращает информацию о длине принятого сообщения. Эта информация не является доступной непосредственно, как поле статусной переменной, и требуется вызов **MPI_GET_COUNT**, чтобы «декодировать» эту информацию.

MPI_GET_COUNT(status, datatype, count)
IN **status** статус операции приема (статус)
IN **datatype** тип данных каждого элемента приемного буфера (дескриптор)
OUT **count** количество полученных единиц (целое)

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

MPI_GET_COUNT(STATUS, DATATYPE, COUNT, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

int Status :: Get_count (const MPI :: Datatype& datatype) const

Операция **MPI_GET_COUNT** возвращает число полученных элементов. Аргумент **datatype** следует сопоставлять с аргументом из операции приема, которая устанавливает статусную переменную.

3.3. СООТВЕТСТВИЕ ТИПОВ ДАННЫХ И ПРЕОБРАЗОВАНИЕ ДАННЫХ

3.3.1. Правила соответствия типов данных

Передача данных содержит следующие три фазы:

1) данные выталкиваются из буфера процесса-отправителя, и части сообщения объединяются;
2) сообщение передается от отправителя к получателю;
3) данные выделяются из получаемого сообщения и помещаются в буфер получателя.

Соответствие типов должно отслеживаться на каждой из трех фаз:

- тип каждой переменной в буфере посылки должен соответствовать типу, указанному для этого элемента в операции посылки;
- тип, описанный в операции посылки, должен соответствовать типу, указанному в операции приема;
- тип каждой переменной в приемном буфере должен соответствовать типу, указанному для нее в операции приема.

Программа, в которой не соблюдаются эти три правила, является неверной.

Типы отправителя и получателя (фаза два) соответствуют друг другу, если обе операции используют одинаковые названия. Это означает, что **MPI_INTEGER** соответствует **MPI_INTEGER, MPI_REAL** соответствует **MPI_REAL**, и так далее.

Тип переменной в хост–программе (главной программе) соответствует типу, указанному в операции обмена, если название типа данных, используемое этой операцией, соответствует базисному типу переменной хост–программы. Например, элемент с названием типа **MPI_INTEGER** соответствует в языке Fortran переменной типа **INTEGER**. Таблица, описывающая соответствие для языков Fortran и C, представлена в параграфе 3.2.2.

Имеется два исключения из этого последнего правила: элемент с названием типа **MPI_BYTE** или **MPI_PACKED** может соответствовать любому байту памяти (на байт-адресуемой машине), без учета типа переменной, которая содержит этот байт. Тип **MPI_PACKED** используется для передачи данных, которые были явно упакованы, или для получения данных, которые будут явно распакованы (3.10). Тип **MPI_BYTE** позволяет передавать двоичное значение байта из памяти.

Правила соответствия типов можно разделить на три категории:

- Коммуникация типизированных значений: типы данных соответствующих элементов в программе передачи, в вызове операции передачи, в вызове операции приема и в программе приема должны соответствовать друг другу.
- Коммуникация нетипизированных значений: нет никаких требований по типам соответствующих элементов в передающей и принимающей программах.
- Коммуникация, применяющая упакованные данные, где используется **MPI_PACKED**.

Пример 3.1. Отправитель и получатель указывают типы соответствия.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
   CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE
   CALL MPI_RECV(b(1), 15, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

Код корректен, если a, b – действительные массивы размера ≥ 10.

Пример 3.2. Отправитель и получатель указывают разные типы.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
   CALL MPI_SEND(a(1), 10, MPI_REAL, 1, tag, comm, ierr)
ELSE
   CALL MPI_RECV(b(1), 40, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

Код ошибочен: отправитель и получатель описывают различные типы данных.

Пример 3.3. Отправитель и получатель описывают передачу нетипизированных значений.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF(rank.EQ.0) THEN
   CALL MPI_SEND(a(1), 40, MPI_BYTE, 1, tag, comm, ierr)
ELSE
   CALL MPI_RECV(b(1), 60, MPI_BYTE, 0, tag, comm, status, ierr)
END IF
```

Код правилен безотносительно к типу и размеру a и b (кроме случая, когда эти результаты выходят за границы памяти).

3.3.2. Преобразование данных

Одной из целей MPI является поддержка параллельных вычислений в неоднородной среде. Связь в такой среде может потребовать следующего преобразования данных:

- преобразования типа – изменяется тип данных значения, например округлением REAL в INTEGER;
- преобразования представления – изменяется двоичное представление значения, например, от Hex floating point к IEEE floating point.

Правила соответствия типов приводят к тому, что обмен в MPI никогда не влечет за собой преобразования типов. С другой стороны MPI требует, чтобы преобразование представления выполнялось, когда типизированное значение передается через среды, которые используют различные представления для типов данных этих значений. MPI не описывает правила для преобразования представления. Предполагается, что такое преобразование должно сохранять целые, логические или знаковые значения и преобразовывать значения с плавающей точкой к ближайшему значению, которое может быть представлено на целевой системе.

Во время преобразования с плавающей точкой могут иметь место исключения по переполнению и потере значимости. Преобразование целых также может приводить к исключениям, когда значения, которые могут быть представлены в одной системе, не могут быть представлены в другой системе. Исключения при преобразовании представления приводят к невозможности обмена. Ошибка имеет место либо на операции посылки, либо на операции приема, либо на обеих операциях.

Если значение, посылаемое в сообщении, не типизировано (например, типа **MPI_BYTE**), тогда двоичное представление байта, хранимое на стороне получателя, идентично двоичному представлению байта, загруженного на стороне отправителя. Это сохраняется вне зависимости от того, работают ли отправитель и получатель в одной и той же или различающихся средах.

Никакого преобразования не нужно, когда программа MPI работает в однородной системе – все процессы выполняются в той же самой среде.

Рассмотрим примеры 3.1 – 3.3. Первая программа правильна, если a и b являются действительными массивами размера ≥ 10. Если отправитель и получатель работают в различных средах, тогда десять действительных значений, которые извлекаются из буфера отправителя, будут преобразованы в представление для действительных чисел на приемной стороне прежде, чем они будут записаны в приемный буфер. В то время, как число действительных чисел, извлекаемых из буфера отправителя, равно числу действительных чисел, хранимых в приемном буфере, то число хранимых байтов не обязано быть равным числу загруженных байтов. Например, отправитель может использовать четырехбайтовое представление для действительных чисел, а получатель – восьмибайтовое.

Вторая программа содержит ошибки, и ее поведение является неопределенным.

Третья программа правильная. Точно та же последовательность из сорока байтов, которая была загружена из буфера посылки, будет записана в приемный буфер, даже если получатель и отправитель работают в различных средах. Посланное сообщение имеет точно ту же длину и точно то же двоичное представление, как и принятое сообщение. Если a и b принадлежат к различным типам или они имеют одинаковый тип, но различное представление, то биты, хранимые в приемном буфере, могут кодировать значения, которые отличаются от значений, закодированных теми же битами в буфере передачи.

Преобразование представления также относится и к атрибутам сообщения: источник, приемник и тэг являются целыми числами и могут нуждаться в преобразовании. MPI не поддерижвает межъязыковый обмен. Поведение программы не определено, если сообщение послано процессом в языке C, а принято процессом в языке Fortran.

3.4. КОММУНИКАЦИОННЫЕ РЕЖИМЫ

Вызов **send**, описанный в параграфе 3.2.1, является блокирующим: он не возвращает управления до тех пор, пока данные и атрибуты сообщения не сохранены в другом месте так, чтобы процесс-отправитель мог обращаться к буферу посылки и перезаписывать его. Сообщение может быть скопировано прямо в соответствующий приемный буфер или во временный системный буфер.

Буферизация сообщения связывает операции посылки и приема. Блокирующая передача может завершаться сразу после буферизации сообщения, даже если приемник не выполнил соответствующий при-

ем. С другой стороны, буферизация сообщения может оказаться дорогой, так как она вовлекает дополнительное копирование память–память, и это требует выделения памяти для буферизации. MPI имеет выбор из нескольких коммуникационных режимов, которые позволяют управлять выбором коммуникационного протокола.

Стандартный коммуникационный режим используется в вызове **send**, описанный в параграфе 3.2.1. В этом режиме решение о том, будет ли исходящее сообщение буферизовано или нет, принимает MPI. MPI может буферизовать исходящее сообщение. В таком случае операция посылки может завершиться до того, как будет вызван соответствующий прием. С другой стороны, буферное пространство может отсутствовать или MPI может отказаться от буферизации исходящего сообщения из-за ухудшения характеристик обмена. В этом случае вызов **send** не будет завершен, пока данные не будут перемещены в процесс-получатель.

В стандартном режиме посылка может стартовать вне зависимости от того, выполнен ли соответствующий прием. Она может быть завершена до окончания приема. Посылка в стандартном режиме является нелокальной операцией: она может зависеть от условий приема. Нежелание разрешать в стандартном режиме буферизацию происходит от стремления сделать программы переносимыми. Поскольку при повышении размера сообщения в любой системе буферных ресурсов может оказаться недостаточно, MPI занимает позицию, что правильная (и, следовательно, переносимая) программа не должна зависеть в стандартном режиме от системной буферизации. Буферизация может улучшить характеристики правильной программы, но она не влияет на результат выполнения программы.

Буферизованный режим операции посылки может стартовать вне зависимости от того, инициирован ли соответствующий прием. Однако, в отличие от стандартной посылки, эта операция является локальной, и ее завершение не зависит от приема. Следовательно, если посылка выполнена и никакого соответствующего приема не инициировано, то MPI буферизует исходящее сообщение, чтобы позволить завершиться вызову **send**. Если не имеется достаточного объема буферного пространства, будет иметь место ошибка. Объем буферного пространства задается пользователем (параграф 3.6).

При **синхронном** режиме посылка может стартовать вне зависимости от того, был ли начат соответствующий прием. Посылка будет

завершена успешно, только если соответствующая операция приема стартовала. Завершение синхронной передачи не только указывает, что буфер отправителя может быть повторно использован, но также и отмечает, что получатель достиг определенной точки в своей работе, а именно, что он начал выполнение приема. Если и посылка, и прием являются блокирующими операциями, то использование синхронного режима обеспечивает синхронную коммуникационную семантику: посылка не завершается на любой стороне обмена, пока оба процесса не выполнят рандеву в процессе операции обмена. Выполнение обмена в этом режиме не является локальным.

При обмене **по готовности** посылка может быть запущена только тогда, когда прием уже инициирован. В противном случае операция является ошибочной, и результат будет неопределенным. Завершение операции посылки не зависит от состояния приема и указывает, что буфер посылки может быть повторно использован. Операция посылки, которая использует режим готовности, имеет ту же семантику, как и стандартная или синхронная передача. Это означает, что отправитель обеспечивает систему дополнительной информацией (именно, что прием уже инициирован), которая может уменьшить накладные расходы. Вследствие этого в правильной программе посылка по готовности может быть замещена стандартной передачей без влияния на поведение программы (но не на характеристики).

Для трех дополнительных коммуникационных режимов используются три дополнительные функции передачи. Коммуникационный режим отмечается одной префиксной буквой: **B** – для буферизованного, **S** – для синхронного и **R** – для режима готовности.

MPI_BSEND(buf, count, datatype, dest, tag, comm)

IN	**buf**	начальный адрес буфера посылки (альтернатива)
IN	**count**	число элементов в буфере посылки (неотрицательное целое)
IN	**datatype**	тип данных каждого элемента в буфере посылки (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**tag**	тэг сообщения (целое)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_BSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR
```

```
void MPI::Comm::Bsend(const void* buf, int count, const MPI::Datatype& datatype,
                      int dest, int tag) const
```

MPI_SSEND (buf, count, datatype, dest, tag, comm)

IN	**buf**	начальный адрес буфера посылки (альтернатива)
IN	**count**	число элементов в буфере посылки (неотрицательное целое)
IN	**datatype**	тип данных каждого элемента в буфере посылки (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**tag**	тэг сообщения (целое)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_SSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

void MPI::Comm::Ssend(const void* buf, int count, const MPI::Datatype& datatype,
                      int dest, int tag) const
```

MPI_RSEND (buf, count, datatype, dest, tag, comm)

IN	**buf**	начальный адрес буфера посылки (альтернатива)
IN	**count**	число элементов в буфере посылки (неотрицательное целое)
IN	**datatype**	тип данных каждого элемента в буфере посылки (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**tag**	тэг сообщения (целое)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_RSEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, IERROR

void MPI::Comm::Rsend(const void* buf, int count, const MPI::Datatype& datatype,
                      int dest, int tag) const
```

Имеется только одна операция приема, которая может соответствовать любому режиму передачи. Эта операция – блокирующая: она завершается только после того, когда приемный буфер уже содержит новое сообщение. Прием может завершаться перед завершением соответствующей передачи (конечно, он может завершаться только после того, как передача стартует).

3.5. СЕМАНТИКА ПАРНОГО ОБМЕНА МЕЖДУ ПРОЦЕССАМИ

Правильная реализация MPI гарантирует определенные общие свойства парного обмена.

Очередность. Сообщения не обгоняют друг друга: если отправитель последовательно посылает два сообщения в один адрес, то они должны быть приняты в том же порядке. Если получатель инициирует два приема последовательно и оба соответствуют тому же сообщению, то вторая операция не будет принимать это сообщение, если первая операция все еще не выполнена. Это требование облегчает установление соответствия посылки и приема. Оно гарантирует, что программа с передачей сообщений детерминирована, если процессы однопоточные и константа **MPI_ANY_SOURCE** не используется при приеме.

Если процесс имеет однопоточное исполнение, тогда любые два обмена, выполняемые этим процессом, упорядочены.

Если процесс многопоточный, тогда семантика потокового исполнения может быть неопределенной относительно порядка для двух операций посылки, выполняемых двумя различными ветвями. Операции логически являются конкурентами, даже если одна физически предшествует другой. Если две операции приема, которые логически конкурентны, принимают два последовательно посланных сообщения, то два сообщения могут соответствовать двум приемам в различном порядке.

Пример 3.4. Пример необгоняемых сообщений

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.EQ.0) THEN
    CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag, comm, ierr)
    CALL MPI_BSEND(buf2, count, MPI_REAL, 1, tag, comm, ierr)
ELSE
    CALL MPI_RECV(buf1,count,MPI_REAL,0,MPI_ANY_TAG,comm,status,ierr)
    CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

Сообщение, посланное в первой передаче, обязано быть принято в первом приеме; сообщение, посланное во второй передаче, обязано быть принято во втором приеме.

Продвижение обмена. Если пара соответствующих посылки и приема инициализирована двумя процессами, то по крайней мере одна

из этих двух операций будет завершена независимо от других действий в системе: операция посылки будет завершена, несмотря на то, что прием завершен другим сообщением; операция приема будет завершена, не глядя на то, что посланное сообщение будет поглощено другим соответствующим приемом, который был установлен на том же самом процессе-получателе.

Пример 3.5. Пример двух пересекающихся пар.

```
   CALL MPI_COMM_RANK(comm, rank, ierr)
   IF (rank.EQ.0) THEN
      CALL MPI_BSEND(buf1, count, MPI_REAL, 1, tag1, comm, ierr)
      CALL MPI_SSEND(buf2, count, MPI_REAL, 1, tag2, comm, ierr)
   ELSE
!  rank.EQ.1
      CALL MPI_RECV(buf1, count, MPI_REAL, 0, tag2, comm, status, ierr)
      CALL MPI_RECV(buf2, count, MPI_REAL, 0, tag1, comm, status, ierr)
   END IF
```

Оба процесса начинают первый коммуникационный вызов. Поскольку первая посылка процесса с номером 0 использует буферизованный режим, она обязана завершиться безотносительно к состоянию процесса с номером 1. Поскольку никакого соответствующего приема не инициировано, сообщение будет скопировано в буферное пространство. Затем начинается вторая посылка. На этот момент соответствующая пара операций приема и посылки запущена, и обе операции обязаны завершиться. Процесс с номером 1 затем вызывает свою вторую операцию приема, которая получит буферизованное сообщение. Заметим, что процесс 1 получает сообщения в порядке, обратном порядку их передачи.

Однозначность выполнения коммуникаций в MPI должен обеспечить программист. Предположим, что посылка инициирована. Тогда возможно, что процесс-получатель повторно инициирует прием, соответствующий этой посылке, хотя сообщение все еще не принято, поскольку оно всякий раз обгоняется другим сообщением, посланным другим источником. Аналогично предположим, что прием был установлен многопоточным процессом. Тогда возможно, что сообщения, соответствующие этому приему, принимаются повторно, хотя прием все еще не закрыт, поскольку он обгоняется другими приемами, установленными на этом узле. Предупредить зависание в такой ситуации является обязанностью программиста.

Ограничение по ресурсам. Любое выполнение операций обмена предполагает наличие ресурсов, которые могут быть ограниченными. Может иметь место ошибка, когда недостаток ресурсов ограничивает выполнение вызова.

Хорошая реализация MPI обеспечивает фиксированный объем ресурсов для каждой ждущей посылки в режиме готовности или синхронном режиме и для каждого ждущего приема. Однако буферное пространство может быть израсходовано на хранение сообщений, посланных в стандартном режиме, или занято на хранение сообщений, посланных в буферном режиме, когда прием для соответствующей пары недоступен. В таких случаях объем пространства, доступного для буферизации, будет много меньше, чем объем памяти данных на многих системах.

MPI позволяет пользователю обеспечить буферную память для сообщений, посланных в буферизованном режиме. Более того, MPI описывает детализированную операционную модель для использования этого буфера.

Операции буферизованной передачи, которые не могут завершиться из-за недостатка буферного пространства, являются ошибочными. Когда такая ситуация выявлена, появляется сигнал об ошибке, и это может вызвать ненормальное окончание программы. С другой стороны, операция стандартной посылки, которая не может завершиться из-за недостатка буферного пространства, будет просто блокирована до освобождения буферного пространства или установления соответствующего приема.

Это поведение предпочтительно во многих ситуациях. Рассмотрим ситуацию, в которой поставщик многократно генерирует новые значения и посылает их потребителю. Предположим, что генерация производится быстрее, чем потребитель может принять их. Если используются буферизованные посылки, то результатом будет перегрузка буфера. Чтобы предупредить такую ситуацию, в программу необходимо ввести дополнительную синхронизацию. Если используются стандартные передачи, то производитель будет автоматически следовать за блокированием его операций из-за недостатка буферного пространства. В некоторых ситуациях недостаток буферного пространства ведет к дедлоку.

Пример 3.6. Пример корректного обмена сообщениями.

```
  CALL MPI_COMM_RANK(comm, rank, ierr)
  IF (rank.EQ.0) THEN
     CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
     CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag, comm, status, ierr)
  ELSE
     CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag, comm, status, ierr)
     CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
  END IF
```

Эта программа будет успешной, даже если не будет буферного пространства для данных. Операция стандартной передачи данных в этом примере может быть заменена синхронной передачей.

Пример 3.7. Пример некорректного обмена сообщениями.

```
  CALL MPI_COMM_RANK(comm, rank, ierr)
  IF (rank.EQ.0) THEN
     CALL MPI_RECV(recvbuf,count,MPI_REAL,1,tag, comm, status, ierr)
     CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
  ELSE
     CALL MPI_RECV(recvbuf,count,MPI_REAL,0,tag, comm, status, ierr)
     CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
  END IF
```

Операция приема первого процесса обязана завершиться перед его посылкой и может завершиться только в том случае, если выполнена соответствующая посылка второго процесса. Операция приема второго процесса обязана завершиться перед его посылкой и может завершиться только тогда, если выполнена соответствующая посылка первого процесса. Эта программа будет всегда в состоянии взаимного блокирования.

Пример 3.8. Пример обмена: результат зависит от буферизации.

```
  CALL MPI_COMM_RANK(comm, rank, ierr)
  IF (rank.EQ.0) THEN
     CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
     CALL MPI_RECV(recvbuf,count,MPI_REAL,1, tag, comm, status, ierr)
!  rank.EQ.1
  ELSE
     CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
     CALL MPI_RECV(recvbuf,count,MPI_REAL, 0, tag, comm, status, ierr)
END IF
```

Сообщение, посланное каждым процессом, должно быть скопировано перед окончанием операции посылки и до старта операции приема. Чтобы завершить программу, необходимо, чтобы по крайней мере одно из двух сообщений было буферизовано. Значит, программа может быть буферизованной, только если коммуникационная система может буферизовать, по крайней мере, count слов данных.

3.6. РАСПРЕДЕЛЕНИЕ И ИСПОЛЬЗОВАНИЕ БУФЕРОВ

Пользователь может описать буфера, используемые для буферизации сообщений, посылаемых в режиме буферизации. Буферизация выполняется отправителем.

MPI_BUFFER_ATTACH (buffer, size)
IN **buffer** начальный адрес буфера (альтернатива)
IN **size** размер буфера в байтах (целое)

int MPI_Buffer_attach (void* buffer, int size)

MPI_BUFFER_ATTACH (BUFFER, SIZE, IERROR)
<type> BUFFER(*)
INTEGER SIZE, IERROR

void MPI::Attach_buffer(void* buffer, int size)

Предусмотренный в MPI буфер в памяти пользователя используется для буферизации исходящих сообщений. Буфер используется только сообщениями, посланными в буферизованном режиме. Только один буфер может быть присоединен к процессу за один раз.

MPI_BUFFER_DETACH (buffer_addr, size)
OUT **buffer_addr** начальный адрес буфера (альтернатива)
OUT **size** размер буфера в байтах (целое)

int MPI_Buffer_detach(void* buffer_addr, int * size)

MPI_BUFFER_DETACH(BUFFER_ADDR, SIZE, IERROR)
<type> BUFFER_ADDR(*)
INTEGER SIZE, IERROR

Int MPI :: Detach_buffer (void*& buffer)

Вызов возвращает адрес и размер отключенного буфера. Операция будет блокирована, пока находящееся в буфере сообщение не будет передано. После выполнения этой функции пользователь может повторно использовать или перераспределять объем, занятый буфером.

Пример 3.9. Обращение к функциям использования буферов.

```
#define BUFFSIZE 10000
int size
char *buff;
MPI_Buffer_attach( malloc(buf, BUFFSIZE);
                                /* буфер может теперь быть  использован MPI_Bsend */
MPI_Buffer_detach( &buff,&size);        /* размер буфера уменьшен до нуля */
MPI_Buffer_attach( buff, size);         /* буфер на  10000 байтов доступен снова */
```

Если никакого буфера не подключено, то MPI ведет себя, как если бы с процессом был связан буфер нулевого размера.

3.7. НЕБЛОКИРУЮЩИЙ ОБМЕН

На многих системах можно улучшить характеристики путем совмещения во времени процессов обмена и вычислений. Механизмом, который часто приводит к лучшим характеристикам, является **неблокирующий обмен**.

Неблокирующий вызов инициирует операцию посылки, но не завершает ее. Вызов начала посылки будет возвращать управление перед тем, как сообщение будет послано из буфера отправителя. Отдельный вызов для завершения посылки необходим, чтобы завершить обмен, то есть убедиться, что данные уже извлечены из буфера отправителя.

Посылка данных из памяти отправителя может выполняться параллельно с вычислениями, выполняемыми на процессе-отправителе после того, как передача была инициирована до ее завершения. Аналогично, неблокирующий вызов инициирует операцию приема, но не завершает ее. Вызов будет закончен до записи сообщения в приемный буфер. Необходим отдельный вызов завершения приема, чтобы завершить операцию приема и проверить, что данные получены в приемный буфер. Посылка данных в память получателя может выполняться параллельно с вычислениями, производимыми после того, как прием был инициирован, и до его завершения. Использование неблокируемого приема позволит также избежать системной буферизации и копирования память–память, когда информация появилась преждевременно на приемном буфере.

Неблокируемые вызовы начала посылки могут использовать четыре режима: стандартный, буферизуемый, синхронный и по готовности с сохранением их семантики. Передачи во всех режимах, исключая режим по готовности, могут стартовать вне зависимости от того, был

ли инициирован соответствующий прием; неблокируемая посылка по готовности может быть начата, только если инициирован соответствующий прием. Во всех случаях вызов начала посылки является локальным: он заканчивается немедленно, безотносительно к состоянию других процессов. Если при вызове обнаруживается нехватка некоторых системных ресурсов, тогда он не может быть выполнен и возвращает код ошибки.

Вызов завершения посылки заканчивается, когда данные извлечены из буфера отправителя. Если режим передачи синхронный, тогда передача может завершиться, только если соответствующий прием стартовал, то есть прием инициирован и соответствует передаче. В этом случае вызов **send** является нелокальным. Синхронная неблокирующая передача может быть завершена, если перед вызовом **receive** имеет место соответствующий неблокирующий прием. Если используется режим буферизуемой передачи, то сообщение должно быть буферизовано, если не имеется ждущего приема. В этом случае вызов **send** является локальным и обязан быть успешным независимо от состояния соответствующего приема. Если используется стандартный режим передачи, тогда вызов **send** может заканчиваться перед тем, как имеет место соответствующий прием, если сообщение буферизованное. С другой стороны, **send** может не завершаться до тех пор, пока имеет место соответствующий прием и сообщение было скопировано в приемный буфер. Неблокирующие передачи могут соответствовать блокирующим приемам и наоборот.

3.7.1. Коммуникационные объекты

Неблокирующие обмены используют скрытые **запросы**, чтобы идентифицировать операции обмена и сопоставить операцию, которая инициирует обмен с операцией, которая заканчивает его. Они являются системными объектами, которые становятся доступными в процессе обработки. Объект запроса указывает различные свойства операции обмена, такие как режим передачи, связанный с ней буфер обмена, ее контекст, тэг и номер процесса-приемника, которые используются для посылки сообщения, или тэг и номер процесса-отправителя, которые используются для приема. В дополнение этот объект хранит информацию о состоянии ждущих операций обмена.

3.7.2. Инициация обмена

Далее используются те же обозначения, что и для блокирующего обмена: преффикс B, S или R используются для буферизованного, синхронного режима или для режима готовности, преффикс I – для неблокирующего обмена.

MPI_ISEND(buf, count, datatype, dest, tag, comm, request)

IN	**buf**	начальный адрес буфера посылки (альтернатива)
IN	**count**	число элементов в буфере посылки (целое)
IN	**datatype**	тип каждого элемента в буфере посылки (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**tag**	тэг сообщения (целое)
IN	**comm**	коммуникатор (дескриптор)
OUT	**request**	запрос обмена (дескриптор)

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

MPI_ISEND (BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
           IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI::Request Comm::Isend(const void* buf, int count, const MPI::Datatype& datatype,
                         int dest, int tag) const
```

MPI_IBSEND(buf, count, datatype, dest, tag, comm, request)

IN	**buf**	начальный адрес буфера посылки (альтернатива)
IN	**count**	число элементов в буфере посылки (целое)
IN	**datatype**	тип каждого элемента в буфере посылки (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**tag**	тэг сообщения (целое)
IN	**comm**	коммуникатор (дескриптор)
OUT	**request**	запрос обмена (дескриптор)

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

MPI_IBSEND (BUF, COUNT, DATATYPE, DEST, TAG, COMM,
            REQUEST, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI::Request MPI::Comm::Ibsend(const void* buf, int count,
                               const MPI::Datatype& datatype, int dest, int tag) const
```

MPI_ISSEND (buf, count, datatype, dest, tag, comm, request)

IN	**buf**	начальный адрес буфера посылки (альтернатива)
IN	**count**	число элементов в буфере посылки (целое)
IN	**datatype**	тип каждого элемента в буфере посылки (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**tag**	тэг сообщения (целое)
IN	**comm**	коммуникатор (дескриптор)
OUT	**request**	запрос обмена (дескриптор)

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

MPI_ISSEND (BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
            IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI::Request MPI::Comm::Issend(const void* buf, int count,
                               const MPI::Datatype& datatype, int dest, int tag) const
```

MPI_IRSEND (buf, count, datatype, dest, tag, comm, request)

IN	**buf**	начальный адрес буфера посылки (альтернатива)
IN	**count**	число элементов в буфере посылки (целое)
IN	**datatype**	тип каждого элемента в буфере посылки (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**tag**	тэг сообщения (целое)
IN	**comm**	коммуникатор (дескриптор)
OUT	**request**	запрос обмена (дескриптор)

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

MPI_IRSEND (BUF, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
            IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR

MPI::Request MPI::Comm::Irsend(const void* buf, int count,
                               const MPI::Datatype& datatype, int dest, int tag) const
```

MPI_IRECV(buf, count, datatype, source, tag, comm, request)

IN	**buf**	начальный адрес буфера посылки (альтернатива)
IN	**count**	число элементов в буфере посылки (целое)
IN	**source**	тип каждого элемента в буфере посылки (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**tag**	тэг сообщения (целое)
IN	**comm**	коммуникатор (дескриптор)
OUT	**request**	запрос обмена (дескриптор)

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

MPI_IRECV (BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST,
           IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR

MPI::Request MPI::Comm::Irecv(void* buf, int count, const MPI::Datatype& datatype,
                              int source, int tag) const
```

Эти вызовы создают объект коммуникационного запроса и связывают его с дескриптором запроса (аргумент **request**). Запрос может быть использован позже, чтобы узнать статус обмена или чтобы ждать его завершения.

Неблокирующий вызов посылки указывает, что система может стартовать, копируя данные из буфера отправителя. Отправитель не должен обращаться к любой части буфера посылки после того, как вызвана операция неблокируемой передачи, пока посылка не завершится. Неблокирующий прием указывает, что система может стартовать, записывая данные в приемный буфер. Приемник не должен обращаться в любую часть приемного буфера после того, как вызвана операция неблокируемого приема, пока прием не завершен.

3.7.3. Завершение обмена

Чтобы завершить неблокирующий обмен, используются функции **MPI_WAIT** и **MPI_TEST**. Завершение операции посылки указывает, что отправитель теперь может изменять содержимое ячеек буфера посылки (операция посылки сама не меняет содержание буфера). Операция завершения не извещает, что сообщение было получено, но дает сведения, что оно было буферизовано коммуникационной подсистемой. Однако, если был использован синхронный режим, завершение операции посылки указывает, что соответствующий прием был инициирован и что это сообщение будет в конечном итоге принято этим соответствующим получателем.

Завершение операции приема указывает, что приемный буфер содержит принятое сообщение, что процесс-получатель теперь может обращаться к нему и что статусный объект установлен. Это не означает, что операция посылки завершена.

Нулевой дескриптор имеет значение **MPI_REQUEST_NULL**. Дескриптор является активным, если он не является нулевым или неак-

тивным. Состояние **empty** (пусто) возвращает **tag = MPI_ANY_TAG, error = MPI_SUCCESS, source = MPI_ANY_SOURCE**, вызов **MPI_GET_ELEMENTS** и **MPI_TEST_CANCELLED** возвращают **false**, а вызов **MPI_GET_COUNT** возвращает **count = 0**. Переменная состояния устанавливается на **empty**, когда возвращаемое ею значение несущественно. Состояние устанавливается таким образом, чтобы предупредить ошибки из-за устаревшей информации.

MPI_WAIT (request, status)

INOUT	**request**	запрос (дескриптор)
OUT	**status**	объект состояния (статус)

int MPI_Wait (MPI_Request *request, MPI_Status *status)

MPI_WAIT(REQUEST, STATUS, IERROR)
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::Request::Wait (MPI::Status& status)

Обращение к **MPI_WAIT** заканчивается, когда завершена операция, указанная в запросе. Если коммуникационный объект, связанный с этим запросом, был создан вызовом неблокирующей посылки или приема, тогда этот объект удаляется при обращении к **MPI_WAIT**, и дескриптор запроса устанавливается в **MPI_REQUEST_NULL. MPI_WAIT** является нелокальной операцией.

Вызов возвращает в **status** информацию о завершенной операции. Содержание статусного объекта для приемной операции может быть получено, как описано в параграфе 3.2.5.

Разрешается вызывать **MPI_WAIT** с нулевым или неактивным аргументом запроса. В этом случае операция заканчивается немедленно со статусом **empty**.

MPI_TEST (request, flag, status)

INOUT	**request**	коммуникационный запрос (дескриптор)
OUT	**flag**	true, если операция завершена (логический тип)
OUT	**status**	статусный объект (статус)

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

MPI_TEST(REQUEST, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR

bool MPI::Request::Test (MPI::Status& status)

Обращение к **MPI_TEST** возвращает **flag = true**, если операция, указанная в запросе, завершена. В таком случае статусный объект содержит информацию о завершенной операции; если коммуникационный объект был создан неблокирующей посылкой или приемом, то он входит в состояние дедлока, и обработка запроса устанавливается в **MPI_REQUEST_NULL**. Другими словами, вызов возвращает **flag = false**. В этом случае значение статуса не определено. **MPI_TEST** является локальной операцией.

Возвращенный статус для операции приема несет информацию, которая может быть получена, как описано в параграфе 3.2.5. Статусный объект для операции посылки несет информацию, которая может быть получена обращением к **MPI_TEST_CANCELLED** (параграф 3.8). Можно вызывать **MPI_TEST** с нулевым или неактивным аргументом запроса. В таком случае операция возвращает **flag = true** и **empty** для **status**.

Функции **MPI_WAIT** и **MPI_TEST** могут быть использованы как для завершения, так и для приема.

Пример 3.10. Простое использование неблокируемой операции.

```
   CALL MPI_COMM_RANK(comm, rank, ierr)
   IF(rank.EQ.0) THEN
       CALL MPI_ISEND(a(1), 10, MPI_REAL, 1, tag, comm, request, ierr)
!      выполним вычисления до завершения операции посылки
       CALL MPI_WAIT(request, status, ierr)
   ELSE
       CALL MPI_IRECV(a(1), 15, MPI_REAL, 0, tag, comm, request, ierr)
!      выполним  вычисления до завершения операции приема
       CALL MPI_WAIT(request, status, ierr)
    END IF
```

3.7.4. Семантика неблокирующих коммуникаций

Очередность. Операции неблокирующих коммуникаций упорядочены согласно порядку исполнения вызовов, которые инициируют обмен. Требование отсутствия обгона, описанного в параграфе 3.5, расширено на неблокирующий обмен.

Пример 3.11. Установление очереди для неблокирующих операций.

```
   CALL MPI_COMM_RANK(comm, rank, ierr)
   IF (RANK.EQ.0) THEN
       CALL MPI_ISEND(a, 1, MPI_REAL, 1, 0, comm, r1, ierr)
```

```
    CALL MPI_ISEND(b, 1, MPI_REAL, 1, 0, comm, r2, ierr)
ELSE
    CALL MPI_IRECV(a,1,MPI_REAL,0,MPI_ANY_TAG, comm, r1, ierr)
    CALL MPI_IRECV(b, 1, MPI_REAL, 0, 0, comm, r2, ierr)
END IF
CALL MPI_WAIT(r1,status)
CALL MPI_WAIT(r2,status)
```

Первая посылка процесса с номером 0 будет соответствовать первому приему процесса с номером 1, даже если оба сообщения посланы до того, как процесс с номером 1 выполнит тот или другой прием.

Продвижение обмена. Вызов **MPI_WAIT**, который завершает прием, будет в конечном итоге заканчиваться, если соответствующая посылка была начата и не закрыта другим приемом. Если соответствующая посылка неблокирующая, тогда прием должен завершиться, даже если отправитель не выполняет никакого вызова, чтобы завершить передачу. Аналогично, обращение к **MPI_WAIT**, которое завершает посылку, будет заканчиваться, если соответствующий прием инициирован.

Пример 3.12. Иллюстрация семантики продвижения.

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (RANK.EQ.0) THEN
    CALL MPI_SSEND(a, 1, MPI_REAL, 1, 0, comm, ierr)
    CALL MPI_SEND(b, 1, MPI_REAL, 1, 1, comm, ierr)
ELSE
    CALL MPI_IRECV(a, 1, MPI_REAL, 0, 0, comm, r, ierr)
    CALL MPI_RECV(b, 1, MPI_REAL, 0, 1, comm, ierr)
    CALL MPI_WAIT(r, status, ierr)
END IF
```

Код не имеет дедлока. Первая синхронная посылка процесса с номером 0 обязана завершиться после того, как процесс с номером 1 установит соответствующий (неблокирующий) прием, даже если процесс 1 не достиг еще завершения вызова wait. Поэтому процесс с номером 0 будет продолжаться и выполнит вторую посылку, позволяя процессу 1 завершить передачу.

Если **MPI_TEST**, который завершает прием, вызывается повторно с тем же аргументом и соответствующая посылка стартовала, тогда вызов рано или поздно возвратит flag = true, если посылка не закрыта другим приемом. Если **MPI_TEST**, который завершает посылку, по-

вторяется с тем же аргументом и соответствующий прием стартовал, тогда вызов рано или поздно возвратит **flag = true**, если не будет закрыт другой посылкой.

3.7.5. Множественные завершения

Удобно иметь возможность ожидать завершения любой или всех операций в списке, а не ждать только специального сообщения. Вызовы **MPI_WAITANY** или **MPI_TESTANY** можно использовать для ожидания завершения одной из нескольких операций. Вызовы **MPI_WAITALL** или **MPI_TESTALL** могут быть использованы для всех ждущих операций в списке. Вызовы **WAITSOME** или **MPI_TESTSOME** можно использовать для завершения всех разрешенных операций в списке.

MPI_WAITANY (count, array_of_requests, index, status)

IN	**count**	длина списка (целое)
INOUT	**array_of_requests**	массив запросов (массив дескрипторов)
OUT	**index**	индекс дескриптора для завершенной операции (целое)
OUT	**status**	статусный объект (статус)

```
int MPI_Waitany (int count, MPI_Request *array_of_requests,  int *index,
                 MPI_Status *status)

MPI_WAITANY(COUNT, ARRAY_OF_REQUESTS, INDEX, STATUS, IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX,
          STATUS(MPI_STATUS_SIZE),  IERROR

static int MPI::Request::Waitany(int count, MPI::Request array_of_requests[],
                                 MPI::Status& status)
```

Операция блокирует работу до тех пор, пока не завершится одна из операций из массива активных запросов. Если более чем одна операция задействована и может закончиться, выполняется произвольный выбор. Операция возвращает в **index** индекс этого запроса в массиве и возвращает в **status** статус завершаемого обмена. Если запрос был создан операцией неблокирующего обмена, то он удаляется, и дескриптор запроса устанавливается в **MPI_REQUEST_NULL**.

Список **array_of_request** может содержать нуль или неактивные дескрипторы. Если список не содержит активных дескрипторов (список имеет нулевую длину или все элементы являются нулями или неактивны), тогда вызов заканчивается немедленно с **index = MPI_UNDEFINED** и со статусом **empty**.

Выполнение **MPI_WAITANY (count, array_of_requests, index, status**) имеет тот же эффект, что и выполнение **MPI_WAIT (&array_of_requests[i], status)**, где **i** есть значение, возвращенное в аргументе **index** (если значение **index** не **MPI_UNDEFINED**). **MPI_WAITANY** с массивом, содержащим один активный элемент, эквивалентно **MPI_WAIT**.

Функция **MPI_TESTANY** тестирует завершение либо одной либо никакой из операций, связанных с активными дескрипторами. В первом случае она возвращает **flag = true**, индекс этого запроса в массиве **index** и статус этой операции в **status**; если запрос был создан вызовом неблокирующего обмена, то запрос удаляется, и дескриптор устанавливается в **MPI_REQUEST_NULL**. Массив индексируется от нуля в языке Си и от единицы в языке Fortran. В последнем случае (не завершено никакой операции) возвращается **flag = false**, значение **MPI_UNDEFINED** в **index** и состояние аргумента **status** является неопределенным. Массив может содержать нуль или неактивные дескрипторы. Если массив не содержит активных дескрипторов, то вызов заканчивается немедленно с **flag = true, index = MPI_UNDEFINED** и **status** = **empty**. Если массив запросов содержит активные дескрипторы, тогда выполнение **MPI_TESTANY**(count, array_of_requests, index, status) имеет тот же эффект, как и выполнение **MPI_TEST (&array_of_requests[i], flag, status)** для **i = 0, 1 ,..., count-1** в некотором произвольном порядке, пока один вызов не возвратит **flag = true**, или все вызовы не могут быть выполнены. В первом случае индекс устанавливается на последнее значение **i**, и в последнем случае устанавливается в **MPI_UNDEFINED**. **MPI_TESTANY** с массивом, содержащим один активный элемент, эквивалентен **MPI_TEST**.

MPI_TESTANY (count, array_of_requests, index, flag, status)

IN	**count**	длина списка (целое)
INOUT	**array_of_requests**	массив запросов (массив дескрипторов)
OUT	**index**	индекс дескриптора для завершенной операции (целое)
OUT	**flag**	true, если одна из операций завершена (логический тип)
OUT	**status**	статусный объект (статус)

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

MPI_TESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, FLAG, STATUS,
            IERROR)
LOGICAL FLAG
```

```
INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX,
          STATUS(MPI_STATUS_SIZE), IERROR

static bool MPI::Request::Testany (int count, MPI::Request array_of_requests[],
                                   int& index, MPI::Status& status)
```

Функция **MPI_WAITALL** блокирует работу, пока все операции обмена, связанные с активными дескрипторами в списке, не завершатся, и возвращает статус всех операций. Оба массива имеют то же самое количество элементов. Элемент с номером **i** в **array_of_statuses** устанавливается в возвращаемый статус **i**-й операции. Запросы, созданные операцией неблокирующего обмена, удаляются, и соответствующие дескрипторы устанавливаются в **MPI_REQUEST_NULL**. Список может содержать нуль или неактивные дескрипторы. Вызов устанавливает статус каждого такого элемента в состояние **empty**.

Когда один или более обменов, завершенных обращением к **MPI_WAITALL**, оказались неудачны, желательно возвратить специальную информацию по каждому обмену. Функция **MPI_WAITALL** возвращает в таком случае код **MPI_ERR_IN_STATUS** и устанавливает в поля ошибки каждого статуса специфический код ошибки. Этот код будет **MPI_SUCCESS**, если обмен завершен, или другой код, если обмен не состоялся; или он может быть **MPI_ERR_PENDING**, если код не завершен и не в состоянии отказа. Функция **MPI_WAITALL** будет возвращать **MPI_ SUCCESS**, если никакой из запросов не имеет ошибки, или будет возвращать другой код ошибки, если не выполнился по другим причинам (таким как неверный аргумент). В таком случае он не будет корректировать поле ошибки в статусе.

MPI_WAITALL (count, array_of_requests, array_of_statuses)

IN	**count**	длина списков (целое)
INOUT	**array_of_requests**	массив запросов (массив дескрипторов)
OUT	**array_of_statuses**	массив статусных объектов (массив статусов)

```
int MPI_Waitall (int count, MPI_Request *array_of_requests,
                 MPI_Status *array_of_statuses)

MPI_WAITALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_STATUSES,
            IERROR)
INTEGER COUNT, ARRAY_OF_REQUESTS(*)
INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

static void MPI::Request::Waitall (int count, MPI::Request array_of_requests[],
                                   MPI::Status array_of_statuses[])
```

Функция **MPI_TESTALL** возвращает **flag=true**, если обмены, связанные с активными дескрипторами в массиве, завершены. В этом случае каждый статусный элемент, который соответствует активному дескриптору, устанавливается в статус соответствующего обмена; если запрос был размещен вызовом неблокирующего обмена, то он удаляется, и дескриптор устанавливается в **MPI_REQUEST_NULL**. Каждый статусный элемент, который соответствует нулю или неактивному дескриптору, устанавливается в состояние **empty**. В противном случае возвращается **flag=false**, никакие запросы не модифицируются, и значения статусных элементов неопределенные. Это локальная операция.

MPI_TESTALL (count, array_of_requests, flag, array_of_statuses)

IN	**count**	длина списка (целое)
INOUT	**array_of_requests**	массив запросов (массив дескрипторов)
OUT	**flag**	(логический тип)
OUT	**array_of_statuses**	массив статусных объектов(массив статусов)

```
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag,
                MPI_Status  *array_of_statuses)

MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, FLAG,
            ARRAY_OF_STATUSES, IERROR)
LOGICAL FLAG
INTEGER COUNT, ARRAY_OF_REQUESTS(*),
        ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR

static bool MPI::Request::Testall (int count, MPI::Request array_of_requests[],
                                   MPI::Status array_of_statuses[])
```

Функция **MPI_WAITSOME** ожидает, пока, по крайней мере, одна операция, связанная с активным дескриптором в списке, не завершится. Возвращает в **outcount** число запросов из списка **array_of_indices**, которые завершены. Возвращает в первую **outcount** ячейку массива **array_of_indices** индексы этих операций (индекс внутри **array_of_requests**). Возвращает в первую ячейку **outcount** массива **array_of_status** статус этих завершенных операций. Если завершенный запрос был создан вызовом неблокирующего обмена, то он удаляется, и связанный дескриптор устанавливается в **MPI_REQUEST_NULL**.

Если список не содержит активных дескрипторов, то вызов заканчивается немедленно со значением **outcount = MPI_UNDEFINED**.

MPI_WAITSOME (incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN	**incount**	длина массива запросов (целое)
INOUT	**array_of_requests**	массив запросов (массив дескрипторов)
OUT	**outcount**	число завершенных запросов (целое)
OUT	**array_of_indices**	массив индексов операций, которые завершены (массив целых)
OUT	**array_of_statuses**	массив статусных операций для завершенных операций (массив статусов)

```
int MPI_Waitsome (int incount, MPI_Request *array_of_requests, int *outcount,
                  int *array_of_indices, MPI_Status *array_of_statuses)

MPI_WAITSOME (INCOUNT, ARRAY_OF_REQUESTS,  OUTCOUNT,
                  ARRAY_OF_INDICES,  ARRAY_OF_STATUSES, IERROR)
INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT,
    ARRAY_OF_INDICES(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),
    IERROR

static int MPI::Request::Waitsome(int incount, MPI::Request array_of_requests[],
                                  int array_of_indices[], MPI::Status array_of_statuses[])
```

Если один или более обменов, завершенных **MPI_WAITSOME**, не могут быть выполнены, надо возвращать по каждому обмену специфическую информацию.

Аргументы **outcount, array_of_indices** и **array_of_statuses** будут индицировать завершение всех обменов, успешных или неуспешных.

Вызов будет возвращать код ошибки **MPI_ERR_IN_STATUS** и устанавливать поле ошибки каждого возвращенного статуса, чтобы указать на успешное завершение или возвратить специфический код ошибки.

Вызов будет возвращать **MPI_SUCCESS**, если ни один запрос не содержал ошибки, и будет возвращен другой код ошибки, если запрос не может быть выполнен по какой-то причине. В таких случаях поля ошибок статуса не будут корректироваться.

Функция **MPI_TESTSOME** ведет себя подобно **MPI_WAITSOME** за исключением того, что заканчивается немедленно. Если ни одной операции не завершено, она возвращает **outcount = 0**. Если не имеется активных дескрипторов в списке, она возвращает **outcount = MPI_UNDEFINED**.

MPI_TESTSOME (incount, array_of_requests, outcount, array_of_indices, array_of_statuses)

IN **incount**	длина массива запросов (целое)
IN OUT **array_of_requests**	массив запросов (массив дескрипторов)
OUT **outcount**	число завершенных запросов (целое)
OUT **array_of_indices**	массив индексов завершенных операций (массив целых)
OUT **array_of_statuses**	массив статусных объектов завершенных операций (массив статусов)

```
int MPI_Testsome (int incount, MPI_Request *array_of_requests, int *outcount,
                  int *array_of_indices, MPI_Status *array_of_statuses)

MPI_TESTSOME (INCOUNT, ARRAY_OF_REQUESTS,  OUTCOUNT,
                  ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERROR)
INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), OUTCOUNT,
    ARRAY_OF_INDICES(*), ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*),
    IERROR

static int MPI::Request::Testsome(int incount, MPI::Request array_of_requests[],
                                  int array_of_indices[], MPI::Status array_of_statuses[])
```

MPI_TESTSOME является локальной операцией, которая заканчивается немедленно, тогда как **MPI_WAITSOME** будет блокироваться до завершения обменов, если в списке содержится хотя бы один активный дескриптор. Оба вызова выполняют требование однозначности: если запрос на прием повторно появляется в списке запросов, передаваемых **MPI_WAITSOME** или **MPI_TESTSOME**, и соответствующая посылка была инициирована, тогда прием будет завершен успешно, если передача не закрыта другим приемом.

Пример 3.13. Код клиент – сервер (невозможность обмена).

```
   CALL MPI_COMM_SIZE(comm, size, ierr)
   CALL MPI_COMM_RANK(comm, rank, ierr)
!  код клиента
   IF(rank > 0) THEN
      DO WHILE(.TRUE.)
         CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
         CALL MPI_WAIT(request, status, ierr)
      END DO
   ELSE
!     rank=0 – код сервера
      DO i=1, size-1
        CALL MPI_IRECV(a(1,i),n,MPI_REAL,i tag,comm,request_list(i),ierr)
      END DO
```

```
      DO WHILE(.TRUE.)
           CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
           CALL DO_SERVICE(a(1,index))     ! дескриптор одного сообщения
           CALL MPI_IRECV(a(1,index),n,MPI_REAL,index,tag,comm,
                                  request_list(index), ierr)
      END DO
   END IF
```

Пример 3.14. Код с использованием MPI_WAITSOME.

```
   CALL MPI_COMM_SIZE(comm, size, ierr)
   CALL MPI_COMM_RANK(comm, rank, ierr)
!  код клиента
   IF(rank > 0) THEN
      DO WHILE(.TRUE.)
           CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
          CALL MPI_WAIT(request, status, ierr)
      END DO
   ELSE
!     rank=0 – код сервера
      DO i=1, size-1
           CALL MPI_IRECV(a(1,i), n, MPI_REAL,i,tag,comm, requests(i), ierr)
      END DO
      DO WHILE(.TRUE.)
           CALL MPI_WAITSOME(size,request_list,numdone,indices,statuses, ierr)
           DO i=1, numdone
               CALL DO_SERVICE(a(1, indices(i)))
               CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag,
                                      comm, requests(indices(i)), ierr)
           END DO
      END DO
   END IF
```

3.8. ПРОБА И ОТМЕНА

Операции **MPI_PROBE** и **MPI_IPROBE** позволяют проверить входные сообщения без реального их приема. Пользователь затем может решить, как ему принимать эти сообщения, основываясь на информации, возвращенной при пробе (преимущественно на информации, возвращенной аргументом **status**). В частности, пользователь может выделить память для приемного буфера согласно длине опробованного сообщения.

Операция **MPI_CANCEL** позволяет отменить ждущие сообщения. Это необходимо для очистки. Инициация операций получения или отправки связывает пользовательские ресурсы, и может оказаться необходимой отмена, чтобы освободить эти ресурсы.

MPI_IPROBE (source, tag, comm, flag, status)

IN	**source**	номер процесса-отправителя или MPI_ANY_SOURCE (целое)
IN	**tag**	значение тэга или MPI_ANY_TAG (целое)
IN	**comm**	коммуникатор (дескриптор)
OUT	**flag**	(логическое значение)
OUT	**status**	статус (статус)

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

MPI_IPROBE(SOURCE, TAG, COMM, FLAG, STATUS, IERROR)
LOGICAL FLAG
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

bool MPI::Comm::Iprobe(int source, int tag, MPI::Status& status) const

MPI_IPROBE (**source, tag, comm, flag, status**) возвращает **flag = true**, если имеется сообщение, которое может быть получено и которое соответствует образцу, описанному аргументами **source**, **tag**, и **comm**. Вызов соответствует тому же сообщению, которое было бы получено с помощью вызова **MPI_RECV (..., source, tag, comm, status**), выполненного на той же точке программы, и возвращает статус с теми же значениями, которые были бы возвращены **MPI_RECV()**. Другими словами, вызов возвращает **flag = false** и оставляет статус неопределенным. Если **MPI_IPROBE** возвращает **flag = true**, тогда содержание статусного объекта может быть впоследствии получено, как описано в параграфе 3.2.5, чтобы определить источник, тэг и длину опробованного сообщения. Последующий прием, выполненный с тем же самым контекстом и тэгом, возвращенным в **status** вызовом **MPI_IPROBE**, будет получать сообщение, которое соответствует пробе, если после пробы не вмешается какое-либо другое сообщение. Если принимающий процесс многопоточный, ответственность за выполнение условия возлагается на пользователя.

MPI_PROBE (source, tag, comm, status)

IN	**source**	номер источника или MPI_ANY_SOURCE (целое)
IN	**tag**	значение тэга или MPI_ANY_TAG (целое)
IN	**comm**	коммуникатор (дескриптор)
OUT	**status**	статус (статус)

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

MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)
INTEGER SOURCE, TAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::Comm::Probe(int source, int tag, MPI::Status& status) const

MPI_PROBE ведет себя подобно **MPI_IPROBE**, исключая то, что функция **MPI_PROBE** является блокирующей и заканчивается после того, как соответствующее сообщение было найдено.

Аргумент **source** функции **MPI_PROBE** может быть **MPI_ANY_SOURCE**, это позволяет опробовать сообщения из произвольного источника и/или с произвольным тэгом. Однако специфический контекст обмена обязан создаваться только при помощи аргумента **comm**. Сообщение не обязательно должно быть получено сразу после опробования, оно может опробоваться несколько раз перед его получением.

MPI–реализация **MPI_PROBE** и **MPI_IPROBE** нуждается в гарантии продвижения: если обращение к **MPI_PROBE** уже было запущено процессом и посылка, которая соответствует пробе, уже инициирована тем же процессом, то вызов **MPI_PROBE** будет завершен, если сообщение не получено другой конкурирующей операцией приема (которая выполняется другой ветвью опробуемого процесса).

Аналогично, если процесс ожидает выполнения **MPI_IPROBE** и соответствующее сообщение было запущено, то обращение к **MPI_IPROBE** возвратит **flag = true**, если сообщение не получено другой конкурирующей приемной операцией.

Пример 3.15. Использование блокируемой пробы для ожидания входного сообщения.

```
   CALL MPI_COMM_RANK(comm, rank, ierr)
   IF (rank.EQ.0) THEN
      CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
   ELSE
      IF(rank.EQ.1) THEN
         CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
      ELSE
!     rank.EQ.2
         DO i=1, 2
            CALL MPI_PROBE(MPI_ANY_SOURCE, 0, comm, status, ierr)
            IF (status(MPI_SOURCE) = 0) THEN
100            CALL MPI_RECV(i,1,MPI_INTEGER,0,0,comm, status, ierr)
            ELSE
200            CALL MPI_RECV(x,1,MPI_REAL,1,0,comm, status, ierr)
            END IF
         END DO
      END IF
```

Каждое сообщение принимается с правильным типом.

Пример 3.16. Некорректная программа с использованием блокируемой пробы для ожидания входного сообщения.

```
   CALL MPI_COMM_RANK(comm, rank, ierr)
   IF (rank.EQ.0) THEN
      CALL MPI_SEND(i, 1, MPI_INTEGER, 2, 0, comm, ierr)
   ELSE
      IF(rank.EQ.1) THEN
         CALL MPI_SEND(x, 1, MPI_REAL, 2, 0, comm, ierr)
      ELSE
        DO i=1, 2
            CALL MPI_PROBE(MPI_ANY_SOURCE, 0,comm, status, ierr)
            IF (status(MPI_SOURCE) = 0) THEN
100             CALL MPI_RECV(i,1,MPI_INTEGER,MPI_ANY_SOURCE,
                                    0,comm,status, ierr)
            ELSE
200             CALL MPI_RECV(x,1,MPI_REAL,MPI_ANY_SOURCE,0,
                                   comm,status,ierr)
            END IF
        END DO
      END IF
```

Модифицируем пример 3.15, используя **MPI_ANY_SOURCE**, как аргумент **sourse** в двух вызовах приема, обозначенных метками 100 и 200. Теперь программа некорректна: операция приема может получать сообщение, которое отличается от сообщения, опробованного предыдущим обращением к **MPI_PROBE**.

MPI_CANCEL (request)
IN **request** коммуникационный запрос (дескриптор)

```
int MPI_Cancel(MPI_Request *request)

MPI_CANCEL(REQUEST, IERROR)
INTEGER REQUEST, IERROR

void MPI::Request::Cancel () const
```

Обращение к **MPI_CANCEL** маркирует для отмены ждущие неблокирующие операции обмена (передача или прием). Вызов cancel является локальным. Он заканчивается немедленно, возможно перед действительной отменой обмена. После маркировки необходимо завершить эту операцию обмена, используя вызов **MPI_WAIT** или **MPI_TEST** (или любые производные операции). Если обмен отмечен для отмены, то вызов **MPI_WAIT** для этой операции гарантирует за-

вершение, не глядя на активность других процессов (то есть **MPI_WAIT** ведет себя как локальная функция); аналогично, если **MPI_TEST** вызывается повторно в цикле занятого ожидания для отмены обмена, тогда **MPI_TEST** будет неизбежно успешно закончен. Успешная отмена буферизованной передачи освобождает буферное пространство, занятое ждущим сообщением.

Должно выполняться следующее условие: либо отмена имеет успех, либо имеет успех обмен, но не обе ситуации вместе. Если передача маркирована для отмены, то обязательно должна быть ситуация, что когда-либо передача завершается нормально (посланное сообщение принято процессом назначения) или передача отменена успешно (никакая часть сообщения не принята по адресу назначения). Тогда любой соответствующий прием закрывается другой передачей. Если прием маркирован для отмены, то обязан быть случай, когда прием завершился нормально или этот прием успешно отменен (никакая часть приемного буфера не изменена). Тогда любая соответствующая передача должна быть удовлетворена другим приемом. Если операция была отменена, тогда информация об этом будет возвращена в аргумент статуса операции, которая завершает обмен.

MPI_TEST_CANCELLED (status, flag)
IN **status** статус (Status)
OUT **flag** (логический тип)

```
int MPI_Test_cancelled(MPI_Status *status, int *flag)

MPI_TEST_CANCELLED(STATUS, FLAG, IERROR)
LOGICAL FLAG
INTEGER STATUS(MPI_STATUS_SIZE), IERROR

bool MPI::Status::Is_cancelled () const
```

Функция **MPI_TEST_CANCELLED** возвращает **flag = true**, если обмен, связанный со статусным объектом, был отменен успешно. В таком случае все другие поля статуса (такие как **count** или **tag**) не определены. В противном случае возвращается **flag = false**.

3.9. СОВМЕЩЕННЫЕ ПРИЕМ И ПЕРЕДАЧА СООБЩЕНИЙ

Операция **send–receive** комбинирует в одном обращении посылку сообщения одному получателю и прием сообщения от другого отправителя. Получателем и отправителем может быть тот же самый процесс. Эта операция весьма полезна для выполнения сдвига по цепи

процессов. Если для такого сдвига были использованы блокирующие приемы и передачи, тогда нужно корректно упорядочить эти приемы и передачи так, чтобы предупредить циклические зависимости, которые могут привести к дедлоку.

MPI_SENDRECV выполняет операции блокируемой передачи и приема. Передача и прием используют тот же самый коммуникатор, но, возможно, различные тэги. Буфера отправителя и получателя должны быть разделены и могут иметь различную длину и типы данных. Сообщение, посланное операцией **send–receive**, может быть получено обычной операцией приема или опробовано операцией **probe**, **send–receive** может также получать сообщения, посланные обычной операцией передачи.

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)

IN	**sendbuf**	начальный адрес буфера отправителя (альтернатива)
IN	**sendcount**	число элементов в буфере отправителя (целое)
IN	**sendtype**	тип элементов в буфере отправителя (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**sendtag**	тэг процесса-отправителя (целое)
OUT	**recvbuf**	начальный адрес приемного буфера (альтернатива)
IN	**recvcount**	число элементов в приемном буфере (целое)
IN	**recvtype**	тип элементов в приемном буфере (дескриптор)
IN	**source**	номер процесса-отправителя (целое)
IN	**recvtag**	тэг процесса-получателя (целое)
IN	**comm**	коммуникатор (дескриптор)
OUT	**status**	статус (статус)

```
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)

MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST,  SENDTAG,
                RECVBUF, RECVCOUNT,  RECVTYPE, SOURCE, RECVTAG,
                COMM,  STATUS, IERROR)

<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVCOUNT,
        RECVTYPE, SOURCE, RECVTAG, COMM,
        STATUS(MPI_STATUS_SIZE), IERROR

void MPI::Comm::Sendrecv(const void *sendbuf, int sendcount,
const MPI::Datatype& sendtype, int dest, int sendtag, void *recvbuf, int recvcount,
const MPI::Datatype& recvtype, int source, int recvtag, MPI::Status& status) const
```

MPI_SENDRECV_REPLACE выполняет блокируемые передачи и приемы. Тот же самый буфер используется для отправки и получения, так что посланное сообщение замещается полученным.

MPI_SENDRECV_REPLACE (buf, count, datatype, dest, sendtag, source, recvtag, comm, status)

INOUT	**buf**	начальный адрес буфера отправителя и получателя (альтернатива)
IN	**count**	число элементов в буфере отправителя и получателя (целое)
IN	**datatype**	тип элементов в буфере отправителя и получателя (дескриптор)
IN	**dest**	номер процесса-получателя (целое)
IN	**sendtag**	тэг процесса-отправителя (целое)
IN	**source**	номер процесса-отправителя (целое)
IN	**recvtag**	тэг процесса-получателя (целое)
IN	**comm**	коммуникатор (дескриптор)
OUT	**status**	статус (статус)

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

MPI_SENDRECV_REPLACE(BUF, COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS, IERROR)
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, SENDTAG, SOURCE, RECVTAG, COMM, STATUS(MPI_STATUS_SIZE), IERROR

void MPI::Comm::Sendrecv_replace (void* buf, int count, const MPI::Datatype& datatype, int dest, int sendtag, int source, int recvtag, MPI::Status& status) const

Семантика операции **send–receive** похожа на запуск двух конкурирующих потоков, один выполняет передачу, другой – прием, с последующим объединением этих потоков. Во многих случаях удобно описать "фиктивный" отправитель или получатель для коммуникаций. Это упрощает код, который необходим для работы с границами, например, в случае нециклического сдвига, выполненного по вызову **send–receive**. Когда в вызове нужны аргументы отправителя или получателя, вместо номера может быть использовано специальное значение **MPI_PROC_NULL.** Обмен с процессом, который имеет значение **MPI_PROC_NULL,** не дает результата. Передача в процесс с **MPI_PROC_NULL** успешна и заканчивается сразу, как только возможно. Прием от процесса с **MPI_PROC_NULL** успешен и заканчивается сразу, как только возможно без изменения буфера приема. Когда выполняется прием с **source = MPI_PROC_NULL**, тогда статус возвращает **source = MPI_PROC_NULL, tag = MPI_ANY_TAG** и **count = 0**.

3.10. ПРОИЗВОДНЫЕ ТИПЫ ДАННЫХ

До сих пор все парные обмены использовали только непрерывные буфера, содержащие последовательности элементов одного типа. Часто необходимо передавать сообщения, которые содержат значения различных типов или посылать несмежные данные. Одно из решений состоит в том, чтобы упаковать несмежные данные в смежный буфер на стороне отправителя и распаковать обратно на приемной стороне. Это неэффективно, поскольку требуется дополнительная операция копирования память–память на обеих сторонах. MPI обеспечивает механизм для описания общих буферов для несмежных коммуникаций, в которых используются производные типы данных, образуемые конструкторами, описанными в этом разделе.

Универсальный тип данных есть скрытый объект, который описывается двумя составляющими: последовательностью базисных типов и последовательностью целых (байтовых) смещений.

Не требуется, чтобы смещения были положительными, различными или возрастающего порядка. Порядок объектов не обязан совпадать с их порядком в памяти, и объект может появляться более чем один раз.

Последовательность указанных выше пар называется **картой типа**. Последовательность базисных типов данных (смещения игнорируются) есть **сигнатура типа.** Можно использовать дескриптор общего типа данных как аргумент в операциях передачи или приема вместо аргумента базисного типа данных.

Операция **MPI_SEND (buf, 1, datatype,...)** будет использовать буфер посылки, определенный базовым адресом **buf** и общим типом данных, связанным с **datatype**; она будет генерировать сообщение с сигнатурой типа, определенной аргументом **datatype.**

Базисные типы данных, представленные в 3.2.2, – частные случаи универсального типа и являются предопределенными (например, **MPI_INT** есть предопределенный указатель на тип данных с одним элементом типа **int** и смещением равным нулю).

Экстент (extent) типа данных определяется как пространство, от первого байта до последнего байта, занятое элементами этого типа данных, округленное вверх с учетом требований выравнивания данных.

3.10.1. Конструкторы типа данных

Простейшим типом конструктора типа данных является конструктор **MPI_TYPE_CONTIGUOUS**, который позволяет копировать тип данных в смежные области.

MPI_TYPE_CONTIGUOUS (count, oldtype, newtype)

IN	**count**	число повторений (неотрицательное целое)
IN	**oldtype**	старый тип данных (дескриптор)
OUT	**newtype**	новый тип данных (дескриптор)

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

MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, OLDTYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Create_contiguous (int count) const

Новый тип **newtype** есть тип, полученный конкатенацией (сцеплением) **count** копий старого типа **oldtype**.

Функция **MPI_TYPE_VECTOR** является универсальным конструктором, который позволяет реплицировать типы данных в области, которые состоят из блоков равного объема. Каждый блок получается как конкатенация некоторого количества копий старого типа. Пространство между блоками кратно размеру **old datatype**.

MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype)

IN	**count**	число блоков (неотрицательное целое)
IN	**blocklength**	число элементов в каждом блоке (неотрицательное целое)
IN	**stride**	число элементов между началами каждого блока (целое)
IN	**oldtype**	старый тип данных (дескриптор)
OUT	**newtype**	новый тип данных (дескриптор)

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

MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Create_vector (int count, int blocklength,
int stride) const

Функция **MPI_TYPE_HVECTOR** идентична за исключением того, что страйд задается в байтах, а не в элементах. (**H** соответствует слову heterogeneous – неоднородный.)

MPI_TYPE_HVECTOR (count, blocklength, stride, oldtype, newtype)

IN	**count**	число блоков (неотрицательное целое)
IN	**blocklength**	число элементов в каждом блоке (неотрицательное целое)
IN	**stride**	число байтов между началом каждого блока (целое)
IN	**oldtype**	старый тип данных (дескриптор)
OUT	**newtype**	новый тип данных (дескриптор)

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

MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE,
                 NEWTYPE, IERROR)
INTEGER COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Create_hvector (int count, int blocklength,
                                             MPI::Aint stride) const
```

Функция **MPI_TYPE_INDEXED** позволяет реплицировать старый тип **old datatype** в последовательность блоков (каждый блок есть конкатенация **old datatype**), где каждый блок может содержать различное число копий и иметь различное смещение. Все смещения блоков кратны длине старого блока **oldtype**.

MPI_TYPE_INDEXED(count, array_of_blocklengths,
array_of_displacements, oldtype, newtype)

IN	**count**	число блоков
IN	**array_of_blocklengths**	число элементов в каждом блоке (массив неотрицательных целых)
IN	**array_of_displacements**	смещение для каждого блока (массив целых)
IN	**oldtype**	старый тип данных (дескриптор)
OUT	**newtype**	новый тип данных (дескриптор)

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

MPI_TYPE_INDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
                 ARRAY_OF_DISPLACEMENTS, OLDTYPE, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
          ARRAY_OF_DISPLACEMENTS(*), OLDTYPE, NEWTYPE, IERROR

MPI::Datatype MPI::Datatype::Create_indexed(int count,
          const int array_of_blocklengths[], const int array_of_displacements[]) const
```

Функция **MPI_TYPE_HINDEXED** идентична.

MPI_TYPE_HINDEXED (count, array_of_blocklengths, array_of_displacements, oldtype, newtype)

IN **count**	число блоков (неотрицательное целое)
IN **array_of_blocklengths**	число элементов в каждом блоке (массив неотрицательных целых)
IN **array_of_displacements**	смещение каждого блока в байтах (массив целых)
IN **oldtype**	старый тип данных (дескриптор)
OUT **newtype**	новый тип данных (дескриптор)

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

MPI_TYPE_HINDEXED(COUNT, ARRAY_OF_BLOCKLENGTHS,
        ARRAY_OF_DISPLACEMENTS, OLDTYPE,  NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
            ARRAY_OF_DISPLACEMENTS(*), OLDTYPE,  NEWTYPE, IERROR
```

Смещения блоков в массиве **array_of_displacements** задаются в байтах, а не в кратностях ширины старого типа **oldtype**.

MPI_TYPE_STRUCT является общим типом конструктора. Он отличается от предыдущего тем, что позволяет каждому блоку состоять из репликаций различного типа.

MPI_TYPE_STRUCT (count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)

IN **count**	число блоков (целое)
IN **array_of_blocklength**	число элементов в каждом блоке (массив целых)
IN **array_of_displacements**	смещение каждого блока в байтах (массив целых)
IN **array_of_types**	тип элементов в каждом блоке (массив дескрипторов объектов типов данных)
OUT **newtype**	новый тип данных (дескриптор)

```
int MPI_Type_struct(int count, int *array_of_blocklengths,  MPI_Aint
   *array_of_displacements,  MPI_Datatype *array_of_types, MPI_Datatype *newtype)

MPI_TYPE_STRUCT(COUNT, ARRAY_OF_BLOCKLENGTHS,
      ARRAY_OF_DISPLACEMENTS, ARRAY_OF_TYPES, NEWTYPE, IERROR)
INTEGER COUNT, ARRAY_OF_BLOCKLENGTHS(*),
 ARRAY_OF_DISPLACEMENTS(*), ARRAY_OF_TYPES(*), NEWTYPE, IERROR
```

Обращение к **MPI_TYPE_HINDEXED** (count, B, D, oldtype, newtype) эквивалентно обращению к **MPI_TYPE_STRUCT**(count, B, D, T, newtype), где каждый вход T равен oldtype.

3.10.2. Адресные функции и функции экстентов

Смещения в универсальном типе данных задаются относительно начального буферного адреса. Этот начальный адрес "нуль" отмечается 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 путем использования функции **MPI_ADDRESS**.

MPI_ADDRESS (location, address)
IN **location** ячейка в памяти (альтернатива)
OUT **address** адрес ячейки (целое)

```
int MPI_Address(void* location, MPI_Aint *address)

MPI_ADDRESS(LOCATION, ADDRESS, IERROR)
<type> LOCATION(*)
INTEGER ADDRESS, IERROR
```

Функция **MPI_ADDRESS** возвращает байтовый адрес ячейки.

Пример 3.17. Использование **MPI_ADDRESS** для массива.

```
   REAL A(100,100)
   INTEGER I1, I2, DIFF
   CALL MPI_ADDRESS(A(1,1), I1, IERROR)
   CALL MPI_ADDRESS(A(10,10), I2, IERROR)
   DIFF = I2 – I1
!  значение DIFF есть 909*sizeofreal;  значение I1 и  I2 зависят от реализации.
```

Функция **MPI_TYPE_EXTENT** возвращает экстент типа данных.

MPI_TYPE_EXTENT(datatype, extent)
IN **datatype** тип данных (дескриптор)
OUT **extent** экстент типа данных (целое)

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

MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)
INTEGER DATATYPE, EXTENT, IERROR
```

Функция **MPI_TYPE_SIZE** возвращает общий размер в байтах элементов в сигнатуре типа, связанной с **datatype**, то есть общий размер данных в сообщении, которое было бы создано с этим типом данных. Элементы, которые появляются несколько раз в типе данных, подсчитываются с учетом их кратности.

MPI_TYPE_SIZE (datatype, size)
IN **datatype** тип данных (дескриптор)
OUT **size** размер типа данных (целое)

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

MPI_TYPE_SIZE(DATATYPE, SIZE, IERROR)
INTEGER DATATYPE, SIZE, IERROR

int MPI::Datatype::Get_size ( ) const
```

3.10.3. Маркеры нижней и верхней границ

Часто удобно явно указать нижнюю и верхнюю границы карты типа. Это позволяет определить тип данных, который имеет «дыры» в начале или конце, или тип с элементами, которые идут выше верхней или ниже нижней границы. Пользователь может явно описать границы типа данных, которые соответствуют этим структурам. Чтобы достичь этого, вводятся два дополнительных «псевдо-типа данных»: **MPI_LB** и **MPI_UB**, – которые могут быть использованы соответственно для маркировки нижней и верхней границ типа данных. Эти псевдотипы не занимают места (*экстент* (**MPI_LB**) = *экстент* (**MPI_UB**) =0). Они не меняют **size** или **count** типа данных и не влияют на содержание сообщения, созданного с этим типом данных. Они влияют на определение экстента типа данных и, следовательно, влияют на результат репликации этого типа данных конструктором типа данных.

Две функции могут быть использованы для нахождения нижней и верхней границ типа данных.

MPI_TYPE_LB(datatype, displacement)
IN **datatype** тип данных (дескриптор)
OUT **displacement** смещение нижней границы от исходной в байтах (целое)

```
int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)
```

MPI_TYPE_LB(DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR

MPI_TYPE_UB(datatype, displacement)
IN **datatype** тип данных (дескриптор)
OUT **displacement** смещение верхней границы от исходной в байтах (целое)

int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)

MPI_TYPE_UB(DATATYPE, DISPLACEMENT, IERROR)
INTEGER DATATYPE, DISPLACEMENT, IERROR

3.10.4. Объявление и удаление объектов типа данных

Объекты типов данных должны быть объявлены перед их использованием в коммуникации. Объявленный тип данных может быть использован как аргумент в конструкторах типов данных. Базисные типы данных объявлять не нужно, поскольку они предопределены.

MPI_TYPE_COMMIT(datatype)
INOUT **datatype** тип данных, который объявлен (дескриптор)

int MPI_Type_commit(MPI_Datatype *datatype)

MPI_TYPE_COMMIT(DATATYPE, IERROR)
INTEGER DATATYPE, IERROR

void MPI::Datatype::Commit ( )

Операция **commit** объявляет тип данных, то есть формально описывает коммуникационный буфер, но не содержимое этого буфера. Поэтому после того, как тип данных объявлен, он может быть многократно использован, чтобы передавать изменяемое содержимое буфера или различных буферов с различными стартовыми адресами.

MPI_TYPE_FREE (datatype)
INOUT **datatype** тип данных, который освобождается (дескриптор)

int MPI_Type_free(MPI_Datatype *datatype)

MPI_TYPE_FREE (DATATYPE, IERROR)
INTEGER DATATYPE, IERROR

void MPI::Datatype::Free ( )

Функция **MPI_TYPE_FREE** маркирует объекты типа данных, связанные с **datatype** для удаления и установки типа данных в **MPI_DATATYPE_NULL**. Любой обмен, который использует этот тип данных, будет завершен нормально. Производные типы данных, которые произошли от удаленного типа, не меняются.

Пример 3.18. Пример использования **MPI_TYPE_COMMIT.**

```
   INTEGER type1, type2
   CALL MPI_TYPE_CONTIGUOUS(5, MPI_REAL, type1, ierr)
!  создан новый объект типа данных
   CALL MPI_TYPE_COMMIT(type1, ierr)
!  теперь type1 может быть использован для обмена
   type2 = type1
!  type2 может быть использован для обмена
   CALL MPI_TYPE_VECTOR(3, 5, 4, MPI_REAL, type1, ierr)
!  создан новый необъявленный объект типа
   CALL MPI_TYPE_COMMIT(type1, ierr)
!  теперь type1 может быть использован снова для обмена
```

Удаление типа данных не воздействует на другой тип, который был построен от удаленного типа. Система ведет себя как если бы аргументы входного типа данных были переданы конструктору производного типа данных по значению.

3.10.5. Использование универсальных типов данных

Пример 3.19. Пример использования производных типов.

```
   CALL MPI_TYPE_CONTIGUOUS( 2, MPI_REAL, type2, ...)
   CALL MPI_TYPE_CONTIGUOUS( 4, MPI_REAL, type4, ...)
   CALL MPI_TYPE_CONTIGUOUS( 2, type2, type22, ...)
    ...
   CALL MPI_SEND( a, 4, MPI_REAL, ...)
   CALL MPI_SEND( a, 2, type2, ...)
   CALL MPI_SEND( a, 1, type22, ...)
   CALL MPI_SEND( a, 1, type4, ...)
   ...
   CALL MPI_RECV( a, 4, MPI_REAL, ...)
   CALL MPI_RECV( a, 2, type2, ...)
   CALL MPI_RECV( a, 1, type22, ...)
   CALL MPI_RECV( a, 1, type4, ...)
```

Каждой передаче соответствует операция приема. Тип данных может описывать перекрывающиеся элементы. Использование такого типа в операциях приема неверно. Предположим, что выполнена операция **MPI_RECV** (**buf, count, datatype, dest, tag, comm, status**), где карта типа имеет **n** элементов. Принятое сообщение не обязано ни заполнять весь буфер, ни заполнять число ячеек, которое кратно ***n***. Может быть принято любое число ***k*** базисных элементов, где $\mathbf{0} \le k \le \mathbf{count} \cdot n$. Номер полученных базисных элементов может быть получен из статуса с помощью функции **MPI_GET_ELEMENTS**.

Ранее определенная функция **MPI_GET_COUNT** имеет различное поведение. Она возвращает количество полученных «элементов верхнего уровня», то есть количество «копий» типа данных. В предыдущем примере **MPI_GET_COUNT** может возвратить любое целое число ***k***, где $\mathbf{0} \le k \le \mathbf{count} \cdot n$. Если **MPI_GET_COUNT** возвращает ***k***, тогда число принятых базисных элементов (и значение, возвращенное **MPI_GET_ELEMENTS**) есть $n \cdot k$. Если число полученных базисных элементов не кратно ***n***, то есть операция приема не получила общее число «копий» **datatype**, то **MPI_GET_COUNT** возвращает значение **MPI_UNDEFINED**.

MPI_GET_ELEMENTS (status, datatype, count)

IN	**status**	возвращает статус операции приема (статус)
IN	**datatype**	тип данных операции приема (дескриптор)
OUT	**count**	число принятых базисных элементов (целое)

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

MPI_GET_ELEMENTS (STATUS, DATATYPE, COUNT, IERROR)
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE, COUNT, IERROR

int MPI::Status::Get_elements(const MPI::Datatype& datatype) const
```

Пример 3.20. Использование MPI_GET_COUNT,MPI_GET_ELEMENT

```
   CALL MPI_TYPE_CONTIGUOUS(2, MPI_REAL, Type2, ierr)
   CALL MPI_TYPE_COMMIT(Type2, ierr)
   CALL MPI_COMM_RANK(comm, rank, ierr)
   IF(rank.EQ.0) THEN
      CALL MPI_SEND(a, 2, Type2, 1, 0, comm, ierr)
      CALL MPI_SEND(a, 3, Type2, 1, 0, comm, ierr)
   ELSE
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat, Type2, i, ierr)
!    возвращает i=1
```

```
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
!   возвращает i=2
      CALL MPI_RECV(a, 2, Type2, 0, 0, comm, stat, ierr)
      CALL MPI_GET_COUNT(stat,Type2,i,ierr)
!    возвращает i=MPI_UNDEFINED
      CALL MPI_GET_ELEMENTS(stat, Type2, i, ierr)
!   возвращает i=3
   END IF
```

Функция **MPI_GET_ELEMENTS** также может использоваться после операции **probe**, чтобы найти число элементов в опробованном сообщении. Заметим, что две функции **MPI_GET_COUNT** и **MPI_GET_ELEMENTS** возвращают то же самое значение, когда они используются с базисным типом данных.

3.10.6. Примеры

Пример 3.21. Передача и прием части трехмерного массива.

```
   REAL a(100,100,100), e(9,9,9)
   INTEGER oneslice, twoslice, threeslice, sizeofreal, myrank, ierr
   INTEGER status(MPI_STATUS_SIZE)
!  извлекает часть a(1:17:2, 3:11, 2:10)и запоминает ее в e(:,:,:).
   CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
   CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)
!  создает тип данных для секции 1D
   CALL MPI_TYPE_VECTOR( 9, 1, 2, MPI_REAL, oneslice, ierr)
!  создает тип данных для секции 2D
   CALL MPI_TYPE_HVECTOR(9, 1, 100*sizeofreal, oneslice, twoslice,  ierr)
!  создает тип данных для секции в целом
   CALL MPI_TYPE_HVECTOR( 9, 1, 100*100*sizeofreal, twoslice, threeslice, ierr)
   CALL MPI_TYPE_COMMIT( threeslice, ierr)
   CALL MPI_SENDRECV(a(1,3,2), 1, threeslice, myrank, 0,e, 9*9*9, REAL,
                                 myrank, 0,MPI_COMM_WORLD, status, ierr)
```

Пример 3.22. Копирование нижней треугольной части матрицы

```
   REAL a(100,100), b(100,100)
   INTEGER  disp(100), blocklen(100), ltype, myrank, ierr
   INTEGER status(MPI_STATUS_SIZE)
!  копирует нижнюю треугольную часть массива a  в нижнюю
!  треугольную часть массива b
   CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
!  вычисляет начало и размер каждого столбца
```

```
 DO i=1, 100
        disp(i) = 100*(i-1) + i
        block(i) = 100-i
   END DO
!  создает тип данных для нижней треугольной части
   CALL MPI_TYPE_INDEXED(100,block, disp, MPI_REAL, ltype, ierr)
   CALL MPI_TYPE_COMMIT(ltype, ierr)
   CALL MPI_SENDRECV(a, 1, ltype, myrank, 0, b, 1, ltype, myrank, 0,
                                  MPI_COMM_WORLD,status, ierr)
```

Пример 3.23. Транспонирование матрицы.

```
   REAL a(100,100), b(100,100)
   INTEGER row, xpose, sizeofreal, myrank, ierr
   INTEGER status(MPI_STATUS_SIZE)
!  транспонирование матрицы  a в матрицу b
   CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
   CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)
!  создание типа данных для одной строки
   CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
!   создание типа данных для матрицы с расположением по строкам
   CALL MPI_TYPE_HVECTOR( 100, 1, sizeofreal, row, xpose, ierr)
   CALL MPI_TYPE_COMMIT( xpose, ierr)
!  посылка матрицы с расположением по строкам
!  и получение матрицы с расположением по столбцам
   CALL MPI_SENDRECV( a, 1, xpose, myrank, 0, b, 100*100,
                          MPI_REAL, myrank, 0, MPI_COMM_WORLD, status,ierr)
```

Пример 3.24. Другой способ транспонирования матрицы.

```
   REAL a(100,100), b(100,100)
   INTEGER disp(2), blocklen(2), type(2), row, row1, sizeofreal
   INTEGER  myrank, ierr
   INTEGER status(MPI_STATUS_SIZE)
   CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank)
!  транспонирование матрицы a в матрицу b
   CALL MPI_TYPE_EXTENT( MPI_REAL, sizeofreal, ierr)
!  создание типа данных для одной строки
   CALL MPI_TYPE_VECTOR( 100, 1, 100, MPI_REAL, row, ierr)
!  создание типа данных для одной строки
   disp(1) = 0
   disp(2) = sizeofreal
   type(1)  = row
   type(2)  = MPI_UB
   blocklen(1)  = 1
   blocklen(2)  = 1
```

```
   CALL MPI_TYPE_STRUCT( 2, blocklen, disp, type, row1, ierr)
   CALL MPI_TYPE_COMMIT( row1, ierr)
!  посылка 100 строк и получение с расположением по столбцам
   CALL MPI_SENDRECV( a, 100, row1, myrank, 0, b, 100*100,
                      MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)
```

Пример 3.25. Посылка массива структур.

```
struct Partstruct
  {   int    class;        /* класс частицы */
      double d[6];         /* координаты частицы */
      char   b[7];         /* некоторая дополнительная информация */
  };
struct Partstruct    particle[1000];
int                  i, dest, rank;
MPI_Comm     comm;
                        /* построение типа данных описываемой структуры */
MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int          blocklen[3] = {1, 6, 7};
MPI_Aint     disp[3];
int          base;
                        /* вычисление смещений компонент структуры */
MPI_Address( particle, disp);
MPI_Address( particle[0].d, disp+1);
MPI_Address( particle[0].b, disp+2);
base = disp[0];
for (i=0; i <3; i++) disp[i] -= base;
MPI_Type_struct( 3, blocklen, disp, type, &Particletype);
MPI_Type_commit( &Particletype);
MPI_Send( particle, 1000, Particletype, dest, tag, comm);
```

Пример 3.26. Обработка объединений.

```
union {
          int     ival;
          float   fval;
        } u[1000]
int     utype;
MPI_Datatype   type[2];
int            blocklen[2] = {1,1};
MPI_Aint       disp[2];
MPI_Datatype   MPI_utype[2];
MPI_Aint       i,j;
        /* вычисляет тип данных MPI для каждого возможного типа union;
           считаем, что значения в памяти union выровнены по левой границе.*/
```

```
MPI_Address( u, &i);
MPI_Address( u+1, &j);
disp[0] = 0;
disp[1] = j-i;
type[1] = MPI_UB;
type[0] = MPI_INT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[0]);

type[0] = MPI_FLOAT;
MPI_Type_struct(2, blocklen, disp, type, &mpi_utype[1]);

for(i=0; i<2; i++) MPI_Type_commit(&mpi_utype[i]);
                              /* фактический обмен */
 MPI_Send(u, 1000, MPI_utype[utype], dest, tag, comm);
```

3.11. УПАКОВКА И РАСПАКОВКА

Некоторые существующие библиотеки для передачи сообщений обеспечивают функции **pack/unpack** для передачи несмежных данных. При этом пользователь явно пакует данные в смежный буфер перед их посылкой и распаковывает смежный буфер при приеме. Производные типы данных, которые описаны в параграфе 3.10, позволяют избежать упаковки и распаковки. Пользователь описывает размещение данных, которые должны быть посланы или приняты, и коммуникационная библиотека прямо обращается в несмежный буфер.

Процедуры **pack/unpack** обеспечивают совместимость с предыдущими библиотеками. К тому же они обеспечивают некоторые возможности, которые другим образом недоступны в MPI. Например, сообщение может быть принято в нескольких частях, где приемная операция, сделанная на поздней части, может зависеть от содержания первой части. Другое удобство состоит в том, что исходящее сообщение может быть явно буферизовано в предоставленном пользователю пространстве, превышая возможности системной политики буферизации. Доступность операций **pack** и **unpack** облегчает развитие дополнительных коммуникационных библиотек, расположенных на верхнем уровне MPI.

Операция **MPI_PACK** пакует сообщение в буфер посылки, описанный аргументами **inbuf, incount, datatype** в буферном пространстве, описанном аргументами **outbuf** и **outsize**. Входным буфером может быть любой коммуникационный буфер, разрешенный в **MPI_SEND**. Выходной буфер есть смежная область памяти, содержащая **outsize**

байтов, начиная с адреса **outbuf** (длина подсчитывается в байтах, а не элементах, как если бы это был коммуникационный буфер для сообщения типа **MPI_PACKED**).

MPI_PACK (inbuf, incount, datatype, outbuf, outsize, position, comm)

IN	**inbuf**	начало входного буфера (альтернатива)
IN	**incount**	число единиц входных данных (целое)
IN	**datatype**	тип данных каждой входной единицы (дескриптор)
OUT	**outbuf**	начало выходного буфера (альтернатива)
IN	**outsize**	размер выходного буфера в байтах (целое)
INOUT	**position**	текущая позиция в буфере в байтах (целое)
IN	**comm**	коммуникатор для упакованного сообщения (дескриптор)

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

MPI_PACK(INBUF, INCOUNT, DATATYPE, OUTBUF, OUTSIZE, POSITION,
         COMM, IERROR)
<type> INBUF(*), OUTBUF(*)
INTEGER INCOUNT, DATATYPE, OUTSIZE, POSITION, COMM, IERROR

void MPI::Datatype::Pack (const void* inbuf, int incount, void *outbuf, int outsize,
                          int& position, const MPI::Comm &comm) const
```

Входное значение **position** есть первая ячейка в выходном буфере, которая должна быть использована для упаковки. **position** инкрементируется размером упакованного сообщения, выходное значение **position** есть первая ячейка в выходном буфере, следующая за ячейками, занятыми упакованным сообщением. Аргумент **comm** – коммуникатор, используемый для передачи упакованного сообщения.

Функция **MPI_UNPACK** распаковывает сообщение в приемный буфер, описанный аргументами **outbuf, outcount, datatype** из буферного пространства, описанного аргументами **inbuf** и **insize**.

Выходным буфером может быть любой буфер, разрешенный в **MPI_RECV**. Входной буфер есть смежная область памяти, содержащая **insize** байтов, начиная с адреса **inbuf**. Входное значение **position** есть первая ячейка во входном буфере, занятом упакованным сообщением. **position** инкрементируется размером упакованного сообщения, так что выходное значение **position** есть первая ячейка во входном буфере после ячеек, занятых сообщением, которое было упаковано. **comm** есть коммуникатор для приема упакованного сообщения.

MPI_UNPACK (inbuf, insize, position, outbuf, outcount, datatype, comm)

IN	**inbuf**	начало входного буфера (альтернатива)
IN	**insize**	размер входного буфера в байтах (целое)
INOUT	**position**	текущая позиция в байтах (целое)
OUT	**outbuf**	начало выходного буфера (альтернатива)
IN	**outcount**	число единиц для распаковки (целое)
IN	**datatype**	тип данных каждой выходной единицы данных (дескриптор)
IN	**comm**	коммуникатор для упакованных сообщений (дескриптор)

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

MPI_UNPACK(INBUF, INSIZE, POSITION, OUTBUF,  OUTCOUNT, DATATYPE,
                  COMM, IERROR)
<type> INBUF(*), OUTBUF(*)
INTEGER INSIZE, POSITION, OUTCOUNT, DATATYPE, COMM, IERROR

void MPI::Datatype::Unpack (const void* inbuf, int insize, void *outbuf, int outcount,
                                  int& position, const MPI::Comm& comm) const
```

Чтобы понять поведение **pack** и **unpack**, предположим, что часть данных сообщения есть последовательность, полученная конкатенацией последующих значений, посланных в сообщении. Операция **pack** сохраняет эту последовательность в буферном пространстве, как при посылке сообщения в этот буфер. Операция **unpack** обрабатывает последовательность из буферного пространства, как при приеме сообщения из этого буфера. Несколько сообщений могут быть последовательно упакованы в один объект. Это достигается несколькими последовательными связанными обращениями к **MPI_PACK**, где первый вызов обеспечивает **position = 0**, и каждый последующий вызов вводит значение **position**, которое было выходом для предыдущего вызова, и то же самое значение для **outbuf, outcount** и **comm.** Этот упакованный объект содержит эквивалентную информацию, которая хранилась бы в сообщении по одной передаче с буфером передачи, который есть «конкатенация» индивидуальных буферов передачи.

Упакованный объект может быть послан типом **MPI_PACKED**. Любая парная или коллективная коммуникационная функция может быть использована для передачи последовательности байтов, которая формирует упакованный объект, из одного процесса в другой. Этот упакованный объект также может быть получен любой приемной операцией, поскольку типовые правила соответствия ослаблены для сообщений, посланных с помощью **MPI_PACKED**.

Сообщение, посланное с любым типом может быть получено с помощью **MPI_PACKED**. Такое сообщение может быть распаковано обращением к **MPI_UNPACK**.

Упакованный объект может быть распакован в несколько последовательных сообщений. Это достигается несколькими последовательными обращениями к **MPI_UNPACK**, где первое обращение обеспечивает **position = 0,** и каждый последовательный вызов вводит значение **position**, которое было выходом предыдущего обращения, и то же самое значение для **inbuf, insize** и **comm**.

Конкатенация двух упакованных объектов не обязательно является упакованным объектом; подстрока упакованного объекта также не обязательно есть упакованный объект. Поэтому нельзя производить конкатенацию двух упакованных объектов и затем распаковывать результат как один упакованный объект; ни распаковывать подстроку упакованного объекта, как отдельный упакованный объект. Каждый упакованный объект, который был создан соответствующей последовательностью операций упаковки или регулярными **send**, обязан быть распакован как объект последовательностью связанных распаковывающих обращений. Следующий вызов позволяет пользователю выяснить, сколько пространства нужно для упаковки объекта, что позволяет управлять распределением буферов.

Обращение к **MPI_PACK_SIZE(incount, datatype, comm, size**) возвращает в **size** верхнюю границу по инкременту в **position**, которая создана обращением к **MPI_PACK(inbuf, incount, datatype, outbuf, outcount, position, comm**).

MPI_PACK_SIZE(incount, datatype, comm, size)

IN	**incount**	аргумент count для упакованного вызова (целое)
IN	**datatype**	аргумент datatype для упакованного вызова (дескриптор)
IN	**comm**	аргумент communicator для упакованного вызова (дескриптор)
OUT	**size**	верхняя граница упакованного сообщения в байтах (целое)

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

MPI_PACK_SIZE(INCOUNT, DATATYPE, COMM, SIZE, IERROR)
INTEGER INCOUNT, DATATYPE, COMM, SIZE, IERROR

int MPI::Datatype::Pack_size(int incount, const MPI::Comm& comm) const

Пример 3.27. Пример использования MPI_PACK.

```
int position, i, j, a[2];
char buff[1000];
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
                              / * код отправителя */
if (myrank == 0)
{    position = 0;
     MPI_Pack(&i,1,MPI_INT,buff,1000, &position,MPI_COMM_WORLD);
     MPI_Pack(&j,1,MPI_INT,buff,1000,&position, MPI_COMM_WORLD);
     MPI_Send( buff, position, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
}
else                                  /* код получателя */
     MPI_Recv( a, 2, MPI_INT, 0, 0, MPI_COMM_WORLD)
```

Пример 3.28. Более сложный пример.

```
int position, i;
float a[1000];
char buff[1000]
MPI_Comm_rank(MPI_Comm_world, &myrank);
if (myrank == 0)                        / * код отправителя */
{    int len[2];
     MPI_Aint disp[2];
     MPI_Datatype type[2], newtype;
                    /* построение типа данных для i с последующими a[0]...a[i-1] */
     len[0] = 1;        len[1] = i;
     MPI_Address( &i, disp);
     MPI_Address( a, disp+1);
     type[0] = MPI_INT;   type[1] = MPI_FLOAT;
     MPI_Type_struct( 2, len, disp, type, &newtype);
     MPI_Type_commit( &newtype);
                    /* упаковка i с последующими a[0]...a[i-1]*/
     position = 0;
     MPI_Pack(MPI_BOTTOM,1, newtype,buff,1000, &position,
                MPI_COMM_WORLD);
                                          /* посылка */
     MPI_Send( buff, position, MPI_PACKED, 1, 0,  MPI_COMM_WORLD)
                    /*    можно заменить последние три строки
     MPI_Send( MPI_BOTTOM, 1, newtype, 1, 0, MPI_COMM_WORLD); */
}
```

```
else                    /* myrank == 1 */
{    MPI_Status status;          /* прием */
     MPI_Recv( buff, 1000, MPI_PACKED, 0, 0, &status);
                    /* распаковка i */
    position = 0;
    MPI_Unpack(buff, 1000, &position, &i, 1, MPI_INT,  MPI_COMM_WORLD);
                    /* распаковка a[0]...a[i-1] */
    MPI_Unpack(buff, 1000, &position, a, i, MPI_FLOAT,  MPI_COMM_WORLD);
}
```

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ К ГЛАВЕ 3

Контрольные вопросы к 3.2

1. Что такое атрибуты сообщения?
2. Возвращает ли функция MPI_Send код ошибки, номер процесса, которому адресована передача?
3. Происходит ли возврат из функции MPI_Send, когда можно повторно использовать параметры данной функции или когда сообщение покинет процесс, или когда сообщение принято адресатом или когда адресат инициировал приём данного сообщения?
4. Может ли значение переменной count быть равным нулю?
5. Можно ли в качестве значения тэга в команде посылки передать значение номера процесса в коммуникаторе?
6. Может ли длина буфера получателя быть большей, чем длина принимаемого сообщения? А меньшей?
7. Каков диапазон значений принимаемых константами MPI_ANY_SOURCE, MPI_ANY_TAG?
8. Можно ли в операции посылки использовать MPI_ANY_SOURCE? Почему?
9. Как определить номер процесса-отправителя, если при приеме используются MPI_ANY_SOURCE, MPI_ANY_TAG?
10. Означает ли возврат из функции MPI_Recv, что произошла ошибка?
11. Нужно ли перед вызовом функции MPI_Recv обратиться к функции MPI_Get_Count?
12. Можно ли использовать функцию MPI_Recv, если не известен отправитель сообщения или тэг сообщения?
13. Как определить длину переданного сообщения?
14. Всегда ли status.MPI_SOURCE= SOURCE, status.MPI_TAG= MPI_TAG?

Контрольные вопросы к 3.3

1. Сформулируйте основные правила соответствия типов данных.
2. Правильна ли программа в примере 3.1, если отправитель использует четырехбайтовое представление для действительных чисел, а получатель – восьмибайтовое?
3. Объясните, почему пример 3.3 корректен, хотя посылаем 40 элементов, а в операции приема указываем 60 элементов?
4. Как изменить программу в примере 3.2, чтобы она стала корректна?

5. Влечет ли обмен в MPI преобразование типов данных при их несоответствии?
6. Являются ли MPI–программы, в которых смешаны языки, переносимыми?

Контрольные вопросы к 3.4

1. Что такое стандартный коммуникационный режим?
2. Буферизуется ли исходящее сообщение при стандартном коммуникационном режиме?
3. В каком случае использование функций MPI_Send и MPI_Recv приведет к дедлоку?
4. Перечислите три дополнительных коммуникационных режима.
5. Сколько операций приема существует для трех разных режимов передачи?
6. Почему потребовалось введение различных коммуникационных режимов?
7. Почему посылка в стандартном режиме коммуникации является локальной операцией?
8. Что произойдет, если при буферизованной посылке недостаточно места для сообщения?
9. В чем основное различие между буферизованным и стандартным режимами?
10. Можно ли рассматривать синхронный режим коммуникаций как способ синхронизации процессов?
11. В чем основное различие между синхронным и стандартным режимами?
12. В чем основное различие между стандартным и режимом по готовности?

Контрольные вопросы к 3.5

1. Перечислите основные свойства парного обмена, которые гарантирует правильная реализация MPI.
2. Что произойдет при выполнении программы из примера 3.4. если не будет выполняться свойство очередности?
3. Что произойдет при выполнении программы из примера 3.5. если не будет выполняться свойство продвижения обмена?
4. Гарантируют ли свойства парного обмена, что программа с передачей сообщений детерминирована?
5. В каких ситуациях недостаток буферного пространства ведет к дедлоку?
6. Как разрешить ситуацию дедлока в примере 3.7?
7. В каком случае возникнет ситуация дедлока в примере 3.8?

Контрольные вопросы к 3.6.

1. Необходимы ли функции, описанные в данном разделе, при стандартном коммуникационном режиме? Почему?
2. Что произойдет при вызове функции MPI_Buffer_attach, если недостаточно памяти для буфера?
3. Сколько буферов можно присоединить к процессу за один вызов MPI_Buffer_attach?
4. В каком режиме должны быть посланы сообщения для использования буфера в MPI_Buffer_attach?
5. Когда функция MPI_Buffer_detach может быть вызвана?

Контрольные вопросы к 3.7.

1. В чем различие между блокирующим и неблокирующим обменом?
2. Что такое скрытые запросы?
3. Верен ли вызов функции MPI_IRecv(buf, 1, MPI_INT, 3, tag, comm, &status); для того, чтобы принять одно целое число от процесса 3 (считаем, что все переменные описаны корректно, значение переменной tag – правильное)?
4. Определите понятие “завершение операции посылки” для разных коммуникационных режимов.
5. Определите понятие “завершение операции приема” для разных коммуникационных режимов.
6. Что такое активный и неактивный дескриптор?
7. В чем различие использования MPI_Wait и MPI_Test?
8. Означает ли возврат из функции MPI_Wait, что все процессы дошли до барьера или отправитель вернулся из функции MPI_Send, или получатель вернулся из функции MPI_Recv, или закончилась асинхронно запущенная операция?
9. Как изменится код программы в примере 3.10, если вместо MPI_Wait использовать MPI_Test?
10. Какие значения имеет status в примере 3.10 при вызове MPI_Wait?
11. Какие свойства неблокирующего парного обмена должны быть гарантированы правильной реализацией MPI?
12. Что произойдет при выполнении программы из примера 3.11, если не будет выполняться свойство очередности?
13. Что произойдет при выполнении программы из примера 3.12, если не будет выполняться свойство продвижение обмена?
14. Закончилась одна или все из асинхронно запущенных операций, ассоциированных с указанными в списке параметров идентификаторами requests, если значение параметра flag равно 1 при возврате из функции MPI_TestAll?
15. В чем различие использования функций MPI_Waitany, MPI_Waitall, MPI_Waitsome?
16. В чем различие использования функций MPI_Waitany, MPI_Testany?
17. Если несколько операций из массива активных запросов могут завершиться одновременно при вызове MPI_Waitany, какой номер процесса будет выбран?
18. Что означает код MPI_ERR_Pending в статусе при выполнении функции MPI_Waitall?
19. Почему код, в примере 3.14 с использованием MPI_Waitsome дает правильный результат, в отличие от кода в примере 3.13 с использованием MPI_Waitany, при котором имеет место невозможность обмена?

Контрольные вопросы к 3.8

1. В каком случае возникает необходимость проверять входные сообщения без их реального приема?
2. Какая функция позволяет отменить ждущие сообщения? В каких ситуациях это необходимо?
3. В чем различие в использовании MPI_Probe и MPI_Iprobe?

4. Что может произойти при выполнении кода в примере 3.15, если не использовать блокируемую пробу для ожидания входного сообщения?
5. Что произойдет при выполнении программы в примере 3.15, если количество процессов будет 1 , 2, больше 3?
6. Как изменить код программы в примере 3.16, чтобы она была корректна и выполнялась для количества процессов больше 3?
7. Как завершить операцию обмена, которую маркирует функция MPI_Cancel?
8. Как определить, что операция обмена была отменена функцией MPI_Cancel?

Контрольные вопросы к 3.9

1. Может ли быть получателем и отправителем один и тот же процесс в комбинируемой операции send–receive?
2. Может ли быть получено сообщение обычной операцией приема, если оно послано операцией send–receive?
3. Какой режим обменов осуществляет операция send–receive – блокирующий или неблокирующий?
4. В чем отличие функций MPI_Sendrecv и MPI_Sendrecv_replace?
5. В каком случае может понадобиться "фиктивный" отправитель или получатель для коммуникаций?
6. Каков результат обмена с процессом, который имеет значение MPI_PROC_NULL?
7. Каковы значения статуса и тэга, если прием происходит от процесса, который имеет значение MPI_PROC_NULL?

Контрольные вопросы к 3.10

1. Как осуществить обмен данными, которые не непрерывно расположены в памяти, не используя производные типы данных?
2. Почему лучше один раз вызвать функцию приемопередачи со сложным типом, чем много раз с простым?
3. Есть ли различие при пересылке массива следующими двумя способами:
   MPI_Send(a, 16, MPI_INT, ...);
   MPI_Send(a, 1, intArray16, ...);
   если
   int a[16];
   MPI_Datatype intArray16;
   MPI_Type_contiguous(16, MPI_INT, &intArray16)
   MPI_Type_commit(&intArray16)?
4. Опишите структуру данных, которая создается при вызове:
   MPI_Type_vector(5, 1, 7, MPI_DOUBLE, &newtype);
5. Эквивалентны ли следующие обращения (n – произвольно):
   MPI_Type_contiguous (count, oldtype, newtype);
   MPI_Type_vector (count, 1, 1, oldtype, newtype);
   MPI_Type_vector (1, count, n, oldtype, newtype);
6. В чем различие между производными типами, созданными двумя способами?
   MPI_Type_vector (5, 1, 7, MPI_DOUBLE, &newtype);
   MPI_Type_hvector(5, 1, 7, MPI_DOUBLE, &newtype);

7. Опишите параметры функции
   MPI_Type_struct(3, block_lengths, displs, typelist,message_typ);
   которая позволит создать новый тип данных для осуществления передачи данных следующего типа:
   typedef struct { float a; float b; int n; } INDATA_TYPE
8. Какая операция в С эквивалентна вызову функции MPI_Address?
9. Функция MPI_Type_size возвращает размер элемента в байтах или в количестве элементов старого типа, содержащегося в новом?
10. Нужно ли перед вызовом функции MPI_Type_struct обратиться к функции MPI_Type_commit?
11. Когда и почему необходимо использовать функцию MPI_Type_free?
12. Обьясните разные результаты, полученные функциями MPI_GET_COUNT и MPI_GET_ELEMENTS в примере 3.20.
13. Эквивалентен ли вызов
    MPI_SEND (buf, count, datatype, dest, tag, comm)
    вызовам
    MPI_Type_contiguous(count, datatype, newtype)
    MPI_Type_commit(newtype)
    MPI_Send(buf, 1, newtype, dest, tag, comm)?
14. Сколько производных типов данных создается в примере 3.21? Предложите вариант эквивалентного кода, заменив MPI_TYPE_HVECTOR на MPI_TYPE_VECTOR.
15. Предложите другие способы задания типа, эквивалентные по результату функции MPI_TYPE_INDEXED в примере 3.22.
16. Какое значение получаем при вызове
    CALL MPI_TYPE_EXTENT (MPI_REAL, sizeofreal, ierr)
    в примере 3.23?
17. Сравните программы в примерах 3.23 и 3.24 для решения одной и той же задачи транспонирования матриц. Какой вариант предпочтительнее? Почему?
18. Как выполнить посылку массива структур из примера 3.25, не используя MPI_Type_struct?
19. Как измениться код в примере 3.26, если в union будет полей больше?

Контрольные вопросы к 3.11

1. Какие возможности обеспечивают функции pack/unpack, которые недоступны в MPI другим способом?
2. Укажите разницу при использовании параметра count в разных функциях: MPI_Recv (…, count, …) и MPI_Unpack (…,count,…) .
3. Можно ли распаковывать результат как один упакованный объект, если перед посылкой происходила упаковка двух объектов?
4. Предложите вариант посылки данных в примере 3.26 без использования MPI_Pack.
5. Почему в примере 3.27 не нужно вызывать функцию MPI_Unpack после приема результата?
6. Почему в пример 3.28 используется две функции MPI_Unpack при приеме, если при посылке использовали одну функцию MPI_Pack?

Задания для самостоятельной работы

3.1. Напишите программу, в которой каждый процесс MPI печатает “Hello Word from process i for n”, где i – номер процесса в MPI_COMM_WORLD, а n – размер MPI_COMM_WORLD.

3.2. Напишите программу, в которой определено только два процесса. В одном из них осуществляется генерация случайных чисел, количество которых задает пользователь. Второй процесс получает эти числа, складывает их, посылает результат в первый процесс. Результат выводится на экран.

3.3. Напишите программу, которая рассылает данные от процесса с номером 0 всем другим процессам по конвейеру. Этот означает, что процесс i должен принять данные от i-1 и послать их процессу i + 1, и так до тех пор, пока не будет достигнут последний процесс. Процесс с номером 0 читает данные (целые) от пользователя, пока не будет введено отрицательное целое число.

3.4. Выполните задание 3.3, замкнув вычисления по кругу, т.е. последний процесс передает данные нулевому, который выводит их на экран.

3.5. Выполните задание 3.3, используя обмен MPI_Sendrecv.

3.6. Напишите программу, которая проверяет, в каком порядке осуществляется передача сообщений. Для этого можно допустить, что все процессы за исключением процесса 0 посылают 100 сообщений процессу 0. Пусть процесс 0 распечатывает сообщения в порядке их приема, используя MPI_ANY_SOURCE и MPI_ANY_TAG в MPI_Recv.

3.7. Выполните задание 3.6, используя неблокируемый прием MPI_IRecv.

3.8. Напишите программу для измерения времени передачи вещественных данных двойной точности от одного процесса другому. Для усреднения накладных расходов следует: повторить достаточное количество операций пересылок для получения времени в пределах долей секунды (образцовое решение делает 100000/size итераций для целых size), повторить тестирование несколько раз (например, 10) и усреднить результаты.

3.9. Выполните задание 3.8, используя MPI_SSend. Сравните характеристики с теми, которые получаются при использовании MPI_Send.

3.10. Выполните задание 3.8, используя MPI_BSend. Сравните характеристики с теми, которые получаются при использовании MPI_Send.

3.11. Выполните задание 3.8, используя MPI_RSend. Сравните характеристики с теми, которые получаются при использовании MPI_Send .

3.12. Выполните задание 3.8, используя MPI_ISend, MPI_IRecv, MPI_Wait. Сравните характеристики с теми, которые получаются при использовании MPI_Send и MPI_Recv. Этот тест измеряет эффективную полосу пропускания и задержку, когда процессор одновременно посылает и принимает данные.

3.13. Напишите программу для измерения времени, необходимого для выполнения пересылки вектора из 1000 элементов MPI_DOUBLE со страйдом 24 между элементами. Используйте MPI_Type_vector для описания данных. Используйте те же приемы, как в задании 3.8 для усреднения вариаций и накладных расходов.

3.14. Выполните задание 3.13, используя MPI_Type_struct, чтобы сформировать структуру со страйдом. Сравните результаты с результатами в 3.13.

3.15. Выполните задание 3.13, используя MPI_DOUBLE и цикл для самостоятельной упаковки и распаковки (то есть не используйте типы данных MPI). Сравните результаты с результатами в 3.13, 3.14.

3.16. Напишите программу, которая пересылает процессу (можно самому себе) часть трехмерного массива (задаются значения для каждой размерности), используя производные типы данных MPI_Type_vector, MPI_Type_Hvector.

3.17. Выполните задание 3.16, используя MPI_Pack, MPI_Unpack.

3.18. Напишите программу, которая позволяет копировать нижнюю треугольную часть массива A в нижнюю треугольную часть массива B, используя MPI_Type_indexed.

3.19. Напишите программу транспонирования матрицы, используя типы данных MPI_Type_vector, MPI_Type_Hvector.

3.20. Напишите программу транспонирования матрицы, используя производные типы данных MPI_Type_struct.

Глава 4. КОЛЛЕКТИВНЫЕ ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ

4.1. ВВЕДЕНИЕ

К операциям коллективного обмена относятся (рис. 4.1):

- Барьерная синхронизация всех процессов группы (параграф 4.2.1).
- Широковещательная передача (broadcast) от одного процесса всем остальным процессам группы (параграф 4.2.2).
- Сбор данных из всех процессов группы в один процесс (параграф 4.2.3).
- Рассылка данных одним процессом группы всем процессам группы (параграф 4.2.4).
- Сбор данных, когда все процессы группы получают результат (параграф 4.2.5). Этот вариант представлен на рис. 4.1 как “allgather”.
- Раздача/сбор данных из всех процессов группы всем процессам группы (параграф 4.2.6). Этот вариант называется также полным обменом или all-to-all.
- Глобальные операции редукции, такие как сложение, нахождение максимума, минимума или определенные пользователем функции, где результат возвращается всем процессам группы или в один процесс (параграф 4.3).
- Составная операция редукции и раздачи (параграф 4.3.5).
- Префиксная операция редукции, при которой в процессе i появляется результат редукции 0, 1, …, i, $i \leq n$, где n – число процессов в группе (параграф 4.3.6).

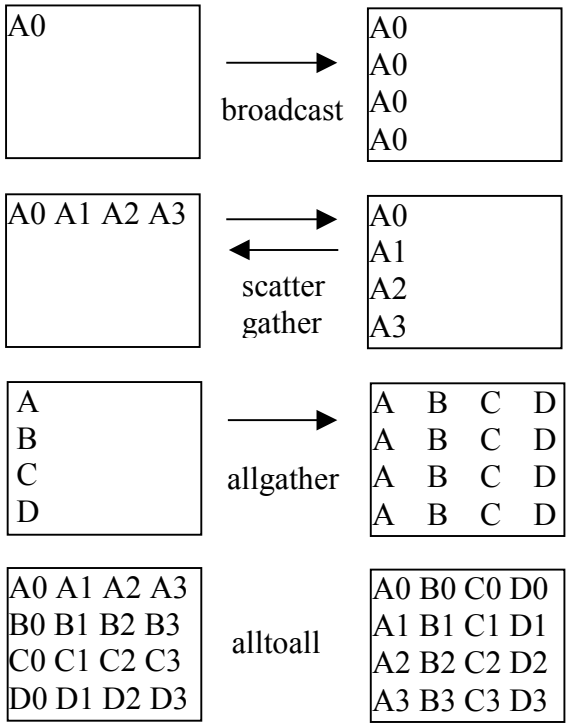


Рис. 4.1. Операции коллективного обмена.
В каждом случае строка прямоугольника представляет данные одного процесса, поэтому, например, в операции broadcast сначала один процесс содержит A0, но после операции все процессы содержат это значение.

Коллективная операция исполняется путем вызова всеми процессами в группе коммуникационных функций с соответствующими аргументами. В коллективных операциях используются основные типы данных, и они должны совпадать у процесса-отправителя и процесса-получателя. Один из ключевых аргументов – это коммуникатор, который определяет группу участвующих в обмене процессов и обеспечивает контекст для этой операции.

Различные коллективные операции (широковещание, сбор данных) имеют единственный процесс-отправитель или процесс-получатель. Такие процессы называются корневыми (root). Некоторые аргументы в коллективных функциях определены как “существенные только для корневого процесса” и игнорируются для всех других участников операции.

Условия соответствия типов для коллективных операций более строгие, чем аналогичные условия для парного обмена. А именно для коллективных операций количество посланных данных должно точно соответствовать количеству данных, описанных в процессе-получа-

теле. Вызов коллективной функции может возвращать управление сразу, как только его участие в коллективной операции завершено. Завершение вызова показывает, что процесс-отправитель может обращаться к буферу обмена. Это не означает, что другие процессы в группе завершили операцию. Таким образом, вызов операции коллективного обмена может иметь эффект синхронизации всех процессов в группе. Это утверждение не относится к барьерной функции.

Вызовы коллективных операций могут использовать те же коммуникаторы, что и парный обмен, при этом MPI гарантирует, что сообщения, созданные коллективными операциями, не будут смешаны с сообщениями, созданными парным обменом.

Ключевым понятием в коллективных функциях является группа участвующих процессов, но в качестве явного аргумента выступает коммуникатор. Коммуникатор понимается как идентификатор группы, связанный с контекстом. Не разрешается использовать в качестве аргумента коллективной функции интер-коммуникатор (коммуникатор, соединяющий две группы).

4.2. КОЛЛЕКТИВНЫЕ ОПЕРАЦИИ

4.2.1. Барьерная синхронизация

MPI_BARRIER (comm)
IN **comm** коммуникатор (дескриптор)

int MPI_Barrier (MPI_Comm comm)

MPI_BARRIER (COMM, IERROR)
INTEGER COMM, IERROR

void MPI::Intracomm::Barrier() const

Функция барьерной синхронизации **MPI_BARRIER** блокирует вызывающий процесс, пока все процессы группы не вызовут ее. В каждом процессе управление возвращается только тогда, когда все процессы в группе вызовут процедуру.

4.2.2. Широковещательный обмен

Функция широковещательной передачи **MPI_BCAST** посылает сообщение из корневого процесса всем процессам группы, включая себя. Она вызывается всеми процессами группы с одинаковыми аргументами для **comm, root**. В момент возврата управления содержимое корневого буфера обмена будет уже скопировано во все процессы.

MPI_BCAST(buffer, count, datatype, root, comm)

INOUT	buffer	адрес начала буфера (альтернатива)
IN	count	количество записей в буфере (целое)
IN	datatype	тип данных в буфере (дескриптор)
IN	root	номер корневого процесса (целое)
IN	comm	коммуникатор (дескриптор)

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

```
MPI_BCAST(BUFFER, COUNT, DATATYPE, ROOT, COMM,  IERROR)
<type> BUFFER(*)
INTEGER COUNT, DATATYPE, ROOT, COMM, IERROR
```

```
void MPI::Intracomm::Bcast(void* buffer, int count,
                           const Datatype& datatype,int root) const
```

В аргументе **datatype** можно задавать производные типы данных. Сигнатура типа данных **count, datatype** любого процесса обязана совпадать с соответствующей сигнатурой в корневом процессе. Это требует, чтобы количество посланных и полученных данных совпадало попарно для корневого и каждого другого процессов. Такое ограничение имеют и все другие коллективные операции, выполняющие перемещение данных.

Пример 4.1. Широковещательная передача 100 целых чисел от процесса 0 каждому процессу в группе.

```
MPI_Comm comm;
int array[100];
int root = 0;
MPI_Bcast( array, 100, MPI_INT, root, comm );
```

4.2.3. Сбор данных

При выполнении операции сборки данных **MPI_GATHER** каждый процесс, включая корневой, посылает содержимое своего буфера в корневой процесс.

Корневой процесс получает сообщения, располагая их в порядке возрастания номеров процессов. Результат будет такой же, как если бы каждый из n процессов группы (включая корневой процесс) выполнил вызов

MPI_Send (sendbuf, sendcount, sendtype, root, …),

и корневой процесс выполнил **n** вызовов

MPI_Recv(recvbuf + i * recvcount * extent(recvtype), recvcount, recvtype, i, …),

где **extent(recvtype)** – размер типа данных, получаемый с помощью **MPI_Type_extent().**

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	**sendbuf**	начальный адрес буфера процесса-отправителя (альтернатива)
IN	**sendcount**	количество элементов в отсылаемом сообщении (целое)
IN	**sendtype**	тип элементов в отсылаемом сообщении (дескриптор)
OUT	**recvbuf**	начальный адрес буфера процесса сборки данных (альтернатива, существенен только для корневого процесса)
IN	**recvcount**	количество элементов в принимаемом сообщении (целое, имеет значение только для корневого процесса)
IN	**recvtype**	тип данных элементов в буфере процесса-получателя (дескриптор)
IN	**root**	номер процесса-получателя (целое)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_GATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,
           RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
        COMM, IERROR

void MPI::Intracomm::Gather(const void* sendbuf, int sendcount, const Datatype&
     sendtype, void* recvbuf, int recvcount, const Datatype& recvtype, int root) const
```

В общем случае как для **sendtype,** так и для **recvtype** разрешены производные типы данных. Сигнатура типа данных **sendcount, sendtype** у процесса **i** должна быть такой же, как сигнатура **recvcount, recvtype** корневого процесса. Это требует, чтобы количество посланных и полученных данных совпадало попарно для корневого и каждого другого процессов. Разрешается различие в картах типов между отправителями и получателями.

В корневом процессе используются все аргументы функции, в то время как у остальных процессов используются только аргументы **sendbuf, sendcount, sendtype, root, comm.** Аргументы **comm** и **root** должны иметь одинаковые значения во всех процессах.

Описанные в функции **MPI_GATHER** количества и типы данных не должны являться причиной того, чтобы любая ячейка корневого процесса записывалась бы более одного раза. Такой вызов является неверным.

Аргумент **recvcount** в главном процессе показывает количество элементов, которые он получил от каждого процесса, а не общее количество полученных элементов.

MPI_GATHERV (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)

IN	**sendbuf**	начальный адрес буфера процесса-отправителя (альтернатива)
IN	**sendcount**	количество элементов в отсылаемом сообщении (целое)
IN	**sendtype**	тип элементов в отсылаемом сообщении (дескриптор)
OUT	**recvbuf**	начальный адрес буфера процесса сборки данных (альтернатива, существенно для корневого процесса)
IN	**recvcounts**	массив целых чисел (по размеру группы), содержащий количество элементов, которые получены от каждого из процессов (используется корневым процессом)
IN	**displs**	массив целых чисел (по размеру группы). Элемент i определяет смещение относительно recvbuf, в котором размещаются данные из процесса i (используется корневым процессом)
IN	**recvtype**	тип данных элементов в буфере процесса-получателя (дескриптор)
IN	**root**	номер процесса-получателя (целое)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_GATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                RECVCOUNTS, DISPLS, RECVTYPE, ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*),
          RECVTYPE, ROOT, COMM, IERROR

void MPI::Intracomm::Gatherv(const void* sendbuf, int sendcount,
    const Datatype& sendtype, void* recvbuf, const int recvcounts[], const int displs[],
    const Datatype& recvtype, int root) const
```

По сравнению с **MPI_GATHER** при использовании функции **MPI_GATHERV** разрешается принимать от каждого процесса переменное число элементов данных, поэтому в функции **MPI_GATHERV** аргумент **recvcount** является массивом. Она также

обеспечивает большую гибкость в размещении данных в корневом процессе. Для этой цели используется новый аргумент **displs**.

Выполнение **MPI_GATHERV** будет давать такой же результат, как если бы каждый процесс, включая корневой, посылал корневому процессу сообщение:

MPI_Send(sendbuf, sendcount, sendtype, root, …),

и корневой процесс выполнял **n** операций приема:

MPI_Recv(recvbuf+displs[i]*extern(recvtype), recvcounts[i], recvtype, i, …).

Сообщения помещаются в принимающий буфер корневого процесса в порядке возрастания их номеров, то есть данные, посланные процессом **j**, помещаются в **j**-ю часть принимающего буфера **recvbuf** на корневом процессе. **j**-я часть **recvbuf** начинается со смещения **displs[j].** Номер принимающего буфера игнорируется во всех некорневых процессах. Сигнатура типа, используемая **sendcount**, **sendtype** в процессе **i** должна быть такой же, как и сигнатура, используемая **recvcounts[i]**, **recvtype** в корневом процессе. Это требует, чтобы количество посланных и полученных данных совпадало попарно для корневого и каждого другого процессов. Разрешается различие в картах типов между отправителями и получателями.

В корневом процессе используются все аргументы функции **MPI_GATHERV**, а на всех других процессах используются только аргументы **sendbuf, sendcount, sendtype, root, comm**. Переменные **comm** и **root** должны иметь одинаковые значения во всех процессах. Описанные в функции **MPI_GATHERV** количества, типы данных и смещения не должны приводить к тому, чтобы любая область корневого процесса записывалась бы более одного раза.

Пример 4.2. Сбор 100 целых чисел с каждого процесса группы в корневой процесс (рис. 4.2).

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather(sendarray,100, MPI_INT, rbuf,100,MPI_INT,root, comm);
```

Пример 4.3 Предыдущий пример модифицирован – только корневой процесс назначает память для буфера приема.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, myrank, *rbuf;
MPI_Comm_rank( comm, myrank);
if ( myrank == root)
{
    MPI_Comm_size( comm, &gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather(sendarray,100,MPI_INT, rbuf,100, MPI_INT, root,comm);
```

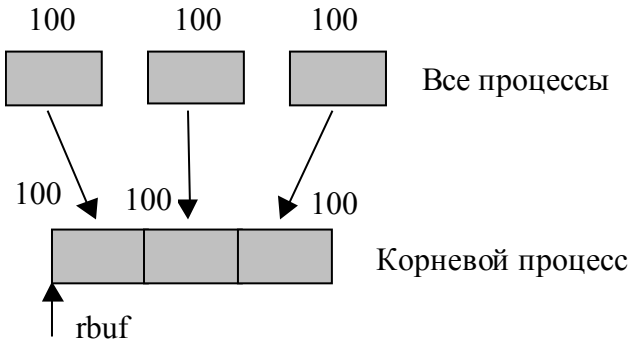


Рис. 4.2. Корневой процесс собирает по 100 целых
чисел из каждого процесса в группе

Пример 4.4. Программа делает то же, что и в предыдущем примере, но использует производные типы данных.

```
MPI_Comm comm;
int gsize,sendarray[100];
int root, *rbuf;
MPI_Datatype rtype;
MPI_Comm_size( comm, &gsize);
MPI_Type_contiguous( 100, MPI_INT, &rtype );
MPI_Type_commit( &rtype );
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 1, rtype, root, comm);
```

Пример 4.5. Каждый процесс посылает 100 чисел **int** корневому процессу, но каждое множество (100 элементов) размещается с некоторым шагом (**stride)** относительно конца размещения предыдущего множества. Для этого нужно использовать **MPI_GATHERV** и аргумент **displs**. Полагаем, что **stride ≥ 100** (рис. 4.3).

```
MPI_Comm comm;
int gsize,sendarray[100], root, *rbuf, stride, *displs,i,*rcounts;
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
MPI_Gatherv( sendarray, 100, MPI_INT, rbuf, rcounts, displs,
                MPI_INT, root, comm);
```

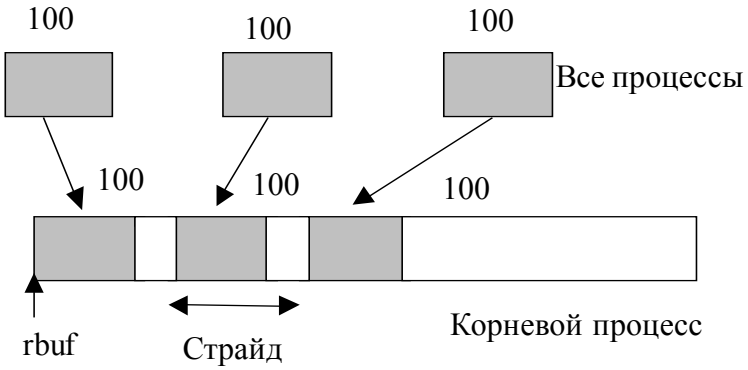


Рис. 4.3. Корневой процесс собирает множество из 100 целых чисел из каждого процесса в группе, и каждое множество размещается с некоторым страйдом относительно предыдущего размещения

Программа неверна, если **stride < 100**.

Пример 4.6. Со стороны процесса-получателя пример такой же, как и 4.5, но посылается 100 чисел типа **int** из 0-го столбца массива 100×150 чисел типа **int** (рис. 4.4).

```
MPI_Comm comm;
int gsize,sendarray[100][150], root, *rbuf, stride, *displs,i,*rcounts;
MPI_Datatype stype;
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;    }
```

```
                    /* Create datatype for 1 column of array  */
MPI_Type_vector( 100, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
MPI_Gatherv( sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT, root, comm);
```

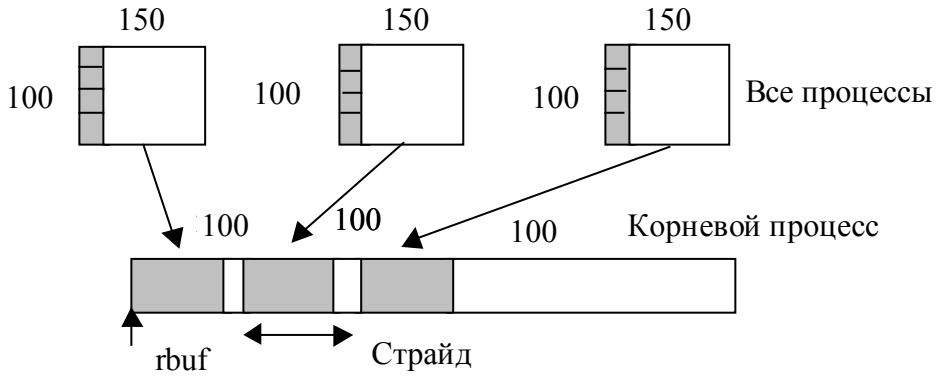


Рис. 4.4. Корневой процесс собирает столбец 0 массива 100×150, и каждое множество размещается с некоторым страйдом по отношению к предыдущему

Пример 4.7. Процесс **i** посылает 100 чисел типа **int** из **i**-го столбца массива 100×150 чисел типа **int** (рис. 4.5)

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr, root, *rbuf, stride, myrank, *displs,i,*rcounts;
MPI_Datatype stype;
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i)
{   displs[i] = i*stride;
    rcounts[i] = 100-i;
}                   /* отличие от предыдущего примера */
                    /* создается тип данных для посылаемого столбца */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
                    /* sptr есть адрес начала  столбца "myrank"  */
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr,1,stype,rbuf,rcounts,displs,MPI_INT,root,comm);
```

Из каждого процесса получено различное количество данных.

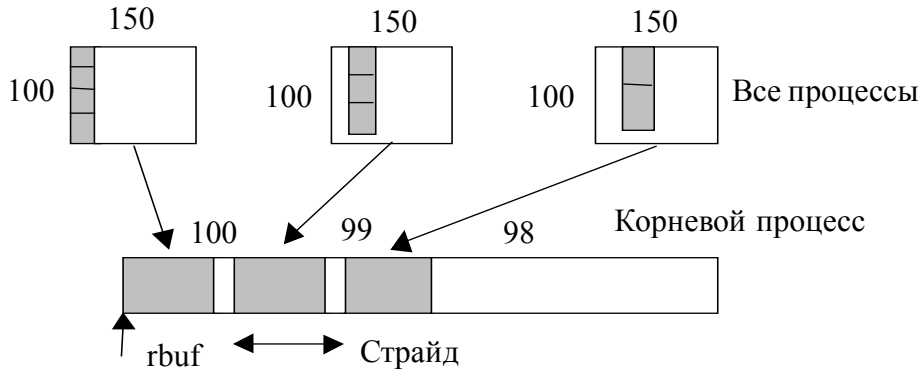


Рис. 4.5. Корневой процесс собирает множество из100-i целых чисел столбца i массива 100×150, и каждое множество размещается с отдельным страйдом

Пример 4.8. Пример такой же, как и 4.7, но содержит отличие на передающей стороне. Создается тип данных со страйдом на передающей стороне для чтения столбца массива.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr,root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype,type[2];
int *displs,i,*rcounts;
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i)
{   displs[i] = i*stride;
    rcounts[i] = 100-i;
 }       /* создается тип  для  числа  int с расширением на полную строку   */
disp[0] = 0;
disp[1] = 150*sizeof(int);
type[0] = MPI_INT;
type[1] = MPI_UB;
blocklen[0] = 1;
 blocklen[1] = 1;
MPI_Type_struct( 2, blocklen, disp, type, &stype );
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv( sptr, 100-myrank, stype, rbuf, rcounts, displs, MPI_INT,
                    root, comm);
```

Пример 4.9. Аналогичен примеру 4.7 на передающей стороне, но на приемной стороне устанавливается страйд между принимаемыми блоками, изменяющийся от блока к блоку (рис. 4.6).

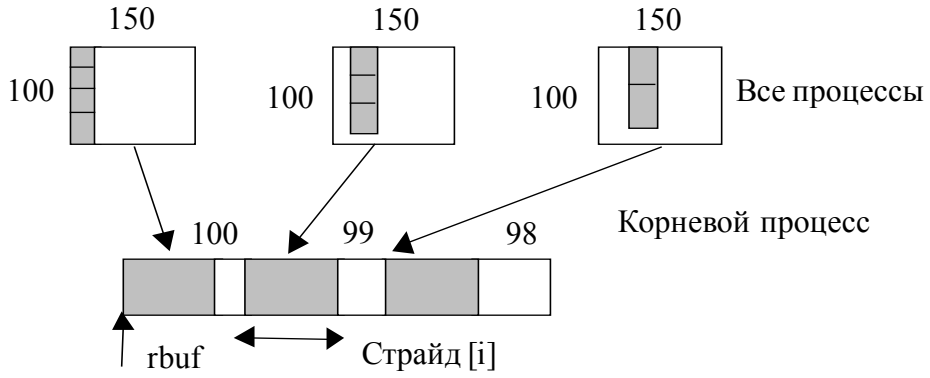


Рис. 4.6. Корневой процесс собирает множество 100-i целых чисел из столбца i массива 100×150, и каждое множество размещает с переменным страйдом [i]

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr;
int root, *rbuf, *stride, myrank, bufsize;
MPI_Datatype stype;
int *displs,i,*rcounts,offset;
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );
stride = (int *)malloc(gsize*sizeof(int));
                    /* сначала устанавливаются вектора displs и rcounts    */
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i)
{   displs[i] = offset;
    offset += stride[i];
    rcounts[i] = 100-i;
}
          /* теперь легко получается требуемый размер буфера для rbuf      */
bufsize = displs[gsize-1]+rcounts[gsize-1];
rbuf = (int *)malloc(bufsize*sizeof(int));
          /* создается тип данных для посылаемого столбца     */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &stype);
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr,1,stype,rbuf, rcounts, displs, MPI_INT,root,comm);
```

Пример 4.10. В этом примере процесс **i** посылает **num** чисел типа **int** из **i**-го столбца массива 100× 150 чисел типа **int**. Усложнение состоит в том, что различные значения **num** неизвестны корневому процессу, поэтому требуется выполнить отдельную операцию **gather**, чтобы найти их. Данные на приемной стороне размещаются непрерывно.

```
MPI_Comm comm;
int gsize,sendarray[100][150],*sptr,root, *rbuf, stride, myrank, disp[2], blocklen[2];
MPI_Datatype stype,types[2];
int *displs,i,*rcounts,num;
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );
                          /*  сначала собираются nums для root      */
rcounts = (int *)malloc(gsize*sizeof(int));
MPI_Gather( &num, 1, MPI_INT, rcounts, 1, MPI_INT, root, comm);
 /* root теперь имеет правильные rcounts, это позволяет установить displs[]
    так, чтобы данные на приемной стороне размещались непрерывно   */
displs = (int *)malloc(gsize*sizeof(int));
displs[0] = 0;
for (i=1; i<gsize; ++i)
     displs[i] = displs[i-1]+rcounts[i-1];
                          /* создается буфер получения      */
rbuf=(int*)malloc(gsize*(displs[gsize-1]+rcounts[gsize-1])*sizeof(int));
   /* создается тип данных для одного int с расширением на полную строку */
disp[0] = 0;
disp[1] = 150*sizeof(int);
type[0] = MPI_INT;
type[1] = MPI_UB;
blocklen[0] = 1;
blocklen[1] = 1;
MPI_Type_struct( 2, blocklen, disp, type, &stype );
MPI_Type_commit( &stype );
sptr = &sendarray[0][myrank];
MPI_Gatherv(sptr,num,stype,rbuf, rcounts, displs, MPI_INT,root, comm);
```

4.2.4. Рассылка

Операция **MPI_SCATTER** обратна операции **MPI_GATHER**. Результат ее выполнения таков, как если бы корневой процесс выполнил **n** операций посылки:

MPI_Send(senbuf + i * extent(sendtype), sendcount, sendtype, i, …),

и каждый процесс выполнит притм:

MPI_Recv(recvbuf, recvcount, recvtype, i, …).

MPI_SCATTER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	**sendbuf**	начальный адрес буфера рассылки (альтернатива, используется только корневым процессом)
IN	**sendcount**	количество элементов, посылаемых каждому процессу (целое, используется только корневым процессом)
IN	**sendtype**	тип данных элементов в буфере посылки (дескриптор, используется только корневым процессом)
OUT	**recvbuf**	адрес буфера процесса-получателя (альтернатива)
IN	**recvcount**	количество элементов в буфере корневого процесса (целое)
IN	**recvtype**	тип данных элементов приемного буфера (дескриптор)
IN	**root**	номер процесса-получателя (целое)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_SCATTER (SENDBUF, SENDCOUNT, SENDTYPE,  RECVBUF,
                  RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)
  <type> SENDBUF(*), RECVBUF(*)
  INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, ROOT,
          COMM, IERROR

void MPI::Intracomm::Scatter(const void* sendbuf, int sendcount, const Datatype&
      sendtype, void* recvbuf, int recvcount, const Datatype& recvtype, int root) const
```

Буфер отправки игнорируется всеми некорневыми процессами. Сигнатура типа, связанная с **sendcount, sendtype**, должна быть одинаковой для корневого процесса и всех других процессов. Это требует, чтобы количество посланных и полученных данных совпадало попарно для корневого и каждого другого процессов. Корневой процесс использует все аргументы функции, а другие процессы используют только аргументы **recvbuf, recvcount, recvtype, root, comm**. Аргументы **root** и **comm** должны быть одинаковыми во всех процессах. Описанные в функции **MPI_SCATTER** количества и типы данных не должны являться причиной того, чтобы любая ячейка корневого процесса записывалfсь бы более одного раза. Такой вызов является неверным.

Операция **MPI_SCATERV** обратна операции **MPI_GATHERV**. Аргумент **sendbuf** игнорируется во всех некорневых процессах. Сигнатура типа, связанная с **sendcount [i], sendtype** в главном процессе, должна быть той же, что и сигнатура, связанная с **recvcount, recvtype** в процессе **i**. Это требует, чтобы количество посланных и полученных данных совпадало попарно для корневого и каждого другого процес-

сов. Разрешается различие в картах типов между отправителями и получателями. Корневой процесс использует все аргументы функции, а другие процессы используют только аргументы **recvbuf, recvcount, recvtype, root, comm**. Аргументы **root** и **comm** должны быть одинаковыми во всех процессах. Описанные в функции **MPI_SCATTER** количества и типы данных не должны приводить к тому, чтобы любая область корневого процесса записывалfсь бы более одного раза.

MPI_SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	**sendbuf**	адрес буфера посылки (альтернатива, используется только корневым процессом)
IN	**sendcounts**	целочисленный массив (размера группы), определяющий число элементов, для отправки каждому процессу
IN	**displs**	целочисленный массив (размера группы). Элемент i указывает смещение (относительно sendbuf, из которого берутся данные для процесса)
IN	**sendtype**	тип элементов посылающего буфера (дескриптор)
OUT	**recvbuf**	адрес принимающего буфера (альтернатива)
IN	**recvcount**	число элементов в посылающем буфере (целое)
IN	**recvtype**	тип данных элементов принимающего буфера (дескриптор)
IN	**root**	номер посылающего процесса (целое)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_SCATTERV(SENDBUF, SENDCOUNTS, DISPLS, SENDTYPE,  RECVBUF,
                    RECVCOUNT,   RECVTYPE,  ROOT, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNTS(*), DISPLS(*), SENDTYPE, RECVCOUNT,
            RECVTYPE, ROOT, COMM, IERROR

void MPI::Intracomm::Scatterv(const void* sendbuf, const int sendcounts[],
          const int displs[], const Datatype& sendtype, void* recvbuf, int recvcount,
          const Datatype& recvtype, int root) const
```

Пример 4.11. Обратен примеру 4.2, **MPI_SCATTER** рассылает 100 чисел из корневого процесса каждому процессу в группе (рис. 4.7).

```
MPI_Comm comm;
int gsize,*sendbuf;
int root, rbuf[100];
MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

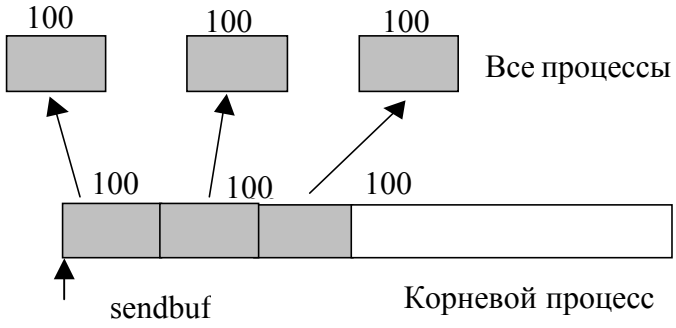


Рис. 4.7. Корневой процесс рассылает по 100 целых чисел каждому процессу в группе

Пример 4.12. Обратен примеру 4.5. Корневой процесс рассылает множества из 100 чисел типа **int** остальным процессам, но множества размещены в посылающем буфере с шагом **stride**, поэтому нужно использовать **MPI_SCATTERV**. Полагаем **stride ≥ 100** (рис. 4.8).

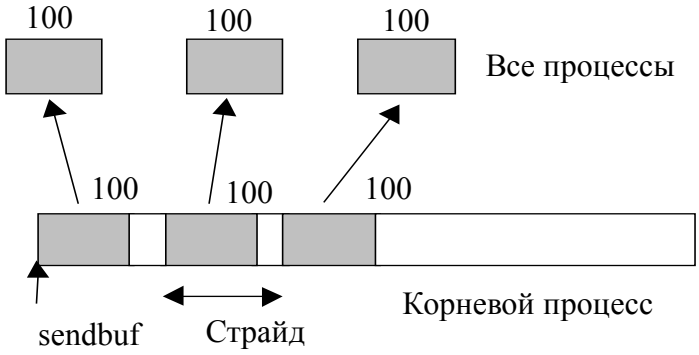


Рис. 4.8. Корневой процесс рассылает по 100 целых чисел, передавая данные со страйдом

```
MPI_Comm comm;
int gsize,*sendbuf, root, rbuf[100], i, *displs, *scounts;
MPI_Comm_size( comm, &gsize);
sendbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i)
{   displs[i] = i*stride;
    scounts[i] = 100;
}
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rbuf, 100, MPI_INT,
                root, comm);
```

Пример 4.13. Обратен примеру 4.9, на стороне корневого процесса используется изменяющийся stride между блоками чисел, на принимаемой стороне производится прием в **i**-й столбец С-массива размера 100×150 (рис. 4.9).

```
MPI_Comm comm;
int gsize,recvarray[100][150],*rptr;
int root, *sendbuf, myrank, bufsize, *stride;
MPI_Datatype rtype;
int i, *displs, *scounts, offset;
MPI_Comm_size( comm, &gsize);
MPI_Comm_rank( comm, &myrank );
stride = (int *)malloc(gsize*sizeof(int));
          /* stride[i] для  i = 0 до gsize-1 – множество  различных значений  */
 displs = (int *)malloc(gsize*sizeof(int));
 scounts = (int *)malloc(gsize*sizeof(int));
 offset = 0;
 for (i=0; i<gsize; ++i)
{    displs[i] = offset;
     offset += stride[i];
     scounts[i] = 100 – i;
}            /* создается тип данных для посылаемого столбца    */
MPI_Type_vector( 100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit( &rtype );
rptr = &recvarray[0][myrank];
MPI_Scatterv( sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype,  root, comm);
```

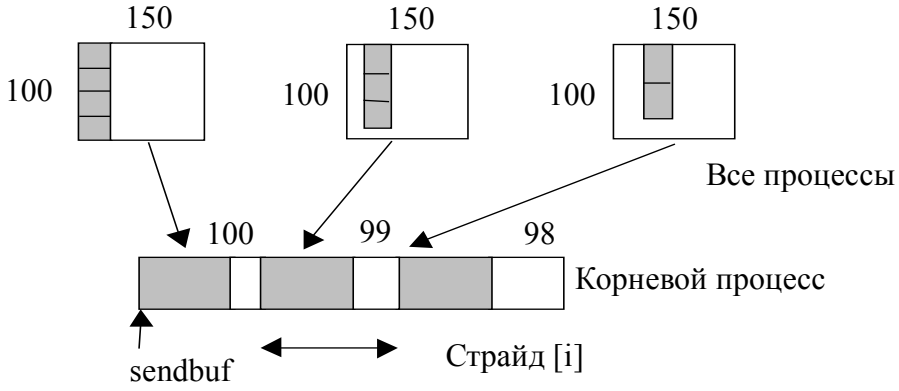


Рис. 4.9. Корневой процесс рассылает блоки из 100-i целых чисел в столбец i массива 100×150 . На стороне отправки блоки размещены со страйдом stride[i]

4.2.5. Сбор для всех процессов

MPI_ALLGATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	**sendbuf**	начальный адрес посылающего буфера (альтернатива)
IN	**sendcount**	количество элементов в буфере (целое)
IN	**sendtype**	тип данных элементов в посылающем буфере (дескриптор)
OUT	**recvbuf**	адрес принимающего буфера (альтернатива)
IN	**recvcount**	количество элементов, полученных от любого процесса (целое)
IN	**recvtype**	тип данных элементов принимающего буфера (дескриптор)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_ALLGATHER(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                RECVCOUNT, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM,
        IERROR

void MPI::Intracomm::Allgather(const void* sendbuf, int sendcount, const Datatype&
        sendtype, void* recvbuf, int recvcount, const Datatype& recvtype) const
```

Функцию **MPI_ALLGATHER** можно представить как функцию **MPI_GATHER**, где результат принимают все процессы, а не только главный. Блок данных, посланный **j**-м процессом? принимается каждым процессом и помещается в **j**-й блок буфера **recvbuf.**

Результат выполнения вызова **MPI_ALLGATHER(...)** такой же, как если бы все процессы выполнили **n** вызовов

MPI_GATHER(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype root, comm),

для **root=0,…,n-1.** Правила использования **MPI_ALLGATHER** соответствуют правилам для **MPI_GATHER**. Сигнатура типа, связанная с **sendcount, sendtype**, должна быть одинаковой во всех процессах.

MPI_ALLGATHERV можно представить как **MPI_GATHERV**, но при ее использовании результат получают все процессы, а не только один корневой. **j**-й блок данных, посланный каждым процессом, принимается каждым процессом и помещается в **j**-й блок буфера **recvbuf.** Эти блоки не обязаны быть одинакового размера.

MPI_ALLGATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)

IN	**sendbuf**	начальный адрес посылающего буфера (альтернатива)
IN	**sendcount**	количество элементов в посылающем буфере (целое)
IN	**sendtype**	тип данных элементов в посылающем буфере (дескриптор)
OUT	**recvbuf**	адрес принимающего буфера (альтернатива)
IN	**recvcounts**	целочисленный массив (размера группы), содержащий количество элементов, полученных от каждого процесса
IN	**displs**	целочисленный массив (размера группы). Элемент i представляет смещение области (относительно recvbuf), где помещаются принимаемые данные от процесса i
IN	**recvtype**	тип данных элементов принимающего буфера (дескриптор)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_ALLGATHERV(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                        RECVCOUNTS, DISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNTS(*), DISPLS(*),
            RECVTYPE, COMM, IERROR

void MPI::Intracomm::Allgatherv(const void* sendbuf, int sendcount,
                    const Datatype& sendtype, void* recvbuf, const int recvcounts[],
                    const int displs[], const Datatype& recvtype) const
```

Сигнатура типа, связанного с **sendcount, sendtype** в процессе **j** должна быть такой же, как сигнатура типа, связанного с **recvcounts[j], recvtype** в любом другом процессе.

Результат вызова **MPI_ALLGATHERV(...)** такой же, как если бы все процессы выполнили **n** вызовов:

MPI_GATHERV(sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm),

для **root = 0, …, n-1.** Правила корректного использования функции **MPI_ALLGATHERV** соответствуют правилам для **MPI_GATHERV**.

Пример 4.14. Сбор 100 чисел типа **int** от каждого процесса в группе для каждого процесса.

```
MPI_Comm comm;
int gsize,sendarray[100], *rbuf;
MPI_Comm_size( comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

После исполнения вызова каждый процесс содержит конкатенацию данных всей группы.

4.2.6. Функция all-to-all Scatter/Gather

MPI_ALLTOALL – расширение функции **MPI_ALLGATHER** для случая, когда каждый процесс посылает различные данные каждому получателю. **j-**й блок, посланный процессом **i,** принимается процессом **j** и помещается в **i**-й блок буфера **recvbuf.**

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)

IN	**sendbuf**	начальный адрес посылающего буфера (альтернатива)
IN	**sendcount**	количество элементов посылаемых в каждый процесс (целое)
IN	**sendtype**	тип данных элементов посылающего буфера (дескриптор)
OUT	**recvbuf**	адрес принимающего буфера (альтернатива)
IN	**recvcount**	количество элементов, принятых от какого-либо процесса (целое)
IN	**recvtype**	тип данных элементов принимающего буфера (дескриптор)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_ALLTOALL(SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF,
                  RECVCOUNT, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER SENDCOUNT, SENDTYPE, RECVCOUNT, RECVTYPE, COMM,
          IERROR

void MPI::Intracomm::Alltoall(const void* sendbuf, int sendcount,
           const Datatype& sendtype, void* recvbuf, int recvcount,
           const Datatype& recvtype) const
```

Результат выполнения функции **MPI_ALLTOALL** такой же, как если бы каждый процесс выполнил посылку данных каждому процессу (включая себя) вызовом:

MPI_Send(sendbuf + i * sendcount * extent(sendtype), sendcount, sendtype, i, ...),

и принял данные от всех остальных процессов путем вызова:

MPI_Recv(recvbuf + i* recvcount* extent(recvtype), recvcount, i,…).

Сигнатура типа, связанная с **sendcount, sendtype** в каждом процессе должна быть такой же, как и в любом другом процессе. Это требу-

ет, чтобы количество посланных данных было равно количеству полученных данных между каждой парой процессов, карты типа могут отличаться. Все аргументы используются всеми процессами. Аргумент **comm** должен иметь одинаковое значение во всех процессах.

MPI_ALLTOALLV обладает большей гибкостью, чем функция **MPI_ALLTOALL,** поскольку размещение данных на передающей стороне определяется аргументом **sdispls**, а на стороне приема – независимым аргументом **rdispls**. **j**-й блок, посланный процессом **i**, принимается процессом **j** и помещается в **i**-й блок **recvbuf.** Эти блоки не обязаны быть одного размера. Сигнатура типа, связанная с **sendcount[j], sendtype** в процессе **i**, должна быть такой же и для процесса **j**. Это подразумевает, что количество посланных данных должно быть равно количеству полученных данных для каждой пары процессов. Карты типа для отправителя и приемника могут отличаться.

MPI_ALLTOALLV(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)

IN	**sendbuf**	начальный адрес посылающего буфера (альтернатива)
IN	**sendcounts**	целочисленный массив (размера группы), определяющий количество посылаемых каждому процессу элементов
IN	**sdispls**	целочисленный массив (размера группы). Элемент j содержит смещение области (относительно sendbuf), из которой берутся данные для процесса j
IN	**sendtype**	тип данных элементов посылающего буфера (дескриптор)
OUT	**recvbuf**	адрес принимающего буфера (альтернатива)
IN	**recvcounts**	целочисленный массив (размера группы), содержит число элементов, которые могут быть приняты от каждого процесса
IN	**rdispls**	целочисленный массив (размера группы). Элемент i определяет смещение области (относительно recvbuf), в которой размещаются данные, получаемые из процесса i
IN	**recvtype**	тип данных элементов принимающего буфера (дескриптор)
IN	**comm**	коммуникатор (дескриптор)

int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls, MPI_Datatype sendtype, void* recvbuf,int *recvcounts,int *rdispls, MPI_Datatype recvtype, MPI_Comm comm)

MPI_ALLTOALLV(SENDBUF, SENDCOUNTS, SDISPLS, SENDTYPE,
RECVBUF, RECVCOUNTS, RDISPLS, RECVTYPE, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)

INTEGER SENDCOUNTS(*), SDISPLS(*), SENDTYPE, RECVCOUNTS(*),
RDISPLS(*), RECVTYPE, COMM, IERROR

```
void MPI::Intracomm::Alltoallv(const void* sendbuf, const int sendcounts[],
        const int sdispls[], const Datatype& sendtype, void* recvbuf,
        const int recvcounts[], const int rdispls[], const Datatype& recvtype) const
```

Результат выполнения **MPI_ALLTOALLV** такой же, как если бы процесс посылал сообщение всем остальным процессам с помощью функции

MPI_Send(sendbuf + displs[i] * extent(sendtype), sendcounts[i], sendtype, i,…)

и принимал сообщение от всех остальных процессов, вызывая

MPI_Recv(recvbuf + displs[i] * extent(recvtype), recvcounts[i], recvtype, i,…).

Все аргументы используются всеми процессами. Значение аргумента **comm** должно быть одинаковым во всех процессах.

4.3. ГЛОБАЛЬНЫЕ ОПЕРАЦИИ РЕДУКЦИИ

Функции в этом разделе предназначены для выполнения операций глобальной редукции (суммирование, нахождение максимума, логическое И, и т.д.) для всех элементов группы. Операция редукции может быть одной из предопределенного списка операций или определяться пользователем. Функции глобальной редукции имеют несколько разновидностей: операции, возвращающие результат в один узел; функции, возвращающие результат во все узлы; операции просмотра.

4.3.1. Функция Reduce

Функция **MPI_REDUCE** объединяет элементы входного буфера каждого процесса в группе, используя операцию **op**, и возвращает объединенное значение в выходной буфер процесса с номером **root.**

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	**sendbuf**	адрес посылающего буфера (альтернатива)
OUT	**recvbuf**	адрес принимающего буфера (альтернатива, используется только корневым процессом)
IN	**count**	количество элементов в посылающем буфере (целое)
IN	**datatype**	тип данных элементов посылающего буфера (дескриптор)
IN	**op**	операция редукции (дескриптор)
IN	**root**	номер главного процесса (целое)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM,
                IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR

void MPI::Intracomm::Reduce(const void* sendbuf, void* recvbuf, int count,
                            const Datatype& datatype, const Op& op, int root) const
```

Буфер ввода определен аргументами **sendbuf, count** и **datatype;** буфер вывода определен параметрами **recvbuf, count** и **datatype;** оба буфера имеют одинаковое число элементов одинакового типа. Функция вызывается всеми членами группы с одинаковыми аргументами **count, datatype, op, root** и **comm.** Таким образом, все процессы имеют входные и выходные буфера одинаковой длины и с элементами одного типа. Каждый процесс может содержать один элемент или последовательность элементов, в последнем случае операция выполняется над всеми элементами в этой последовательности. Например, если выполняется операция **MPI_MAX** и посылающий буфер содержит два элемента – числа с плавающей точкой (**count = 2, datatype = MPI_FLOAT**), то **recvbuf(1) = sendbuf(1)** и **recvbuf(2) = sendbuf(2).**

4.3.2. Предопределенные операции редукции

Имя	**Значение**
MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логическое И
MPI_BAND	поразрядное И
MPI_LOR	логическое ИЛИ
MPI_BOR	поразрядное ИЛИ
MPI_LXOR	логическое исключающее ИЛИ
MPI_BXOR	поразрядное исключающее ИЛИ
MPI_MAXLOC	максимальное значение и местонахождения
MPI_MINLOC	минимальное значение и местонахождения

Группы основных типов данных MPI:

C integer:	MPI_INT,MPI_LONG,MPI_SHORT, MPI_UNSIGNED_SHORT,MPI_UNSIGNED, MPI_UNSIGNED_LONG

Fortran integer:	MPI_INTEGER
Floating point:	MPI_FLOAT,MPI_DOUBLE,MPI_REAL, MPI_DOUBLE_PRECISION,MPI_LONG_DOUBLE
Logical:	MPI_LOGICAL
Complex:	MPI_COMPLEX
Byte:	MPI_BYTE

Типы данных для каждой операции:

Op	**Разрешённые типы**
MPI_MAX, MPI_MIN	C integer, Fortran integer, Floating point
MPI_SUM, MPI_PROD	C integer, Fortran integer, Floating point, Complex
MPI_LAND, MPI_LOR, MPI_LXOR	C integer, Logical
MPI_BAND, MPI_BOR, MPI_BXOR	C integer, Fortran integer, Byte

Пример 4.15. Процедура вычисляет скалярное произведение двух векторов, распределенных в группе процессов, и возвращает результат в нулевой узел.

```
   SUBROUTINE PAR_BLAS1(m, a, b, c, comm)
   REAL a(m), b(m)      ! локальная часть массива
   REAL c               ! результат (на узле ноль)
   REAL sum
   INTEGER m, comm, i, ierr
!  локальная сумма
   sum = 0.0
   DO i = 1, m
        sum = sum + a(i)*b(i)
   END DO
!  глобальная сумма
   CALL MPI_REDUCE(sum, c, 1, MPI_REAL, MPI_SUM, 0, comm, ierr)
   RETURN
```

Пример 4.16 Процедура вычисляет произведение вектора на массив, которые распределены в группе процессов, и возвращает результат в нулевой узел.

```
   SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
   REAL a(m), b(m,n)    ! локальная часть массива
   REAL c(n)            ! результат
   REAL sum(n)
```

```
   INTEGER n, comm, i, j, ierr
!  локальная сумма
   DO j= 1, n
        sum(j) = 0.0
        DO i = 1, m
             sum(j) = sum(j) + a(i)*b(i,j)
        END DO
   END DO
!  глобальная сумма
   CALL MPI_REDUCE(sum, c, n, MPI_REAL, MPI_SUM, 0, comm, ierr)
   RETURN
```

4.3.3. MINLOC и MAXLOC

Оператор **MPI_MINLOC** используется для расчета глобального минимума и соответствующего ему индекса. **MPI_MAXLOC** аналогично считает глобальный максимум и индекс. Обе операции ассоциативны и коммутативны. Если каждый процесс предоставляет значение и свой номер в группе, то операция редукции с **op = MPI_MAXLOC** возвратит значение максимума и номер первого процесса с этим значением. Аналогично, **MPI_MINLOC** может быть использована для получения минимума и его индекса.

Чтобы использовать **MPI_MINLOC** и **MPI_MAXLOC** в операции редукции, нужно обеспечить аргумент **datatype,** который представляет пару (значение и индекс). MPI предоставляет девять таких предопределенных типов данных:

Name	**Description**
Fortran:	
MPI_2REAL	пара переменных типа REAL
MPI_2DOUBLE_PRECISION	пара переменных типа DOUBLE PRECISION
MPI_2INTEGER	пара переменных типа INTEGERs
C:	
MPI_FLOAT_INT	переменные типа float и int
MPI_DOUBLE_INT	переменные типа double и int
MPI_LONG_INT	переменные типа long и int
MPI_2INT	пара переменных типа int
MPI_SHORT_INT	переменные типа short и int
MPI_LONG_DOUBLE_INT	переменные типа long double и int

Тип данных **MPI_2REAL** аналогичен тому, как если бы он был определен следующим образом:

MPI_TYPE_CONTIGOUS(2, MPI_REAL, MPI_2REAL).

Аналогичными выражениями задаются **MPI_2INTEGER, MPI_2DOUBLE_PRECISION** и **MPI_2INT.**

Тип данных **MPI_FLOAT_INT** аналогичен тому, как если бы он был объявлен следующей последовательностью инструкций.

```
type[0] = MPI_FLOAT
type[1] = MPI_INT
disp[0] = 0
disp[1] = sizeof(float)
block[0] = 1
block[1] = 1
MPI_TYPE_STRUCT(2, block, disp, type, MPI_FLOAT_INT)
```

Подобные выражения относятся и к функциям **MPI_LONG_INT** и **MPI_DOUBLE_INT.**

Пример 4.17. Каждый процесс имеет массив 30 чисел типа double. Для каждой из 30 областей надо вычислить значение и номер процесса, содержащего наибольшее значение.

```
/* каждый процесс имеет массив из чисел  двойной точности:   ain[30]*/
double ain[30], aout[30];
int  ind[30];
struct {
    double val;
    int   rank;
} in[30], out[30];
int i, myrank, root;

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
for (i=0; i<30; ++i) {
    in[i].val = ain[i];
    in[i].rank = myrank;
}
MPI_Reduce(in,out,30,MPI_DOUBLE_INT,MPI_MAXLOC,root,comm );
                    /* в этой точке результат помещается на корневой процесс    */
if (myrank == root) {              /* читаются выходные номера         */
    for (i=0; i<30; ++i) {
      aout[i] = out[i].val;
      ind[i] = out[i].rank;          /* номер обратно преобразуется в целое */
 }
```

Пример 4.18. Каждый процесс имеет не пустой массив чисел. Требуется найти минимальное глобальное число, номер процесса, хранящего его, его индекс в этом процессе.

```
#define  LEN   1000
float val[LEN];          /* локальный массив значений */
int count;               /* локальное количество значений */
int myrank, minrank, minindex;
float minval;
 struct {
    float value;
    int   index;
} in, out;
                         /* локальный minloc */
in.value = val[0];
in.index = 0;
for (i=1; i < count; i++)
     if (in.value > val[i])
     {        in.value = val[i];
              in.index = i;
     }
                         /* глобальный minloc */
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
in.index = myrank*LEN + in.index;
MPI_Reduce(in, out, 1, MPI_FLOAT_INT, MPI_MINLOC, root, comm );
        /* в этой точке результат помещается на корневой процесс      */
if (myrank == root)
{  minval = out.value;
   minrank = out.index / LEN;
   minindex = out.index % LEN;
}
```

4.3.4. Функция All-Reduce

MPI имеет варианты каждой из операций редукции, где результат возвращается всем процессам группы. MPI требует, чтобы все процессы, участвующие в этих операциях, получили идентичные результаты.

MPI_ALLREDUCE(sendbuf, recvbuf, count, datatype, op, comm)

IN	**sendbuf**	начальный адрес посылающего буфера (альтернатива)
OUT	**recvbuf**	начальный адрес принимающего буфера (альтернатива)
IN	**count**	количество элементов в посылающем буфере (целое)
IN	**datatype**	тип данных элементов посылающего буфера ()
IN	**op**	операция (дескриптор)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_ALLREDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM,
              IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR

void MPI::Intracomm::Allreduce(const void* sendbuf, void* recvbuf, int count,
                               const Datatype& datatype, const Op& op) const
```

Функция **MPI_ALLREDUCE** отличается от **MPI_REDUCE** тем, что результат появляется в принимающем буфере всех членов группы.

Пример 4.19. Процедура вычисляет произведение вектора и массива, которые распределены по всем процессам группы, и возвращает ответ всем узлам.

```
   SUBROUTINE PAR_BLAS2(m, n, a, b, c, comm)
!  локальная часть массива
   REAL a(m), b(m,n)
!  результат
   REAL c(n)
   REAL sum(n)
   INTEGER n, comm, i, j, ierr
!  локальная сумма
   DO j= 1, n
        sum(j) = 0.0
        DO i = 1, m
             sum(j) = sum(j) + a(i)*b(i,j)
        END DO
   END DO
!  глобальная сумма
   CALL MPI_ALLREDUCE(sum,c,n,MPI_REAL,MPI_SUM,comm,ierr)
!  возвращение результата всем узлам
   RETURN
```

4.3.5. Функция Reduce-Scatter

MPI имеет варианты каждой из операций редукции, когда результат рассылается всем процессам в группе в конце операции.

Функция **MPI_REDUCE_SCATTER** сначала производит поэлементную редукцию вектора из **count** = $\sum_i$ **recvcount[i]** элементов в посылающем буфере, определенном **sendbuf, count** и **datatype**. Далее полученный вектор результатов разделяется на **n** непересекающихся

сегментов, где **n** – число членов в группе. Сегмент **i** содержит **recvcount[i]** элементов. **i-**й сегмент посылается **i-**му процессу и хранится в приемном буфере, определяемом **recvbuf, recvcounts[i]** и **datatype**.

MPI_REDUCE_SCATTER(sendbuf, recvbuf, recvcounts, datatype, op, comm)

IN	**sendbuf**	начальный адрес посылающего буфера (альтернатива)
OUT	**recvbuf**	начальный адрес принимающего буфера (альтернатива)
IN	**recvcounts**	целочисленный массив, определяющий количество элементов результата, распределенных каждому процессу. Массив должен быть идентичен во всех вызывающих процессах
IN	**datatype**	тип данных элементов буфера ввода (дескриптор)
IN	**op**	операция (дескриптор)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_REDUCE_SCATTER(SENDBUF, RECVBUF, RECVCOUNTS,
                   DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER RECVCOUNTS(*), DATATYPE, OP, COMM, IERROR

void Intracomm::Reduce_scatter(const void* sendbuf, void* recvbuf, int recvcounts[],
                               const Datatype& datatype, const Op& op) const
```

4.3.6. Функция Scan

MPI_SCAN(sendbuf, recvbuf, count, datatype, op, comm)

IN	**sendbuf**	начальный адрес посылающего буфера (альтернатива)
OUT	**recvbuf**	начальный адрес принимающего буфера (альтернатива)
IN	**count**	количество элементов в принимающем буфере (целое)
IN	**datatype**	тип данных элементов в принимающем буфере (дескриптор)
IN	**op**	операция (дескриптор)
IN	**comm**	коммуникатор (дескриптор)

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

MPI_SCAN(SENDBUF, RECVBUF, COUNT, DATATYPE, OP, COMM, IERROR)
<type> SENDBUF(*), RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, COMM, IERROR

void Intracomm::Scan(const void* sendbuf, void* recvbuf, int count,
                     const Datatype& datatype, const Op& op) const
```

Функция **MPI_SCAN** используется, чтобы выполнить префиксную редукцию данных, распределенных в группе. Операция возвращает в

приемный буфер процесса **i** редукцию значений в посылающих буферах процессов с номерами **0, ..., i** (включительно). Тип поддерживаемых операций, их семантика и ограничения на буфера посылки и приема – такие же, как и для **MPI_REDUCE**.

4.4. КОРРЕКТНОСТЬ

Пример 4.20. Следующий отрезок программы неверен.

```
switch(rank) {
   case 0:
      MPI_Bcast(buf1, count, type, 0, comm);
      MPI_Bcast(buf2, count, type, 1, comm);
      break;
   case 1:
      MPI_Bcast(buf2, count, type, 1, comm);
      MPI_Bcast(buf1, count, type, 0, comm);
      break;
}
```

Предполагается, что группа **comm** есть {**0,1**}. Два процесса выполняют две операции широковещания в обратном порядке. Если операция синхронизирующая, произойдёт взаимоблокирование. Коллективные операции должны быть выполнены в одинаковом порядке во всех элементах группы.

Пример 4.21. Следующий отрезок программы неверен.

```
switch(rank) {
   case 0:
      MPI_Bcast(buf1, count, type, 0, comm0);
      MPI_Bcast(buf2, count, type, 2, comm2);
      break;
   case 1:
      MPI_Bcast(buf1, count, type, 1, comm1);
      MPI_Bcast(buf2, count, type, 0, comm0);
      break;
   case 2:
      MPI_Bcast(buf1, count, type, 2, comm2);
      MPI_Bcast(buf2, count, type, 1, comm1);
      break;
}
```

Предположим, что группа из **comm0** есть **{0,1},** группа из **comm1 – {1, 2}** и группа из **comm2 – {2,0}.** Если операция широковещания син-

хронизирующая, то имеется циклическая зависимость: широковещание в **comm2** завершается только после широковещания в **comm0**; широковещание в **comm0** завершается только после широковещания в **comm1**; и широковещание в **comm1** завершится только после широковещания в **comm2**. Таким образом, будет иметь место дедлок. Коллективные операции должны быть выполнены в таком порядке, чтобы не было циклических зависимостей.

Пример 4.22. Следующий отрезок программы неверен.

```
switch(rank) {
   case 0:
      MPI_Bcast(buf1, count, type, 0, comm);
      MPI_Send(buf2, count, type, 1, tag, comm);
      break;
   case 1:
      MPI_Recv(buf2, count, type, 0, tag, comm, status);
      MPI_Bcast(buf1, count, type, 0, comm);
      break;
}
```

Процесс с номером 0 выполняет широковещательную рассылку (**bcast**), сопровождаемую блокирующей посылкой данных (**send**). Процесс с номером 1 выполняет блокирующий приём (**receive**), который соответствует посылке с последующей широковещательной передачей, соответствующей широковещательной операции процесса с номером 0. Такая программа может вызвать дедлок. Операция широковещания на процессе с номером 0 может вызвать блокирование, пока процесс с номером 1 не выполнит соответствующую широковещательную операцию, так что посылка не будет выполняться. Процесс 0 будет неопределенно долго блокироваться на приеме, в этом случае никогда не выполнится операция широковещания. Относительный порядок выполнения коллективных операций и операций парного обмена должен быть такой, чтобы даже в случае синхронизации не было дедлока.

Пример 4.23. Правильная, но недетерминированная программа.

```
switch(rank) {
   case 0:
      MPI_Bcast(buf1, count, type, 0, comm);
      MPI_Send(buf2, count, type, 1, tag, comm);    break;
   case 1:
```

```
      MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
      MPI_Bcast(buf1, count, type, 0, comm);
      MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, status);
       break;
    case 2:
       MPI_Send(buf2, count, type, 1, tag, comm);
       MPI_Bcast(buf1, count, type, 0, comm);     break;
}
```

Все три процесса участвуют в широковещании (broadcast). Процесс 0 посылает сообщение процессу 1 после операции широковещания, а процесс 2 посылает сообщение процессу 1 перед операцией широковещания. Процесс 1 принимает данные перед и после операции широковещания, с произвольным номером процесса-отправителя.

У этой программы существует два возможных варианта выполнения, с разными соответствиями между отправлением и получением. Заметим, что второй вариант выполнения имеет специфику, заключающуюся в том, что посылка выполненная после операции широковещания получена в другом узле перед операцией широковещания. Этот пример показывает, что нельзя полагаться на специфические эффекты синхронизации. Программа, которая работает правильно только тогда, когда выполнение происходит по первой схеме (только когда операция широковещания выполняет синхронизацию), неверна.

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ К ГЛАВЕ 4

Контрольные вопросы к 4.1

1. Дайте определение локальной и коллективной операций.
2. Участвуют ли в коллективных операциях все процессы приложения?
3. Должна ли быть вызвана функция, соответствующая коллективной операции, каждым процессом, быть может, со своим набором параметров?
4. В коллективных операциях участвуют все процессы коммуникатора?
5. Что такое корневой процесс?
6. Означает ли возврат процесса из функции, реализующей коллективную операцию, что операция завершена?
7. В чем преимущество использования коллективных операций перед парными?
8. Приведите пример некорректного использования коллективных операций, приводящий к дедлоку.
9. Можно ли использовать в качестве аргумента коллективной функции интеркоммуникатор?

Контрольные вопросы к 4.2

1. Какие коллективные операции используются для синхронизации процессов?

2. Означает ли вызов функции MPI_Barrier, что вызывающий процесс блокируется, пока все процессы приложения не вызовут ее?
3. Можно ли в качестве номера корневого процесса в функции широковещательной передачи MPI_Bcast указывать номер любого процесса коммуникатора?
4. Означает ли возврат из функции MPI_Bcast, что содержимое буфера обмена скопировано во все процессы?
5. Как изменить код программы в примере 4.1. для осуществления широковещательной передачи 200 вещественных чисел от процесса 2 каждому процессу в группе works?
6. В каком порядке располагает сообщения корневой процесс при выполнении операции сборки данных MPI_Gather?
7. Какие из аргументов функции MPI_Gather не используются в процессах, не являющихся корневыми?
8. Сколько сообщений от каждого процесса может принимать корневой процесс при выполнении операции MPI_Gatherv?
9. В чем состоит особенность размещения данных в корневом процессе при выполнении операции MPI_Gatherv?
10. В чем различие использования функции MPI_Gather в примерах 4.2, 4.3 и 4.4?
11. Почему неверна программа в примере 4.5, если stride < 100?
12. Как в примере 4.6 осуществить посылку от каждого процесса 100 элементов n-го столбца, где n<100?
13. Поясните, почему из каждого процесса получено различное количество данных в примере 4.7?
14. Сравните реализации программы примеров 4.7, 4.8 и 4.9. Какой из примеров предпочтительней? Почему?
15. Можно ли выполнить задание примера 4.10, не используя функцию MPI_Gather? Предложите варианты решения.
16. Может ли начальный адрес буфера рассылки совпадать с адресом буфера процесса-получателя при вызове функции MPI_Scatter?
17. Разрешается ли изменять количества данных, посылаемых каждому процессу в функции MPI_Scatter? А в функции MPI_Scatterv?
18. В чем различие между двумя вызовами

    MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);

    и MPI_Bcast (sendbuf, 100, MPI_INT, root, comm)?
19. Как изменить код программы в примере 4.11 для осуществления передачи 200 вещественных чисел от процесса 2 каждому процессу в группе works?
20. Что произойдет при реализации программы из примера 4.12, если stride<100?
21. Как изменится код программы из примера 4.13, если на принимаемой стороне производится прием в 0-й столбец С-массива?
22. Будет ли различие в результатах для корневого процесса при использовании MPI_Gather и MPI_Allgather с одинаковыми параметрами?
22. Сколько процессов получают данные при использовании MPI_Allgather? А при использовании MPI_Allgatherv?
23. Каким набором посылок и приемов (MPI_Send и MPI_Recv) можно заменить вызов функции MPI_Alltoall? А – MPI_Alltoallv?

Контрольные вопросы к 4.3

1. В каком порядке производится операция редукции над данными из разных процессов при вызове функции MPI_Reduce?
2. Гарантирована ли однозначность результата в стандарте MPI?
3. Какие типы данных можно использовать для операций MPI_MINLOC и MPI_MAXLOC в операциях редукции?
4. Сколько процессов получают данные при использовании MPI_Allreduce?
5. Предложите вариант эквивалентной по результату выполнения замены операции MPI_Allreduce двумя операциями: MPI_Reduce и MPI_Bcast.
6. Сколько процессов получают данные при использовании MPI_Reduce? А при – MPI_Reduce_scatter?
7. Предложите вариант эквивалентной замены операции MPI_Scan набором посылок и приемов (MPI_Send и MPI_Recv).

Задания для самостоятельной работы

4.1. Напишите программу, которая читает целое значение с терминала и посылает это значение всем MPI–процессам. Каждый процесс должен печатать свой номер и полученное значение. Значения следует читать, пока не появится на входе отрицательное целое число.

4.2. Напишите программу, которая реализует параллельный алгоритм для задачи нахождения скалярного произведение двух векторов.

4.3. Напишите программу, которая реализует параллельный алгоритм для произведения вектора на матрицу. Будем считать, что матрица и вектор генерируются в нулевом процессе, затем рассылаются всем процессам. Каждый процесс считает n/size элементов результирующего вектора, где n – количество строк матрицы, size – число процессов приложения.

4.4. Напишите программу пересылки по num чисел типа int из i-го столбца массива 100*150 от каждого процесса в корневой.

4.5. Напишите программу, которая читает целое значение и значение двойной точности с терминала и посылает одной командой MPI_Bcast эти значения всем MPI–процессам. Каждый процесс должен печатать свой номер и полученные значения. Значения следует читать, пока не появится на входе отрицательное целое число. Используйте возможности MPI для создания новых типов данных: Type_struct.

4.6. Выполните задание 4.5, используя MPI_Pack и MPI_Unpack для обеспечения коммуникации различных типов данных.

4.7. Напишите программу для измерения времени передачи вещественных данных двойной точности от одного процесса другому. Выполните задание при условии, что каждый процесс передает и принимает от процесса, находящегося на расстоянии size/2, где имеется size процессов в MPI_COMM_WORLD. Лучшее решение будет получено при использовании MPI_SendRecv, MPI_Barrier, чтобы гарантировать, что различные пары стартуют почти одновременно, однако возможны другие решения. Для усреднения накладных расходов следует: повторить достаточное количество операций пересылок для получения времени в пределах

долей секунды (образцовое решение делает 100000/size итераций для целых size), повторить тестирование несколько раз (например, 10) и усреднить результаты.

4.8. Напишите программу для измерения времени, необходимого для выполнения MPI_Allreduce на MPI_COMM_WORLD. Как изменяются характеристики для MPI_Allreduce при изменении размера MPI_COMM_WORLD?

4.9. Напишите программу для измерения времени, необходимого для выполнения MPI_Barrier на MPI_COMM_WORLD. Как изменяются характеристики для MPI_Barrier при изменении размера MPI_COMM_WORLD?

4.10. Напишите программу для определения объема буферизации, необходимого для выполнения MPI_Send. Это означает, что нужно написать программу, которая определяет, насколько большого объема сообщение может быть послано без включения соответствующего приема в процессе назначения.

4.11. Пусть A(n,m) – матрица, созданная в процессе 0. Например, может быть прочитана из памяти или уже была вычислена. Пусть имеем 4 процесса и процесс 0 посылает части этой матрицы другим процессам. Процессор 1 получает A(i,j) для i=n/2+1,...,n, и j=1,...,m/2. Процессор 2 получает A(i,j) для i=1,...,n/2 и j=m/2+1,...,m и процессор 3 получает A(i,j) для i=n/2+1,...,n and j=m/2,...,m . Это двумерная декомпозиция A на четыре процесса. Напишите программу рассылки частей матрицы по процессам, используйте MPI_Scatterv, чтобы послать данные из процессора 0 всем другим процессам (включая процесс 0).

4.12. Пусть имеем двумерный массив X размера maxn*maxn. Эта структура есть двумерная регулярная сетка точек, которую разделим на слои, каждый из которых будет обрабатывать отдельный процесс. Пусть вычисления, которые необходимо выполнить, нуждаются в смежных значениях. Это означает, что для вычисления нового значения x[i][j] необходимо знать: x[i][j+1], x[i][j-1], x[i+1][j], x[i-1][j]. Последние два могут быть проблемой, если они находятся в смежных процессах. Чтобы разрешить это, определим теневые точки в каждом процессе, которые будут содержать эти смежные точки. Элементы среды, которые используются, чтобы сохранять данные из других процессов, называются «теневыми». Напишите программу для правильного заполнения теневых точек в каждом процессе. Для простоты предположите:

1) maxn = 12 и количество процессов =4;
2) каждый процесс заполняет свою часть массива собственным номером в коммуникаторе, а теневые точки значениями -1. После обмена с соседними процессами необходимо проверить правильность заполнения теневых точек;
3) область непериодическая, то есть верхний процесс (номер = size-1) только посылает и принимает данные от нижележащего процесса (номер = size–2), а самый нижний процесс (номер = 0) передает и принимает данные только от процесса выше него (номер = 1).

4.13. Выполните задание 4.12, используя неблокируемые парные обмены вместо блокируемых. Замените MPI_Send и MPI_Recv процедурами MPI_ISend и MPI_IRecv и используйте MPI_Wait или MPI_Waitall для теста на окончание неблокируемой операции.

4.14. Выполните задание 4.12, заменив в решении вызовы MPI_Send и MPI_Recv двумя вызовами MPI_SendRecv. Первый вызов должен сдвинуть данные вверх, то есть послать данные процессору, который расположен выше и при-

нять данные от процессора, расположенного ниже. Второе обращение к MPI_SendRecv должно быть обратным первому: послать данные нижележащему процессору и получить данные от процессора, расположенного выше.

4.15. Выполните задание 4.12, используя MPI_SendRecv для обмена данными с соседними процессами. Это означает, что обмениваются процессы 0 и 1, 2 и 3 и так далее.

4.16. В этом задании необходимо решить уравнение Лапласа на сетке двух измерений методом итераций Якоби. Любой текст численного анализа показывает, что итерации будут вычислять аппроксимацию для решения уравнения Лапласа, причем новое значение xnew замещается средним значением точек вокруг него для внутренних точек, а граничные значения остаются фиксированными.

```
while (not converged) {
   for (i,j)   xnew[i][j] = (x[i+1][j] + x[i-1][j] + x[i][j+1] + x[i][j-1])/4;
   for (i,j)   x[i][j] = xnew[i][j];  }
```

На практике это означает, что если сетка имеет размер n*n, тогда значения x[0][j], x[n-1][j], x[i][0], x[i][n-1] не изменяются. Для проверки сходимости выберем следующий критерий:

```
diffnorm = 0;
for (i,j)
   diffnorm += (xnew[i][j] – x[i][j]) * (xnew[i][j] – x[i][j]);
   diffnorm = sqrt(diffnorm).
```

Когда diffnorm станет меньше 1.0e-2, будем считать, что результат достигнут.

Напишите программу исполнения этой аппроксимации. Для простоты рассмотрите сетку 12*12 на 4 процессах, заполненную значениями предыдущего примера: это значения равные -1 на границах, значения, равные номеру процесса во внутренних точках сетки.

4.17. Выполните задание 4.16, модифицируя его так, чтобы вычисленное решение собиралось в процессоре 0, который затем выводит этот результат.

4.18. Во многих случаях невозможно разделить параллельную структуру данных так, чтобы каждый процесс имел одинаковое количество данных. Это даже может быть нежелательным, когда сумма работы, которую необходимо сделать, является переменной. Выполните задание 4.17, изменив код так, чтобы каждый процесс мог иметь различное число строк разделяемой сетки.

4.19.Выполните задание 4.17 при условии, что распределяемый массив заполняется значениями из файла.

4.20. Напишите программу, в которой каждый процесс генерирует часть матрицы псевдослучайных чисел и передает все части в матрицу главному процессу.

4.21. Напишите программу нахождения максимального значения и его индекс из массива чисел равномерно распределенного по процессам.

Глава 5. ГРУППЫ И КОММУНИКАТОРЫ

5.1. ВВЕДЕНИЕ

В этом разделе рассматриваются средства MPI для поддержки разработки параллельных библиотек. Для создания устойчивых параллельных библиотек интерфейс MPI должен обеспечить:

- Возможность создавать безопасное коммуникационное пространство, которое гарантировало бы, что библиотеки могут выполнять обмен, когда им нужно, без конфликтов с обменами, внешними по отношению к данной библиотеке.
- Возможность определять границы области действия коллективных операций, которые позволяли бы библиотекам избегать ненужного запуска невовлеченных процессов.
- Возможность абстрактного обозначения процессов, чтобы библиотеки могли описывать свои обмены в терминах, удобных для их собственных структур данных и алгоритмов.
- Возможность создавать новые пользовательские средства, такие как дополнительные операции для коллективного обмена. Этот механизм должен обеспечить пользователя или создателя библиотеки средствами эффективного расширения состава операций для передачи сообщений.

Для поддержки библиотек MPI обеспечивает группы процессов (groups), виртуальные топологии (virtual topologies), коммуникаторы (communicators).

Коммуникаторы создают область для всех операций обмена в MPI. Коммуникаторы разделяются на два вида: интра-коммуникаторы (внутригрупповые коммуникаторы), предназначенные для операций в пределах отдельной группы процессов, и интер-коммуникаторы (межгрупповые коммуникаторы), предназначенные для обменов между двумя группами процессов.

Группы. Группы определяют упорядоченную выборку процессов по именам. Таким образом, группы определяют область для парных и коллективных обменов. В MPI группы могут управляться отдельно от коммуникаторов, но в операциях обмена могут использоваться только коммуникаторы.

Виртуальная топология определяет специальное отображение номеров процессов в группе на определенную топологию, и наоборот.

Чтобы обеспечить эту возможность, в главе 6 для коммуникаторов определены специальные конструкторы.

5.2. БАЗОВЫЕ КОНЦЕПЦИИ

Группа есть упорядоченный набор идентификаторов процессов; **процессы** есть зависящие от реализации объекты. Каждый процесс в группе связан с целочисленным номером. Нумерация является непрерывной и начинается с нуля. Группы представлены скрытыми объектами группы и, следовательно, не могут быть непосредственно переданы от одного процесса к другому. Группа используется в пределах коммуникатора для описания участников коммуникационной области и ранжирования этих участников путем предоставления им уникальных имен.

Имеется предопределенная группа: **MPI_GROUP_EMPTY**, которая является группой без членов. Предопределенная константа **MPI_GROUP_NULL** является значением, используемым для ошибочных дескрипторов группы.

Контекст есть свойство коммуникаторов, которое позволяет разделять пространство обмена. Сообщение, посланное в одном контексте, не может быть получено в другом контексте. Более того, где это разрешено, коллективные операции независимы от ждущих операций парного обмена. Контексты не являются явными объектами MPI; они проявляются только как часть реализации коммуникаторов.

Интра-коммуникаторы объединяют концепции группы и контекста для поддержки реализационно-зависимых оптимизаций и прикладных топологий (глава 6). Операции обмена в MPI используют коммуникаторы для определения области, в которой должны выполняться парная или коллективная операции.

Каждый коммуникатор содержит группу участников; эта группа всегда участвует в локальном процессе. Источник и адресат сообщения определяются номером процесса в пределах этой группы.

Для коллективной связи интра-коммуникатор определяет набор процессов, которые участвуют в коллективной операции (и их порядок, когда это существенно). Таким образом, коммуникатор ограничивает "пространственную" область коммуникации и обеспечивает машинно-независимую адресацию процессов их номерами.

Начальный для всех возможных процессов интра-коммуникатор **MPI_COMM_WORLD** создается сразу при обращении к функции **MPI_INIT**. Кроме того, существует коммуникатор, который содержит

только себя как процесс – **MPI_COMM_SELF**. Предопределенная константа **MPI_COMM_NULL** есть значение, используемое для неверных дескрипторов коммуникатора.

В реализации MPI со статической моделью обработки коммуникатор **MPI_COMM_WORLD** имеет одинаковое значение во всех процессах. В реализации MPI, где процессы могут порождаться динамически, возможен случай, когда процесс начинает вычисления, не имея доступа ко всем другим процессам. В таких ситуациях, коммуникатор **MPI_COMM_WORLD** является коммуникатором, включающим все процессы, с которыми подключающийся процесс может немедленно связаться. Поэтому **MPI_COMM_WORLD** может одновременно иметь различные значения в различных процессах.

Все реализации MPI должны обеспечить наличие коммуникатора **MPI_COMM_WORLD**. Он не может быть удален в течение времени существования процесса. Группа, соответствующая этому коммуникатору, не появляется как предопределенная константа, но к ней можно обращаться, используя **MPI_COMM_GROUP**. MPI не определяет соответствия между номером процесса в **MPI_COMM_WORLD** и его абсолютным адресом.

5.3. УПРАВЛЕНИЕ ГРУППОЙ

Операции управления являются локальными, и их выполнение не требует межпроцессного обмена.

5.3.1. Средства доступа в группу

MPI_GROUP_SIZE (group, size)

IN	**group**	группа (дескриптор)
OUT	**size**	количество процессов в группе (целое)

```
int MPI_Group_size(MPI_Group group, int *size)

MPI_GROUP_SIZE(GROUP, SIZE, IERROR)
INTEGER GROUP, SIZE, IERROR

int MPI::Group::Get_size() const
```

Функция **MPI_GROUP_SIZE** позволяет определить размер группы.

MPI_GROUP_RANK (group, rank)

IN	**group**	группа (дескриптор)
OUT	**rank**	номер процесса в группе или MPI_UNDEFINED, если процесс не является членом группы (целое)

```
int MPI_Group_rank(MPI_Group group, int *rank)

MPI_GROUP_RANK(GROUP, RANK, IERROR)
INTEGER GROUP, RANK, IERROR

int MPI::Group::Get_rank() const
```

Функция **MPI_GROUP_RANK** служит для определения номера процесса в группе.

MPI_GROUP_TRANSLATE_RANKS(group1, n, ranks1, group2, ranks2)

IN	**group1**	Группа1 (дескриптор)
IN	**n**	число номеров в массивах ranks1 и ranks2 (целое)
IN	**ranks1**	массив из нуля или более правильных номеров в группе1
IN	**group2**	группа2 (дескриптор)
OUT	**ranks2**	массив соответствующих номеров в группе2, MPI_UNDEFINED, если соответствие отсутствует.

```
int MPI_Group_translate_ranks (MPI_Group group1, int n, int *ranks1,
                               MPI_Group group2, int *ranks2)

MPI_GROUP_TRANSLATE_RANKS(GROUP1, N, RANKS1, GROUP2, RANKS2,
                          IERROR)
INTEGER GROUP1, N, RANKS1(*), GROUP2, RANKS2(*), IERROR

static void MPI::Group::Translate_ranks(const MPI::Group& group1, int n,
                const int ranks[], const MPI::Group& group2, int ranks2[])
```

Эта функция важна для определения относительной нумерации одинаковых процессов в двух различных группах. Например, если известны номера некоторых процессов в **MPI_COMM_WORLD**, то можно узнать их номера в подмножестве этой группы.

MPI_GROUP_COMPARE(group1, group2, result)

IN	**group1**	первая группа (дескриптор)
IN	**group2**	вторая группа (дескриптор)
OUT	**result**	результат (целое)

```
int MPI_Group_compare(MPI_Group group1,MPI_Group group2, int *result)

MPI_GROUP_COMPARE(GROUP1, GROUP2, RESULT, IERROR)
INTEGER GROUP1, GROUP2, RESULT, IERROR

static int MPI::Group::Compare(const MPI::Group& group1,
                               const MPI::Group& group2)
```

Если члены группы и их порядок в обеих группах совершенно одинаковы, вырабатывается результат **MPI_IDENT**. Это происходит,

например, если **group1** и **group2** имеют тот же самый дескриптор. Если члены группы одинаковы, но порядок различен, то вырабатывается результат **MPI_SIMILAR**. В остальных случаях получается значение **MPI_UNEQUAL**.

5.3.2. Конструкторы групп

Конструкторы групп применяются для подмножества и расширенного множества существующих групп. Эти конструкторы создают новые группы на основе существующих групп. Данные операции являются локальными, и различные группы могут быть определены на различных процессах; процесс может также определять группу, которая не включает себя.

MPI не имеет механизма для формирования группы с нуля – группа может формироваться только на основе другой, предварительно определенной группы. Базовая группа, на основе которой определены все другие группы, является группой, связанной с коммуникатором **MPI_COMM_WORLD** (через функцию **MPI_COMM_GROUP**).

MPI_COMM_GROUP(comm, group)

IN	**comm**	коммуникатор (дескриптор)
OUT	**group**	группа, соответствующая comm (дескриптор)

```
int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

MPI_COMM_GROUP(COMM, GROUP, IERROR)
INTEGER COMM, GROUP, IERROR

MPI::Group MPI::Comm::Get_group() const
```

Функция **MPI_COMM_GROUP** возвращает в **group** дескриптор группы из **comm.**

MPI_GROUP_UNION(group1, group2, newgroup)

IN	**group1**	первая группа (дескриптор)
IN	**group2**	вторая группа (дескриптор)
OUT	**newgroup**	объединенная группа (дескриптор)

```
int MPI_Group_union(MPI_Group group1, MPI_Group group2,
                    MPI_Group *newgroup)

MPI_GROUP_UNION(GROUP1, GROUP2, NEWGROUP, IERROR) INTEGER
GROUP1, GROUP2, NEWGROUP, IERROR

static MPI::Group MPI::Group::Union(const MPI::Group& group1,
                    const MPI::Group& group2)
```

Операции над множествами определяются следующим образом:

- **Объединение** (**union**) **–** содержит все элементы первой группы (**group1**) и следующие за ними элементы второй группы (**group2**), не входящие в первую группу.
- **Пересечение** (**intersect**) **–** содержит все элементы первой группы, которые также находятся во второй группе, упорядоченные, как в первой группе.
- **Разность** (**difference**) **–** содержит все элементы первой группы, которые не находятся во второй группе.

MPI_GROUP_INTERSECTION(group1, group2, newgroup)

IN	**group1**	первая группа (дескриптор)
IN	**group2**	вторая группа (дескриптор)
OUT	**newgroup**	группа, образованная пересечением (дескриптор)

```
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2,
                           MPI_Group *newgroup)

MPI_GROUP_INTERSECTION(GROUP1, GROUP2, NEWGROUP, IERROR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

static MPI::Group MPI::Group::Intersect(const MPI::Group& group1,
                                        const MPI::Group& group2)
```

MPI_GROUP_DIFFERENCE(group1, group2, newgroup)

IN	**group1**	первая группа(дескриптор)
IN	**group2**	вторая группа (дескриптор)
OUT	**newgroup**	исключенная группа (дескриптор)

```
int MPI_Group_difference(MPI_Group group1, MPI_Group group2,
                         MPI_Group *newgroup)

MPI_GROUP_DIFFERENCE(GROUP1, GROUP2, NEWGROUP, IERROR)
INTEGER GROUP1, GROUP2, NEWGROUP, IERROR

static MPI::Group MPI::Group::Difference(const MPI::Group& group1,
                                         const MPI::Group& group2)
```

Для этих операций порядок процессов в результирующей группе определен, прежде всего, в соответствии с порядком в первой группе и затем, в случае необходимости, в соответствии с порядком во второй группе. Ни объединение, ни пересечение не коммутативны, но обе ассоциативны. Новая группа может быть пуста (эквивалентна **MPI_GROUP_EMPTY)**.

MPI_GROUP_INCL(group, n, ranks, newgroup)

IN	**group**	группа (дескриптор)
IN	**n**	количество элементов в массиве номеров (и размер newgroup, целое)
IN	**ranks**	номера процессов в group, перешедших в новую группу (массив целых)
OUT	**newgroup**	новая группа, полученная из прежней, упорядоченная согласно ranks (дескриптор)

int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI::Group MPI::Group::Incl(int n, const int ranks[]) const

Функция **MPI_GROUP_INCL** создает группу **newgroup**, которая состоит из **n** процессов из **group** с номерами **rank[0],..., rank[n-1];** процесс с номером **i** в **newgroup** есть процесс с номером **ranks[i]** в **group**. Каждый из **n** элементов **ranks** должен быть правильным номером в **group**, и все элементы должны быть различными, иначе программа будет неверна. Если **n = 0**, то **newgroup** имеет значение **MPI_GROUP_EMPTY**. Эта функция может, например, использоваться, чтобы переупорядочить элементы группы.

MPI_GROUP_EXCL(group, n, ranks, newgroup)

IN	**group**	группа (дескриптор)
IN	**n**	количество элементов в массиве номеров (целое)
IN	**ranks**	массив целочисленных номеров в group, не входящих в newgroup
OUT	**newgroup**	новая группа, полученная из прежней, сохраняющая порядок, определенный group (дескриптор)

int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

MPI::Group MPI::Group::Excl(int n, const int ranks[]) const

Функция **MPI_GROUP_EXCL** создает группу **newgroup**, которая получена путем удаления из **group** процессов с номерами **ranks[0] ,... ranks[n-1].** Упорядочивание процессов в **newgroup** идентично упорядочиванию в **group**. Каждый из **n** элементов **ranks** должен быть правильным номером в **group**, и все элементы должны быть различными;

в противном случае программа неверна. Если **n = 0**, то **newgroup** идентична **group**.

5.3.3. Деструкторы групп

MPI_GROUP_FREE(group)
INOUT **group** идентификатор группы (дескриптор)

int MPI_Group_free(MPI_Group *group)

MPI_GROUP_FREE(GROUP, IERROR)
INTEGER GROUP, IERROR

void MPI::Group::Free()

Эта операция маркирует объект группы для удаления. Дескриптор **group** устанавливается вызовом в состояние **MPI_GROUP_NULL**. Любая выполняющаяся операция, использующая эту группу, завершится нормально.

5.4. УПРАВЛЕНИЕ КОММУНИКАТОРАМИ

Этот параграф описывает управление коммуникаторами в **MPI**. Операции обращения к коммуникаторам являются локальными, их выполнение не требует обмена между процессами. Операции, которые создают коммуникаторы, являются коллективными и могут потребовать обмена между процессами.

5.4.1. Доступ к коммуникаторам

Все следующие операции являются локальными.

MPI_COMM_SIZE(comm, size)
IN **comm** коммуникатор (дескриптор)
OUT **size** количество процессов в группе comm (целое)

int MPI_Comm_size(MPI_Comm comm, int *size)

MPI_COMM_SIZE(COMM, SIZE, IERROR)
INTEGER COMM, SIZE, IERROR

int MPI::Comm::Get_size() const

Эта функция указывает число процессов в коммуникаторе. Для **MPI_COMM_WORLD** она указывает общее количество доступных процессов.

MPI_COMM_RANK(comm, rank)
IN **comm** коммуникатор (дескриптор)
OUT **rank** номер вызывающего процесса в группе comm (целое)

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)

MPI_COMM_RANK(COMM, RANK, IERROR)
INTEGER COMM, RANK, IERROR

int MPI::Comm::Get_rank() const
```

Функция **MPI_COMM_RANK** возвращает номер процесса в частной группе коммуникатора. Ее удобно использовать совместно с **MPI_COMM_SIZE**.

MPI_COMM_COMPARE(comm1, comm2, result)
IN **comm1** первый коммуникатор (дескриптор)
IN **comm2** второй коммуникатор (дескриптор)
OUT **result** результат (целое)

```
int MPI_Comm_compare(MPI_Comm comm1,MPI_Comm comm2,  int *result)

MPI_COMM_COMPARE(COMM1, COMM2, RESULT, IERROR)
INTEGER COMM1, COMM2, RESULT, IERROR

static inc MPI::Comm::Compare(const MPI::Comm& comm1,
                              const MPI::Comm& comm2)
```

Результат **MPI_IDENT** появляется тогда и только тогда, когда **comm1** и **comm2** являются дескрипторами для одного и того же объекта. Результат **MPI_CONGRUENT** появляется, если исходные группы идентичны по компонентам и нумерации; эти коммуникаторы отличаются только контекстом. Результат **MPI_SIMILAR** имеет место, если члены группы обоих коммуникаторов являются одинаковыми, но порядок их нумерации различен. В противном случае выдается результат **MPI_UNEQUAL**.

5.4.2. Конструкторы коммуникаторов

Нижеперечисленные функции являются коллективными и вызываются всеми процессами в группе, связанной с **comm**. В MPI для создания нового коммуникатора необходим исходный коммуникатор. Основным коммуникатором для всех MPI коммуникаторов является коммуникатор **MPI_COMM_WORLD.**

Функция MPI_COMM_DUP дублирует существующий коммуникатор **comm,** возвращает в аргументе **newcomm** новый коммуникатор с той же группой.

MPI_COMM_DUP(comm, newcomm)

IN	**comm**	коммуникатор (дескриптор)
OUT	**newcomm**	копия comm (дескриптор)

```
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcomm)

MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR

MPI::Intracomm MPI::Intracomm::Dup() const
```

MPI_COMM_CREATE(comm, group, newcomm)

IN	**comm**	коммуникатор (дескриптор)
IN	**group**	группа, являющаяся подмножеством группы comm (дескриптор)
OUT	**newcomm**	новый коммуникатор (дескриптор)

```
int MPI_Comm_create(MPI_Comm comm, MPI_Group group,
                    MPI_Comm *newcomm)

MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
INTEGER COMM, GROUP, NEWCOMM, IERROR

MPI::Intracomm MPI::Intracomm::Create(const MPI::Group& croup) const
```

Эта функция создает новый коммуникатор **newcomm** с коммуникационной группой, определенной аргументом **group** и новым контекстом. Из **comm** в **newcomm** не передается никакой кэшированной информации. Функция возвращает **MPI_COMM_NULL** для процессов, не входящих в **group**. Запрос неверен, если не все аргументы в **group** имеют одинаковое значение или если **group** не является подмножеством группы, связанной с **comm**. Заметим, что запрос должен быть выполнен всеми процессами в **comm**, даже если они не принадлежат новой группе.

MPI_COMM_SPLIT(comm, color, key, newcomm)

IN	**comm**	коммуникатор (дескриптор)
IN	**color**	управление созданием подмножества (целое)
IN	**key**	управление назначением номеров целое)
OUT	**newcomm**	новый коммуникатор (дескриптор)

```
int MPI_Comm_split(MPI_Comm comm, int color, int key,  MPI_Comm *newcomm)
```

MPI_COMM_SPLIT(COMM, COLOR, KEY, NEWCOMM, IERROR)
INTEGER COMM, COLOR, KEY, NEWCOMM, IERROR

MPI::Intracomm MPI::Intracomm::Split(int color, int key) const

Эта функция делит группу, связанную с **comm,** на непересекающиеся подгруппы по одной для каждого значения **color**. Каждая подгруппа содержит все процессы одного цвета. В пределах каждой подгруппы процессы пронумерованы в порядке, определенном значением аргумента **key**, со связями, разделенными согласно их номеру в старой группе.

Для каждой подгруппы создается новый коммуникатор и возвращается в аргументе **newcomm**. Процесс может иметь значение цвета **MPI_UNDEFINED**, тогда переменная **newcomm** возвращает значение **MPI_COMM_NULL**. Это коллективная операция, но каждому процессу разрешается иметь различные значения для **color** и **key**.

Обращение к **MPI_COMM_CREATE (comm, group, newcomm)** эквивалентно обращению к **MPI_COMM_SPLIT (comm, color, key, newcomm),** где все члены **group** имеют **color =0** и **key** = номеру в **group**, и все процессы, которые не являются членами **group**, имеют **color** = **MPI_UNDEFINED**.

Функция **MPI_COMM_SPLIT** допускает более общее разделение группы на одну или несколько подгрупп с необязательным переупорядочением. Этот запрос используют только интра-коммуникаторы. Значение **color** должно быть неотрицательно.

5.4.3. Деструкторы коммуникаторов

MPI_COMM_FREE(comm)
INOUT **comm** удаляемый коммуникатор (handle)

int MPI_Comm_free(MPI_Comm *comm)

MPI_COMM_FREE(COMM, IERROR)
INTEGER COMM, IERROR

void MPI::Comm:Free()

Эта коллективная операция маркирует коммуникационный объект для удаления. Дескриптор устанавливается в **MPI_COMM_NULL**. Любые ждущие операции, которые используют этот коммуникатор, будут завершаться нормально; объект фактически удаляется только в том случае, если не имеется никаких других активных ссылок на него.

5.5. ПРИМЕРЫ

Пример 5.1.

```
main(int argc, char **argv)
{
 int me, count, count2;
 void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
 MPI_Group MPI_GROUP_WORLD, grprem;
 MPI_Comm commslave;
 static int ranks[ ] = {0};

 MPI_Init(&argc, &argv);
 MPI_Comm_group(MPI_COMM_WORLD,&MPI_GROUP_WORLD);
 MPI_Comm_rank(MPI_COMM_WORLD, &me);
                              /* локально */
 MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem);
                             /* локально */
 MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);

 if(me != 0)
 { …                          /* вычисления на подчиненном процессе */
       MPI_Reduce(send_buf,recv_buff,count, MPI_INT,
                        MPI_SUM, 1, commslave);
  }
         /* процесс 0 останавливается немедленно после выполнения этого
            reduce,  другие процессы – позже... */
 MPI_Reduce(send_buf2, recv_buff2, count2, MPI_INT,  MPI_SUM, 0,
                  MPI_COMM_WORLD);
 MPI_Comm_free(&commslave);
 MPI_Group_free(&MPI_GROUP_WORLD);
 MPI_Group_free(&grprem);
 MPI_Finalize();
}
```

Этот пример иллюстрирует, как из исходной группы создается группа, содержащая все процессы, кроме процесса 0, затем, как формируется коммуникатор (**commslave**) для новой группы. Новый коммуникатор используется в коллективном обмене, и все процессы выполняются в контексте **MPI_COMM_WORLD**. Пример иллюстрирует, как два коммуникатора (которые обязательно имеют различные контексты) защищают обмен. Это означает, что обмен в **commslave** изолирован от обмена в **MPI_COMM_WORLD,** и наоборот.

Пример 5.2. Следующий пример иллюстрирует "безопасное" выполнение парного и коллективного обменов в одном коммуникаторе.

```
#define TAG_ARBITRARY 12345
#define SOME_COUNT      50
 main(int argc, char **argv)
{  int me;
  MPI_Request request[2];
  MPI_Status status[2];
  MPI_Group MPI_GROUP_WORLD, subgroup;
  int ranks[] = {2, 4, 6, 8};
  MPI_Comm the_comm;
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD,&MPI_GROUP_WORLD);
  MPI_Group_incl(MPI_GROUP_WORLD,4,ranks, &subgroup);
                                        /*локально */
  MPI_Group_rank(subgroup, &me);     /* локально */
  MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);
                                        /* парный обмен  */
  if(me != MPI_UNDEFINED)    {
        MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE,
                    TAG_ARBITRARY,  the_comm, request);
        MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4,
                     TAG_ARBITRARY, the_comm, request+1);
   }                                    /* коллективная операция */
    for(i = 0; i < SOME_COUNT, i++)
          MPI_Reduce(..., the_comm);

   MPI_Waitall(2, request, status);
   MPI_Comm_free(t&he_comm);
   MPI_Group_free(&MPI_GROUP_WORLD);
   MPI_Group_free(&subgroup);
   MPI_Finalize();
}
```

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ К ГЛАВЕ 5

Контрольные вопросы к 5.1

1. Какие преимущества получает пользователь, использующий параллельные библиотеки?
2. Перечислите проблемы при написании параллельных библиотек.
3. Какие специфические структуры введены в MPI, необходимые для создания и использования параллельных библиотек?
4. В чем различие между группой процессов и коммуникатором?
5. Могут ли общаться процессы, принадлежащие разным коммуникаторам?
6. Что такое виртуальная топология?

Контрольные вопросы к 5.2

1. Можно ли ввести собственную нумерацию процессов в группе, например: 2,4,6,8,…?
2. Какая предопределенная константа является группой без членов?
3. В какой момент выполнения приложения определяется начальный коммуникатор MPI_COMM_WORLD?
4. MPI_COMM_WORLD является интер или интра-коммуникатором?
5. Можно ли удалить коммуникатор MPI_COMM_WORLD в течение времени существования процесса?

Контрольные вопросы к 5.3

1. Функции управления группой являются локальными или коллективными?
2. Что возвращает функция MPI_Group_rank, если процесс, в котором производится вызов этой функции, не является членом этой группы?
3. Как определить номер процесса в группе, зная его номер в группе, соответствующей коммуникатору MPI_COMM_WORLD?
4. Перечислите три возможных варианта результата, которые можно получить после вызова функции MPI_Group_compare.
5. Функции-конструкторы групп являются локальными?
6. Что такое базовая группа? Приведите пример базовой группы.
7. Можно ли определить в каждом процессе приложения собственную группу процессов?
8. Какая функция позволяет определить группу какого-либо коммуникатора?
9. Можно ли определить группу, которая не включает в себя процесс, ее создающий?
10. Перечислите операции над группами процессов, определенные для создания новой группы.

Контрольные вопросы к 5.4

1. Можно ли изменить число процессов приложения после инициализации MPI?
2. Могут ли быть строго вложенными множества процессов, определяющие различные коммуникаторы?
3. Должны ли быть непересекающимися множества процессов, определяющие различные коммуникаторы?
4. Должны ли множества процессов, определяющие различные коммуникаторы, состоять более чем из одного процесса?
5. Определены ли в каждой программе множества процессов, задающие различные коммуникаторы в количестве большем, чем 2?
6. Функции доступа к коммуникаторам локальные или коллективные?
7. В чем сходство и различие использования функций MPI_Group_size и MPI_Comm_size, MPI_Group_rank и MPI_Com_rank, MPI_Group_compare и MPI_Com_compare?
8. Функции создания коммуникаторов локальные или коллективные?
9. Какой коммуникатор создает MPI_Comm_dup?

10. Какой коммуникатор создает функция MPI_Comm_create, если она вызывается в процессе, не входящем в коммуникационную группу?
11. Сколько коммуникаторов создает MPI_Comm_split?

Контрольные вопросы к 5.5

1. В каком процессе находится результат коллективной операции MPI_Reduce в примере 5.1, которая выполняется в новом коммуникаторе commslave?
2. Объясните необходимость использования проверки me!=MPI_UNDEFINED в примере 5.2. В каком случае процесс получает номер MPI_UNDEFINED?
3. Сколько процессов и какие входят в коммуникатор the_comm в примере 5.2?
4. Для какой цели в примере 5.2 приведен цикл, в котором MPI_Reduce повторяется много раз?
5. По какой схеме осуществляется обмен данными в примере 5.2?

Задания для самостоятельной работы

5.1. Организуйте передачу данных внутри коммуникатора, который создан на основании группы, созданной из произвольного числа процессов, заданных пользователем. Например, если 10 процессов в MPI_COMM_WORLD, то создайте группу из 2,3,6,8,9 процессов.

5.2. Постройте ввод-вывод по принципу master/slave. Главный процесс должен принимать сообщения от подчиненных и печатать их в порядке нумерации (сначала 0, затем 1 и так далее). Подчиненные должны посылать каждый по 2 сообщения мастеру. Например, сообщения:

Hello from slave 3

Goodbye from slave 3

(с соответствующими сообщениями для каждого подчиненного процесса). Для решения разделите процессы из MPI_COMM_WORLD на два коммуникатора: один процесс в качестве главного и остальные процессы – подчиненные. Используйте новый коммуникатор для подчиненных процессов для того, чтобы получить номер подчиненного процесса в его коммуникаторе. Подчиненные процессы могут также выполнять любые вычисления.

5.3. Выполните задание 5.2, изменив код мастера, чтобы он позволял принимать три типа сообщений от подчиненных процессов: упорядоченный вывод (такой же, как и в примере 1); неупорядоченный вывод (как если бы каждый подчиненный процесс печатал непосредственно); сообщение о выходе. Мастер продолжает получать сообщения до тех пор, пока он не получит сообщения о выходе от всех подчиненных процессов. Для простоты программирования пусть каждый подчиненный процесс посылает сообщения:

Hello from slave 3
Goodbye from slave 3 в упорядоченном выходном режиме и
I'm exiting (3) в неупорядоченном режиме.

5.4. Напишите программу вычисления числа π методом Монте-Карло. Суть метода в следующем. Если радиус круга равен 1, тогда площадь круга равна π, а площадь квадрата вокруг круга равна 4. Следовательно, отношение площади кру-

га к площади квадрата равно π/4. Если теперь случайным образом генерировать точки в пределах контура квадрата, то отношение числа точек, попавших в круг, к общему количеству сгенерированных точек даст величину π/4. Сгенерированная точка находится в круге, если ее координаты соответствуют выражению $x^2+ y^2<1$. Для генерации случайных чисел создайте отдельный процесс, который будет рассылать эти числа другим процессам. Поскольку другие процессы должны будут выполнить коллективные операции, в которых не участвует этот процесс, необходимо определить коммуникатор, чья группа не включает генерацию случайных чисел.

5.5. Пусть имеем n процессов, где n – произвольно. Создайте два коммуникатора: отдельно из процессов с четными и нечетными номерами. Осуществите передачу данных по кольцу внутри каждого коммуникатора.

5.6. Пусть количество процессов n=l*m. Тогда процессы можно представить решеткой: l – строк и m столбцов. Создайте коммуникаторы для каждой строки и каждого столбца решетки. Осуществите передачу данных по кольцу внутри каждого коммуникатора.

Глава 6. ТОПОЛОГИИ ПРОЦЕССОВ

6.1. ВИРТУАЛЬНАЯ ТОПОЛОГИЯ

Топология является необязательным атрибутом, который дополняет систему интра-коммуникаторов и не применяется в интер-коммуникаторах. Топология обеспечивает удобный способ обозначения процессов в группе (внутри коммуникатора) и оказывает помощь исполнительной системе при размещении процессов в аппаратной среде. Во многих параллельных приложениях линейное распределение номеров не отражает в полной мере логической структуры обменов между процессами, которая зависит от структуры задачи и ее математического алгоритма. Часто процессы описываются в двух- и трехмерных топологических средах. Наиболее общим видом топологии является организация процессов, описываемая графовой структурой. В этой главе логическая организации процессов будет именоваться как “ виртуальная топология ”.

Следует различать виртуальную топологию процессов и физическую топологию процессов. Виртуальная топология должна применяться для назначения процессов физическим процессорам, если это позволит увеличить производительность обменов на данной машине. С другой стороны, описание виртуальной топологии зависит от конкретного приложения и является машинно-независимой.

Кроме возможности получить выгоду в производительности, виртуальная топология может быть использована как средство обозначения процессов, которое значительно улучшает читаемость программ.

Взаимосвязь процессов может быть представлена графом. Узлы такого графа представляют процессы, ребра соответствуют связям между процессами. Стандарт MPI предусматривает передачу сообщений между любой парой процессов в группе. Не обязательно указывать канал связи явно. Следовательно, отсутствие канала связи в граф-схеме процессов не запрещает соответствующим процессам обмениваться сообщениями. Из этого следует, что такая связь в виртуальной топологии может отсутствовать, например, из-за того, что топология не представляет удобного способа обозначения этого канала обмена. Возможным следствием такого отсутствия может быть то, что автоматическая программа для размещения процессов по процессорам (если такая будет существовать) не будет учитывать эту связь, и уменьшится эффективность вычислений. Ребра в графе не взвешены, поэтому процесс может изображаться только подключенным или неподключенным.

Графовое представление топологии пригодно для всех приложений. Однако во многих приложениях графовая структура является регулярной и довольно простой, поэтому использование всех деталей графового представления будет неудобным и малоэффективным в использовании. Большая доля всех параллельных приложений использует топологию колец, двумерных массивов или массивов большей размерности. Эти структуры полностью определяются числом размерностей и количеством процессов для каждой координаты. Кроме того, отображение решеток и торов на аппаратуру обычно более простая задача, чем отображение сложных графов.

Координаты в декартовой системе нумеруются от 0. Например, соотношение между номером процесса в группе и координатами в решетке (2 × 2) для четырех процессов будет следующим:

```
coord (0,0):        rank 0
coord (0,1):        rank 1
coord (1,0):        rank 2
coord (1,1):        rank 3
```

Функции **MPI_GRAPH_CREATE** и **MPI_CART_CREATE** используются для создания универсальной (графовой) и декартовой топологий соответственно. Эти функции являются коллективными. Как

и для других коллективных обращений, программа для работы с топологией должна быть написана корректно, вне зависимости от того, являются ли обращения синхронизированными или нет.

Функции создания топологии используют в качестве входа существующий коммуникатор **comm_old**, который определяет множество процессов, для которых и создается топология.

Функция **MPI_CART_CREATE** может использоваться для описания декартовой системы произвольной размерности. Для каждой координаты она определяет, является ли структура процессов периодической или нет. Таким образом, никакой специальной поддержки для структур типа гиперкуб не нужно. Локальная вспомогательная функция **MPI_DIMS_CREATE** используется для вычисления сбалансированности процессов для заданного количества размерностей.

Функция **MPI_TOPO_TEST** может использоваться для запроса о топологии, связанной с коммуникатором. Информация о топологии может быть извлечена из коммуникаторов с помощью функций **MPI_GRAPHDIMS_GET** и **MPI_GRAPH_GET** для графов и **MPI_CARTDIM_GET** и **MPI_CART_GET** – для декартовой топологии. Дополнительные функции обеспечивают работу с декартовой топологией: **MPI_CART_RANK** и **MPI_CART_COORDS** переводят декартовы координаты в номер группы и обратно; функция **MPI_CART_SUB** может использоваться для выделения декартова подпространства (аналогично функции **MPI_COMM_SPLIT**). Функция **MPI_CART_SHIFT** обеспечивает необходимую информацию для обмена между соседними процессами в декартовой системе координат. Две функции **MPI_GRAPH_NEIGHBORS_COUNT** и **MPI_GRAPH_NEIGHBORS** используются для выделения соседних процессов на топологической схеме. Функция **MPI_CART_SUB** является коллективной для группы входного коммуникатора, остальные функции локальные.

6.2. ТОПОЛОГИЧЕСКИЕ КОНСТРУКТОРЫ

6.2.1. Конструктор декартовой топологии

Функция **MPI_CART_CREATE** создает новый коммуникатор, к которому подключается топологическая информация.

Если **reorder = false**, то номер каждого процесса в новой группе идентичен номеру в старой группе. Иначе функция может переупорядочивать процессы (возможно, чтобы обеспечить хорошее наложение

виртуальной топологии на физическую систему). Если полная размерность декартовой решетки меньше, чем размер группы коммуникаторов, то некоторые процессы возвращаются с результатом **MPI_COMM_NULL** по аналогии с **MPI_COMM_SPLIT.** Вызов будет неверным, если он задает решетку большего размера, чем размер группы.

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

IN	**comm_old**	исходный коммуникатор (дескриптор)
IN	**ndims**	размерность создаваемой декартовой решетки (целое)
IN	**dims**	целочисленный массив размера ndims, хранящий количество процессов по каждой координате
IN	**periods**	массив логических элементов размера ndims, определяющий, периодична (true) или нет (false) решетка в каждой размерности
IN	**reorder**	нумерация может быть сохранена (false) или переупорядочена (true) (логическое значение)
OUT	**comm_cart**	коммуникатор новой декартовой топологии (дескриптор)

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims,  int *periods,
                    int reorder, MPI_Comm *comm_cart)

MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS,  REORDER,
                    COMM_CART, IERROR)
INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
LOGICAL PERIODS(*), REORDER

MPI::Cartcomm MPI::Intracomm::Create_cart(int ndims, const int dims[],
                                          const bool periods[], bool reorder) const
```

6.2.2. Декартова функция MPI_DIMS_CREATE

В декартовой топологии функция **MPI_DIMS_CREATE** помогает пользователю выбрать выгодное распределение процессов по каждой координате в зависимости от числа процессов в группе и некоторых ограничений, определенных пользователем. Эта функция используется, чтобы распределить все процессы группы в **n**-мерную топологическую среду (размер группы **MPI_COMM_WORLD**).

MPI_DIMS_CREATE(nnodes, ndims, dims)

IN	**nnodes**	количество узлов решетки (целое)
IN	**ndims**	число размерностей декартовой решетки(целое)
INOUT	**dims**	целочисленный массив размера ndims, указывающий количество вершин в каждой размерности.

```
int MPI_Dims_create(int nnodes, int ndims, int *dims)

MPI_DIMS_CREATE(NNODES, NDIMS, DIMS, IERROR)
INTEGER NNODES, NDIMS, DIMS(*), IERROR

void MPI::Compute_dims(int nnodes, int ndims, int dims[])
```

Элементы массива **dims** описывают декартову решетку координат с числом размерностей **ndims** и общим количеством узлов **nnodes**. Количество узлов по размерностям нужно сделать, насколько возможно, близкими друг другу, используя соответствующий алгоритм делимости. Вызывающая подпрограмма может дальше ограничить выполнение этой функции, задавая количество элементов массива **dims**. Если размерность **dims[i]** есть положительное число, функция не будет изменять число узлов в направлении **i**; будут изменены только те элементы, для которых **dims[i] = 0**.Отрицательные значения **dims[i]** неверны. Также будет неверно, если значение **nnodes** не кратно произведению **dims[i]**. Аргумент **dims[i]**, установленный вызывающей программой, будет упорядочен по убыванию. Массив **dims** удобен для использования в функции **MPI_CART_CREATE** в качестве входного. Функция является локальной.

Пример 6.1

dims перед вызовом	**вызов функции**	**dims после возврата управления**
(0,0)	MPI_DIMS_CREATE(6, 2, dims)	(3,2)
(0,0)	MPI_DIMS_CREATE(7, 2, dims)	(7,1)
(0,3,0)	MPI_DIMS_CREATE(6, 3, dims)	(2,3,1)
(0,3,0)	MPI_DIMS_CREATE(7, 3, dims)	Ошибка

6.2.3. Конструктор универсальной (графовой) топологии

Функция **MPI_GRAPH_CREATE** передает дескриптор новому коммуникатору, к которому присоединяется информация о графовой топологии. Если **reorder = false**, то номер каждого процесса в новой группе идентичен его номеру в старой группе. В противном случае функция может переупорядочивать процессы. Если количество узлов графа **nnodes** меньше, чем размер группы коммуникатора, то некоторые процессы возвращают значение **MPI_COMM_NULL** по аналогии с **MPI_CART_SPLIT** и **MPI_COMM_SPLIT.** Вызов будет неверным, если он определяет граф большего размера, чем размер группы исходного коммуникатора.

MPI_GRAPH_CREATE(comm_old, nnodes, index, edges, reorder, comm_graph)

IN	**comm_old**	входной коммуникатор (дескриптор)
IN	**nnodes**	количество узлов графа (целое)
IN	**index**	массив целочисленных значений, описывающий степени вершин
IN	**edges**	массив целочисленных значений, описывающий ребра графа
IN	**reorder**	номера могут быть переупорядочены (true) или нет (false)
OUT	**comm_graph**	построенный коммуникатор с графовой топологией (дескриптор)

```
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
                     int reorder, MPI_Comm *comm_graph)

MPI_GRAPH_CREATE(COMM_OLD, NNODES, INDEX, EDGES,  REORDER,
                 COMM_GRAPH, IERROR)
INTEGER COMM_OLD, NNODES, INDEX(*), EDGES(*),  COMM_GRAPH,
        IERROR
LOGICAL REORDER

int  MPI::Cartcomm::Get_dim() const.
```

Структуру графа определяют три параметра: **nnodes, index** и **edges**. **Nnodes –** число узлов графа от **0 до nnodes-1. i-**ый элемент массива **index** хранит общее число соседей первых **i** вершин графа. Списки соседей вершин **0, 1, ..., nnodes-1** хранятся в последовательности ячеек массива **edges**. Общее число элементов в **index** есть **nnodes**, а общее число элементов в **edges** равно числу ребер графа.

6.2.4. Топологические функции запроса

Если топология была определена одной из вышеупомянутых функций, то информация об этой топологии может быть получена с помощью функций запроса. Все они являются локальными вызовами.

MPI_TOPO_TEST(comm, status)

IN	**comm**	коммуникатор (дескриптор)
OUT	**status**	тип топологии коммуникатора comm (альтернатива)

```
int MPI_Topo_test(MPI_Comm comm, int *status)

MPI_TOPO_TEST(COMM, STATUS, IERROR)
INTEGER COMM, STATUS, IERROR

int MPI::Comm::Get_topology() const
```

Функция **MPI_TOPO_TEST** возвращает тип топологии, переданной коммуникатору.

Выходное значение **status** имеет одно из следующих значений:

MPI_GRAPH	топология графа
MPI_CART	декартова топология
MPI_UNDEFINED	топология не определена

Функции **MPI_GRAPHDIMS_GET** и **MPI_GRAPH_GET** возвращают графо-топологическую информацию, которая была связана с коммуникатором с помощью функции **MPI_GRAPH_CREATE**.

MPI_GRAPHDIMS_GET(comm, nnodes, nedges)

IN	**comm**	коммуникатор группы с графовой топологией (дескриптор)
OUT	**nnodes**	число вершин графа (целое, равно числу процессов в группе)
OUT	**nedges**	число ребер графа (целое)

int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)

MPI_GRAPHDIMS_GET(COMM, NNODES, NEDGES, IERROR)
INTEGER COMM, NNODES, NEDGES, IERROR

void MPI::Graphcomm::Get_dims(int nnodes[], int nedges[]) const

Информация, представляемая **MPI_GRAPHDIMS_GET,** может быть использована для корректного определения размера векторов **index** и **edges** для последующего вызова функции **MPI_GRAPH_GET**.

MPI_GRAPH_GET(comm, maxindex, maxedges, index, edges)

IN	**comm**	коммуникатор с графовой топологией (дескриптор)
IN	**maxindex**	длина вектора index (целое)
IN	**maxedges**	длина вектора edges (целое)
OUT	**index**	целочисленный массив, содержащий структуру графа (подробнее в описании функции MPI_GRAPH_CREATE)
OUT	**edges**	целочисленный массив, содержащий структуру графа

int MPI_Graph_get(MPI_Comm comm, int maxindex, int maxedges, int *index,
int *edges)

MPI_GRAPH_GET(COMM, MAXINDEX, MAXEDGES, INDEX, EDGES,
IERROR)
INTEGER COMM, MAXINDEX, MAXEDGES, INDEX(*), EDGES(*), IERROR

void MPI::Graphcomm::Get_topo (int maxindex, int maxedges, int index[],
int edges[]) const

Функции **MPI_CARTDIM_GET** и **MPI_CART_GET** возвращают информацию о декартовой топологии, которая была связана с функцией **MPI_CART_CREATE**.

MPI_CARTDIM_GET(comm, ndims)

IN	**comm**	коммуникатор с декартовой топологией (дескриптор)
OUT	**ndims**	число размерностей в декартовой топологии системы (целое)

```
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)

MPI_CARTDIM_GET(COMM, NDIMS, IERROR)
INTEGER COMM, NDIMS, IERROR

int MPI::Cartcomm::Get_dim() const
```

MPI_CART_GET(comm, maxdims, dims, periods, coords)

IN	**comm**	коммуникатор с декартовой топологией (дескриптор)
IN	**maxdims**	длина векторов dims, periods и coords (целое)
OUT	**dims**	число процессов по каждой декартовой размерности (целочисленный массив)
OUT	**periods**	периодичность (true/ false) для каждой декартовой размерности (массив логических элементов)
OUT	**coords**	координаты вызываемых процессов в декартовой системе координат (целочисленный массив)

```
int MPI_Cart_get(MPI_Comm comm, int maxdims, int *dims,  int *periods,
                 int *coords)

MPI_CART_GET(COMM, MAXDIMS, DIMS, PERIODS, COORDS, IERROR)
INTEGER COMM, MAXDIMS, DIMS(*), COORDS(*), IERROR
LOGICAL PERIODS(*)

Void MPI::Cartcomm::Get_topo(int maxdims, int dims[], bool periods[],
                             int coords[]) const
```

Для группы процессов с декартовой структурой функция **MPI_CART_RANK** переводит логические координаты процессов в номера, которые используются в процедурах парного обмена.

MPI_CART_RANK(comm, coords, rank)

IN	**comm**	коммуникатор с декартовой топологией (дескриптор)
IN	**coords**	целочисленный массив (размера ndims), описывающий декартовы координаты процесса
OUT	**rank**	номер указанного процесса (целое)

```
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

MPI_CART_RANK(COMM, COORDS, RANK, IERROR)
INTEGER COMM, COORDS(*), RANK, IERROR
```

Int MPI::Cartcomm::Get_cart_rank(const int coords[]) const.

Для размерности **i** с **periods(i) = true**, если координата **cords(i)** выходит за границу диапазона – **coords(i)<0** или **coords(i)≥dims(i)**, она автоматически сдвигается назад к интервалу **0≤coords(i)<dims(i)**. Выход координат за пределы диапазона неверен для непериодических размерностей.

Функция **MPI_CART_COORDS** используется для перевода номера в координату.

MPI_CART_COORDS(comm, rank, maxdims, coords)

IN	**comm**	коммуникатор с декартовой топологией (дескриптор)
N	**rank**	номер процесса внутри группы comm (целое)
IN	**maxdims**	длина вектора coord (целое)
OUT	**coords**	целочисленный массив (размера ndims), содержащий декартовы координаты указанного процесса (целое)

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

MPI_CART_COORDS(COMM, RANK, MAXDIMS, COORDS, IERROR)
INTEGER COMM, RANK, MAXDIMS, COORDS(*), IERROR

Void MPI::Cartcomm_Get_coords (int rank, int maxdims, int coords[]) const

MPI_GRAPH_NEIGHBORS_COUNT и **MPI_GRAPH_NEIGHBORS** предоставляют похожую информацию для графической топологии.

MPI_GRAPH_NEIGHBORS_COUNT(comm, rank, nneighbors)

IN	**comm**	коммуникатор с графовой топологией (дескриптор)
IN	**rank**	номер процесса в группе comm (целое)
OUT	**nneighbors**	номера процессов, являющихся соседними указанному процессу (целочисленный массив)

int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)

MPI_GRAPH_NEIGHBORS_COUNT(COMM, RANK, NNEIGHBORS, IERROR)
INTEGER COMM, RANK, NNEIGHBORS, IERROR

int MPI::Graphcomm::Get_neighbors_count (int rank) const

MPI_GRAPH_NEIGHBORS(comm, rank, maxneighbors, neighbors)

IN	**comm**	коммуникатор с графовой топологией (дескриптор)
IN	**rank**	номер процесса в группе comm (целое)
IN	**maxneighbors**	размер массива neighbors (целое)
OUT	**neighbors**	номера процессов, соседних данному (целочисленный массив)

int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors,
int *neighbors)

```
MPI_GRAPH_NEIGHBORS(COMM, RANK, MAXNEIGHBORS, NEIGHBORS,
                                    IERROR)
INTEGER COMM, RANK, MAXNEIGHBORS, NEIGHBORS(*),  IERROR

void MPI::Graphcomm::Get_neighbors (int rank, int maxneighbors,
                                    int neighbors[] )  const
```

Пример 6.2. Предположим, что **comm** является коммуникатором с топологией типа “тасовка”.

Пусть группа содержит 2^n процессов. Каждый процесс задан двоичным **n** – разрядным кодом $a_1 , a_2 , \ldots, a_n$, где $a_i \in \{0,1\}$ и имеет трех соседей:

exchange $(a_1 , \ldots, a_n) = a_1 , \ldots, a_{n-1} , \hat{a}_n$ ($\hat{a} = 1-a$);

shuffle $(a_1 , \ldots, a_n) = a_2 , \ldots, a_n , a_1$;

unshuffle$(a_1 , \ldots, a_n) = a_n , a_1 , \ldots, a_{n-1}$.

Таблица связей каждого процесса с соседями для n=3

узлы	exchange соседи(1)	shuffle соседи(2)	unshuffle соседи(3)
0 (000)	1	0	0
1 (001)	0	2	4
2 (010)	3	4	1
3 (011)	2	6	5
4 (100)	5	1	2
5 (101)	4	3	6
6 (110)	7	5	3
7 (111)	6	7	7

Предположим, что коммуникатор **comm** имеет эту топологию. Представленный ниже фрагмент программы в цикле обходит трех соседей и делает соответствующую перестановку для каждого.

```
!  пусть каждый процесс хранит действительное число A
!  получение информации о соседях
     CALL MPI_COMM_RANK(comm, myrank, ierr)
     CALL MPI_GRAPH_NEIGHBORS(comm, myrank, 3, neighbors, ierr)
!  выполнение перестановки для exchange
     CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL,
                    neighbors(1), 0, neighbors(1), 0, comm, status, ierr)

!  выполнение перестановки для shuffle
     CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL,
                     neighbors(2), 0, neighbors(3), 0, comm, status, ierr)
!  выполнение перестановки для unshuffle
```

```
CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL,
                neighbors(3), 0, neighbors(2), 0, comm, status, ierr)
```

6.2.5. Сдвиг в декартовых координатах

Если используется декартова топология, то операцию **MPI_SENDRECV** можно выполнить путем сдвига данных вдоль направления координаты. В качестве входного параметра **MPI_SENDRECV** использует номер процесса-отправителя для приема и номер процесса-получателя – для передачи. Если функция **MPI_CART_SHIFT** выполняется для декартовой группы процессов, то она передает вызывающему процессу эти номера, которые затем могут быть использованы для **MPI_SENDRECV**. Пользователь определяет направление координаты и величину шага (положительный или отрицательный). Эта функция является локальной.

MPI_CART_SHIFT(comm, direction, disp, rank_source, rank_dest)

IN	**comm**	коммуникатор с декартовой топологией (дескриптор)
IN	**direction**	координата сдвига (целое)
IN	**disp**	направление смещения (> 0: смещение вверх, < 0: смещение вниз) (целое)
OUT	**rank_source**	номер процесса-отправителя (целое)
OUT	**rank_dest**	номер процесса-получателя (целое)

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
                   int *rank_dest)

MPI_CART_SHIFT(COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST,
                IERROR)
INTEGER COMM, DIRECTION, DISP, RANK_SOURCE, RANK_DEST, IERROR

Void MPI::Cartcomm::Shift(int direction, int disp, int& rank_source,
                          int& rank_dest) const
```

Аргумент **direction** указывает размерность (координату) по которой сдвигаются данные. Координаты маркируются от **0** до **ndims-1**, где **ndims** – число размерностей. В зависимости от периодичности декартовой группы в указанном направлении координаты **MPI_CART_SHIFT** указывает признаки для кольцевого сдвига или для сдвига без переноса. В случае сдвига без переноса в **rank_source** или **rank_dest** может быть возвращено значение **MPI_PROC_NULL** для указания, что процесс-отправитель или процесс-получатель при сдвиге вышли из диапазона.

Пример 6.3. Коммуникатор **comm** имеет двумерную периодическую декартову топологию. Двумерная матрица элементов типа **REAL** хранит один элемент на процесс в переменной **A**. Нужно выполнить скошенное преобразование матрицы путем сдвига столбца **i** (вертикально, то есть вдоль столбца) **i** раз.

```
!  определяется номер процесса
     CALL MPI_COMM_RANK(comm, rank, ierr)
!  определяются декартовы координаты
     CALL MPI_CART_COORDS(comm, rank, maxdims, coords, ierr)
!  вычисляются номера процесса-отправителя и процесса-получателя  при  сдвиге
     CALL MPI_CART_SHIFT(comm, 0, coords(2), source, dest, ierr)
!  «скошенный» массив
     CALL MPI_SENDRECV_REPLACE(A, 1, MPI_REAL, dest, 0,  source, 0, comm,
                                                    status, ierr)
```

6.2.6. Декомпозиция декартовых структур

MPI_CART_SUB(comm, remain_dims, newcomm)

IN	**comm**	коммуникатор с декартовой топологией (дескриптор)
IN	**remain_dims**	i-й элемент в remain_dims показывает, содержится ли i-я размерность в подрешетке (true) или нет (false) (вектор логических элементов)
OUT	**newcomm**	коммуникатор, содержащий подрешетку, которая включает вызываемый процесс (дескриптор)

```
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims,  MPI_Comm *newcomm)

MPI_CART_SUB(COMM, REMAIN_DIMS, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR
LOGICAL REMAIN_DIMS(*)

MPI::Cartcomm MPI::Cartcomm::Sub(const bool remain_dims[]) comst
```

Если декартова топология была создана с помощью функции **MPI_CART_CREATE**, то функция **MPI_CART_SUB** может использоваться для декомпозиции группы в подгруппы, которые являются декартовыми подрешетками, и построения для каждой подгруппы коммуникаторов с подрешеткой с декартовой топологией.

Пример 6.4

Допустим, что **MPI_CART_CREATE** создала решетку размером (2 × 3 × 4). Пусть **remain_dims = (true, false, true**). Тогда обращение к **MPI_CART_SUB (comm, remain_dims, comm_new)** создаст три коммуникатора, каждый с восьмью процессами размерности 2×4 в декар-

товой топологии. Если **remain_dims = (false, false, true**), то обращение к функции **MPI_CART_SUB (comm, remain_dims, comm_new)** создает шесть непересекающихся коммуникаторов, каждый с четырьмя процессами в одномерной декартовой топологии.

Пример 6.5

Дифференциальные уравнения в частных производных, например уравнение Пуассона, могут быть решены на прямоугольной решетке. Пусть процессы располагаются в двумерной структуре. Каждый процесс запрашивает номера соседей в четырех направлениях (вверх, вниз, вправо, влево). Числовая задача решается итерационным методом, детали которого скрыты в подпрограмме. На каждой итерации каждый процесс вычисляет новые значения для функции в области решетки, за которую он ответственен. Затем процесс на своих границах должен обменяться значениями функции с соседними процессами. Например, подпрограмма обмена может содержать вызов функции MPI_SEND (..., neigh_rank (1), ...) , чтобы затем послать модифицированные значения функции левому соседу (i-1, j).

```
    integer ndims, num_neihg
    logical reorder
    parameter (ndims=2,num_neigh=4,reorder=.true.)
    integer comm, comm_cart, dims (ndims), neigh_def (ndims), ierr
    integer neigh_rank (num_neigh), own_position (ndims), i,j
    logical periods (ndims)
    real*8 u(0:101,0:101), f(0:101,0:101)
    data dims /ndims*0/
    comm = MPI_COMM_WORLD
!   устанавливает размер решетки и периодичность
    call MPI_DIMS_CREATE (comm, ndims, dims, ierr)
    periods (1) = .TRUE.
    periods (2) = .TRUE.
!   создает структуру  в группе WORLD и запрашивает   собственную позицию
   call MPI_CART_CREATE (comm, ndims, dims, periods, reorder, comm_cart, ierr)
   call MPI_CART_GET (comm_cart, ndims, dims, periods, own_position, ierr)

!  просматривает номера соседей, собственные координаты есть  (i,j).
!  соседи – процессы с номерами (i-1,j), (i+1,j), (i,j-1), (i,j+1)
   i = own_position(1)
   j = own_position(2)
   neigh_def(1)= i-1
   neigh_def(2)= j
   call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(1), ierr)
```

```
  neigh_def(1)= i+1
  neigh_def(2)= j
  call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(2), ierr)
  neigh_def(1)= i
  neigh_def(2)= j-1
  call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(3), ierr)
  neigh_def(1)= i
  neigh_def(2)= j+1
  call MPI_CART_RANK (comm_cart, neigh_def, neigh_rank(4), ierr)

!  инициализация функций решеток и начало итерации
  call init (u,f)
   do 10 it=1,100
      call relax (u,f)
!     обмен данными с соседними процессами
      call exchange (u, comm_cart, neigh_rank, num_neigh)
10  continue
    call output (u)
end
```

КОНТРОЛЬНЫЕ ВОПРОСЫ К ГЛАВЕ 6

Контрольные вопросы к 6.1

1. Топология – это частный случай коммуникатора?
2. Топология применима к интра-коммуникаторам или интер-коммуникаторам?
3. В чем различие между виртуальной и физической топологией процессов?
4. Можно ли создать виртуальную топологию на основании любого коммуникатора, полученного с помощью процедур из главы 5?
5. Приведите примеры задач, где использование декартовой топологии было бы очень удобным.

Контрольные вопросы к 6.2

1. Функции создания виртуальной топологии коллективные или локальные?
2. Как создать декартову топологию размерности 1?
3. Что произойдет, если полная размерность декартовой решетки меньше, чем размер коммуникатора? А если больше?
4. Опишите математическую функцию, которая реализуется при вызове MPI_Dims_create.
5. Объясните, почему произойдет ошибка в четвертом случае примера 6.1.
6. Что возвращает функция MPI_Topo_test?
7. Необходимо ли вызывать функцию MPI_Graphdims_get перед вызовом функции MPI_Graph_get? Почему?
8. Какую информацию о декартовой топологии возвращает функция MPI_Cartdim_get? А – MPI_Cart_get?
9. Пусть имеются четыре процесса 0, 1, 2, 3 со следующей матрицей смежности:

Процесс	Соседи
0	1, 3
1	0
2	3
3	0, 2

Опишите параметры nnodes, index, edges при создании графовой топологии вызовом функции MPI_Graph_create (…, nnodes, *index, *edges, …) для данного случая.

10. Какие функции переводят логические координаты процесса с декартовой структурой, графовой топологией в номер, который используется в процедурах парного обмена?
11. Какие функции переводят номер, который используется в процедурах парного обмена в логические координаты процесса с декартовой структурой и графовой топологией?
12. Опишите создание коммуникатора comm в примере 6.2.
13. Как определить “соседей” по топологии?
14. Сколько всевозможных коммуникаторов можно получить при вызове функции MPI_Cart_sub для одной декартовой топологии, созданной функцией MPI_Cart_create?

Задания для самостоятельной работы

6.1. Напишите программу, которая рассылает данные от процесса ноль всем другим процессам по конвейеру. Этот означает, что процесс i должен принять данные от i-1 и послать их процессу i + 1, и так до тех пор, пока не будет достигнут последний процесс. Будем считать, что данные содержат единственное целое. Процесс ноль читает данные от пользователя, пока не появится на входе отрицательное целое число. Выполните задание, используя картезианскую топологию процессов размерности 1; для определения номеров соседних процессов используйте MPI_Cart_shift.

6.2. Выполните задание 6.1, используя MPI_Isend и MPI_Irecv, MPI_Wait. Для возможности совмещения вычислений и обмена введите процедуру для вычисления произвольного характера.

6.3. Выполните задание 6.1, организовав передачу данных по кольцу, т.е. последний процесс должен передать данные нулевому, который их выводит на экран.

6.4. Выполните задание 5.5 с использованием декартовой топологии.

Р А З Д Е Л 3. **ПРОГРАММИРОВАНИЕ ПРИЛОЖЕНИЙ**

Глава 7. МАТРИЧНЫЕ ЗАДАЧИ

Задача умножения матриц является базовой операцией для многих приложений. В этой главе рассматриваются различные методы параллельного решения этой задачи [4].

7.1. САМОПЛАНИРУЮЩИЙ АЛГОРИТМ УМНОЖЕНИЯ МАТРИЦ

Рассмотрим задачу вычисления произведения матрицы на вектор, которая естественным образом обобщается на задачу умножения матриц. Для решения задачи используем самопланирующий алгоритм, в котором один процесс (главный) является ответственным за координацию работы других процессов (подчиненных). Соответствующая программа представлена ниже.

Для наглядности единая программа матрично-векторного умножения разбита на три части: общую часть, код главного процесса и код подчиненного процесса.

В общей части программы описываются основные объекты задачи: матрица *A*, вектор *b*, результирующий вектор *c*, определяется число процессов (не меньше двух). Задача разбивается на две части: главный процесс (master) и подчиненные процессы.

В задаче умножения матрицы на вектор единица работы, которую нужно раздать процессам, состоит из скалярного произведения строки матрицы *A* на вектор *b*.

```
   program main
   use mpi
   integer MAX_ROWS, MAX_COLS, rows, cols
   parameter (MAX_ROWS = 1000, MAX_COLS = 1000)
!  матрица A, вектор b, результирующий вектор c
   double precision a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_ROWS)
   double precision buffer (MAX_COLS), ans
   integer myid, master, numprocs, ierr, status (MPI_STATUS_SIZE)
   integer i, j, numsent, sender, anstype, row
   call MPI_INIT( ierr )
   call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
   call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
!  главный процесс –  master
   master = 0
!  количество строк и столбцов матрицы A
     rows   = 10
```

```
   cols   = 100
   if ( myid .eq. master ) then
!  код главного процесса
   else
!  код подчиненного процесса
   endif
   call MPI_FINALIZE(ierr)
   stop
   end
```

Рис. 7.1. Программа умножения матрицы на вектор: общая часть

Код главного процесса представлен на рис. 7.2. Сначала главный процесс передает вектор *b* в каждый подчиненный процесс. Затем главный процесс пересылает одну строку матрицы *A* в каждый подчиненный процесс. Главный процесс, получая результат от очередного подчиненного процесса, передает ему новую работу. Цикл заканчивается, когда все строки будут розданы и от каждого подчиненного процесса получен результат.

```
!    инициализация  A и b
     do 20 j = 1, cols
        b(j) = j
        do 10 i = 1, rows
           a(i,j) = i
10      continue
20    continue
      numsent = 0
!     посылка b каждому подчиненному процессу
      call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
                          MPI_COMM_WORLD, ierr)
!     посылка строки каждому подчиненному процессу;  в   TAG – номер строки = i
      do 40 i = 1,min(numprocs-l, rows)
         do 30 j = I, cols
            buffer(j) = a(i,j)
30       continue
         call MPI_SEND(buffer, cols,  MPI_DOUBLE_PRECISION, i, i,
                          MPI_COMM_WORLD, ierr)
         numsent = numsent + l
40    continue
!     прием результата от подчиненного процесса
      do 70 i = 1, rows
!        MPI_ANY_TAG – указывает, что принимается любая строка
         call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,
                             MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
         sender     = status (MPI_SOURCE)
```

```
        anstype   = status (MPI_TAG)
!       определяем номер строки
        c(anstype) = ans
        if (numsent .lt. rows) then
!       посылка следующей строки
          do 50 j = 1, cols
               buffer(j) = a(numsent+l, j)
50        continue
          call MPI_SEND (buffer, cols, MPI_DOUBLE_PRECISION,  sender,
                              numsent+l, MPI_COMM_WORLD, ierr)
          numsent = numsent+l
        else
!       посылка признака конца работы
          call MPI_SEND(MPI_BOTTQM, 0, MPI_DOUBLE_PRECISION,sender,
                              0, MPI_COMM_WORLD, ierr)
        endif
70  continue
```

Рис. 7.2. Программа для умножения матрицы на вектор: код главного процесса

При передаче данных из главного процесса в параметре **tag** указывается номер передаваемой строки. Этот номер после вычисления произведения вместе с результатом будет отправлен в главный процесс, чтобы главный процесс знал, где размещать результат.

Подчиненные процессы посылают результаты в главный процесс и параметр **MPI_ANY_TAG** в операции приема главного процесса указывает, что главный процесс принимает строки в любой последовательности. Параметр **status** обеспечивает информацию, относящуюся к полученному сообщению. В языке Fortran это – массив целых чисел размера **MPI_STATUS_SIZE**. Аргумент **SOURCE** содержит номер процесса, который послал сообщение, по этому адресу главный процесс будет пересылать новую работу. Аргумент **TAG** хранит номер обработанной строки, что обеспечивает правильное размещение полученного результата. После того как главной процесс разослал все строки матрицы A, на запросы подчиненных процессов он отвечает сообщением с отметкой 0. На рис. 7.3. представлена программа подчиненного процесса.

```
!    прием вектора b всеми подчиненными процессами
     call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master,
                          MPI_COMM_WORLD, ierr)
!    выход, если процессов больше количества строк матрицы
     if (numprocs  .gt. rows)    goto 200
!    прием строки матрицы
```

```
90  call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, master,
                          MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
     if (status (MPI_TAG) .eq. 0) then  go to 200
!    конец работы
     else
          row = status (MPI_TAG)
!         номер полученной строки
          ans = 0.0
          do 100 i = 1, cols
!              скалярное произведение векторов
               ans = ans+buffer(i)*b(i)
100       continue
!         передача результата головному процессу
          call MPI_SEND(ans,1,MPI_DOUBLE_PRECISION,master,row,
                               MPI_COMM_WORLD, ierr)
          go to 90
!         цикл для приема следующей строки матрицы
     endif
200 continue
```

Рис. 7.3. Программа для матрично-векторного умножения: подчиненный процесс

Каждый подчиненный процесс получает вектор b. Затем организуется цикл, состоящий в том, что подчиненный процесс получает очередную строку матрицы *A*, формирует скалярное произведение строки и вектора b, посылает результат главному процессу, получает новую строку и так далее.

Обобщим вышеописанный алгоритм для умножения матриц. На рис. 7.4. представлена программа главного процесса.

```
   program main
   use mpi
   integer MAX_AROWS, MAX_ACOLS, MAX_BCOLS
   parameter (MAX_AROWS = 20, MAX_ACOLS = 1000, MAX_BCOLS = 20)
!  матрицы A,B,C
   double precision a(MAX_AROWS,MAX_ACOLS),
   double precision b(MAX_ACOLS,MAX_BCOLS)
   double precision c (MAX_AROWS, MAX_BCOLS)
   double precision buffer (MAX_ACOLS), ans (MAX_ACOLS)
   double precision starttime, stoptime
   integer myid, master, numprocs, ierr, status (MPI_STATUS_SIZE)
   integer i, j , numsent, sender, anstype, row, arows, acols, brows, bcols, crows, ccols
   call MPI_INIT( ierr )
   call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
   call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
```

```
!   количество строк и столбцов матрицы A
    arows  = 10
    acols  = 20
!   количество строк и столбцов матрицы B
    brows  = 20
    bcols  = 10
!   количество строк и столбцов матрицы C
    crows  = arows
    ccols  = bcols
    if ( myid .eq. 0 ) then
!   код главного процесса
    else
!   код починенного процесса
    endif
    call MPI_FINALIZE(ierr)
    stop
    end
```

Рис. 7.4. Программа умножения матриц: общая часть

Программа на рис. 7.4. отличается от программы общей части на рис. 7.1. описанием основных объектов: матриц *A,B,C* и их размерностей. На рис. 7.5. представлена программа главного процесса.

```
!   инициализация  A и B
    do 10 j = 1, acols
        do 10 i = 1, arows
            a(i,j) = i
10      continue
    do 20 j = 1, bcols
        do 20 i = 1, brows
             b(i,j) = i
20      continue
!    посылка матрицы B каждому подчиненному процессу
    do 25 i = l,bcols
25      call MPI_BCAST(b(l,i), brows, MPI_DOUBLE_PRECISION, master,
                                MPI_COMM_WORLD, ierr)
    numsent = 0
!   посылка строки каждому подчиненному процессу;
!   в TAG – номер строки = i     для простоты полагаем  arows >= numprocs –  1
    do 40 i = l,numprocs-1
         do 30 j = 1, acols
30           buffer(j) = a(i,j)
         call MPI_SEND (buffer, acols, MPI_DOUBLE_PRECISION, i, i,
                                MPI_COMM_WORLD, ierr)
40     numsent = numsent+l
```

```
      do 70 i = 1, crows
         call MPI_RECV(ans, ccols, MPI_DOUBLE_PRECISION,
                       MPI_ANY_SOURCE, MPI_ANY_TAG,
                       MPI_COMM_WORLD, status, ierr)
         sender    = status (MPI_SOURCE)
         anstype   = status (MPI_TAG)
         do 45 j = 1, ccols
45          c(anstype, j) = ans(j)
         if (numsent .lt. arows) then
            do 50 j = 1, acols
50             buffer(j) = a(numsent+l,j)
            call MPI_SEND (buffer, acols, MPI_DOUBLE_PRECISION,
                           sender, numsent+l, MPI_COMM_WORLD, ierr)
            numsent = numsent+l
         else
!        посылка признака конца работы
            call MPI_SEND(MPI_BOTTQM, 1, MPI_DOUBLE_PRECISION,
                          sender, 0, MPI_COMM_WORLD, ierr)
         endif
70    continue
```

Рис.7.5. Умножение матриц: программа главного процесса.

В главном процессе можно выделить следующие основные этапы: передачу матрицы *B* в каждый процесс, посылку строки матрицы *A* после каждого получения результирующей строки матрицы *C* от процессов.

На рис. 7.6. представлена программа подчиненного процесса программы умножения матриц.

```
!    прием матрицы B каждым подчиненным процессом
     do 85 i = l,bcols
        call MPI_BCAST(b(l,i), brows, MPI_DOUBLE_PRECISION, master,
                       MPI_COMM_WORLD, ierr)
85   continue
!    прием строки матрицы A каждым подчиненным процессом
90   call MPI_RECV (buffer, acols, MPI_DOUBLE_PRECISION, master,
                    MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)
     if (status (MPI_TAG) .eq. 0) then   go to 200
     else
        row = status (MPI_TAG)
        do 100 i = l,bcols
           ans(i) = 0.0
           do 95 j = 1, acols
!             вычисление результатов
```

```
                ans(i) = ans(i) + buffer(j)*b(j,i)
95            continue
100     continue
!       посылка результата
        call MPI_SEND(ans, bcols, MPI_DOUBLE_PRECISION, master,
                              row, MPI_COMM_WORLD, ierr)
         go to 90
      endif
200 continue
```

Рис. 7.6. Умножение матриц: программа подчиненного процесса

В подчиненном процессе основными этапами являются: получение матрицы *B* (операция широковещания), получение строки *A*, вычисление строки *C,* посылка строки результирующей матрицы *C* главной программе.

7.2. КЛЕТОЧНЫЙ АЛГОРИТМ УМНОЖЕНИЯ МАТРИЦ

Рассмотрим другой алгоритм умножения матриц, получивший название клеточного [8]. В отличие от самопланирующего алгоритма, описанного в параграфе 7.1, этот алгоритм демонстрирует использование основных особенностей MPI относительно большинства других систем передач сообщений: коммуникаторов и топологии.

7.2.1. Клеточный алгоритм

Пусть матрицы $A = (a_{ij})$ и $B = (b_{ij})$ имеют порядок n, число процессов p является полным квадратом, n нацело делится на q:

$$p = q^2, \quad n' = n/q.$$

В клеточном алгоритме матрица распредляется по процессам по принципу шахматной доски. Будем рассматривать процессы как виртуальную двумерную сетку размером $q \times q$, и каждый процесс связан с подматрицей $n' \times n'$. Более формально можно определить так: имеется взаимно однозначное соответствие между числом процессов и частями матриц:

$$\phi : \{0,1,..., p-1\} \to \{(s,t) : 0 \le s,t \le q-1\}$$

Это соответствие определяет сетку процессов: процесс i работает со строками и столбцами, номера которых определяет $\phi(i)$. Процесс с номером $\phi^{-1}(s, t)$ назначен подматрице

$$A_{st} = \begin{pmatrix} a_{s*n',t*n'} \ldots a_{(s+1)*n'-1,t*n'} \\ \ldots \\ .a_{s*n',(t+1)*n'-1} \ldots a_{(s+1)*n'-1,(t+1)*n'-1} \end{pmatrix}$$

и подматрице B_{st}, определенной аналогично. В клеточном алгоритме подматрицы A_{rs} и B_{st} при $s = 0, 1, \ldots, q$ -1, перемножаются и сохраняются в процессе $\phi^{-1}(r; t)$. Например, если p=9; $\phi(x) = (x/3; x \bmod 3)$ и $n = 6$, то A будет разделена следующим образом:

Процесс 0	Процесс 1	**Процесс 2**
$A_{00} = \begin{pmatrix} a_{00}a_{01} \\ a_{10}a_{11} \end{pmatrix}$	$A_{01} = \begin{pmatrix} a_{02}a_{03} \\ a_{12}a_{13} \end{pmatrix}$	$A_{02} = \begin{pmatrix} a_{04}a_{05} \\ a_{14}a_{15} \end{pmatrix}$
Процесс 3	Процесс 4	Процесс 5
$A_{10} = \begin{pmatrix} a_{20}a_{21} \\ a_{30}a_{31} \end{pmatrix}$	$A_{11} = \begin{pmatrix} a_{22}a_{23} \\ a_{32}a_{33} \end{pmatrix}$	$A_{12} = \begin{pmatrix} a_{24}a_{25} \\ a_{34}a_{35} \end{pmatrix}$
Процесс 6	Процесс 7	Процесс 8
$A_{20} = \begin{pmatrix} a_{40}a_{41} \\ a_{50}a_{51} \end{pmatrix}$	$A_{21} = \begin{pmatrix} a_{42}a_{43} \\ a_{52}a_{53} \end{pmatrix}$	$A_{22} = \begin{pmatrix} a_{44}a_{45} \\ a_{54}a_{55} \end{pmatrix}$

Алгоритм клеточного умножения матриц представлен ниже:

for(step = 0; step < q; step++)
{

1. Выбрать подматрицу из A в каждой строке процессов. Подматрица, выбранная в r-й строке, есть A_{ru}, где $u = (r + step) \bmod q$.
2. Для каждой строки процессов переслать всем другим процессам той же строки выбранную подматрицу.
3. Каждый процессе умножает полученную подматрицу из A на подматрицу из B, в настоящее время находящуюся в процессе.
4. Для каждого процесса переслать подматрицу из B в процесс, находящийся выше. (В процессах первой строки осуществляется пересылка подматрицы в последнюю строку.)

}

7.2.2. Способы создания коммуникаторов

Из клеточного алгоритма следует, что было бы удобным трактовать каждую строку и каждый столбец процессов как новый коммуникатор. Это можно организовать несколькими способами.

Способ 1. Для иллюстрации создадим коммуникатор, который составляет группа из процессов первой строки виртуальной сетки. Предположим, что **MPI_COMM_WORLD** состоит из p процессов, где $q^2 = p$. Предположим, что $\phi(x) = (x/q;\ x \bmod q)$. Тогда первая строка процессов состоит из процессов с номерами 0, 1,..., q-1.

Чтобы создать группу нового коммуникатора, необходимо воспользоваться функциями, описанными в главе 5, для создания новой группы и коммуникатора, например так, как это сделано в следующем фрагменте программы.

```
MPI_Group MPI_GROUP_WORLD;
MPI_Group first_row_group;
MPI_Comm first_row_comm;
int row_size;
int* process_ranks;
                    /* Создание списка процессов в новом коммуникаторе */
process_ranks = (int*) malloc(q*sizeof(int));
for (proc = 0; proc < q; proc++) process_ranks[proc] = proc;
                     /* Получение группы MPI_COMM_WORLD */
MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
                     /* Создание новой группы */
MPI_Group_incl(MPI_GROUP_WORLD, q, process_ranks, &first_row_group);
                     /* Создание нового коммуникатора */
MPI_Comm_create(MPI_COMM_WORLD, first_row_group,&first_row_comm);
```

Этот фрагмент демонстрирует способ построения нового коммуникатора. Сначала создается список процессов для нового коммуникатора. Затем создается группа, состоящая из этих процессов. На это необходимо три команды: получение группы, связанной с **MPI_COMM_WORLD**; создание группы – **MPI_Group_incl**; создание коммуникатора – **MPI_Comm_create**. Результатом является коммуникатор первой строки **comm**. Теперь процессы в первой строке **comm** могут совершать коллективные операции связи. Например, процесс 0 (в группе первой строки) может передавать A_{00} другим процессам своей группы. Фрагмент кода программы для выполнения коллективной операции представлен ниже.

```
int my_rank_in_first_row;
float* A_00;
                        /* my_rank есть номер процесса в  MPI_GROUP_WORLD */
if (my_rank < q)
{
    MPI_Comm_rank( first_row_comm, &my_rank_in_first_row);
                        /* Выделяем память для A_00, order = n_bar */
    A_00 = (float*) malloc (n_bar*n_bar*sizeof(float));
    if (my_rank_in_first_row == 0)
        {  … }          /* Инициализация A_00 */
    MPI_Bcast(A_00, n_bar*n_bar, MPI_FLOAT, 0, first_row_comm);
}
```

В программе умножения матриц необходимо создать несколько коммуникаторов: по одному для каждой строки процессов и по одному для каждого столбца. Это чрезвычайно трудоемко, если количество процессов большое. Используя функцию **MPI_Comm_split**, которая может создавать несколько коммуникаторов одновременно, можно упростить решение задачи. В качестве примера использования этой функции создадим один коммуникатор для каждой строки процессов.

```
MPI_Comm my_row_comm;
int my_row;
          /* my_rank есть номер процесса в  MPI_COMM_WORLD.  q*q = p */
my_row = my_rank/q;
MPI_Comm_split( MPI_COMM_WORLD, my_row, my_rank, &my_row_comm);
```

Единственный вызов **MPI_Comm_split** создает **q** новых коммуникаторов, имеющих одно и то же имя **my_row_comm**.

Способ 2. В клеточном алгоритме удобно идентифицировать процессы в **MPI_COMM_WORLD** координатами квадратной сетки, то есть для каждой строки и каждого столбца сетки следует сформировать собственный коммуникатор. Для этого необходимо связать квадратную сетку с **MPI_COMM_WORLD**. Чтобы сделать это, нужно определить:

1. Число размерностей сетки (равно двум).
2. Размер каждого измерения (q строк и q столбцов).
3. Периодичность каждого измерения. Эта информация определяет, является ли первая входящая величина в каждой строке или столбце смежной с последней входящей величиной в той же строке или столбце. Так как по алгоритму необходимо циклическое смеще-

ние подматриц в каждом столбце, то периодичность второго измерения нужна.

4. MPI дает пользователю возможность выбрать систему так, чтобы оптимизировать наложение сетки процессов на структуру физических процессоров, обеспечивая в том числе и возможность переупорядочивания процессов в группе, лежащей в основе коммуникатора. Из этого следует, что не обязательно сохранять порядок процессов в **MPI_COMM_WORLD**, нужно допустить, что система может быть переупорядочена.

Чтобы создать новый коммуникатор, необходимо прежде всего вызвать функцию **MPI_Cart_create** с соответствующими параметрами. Это показано в следующм фрагменте программы.

```
MPI_Comm grid_comm;
int dimensions[2];
int wrap_around[2];
int reorder = 1;
dimensions[0] = dimensions[1] = q;
wrap_around[0] = wrap_around[1] = 1;
MPI_Cart_create(MPI_COMM_WORLD,2,dimensions,wrap_around,reorder,
                grid_comm);
```

После выполнения этого фрагмента сеточный коммуникатор **comm** будет содержать все процессы из **MPI_COMM_WORLD** (возможно, переупорядоченные) и иметь двумерную декартову связанную систему координат. Чтобы определить координаты процесса, необходим вызов функций – **MPI_Comm_rank** и **MPI_Cart_coords**.

```
MPI_Cart coords:
int coordinates[2], my_grid_rank;
MPI_Comm_rank(grid_comm, &my_grid_rank);
MPI_Cart_coords(grid_comm, my_grid_rank, 2,coordinates);
```

MPI_Cart_rank возвращает номер в декартовом коммуникаторе процесса с декартовыми координатами. **MPI_Cart_coords** является обратной функцией по отношению к **MPI_Cart_rank**: она возвращает координаты процесса с номером rank в декартовом коммуникаторе **comm**.

Можно разбить сетку на коммуникаторы более низкого порядка. Чтобы создать коммуникатор для каждой строки сетки, можно воспользоваться функцией **MPI_Cart_sub** с соответствующими параметрами, что и показано ниже.

```
int varying_coords[2];
MPI_Comm row_comm;
varying_coords[0] = 0; varying_coords[1] = 1;
MPI_Cart_sub(grid_comm, varying_coords, row_comm);
```

Вызов **MPI_Cart_sub** создает q новых коммуникаторов. Переменная **Varying_coords** есть массив булевых элементов. Она определяет, принадлежит ли каждое измерение новому коммуникатору. Так как создаются коммуникаторы для строк сетки, каждый новый коммуникатор состоит из процессов, полученных при фиксировании номера строки и изменения номеров столбцов. В каждом процессе новый коммуникатор возвращается в **row_comm**. Чтобы создать коммуникаторы для столбцов, значения для **varying_coords** подаются в другом порядке. Вызов функции тогда будет таким.

```
MPI_Comm col_comm;
varying_coords[0] = 1; varying_coords[1] = 0;
MPI_Cart_sub(grid_comm, varying_coord, col_comm);
```

Заметим схожесть функций **MPI_Cart_sub** и **MPI_Comm_split**. Они выполняют похожие действия, то есть разбивают коммуникатор на несколько новых. Однако **MPI_Cart_sub** может использоваться с коммуникатором, который имеет связанную декартову топологию, и новые коммуникаторы могут быть созданы только в том случае, когда происходит фиксирование (или изменение) одного или больше размерностей старого коммуникатора.

7.2.3. Параллельная программа для клеточного алгоритма

Выберем 2-й способ создания коммуникаторов для решения задачи умножения матриц. Опишем функцию, которая создает различные коммуникаторы и связанную с ними информацию. Так как это требует большого количества переменных и эта информация будет использоваться в других функциях, введем следующую структуру:

```
typedef struct {
int p;                  /* общее количество процессов */
MPI_Comm comm;          /* коммуникатор сетки */
MPI_Comm row_comm;      /* коммуникатор строки */
MPI_Comm col_comm;      /* коммуникатор столбца */
int q;                  /* порядок сетки */
int my_row;             /* номер строки */
int my_col;             /* номер столбца */
```

```
int my_rank;                    /* номер в коммуникаторе сетки */
} GRID_INFO_TYPE;
```

Тогда подпрограмма создания коммуникаторов для решения задачи будет иметь следующий код:

```
void Setup_grid(GRID_INFO_TYPE* grid)
{   int old_rank, dimensions[2], periods[2], coordinates[2], varying_coords[2];
                                /* определение  параметров сетки */
  MPI_Comm_size(MPI_COMM_WORLD, &(grid->p));
  MPI_Comm_rank(MPI_COMM_WORLD, &old_rank);
  grid->q = (int) sqrt((double) grid->p);
  dimensions[0] = dimensions[1] = grid->q;
  periods[0] = periods[1] = 1;
          /*создание коммуникатора с картезианской  топологии размерности 2 */
  MPI_Cart_create(MPI_COMM_WORLD, 2, dimensions, periods,1, &(grid->comm));
  MPI_Comm_rank(grid->comm, &(grid->my_rank));
  MPI_Cart_coords(grid->comm, grid->my_rank, 2,coordinates);
  grid->my_row = coordinates[0];       grid->my_col = coordinates[1];
                          /* создание коммуникатора строки */
  varying_coords[0] = 0; varying_coords[1] = 1;
  MPI_Cart_sub(grid->comm, varying_coords,&(grid->row_comm));
                          /* создание коммуникатора столбца */
  varying_coords[0] = 1; varying_coords[1] = 0;
 MPI_Cart_sub(grid->comm, varying_coords,&(grid->col_comm));
}
```

Ниже представлен текст обобщающей программы клеточного умножения матриц. **LOCAL_MATRIX_TYPE –** тип элементов умножаемых матриц, **DERIVED_LOCAL_MATRIX** – тип перемножаемых матриц**.**

```
void Mult_Matr (int n, GRID_INFO_TYPE* grid,
          LOCAL_MATRIX_TYPE* local_A, local_B,  local_C)
{  LOCAL_MATRIX_TYPE* temp_A;        /* порядок  подматриц = n/q */
  int step, bcast_root, n_bar; int source, dest, tag = 43;
  MPI_Status status;
  n_bar = n/grid->q;
  Set_to_zero(local_C);      /*  обнуление элементов результирующей матрицы */
                             /* вычисление адресов для циклического смещения B */
  source = (grid->my_row + 1) % grid->q;
  dest = (grid->my_row + grid->q –  1) % grid->q;
                            /* выделение памяти для пересылки блоков A */
  temp_A = Local_matrix_allocate(n_bar);
```

```
 for (step = 0; step < grid->q; step++)
{
     bcast_root = (grid->my_row + step) % grid->q;
     if (bcast_root == grid->my_col)  {
        MPI_Bcast(local_A,1,DERIVED_LOCAL_MATRIX,bcast_root,
                    grid->row_comm);
                        /* умножение подматриц */
        Local_matrix_multiply(local_A, local_B,local_C);
     }
     else {
        MPI_Bcast(temp_A,1,DERIVED_LOCAL_MATRIX,bcast_root,
                    grid->row_comm);
                        /* умножение подматриц */
        Local_matrix_multiply(temp_A, local_B, local_C);
     }                  /*  пересылка подматриц B*/
     MPI_Send(local_B, 1, DERIVED_LOCAL_MATRIX, dest, tag,
                grid->col_comm);
     MPI_Recv(local_B,1,DERIVED_LOCAL_MATRIX,source,tag,
                grid->col_comm, &status);
   }
}
```

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ К ГЛАВЕ 7

Контрольные вопросы к 7.1

1. Объясните понятие “самопланирующий алгоритм”.
2. Можно ли в главном процессе программы умножения матрицы на вектор заменить коллективную операцию MPI_Bcast на обычный межпроцессный обмен MPI_Send/MPI_Recv? Как изменится код подчиненного процесса?
3. Какую функцию выполняют параметры status, tag при передаче данных в программе умножения матрицы на вектор?
4. Как изменится код программы умножения матрицы на вектор, если необходимо умножить вектор на матрицу?
5. Можно ли в программе умножения матриц использовать другие виды коммуникаций?

Контрольные вопросы к 7.2

1. В чем преимущества и недостатки двух способов создания коммуникаторов для алгоритма клеточного умножения?
2. Как определить тип DERIVED_LOCAL_MATRIX средствами MPI?
3. Чему равны значения source, tag в операциях обмена процедуры Mult_Matr?
4. Каково минимальное количество процессов для клеточного алгоритма?
5. Проведите анализ эффективности для алгоритма клеточного умножения.

Задания для самостоятельной работы

7.1. Напишите программу, реализующую самопланирующий алгоритм умножения матрицы на вектор. Исследуйте зависимость ускорения от размеров вектора и матрицы и разных способов коммуникаций.

7.2. Напишите программу, реализующую самопланирующий алгоритм умножения матриц. Исследуйте зависимость ускорения от размеров умножаемых матриц и разных способов коммуникаций.

7.3. Напишите программу, реализующую алгоритм умножения матриц, при котором происходит равномерное распределение частей матрицы A по процессам, а затем независимо вычисляются части результирующей матрицы. Сравните эффективность этого и предыдущего алгоритмов.

7.4. Напишите программу, реализующую клеточный алгоритм умножения матриц.

7.5. Напишите программу, реализующую клеточный алгоритм умножения матриц, используя первый способ создания коммуникаторов, описанный в параграфе 7.2.

Глава 8. РЕШЕНИЕ ДИФФЕРЕНЦИАЛЬНЫХ УРАВНЕНИЙ В ЧАСТНЫХ ПРОИЗВОДНЫХ

Решение дифференциальных уравнений в частных производных является ядром многих приложений. Решение задачи на конечно-разностной вычислительной сетке методом Якоби, рассмотренное в этом параграфе, дает возможность продемонстрировать использование «виртуальной топологии», которая делает более удобным размещение процессов на вычислительной среде, а также позволяет исследовать эффективность различных способов коммуникаций для решения одной и той же задачи. Метод Якоби был выбран благодаря простоте вычислительной части [4, 21].

8.1. ЗАДАЧА ПУАССОНА

Задача Пуассона, то есть решение дифференциальных уравнений в частных производных, выражается следующими уравнениями:

$$\Delta^2 u = f(x,y) \quad \text{– внутри области} \qquad (8.1)$$

$$u(x,y) = k(x,y) \quad \text{– на границе} \qquad (8.2)$$

Будем считать, что область решения является квадратной. Чтобы найти приближенное решение, определим квадратную сетку, содержащую точки (x_i, y_i), задаваемые как

$$x_i = \frac{i}{n+1}, i = 0,...,n+1\ , \qquad y_j = \frac{j}{n+1}, j = 0,...,n+1\ .$$

Таким образом, вдоль каждого края сетки имеется n+2 точки. Следует найти аппроксимацию для u(x,y) в точках (x_i, y_j) выбранной сетки. Обозначим через $u_{i,j}$ значения u в (x_i, y_j), через h расстояние между точками, равное 1/(n+1). Тогда формула 8.1. для каждой из точек будет выглядеть следующим образом:

$$\frac{u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - 4u_{i,j}}{h^2} = f_{i,j}. \qquad (8.3)$$

Вычисляем значения $u_{i,j}$ в каждой точке сетки, которые замещают предыдущие значения, используя выражение

$$u_{i,j}^{k+1} = (u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} + u_{i=1,j}^{k} - h^2 f_{i,j}) / 4.$$

Этот процесс называется итерациями Якоби и повторяется до получения решения. Фрагмент программы для этого случая таков:

```
   integer i, j, n
   double precision, u (0:n+l, 0:n+l), unew(0:n+l, 0:n+l)
   do 10 j = l, n
        do 10 i = l, n
           unew (i, j) =   0.25*(u(i-l, j) + u(i, j+l) + u(i, j-l)+u(i+l, j)) – h * h * f(i, j)
10      continue
```

8.2. ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ ДЛЯ МЕТОДА ИТЕРАЦИЙ ЯКОБИ

8.2.1. Параллельный алгоритм для 1D композиции

Чтобы распараллелить последовательный алгоритм, нужно параллелизовать цикл. Для этого необходимо распределить данные (в нашем случае массивы u, unew, f) по процессам. Одна из простейших декомпозиций состоит в следующем: физическая область разделяется на слои, каждый из них обрабатывается отдельным процессом. Эта декомпозиция может быть описана следующим образом:

```
  integer i, j, n, s, e
  double precision u(0:n+l, s:e), unew(0:n+l, s:e)
  do 10 j = s, e
      do 10 i = l, n
          unew(i, j) = 0.25*(u(i-l, j)+ u(i, j+l)+ u(i, j-l) +  u(i+l,j)) –  h * h * f(i,j)
10    continue
```

Здесь s,e указывают значения номеров строк слоя, за которые ответственен данный процесс.

При выполнении итераций каждый процесс использует не только элементы, принадлежащие его слою, но и элементы из смежных процессов. В этом случае организовать вычисления можно следующим образом. Расширим слой сетки для каждого процесса так, чтобы он содержал необходимые данные (рис. 8.1). Элементы среды, которые используются, чтобы сохранять данные из других процессов, называются “теневыми”.

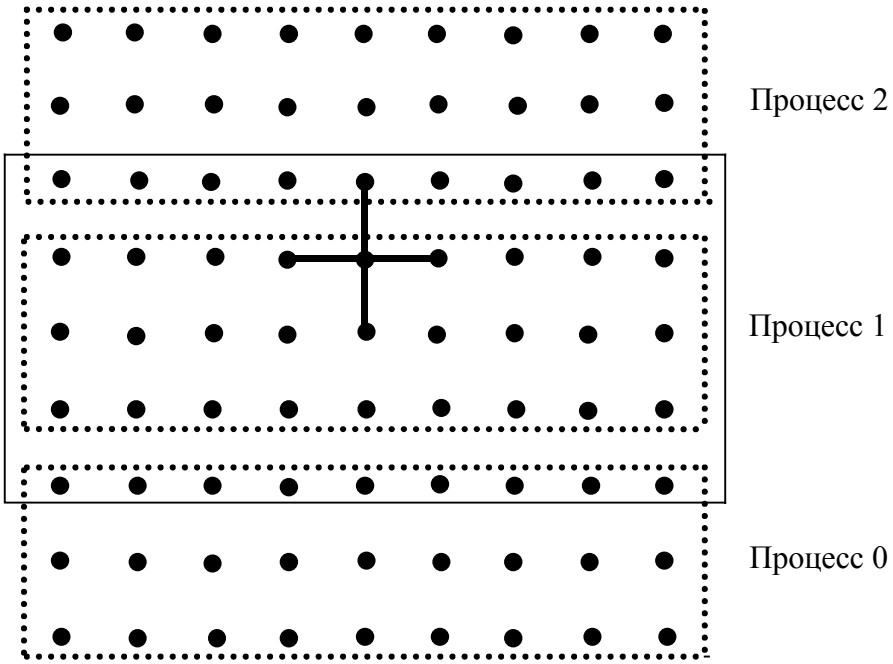


Рис. 8.1. Область с теневыми точками для процесса 1

Вследствие этого размерность массивов изменится:

double precision u (0:n+1, s-1: e+1).

Следующая задача – решить, как назначать процессы разделенным областям массива. В качестве декомпозиции процессов для решения задачи выберем картезианскую топологию размерности 1. MPI имеет набор процедур для определения, исследования и манипулирования картезианскими топологиями.

При одном измерении можно просто использовать номер в новом коммуникаторе плюс или минус 1, чтобы найти соседа и не использовать фунцию **MPI_CART_CREATE**, но даже в этом случае выбор может быть не наилучшим, поскольку соседи, определенные таким

образом, могут не быть соседями по аппаратуре. Если размерностей больше, чем одна, задача становится еще труднее.

Каждый процесс посылает и получает сообщения от соседей. В декомпозиции 1D это соседи, расположенные выше и ниже. Способ определения соседей заключается в следующем: копия верхней строки одного процесса является дном теневой строки процесса, который расположен выше. Следовательно, каждый процесс является как принимающим, так и посылающим данные. Фактически данные сдвигаются от одного процесса к другому, и MPI имеет процедуру **MPI_CART_SHIFT**, которая может быть использована, чтобы найти соседей. Опишем процедуру **MPE_DECOMP1D**, которая определяет декомпозицию массивов и вызывается функцией

call MPE_DECOMP1D(n, nprocs, myrank, s, e),

где **nprocs** – число процессов в картезианской топологии, **myrank** – координаты процесса, **n** – размерность массива. **MPE_Decomp1d** вычисляет значения **s** и **e.** Например следующим образом.

Если n делится на nprocs без остатка, то

s = 1 + myrank * (n/nprocs)
e = s + (n/nprocs) – 1.

Когда деление без остатка не получается, если floor(x) – целая часть x плюс 1, то наиболее простой выход:

```
s = 1 + myrank * floor (n/nprocs)
if  (myrank  .eq. nprocs – 1)    then  e = n
 else  e = s + floor (n/nprocs) – 1
endif.
```

Для каждого процесса мы должны получить теневые данные для строки с номером **s–1** от процесса ниже и данные для строки с номером **e+1** от процесса выше. Ниже представлена процедура обмена данными, для которой определены следующие параметры: **a** – массив, **n*x*** – количество пересылаемых данных строки, **s** – номер первой строки массива в данном процессе, **e** - номер последней строки массива данного процесса, **nbrbottom** – номер процесса, расположенного ниже данного, а **nbrtop** – номер процесса, расположенного выше данного.

```
subroutine EXCHG1(a, nx, s, e, comm1d, nbrbottom, nbrtop )
  use mpi
  integer nx, s, e
  double precision a(0:nx+l, s-l:e+l)
```

```
  integer comm1d, nbrbottom, nbrtop, status (MPI_STATUS_SIZE), ierr
  call MPI_SEND( a(l,e), nx, MPI_DOUBLE_PRECISION,  nbrtop, 0,  comm1d, ierr )
  call MPI_RECV( a(l,s-l), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0,
                    comm1d, status, ierr)
  call MPI_SEND( a(l,s),nx, MPI_DOUBLE_PRECISION,nbrbottom, 1,comm1d, ierr )
  call MPI_RECV( a(l,e+l), nx, MPI_DOUBLE_PRECISION, nbrtop, 1,
                    comm1d, status, ierr )
  return
end
```

В этой процедуре каждый процесс посылает данные процессу **nbrtop,** расположенному выше его, и затем принимает данные от процесса **nbrbottom** ниже его. Затем порядок меняется на обратный - данные посылаются процессу ниже и принимаются от процесса, выше данного.

Опишем все части программы для решения задачи Пуассона. В основной части программы для назначения процессов разделенным областям массива будем использовать картезианскую топологию единичной размерности. Используем процедуру **MPI_CART_CREATE** для создания декомпозиции процессов и процедуру **MPE_Decomp1d**, чтобы определить декомпозицию массива. Процедура **ONEDINIT** инициализирует элементы массивов **a**, **f**. Решение вычисляется попеременно в массивах **a** и **b**, так как в цикле имеется два обращения к **EXCHNG1** и **SWEEP1D**. Итерации заканчиваются, когда разница между двумя смежными значениями аппроксимаций становится меньше, чем 10^{-5}. Разность между двумя локальными частями **a** и **b** вычисляется процедурой **DIFF**. Процедура **MPI_ALLREDUCE** нужна, чтобы гарантировать, что во всех процессах достигнута точность вычислений. Программа печатает как результаты итераций, так и конечный результат. Цикл **DO** с максимумом итераций **maxit** гарантирует, что программа закончится, даже если итерации не сходятся.

Вычислительная часть программы представлена на рис. 8.2.

```
!  определение нового коммуникатора картезианской топологии
!  размерности 1   для декомпозиции процессов
     call MPI_CART_CREATE(MPI_COMM_WORLD,1,numprocs, .false., .true,
                                    comm1d, ierr )
!  определение позиции процесса в данном коммуникаторе
     call MPI_COMM_RANK( comm1d, myid, ierr )
!  определение номеров процессов для заполнения теневых точек
     call MPI_CART_SHIFT( comm1d, 0, 1, nbrbottom, nbrtop, ierr)
!  определение декомпозиции исходного массива
     call MPE_DECOMP1D( ny, numprocs, myid, s, e )
```

```
!   инициализация  массивов f и a
    call ONEDINIT( a, b, f, nx, s, e )
!  Вычисление итераций Якоби ,  maxit – максимальное число итераций
    do 10 it=l, maxit
!   определение теневых точек
        call EXCHNG1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
!   вычисляем одну итерацию Jacobi
        call SWEEP1D( a, f, nx, s, e, b )
!     повторяем, чтобы получить решение в a
        call EXCHNG1( b, nx, s, e, comm1d, nbrbottom, nbrtop )
        call SWEEP1D( b, f, nx, s, e, a )
!   проверка точности вычислений
        diffw = DIFF( a, b, nx, s, e )
        call MPI_ALLREDUCE( diffw, diffnorm, 1,
                                  MPI_DOUBLE_PRECISION,MPI_SUM, comm1d, ierr )
        if (diffnorm .It.  1.Oe-5) goto 20
        if (myid .eq. 0)  print *, 2*it, ' Difference is ', diffnorm
10  continue
    if (myid .eq. 0) print *, 'Failed to converge'
20  continue
    if (myid .eq. 0) then   print *, 'Converged after ', 2*it, ' Iterations'
    endif
```

Рис. 8.2. Реализация итераций Якоби для решения задачи Пуассона

8.2.2. Параллельный алгоритм для 2D композиции

Небольшая модификация программы преобразует ее из одномерной в двухмерную. Прежде всего следует определить декомпозицию двумерной среды, используя функцию **MPI_CART_CREATE.**

```
isperiodic(l)  = .false.
isperiodic(2)  = .false.
reorder        = .true.
call MPI_CART_CREATE(MPI_COMM_WORLD, 2, dims, isperiodic,
                                reorder, comm2d, ierr )
```

Для получения номеров левых **nbrleft** и правых **nbrright** процессов-соседей, так же как и верхних и нижних в предыдущем параграфе, воспользуемся функцией **MPI_CART_SHIFT**.

```
call MPI_CART_SHIFT(comm2d, 0, 1, nbrleft, nbrright, ierr )
call MPI_CART_SHIFT(comm2d, 1, 1, nbrbottom, nbrtop, ierr )
```

Процедура **SWEEP** для вычисления новых значений массива **unew** изменится следующим образом:

```
  integer i, j, n
  double precision u(sx-l:ex+l, sy-l:ey+l), unew(sx-l:ex+l, sy-l:ey+l)
  do 10 j=sy, ey
     do 10 i=sx, ex
        unew(i,j)  = 0.25*(u(i-l,j) + ui(i,j+l) + u(i,j-l) +  u(i+l,j)) –   h * h * f(i,j)
10    continue
```

Последняя процедура, которую нужно изменить, – процедура обмена данными (**EXCHNG1** в случае 1D композиции). Возникает проблема: данные, посылаемые к верхним и нижним процессам, хранятся в непрерывной памяти, а данные для левого и правого процессов – нет.

Если сообщения занимают непрерывную область памяти, то такие типы данных, как **MPI_INTEGER** и **MPI_DOUBLE_PRECISION,** используются при передаче сообщений. В других случаях необходимы наследуемые типы данных. Определим новый тип данных, описывающий группу элементов, адрес которых отличается на некоторую постоянную величину (страйд).

```
call MPI_TYPE_VECTOR ( ey – sy + 3, 1, ex – sx + 3,
                                  MPI_DOUBLE_PRECISION, stridetype, ierr )
call MPI_TYPE_COMMIT( stridetype, ierr )
```

Аргументы **MPI_TYPE_VECTOR** описывают блок, который состоит из нескольких копий входного типа данных. Первый аргумент – это число блоков; второй – число элементов старого типа в каждом блоке (часто это один элемент); третий аргумент есть страйд (дистанция в терминах входного типа данных между последовательными элементами); четвертый аргумент – это старый тип; пятый аргумент есть создаваемый наследуемый тип.

После того как с помощью функции **MPI_TYPE_VECTOR** создан новый тип данных**,** он передается системе с помощью функции **MPI_TYPE_COMMIT**. Эта процедура получает новый тип данных и дает системе возможность выполнить любую желаемую возможную оптимизацию. Все сконструированные пользователем типы данных должны быть переданы до их использования в операциях обмена. Когда тип данных больше не нужен, он должен быть освобожден процедурой **MPI_TYPE_FREE**. После определения новых типов данных программа для передачи строки отличается от программы для переда-

чи столбца только в аргументах типа данных. Конечная версия процедуры обмена **EXCHNG2** представлена ниже. В ней определены следующие параметры: **a** – массив, **n*x*** – количество пересылаемых данных строки, **sx** – номер первой строки массива в данном процессе, **ex** – номер последней строки массива данного процесса, **sy** – номер первого столбца массива в данном процессе, **ey** – номер последнего столбца массива данного процесса, **nbrbottom** – номер процесса-соседа выше , **nbrtop** – номер процесса-соседа ниже, **nbrleft** – номер процеса-соседа слева, **nbrright** – номер процеса-соседа справа, **stridetype –** новый тип данных для элементов столбца.

```
subroutine EXCHNG2 (a, sx, ex, sy, ey, comm2d, stridetype, nbrleft,
                          nbrright, nbrtop, nbrbottom )
 use mpi
 integer sx, ex, sy, ey, stridetype, nbrleft, nbrright, nbrtop, nbrbottom, comm2d
 double precision a(sx-l:ex+l,  sy-l:ey+l)
 integer status (MPI_STATUS_SIZE), ierr, nx
 nx = ex –  sx + I
 call MPI_SENDRECV( a(sx,ey),  nx,  MPI_DOUBLE_PRECISION,  nbrtop,0,
                           a(sx, sy-l), nx, MPI_DOUBLE_PRECISION, nbrbottom,
                           0, comm2d, status, ierr )
 call MPI_SENDRECV( a(sx,sy), nx, MPI_DOUBLE_PRECISION,  brbottom,1,
                           a(sx,ey+l), nx, MPI_DOUBLE_PRECISION, nbrtop, 1,
                           comm2d, status, ierr )
 call MPI_SENDRECV(a(ex,sy), 1, stridetype, nbrright, 0, a(sx-l,sy),
                          1, stridetype, nbrleft, 0,comm2d, status, ierr )
 call MPI_SENDRECV( a(sx,sy), 1, stridetype, nbrieft, 1, a(ex+l,sy),
                           1, stridetype,nbrright, 1, comm2d, status, ierr )
 return
end
```

8.2.3. Способы межпроцессного обмена

В процедуре **EXCHG1** применен наиболее часто используемый вид обмена. Параллелизм зависит от величины буфера, обеспечиваемого системой передачи сообщений. Если запустить задачу на системе с малым объемом буферного пространства и с большим размером сообщений, то посылки не завершатся до тех пор, пока принимающие процессы не закончат прием. В процедуре **EXCHG1** только один последний процесс не посылает сообщений на первом шаге, следовательно, он может получить сообщения от процесса выше, затем происходит прием в предыдущем процессе и так далее. Эта лестничная процедура посылок и приема представлена на рис. 8.3.

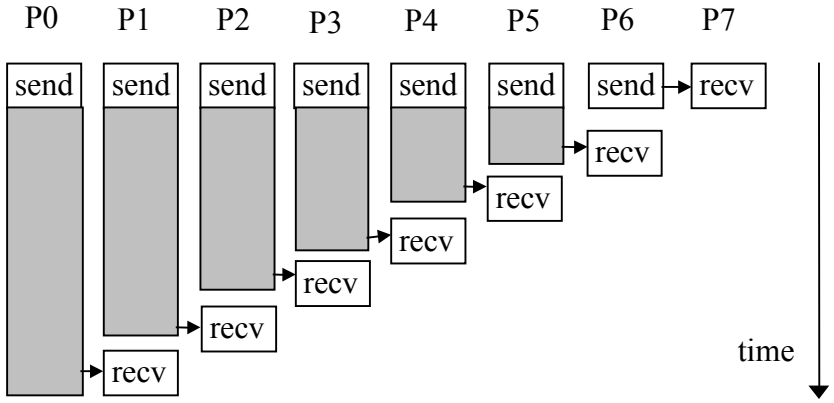


Рис. 8.3. Порядок посылок и соответствующих приемов во времени.

Длина заштрихованной области указывает время, когда процесс ожидает возможности передать данные соседнему процессу.

Приведем несколько вариантов реализации процедуры EXCHG1, которые могут быть более эффективными для различных MPI реализаций.

Упорядоченный send и reseive. Простейший путь скорректировать зависимости из-за буферизации состоит в упорядочении посылок и приемов: если один из процессов посылает другому, то принимающий процесс выполнит у себя сначала прием, а потом только начнет свою посылку, если она у него есть. Соответствующая программа представлена ниже.

```
subroutine exchng1 a, nx, s, e, comm1d, nbrbottom, nbrtop)
use mpi
integer nx, s, e,  comm1d, nbrbottom, nbrtop, rank, coord
double precision a(0:nx+l, s-l:e+l)
integer status (MPI_STATUS_SIZE), ierr
call MPI_COMM_RANK( comm1d, rank, ierr )
call MPI_CART_COORDS( comm1d, rank, 1, coord, ierr )
if (mod( coord, 2 ) .eq. 0) then
   call MPI_SEND( a(l,e), nx, MPI_DOUBLE_PRECISION,nbrtop, 0,
                        comm1d, ierr )
   call MPI_RECV( a(l, s-l), nx, MPI_DOUBLE_PRECISION,  nbrbottom,
                        0, comm1d,status, ierr )
   call MPI_SEND( a(l,s), nx, MPI_DOUBLE_PRECISION,  nbrbottom,
                       1, comm1d, ierr)
```

```
    call MPI_RECV( a(l, e+l), nx, MPI_DOUBLE_PRECISION,  nbrtop, 1,
                          comm1d, status, ierr )
  else
    call MPI_RECV ( a(l, s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom,
                           0,comm1d,status,ierr)
    call MPI_SEND( a(l, e), nx, MPI_DOUBLE_PRECISION,  nbrtop, 0,
                          comm1d, ierr )
    call MPI_RECV( a(l, e+l), nx, MPI_DOUBLE_PRECISION,  nbrtop, 1,
                         comm1d, status, ierr )
    call MPI_SEND( a(l, s), nx, MPI_DOUBLE_PRECISION, nbrbottom,
                        1,comm1d, ierr)
  endif
  return
end
```

В этой программе четные процессы посылают первыми, а нечетные процессы принимают первыми.

Комбинированные send и receive. Спаривание посылки и приема эффективно, но может быть трудным в реализации при сложном взаимодействии процессов (например, на нерегулярной сетке). Альтернативой является использование процедуры **MPI_SENDRECV**. Эта процедура позволяет послать и принять данные, не заботясь о том, что может иметь место дедлок из-за нехватки буферного пространства. В этом случае каждый процесс посылает данные процессу, расположенному выше, и принимает данные от процесса, расположенного ниже, как это показано в следующей программе:

```
subroutine exchng1 ( a, nx, s, e, comm1d, nbrbottom, nbrtop )
use mpi
integer nx, s, e,  comm1d, nbrbottom, nbrtop
double precision a(0:nx+l, s-l:e+l)
integer status (MPI_STATUS_SIZE), ierr
call MPI_SENDRECV( a(l,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0,
   a(l,s-l), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, comm1d, status, ierr )
call MPI_SENDRECV( a(l,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, I,
     a(l,e+l), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, comm1d, status, ierr )
return
end
```

Буферизованные Sends. MPI позволяет программисту зарезервировать буфер, в котором данные могут быть размещены до их доставки по назначению. Это снимает с программиста ответственность за

безопасное упорядочивание операций посылки и приема. Изменение в обменных процедурах будет простым – вызов **MPI_SEND** замещается на вызов **MPI_BSEND:**

```
subroutine exchng1 (a, nx, s, e, comm1d, nbrbottom, nbrtop )
use mpi
integer nx, s, e, integer coimn1d, nbrbottom, nbrtop
double precision a(0:nx+l, s-l:e+l)
integer status (MPI_STATUS_SIZE), ierr
call MPI_BSEND( a(l,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0,
                      comm1d, ierr )
call MPI_RECV( a(l, s-l), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0,
                     comm1d,status,ierr )
call MPI_BSEND( a(l,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1,
                      comm1d, ierr )
call MPI_RECV( a(l,e+l), nx, MPI_DOUBLE_PRECISION, nbrtop,  1, commid,
                    status, ierr )
return
end
```

В дополнение к изменению обменных процедур MPI требует, чтобы программист зарезервировал память, в которой сообщения могут быть размещены с помощью процедуры **MPI_BUFFER_ATTACH**. Этот буфер должен быть достаточно большим, чтобы хранить все сообщения, которые обязаны быть посланными перед тем, как будут выполнены соответствующие вызовы для приема. В нашем случае нужен буфер размером 2*nx значений двойной точности. Это можно сделать с помощью нижеприведенного отрезка программы (8 – число байтов в значениях двойной точности).

```
double precision buffer(2*MAXNX+2*MPI_BSEND_OVERHEAD)
call MPI_BUFFER_ATTACH( buffer,
                                MAXNX*8+2*MPI_BSEND_OVERHEAD*8, ierr )
```

Заметим, что дополнительное пространство размещено в буфере. Для каждого сообщения, посланного при помощи **MPI_BSEND**, должен быть зарезервирован дополнительный объем памяти размером **MPI_BSEND_OVERHEAD** байтов. Реализация MPI использует это внутри процедуры **MPI_BSEND**, чтобы управлять буферной областью и коммуникациями. Когда программа больше не нуждается в буфере, необходимо вызвать процедуру **MPI_BUFFER_DETACH**.

Имеется следующая особенность использования процедуры **MPI_BSEND**. Может показаться, что цикл, подобный следующему, способен посылать 100 байтов на каждой итерации:

```
    size = 100 + MPI_BSEND_OVERHEAD
    call MPI_BUFFER_ATTACH(buf, size, ierr )
    do 10 i=l, n
       call MPI_BSEND(sbuf,100,MPI_BYTE,0, dest, MPI_COMM_WORLD,  ierr )
          . . . other work
10  continue
    call MPI_BUFFER_DETACH( buf, size, ierr )
```

Здесь проблема состоит в том, что сообщение, посланное в *i*-й итерации, может быть не доставлено, когда следующее обращение к **MPI_BSEND** имеет место на *i*+1-й итерации. Чтобы в этом случае правильно использовать **MPI_BSEND**, необходимо, чтобы либо буфер, описанный с помощью **MPI_BUFFER_DETACH**, был достаточно велик, чтобы хранить все посланные данные, либо **buffer attach** и **detach** должны быть перемещены внутрь цикла.

Неблокирующий обмен. Для большинства параллельных процессоров обмен данными между процессами требует больше времени, чем внутри одиночного процесса. Чтобы избежать падения эффективности, во многих параллельных процессорах используется совмещение вычислений с одновременным обменом данными. В этом случае должны использоваться неблокирующие приемопередачи.

Процедура **MPI_ISEND** начинает неблокирующую передачу. Процедура аналогична **MPI_SEND** за исключением того, что для **MPI_ISEND** буфер, содержащий сообщение, не должен модифицироваться до тех пор, пока сообщение не будет доставлено. Самый легкий путь проверки завершения операции состоит в использовании операции **MPI_TEST**:

```
    call MPI_ISEND( buffer, count, datatype, dest, tag, cornm, request, ierr )
    < do other work >
10  call MPI_TEST ( request, flag, status, ierr )
    if (.not. flag) goto 10
```

Часто желательно подождать до окончания передачи. Тогда вместо того, чтобы писать цикл, как в предыдущем примере, можно использовать **MPI_WAIT**:

```
     call MPI_WAIT ( request, status, ierr )
```

Когда неблокирующая операция завершена (то есть **MPI_WAIT** или **MPI_TEST** возвращают результат с **flag=.true.),** устанавливается значение **MPI_REQUEST_NULL**.

Процедура **MPI_IRECV** начинает неблокирующую операцию приема. Точно так же как для **MPI_ISEND**, функция **MPI_TEST** может быть использована для проверки завершения операции приема, начатой **MPI_IRECV**, а функция **MPI_WAIT** может быть использована для ожидания завершения такого приема. MPI обеспечивает способ ожидания окончания всех или некоторой комбинации неблокирующих операций (функции **MPI_WAITALL** и **MPI_WAITANY**). Процедура **EXCHNG1** с использованием неблокирующего обмена будет выглядеть следующим образом:

```
subroutiтe exchng1 ( a, nx, s, e, comm1d, nbrbottom, nbrtop )
use mpi
integer nx, s, e,  comm1d, nbrbottom, nbrtop
double precision a(0:nx+l, s-l:e+l)
integer status_array(MPI_STATUS_SIZE,4), ierr, req(4)
call MPI_IRECV ( a(l, s-l), nx, MPI_DOUBLE_PRECISION,
                  nbrbottom,0, comm1d, req(l), ierr )
call MPI_IRECV ( a(l, e+l), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, comm1d,
                  req(2), ierr )
call MPI_ISEND (  a(l, e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, comm1d,
                  req(3), ierr )
call MPI_ISEND ( a(l, s), nx, MPI_DOUBLE_PRECISION, nbrbottom, comm1d,
                  req(4), ierr )
MPI_WAITALL ( 4, req, status_array, ierr )
return
end
```

Этот подход позволяет выполнять в одно и то же время как передачу, так и прием. При этом возможно двойное ускорение по сравнению с вариантом на рис. 8.4, хотя немногие существующие системы допускают это. Чтобы совмещать вычисления с обменом, необходимы некоторые изменения в программах. В частности, необходимо изменить процедуру **SWEEP** так, чтобы можно было совершать некоторую работу, пока ожидается приход данных.

Синхронизируемые Sends. Что надо сделать, чтобы гарантировать, что программа не зависит от буферизации? Общего ответа на этот вопрос нет, но во многих специальных случаях можно показать, что, если программа работает успешно без буферизации, она будет выполняться и с некоторым размером буферов. MPI обеспечивает

способ послать сообщение, при котором ответ не приходит до тех пор, пока приемник не начнет прием сообщения. Процедура называется **MPI_SSEND**. Программы, не требующие буферизации, называют «экономными».

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ К ГЛАВЕ 8

Контрольные вопросы к 8.2

1. Почему для 1D декомпозиции процессов использование картезианской топологии предпочтительнее, чем использование стандартного коммуникатора MPI_COMM_WORLD?
2. Что такое “теневая ” точка? Зачем необходимо введение “теневых” точек для решения задачи Пуассона методом итераций Якоби?
3. Предложите способ декомпозиции массивов по процессам, который будет лучше учитывать балансировку нагрузки, чем способ в MPE_DECOMP1D.
4. Почему процедуры EXCHNG1, SWEEP1D вызываются дважды в реализации итераций Якоби для решения задачи Пуассона на рис. 8.4.?
5. В чем особенность метода итераций Якоби для 2D декомпозиции процессов?

Контрольные вопросы к 8.3

1. Объясните, почему при выполнении обмена на рис. 8.9. не возникает дедлок?
2. В чем различие между обменами: упорядоченныv send/reseive и комбинированныv send/receive?
3. Как зависит размер буфера для буферизованного обмена от размерности вычислений?
4. Какие изменения необходимы в программах, чтобы можно было совершать некоторую работу, пока ожидаются данные в случае неблокирующего обмена?
5. Что такое «экономные» программы?

Задания для самостоятельной работы

8.1. Напишите программу реализации метода итераций Якоби для одномерной декомпозиции. Используйте для топологии квадратную сетку произвольного размера maxn*maxn. Рассмотрите несколько вариантов обмена:

1) блокирующий обмен;
2) операции сдвига вверх и вниз, реализуемые с помощью MPI_SendRecv;
3) неблокирующий обмен.

8.2. Напишите программу реализации метода итераций Якоби для двумерной декомпозиции.

Глава 9. ПАРАЛЛЕЛИЗМ В РЕШЕНИИ ЗАДАЧ КРИПТОАНАЛИЗА

9.1. КРИПТОЛОГИЯ И КРИПТОАНАЛИЗ

Криптология – раздел прикладной математики, изучающий методы, алгоритмы, программные и аппаратные средства для защиты информации (криптография), а также предназначенный для оценки эффективности такой защиты (криптоанализ). Общие сведения по криптологии, представленные в параграфах 9.1 и 9.2, с разрешения авторов взяты из книги [22]. Информация по криптографии также содержится в [23]. В параграфе 9.3 приводится пример разработки параллельной программы, реализующей некоторый алгоритм криптоанализа.

Любая математическая теория наряду с прямыми задачами рассматривает обратные задачи, которые часто оказываются вычислительно более сложными. Криптоанализ занимается задачами, обратными по отношению к криптографии. Основная цель криптоанализа состоит не в проникновении в компьютерные сети для овладения конфиденциальной информацией, а в оценке стойкости (надежности) используемых и разрабатываемых криптосистем. Методы криптоанализа позволяют оценивать стойкость (уровень безопасности) криптосистем количественно в виде числа W компьютерных операций, необходимых криптоаналитику для вскрытия ключа (или исходного текста). Для пользователя криптосистемы важно, чтобы это число W было настолько велико (например, несколько лет счета на современной вычислительной технике), чтобы за время криптоанализа информация потеряла свою ценность. Под криптоатакой здесь понимается задача оценивания ключевой информации при условии, что сама используемая криптосистема известна.

В зависимости от условий взаимодействия криптоаналитика с криптосистемой различают следующие четыре типа криптоатак:

1) криптоатака с использованием криптограмм;
2) криптоатака с использованием открытых текстов и соответствующих им криптограмм;
3) криптоатака с использованием выбираемых криптоаналитиком открытых текстов и соответствующих им криптограмм;
4) криптоатака с использованием активного аппаратного воздействия на криптосистему.

Эти четыре типа криптоатак упорядочены по возрастанию степени активности криптоаналитика при взаимодействии с криптосистемой.

Далее мы рассмотрим только криптоатаку типа 2. Для нее возможны четыре метода криптоанализа: метод “опробования” (полного перебора); метод криптоанализа с использованием теории статистических решений; линейный криптоанализ; разностный криптоанализ.

Из этих четырех методов далее рассматривается только метод “опробования” ввиду его относительной простоты и наглядности в компmютерной реализации.

Пусть рассматривается произвольная симметричная криптосистема. Пусть $X = (x_1,...,x_n) \in V_N^n$ – открытый текст, подлежащий шифрованию; $u^0 = (u_1^0,...,u_L^0) \in И \subseteq V_m^L$ – истинное значение ключа, использованное в данном сеансе шифрования; $f(\cdot)$ – криптографическое преобразование, а $Y = (y_1,..., y_n) \in V_N^n$ – криптограмма (зашифрованный текст), тогда:

$$Y = f(X; u^0) \tag{9.1}$$

Обязательным условием любой криптосистемы является условие биективности преобразования при фиксированном значении ключа (параметра) θ^0, так что выполняется соотношение

$$f^{-1}(Y; u^0) = X. \tag{9.2}$$

Соотношение (9.1) определяет алгоритм расшифрования зарегистрированной криптограммы Y санкционированным получателем информации, который знает ключ θ^0.

Метод опробования базируется на соотношении (9.2) и заключается в следующем:

1. На основе имеющейся криптограммы $Y \in V_N^n$ и соответствующего ей открытого текста $X \in V_N^n$ составляется система уравнений относительно $\theta = (\theta_1,...,\theta_L)$

$$f_i^{-1}(Y;\theta) = x_i, \qquad i = \overline{1,n}. \tag{9.3}$$

2. Полным перебором всех возможных значений ключевого параметра $\theta \in \Theta \subseteq V_m^L$ находится подмножество $\Theta_0 = \{\theta^{(1)},...,\theta^{(l)}\}$ l решений этой системы;
3. Если число найденных решений l = 1, то в силу (9.2) определено истинное значение параметра $\theta = \theta^0$; в противном случае Θ_0 не

является одноточечным множеством и определено подмножество l значений искомого параметра, одно из которых в силу (9.2) совпадает с истинным значением θ^0.

В случае $l>1$ целесообразно увеличить число уравнений в системе (9.3). Для этого необходимо увеличить число символов n в исходном сообщении X либо получить и использовать в (9.3) $q>1$ пар (сообщение, криптограмма) $(X^{(1)},Y^{(1)}),...,(X^{(q)},Y^{(q)})$:

$$f^{-1}(Y^{(j)};\theta) = X^{(j)}, \qquad j = \overline{1,q}. \tag{9.4}$$

Увеличивая n или q, можно достичь случая $l = 1$ и, следовательно, достичь безошибочного оценивания ключа: $\hat{\theta} = \theta^0$.

Вычислительная сложность метода «опробования» характеризуется количеством компьютерных операций, необходимых для реализации этого метода:

$$W = W(n,q,|\Theta| = |\Theta| \cdot W_1(n,q), \tag{9.5}$$

где $|\Theta|$ – количество всевозможных допустимых значений ключа, $W_1(n, q)$ – число компьютерных операций, затрачиваемых на "опробование (проверку (9.4) для одного значения θ). Обычно $W_1(n, q)$ зависит от q линейно:

$$W_1(n,q) = \omega(n) \cdot q, \tag{9.6}$$

где $\omega(n)$ – число компьютерных операций, затрачиваемых на проверку (9.3) применительно к n – символьным сообщениям для единичного значения $\theta \in \Theta$; $\omega(n)$ фактически совпадает с числом компьютерных операций при расшифровании криптограммы (вычисление обратной функции $f^{-1}(\cdot)$ при известном ключе). Заметим, что величина $\omega(n)$ является известной характеристикой для каждой реальной криптосистемы, причем чем выше быстродействие криптосистемы, тем эта величина меньше.

9.2. КРИПТОСИСТЕМА DES

В широко известной криптосистеме DES используется блочный принцип шифрования двоичного текста. Длина блока шифрования составляет 64 бита. Размер ключа также составляет 64 бита. При этом каждый 8-й бит является служебным и в шифровании не участвует. Каждый такой бит является двоичной суммой семи предыдущих и

служит лишь для обнаружения ошибок при передаче ключа по каналу связи. Процесс криптопреобразования включает следующие три основных этапа.

1. Биты исходного сообщения *x* подвергаются начальной подстановке *IP* в соответствии с табл. 9.1.

Таблица 9.1

Система замены битов исходного сообщения

IP	58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
	62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
	57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
	61	53	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Это означает, что 58-й бит становится первым, 50-й – вторым и т.д. Затем полученный вектор $x_0 = IP(x)$ представляется в виде $x_0 = L_0 R_0$, где L_0 – левая половина из 32 бит, а R_0 – правая половина из 32 бит.

2. Сообщение $L_0 R_0$ преобразуется далее 16 раз по так называемой схеме Фейстеля, приведенной на рис.9.1:

$$L_i = R_{i-1}, \qquad R_i = L_{i-1} \oplus f(R_{i-1}, K_i), \qquad i = 1,2,...,16,$$

где функция f и расписание ключей $K_1, K_2, .., K_{16}$ будут описаны отдельно.

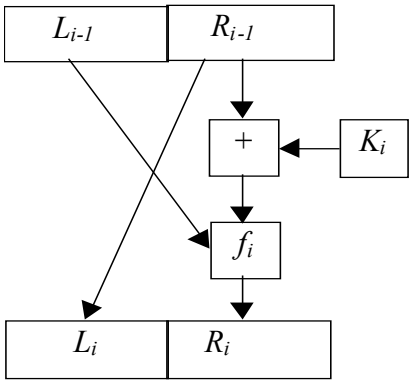


Рис. 9.1 Алгоритм криптопреобразования

3. Сообщение $L_{16} R_{16}$ перемешивается подстановкой IP^{-1}:

$$y = IP^{-1}(R_{16} L_{16})$$

есть зашифрованное сообщение.

Функция *f*. Эта функция имеет два аргумента *A* и *B*. Первый из них состоит из 32 бит, а второй из 48 бит. Результат имеет 32 бита.

1. Первый аргумент *A,* имеющий *32* бита, преобразуется в 48-битовый вектор *P(A)* путем перестановки с повторениями исходного вектора *A*. Эта процедура одна и та же для всех раундов. Она задается табл. 9.2.

Таблица 9.2

Преобразование 32-разрядного аргумента A в 48-разрядный вектор P(A)

P_1	32	1	2	3	4	5	4	5	6	7	8	9	8	9	10	11
	12	13	12	13	14	15	16	17	16	17	18	19	20	21	20	21
	22	23	24	25	24	25	26	27	28	29	28	29	30	31	30	1

2. Далее вычисляется сумма $P(A) \oplus B$ и записывается в виде конкатенации восьми 6-битовых слов:

$$P(A) \oplus B = B_1B_2B_3B_4B_5B_6B_7B_8.$$

3. На этом этапе каждое слово B_i поступает на соответствующий *S*-блок S_i . Блок S_i преобразует 6-битовый вход B_i в 4-битовый выход C_i . *S*-блок есть матрица 4x16 с целыми элементами в диапазоне от 0 до 16. Два первых бита слова B_i, если их рассматривать как двоичную запись числа, определяют номер строки матрицы. Четыре последних бита определяют некоторый столбец. Тем самым найден некоторый элемент матрицы. Его двоичная запись является выходом. В табл. 9.3 представлен один из *S*-блоков.

Таблица 9.3

Содержание блока S_1

S_1	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	5	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

4. Выход $C = C_1, C_2 \ldots C_s$ перемешивается фиксированной подстановкой P_2:

***Таблица* 9.4**

Подстановка P_2

P_2	16	7	20	21	29	12	28	17	1	15	23	26	5	18	31	10
	2	8	24	14	32	27	3	9	19	13	30	6	22	11	4	25

Расписание ключей определяется следующим алгоритмом:

1. В 64-битовом ключе K устраняются биты 8, 16, …, 64. Оставшиеся биты перемешиваются подстановкой P_3.

Таблица 9.5

Подстановка для перемешивания ключей с устраненными битами

P_3	57	49	41	33	25	17	9	1	58	50	42	34	26	18
	10	2	59	51	43	35	27	19	11	3	60	52	44	36
	53	55	47	39	31	23	15	7	62	54	46	38	30	22
	14	6	61	53	45	37	29	2	13	5	28	20	12	4

Выход P_3 (K) представляется в виде $P_3(K) = C_0 D_0$, где C_0 – левая половина, D_0 – правая половина.

2. Очередные значения $C_i D_i$ вычисляются по схеме

$$C_i = L_i(C_{i-1}), \qquad D_i = L_i(D_{i-1}),$$

где L_i – циклический сдвиг влево на одну позицию, если i=1,2,9,16. В остальных случаях L_i – сдвиг влево на две позиции.

3. На этом этапе выход перемешивается подстановкой P_4.

Таблица 9.6

Подстановка для перемешивания

P_4	14	17	11	24	1	5	3	28	15	6	2	10
	23	19	12	4	26	8	16	7	27	20	13	2
	41	52	31	37	47	55	30	40	51	45	33	48
	44	49	39	56	34	53	46	42	50	36	29	32

Дешифрация в стандарте DES. В процессе дешифрования последозвательно используются преобразования

$$IP, \Phi_1, \Phi_2, ..., \Phi_{16}, T, IP^{-1}.$$

Каждое из преобразований Фейстела остается композицией двух преобразований. Первое из них – транспозиция L и R: $T(L,R)= (R,L)$. Таким образом, для любого i $\Phi_i = V_i T$. Кроме того, легко убедиться, что $V_i^2 = T^2 = id$ – тождественное преобразование. Такие преобразования (элементы группы) принято называть *инволюциями.* Для них, в частности, верно $V_i^{-1} = V_i, T^{-1} = T$.

По правилу обращения элементов неабелевой группы имеем

$$D = E^{-1} = ((IP)^{-1} T V_{16} T V_{15} T ... V_1 T (IP))^{-1} = (IP)^{-1} T V_1 T V_2 ... V_{16} T (IP).$$

Это означает, что дешифрование осуществляется теми же алгоритмом и ключом, но в расписание ключей надо внести некоторое изменение: поменять на обратный порядок генерации ключей.

Из (9.5), (9.6) следует, что основным фактором, определяющим вычислительную сложность метода "опробования" и одновременно криптостойкость алгоритма шифрования по отношению к рассматриваемому методу криптоатаки, является мощность пространства ключей $|\Theta|$. С учетом этого оценим вычислительную сложность метода "опробования" криптосистемы *DES*.

В этом случае двоичный текст (N = 2) шифруется блоками размера n= 64 и используется 64-битовый ключ:

$$\Theta = \{\theta_1,...,\theta_{56}) : \theta_i \ э \ V_2, i = \overline{1,56}\},$$

поэтому

$$|\Theta| = 2^{56}, \qquad W = 2^{56} q\omega(64).$$

Для суперЭВМ Intel ASCI RED (9152 процессора) среднее время перебора для системы ***DES*** составляет 9.4 часа.

9.3. ПАРАЛЛЕЛЬНАЯ РЕАЛИЗАЦИЯ DES АЛГОРИТМА

Пусть имеется процедура шифрации двоичного текста по DES алгоритму, описанному в парграфе 9.2:

procedure_shifr(in,out);

где **in** – блок исходного сообщения, длина блока шифрования составляет 64 бита, **out** – зашифрованное сообщение. Пусть также имеется процедура дешифрации двоичного текста:

procedure_de_shifr(out,outin,key);

где **out** – зашифрованное сообщение, **outin** – результат дешифрации сообщения с помощью ключа **key**.

Тогда метод «опробования» сводится к следующей задаче. Сначала открытый текст шифруется с помощью некоторого ключа. Выполняем полный перебор значений ключа в некотором диапазоне. При этом для каждого значения вызываем процедуру дешифрации с этим значением ключа. Для простоты будем считать: если дешифрованный текст совпадает с исходным, ключ найден (иначе см. параграф 9.1).

Ключ в DES алгоритме 64-битовый, следовательно, можно его представлять двумя 32-разрядными (unsigned long), четырьмя 16-раз-

рядными (unsigned int) или восемью 8-разрядными беззнаковыми целыми (unsigned char). Выберем первый способ представления ключа:

unsigned long key_data[2].

Последовательный алгоритм для метода «опробования» при определении ключа будет выглядеть следующим образом.

```
int main(argc,argv)
int argc;
char argv[];
{  int i,j,err=0;
   unsigned char in[8],out[8],outin[8];      /* 64-битовые сообщения */
   unsigned long key_data[2];                /* 64-битовый ключ   */
   unsigned long i1=0,i2=0;
   unsigned long imin[2]={0,0};              /* нижняя граница  */
   unsigned long imax[2]={0xffff,0xffff}; /* верхняя граница  диапазона  ключа */
   procedure_shifr(in,out);                  /* вызов процедуры шифрации */
   i1=0;i2=0;
                                          /*перебор значений старшей части ключа*/
  for (i1=imin[0]; i1<imax[0]; i1++)
  {         key_data[i][0]=i1;
                                         /*перебор значений младшей части ключа*/
          for (i2=imin[1]; i2<imax[1]; i2++)
          {         key_data[i][1]=i2;
                     /* вызов процедуры дешифрации с помощью ключа key_data */
                   procedure_de_shifr(out,outin,key_data);
                   if (memcmp(in,outin,8) == 0)
                   {        printf("key is: 0x%x 0x%x \n",i1,i2);          /* ключ найден */
                            return 0;
                   }
          }
  }
  return 0;
}
```

Рис. 9.2. Последовательная программа определения ключа по DES алгоритму

Рассмотрим параллельный алгоритм для решения этой задачи. Поиск ключа – перебор всевозможных значений из заданного диапазона. Естественно разделить диапазон изменения ключа (верхний диапазон **key_data[1]** или нижний диапазон **key_data[2]**) по процессам так, чтобы каждый процесс независимо осуществлял перебор значений своего диапазона. Пусть **min** – нижняя, **max -** верхняя граница значений диапазона ключа, **myid** – собственный номер процесса, **numprocs** – коли-

чество процессов приложения, **n1** – нижняя, **n2** – верхняя границы диапазона ключа для данного процесса. Тогда можно предложить следующий способ распределения работы по процессам:

```
h= (max-min+1)/numprocs                 /*  частное  */
ost=(max-min+1)%numprocs                /*  остаток   */
n1=min+myid*h
n2=n1+h
if ((ost !=0) &&(myid==numprocs)) n2=n2+ost
                        /* в последний процесс –  остаток работы, если он есть */
```

Так как количество вариантов ключей намного больше количества процессов, то можно считать, что общий объем вычислений будет распределен между процессами равномерно.

Чтобы узнать, найден ли ключ, удобно использовать коллективную функцию **MPI_Allgather**, которая позволяет каждому процессу иметь информацию обо всех процессах.

```
MPI_Allgather( &num, 1, MPI_INT, buf, 1, MPI_INT, MPI_COMM_WORLD).
```

В результате вызова этой функции в массиве **buf** будут находиться значения переменной **num** всех процессов коммуникатора, которая будет иметь значение **0xff**, если ключ найден, и нуль – в противном случае.

Если ключ найден, необходимо остановить работу всех процессов. Имеется два варианта остановки процессов. Если при проверке каждого варианта ключа опрашивать процессы о необходимости продолжать работу, то получится, что все процессы синхронизируют свои действия и постоянно опрашивают друг друга, что, естественно, нерационально. Если опрашивать процессы не на каждой итерации цикла перебора, а, например, при каждом новом значении старшей части ключа (после полного перебора всех значений младшей части), то это приведет к дополнительному счету по поиску ключа, когда он уже найден, но процессы не знают об этом. Анализ показывает, что реализация второго способа позволяет получить хороший параллельный алгоритм для решения задачи.

Возможны следующие случаи нахождения ключа процессами при полном переборе вариантов:

- ключ найден одним из процессов,
- ключ из заданного диапазона не найден ни одним процессом,
- часть процессов закончила перебор значений ключа, а другая – продолжает перебор значений.

В первом случае коллективный опрос процессов приведет к прекращению работы всех процессов приложения в случае нахождения ключа. Это можно реализовать вызовом функции **MPI_Allgather** и анализом полученных данных. Программа представлена ниже.

```
MPI_Allgather( &num,1,MPI_INT,buf,1,MPI_INT,MPI_COMM_WORLD);
for (i=0; i<numprocs; i++)
  if (buf[i]==0xFF)
  {                                /* значение num=0xFF, если  ключ найден  */
      printf("process %d stopped \n",myid);
       MPI_Finalize();             /* прерывание работы процессов */
       return 0;
     }
```

В двух оставшихся случаях может возникнуть ошибка вызова коллективной функции. Коллективная функция должна вызываться всеми процессами коммуникатора без исключения, но некоторые из процессов могут закончить работу по перебору ключей, что вызовет “зависание” программы. Поэтому необходимо введение цикла после перебора ключей, в будем выполнять коллективный опрос процессов о нахождении ключа до тех пор, пока либо он не будет найден, либо перебор ключей закончат все процессы приложения. Это можно осуществить следующим образом:

```
num=myid;                        /* номер процесса, закончившего перебор */
do                               /* опрос всех процессов, найден ли ключ  */
{
    MPI_Allgather(&num,1,MPI_INT,buf,1,MPI_INT,MPI_COMM_WORLD);
    for (i=0; i<numprocs; i++)
       if (buf[i]==0xFF)
       {      printf("process %d stopped \n",myid);
              MPI_Finalize();       /* прерывание работы процессов  */
              return 0;
       }
                 /* возможен вариант, когда ключ не найден ни одним процессом */
    PR=0;
    for (i=0; i<numprocs; i++)
         if (buf[i]!=i)
            { PR=0xFF; break; }
}while (PR!=0);
```

В **num** заносится собственный номер процесса, закончившего перебор вариантов ключей. Тогда полное окончание работы всех процессов

произойдет, когда после выполнения коллективной функции в массиве **buf** будут номера всех процессов приложения: 0,…**numprocs**-1.

Ниже приведен текст параллельной программы метода опробования для DES алгоритма криптоатаки с использованием открытых текстов, который является обобщением вышеописанного алгоритма.

```
int main(argc,argv)
int argc;
char *argv[];
{  int i,numprocs,myid,PR;
   unsigned long key_data[2];                  /* 64-битовый ключ */
   unsigned char in[8],out[8],outin[8];        /* 64-битовые тексты */
   unsigned long i1=0,i2=0;
   int h,n1,n2,ost,num,*buf;
   unsigned long imin[2]={0,0};                /* нижняя граница  */
   unsigned long imax[2]={0xffff,0xffff};      /* верхняя граница   диапазона  ключа */
   double t1,t2;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
                                               /* вызов процедуры шифрации */
   procedure_shifr(in,out);
          /* определение диапазонов значений ключей для каждого процесса */
   h=(imax[0]-imin[0]+1)/numprocs;
   ost=(imax[0]-imin[0]+1)%numprocs;
   n1=imin[0]+myid*h;
   n2=n1+h;
   if ((ost!=0)&&(myid==numprocs-1))
      n2=n2+ost;
   num=0;
   buf= (int *)malloc(numprocs*sizeof(int));
   for (i=0; i<numprocs; i++) buf[i]=0;
   t1 = MPI_Wtime();
          /* перебор значений старшей части ключа для каждого процесса свой */
   for (i1=n1; i1<n2; i1++)
   {
       key_data[0]=i1;
          /* перебор значений младшей части ключа для всех процессов одинаков*/
       for (i2=imin[1]; i2<imax[1]; i2++)
       {    key_data[1]=i2;
                       /* вызов процедуры дешифрации с помощью ключа key_data */
            procedure_de_shifr(out,outin,key_data);
            if (memcmp(in,outin,8) == 0)           /* ключ найден */
            {    t2 = MPI_Wtime();
```

```
            printf("key is: 0x%x 0x%x \n",i1,i2);
            printf("process %d  %d \n",myid,i1);
            printf("time %f \n",t2-t1);
            num=0xFF;   }                    /* ключ найден  */
        }
                /* опрос всех процессов, найден ли ключ   */
        MPI_Allgather( &num,1,MPI_INT,buf,1,MPI_INT,MPI_COMM_WORLD);
        for (i=0; i<numprocs; i++)
          if (buf[i]==0xFF)
          {
              printf("process %d stopped \n",myid);
              MPI_Finalize();          /* прерывание работы процессов  */
              return 0;
          }
     }
                          /* данный процесс ключ не нашел  */
    if (num==0) printf(" no key! in process %d \n",myid);
    num=myid;             /* номер процесса, закончившего перебор */
   do
   {                      /* опрос всех процессов, найден ли ключ     */
      MPI_Allgather(&num,1,MPI_INT,buf,1,MPI_INT,MPI_COMM_WORLD);
     for (i=0; i<numprocs; i++)
        if (buf[i]==0xFF)
        {
            printf("process %d stopped \n",myid);
            MPI_Finalize();              /* прерывание работы процессов */
            return 0;
        }
          /* возможен вариант, когда ключ не найден ни одним процессом  */
     PR=0;
     for (i=0; i<numprocs; i++)
         if (buf[i]!=i) { PR=0xFF; break; }
   }while (PR!=0);
            /* все процессы закончили работу, ключ не найден */
 MPI_Finalize();
 return 0;
}
```

Рис. 9.3. Параллельная программа определения ключа по DES алгоритму

КОНТРОЛЬНЫЕ ВОПРОСЫ К ГЛАВЕ 9

Контрольные вопросы к 9.1

1. Что такое криптология, крипография, криптоанализ? Каково основное назначение криптоанализа?
2. Какие виды криптоатак Вы знаете?

3. Что такое криптоатака с использованием открытых текстов и соответствующих им криптограмм?
4. Что такое метод «опробования»?
5. Что означают преобразования: прямое преобразование $Y = f(X;\theta^0)$ и обратное ему преобразование $f^{-1}(Y;u^{0)} = X$?
6. В чем заключается метод полного перебора?
7. Возможны ли множественные решения при поиске ключей? Как это устранить?
8. Каковы временные затраты на проведение криптоанализа для системы DES?

Контрольные вопросы к 9.2

1. Что такое криптосистема DES? Опишите основные этапы шифрования в системе DES.
2. Что такое преобразование Фейстела? Что такое функция *f*, роль ключа в реализации этой функции?
3. Как выглядит дешифрация в системе DES?

Контрольные вопросы к 9.3

1. Предложите другой (более равномерный) способ распределения работы по процессам в параллельной программе криптографии.
2. Почему для опроса процессов о нахождении ключа удобно использовать коллективную функцию MPI_Allgather?
3. Почему необходимо введение цикла do … while после циклов перебора всех значений ключа?
4. Предложите вариант параллельной программы для данной задачи без использования коллективной функции.
5. Почему опрос процессов о том, найден ли ключ, на каждой итерации цикла перебора не позволяет получить ускорение на сети компьютеров?

Глава 10. СИСТЕМЫ ЛИНЕЙНЫХ АЛГЕБРАИЧЕСКИХ УРАВНЕНИЙ

10.1. МЕТОДЫ РЕШЕНИЯ СЛАУ

Основная задача вычислительной алгебры – решение систем линейных алгебраических уравнений (СЛАУ):

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &= b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &= b_2, \\ \ldots & \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n. \end{aligned}$$

Предполагается, что матрица A неособенная, т. е. $detA \neq 0$, и решение единственно.

Прямые и итерационные методы. Численные методы решения СЛАУ делятся на две большие группы: прямые и итерационные [24,25]. Прямые методы при отсутствии ошибок округления за конечное число арифметических операций позволяют получить точное решение x^*. В итерационных методах задается начальное приближение x^0 и строится последовательность $\{x^k\} \xrightarrow[k\to\infty]{} x^*$, где k – номер итерации. В действительности итерационный процесс прекращается, как только x^k становится достаточно близким к x^*.

Итерационные методы привлекательнее с точки зрения объема вычислений и требуемой памяти, когда решаются системы с матрицами высокой размерности. При небольших порядках системы используют прямые методы либо прямые методы в сочетании с итерационными методами.

В этой главе рассмотрены два метода решения СЛАУ – Якоби и Гаусса–Зейделя. Метод Якоби рассмотрен в силу того, что вычисления компонент вектора внутри одной итерации независимы друг от друга и параллелизм очевиден, поэтому легко написать MPI программу.

Метод Гаусса–Зейделя более эффективен, поскольку требует заметно меньше итераций. Высокая сходимость к решению в этом методе достигается за счет выбора матрицы расщепления, лучше аппроксимирующей матрицу A. Однако в методе Гаусса–Зейделя вычисление каждой компоненты вектора зависит от компонент вектора, вычисленных на этой же итерации. Поэтому на первый взгляд метод носит последовательный характер. В этой главе предложен параллельный алгоритм для модифицированного метода Гаусса–Зейделя.

Метод простой итерации (метод Якоби). Пусть требуется решить систему

$$Ax = b\,;\, x,b \in R^n.$$

Представим $A = A^- + D + A^+$, где D – диагональная матрица с диагональными членами матрицы A; A^- – часть матрицы A, лежащая ниже центральной диагонали; A^+ – часть матрицы A, лежащая выше центральной диагонали. Тогда

$$\left(A^- + D + A^+\right)x = b,$$

или

$$Dx = -\left(A^- + A^+\right)x + b.$$

Запишем итерационный метод в виде

$$\mathrm{D}\mathrm{x}^{k+1} = -\left(\mathrm{A}^{-} + \mathrm{A}^{+}\right)\mathrm{x}^{k} + \mathrm{b}\,;\ k = 0,1,2,\ldots\ .$$

Разрешим его относительно x^{k+1}:

$$x^{k+1} = -D^{-1}(A^{-} + A^{+})x^{k} + D^{-1}b\,;\ k = 0,1,\ldots$$

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}\,x_j^{k} - \sum_{j=i+1}^{n} a_{ij}\,x_j^{k}\right);\ i = \overline{1,n}. \qquad (10.1)$$

Нетрудно убедиться, что метод Якоби в координатной форме есть не что иное, как разрешение каждого из уравнений системы относительно одной из компонент вектора. Из первого уравнения системы выражается x_1 и его значение принимается за значение x_1^{k+1}. Из второго уравнения определяется x_2 и его значение принимается за x_2^{k+1} и т. д. Переменные в правой части этих соотношений при этом полагаются равными их значениям на предыдущей итерации.

Метод Гаусса–Зейделя. Пусть решаемая система представлена в виде

$$(\mathrm{A}^{-} + \mathrm{D} + \mathrm{A}^{+})\mathrm{x} = \mathrm{b}\,.$$

Итерационная схема Гаусса–Зейделя следует из этого представления системы:

$$(\mathrm{A}^{-} + \mathrm{D})\mathrm{x}^{k+1} = \mathrm{b} - \mathrm{A}^{+}\mathrm{x}^{k}\ \ ;\ k = 0,1,2,\ldots,$$

или

$$\mathrm{D}\mathrm{x}^{k+1} = -\mathrm{A}^{-}\mathrm{x}^{k+1} - \mathrm{A}^{+}\mathrm{x}^{k} + \mathrm{b}\ \ ;\ k = 0,1,2,\ldots.$$

Приведем метод Гаусса–Зейделя к стандартному виду:

$$\mathrm{x}^{k+1} = -(\mathrm{A}^{-} + \mathrm{D})^{-1}\mathrm{A}^{+}\mathrm{x}^{k} + (\mathrm{A}^{-} + \mathrm{D})^{-1}\mathrm{b}\,;\ k = 0,1,\ldots.$$

Стандартная форма метода позволяет выписать его итерационную матрицу и провести над ней очевидные преобразования:

$$\mathrm{B} = -(\mathrm{A}^{-} + \mathrm{D})^{-1}\mathrm{A}^{+} = -(\mathrm{A}^{-} + \mathrm{D})^{-1}(\mathrm{A} - \mathrm{D} - \mathrm{A}^{-}) = \mathrm{I} - (\mathrm{A}^{-} + \mathrm{D})^{-1}\mathrm{A}\,.$$

Представим метод Гаусса–Зейделя в координатной форме для системы общего вида:

$$x_i^{k+1} = \left(b_i - \sum_{j=1}^{i-1} a_{ij}\,x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}\,x_j^{k}\right) \Big/ a_{ii}\,;\ i = \overline{1,n};\ k = 0,1,2,\ldots. \qquad (10.2)$$

Координатная форма метода Гаусса–Зейделя отличается от координатной формы метода Якоби лишь тем, что первая сумма в правой части итерационной формулы содержит компоненты вектора решения не на *k*-й, а на *(k+1)*-й итерации.

10.2. ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ РЕШЕНИЯ СЛАУ

10.2.1. Последовательный алгоритм метода простой итерации

Для реализации метода простой итерации должны быть заданы матрица СЛАУ *A,* вектор свободных членов *B***,** начальное приближение вектора *X*, точность вычислений ε. Тогда новые значения вектора *X* вычисляются по формуле (10.1).

Критерий завершения процесса вычислений

$$\left\| x^{k+1} - x^{k} \right\| = \max \left| x_i^{\,k+1} - x_i^{\,k} \right| < \varepsilon, 1 \le i \le n,$$

где x^k – приближенное значение решения на k-м шаге численного метода.

Ниже приведен текст процедуры **Iter_Jacoby** для вычисления новых значений компонент вектора *X* по формуле 10.1.

```
int ind(int i,int j,int SIZE)       /* процедура обращения к элементу матрицы */
{        return (i*(SIZE+1)+j);   }

void Iter_Jacoby(double *A, double *X, double *X_old, int size)
            /* задана матрица A, начальное приближение вектора X_old,
               размерность матрицы size, вычисляем новое значение вектора X */
{
   unsigned int i, j;
   double Sum;
   for (i = 0; i < size; ++i)
   {
       Sum = 0;
       for (j = 0; j < i; ++j)
           Sum += A[ind(i,j,size)] * X_old[j];
       for (j = i+1; j < size; ++j)
           Sum += A[ind(i,j,size)] * X_old[j];
       X[i]=(A[ind(i,size,size)] –  Sum) / A[ind(i,i,size)];
   }
}
```

Рис. 10.1. Процедура вычисления значений вектора по методу простой итерации

Тогда процедура решения СЛАУ методом простой итерации будет следующей.

```
unsigned long int SolveSLAE(double *A, double *X, double Error, int size)
  /* задана матрица A размерности size+1  (матрица+столбец свободных
     членов), начальное приближение вектора X, погрешность вычислений Error */
{  double X_old;                          /*  предыдущее значение вектора X */
   int i,Iter = 0;                        /* число итераций  */
   double dNorm, dVal;
   X_old = malloc(sizeof(double) * size);   /* выделяем память для X_old */
   do{
        ++Iter;
                                          /*сохраняем предыдущее значение вектора X */
        memcpy(X_old, X, size);
                                          /* вычисляем новое значение вектора X */
        Iter_Jacoby(A, X, X_old, size);
        dNorm = 0;                        /* считаем норму погрешности */
        for (i = 0; i < size; ++i)
        {
             dVal = fabs(X[i] –  X_old[i]);
             if (dNorm < dVal) dNorm = dVal;
        }
   }while(Error < dNorm);                 /* цикл до достижения необходимой точности */
   free(X_old );
   return Iter;
}
```

Рис. 10.2. Последовательная программа реализации метода простой итерации

В основном цикле процедуры сохраняется значение вектора, вычисленное на предыдущей итерации цикла, новое значение вектора вычисляется в процедуре **Iter_Jacoby**, затем вычисляется норма погрешности. Если достигнута необходимая точность вычислений, осуществляется выход из цикла,.

10.2.2. Параллельный алгоритм метода простой итерации

Следующая система уравнений описывает метод простой итерации:

$$X_1^{k+1}=f1(x_1^k,x_2^k,\ldots x_n^k)$$
$$X_2^{k+1}=f2(x_1^k,x_2^k,\ldots x_n^k)$$
$$\ldots$$
$$X_n^{k+1}=fn(x_1^k,x_2^k,\ldots x_n^k)$$

Вычисления каждой координаты вектора зависят от значений вектора, вычисленных на предыдущей итерации, и не зависят между собой. Поэтому, естественно, для параллельной реализации можно редложить следующий алгоритм.

Количество координат вектора, которые вычисляются в каждом процессе, определим следующим образом.

```
size = (MATR_SIZE/numprocs)+((MATR_SIZE % numprocs) > myid ? 1 : 0 );
```

где **size** – количество координат вектора, вычисляемых в данном процессе, **myid** – собственный номер процесса, **numprocs** – количество процессов приложения, **MATR_SIZE** – размерность вектора.

Тогда каждый процесс может вычислять свое количество size новых значений координат вектора. После этого процессы могут обменяться вновь вычисленными значениями, что позволит главному процессу произвести оценку точности вычислений.

Процедура **Iter_Jacoby** для вычисления значений вектора методом простой итерации в параллельной версии изменится следующим образом: в ней вычисляется **size** значений вектора **X** с номера **first**.

```
void Iter_Jacoby(double *X_old, int size, int MATR_SIZE, int first)
 /* задана матрица A, начальное приближение вектора X_old, размерность
    матрицы  MATR_SIZE, количество вычисляемых элементов вектора
    в данном процессе size, вычисляем новые значение вектора X с номера first */
{
  int i, j;
  double Sum;
  for (i = 0; i < size; ++i)
  {
     Sum = 0;
     for (j = 0; j < i+first; ++j)
         Sum += A[ind(i,j,MATR_SIZE)] * X_old[j];
     for (j = i+1+first; j < MATR_SIZE; ++j)
          Sum += A[ind(i,j,MATR_SIZE)] * X_old[j];
     X[i+first]=(A[ind(i,MATR_SIZE,MATR_SIZE)] –  Sum) /
                    A[ind(i,i+first, MATR_SIZE)];
  }
}
```

Рис. 10.3. Процедура вычисления значений вектора по методу простой итерации (параллельная версия)

Ниже представлена процедура **SolveSLAE**, которая реализует метод простой итерации.

```
void SolveSLAE(int MATR_SIZE, int size, double Error)
        /* задана матрица A, размерность матрицы MATR_SIZE, количество
          элементов,  вычисляемых в данном процессе size, погрешность
          вычислений Error*/
{ double *X_old ;
  int Iter = 0, i, Result, first;
  double dNorm = 0, dVal;
        /* определение номера элемента first вектора X, с которого
           будут  вычисляться новые значения в данном процессе*/
 MPI_Scan(&size, &first,1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
 first-= size;
        /* заполнение массива sendcounts значениями size   от каждого процесса*/
 MPI_Allgather(&size,1,MPI_INT,sendcounts, 1, MPI_INT, MPI_COMM_WORLD);
        /* заполнение массива displs –  расстояний между элементами вектора,
           распределенных по процессам   */
 displs[0] = 0;
 for (i = 0; i<size; ++i)
    displs[i] = displs[i-1] + sendcounts[i-1];
                        /* выделяем память для X_old */
 X_old = (double *)malloc(sizeof(double) * MATR_SIZE);
 do
 {   ++Iter;
                        /*сохраняем предыдущее значение вектора X */
     memcpy(X_old, X, MATR_SIZE);
                        /* вычисляем новое значение вектора X */
     Iter_Jacoby(X_old, size, MATR_SIZE, first);
                        /* Рассылаем вектор X всем процессам  */
     MPI_Allgatherv(&X[first], size, MPI_DOUBLE,  X, sendcounts, displs,
                        MPI_DOUBLE, MPI_COMM_WORLD );
                        /* Считаем норму */
     if (myid == Root)   {
         for (i = 1; i <= MATR_SIZE; ++i) {
             dVal = fabs(X[i] –  X_old[i]);
             if (dNorm < dVal) dNorm = dVal;
         }
         Result = Error < dNorm;
     }
                        /* рассылаем результат всем процессам */
     MPI_Bcast(&Result, 1, MPI_INT, Root, MPI_COMM_WORLD);
 }while(Result);
 free(X_old);
 return ;
}
```

Рис. 10.4. Процедура реализации метода простой итерации (параллельная версия)

Прежде всего определяется номер элемента вектора **first**, с которого вычисляются новые значения в каждом процессе. Переменная **first** будет в общем случае иметь разные значения, так как количество элементов вектора **size** различно в процессах. Вычислить значение переменной удобно вызовом коллективной функции **MPI_Scan:**

```
MPI_Scan(&size, &first,1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
first-= size;
```

Затем в цикле (аналогично последовательному варианту) сохраняется значение вектора, вычисленного на предыдущей итерации; вызовом процедуры **Iter_Jacoby** вычисляются новые значения вектора, осуществляется рассылка вычисленных значений вектора от всех – всем:

```
MPI_Allgatherv(&X[first], size, MPI_DOUBLE, X, sendcounts, displs,
                    MPI_DOUBLE, MPI_COMM_WORLD );
```

Для выполнения коллективной функции **MPI_ Allgatherv** необходимо заполнить массивы **sendcounts** и **displs** (заполнение массивов лучше выполнить до цикла). После обмена каждый процесс имеет новый вектор **X**. В корневом процессе вычисляется норма погрешности и сравнивается с заданной. Условие выхода из цикла **Result ≠ 0**, поэтому корневой процесс рассылает значение **Result** всем процессам:

```
MPI_Bcast(&Result, 1, MPI_INT, Root, MPI_COMM_WORLD).
```

Ниже представлена главная программа метода простой итерации.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <memory.h>
  /* myid – номер процесса, numprocs – количество процессов, Root – номер
     корневого процесса,   sendcounts, displs – массивы для организации обменов  */
int myid,numprocs,Root=0, *sendcounts, *displs;
   /*  AB – исходная матрица метода, может быть задана любым способом
       X – вектор, содержит начальное приближение */
double *AB, *A,* X;
int main(int argc,char *argv[])
   /*  MATR_SIZE – размерность системы,  size – количество строк матрицы  в
     каждом процессе, SIZE – количество элементов матрицы в каждом процессе */
{   int i, size, MATR_SIZE, SIZE;
    double Error;                                /* точность вычислений */
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
```

```
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  if (myid=Root)  {
        /*  получаем размерность системы MATR_SIZE */
        /*  выделяем память для матрицы AB */
        AB = (double *)malloc(sizeof(double) * MATR_SIZE * (MATR_SIZE + 1));
        /* получение матрицы AB размерности MATR_SIZE+1(матрица+столбец
           свободных  членов), задание точности вычислений Error */
  }
                              /* рассылка значений всем процессам */
  MPI_Bcast(&MATR_SIZE, 1, MPI_INT, Root, MPI_COMM_WORLD);
  MPI_Bcast(&Error, 1, MPI_DOUBLE, Root, MPI_COMM_WORLD);
                              /*  выделяем память для вектора X */
  X = (double *)malloc(sizeof(double) * MATR_SIZE);
  if (myid=Root)  {         /*получаем начальное значение для вектора X  */  }
                              /* рассылка вектора X всем процессам */
  MPI_Bcast(X, MATR_SIZE, MPI_DOUBLE, Root, MPI_COMM_WORLD);
                  /* определение количества элементов вектора, которые будут
                     вычисляться в каждом процессе (равно количеству строк
                     матрицы, находящихся  в данном  процессе) */
  size = (MATR_SIZE/numprocs)+((MATR_SIZE % numprocs) > myid ? 1 : 0 );
                              /*  выделяем память для матрицы A в каждом процессе*/
  A = (double *)malloc(sizeof(double) * (MATR_SIZE+1)*size);
  displs= (int *)malloc(numprocs*sizeof(int));
  sendcounts = (int *)malloc(numprocs*sizeof(int));
                             /* рассылка частей матрицы по процессам */
  SIZE = (MATR_SIZE+1) * size;
  MPI_Gather(&SIZE,1,MPI_INT,endcounts,1,MPI_INT,Root,
                    MPI_COMM_WORLD);
  displs[0] = 0;
  for (i = 1;i<numprocs; ++i)
       displs[i] = displs[i-1] + sendcounts[i-1];
  MPI_Scatterv(AB, sendcounts, displs,  MPI_DOUBLE,  A,
               (MATR_SIZE+1) * size, MPI_DOUBLE,Root, MPI_COMM_WORLD );
                              /* решение СЛАУ методом простой итерации */
  SolveSLAE(MATR_SIZE, size, Error);
                              /* освобождение памяти */
  free(sendcounts);  free(displs);
  free(AB); free(A); free(X);
  MPI_Finalize();
  return 0;
}
```

Рис. 10.5. Главная программа параллельного алгоритма метода простой итерации

В корневом процессе происходит задание размерности исходной системы **MATR_SIZE**, заполняется матрица значений системы **AB** (матрица + столбец свободных членов), начальное приближение вектора решения **X**, точность вычислений **Error**. После инициализации MPI, определения количества процессов в приложении **nimprocs**, собственного номера процесса **myid** каждый процесс получает от корневого процесса заданную размерность исходной матрицы, точность вычислений, начальное приближение вектора **X**:

```
MPI_Bcast(&MATR_SIZE, 1, MPI_INT, Root, MPI_COMM_WORLD);
MPI_Bcast(&Error, 1, MPI_DOUBLE, Root, MPI_COMM_WORLD);
MPI_Bcast(X, MATR_SIZE, MPI_DOUBLE, Root, MPI_COMM_WORLD);
```

После этого каждый процесс определяет количество координат вектора **size**, которые будут вычисляться в данном процессе (разные процессы, возможно, будут иметь разные значения size):

```
size = (MATR_SIZE/numprocs)+((MATR_SIZE % numprocs) > myid ? 1 : 0 );
```

Далее необходимо разделить исходную матрицу AB по процессам: по **size** строк матрицы в каждый процесс:

```
MPI_Scatterv(AB, sendcounts, displs,  MPI_DOUBLE,  A,
        (MATR_SIZE+1) * size, MPI_DOUBLE,Root, MPI_COMM_WORLD );
```

Для выполнения функции **MPI_Scatterv** необходимо заполнить массивы **sendcounts, displs**. В первом из них находится количество элементов матрицы **SIZE** каждого процесса, во втором – расстояния между элементами вектора, распределенного по процессам, причем

```
SIZE=(MATR_SIZE+1)*size,
```

где **MATR_SIZE+1** – количество элементов в строке матрицы, **size** – количество строк матрицы в процессе. Тогда заполнение первого массива **sendcounts** выполняется вызовом функции:

```
MPI_Gather(&SIZE, 1, MPI_INT, sendcounts, 1, MPI_INT, Root,
                MPI_COMM_WORLD);
```

Массив **displs** заполняется следующим образом:

```
displs[0] = 0;
for (i = 1;i<numprocs; ++i)
    displs[i] = displs[i-1] + sendcounts[i-1];
```

10.2.3. Параллельный алгоритм метода Гаусса–Зейделя.

Отличие метода Гаусса–Зейделя от метода простой итерации заключается в том, что новые значения вектора вычисляются не только на основании значений предыдущей итерации, но и с использованием значений уже вычисленных на данной итерации (формула (10.2)). Текст последовательной программы для вычисления новых значений компонент вектора представлен ниже.

```
void GaussZeidel (double *A, double *X, int size)
      /* задана матрица A, начальное приближение вектора X,
         размерность матрицы size, вычисляем новое значение вектора X */
{  unsigned int i, j;
   double Sum;
   for (i = 0; i < size; ++i)  {
       Sum = 0;
       for (j = 0; j < i; ++j)
           Sum += A[ind(i,j,size)] * X[j];
        for (j = i+1; j < size; ++j)
            Sum += A[ind(i,j,size)] * X[j];
        X[i]=(A[ind(i,size,size)] –  Sum) / A[ind(i,i,size)];
   }
}
```

Рис. 10.6. Процедура вычисления значений вектора по методу Гаусса-Зейделя

Следующая система уравнений описывает метод Гаусса-Зейделя.

$$X_1^{k+1}=f1(x_1^k,x_2^k,\ldots x_n^k)$$
$$X_2^{k+1}=f2(x_1^{k+1},x_2^k,\ldots x_n^k)$$
$$\ldots$$
$$X_n^{k+1}=fn(x_1^{k+1},x_2^{k+1},\ldots x_{n-1}^{k+1},x_n^k)$$

Вычисления каждой координаты вектора зависят от значений, вычисленных на предыдущей итерации, и значений координат вектора вычисленных на данной итерации. Поэтому нельзя реализовывать параллельный алгоритм, аналогичный методу простой итерации: каждый процесс не может начинать вычисления пока, не закончит вычисления предыдущий процесс.

Можно предложить следующий модифицированный метод Гаусса–Зейделя для параллельной реализации. Разделим вычисления координат вектора по процессам аналогично методу простой итерации. Будем в каждом процессе вычислять свое количество координат вектора по методу Гаусса-Зейделя, используя только вычисленные значения

вектора данного процесса. Различие в параллельной реализации по сравнению с методом простой итерации заключается только в процедуре вычисления значений вектора (вместо процедуры **Iter_Jacoby** используем процедуру **GaussZeidel**).

```
void GaussZeidel(int size, int MATR_SIZE, int first)
/* задана матрица A, размерность матрицы MATR_SIZE, количество
   вычисляемых элементов вектора в данном процессе size, вычисляем новые
   значения вектора X с номера first, используя значения   вектора X */
{  int i, j;
  double Sum;
  for (i = 0; i < size; ++i)  {
       Sum = 0;
       for (j = 0; j < i+first; ++j)
            Sum += A[ind(i,j,MATR_SIZE)] * X[j];
       for (j = i+1+first; j < MATR_SIZE; ++j)
            Sum += A[ind(i,j,MATR_SIZE)] * X[j];
       X[i+first]=(A[ind(i,MATR_SIZE,MATR_SIZE)] –  Sum) /
                      A[ind(i,i+first, MATR_SIZE)];
  }
}
```

Рис. 10.7. Процедура вычисления значений вектора по методу Гаусса–Зейделя (параллельная версия)

КОНТРОЛЬНЫЕ ВОПРОСЫ И ЗАДАНИЯ К ГЛАВЕ 10

Контрольные вопросы к 10.1

1. В чем различие между прямыми и итерационными методами решения СЛАУ?
2. В чем различие между методоми простой итерации и Гаусса–Зейделя для решения СЛАУ?
3. Почему метод Гаусса–Зейделя эффективнее метода простой итерации?

Контрольные вопросы к 10.2

1. Сравните распределение работ между процессами в методе простой итерации и в задаче криптографии в предыдущей главе.
2. В чем заключается распараллеливание метода простой итерации?
3. Почему для распределения матрицы системы по процессам нужно использовать коллективную функцию MPI_Scatterv, а не MPI_Scatter?
4. Для чего необходимы массивы sendcounts, displs в процедуре SolveSLAE?
5. Объясните суть переменной first в процедуре Iter_Jacoby параллельной реализации. Как иначе можно вычислить ее значение?
6. Как изменится параллельная программа метода простой итерации в случае, если не использовать коллективные функции?
7. В чем различие между методом Гаусса–Зейделя и его модификацией, предложенной в данной главе?

Р А З Д Е Л 4. ОРГАНИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Глава 11. ОБРАБОТКА ИСКЛЮЧЕНИЙ И ОТЛАДКА

11.1. ОБРАБОТКА ИСКЛЮЧЕНИЙ

Реализация MPI может обрабатывать некоторые ошибки, которые возникают при выполнении вызовов MPI. Это могут быть ошибки, которые генерируют исключения или прерывания, например, ошибки для операций с плавающей точкой или при нарушении доступа. Набор ошибок, которые корректно обрабатываются MPI, зависит от реализации. Каждая такая ошибка генерирует исключение MPI.

Пользователь может связывать обработчик ошибок с коммуникатором. Вновь созданный коммуникатор наследует обработчик ошибок, который связан с “родительским” коммуникатором. В частности, пользователь может определить “глобальный” обработчик ошибок для всех коммуникаторов, связывая этот обработчик с коммуникатором **MPI_COMM_WORLD** сразу после инициализации.

В **MPI** доступны предопределенные обработчики ошибок:

- **MPI_ERRORS_ARE_FATAL –** обработчик, который после вызова прерывает работу программы на всех процессах. Это имеет тот же эффект, как если бы процессом, который запустил обработчик, был вызван **MPI_ABORT**.
- **MPI_ERRORS_RETURN –** обработчик не делает ничего, кроме представления кода ошибки пользователю.

Реализации могут обеспечивать дополнительные обработчики ошибок, программисты также могут написать свои собственные обработчики ошибок.

Обработчик ошибок **MPI_ERRORS_ARE_FATAL** связан по умолчанию с **MPI_COMM_WORLD** после его инициализации. Таким образом, если пользователь не желает управлять обработкой ошибок самостоятельно, то каждая ошибка в MPI обрабатывается как фатальная. Так как все вызовы MPI возвращают код ошибки, пользователь может работать с ошибками в головной программе, используя возвращенные вызовами MPI коды и выполняя подходящую программу восстановления при неуспешном вызове. В этом случае будет использоваться обработчик ошибок **MPI_ERRORS_RETURN**. Обычно

более удобно и более эффективно не проверять ошибки после каждого вызова, а иметь специализированный обработчик ошибок.

После того, как ошибка обнаружена, состояние MPI является неопределенным. Это означает, что даже если используется определенный пользователем обработчик ошибок или обработчик **MPI_ERRORS_RETURN**, не обязательно, что пользователю будет разрешено продолжить использовать MPI после того, как ошибка определена. Цель таких обработчиков состоит в том, чтобы пользователь получил определенное им сообщение об ошибке и предпринял действия, не относящиеся к MPI (такие, как очистка буферов ввода/вывода) перед выходом из программы. Реализация MPI допускает продолжение работы приложения после возникновения ошибки, но не требует, чтобы так было всегда. Обработчик ошибок MPI является скрытым объектом, связанным с дескриптором. Процедуры MPI обеспечивают создание новых обработчиков ошибок, связывают обработчики ошибок с коммуникаторами и проверяют, какой обработчик ошибок связан с коммуникатором. Существует несколько функций MPI, обеспечивающих обработку ошибок.

MPI_ERRHANDLER_CREATE(function, errhandler)
IN **function** установленная пользователем процедура обработки ошибок
OUT **errhandler** MPI обработчик ошибок (дескриптор)

```
int MPI_Errhandler_create(MPI_Handler_function *function,
                          MPI_Errhandler *errhandler)
```

```
MPI_ERRHANDLER_CREATE(FUNCTION, ERRHANDLER, IERROR)
EXTERNAL FUNCTION
INTEGER ERRHANDLER, IERROR
```

Функция **MPI_ERRHANDLER_CREATE** регистрирует процедуру пользователя в качестве обработчика исключений. Возвращает в **errhandler** дескриптор зарегистрированного обработчика исключений. В языке С процедура пользователя должна быть функцией типа **MPI_Handler_function**, которая определяется как:

typedef void (MPI_Handler_function) (MPI_Comm *, int *, ...);

Первый аргумент является идентификатором используемого коммуникатора, второй является кодом ошибки, который будет возвращен процедурой MPI, выявившей ошибку. Если процедура возвратила **MPI_ERR_IN_STATUS**, то это значит, что код ошибки возвращен в статусный объект обращения, которое запустило обработчик ошибок.

Остающиеся аргументы есть аргументы “**stdargs**”, чьи номер и значение являются зависимыми от реализации. В реализации должны быть ясно документированы эти аргументы. Адреса используются так, чтобы обработчик мог быть написан на языке Fortran.

MPI_ERRHANDLER_SET(comm, errhandler)
IN **comm** Коммуникатор для установки обработчика ошибок (дескриптор)
IN **errhandler** новый обработчик ошибок для коммуникатора (дескриптор)

int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)

MPI_ERRHANDLER_SET(COMM, ERRHANDLER, IERROR)
INTEGER COMM, ERRHANDLER, IERROR

Функция **MPI_ERRHANDLER_SET** связывает новый обработчик ошибок **errorhandler** с коммуникатором **comm** на вызывающем процессе. Обработчик ошибок всегда связан с коммуникатором.

MPI_ERRHANDLER_GET(comm, errhandler)

IN **comm**	коммуникатор, из которого получен обработчик ошибок (дескриптор)
OUT **errhandler**	MPI обработчик ошибок, связанный с коммуникатором (дескриптор)

int MPI_Errhandler_get(MPI_Comm comm, MPI_Errhandler *errhandler)

MPI_ERRHANDLER_GET(COMM, ERRHANDLER, IERROR)
INTEGER COMM, ERRHANDLER, IERROR

Функция **MPI_ERRHANDLER_GET** возвращает в **errhandler** дескриптор обработчика ошибок, связанного с коммуникатором **comm.** Пример: библиотечная функция может записать на входной точке текущий обработчик ошибок для коммуникатора, затем установить собственный частный обработчик ошибок для этого коммуникатора и восстановить перед выходом предыдущий обработчик ошибок.

MPI_ERRHANDLER_FREE(errhandler)
INOUT **errhandler** MPI обработчик ошибок (дескриптор)

int MPI_Errhandler_free(MPI_Errhandler *errhandler)

MPI_ERRHANDLER_FREE(ERRHANDLER, IERROR)
INTEGER ERRHANDLER, IERROR

void MPI::Errhandler::Free()

Эта функция маркирует обработчик ошибок, связанный с **errhandler**, для удаления и устанавливает для **errhandler** значение **MPI_ERRHANDLER_NULL**. Обработчик ошибок будет удален после того, как все коммуникаторы, связанные с ним, будут удалены.

MPI_ERROR_STRING(**errorcode, string, resultlen**)
IN **errorcode** код ошибки, возвращаемый процедурой MPI
OUT **string** текст, соответствующий **errorcode**
OUT **resultlen** длина (в печатных знаках) результата, возвращаемого в **string**

int MPI_Error_string(int errorcode, char *string, int *resultlen)

MPI_ERROR_STRING(ERRORCODE, STRING, RESULTLEN, IERROR)
INTEGER ERRORCODE, RESULTLEN, IERROR
CHARACTER*(*) STRING

void MPI::Get_error_string (int errorcode, char* name, int& resulten)

Функция **MPI_ERROR_STRING** возвращает текст ошибки, связанный с кодом или классом ошибки. Аргумент **string** обязан иметь длину не менее **MAX_ERROR_STRING** знаков. Число фактически записанных символов возвращается в выходном аргументе **resultlen**.

Коды ошибок, возвращаемых MPI, приведены в реализации MPI (за исключением **MPI_SUCCESS**). Это сделано для того, чтобы позволить реализации представить как можно больше информации об ошибках (для использования с **MPI_ERROR_STRING**).

Чтобы приложения могли интерпретировать код ошибки, процедура **MPI_ERROR_CLASS** преобразует код любой ошибки в один из кодов небольшого набора кодов стандартных ошибок, названный *классом ошибок*.

Классы ошибок являются подмножеством кодов ошибок: функция MPI может возвращать номер класса ошибки, а функция **MPI_ERROR_STRING** может использоваться, чтобы вычислить строку ошибки, связанную с классом ошибки.

Коды ошибок удовлетворяют выражению:

$$0 = MPI\_SUCCESS < MPI\_ERR\_... \le MPI\_ERR\_LASTCODE.$$

MPI_ERROR_CLASS(**errorcode, errorclass**)
IN **errorcode** код ошибки, возвращаемый процедурой MPI
OUT **errorclass** класс ошибки, связаный с **errorcode**

int MPI_Error_class(int errorcode, int *errorclass)

```
MPI_ERROR_CLASS(ERRORCODE, ERRORCLASS, IERROR)
INTEGER ERRORCODE, ERRORCLASS, IERROR

int MPI::Get_error_class(int errorcode)
```

Функция **MPI_ERROR_CLASS** отображает код каждой стандартной ошибки (класс ошибки) на себя.

Правильные классы ошибок включают:

MPI_SUCCESS	Ошибки нет
MPI_ERR_BUFFER	Неправильный указатель буфера
MPI_ERR_COUNT	Неверное количество аргумента
MPI_ERR_TYPE	Неправильный тип аргумента
MPI_ERR_TAG	Неправильный тэг аргумента
MPI_ERR_COMM	Неправильный коммуникатор
MPI_ERR_RANK	Неправильный номер
MPI_ERR_REQUEST	Неверный запрос (дескриптор)
MPI_ERR_ROOT	Неверный корневой идентификатор
MPI_ERR_GROUP	Неправильная группа
MPI_ERR_OP	Неправильная операция
MPI_ERR_TOPOLOGY	Неверная топология
MPI_ERR_DIMS	Неправильная размерность аргумента
MPI_ERR_ARG	Ошибка аргумента некоторого другого типа
MPI_ERR_UNKNOWN	Неизвестная ошибка
MPI_ERR_TRUNCATE	Неправильное округление
MPI_ERR_OTHER	Известная ошибка не из этого списка
MPI_ERR_INTERN	Внутренняя ошибка реализации MPI
MPI_ERR_IN_STATUS	Неправильный код статуса
MPI_ERR_PENDING	Зависший запрос
MPI_ERR_LASTCODE	Последний код в списке

11.2. ОТЛАДКА ПАРАЛЛЕЛЬНЫХ ПРИЛОЖЕНИЙ

Средства отладки являются необходимой принадлежностью любой системы программирования. Отладчик должен позволять: запустить программу; остановить программу в заданной точке и обеспечить в случае необходимости пошаговый просмотр программы; просмотреть значения нужных переменных; изменить некоторые части программы.

Для реализации этих функций существуют несколько общеизвестных возможностей отладки:

- Трассировка отлаживаемой программы
- Использование последовательных отладчиков

- Использование псевдопараллельных отладчиков
- Использование полноценных параллельных отладчиков

Большой срок существования ОС Unix и систем на ее основе привели к тому, что состав средств параллельной отладки для них богаче, чем для Windows NT, поэтому далее рассматриваются в основном средства отладки для систем на базе Unix.

11.2.1. Трассировка

Как и при обычной отладке, чтобы отследить последовательность событий в программе, необходимо выводить соответствующие сообщения. Трасса – журнал событий, произошедших во время выполнения программы. Для трассировки в текст программы вставляются операторы, которые позволяют получить временной срез – состояние всех процессов программы в некоторый фиксированный момент времени, выводимый на экран дисплея или на печать. Для параллельных приложений необходимо идентифицировать сообщения номером процесса, пославшего его. Это делается непосредственным помещением номера в сообщение трассировки.

В программе для вычисления значения π на языке С (параграф 1.5) это может быть, например, сделано так:

```
#include "mpi.h"
#include <math.h>
int main ( int argc, char *argv[ ] )
{ int n, myid, numprocs, i;
  double PI25DT = 3.14159265358979323846264 3;
   double mypi, pi, h, sum, x;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   printf ("begin process %d /n",myid);
   while (1)
   {    if (myid == 0) {
            printf ("Enter the number of intervals: (0 quits) ");
            scanf ("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        printf (" process %d recieve n /n",myid);
        if (n == 0)     break;
        else {
            h  = 1.0/ (double) n;
            sum = 0.0;
```

```
            for (i = myid +1; i <= n; i+= numprocs) {
                  x = h * ( (double)i – 0.5);
                  sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
            printf (" mypi in  process %d = %f /n",myid,mypi);
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                              MPI_COMM_WORLD);
            if (myid == 0)
                printf ("pi is approximately %.16f. Error is %.16f\n", pi,
                        fabs(pi – PI25DT));
        }
    }
    MPI_Finalize();
    return 0;
}
```

Каждый процесс отмечает начало работы, получение переменной **n** и вычисление частичной суммы. Эти операции в программе выделены жирным шрифтом. В результате будет получена трассировка программы, анализ которой позволит найти ошибку.

11.2.2. Использование последовательных отладчиков

Параллельная программа может содержать различные исполняемые файлы. Этот стиль параллельного программирования называется MPMD (множество программ при множестве данных). Во многих случаях программу MPMD можно преобразовать в программу SPMD (одна программа при множестве данных), которая использует номер процесса для вызова различных процедур. Это облегчает старт параллельных программ и их отладку, поскольку тексты процессов становятся идентичными. Для таких программ часто достаточно отладить единственный процесс, обычно процесс 0. Такая отладка производится обычными отладчиками, например, отладчиками dbx, gdb, xdbx для Unix или обычным отладчиком msdev studio для Windows.

В Unix существует возможность запустить один из процессов с помощью обычного отладчика. Например, команда:

```
mpirun -dbg=<name of debugger> -np 2 program
```

запускает **program** на двух машинах, но отладчик работает только на одной, а, именно, на хост машине.

Часто удобно запускать отладчик, когда в программе возникает ошибка. Если mpich сконфигурирован с опцией **-mpedbg**, то это вы-

зовет попытку mpich запустить отладчик (обычно dbx или gdb), когда возникает ошибка. Команда может, например, выглядеть следующим образом:

```
mpirun –np 4 a.out -mpedbg
```

Если преобразование программы MPMD в SPMD невозможно или принципиально необходимо отлаживать одновременно несколько ветвей (пусть идентичных по тексту), то в этом случае нельзя для запуска программ использовать mpirun. Вместо этого можно запустить процессы вручную, каждый под управлением обычного отладчика.

Опишем возможность параллельной отладки двух процессов произвольной параллельной программы program для ОС Windows.

Для организации параллельной отладки создадим два файла с расширением bat, в которых произведем установку необходимых переменных окружения для запуска приложений вручную [1] и вызов msdev studio. Файлы могут выглядеть следующим образом:

Первый файл:	**Второй файл:**
set MPICH_JOBID=parallel.2000	set MPICH_JOBID=parallel.2000
rem номер процесса = 0	rem номер процесса = 1
set MPICH_IPROC=0	set MPICH_IPROC=1
rem всего 2 процесса	rem всего 2 процесса
set MPICH_NPROC=2	set MPICH_NPROC=2
set MPICH_ROOT=parallel:12345	set MPICH_ROOT=parallel:12345
msdev program.exe	msdev program.exe

Из командной строки (command prompt) запускаем файл номер один. В результате его работы запустится msdev studio с программой **program** и номером процесса, равным **0**. Аналогично из другой командной строки (second command prompt) запускаем файл номер два. Запускается еще раз msdev studio с программой **program** и номером процесса, равным **1**. Затем вызываем в обоих окнах отладчик msdev. Переключаясь между окнами, можно шаг за шагом выполнять оба процесса. При этом имеем все необходимые возможности отладки: выполнение по шагам, установки точек прерываний, просмотр необходимых переменных и т. д.

11.2.3. Псевдопараллельный отладчик

Устройство ADI ch_p4mpd в mpich содержит “параллельный отладчик”, который состоит просто из нескольких копий отладчика gdb и механизма перенаправления stdin. Команда **mpigdb** является верси-

ей **mpirun**, оторая запускает каждый процесс под управлением gdb и управляет stdin для gdb. Команда **`z'** позволяет направить ввод с терминала в определенный процесс или разослать его всем процессам. Продемонстрируем это запуском просраммы cpi на языке Си для вычисления числа π под управлением простого отладчика:

```
donner% mpigdb -np 5 cpi                    # по умолчанию вывод от всех
(mpigdb) b 29                               # задать точку останова для  всех
0-4: Breakpoint 1 at 0x8049e93: file cpi.c, line 29.
(mpigdb) r                                  # запустить все 0-4:
Starting program:/home/lusk/mpich/examples/basic/cpi
0: Breakpoint 1, main (argc=1, argv=0xbffffa84) at cpi.c:29
1-4: Breakpoint 1, main (argc=1, argv=0xbffffa74) at cpi.c:29
0-4: 29 n = 0;                              # все достигли точки останова
(mpigdb) n                                  # пошаговый режим для всех
0: 38 if (n==0) n=100; else n=0;
1-4: 42 MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
(mpigdb) z 0                                # stdin только для процесса 0
(mpigdb) n                                  # пошаговый режим процесса 0
0: 40 startwtime = MPI_Wtime ();
(mpigdb) n                                  # до останова
0: 42 MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
(mpigdb) z                                  # stdin возвращен для всех  процесов
(mpigdb) n                                  # пошаговый режим для всех  до
                                            # интересующего места
…..
(mpigdb) n
0-4: 52 x = h * ( (double)i -0.5);
(mpigdb) p x                                # вывод от всех процессов
0: $1 = 0.0050000000000000001               # значение x процесса 0
1: $1 = 0.014999999999999999                # значение x процесса 1
2: $1 = 0.025000000000000001                # значение x процесса 2
3: $1 = 0.035000000000000003                # значение x процесса 3
4: $1 = 0.044999999999999998                # значение x процесса 4
(mpigdb) c                                  # продолжить все
0: pi is approximately  3.141600986923,  Error is 0.000008333333
0-4 Program exited normally.
(mpigdb) q                                  # выход
donner%
```

Если отлаживаемый процесс зависает (нет приглашения **mpigdb**) из-за ожидания текущим процессом действий другого процесса, ctrl-C вызовет меню, позволяющее переключать процессы. mpigdb не так развит, как параллельный отладчик TotalView, но часто полезен и свободно распространяется с mpich.

11.2.4. Отладка программ MPI с помощью TotalView

Коммерческий отладчик TotalView (TV) является переносимым отладчиком для параллельных программ [26]. TotalView – единственный встроенный высокоуровневый отладчик оконного типа, специально спроектированный для современных распределенных многозадачных систем. TotalView воспринимает множество реализаций MPI, включая mpich. С помощью TV разработчики способны быстро и точно отладить приложения с многими процессами, потоками и процессорами. TV имеет следующие возможности:

- Процессы и потоки могут быть легко запущены, остановлены, перезапущены, просмотрены и удалены.
- Пользователь может легко подключиться к любому процессу в системе простым щелчком мыши. Точки останова легко управляются и изменяются.
- Программа может быть исправлена «на лету», поэтому различные сценарии исполнения проверяются быстро и без перекомпиляции.
- Множественные процессы на множественных процессорах могут быть сгруппированы, и когда один процесс достигает точки останова, все сгруппированные процессы будут остановлены. Это может существенно помочь при отладке распределенных систем, построенных по принципу клиент-сервер.
- Распределенная архитектура TV позволяет отлаживать удаленные программы на всей сети.

Для реализации указанных возможностей в TV имеется три окна:

- Root Window – корневое окно,
- Process Window – окно процесса,
- Variable Window – вспомогательное окно.

Root Window дает обзор состояния выполняемой программы. Его также можно использовать как навигационный инструмент. Это окно появляется при запуске TV. Окно имеет следующие четыре выделенные страницы:

1. **Attached (**подключенные процессы). Показывает список всех отлаживаемых процессов и нитей.
2. **Unattached** (неподключенные процессы). Показывает процессы, над которыми TV имеет контроль. Если нельзя подключиться к какому-то процессу, например, нельзя подключиться к процессу TV, то TV изображает этот процесс серым.

3. **Groups** (группы). Представляет список групп, используемых выполняемой программой.
4. **Log** (регистрация). Окно отображает собранную отладочную информацию о процессе и потоке внутри этого процесса. Панели внутри этого окна показывают трассу стека, границу стека и код для избранной нити. В этом окне в основном и затрачивается основная часть времени отладки.

Process Window имеет пять информационных панелей:

1. **Source Pane** (панель исходного кода). Эта панель содержит исходный код. Левый край этой панели, называемый областью поля тэга, показывает номера строк и иконки, указывающие определенные свойства программы. Можно разместить точку останова на любой строке программы. При этом TV заменяет номер строки на иконку STOP. Стрелка в поле тэгов указывает текущее положение программного счетчика.
2. **Threads Pane** (панель нитей) – показывает список нитей, которые существуют в процессе. Когда выбирается некоторая нить в этом процессе, TV изменяет содержимое Stack Trace Pane, Stack Frame Pane и Source Pane, показывая информацию по этой нити.
3. **Stack Trace Pane** (панель трассы стека) показывает стек вызова процедур, которые выполняет выделенная нить.
4. **Stack Frame Pane** (панель фрейма стека) показывает функциональные параметры, локальные переменные и регистры для избранного фрейма стека. Информация, представляемая панелями Stack Trace и Stack Frame, отражает состояние процесса на момент его последней остановки. Соответственно, эта информация не корректируется во время выполнения нити.
5. **Action Points Pane (**панель активных точек) показывает список точек останова, точек вычислений и точек наблюдения за процессом.

Variable Window (переменное окно) содержит список адресов, типов данных, значения локальных переменных, регистров или глобальных переменных. Окно также показывает значения, хранимые в памяти.

КОНТРОЛЬНЫЕ ВОПРОСЫ К ГЛАВЕ 11

Контрольные вопросы к 11.1

1. Что такое ошибки времени исполнения?
2. Какие предопределенные обработчики ошибок доступны в MPI?

3. Опишите возможности предопределенных обработчиков ошибок.
4. Какие функции используются для создания новых обработчиков ошибок?
5. Что такое классы ошибок?

Контрольные вопросы к 11.2

1. Назовите основные функции отладчика.
2. Какие способы отладки параллельных приложений используются на практике?
3. Что такое трассировка?
4. Как использовать последовательный отладчик для параллельных приложений?
5. Как можно использовать последовательный отладчик для отладки нескольких параллельных процессов, выполняемых под Windows NT?
6. Что такое псевдопараллельные отладчики и как они используются?
7. Опишите возможности параллельного отладчика программ MPI TotalView.

Глава 12. ЭФФЕКТИВНОСТЬ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

12.1. АНАЛИТИЧЕСКАЯ ОЦЕНКА ЭФФЕКТИВНОСТИ ВЫЧИСЛЕНИЙ

Время выполнения параллельной программы зависит от многих факторов: от параллелизма математического алгоритма, от его программного представления, от возможностей и качества реализации языка параллельного программирования, количества процессоров, объема памяти, размещения данных по процессорам, быстродействия коммуникационного оборудования и от множества других обстоятельств. При некоторых упрощениях ускорение вычислений характеризуется сетевым законом Амдала (глава 1):

$$R_c = \frac{1}{a + \frac{1-a}{n} + C}, \tag{12.1}$$

где

$$C = \frac{W_c \cdot t_c}{W \cdot t} = C_A \cdot C_T, \tag{12.2}$$

есть коэффициент сетевой деградации. При этом $C_A = Wc/W$ определяет алгоритмическую составляющую коэффициента деградации, обусловленную свойствами алгоритма, а C_T – техническую составляющую, которая зависит от соотношения технического быстродействия процессора и аппаратуры сети. Таким образом, для повышения скоро-

сти вычислений следует воздействовать на обе составляющие коэффициента деградации. Как правило, обе составляющие коэффициента деградации C известны заранее для многих алгоритмов[14] и аппаратных средств, или их можно при некоторых упрощениях относительно легко вычислить. В качестве примера рассмотрим сетевую эффективность вычислений умножения матрицы на вектор и умножение матрицы на матрицу [4].

Умножение матрицы на вектор. Предположим, что матрица A квадратная, размера $m*m$. Тогда для каждого элемента результирующего вектора C, получаемого от перемножения матрицы A на вектор b, нужно выполнить m умножений, и m-1 сложений. Имеется m элементов вектора C, поэтому общее количество операций с плавающей точкой будет

$$m * (m + (m - 1)) = 2m^2 - m. \qquad (12.3)$$

Для простоты предполагаем, что на сложение и умножение затрачивается одинаковое количество времени, равное $T_{выч}$. Далее предполагаем, что полное время вычисления в основном зависит от операций с плавающей точкой. Таким образом, грубая оценка времени на вычисления равна $(2m^2 - m) * T_{выч}$.

Оценим затраты на коммуникацию. Не будем учитывать затраты на посылку b каждому подчиненному процессу, предположим, что он прибывает туда другим способом (например, может быть там вычислен). Тогда количество чисел с плавающей запятой, которые должны быть переданы, равно m (чтобы послать строку матрицы A), + 1 (чтобы послать ответ назад) для каждого из m элементов C, что дает в результате

$$m \times (m + 1) = m^2 + m. \qquad (12.4)$$

Если предположить, что время, требуемое на сообщение числа с плавающей запятой равняется $T_{КОМ}$, то полное время связи примерно равно $(m^2 + m) * T_{КОМ}$. Поэтому отношение коммуникации к вычислению равно

$$C = C_A \times C_T = \left(\frac{m^2 + m}{2m^2 - m} \right) \times \left(\frac{T_{ком}}{T_{выч}} \right) \qquad (12.5)$$

Предположим, что:

1. Для больших m (именно для таких задач нужны параллельные ЭВМ) можно пренебречь m по сравнению с m^2.

2. В качестве коммуникационной сети используется сеть Fast Ethernet с пропускной способность V = 10 Мбайт/сек.
3. Объединенное время $t_{ОП}$ плавающей операции (умножение, сложение, 2 обращения в память) требует 20 нс при частоте процессора 500 Мгц.
4. Длина слова с плавающей запятой может составлять 10 байтов.

Тогда согласно выражению (12.5) получаем:

$$C = C_A * C_T = \frac{1}{2} * \frac{T_{ком}}{T_{выч}} = \frac{1}{2} * \frac{10}{V * t_{оп}} = \frac{1}{2} * \frac{10}{10^7 * 20 * 10^{-9}} = 25 \qquad (12.6)$$

Следовательно, даже при идеальном распараллеливании (a=0)

$$R_c = \frac{1}{1/n + 25} , \qquad (12.7)$$

матрично-векторное умножение всегда будет выполняться в многопроцессорной системе с замедлением по отношению к однопроцессорному варианту. Это же верно и для самопланирующего алгоритма умножения матриц, так как здесь основной операцией также является пересылка одной строки и умножение этой строки на один столбец.

Необходимы другие алгоритмы матрично-векторного умножения, обеспечивающие ускорение.

Умножение матрицы на матрицу. Для матрично-матричного умножения вектор b становится матрицей B, которая копируется в каждом процессе, и после вычислений каждый процесс посылает обратно целую строку конечной матрицы C.

Пусть A и B – квадратные матрицы. Тогда число операций для каждого элемента матрицы C будет (как и прежде) равно m умножений и m-1 сложений, но теперь вычисляется m^2 элементов, а не m. Поэтому число операций с плавающей запятой равно

$$m^2 \times (2m\text{-}1) = 2m^3 - m^2 . \qquad (12.8)$$

Количество чисел с плавающей запятой, передаваемых каждой строке, равно m (чтобы послать строку матрицы A плюс m, чтобы послать строку матрицы C назад), и имеются m строк, отсюда ответом является $m * 2m$. Тогда отношение коммуникации к вычислению можно вычислить по формуле 12.9, результат стремится к $1/m$ при увеличении m. Поэтому для этой задачи мы должны ожидать, что коммуникационные перерасходы будут играть все меньшую роль при увеличении размера задачи.

$$\left(\frac{2m^2}{2m^3 - m^2}\right) \times \left(\frac{T_{ком}}{T_{выч}}\right) \qquad (12.9)$$

Тогда с помощью рассуждений, которые проводились выше для векторно-матричного умножения, учитывая формулу (12.9) и необходимость посылки равного объема данных туда и обратно, получаем:

$$C = C_A * C_T = \frac{1}{m} * \frac{10}{V * t_{on}} = \frac{1}{m} * \frac{10}{V * 20 * 10^{-9}} \qquad (12.10)$$

Поскольку для коммуникационной системы SCI (Scalable Coherent Interface) V = 80 Мбайт/сек, то для сетей Fast Ethernet (FE) и SCI получаем:

$$R_c^{FE} = \frac{1}{\frac{1}{n} + \frac{100}{m}}, \quad (12.11) \qquad R_c^{SCI} = \frac{1}{\frac{1}{n} + \frac{12.5}{m}}. \qquad (12.12)$$

Ниже для операции умножения матрицы на матрицу приводятся два графика (рис. 12.1, 12.2), построенные по выражениям (12.11) и (12.12). Справа на графиках указано число процессоров в порядке расположения графиков (для n = 32 график и значение n расположены вверху).

Приведенные графики позволяют сделать следующие выводы:

Ускорение существенно зависит от объема вычислений, вызванного однократной передачей данных. Согласно (12.7) для самопланирующего алгоритма это соотношение неудовлетворительное, поэтому при любом числе процессоров происходит замедление, а не ускорение вычислений. В алгоритме с дублированием матрицы B объем вычислений при однократной передаче данных велик и повышается при увеличении размера матриц. Это видно из обоих графиков.

Сравнение графиков для Fast Ethernet и SCI также показывает сильную зависимость ускорения от пропускной способности коммуникационной сети, причем для SCI высокое ускорение достигается при меньших размерах матриц.

Реальные данные по производительности 36-процессорного кластера SCI и 40-процессорного кластера SKY, установленных в НИВЦ МГУ, приведены в [27].

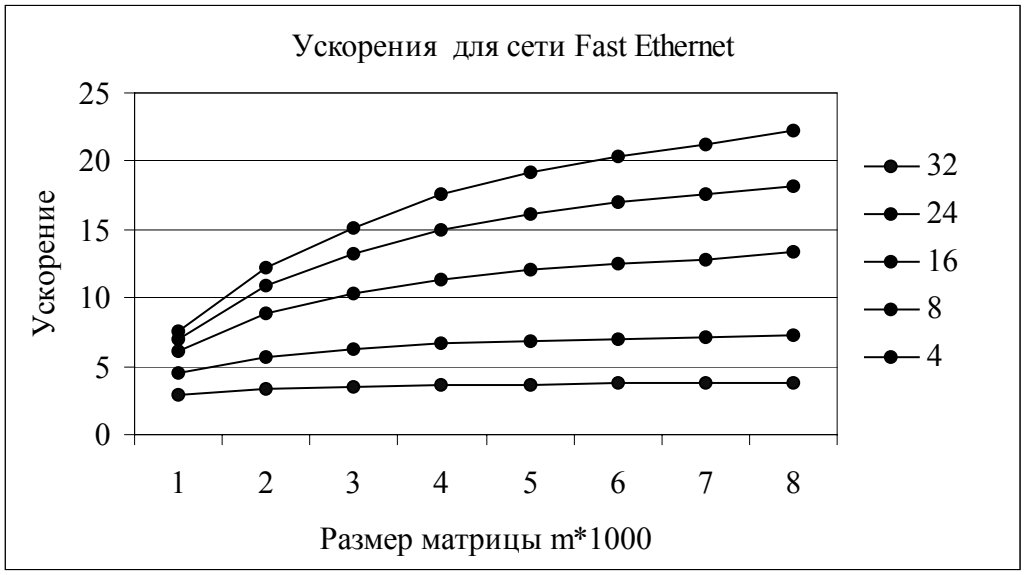


Рис. 12.1. Зависимость ускорения от размера матриц и числа процессоров для Fast Ethernet

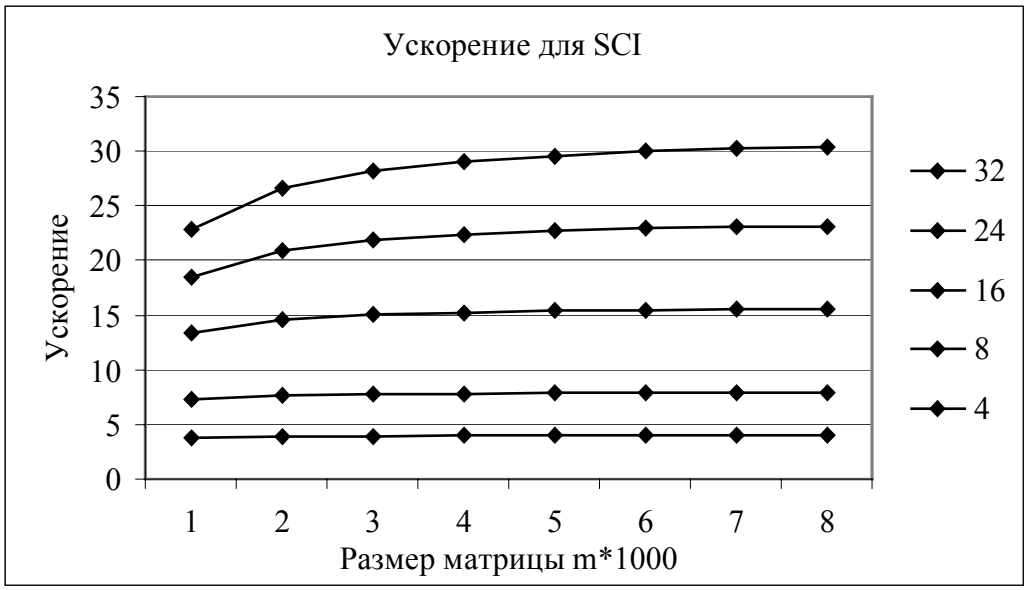


Рис. 12.2. Зависимость ускорения от размера матриц и числа процессоров SCI

12.2. СПОСОБЫ ИЗМЕРЕНИЯ ЭФФЕКТИВНОСТИ ВЫЧИСЛЕНИЙ

Последовательные программы тестируют, чтобы убедиться, дают ли они правильный результат. Для параллельных программ этого недостаточно – для них необходимо определять ускорение, а следовательно, необходимо измерять время выполнения параллельной про-

граммы. MPI имеет простые функции для измерения времени выполнения программы или ее частей.

Функция **MPI_Wtime** возвращает число с плавающей запятой, которое является текущим отсчетом времени в секундах. Следовательно, чтобы узнать время выполнения отрезка программы, нужно выполнить **MPI_Wtime** в начале и в конце отрезка, а затем вычесть отсчеты. Для получения отсчетов используются таймеры высокого разрешения. Если их нет в конкретном процессоре, то и измерения невозможны. Программа для вычисления значения π для языка Си, использующая функцию **MPI_Wtime**, представлена в параграфе 1.5.

Функция **MPI_Wtick** не имеет никаких аргументов. Она возвращает число с плавающей точкой, которое указывает время между двумя последовательными тактами (период) тактового генератора.

В MPI используются и более мощные средства измерения и анализа эффективности параллельных вычислений. Они связаны с созданием логфайлов, в которых регистрируются заранее заданные пользователем события. Затем производится анализ и визуализация результатов регистрации. Все эти средства реализованы в основном в библиотеке MPE и представлены в параграфах 2.3 и 2.4.

12.3. ИНТЕРФЕЙС ПРОФИЛИРОВАНИЯ

Профилем некоторого процесса или объекта называют совокупность его основных параметров. Профилирование – это процесс сбора этих параметров. В главе 2 и предыдущем параграфе были представлены некоторые фиксированные для MPICH и MPE методы профилирования. Однако получаемый с помощью этих методов набор результатов не всегда удовлетворяет пользователей. Поэтому разработчики MPI предоставили непосредственно пользователю вариант профилирующего механизма, с помощью которого он сам сможет запрограммировать сбор любой нужной ему информации.

Чтобы обеспечить это, реализация MPI должна:

1. Обеспечить механизм, с помощью которого ко всем определенным в MPI функциям можно обращаться по смещенному имени. Таким образом все функции MPI**,** которые начинаются с префиксом “**MPI_**“, должны быть также доступны с префиксом “**PMPI_**”.
2. Гарантировать, что не измененные функции также можно включать в исполнительный файл без конфликтов в отношении имен.
3. Обеспечивать подпрограмму **MPI_PCONTROL** холостой командой.

Если реализация MPI отвечает вышеупомянутым требованиям, система профилирования в состоянии перехватить все запросы MPI, сделанные в соответствии с программой пользователя. Затем можно собрать любую требуемую информацию перед обращением к основной реализации MPI (по ее имени, смещенному по отношению к входной точке) для достижения желаемого результата.

Программа пользователя должна иметь возможность управлять профилированием во время своего исполнения. Обычно это необходимо для достижения следующих целей: для активизации профилирования и его прекращения в зависимости от состояния вычислений; для очистки буферов трассировки в некритических точках вычисления и для добавления пользовательских событий в файл трассировки.

Эти требования удовлетворяются при использовании функции **MPI_PCONTROL**.

MPI_PCONTROL(level, ...)
IN **level** уровень профилирования

```
int MPI_Pcontrol(const int level, ...)

MPI_PCONTROL(LEVEL)
INTEGER LEVEL, ...

void Pcontrol(const int level, ...)
```

MPI библиотеки непосредственно не используют эту подпрограмму, они просто возвращаются немедленно к коду пользователя. Однако возможность обращения к этой процедуре позволяет пользователю явно вызвать блок профилирования.

Поскольку MPI не контролирует реализацию профилирующего кода, нельзя точно определить семантику, которая будет обеспечивать вызовы **MPI_PCONTROL**. Эта неопределенность распространяется на число аргументов функции и на их тип.

Однако чтобы обеспечить некоторый уровень переносимости кода пользователя относительно различных библиотек профилирования, необходимо, чтобы определенные значения уровня имели следующий смысл:

- **level==0** Профилирование не используется.
- **level==1** Профилирование установлено для нормального по умолчанию уровня детализации.

- **level==2** Буфера профилирования очищены (в некоторых системах профилирования это может быть холостая команда).
- Все другие значения **level** имеют эффекты, определенные библиотекой профилирования и дополнительными аргументами.

Также требуется, чтобы после выполнения **MPI_INIT** по умолчанию было установлено профилирование (как если бы была выполнена операция **MPI_PCONTROL** с аргументом 1). Это позволяет пользователю связаться с библиотекой профилирования и получить результат профилирования без модификации их исходного текста.

Рассмотрим пример автоматического создания логфайлов для измерения времени выполнения всех операций **MPI_Bcast** в программе вычисления числа π (параграф 2.3).

```
Int MPI_Bcast(void*buf, int count, MPI_Datatype, int root, MPI_Comm comm)
{  int result;
   MPE_log_event (S_BCAST_EVENT, Bcast_ncalls, (char*)0);
   result = PMPI_Bcast(buf, count, datatype, root, comm);
   MPE_Log_event(E_BCAST_EVENT, Bcast_ncalls, (char*)0);
   return result;
}
```

Идея состоит в том, чтобы выполнить перехват вызовов на *этапе компановки*, а не на этапе компиляции. Стандарт MPI требует, чтобы каждая процедура MPI могла быть вызвана по альтернативному имени. В частности, каждая процедура вида MPI_xxx должна также вызываться по имени xxx. Более того, пользователю должно быть позволено использовать свою собственную версию MPI_xxx.

Эта схема позволяет пользователю написать некоторое число оболочек для процедур MPI и выполнить некоторые действия внутри этих оболочек. Чтобы вызвать реальную процедуру MPI, следует обозначить ее префиксом PMPI_. Необходимо только гарантировать, что эта версия MPI_Bcast является единственной, которая используется компановщиком для обращения к ней из прикладного кода. Эта процедура вызывает PMPI_Bcast, чтобы выполнить нормальную работу. Последовательность библиотек, используемых компановщиком, показана на рис.12.3.

Механизм профилирования MPI позволил создать ряд библиотек, предназначенных для анализа эффективности вычислений. Библиотеки эти входят в состав MPE и частично описаны в параграфе 2.3.

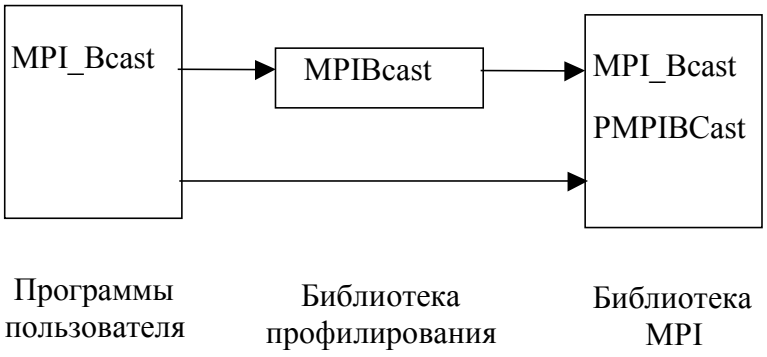


Рис. 12.3. Выполнение процедур с использованием библиотек

Для каждой из этих библиотек процессы построения подобны. Сначала должны быть написаны профилирующие версии MPI_Init и MPI_Finalize. Профилирующие версии других процедур MPI подобны по стилю. Код каждой из них выглядит как:

```
int MPI_Xxx (...)
{  сделать что-либо для профилирующей библиотеки
   retcode = PMPI_Xxx ( . . . );
   сделать что-либо еще для профилирующей библиотеки
   return retcode;
}
```

Эти процедуры создаются только написанием частей “сделать что-либо”, а затем обрамляются автоматически вызовами PMPI_. Поэтому генерация профилирующих библиотек очень проста. Детали генерации находятся в MPICH в подкаталоге `mpe/profiling/lib'.

КОНТРОЛЬНЫЕ ВОПРОСЫ К ГЛАВЕ 12

Контрольные вопросы к 12.1

1. Что такое сетевой закон Амдала?
2. Объясните смысл составляющих c_A и c_T коэффициента сетевой деградации.
3. Почему при умножении матрицы на вектор имеет место замедление?
4. В чем состоят характерные различия графиков 12.1 и 12.2?

Контрольные вопросы к 12.2

1. Для чего предназначена функция MPI_Wtime?
2. На примере программы вычисления числа π расскажите о способе использования функции MPI_Wtime.
3. Дайте перечень средств измерения эффективности?
4. Что такое библиотека MPE?

Контрольные вопросы к 12.3

1. Что такое профилирование?
2. Какие предопределенные средства профилирования Вы знаете?
3. Зачем нужен пользовательский интерфейс профилирования?
4. Определите назначение функции MPI_PCONTROL(level, ...).
5. Объясните пример профилирования, представленный программой и рис.12.3.
6. Какие библиотеки используются в этом примере?
7. Объясните способ автоматической генерация профилирующих библиотек.

Глава 13. ПАРАЛЛЕЛЬНЫЕ БИБЛИОТЕКИ

Существует достаточно большое количество библиотек, содержащих оптимизированные наборы программ различного назначения, предназначенных для систем с распределенной памятью. Среди них выделяются две библиотеки, связанные с MPI:

- ScaLAPACK (Scalable LAPACK) – для параллельного решения разнообразных задач линейной алгебры,
- PETSc (Portable Extensible Toolkit for Scientific Computation) – для параллельного решения линейных и нелинейных систем уравнений, возникающих при дискретизации уравнений в частных производных.

13.1. БИБЛИОТЕКА SCALAPACK

Основой для построения ScaLAPACK послужили следующие библиотеки, которые в большинстве своем используются на нижних уровнях иерархии организации ScaLAPACK [28].

BLAS (Basic Linear Algebra Subprograms) – библиотека высококачественных процедур, “строительными блоками” для которых являются вектора, матрицы или их части. Уровень 1 BLAS используется для выполнения операций вектор-вектор, уровень 2 BLAS – для выполнения матрично-векторных операций, наконец, уровень 3 BLAS – для выполнения матрично-матричных операций. Поскольку программы BLAS являются эффективными, переносимыми и легко доступными, они используются как базовые компоненты для развития высококачественного программного обеспечения для задач линейной алгебры. Так были созданы широко известные пакеты LINPACK, EISPACK, которые затем были перекрыты пакетом LAPACK, получившим большое распространение.

LAPACK – набор реализаций “продвинутых” методов линейной алгебры. LAPACK содержит процедуры для решения систем линей-

ных уравнений, задач нахождения собственных значений и факторизации матриц. Обрабатываются заполненные и ленточные матрицы, но не разреженные матрицы общего вида. Во всех случаях можно обрабатывать действительные и комплексные матрицы с одиночной и удвоенной точностью. Процедуры LAPACK построены таким образом, чтобы как можно больше вычислений выполнялось с помощью обращений к BLAS с использованием всех его трех уровней. Вследствие крупнозернистости операций уровня 3 BLAS, их использование обеспечивает высокую эффективность на многих высокопроизводительных компьютерах, в частности, если производителем создана специализированная реализация. ScaLAPACK является успешным результатом переноса пакета LAPACK на системы с передачей сообщений. Но в таких системах необходимы процедуры, обеспечивающие обмен данными между взаимодействующими процессами. Для этого в ScaLAPACK используется библиотека BLACS.

BLACS (Basic Linear Algebra Communication Subprograms) – набор базовых коммуникационных процедур линейной алгебры, созданных для линейной алгебры. Вычислительная модель состоит из одномерной или двумерной решетки процессов, где каждый процесс содержит часть матрицы или вектора. BLACS включает синхронизируемые send/receive процедуры для передачи матриц или подматриц от одного процесса другому, для передачи подматриц многим процессам, или для вычисления глобальной редукции (sum, max или min). Имеются процедуры для создания, изменения или опроса решетки процессов. Поскольку несколько ScaLAPACK алгоритмов требуют операций broadcasts или reductions среди различных подмножеств процессов, BLACS позволяет процессу быть членом нескольких перекрывающихся или изолированных решеток процессов, каждая из которых связана с контекстом. BLACS используется как коммуникационный слой проекта ScaLAPACK, который переносит библиотеку LAPACK на машины с распределенной памятью. Пример операции посылки данных в BLACS, которая аналогична операции SEND в MPI:

vTRSD2D(ICONTXT, UPLO, DIAG, M, N, A, LDA, RDEST, CDEST),

где **ICONTXT** – целочисленный дескриптор, указывающий на контекст, **M** – количество обрабатываемых строк матрицы, **N** – число обрабатываемых столбцов матрицы, **A** – указатель на начало посылаемого (под)массива, **LDA** – расстояние между двумя элементами в строке матрицы, **UPLO** указывает, является треугольная матрица

верхней или нижней, **DIAG** указывает, является ли диагональ матрицы unit diagonal, **RDEST** – координаты строки принимающего процесса, **CDEST** – координаты столбца принимающего процесса.

Подпрограммы библиотеки ScaLAPACK разделяются на три категории: *драйверные* подпрограммы, каждая из которых решает некоторую законченную задачу, например решение системы линейных алгебраических уравнений; *вычислительные* подпрограммы выполняют отдельные подзадачи, например LU разложение матрицы; *служебные* подпрограммы выполняют некоторые внутренние вспомогательные действия.

Имена всех драйверных и вычислительных подпрограмм совпадают с именами соответствующих подпрограмм из пакета LAPACK, с той лишь разницей, что в начале имени добавляется символ P, указывающий на то, что это параллельная версия. Соответственно, принцип формирования имен подпрограмм имеет ту же самую схему, что и в LAPACK. В соответствие с этой схемой имена подпрограмм пакета имеют вид **PTXXYYY**, где **T** – код типа исходных данных; **XX** – вид матрицы: ленточные, общего вида, трехдиагональные и т.д.; **YYY** – выполняемые действия данной подпрограммой: факторизация; решение СЛАУ, вычисление сингулярных значений и др.

Библиотека ScaLAPACK требует, чтобы все объекты (векторы и матрицы), являющиеся параметрами подпрограмм, были предварительно распределены по процессорам. Исходные объекты классифицируются как глобальные объекты, и параметры, описывающие их, хранятся в специальном описателе объекта – дескрипторе. Дескриптор некоторого распределенного по процессорам глобального объекта представляет собой массив целого типа, в котором хранится вся необходимая информация об исходном объекте. Части этого объекта, находящиеся в каком-либо процессоре, и их параметры являются локальными данными. Для того, чтобы воспользоваться драйверной или вычислительной подпрограммой из библиотеки ScaLAPACK, необходимо выполнить 4 шага:

1. **Инициализировать сетку процессоров**. Инициализация сетки процессоров выполняется с помощью подпрограмм из библиотеки BLACS. Вызов **CALL BLACS_PINFO (IAM, NPROCS)** инициализирует библиотеку BLACS, устанавливает некоторый стандартный контекст для ансамбля процессоров (аналог коммуникатора в MPI), сообщает процессору его номер в ансамбле (**IAM**) и количество доступных задаче процессоров (**NPROCS**).

2. **Распределить матрицы на сетку процессоров.** Точное значение того, сколько строк и столбцов должно находится в каждом процессоре, позволяет вычислить подпрограмма-функция из вспомогательной библиотеки:

   NP = NUMROC (M, MB, MYROW, RSRC, NPROW)
   NQ = NUMROC (N, NB, MYCOL, CSRC, NPCOL),

   здесь **NP** – число строк локальной подматрицы в процессоре; **NQ** – число столбцов локальной подматрицы в процессоре. Входные параметры: **M**, **N** число строк и столбцов исходной матрицы; **MB**, **NB** – размеры блоков по строкам и по столбцам; **MYROW**, **MYCOL** –координаты процессора в сетке процессоров; **IRSRC**, **ICSRC** – координаты процессора, начиная с которого выполнено распределение матрицы (подразумевается возможность распределения не по всем процессорам); **NPROW**, **NPCOL** – число строк и столбцов в сетке процессоров.
3. **Вызвать вычислительную подпрограмму.** Вызов подпрограммы вычислений рассмотрим на примере решения систем линейных алгебраических уравнений с матрицами общего вида. Имя подпрограммы **PDGESV** указывает, что: тип матриц – double precision (второй символ D); матрица общего вида, т. е. не имеет ни симметрии, ни других специальных свойств (3-й и 4-й символы GE); подпрограмма выполняет решение системы линейных алгебраических уравнений $A * X = B$ (последние символы SV). Обращение к подпрограмме и ее параметры имеют вид:

   CALL PDGESV(N, NRHS, A, IA, JA, DESCA, IPIV, B,
   IB, JB, DESCB, INFO),

   где **N** – размерность исходной матрицы **A** (полной); **NRHS** – количество правых частей в системе (сколько столбцов в матрице **B**); **A** – на входе локальная часть распределенной матрицы **A**, на выходе локальная часть LU разложения; **IA**, **JA** – индексы левого верхнего элемента подматрицы матрицы **A**, для которой находится решение (т. е. подразумевается возможность решать систему не для полной матрицы, а для ее части); **DESCA** – дескриптор матрицы **A** (подробно рассмотрен выше); **IPIV** – рабочий массив целого типа, который на выходе содержит информацию о перестановках в процессе факторизации, длина массива должна быть не меньше количества строк в локальной подматрице; **B** – на входе локальная часть распределенной матрицы **B**, на выходе локальная часть по-

лученного решения (если решение найдено); **IB**, **JB** – то же самое, что **IA**, **JA** для матрицы **A**; **DESCB** –дескриптор матрицы **B**; **INFO** – целая переменная, которая на выходе содержит информацию о том, успешно или нет завершилась подпрограмма, и причину аварийного завершения.

4. **Освободить сетку процессоров.** Программы, использующие пакет ScaLAPACK, должны заканчиваться закрытием всех BLACS процессов, то есть освобождением сетки процессоров. Это осуществляется с помощью вызова подпрограммы **BLACS_EXIT (0).**

Следующий пример демонстрирует возможности пакета (заимствован из [29]) ScaLAPACK. Используется подпрограмма **PDGEMM** из PBLAS (комбинация библиотек BLAS и BLACS), которая выполняет матричную операцию $C = aA*B + bC$, где A, B и C – матрицы, a и b – константы. В нашем случае полагаем $a = 1$, $b = 0$.

```
   program abcsl
   include 'mpif.h'
!  параметр nm определяет максимальную размерность блока матрицы
!  на одном процессоре, массивы описаны как одномерные
   parameter (nm = 1000, nxn = nm*nm)
   double precision a(nxn), b(nxn), c(nxn), mem(nm)
   double precision time(6), ops, total, t1
!  параметр NOUT – номер выходного устройства (терминал)
   PARAMETER ( NOUT = 6 )
   DOUBLE PRECISION ONE
   PARAMETER ( ONE = 1.0D+0 )
   INTEGER DESCA(9), DESCB(9), DESCC(9)
!  Инициализация BLACS
   CALL BLACS_PINFO( IAM, NPROCS )
!  вычисление формата сетки процессоров, наиболее близкого к квадратному
   NPROW = INT(SQRT(REAL(NPROCS)))
   NPCOL = NPROCS/NPROW
!  считывание параметров решаемой задачи 0-м процессором и печать этой
!  информации ( N – размер матриц и NB – размер блоков )
   IF( IAM.EQ.0 ) THEN
      WRITE(*,* ) ' Input N and NB: '
      READ( *, * ) N, NB
      WRITE( NOUT, FMT = * )
      WRITE( NOUT, FMT = 9999 ) 'The following parameter values will be used:'
      WRITE( NOUT, FMT = 9998 ) 'N ', N
      WRITE( NOUT, FMT = 9998 ) 'NB ', NB
      WRITE( NOUT, FMT = 9998 ) 'P ', NPROW
      WRITE( NOUT, FMT = 9998 ) 'Q ', NPCOL
```

```
      WRITE( NOUT, FMT = * )
    END IF
!  Рассылка считанной информации всем процессорам
    call MPI_BCAST(N, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
    call MPI_BCAST(NB,1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
!  теоретическое количество операций при умножении двух квадратных матриц
    ops = (2.0d0*dfloat(n)-1)*dfloat(n)*dfloat(n)
!  инициализация сетки процессоров
    CALL BLACS_GET( -1, 0, ICTXT )
    CALL BLACS_GRIDINIT( ICTXT, 'Row-major', NPROW, NPCOL )
    CALL BLACS_GRIDINFO( ICTXT, NPROW, NPCOL, MYROW, MYCOL )
!  если процессор не вошел в сетку, то он ничего не делает;
!  такое может случиться, если заказано, например, 5 процессоров
    IF( MYROW.GE.NPROW .OR. MYCOL.GE.NPCOL )  GO TO 500
!  вычисление реальных размеров матриц на процессоре
    NP = NUMROC( N, NB, MYROW, 0, NPROW )
    NQ = NUMROC( N, NB, MYCOL, 0, NPCOL )
!  инициализация дескрипторов для 3-х матриц
    CALL DESCINIT( DESCA, N, N, NB, NB, 0, 0, ICTXT, MAX(1,NP ), INFO )
    CALL DESCINIT( DESCB, N, N, NB, NB, 0, 0, ICTXT, MAX(1,NP ), INFO )
    CALL DESCINIT( DESCC, N, N, NB, NB, 0, 0, ICTXT, MAX(1,NP ), INFO )
    lda = DESCA(9)
!  вызов процедуры генерации матриц A и B
    call pmatgen(a, DESCA, np, nq, b, DESCB, nprow, npcol, myrow, mycol)
    t1 = MPI_Wtime()
!  вызов процедуры перемножения матриц
    CALL PDGEMM('N','N', N, N, N, ONE, A, 1, 1, DESCA,
                              B, 1, 1, DESCB, 0. 0, C, 1, 1, DESCC)
    time(2) = MPI_Wtime() – t1
!  печать угловых элементов матрицы C с помощью служебной подпрограммы
    if (IAM.EQ.0) write(*,*) 'Matrix C...'
    CALL PDLAPRNT( 1, 1, C, 1, 1, DESCC, 0, 0, 'C', 6, MEM )
    CALL PDLAPRNT( 1, 1, C, 1, N, DESCC, 0, 0, 'C', 6, MEM )
    CALL PDLAPRNT( 1, 1, C, N, 1, DESCC, 0, 0, 'C', 6, MEM )
    CALL PDLAPRNT( 1, 1, C, N, N, DESCC, 0, 0, 'C', 6, MEM )
!  вычисление времени, затраченного на перемножение,
!  и оценка производительности в Mflops.
    total = time(2)
    time(4) = ops/(1.0d6*total)
    if (IAM.EQ.0) then
       write(6,80) lda
 80 format(' times for array with leading dimension of',i4)
      write(6,110) time(2), time(4)
 110 format(2x,'Time calculation: ',f12.4, ' sec.', ' Mflops = ',f12.4)
    end if
!  Закрытие BLACS процессов
```

```
   CALL BLACS_GRIDEXIT( ICTXT )
   CALL BLACS_EXIT(0)
 9998 FORMAT( 2X, A5, '   :   ', I6 )
 9999 FORMAT( 2X, 60A )
  500 continue
   stop
   end

   subroutine pmatgen(a,DESCA,np,nq,b,DESCB,nprow,npcol,myrow,mycol)
   integer n, i, j, DESCA(*), DESCB(*), nprow, npcol, myrow, mycol
   double precision a(*), b(*)
   nb = DESCA(5)
! Заполнение локальных частей матриц A и B,
! матрица A формируется по алгоритму A(I,J) = I, a
! матрица B(I,J) = 1./J здесь имеются в виду глобальные индексы.
   k = 0
   do 250 j = 1,nq
       jc = (j-1)/nb
       jb = mod(j-1,nb)
       jn = mycol*nb + jc*npcol*nb + jb + 1
      do 240 i = 1,np
          ic = (i-1)/nb
          ib = mod(i-1,nb)
          in = myrow*nb + ic*nprow*nb + ib + 1
          k = k + 1
        a(k) = dfloat(in)
        b(k) = 1.D+0/dfloat(jn)
 240 continue
 250 continue
   return
   end
```

13.2. БИБЛИОТЕКА PETSC

Библиотека PETSc [30] облегчает разработку крупномасштабных приложений на языках Fortran, C и C++, является мощным средством для численного решения дифференциальных уравнений в частных производных (ДУЧП) и связанных с этим проблем на быстродействующих компьютерах. PETSc использует MPI стандарт для взаимодействия процессов. Библиотека PETSc включает различные компоненты (подобно классам в C++), каждый компонент имеет дело с частным семейством объектов (например, векторами) и операциями, которые нужно выполнять над этими объектами. Объекты и операции в PETSc определены на основе долгого опыта научных вычислений. Некоторые основные объекты PETSc представлены на рис. 13.1.

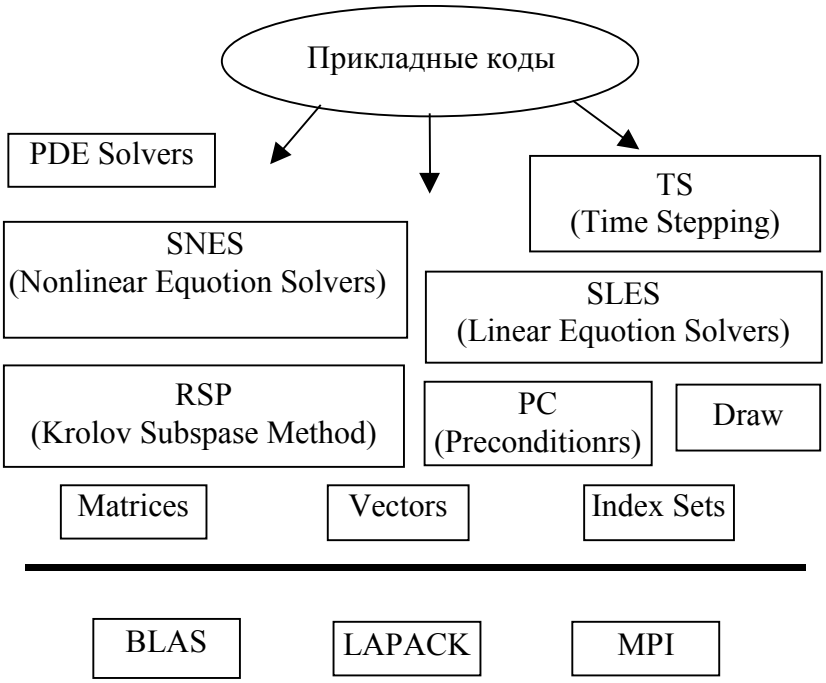


Рис. 13.1. Организация библиотеки PETSc

Библиотеки BLAS, LAPACK и MPI уже были описаны ранее. Состав векторных и матричных операций будет представлен несколько позже. Индексные ряды (Index Sets) являются обобщением рядов целочисленных индексов, которые используются для определения элементов векторов и матриц при выполнении операций рассылки и сборки. Частным случаем такого ряда является обычный список координат необходимых элементов.

На основе нижеуказанных программ строятся программы более высокого уровня абстракции, которые в свою очередь являются базой следующего уровня программ: программ – решателей.

Для подпространств Крылова (KSP) программно реализованы методы сопряженных градиентов, обобщенных сопряженных градиентов, би-сопряженных градиентов, методы Чебышева, Ричардсона и другие. Программы предобуславливателей (PC) используют методы полной и неполной LU-факторизации, методы Якоби, аддитивный метод Шварца и другие. PETSc содержит небольшой набор средств для рисования простых фигур (Draw). Этот набор не может конкурировать с более совершенными системами графики, но его наличие в PETSc повышает удобство использования последнего.

Программы SLES являются сердцем PETSc, поскольку обеспечивают единообразный и эффективный доступ ко всем пакетам решения линейных систем, включая параллельные и последовательные, прямые и итерационные. Программы SNES предназначены для решения нелинейных систем и используют верхние уровни SLES.

Все перечисленные ранее уровни используются для решения систем ДУЧП (PDE Solvers). Компонент TS (Time Stepping) обеспечивает решение дифференциальных уравнений, возникающих при дискретизации времязависимых PDEs.

Более полную информацию о возможностях можно получить в “PETSc Users Manual” [30].

Поскольку PETSc использует MPI для всех межпроцессорных обменов, пользователь свободен в использовании процедур MPI в своем приложении. Однако пользователь изолирован от многих деталей передачи сообщений внутри PETSc, поскольку они невидимы внутри параллельных объектов, таких как вектора, матрицы и методы решения систем. Кроме того, PETSc обеспечивает средства, такие как обобщенные векторные scatters/gathers и распределенные массивы, чтобы помочь в управлении параллельными данными.

Большинство программ PETSc начинаются обращением:

PetscInitialize(int *argc, char ***argv, char *file, char *help);

которое инициализирует PETSc и MPI. Аргументы argc и argv являются аргументами командной строки и поставляются во всех C и C++ программах. Файл аргументов в качестве опции указывает на альтернативное имя файла опций, который размещается по умолчанию в пользовательском директории. Конечный аргумент help есть знаковая строка, которая будет печататься, если программа выполняется с опцией –help.

PetscInitialize автоматически вызывает **MPI_Init,** если он не был инициализирован ранее. В определенных обстоятельствах, когда нужно инициализировать MPI (или инициализация производится некоторой другой библиотекой) необходимо сначала вызвать **MPI_Init**, затем выполнить обращение к **PetscInitialize**. По умолчанию **PetscInitialize** добавляет к **MPI_COMM_WORLD** коммуникатор с фиксированным именем **PETSC_COMM_WORLD.**

В большинстве случаев пользователь может использовать коммуникатор **PETSC_COMM_WORLD,** чтобы указать все процессы в данном приложении, а **PETSC_COMM_SELF** указывает одиночный процесс.

Пользователи, которые хотят использовать процедуры PETSc только для подмножества процессоров внутри большой работы, или если необходимо использовать главный процесс для координации работы подчиненных PETSc процессов, должны описать альтернативный коммуникатор для **PETSC_COMM_WORLD** обращением:

PetscSetCommWorld(MPI_Comm comm);

перед вызовом **PetscInitialize,** но после обращения к **MPI_Init**. **PetscSetCommWorld** в большинстве случаев вызывается однажды на процесс.

Все PETSc процедуры возвращают целое значение, указывающее, имела ли место ошибка на протяжении вызова. Код ошибки устанавливается в ненулевое значение, если ошибка была обнаружена, в противном случае устанавливается нуль.

Все PETSc программы должны вызывать **PetscFinalize** как их финальное предложение:

PetscFinalize().

Эта процедура вызывает **MPI_Finalize,** если MPI был запущен вызовом **PetscInitialize**. Если MPI был запущен внешним по отношению к PETSc образом, например, пользователем, то пользователь несет ответственность за вызов **MPI_Finalize.**

Пользователь обязан описывать коммуникатор при создании любого объекта PETSc (вектора, матрицы, метода решения), чтобы указать процессы, на которые распределяется объект. Например, некоторые команды для создания матриц, векторов и методов решения систем линейных уравнений таковы:

```
MatCreate(MPI_Comm comm, int M, int N, Mat *A);
VecCreate(MPI_Comm comm, int m, int M, Vec *x);
SLESCreate(MPI_Comm comm, SLES *sles);
```

Процедуры создания объектов являются коллективными над всеми процессами в коммуникаторе, все процессы в коммуникаторе обязаны обращаться к процедурам создания объектов. Кроме того, если используется последовательность коллективных процедур, они обязаны вызываться в том же самом порядке на каждом процессе.

Опишем некоторые основные процедуры библиотеки, которые дадут представление о работе с новыми структурами данных.

Вектора. PETSc обеспечивает два основных векторных типа: последовательный и параллельный (основанный на MPI). Для создания последовательного вектора с m компонентами служит команда

VecCreateSeq(PETSC_COMM_SELF,int m,Vec *x).

Чтобы создать параллельный вектор необходимо указать число компонент, которые распределяются между процессами. Команда

VecCreateMPI(MPI_Comm comm, int m, int M,Vec *x);

создает вектор, распределенный между всеми процессами коммуникатора **comm, m** указывает число компонент вектора в локальном процессе, **M** – общее число компонент вектора. Или локальную или глобальную размерность, но не обе одновременно, можно установить равными **PETSC_DECIDE**, чтобы указать, что PETSc определит их на этапе выполнения.

Параллельный или последовательный вектор **x** с глобальным размером **M** можно создать так:

VecCreate(MPI_Comm comm,int m,int M,Vec *x);
VecSetFromOptions().

В этом случае автоматически генерируется тип вектора (последовательный или параллельный). Дополнительные векторы того же типа могут быть сформированы с помощью вызова:

VecDuplicate(Vec old,Vec *new);

Команды

VecSet(Scalar *value,Vec x);
VecSetValues(Vec x,int n,int *indices, Scalar *values,INSERT_VALUES);

позволяют заполнить векторы некоторыми значениями: или устанавливают все компоненты вектора равными некоторому скалярному значению, или присваивают различные значение каждому компоненту. **Scalar** определяется как **double** в C/C++ (или соответственно **double precision** в языке Фортран) для версий PETSc, которая не была скомпилирована для использования с комплексными числами.

Для параллельных векторов, которые распределяются по процессорам в соответствии с их номерами, процедура:

VecGetOwnershipRange (Vec vec, int *low, int *high)

позволяет определить локальную для данного процессора часть вектора. Здесь аргумент **low** указывает первый компонент вектора, принадлежащий данному процессору, а аргумент **high** указывает число,

на единицу большее, чем номер последнего принадлежащего процессору элемента вектора.

Основные векторные операции представлены в табл.13.1.

Таблица 13.1

Основные векторные операции

№	Название функции	Операция
1	VecAXPY(Scalar *a,Vec x, Vec y);	$y = y + a * x$
2	VecAYPX(Scalar *a,Vec x, Vec y);	$y = x + a * y$
3	VecWAXPY(Scalar *a,Vec x,Vec y, Vec w);	$w = a * x + y$
4	VecAXPBY(Scalar *a,Scalar *,Vec x,Vec y);	$y = a * x + b * y$
5	VecScale(Scalar *a, Vec x);	$x = a * x$
6	VecDot(Vec x, Vec y, Scalar *r);	$r = \bar{x}' * y$
7	VecTDot(Vec x, Vec y, Scalar *r);	$r = x' * y$
8	VecNorm(Vec x,NormType type, double *r);	$r = \|x\|_{type}$
9	VecSum(Vec x, Scalar *r);	$r = \sum x_i$
10	VecCopy(Vec x, Vec y);	$y = x$
11	VecSwap(Vec x, Vec y);	$y = x\ while\ x = y$
12	VecPointwiseMult(Vec x,Vec y, Vec w);	$w_i = x_i * y_i$
13	VecPointwiseDivide(Vec x,Vec y, Vec w);	$w_i = x_i / y_i$
14	VecMDot(int n,Vec x, Vec *y,Scalar *r);	$r[i] = \bar{x}' * y[i]$
15	VecMTDot(int n,Vec x, Vec *y,Scalar *r);	$r[i] = x' * y[i]$
16	VecMAXPY(int n, Scalar *a,Vec y, Vec *x);	$y = y + \sum_i a_i * x[i]$
17	VecMax(Vec x, int *idx, double *r);	$r = \max x_i$
18	VecMin(Vec x, int *idx, double *r);	$r = \min x_i$
19	VecAbs(Vec x);	$x_i = \|x_i\|$
20	VecReciprocal(Vec x);	$x_i = 1 / x_i$
21	VecShift(Scalar *s,Vec x);	$x_i = s + x_i$

Матрицы. Использование матриц и векторов в PETSc сходно. Пользователь может создавать новую параллельную или последовательную матрицу **A**, которая имеет **M** глобальных строк и **N** глобальных столбцов с помощью процедуры:

MatCreate(MPI_Comm comm, int m, int n, int M, int N, Mat *A);

где распределение элементов матрицы по процессам может быть описано на этапе выполнения, или описано для каждого номера процесса через локальные номера строк и столбцов, используя m и n. Номера строк в локальном процессе можно определить командой:

MatGetOwnershipRange(Mat A, int *first_row, int *last_row).

Значения элементов матрицы можно установить командой:

MatSetValues(Mat A, int m, int *im, int n, int *in, Scalar *values, INSERT_VALUES);

После того, как все элементы помещены в матрицу, она должна быть обработана парой команд:

MatAssemblyBegin(Mat A,MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(Mat A,MAT_FINAL_ASSEMBLY);

Основные матричные операции представлены в табл. 13.2.

Таблица 13.2

Основные матричные операции

№	Название функции	Операция
1	MatAXPY(Scalar *a,Mat X, Mat Y);	$Y = Y + a * X$
2	MatMult(Mat A,Vec x, Vec y);	$y = A * x$
3	MatMultAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A * x$
4	MatMultTrans(Mat A,Vec x, Vec y);	$y = A^T * x$
5	MatMultTransAdd(Mat A,Vec x, Vec y,Vec z);	$z = y + A^T * x$
6	MatNorm(Mat A,NormType type, double *r);	$r = \|A\|_{type}$
7	MatDiagonalScale(Mat A,Vec l,Vec r);	$A = diag(l) * A * diag(r)$
8	MatScale(Scalar *a,Mat A);	$A = a * A$
9	MatConvert(Mat A,MatType type,Mat *B);	$B = A$
10	MatCopy(Mat A,Mat B,MatStructure);	$B = A$
11	MatGetDiagonal(Mat A,Vec x);	$x = diag(A)$
12	MatTranspose(Mat A,Mat* B);	$B = A^T$
13	MatZeroEntries(Mat A);	$A = 0$
14	MatShift(Scalar *a,Mat Y);	$Y = Y + a * I$

Решение систем линейных уравнений. После создания матрицы и векторов, которые определяют линейную систему $Ax=b$, пользователь может решить эту систему с помощью SLES, применив последовательность команд:

SLESCreate(MPI_Comm comm,SLES *sles);
SLESSetOperators(SLES sles,Mat A,Mat PrecA,MatStructure flag);
SLESSetFromOptions(SLES sles);
SLESSolve(SLES sles,Vec b,Vec x,int *its);
SLESDestroy(SLES sles);

Пользователь сначала создает SLES контекст и устанавливает операторы, связанные с системой, затем устанавливает различные опции для настройки решения, решает линейную систему и, наконец, разрушает SLES контекст. Команда **SLESSetFromOptions()** разрешает пользователю настраивать метод линейного решения на этапе выполнения, используя определенный набор опций. Используя этот набор, пользователь может не только избрать итеративный метод и предобработку, но и предписать устойчивость сходимости, установить различные мониторинговые процедуры и т.д.

Решение систем нелинейных уравнений. Большинство ДУЧП задач – нелинейны. PETSc обеспечивает интерфейс для решения нелинейных задач вызовом SNES аналогично SLES**.**

13.3. ПРИМЕРЫ

Рассмотрим несколько простейших задач с использованием библиотеки PETSc. Задачи иллюстрируют общие принципы создания программ и использования новых структур данных.

Все программы должны включать заголовочные файлы для PETSc:

```
#include "petscsles.h",
```

где **petscsles.h** есть include файл для **SLES** компонента. Каждая программа PETSc обязана описать include файл, который соответствует наиболее высокому уровню объектов PETSc, необходимых внутри программы; все необходимые include файлы нижнего уровня включаются автоматически внутри файлов верхнего уровня. Например, **petscsles.h** включает **petscmat.h** (матрицы), **petscvec.h** (вектора), и **petsc.h** (основной PETSc файл).

Пользователь может вводить управляющие данные во время исполнения, используя опции базы данных. Например, команда

```
OptionsGetInt(PETSC_NULL,"-n", &n, &flg);
```

проверяет, обеспечил ли пользователь опцию командной строки, чтобы установить значение **n**, которое определяет размер задачи. Если это сделано, соответственно устанавливается переменная **n**, в противном случае **n** не изменяется.

Пример 1. Умножение матрицы на вектор

Рассмотрим параллельную программу умножения матрицы на вектор. Умножаем матрицу **A**, распределенную полосами из строк по процессам, на параллельный вектор **b**, распределенный по процессам.

Результирующий вектор **u** будет параллельный и распределен по процессам.

```
static char help[] = "умножение матрицы на вектор \n\
Входные параметры:\n\  -n <n>       : порядок квадратной матрицы \n";

#include "petscmat.h"
int main(int argc,char **args)
{ Vec        b,u;            /* вектора  */
  Mat        A;              /* матрица  */
  int        I,J,Istart,Iend,ierr,n = 8;
  Scalar     zz,one = 1.0;

  PetscInitialize(&argc,&args,(char *)0,help);                  /* инициализация Petsc  */
                                    /* задаем порядок матрицы или по умолчанию  */
   ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);

/* Создаем параллельную структуру типа Mat  для матрицы A.  Матрица будет
разделена полосами из строк по  процессам.  Создаем параллельную матрицу,
описывая только ее глобальные размеры. При использовании  MatCreate() формат
распределения элементов матрицы по процессам будет определен на
 этапе  исполнения: PETSC_DECIDE . */

  ierr = MatCreate (PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,
                    n,n, &A); CHKERRQ(ierr);
  ierr = MatSetFromOptions(A);CHKERRQ(ierr);
                        /*  Определяем, какие строки матрицы размещены локально.
                            Istart,Iend  номера строк матрицы A в  процессе */
  ierr = MatGetOwnershipRange(A,&Istart,&Iend);CHKERRQ(ierr);
  ierr = PetscPrintf(PETSC_COMM_WORLD," %d %d\n",Istart,Iend);
        CHKERRQ(ierr);
                        /* заполняем матрицу A некоторыми значениями */
  for (I=0; I<n; I++)
     for (J=0; J<n; J++)
     {    zz=I*n+J;
          ierr =  MatSetValues(A,1,&I,1,&J,&zz,INSERT_VALUES);CHKERRQ(ierr);
     }
  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
                        /* вывод на экран  */
  ierr = MatView(A,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);

          /* создаем структуру Vect для векторов u, b.  Создаем  параллельный
             вектор, описывая только его глобальные размеры. При использовании
             VecCreate() формат распределения элементов вектора по процессам
             будет определен на этапе    исполнения:   PETSC_DECIDE . */
```

```
  ierr = VecCreate(PETSC_COMM_WORLD,PETSC_DECIDE,n,&u);
         CHKERRQ(ierr);
  ierr = VecSetFromOptions(u);CHKERRQ(ierr);
  ierr = VecDuplicate(u,&b);CHKERRQ(ierr);
  ierr = VecSet(&one,b);CHKERRQ(ierr);       /* заполняем вектор b единицами */
                                             /* вывод на экран вектора b */
   ierr = VecView(b,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
                                             /*  параллельное умножение */
  ierr = MatMult(A,b,u);CHKERRQ(ierr);
                                    /* вывод на экран результирующего вектора u */
  ierr = VecView(u,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}
```

Пример 2. Умножение матриц

Обобщим предыдущую программу для умножения матрицы на матрицу. Умножаем параллельную матрицу **A**, распределенную полосами строк по процессам на последовательную матрицу **B** в каждом процессе. Результирующая матрица **C** будет распределена полосами столбцов по процессам.

```
static char help[] = "умножение матрицы на матрицу.\n\
Входные параметры:\n\ -n <n>       : порядок квадратной матрицы \n";
#include "petscmat.h"
int main(int argc,char **args)
{ Vec        b,u;                          /* вектора */
  Mat        A;                            /* матрица */
  int        I,J,ierr,n = 8,*cc,size;
  Scalar     zz,*ci,**B,**C;

  PetscInitialize(&argc,&args,(char *)0,help);          /* инициализация Petsc  */
                                  /* задаем порядок матрицы или по умолчанию  */
  ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);CHKERRQ(ierr);
                                /* определяем количество процессов приложения */
  ierr = MPI_Comm_size(PETSC_COMM_WORLD,&size);CHKERRQ(ierr);
                                          /* выделяем память для матрицы B  */
  ierr = PetscMalloc(n*sizeof(double *),&B);CHKERRQ(ierr);
  for (I=0; I<n; I++)
      ierr = PetscMalloc(n*sizeof(double),&B[I]);CHKERRQ(ierr);
                    /* заполняем матрицу некоторыми значениями по столбцам */
  for (I=0; I<n; I++)
      for (J=0; J<n; J++)
          B[I][J]=I+1;
```

```
  for (J=0; J<n; J++)          /* вывод на экран */
  {
      for (I=0; I<n; I++)
         ierr = PetscPrintf(PETSC_COMM_WORLD," %f \n", B[I][J]);
                CHKERRQ(ierr);
         ierr = PetscPrintf(PETSC_COMM_WORLD," \n"); CHKERRQ(ierr);
  }
                                          /*  выделяем память для матрицы C */
  ierr = PetscMalloc(n*sizeof(double *),&C);CHKERRQ(ierr);
  for (I=0; I<n; I++)
      ierr = PetscMalloc(n/size*sizeof(double),&C[I]);CHKERRQ(ierr);
```

/* Создаем параллельную структуру типа Mat для матрицы A. Матрица будет разделена полосами из строк по процессам. Создаем параллельную матрицу, описывая только ее глобальные размеры. При использовании MatCreate() формат распределения элементов матрицы по процессам будет определен на этапе исполнения: PETSC_DECIDE . */

```
  ierr = MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE, PETSC_DECIDE,
                       n,n, &A); CHKERRQ(ierr);
  ierr = MatSetFromOptions(A);CHKERRQ(ierr);
                              /* заполняем матрицу A некоторыми значениями */
  for (I=0; I<n; I++)
    for (J=0; J<n; J++)
    {   zz=I+1;
        ierr = MatSetValues(A,1,&I,1,&J,&zz,INSERT_VALUES);CHKERRQ(ierr);
    }
  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
                                                    /* вывод на экран  */
  ierr = MatView(A,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
```

/* создаем структуру Vect для столбцов матрицы B и строк матрицы C. Создаем параллельный вектор, описывая только его глобальные размеры. При использовании VecCreate() формат распределения элементов вектора по процессам будет определен на этапе исполнения: PETSC_DECIDE . */

```
  ierr = VecCreate(PETSC_COMM_WORLD,PETSC_DECIDE,n,&u);
         CHKERRQ(ierr);
  ierr = VecSetFromOptions(u);CHKERRQ(ierr);
  ierr = VecDuplicate(u,&b);CHKERRQ(ierr);
                                        /* заполняем массив порядка индексов  */
  ierr = PetscMalloc(n*sizeof(int),&cc);CHKERRQ(ierr);
  for (J=0; J<n; J++)
        cc[J]=J;
```

```
        /*основной цикл умножения матрицы A на строку-вектор матрицы B */
 for (I=0; I<n; I++)
 {                          /* заполняем вектор u значениями столбца матрицы B */
       ierr = VecSetValues(u,n,cc,B[I],INSERT_VALUES);CHKERRQ(ierr);
                                                  /*  параллельное умножение */
       ierr = MatMult(A,u,b);CHKERRQ(ierr);
                                          /* заполнение результатов в  матрицу C */
       ierr = VecGetArray(b,&ci);CHKERRQ(ierr);
       for (J=0; J<n/size; J++)    C[I][J]=ci[J];
       ierr = VecRestoreArray(b,&ci);CHKERRQ(ierr);
 }
                                              /* вывод результирующей матрицы C */
  for (I=0; I<n; I++)
  {
       for (J=0; J<n/size; J++)
           ierr = PetscPrintf(PETSC_COMM_WORLD," %f \n", C[I][J]);
                 CHKERRQ(ierr);
       ierr = PetscPrintf(PETSC_COMM_WORLD," \n"); CHKERRQ(ierr);
  }
  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}
```

Пример 3. Умножение матриц (вариант с рассылкой)

Если необходимо иметь результирующую матрицу во всех процессах, можно использовать механизм scatter/gather. Рассмотрим, как это организовано в библиотеке PETSc. Умножаем параллельную матрицу *A*, распределенную полосами строк по процессам на последовательную матрицу *B* в каждом процессе. Результирующая последовательная матрица *C* будет получена всеми процессами.

```
static char help[] = "умножение матрицы на матрицу.\n\
Входные параметры:\n\ -n <n>       : порядок квадратной матрицы \n";

#include "petscmat.h"
int main(int argc,char **args)
{  Vec         b,u,c;          /* вектора */
   Mat         A;              /* матрица */
   int         I,J,ierr,n = 8,*cc;
  Scalar      zz,**B,*ci,**C;
  IS          is;
  VecScatter  ctx;                  /* описание переменной для организации рассылки */

  PetscInitialize(&argc,&args,(char *)0,help);                  /* инициализация Petsc  */
                                    /* задаем порядок матрицы или по умолчанию  */
```

```
 ierr = PetscOptionsGetInt(PETSC_NULL,"-n",&n,PETSC_NULL);
        CHKERRQ(ierr);
                                                  /* выделяем память для матрицы B  */
  ierr = PetscMalloc(n*sizeof(double *),&B);CHKERRQ(ierr);
  for (I=0; I<n; I++)
      ierr = PetscMalloc(n*sizeof(double),&B[I]);CHKERRQ(ierr);
                                 /* выделяем память для результирующей матрицы C  */
  ierr = PetscMalloc(n*sizeof(double *),&C);CHKERRQ(ierr);
  for (I=0; I<n; I++)
       ierr = PetscMalloc(n*sizeof(double),&C[I]);CHKERRQ(ierr);
                       /* заполняем матрицу B некоторыми значениями по столбцам */
  for (I=0; I<n; I++)
       for (J=0; J<n; J++)
       {   B[I][J]=I+1;        C[I][J]=0;         }
                                                    /* вывод на экран матрицы B  */
  for (J=0; J<n; J++)
  {   for (I=0; I<n; I++)
          ierr = PetscPrintf(PETSC_COMM_WORLD," %f \n", B[I][J]);
                CHKERRQ(ierr);
      ierr = PetscPrintf(PETSC_COMM_WORLD," \n"); CHKERRQ(ierr);
  }
```

/* Создаем параллельную структуру типа Mat для матрицы A. Матрица будет разделена полосами из строк по процессам. Создаем параллельную матрицу, описывая только ее глобальные размеры. При использовании MatCreate() формат распределения элементов матрицы по процессам будет определен на этапе исполнения: PETSC_DECIDE . */

```
  ierr = MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,
                   n,n,&A); CHKERRQ(ierr);
  ierr = MatSetFromOptions(A);CHKERRQ(ierr);
                                    /* заполняем матрицу A некоторыми значениями */
  for (I=0; I<n; I++)
       for (J=0; J<n; J++)
       {    zz=I+1;
            ierr = MatSetValues(A,1,&I,1,&J,&zz,INSERT_VALUES);
                 CHKERRQ(ierr);
       }
  ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
  ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
                                                          /* вывод на экран  */
  ierr = MatView(A,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
```

/* создаем структуру Vect для столбцов матрицы B и строк матрицы C. Создаем параллельный вектор, описывая только его глобальные размеры. При использовании VecCreate() формат распределения элементов вектора по процессам будет определен на этапе исполнения: PETSC_DECIDE . */

```
  ierr = VecCreate(PETSC_COMM_WORLD,PETSC_DECIDE,n,&u);
         CHKERRQ(ierr);
  ierr = VecSetFromOptions(u);CHKERRQ(ierr);
  ierr = VecDuplicate(u,&b);CHKERRQ(ierr);

                                    /* создаем структуры  для организации рассылки  */
  ierr = VecCreateSeq(PETSC_COMM_SELF,n,&c);CHKERRQ(ierr);
  ierr = ISCreateStride(PETSC_COMM_SELF,n,0,1,&is);CHKERRQ(ierr);
  ierr = VecScatterCreate(b,is,c,is,&ctx);CHKERRQ(ierr);
                                              /* заполняем массив порядка индексов  */
  ierr = PetscMalloc(n*sizeof(int),&cc);CHKERRQ(ierr);
  for (J=0; J<n; J++)
       cc[J]=J;

            /* основной цикл умножения матрицы A на строку-вектор матрицы B */
  for (I=0; I<n; I++)
  {                              /* заполняем вектор u значениями столбца матрицы B */
       ierr = VecSetValues(u,n,cc,B[I],INSERT_VALUES);CHKERRQ(ierr);
                                                        /*  параллельное умножение */
       ierr = MatMult(A,u,b);CHKERRQ(ierr);
                                               /* рассылка вектора всем процессам */
       ierr = VecScatterBegin(b,c,INSERT_VALUES,SCATTER_FORWARD,ctx);
              CHKERRQ(ierr);
       ierr = VecScatterEnd(b,c,INSERT_VALUES,SCATTER_FORWARD,ctx);
              CHKERRQ(ierr);
                                              /* заполнение результатов в матрицу C */
       ierr = VecGetArray(c,&ci);CHKERRQ(ierr);
       for (J=0; J<n; J++)
            C[I][J]=ci[J];
       ierr = VecRestoreArray(c,&ci);CHKERRQ(ierr);
  }
                                               /* вывод результирующей матрицы C */
  for (I=0; I<n; I++)
  {    for (J=0; J<n; J++)
           ierr = PetscPrintf(PETSC_COMM_WORLD," %f \n", C[I][J]);
                  CHKERRQ(ierr);
       ierr = PetscPrintf(PETSC_COMM_WORLD," \n"); CHKERRQ(ierr);
  }
  ierr = PetscFinalize();CHKERRQ(ierr);
  return 0;
}
```

Пример 4. Решение системы линейных уравнений

Следующий пример иллюстрирует параллельное решение системы линейных уравнений с помощью SLES. Пользовательский интерфейс для инициации программы, создания векторов и матриц и решения линейной системы является тем же для однопроцессорного и многопроцессорного примеров. Главное различие состоит в том, что каждый процесс формирует локальную часть матрицы и вектора в параллельном случае.

```
static char help[] = "Solves a linear system in parallel with SLES.\n\
Input parameters include:\n\
-random_exact_sol : use a random exact solution vector\n\
-view_exact_sol : write exact solution vector to stdout\n\
-m <mesh_x> : number of mesh points in x-direction\n\
-n <mesh_n> : number of mesh points in y-direction\n\n";

#include "petscsles.h"
int main(int argc,char **args)
{
Vec x,b,u;              /* приближенное решение, RHS, точное решение */
Mat A;                  /* матрица линейной системы */
SLES sles;              /* метод решения линейной системы */
PetscRandom rctx;       /*  генератор случайных чисел */
double norm;            /* норма ошибки решения */
int i,j,I,J,Istart,Iend,ierr,m = 8,n = 7,its;
PetscTruth flg;
Scalar v,one = 1.0,neg_one = -1.0;
KSP ksp;

PetscInitialize(&argc,&args,(char *)0,help);
ierr = PetscOptionsGetInt(PETSC_NULL,"-",&m,PETSC_NULL);CHKERRQ(ierr);
ierr = PetscOptionsGetInt(PETSC_NULL,"-",&n,PETSC_NULL);CHKERRQ(ierr);

/* Организуем матрицу и правосторонний вектор, который определяет линейную
систему, Ax = b. Создаем параллельную матрицу, описывая только ее глобальные
размеры. Когда используется MatCreate(), формат матрицы может быть описан на
этапе исполнения. */

ierr=MatCreate(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,
                m*n,m*n,&A);  CHKERRQ(ierr);
ierr = MatSetFromOptions(A);CHKERRQ(ierr);
/*  Матрица разделена крупными кусками строк по процессам.  Определяем, какие
    строки матрицы   размещены локально.*/
ierr = MatGetOwnershipRange(A,&Istart,&Iend);CHKERRQ(ierr);
```

```
/*  Размещаем матричные элементы. Каждому процессу нужно разместить только
те элементы, которые принадлежат ему локально (все нелокальные элементы бу-
дут посланы в соответствующий процессор во время сборки  матрицы). */
for (I=Istart; I<Iend; I++)
{     v = -1.0; i = I/n; j = I – i*n;
      if (i>0)
      {  J = I – n;
         ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);
                CHKERRQ(ierr);
      }
      if (i<m-1)
      {   J = I + n;
          ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);
                 CHKERRQ(ierr);
      }
      if (j>0)
      {   J = I – 1;
          ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);
                CHKERRQ(ierr);
      }
      if (j<n-1)
      {   J = I + 1;
         ierr = MatSetValues(A,1,&I,1,&J,&v,INSERT_VALUES);
                CHKERRQ(ierr);
      }
      v = 4.0;
      ierr = MatSetValues(A,1,&I,1,&I,&v,INSERT_VALUES);
             CHKERRQ(ierr);
}
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);

/* Создаем параллельные векторы. Формируем  вектор  и затем дублируем, если
необходимо.  Когда  в  этом  примере  используется  VecCreate()  и
VecSetFromOptions(),  указывается  только глобальный размер вектора; парал-
лельное разбиение определяется на этапе исполнения. Когда решается линейная
система, векторы и матрицы обязаны быть разбиты  соответственно. PETSc авто-
матически генерирует соответствующее разбиение матриц и векторов, когда
MatCreate() и VecCreate() используются с тем же самым коммуникатором. Поль-
зователь может альтернативно описать размеры локальных векторов и матриц,
когда необходимо более сложное разбиение (путем замещения аргумента
PETSC_DECIDE  в VecCreate()). */

ierr = VecCreate(PETSC_COMM_WORLD,PETSC_DECIDE,m*n,&u);
       CHKERRQ(ierr);
```

```
ierr = VecSetFromOptions(u);CHKERRQ(ierr);
ierr = VecDuplicate(u,&b);CHKERRQ(ierr);
ierr = VecDuplicate(b,&x);CHKERRQ(ierr);

/* Установим точное решение, затем вычислим правосторонний вектор. По умол-
чанию используем точное решение вектора для случая, когда все элементы
вектора равны единице. Альтернативно, используя опцию – random_sol, фор-
мируем решение вектора со случайными компонентами. */
ierr = PetscOptionsHasName(PETSC_NULL,"-random_exact_sol",&flg);
       CHKERRQ(ierr);
if (flg)
{ ierr = PetscRandomCreate(PETSC_COMM_WORLD,RANDOM_DEFAULT,&rctx);
         CHKERRQ(ierr);
  ierr = VecSetRandom(rctx,u);CHKERRQ(ierr);
  ierr = PetscRandomDestroy(rctx);CHKERRQ(ierr);
}
else   ierr = VecSet(&one,u);CHKERRQ(ierr);

ierr = MatMult(A,u,b);CHKERRQ(ierr);
                              /*   Выводим точное решение вектора, если необходимо */
ierr = PetscOptionsHasName(PETSC_NULL,"-view_exact_sol",&flg);
       CHKERRQ(ierr);
if (flg)
{    ierr = VecView(u,PETSC_VIEWER_STDOUT_WORLD);
     CHKERRQ(ierr);
}
                              /*  Создаем метод решения системы линейных уравнений  */
ierr = SLESCreate(PETSC_COMM_WORLD,&sles);CHKERRQ(ierr);
         /* Устанавливаем операторы. Здесь матрица, которая определяет линейную
            систему,  служит как preconditioning матрица.    */
ierr = SLESSetOperators(sles,A,A,DIFFERENT_NONZERO_PATTERN);
       CHKERRQ(ierr);
                                                  /* Устанавливается  метод решения */
ierr = SLESGetKSP(sles,&ksp);CHKERRQ(ierr);
ierr = KSPSetTolerances(ksp,1.e-2/((m+1)*(n+1)),1.e-50,
                                PETSC_DEFAULT,PETSC_DEFAULT);CHKERRQ(ierr);
ierr = SLESSetFromOptions(sles);CHKERRQ(ierr);
                                                        /*  Решается линейная система */
ierr = SLESSolve(sles,b,x,&its);CHKERRQ(ierr);
                         /* Проверяется решение  norm *= sqrt(1.0/((m+1)*(n+1))); */
ierr = VecAXPY(&neg_one,u,x);CHKERRQ(ierr);
ierr = VecNorm(x,NORM_2,&norm);CHKERRQ(ierr);

/* Печатается информация о сходимости.  PetscPrintf() создает одно предложение
печати из всех процессов, которые разделяют коммуникатор. Альтернативой яв-
ляется PetscFPrintf(), которая печатает в файл. */
```

```
ierr = PetscPrintf(PETSC_COMM_WORLD,"Norm of error %A iterations d\n",
                   norm,its); CHKERRQ(ierr);

/* Освобождается рабочее пространство. Все PETSc объекты должны быть разрушены, когда они больше не нужны. */
ierr = SLESDestroy(sles);CHKERRQ(ierr);ierr = VecDestroy(u);CHKERRQ(ierr);
ierr = VecDestroy(x);CHKERRQ(ierr);ierr = VecDestroy(b);CHKERRQ(ierr);
ierr = MatDestroy(A);CHKERRQ(ierr);
ierr = PetscFinalize();CHKERRQ(ierr);
return 0;
}
```

КОНТРОЛЬНЫЕ ВОПРОСЫ К ГЛАВЕ 13

Контрольные вопросы к 13.1

1. Для чего предназначена библиотека ScaLAPACK?
2. Для чего предназначена библиотека PETSc?
3. Опишите уровень операций в библиотеки BLAS?
4. Для чего предназначена библиотека BLAS?
5. Для чего предназначена библиотека LAPACK?
6. Каково назначение библиотеки BLACS?
7. Объясните назначение параметров функции vTRSD2D.
8. Какие четыре шага надо сделать, чтобы выполнить программу ScaLAPACK?
9. Опишите каждый из этих четырех шагов.
10. Объясните на примере программы умножения матриц особенности работы библиотеки ScaLAPACK.

Контрольные вопросы к 13.2

1. Какие задачи позволяет решать библиотека PETSc?
2. Нужно ли вызывать MPI_Init при написании PETSc программы?
3. В каком случае необходимо создание коммуникатора и как это сделать?
4. Какие типы векторов существуют в библиотеке? Как создать вектор того или иного типа?
5. В каком случае нужно создавать параллельные векторы?
6. Можно ли распределить компоненты параллельного вектора по процессам не равными частями?
7. Как определить номера строк в локальном процессе при создании параллельной матрицы?
8. Какие матричные операции библиотеки можно использовать для параллельного решения задачи умножения матрицы на матрицу?
9. Необходим ли MPICH для инсталляции PETSc?
10. Как выполнить PETSc-программу на 4 процессах, задавая различные размерности задачи?

ПРИЛОЖЕНИЯ

Приложение 1. КОНСТАНТЫ ДЛЯ ЯЗЫКОВ С И FORTRAN

Здесь приведены константы, определенные в файлах mpi.h (для С) и mpif.h (для языка Fortran).

```
        /* возвращаемые коды (как для С, так и для  Fortran) */
MPI_SUCCESS
MPI_ERR_BUFFER
MPI_ERR_COUNT
MPI_ERR_TYPE
MPI_ERR_TAG
MPI_ERR_COMM
MPI_ERR_RANK
MPI_ERR_REQUEST
MPI_ERR_ROOT
MPI_ERR_GROUP
MPI_ERR_OP
MPI_ERR_TOPOLOGY
MPI_ERR_DIMS
MPI_ERR_ARG
MPI_ERR_UNKNOWN
MPI_ERR_TRUNCATE
MPI_ERR_OTHER
MPI_ERR_INTERN
MPI_ERR_PENDING
MPI_ERR_IN_STATUS
MPI_ERR_LASTCODE
        /*  константы (как для С, так и для  Fortran) */
MPI_BOTTOM
MPI_PROC_NULL
MPI_ANY_SOURCE
MPI_ANY_TAG
MPI_UNDEFINED
MPI_BSEND_OVERHEAD
MPI_KEYVAL_INVALID
        /* размер статуса и резервируемые индексные значения (Fortran) */
MPI_STATUS_SIZE
MPI_SOURCE
MPI_TAG
MPI_ERROR
        /* описатели обработчика ошибок (С и Fortran) */
MPI_ERRORS_ARE_FATAL
MPI_ERRORS_RETURN
```

```
        /* максимальные размеры строк */
MPI_MAX_PROCESSOR_NAME
MPI_MAX_ERROR_STRING
        /* элементарные типы данных (C) */
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_UNSIGNED_SHORT
MPI_UNSIGNED
MPI_UNSIGNED_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_BYTE
MPI_PACKED
        /* элементарные типы данных(Fortran) */
MPI_INTEGER
MPI_REAL
MPI_DOUBLE_PRECISION
MPI_COMPLEX
MPI_LOGICAL
MPI_CHARACTER
MPI_BYTE
MPI_PACKED
        /* типы данных для функций редукции (C) */
MPI_FLOAT_INT
MPI_DOUBLE_INT
MPI_LONG_INT
MPI_2INT
MPI_SHORT_INT
MPI_LONG_DOUBLE_INT
        /* типы данных для функций редукции (Fortran) */
MPI_2REAL
MPI_2DOUBLE_PRECISION
MPI_2INTEGER
        /* дополнительные типы данных (Fortran) */
MPI_INTEGER1
MPI_INTEGER2
MPI_INTEGER4
MPI_REAL2
MPI_REAL4
MPI_REAL8
etc.
```

```
        /* дополнительные типы данных (C) */
MPI_LONG_LONG_INT
etc.
        /*специальные типы данных для создания производных типов данных */
MPI_UB
MPI_LB
        /* зарезервированные коммуникаторы (C и Fortran) */
MPI_COMM_WORLD
MPI_COMM_SELF
        /* результаты сравнения коммуникаторов и групп */
MPI_IDENT
MPI_CONGRUENT
MPI_SIMILAR
MPI_UNEQUAL
        /* ключи запроса среды (C и Fortran) */
MPI_TAG_UB
MPI_IO
MPI_HOST
MPI_WTIME_IS_GLOBAL
        /* коллективные операции (C и Fortran) */
MPI_MAX
MPI_MIN
MPI_SUM
MPI_PROD
MPI_MAXLOC
MPI_MINLOC
MPI_BAND
MPI_BOR
MPI_BXOR
MPI_LAND
MPI_LOR
MPI_LXOR
        /* нулевые дескрипторы */
MPI_GROUP_NULL
MPI_COMM_NULL
MPI_DATATYPE_NULL
MPI_REQUEST_NULL
MPI_OP_NULL
MPI_ERRHANDLER_NULL
        /* пустая группа */
MPI_GROUP_EMPTY
        /* топологии (C и Fortran) */
MPI_GRAPH
MPI_CART
        /* предопределенные функции в C и Fortrun*/
MPI_NULL_COPY_FN
```

MPI_NULL_DELETE_FN
MPI_DUP_FN

Следующее есть типы C, также включенные в файл mpi.h.

```
        /* скрытые типы */
MPI_Aint
MPI_Status
        /* дескрипторы различных структур */
MPI_Group
MPI_Comm
MPI_Datatype
MPI_Request
MPI_Op
MPI_Errhandler
```

Приложение 2. ПЕРЕЧЕНЬ ФУНКЦИЙ MPI-1.1

1. **MPI_ABORT** (comm, errorcode)
   Прекращает выполнение операций
2. **MPI_ADDRESS** (location, address)
   Получает адрес в ячейке памяти
3. **MPI_ALLGATHER** (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
   Собирает данные от всех процессов и распределяет их всем процессам
4. **MPI_ALLGATHERV** (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)
   Собирает данные от всех процессов и поставляет их всем процессам
5. **MPI_ALLREDUCE** (sendbuf, recvbuf, count, datatype, op, comm)
   Выполняет глобальную операцию над данными от всех процессов и результат посылает обратно всем процессам
6. **MPI_ALLTOALL** (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
   Посылает данные от всех процессов всем процессам
7. **MPI_ALLTOALLV** (sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)
   Посылает данные от всех процессов всем процессам со смещением
8. **MPI_BARRIER** (comm)
   Блокирует дальнейшее исполнение, пока все процессы не достигнут этой функции
9. **MPI_BCAST**(buffer, count, datatype, root, comm)
   Широковещательная рассылка сообщения от одного процесса всем
10. **MPI_BSEND** (buf, count, datatype, dest, tag, comm)
    Операция посылки сообщения с определенным пользователем буфером
11. **MPI_BSEND_INIT** (buf, count, datatype, dest, tag, comm, request)
    Создает дескриптор для буферизованной посылки

12. **MPI_BUFFER_ATTACH** (buffer, size)
    Создает определяемый пользователем буфер для посылки
13. **MPI_BUFFER_DETACH** (buffer_addr, size)
    Удаляет существующий буфер
14. **MPI_CANCEL** (request)
    Отменяет коммуникационный запрос
15. **MPI_CART_COORDS** (comm, rank, maxdims, coords)
    Определяет координаты процесса в картезианской топологии в соответствии с его номером
16. **MPI_CART_CREATE** (comm_old, ndims, dims, periods, reorder, comm_cart)
    Создает новый коммуникатор по заданной топологической информации
17. **MPI_CART_GET** (comm, maxdims, dims, periods, coords)
    Получает связанную с коммуникатором информацию о картезианской топологии
18. **MPI_CART_MAP** (comm, ndims, dims, periods, newrank)
    Отображает процесс в соответствии с топологической информацией
19. **MPI_CART_RANK** (comm, coords, rank)
    Определяет номер процесса в коммуникаторе с картезианской топологией
20. **MPI_CART_SHIFT** (comm, direction, disp, rank_source, rank_dest)
    Возвращает новые номера процессов отправителя и получателя после сдвига в заданном направлении
21. **MPI_CART_SUB** (comm, remain_dims, newcomm)
    Разделяет коммуникатор на подгруппы, которые формируют картезианские решетки меньшей размерности
22. **MPI_CARTDIM_GET** (comm, ndims)
    Получает информацию о связанной с коммуникатором картезианской топологии
23. **MPI_COMM_COMPARE** (comm1, comm2, result)
    Сравнивает два коммуникатора
24. **MPI_COMM_CREATE** (comm, group, newcomm)
    Создает новый коммуникатор
25. **MPI_COMM_DUP** (comm, newcomm)
    Создает новый коммуникатор путем дублирования существующего со всеми его параметрами
26. **MPI_COMM_FREE** (comm)
    Маркирует коммуникатор для удаления
27. **MPI_COMM_GROUP** (comm, group)
    Осуществляет доступ к группе, связанной с данным коммуникатором
28. **MPI_COMM_RANK** (comm, rank)
    Определяет номер процесса в коммуникаторе
29. **MPI_COMM_SIZE** (comm, size)
    Определяет размер связанной с коммуникатором группы
30. **MPI_COMM_SPLIT** (comm, color, key, newcomm)
    Создает новый коммуникатор на основе признаков и ключей
31. **MPI_DIMS_CREATE** (nnodes, ndims, dims)
    Распределяет процессы по размерностям

32. **MPI_FINALIZE** ( )
Завершает выполнение программы MPI
33. **MPI_ GATHER** (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
Собирает в один процесс данные от группы процессов
34. **MPI_GATHERV** (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)
Векторный вариант функции GATHER
35. **MPI_GET_COUNT** (status, datatype, count)
Получает номер старших элементов
36. **MPI_GET_ELEMENTS** (status, datatype, count)
Возвращает номер базового элемента в типе данных
37. **MPI_GET_PROCESSOR_NAME** (name, resultlen)
Получает номер процессора
38. MPI_GET_VERSION (version, subversion)
Возвращает версию MPI
39. **MPI_GRAPH_CREATE** (comm_old, nnodes, index, edges, reorder, comm_graph)
Создает новый коммуникатор согласно топологической информации
40. **MPI_GRAPH_GET**(comm, maxindex, maxedges, index, edges)
Получает информацию о связанной с коммуникатором графовой топологии.
41. **MPI_GRAPH_MAP** (comm, nnodes, index, edges, newrank)
Размещает процесс согласно информации о топологии графа
42. **MPI_GRAPH_NEIGHBORS_COUNT**(comm, rank, nneighbors)
Возвращает число соседей узла в графовой топологии
43. **MPI_GRAPH_NEIGHBORS** (comm, rank, maxneighbors, neighbors)
Возвращает соседей узла в графовой топологии
44. **MPI_GRAPHDIMS_GET** (comm, nnodes, nedges)
Получает информацию о связанной с коммуникатором топологии
45. **MPI_GROUP_COMPARE** (group1, group2, result)
Сравнивает две группы
46. **MPI_GROUP_DIFFERENCE** (group1, group2, newgroup)
Создает группу по разности двух групп
47. **MPI_GROUP_EXCL** (group, n, ranks, newgroup)
Создает группу путем переупорядочивания существующей группы и отбора только тех элементов, которые не указаны в списке
48. **MPI_GROUP_FREE** (group)
Освобождает группу
49. **MPI_GROUP_INCL** (group, n, ranks, newgroup)
Создает группу путем переупорядочивания существующей группы и отбора только тех элементов, которые указаны в списке
50. **MPI_GROUP_INTERSECTION** (group1, group2, newgroup)
Создает группу на основе пересечения двух групп
51. **MPI_GROUP_RANGE_EXCL** (group, n, ranges, newgroup)
Создает группу путем исключения ряда процессов из существующей группы
52. **MPI_GROUP_RANGE_INCL** (group, n, ranges, newgroup)
Создает новую группу из ряда номеров существующей группы

53. **MPI_GROUP_RANK** (group, rank)
    Возвращает номер процесса в данной группе
54. **MPI_GROUP_SIZE** (group, size)
    Возвращает размер группы
55. **MPI_GROUP_TRANSLATE_RANKS** (group1, n, ranks1, group2, ranks2)
    Переводит номер процесса в одной группе в номер в другой группе
56. **MPI_GROUP_UNION**(group1, group2, newgroup)
    Создает новую группу путем объединения двух групп
57. **MPI_IBSEND** (buf, count, datatype, dest, tag, comm, request)
    Запускает неблокирующую буферизованную посылку
58. **MPI_INIT** ( )
    Инициализация параллельных вычислений
59. **MPI_INITIALIZED** (flag)
    Указывает, был ли выполнен MPI_INIT
60. **MPI_IPROBE** (source, tag, comm, flag, status)
    Неблокирующий тест сообщения
61. **MPI_IRECV** (buf, count, datatype, source, tag, comm, request)
    Начинает неблокирующий прием
62. **MPI_IRSEND** (buf, count, datatype, dest, tag, comm, request)
    Запускает неблокирующую посылку по готовности
63. **MPI_ISEND** (buf, count, datatype, dest, tag, comm, request)
    Запускает неблокирующую посылку
64. **MPI_ISSEND** (buf, count, datatype, dest, tag, comm, request)
    Запускает неблокирющую синхронную передачу
65. **MPI_PACK** (inbuf, incount, datatype, outbuf, outsize, position, comm)
    Упаковывает данные в непрерывный буфер
66. **MPI_PACK_SIZE** (incount, datatype, comm, size)
    Возвращает размер, необходимый для упаковки типа данных
67. **MPI_PROBE** (source, tag, comm, status)
    Блокирующий тест сообщения
68. **MPI_RECV** (buf, count, datatype, source, tag, comm, status)
    Основная операция приема
69. **MPI_RECV_INIT**(buf, count, datatype, source, tag, comm, request)
    Создает handle для приема
70. **MPI_REDUCE** (sendbuf, recvbuf, count, datatype, op, root, comm)
    Выполняет глобальную операцию над значениями всех процессов и возвращает результат в один процесс
71. **MPI_REDUCE_SCATTER** (sendbuf, recvbuf, recvcounts, datatype, op, comm)
    Выполняет редукцию и рассылает результаты
72. **MPI_REQUEST_FREE** (request)
    Освобождает объект коммуникационного запроса
73. **MPI_RSEND** (buf, count, datatype, dest, tag, comm)
    Операция посылки по готовности
74. **MPI_RSEND_INIT** (buf, count, datatype, dest, tag, comm, request)
    Создает дескриптор для посылки по готовности

75. **MPI_SCAN** (sendbuf, recvbuf, count, datatype, op, comm)
Вычисляет частичную редукцию данных на совокупности процессов
76. **MPI_SCATTER** (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root,comm)
Рассылает содержимое буфера одного процесса всем процессам в группе
77. **MPI_SCATTERV**(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)
Рассылает части буфера одного процесса всем процессам в группе
78. **MPI_SEND** (buf, count, datatype, dest, tag, comm)
Основная операция посылки
79. **MPI_SEND_INIT** (buf, count, datatype, dest, tag, comm, request)
Строит дескриптор для стандартной посылки
80. **MPI_SENDRECV** (sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)
Посылает и принимает сообщение
81. **MPI_SENDRECV_REPLACE** (buf, count, datatype, dest, sendtag, source, recvtag,comm, status)
Посылает и принимает сообщение, используя один буфер
82. **MPI_SSEND** (buf, count, datatype, dest, tag, comm)
Базисная синхронная передача
83. **MPI_SSEND_INIT** (buf, count, datatype, dest, tag, comm, request)
Строит дескриптор для синхронной передачи
84. **MPI_START** (request)
Инициирует обмен с персистентным дескриптором запросов
85. **MPI_STARTALL** (count, array_of_requests)
Запускает совокупность запросов
86. **MPI_TEST**(request, flag, status)
Проверяет завершение посылки или приема
87. **MPI_TESTALL** (count, array_of_requests, flag, array_of_statuses)
Проверяет завершение всех ранее начатых операций обмена
88. **MPI_TESTANY** (count, array_of_requests, index, flag, status)
Проверяет завершение любой ранее начатой операции
89. **MPI_TESTSOME** (incount, array_of_requests, outcount, array_of_indices,array_of_statuses)
Проверяет завершение заданных операций
90. **MPI_TEST_CANCELLED** (status, flag)
Проверяет отмену запроса
91. **MPI_TOPO_TEST** (comm, status)
Определяет тип связанной с коммуникатором топологии
92. **MPI_TYPE_COMMIT**(datatype)
Объявляет тип данных
93. **MPI_TYPE_CONTIGUOUS** (count, oldtype, newtype)
Создает непрерывный тип данных
94. **MPI_TYPE_EXTENT**(datatype, extent)
Определяет экстент типа данных

95. **MPI_TYPE_FREE** (datatype)
Отмечает объект типа данных для удаления
96. **MPI_TYPE_INDEXED** (count, array_of_blocklengths, array_of_displacements, oldtype, newtype)
Создает индексированный тип данных
97. **MPI_TYPE_LB** (datatype, displacement)
Возвращает нижнюю границу типа данных
98. **MPI_TYPE_SIZE** (datatype, size)
Возвращает число байтов, занятых элементами типа данных
99. **MPI_TYPE_STRUCT** (count, array_of_blocklengths, array_of_displacements, array_of_types, newtype)
Создает новый тип данных
100. **MPI_TYPE_UB** (datatype, displacement)
Возвращает верхнюю границу типа данных
101. **MPI_TYPE_VECTOR** (count, blocklength, stride, oldtype, newtype)
Создает векторный тип данных
102. **MPI_UNPACK** (inbuf, insize, position, outbuf, outcount, datatype, comm)
Распаковывает данные из непрерывного буфера
103. **MPI_WAIT** (request, status)
Ожидает завершения посылки или приема
104. **MPI_WAITALL** (count, array_of_requests, array_of_statuses)
Ожидает завершения всех обменов
105. **MPI_WAITANY** (count, array_of_requests, index, status)
Ожидает завершения любой из описанных посылки или приема
106. **MPI_WAITSOME** (incount, array_of_requests, outcount, array_of_indices, array_of_statuses)
Ожидает завершения некоторых заданных обменов
107. **MPI_WTICK ( )**
Возвращает величину разрешения при измерении времени
108. **MPI_WTIME** ( )
Возвращает полное время выполнения операций на используемом процессоре

Приложение 3. ОРГАНИЗАЦИИ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ В СЕТИ ПОД УПРАВЛЕНИЕМ WINDOWS NT

3.1. ОРГАНИЗАЦИЯ ПАРАЛЛЕЛЬНЫХ ВЫЧИСЛЕНИЙ

Для выполнения параллельных программ на базе интерфейса MPI необходимо:

1. Создать и настроить локальную сеть (Fast Ethernet, SCI др.) под управлением Windows NT.
2. Инсталлировать MPICH (версий 1.2.1, 1.2.2 или 1.2.3).
3. Настроить компилятор C++ для работы с MPICH.
4. Запустить приложение.

Пункт 1 является штатной для локальных сетей операцией и далее не рассматривается. Пункт 2 содержит ряд вариантов для для Unix подобных систем. Однако

для Windows NT возможности существенно ограничены и обычно используются дистрибутивы с расширением **.exe**. Это самоинсталлирующиеся версии, которые бесплатно можно получить по адресу http://www.mcs.anl.gov/mpi/mpich в Интернет. Последняя версия называется mpich.nt.1.2.3.src.exe (6.65 MB Jan 11, 2002). Для более подробного изучения инсталляции и работы с MPICH существуют руководства:

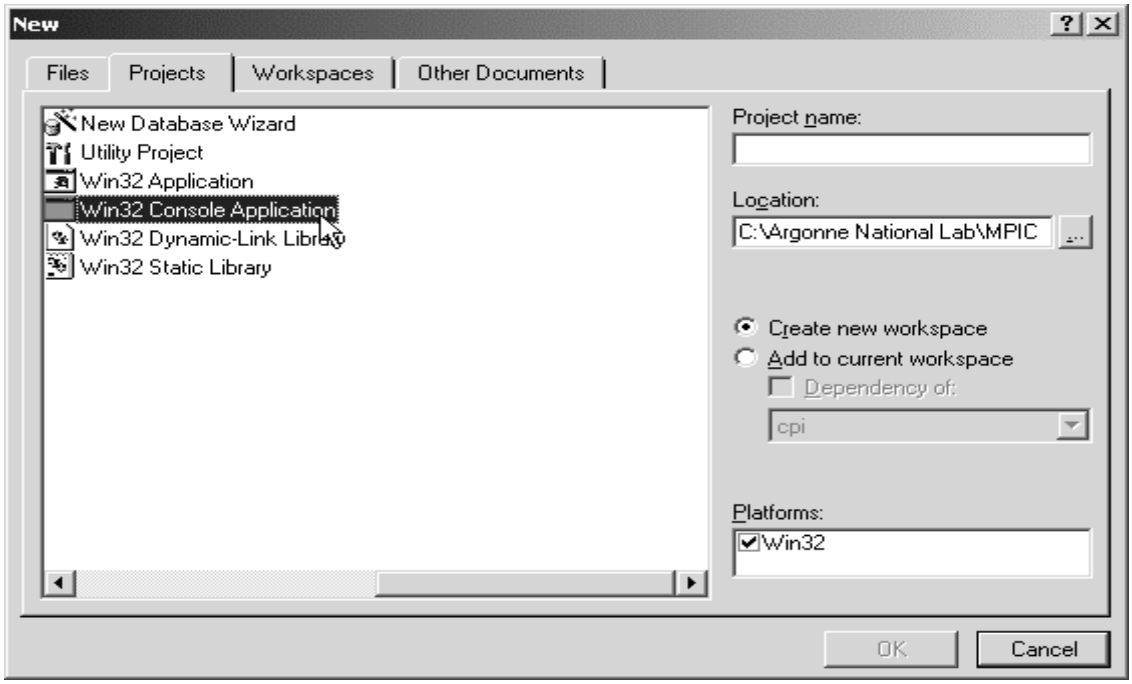
1. Instsllation Guide to mpich , a Portable Implementation of MPI
2. User's Guide for mpich , a Portable Implementation of MPI,

которые можно получить по адресу: http://www.mcs.anl/gov/mpi/mpich или ftp.mcs.anl.gov в директории pub/mpi. Если файл слишком велик, можно попытаться получить его по частям из pub/mpi/mpisplit и затем объединить. В дистрибутив MPICH входит также библиотека MPE.

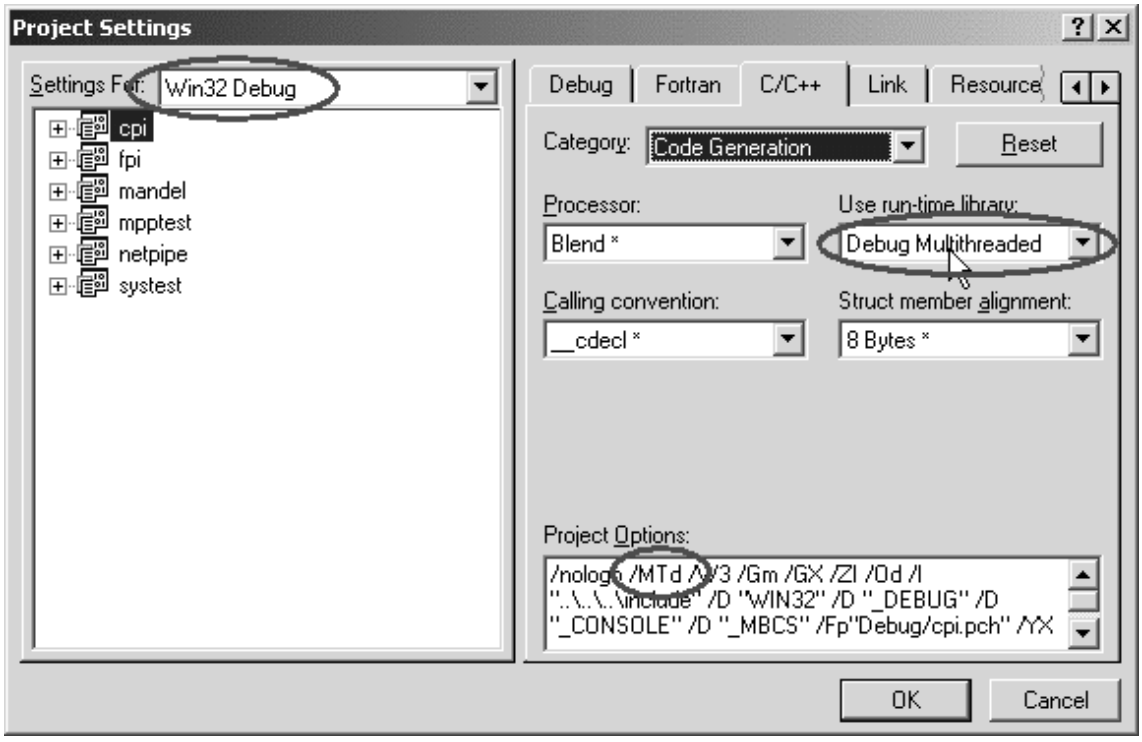
3.2. НАСТРОЙКИ VISUAL C++ 6.0 ДЛЯ РАБОТЫ С MPICH

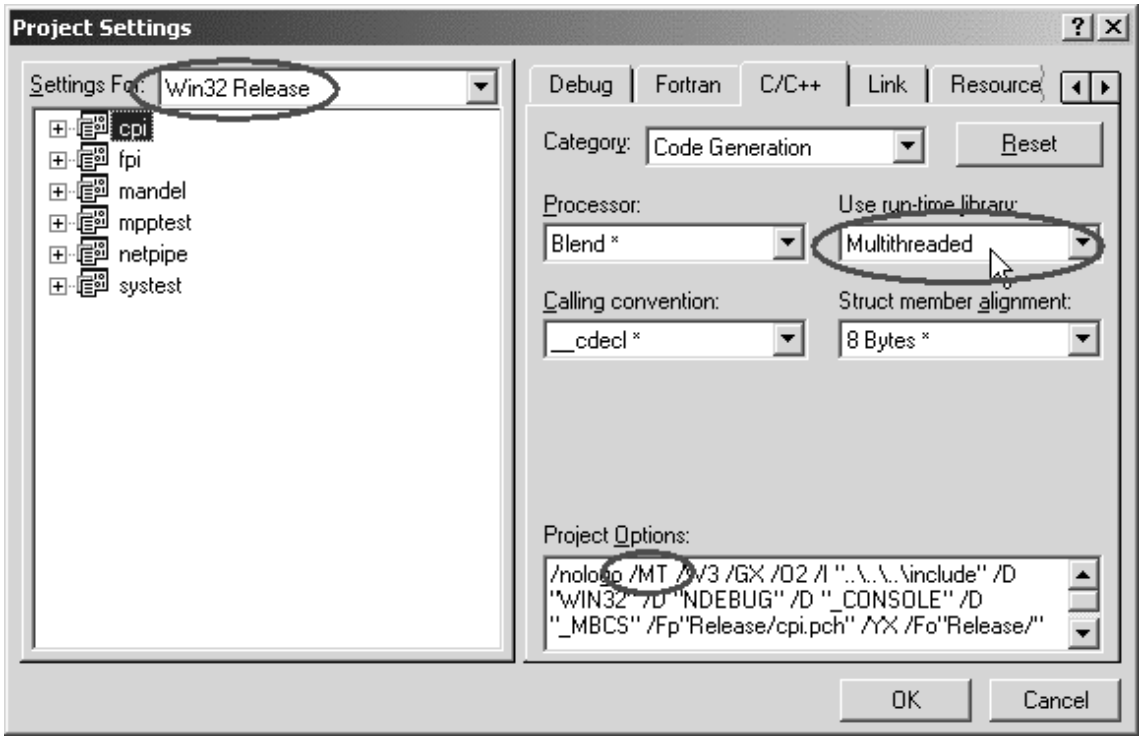
Чтобы создать новый проект mpich.nt с MSDEV, после того, как уже инсталлирован mpich.nt, необходимо выполнить следующие действия:

1. Открыть MS Developer Studio - Visual C++.
2. Создать новый проект с любым желаемым именем и в любой директории. Самым простым проектом является консольное приложение Win32 без файлов в нем.

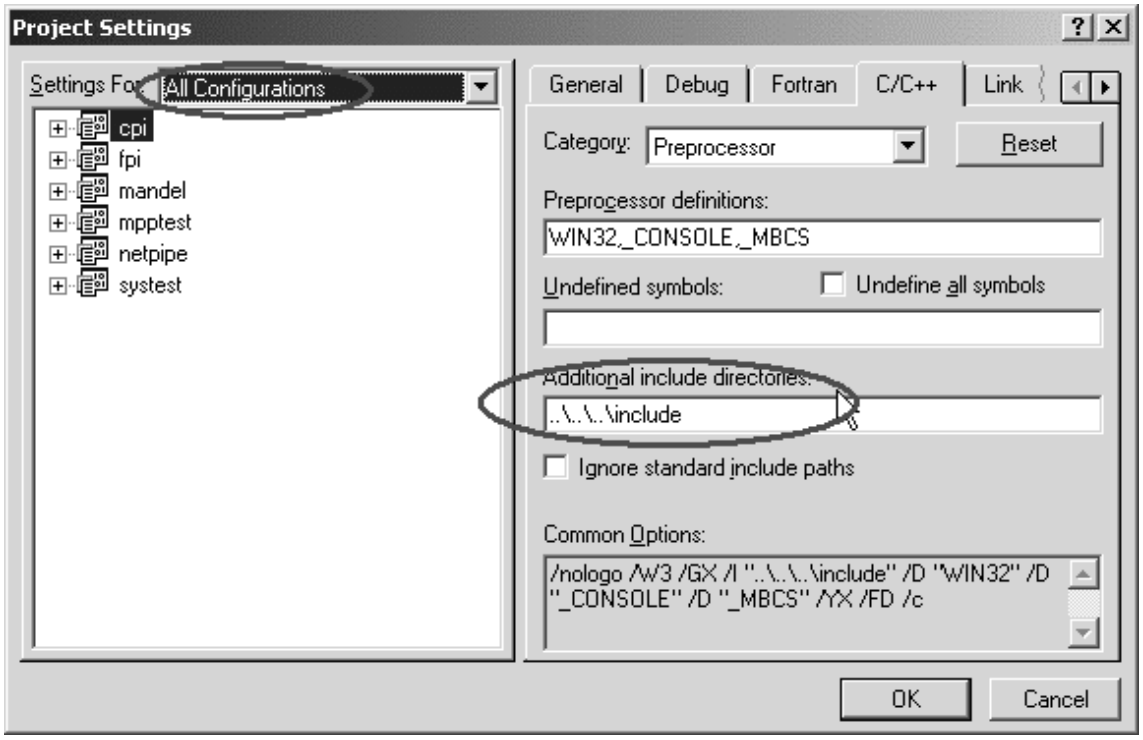


3. Закончить работу мастера по установке нового проекта.
4. Перейти на Project->Settings или нажать клавишу Alt F7, чтобы вызвать диалоговое окно настройки параметров нового проекта.
5. Установить параметры для использования многопоточных (multithreaded) библиотек. Установить указатели Debug и Release режимов.

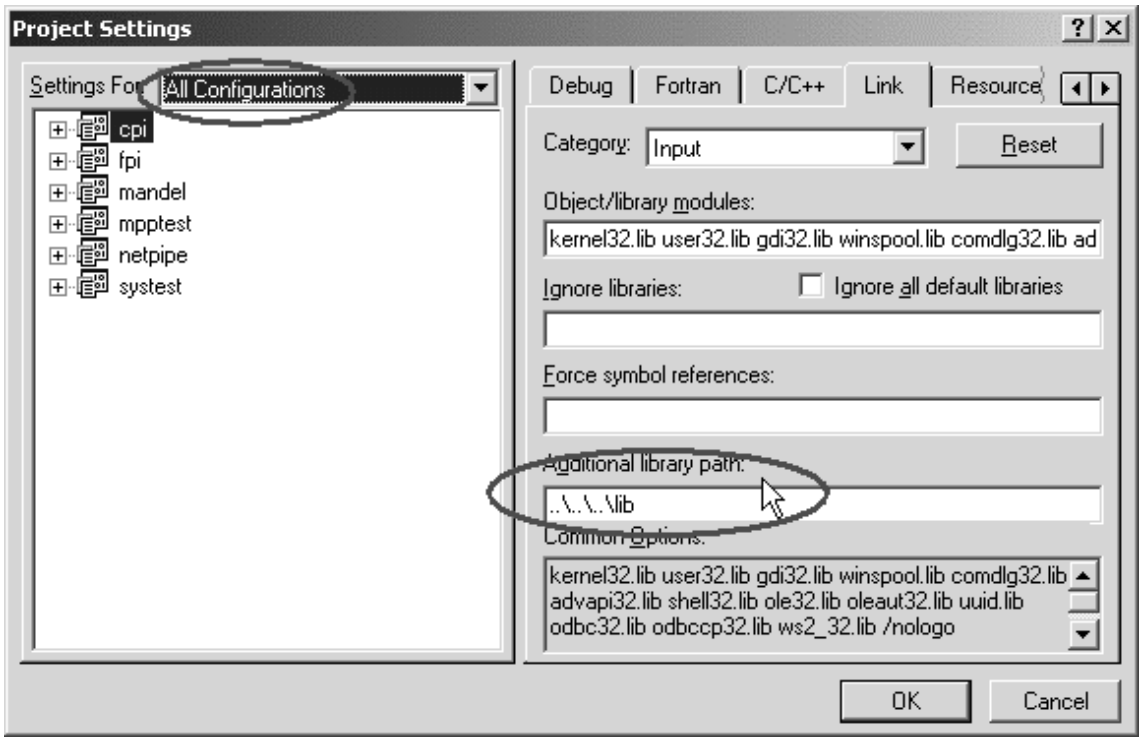

Project Settings
Settings For: Win32 Debug
cpi
fpi
mandel
mpptest
netpipe
systest
Debug
Fortran
C/C++
Link
Resource
Category: Code Generation
Reset
Processor:
Blend *
Use run-time library:
Debug Multithreaded
Calling convention:
__cdecl *
Struct member alignment:
8 Bytes *
Project Options:
/nologo /MTd /W3 /Gm /GX /ZI /Od /I
"..\..\..\include" /D "WIN32" /D "_DEBUG" /D
"_CONSOLE" /D "_MBCS" /Fp"Debug/cpi.pch" /YX
OK
Cancel


Project Settings
Settings For: Win32 Release
cpi
fpi
mandel
mpptest
netpipe
systest
Debug
Fortran
C/C++
Link
Resource
Category: Code Generation
Reset
Processor:
Blend *
Use run-time library:
Multithreaded
Calling convention:
__cdecl *
Struct member alignment:
8 Bytes *
Project Options:
/nologo /MT /W3 /GX /O2 /I "..\..\..\include" /D
"WIN32" /D "NDEBUG" /D "_CONSOLE" /D
"_MBCS" /Fp"Release/cpi.pch" /YX /Fo"Release/"
OK
Cancel

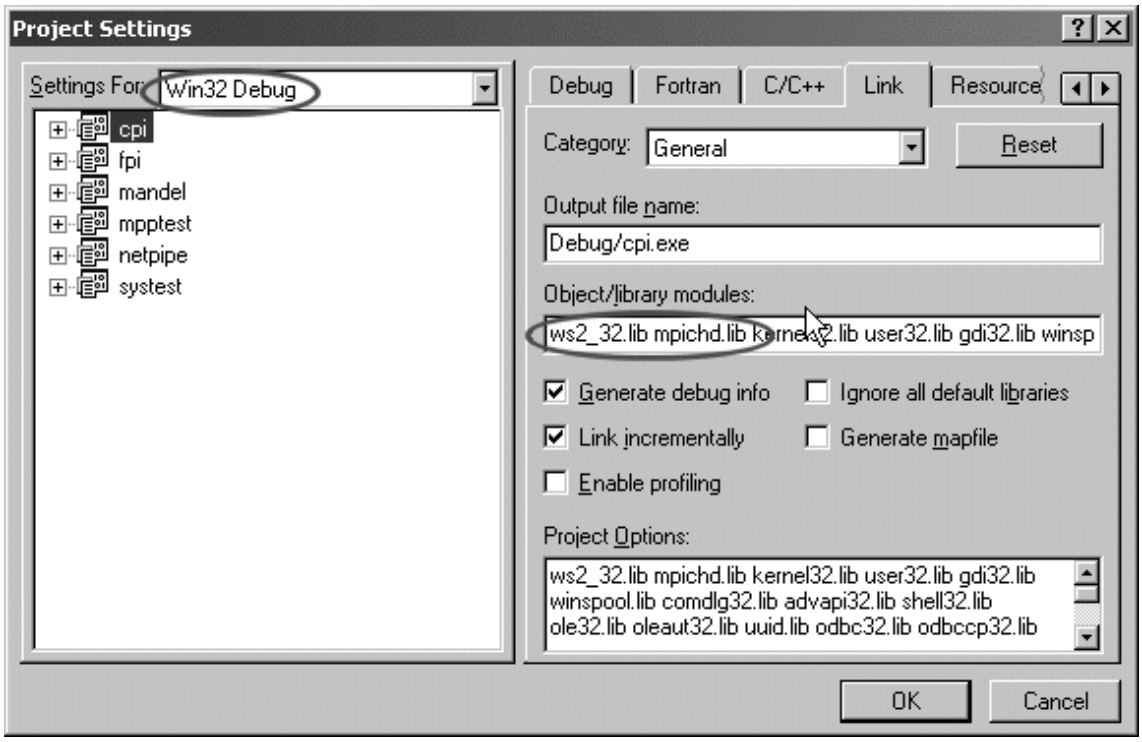
6. Установить include путь для всех целевых конфигураций: Argonne National Lab\MPICH.NT.1.2.1\SDK\include.

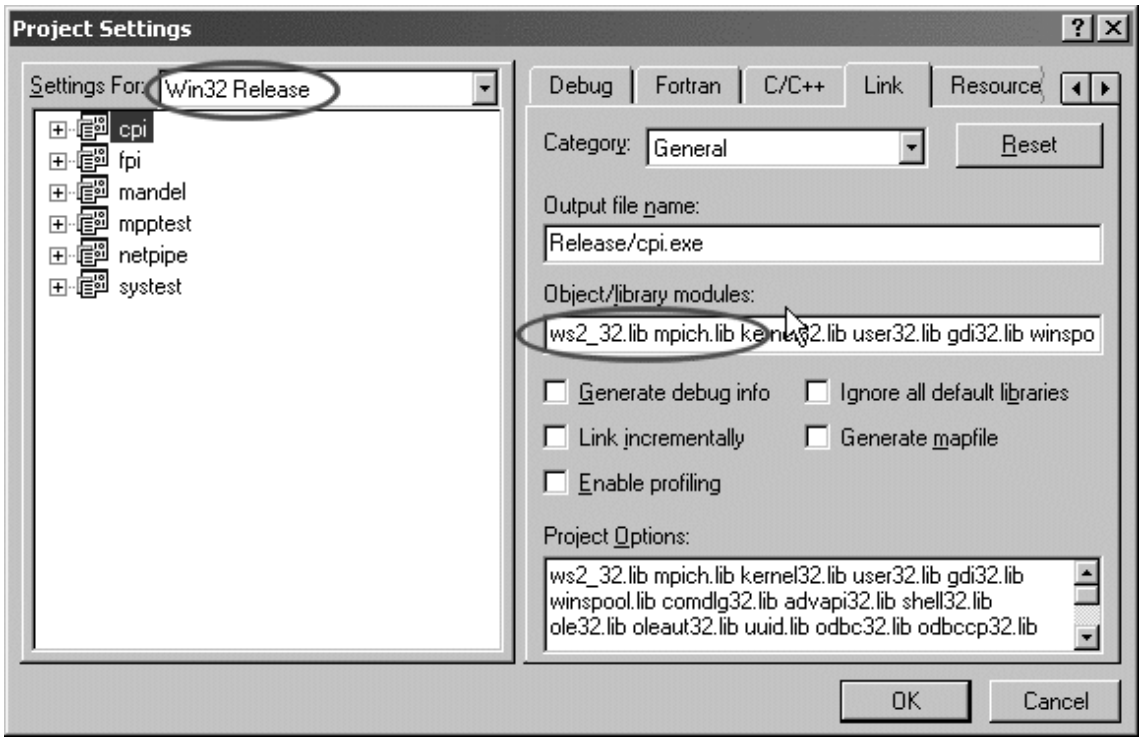


7. Установить lib путь для всех целевых конфигураций: Argonne National Lab\MPICH.NT.1.2.1\SDK\lib.

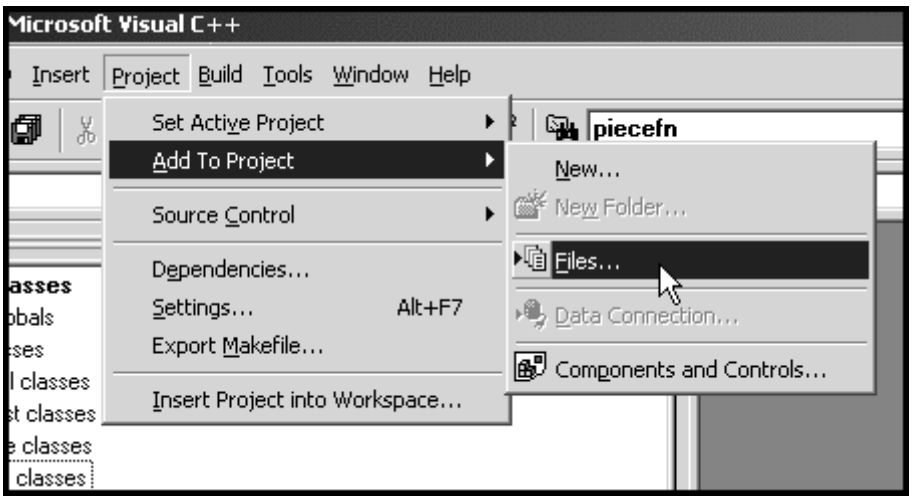


8. Добавить для всех конфигураций библиотеку ws2_32.lib (Это библиотека Microsoft Winsock2 library. Она по умолчанию находится в вашем пути к библиотекам). Добавить mpich.lib в режим release и mpichd.lib в режим debug.





9. Закрыть диалоговое окно установки проекта.Добавить к проекту исходные файлы.



10. Build.

Запуск приложения обычно производится из командной строки с помощью оператора mpirun, например:

mpi run -np 4 cpi

В этой команде могут быть также указаны различные опции, связанные с отладкой приложений, спецификой выполнения для различных устройств и платформ. Более подробно запуск приложений рассмотрен в главе 2.

Приложение 4. ХАРАКТЕРИСТИКИ КОММУНИКАЦИОННЫХ СЕТЕЙ ДЛЯ КЛАСТЕРОВ

Ниже приведены сравнительные характеристики некоторых сетей.

Таблица 1

Характеристики некоторых коммуникационных технологий

	SCI	**Myrinet**	**cLAN**	**ServerNet**	**Fast Ethernet**
Латентность	5,6 мкс	17 мкс	30 мкс	13 мкс	170 мкс
Пропускная способность (MPI)	80 Мбайт/с	40 Мбайт/с	100 Мбайт/с	180 Мбайт/с	10 Мбайт/с
Пропускная способность (аппаратная)	400 Мбайт/с	160 Мбайт/с	150 Мбайт/с	н/д	12,5 Мбайт/с
Реализация MPI	ScaMPI	HPVMи др.	MPI /Pro	MVICH	MPICH

Коммуникационные сети. Производительность коммуникационных сетей в кластерных системах определяют две основные характеристики: латентность – время начальной задержки при посылке сообщений и пропускная способность сети, определяющая скорость передачи информации по каналам связи. При этом важны не столько пиковые характеристики, заявляемые производителем, сколько реальные, достигаемые на уровне пользовательских приложений, например, на уровне MPI–приложений. В частности, после вызова пользователем функции посылки сообщения Send() сообщение последовательно пройдет через целый набор слоев, определяемых особенностями организации программного обеспечения и аппаратуры, прежде, чем покинуть процессор - отсюда и вариации на тему латентности. Кстати, наличие латентности определяет и тот факт, что максимальная скорость передачи по сети не может быть достигнута на сообщениях с небольшой длиной.

Коммуникационная технология Fast Ethernet. Сети этого типа используются чаще всего благодаря низкой стоимости оборудования. Однако большие накладные расходы на передачу сообщений в рамках Fast Ethernet приводят к серьезным ограничениям на спектр задач, которые можно эффективно решать на таком кластере. Если от кластера требуется большая универсальность, то нужно переходить на другие, более производительные коммуникационные технологии.

Коммуникационная технология SCI. Основа технологии SCI – это кольца, состоящие из быстрых однонаправленных линков с пиковой пропускной способностью на аппаратном уровне 400 Мбайт/с. Реальная пропускная способность на уровне MPI–приложений с использованием 32-разрядной шины PCI с частотой 33 МГц достигает 80 Мбайт/с, латентность – порядка 5,6 мкс.

Основной поставщик промышленных SCI–компонентов на современном рынке – норвежская компания Dolphin Interconnect Solutions. Вместе с компанией Scali Computer она предлагает интегрированное кластерное решение Wulfkit, в состав которого входят “основная” и “дочерняя” сетевые платы, два специальных кабеля и соответствующее программное обеспечение. Программный пакет Scali Software Platform включает средства конфигурирования и администрирования кластеров, и, что немаловажно, ScaMPI – оптимизированную под SCI реализацию интерфейса MPI (Message Passing Interface). Поддерживаются операционные системы Linux, Solaris и NT

Существующие кластеры, построенные на основе технологии SCI, содержат до 100 узлов, в качестве которых используются одно-, двух- и четырехпроцессорные компьютеры на базе Intel или UltraSPARC. Все узлы объединяются в топологию “двухмерный тор”, образуемую двумя SCI-кольцами с использованием двух сетевых адаптеров на каждом узле. Одним из преимуществ подобного решения является отказ от дорогостоящих многопортовых коммутаторов. Самый большой кластер на базе SCI установлен в университете города Падеборн (Германия) – 96 двухпроцессорных узлов на базе Pentium II.

Коммуникационная технология Myrinet. Сетевую технологию Myrinet представляет компания Myricom, которая впервые предложила свою коммуникационную технологию в 1994 году, а на сегодняшний день имеет уже более 1000 инсталляций по всему миру.

Узлы в Myrinet соединяются друг с другом через коммутатор (до 16 портов). Максимальная длина линий связи варьируется в зависимости от конкретной реализации. На данный момент наиболее распространены реализации сетей LAN и SAN. В последнем случае, при использовании в рамках вычислительной системы, длина кабеля не может превышать 3-х метров, а в LAN - 10,7 метра.

Приложение 5. ВАРИАНТЫ РЕШЕНИЯ ЗАДАНИЙ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Задание 3.1.

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int  argc;
char **argv;
{  int rank, size;
   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   printf( "Hello world from process %d of %d\n", rank, size );
   MPI_Finalize();
   return 0;
}
```

Задание 3.3.

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{  int rank, value, size;
   MPI_Status status;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   do
   {  if (rank == 0)
      {  scanf( "%d", &value );
         MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
      }
      else
      {  MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status );
         if (rank < size - 1)
              MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
      }
```

```
        printf( "Process %d got %d\n", rank, value );
        } while (value >= 0);
    MPI_Finalize( );
    return 0;   }
```

Задание 3.6.

```
#include "mpi.h"
#include <stdio.h>
int main(argc, argv)
int argc;
char **argv;
{ int rank, size, i, buf[1];
  MPI_Status status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  if (rank == 0)
  {    for (i=0; i<100*(size-1); i++)
       {  MPI_Recv( buf,1,MPI_INT,MPI_ANY_SOURSE, MPI_ANY_TAG,
                        MPI_COMM_WORLD, &status);
          printf( "Msg from %d with tag %d\n",  status.MPI_SOURCE, status.MPI_TAG );
       }
  }
  else
  {  for (i=0; i<100; i++)
           MPI_Send( buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD );
   }
   MPI_Finalize();
   return 0;
}
```

Задание 3.13.

```
/* пересылка вектора из 1000 элементов типа MPI_DOUBLE
   со страйдом 24 между элементами. Используем MPI_Type_vector  */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define NUMBER_OF_TESTS 10
int main( argc, argv )
int argc;
char **argv;
{  MPI_Datatype vec1, vec_n, old_types[2];
   MPI_Aint    indices[2];
   double    *buf, *lbuf,  t1, t2, tmin;
   register double *in_p, *out_p;
   int   i, j, k, nloop,  rank, n, stride, blocklens[2];
```

```
  MPI_Status   status;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );
  n   = 1000;    stride = 24;    nloop  = 100000/n;
  buf = (double *) malloc( n * stride * sizeof(double) );
  if (!buf)
  {  fprintf( stderr, "Could not allocate send/recv buffer of size %d\n",n * stride );
     MPI_Abort( MPI_COMM_WORLD, 1 );
  }
  lbuf = (double *) malloc( n * sizeof(double) );
  if (!lbuf)
  {   fprintf( stderr, "Could not allocated send/recv lbuffer of size %d\n", n );
      MPI_Abort( MPI_COMM_WORLD, 1 );
  }
  if (rank == 0) printf( "Kind\tn\tstride\ttime (sec)\tRate (MB/sec)\n" );
                        /* создаем новый векторный тип с заданным страйдом */
  MPI_Type_vector( n, 1, stride, MPI_DOUBLE, &vec1 );
  MPI_Type_commit( &vec1 );
  tmin = 1000;
  for (k=0; k<NUMBER_OF_TESTS; k++)
  {  if (rank == 0)
     {              /* убедимся, что оба процесса готовы к приему/передаче */
        MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, 14,
                MPI_BOTTOM, 0, MPI_INT, 1, 14, MPI_COMM_WORLD, &status );
        t1 = MPI_Wtime();
        for (j=0; j<nloop; j++)
        {
          MPI_Send( buf, 1, vec1, 1, k, MPI_COMM_WORLD );
          MPI_Recv( buf, 1, vec1, 1, k, MPI_COMM_WORLD, &status );
         }
        t2 = (MPI_Wtime() - t1) / nloop; ;            /*  время  для  пересылок*/
        if (t2 < tmin) tmin = t2;
      }
     else if (rank == 1)
      {                     /* убедимся, что оба процесса готовы к приему/передаче */
         MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, 14,
                MPI_BOTTOM, 0, MPI_INT, 0, 14, MPI_COMM_WORLD, &status );
         for (j=0; j<nloop; j++)
         {  MPI_Recv( buf, 1, vec1, 0, k, MPI_COMM_WORLD, &status );
            MPI_Send( buf, 1, vec1, 0, k, MPI_COMM_WORLD );
         }
       }
  }
tmin = tmin / 2.0;
if (rank == 0)
  printf( "Vector\t%d\t%d\t%f\t%f\n", n, stride, tmin, n*sizeof(double)*1.0e-6 / tmin );
```

```
 MPI_Type_free( &vec1 );
 MPI_Finalize( );
 return 0;
}
```

Задание 3.14.

```
/* пересылка вектора из 1000 элементов типа MPI_DOUBLE
   со страйдом 24 между элементами. Используем MPI_Type_struct  */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define NUMBER_OF_TESTS 10
int main( argc, argv )
int argc;
char **argv;
{ MPI_Datatype vec1, vec_n, old_types[2];
  MPI_Aint     indices[2];
  double       *buf, *lbuf,  t1, t2, tmin;
   register double *in_p, *out_p;
   int          i, j, k, nloop,  rank, n, stride,  blocklens[2];
   MPI_Status   status;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   n   = 1000;
   stride = 24;
   nloop  = 100000/n;
   buf = (double *) malloc( n * stride * sizeof(double) );
   if (!buf)
   {   fprintf( stderr, "Could not allocate send/recv buffer of size %d\n",n * stride );
       MPI_Abort( MPI_COMM_WORLD, 1 );
   }
   lbuf = (double *) malloc( n * sizeof(double) );
   if (!lbuf)
  {  fprintf( stderr, "Could not allocated send/recv lbuffer of size %d\n", n );
     MPI_Abort( MPI_COMM_WORLD, 1 );
  }
   if (rank == 0)   printf( "Kind\tn\tstride\ttime (sec)\tRate (MB/sec)\n" );
                                                /* создаем новый тип */
   blocklens[0] = 1;    blocklens[1] = 1;
   indices[0]   = 0;      indices[1]   = stride * sizeof(double);
   old_types[0] = MPI_DOUBLE;    old_types[1] = MPI_UB;
   MPI_Type_struct( 2, blocklens, indices, old_types, &vec_n );
   MPI_Type_commit( &vec_n );
   tmin = 1000;
   for (k=0; k<NUMBER_OF_TESTS; k++)
   {  if (rank == 0)  {   /* убедимся, что оба процесса готовы к приему/передаче */
```

```
        MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, 14,
                MPI_BOTTOM, 0, MPI_INT, 1, 14, MPI_COMM_WORLD, &status );
        t1 = MPI_Wtime();
        for (j=0; j<nloop; j++)
        {   MPI_Send( buf, n, vec_n, 1, k, MPI_COMM_WORLD );
            MPI_Recv( buf, n, vec_n, 1, k, MPI_COMM_WORLD, &status );
        }
        t2 = (MPI_Wtime() - t1) / nloop; ;            /*  время  для  пересылок*/
        if (t2 < tmin) tmin = t2;
      }
      else
        if (rank == 1)
        {    /* убедимся, что оба процесса готовы к приему/передаче */
           MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, 14,
              MPI_BOTTOM, 0, MPI_INT, 0, 14, MPI_COMM_WORLD,  &status );
           for (j=0; j<nloop; j++)
           {MPI_Recv( buf, n, vec_n, 0, k, MPI_COMM_WORLD, &status );
            MPI_Send( buf, n, vec_n, 0, k, MPI_COMM_WORLD );
           }
        }
   }
  tmin = tmin / 2.0;
  if (rank == 0)
    printf("Struct\t%d\t%d\t%f\t%f\n",n,stride,tmin, n*sizeof(double)*1.0e-6 / tmin );
  MPI_Type_free( &vec_n );
  MPI_Finalize( );
   return 0;
}
```

Задание 3.15.

```
/* пересылка вектора из 1000 элементов типа MPI_DOUBLE   со страйдом 24
   между элементами.  Самостоятельная упаковка и распаковка (не используем
   типы данных MPI) */
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define NUMBER_OF_TESTS 10
int main( argc, argv )
int argc;
char **argv;
{  MPI_Datatype vec1, vec_n, old_types[2];
  MPI_Aint     indices[2];
  double       *buf, *lbuf,  t1, t2, tmin;
  register double *in_p, *out_p;
  int          i, j, k, nloop,  rank, n, stride, blocklens[2];
  MPI_Status   status;
```

```
MPI_Init( &argc, &argv );
MPI_Comm_rank( MPI_COMM_WORLD, &rank );

 n   = 1000;    stride = 24;    nloop  = 100000/n;
 buf = (double *) malloc( n * stride * sizeof(double) );
 if (!buf)
 {   fprintf( stderr, "Could not allocate send/recv buffer of size %d\n",n * stride );
     MPI_Abort( MPI_COMM_WORLD, 1 );
 }
 lbuf = (double *) malloc( n * sizeof(double) );
 if (!lbuf)
 {   fprintf( stderr, "Could not allocated send/recv lbuffer of size %d\n", n );
     MPI_Abort( MPI_COMM_WORLD, 1 );
 }
 if (rank == 0)  printf( "Kind\tn\tstride\ttime (sec)\tRate (MB/sec)\n" );
 tmin = 1000;
 for (k=0; k<NUMBER_OF_TESTS; k++)
 {   if (rank == 0) {   /* убедимся, что оба процесса готовы к приему/передаче */
         MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 1, 14,
                 MPI_BOTTOM, 0, MPI_INT, 1, 14, MPI_COMM_WORLD,  &status );
         t1 = MPI_Wtime();
        for (j=0; j<nloop; j++)
        {    for (i=0; i<n; i++)                  /* создаем пользовательский тип  */
               lbuf[i] = buf[i*stride];
           MPI_Send( lbuf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD );
           MPI_Recv( lbuf, n, MPI_DOUBLE, 1, k, MPI_COMM_WORLD, &status );
           for (i=0; i<n; i++)     buf[i*stride] = lbuf[i];
         }
         t2 = (MPI_Wtime() - t1) / nloop;
         if (t2 < tmin) tmin = t2;
    }
    else
       if (rank == 1)
       {             /* убедимся, что оба процесса готовы к приему/передаче */
          MPI_Sendrecv( MPI_BOTTOM, 0, MPI_INT, 0, 14,
                 MPI_BOTTOM, 0, MPI_INT, 0, 14, MPI_COMM_WORLD,  &status );
          for (j=0; j<nloop; j++)
          {  MPI_Recv( lbuf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD, &status );
             for (i=0; i<n; i++)     buf[i*stride] = lbuf[i];
             for (i=0; i<n; i++)     lbuf[i] = buf[i*stride];
             MPI_Send( lbuf, n, MPI_DOUBLE, 0, k, MPI_COMM_WORLD );
          }
       }
 }
 tmin = tmin / 2.0;
```

```
 if (rank == 0)
   printf( "User\t%d\t%d\t%f\t%f\n",n,stride,tmin, n *sizeof(double)*1.0e-6 / tmin );
 MPI_Finalize( );
 return 0;
}
```

Задание 3.18.

```
#include <mpi.h>
#include <memory.h>
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
int main(int argc, char* argv[])
{  int iMyRank, i, gsize;
   MPI_Comm _Comm = MPI_COMM_WORLD;          /* в группе все процессы */
  MPI_Status status;
  double x[100][100][100], e[9][9][9];
  int oneslice, twoslice, threeslice, sizeof_double;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(_Comm, &gsize);
  MPI_Comm_rank(_Comm, &iMyRank);
                          /* извлекаем секцию a(1-17-2, 3-11, 2-10) и записываем в e */
 MPI_Type_extent(MPI_DOUBLE, &sizeof_double);
 MPI_Type_vector(9, 1, 2, MPI_DOUBLE, &oneslice);
 MPI_Type_hvector(9, 1, 100 * sizeof_double, oneslice, &twoslice);
 MPI_Type_hvector(9, 1, 100 * 100 * sizeof_double, twoslice, &threeslice);
 MPI_Type_commit(&threeslice);
 MPI_Sendrecv(&x[1][3][2], 1, threeslice, iMyRank, 0, e, 9*9*9, MPI_DOUBLE,
                    iMyRank, 0,  MPI_COMM_WORLD, &status);
 if (iMyRank == 0)
 {  printf("x = (");
    for (i = 0; i < 100; ++i)   printf("%f", x[i]);
    printf(")");
 }
MPI_Finalize();
return 0;
}
```

Задание 3.19.

```
#include <mpi.h>
#include <memory.h>
#include <stdlib.h>
#include <stdio.h>
#include <malloc.h>
#define MATRIXSIZE 10
int main(int argc, char* argv[])
```

```
{  int iMyRank, iRoot = 0, iRow, iColumn, gsize, sizeof_double;
   MPI_Comm _Comm = MPI_COMM_WORLD; /* в группе все процессы */
   MPI_Status status;
   double x[MATRIXSIZE][MATRIXSIZE], y[MATRIXSIZE][MATRIXSIZE];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(_Comm, &gsize);
  MPI_Comm_rank(_Comm, &iMyRank);
  if (iMyRank == iRoot)
  {  printf("Transpose the matrix\n");
     srand((int)((double)MPI_Wtime()*1e7));
     for (iRow = 0; iRow < MATRIXSIZE; ++iRow)
     {   for (iColumn = 0; iColumn < MATRIXSIZE; ++iColumn)
          {  x[iRow][iColumn] = rand() % 100;
          printf("%3.0f ", x[iRow][iColumn]);
          }
          printf("\n");
     }
  }
  MPI_Type_extent(MPI_DOUBLE, &sizeof_double);
  MPI_Type_vector(MATRIXSIZE, 1, MATRIXSIZE, MPI_DOUBLE, &iRow);
  MPI_Type_hvector(MATRIXSIZE, 1, sizeof_double, iRow, &iColumn);
  MPI_Type_commit(&iColumn);
  MPI_Sendrecv(x, 1, iColumn, iMyRank, 0, y, MATRIXSIZE * MATRIXSIZE,
                    MPI_DOUBLE, iMyRank, iRoot, MPI_COMM_WORLD, &status);
  if (iMyRank == iRoot)
  {   for (iRow = 0; iRow < MATRIXSIZE; ++iRow)
      { printf("\n");
        for (iColumn = 0; iColumn < MATRIXSIZE; ++iColumn)
                    printf("%3.0f ", y[iRow][iColumn]);
      }
  }
 MPI_Finalize();
return 0;
}
```

Задание 4.1.

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{  int rank, value;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   do {
         if (rank == 0) scanf( "%d", &value );
```

```
        MPI_Bcast( &value, 1, MPI_INT, 0, MPI_COMM_WORLD );
        printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);
    MPI_Finalize( );
    return 0;
}
```

Задание 4.2.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <mpi.h>
int main(int argc, char* argv[])
{  int  iRoot=0, iMyRank, iSize =0, i;
   double dSum = 0, dResult = 0, Vec_1[20], Vec_2[20];
   MPI_Comm _Comm = MPI_COMM_WORLD;       /* в группе все процессы */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(_Comm, &iMyRank);
   MPI_Comm_size(_Comm, &iSize);
  if ( iMyRank == iRoot )
  {  srand((int)(MPI_Wtime()*1e4));                    /* заполнение векторов */
     printf("Vector1=\n");
     for (i = 0; i < 20; ++i)
     {   Vec_1[i] = rand() % 11;
         printf("%3.0f ", Vec_1[i]);
      }
      printf("\nVector2=\n");
     for (i = 0; i < 20; ++i)
     {   Vec_2[i] = rand() % 11;
         printf("%3.0f ", Vec_2[i]);
     }
     printf("\n");
   }
  MPI_Bcast(Vec_1, 20, MPI_DOUBLE, iRoot, _Comm);
  MPI_Bcast(Vec_2, 20, MPI_DOUBLE, iRoot, _Comm);
                                          /* считаем локальную сумму */
  for (i = iMyRank, dSum = 0; i < 20; i += iSize)      dSum += Vec_1[i] * Vec_2[i];
  MPI_Reduce(&dSum, &dResult, 1, MPI_DOUBLE, MPI_SUM, iRoot, _Comm);
  if (iMyRank == iRoot)        printf("Result is %f\n", dResult);
  MPI_Finalize();
  return 0;
}
```

Задание 4.3.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <mpi.h>
#define n 10
#define m  5
int main(int argc, char* argv[])
{   int  iRoot=0, iMyRank, iSize =0, i, j;
    double dSum[n], dResult[n], Vec_1[m], Matr[m][n];

    MPI_Comm _Comm = MPI_COMM_WORLD;    /* в группе все процессы */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(_Comm, &iMyRank);
    MPI_Comm_size(_Comm, &iSize);
    if ( iMyRank == iRoot )
    {   srand((int)(MPI_Wtime()*1e4));                    /* заполнение векторов */
        printf("Vector1=\n");
        for (i = 0; i < m; ++i)
        { Vec_1[i] = rand() % 11;
          printf("%2.0f ", Vec_1[i]);
         }
         printf("\nMatrix=\n");
        for (i = 0; i < m; ++i)
        { for (j = 0; j < n; ++j)
          {  Matr[i][j] = rand() % 11;
             printf("%2.0f ", Matr[i][j]);
          }
          printf("\n");
        }
          printf("\n");
      }
   MPI_Bcast(Vec_1, m, MPI_DOUBLE, iRoot, _Comm);
   for (i = 0; i < n; ++i)
         MPI_Bcast(Matr[i], m, MPI_DOUBLE, iRoot, _Comm);
                                        /* считаем локальную сумму */
    for (j = 0; j < n; ++j)
    {  dSum[j] = 0;
       for (i = iMyRank; i < m; i += iSize)
          dSum[j] += Vec_1[i] * Matr[i][j];
    }

   MPI_Reduce(&dSum, dResult, n, MPI_DOUBLE, MPI_SUM, iRoot, _Comm);

   if (iMyRank == iRoot)
```

```
  {  printf("Result is:\n");
     for (i = 0; i < n; ++i)  printf("%3.0f ", dResult[i]);
   }
  MPI_Finalize();
  return 0;
}
```

Задание 4.5.

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{  struct { int a; double b } value;
   MPI_Datatype mystruct, old_types[2];
   int          blocklens[2],   rank;
   MPI_Aint     indices[2];
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   blocklens[0] = 1;    blocklens[1] = 1;     /* базовые типы */
   old_types[0] = MPI_INT;    old_types[1] = MPI_DOUBLE;
                                              /* адреса каждого элемента */
   MPI_Address( &value.a, &indices[0] );    MPI_Address( &value.b, &indices[1] );
   indices[1] = indices[1] - indices[0];    indices[0] = 0;        /*новый  тип  данных */
   MPI_Type_struct( 2, blocklens, indices, old_types, &mystruct );
   MPI_Type_commit( &mystruct );
   do
   {   if (rank == 0)    scanf( "%d %lf", &value.a, &value.b );
       MPI_Bcast( &value, 1, mystruct, 0, MPI_COMM_WORLD );
      printf( "Process %d got %d and %lf\n", rank, value.a, value.b );
   } while (value.a >= 0);
   MPI_Type_free( &mystruct );
   MPI_Finalize( );
   return 0;
}
```

Задание 4.6.

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{  int          rank,  Packsize, position, a;
   double       b;
   char         Packbuf[100];
   MPI_Init( &argc, &argv );
```

```
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   do
   {  if (rank == 0)
      {    scanf( "%d %lf", &a, &b );
           Packsize = 0;
           MPI_Pack( &a, 1, MPI_INT, Packbuf, 100, &Packsize,
                     MPI_COMM_WORLD );
           MPI_Pack( &b, 1, MPI_DOUBLE, Packbuf, 100,
                     &Packsize,MPI_COMM_WORLD);
      }
      MPI_Bcast( &Packsize, 1, MPI_INT, 0, MPI_COMM_WORLD );
      MPI_Bcast( Packbuf, Packsize, MPI_PACKED, 0, MPI_COMM_WORLD );
      if (rank != 0)
      {    position = 0;
           MPI_UnPack( Packbuf, Packsize, &position, &a, 1, MPI_INT,
                       MPI_COMM_WORLD );
           MPI_UnPack( Packbuf, Packsize, &position, &b, 1,
                       MPI_DOUBLE,MPI_COMM_WORLD );
       }
       printf( "Process %d got %d and %lf\n", rank, a, b );
   } while (a >= 0);
   MPI_Finalize( );
   return 0;
}
```

Задание 4.8.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define NUMBER_OF_TESTS 10
int main( argc, argv )
int argc;
char **argv;
{  int         rank,    size, j, k, nloop;
   double      t1, t2, tmin, d_in, d_out;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (rank == 0 && size == 1)             printf( "Kind\t\tnp\ttime (sec)\n" );
   nloop = 1000;    tmin  = 1000;
   for (k=0; k<NUMBER_OF_TESTS; k++)
   {    MPI_Barrier( MPI_COMM_WORLD );
        d_in = 1.0;
        t1 = MPI_Wtime();
        for (j=0; j<nloop; j++)
           MPI_Allreduce( &d_in, &d_out, 1, MPI_DOUBLE, MPI_SUM,
```

```
                                MPI_COMM_WORLD );
        t2 = (MPI_Wtime() - t1) / nloop;          if (t2 < tmin) tmin = t2;
    }
   if (rank == 0) printf( "Allreduce\t%d\t%f\n", size, tmin );
   MPI_Finalize( );
   return 0;    }
```

Задание 4.9.

```
#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"
#define NUMBER_OF_TESTS 10
int main( argc, argv )
int argc;
char **argv;
{  double      t1, t2, tmin;
   int         j, k, nloop, rank,   size;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (rank == 0 && size == 1)              printf( "Kind\tnp\ttime (sec)\n" );
   nloop = 1000;    tmin  = 1000;
   for (k=0; k<NUMBER_OF_TESTS; k++)
   {  MPI_Barrier( MPI_COMM_WORLD );
      t1 = MPI_Wtime();
      for (j=0; j<nloop; j++)
          MPI_Barrier( MPI_COMM_WORLD );
       t2 = (MPI_Wtime() - t1) / nloop;
       if (t2 < tmin) tmin = t2;
   }
   if (rank == 0)     printf( "Barrier\t%d\t%f\n", size, tmin );
   MPI_Finalize( );
   return 0;
}
```

Задание 4.11.

```
int main( argc, argv )
int argc;
char *argv[];
{  double A[8][8], alocal[4][4];
   int i, j, r, rank, size;
   MPI_Datatype stype, t[2], vtype;
   MPI_Aint     displs[2];
   int          blklen[2],   sendcount[4], sdispls[4];
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
```

```
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (size != 4)
   {    fprintf( stderr, "This program requires exactly four processors\n" );
        MPI_Abort( MPI_COMM_WORLD, 1 );
   }

   if (rank == 0)
   {                                        /* Инициализация матрицы */
      for (j=0; j<8; j++)
        for (i=0; i<8; i++)        A[i][j] = 1.0 + i / 10.0 + j / 100.0;
                                            /* новый векторный тип для подматриц */
      MPI_Type_vector( 4, 4, 8, MPI_DOUBLE, &vtype );
      t[0] = vtype;    t[1] = MPI_UB;
     displs[0] = 0;    displs[1] = 4 * sizeof(double);
     blklen[0] = 1;    blklen[1] = 1;
     MPI_Type_struct( 2, blklen, displs, t, &stype );
     MPI_Type_commit( &stype );
     sendcount[0] = 1;  sendcount[1] = 1;   sendcount[2] = 1;   sendcount[3] = 1;
     sdispls[0] = 0;         sdispls[1] = 1;      sdispls[2] = 8;        sdispls[3] = 9;
     MPI_Scatterv( &A[0][0], sendcount, sdispls, stype, &alocal[0][0], 4*4,
                         MPI_DOUBLE, 0, MPI_COMM_WORLD );
   }
   else
      MPI_Scatterv( (void *)0, (void *)0, (void *)0, MPI_DATATYPE_NULL,
                          &alocal[0][0], 4*4, MPI_DOUBLE, 0, MPI_COMM_WORLD );
                                 /* каждый процесс печатает свою часть матрицы */
   for (r = 0; r<size; r++)
   {  if (rank == r)
       {    printf( "Output for process %d\n", r );
            for (j=0; j<4; j++)
           {for (i=0; i<4; i++)     printf( "%.2f ", alocal[i][j] );
            printf( "\n" );
            }
           fflush( stdout );
        }
        MPI_Barrier( MPI_COMM_WORLD );
    }
   MPI_Finalize( );
   return 0;
}
```

Задание 4.12.

```
#include <stdio.h>
#include "mpi.h"
#define maxn 12       /* В этом примере сетка 12 x 12,  4 процесса */
int main( argc, argv )
```

```
int argc;
char **argv;
{  int rank, value, size, errcnt, toterr, i, j;
   MPI_Status status;
   double x[12][12], xlocal[(12/4)+2][12];

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (size != 4) MPI_Abort( MPI_COMM_WORLD, 1 );
                  /* xlocal[][0] – нижние теневые точки, xlocal[][maxn+2] - верхние */
   for (i=1; i<=maxn/size; i++)
       for (j=0; j<maxn; j++)     xlocal[i][j] = rank;
   for (j=0; j<maxn; j++)
   {   xlocal[0][j] = -1;      xlocal[maxn/size+1][j] = -1;     }
 /* передаем - вверх, получаем -  снизу используем xlocal[i]  вместо xlocal[i][0] */
   if (rank < size - 1)
      MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1, 0,
                    MPI_COMM_WORLD );
   if (rank > 0)
      MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                     &status );
                                          /* передаем  вниз, получаем  сверху*/
   if (rank > 0)
     MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD );
   if (rank < size - 1)
     MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1,
                    MPI_COMM_WORLD, &status );
                                 /* Проверяем на корректность результаты */
   errcnt = 0;
   for (i=1; i<=maxn/size; i++)
       for (j=0; j<maxn; j++)     if (xlocal[i][j] != rank) errcnt++;
   for (j=0; j<maxn; j++)
   {   if (xlocal[0][j] != rank - 1) errcnt++;
       if (rank < size-1 && xlocal[maxn/size+1][j] != rank + 1) errcnt++;
   }

  MPI_Reduce( &errcnt, &toterr, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );
  if (rank == 0)
   {   if (toterr)    printf( "! found %d errors\n", toterr );
       else     printf( "No errors\n" );
   }
   MPI_Finalize( );
   return 0;
}
```

Задание 4.14.

```
#include <stdio.h>
#include "mpi.h"
#define maxn 12              /* В этом примере сетка 12 x 12,  4 процесса */
int main( argc, argv )
int argc;
char **argv;
{  int rank, value, size, errcnt, toterr, i, j, up_nbr, down_nbr;
   MPI_Status status;
   double x[12][12],xlocal[(12/4)+2][12];

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (size != 4) MPI_Abort( MPI_COMM_WORLD, 1 );
                /* xlocal[][0] – нижние теневые точки, xlocal[][maxn+2] - верхние */
   for (i=1; i<=maxn/size; i++)
     for (j=0; j<maxn; j++)     xlocal[i][j] = rank;
   for (j=0; j<maxn; j++)
   {   xlocal[0][j] = -1;
       xlocal[maxn/size+1][j] = -1;
   }
          /* Передаем и получаем от нижних процессов.
                                  Используем xlocal[i] вместо xlocal[i][0] */
   up_nbr = rank + 1;
   if (up_nbr >= size) up_nbr = MPI_PROC_NULL;
   down_nbr = rank - 1;
   if (down_nbr < 0) down_nbr = MPI_PROC_NULL;
   MPI_Sendrecv( xlocal[maxn/size], maxn, MPI_DOUBLE, up_nbr, 0,
     xlocal[0], maxn, MPI_DOUBLE, down_nbr, 0,   MPI_COMM_WORLD, &status );
                                     /* Передаем и получаем от верхних процессов. */
   MPI_Sendrecv( xlocal[1], maxn, MPI_DOUBLE, down_nbr, 1, xlocal[maxn/size+1],
                  maxn, MPI_DOUBLE, up_nbr, 1,   MPI_COMM_WORLD, &status );
                               /* Проверяем на корректность результаты */
   errcnt = 0;
   for (i=1; i<=maxn/size; i++)
      for (j=0; j<maxn; j++)     if (xlocal[i][j] != rank) errcnt++;
   for (j=0; j<maxn; j++)
   {   if (xlocal[0][j] != rank - 1) errcnt++;
       if (rank < size-1 && xlocal[maxn/size+1][j] != rank + 1) errcnt++;
   }

  MPI_Reduce( &errcnt, &toterr, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );
   if (rank == 0)    {
      if (toterr)    printf( "! found %d errors\n", toterr );
```

```
    else   printf( "No errors\n" );
  }
  MPI_Finalize( );
  return 0;
}
```

Задание 4.16.

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#define maxn 12           /* В этом примере сетка 12 x 12,  4 процесса */
int main( argc, argv )
int argc;
char **argv;
{  int       rank, value, size, errcnt, toterr, i, j, itcnt,  i_first, i_last;
   MPI_Status status;
   double     diffnorm, gdiffnorm, xlocal[(12/4)+2][12],  xnew[(12/3)+2][12];

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (size != 4) MPI_Abort( MPI_COMM_WORLD, 1 );
        /* xlocal[][0] – нижние теневые точки, xlocal[][maxn+2] - верхние */
  /* В первом и последних процессах на одну строку меньше внутренних точек */
   i_first = 1;    i_last  = maxn/size;
   if (rank == 0)       i_first++;
   if (rank == size - 1) i_last--;
   for (i=1; i<=maxn/size; i++)
      for (j=0; j<maxn; j++)     xlocal[i][j] = rank;
   for (j=0; j<maxn; j++)
   {   xlocal[i_first-1][j] = -1;
       xlocal[i_last+1][j] = -1;
   }
   itcnt = 0;
   do
   {        /* передаем  вверх, получаем  снизу,   xlocal[i]  вместо xlocal[i][0] */
      if (rank < size - 1)
        MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1, 0,
                        MPI_COMM_WORLD );
    if (rank > 0)
      MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,  MPI_COMM_WORLD,
                      &status );
                                        /* передаем  вниз, получаем  сверху*/
   if (rank > 0)
     MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD );
   if (rank < size - 1)
```

```
      MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1,
                              MPI_COMM_WORLD,  &status );
                                    /* Вычисляем новые значения  ( не на границах) */
    itcnt ++;
    diffnorm = 0.0;
     for (i=i_first; i<=i_last; i++)
        for (j=1; j<maxn-1; j++)
       {  xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] + xlocal[i+1][j] + xlocal[i-1][j]) / 4.0;
          diffnorm += (xnew[i][j] - xlocal[i][j]) * (xnew[i][j] - xlocal[i][j]);
        }
                                    /* присваиваем новые значения внутренним точкам */
      for (i=i_first; i<=i_last; i++)
        for (j=1; j<maxn-1; j++)              xlocal[i][j] = xnew[i][j];
      MPI_Allreduce( &diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                             MPI_COMM_WORLD);
       gdiffnorm = sqrt( gdiffnorm );
       if (rank == 0) printf( "At iteration %d, diff is %e\n", itcnt,  gdiffnorm );
   } while (gdiffnorm > 1.0e-2 && itcnt < 100);
   MPI_Finalize( );
   return 0;
}
```

Задание 4.17.

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"
#define maxn 12         /* В этом примере сетка 12 x 12,  4 процесса */
int main( argc, argv )
int argc;
char **argv;
{  int       rank, value, size, errcnt, toterr, i, j, itcnt,   i_first, i_last;
   MPI_Status status;
   double     diffnorm, gdiffnorm,  xlocal[(12/4)+2][12], xnew[(12/3)+2][12],
              x[maxn][maxn];

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (size != 4) MPI_Abort( MPI_COMM_WORLD, 1 );
            /* xlocal[][0] – нижние теневые точки, xlocal[][maxn+2] - верхние */
                        /* заполняем данные из файла */
   if (rank == 0)
   {   FILE *fp; fp =fopen( "in.dat", "r" );
       if (!fp) MPI_Abort( MPI_COMM_WORLD, 1 );
                     /* заполняем все внутренние точки и граничные */
       for (i=maxn-1; i>=0; i--)
```

```
    {   for (j=0; j<maxn; j++)              fscanf( fp, "%lf", &x[i][j] );
        fscanf( fp, "\n" );
    }
  }
  MPI_Scatter( x[0], maxn * (maxn/size), MPI_DOUBLE,
           xlocal[1], maxn * (maxn/size), MPI_DOUBLE, 0, MPI_COMM_WORLD );
/* В первом и последних процессах на одну строку меньше внутренних точек */
  i_first = 1;    i_last  = maxn/size;
  if (rank == 0)   i_first++;
  if (rank == size - 1) i_last--;
  itcnt = 0;
  do   {                                    /* передаем  вверх, получаем  снизу*/
     if (rank < size - 1)
       MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1, 0,
                    MPI_COMM_WORLD );
     if (rank > 0)
      MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                    &status );
                                        /* передаем  вниз, получаем  сверху*/
   if (rank > 0)
     MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,MPI_COMM_WORLD );
   if (rank < size - 1)
      MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1,
                    MPI_COMM_WORLD,  &status );
                              /* Вычисляем новые значения  ( не на границах) */
  itcnt ++;             diffnorm = 0.0;
  for (i=i_first; i<=i_last; i++)
    for (j=1; j<maxn-1; j++)
     { xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] +  xlocal[i+1][j] + xlocal[i-1][j]) / 4.0;
       diffnorm += (xnew[i][j] - xlocal[i][j]) *  (xnew[i][j] - xlocal[i][j]);
     }
                              /* присваиваем новые значения внутренним точкам */
  for (i=i_first; i<=i_last; i++)
      for (j=1; j<maxn-1; j++)  xlocal[i][j] = xnew[i][j];
       MPI_Allreduce( &diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                          MPI_COMM_WORLD);
   gdiffnorm = sqrt( gdiffnorm );
   if (rank == 0) printf( "At iteration %d, diff is %e\n", itcnt,   gdiffnorm );
  } while (gdiffnorm > 1.0e-2 && itcnt < 100);

                                        /* собираем данные в  x и печатаем */
  MPI_Gather( xlocal[1], maxn * (maxn/size), MPI_DOUBLE,
                  x, maxn * (maxn/size), MPI_DOUBLE, 0, MPI_COMM_WORLD );
  if (rank == 0)
  {   printf( "Final solution is\n" );
      for (i=maxn-1; i>=0; i--) {
```

```
        for (j=0; j<maxn; j++)  printf( "%f ", x[i][j] );
        printf( "\n" );
      }
   }
   MPI_Finalize( );
   return 0;
}
```

Задание 4.20.

```
#include <stdio.h>
#include <malloc.h>
#include <STDLIB.H>
#include <mpi.h>
int main(int argc, char* argv[])
{  int iGatherSize=0, iRoot=0, iMyRank, i, j;
   int sendarray[100][150], *RBuffer=0;
   int *displs=0, *rcounts=0, stride = 255/*>100*/;
   MPI_Comm _Comm = MPI_COMM_WORLD;   /* в группе все процессы */
   MPI_Datatype stype;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(_Comm, &iMyRank);
                        /* Главному процессу нужно выделить память для буфера */
   if (iMyRank == iRoot)
   {    MPI_Comm_size(_Comm, &iGatherSize);
        RBuffer = (int*)malloc(stride*iGatherSize*sizeof(int));
       displs  = (int*)malloc(iGatherSize * sizeof(int));
       rcounts = (int*)malloc(iGatherSize * sizeof(int));
       for (i = 0; i < iGatherSize; ++i)
       { displs[i] = stride * i;rcounts[i] = 100;  }
    }
    srand((int)(MPI_Wtime()*1e4));
    for (j = 0; j < 100; ++j)
      for (i = 0; i < 150; ++i)   sendarray[j][i] = rand() % 999;
   MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
   MPI_Type_commit(&stype);
                                        /* сбор данных */
    MPI_Gatherv(sendarray,1,stype,RBuffer,rcounts,displs,MPI_INT,iRoot, _Comm);

              /* выведем данные на экран (выводятся только главном процессе,
                 в остальных iGatherSize == 0) */
    for (i = 0; i < iGatherSize; ++i)
    {   printf("Process %i retuned:\n",i);
       for (j = 0; j < 100; ++j)   printf("%4i ", RBuffer[i*stride + j]);
        printf("\n");
     }
```

```
    if (iMyRank == iRoot)
    {  free(RBuffer);   free(displs);     free(rcounts);}
   MPI_Finalize();
   return 0;
}
```

Задание 4.21.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <mpi.h>
#define LEN 1000
int main(int argc, char* argv[])
{  int  iRoot=0, iMyRank, iSize =0, i;
   double dVal[LEN];
   MPI_Comm _Comm = MPI_COMM_WORLD;    /* в группе все процессы */
   struct {  double value;        int index;   } In, Out;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(_Comm, &iMyRank);
   MPI_Comm_size(_Comm, &iSize);
   srand((int)(MPI_Wtime()*1e4));
   for (i = 0; i < LEN; ++i)
   {   dVal[i] = rand() * (double)((double)rand() / ((unsigned)rand()+3));
       printf("%4.2f ", dVal[i]);
    }
                                          /* выбор локального минимума */
    In.value = dVal[0];In.index = 0;
    for (i = 1; i < LEN; ++i)
     if (In.value < dVal[i])
      {   In.value = dVal[i];       In.index = i;  }
   In.index  += iMyRank*LEN;
   MPI_Reduce(&In, &Out, 1, MPI_DOUBLE_INT, MPI_MAXLOC, iRoot, _Comm);
   if (iMyRank == iRoot)
     printf("\n\nMaximum is %f, found in process %i with index %i\n", Out.value,
               Out.index / LEN, Out.index % LEN);
     MPI_Finalize();
}
```

Задание 5.2.

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{  int rank, size;
```

```
   MPI_Comm new_comm;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_split( MPI_COMM_WORLD, rank == 0, 0, &new_comm );
   if (rank == 0)   master_io( MPI_COMM_WORLD, new_comm );
   else             slave_io( MPI_COMM_WORLD, new_comm );
   MPI_Finalize( );
   return 0;
}
                                                    /* подпрограмма  master */
int master_io( master_comm, comm )
MPI_Comm comm;
{  int       i,j, size;
   char      buf[256];
   MPI_Status status;
   MPI_Comm_size( master_comm, &size );
   for (j=1; j<=2; j++)
   {  for (i=1; i<size; i++)
        {   MPI_Recv( buf, 256, MPI_CHAR, i, 0, master_comm, &status );
            fputs( buf, stdout );
        }
    }
}
                                                    /* подпрограмма  slave */
int slave_io( master_comm, comm )
MPI_Comm comm;
{  char buf[256];
   int  rank;
   MPI_Comm_rank( comm, &rank );
   sprintf( buf, "Hello from slave %d\n", rank );
   MPI_Send( buf, strlen(buf) + 1, MPI_CHAR, 0, 0, master_comm );
    sprintf( buf, "Goodbye from slave %d\n", rank );
   MPI_Send( buf, strlen(buf) + 1, MPI_CHAR, 0, 0, master_comm );
   return 0;
}
```

Задание 5.4.

```
#include <math.h>
#include "mpi.h"
#include "mpe.h"
#define CHUNKSIZE 1000
#define REQUEST 1
#define REPLY 2
int main.( int argc, char *argv[] )
{  int in, out, i, iters, max, ix, iy, ranks [1], done, temp,iter;
   double x, y, Pi, error, epsilon;
```

```
   int numprocs,myid, server, totalin, totalout, workerid, rands [CHUNKSIZE] , request;
   MPI_Comm world, workers;
   MPI_Group world_group, worker_group;
   MPI_Status status;

   MPI_Init(&rgc,&argv);    world = MPI_COMM_WORLD;
   MPI_Comm_size(world,&numprocs);
   MPI_Comm_rank (world, &myid);
   server = numprocs-1;                    /* последний процесс – сервер */
   if (myid == 0)        sscanf( argv[l], "%f", &epsilon );
    MPI_Bcast( &epsilon, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD );
                                                  /* создание нового коммуникатора */
  MPI_Comm_group( world, &world_group );
  ranks [0] = server;
  MPI_Group_excl( world_group, 1, ranks, &worker_group );
  MPI_Comm_create( world, worker_group, &workers );
  MPI_Group_free(&worker_group);
   if (myid == server) /*  сервер */
   { do
       {MPI_Recv(&request, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST,world,
                    &status);
        if (request)
        {  for (i = 0; i < CHUNKSIZE; i++)   rands[i] = random();
             MPI_Send (rands, CHUNKSIZE, MPI_INT, status .MPI_SOURCE,
                        REPLY, world);
        }
      }   while ( request>0 );
   }
else                                  /* рабочий процесс */
{  request = 1;   done = in = out = 0;    max = INT_MAX;
    MPI_Send( &request, 1, MPI_INT, server, REQUEST, world );
    MPI_Comm_rank( workers, &workerid );
    iter = 0;
    while (! done)
    {   iter++;   request = 1;
        MPI_Recv( rands, CHUNKSIZE, MPI_INT, server, REPLY, world, &status );
        for (i=0; i < CHUNKSIZE; )
           {   x = (((double) rands [i++])/max) * 2 - 1;
               y = (((double) rands [i++]) /max) * 2 - 1;
               if (x*x + y*y < 1.0)   in++;
               else   out++;
            }
           MPI_Allreduce(&in, &totalin, 1, MPI_INT, MPI_SUM, workers);
           MPI_Allreduce(&out, &totalout, 1, MPI_INT, MPI_SUM, workers);

            Pi = (4.0*totalin)/(totalin + totalout);
```

```
            error = fabs( Pi-3.14159265358979323846264 3);
             done = (error < epsilon || (totalin+totalout) > 1000000);
             request = (done) ? 0 : 1;
            if (myid == 0)
            {  printf( "\rpi = %23.20f", Pi );
               MPI_Send( &request, 1, MPI_INT, server, REQUEST, world );
            }
            else
            {  if (request)
                    MPI_Send(&request, 1, MPI_INT, server, REQUEST, world);    }
        }
    }
    if (myid == 0)
    {   printf( "\npoints: %d\nin: %d, out: %d, <ret> to exit\n",
        totalin+totalout, totalin, totalout );
        get char ();
    }
    MPI_Comm_free(&workers);
    MPI_Finalize();
    }
```

Задание 6.1.

```
#include <stdio.h>
#include "mpi.h"
int main( argc, argv )
int argc;
char **argv;
{  int rank, value, size, false=0, right_nbr, left_nbr;

   MPI_Comm   ring_comm;
   MPI_Status status;
   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   MPI_Cart_create( MPI_COMM_WORLD, 1, &size, &false, 1, &ring_comm );
   MPI_Cart_shift( ring_comm, 0, 1, &left_nbr, &right_nbr );
   MPI_Comm_rank( ring_comm, &rank );
   MPI_Comm_size( ring_comm, &size );

   do
   {  if (rank == 0)
      {    scanf( "%d", &value );
           MPI_Send( &value, 1, MPI_INT, right_nbr, 0, ring_comm );
      }
      else
      {    MPI_Recv( &value, 1, MPI_INT, left_nbr, 0, ring_comm, &status );
          MPI_Send( &value, 1, MPI_INT, right_nbr, 0, ring_comm );
```

```
        }
        printf( "Process %d got %d\n", rank, value );
    } while (value >= 0);
    MPI_Finalize( );
    return 0;
}
```

Задание 6.2.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void Compute( cnt, size, databuf )                    /* подпрограмма вычислений */
int    cnt, size;
double *databuf;
{  int i,j;
   for (j = 0; j < cnt; j++)
   {  for (i = 0; i < size; i++)    databuf[i] = sqrt(sqrt(databuf[i]));    }
}

int main( argc, argv )
int argc;
char **argv;
{  int        rank, size, left_nbr, right_nbr,   false = 0,   true = 1,  i, k, n, m, args[2];
   double      *rbuf, *sbuf, *databuf,  t_comm, t_compute, t_both, t1;
   MPI_Comm    comm;
   MPI_Request r_recv, r_send, r[2];
   MPI_Status  status, statuses[2];
   MPI_Init( &argc, &argv );                    /* получаем  n и m */
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );

   if (rank == 0)
   {    args[0] = 20000;          args[1] = 20000;
        for (i=0; i<argc; i++)
        {    if (!argv[i]) continue;
             if (strcmp( argv[i], "-n" ) == 0)
             {  args[0] = atoi( argv[i+1] );  i++;           }
             else
               if (strcmp( argv[i], "-m" ) == 0)
               { args[1] = atoi( argv[i+1] );  i++;          }
        }
    }
    MPI_Bcast( args, 2, MPI_INT, 0, MPI_COMM_WORLD );
    n = args[0];     m = args[1];
```

```
                        /* создаем новый коммуникатор и получаем соседей */
MPI_Comm_size( MPI_COMM_WORLD, &size );
MPI_Cart_create( MPI_COMM_WORLD, 1, &size, &false, true, &comm );
MPI_Cart_shift( comm, 0, 1, &left_nbr, &right_nbr );
MPI_Comm_size( comm, &size );
MPI_Comm_rank( comm, &rank );
rbuf = (double *) malloc( n * sizeof(double) );
sbuf = (double *) malloc( n * sizeof(double) );
databuf = (double *)malloc( m * sizeof(double) );
if (!rbuf || !sbuf)
{  fprintf( stderr, "Unable to allocate buffers of size %n\n", n );
   MPI_Abort( MPI_COMM_WORLD, 1 );
}
if (!databuf)
{   fprintf( stderr, "Unable to allocate buffers of size %n\n", m );
    MPI_Abort( MPI_COMM_WORLD, 1 );
}
for (k=0; k<m; k++)          databuf[k] = 1000.0;
                                /*  время вычислений*/
Compute( 1, m, databuf );
t1 = MPI_Wtime();
Compute( 1, m, databuf );
t_compute= MPI_Wtime() - t1;
                        /* для сравнения  время Irecv/Isend/Wait без вычислений */
MPI_Barrier( comm );
t1 = MPI_Wtime();
MPI_Irecv( rbuf, n, MPI_DOUBLE, left_nbr, 5, comm, &r[0] );
MPI_Isend( sbuf, n, MPI_DOUBLE, right_nbr, 5, comm, &r[1] );
MPI_Waitall( 2, r, statuses );
t_comm = MPI_Wtime() - t1;
                                            /* совмещаем вычисления и передачи */
MPI_Barrier( comm );
t1 = MPI_Wtime();
r_recv = MPI_REQUEST_NULL;
for (k=0;k<3;k++)
{                                                   /* ожидаем предыдущий  recv */
    MPI_Wait( &r_recv, &status );
    MPI_Irecv( rbuf, n, MPI_DOUBLE, left_nbr, k, comm, &r_recv );
    MPI_Isend( sbuf, n, MPI_DOUBLE, right_nbr, k, comm, &r_send );
    Compute( 1, m, databuf );
    MPI_Wait( &r_send, &status );
}
MPI_Wait( &r_recv, &status );
t_both = MPI_Wtime() - t1;    t_both /= 3.0;
if (rank == 0)
  {  printf(  "For n = %d, m = %d, T_comm = %f, T_compute = %f, sum = %f,
```

```
            T_both = %f\n", n, m, t_comm, t_compute, t_comm + t_compute, t_both );
    }
  MPI_Finalize( );
  return 0;
}
```

Задание 7.1.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

int f(int i,int j,int n)          /*  функция обращения к элементу i,j матрицы */
{  return n*i+j;   }

void main(int argc, char *argv[])
{  int  myid, numprocs, i, j,numsent,x, *buffer, *a, *b, *c, master,sender,anstype, ans;
  double startwtime, endwtime, t_calc,t_send;
  int  cols,rows,row,max_cols,max_rows;
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  master = 0;
  if (myid==master)
    {    printf("max_cols=");        scanf("%d",&max_cols);   /* задаем размерности */
         printf("max_rows=");        scanf("%d",&max_rows);
     }
                              /* пересылаем всем процессам размерности матрицы */
    MPI_Bcast(&max_cols,1,MPI_INT,master,MPI_COMM_WORLD);
    MPI_Bcast(&max_rows,1,MPI_INT,master,MPI_COMM_WORLD);
    rows   = max_rows;      cols   =  max_cols;      t_calc = 0;          t_send = 0;
    if (myid==master)
    {  a=(int*) malloc(max_cols*max_rows*sizeof(int));   /* выделяем память */
       buffer=(int*) malloc(max_cols*sizeof(int));
       b=(int*) malloc(max_cols*sizeof(int));
       c=(int*) malloc(max_rows*sizeof(int));
                                          /* инициализируем матрицу и вектор*/
       for (j=0;j<cols;j++)
       {    b[j]=j;
          for (i=0;i<rows;i++)    a[f(i,j,cols)]=j;
       }
       numsent=0;
       startwtime=MPI_Wtime();
                                     /*  пересылаем вектор всем процессам */
       MPI_Bcast(b,cols,MPI_INT,master,MPI_COMM_WORLD);
```

```
      if (numprocs-1<rows)            x=numprocs-1;
      else x=rows;
      for (i=0;i<x;i++)
      {  for (j=0;j<cols;j++) buffer[j]=a[f(i,j,cols)];    /* посылка строки матрицы a */
           MPI_Send(buffer,cols,MPI_INT,i+1,i,MPI_COMM_WORLD);
           numsent=numsent+1;
      }
      for(i=0;i<rows;i++)
      {       /* получаем результат и выдаем следующую строку матрицы a */
        MPI_Recv(&ans,1,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
                      MPI_COMM_WORLD,&status);
        sender  = status.MPI_SOURCE;      anstype  =status.MPI_TAG;
        c[anstype]=ans;
        if (numsent<rows)
          {  for (j=0;j<cols;j++) buffer[j]=a[f(numsent,j,cols)];
             MPI_Send(buffer,cols,MPI_INT,sender,numsent,MPI_COMM_WORLD);
             numsent=numsent+1;
          }
          else                                          /* признак конца работы */
           MPI_Send(MPI_BOTTOM,0,MPI_INT,sender,999,MPI_COMM_WORLD);
      }
      endwtime=MPI_Wtime();
      t_send=t_send+(endwtime-startwtime);
                                                          /* результаты  */
      printf("\n c=");
      for (i=0;i<rows;i++)  printf(" %d ",c[i]);          printf("\n");
      printf("send & handle time = %f\n",t_send);
  }

  else                                  /* не корневые процессы */
  {                                     /*    выделяем память */
    buffer=(int*) malloc(max_cols*sizeof(int));
    b=(int*) malloc(max_cols*sizeof(int));

                                              /* получаем вектор b*/
    MPI_Bcast(b,cols,MPI_INT,master,MPI_COMM_WORLD);

again_sycle:                            /* получаем строку матрицы a*/
    MPI_Recv(buffer,cols,MPI_INT,master,MPI_ANY_TAG,MPI_COMM_WORLD,
                &status);
    if(status.MPI_TAG==999) goto end_program;
    else
      {   row =status.MPI_TAG;   ans=0;
                                                /* считаем результат   */
        for (i=0;i<cols;i++) ans=ans+buffer[i]*b[i];
                                               /* отправляем результат */
```

```
        MPI_Send(&ans,1,MPI_INT,master,row,MPI_COMM_WORLD);
        goto again_sycle;
       }
   }
 end_program:
 MPI_Finalize();
}
```

Задание 7.2.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

int f(int i,int j,int n)        /*  функция обращения к элементу i,j матрицы */
  {  return n*i+j;   }

void main(int argc, char *argv[])
{   int  myid, numprocs, i, j,numsent,x,row, master,sender,anstype, ans;
    int *buffera, *bufferc, *a, *b, *c,  colsa,rowsa,colsb,rowsb;
    double startwtime, endwtime, t_send;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    master = 0;
    if (myid==master)
    {  printf("cols a =");         scanf("%d",&colsa);          /* задаем размерности a*/
       printf(" rows a =");       scanf("%d",&rowsa);
       printf("cols b =");        scanf("%d",&colsb);          /* задаем размерности b*/
       printf(" rows b =");      scanf("%d",&rowsb);
       if (colsa != rowsb)    MPI_Abort( MPI_COMM_WORLD, 1 );
  }
                              /* пересылаем всем процессам размерности матрицы */
   MPI_Bcast(&colsa,1,MPI_INT,master,MPI_COMM_WORLD);
   MPI_Bcast(&rowsa,1,MPI_INT,master,MPI_COMM_WORLD);
   MPI_Bcast(&colsb,1,MPI_INT,master,MPI_COMM_WORLD);
   rowsb=colsa;
    if (myid==master)
   {  a=(int*) malloc(colsa*rowsa*sizeof(int));          /* выделяем память */
      b=(int*) malloc(rowsb*colsb*sizeof(int));
      c=(int*) malloc(rowsa*colsb*sizeof(int));
      buffera=(int*) malloc(colsa*sizeof(int));
      bufferc=(int*) malloc(colsb*sizeof(int));
                        /* инициализируем матрицы*/
     for (j=0;j<colsa;j++)
```

```
      for (i=0;i<rowsa;i++)    a[f(i,j,colsa)]=j+1;
    for (j=0;j<colsb;j++)
      for (i=0;i<rowsb;i++)    b[f(i,j,colsb)]=j+1;

 numsent=0;
 startwtime=MPI_Wtime();
                                        /*  пересылаем матрицу b всем процессам */
  for (i=0;i<rowsb;i++)
     MPI_Bcast(&b[f(i,0,colsb)],colsb,MPI_INT,master,MPI_COMM_WORLD);
  if (numprocs-1<rowsa)       x=numprocs-1;
  else x=rowsa;
 for (i=0;i<x;i++)
 {   for (j=0;j<colsa;j++) buffera[j]=a[f(i,j,colsa)];       /* посылка строки матрицы a */
        MPI_Send(buffera,colsa,MPI_INT,i+1,i,MPI_COMM_WORLD);
     numsent=numsent+1;
 }
 for(i=0;i<rowsa;i++)
 {                    /* получаем результат и выдаем следующую строку матрицы a */
     MPI_Recv(bufferc,colsb,MPI_INT,MPI_ANY_SOURCE,MPI_ANY_TAG,
                  MPI_COMM_WORLD,&status);
     sender  = status.MPI_SOURCE;
     anstype  =status.MPI_TAG;
     for (j=0;j<colsb;j++) c[f(anstype,j,colsb)]=bufferc[j];
    if (numsent<rowsa)
    {      for (j=0;j<colsa;j++) buffera[j]=a[f(numsent,j,colsa)];
           MPI_Send(buffera,colsa,MPI_INT,sender,numsent,MPI_COMM_WORLD);
          numsent=numsent+1;
     }
     else                                              /* признак конца работы */
         MPI_Send(MPI_BOTTOM,0,MPI_INT,sender,999,MPI_COMM_WORLD);
    }
    endwtime=MPI_Wtime();
    t_send=endwtime-startwtime;
   printf("\n c=");
   for (i=0;i<rowsa;i++)
  {   for (j=0;j<colsb;j++)     printf(" %d ",c[f(i,j,colsb)]);
      printf("\n");
   }
   printf("send & handle time = %f\n",t_send);
}
 else                        /* некорневые процессы */
 {                             /*    выделяем память */
   buffera=(int*) malloc(colsa*sizeof(int));
   bufferc=(int*) malloc(colsb*sizeof(int));
   b=(int*) malloc(colsb*rowsb*sizeof(int));
                                          /* получаем  b*/
```

```
    for (i=0;i<rowsb;i++)
       MPI_Bcast(&b[f(i,0,colsb)],colsb,MPI_INT,master,MPI_COMM_WORLD);
again_sycle:
                                    /* получаем строку матрицы a*/
    MPI_Recv(buffera,colsa,MPI_INT,master,MPI_ANY_TAG,
                MPI_COMM_WORLD,&status);
    if(status.MPI_TAG==999) goto end_program;
    else
    {      row =status.MPI_TAG;
                                                    /* считаем результат   */
           for (i=0;i<colsb;i++)
             {    bufferc[i]=0;
                  for (j=0;j<colsa;j++) bufferc[i]=bufferc[i]+buffera[j]*b[f(j,i,colsb)];
             }
                                                     /* отправляем результат     */
           MPI_Send(bufferc,colsb,MPI_INT,master,row,MPI_COMM_WORLD);
           goto again_sycle;
   }
}
 end_program:
 MPI_Finalize();
}
```

Задание 8.1.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
#include "jacobi.h"                         /* описание Mesh как структуры */
static int do_print = 0;
static int max_it   = 100;
int main( argc, argv )
int argc;
char **argv;
{  int       rank, size, i, j, itcnt, maxm, maxn, lrow, k;
   Mesh      mesh;
   double    diffnorm, gdiffnorm, *swap,t, *xlocalrow, *xnewrow;

   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   Get_command_line( rank, argc, argv, &maxm, &maxn, &do_print, &max_it );
   Setup_mesh( maxm, maxn, &mesh );
   for (k=0; k<2; k++)
   {  itcnt = 0;
      Init_mesh( &mesh );                           /* инициализация */
```

```
      MPI_Barrier( mesh.ring_comm );
       t = MPI_Wtime();
      if (do_print && rank == 0)
          printf( "Starting at time %f\n", t ); fflush(stdout);
      lrow = mesh.lrow;
     do
     {    Exchange( &mesh );                  /* заполнение теневых точек */
          itcnt ++;          diffnorm  = 0.0;
          xnewrow  = mesh.xnew   + 1 * maxm;
          xlocalrow = mesh.xlocal + 1 * maxm;
                                  /*  вычисляем новые значения  (не на границах)  */
          for (i=1; i<=lrow; i++)
          {    for (j=1; j<maxm-1; j++)
                 {    double diff;      /* умножаем на 0.25 вместо деления на  4.0 */
                   xnewrow[j] = (xlocalrow[j+1] +   xlocalrow[j-1] +
                           xlocalrow[maxm + j] +   xlocalrow[-maxm + j]) * 0.25;
                    diff   = xnewrow[j] - xlocalrow[j];
                    diffnorm += diff * diff;
                 }
               xnewrow   += maxm;
               xlocalrow += maxm;             }
          swap = mesh.xlocal; mesh.xlocal = mesh.xnew; mesh.xnew = swap;
                                              /* точность вычислений */
          MPI_Allreduce( &diffnorm, &gdiffnorm, 1, MPI_DOUBLE, MPI_SUM,
                            mesh.ring_comm );
         gdiffnorm = sqrt( gdiffnorm );
         if (do_print && rank == 0)
         {  printf( "At iteration %d, diff is %e\n", itcnt, gdiffnorm );
           fflush( stdout );
          }
     } while (gdiffnorm > 1.0e-2 && itcnt < max_it);
     t = MPI_Wtime() - t;
   }
  if (rank == 0)
   { printf( "%s: %d iterations in %f secs (%f MFlops), m=%d n=%d np=%d\n",
   TESTNAME, itcnt, t, itcnt * (maxm-2.0)*(maxn-2)*(4)*1.0e-6/t, maxm, maxn, size);
   }
   MPI_Comm_free( &mesh.ring_comm );
   MPI_Finalize( );
   return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
```

```
#include "jacobi.h"
          /*  процедура для получение размеров сетки из командной строки */
void Get_command_line( rank, argc, argv, maxm, maxn, do_print, maxit )
int rank, argc, *maxm, *maxn, *do_print, *maxit;
char **argv;
{  int args[4], i;
   if (rank == 0)                          /* получение  maxn из командной строки */
   {    *maxn = DEFAULT_MAXN;
        *maxm = -1;
         for (i=1; i<argc; i++)
         {   if (!argv[i]) continue;
             if (strcmp( argv[i], "-print" ) == 0) *do_print = 1;
             else
                if (strcmp( argv[i], "-n" ) == 0)
                {    *maxn = atoi( argv[i+1] );       i++;        }
                else
                   if (strcmp( argv[i], "-m" ) == 0)
                   { *maxm = atoi( argv[i+1] );         i++;    }
                 else
                      if (strcmp( argv[i], "-maxit" ) == 0)
                      {   *maxit = atoi( argv[i+1] );   i++;   }
         }
         if (*maxm < 0) *maxm = *maxn;
        args[0] = *maxn;
        args[1] = *maxm;   args[2] = *do_print;     args[3] = *maxit;
   }
    MPI_Bcast( args, 4, MPI_INT, 0, MPI_COMM_WORLD );
   *maxn     = args[0];    *maxm     = args[1];
   *do_print = args[2];    *maxit    = args[3];
}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
#include "jacobi.h"
                          /* процедура размещения элементов сетки по процессам */
void Setup_mesh( maxm, maxn, mesh )
int  maxm, maxn;
Mesh *mesh;
{  int false = 0;    int true = 1;
   int lrow, rank, size;
   register double *xlocal, *xnew;
                                          /*   картезианская топология */
  MPI_Comm_size( MPI_COMM_WORLD, &size );
  MPI_Cart_create( MPI_COMM_WORLD,1,&size,&false,true,&mesh->ring_comm );
```

```
                                        /* определяем соседей*/
  MPI_Cart_shift( mesh->ring_comm, 0, 1, &mesh->down_nbr, &mesh->up_nbr );
  MPI_Comm_rank( mesh->ring_comm, &rank );
 MPI_Comm_size( mesh->ring_comm, &size );
 lrow = (maxn - 2) / size;
 if (rank < ((maxn - 2) % size)) lrow++;
 mesh->lrow = lrow;
 mesh->maxm = maxm;
 mesh->maxn = maxn;
                                /* размещение частей сетки по процессам */
 mesh->xlocal = xlocal = (double *)malloc( maxm * ( lrow + 2 ) * sizeof(double) );
 mesh->xnew   = xnew = (double *)malloc( maxm * ( lrow + 2 ) * sizeof(double) );
 if (!mesh->xlocal || !mesh->xnew)
   {  fprintf( stderr, "Unable to allocate local mesh\n" );
      MPI_Abort( MPI_COMM_WORLD, 1 );
   }
}

void Init_mesh( mesh )                  /*  процедура инициализации сетки */
Mesh *mesh;
{
   int i, j, lrow, rank, maxm;
   register double *xlocal, *xnew;
   xlocal = mesh->xlocal;
   xnew   = mesh->xnew;
   lrow   = mesh->lrow;
   maxm   = mesh->maxm;
   MPI_Comm_rank( mesh->ring_comm, &rank );
   for (i=1; i<=lrow; i++)
     for (j=0; j<maxm; j++)
     {    xlocal[i*maxm+j] = rank;          xnew[i*maxm+j] = rank;       }
                                     /* заполнение граничных значений  */
   for (j=0; j<maxm; j++)
   {    xlocal[j]  = -1;             xlocal[(lrow+1)*maxm + j] = rank + 1;
        xnew[j]  = -1; xnew[(lrow+1)*maxm + j]  = rank + 1;
   }
   for (i=1; i<=lrow; i++)
   {   xnew[i*maxm]                = rank;
       xnew[i*maxm+maxm-1] = rank;
   }
}

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mpi.h"
```

```
#include "jacobi.h"

void Exchange( mesh )                /* процедура обменов для заполнения теневых то-
чек */
Mesh *mesh;
{  MPI_Status status;
   double *xlocal     = mesh->xlocal;
   int     maxm       = mesh->maxm;
   int     lrow       = mesh->lrow;
   int     up_nbr     = mesh->up_nbr;
   int     down_nbr   = mesh->down_nbr;
   MPI_Comm ring_comm = mesh->ring_comm;
                                        /* передаем вверх,  получаем снизу */
   MPI_Send( xlocal + maxm * lrow, maxm, MPI_DOUBLE, up_nbr, 0,  ring_comm );
   MPI_Recv( xlocal, maxm, MPI_DOUBLE, down_nbr, 0, ring_comm, &status );
                                       /* передаем вниз,  получаем сверху */
   MPI_Send( xlocal + maxm, maxm, MPI_DOUBLE, down_nbr, 1, ring_comm );
   MPI_Recv( xlocal + maxm * (lrow + 1), maxm, MPI_DOUBLE, up_nbr, 1,
                 ring_comm, &status );
}
```

ИСТОЧНИКИ И НФОРМАЦИИ

1. Argonne National Laboratory (http://www.mcs.anl.gov/mpi).
2. Кластерные установки в России (http://parallel.ru/parallel/russia/russian_clusters.html).
3. Научно-исследовательский вычислительный центр МГУ (http://www.parallel.ru).
4. Gropp W., Lusk E., Skjellum A. Using MPI: Portable Parallel Programming with the Message-Passing Interface. Second edition, published in 1999 by MIT Press, 371 p.
5. Snir M., Otto S., Huss-Lederman S. etc. MPI – The Complete Reference: Vol.1, The MPI Core. Second edition, published in 1998 by MIT Press, 426 p.
6. Gropp W., .Huss-Lederman S., Lumsdaine A. etc. MPI – The Complete Reference: Vol.2, The MPI-2 Extensions. Published in 1998 by MIT Press.
7. Gropp W., Lusk E., Thakur R. Using MPI - 2: advanced features of the message-passing interface. Published in 1999 by MIT Press, 382 p.
8. Peter S. Pacheco. Parallel Programming With MPI. Published by Morgan Kaufmann, San Francisco, California, 1997.
9. Программа Союзного государства СКИФ по разработке кластерных систем, 2001 г. (http://www.botik.ru).
10. Шпаковский Г.И., Серикова Н.В. Пособие по программированию для многопроцессорных систем в среде MPI. Мн., 2001г. (http://www.bsu.by).
11. MPI: Стандарт интерфейса передачи сообщений. Перевод с англ. Шпаковского Г.И., Мн., 2001 г. (http://www.bsu.by).
12. Белорусский государственный университет (http://www.bsu.by).
13. Шпаковский Г.И. Организация параллельных ЭВМ и суперскалярных процессоров: Учеб. пособие. Мн.: Белгосуниверситет, 1996. 284 с.
14. Кузюрин Н.Н., Фрумкин М.А. Параллельные вычисления: теория и алгоритмы // Программирование. 1991. N 2. С. 3–19.
15. Bacon D.F., Grahem S.L., Sharp O.J. Compiler transformations for high performance computing // ASM Computing Surveys. 1994. V. 26. № 4.
16. Корнеев В.В. Параллельные вычислительные системы. М.: “Нолидж”, 1999. 320 с.
17. Буза М.К. Введение в архитектуру компьютеров: Учеб. пособие. Мн.: БГУ, 2000. 253 с.
18. Андреев А.Н. Что такое OpenMP? (http://parallel.ru/tech/tech_dev/openmp.html).
19. User’s Guide for MPE (unix.mcs.anl.gov/mpi/mpich/docs/mpeguide).
20. MPI: A Message – Passing Interface Standart. June 12, 1995 (http://www.mcs.anl.gov/mpi).
21. Ортега Д., Пул У. Введение в численные методы решения дифференциальных уравнений. М., 1986.
22. Математические основы криптологии: Учеб. пособие / Харин Ю.С., Берник В.И., Матвеев Г.В. Мн.: БГУ, 1999. 319 с.
23. Молдовян А.А., Молдовян Н.А., Советов Б.Я. Криптография. СПб.:Из–во “Лань”, 2001. 224с.

24. Мулярчик С.Г. Численные методы: Конспект лекций. Мн., БГУ, 2001. 127с.
25. Самарский А.А., Гулис А.В. Численные методы. М., 1989.
26. Параллельный отладчик TotalView (http://www.etnus.com).
27. Производительность кластеров: http://parallel.ru/cluster/performance.html.
28. Библиотеки BLAS, LAPACK, BLACS, ScaLAPACK: (http://www.netlib.org).
29. Дацюк В.Н., Букатов А.А., Жегуло А.И. Многопроцессорные системы и параллельное программирование. Метод. пособие, части 1,2,3 (http://rsusu1.rnd.runnet.ru/ncube/koi8/method/m1/context.html).
30. PETSc: http://www-fp.mcs.anl.gov/petsc/index.html.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

УКАЗАТЕЛЬ ФУНКЦИЙ