Параллельное программирование с использованием технологии MPI

Тулебаев С.Д.

Пособие предназначено для начального освоения практического курса параллельного программирования с использованием технологии MPI. Курс включает в себя описание основных процедур стандарта MPI-1.1 с примерами их применения, а также практические сведения, которые могут потребоваться при написании реальных программ. Основное описание ведется с использованием вызовов процедур MPI из программ, написанных на языке Си, однако указаны также основные отличия в использовании вызовов аналогичных функций из программ на языке Фортран. Приводятся примеры небольших законченных параллельных программ.

Для студентов, аспирантов и научных работников, чья деятельность связана с параллельными вычислениями.

Оглавление

Введение

Настоящее учебное пособие представляет собой введение в методы параллельного программирования и является практическим руководством по разработке эффективных программ для многопроцессорных ЭВМ. Сегодня наиболее распространенным средством программирования для кластерных систем и компьютеров с распределенной памятью является технология MPI.

В разделе, посвященном описанию интерфейса MPI, рассмотрены функциональные возможности этой коммуникационной библиотеки и подробно описан базовый набор подпрограмм, достаточный для разработки параллельных приложений. Предполагается, что приводимой информации достаточно для начала работы на вычислительных кластерах и создания реальных эффективных параллельных программ. Необходимо отметить, что предлагаемое вниманию читателя методическое пособие не является полным изложением MPI, хотя и содержит достаточное количество фактического материала. Полное и строгое описание среды программирования MPI можно найти в авторском описании разработчиков [1]. Большую подборку материалов по MPI и вообще по параллельному программированию можно найти на сервере лаборатории Параллельных Информационных Технологий НИВЦ МГУ [2].

Вопросы распараллеливания вычислительных алгоритмов рассматриваются на различных примерах параллельных программ, приведенных в последнем разделе. Изучение системы MPI строится на практической основе — предлагается ряд конкретных задач, в ходе решения которых рассматриваются основные методы параллельного программирования. В качестве языка программирования использован язык C, хотя все программы, использующие интерфейс MPI, могут быть легко переписаны на языке Fortran. Вопросы трансляции и выполнения MPI-программ освещены в самом минимальном объеме. Для более подробного рассмотрения этого предмета можно рекомендовать пособие [3].

Глава 1

Параллельная модель программирования

1.1 Нам необходимо больше вычислительной мощности

Вычислительные возможности современных компьютеров увеличиваются стремительными темпами. Совсем недавно скорость работы компьютеров исчислялась тысячами операций в секунду, затем мы перешли к миллионам и даже к миллиардам операций в секунду. Но триллион операций в секунду? Не слишком ли это много? Однако вот простой пример, демонстрирующий, что терафлоп[1] является весьма скромным достижением.

Допустим, что мы хотим предсказать погоду на территории России и прилегающих к ней стран на ближайшие два дня. Также предположим, что в нашем распоряжении имеется математическая модель земной атмосферы от уровня Мирового океана до высоты в 20 километров. Стандартный подход к такому типу задач состоит в том, что интересующая нас область покрывается расчетной сеткой, в узлах которой и производятся вычисления. Пусть мы используем кубическую сетку, причем каждая ячейка представляет собой куб с ребром в 0.1 километров. Оценим площадь рассматриваемой территории в 20 миллионов км^2. Тогда наша расчетная сетка будет

[1] TFlop – 10^{12} float operations per second

состоять из

$$20 \times 10^6 \text{ км}^2 \times 20 \text{ км} \,/\, (0.1 \text{ км})^3 = 4 \times 10^{11} \text{ узлов.}$$

Если для определения атмосферных показателей нам необходимо произвести 100 арифметических операций в каждой расчетной точке, а шаг по времени был выбран величиной в один час, то для предсказания погоды на период двое суток (48 часов) всего потребуется

$$100 \times 4 \times 10^{11} \text{ узлов} \times 48 \text{ часов} \approx 2 \times 10^{15} \text{ вычислений.}$$

Если наш компьютер способен выполнять миллиард (10^9) операций в секунду, то на проведение этого количества вычислений на потребуется

$$2 \times 10^{15} / \, 10^9 = 2 \times 10^6 \text{ секунд} \approx 23 \text{ дня!}$$

Иными словами, прогноз погоды на два дня вперед мы получим только спустя три недели. Все наши расчеты теряют всяческий смысл. Известная шутка "Прогноз погоды на завтра слушайте послезавтра" в таком разрезе получается даже слишком оптимистичной.

Если же мы будем иметь возможность проводить вычисления со скоростью триллион операций в секунду, то весь расчет займет примерно 30 минут. Это уже является приемлемым, но весьма несложно представить небольшую модификацию данной задачи, при которой заявленной вычислительной мощности в один терафлоп будет уже недостаточно. Например, если мы заинтересуемся погодой уже на всем земном шаре. Площадь расчетной области тогда возрастет до $5 \times 10^8 \text{ км}^2$, а время, потраченное на вычисления, увеличится до 13 часов — так что первые 12 предсказаний будут уже бесполезны.

Продолжая таким образом наши рассуждения, можно сказать, что всегда найдется задача, для решения которой в разумные сроки не хватит производительности имеющегося в нашем распоряжении компьютера. Например, численное моделирование взаимодействия на атомном уровне объектов молекулярной биологии, разведка месторождений нефти и газа, расчеты для разработки и производства больших интегральных схем и т.п. К слову, правительство США вкладывает миллионы долларов в исследовательские программы по созданию вычислительных средств для решения так называемых задач "Большого Вызова", имеющих огромное значение для развития фундаментального знания — создание новых лекарственных средств, расшифровка генома человека, имитационное моделирование ядерных взрывов, синтез новых материалов и многие другие.

Необходимость в получении значительных вычислительных мощностей для высокопроизводительных вычислений требует новых решений в архитектуре компьютеров. Основное место среди них занимает принцип *параллельной обработки данных*, воплощающий идею одновременного (параллельного) выполнения нескольких действий.

1.2 Использование параллелизма

Если некоторое устройство выполняет одну операцию за единицу времени, то тысячу операций оно выполнит за тысячу единиц. Если предположить, что есть пять таких же независимых устройств, способных работать одновременно, то ту же тысячу операций система из пяти устройств может выполнить уже не за тысячу, а за двести единиц времени. Аналогично система из N устройств ту же работу выполнит за 1000/N единиц времени. Подобные аналогии можно найти и в жизни: если один солдат вскопает огород за 10 часов, то рота солдат из пятидесяти человек с такими же способностями, работая одновременно, справятся с той же работой за 12 минут — принцип параллельности в действии!

Одним из простейших способов использования параллелизма при программировании является *распараллеливание по данным*. Кратко, суть этого подхода заключается в следующем. Исходные данные задачи распределяются по процессам (ветвям параллельного алгоритма), а алгоритм является одним и тем же для всех процессов. Однако действия этого алгоритма зависят от имеющихся в текущем процессе данных. Распределение действий алгоритма заключается, например, в присвоении разных значений переменным одних и тех же циклов в разных ветвях, либо в исполнении в разных ветвях разного количества итераций одних и тех же циклов и т.п. Сказанное можно проиллюстрировать следующей программной конструкцией:

```
if (MyProc == 0) {/* операции, выполняемые 0-ым процессором */}
...
if (MyProc == K) {/* операции, выполняемые K-ым процессором */}
...
```

При этом предполагается, что каждый процессор каким-то образом может получить уникальный номер, присвоить его переменной `MyProc` и использовать в дальнейшем для получения участка кода для независимого исполнения. Таким образом, в приведенном примере операции в первых фигурных

скобках будут выполнены только процессором с номером `0`, операции во вторых фигурных скобках – процессором с номером `K` и т.д.

Разумеется, исходный алгоритм решения поставленной задачи должен иметь возможность быть разложенным на параллельные блоки. Для того чтобы написать параллельную программу, необходимо выделить в алгоритме те группы операции, которые могут вычисляться одновременно и независимо разными процессорами. Возможность этого определяется отсутствием в программе информационных зависимостей. Две операции программы (а в данном случае под операцией можно понимать как отдельное срабатывание некоторого оператора, так и более крупные куски кода программы) называются *информационно зависимыми*, если результат выполнения одной операции используется в качестве аргумента в другой. Очевидно, что если операция B информационно зависит от операции A (то есть использует какие-то результаты операции A в качестве своих аргументов), то операция B может быть выполнена только по завершении операции A. С другой стороны, если операции A и B не являются информационно зависимыми, то алгоритмом не накладывается никаких ограничений на порядок их выполнения, в частности, они могут быть выполнены одновременно. Таким образом, задача распараллеливания алгоритма обычно сводится к нахождению в нем достаточного количества информационно независимых операций и распределению их между вычислительными устройствами [4, 5].

Итак, параллельная программа представляет собой набор обычных последовательных программ (*подзадач*), которые отрабатываются одновременно. Каждая из этих подзадач выполняется на своем процессоре и имеет доступ к своей локальной памяти. Очевидно, в таком случае требуется механизм, обеспечивающий согласованную работу всех частей параллельной программы. Эту проблему можно решить, реализуя *модель передачи сообщений*.

Для обеспечения согласованной работы подзадачи должны обмениваться между собой различной информацией. Это могут быть данные, обработку которых выполняет программа, или управляющие сигналы. В модели передачи сообщений пересылка данных и управляющих сигналов происходят с помощью сообщений. Самым важным моментом здесь является то, что обмен происходит не через общую или разделяемую память, а через коммуникационную среду, поэтому данная модель ориентирована на вычислительные системы с распределенной памятью. Пересылка сообщений — их отправка и прием реализуются программистом с помощью вызова соответствующих функций из библиотеки передачи сообщений.

1.3 Эффективность распараллеливания

Естественно, что используя параллельную систему с p вычислительными устройствами, пользователь ожидает получить ускорение своей программы в p раз по сравнению с последовательным вариантом. Но действительность практически всегда оказывается далека от идеала.

Предположим, что структура информационных зависимостей программы определена (что в общем случае является весьма непростой задачей), и доля операций, которые нужно выполнять последовательно, равна f, где $0 \leq f \leq 1$ (при этом доля понимается не по статическому числу строк кода, а по времени выполнения последовательной программы). Крайние случаи в значениях f соответствуют полностью параллельным ($f = 0$) и полностью последовательным ($f = 1$) программам. Тогда для того, чтобы оценить, какое ускорение s может быть получено на компьютере из p процессоров при данном значении f, можно воспользоваться *законом Амдала*:

$$s \leq \frac{1}{f + \frac{1-f}{p}}$$

Например, если 9/10 программы исполняется параллельно, а 1/10 по-прежнему последовательно, то ускорения более 10 раз получить в принципе невозможно вне зависимости от качества реализации параллельной части кода и числа используемых процессоров (ясно, что 10 получается только в том случае, когда время исполнения параллельной части равно 0). Отсюда можно сделать вывод, что не любая программа может быть эффективно распараллелена. Для того чтобы это было возможно, необходимо, чтобы доля информационно независимых операции была очень большой. В принципе, это не должно отпугивать от параллельного программирования, потому что, как показывает практика, большинство вычислительных алгоритмов устроено в этом смысле достаточно хорошим образом.

Предположим теперь, что в программе относительно немного последовательных операций. Казалось бы, в данном случае все проблемы удалось разрешить. Но представьте, что доступные вам процессоры разнородны но своей производительности. Значит, будет такой момент, когда кто-то из них еще трудится, а кто-то уже все сделал и бесполезно простаивает в ожидании. Если разброс в производительности процессоров большой, то и эффективность всей системы при равномерной загрузке будет крайне низкой.

Но предположим, что все процессоры одинаковы. Проблемы кончились? Опять нет! Процессоры выполнили свою работу, но результатами чаще всею

надо обмениваться для продолжения вычислений, а на передачу данных уходит время, и в это время процессоры опять простаивают... Кроме указанных, есть и еще большое количество факторов, влияющих на эффективность выполнения параллельных программ, причем все они действуют одновременно, а значит, все в той или иной степени должны учитываться при распараллеливании.

Таким образом, заставить параллельную вычислительную систему работать с максимальной эффективностью на конкретной программе — это задача не из простых, поскольку необходимо тщательное согласование структуры программ и алгоритмов с особенностями архитектуры параллельных вычислительных систем.

Сложность программирования с использованием механизма передачи сообщений долгое время оставалась основным сдерживающим фактором на пути широкого использования многопроцессорных систем с распределенной памятью. Однако в последние годы ситуация значительно улучшилась благодаря появлению достаточно эффективных библиотек подпрограмм для решения широкого круга задач (ScaLAPACK, Aztec, PETSc, FFTW и другие). Такие библиотеки избавляют программистов от рутинной работы по написанию подпрограмм для решения стандартных задач численных методов и позволяют сконцентрироваться на предметной области. Однако использование этих библиотек не избавляет от необходимости ясного понимания принципов параллельного программирования и требует достаточно объемной подготовительной работы.

1.4 Технология MPI

Практическое воплощение модель передачи сообщений нашла в спецификации, которая получила название MPI (*Message Passing Interface* — взаимодействие через передачу сообщений). Эта спецификация была разработана в 1993-1994 годах группой MPI Forum, в состав которой входили представители академических и промышленных кругов. Она стала первым стандартом систем передачи сообщений. В MPI были учтены достижения других проектов по созданию систем передачи сообщений: NX/2, nCUBE, p4, PVM и др. Ее реализации представляют собой библиотеки функций, призванные обеспечить совместную работу параллельных процессов путем организации передачи сообщений между ними. В настоящее время имеется несколько программных пакетов, удовлетворяющих спецификации MPI — LAM, HPVM, MPICH (последний относится к свободно распространяемым

продуктам).

Стандарт MPI фиксирует интерфейс, который должна соблюдать как система программирования MPI на каждой вычислительной системе, так и пользователь при создании своих программ. Современные реализации, чаще всего, соответствуют стандарту MPI версии 1.1. В 1997-1998 годах появился стандарт MPI-2.0, значительно расширивший функциональность предыдущей версии. Однако до сих пор этот вариант MPI не получил широкого распространения. Везде далее, если иного не оговорено, мы будем иметь дело со стандартом 1.1.

Механизм обмена сообщениями, реализованный в виде библиотеки подпрограмм MPI, не зависит от физической привязки того или иного процесса. Этим достигается основная цель MPI — обеспечить переносимость разрабатываемого приложения на другие платформы[2]. Таким образом, использование MPI позволяет создавать параллельные приложения, не зависящие ни от машинной архитектуры (одно и то же приложение может выполняться как на однопроцессорной, так и на многопроцессорной машине), ни от взаимного расположения отдельных процессов (на одном или на разных процессорах), ни от операционной системы. Это, в частности, дает возможность отлаживать параллельные программы на обычном персональном компьютере перед выходом на большие многопроцессорные системы.

MPI позволяет создавать хорошо масштабируемые приложения. Под *масштабируемостью* понимается существенное ускорение работы программы при увеличении количества параллельно выполняемых подзадач. Кроме того, приложения, созданные на основе MPI, могут выполняться на разнородных (гетерогенных) системах — множестве процессоров с различной архитектурой. В этом случае MPI автоматически производит конвертирование данных и подбирает корректный протокол взаимодействия. Другой важной особенностью MPI является слабая зависимость его семантики от языка, на котором пишется приложение. MPI поддерживает работу с языками Си и Фортран. В данном пособии все примеры и описания всех функции будут даны с использованием языка Си. Однако это совершенно не является принципиальным, поскольку основные идеи MPI и правила оформления отдельных конструкций для этих языков во многом схожи. Полная версия интерфейса содержит описание более 120 функций. Наша задача — объяснить идею технологии и помочь освоить необходимые на практике компоненты.

[2] *Платформой* называется сочетание архитектуры компьютера и установленной на нем операционной системы.

Глава 2

Интерфейс MPI

2.1 Структура MPI

Спецификация MPI обеспечивает переносимость программ на уровне исходных кодов и большую функциональность. Поддерживается работа на гетерогенных кластерах и симметричных многопроцессорных системах. В спецификации MPI отсутствуют описания параллельного ввода/вывода и отладки параллельных программ. Эти возможности могут быть включены в состав конкретной реализации MPI в виде дополнительных пакетов и утилит. Совместимость разных реализации MPI не гарантируется.

Стандарт MPI-1 предусматривает обязательное наличие следующих типов операций:

- операции межпроцессного взаимодействия типа “точка-точка”;
- операции коллективного взаимодействия;
- операции над группами процессов;
- операции с областями коммуникации;
- операции с топологией процессов.

При этом не включены в стандарт, однако практически всегда имеются следующие средства:

- средства конструирования программы;

- средства отладки;
- явная поддержка подпроцессов;
- поддержка работы с задачами;
- функции ввода-вывода.

В MPI каждое сообщение имеет свой идентификатор, по которому программа и библиотека отличают их друг от друга. Для совместного выполнения определенных операций процессы внутри приложения объединяются в группы. Каждый процесс может узнать свой номер внутри группы, и, в зависимости от номера выполнять соответствующую часть приложения. MPI всегда идентифицирует процессы в соответствии с их номером в группе (начиная с 0).

Важнейшим понятием MPI является область связи (*communication domain*). При запуске приложения все процессы помещаются в создаваемую для приложения общую область связи. При необходимости они могут создавать новые области связи на базе существующих. Все области связи имеют независимую друг от друга нумерацию процессов (нумерация начинается с 0). Каждая область связи имеет свой идентификатор, называемый коммуникатором. Многие функции MPI имеют среди входных аргументов коммуникатор, который ограничивает сферу их действия той областью связи, к которой он прикреплен. Для одной области связи может существовать несколько коммуникаторов. В этом случае приложение работает с такой областью связи, как с несколькими разными областями. При запуске приложения MPI автоматически создает для него стартовую область связи, объединяющую все процессы приложения и имеющую предопределенный коммуникатор `MPI_COMM_WORLD`.

В стандарте MPI выделяются несколько основных категорий функций: *блокирующие* и *неблокирующие*, *локальные*, *глобальные* и *коллективные*. Блокирующие функции останавливают (блокируют) выполнение процесса до тех пор, пока производимая ими операция не будет выполнена. Неблокирующие функции возвращают управление немедленно, а выполнение операции продолжается в фоновом режиме; за завершением операции надо проследить особо. Неблокирующие функции возвращают квитанции (*requests*), которые погашаются при завершении. До погашения квитанции с переменными и массивами, которые были аргументами неблокирующей функции, ничего делать нельзя. Локальные функции MPI выполняют операции, которые не требуют явного взаимодействия с другими процессами. Глобальные

предполагают обязательное выполнение какой-либо MPI процедуры в другом процессе. Коллективные — это функции, которые вызываются всеми процессами в группе, описываемой определенным коммуникатором.

2.2 Общее устройство MPI-программы

В MPI принята следующая нотация записи: все идентификаторы (имена функций, именованные константы, типы данных и пр.) начинаются с префикса `MPI_` и описаны в заголовочном файле `mpi.h`. Порядок слов в составном идентификаторе выбирается по принципу *от общего к частному*: сначала префикс `MPI_`, потом название категории (`Comm`, `Group`, `Type` и т.д.), потом название операции (`MPI_COMM_SIZE`, `MPI_Group_free`, ...). При этом регистр букв важен в C и не играет роли в Фортране. В C имена констант и неизменяемых пользователем переменных записываются прописными буквами, а в имени функции после префикса `MPI_` первая буква прописная, остальные — строчные.

Все функции (за исключением `MPI_Wtime` и `MPI_Wtick`) имеют тип возвращаемого значения `int` и возвращают код ошибки или `MPI_SUCCESS` в случае успеха. Однако в случае ошибки перед возвратом из вызвавшей ее функции вызывается стандартный обработчик ошибки, который аварийно завершает программу. Поэтому проверять возвращаемое значение не имеет смысла. Стандартный обработчик ошибки можно заменить (с помощью функции `MPI_Errhandler_set`), но стандарт MPI не гарантирует, что программа может продолжать работать после ошибки. Так что это тоже обычно не имеет смысла.

До первого вызова любой MPI-функции необходимо вызвать функцию `MPI_Init`. Ее прототип:

```
int MPI_Init(int *argc, char **argv[]);
```

где `argc, argv` — указатели на число аргументов программы и на вектор аргументов соответственно (это адреса аргументов функции main программы). Многие реализации MPI требуют, чтобы процесс до вызова `MPI_Init` не делал ничего, что могло бы изменить его состояние, например открытие или чтение/запись файлов, включая стандартные ввод и вывод.

После окончания работы с MPI-функциями необходимо вызвать функцию `MPI_Finalize`. Ее прототип:

```
int MPI_Finalize(void);
```

Количество работающих параллельных процессов после вызова этой функции не определено, поэтому после ее вызова лучше всего сразу же закончить работу программы.

Общий вид MPI-программы:

```
#include "mpi.h"
/* Другие описания */
int main (int argc, char *argv[])
{
  /* Описания локальных переменных */

  MPI_Init(&argc, &argv);

  /* Тело программы */

  MPI_Finalize();
  return 0;
}
```

Для компиляции и компоновки MPI-программ используется команда `mpicc`. Это скрипт, который после соответствующей настройки окружения (пути к библиотекам и т.д.) вызывает стандартный компилятор языка C. Необходимо отметить, что MPI не содержит механизмов для начального распределения процессов по вычислительным узлам и для запуска их на исполнение. Этим занимается та система, на которой реализован MPI, предоставляя в распоряжение пользователя команду `mpirun`. По этой команде создается заданное количество *виртуальных компьютеров* (*процессов*), объединенных виртуальными каналами связи со структурой *полный граф* (*каждый с каждым*). Этой группе виртуальных компьютеров присваивается стандартное системное имя `MPI_COMM_WORLD`. После этого пользовательская программа загружается (копируется) в память каждого из созданных виртуальных компьютеров и стартует. Если число виртуальных компьютеров больше числа имеющихся вычислительных узлов (*физических компьютеров*), то в некоторых (или всех) физических компьютерах будет создано несколько виртуальных. Виртуальные компьютеры, расположенные на одном физическом, будут работать в режиме интерпретации их физическим процессором с разделением времени.

Отображение виртуальных компьютеров и структуры их связи на конкретную физическую систему осуществляется системой MPI автоматически, т.е. пользователю не нужно переделывать свою программу для разных

вычислительных систем (с другими компьютерами и другой архитектурой). Заметим, что стандарт MPI-1 не обеспечивает динамическое порождение и удаление процессов, т.е. перераспределение виртуальных компьютеров по физическим (эти функции должны появиться в новом стандарте MPI-2).

2.3 Сообщения

Параллельно работающие MPI-процессы обмениваются между собой информацией и обеспечивают взаимную синхронизацию посредством **сообщений**.

Различают обмен сообщениями:

- **попарный (point-to-point)** — сообщение посылается одним процессом другому, в обмене участвуют только два процесса;

- **коллективный** — сообщение посылается процессом всем процессам из его группы (коммуникатора, см. ниже) и получается всеми процессами из его группы (коммуникатора). Коллективный обмен может быть представлен как последовательность попарных обменов, однако специализированные MPI-функции для коллективных обменов учитывают специфику построения конкретной вычислительной установки и могут выполняться значительно быстрее соответствующей последовательности попарных обменов.

- **синхронный** — процесс-отправитель сообщения переходит в состояние ожидания, пока процесс-получатель не будет готов взять сообщение, а процесс-получатель сообщения переходит в состояние ожидания, пока процесс-отправитель не будет готов послать сообщение;

- **асинхронный** — процесс-отправитель сообщения не ждет готовности процесса-получателя, сообщение копируется подсистемой MPI во внутренний буфер, и отправитель продолжает работу.

В первых реализациях MPI все обмены были синхронными. Для повышения эффективности работы (за счет параллельности работы процесса и пересылки сообщения) последние реализации MPI стараются сделать как можно больше обменов асинхронными. Например, если размер сообщения небольшой, то его обычно буферизуют и устраивают асинхронный обмен. Длительное время все коллективные обмены были синхронными.

Отметим, что поведение некачественной программы может быть разным при использовании синхронного или асинхронного режима. Например, если процесс A посылает сообщение процессу B, который не вызывает функцию получения сообщения, то это приводит к "зависанию" процесса A в синхронном режиме. В асинхронном режиме такая программа будет работать.

Основные составляющие сообщения:

1. **Блок данных сообщения** — представляется типом `void*`.
2. **Информация о данных сообщения**:
   (a) **тип данных** — представляется типом `MPI_Datatype`. Соответствие MPI-типов данных и типов языка C приведено в табл. 2.3.
   (b) **количество данных** — количество единиц данного типа в блоке сообщения. Представляется типом `int`. Причина появления этого поля достаточно очевидна: выгоднее за один раз передать большое сообщение с 10-ю элементами, чем посылать 10 сообщений с одним элементом.
3. **Информация о получателе и отправителе сообщения**:
   (a) **коммуникатор** — идентификатор группы запущенных программой параллельных процессов, которые могут обмениваться между собой сообщениями. Представляется типом `MPI_Comm`. Как получатель, так и отправитель должны принадлежать указанной группе. Процесс может принадлежать одновременно нескольким группам.
   (b) **ранг получателя** — номер процесса-получателя в указанной группе (коммуникаторе). Представляется типом `int`. Это поле отсутствует при коллективных обменах сообщениями.
   (c) **ранг отправителя** — номер процесса-отправителя в указанной группе (коммуникаторе). Представляется типом `int`. Ранг отправителя дает возможность различать сообщения, приходящие от разных процессов. Получатель может указать, что он будет принимать только сообщения от процесса с определенным рангом, или использовать константу `MPI_ANY_SOURCE`, позволяющую принимать сообщения от любого отправителя в группе с указанным идентификатором. Это поле отсутствует при коллективных обменах сообщениями.

Таблица 2.1: Соответствие типов данных MPI и типов языка C

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

4. **Тег сообщения** — задаваемое отправителем число типа `int` (стандарт гарантирует возможность использования чисел от 0 до 32767), которое играет роль идентификатора сообщения и позволяет различать сообщения, приходящие от одного процесса. Получатель может специально указать, что он будет принимать только сообщения, имеющие определенный тег, или же использовать константу `MPI_ANY_TAG`, позволяющую принимать любые сообщения. Это поле отсутствует при коллективных обменах сообщениями, поскольку все процессы обмениваются одинаковыми по структуре данными.

2.4 Коммуникаторы

Напомним, **коммуникатор** — это объект типа `MPI_Comm`, представляющий собой идентификатор группы запущенных программой параллельных процессов, которые могут обмениваться сообщениями. Коммуникатор является аргументом всех функций, осуществляющих обмен сообщениями. Програм-

ма может разделять исходную группу всех запущенных процессов, идентифицируемую коммуникатором MPI_COMM_WORLD, на подгруппы для:

- организации коллективных обменов внутри этих подгрупп,
- изоляции одних обменов, от других (так, часто поступают параллельные библиотечные функции, чтобы их сообщения не пересекались с сообщениями вызвавшего их процесса),
- учета топологии распределенной вычислительной установки (например, часто распределенный кластер можно представить в виде пространственной решетки, составленной из скоростных линий связи, в узлах которой находятся рабочие станции; скорости обмена данными между узлами тут различны и это можно учесть введением соответствующих коммуникаторов).

Получить количество процессов в группе (коммуникаторе) можно с помощью функции MPI_Comm_size. Ее прототип:

```
int MPI_Comm_size(MPI_Comm comm, int *size);
```

где comm — коммуникатор (входной параметр), size — указатель на результат. Например, общее количество запущенных процессов можно получить следующим образом:

```
int numprocs;
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
```

Получить ранг (номер) процесса в группе (коммуникаторе) можно с помощью функции MPI_Comm_rank. Ее прототип:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

где comm — коммуникатор (входной параметр), rank — указатель на результат. Например, номер текущего процесса среди всех запущенных можно получить следующим образом:

```
int my_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

2.5 Ввод-вывод в MPI-программах

Рассмотрим вначале стандартный ввод-вывод (на экран). Здесь ситуация зависит от конкретной реализации MPI:

- Существуют реализации MPI, в которых всем работающим процессам разрешено осуществлять ввод-вывод на экран. В этом случае на экране будет наблюдаться перемешанный вывод от всех процессов, в котором часто трудно разобраться.

- Для предотвращения перемешивания вывода разных процессов некоторые реализации MPI (и таких большинство) разрешают осуществлять ввод-вывод на экран только процессу с номером 0 в группе `MPI_COMM_WORLD`. В этом случае остальные процессы посылают этому процессу запросы на производство своего ввода-вывода на экран в качестве сообщений. Этим обеспечивается последовательный вывод на экран (без перемешивания).

- Существуют реализации MPI, в которых всем работающим процессам запрещено осуществлять ввод-вывод на экран. При выводе на экран создается файл, в котором содержится все выведенное, а при вводе с клавиатуры фиксируется ошибка ввода. Обычно такие реализации работают на распределенных вычислительных установках, где запуск программы на исполнение осуществляется с выделенного компьютера, не входящего в состав узлов, на которых собственно исполняется MPI-программа. В таких реализациях весь ввод-вывод осуществляется через файлы.

На файловый ввод-вывод какие-либо ограничения формально отсутствуют. Однако при одновременной записи данных в файл несколькими процессами может происходить их перемешивание и даже потеря. Одновременное чтение данных из файла несколькими процессами на распределенной вычислительной установке может приводить к блокировке всех процессов, кроме одного, и их последовательному выполнению. Это связано с работой сетевой файловой системы (NFS, network file system). Поэтому для написания переносимой между различными реализациямй MPI-программы следует придерживаться предположения о том, что осуществлять файловый ввод-вывод можно только процессу с номером 0 в группе `MPI_COMM_WORLD`. Остальные процессы получают или передают ему свои данные посредством сообщений. Естественно, это не относится к ситуации,

когда каждый из процессов осуществляет ввод-вывод в свой собственный файл.

2.6 Работа с временем

Для хронометрирования работы программы обычно вычисляется процессорное и/или астрономическое время. Однако для MPI-приложения процессорное время не является показателем скорости работы, поскольку оно не учитывает время передачи сообщений между процессами (которое может быть больше процессорного времени!). С другой стороны, и абсолютное значение астрономического времени может быть различным в разных процессах из-за несинхронности часов на узлах параллельного компьютера. Поэтому MPI предоставляет свои функции работы с временем. Кроме того, такое решение позволяет добиться полной независимости MPI-приложений от платформы разработки.

Узнать астрономическое время можно с помощью функции `MPI_Wtime`. Ее прототип:

```
double MPI_Wtime();
```

Она возвращает для текущего процесса астрономическое время в секундах от некоторого фиксированного момента в прошлом (точки отсчета). Гарантируется, что эта точка не будет изменена в течении жизни процесса.

Точность, с которой производится измерение времени, можно узнать при помощи функции `MPI_Wtick`. Ее прототип:

```
double MPI_Wtick();
```

Она возвращает частоту внутреннего таймера (минимальное значение кванта времени). Например, если счетчик времени увеличивается 100 раз в секунду, то эта функция вернет 10^{-2}.

Почти на всех вычислительных установках, где используется MPI, на каждом процессоре запускается только один процесс (иначе очень тяжело балансировать загруженность всех узлов параллельного компьютера). В такой ситуации астрономическое время почти совпадает с временем работы задачи.

Типичная процедура измерения времени работы программы в MPI выглядит следующим образом:

```
double t;
```

```
. . .
/* Синхронизация всех процессов */
MPI_Barrier(MPI_COMM_WORLD);
/* Время начала */
t = MPI_Wtime();
. . .
/* Программный код, время работы которого надо измерить */
. . .
/* Синхронизация всех процессов */
MPI_Barrier(MPI_COMM_WORLD);
/* Время работы */
t = MPI_Wtime() - t;
. . .
```

2.7 Попарный обмен сообщениями

Послать сообщение можно с помощью функции `MPI_Send`. Ее прототип:

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

где входные параметры:

- `buf` — адрес буфера с данными,
- `count` — число элементов (типа datatype) в буфере,
- `datatype` — тип данных каждого из элементов буфера,
- `dest` — ранг (номер) получателя в коммуникаторе (группе) `comm`,
- `tag` — тег сообщения,
- `comm` — коммуникатор.

Эта функция может перевести текущий процесс в состояние ожидания, пока получатель с номером `dest` в группе `comm` не примет сообщение с тегом tag (если используется синхронный режим обмена).

Получить сообщение можно с помощью функции `MPI_Recv`. Ее прототип:

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

где входные параметры:

- `buf` — адрес буфера с данными, где следует разместить полученное сообщение,
- `count` — максимальное число элементов (типа datatype) в буфере,
- `datatype` — тип данных каждого из элементов буфера,
- `source` — ранг (номер) отправителя в коммуникаторе (группе) `comm` (может быть `MPI_ANY_SOURCE`),
- `tag` — тег сообщения (может быть `MPI_ANY_TAG`),
- `comm` — коммуникатор,

и выходные параметры (результаты):

- `buf` — полученное сообщение,
- `status` — информация о полученном сообщении (см. ниже).

Эта функция переводит текущий процесс в состояние ожидания, пока отправитель с номером `source` (или любым номером, если в качестве ранга используется константа `MPI_ANY_SOURCE`) в группе `comm` не отправит сообщение с тегом `tag` (или любым тегом, если в качестве тега используется константа `MPI_ANY_TAG`).

При попарных обменах сообщениями получатель сообщения может получить через переменную `status` типа `MPI_Status` информацию о полученном сообщении. Структура данных `MPI_Status` включает в себя следующие поля:

- `MPI_SOURCE` — реальный ранг отправителя (может потребоваться, если в качестве ранга отправителя использована константа `MPI_ANY_SOURCE`),
- `MPI_TAG` — реальный тег сообщения (может потребоваться, если в качестве тега сообщения использована константа `MPI_ANY_TAG`),
- `MPI_ERROR` — код ошибки,
- дополнительные служебные поля, зависящие от реализации MPI.

Размер полученного сообщения непосредственно в структуре данных `MPI_Status` не хранится, но может быть получен из нее с помощью функции `MPI_Get_count`. Ее прототип:

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

где входные параметры:

- `status` — информация о полученном сообщении,
- `datatype` — тип данных, в единицах которого требуется получить размер сообщения,

и выходной параметр (результат):

- `count` — количество полученных элементов в единицах типа `datatype` или константа `MPI_UNDEFINED`, если длина данных сообщения не делится нацело на размер типа `datatype`.

Заметим, что при коллективных обменах получатель информацию о сообщении не получает, поскольку процессы обмениваются одинаковыми по структуре данными.

Еще раз отметим, что поведение некачественной программы может быть разным при использовании синхронного или асинхронного режима отправки сообщений. Например, если процесс A последовательно посылает процессу B два сообщения с тегами 0 и 1, а процесс B последовательно вызывает функцию получения сообщения; с тегами 1 и 0, то это приводит к “зависанию” обоих процессов в синхронном режиме. В асинхронном режиме такая программа будет работать.

2.8 Дополнительные функции для попарного обмена сообщениями

Для удобства программиста стандарт MPI предоставляет ряд дополнительных функций для попарного обмена сообщениями. Любая программа может быть написана без их использования, но для многих вычислительных установок они могут значительно увеличить производительность.

Если программе важно, чтобы был использован именно синхронный способ посылки сообщений, то вместо функции `MPI_Send` можно использовать

функцию MPI_Ssend, имеющую те же аргументы и возвращаемое значение (дополнительная буква s в имени — от synchronous).

При попарных обменах между процессами очень часто требуется именно обменяться данными, т.е. послать и получить разные сообщения. При этом программный код для этих процессов получается несимметричным: для одного из них надо вначале использовать MPI_Send, а затем MPI_Recv, для второго — эти же функции, но в обратной последовательности. Если же оба процесса вызывают эти функции в одинаковом порядке, то в случае блокирующего синхронного обмена сообщениями они оба "зависают". Для этой ситуации стандарт MPI предоставляет функцию MPI_Sendrecv, являющуюся своеобразным гибридом MPI_Send и MPI_Recv. Ее прототип:

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

где входные параметры:

- sendbuf — адрес буфера с посылаемыми данными,
- sendcount — число элементов (типа sendtype) в буфере sendbuf,
- sendtype — тип данных каждого из элементов буфера sendbuf,
- dest — ранг (номер) получателя в коммуникаторе (группе) comm,
- sendtag — тег посылаемого сообщения,
- recvbuf — адрес буфера с данными, где следует разместить полученное сообщение,
- recvcount — максимальное число элементов (типа recvtype) в буфере recvbuf,
- recvtype — тип данных каждого из элементов буфера recvbuf,
- source — ранг (номер) отправителя в коммуникаторе (группе) comm (может быть MPI_ANY_SOURCE),
- recvtag — тег получаемого сообщения (может быть MPI_ANY_TAG),
- comm — коммуникатор,

и выходные параметры (результаты):

- `recvbuf` — полученное сообщение,
- `status` — информация о полученном сообщении.

Эта функция может перевести текущий процесс в состояние ожидания, пока получатель с номером `dest` в группе `comm` не примет сообщение с тегом `sendtag` (если используется синхронный режим обмена) или пока отправитель с номером `source` (или любым номером, если в качестве ранга используется константа `MPI_ANY_SOURCE`) в группе `comm` не отправит сообщение с тегом `recvtag` (или любым тегом, если в качестве тега используется константа `MPI_ANY_TAG`). Посылать сообщения для приема этой функцией можно с помощью любой функции для попарных обменов, например, `MPI_Send` или ее самой, принимать сообщения от этой функции можно посредством любой функции для попарных обменов, например, `MPI_Recv` или ее самой.

Стандарт MPI запрещает использовать одинаковые указатели для любых двух аргументов любой MPI-функции, если хотя бы один из них является выходным параметром. Для функции `MPI_Sendrecv` это означает, что нельзя использовать один и тот же массив в качестве входного и выходного буферов. Если программе это требуется, то надо применять функцию `MPI_Sendrecv_replace`. Ее прототип:

```
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);
```

где входные параметры:

- `buf` — адрес буфера с посылаемыми данными (также является выходным параметром),
- `count` — число элементов (типа `datatype`) в буфере,
- `datatype` — тип данных каждого из элементов буфера,
- `dest` — ранг (номер) получателя в коммуникаторе (группе) `comm`,
- `sendtag` — тег посылаемого сообщения,
- `source` — ранг (номер) отправителя в коммуникаторе (группе) `comm` (может быть `MPI_ANY_SOURCE`),

- `recvtag` — тег получаемого сообщения (может быть `MPI_ANY_TAG`),
- `comm` — коммуникатор,

и выходные параметры (результаты):

- `buf` — полученное сообщение (также является входным параметром),
- `status` — информация о полученной сообщений.

Работа этой функции аналогична `MPI_Sendrecv`, за исключением того, что полученные данные замещают посланные.

В некоторых вычислительных установках для обмена сообщениями существует специальный коммуникационный процессор, способный работать параллельно с основным. Следовательно, передача сообщений и вычислительная работа могут идти параллельно, если для проведения дальнейших вычислений данные сообщения не нужны (получателю) и не будут модифицироваться (отправителем). Послать сообщение без блокирования процесса можно с помощью функции `MPI_Isend` (дополнительная буква `I` в имени — от immediate). Ее прототип:

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

где входные параметры:

- `buf` — адрес буфера с данными,
- `count` — число элементов (типа `datatype`) в буфере,
- `datatype` — тип данных каждого из элементов буфера,
- `dest` — ранг (номер) получателя в коммуникаторе (группе) `comm`,
- `tag` — тег сообщения,
- `comm` — коммуникатор,

и выходной параметр (результат);

- `request` — идентификатор запроса на обслуживание сообщения; имеет тип `MPI_Request`, не доступный пользователю (указатель на объект этого типа возвращается процессу и может быть использован в качестве аргумента для других функций).

Принимать сообщения от этой функции можно с помощью любой функции для попарных обменов, например, MPI_Recv.

Получить сообщение без блокирования процесса можно при помощи функции MPI_Irecv (дополнительная буква I в имени — от immediate). Ее прототип:

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

где входные параметры:

- buf — адрес буфера с данными, где следует разместить полученное сообщение,
- count — максимальное число элементов (типа datatype) в буфере,
- datatype — тип данных каждого из элементов буфера,
- source — ранг (номер) отправителя в коммуникаторе (группе) comm (может быть MPI_ANY_SOURCE),
- tag — тег сообщения (может быть MPI_ANY_TAG),
- comm — коммуникатор,

и выходные параметры (результаты):

- buf — полученное сообщение,
- request — идентификатор запроса на обслуживания сообщения.

Посылать сообщения для приема этой функцией можно с помощью любой функции для попарных обменов, например, MPI_Send.

Узнать, доставлено ли сообщение (как для функции MPI_Isend, так и для MPI_Irecv), можно с помощью функции MPI_Test. Ее прототип:

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

где входной параметр:

- request — идентификатор запроса на обслуживание сообщения (также является выходным параметром: если обслуживание завершено, то он становится равным специальному значению MPI_REQUEST_NULL),

и выходные параметры (результаты):

- `flag` — признак окончания обслуживания, значение `flag` становится истинным, если запрос обработан,
- `status` — если запрос обработан, то содержит информацию о доставленном сообщении (используется только при получении сообщения и в этом случае играет роль одноименного параметра функции `MPI_Recv`).

Функция `MPI_Test` не блокирует вызвавший ее процесс, а только устанавливает свои аргументы указанным выше образом. Если требуется проверить доставку одного из нескольких сообщений, то можно использовать функцию `MPI_Testany`. Ее прототип:

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

где входные параметры

- `count` — количество запросов в массиве `array_of_requests`,
- `array_of_requests` — массив идентификаторов запросов на обслуживание сообщений (также является выходным параметром: если обслуживание сообщения с номером `index` в массиве `array_of_requests` завершено, то соответствующий элемент `array_of_requests[index]` становится равным `MPI_REQUEST_NULL`),

и выходные параметры (результаты):

- `flag` — признак окончания обслуживания, значение `flag` становится истинным, если один из запросов обработан,
- `index` — если один из запросов обработан, то содержит номер обработанного сообщения в массиве `array_of_requests`,
- `status` — если один из запросов обработан, то содержит информацию о доставленном сообщении (используется только при получении сообщения и в этом случае играет роль одноименного параметра функции `MPI_Recv`).

Дождаться доставки сообщения (как для функции `MPI_Isend`, так и для `MPI_Irecv`) можно с помощью функции `MPI_Wait`. Ее прототип:

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

где входной параметр

- `request` — идентификатор запроса на обслуживание сообщения (также является выходным параметром: после завершения этой функции он становится равным `MPI_REQUEST_NULL`),

и выходной параметр (результат):

- `status` — информация о доставленном сообщении (используется только при получении сообщения и в этом случае играет роль одноименного параметра функции `MPI_Recv`).

Функция `MPI_Wait` блокирует вызвавший ее процесс до окончания обслуживания сообщения с идентификатором `request`. Если требуется дождаться доставки одного из нескольких сообщений, то можно использовать функцию `MPI_Waitany`. Ее прототип:

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

где входные параметры:

- `count` — количество запросов в массиве `array_of_requests`,

- `array_of_requests` — массив идентификаторов запросов на обслуживание сообщений (также является выходным параметром: после завершения этой функции соответствующий элемент массива запросов `array_of_requests[index]` становится равным `MPI_REQUEST_NULL`),

и выходные параметры (результаты):

- `index` — содержит номер обработанного сообщения в массиве запросов `array_of_requests`,

- `status` — информация о доставленном сообщении (используется только при получении сообщения и в этом случае играет роль одноименного параметра функции `MPI_Recv`).

Функция `MPI_Waitany` блокирует вызвавший ее процесс до окончания обработки по крайней мере одного из запросов на обслуживание в массиве `array_of_requests`.

2.9 Коллективный обмен сообщениями

Все описываемые ниже возможности MPI являются избыточными в том смысле, что любая программа может быть написана без их использования. Действительно, всякий коллективный обмен сообщениями может быть заменен на соответствующий цикл попарных обменов. Однако это приведет к значительному снижению скорости работы программы, так как, во-первых, простейшее решение (один из процессов рассылает данные всем остальным) не годится (в каждый момент времени работают только два процесса), а, во-вторых, любое решение (например, рассылка по принципу бинарного дерева: первый процесс посылает данные второму, затем первый и второй посылают данные третьему и четвертому соответственно, и т.д.) не будет учитывать специфику конкретной вычислительной установки.

Сообщения, посылаемые с помощью функций коллективного обмена, не могут быть получены с помощью функций попарного обмена и наоборот. Напомним также, что в первых версиях MPI все функции коллективного обмена были синхронными и это наложило отпечаток на их синтаксис.

Синхронизацию процессов можно осуществить при помощи функции `MPI_Barrier`. Ее прототип:

```
int MPI_Barrier(MPI_Comm comm);
```

где входной параметр

- `comm` — коммуникатор.

Функция `MPI_Barrier` блокирует вызвавший ее процесс до тех пор, пока все процессы в группе с идентификатором `comm` (коммуникаторе) не вызовут эту функцию.

Для рассылки данных из одного процесса всем остальным в его группе можно использовать функцию `MPI_Bcast`. Ее прототип:

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

где входные параметры:

- `buf`
  — в процессе с номером `root` — адрес буфера с посылаемыми данными (входной параметр),
  — в остальных процессах группы `comm` — адрес буфера с данными, где следует разместить полученное сообщение (выходной параметр),

- `count`
  — в процессе с номером `root` — число элементов (типа `datatype`) в буфере,
  — в остальных процессах группы `comm` — максимальное число элементов (типа `datatype`) в буфере,

- `datatype` — тип данных каждого из элементов буфера,

- `root` — ранг (номер) отправителя в коммуникаторе (группе) `comm`;

- `comm` — коммуникатор,

и выходной параметр (результат):

- `buf` — в процессах группы `comm` с номерами, отличными от `root`, — полученное сообщение.

Эта функция должна быть вызвана **во всех** процессах группы с `comm` с **одинаковыми** значениями для аргументов `root` и `comm`. Также в большинстве реализации MPI требуется, чтобы значения аргументов `count` и `datatype` были одинаковыми во всех процессах группы `comm`. Эта функция рассылает сообщение `buf` из процесса с номером `root` всем процессам группы `comm`. Поскольку все процессы вызывают эту функцию одновременно и с одинаковыми аргументами, то, в отличие от функций `MPI_Send` и `MPI_Recv`, поля `tag` и `status` отсутствуют.

Часто требуется выполнить в каком-то смысле обратную к производимой функцией `MPI_Bcast` операцию — переслать данные одному из процессов группы от всех остальных процессов этой группы; при этом часто нужно получить не сами эти данные, а некоторую функцию от них, например, сумму всех данных. Для этого можно использовать функцию `MPI_Reduce`. Ее прототип:

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

где входные параметры:

- `sendbuf` — адрес буфера с данными,

- `count` — число элементов (типа `datatype`) в буфере,

- `datatype` — тип данных каждого из элементов буфера,

Таблица 2.2: Операции над данными в MPI

Операция MPI	Значение
MPI_MAX	максимум
MPI_MIN	минимум
MPI_SUM	сумма
MPI_PROD	произведение
MPI_LAND	логическое "и"
MPI_BAND	побитовое "и"
MPI_LOR	логическое "или"
MPI_BOR	побитовое "или"
MPI_LXOR	логическое "исключающее или"
MPI_BXOR	побитовое "исключающее или"
MPI_MAXLOC	максимум и его позиция
MPI_MINLOC	минимум и его позиция

- `op` — идентификатор операции (типа `MPI_Op`), которую нужно осуществить над пересланными данными для получения результата в буфере `recvbuf`; см. в табл. 2.9 возможные значения этого аргумента,
- `root` — ранг (номер) получателя в коммуникаторе (группе) `comm`,
- `comm` — коммуникатор,

и выходной параметр (результат):

- `recvbuf` — указатель на буфер, где требуется получить результат; используется только в процессе с номером `root` в группе `comm`.

Эта функция должна быть вызвана **во всех** процессах группы `comm` с **одинаковыми** значениями для аргументов `root`, `comm`, `count`, `datatype`, `op`.

Если нам требуется, чтобы результат, полученный посредством функции `MPI_Reduce`, стал известен не только одному процессу (с номером `root` в группе `comm`), а всем процессам группы, то можно использовать `MPI_Bcast`, хотя более эффективным решением является функция `MPI_Allreduce`. Ее прототип:

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

где входные параметры:

- `sendbuf` — адрес буфера с данными,
- `count` — число элементов (типа `datatype`) в буфере,
- `datatype` — тип данных каждого из элементов буфера,
- `op` — идентификатор операций (типа `MPI_Op`), которую нужно осуществить над пересланными данными для получения результата в буфере `recvbuf`; см. в табл. 2.9 возможные значения этого аргумента,
- `comm` — коммуникатор,

и выходной параметр (результат):

- `recvbuf` — указатель на буфер, где требуется получить результат.

Функция, аналогична `MPI_Reduce`, но результат образуется в буфере `recvbuf` во всех процессах группы `comm`, поэтому нет аргумента `root`.

Важным случаем коллективных обменов является запрос на завершение всех процессов, например, в случае ошибки в одном из них. Для этого можно использовать функцию `MPI_Abort`. Ее прототип:

```
int MPI_Abort(MPI_Comm comm, int errorcode);
```

где входные параметры:

- `comm` — коммуникатор, описывающий группу задач, которую надо завершить,
- `errorcode` — код завершения (аналог аргумента функции `exit`).

Функция завершает все процессы в группе `comm`. В большинстве реализаций завершаются **все** запущенные процессы.

2.10 Дополнительные функции коллективного обмена сообщениями

Рассмотрим ситуацию, когда распределенные по процессам данные необходимо собрать в один массив некоторого процесса (рис. 2.1). Конечно же, можно переслать блоки данных от других процессов с помощью функций попарного обмена сообщениями, но удобнее и быстрее это сделать посредством функции `MPI_Gather`. Ее прототип:

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

где входные параметры:

- `sendbuf` — адрес буфера с посылаемыми данными,
- `sendcount` — число элементов (типа `sendtype`) в буфере `sendbuf`,
- `sendtype` — тип данных каждого из элементов буфера `sendbuf`,
- `recvbuf` — адрес буфера с данными, где следует разместить полученное сообщение (используется только в процессе с номером `root` группы `comm`),
- `recvcount` — максимальное число элементов (типа `recvtype`) в буфере `recvbuf` (используется только в процессе с номером `root` группы `comm`),
- `recvtype` — тип данных каждого из элементов буфера `recvbuf` (используется только в процессе с номером `root` группы `comm`),
- `root` — ранг (номер) получателя в коммуникаторе (группе) `comm`,
- `comm` — коммуникатор,

и выходной параметр (результат) в процессе с номером `root` группы `comm`:

- `recvbuf` — указатель на полученные данные.

Эта функция:

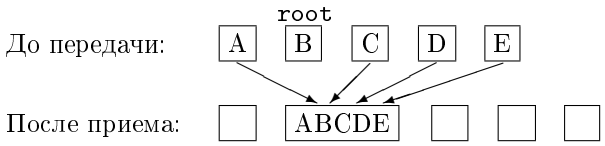

Рис. 2.1: Схема операции сбора данных (`MPI_Gather`)

- в процессе с номером `root` группы `comm` — принимает `recvcount` данных типа `recvtype` от всех остальных процессов группы `comm` и размещает их последовательно в буфере `recvbuf`: вначале данные от процесса с номером 0 группы `comm`, затем данные от процесса с номером 1 группы `comm` и т.д.; от самого себя процесс данные, конечно, не принимает, а просто копирует их из `sendbuf` в соответствующее место `recvbuf`;

- в процессе группы `comm` с номером, отличным от `root`, — посылает `sendcount` данных типа `sendtype` из буфера `sendbuf` процессу с номером `root`.

Эта функция должна быть вызвана **во всех** процессах группы `comm` с одинаковыми значениями для аргументов `root` и `comm`. Также в большинстве случаев требуется, чтобы значения аргументов `sendcount` и `sendtype` были одинаковыми во всех процессах группы `comm`, причем равными значениям `recvcount` и `recvtype` соответственно.

Если нам требуется, чтобы результат, полученный посредством функции `MPI_Gather`, стал известен не только одному процессу (с номером `root` в группе `comm`), а всем процессам группы, то можно использовать `MPI_Bcast`, но более эффективным решением является функция `MPI_Allgather` (ср. ситуацию с функциями `MPI_Reduce` и `MPI_Allreduce`). Ее прототип:

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

где входные параметры:

- `sendbuf` — адрес буфера с посылаемыми данными,

- `sendcount` — число элементов (типа `sendtype`) в буфере `sendbuf`,

- `sendtype` — тип данных каждого из элементов буфера `sendbuf`,
- `recvbuf` — адрес буфера с данными, где следует разместить полученное сообщение,
- `recvcount` — максимальное число элементов (типа `recvtype`) в буфере `recvbuf`,
- `recvtype` — тип данных каждого из элементов буфера `recvbuf`,
- `comm` — коммуникатор,

и выходной параметр (результат):

- `recvbuf` — указатель на полученные данные.

Функция аналогична `MPI_Gather`, но результат образуется в буфере `recvbuf` во всех процессах группы `comm`, поэтому нет аргумента `root`.

MPI предоставляет функцию `MPI_Scatter`, в некотором смысле обратную к `MPI_Gather` (рис. 2.2). Ее прототип:

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

где входные параметры:

- `sendbuf` — адрес буфера с посылаемыми данными (используется только в процессе с номером `root` группы `comm`),
- `sendcount` — число элементов (типа `sendtype`) в буфере `sendbuf` (используется только в процессе с номером `root` группы `comm`),
- `sendtype` — тип данных каждого из элементов буфера `sendbuf` (используется только в процессе с номером `root` группы `comm`),
- `recvbuf` — адрес буфера с данными, где следует разместить полученное сообщение,
- `recvcount` — максимальное число элементов (типа `recvtype`) в буфере `recvbuf`,
- `recvtype` — тип данных каждого из элементов буфера `recvbuf`,
- `root` — ранг (номер) отправителя в коммуникаторе (группе) `comm`,

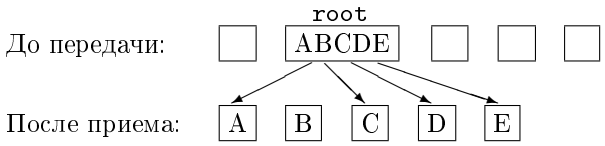

Рис. 2.2: Схема операции распределения данных (`MPI_Scatter`)

- `comm` — коммуникатор,

и выходной параметр (результат):

- `recvbuf` — указатель на полученные данные.

Эта функция:

- в процессе с номером `root` группы `comm` — посылает первые `sendcount` элементов типа `sendtype` в буфере `sendbuf` процессу с номером 0 группы `comm`, следующие `sendcount` элементов типа `sendtype` в буфере `sendbuf` — процессу с номером 1 группы `comm` и т.д.; сам себе процесс данные, конечно, не посылает, а просто копирует их из соответствующего места `sendbuf` в `recvbuf`;

- в процессе группы `comm` с номером, отличным от `root`, — принимает `recvcount` данных типа `recvtype` от процесса с номером `root` и складывает в буфер `recvbuf`.

Эта функция должна быть вызвана **во всех** процессах группы `comm` с **одинаковыми** значениями для аргументов `root` и `comm`. Также в большинстве случаев требуется, чтобы значения аргументов `recvcount` и `recvtype` были одинаковыми во всех процессах группы `comm`, причем равными значениям `sendcount` и `sendtype` соответственно.

Подпрограммы `MPI_Scatterv` и `MPI_Gatherv` являются расширенными (“векторными”) версиями подпрограмм `MPI_Scatter` и `MPI_Gather`. Они позволяют пересылать разным процессам (или собирать от них) разное количество элементов данных. Благодаря аргументу `displacements` (см. ниже), который задает расстояние между элементами, пересылаемые элементы могут не располагаться непрерывно в памяти главного процесса. Это может оказаться полезным, например, при пересылке частей массивов.

Векторная подпрограмма распределения — `MPI_Scatterv`. Ее прототип:

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

где входные параметры:

- `sendbuf` — адрес буфера передачи,
- `sendcounts` — целочисленный одномерный массив, содержащий количество элементов, передаваемых каждому процессу (индекс равен рангу адресата); его длина равна количеству процессов в группе,
- `displacements` — целочисленный массив, длина которого равна количеству процессов в группе; элемент с индексом `i` задает смещение относительно начала буфера передачи; ранг адресата равен значению индекса,
- `sendtype` — тип данных в буфере передачи,
- `recvcount` — количество элементов в буфере приема,
- `recvtype` — тип данных в буфера приема,
- `root` — ранг передающего процесса,
- `comm` — коммуникатор.

Выходным параметром является адрес буфера приема `recvbuf`.

Подпрограмма `MPI_Gatherv` используется для сбора данных от всех процессов в заданном коммуникаторе и записи их в буфер приема с указанным смещением:

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

Список праметров у этой подпрограммы весьма похож на список параметров подпрограммы `MPI_Scatterv`. Поэтому читателю не составит труда самостоятельно разобраться с их назначением.

Глава 3

Примеры MPI-программ

3.1 Пример простейшей MPI-программы

Рассмотрим в качестве примера программу, выводящую на экран сообщение "Hello"от каждого из процессов.

```
/* MPI_simple.c */
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#define BUF_LEN 256 /* длина буфера сообщений */
#define MSG_TAG 100 /* тег сообщений */

int main(int argc, char *argv[])
{
  int my_rank;          /* ранг текущего процесса */
  int numprocs;         /* общее число процессов */
  int source;           /* ранг отправителя */
  int dest;             /* ранг получателя */
  char message[BUF_LEN];/* буфер для сообщения */
  MPI_Status status;    /* информация о полученном сообщении */

  /* Начать работу с MPI */
  MPI_Init(&argc, &argv);
```

```
  /* Получить номер текущего процесса в группе всех процессов */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  /* Получить общее количество запущенных процессов */
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  /* Посылаем сообщения процессу 0, который их выводит
     на экран */
  if (my_rank != 0) {
    /* Создаем сообщение */
    sprintf(message, "Hello from process %d!", my_rank);
    /* Отправляем его процессу 0 */
    dest = 0;
    MPI_Send(message, strlen(message) + 1, MPI_CHAR,
             dest, MSG_TAG, MPI_COMM_WORLD);
  }
  else {
    /* В процессе 0: получаем сообщение от процессов
       1,...,numprocs-1 и выводим его на экран */
    for (source = 1; source < numprocs; source++) {
      MPI_Recv(message, BUF_LEN, MPI_CHAR, source, MSG_TAG,
               MPI_COMM_WORLD, &status);
      printf("%s\n", message);
     }
  }
  /* Заканчиваем работу с MPI */
  MPI_Finalize();
  return 0;
}
```

Компиляция этого файла:

```
mpicc MPI_simple.c -o MPI_simple
```

и запуск

```
mpirun -np 2 MPI_simple
```

где опция `-np` задает количество параллельно работающих процессов (в данном случае 2). Попробуйте его изменить, чтобы выяснить, как это отражается на работе программы.

3.2 Вычисление интеграла

Рассмотрим в качестве примера задачу вычисления определенного интеграла от функции $f(x)$ по отрезку $[a, b]$. Для ускорения работы программы на вычислительной установке с p процессорами мы воспользуемся аддитивностью интеграла:

$$\int_a^b f(x)\,dx = \sum_{i=0}^{p-1} \int_{a_i}^{b_i} f(x)\,dx,$$

где $a_i = a + i * l$, $b_i = a_i + l$, $l = (b - a)/p$. Использовав для приближенного определения каждого из слагаемых $\int_{a_i}^{b_i} f(x)\,dx$ этой суммы составную формулу трапеций, и поручив эти вычисления своему процессору, мы получим p-кратное ускорение работы. Простейшая реализация этой задачи приведена ниже.

Каждый из процессов инициализирует подсистему MPI с помощью вызова `MPI_Init`, получает свой номер и общее количество процессов посредством `MPI_Comm_rank` и `MPI_Comm_size`. Основная работа производится в функции `integrate`, по окончании которой процесс заканчивает работу с MPI с помощью `MPI_Finalize` и завершается. Функция `integrate` вычисляет свою часть интеграла и прибавляет его к ответу `total` в процессе с номером 0 посредством `MPI_Reduce`. Процесс с номером 0 перед окончанием своей работы выводит переменную `total` на экран.

Входные данные (границы отрезка и количество точек разбиения интервала) принимаются от пользователя в процессе с номером 0. Затем эти данные рассылаются всем остальным процессам с помощью `MPI_Bcast`.

Текст программы:

```
/* MPI_integral.c */
#include <stdio.h>
#include "mpi.h"

/* Интегрируемая функция */
double f(double x)
{
  return x;
}

/* Вычислить интеграл по отрезку [a,b] с числом точек разбиений n
   по формуле трапеций */
```

```
double integrate(double a, double b, int n)
{
  double res; /* результат */
  double h;   /* шаг интегрирования */
  double x;
  int i;

  h = (b-a)/n;
  res = 0.5*(f(a)+f(b))*h;
  x = a;
  for (i=1; i<n; i++) {
    x += h;
    res += f(x)*h;
  }
  return res;
}

int main(int argc, char *argv[])
{
  int my_rank;     /* ранг текущего процесса */
  int numprocs;    /* общее число процессов */
  double a;        /* левый конец интервала */
  double b;        /* правый конец интервала */
  int n;           /* число точек разбиения */
  double len;      /* длина отрезка интегрирования для текущего
                      процесса */
  double local_a;  /* левый конец интервала для текущего
                      процесса */
  double local_b;  /* правый конец интервала для текущего
                      процесса */
  int local_n;     /* число точек разбиения для текущего
                      процесса */
  double local_res;/* значение интеграла в текущем процессе */
  double result;   /* результат интегрирования */
  double wtime;    /* время работы программы */

  /* Начать работу с MPI */
  MPI_Init(&argc, &argv);
  /* Получить номер текущего процесса в группе всех процессов */
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
/* Получить общее количество запущенных процессов */
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
/* Получить данные */
if (my_rank == 0) {
  printf("Input a: ");
  scanf("%lf", &a);
  printf("Input b: ");
  scanf("%lf", &b);
  printf("Input n: ");
  scanf("%d", &n);
}
/* Рассылаем данные из процесса 0 остальным */
MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
/* Синхронизация процессов */
MPI_Barrier(MPI_COMM_WORLD);
/* Запускаем таймер */
wtime = MPI_Wtime();
/* Вычисляем отрезок интегрирования для текущего процесса */
len = (b-a)/numprocs;
local_n = n/numprocs;
local_a = a + my_rank*len;
local_b = local_a + len;
/* Вычислить интеграл на каждом из процессов */
local_res = integrate(local_a, local_b, local_n);
/* Сложить все ответы и передать процессу 0 */
MPI_Reduce(&local_res, &result, 1, MPI_DOUBLE, MPI_SUM, 0,
           MPI_COMM_WORLD);
/* Синхронизация процессов */
MPI_Barrier(MPI_COMM_WORLD);
/* Вычисляем время работы */
wtime = MPI_Wtime() - wtime;
/* Напечатать ответ */
if (my_rank == 0) {
  printf("Integral from %.2lf to %.2lf = %.8lf\n",
          a, b, result);
  printf("Working time: %.2lf seconds\n", wtime);
```

```
  }
  /* Заканчиваем работу с MPI */
  MPI_Finalize();
  return 0;
}
```

Компиляция этого файла:

```
  mpicc MPI_integral.c -o MPI_integral
```

и запуск

```
  mpirun -np 2 MPI_integral
```

Имеет смысл поэкспериментировать с количеством работающих параллельно процессов, чтобы выяснить, как меняется время работы программы.

3.3 Нахождение минимума

Довольно часто в практических задачах встречается проблема оптимизации – поиска экстремума (минимума или максимума) некоторой функции. Одним из методов решения этой задачи является метод сканирования. Рассмотрим работу этого метода для решения задачи нахождения минимума функции двух переменных $f(x, y)$. Суть метода заключается в переборе всех значений $x_{min} \leq x \leq x_{max}$ с шагом h_x и $y_{min} \leq y \leq y_{max}$ с шагом h_y с вычислением значения функции $f(x, y)$ в каждой точке. Решение задачи находится путем выбора наименьшего из всех вычисленных значений функции. Величины h_x, h_y определяют погрешность решения, т.е. с какой точностью мы локализуем точку минимума.

К недостаткам метода сканирования относится значительное число повторных вычислений функции $f(x, y)$, что требует существенных затрат времени. Однако к задаче оптимизации вполне возможно применить концепцию распараллеливания вычислений, "назначив"каждому процессу для обработки свою часть двумерной области изменения параметров.

Текст программы:

```
/* MPI_minima.c */
#include <stdio.h>
#include <math.h>
#include "mpi.h"
```

```
/* Исследуемая функция */
double f(double x, double y) {
  return (exp(-x) + 3.0*exp(-y) + log(exp(x)+exp(y)));
}

#define MSG_TAG 100 /* тег сообщения */

int main(int argc, char *argv[]) {
  int my_rank;     /* ранг текущего процесса */
  int numprocs;    /* общее число процессов */
  int source;      /* ранг отправителя */
  MPI_Status status;  /* информация о полученном сообщении */
  double minx, maxx,
         miny, maxy;  /* границы области поиска */
  double hx, hy;      /* погрешность поиска */
  double global_minf; /* значение глобального минимума функции */
  double global_posx,
         global_posy; /* положение глобального минимума */
  double start_x,
         end_x, len;  /* интервал поиска для текущего процесса */
  double posx, posy;  /* положение локального минимума */
  double minf; /* значение минимума функции в текущем процессе */
  double wtime;       /* время работы программы */
  double x, y, value; /* рабочие переменные */
  double exchange_array[6]; /* массив для рассылки */

  /* Начать работу с MPI */
  MPI_Init(&argc, &argv);
  /* Получить номер текущего процесса в группе всех процессов */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  /* Получить общее количество запущенных процессов */
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  /* Получить данные */
  if (my_rank == 0) {
    printf("Input the search domain:\n");
    printf(" min x: ");
    scanf("%lf", &exchange_array[0]);
    printf(" max x: ");
```

```
  scanf("%lf", &exchange_array[1]);
  printf(" min y: ");
  scanf("%lf", &exchange_array[2]);
  printf(" max y: ");
  scanf("%lf", &exchange_array[3]);
  printf("Input the step of searching:\n");
  printf(" hx: ");
  scanf("%lf", &exchange_array[4]);
  printf(" hy: ");
  scanf("%lf", &exchange_array[5]);
}
/* Рассылаем данные из процесса 0 остальным */
MPI_Bcast(exchange_array, 6, MPI_DOUBLE, 0, MPI_COMM_WORLD);
minx = exchange_array[0]; maxx = exchange_array[1];
miny = exchange_array[2]; maxy = exchange_array[3];
hx = exchange_array[4]; hy = exchange_array[5];
/* Синхронизация процессов */
MPI_Barrier(MPI_COMM_WORLD);
/* Запускаем таймер */
wtime = MPI_Wtime();
/* Вычисляем интервал поиска для текущего процесса */
len = (maxx-minx)/numprocs;
start_x = minx + len*my_rank;
end_x = start_x + len;
/* Найти минимум функции в своей подобласти */
minf = +1.0e+38;
x = start_x;
while (x <= end_x) {
  y = miny;
  while (y <= maxy) {
    value = f(x,y);
    if (minf > value) {
      minf = value; posx = x; posy = y;
    }
    y += hy;
  }
  x += hx;
}
/* Передать результаты процессу 0 */
```

```
  if (my_rank != 0) {
    exchange_array[0] = minf;
    exchange_array[1] = posx; exchange_array[2] = posy;
    MPI_Send(&exchange_array, 3, MPI_DOUBLE, 0, MSG_TAG,
             MPI_COMM_WORLD);
  }
  /* Синхронизация процессов */
  MPI_Barrier(MPI_COMM_WORLD);
  /* Вычисляем время работы */
  wtime = MPI_Wtime() - wtime;
  if (my_rank == 0) {
    /* Выбрать среди результатов наименьший */
    global_minf = minf; global_posx = posx; global_posy = posy;
    for (source = 1; source < numprocs; source++) {
      MPI_Recv(exchange_array, 3, MPI_DOUBLE, source, MSG_TAG,
               MPI_COMM_WORLD, &status);
      minf = exchange_array[0];
      posx = exchange_array[1]; posy = exchange_array[2];
      if (global_minf > minf) {
        global_minf = minf;
        global_posx = posx; global_posy = posy;
      }
    }
    /* Напечатать ответ */
    printf("f(x,y) has a minimum %.5f at (%.3f,%.3f).\n",
           global_minf, global_posx, global_posy);
    printf("Working time: %.2lf seconds\n", wtime);
  }
  /* Заканчиваем работу с MPI */
  MPI_Finalize();
  return 0;
}
```

Командная строка для компиляции этой программы выглядит следующим образом:

```
mpicc MPI_minima.c -o MPI_minima -lm
```

Ключ `-lm` указывает компоновщику на необходимость подключения математической библиотеки (так как мы используем функции exp и log).

И вновь, в качестве упражнения, мы предлагаем запустить программу на исполнение с различным количеством параллельно работающих процессов, чтобы выяснить, насколько хорошо масштабируется данная задача.

3.4 Параллельная сортировка

Сортировка является одной из типовых проблем обработки данных, и обычно понимается как задача размещения элементов неупорядоченного набора значений в порядке монотонного возрастания или убывания. Вычислительная трудоемкость процедуры упорядочивания является достаточно высокой, однако ускорить сортировку, если использовать несколько процессоров. В этом случае исходный упорядочиваемый набор разделяется между процессорами, которые производят локальную сортировку своей части данных при помощи какого-либо быстрого алгоритма. Затем производится объединение уже упорядоченных фрагментов, используя схему так называемого P-путевого слияния:
1. Считать начальные элементы оставшихся непустых последовательностей.
2. Записать минимальный элемент в выходной массив.
3. Удалить записанный элемент из соответствующей последовательности.
4. Если имеются непустые последовательности, то перейти к пункту 1.

Дополнительно, в программе показано, как следует отслеживать ошибки динамического выделения памяти. В случае нехватки памяти на каком-либо вычислительном узле, необходимо корректно завершить **все** процессы.

Текст программы:

```
/* MPI_sorting.c */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "mpi.h"

#define NUM_ITEMS 100
#define ERROR_CODE_TAG 300
#define NO_MEMORY_MSG "Not enough memory in %i process!\n"

#define randomize() srand((unsigned)time(NULL))
#define get_random(num) (int)((double)rand()/RAND_MAX*num)

/* Функция сравнения элементов */
```

```
int compare(const void *a, const void *b) {
  if (*(int*)a < *(int*)b) return -1;
  else return +1;
}

int main(int argc, char* argv[]) {
  int my_rank;    /* ранг текущего процесса */
  int numprocs;   /* общее число процессов */
  double wtime;   /* время работы программы */
  int *array = NULL; /* указатель на исходный массив с данными */
  int *array_sorted = NULL;  /* указатель на отсортированные
                                данные */
  int *counts = NULL;        /* размеры фрагментов массива */
  int *displacements = NULL; /* смещения фрагментов массива */
  int *array_local = NULL;   /* локальный массив */
  int error_flag = 0;        /* общий флаг наличия ошибки */
  int local_flag = 0;        /* локальный флаг наличия ошибки */
  MPI_Status status;
  int i, p, n_div, index;

  /* Начать работу с MPI */
  MPI_Init(&argc, &argv);
  /* Получить номер текущего процесса в группе всех процессов */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  /* Получить общее количество запущенных процессов */
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  /* Запрашиваем память в корневом процессе */
  if (my_rank == 0) {
    if (!(array = (int*)malloc((NUM_ITEMS+1)*sizeof(int))) ||
        !(array_sorted = (int*)malloc(NUM_ITEMS*sizeof(int))))
    { /* Ошибка -- недостаточно памяти */
      fprintf(stderr, NO_MEMORY_MSG, 0);
      error_flag = 1;
    }
  }
  MPI_Bcast(&error_flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
  /* Если произошла ошибка в корневом процессе,
     то досрочно завершаем _все_ процессы */
  if (error_flag) goto exit_label;
```

```
/* Задаем размеры и смещения фрагментов массива */
counts = (int*)malloc(numprocs*sizeof(int));
displacements = (int*)malloc(numprocs*sizeof(int));
n_div = NUM_ITEMS/numprocs;
for (p = 0; p < numprocs; p++) {
  counts[p] = n_div;
  displacements[p] = p*n_div;
}
/* Корректировка размера последнего фрагмента в случае,
   когда число элементов массива не кратно числу процессов */
counts[numprocs-1] += NUM_ITEMS%numprocs;
/* Запрашиваем память под локальные массивы */
if (!(array_local=(int*)malloc((counts[my_rank])*sizeof(int))))
  local_flag = 1;
/* Сообщаем о результате выделения памяти корневому процессу */
MPI_Send(&local_flag, 1, MPI_INT, 0, ERROR_CODE_TAG,
         MPI_COMM_WORLD);
/* Проверка выделения памяти под локальные массивы */
if (my_rank == 0) {
  for (p = 0; p < numprocs; p++) {
    MPI_Recv(&local_flag, 1, MPI_INT, p, ERROR_CODE_TAG,
             MPI_COMM_WORLD, &status);
    if (local_flag) {
      fprintf(stderr, NO_MEMORY_MSG, status.MPI_SOURCE);
      error_flag = 1;
    }
  }
}
MPI_Bcast(&error_flag, 1, MPI_INT, 0, MPI_COMM_WORLD);
/* Была ошибка в каком-либо процессе */
if (error_flag) goto exit_label;
/* Подготовительные операции завершены -- начинаем работу */
if (my_rank == 0) {
  /* Инициализируем исходный массив случайными числами */
  randomize();
  for (i = 0; i < NUM_ITEMS; i++) array[i] = get_random(1000);
}
/* Запускаем таймер */
MPI_Barrier(MPI_COMM_WORLD);
```

```
  wtime = MPI_Wtime();
  /* Поизводим разделение исходного массива и его рассылку
     по процессам */
  MPI_Scatterv(array, counts, displacements, MPI_INT,
               array_local, counts[my_rank], MPI_INT,
               0, MPI_COMM_WORLD);
  /* Производим локальную сортировку */
  qsort(array_local, counts[my_rank], sizeof(int), compare);
  /* Производим сбор данных от всех процессоров */
  MPI_Gatherv(array_local, counts[my_rank], MPI_INT, array,
              counts, displacements, MPI_INT, 0, MPI_COMM_WORLD);
  /* Производим слияние упорядоченных фрагментов */
  if (my_rank == 0) {
    /* ставим как ограничитель число, заведомо большее любого
       из массива данных */
    array[NUM_ITEMS] = INT_MAX;
    for (i = 0; i < NUM_ITEMS; i++) {
      index = 0;
      for (p = 1; p < numprocs; p++) {
       if (array[displacements[p]] < array[displacements[index]])
         index = p;
      }
      array_sorted[i] = array[displacements[index]];
      displacements[index]++;
      counts[index]--;
      if (counts[index] == 0) displacements[index] = NUM_ITEMS;
    }
  }
  /* Вычисляем время работы */
  MPI_Barrier(MPI_COMM_WORLD);
  wtime = MPI_Wtime() - wtime;
  /* Выводим отсортированный массив */
  if (my_rank == 0) {
    for (i = 0; i < NUM_ITEMS; i++)
      printf("%3i ", array_sorted[i]);
    printf("\n");
    printf("Working time: %.2lf seconds\n", wtime);
   }
exit_label:
```

```
  /* Освобождаем память и заканчиваем работу MPI */
  free(counts);
  free(displacements);
  free(array_local);
  free(array);
  free(array_sorted);
  MPI_Finalize();
  preturn(error_flag);
}
```

3.5 Решение систем линейных уравнений

Рассмотрим систему n линейных уравнений с неизвестными $x_1, x_2, \ldots, x_n$:

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + & \ldots & + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + & \ldots & + a_{2n}x_n = b_2, \\ & \ldots & \\ a_{n1}x_1 + a_{n2}x_2 + & \ldots & + a_{nn}x_n = b_n, \end{array} \tag{3.1}$$

где коэффициенты a_{ij} $(i, j = 1, \ldots, n)$ – действительные числа. Запишем систему (3.1) в матричной форме:

$$Ax = b, \tag{3.2}$$

где A – матрица размерности $n \times n$, $x = (x_1, \ldots, x_n)^T$ – искомый вектор и $b = (b_1, \ldots, b_n)^T$ – заданный вектор.

Решение систем линейных уравнений является одной из важных вычислительных задач, часто встречающихся в прикладной математике. К решению систем линейных уравнений сводится ряд задач анализа, связанных с приближением функций, решением дифференциальных уравнений и т.д. Весьма распространенными методами решения систем линейных уравнений являются *итерационные методы*, которые состоят в том, что решение x системы (3.1) находится как предел при $k \to \infty$ последовательных приближений x^k, где $k = 0, 1, \ldots$ – номер итерации.

Мы рассмотрим простейший из таких алгоритмов – *метод Якоби*. Согласно ему, последовательные приближения, начиная с некоторого началь-

ного x^0, вычисляются с использованием следующих формул:

$$x_i^{k+1} = \frac{1}{a_{ii}} \left(- \sum_{j \neq i} a_{ij} x_j^k + b_i \right), \quad i = 1, \ldots, n \tag{3.3}$$

Метод Якоби не является приемлемым для большинства задач, поскольку он сходится не всегда, а только в случае диагонального преобладания матрицы A. Однако для наших учебных целей этот метод представляет собой удобную отправную точку в изучении способов распараллеливания итерационных алгоритмов.

Для решения задачи (3.1) на параллельной системе исходную матрицу коэффициентов A "разрезают"на p горизонтальных полос по строкам, где p – количество компьютеров в системе. Аналогично, горизонтальными полосами, разделяются вектор правых частей b и вектор текущего приближения x^k. Полосы распределяются по соответствующим компьютерам и реализуется параллельный алгоритм умножения матрицы на вектор.

Качество полученного приближения x^k характеризует норма вектора $r^k = x^k - x^{k-1}$ – так называемая *невязка*. Обычно вычисления в процессе (3.3) продолжают до тех пор, пока норма невязки не станет достаточно малой:

$$||x^k - x^{k-1}|| \leq \varepsilon, \tag{3.4}$$

где $\varepsilon > 0$ есть некоторое наперед заданное малое число.

Если условие (3.4) не выполняется в течении максимального числа итераций, то имеет место расходимость процесса и алгоритм завершает работу с соответствующим диагностическим сообщением.

Текст программы:

```
/* MPI_Jacobi.c */
#include <stdio.h>
#include <math.h>
#include "mpi.h"

/* Максимальная размерность */
#define MAX_DIM 12
typedef float MATRIX_T[MAX_DIM][MAX_DIM];

#define Swap(x,y) { float* temp; temp = x; x = y; y = temp; }
/* Прототипы функций */
```

```
float Distance(float*, float*, int);
int Jacobi(MATRIX_T, float*, float*, int, float, int, int, int);
void Read_matrix(char*, MATRIX_T, int, int, int);
void Read_vector(char*, float*, int, int, int);
void Print_matrix(char*, MATRIX_T, int, int, int);
void Print_vector(char*, float*, int, int, int);

int main(int argc, char* argv[])
{
  int numprocs, my_rank, n, max_iter;
  float tol;
  MATRIX_T A_local;
  float x_local[MAX_DIM];
  float b_local[MAX_DIM];
  int converged;

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

  if (my_rank == 0) {
    printf("Enter order of system, tolerance,\
            and max number of iterations:\n");
    scanf("%d %f %d", &n, &tol, &max_iter);
  }
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&tol, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
  MPI_Bcast(&max_iter, 1, MPI_INT, 0, MPI_COMM_WORLD);
  Read_matrix("Enter the matrix:", A_local, n, my_rank,
              numprocs);
  Read_vector("Enter the right-hand side:", b_local, n,
              my_rank, numprocs);
  converged = Jacobi(A_local, x_local, b_local, n, tol,
                     max_iter, numprocs, my_rank);
  if (converged)
    Print_vector("The solution is:", x_local, n, my_rank,
                 numprocs);
  else
    if (my_rank == 0)
```

```
      printf("Failed to converge in %d iterations.\n", max_iter);
  MPI_Finalize();
  return 0;
}  /* main */

/* Возвращает 1, если процесс решения сходится,
   0 в противном случае */
int Jacobi(MATRIX_T A_local, float x_local[], float b_local[],
           int n, float tol, int max_iter, int numprocs,
           int my_rank)
{
  int n_bar, iter_num;
  int i_local, i_global, j;
  float x_temp1[MAX_DIM];
  float x_temp2[MAX_DIM];
  float* x_old;
  float* x_new;

  n_bar = n/numprocs;
  /* Инициализируем x */
  MPI_Allgather(b_local, n_bar, MPI_FLOAT, x_temp1, n_bar,
                MPI_FLOAT, MPI_COMM_WORLD);
  x_new = x_temp1;
  x_old = x_temp2;
  iter_num = 0;
  do {
    /* Переставляем x_old и x_new */
    Swap(x_old, x_new);
    /* Производим параллельное умножение матрицы на вектор */
    for (i_local = 0; i_local < n_bar; i_local++) {
      i_global = i_local + my_rank*n_bar;
      x_local[i_local] = b_local[i_local];
      for (j = 0; j < i_global; j++)
        x_local[i_local] =
          x_local[i_local] - A_local[i_local][j]*x_old[j];
      for (j = i_global+1; j < n; j++)
        x_local[i_local] =
          x_local[i_local] - A_local[i_local][j]*x_old[j];
      x_local[i_local] =
```

```
        x_local[i_local]/A_local[i_local][i_global];
    }
    /* Рассылаем новое решение */
    MPI_Allgather(x_local, n_bar, MPI_FLOAT, x_new, n_bar,
                  MPI_FLOAT, MPI_COMM_WORLD);
    iter_num++;
    if (iter_num > max_iter) return 0; /* процесс расходится */
  } while (Distance(x_new,x_old,n) >= tol);
  return 1;
} /* Jacobi */

float Distance(float x[], float y[], int n)
{
  int i;
  float sum = 0.0;

  for (i = 0; i < n; i++) {
    sum = sum + (x[i] - y[i])*(x[i] - y[i]);
  }
  return sqrt(sum);
} /* Distance */

void Read_matrix(char* prompt, MATRIX_T  A_local, int n,
                 int my_rank, int numprocs)
{
  int n_bar, i, j;
  MATRIX_T temp;

  n_bar = n/numprocs;
  /* Заполняем неиспользуемые столбцы в temp нулями */
  for (i = 0; i < n; i++)
    for (j = n; j < MAX_DIM; j++)
      temp[i][j] = 0.0;
  /* Считываем матрицу */
    if (my_rank == 0) {
        printf("%s\n", prompt);
        for (i = 0; i < n; i++)
            for (j = 0; j < n; j++)
                scanf("%f", &temp[i][j]);
```

```
    }
  /* Рассылаем полосы матрицы по процессам */
  MPI_Scatter(temp, n_bar*MAX_DIM, MPI_FLOAT, A_local,
              n_bar*MAX_DIM, MPI_FLOAT, 0, MPI_COMM_WORLD);
} /* Read_matrix */

void Read_vector(char* prompt, float x_local[], int n,
                 int my_rank, int numprocs)
{
  int n_bar, i;
  float temp[MAX_DIM];

  n_bar = n/numprocs;
  /* Считываем вектор */
  if (my_rank == 0) {
    printf("%s\n", prompt);
    for (i = 0; i < n; i++)
      scanf("%f", &temp[i]);
  }
  /* и рассылаем по процессам */
  MPI_Scatter(temp, n_bar, MPI_FLOAT, x_local, n_bar, MPI_FLOAT,
              0, MPI_COMM_WORLD);
} /* Read_vector */

void Print_matrix(char* title, MATRIX_T A_local, int n,
                  int my_rank, int numprocs)
{
  int n_bar, i, j;
  MATRIX_T temp;

  n_bar = n/numprocs;
  /* Собираем матрицу */
  MPI_Gather(A_local, n_bar*MAX_DIM, MPI_FLOAT, temp,
             n_bar*MAX_DIM, MPI_FLOAT, 0, MPI_COMM_WORLD);
  /* и печатаем */
  if (my_rank == 0) {
    printf("%s\n", title);
    for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++)
```

```
        printf("%4.1f ", temp[i][j]);
      printf("\n");
    }
  }
} /* Print_matrix */

void Print_vector(char* title, float x_local[], int n,
                  int my_rank, int numprocs)
{
  int n_bar, i;
  float temp[MAX_DIM];

  n_bar = n/numprocs;
  MPI_Gather(x_local, n_bar, MPI_FLOAT, temp, n_bar, MPI_FLOAT,
             0, MPI_COMM_WORLD);
  if (my_rank == 0) {
    printf("%s\n", title);
    for (i = 0; i < n; i++)
      printf("%4.1f ", temp[i]);
    printf("\n");
  }
} /* Print_vector */
```

Командная строка для компиляции программы выглядит так:

```
mpicc MPI_Jacobi.c -o MPI_Jacobi -lm
```

При большой размерности матрицы коэффициентов ввод данных с клавиатуры может оказаться весьма утомительным. Чтобы избежать этого и уменьшить вероятность ошибки, можно использовать возможность перенаправления стандартного ввода. При командной строке следующего вида

```
mpirun -np 2 MPI_Jacobi < 4Jacobi.dat
```

поведение программы будет таким же, как если бы содержимое текстового файла `4Jacobi.dat` было бы набрано на клавиатуре.

Литература

[1] Snir M, Otto S.W., Huss-Lederman S., Walker D, and Dongarra J. MPI: The Complete Reference. MIT Press. Boston, 1996. *http://www.netlib.org/utk/papers/mpi-book/mpi-book.html*

[2] *htpp://parallel.ru*

[3] Немнюгин С., Чаунин М., Комолкин А. Эффективная работа: UNIX. СПб.:Питер, 2003.

[4] Немнюгин С., Стесик О. Параллельное программирование для многопроцессорных вычислительных систем. СПб.:БХВ-Петербург, 2002.

[5] Богачев К.Ю. Основы параллельного программирования. М.:Бином, 2003.