Введение в MPI

Рахматуллин Д. Я.

Уфа, ИМВЦ УНЦ РАН

УДК 004.438

Содержание

Занятие 1. Суперкомпьютеры и параллельные вычисления

Зачем нужны суперкомпьютеры?

Существует множество вычислительных задач, для решения которых не требуется суперкомпьютер — мощная многопроцессорная вычислительная система, их запросам вполне удовлетворяет обычный персональный компьютер. Эти задачи характеризуются тем, что время их выполнения измеряется лишь секундами или минутами, а объем требуемой ими оперативной памяти обычно не превышает гигабайта. Многие из них уже включены в библиотеки функций известных математических программ, таких как Maple, Mathematica, Matlab и других.

Однако есть множество прикладных задач, для решения которых суперкомпьютер просто необходим. Это, например:

- прогнозирование погоды и другие климатические расчеты;
- обеспечение сейсмической устойчивости мостов;
- компьютерная анимация;
- crash-тесты в автомобилестроении;
- нефте- и газодобыча;
- моделирование столкновения черных дыр;
- моделирование формирования первых звезд;
- моделирование процесса образования белковых молекул.

Параллельные вычисления

Параллельные вычисления — это многопоточные вычисления, направленные на решение единой задачи. Каждый поток вычислений выполняется независимо от других, допуская, однако, взаимный обмен данными или даже наличие общего адресного пространства.

Конечно, если бы существовала возможность создать процессор, не уступающий по мощности нынешним суперкомпьютерам, то счет любой задачи на нем был бы не менее эффективен. Но стоимость такого процессора превысила бы во много раз цену равной ему по суммарной мощности многопроцессорной системы, использующей принципы параллелизма. С другой стороны, суперкомпьютер собранный уже из этих суперпроцессоров будет во много раз быстрее при применении грамотного распараллеливания расчетов. Отметим, что большинство выпускаемых ныне процессоров для персональных компьютеров используют **внутренний** параллелизм вычислений, который позволяет совершать одновременно и независимо несколько

простейших операций. Полученный таким способом «микросуперкомьютер» дает более дешевый прирост скорости вычислений, чем процессор той же мощности, построенный лишь за счет применения сложнейших нанотехнологий. Это еще раз показывает, что слово «многопроцессорный» в определении суперкомпьютера является ключевым, в отличие от слова «мощный», которое весьма условно.

Однако есть задачи, алгоритм которых принципиально не допускает «хорошей» параллелизации, т.е. разбиения на определенное количество частей примерно одинаковой вычислительной сложности, каждая из которых может выполняться независимо (или почти независимо) от других. Чем сильнее каждая подзадача алгоритмически зависит от остальных, тем сложнее структура параллельной программы и меньше эффективность распараллеливания. Поэтому, даже если вычисления длятся часами, но задача плохо параллелизуется, то запускать ее на суперкомпьютере невыгодно. Нередко, однако, встречаются случаи, когда решение можно получить различными методами. При этом бывает, что те из них, которые были лучшими по скоростным качествам на однопроцессорных компьютерах, проигрывают в применении к многопроцессорной технике, и наоборот. Поэтому следует говорить не о «плохих», в известном смысле, задачах, а о не- или малопригодных к параллелизации методах. К таким методам, например, относится метод прогонки. Лучше дело обстоит с методом Гаусса решения систем линейных алгебраических уравнений. Он дает приемлемое ускорение вычислений лишь при ограниченном числе процессоров; их оптимальное количество зависит от размера матрицы системы.

И все же существует множество задач, требующих больших затрат времени и ресурсов компьютера с одной стороны, и допускающих эффективную параллелизацию одного из методов решения — с другой. Грамотное применение многопроцессорных технологий в этих случаях позволяет ускорить вычисления в десятки и сотни раз.

Таким образом, параллельные вычисления необходимы, если выполнены следующие условия:

1) существует потребность в значительном сокращении времени решения данной задачи;

2) имеется достаточно хорошо параллелизуемый метод решения задачи;

3) алгоритм прост настолько, что время, потраченное на составление параллельной программы, окупается полученным ускорением вычислений.

Архитектура суперкомпьютера

Архитектура суперкомпьютера, т.е. структура его основных компонентов, обычно сводится к взаимодействию процессоров, оперативной памяти и каналов связи машины.

При этом общая архитектура складывается из частных архитектур, специфичных для каждого компонента системы. Можно выделить три основных типа (уровня) архитектур:

1. Архитектура процессоров системы: векторная и скалярная, конвейерная и неконвейерная. Поясним эти понятия и подчеркнем их достоинства и недостатки.

   Векторный процессор позволяет оперировать с массивами данных, в отличие от скалярного. Векторный процессор производительнее, но требует более аккуратного и сложного программирования под себя. Уровень развития микроэлектронных технологий не позволяет в настоящее время производить однокристальные векторные процессоры, поэтому системы, построенные с их использованием, довольно громоздки и чрезвычайно дороги.

   Конвейерная организация обработки потока команд — способ организации вычислений внутри каждого из процессоров, при котором сложная операция делится на несколько типичных **зависимых** частей. Каждая такая подоперация может обрабатывать данные (при их наличии) отдельно и независимо, а значит и одновременно с другими «работниками» конвейера. Поэтому, если имеется непрерывный поток задач, возникает **параллельность**, способствующая ускорению вычислений.

   Как правило, векторная и конвейерная архитектуры используются совместно, порождая векторно-конвейерный процессор. При этом возможность процессора работать с массивами данных позволяет эффективно использовать внутренний конвейер, не давая ему простаивать.

2. По способу доступа процессоров к оперативной памяти: SMP архитектуры (Symmetrical Multiprocessing — симметричная многопроцессорность) и MPP архитектуры (Massively Parallel Processing — массовая параллельность).

   SMP система (вычислительная система с SMP архитектурой) характеризуется тем, что ее процессоры имеют прямой и равноправный доступ к любому участку памяти, которая является **общей** для всех процессоров. Как следствие, программировать под такие системы относительно легко, и они сравнительно дешевы. Главный недостаток SMP компьютера — слишком плохая масштабируемость, т.е. невозможность серьезного увеличения количества процессоров в системе. Это происходит из-за конфликтов доступа к памяти, которые сложно устранить при большом количестве процессоров. Поэтому в SMP систему обычно включают не более 32 процессоров.

   В MPP системе оперативная память — **распределенная**, т.е. каждый вычислительный модуль имеет прямой доступ лишь к своей — локальной памяти. Модуль представляет собой один процессор или же SMP систему, состоящую из нескольких процессоров с общей памятью. К нелокальным данным запросы производятся через коммуникационную среду. Главным достоинством MPP суперкомпьютера является масштабируемость — количество процессоров может доходить до сотен тысяч. Недостатки системы вызваны той же причиной, что и ее единственное достоинство. Распределение памяти по вычислительным модулям значительно усложняет распараллеливание программ,

и ведет к значительной потере эффективности использования большого количества процессоров в случае, когда метод плохо параллелизуется. Таким образом, метод решения задачи, прекрасно проявляющий себя на суперкомпьютере с SMP архитектурой, может не реализовываться в сколько-нибудь эффективной программе для MPP системы. Еще два небольших минуса MPP компьютеров: относительная ограниченность локальной памяти каждого вычислительного модуля и высокая цена программного обеспечения.

Особо отметим гибридную архитектуру NUMA (nonuniform memory access — неоднородный доступ к памяти), которая сочетает в себе MPP архитектуру как способ распределения памяти и SMP архитектуру как способ доступа к ней. В такой системе поддерживается единое адресное пространство, но каждый вычислительный модуль имеет физически отдельную, локальную память, доступ к которой осуществляется в несколько раз быстрее, чем к удаленной.

3. По типу технического воплощения вычислительных модулей MPP компьютеры делятся на кластерные и обычные (некластерные) системы. Вычислительные модули суперкомпьютера кластерной архитектуры представляют собой обычные серийные компьютеры, мощности которых могут сильно разниться. Основными достоинствами кластерных систем являются их дешевизна, доступность для многих и простота самостоятельной «сборки». Под дешевизной нужно понимать невысокую стоимость не самих даже персональных компьютеров, а их машинного времени. Эти компьютеры, в основном, используются не для расчетов, а в качестве оргтехники, простаивая значительную часть времени без нагрузки. Следовательно, использование таких компьютеров для выполнения побочных задач практически ничего не стоит. Чаще всего кластерные системы в качестве средства коммуникации используют интернет. Отсюда и главный недостаток этих систем — из всех видов суперкомпьютеров у их процессоров самый медленный доступ к удаленной (нелокальной) памяти. Из этого, а также из того, что мощности вычислительных модулей кластеров могут значительно разниться, вытекает необходимость более сложного программирования, чем для обычных MPP систем.

   В заключение отметим, что каждый конкретный компьютер сочетает на различных уровнях несколько типов архитектур.

Показатели эффективности параллельной программы

Допустим, мы написали программу для многопроцессорных вычислений и убедились, что она считает правильно и даже довольно быстро. Но быстрота — понятие относительное, при разном количестве процессоров время счета может изменяться. При этом мы ожидаем ускорения вычислений при увеличении количества процессоров. Верны ли наши ожидания? Вообще говоря, нет. Нередки случаи, когда при увеличении количества процессоров программа считает все медленнее и медленнее.

Интуитивно понятно, что подобные факты напрямую связаны с качеством распараллеливания вычислений.

Было бы удобно найти численный показатель того, насколько хороша наша параллельная программа или, иными словами, насколько она близка к некому «идеалу». А что такое в данном случае идеал? Естественно назвать идеальной программу, которая при увеличении количества процессоров в N раз, считает в N раз быстрее.

Это определение эквивалентно следующему: пусть T_P — время счета на P процессорах; назовем идеальной такую параллельную программу, для которой при любом $P \geqslant 1$ верно тождество

$$\frac{T_1}{T_P} \equiv P. \tag{1}$$

Ясно, что для реальной программы будет верно лишь неравенство

$$\frac{T_1}{T_P} \geqslant P. \tag{2}$$

Поэтому, вводя обозначение

$$S_P := \frac{T_1}{T_P}, \tag{3}$$

мы получаем удобный численный показатель, с помощью которого можно сравнивать реальный случай с идеальным (обозначим его как S_P^{id}). Подставляя (1) в (3), имеем:

$$S_P^{id} \equiv P. \tag{4}$$

Чем меньше S_P в сравнении с S_P^{id}, тем хуже программа распараллелена. Параметр S_P обычно называют **ускорением** и вполне заслуженно, так как он показывает, во сколько раз ускорилась программа при увеличении числа процессоров с одного до P штук. Для идеальной параллельной программы $S_1^{id} \equiv 1$, $S_2^{id} \equiv 2$, $S_3^{id} \equiv 3$ и т.д.

Но сравнительный анализ ускорений имеет недостаток — параметр S_P не нормирован. Неясно, в каком случае программа ближе к идеалу: если $S_{10} = S_{10}^{id} - 1$ или при $S_{100} = S_{100}^{id} - 1$? Рассудок подсказывает, что в последнем. Проверим это. Разделив обе части тождества (3) на P, получим:

$$\frac{S_P}{P} = \frac{T_1}{T_P P}. \tag{5}$$

Введем обозначение:

$$E_P := \frac{S_P}{P}. \tag{6}$$

Из (4) следует, что

$$E_P^{id} \equiv 1. \tag{7}$$

Мы получили параметр, который в идеальном случае не зависит от числа процессоров и равен единице. Этот параметр называется **эффективностью**. Можно говорить, что чем ближе E_P при данном P к единице, тем эффективнее программа. Как правило, при одинаковых входных данных, по мере увеличения числа процессоров делить задачу между ними поровну становится все труднее. В результате синхронность счета нарушается, время вычислений увеличивается и, следовательно, эффективность снижается. Поэтому обычно E_P монотонно уменьшается, начиная со своего максимального значения — единицы. Однако иногда бывают исключения, например, если P оказывается «круглым» числом относительно разбиения задачи на части.

Итак, анализ параметров S_P и E_P может быть полезен при оценке качества параллельности программы или какой-либо ее части.

Занятие 2. Введение в MPI

Зачем нужен MPI

Предположим, мы решили написать программу для счета на суперкомпьютере. Возникает вопрос, на каком языке писать эту программу? Можно ли использовать обычные, знакомые всем языки или потребуется изобретать что-то новое? Кажется, что с помощью последовательного языка программирования можно написать лишь последовательную программу. Эффект от запуска такой программы на десяти процессорах будет небольшим — пользователь, скорее всего, получит десять одинаковых ответов.

Однако, все не так безнадежно — последовательные языки, тем не менее, применяются — более или менее успешно. Простейший с точки зрения программиста путь таков: на обычном языке пишется последовательная программа, которая автоматически распараллеливается на этапе компиляции или динамически — уже во время выполнения. Но простота такого подхода обманчива. Для того чтобы проанализировать и автоматически параллелизовать программу, необходим сверхинтеллектуальный компилятор. Попытки создать подобный компилятор предпринимаются, но получить удовлетворительный результат очень сложно, и пока эта задача еще решается.

Но и от привычных языков программирования отказываться нерационально — они известны многим, под них написано много готовых библиотек, а главное они прошли проверку временем. Существует компромиссный путь, более эффективный, чем первый и более удобный, чем второй. В чем отличие параллельной программы от последовательной? В наборе операций — обычные, например арифметические, операции сохраняются, но добавляются новые. В первую очередь — это операции доступа к удаленной памяти. Значит все, что нужно сделать, это дополнить последовательный язык программирования набором дополнительных операций для пересылки данных. Как правило, они объединяются в отдельную библиотеку, которая подключается к программе стандартным способом. За основу обычно берут различные версии языков FORTRAN и C/C++.

Стандартом де-факто для MPP машин на данный момент является библиотека параллельных функций MPI (Message Passing Interface — интерфейс передачи сообщений), реализации которой существуют для названных выше языков программирования.

Запуск программы

Приведем две команды, позволяющие компилировать и запускать параллельные программы, написанные на C/C++ с использованием библиотеки MPI. Их следует вводить в командной строке той операционной системы, где установлена реализация MPI.

Чтобы скомпилировать и собрать программу `prog.c`, написанную на C, следует выполнить команду

```
mpicc prog.c -o prog
```

Чтобы скомпилировать и собрать программу `prog.cpp` следует выполнить команду

```
mpiCC prog.cpp -o prog
```

При этом если в программе используется математическая библиотека `math.h`, в конце команды следует добавить ключ `-lm`. Чтобы использовать компилятор C, необходимо заменить `mpiCC` на `mpicc`.

Для того чтобы запустить программу `prog` на `n` процессорах с ограничением времени в `t` минут, следует выполнить такую команду:

```
mpirun -np n -maxtime t prog
```

Из других параметров выделим ключ `-stdiodir`. После него указывается имя директории, в которую будут записываться протокол запуска задачи, файл стандартного вывода и имена вычислительных модулей, использованных для счета задачи.

Две главные функции MPI

Мы будем использовать язык C++ и соответствующий ему синтаксис функций MPI.

Самые главные команды в MPI — функции начала и конца **параллельной** части программы. Именно между ними можно вставлять все остальные функции MPI. Все строки программы, не заключенные между ними, и все функции, не вызывающиеся оттуда, представляют собой **последовательную** часть программы. Процесс, выполняющий такую часть, замкнут в поле своих локальных данных [1] и не использует функции передачи сообщений.

Функция инициализации MPI является первой параллельной функцией в программе. Ее задача — подготовить коммуникационную среду для обмена сообщениями. Она создает **коммуникатор** `a`, который является описателем глобальной **области связи** — коммуникационной среды, позволяющей определенному подмножеству процессоров (в нашем случае — всему множеству) безопасно обмениваться информацией. При этом каждый процессор может «находиться» в нескольких коммуникаторах, получая в каждой из них уникальный номер — целое число от 0 до $P-1$, где P — количество процессов в коммуникаторе.

Перед тем, как привести нотацию функции инициализации, заметим, что сейчас и в дальнейшем мы для удобства будем записывать функции не в виде прототипов, а именно в том их виде, который следует использовать в программе. При этом если

[1] Имеются ввиду данные, расположенные в локальной оперативной памяти. В случае, если имеется общий для всех процессоров жесткий диск, через него по-прежнему возможен обмен данными.

аргумент функции — указатель на переменную какого-либо типа, то на его месте мы будем ставить амперсанд и переменную желаемого типа, допуская при этом замену этой конструкции одним указателем. Результат, возвращаемый почти всеми функциями MPI — это `MPI_SUCCESS` в случае успешного выполнения подпрограммы и код ошибки в противном случае *(ПРИМЕР 2.1)*.

Итак, функция инициализации имеет вид:

```
MPI_Init(&argc, &argv)
```

Аргумент	Тип	Описание
`argc`	`int`	Количество параметров командной строки. Является первым аргументом функции `main`
`argv`	`char**`	Массив параметров командной строки. Является вторым аргументом функции `main`

Функция `MPI_Init` запрашивает параметры командной строки, которые получает транзитом через функцию `main`. Зачем это нужно? Дело в том, что при запуске программы команда `mpirun` добавляет в командную строку информацию, без которой `MPI_Init` не может обойтись *(УПРАЖНЕНИЕ 2.1)*.

Функция завершения параллельной части выглядит следующим образом:

```
MPI_Finalize()
```

Удобнее всего завершать параллельную часть в самом конце программы. Однако если функция `main` возвращает значение, то для корректного завершения работы программы функцию `MPI_Finalize` следует поместить перед оператором `return`. С той же целью ее необходимо вызывать перед каждым выполнением функции `exit`.

ПРИМЕР 2.1

Пусть каждый процесс поприветствует нас и выведет коды возврата функций `MPI_Init` и `MPI_Finalize`.

```
//Пример 2.1
#include <mpi.h>
#include <stdio.h>
int main(int argc, char **argv)
{
 int i;
 //Начало параллельной части программы
 i= MPI_Init(&argc, &argv);
 if (i == MPI_SUCCESS)
  printf("Successful initialization with code %d\n", i);
 else
```

```
  printf("Initialization failed with error code %d\n", i);
 i= MPI_Finalize();
 if (i == MPI_SUCCESS)
  printf("Successful MPI_Finalize() with code %d\n", i);
 else
  printf("MPI_Finalize() failed with error code %d\n", i);
 //Конец параллельной части программы
 return 0;
}
```

УПРАЖНЕНИЕ 2.1

Составьте программу, в которой каждый процесс выводит на экран[2] параметры командной строки. Каждый параметр следует выводить в отдельной строке, сопровождая порядковым номером.

[2]Здесь и далее под выводом «на экран» подразумевается вывод в стандартный поток вывода

Занятие 3. Распараллеливание четырьмя функциями

Информационные функции MPI

С помощью функции

```
MPI_Comm_size(COMM, &size)
```

каждый процесс может узнать общее количество процессов в некоем коммуникаторе `COMM`. Обычно в качестве `COMM` используется глобальный коммуникатор `MPI_COMM_WORLD`. Параметры функции:

Аргумент	Тип	Описание
`COMM`	`MPI_Comm`	Название коммуникатора
size	`int`	Число процессов коммуникатора `COMM`

Здесь и далее аргументы, которые используются функцией для *возврата* значений, выделяются не полужирным шрифтом, а курсивом.

Как уже отмечалось, каждый процесс, относящийся к какой-либо области связи, имеет в ней уникальный номер — целое число от `0` до `size-1`, где `size` — общее количество процессов в ней. Чтобы узнать этот номер процесс должен вызвать функцию

```
MPI_Comm_rank(COMM, &rank)
```

Аргумент	Тип	Описание
`COMM`	`MPI_Comm`	Название коммуникатора
rank	`int`	Номер процесса в коммуникаторе `COMM`

Простейшую программу с использованием информационных функций мы приводим в *ПРИМЕРЕ 3.1.*

Отметим, что всего четырех изученных нами функций MPI уже вполне хватает, чтобы параллелизовать некоторые простые вычисления *(ПРИМЕР 3.2, УПРАЖНЕНИЕ 3.1, УПРАЖНЕНИЕ 3.2).*

Функция досрочного завершения процессов MPI_Abort

Если требуется принудительно завершить работу всех процессов какого-либо коммуникатора, следует использовать специальную функцию MPI — `MPI_Abort`. Она пробует наилучшим способом прервать выполнение всех задач в группе коммуникатора `Comm`, причем вызывать функцию `MPI_Abort` можно из любого процесса коммуникатора. У нее всего два параметра: название коммуникатора и код ошибки, указываемый программистом и возвращаемый системе.

```
MPI_Abort (Comm, error)
```

Аргумент	Тип	Описание
Comm	MPI_Comm	Название коммуникатора, все процессы которого подлежат завершению
error	int	Код ошибки

Использование функции MPI_Abort показано в *ПРИМЕРЕ 3.3.*

ПРИМЕР 3.1

Создадим программу, в которой каждый процесс выводит приветствие, свой порядковый номер в глобальном коммуникаторе и общее количество процессов в нем.

```
//Пример 3.1
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv)
{
 int i, size, rank;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 printf("Hello! I'm a process #%d of %d processes\n", rank, size);
 MPI_Finalize();
 return 0;
}
```

ПРИМЕР 3.2

Приведем пример программы, реализующей простейшее распараллеливание перемножения матрицы размера 10x10 и десятимерного вектора по двум процессам. Работа между процессами разделяется так: первые пять компонентов результирующего вектора вычисляются первым процессом, вторые пять — вторым. Очевидно, что вычисления проходят совершенно независимо, так же как и вывод ответа. С учетом того, что результаты вычислений могут смешаться при выводе, каждый компонент вектора выводится с указанием порядкового номера.

```
//Пример 3.2
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv)
{
```

```
 int size, rank, i, j, k, a[10][10], b[10], c[10];
//Инициализация (заполнение данными) массивов a и b:
 for (i=0;i<10;i++)
 {
  b[i]= i+1;
  for (j=0;j<10;j++)
   a[i][j]= (i+1)*(j+1);
 }
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 //В зависимости от номера процесса варьируются
 //номера начальной и конечной строк счета
 for(i=rank*10/size;i<(rank+1)*10/size;i++)
 {
  c[i]= 0;
  for(j=0;j<10;j++)
   c[i]+= a[i][j]*b[j];
  printf("c[%d]= %d\t(Process #%d)\n", i, c[i], rank);
 }
 MPI_Finalize();
 return 0;
}
```

ПРИМЕР 3.3

Продемонстрируем использование функции `MPI_Abort`. В приведенном примере программа досрочно завершается, если общее количество процессов меньше двух.

```
//Пример 3.3
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv)
{
 int i, size, rank;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 if (size<2)
  MPI_Abort(MPI_COMM_WORLD,1);
 ... //Основная часть программы
 MPI_Finalize();
```

```
 return 0;
}
```

УПРАЖНЕНИЕ 3.1

Параллелизуйте умножение квадратной матрицы размера N x N на вектор длины N по произвольному количеству процессов (начиная с одного)[3] .

УПРАЖНЕНИЕ 3.2

Напишите параллельную программу, выводящую на экран таблицу умножения произвольного размера. Вывод организуйте построчный, не обращая внимания на возможную путаницу строк.

[3]Здесь и далее мы подразумеваем, что алгоритм разбиения задачи на параллельные части должен быть в разумных пределах оптимален.

Занятие 4. Точечный обмен сообщениями

Функции MPI_Send и MPI_Recv

Допустим, что процессу с номером N_1 понадобилась некая информация, которой обладает процесс с номером N_0. При этом предполагается, что существует коммуникатор `Comm`, в который входят оба процесса, а их идентификаторами в нем служат названные выше номера.

Помочь одному процессу отправить сообщение с нужной информацией, а другому его принять могут две функции из библиотеки MPI: `MPI_Send` и `MPI_Recv`. Эти функции реализуют направленный обмен данными между процессами, называемый **точечным**. Такие функции обычно называют функциями типа **«точка-точка»**. Рассмотрим их по порядку.

Начнем с функции отправки. Как правило, информация, которую требуется передать, представляет собой массив каких-либо однотипных данных. Однако это может быть и часть массива, и просто одно число. В любом случае, то, что мы хотим отправить, должно храниться в памяти и иметь идентификатор. Его адрес и будет первым параметром функции отправки сообщений `MPI_Send`. При этом если мы пересылаем массив, то достаточно указать его название, которое и будет являться его адресом.

Вторым параметром функции отправки сообщений является количество отсылаемых элементов. При отправлении целого массива, таким образом, указывается общее количество его элементов, при пересылке некоторой непрерывной (т.е. непрерывающейся) части массива — размер этой части. Если же мы отправляем не массив, а одну переменную, то данный параметр следует указывать равным единице. Для удобства, в дальнейшем мы предполагать, что пересылаются именно элементы массива.

Третьим параметром, как и можно было ожидать, является тип пересылаемых данных. При этом название типа в MPI не совпадает с соответствующим типом в C, но легко из него получается.

Типы данных в MPI	Типы данных в C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

К этому списку можно добавить тип `MPI_BYTE`, служащий для пересылки данных в двоичном виде, а также любые пользовательские типы, правильным образом зарегистрированные.

Четвертым параметром функции `MPI_Send` является номер процесса-получателя сообщения в данном коммуникаторе. У нас это N_1.

Следующий параметр — так называемый **тэг** (идентификатор) сообщения. Это любое целое число от 0 до 32767, задаваемое пользователем по своему усмотрению. Тэг позволяет:

- сделать сообщение уникальным, устранив тем самым возможную путаницу у процесса-приемника;
- повысить удобство работы с сообщениями, сделав тэги осмысленными. Для этого можно задавать тэги символьно, используя оператор макроподстановки `#define` для автоматической замены таких тэгов на числа *(ПРИМЕР 4.1)*.

И, наконец, шестой и последний параметр — название коммуникатора, в рамках которого ведется пересылка данных. У нас это коммуникатор `Comm`.

Приведем всю функцию целиком:

```
MPI_Send(  mass,  count,  type,
           N1,    tag,    Comm   )
```

Аргумент	Тип	Описание
`mass`		Адрес, с которого начинается пересылаемый участок информации
`count`	`int`	Число элементов сообщения
`type`		Тип элемента отправляемого массива
N_1	`int`	Номер процесса-получателя
`tag`	`int`	Тэг (идентификатор) сообщения
`Comm`	`MPI_Comm`	Коммуникатор, в котором идет пересылка данных

Для лучшего запоминания параметры записаны в две строки: первые три относятся непосредственно к отправляемой информации, остальные носят технический характер.

Второй процесс, чтобы получить сообщение, должен вызвать функцию приема сообщения, парную к `MPI_Send` — `MPI_Recv`. Большинство параметров этих функций совпадают:

```
MPI_Recv(  mass,  count,  type,
           N0,    tag,    Comm,  &status  )
```

Аргумент	Тип	Описание
mass		Адрес начала буфера для приема сообщения
count	int	Число элементов принимаемого массива
type		Тип элемента принимающего массива
N_0	int	Номер процесса-отправителя
tag	int	Тэг (идентификатор) сообщения
Comm	MPI_Comm	Коммуникатор, в котором идет пересылка данных
status	MPI_Status	Структура, содержащая информацию о полученном сообщении

Нужно учитывать следующее:

- Аргумент mass может иметь одно и то же название при использовании его в качестве параметра в функциях MPI_Send и MPI_Recv. Однако не нужно забывать, что область памяти, соответствующая этому аргументу, у разных процессов различна.

- Параметр count определяет максимальную емкость массива mass у принимающей стороны. Эта емкость должна быть не меньше, чем длина отправленного сообщения, иначе произойдет ошибка.

- Тип данных type у отправляющей и принимающей функций должен совпадать.

- В качестве значений параметров N_0 и Tag могут использоваться аргументы-джокеры — MPI_ANY_SOURCE и MPI_ANY_TAG соответственно. Использование первого джокера позволяет процессу-получателю функцией MPI_Recv принимать сообщение от любого процесса коммуникатора Comm, не задавая явно его номер. Это может потребоваться, если данный процесс ожидает нескольких сообщений из разных источников, но порядок их поступления заранее неизвестен. Джокер позволяет обрабатывать эти сообщения по мере их реального поступления, экономя, таким образом, время *(ПРИМЕР 4.1)*.

- Название коммуникатора Comm должно быть одним и тем же у отправителя и адресата. Заметим, что не существует джокераю азмер у принимающей стороны должен быть не меньше, чем у отправляюще., которого можно было бы подставлять в качестве названия коммуникатора.

- Количество параметров принимающей функции равно семи. В отличие от функции MPI_Send, у MPI_Recv в конце имеется дополнительный параметр status. Это структура, содержащая три поля: MPI_ERROR, MPI_TAG и MPI_SOURCE. Первое из них содержит код ошибки, второе — тэг отправленного сообщения, а третье — номер отправителя в коммуникаторе Comm.

  Существование полей с тэгом и номером отправителя сообщения не является излишним, несмотря на то, что они и так уже присутствуют в списке параметров функции MPI_Recv как N_0 и Tag. Дело в том, что в качестве их значений

могут использоваться аргументы-джокеры — MPI_ANY_SOURCE и MPI_ANY_TAG соответственно. Но для правильной обработки полученной информации нужно все-таки знать, от кого она пришла и/или каков ее тэг. Как раз для этого и используются поля MPI_SOURCE и MPI_TAG параметра status функции MPI_Recv.

Достоинства и недостатки блокирующих функций

Функции MPI_Send и MPI_Recv являются блокирующими. Это означает, что процесс, вызвавший подобную функцию, приостанавливается до тех пор, пока не операция не будет полностью выполнена.

Конкретнее, процесс переходит к выполнению следующего за MPI_Send оператора только после того, как данные, предназначенные для отправки, будут скопированы в системный буфер (локальный или удаленный — в зависимости от реализации MPI). Такая блокировка защищает программиста от ошибки, которая могла бы произойти, если бы данные успели измениться до их отправки другому процессу. При этом не гарантируется, что по выходу из функции отправки процесс-получатель успеет принять сообщение или, хотя бы начать его прием. Такая передача и прием сообщения называются **асинхронными**.

Функция MPI_Recv приостанавливает выполнение программы до тех пор, пока процесс полностью не примет сообщение и не поместит данные в свое адресное пространство. Это позволяет после выхода из этой функции уверенно пользоваться принимающим массивом, зная, что нужные данные в него уже помещены.

Таким образом, блокирующие функции хороши своей надежностью. Однако, это достигается некоторым замедлением программы. Другой недостаток функций с блокировкой — возможность возникновения **тупиковых ситуаций** *(ПРИМЕР 4.2)*.

ПРИМЕР 4.1

Напишем простейшую программу пересылки сообщений от одного процесса другому с выводом на экран элементов структуры status.

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define mymessage 99
int main (int argc, char **argv)
{
 int size, rank;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 if (size<2) //Досрочный выход, если процесс всего один
```

```
 {
   printf("Error: too few processes");
   MPI_Finalize();
   exit(1);
}
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 int i= (rank+1)*5;
 printf("\nBefore sending: i= %d on %d\n", i, rank);
 if (rank==0)
  MPI_Send(&i, 1, MPI_INT, 1, mymessage, MPI_COMM_WORLD);
 if (rank==1)
  MPI_Recv(&i, 1, MPI_INT, MPI_ANY_SOURCE,
           MPI_ANY_TAG, MPI_COMM_WORLD, &status);
 printf("After sending: i= %d on %d\n", i, rank);
 if (rank==1)
  printf   ("status.MPI_SOURCE=%d\nstatus.MPI_TAG=%d\nstatus.MPI_ERROR=%s",
  status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR);
 MPI_Finalize();
 return 0;
}
```

Результатом выполнения программы будут следующие строки:

```
Before sending: i= 5 on 0
After sending: i= 5 on 0

Before sending: i= 10 on 1
After sending: i= 5 on 1
status.MPI_SOURCE=0
status.MPI_TAG=99
status.MPI_ERROR=(null)
```

ПРИМЕР 4.2

Рассмотрим отрывок программы, в которой наверняка случится тупиковая ситуация. Здесь два процесса обмениваются друг с другом числами. Однако из-за того, что оба процесса вначале используют блокирующие функции приема, каждый из них зависает в ожидании сообщения, которое так никогда и не поступит.

```
 int j, i= (rank+1)*5;
 if (rank==0)
 {
  MPI_Recv(&j,1,MPI_INT,1,message2,MPI_COMM_WORLD,&status);
  MPI_Send(&i,1,MPI_INT,1,message1,MPI_COMM_WORLD);
```

```
}
if (rank==1)
{
 MPI_Recv(&j,1,MPI_INT,0,message1,MPI_COMM_WORLD,&status);
 MPI_Send(&i,1,MPI_INT,0,message2,MPI_COMM_WORLD);
}
```

Гарантированно избежать тупика поможет следующее изменение в программе:

```
int j, i= (rank+1)*5;
if (rank==0)
{
 MPI_Send(&i,1,MPI_INT,1,message1,MPI_COMM_WORLD);
 MPI_Recv(&j,1,MPI_INT,1,message2,MPI_COMM_WORLD,&status);
}
if (rank==1)
{
 MPI_Recv(&j,1,MPI_INT,0,message1,MPI_COMM_WORLD,&status);
 MPI_Send(&i,1,MPI_INT,0,message2,MPI_COMM_WORLD);
}
```

Ситуация, когда оба процесса вначале посылают друг другу сообщения, а потом принимают, может привести, а может и не привести к тупику. В случае, когда обмен буферизованный, то есть процесс блокируется лишь до тех пор, пока сообщение не скопируется в системный буфер, тупиковой ситуации не возникает. Если же отправка и прием в данной реализации MPI осуществляются синхронно, произойдет зависание.

УПРАЖНЕНИЕ 4.1

Параллелизуйте умножение квадратной матрицы произвольного размера N на вектор длины N по произвольному количеству процессов (начиная с одного). Это усовершенствованная версия программы из *УПРАЖНЕНИЯ 3.1*. Здесь ни одному из процессов матрица и вектор заранее неизвестны, их считывает из файла один из процессов и рассылает остальным. Результирующий вектор выводите на экран также лишь одним процессом.

УПРАЖНЕНИЕ 4.2

Усовершенствуйте программу из *УПРАЖНЕНИЯ 3.2*: напишите параллельную программу, выводящую на экран таблицу умножения произвольного размера в виде матрицы. При этом не допускайте путаницы строк.

УПРАЖНЕНИЕ 4.3

Напишите программу, разбивающую число `n` на простые множители. Для этого сопоставьте каждому процессу по одному простому числу. Передавайте число `n` по кольцу процессов коммуникатора, проверяя его делимость на соответствующие простые числа и уменьшая (деля) `n` в случае делимости без остатка.

Занятие 5. Точечный обмен сообщениями (продолжение)

Совмещенный прием и отправка данных (MPI_Sendrecv)

Если процессу требуется совершить одновременную отправку и прием сообщений в одном и том же коммуникаторе, удобно использовать совмещенную функцию пересылки `MPI_Sendrecv`. Рассмотрим ее параметры:

```
MPI_Sendrecv(  mass_s,  count_s,  type_s,  N_s,  tag_s,
               mass_r,  count_r,  type_r,  N_r,  tag_r,
               Comm,    &status                          )
```

Аргумент	Тип	Описание
`mass_s`		Адрес, с которого начинается пересылаемый участок информации
`count_s`	`int`	Число элементов отправляемого сообщения
`type_s`		Тип элемента отправляемого массива
`N_s`	`int`	Номер процесса-получателя
`tag_s`	`int`	Тэг отправляемого сообщения
`mass_r`		Адрес начала буфера для приема сообщения
`count_r`	`int`	Число элементов принимаемого сообщения
`type_r`		Тип элемента принимаемого сообщения
`N_r`	`int`	Номер процесса-отправителя
`tag_r`	`int`	Тэг принимаемого сообщения
`Comm`	`MPI_Comm`	Коммуникатор, в котором идет пересылка данных
`status`	`MPI_Status`	Структура, содержащая информацию о полученном сообщении

Во-первых, заметим, что эта функция совместима с обычными функциями `MPI_Send` и `MPI_Recv`. Далее, она обладает замечательным свойством: выбор решения о том, послать ли вначале сообщение или же принять, производится автоматически, причем так, что тупиковой ситуации с зависанием гарантированно не происходит. Другим достоинством функции совмещенной отправки и приема сообщений является сокращение текста программы и повышение ее удобочитаемости *(ПРИМЕР 5.1)*.

Другие функции типа «точка-точка»

Помимо блокирующих функций `MPI_Send` и `MPI_Recv` библиотека MPI дает в распоряжение функции и без блокировки, а также синхронные, буферизованные и согласованные:

Режимы выполнения	С блокировкой	Без блокировки
Стандартная посылка	MPI_Send	MPI_**I**send
Синхронная посылка	MPI_**S**send	MPI_**Is**send
Буферизованная посылка	MPI_**B**send	MPI_**Ib**send
Согласованная посылка	MPI_**R**send	MPI_**Ir**send
Прием информации	MPI_Recv	MPI_**I**recv

Функции без блокировки лишь инициализируют посылку/прием сообщения и немедленно возвращают управление в программу. Проверка факта отправки/получения данных осуществляется отдельными функциями.

Синхронный режим обмена сообщениями гарантирует, что отправка данных процессом закончится одновременно с их приемом другим процессом.

Буферизованная посылка подразумевает наличие буфера в локальной памяти передающего процесса. Отправляемые данные первым делом копируются именно в этот буфер. Для функции с блокировкой это дает выигрыш во времени, т.к. выход из такой функции происходит сразу же после копирования сообщения в локальный буфер.

При согласованной посылке начало передачи данных происходит лишь после инициализации операции приема, т.е. готовности принимающего процесса их принять сообщение.

Отметим, что все перечисленные функции могут использоваться друг с другом в любой комбинации вне зависимости от спецификации.

Мы не будем рассматривать подробно все эти функции, т.к. в большинстве случаев можно обойтись функциями `MPI_Send` и `MPI_Recv`. Если же в программе часто (в цикле, к примеру) используются обмены сообщениями и необходим максимальный выигрыш во времени, лучше перейти от функций типа «точка-точка» к коллективным функциям, о которых будет рассказано позже.

Функция измерения времени MPI_Wtime

Часто возникает необходимость измерить время выполнения какой-либо параллельной части программы, например, для вычисления показателя эффективности параллелизации. Для этих целей создана функция `MPI_Wtime`, позволяющая независимым от платформы, на которой реализуется программа, способом получить астрономическое время в секундах, прошедшее с какого-то фиксированного момента в прошлом. Нужный нам интервал времени, таким образом, можно измерить, найдя разность между значениями этой функции в конце и в начале промежутка. Добавим, что `MPI_Wtime` возвращает число вещественного типа с удвоенной точностью — `double`. Как видно, это одна из немногих функций MPI, возвращающая не код ошибки, а требуемое значение.

ПРИМЕР 5.1

Рассмотрим программу, реализующую обмен данными «по кольцу». Каждый процесс посылает сообщение процессу с номером на единицу больше или же нулевому, если процесса с б**о**льшим номером не существует. Ясно, что при этом каждый процесс должен и принять сообщение и отправить другое. Вот тут то нам и пригодится функция совмещенной отправки и получения сообщений `MPI_Sendrecv`.

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv)
{
 int rank, size, prev;
 MPI_Status status;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 //Каждый процесс высылает следующему сообщение - свой номер
 MPI_Sendrecv(&rank, 1, MPI_INT,(rank+1)%size,(rank+1)%size,
              &prev, 1, MPI_INT,(rank+size-1)%size, rank,
              MPI_COMM_WORLD, &status);
 printf("#%d received message %d", rank, prev);
 MPI_Finalize();
}
```

УПРАЖНЕНИЕ 5.1

Вставьте в программы из *УПРАЖНЕНИЙ 4.1* и *4.2* временн**ы**е отметки. Составьте таблицу времен при различных размерах матриц. Подсчитайте и проанализируйте параметры ускорения и эффективности программ.

УПРАЖНЕНИЕ 5.2

Напишите программу для демонстрации того, что функция `MPI_Sendrecv` не вызывает тупиковую ситуацию при обмене сообщениями, таком как в *ПРИМЕРЕ 4.1*.

УПРАЖНЕНИЕ 5.3

Подумайте, можно ли сделать *УПРАЖНЕНИЕ 4.3* с использованием функции `MPI_Sendrecv`.

Занятие 6. Коллективные функции MPI

Преимущества коллективных функций

В передаче данных функциями типа «точка-точка» одновременно задействовано не более двух процессов и часто этого достаточно. Если же есть необходимость, например, разослать одни и те же данные двум или более процессам, то придется либо многократно вызывать одни и те же функции с почти идентичными параметрами, либо использовать более быстрые и удобные **коллективные** функции.

Коллективными называются функции, выполняющиеся сразу всеми процессами какого-либо коммуникатора. В приведенном выше примере достаточно использовать всего одну коллективную функцию при условии, что все процессы, входящие в рассылку составляют ровно один коммуникатор. Как правило, таковым является глобальный коммуникатор `MPI_COMM_WORLD`.

Преимущества коллективных функций:

- Простота и удобство использования. Сокращение текста программы и увеличение ее наглядности.
- Существенная экономия времени: если в коммуникаторе `n` процессов, то для рассылки данных от одного процесса всем остальным потребуется `n-1` сообщений; при использовании же коллективной функции автоматически будет сделано порядка `ln(n)` сообщений.
- Наличие готовых решений для многих действий: рассылки и сбора данных, глобальных вычислительных операций (суммирование, нахождение максимума и т.п.). Большинство функций имеют более мощный векторный аналог.
- Уменьшена вероятность ошибок за счет сокращения и упрощения текста программы, отсутствия тэгов, изолированности операций — каждая функция реально использует не аргумент-коммуникатор, а его временный дубликат.

Коллективные функции можно классифицировать следующим образом:

- функции синхронизации (барьеры)
  - `MPI_Barrier`
- функции коллективного обмена данными
  - `MPI_Bcast`
  - `MPI_Gather, MPI_Gatherv`
  - `MPI_Allgather, MPI_Allgatherv`
  - `MPI_Scatter, MPI_Scatterv`
  - `MPI_Alltoall, MPI_Alltoallv`

- глобальные вычислительные функции
  - `MPI_Reduce`
  - `MPI_Allreduce`

Функция синхронизации MPI_Barrier

В MPI существует всего одна функция синхронизации — `MPI_Barrier`. Ее единственным аргументом является название коммуникатора, процессы которого нужно синхронизировать. Каждый из процессов, вызвав функцию `MPI_Barrier`, будет ожидать системный сигнал о том, что все остальные процессы также вызвали эту функцию, и лишь тогда продолжит работу. Говорят, что процессы прошли **точку синхронизации**. Гарантируется, что все процессы одновременно стартуют с этой точки. Синтаксис функции:

```
MPI_Barrier(Comm)
```

Аргумент	Тип	Описание
`Comm`	`MPI_Comm`	Название коммуникатора, в котором происходит синхронизация

Использование функции синхронизации:

- В отладочных целях. С ее помощью можно выявить некоторые ошибки параллелизации.
- При разделении задачи на несколько последовательных этапов, каждый из которых использует данные предыдущего.
- Для гарантированной синхронизации — все остальные коллективные процессы неявно предполагают синхронизацию, но ее может и не быть.

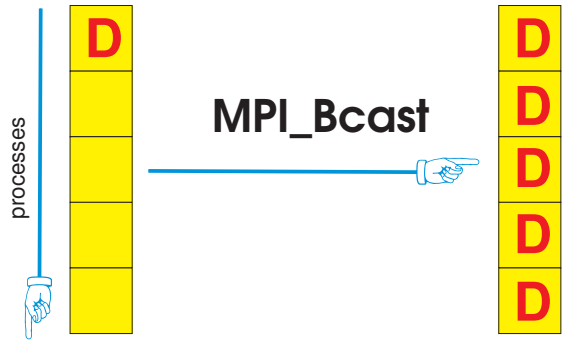
Таким образом, `MPI_Barrier` обычно используется в отладочных целях. Алгоритмической необходимости в барьерах, кажется, нет, поэтому в готовой программе их лучше исключить за счет более выверенного алгоритма.

Рассылка данных функцией MPI_Bcast

Задача рассылки какой-либо информации всем процессам коммуникатора возникает часто. К примеру, если входные данные программы находятся в отдельном файле, их следует считать одним из процессов, а затем разослать всем остальным. При этом не возникнет конфликтов доступа к файлу, а главное, будет выигрыш по времени — последовательное чтение с жесткого диска каждым из процессов происходит обычно медленнее пересылки данных по скоростной сети.

Но, как уже говорилось, использование функций типа «точка-точка» в этом случае возможно, но неудобно и неэффективно. Гораздо выгоднее использовать

специальную функцию рассылки `MPI_Bcast` *(ПРИМЕР 6.1)*. Пусть D — некая часть массива, которую один из процессов рассылает всем процессам коммуникатора. Схема действия функции рассылки показана на рисунке.



Список ее параметров несильно отличается от аргументов функции `MPI_Recv`: отсутствуют за ненадобностью тэг и переменная типа `MPI_Status`.

`MPI_Bcast(mass, count, type,` N_0`, Comm)`

Аргумент	Тип	Описание
mass		Адрес начала буфера для приема сообщения
`count`	`int`	Число элементов принимаемого массива
`type`		Тип элемента принимающего массива
N_0	`int`	Номер процесса-отправителя
`Comm`	`MPI_Comm`	Коммуникатор, в котором идет пересылка данных

Отметим, что **все** процессы коммуникатора `Comm`, как отправляющий, так и принимающие, должны вызвать эту функцию, иначе произойдет ошибка.

Функция-«совок» MPI_Gather и ее векторный вариант MPI_Gatherv

Функция `MPI_Gather` *(ПРИМЕР 6.1)* полезна, когда требуется собрать какую-либо однотипную информацию с процессов некоего коммуникатора и переслать ее одному из процессов. В этом случае неэффективен сбор данных «вручную» — с помощью функций типа «точка-точка».

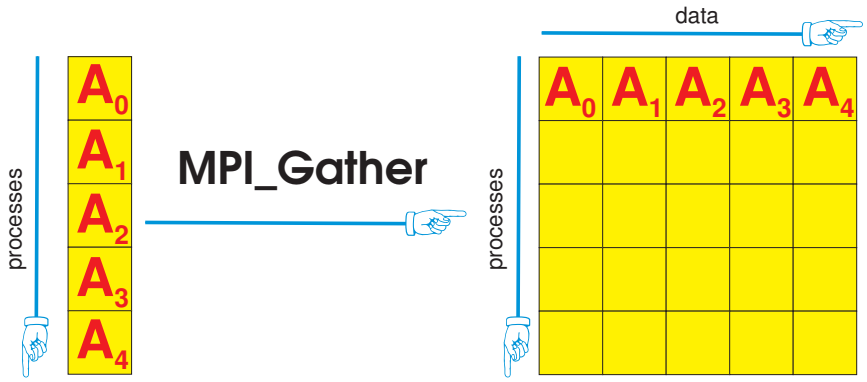
Оптимальное решение таково. Если от каждого процесса необходимо получить лишь по одному значению какой-либо переменной или по части массива одинакового размера, то следует проследить, чтобы требуемые переменные имели одинаковые типы, а затем использовать функцию `MPI_Gather` с такими параметрами:

```
MPI_Gather(  mass_s,  count_s,  type_s,
             mass_r,  count_r,  type_r,
             N0,      Comm              )
```

Аргумент	Тип	Описание
`mass_s`		Адрес, с которого начинается пересылаемый участок информации
`count_s`	`int`	Число элементов отправляемого сообщения
`type_s`		Тип элемента данных отправляемого сообщения
mass_r		Адрес начала буфера для приема сообщения
`count_r`	`int`	Число элементов принимаемого сообщения
`type_r`		Тип элемента данных принимаемого сообщения
N_0	`int`	Номер процесса-«совка»
`Comm`	`MPI_Comm`	Коммуникатор, в котором идет пересылка данных

Каждый процесс, включая собирающий, добавляет к общей информации свой блок — массив `mass_s` с количеством элементов `count_s` типа `type_s`. Все блоки упорядочиваются по возрастанию номеров процессов и объединяются в единый массив `mass_r`, который должен быть заранее инициализирован в локальной памяти процесса с номером N_0, и иметь размер не меньший, чем произведение `count_s` на общее количество процессов в коммуникаторе `Comm`. При этом `count_r` должен равняться `count_s`, а `type_s` должен совпадать с `type_r`.

Приняв, что P-й процесс содержит массив с данными A_P, и все такие массивы собирает 0-й процесс, действие функции `MPI_Gather` можно продемонстрировать на следующей схеме.



Если же в «совок» необходимо собрать данные из массивов разных размеров, следует использовать функцию `MPI_Gatherv` *(ПРИМЕР 6.1)*. В списке ее параметров целый положительный аргумент `count_r` замещен названием массива `mass_count_r`. Элементы этого массива играют ту же роль, что и `count_r`: i-ый его элемент означает, что от i-го процесса нужно получить `mass_count_r[i]` элементов массива `mass_s`. Еще одно отличие векторного варианта функции — наличие параметра смещений — массива `mass_disp_r` (от displacement — смещение, сдвиг). Он позволяет вручную «склеивать» блоки данных, полученных от процессов коммуникатора в единый массив `mass_r`, а именно: положительное целочисленное значение

элемента `mass_disp_r[i]` указывает, на сколько элементов следует сдвинуть блок `mass_s`, полученный от i-го процесса, относительно начала массива-собирателя `mass_r`. Подчеркнем, что каждый элемент массива смещений необходимо задавать, ориентируясь именно на нулевой, а не предыдущий, элемент массива `mass_r`. Ясно, что при желании между блоками в результирующем массиве можно устроить и наложения и разрывы.

```
MPI_Gatherv( mass_s,  count_s,       type_s,
             mass_r,  mass_count_r,  mass_disp_r,  type_r,
             N0,      Comm                                 )
```

Аргумент	Тип	Описание
`mass_s`		Адрес, с которого начинается пересылаемый участок информации
`count_s`	`int`	Число элементов отправляемого сообщения
`type_s`		Тип элемента данных отправляемого сообщения
mass_ r		Адрес начала буфера для приема сообщения
`mass_count_r`	`int*`	Адрес массива размеров принимаемых сообщений
`mass_disp_r`	`int*`	Адрес массива смещений принимаемых сообщений
`type_r`		Тип элемента данных принимаемого сообщения
N_0	`int`	Номер процесса-«совка»
`Comm`	`MPI_Comm`	Коммуникатор, в котором идет пересылка данных

Если принимающих процессов несколько, то можно несколько раз, изменяя лишь параметр N_0, вызвать функцию `MPI_Gather` (или `MPI_Gatherv`). Но если все процессы данного коммуникатора нуждаются в собранных по кусочкам данных, то проще и эффективнее использовать единственный вызов готовой функции `MPI_Allgather` (или `MPI_Allgatherv`). При этом списки параметров этих функций почти полностью соответствуют их «односовковым» вариантам — отличие заключается лишь в отсутствии ненужного параметра N_0 — номера принимающего процесса.

ПРИМЕР 6.1

Следующая программа демонстрирует примеры использования функций `MPI_Bcast`, `MPI_Gather` и `MPI_Gatherv`.

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv)
{
 int i, j, rank, size;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank==0) i = 10;
//Вывод первоначального i каждым процессом
printf("Step 0. i=%d on the process #%d\n", i, rank);
//Рассылка уникального i всем процессам
MPI_Bcast(&i, 1, MPI_INT, 0, MPI_COMM_WORLD);
//Вывод нового i каждым процессом
printf("Step 1. i=%d on the process #%d\n", i, rank);
i*= rank; //Теперь каждый процесс обладает разным i
int *m1 = new int [size];
//Сбор всех значений i в массив m1 нулевого процесса
MPI_Gather(&i, 1, MPI_INT, m1, 1, MPI_INT, 0, MPI_COMM_WORLD);
//Вывод полученной информации
if (rank==0)
 for(j=0; j<size; j++)
  printf("Step 2. i=%d on the process #%d\n", m1[j], j);
//Подготовка каждым из процессов массивов разной длины
int *m2 = new int [rank+1];
for (j=0;j<rank+1;j++)
{
 m2[j]= rank+5;
 printf("Step 3. Process #%d: m2[%d]= %d\n",rank,j,m2[j]);
}
int *mass_count_r, *mass_disp_r, *m3;
if (rank==0)
{
 mass_count_r= new int [size];
 mass_disp_r=    new int [size];
 mass_count_r[0]= 1;
 mass_disp_r[0]= 0;
 for(int i=1; i<size; i++)
 {
  mass_count_r[i]= i+1;
  mass_disp_r[i]= mass_disp_r[i-1]+mass_count_r[i-1];
 }
 m3 = new int [mass_disp_r[size-1]+mass_count_r[size-1]];
}
MPI_Gatherv(m2,rank+1,MPI_INT,m3,mass_count_r,mass_disp_r,
            MPI_INT, 0, MPI_COMM_WORLD);
if (rank==0)
{
 for (j=0;j<mass_disp_r[size-1]+mass_count_r[size-1];j++)
  printf("Step 4. m3[%d]= %d\n",j,m3[j]);
```

```
 }
 MPI_Finalize();
 delete m1,m2;
 if (rank==0)
  delete m3, mass_count_r, mass_disp_r;
}
```

УПРАЖНЕНИЕ 6.1

Усовершенствуйте программы из *УПРАЖНЕНИЙ 4.1, 4.2*, заменив все функции типа «точка-точка» коллективными функциями. Сравните время выполнения полученных программ с их старыми версиями.

УПРАЖНЕНИЕ 6.2

Считайте из файла два вектора одинаковой длины. Разбейте их на примерно одинаковые части по числу процессов и разошлите их им коллективной функцией рассылки данных. После того, как каждый из процессов вычислит локальное скалярное произведение частей векторов, разошлите результаты всем процессам сразу функцией `MPI_Allgather`. После этого нулевой процесс складывает полученные числа, получая полное скалярное произведение; первый процесс складывает все числа, кроме последнего, которое наоборот, вычитает из суммы; второй процесс вычитает уже последние два числа и так далее. Полученные результаты соберите одним из процессов и выведите на экран.

Занятие 7. Распределение данных. Совмещенные функции

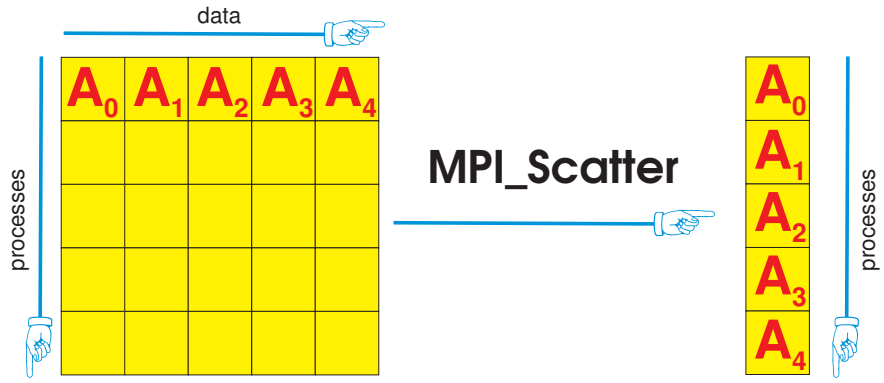
Функции распределения данных MPI_Scatter и MPI_Scatterv

В переводе с английского «scatter» — распылять, рассеивать, разбрасывать. Можно предположить, что функции `MPI_Scatter` и `MPI_Scatterv` являются обратными к функциям `MPI_Gather` и `MPI_Gatherv` («gather» — собирать), и это верная догадка. Если «совки» собирают куски массивов в единый массив какого-либо процесса, то «распылители» разрезают единый массив на части и распределяют его по процессам коммуникатора.

Синтаксис функций-«распылителей» полностью совпадает с синтаксисом функций-«совков». Отличие состоит лишь в использовании параметров функций. И в том и в другом случае один из процессов осуществляет отправку и прием сообщений. Однако если ранее его массив `mass_r` должен был быть достаточно велик для успешного приема всех сообщений, т.е. есть иметь элементов не менее чем `count_s*size`, то сейчас размер массива `mass_r` не должен превышать величины `count_s/size`. Остальные же процессы при использовании функции `MPI_Gather` реально использовали лишь буфер отправки `mass_s`, ничего не принимая в массив с именем `mass_r`. В функции `MPI_Scatter` таким неиспользуемым параметром для этих процессов будет буфер отправки.

```
MPI_Scatter(  mass_s,  count_s,  type_s,
              mass_r,  count_r,  type_r,
              N0,      Comm                )
```

Аргумент	Тип	Описание
`mass_s`		Адрес, с которого начинается пересылаемый участок информации
`count_s`	`int`	Число элементов отправляемого сообщения
`type_s`		Тип элемента данных отправляемого сообщения
mass_r		Адрес начала буфера для приема сообщения
`count_r`	`int`	Число элементов принимаемого сообщения
`type_r`		Тип элемента данных принимаемого сообщения
N_0	`int`	Номер процесса-«распылителя»
`Comm`	`MPI_Comm`	Коммуникатор, в котором идет пересылка данных



Совершенно аналогично функции `MPI_Gatherv`, векторный вариант функции-распылителя получается заменой одного скалярного параметра двумя массивами. Однако, если процессу-совку требовалось знать сколько элементов нужно собрать с каждого процесса и как разместить полученную информацию в своей памяти, то процессу-распылителю требуются данные о том, каковы размеры и положение частей локального массива, которые следует разослать членам коммуникатора. Следовательно, если в первом случае параметр `count_r` мы заменили адресами массивов `mass_count_r` и `mass_disp_r`, то во втором случае параметр `count_s` следует поменять на адреса массивов `mass_count_s` и `mass_disp_s`.

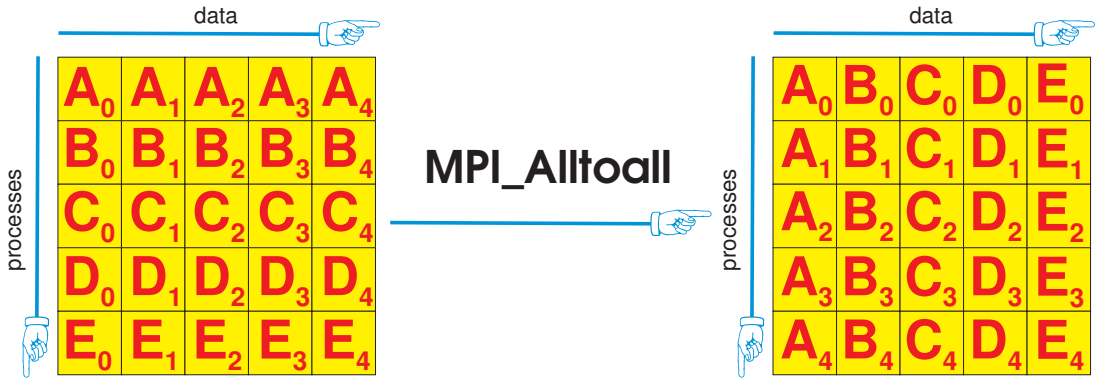
```
MPI_Scatterv( mass_s,   mass_count_s,  mass_disp_s,  type_s,
              mass_r,   count_r,       type_r,
              N0,       Comm                                 )
```

Аргумент	Тип	Описание
`mass_s`		Адрес, с которого начинается пересылаемый участок информации
`mass_count_s`	`int*`	Адрес массива размеров отправляемых сообщений
`mass_disp_s`	`int*`	Адрес массива смещений отправляемых сообщений
`type_s`		Тип элемента данных отправляемого сообщения
mass_r		Адрес начала буфера для приема сообщения
`count_r`	`int`	Число элементов принимаемого сообщения
`type_r`		Тип элемента данных принимаемого сообщения
N_0	`int`	Номер процесса-«распылителя»
`Comm`	`MPI_Comm`	Коммуникатор, в котором идет пересылка данных

Совмещенные функции обмена данных MPI_Alltoall и MPI_Alltoallv

Функция `MPI_Alltoall` симметрично объединяет элементы функций `MPI_Gather` и `MPI_Scatter`. Она сходна с функцией `MPI_Allgather` с тем отличием, что куски, из которых каждый процесс собирает свой массив, являются лишь частями массивов, которыми изначально обладает каждый из процессов. Таким образом, каждый

процесс рассылает свой массив остальным по аналогии с функцией MPI_Scatter, а не MPI_Bcast. Естественно, при этом процессы в итоге собирают, вообще говоря, разные массивы. Поясним сказанное на схеме и приведем формальные параметры функции:




```
MPI_Alltoall(  mass_s,  count_s,  type_s,
               mass_r,  count_r,  type_r,
               Comm                       )
```

Аргумент	Тип	Описание
mass_s		Адрес, с которого начинается пересылаемый участок информации
count_s	int	Число элементов отправляемого сообщения
type_s		Тип элемента данных отправляемого сообщения
mass_r		Адрес начала буфера для приема сообщения
count_r	int	Число элементов принимаемого сообщения
type_r		Тип элемента данных принимаемого сообщения
Comm	MPI_Comm	Коммуникатор, в котором идет пересылка данных

Функция MPI_Alltoallv является векторным вариантом функции MPI_Alltoall, легко получаясь из нее по аналогии с MPI_Gatherv и MPI_Scatterv:

```
MPI_Alltoallv(  mass_s,  mass_count_s,  mass_disp_s,  type_s,
                mass_r,  mass_count_r,  mass_disp_r,  type_r,
                Comm                                          )
```

Аргумент	Тип	Описание
mass_s		Адрес, с которого начинается пересылаемый участок информации
mass_count_s	int*	Адрес массива размеров отправляемых сообщений
mass_disp_s	int*	Адрес массива смещений отправляемых сообщений
type_s		Тип элемента данных отправляемого сообщения
mass_r		Адрес начала буфера для приема сообщения
mass_count_r	int*	Адрес массива размеров принимаемых сообщений
mass_disp_r	int*	Адрес массива смещений принимаемых сообщений
type_r		Тип элемента данных принимаемого сообщения
Comm	MPI_Comm	Коммуникатор, в котором идет пересылка данных

ПРИМЕР 7.1

Приведем пример простейшего применения функции MPI_Scatter. Массив, которым обладает нулевой процесс (его размерность в точности равна общему количеству процессов глобального коммуникатора), «распыляется» по всем процессам коммуникатора, уделяя кусок, в том числе и себе.

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv)
{
 int i, rank, size;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 int *mass;
 if (rank==0)
 {
  mass= new int [size];
  for(int j=0; j<size; j++)
   mass[j]= j*j;
 }
 MPI_Scatter(mass,1,MPI_INT,&i,1,MPI_INT,0,MPI_COMM_WORLD);
 printf("Process #%d received %d\n", rank, i);
 MPI_Finalize();
 if (rank==0)
  delete mass;
}
```

УПРАЖНЕНИЕ 7.1

Оптимизируйте код программы из *УПРАЖНЕНИЯ 6.1*, заменив полную рассылку матрицы, считываемой из файла, на блочную — по строкам. Проанализируйте изменения параметров ускорения и эффективности по сравнению со старой версией программы.

УПРАЖНЕНИЕ 7.2

Реализуйте рассылку данных «всем ото всех» с помощью функции `MPI_Alltoall`. Для этого на каждом из процессов определите и инициализируйте по одному одномерному массиву с разнящимися числами. После коллективного обмена сравните эти массивы с собранными.

Занятие 8. Глобальные вычислительные операции

MPI_Reduce

Глобальные вычислительные операции — это операции сложения, нахождения максима и т.п., выполняемые над векторами, распределенными по процессам какого-либо коммуникатора.

Рассмотрим первую функцию, практически реализующую глобальные вычислительные операции, `MPI_Reduce`. Принцип действия этой функции прост: если все процессы некоего коммуникатора обладают одномерными массивами `mass_s` одного и того же типа и одинаковой длины, то вызов коллективной функции глобального редуцирования с параметром `op` заполняет массив `mass_r` процесса с номером N_0, получая каждый его элемент применением операции `op` к элементам с теми же номерами всех массивов `mass` процессов коммуникатора. Рассмотрим синтаксис функции и перечень доступных операций `op`.

`MPI_Reduce(mass_s, mass_r, count_r, type_r, op,` N_0`, Comm)`

Аргумент	Тип	Описание
`mass_s`		Адрес, с которого начинается пересылаемый участок информации
mass_r		Адрес начала буфера для приема сообщения
`count_r`	`int`	Количество элементов принимаемого сообщения
`type_r`		Тип элемента данных принимаемого сообщения
`op`	`MPI_Op`	Название операции
N_0	`int`	Номер принимающего процесса
`Comm`	`MPI_Comm`	Коммуникатор, в котором идет пересылка данных

Предопределенные операции в функциях редукции MPI.

Название	Операция	Разрешенные типы
`MPI_MAX`	Максимум	Целые, вещественные
`MPI_MIN`	Минимум	
`MPI_SUM`	Сумма	Целые, вещественные
`MPI_PROD`	Произведение	
`MPI_LAND`	Логическое AND	Целые
`MPI_LOR`	Логическое OR	
`MPI_LXOR`	Логическое исключающее OR	
`MPI_BAND`	Поразрядное AND	Целые, байтовые
`MPI_BOR`	Поразрядное OR	
`MPI_BXOR`	Поразрядное исключающее OR	
`MPI_MAXLOC`	Максимальное значение и его индекс	Специальные типы
`MPI_MINLOC`	Минимальное значение и его индекс	для этих функций

Операции MAXLOC и MINLOC выполняются над специальными парными типами, каждый элемент которых хранит две величины: значения, по которым ищется максимум или минимум, и индекс элемента. Имеется 6 таких предопределенных типов:

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

MPI_Allreduce

Функция MPI_Allreduce, в отличие от MPI_Reduce, рассылает результат всем процессам коммуникатора. Список ее параметров отличается, поэтому, лишь отсутствием в нем номера принимающего процесса.

УПРАЖНЕНИЕ 8.1

Напишите программу, вычисляющую глобальный максимум во множестве массивов одинакового типа и размера. Каждый массив является локальным по отношению к одному процессу. Помимо максимума определите номер элемента, на котором он достигается, а также номер процесса, содержащего массив с этим элементом.

УПРАЖНЕНИЕ 8.2

Реализуйте приближенное вычисление интеграла одной переменной методом прямоугольников. При достаточно большом количестве точек разбиения отрезка интегрирования вычислите параметры ускорения и эффективности программы при разном количестве процессов. Сделайте выводы о качестве параллелизации задачи.

Занятие 9. Работа с коммуникаторами

Функции доступа к коммуникаторам

Функции доступа — локальные информационные функции. Основными функциями доступа к коммуникаторам являются `MPI_Comm_size`, `MPI_Comm_rank` и `MPI_Comm_compare`. Первые две из них мы уже рассмотрели на Занятии 3. Третья функция, `MPI_Comm_compare`, служит для сравнения между собой двух коммуникаторов. Она записывается следующим образом:

```
MPI_Comm_compare(Comm1, Comm2, &result)
```

Аргумент	Тип	Описание
`Comm1`	`MPI_Comm`	Первый сравниваемый коммуникатор
`Comm2`	`MPI_Comm`	Второй сравниваемый коммуникатор
result	`int`	Результат сравнения

Параметр *result* может принимать следующие значения:

`MPI_IDENT`	коммуникаторы совпадают
`MPI_CONGRUENT`	группы процессов, соответствующие коммуникаторам, имеют одинаковые атрибуты
`MPI_SIMILAR`	группы процессов, соответствующие коммуникаторам совпадают, но упорядочиваются по-другому
`MPI_UNEQUAL`	в иных случаях

Функции создания и уничтожения коммуникаторов

Простейший способ создания коммуникатора — дублирование существующего. Эту задачу решает следующая функция:

```
MPI_Comm_dup(Comm1, &Comm2)
```

Аргумент	Тип	Описание
`Comm1`	`MPI_Comm`	Коммуникатор-оригинал
`Comm2`	`MPI_Comm`	Коммуникатор-копия

Другой способ — расщепление коммуникатора функцией `MPI_Comm_split`:

```
MPI_Comm_split(Comm1, color, key, &Comm2)
```

Аргумент	Тип	Описание
`Comm1`	`MPI_Comm`	Родительский коммуникатор
`color`	`int`	Признак подгруппы
`key`	`int`	Порядковый параметр
`Comm2`	`MPI_Comm`	Дочерний коммуникатор

Группа процессов, связанная с родительским коммуникатором `Comm1`, расщепляется на подгруппы по признаку совпадения параметра `color`. Иными словами, каждый процесс коммуникатора `Comm1` вызывает функцию `MPI_Comm_split` с собственным произвольным параметром `color`. После этого система разбивает исходную группу процессов «по цветам», относя к одинаковой подгруппе процессы, имеющие одинаковое значение параметра `color`. Параметр `key` отвечает за упорядочивание внутри каждой подгруппы. Чем меньше его значение, тем меньше порядковый номер процесса в новом подкоммуникаторе. Если в группе оказались процессы с одинаковым значением `key`, они ранжируются по их порядку в родительском коммуникаторе *(ПРИМЕР 9.1)*.

В результате каждый процесс получает инициализированный дочерний коммуникатор *`Comm2`*, соответствующий именно той подгруппе процессов, в которую входит данный процесс. Таким образом, мы получаем некоторое количество **различных** коммуникаторов с одинаковым названием — не следует их путать!

Наконец, рассмотрим функцию уничтожения коммуникаторов:

`MPI_Comm_free(Comm)`

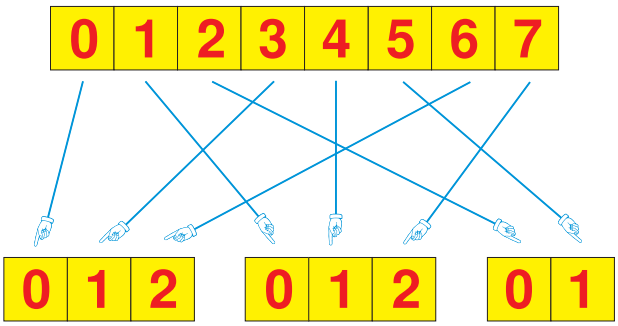
Аргумент	Тип	Описание
`Comm`	`MPI_Comm`	Уничтожаемый коммуникатор

ПРИМЕР 9.1

Продемонстрируем использование функции `MPI_Comm_split`. Мы распределим группу процессов глобального коммуникатора на три подгруппы — по признаку делимости порядкового номера процесса на 3.

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char **argv)
{
 int i, rank, size, newrank;
 MPI_Comm NewComm;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 MPI_Comm_split(MPI_COMM_WORLD, rank%3, rank, &NewComm);
 MPI_Comm_rank(NewComm, &newrank);
 printf("My rank is %d in MPI_COMM_WORLD and %d in NewComm\n", rank, newrank);
 MPI_Finalize();
}
```

К примеру, восемь процессов глобального коммуникатора разобьются следующим образом:



УПРАЖНЕНИЕ 9.1

Сделайте копию глобального коммуникатора. На основе этой копии создайте еще один коммуникатор, отличающийся от него лишь тем, что все процессы в нем имеют обратный порядок нумерации. Сравните: коммуникатор `MPI_COMM_WORLD` и его копию; копию с последним коммуникатором. Удалите созданные коммуникаторы.

УПРАЖНЕНИЕ 9.2

Разбейте коммуникатор `MPI_COMM_WORLD` на две подгруппы: в одну из них включите процессы, номера которых являются простыми числами, в другую — все остальные процессы. Порядок внутри дочерних коммуникаторов сделайте случайным. Удалите дочерние коммуникаторы.

Занятие 10. Проверка знаний

Ниже предлагается список вопросов, который можно использовать для контрольной проверки знаний учащихся. Вопросы можно использовать как в виде списка — для устных ответов, так и для зачета или экзамена, сформировав билеты.

1. Занятие 1

1.1. Что такое суперкомпьютер?

1.2. В каких задачах суперкомпьютер необходим?

1.3. Что такое параллельные вычисления?

1.4. Сформулируйте условия, при которых выгодно применять параллельные вычисления

1.5. Что такое архитектура суперкомпьютера?

1.6. Назовите три основных уровня архитектур суперкомпьютера

1.7. Какие технологии используются в архитектуре процессоров? Перечислите их основные достоинства и недостатки

1.8. Какие технологии используются в архитектуре процессоров? Перечислите их основные достоинства и недостатки

1.9. Какие технологии используются в архитектуре взаимодействия процессоров и оперативной памяти? Перечислите их основные достоинства и недостатки

1.10. Чем отличаются кластерные и некластерные системы? Назовите их основные достоинства и недостатки

1.11. Что такое ускорение параллельной программы?

1.12. Что такое эффективность параллельной программы? Зависит ли она от ускорения?

1.13. Что такое идеальная параллелизация? Чему равна ее эффективность, ускорение?

2. Занятие 2

2.1. Что представляет собой MPI?

2.2. Каков синтаксис командных строк компиляции и запуска MPI-параллельных программ?

2.3. Чем последовательная часть отличается от параллельной?

2.4. Чем отличаются коммуникатор и область связи?

2.5. Что такое функция инициализации? Каковы ее параметры?

2.6. Как записывается функция завершения параллельной части программы? В каких случаях ее использование необходимо?

3. Занятие 3

3.1. Как определить общее количество процессов в коммуникаторе?

3.2. Как процессу узнать свой номер в коммуникаторе? Зачем это нужно?

4. Занятие 4

4.1. Что такое обмен данными типа «точка-точка»?

4.2. Как выглядит функция отправки сообщений? Каковы ее параметры?

4.3. Перечислите основные типы данных библиотеки MPI

4.4. Зачем нужен идентификатор сообщения?

4.5. Как записывается функция приема сообщения?

4.6. Как должны соотноситься емкости передаваемого и принимающего массивов?

4.7. Что такое джокер? Какие бывают джокеры? Зачем они нужны?

4.8. Зачем функции `MPI_Recv` нужен параметр `status`?

4.9. Что такое блокирующая функция? Каковы ее достоинства и недостатки по сравнению с неблокирующими функциями?

5. Занятие 5

5.1. Каким образом можно совместить точечный прием и отправку данных?

5.2. Какие выгоды от такого совмещения?

5.3. Расскажите о второстепенных функциях типа «точка-точка». Почему они нечасто применяются?

5.4. Как измерять время выполнения параллельной части программы?

5.5. Как принудительно завершать процессы коммуникатора?

6. Занятие 6

6.1. Дайте определение коллективной функции.

6.2. Каковы достоинства коллективных функций?

6.3. Как классифицируются коллективные функции?

6.4. Расскажите про функцию синхронизации

6.5. Как действует функция глобальной рассылки данных?

6.6. Что такое функция-«совок»? Каков ее синтаксис?

6.7. Опишите векторный вариант функции-«совка»

7. Занятие 7

7.1. Как действуют функции распределения данных?

7.2. Каков синтаксис функций-«распылителей»?

7.3. Каков результат действий совмещенных коллективных функций обмена данными?

7.4. Приведите списки параметров совмещенных функций

8. Занятие 8

8.1. Что такое глобальная вычислительная операция?

8.2. Каков принцип действия функции `MPI_Reduce`?

8.3. Каков синтаксис функции `MPI_Reduce`?

8.4. Перечислите предопределенные операции, используемые в функции `MPI_Reduce`

8.5. Что такое парный тип данных? Приведите пример

8.6. Какая функция обобщает функцию `MPI_Reduce`?

ЛИТЕРАТУРА

[1] *Message Passing Interface Forum.* MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org/docs/mpi1-report.pdf

[2] *Букатов А. А., Дацюк В. Н., Жегуло А. И.* Программирование многопроцессорных вычислительных систем. Ростов-на-Дону. Издательство ООО «ЦВВР», 2003, 208 с.

[3] *Воеводин В. В., Воеводин Вл. В.* Параллельные вычисления. — СПб.: БХВ-Петербург, 2002. — 600 с.

[4] *Евсеев И.* MPI для начинающих.
http://ilya-evseev.narod.ru/articles/mpi/index.html

[5] *Абрамова В. А., Баранов А. В., Лацис А. О., Храмцов М. Ю.* Руководство программиста.
http://www.jscc.ru/informat/mpiGuide.zip

[6] *Абрамова В. А., Баранов А. В., Лацис А. О., Храмцов М. Ю.* Руководство пользователя системы МВС-1000/М.
http://www.jscc.ru/informat/1000MUsrGuide.zip