Министерство образования Московской области
Государственный университет «Дубна»

Институт системного анализа и управления
Кафедра распределенных информационно-вычислительных систем

**М. В. Башашин, Е. В. Земляная,
О. И. Стрельцова**

Практическое введение в технологию MPI на кластере HybriLIT

УЧЕБНОЕ ПОСОБИЕ

Рекомендовано учебно-методическим советом
университета «Дубна» в качестве учебного пособия для студентов,
обучающихся по направлениям подготовки «Информатика и
вычислительная техника», «Программная инженерия»,
«Фундаментальная информатика и информационные технологии»,
«Информационные системы и технологии», «Прикладная
математика и информатика», «Автоматизация технологических
процессов и производств»
(бакалавриат)

Дубна
2019

УДК 004.42
ББК 32.973я73
Б 33

Р е ц е н з е н т :
доктор физико-математических наук *Д.С. Кулябов*

Башашин, М. В.

Б 33 Практическое введение в технологию MPI на кластере HybriLIT : учебное пособие / М. В. Башашин, Е. В. Земляная, О. И. Стрельцова. — Дубна : Гос. ун-т «Дубна», 2019. — 50 [2] с.

ISBN 978-5-89847-571-0

Данное учебное пособие предназначено для практического овладения основами параллельного программирования на базе технологии *Message Passing Interface* (*MPI*) с использованием ресурсов гетерогенной вычислительной платформы *HybriLIT* Многофункционального информационно-вычислительного центра Лаборатории информационных технологий ОИЯИ.

Пособие ориентировано главным образом на студентов государственного университета «Дубна», изучающих курс «Архитектура вычислительных систем» (бакалавриат), и может быть полезно лицам, самостоятельно либо в рамках других дисциплин изучающих технологию *MPI*.

УДК 004.42
ББК 32.973я73

ISBN 978-5-89847-571-0

© Университет «Дубна», 2019
© Башашин М. В., Земляная Е. В.,
Стрельцова О. И., 2019

Оглавление

Глава 1. Краткая инструкция по работе на учебно-тестовом полигоне HybriLIT. Компиляция и запуск MPI-приложений 5
1.1. Общая характеристика учебно-тестового полигона HybriLIT....... 5
1.2. Работа с MPI-приложениями 7
1.2.1. Компиляция.. 7
1.2.2. Запуск в *batch*-режиме.. 8

Глава 2. MPI: общие принципы, базовые функции............................ 9
2.1. Логическое представление MPI-программы 9
2.2. MPI-функции общего назначения 10
2.2.1. Объявление и закрытие параллельной области 10
2.2.2. Понятия о группе и коммуникаторе 11
2.2.3. Примеры практических заданий 12
2.3. Пример распределения вычислений между параллельными MPI-процессами 13
2.4. Обмены между отдельными MPI-процессами 15
2.4.1. Блокирующая передача сообщений 15
2.4.2. Неблокирующая передача сообщений..................................... 17
2.4.3. Двунаправленный обмен... 19
2.4.4. Примеры практических заданий.. 21

Глава 3. MPI-процедуры для коллективного взаимодействия процессов 23
3.1. Характерные особенности процедур коллективного взаимодействия процессов.. 23
3.2. Барьерная синхронизация всех процессов..................................... 23
3.3. Коллективные коммуникационные операции 24
3.3.1. Рассылка информации от одного процесса всем остальным членам группы 24
3.3.2. Рассылка фрагментов массива данных всем процессам группы 25
3.3.3. Сборка распределенных по процессам данных в один массив 27
3.3.4. Глобальные вычислительные операции над данными, расположенными в адресных пространствах различных процессов.. 28
3.4. Примеры практических заданий... 31

Глава 4. Эффективность параллельных вычислений 33
4.1. Анализ эффективности применения некоторых MPI-функций..... 33
4.1.1. Пример 1. Рассылка данных всем процессам........................... 34

4.1.2. Пример 2. Поэлементное перемножение векторов 35
4.1.3. Примеры самостоятельных заданий 41
4.2. Оценка эффективности параллельной реализации алгоритмов
и программ на основе закона Амдала 41
4.2.1 Формула Амдала для асимптотической оценки ускорения
вычислений при параллельной реализации 41
4.2.2. Примеры самостоятельных заданий 44

Список рекомендованной литературы 45
Приложение 1. Основные команды Linux, редакторы
vim и nano 46
Приложение 2. Основные команды: Module и SLURM 49
Приложение 3. Пять основных шагов запуска MPI-приложений
на кластере HybriLIT 50

Глава 1. Краткая инструкция по работе на учебно-тестовом полигоне HybriLIT. Компиляция и запуск MPI-приложений

1.1. Общая характеристика учебно-тестового полигона HybriLIT

Гетерогенная вычислительная платформа HybriLIT является частью Многофункционального информационно-вычислительного комплекса (МИВК) Лаборатории информационных технологий Объединенного института ядерных исследований (Дубна) [1]. Платформа HybriLIT состоит из суперкомпьютера «ГОВОРУН» и учебно-тестового полигона HybriLIT, имеющих единую программно-информационную среду. Практические занятия по курсу «Архитектура вычислительных систем» проходят на учебно-тестовом полигоне HybriLIT (далее – кластер HybriLIT), но единая программная среда позволяет студентам получить опыт работы на суперкомпьютерах, таких как суперкомпьютер «ГОВОРУН».

Гетерогенная структура вычислительных узлов позволяет разрабатывать параллельные приложения для решения широкого круга математических ресурсоемких задач.

Одной из многих технологий параллельного программирования, работа с которыми поддерживается кластером HybriLIT, является технология *Message Passing Interface* (*MPI*) — интерфейс передачи сообщений [2; 3]. MPI представляет собой один из наиболее широко распространенных инструментариев для создания параллельных приложений. Стандарт MPI задает интерфейс, которому должны следовать как система программирования на каждой вычислительной платформе, так и пользователь при создании своих программ. MPI работает с языками программирования *C*, *C*++, *Fortran*, *Java* путем добавления соответствующей библиотеки.

Для получения доступа на кластер необходимо зарегистрироваться. Для этого нужно заполнить форму на сайте кластера и получить временный пароль, который пользователь должен заменить на постоянный согласно инструкции.

Работа на кластере осуществляется в удаленном режиме по *SSH*-протоколу:

• для пользователей *Windows*: с использованием *SSH*-клиента, например, приложения *PuTTY*;

• для пользователей *Linux* и *MacOS*: ввод в терминале

```
$ ssh USERNAME@hydra.jinr.ru.
```

Кластер HybriLIT находится под управлением операционной системы *Scientific Linux* 7.4.

Для создания и редактирования файлов на кластере HybriLIT доступны несколько редакторов, включая `nano` и `vim`.

На кластере HybriLIT установлены различные пакеты, библиотеки, инструменты для отладки и профилирования параллельных приложений и другое программное обеспечение (ПО). Для удобства использования только необходимого ПО на кластере установлен пакет *Modules*, предназначенный для динамического изменения переменных окружения. Перед компиляцией приложения пользователю следует загрузить модули, необходимые для работы.

Для распределения ресурсов кластера между пользовательскими задачами на кластере в качестве планировщика заданий используется *SLURM* (рис. 1) и для запуска приложения в режиме очереди – команда *sbatch*. Для запуска приложения с помощью команды *sbatch* требуется использовать скрипт-файл.

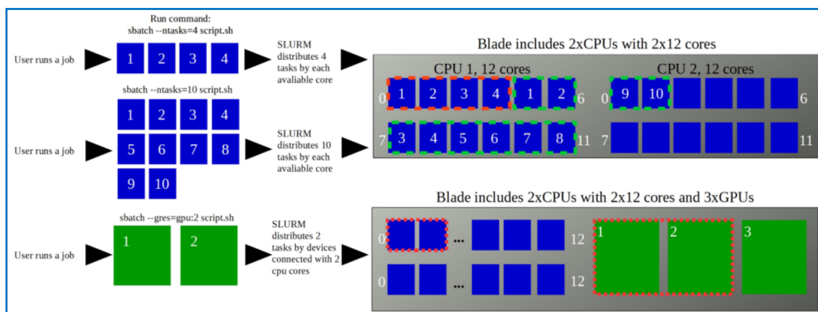


Рис. 1. Пример распределения ресурсов между пользовательскими задачами

Детальная информация о структуре и возможностях кластера, о решаемых задачах, о порядке доступа к ресурсам, а также инструкция по компиляции и запуску различных приложений представлена на сайте [1] (подробная информация о модульной системе,

доступных модулях и компиляторах представлена на сайте http://hybrilit.jinr.ru/how_to_work_on_cluster). В Приложении 1 представлены основные команды пакета *Modules* и планировщика *SLURM*.

В данной главе кратко излагаются основные сведения, необходимые для компиляции и запуска MPI/C(C++)-приложений.

1.2. Работа с MPI-приложениями

1.2.1. Компиляция

Созданный файл с программным кодом необходимо скомпилировать. Для этого нужно загрузить соответствующий модуль и затем вызвать команду компиляции.

Для компиляции MPI/C(C++)-приложения нужно воспользоваться одной из команд для компилятора `openmpi` или для компилятора `Intel` (табл. 1).

Таблица 1. **Команды компиляции, используемые в компиляторах**

Компилятор	Язык программирования	Команды вызова компилятора
OpenMPI	C	**`mpicc`**
	C++	**`mpiCC / mpic++ / mpicxx`**
Intel	C	**`mpiicc`**
	C++	**`mpiicpc`**

Например, загрузка `Intel`-модуля и компиляция файла с именем `mpi_test.c` осуществляются командами:

```
$ module add hlit/intel/v2019.0.018
$ mpiicc mpi_test.c
```

ВАЖНО! Версия Intel-модуля, указанная в примере, может не совпадать с версией, установленной в данный момент на кластере. Чтобы проверить текущую версию модуля, необходимо вывести при помощи команды *module avail* список всех доступных на кластере модулей и выбрать для загрузки нужную версию.

Если компилятор обнаружил ошибки в программе, выдается соответствующая диагностика. Необходимо в режиме редактирования исправить ошибки в программе, сохранить файл с программой и снова запустить команду компиляции. При отсутствии ошибок создается исполняемый модуль, который по умолчанию имеет имя `a.out`.

1.2.2. Запуск в *batch*-режиме

Пользователю необходимо создать специальный скрипт-файл, с помощью которого будет осуществляться запуск задачи на счет.

Пример простейшего скрипт-файла для запуска MPI-задачи:

```
!/bin/sh
#задание входной очереди задач для MPI-программ:
#SBATCH -p cpu
#выделение нужного числа процессоров (ядер), равного
#числу параллельных процессов (здесь задано 4
#параллельных процесса):
#SBATCH -n 4
#установка ограничения на время выполнения расчета
# (здесь задано время счета, равное 1 минуте):
#SBATCH -t 1
#запуск на счет исполняемого MPI-модуля с именем
#a.out:
mpirun ./a.out
```

Для запуска задачи на счет с помощью скрипт-файла с именем `mpi_script.sh` необходимо использовать команду `sbatch`:

```
    sbatch mpi_script
```

После этого на экране появится индивидуальный номер задачи. Результаты выполнения задачи будут находиться в файле с именем `slurm-<id>.out`, где `<id>` – индивидуальный номер задачи, полученный при запуске.

Просмотр *slurm*-файла можно осуществить в режиме редактирования либо с помощью команды просмотра файлов `cat`. Например, команда

```
    cat slurm-3333.out
```

выводит на экран монитора содержимое файла `slurm-3333.out.`

Глава 2. MPI: общие принципы, базовые функции

2.1. Логическое представление MPI-программы

Согласно классическому определению алгоритм представляет собой последовательность действий, выполняемых в строгой очередности. В обыденной жизни подобное определение напоминает, например, принцип действия пулемета: патроны, расположенные в ленте по одному, в строгой последовательности поступают в подаватель, затем извлекаются из ленты и досылаются в ствол для выстрела. Скорость работы в данном случае напрямую зависит от характеристик устройства, которое занимается обработкой. И если для пулеметов высокая скорострельность – это скорее недостаток, чем достоинство, то для компьютерного процессора скорость работы является определяющим фактором.

Прирост мощности современных процессоров достигается все более трудно, и по этой причине архитектуры вычислительных систем стали многопроцессорными и многоядерными. И тут встает вопрос о возможности программы работать в параллельном режиме.

Независимо от того, на каком реальном вычислительном устройстве выполняется MPI-программа, эта технология реализует модель распределенной памяти: предполагается, что MPI-программа выполняется группой так называемых MPI-процессов, каждый из которых имеет свою локальную память. **Все операции в MPI-программе выполняются параллельно всеми MPI-процессами при использовании каждым MPI-процессом данных из собственной локальной памяти.** Получение данных из локальной памяти других процессов организуется явно с помощью специальных процедур обмена, которые и составляют ядро технологии MPI.

Таким образом, если мы сравнивали выполнение последовательной программы с работой пулемета, то работа программы, написанной с использованием технологии MPI, может ассоциироваться с одновременной работой нескольких пулеметов при условии, что ставится задача отстрелять заданное количество лент с патронами.

2.2. MPI-функции общего назначения

2.2.1. Объявление и закрытие параллельной области

Работа с технологией MPI начинается с необходимости подключить соответствующую библиотеку в коде программы. Для языка С это библиотека `mpi.h`. Данная библиотека подключается при помощи команды:

```
#include <mpi.h>.
```

Важной частью работы программы является организация так называемой параллельной области, в которую должны быть заключены все MPI-функции. Чтобы инициализировать параллельную область, нужно воспользоваться функцией `MPI_Init`. В результате выполнения функции создается группа процессов, количество которых было указано при запуске задачи, а также область связи, описываемая коммуникатором с предопределенным именем `MPI_COMM_WORLD`. Функции `MPI_Init` передаются аргументы функции `main`, полученные из командной строки. Таким образом, функция имеет вид:

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

После выполнения всех MPI-функций необходимо закрыть параллельный блок. За это отвечает функция `MPI_Finalize`. Она закрывает все MPI-процессы и уничтожает все области связи. Использование функций MPI после `MPI_Finalize` невозможно. Функция имеет вид:

```
MPI_Finalize().
```

Таким образом, общая схема MPI-программы имеет вид:

```
#include <mpi.h>
………………………………
int main(int argc, char* argv[]){
………………………………
 MPI_Init(&argc,&argv);
 ……………………………
 ……………………………
 MPI_Finalize();
}
```

2.2.2. Понятия о группе и коммуникаторе

В ходе работы параллельной программы неизбежно возникает необходимость в передаче сообщений другим процессам. Но как процесс, передающий сообщение, может обратиться к другому процессу? Для этого все процессы, запрашиваемые у системы, объединяются в группу и между ними устанавливается специальная область связи, за которую отвечает коммуникатор. Таким образом, группой можно назвать некоторое множество процессов, а коммуникатор – объектом, «отвечающим» за связь между этими процессами.

Как уже сказано, по умолчанию система создает группу процессов, количество которых равно числу запрашиваемых через скрипт для запуска, и создает коммуникатор `MPI_COMM_WORLD`.

Другие предопределенные имена коммуникаторов:

– `MPI_COMM_WORLD` – коммуникатор, объединяющий все процессы;

– `MPI_COMM_NULL` – ошибочный коммуникатор;

– `MPI_COMM_SELF` – коммуникатор включает только вызвавший процесс.

Примечание. В MPI существует целый набор функций для организации подгрупп в рамках имеющейся группы и для создания коммуникаторов.

После того как группа определена, каждый процесс, состоящий в группе, может запросить свой порядковый номер в этой группе и общее число процессов в данной группе.

Запрос порядкового номера производится посредством обращения к функции:

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

В качестве параметров функция принимает объект коммуникатора группы, в которой состоит процесс, и указатель на целочисленную переменную, которой будет присвоен порядковый номер процесса.

Запрос количества процессов в группе происходит по аналогичной схеме. За это отвечает функция

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

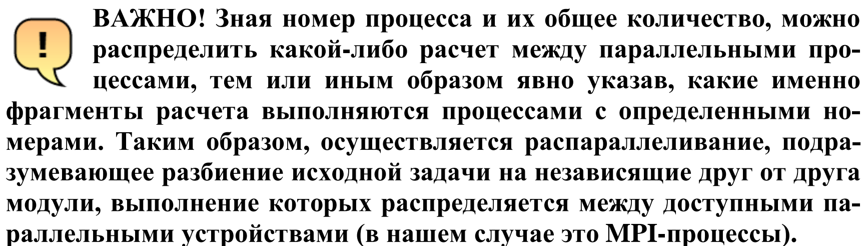
Параметры данной функции аналогичны параметрам функции `MPI_Comm_rank` – это объект коммуникатора группы и указа-

тель на целочисленную переменную, которой будет присвоено число процессов.

В примере представлена программа, в которой каждый MPI-процесс определяет свой порядковый номер и число процессов в группе и выводит на экран эти значения.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[]){
 int myid, numprocs;
 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
 MPI_Comm_rank(MPI_COMM_WORLD,&myid);
// Print number of processes in group and rank of current MPI process
 printf("Numprocs is %d; Hello from %d\n",numprocs, myid);
 MPI_Finalize();
 return 0;
}
```

Примечание. Помимо вышеописанных четырех MPI-функций к процедурам общего назначения относятся функции засечек времени, определения физического имени устройства и ряд других.



ВАЖНО! Зная номер процесса и их общее количество, можно распределить какой-либо расчет между параллельными процессами, тем или иным образом явно указав, какие именно фрагменты расчета выполняются процессами с определенными номерами. Таким образом, осуществляется распараллеливание, подразумевающее разбиение исходной задачи на независящие друг от друга модули, выполнение которых распределяется между доступными параллельными устройствами (в нашем случае это MPI-процессы).

2.2.3. Примеры практических заданий

1. Организовать вывод на экран каждым процессом своего номера; процесс с номером 0 также должен вывести количество процессов в группе.

2. Организовать вывод на экран процессом с максимальным номером в группе его номера и количества процессов в группе.
3. Организовать вывод на экран слова *Hello* процессами с четными номерами.

2.3. Пример распределения вычислений между параллельными MPI-процессами

Рассмотрим, как можно осуществить распараллеливание на примере цикла по умножению каждого элемента массива `A` на 10. Можно выделить два типа распределения расчета между параллельными процессами – блочное и циклическое.

Блочный принцип распределения (рис. 2) предполагает известной конечную длину обрабатываемых данных, что в свою очередь позволяет узнать размер блока, предназначенного для обработки каждым процессом.

Пусть мы имеем целочисленный массив `A` длины `N` и `np` MPI-процессов. Необходимо узнать размер блока, обрабатываемого каждым процессом, а также начальный и конечный индексы этих блоков. Пусть номер процесса хранится в переменной `rank`, начальные и конечные индексы блоков – в `imin` и `imax` соответственно. Нижеприведенный фрагмент программы демонстрирует один из вариантов реализации блочного распределения выполнения цикла.

```
int delta = N/np;
int ost = N%np;
imin = rank*delta;
imax = (rank+1)*delta;
if(rank== np-1) imax += ost;
 for(i=imin;i<imax;i++)
       A[i] *= 10;
```

Здесь каждому процессу назначается к обработке фрагмент массива `A` длины `delta`, начинающийся с позиции `imin`. В случае, если `N` не делится нацело на `np`, процесс с номером `np-1` дополнительно вычисляет `ost` элементов массива `A` (рис. 3).

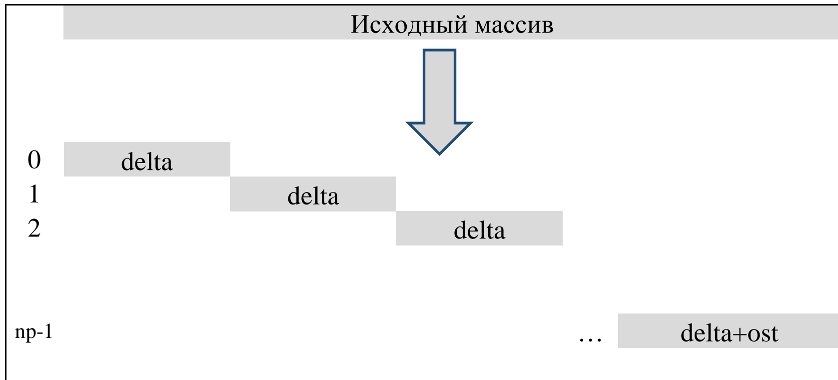


Рис. 2. Блочный тип распараллеливания

Рис. 3. Циклический тип распараллеливания

Другой вариант распределения расчета – циклическое (или итеративное) распределение, заключается в том, что каждый процесс обрабатывает итерации цикла, начиная со своего порядкового номера с шагом, равным числу процессов в группе (см. рис. 3).

Реализация такого распределения демонстрируется на вышеприведенном примере цикла по умножению элементов массива `A` на 10 и имеет следующий вид:

```
    for(i=rank;i<N;i+=np)
        A[i]*=10;
```

2.4. Обмены между отдельными MPI-процессами

2.4.1. Блокирующая передача сообщений

После того как процессы в группе могут позиционировать себя относительно других, можно приступить к передаче сообщений между процессами.

В MPI реализованы возможности как обменов между отдельными процессами (типа «точка – точка»), так и коллективных обменов, в которых участвуют все процессы группы.

Передача сообщений «точка – точка» является основным видом обмена данными между процессами и представляет собой специальный двунаправленный интерфейс. Для начала рассмотрим **блокирующую пересылку сообщения**. Блокирующей она называется потому, что возврат из соответствующих процедур приема\передачи блокируется до возможности корректного доступа к данным буфера приема\передачи. В случае операции приема это означает, что работа принимающего процесса блокируется до тех пор, пока сообщение не будет принято. Для процесса-отправителя выход из блокировки означает, что данные буфера передачи уже отправлены и можно безопасно пользоваться этой областью памяти.

Для организации такой пересылки необходим согласованный вызов процедуры отправки процессом-отправителем и процедуры приема процессом-получателем.

Процессу, который должен послать сообщение, необходимо воспользоваться функцией `MPI_Send`. Данная функция формирует буфер сообщения и отправляет его процессу-адресату. Функция имеет следующие параметры:

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

где `buf` – адрес начала пересылаемых данных;
`count` – число пересылаемых элементов;
`datatype` – тип пересылаемых данных;
`dest` – номер процесса-получателя в группе, которой соответствует коммуникатор `comm`;
`tag` – идентификатор сообщения;
`comm` – коммуникатор, связанный с группой процессов, в рамках которой идет пересылка.

Для приема сообщения процесс-адресат должен воспользоваться функцией MPI_Recv, которая имеет следующий вид:

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

где buf – адрес начала расположения принимаемых данных;
count – максимальное число принимаемых элементов;
datatype – тип принимаемых данных;
source – номер процесса-отправителя в группе, которой соответствует коммуникатор comm;
tag – идентификатор сообщения;
comm – коммуникатор, связанный с группой процессов, в рамках которой идет пересылка;
status – выходной параметр, атрибуты принятого сообщения.

Для инициализации переменной status используется тип данных MPI_Status.

ВАЖНО!!! Операции типа «точка – точка» подразумевают взаимодействие для передачи сообщения только двух процессов: процесса-отправителя и процесса-получателя. По этой причине вызов процедур приема и передачи конкретного сообщения должен быть организован только этими процессами, при этом идентификатор сообщения в вызванных процедурах отправки и получения должен совпадать.

Ниже приведен пример программы, передающей значение целочисленной переменной a=3 из процесса с номером 0 в процесс с номером 1 в группе, соответствующей коммуникатору comm. Принимающий процесс записывает полученное значение в переменную b.

```
#include <mpi.h>
#define comm MPI_COMM_WORLD
int main(int argc, char* argv[]){
 int rank;
 MPI_Status status;
 MPI_Init(&argc,&argv);
```

```
      MPI_Comm_rank(comm,&rank);
      if (rank==0){
            int a;
            a = 3;
            int i;
            MPI_Send(&a,1,MPI_INT,1,55,comm);
      }
      if(rank==1){
            int b;
            MPI_Recv(&b,1,MPI_INT,0,55,comm,
&status);
     }
      MPI_Finalize();
      return 0;
     }
```

2.4.2. Неблокирующая передача сообщений

Еще одним типом пересылки является **неблокирующая пересылка**, которая характеризуется тем, что процесс-отправитель и процесс-получатель сообщения осуществляют немедленный возврат из соответствующих MPI-функций и продолжают свою работу, не заботясь о том, отправлено ли (получено ли) сообщение.

Чтобы воспользоваться неблокирующей пересылкой, процессу-отправителю необходимо вызвать функцию `MPI_Isend`. Она имеет следующие параметры:

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

где `buf` – адрес начала пересылаемых данных;
`count` – число пересылаемых элементов;
`datatype` – тип пересылаемых данных;
`dest` – номер процесса-получателя в группе, соответствующей коммуникатору `comm`;
`tag` – идентификатор сообщения;
`comm` – коммуникатор, связанный с группой процессов, в рамках которой идет пересылка;
`request` – «запрос обмена».

Для организации неблокирующего приема используется функция MPI_Irecv, которую вызывает процесс-адресат. Она имеет следующие параметры:

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

где buf – адрес начала расположения принимаемых данных;
count – максимальное число принимаемых элементов;
datatype – тип принимаемых данных;
source – номер процесса-отправителя в группе, соответствующей коммуникатору comm;
tag – идентификатор сообщения;
comm – коммуникатор, связанный с группой процессов, в рамках которой идет пересылка;
request – «запрос обмена».

Определить окончание приема\передачи можно с помощью функций MPI_Wait или MPI_Test с соответствующим параметром request. Функция MPI_Wait блокирует дальнейшую работу вызвавшего ее процесса до успешного принятия\отправки данных, соответствующих параметру request. Функция MPI_Test выдает признак завершенности операции приема\передачи, соответствующей параметру request (flag=0, если процесс приема\передачи завершен, и flag=1, если не завершен). Процедуры имеют следующий вид:

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

где request – «запрос обмена»;
status – атрибут сообщения;

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

где request – «запрос обмена»;
flag – признак завершенности проверяемой операции;
status – атрибут сообщения.

Ниже приведен фрагмент программы, пересылающей с помощью функций MPI_Isend/MPI_Irecv значение 12 из целочис-

ленной переменной `a` в процессе с номером 0 в процесс с номером 1, где полученное значение записывается в переменную `b`.

```
………………………
int rank, iflag;
int a,b;
MPI_Status status;
MPI_Request send_request,recv_request;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
………………………
if (rank == 0) {
        a=12;
        MPI_Isend(&a,1,MPI_INT,1,100, MPI_COMM_WORLD,
&send_request);
        MPI_Test(&send_request,&iflag,&status);
        printf("iflag=%d\n",iflag);
}
If (rank == 1) {
        b=0;
        MPI_Irecv(&b,1,MPI_INT,0,100, MPI_COMM_WORLD,
&recv_request);
        MPI_Wait(&recv_request,&status);
        printf("b=%d\n",b);
}………………………
```

2.4.3. Двунаправленный обмен

В MPI существует возможность **двунаправленной передачи сообщений** посредством функции `MPI_Sendrecv`. Функция имеет два блока параметров: первый отвечает за отправку сообщения из вызывающего процесса в процесс с указанным номером; второй – за получение вызывающим процессом сообщения от процесса с указанным номером.

```
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 `sendbuf` – адрес начала пересылаемых данных;
`sendcount` – число пересылаемых элементов;
`sendtype` – тип пересылаемых данных;
`dest` – номер процесса-получателя в группе, которой подчинен коммуникатор `comm`;
`sendtag` – идентификатор посылаемого сообщения;
`recvbuf` – адрес начала расположения принимаемых данных;
`recvcount` – максимальное число принимаемых элементов;
`recvdatatype` – тип принимаемых данных;
`source` – номер процесса-отправителя в группе, которой подчинен коммуникатор `comm`;
`recvtag` – идентификатор принимаемого сообщения;
`comm` – коммуникатор, связанный с группой процессов, в рамках которой идет пересылка;
`status` – атрибуты принятого сообщения.

ВАЖНО!!! Буферы данных для приема и передачи сообщения в процедуре `MPI_Sendrecv` не должны совпадать.

Ниже приведен фрагмент программы, передающей с помощью согласованного вызова двух процедур `MPI_Sendrecv` значение целочисленной переменной `a=3` из процесса с номером 0 в процесс с номером 1 и значение целочисленной переменной `c = 11` из процесса с номером 1 в процесс 0.

```
    ..........................................
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Status status;
     ......................................
     if (rank==0) {
           int a, d;
           a = 3;
           d=0;
     MPI_Sendrecv(&a,1,MPI_INT,1,55,&d,1,MPI_INT,1,1
55, MPI_COMM_WORLD,&status);
     }
     if(rank==1) {
```

```
            int b, c;
            c=11;
            b=0;
      MPI_Sendrecv(&c,1,MPI_INT,0,155,&b,1,MPI_INT,0,
55, MPI_COMM_WORLD, &status);
     }
     ............................
```

В результате выполнения данного совмещенного обмена переменная d в процессе с номером 0 принимает значение 11, а переменная b в процессе с номером 1 – значение 3.

Все вышеописанные процедуры обмена полностью совместимы между собой и с другими процедурами обмена типа «точка – точка», существующими в MPI.

Это означает, что отправка сообщения может осуществляться, например, с помощью блокирующей процедуры MPI_Send, а получение – с помощью неблокирующей процедуры MPI_Irecv и т.п.

ВАЖНО!!! Идентификатор отправляемого сообщения в процессе-отправителе и идентификатор, указанный в процедуре приема этого сообщения, вызываемой процессом-получателем, должны совпадать независимо от используемых функций приема-передачи.

2.4.4. Примеры практических заданий

1. В процессе с номером 0 задать произвольный массив и отправить его процессу с максимальным номером в группе. Использовать блокирующую передачу данных.
2. В процессе с номером 0 задать произвольный массив и отправить его всем процессам в группе с ненулевыми номерами. Использовать блокирующую передачу данных.
3. В процессе с номером 0 задать произвольный массив и отправить его процессу с максимальным номером в группе. Использовать неблокирующую передачу данных. Выполнить проверку завершения неблокирующих процедур обмена.
4. В процессе с номером 0 задать произвольный массив и организовать его передачу каждому процессу с возрас-

тающими номерами по цепочке (0→1, 1→2, …, N–1→N). Использовать неблокирующую передачу данных.

5. Задать четное число процессов и организовать двустороннюю «зеркальную» передачу заданной переменной относительно центрального процесса (0↔N, 1↔N–1, 2↔N–2…). Использовать блокирующую передачу данных для отправки и неблокирующую передачу данных для приема сообщений.
6. Организовать взаимную пересылку двух массивов данных из двух процессов с помощью процедуры `MPI_Sendrecv`.

Глава 3. MPI-процедуры для коллективного взаимодействия процессов

3.1. Характерные особенности процедур коллективного взаимодействия процессов

Отличительные особенности коллективных операций, в которых участвуют все процессы в группе:

- коллективные коммуникации не взаимодействуют с процедурами типа «точка – точка»;
- процедуры коллективного взаимодействия должны быть согласованно вызваны всеми процессами в группе;
- процедуры коллективного взаимодействия выполняются в режиме с блокировкой; возврат из процедуры коллективного обмена в каждом процессе происходит тогда, когда его участие в коллективной операции завершилось, однако это не означает, что другие процессы также завершили операцию;
- количество получаемых данных должно быть равно количеству посланных данных;
- типы элементов посылаемых и получаемых сообщений должны совпадать;
- сообщения не имеют идентификаторов.

3.2. Барьерная синхронизация всех процессов

Иногда в параллельном приложении появляется необходимость в синхронизации. Одним из способов синхронизации процессов является так называемый барьер. Соответствующие процедуры имеются практически во всех технологиях параллельного программирования. Смыслом барьера является создание «преграды», которую процессы, состоящие в группе, могут преодолеть только вместе. Таким образом, процесс, выполнивший данную функцию, будет «заморожен» до тех пор, пока все процессы из этой группы не выполнят аналогичную функцию. Функция имеет следующие параметры:

```
int MPI_Barrier (comm),
```

где `comm` – коммуникатор, связанный с группой процессов, в рамках которой происходит синхронизация.

3.3. Коллективные коммуникационные операции

3.3.1. Рассылка информации от одного процесса всем остальным членам группы

Функция MPI_Bcast производит широковещательную рассылку данных от одного процесса всем процессам в группе, что сокращает как объем кода, так и время выполнения рассылки по сравнению с использованием для этой цели процедур типа «точка – точка». Функция имеет следующие параметры:

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

где buf – адрес начала расположения отправляемых данных;
count – количество отправляемых данных;
datatype – тип отправляемых данных;
root – номер процесса-отправителя;
comm – коммуникатор, связанный с группой процессов, в рамках которой происходит операция.

Графическая интерпретация функции MPI_Bcast представлена на рис. 4.

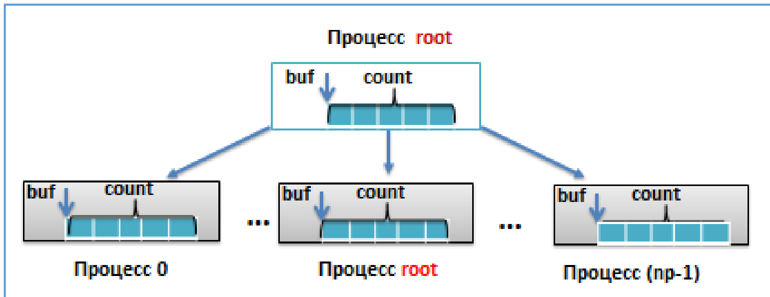


Рис. 4. Схема рассылки данных от одного процесса (root) всем процессам в группе: функция MPI_Bcast, np – количество процессов в группе

Фрагмент программы ниже иллюстрирует рассылку значения переменной a типа double из процесса 0 всем процессам в группе, соответствующей коммуникатору MPI_COMM_WORLD.

```
…
int rank;
double a;
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if(rank==0) a = 0.001;
MPI_Bcast(&a,1,MPI_DOUBLE,0, MPI_COMM_WORLD);
….
```

3.3.2. Рассылка фрагментов массива данных всем процессам группы

Очень часто при распараллеливании программы требуется распределение данных между параллельными процессами. В терминологии технологии MPI каждый процесс должен получить сообщение со своей частью данных, чтобы произвести над ними дальнейшие вычисления. Описанная выше функция `MPI_Bcast` позволяет разослать идентичные данные всем процессам, однако при большом объеме рассылки данная операция может серьезно замедлить выполнение программы за счет повышения объема взаимодействия между процессами. Предположим, нам необходимо распределить между k процессами массив длины N, каждый элемент которого занимает m байт. При использовании функции `MPI_Bcast` через сеть будет переправлено $N*m*(k-1)$ байт данных. Как уже было сказано, на большом объеме данных это может вылиться в существенную задержку времени вычисления, не говоря уже об увеличении используемой программой оперативной памяти. К тому же для последующей обработки зачастую необходима лишь часть переданных данных.

Решением данной проблемы логично представляется организация пересылки только тех данных, которые впоследствии будут обрабатываться конкретным процессом. Такая функция в MPI существует и называется `MPI_Scatter`. Функция распределяет данные между процессами согласно их порядковым номерам (т.е. процесс с номером 0 получит первую порцию данных, процесс 1 – следующую порцию и т.д.) и выглядит так:

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

где sendbuf – адрес начала расположения распределяемых данных;
sendcount – количество передаваемых данных;
sendtype – тип распределяемых данных;
recvbuf – адрес начала расположения принимаемых данных;
recvcount – количество принимаемых данных;
recvtype – тип распределяемых данных;
root – номер процесса-отправителя;
comm – коммуникатор группы.

Графическая интерпретация функции MPI_Scatter представлена на рис. 5.

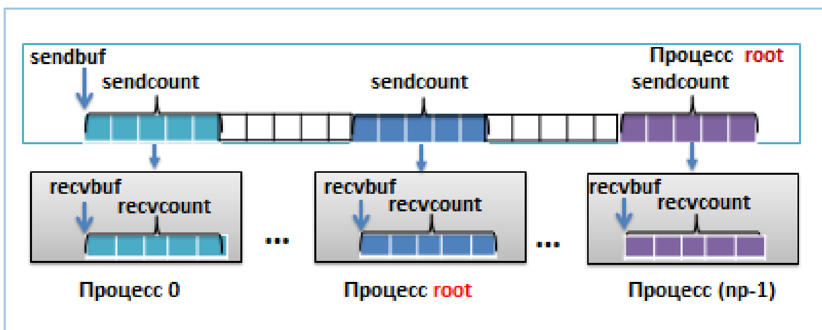


Рис. 5. Схема рассылки фрагментов массива всем процессам согласно их порядковым номерам: функция MPI_Scatter, np – количество процессов в группе

Приведенный ниже фрагмент программы иллюстрирует упорядоченную рассылку по всем MPI-процессам в группе фрагментов длины delta массива a, находящегося в MPI-процессе с номером 0, с записью полученных данных в подмассивы a1 длины delta.

```
…
int rank, size;
int N; //array size
double *a;
double *a1;
a=(double*)malloc(N*sizeof(double));
…
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    int delta = N/size;
    a1=(double*)malloc((delta)*sizeof(double));
    MPI_Scatter(&a[0],delta,MPI_DOUBLE,&a1[0],delta,
MPI_DOUBLE,0,MPI_COMM_WORLD);
    …
```

Примечание. Функция `MPI_Scatter` подразумевает распределение между процессами равных порций данных. Такое распределение не всегда является оптимальным. Организовать рассылку данных произвольными порциями можно с помощью функции `MPI_Scatterv`. В список ее параметров вводятся дополнительно так называемые массивы порций и смещений, имеющие размерность, равную числу процессов в группе. Эти целочисленные массивы содержат размеры порций, предназначенных к рассылке каждому процессу, и номера элементов исходного массива, с которых будет начинаться считывание данных для каждого из процессов.

3.3.3. Сборка распределенных по процессам данных в один массив

Очень часто после параллельной обработки данных появляется необходимость сбора данных в один процесс для финальной обработки, сохранения и\или последующего перераспределения данных. Для этого используется функция `MPI_Gather`, которая является обратной для функции `MPI_Scatter`. `MPI_Gather` собирает распределенные данные из всех процессов и объединяет их в одном root-процессе. Порядок расположения объединенных данных определяется порядковым номером процесса. Функция имеет вид:

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

где `sendbuf` – адрес начала расположения объединяемых данных;
`sendcount` – количество объединяемых данных;
`sendtype` – тип передаваемых данных;
`recvbuf` – адрес начала расположения объединенных данных;
`recvcount` – количество объединенных данных;
`recvtype` – тип объединенных данных;
`root` – номер процесса, в котором происходит объединение данных;

`comm` – коммуникатор группы.

Ниже приведен фрагмент программы, иллюстрирующий упорядоченное объединение подмассивов `c1` размерностью `delta`, расположенных в разных процессах, в массив `c`, находящийся в процессе с номером 0 группы, ассоциированной с коммуникатором `MPI_COMM_WORLD`.

```
     int rank, size;
     int N; //array size
     double *c;
     double *c1;
     c=(double*)malloc(N*sizeof(double));
     …
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     int delta = N/size;
     c1=(double*)malloc((delta)*sizeof(double));
     …
     MPI_Gather(&c1[0],  delta,  MPI_DOUBLE,  &c[0],
delta, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      …
```

Примечание. Также существует функция `MPI_Gatherv`, обратная `MPI_Scatterv`. Эта функция объединяет фрагменты различной длины из всех процессов в один. Существуют также функции `MPI_Allgather` и `MPI_Allgatherv`, позволяющие не только объединить данные из всех процессов, но и отправить в каждый процесс полную копию собранных данных. Функция `MPI_Alltoall` совмещает в себе операции Scatter и Gather и является по сути расширением операции Allgather, когда каждый процесс посылает различные данные разным получателям.

3.3.4. Глобальные вычислительные операции над данными, расположенными в адресных пространствах различных процессов

В параллельном программировании на базе MPI допускаются математические операции над блоками данных, распределенных между MPI-процессами. Такие операции называют глобальными операциями редукции. Например, пусть параллельная программа

на определенном этапе хранит в каждом процессе частичную сумму какой-либо величины. Применение операции глобальной редукции MPI_SUM позволит получить окончательное значение данной величины без организации пересылки этих частичных сумм в один процесс, что существенно упрощает написание программы. Использование подобных операций является одним из важных инструментов организации распределенных вычислений.

Процедура MPI_Reduce возвращает результат глобальной операции (редукции) в один процесс с указанным номером. Функция имеет вид:

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

где sendbuf – адрес начала расположения буфера данных, над которыми выполняется глобальная операция op;
recvbuf – адрес начала расположения в процессе с номером root буфера данных с результатами выполнения операции op;
count – количество данных;
datatype – тип данных;
op – операция редукции, выполняемая над данными sendbuf;
root – номер процесса, в который записываются результаты операции op;
comm – коммуникатор группы.

В MPI определены ряд глобальных операций и типов данных, над которыми они могут выполняться (табл. 2).

Таблица 2. **Операции редукции (глобальные операции) и типы данных, над которыми разрешены эти операции**

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

Фрагмент программы, представленный ниже, иллюстрирует суммирование с помощью процедуры `MPI_Reduce` значений, находящихся в разных процессах переменных `c` типа `double` в переменную `c1`, находящуюся в процессе с номером 0.

```
    …
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    double a = 0.01*rank;
    MPI_Reduce(&a,&b,1,MPI_DOUBLE,MPI_SUM,0,
MPI_COMM_WORLD);
    …
```

Графическая интерпретация этого примера представлена на рис. 6.

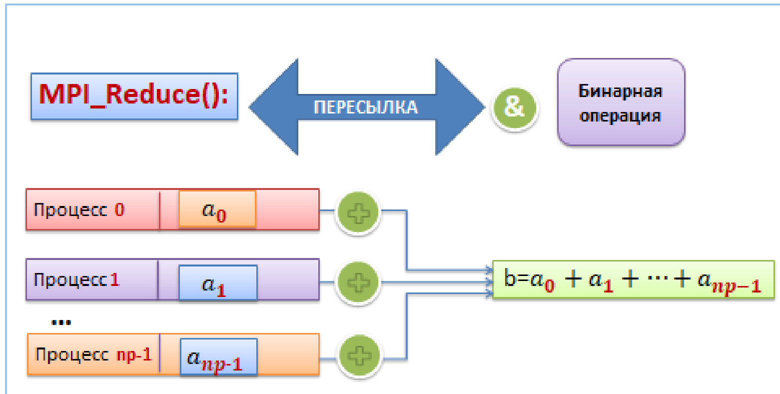


Рис. 6. Схема операции суммирования чисел *a* (на рисунке добавлен индекс – номер процесса), находящихся на разных процессах: функция `MPI_Reduce`, `np` – количество процессов в группе

В отличие от функции `MPI_Reduce`, функция `MPI_Allreduce` возвращает результат редукции всем процессам. Функция имеет вид:

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

где `sendbuf` – адрес начала расположения буфера данных, над которыми выполняется глобальная операция `op`;
`recvbuf` – адрес начала расположения в каждом процессе группы, соответствующей коммуникатору `comm`, буфера данных с результатами выполнения операции `op`;
`count` – количество данных;
`datatype` – тип данных;
`op` – операция редукции, выполняемая над данными `sendbuf`;
`comm` – коммуникатор группы.

3.4. Примеры практических заданий

1. В процессе с максимальным номером в группе задать произвольный массив и разослать его всем процессам в группе с помощью процедуры `MPI_Bcast`.

2. Задать в каждом процессе произвольные массивы равной длины и найти максимальное (минимальное) значение среди элементов с номерами 0, 1, 2, … массивов, находящихся в разных процессах с помощью функции `MPI_Reduce`.
3. Задать в каждом процессе целочисленные переменные, значение которых равно удвоенному номеру процесса. Найти сумму этих элементов с записью результатов во все процессы группы. Использовать функцию `MPI_Allreduce`.
4. Задать произвольный массив в процессе с номером 0. Организовать его рассылку и распределенное поэлементное сложение (перемножение) всех элементов этого массива с использованием функций `MPI_Bcast` и `MPI_Reduce`.
5. Задать произвольный массив в процессе с номером 0. Организовать его рассылку и распределенное поэлементное сложение (перемножение) всех элементов массива с использованием функций `MPI_Scatter` и `MPI_Reduce`.
6. Задать произвольные массивы в каждом процессе. Организовать их сборку в объединенный массив в процессе с номером 1 с помощью процедуры `MPI_Gather`.

Глава 4. Эффективность параллельных вычислений

Материал предыдущих глав направлен главным образом на демонстрацию возможностей технологии MPI и овладение навыками ее применения как для создания параллельной MPI-программы «с нуля», так и для преобразования уже имеющейся последовательной программы в параллельную. При этом, как и в случае написания последовательной программы, перед программистом встает вопрос об эффективности разработанного параллельного кода, наиболее важными составляющими которой представляются:

- эффективность выполнения последовательного кода на каждом из параллельных процессов;
- эффективность параллельной реализации, включающей в нотациях технологии MPI методику разбиения данных, способы передачи сообщений, синхронизацию MPI-процессов и многое другое;
- влияние на эффективность параллельных вычислений особенностей алгоритма, реализуемого в разрабатываемой программе.

В данной главе мы остановимся на двух последних аспектах этого списка. В параграфе 4.1 анализируется эффективность применения различных процедур MPI для организации параллельных вычислений. Параграф 4.2 посвящен априорному анализу эффективности параллельной реализации алгоритмов на основе закона Амдала.

4.1. Анализ эффективности применения некоторых MPI-функций

Оценить эффективность MPI-программы можно посредством замера времени ее выполнения с использованием процедуры `MPI_Wtime`. Для этого необходимо создать две переменные типа `double`, которые будут отвечать за хранение значения времени перед началом выполнения параллельного фрагмента и после завершения его выполнения. Значение времени определяется MPI-функцией `MPI_Wtime`.

Пример использования этой функции:

```
..............................................................................
double start, finish, result_time;
..............................................................................
start= MPI_Wtime();
..............................................................................
finish = MPI_Wtime();
result_time= finish – start;
..............................................................................
```

Приведем примеры различных MPI-реализаций для выполнения одного и того же действия и сравним их эффективность.

4.1.1. Пример 1. Рассылка данных всем процессам

Предположим в качестве первого примера, что перед программистом стоит задача разослать всем процессам целочисленный массив `a` размерности `n`, находящийся в MPI-процессе с номером 0 в группе, соответствующей коммуникатору `comm`. Такую рассылку можно осуществить с помощью процедур обмена между отдельными процессами, с использованием цикла в нулевом процессе-отправителе для поочередной отправки сообщения всем остальным процессам в группе. При этом каждый ненулевой процесс должен вызвать соответствующую процедуру приема. Так, в терминах функций `MPI_Send/MPI_Recv` подобная рассылка имеет вид:

```
..............................................................................
if(rank=0){
for(j=1;j<size;j++){
             MPI_Send(&a[0],n,MPI_INT,j,5,comm);
 }
}
if(rank!=0){
 MPI_Recv(&b[0],n,MPI_INT,0,5,comm, &status);
}
..............................................................................
```

Данная конструкция отлично справится со своей работой, однако стоит подумать, сколько ресурсов системы будет задействовано. Каждый вызов функции `MPI_Send` породит свой соб-

ственный буфер, через который будет осуществлена передача сообщения в соответствующий MPI-процесс.

В качестве альтернативы подобному подходу можно предложить использование коллективной операции `MPI_Bcast`, которая выполняет функционал приведенной выше конструкции с помощью строки.

```
……………………………………………………………………
MPI_Bcast(&a[0],n,MPI_INT,0,comm);
……………………………………………………………………
```

Таким образом, достигается как сокращение числа строк кода программы, так и сокращение издержек, порождаемых излишним вызовом функций пересылки. Убедиться, что рассылка данных с помощью процедуры `MPI_Bcast` является более эффективной с точки зрения затрат времени по сравнению с рассылкой на основе использования функций `MPI_Send`/`MPI_Recv`, можно путем засечек времени выполнения рассылки с применением процедуры `MPI_Wtime`. Предоставляем читателю сделать это самостоятельно.

4.1.2. Пример 2. Поэлементное перемножение векторов

Немаловажной частью MPI-программы является объем передаваемых данных в тех самых сообщениях, которые являются основой технологии MPI.

Для следующего примера возьмем программу, которая будет поэлементно перемножать два вектора длины N, тип данных которых определен как `double`. На первый взгляд задача отлично распараллеливается, поэтому можно распределить данные между N MPI-процессами и произвести соответствующие вычисления, а затем объединить данные в одном процессе для представления результатов. Примеры распараллеливания циклов рассмотрены в параграфе 2.3. Здесь мы остановимся на вариантах реализации (а) рассылки данных перед вычислениями и (б) сборки результатов перемножения в один процесс.

Разберем *первый вариант* программной реализации этой задачи с использованием функций `MPI_Bcast` для стартовой рассылки данных и `MPI_Reduce` для сборки результатов. Ниже приведен первый фрагмент кода, реализующий:

- распределение данных, изначально заданных в процессе с номером 0 группы, ассоциированной с коммуникатором `comm`,
- параллельное выполнение цикла всеми процессами,
- формирование массива с результатами счета в процессе с номером 0:

```
    ………………………………………………………………
    // Broadcasting arrays a and b to all processes
    MPI_Bcast(&a[0],N,MPI_DOUBLE,0,comm);
    MPI_Bcast(&b[0],N,MPI_DOUBLE,0,comm);

    // Bounds of blocks for each process
    delta=N/size;
    ost=N%size;
    imin= rank*delta;
    imax = (rank+1)*delta;
    if(rank== np-1){
    imax= N;
    }
    // Main loop
    for(i=imin;i<imax;i++){
    c[i]=a[i]*b[i];
    }
    // Formation of final result in 0-process
    MPI_Reduce(&c[imin],    &c1[imin],    imax-imin,
MPI_DOUBLE, MPI_SUM, 0, comm);
    ………………………………………………………………
```

Представленный фрагмент прекрасно справляется со своей задачей, но давайте представим, сколько данных передается по сети. Одно число типа `double` занимает в памяти 8 байт. Функция `MPI_Bcast` копирует вектора в каждый из MPI-процессов (за исключением процесса-источника данных). Таким образом, объем передачи будет равен 8*N*(size-1) байт. Таких векторов нужно передать два, а значит, 2*(8*N*(size-1)) байт. Что касается объединения данных, то здесь объем передачи составит 8*N*size байт.

Для векторов размерностью в 1 миллион элементов, распределенных между четырьмя MPI-процессами, общий объем пересы-

лок при выполнении вышеприведенного фрагмента программы составит 60 мегабайт. Для современных компьютеров объем небольшой, тем не менее для его передачи требуется и время, и ресурсы системы.

Для уменьшения объема пересылок можно использовать функцию распределения MPI_Scatter (см. параграф 3.3.2) и функцию сбора MPI_Gather (см. параграф 3.3.3), обеспечивающих пересылку не целых массивов, а их фрагментов заданной длины. Однако объем передаваемых этими функциями фрагментов фиксирован, поэтому остаток данных должен быть передан отдельно. Для этого подойдут классические пересылки типа «точка – точка», например реализованные с помощью функций MPI_Send и MPI_Recv. Приведем фрагмент кода, демонстрирующий *второй вариант* реализации задачи о поэлементном перемножении двух массивов на основе вышеописанной схемы рассылки и сборки фрагментов массивов в рамках нашей задачи с помощью указанных функций.

```
    // Bounds of blocks for each process
    delta=N/np;
    ost=N%np;
    imin= rank*delta;
    imax = (rank+1)*delta;
    if(rank== np-1){
    imax= N;
    }
    //scattering main parts of arrays
    MPI_Scatter(&a [0], delta, MPI_DOUBLE, &a1[0],
delta, MPI_DOUBLE, 0, comm);
    MPI_Scatter(&b[0], delta, MPI_DOUBLE, &b1[0],
delta, MPI_DOUBLE, 0, comm);
    // sending the rest part of arrays
    if(ost!=0){
    if(rank==0){
    MPI_Send(&a[np*delta], ost, MPI_DOUBLE, np-1,
44, comm);
    MPI_Send(&b[np*delta], ost, MPI_DOUBLE, np-1,
45, comm);}
```

```
    if(rank==np-1){
    MPI_Recv(&a1[delta],ost,MPI_DOUBLE,0,44,comm,&
status);
    MPI_Recv(&b1[delta],ost,MPI_DOUBLE,0,45,comm,&
status);}
    }
    //-------------------------------
    // Main loop
    for(i=0;i<delta+ost;i++){
    c1[i]=a1[i]*b1[i];
    }
    //Gathering of final result
    MPI_Gather(&c1[0],  delta,  MPI_DOUBLE,  &c[0],
delta, MPI_DOUBLE, 0, comm);
    if (ost!=0){
    if(rank==np-1){
    MPI_Send(&c1[delta],  ost,  MPI_DOUBLE,  0,  55,
comm);
    }
    if(rank==0){
    MPI_Recv(&c[np*delta],  ost,  MPI_DOUBLE,  np-1,
55, comm, & status);
    }
    }
```

Третий вариант реализации подразумевает использование функций `MPI_Scatterv` и `MPI_Gatherv`. Как уже было отмечено в главе 3, данные функции предназначены соответственно для рассылки и сборки фрагментов различной длины. Естественно, в параметрах этих функций должна присутствовать информация о длине пересылаемых каждому процессу (или из каждого процесса) фрагментов. Эта информация содержится в специальном массиве.

Итак, функция `MPI_Scatterv`, предназначенная для коллективного распределения массива из процесса-источника фрагментами различной длины всем процессам в группе, имеет следующие параметры:

```
int  MPI_Scatterv(void*  sendbuf,  int*  sendcounts,
int* displs, MPI_Datatype sendtype, void* recvbuf,
```

```
int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm),
```

где `sendbuf` – адрес начала всего буфера распределяемых данных;
`sendcounts` – массив, содержащий количество передаваемых данных для каждого процесса;
`displs` – массив, содержащий номер первого элемента для каждого процесса;
`sendtype` – тип распределяемых данных;
`recvbuf` – адрес начала расположения принимаемых данных;
`recvcount` – количество принимаемых данных;
`recvtype` – тип распределяемых данных;
`root` – номер процесса-отправителя;
`comm` – коммуникатор группы.

Функция `MPI_Gatherv`, предназначенная для коллективной сборки массива в принимающем процессе из фрагментов различной длины, находящихся во всех процессах группы, имеет следующие параметры:

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

где `sendbuf` – адрес начала расположения объединяемых данных;
`sendcounts` – массив, содержащий количество объединяемых данных для каждого процесса;
`displs` – массив, содержащий номер первого элемента для каждого процесса;
`sendtype` – тип передаваемых данных;
`recvbuf` – адрес начала расположения объединенных данных;
`recvcount` – количество объединенных данных;
`recvtype` – тип объединенных данных;
`root` – номер процесса, в котором происходит объединение данных;
`comm` – коммуникатор группы.

Как уже сказано, данные функции имеют два специфических параметра, отсутствующих в функциях `MPI_Scatter` и `MPI_Gather`, рассмотренных в главе 3. Это массив, в котором содержатся разме-

ры порций для каждого MPI-процесса (chunks), и массив, содержащий номера элементов, с которых будет отсчитываться каждый пересылаемый фрагмент (disps). Ниже приведен фрагмент кода, демонстрирующий рассылку и сборку фрагментов с помощью указанных функций в рамках той же задачи о поэлементном перемножении двух векторов.

```
     ………………………………………………………………
     delta=N/size;
     for(int i= 0; i<np; i++){
     disps[i]= i*delta;
     if(i==(np-1))
      chunks[i]= N-((np-1)*delta);
     else
      chunks[i]= delta;
     }
     //memory allocation
     a1=(double*)malloc((chunks[rank])*sizeof(double)
);
     b1=(double*)malloc((chunks[rank])*sizeof(double)
);
     c1=(double*)malloc((chunks[rank])*sizeof(double)
);
     // Scattering of parts of input arrays
     MPI_Scatterv(a,  chunks,  disps,  MPI_DOUBLE,  a1,
chunks[rank], MPI_DOUBLE, 0, comm);
     MPI_Scatterv(b,  chunks,  disps,  MPI_DOUBLE,  b1,
chunks[rank], MPI_DOUBLE, 0, comm);

     // Main loop
     for(i=0;i<chunks[rank];i++){
     c1[i]=a1[i]*b1[i];
     }
     //Gathering of the results
     MPI_Gatherv(c1,  chunks[rank],  MPI_DOUBLE,  c,
chunks, disps,  MPI_DOUBLE, 0, comm);
     ………………………………………………………………
```

В двух последних вариантах программы входные массивы разбиваются на части и распределяются фрагментами между процессами. При сборке также происходит пересылка не целых массивов, а их фрагментов. Таким образом, при рассылке данных по MPI-процессам трафик составит всего $2*8*N$ байт, а при приеме – $8*N$ байт. Конкретно в примере с векторами длиной в 1 миллион элементов объем передачи составит 24 мегабайта (экономия более чем в 3 раза по сравнению с реализацией на основе `MPI_Bcast` и `MPI_Reduce`). Проверку, какой из вариантов является более экономичным с точки зрения затрат машинного времени, предоставляем читателям.

4.1.3. Примеры самостоятельных заданий

1. Создать программы, реализующие рассылку массива данных длины 100000 всем процессам, на основе функций (1) `MPI_Send`, `MPI_Recv`, (2) `MPI_Bcast`, (3) `MPI_Isend`, `MPI_Irecv`. Выполнить в каждом варианте замеры времени рассылки на основе функции `MPI_Wtime` при использовании 5 MPI-процессов.
2. Провести анализ времени выполнения вышеуказанных программ в зависимости от длины массива.
3. Провести анализ времени выполнения вышеуказанных программ в зависимости от количества задействованных MPI-процессов.

4.2. Оценка эффективности параллельной реализации алгоритмов и программ на основе закона Амдала

4.2.1 Формула Амдала для асимптотической оценки ускорения вычислений при параллельной реализации

Подавляющее большинство алгоритмов и программ включает в себя некоторый процент непараллельных операций, которые должны выполняться одним процессом. К таким операциям, например, относится финальная обработка результатов параллельного счета с записью в один процесс, стартовый ввод данных и др. К операциям, которые нельзя выполнить параллельно, относятся последовательности команд, имеющие зависимость по данным. Например, операции

$a = b+c$,

$\mathrm{d} = a-f$

обязаны выполняться последовательно, т.к. вычитание во второй строке не может быть выполнено, пока не вычислено значение **а** в первой операции. Для сравнения приведем пример пары операций, которые могут быть выполнены параллельно:

$a = b+c$,

$d = e+f$.

Эффективность реализации какой-либо программы (алгоритма) существенно зависит от количества подобных фрагментов. Закон Амдала описывает предел эффекта от параллельного выполнения какой-либо программы.

Пусть программа содержит n операций, из которых N операций нужно выполнять последовательно. Обозначим через $S = N/n$ долю выполняемых последовательно операций (рис. 7). Отметим, что $0 \leq S \leq 1$. Предельные случаи $S = 0$ и $S = 1$ соответствуют полностью параллельным и полностью последовательным программам. Пусть P – число параллельных вычислительных устройств, выполняющих нашу программу. Пусть τ – время выполнения одной операции на любом из этих устройств.

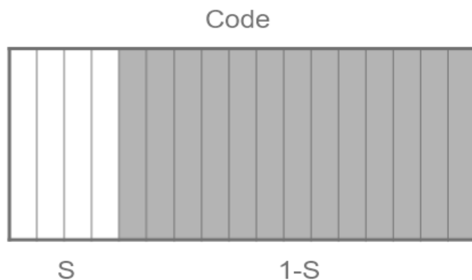


Рис. 7. Доли последовательных (S) и параллельных (1-S) операций в составе программы Code

Оценим ускорение вычислений – отношение времени выполнения программы на одном устройстве ко времени её выполнения на P устройствах.

Время реализации всей программы на одном устройстве составляет:

$$T_1 = n \cdot \tau.$$

Параллельная часть программы ($n–N$ операций) распределяется между P устройствами. Время реализации параллельной части на каждом из устройств равно:

$$T_{n-N} = [(n-N)/P] \cdot \tau.$$

N последовательных операций выполняются только на одном функциональном устройстве за время

$$T_N = N \cdot \tau.$$

Тогда общее время параллельной реализации программы (алгоритма) на P устройствах вычисляется как сумма T_N и T_{n-N}:

$$T_P = T_N + T_{n-N} = N \cdot \tau + [(n-N)/P] \cdot \tau.$$

Таким образом, получаем, что ускорение $A = T_1/T_P$ составляет

$$A = \frac{1}{S + \frac{1-S}{P}}.$$

Легко увидеть, что при $S \to 0$ и при фиксированном, достаточно малом P, имеем предел $A \to P$, т.е. при малой доле последовательных операций и относительно небольшом количестве вычислительных устройств можно ожидать роста A пропорционально числу задействованных параллельных вычислительных устройств.

С другой стороны, при фиксированном S и при $P \to \infty$ получаем предел $A \to 1/S$. Таким образом, прямое увеличение числа процессоров НЕ приводит к пропорциональному ускорению вычислений. При доле последовательных операций в программе 5% нельзя добиться ускорения более чем в 20 раз даже при использовании очень большого количества параллельных вычислительных узлов. Если же доля последовательных вычислений велика, то на значительное ускорение рассчитывать не приходится и нужно думать о модификации алгоритма.

Формулу Амдала применяют для асимптотического прогноза ускорения на основе теоретической оценки доли параллельных вычислений в программе. Формула не учитывает затрат времени на взаимодействие параллельных устройств. Поэтому на практике значения ускорения обычно получаются ниже теоретических. Отметим также, что эффективность работы конкретной программы

может различаться на разных вычислительных системах из-за особенностей их архитектуры и системного обеспечения. Поэтому перед выполнением массовых расчетов обычно проводят серию методических вычислений с целью прогнозирования ожидаемого времени счета и выбора оптимального количества необходимых параллельных вычислительных устройств.

4.2.2. Примеры самостоятельных заданий

1. Рассчитать на основе формулы Амдала, какого максимального ускорения можно достичь при параллельном выполнении программы, в которой доля последовательных операций составляет 1%.
2. Рассчитать на основе формулы Амдала, какого максимального ускорения можно достичь при параллельном выполнении программы, в которой доля параллельных операций составляет 50%.
3. Во сколько раз можно ускорить программу согласно закону Амдала при расчете на пяти параллельных устройствах, если дола последовательных операций составляет 1/10 (т.е. 10%)?

Список рекомендованной литературы

1. Гетерогенная вычислительная платформа HybriLIT. – Электрон. дан. – URL: http://hybrilit.jinr.ru.

2. Антонов, А. С. Технологии параллельного программирования MPI и OpenMP : учебное пособие / А.С. Антонов. – М. : Изд-во МГУ, 2012. – 344 с.

3. Воеводин, В. В. Параллельные вычисления : учебное пособие / В.В. Воеводин, В.В. Воеводин. – СПб : Изд-во БХВ-Петербург, 2015. – 603 с.

4. Лупин, С. А. Технологии параллельного программирования : учебное пособие / С.А. Лупин, М.А. Посыпкин. – М. : Форум, 2008. – 205 с.

5. Гергель, В. П. Современные языки и технологии параллельного программирования : учебник / В.П. Гергель. – М. : Изд-во Моск. ун-та, 2012. – 408 с.

Приложение 1. Основные команды Linux, редакторы vim и nano

Базовые команды **Linux**

Команда	Описание
Ls	выдать список файлов в текущем каталоге
ls -la	выдать подробный список, включая скрытые файлы
cd [каталог]	сменить текущий каталог. Если имя каталога не указывается, то текущим становится домашний каталог пользователя
cp *<что_копировать> <куда_копировать>*	копировать файлы
mv *<что_перемещать> <куда_перемещать>*	переместить или переименовать файл
ln -s *<на_что_сделать_ссылку> <имя_ссылки>*	создать символическую ссылку
rm *<файл(ы)>*	удалить файл(ы)
cat *<имя_файла>*	вывод содержимого файла на стандартный вывод (по умолчанию – на экран)
find *<каталог>* -name *имя_файла*	найти файл *имя_файла* и отобразить результат на экране. Поиск начинается с *<каталог>*
mc	запустить программу управления файлами MidnightCommander
pwd	вывести имя текущего каталога
ps -a	вывести список текущих процессов
<имя_файла> | grep *< фрагмент>*	поиск фрагмента текста в *<имя_файла>*
grep *<User_name>*	отобразить все процессы, запущенные от *User_name*
kill	принудительного завершения работы приложения (процесса)
killall *<имя_программы>*	принудительного завершения работы всех процессов по имени программы

Редактор **Vim**

Вызов:

vi *<имя_файла>* или **vim** *<имя_файла>*

:wq	записать файл и выйти из vi/vim
:w	сохранить изменения
:q!	выход без сохранения изменений
:x	выход с сохранением нового содержимого файла
ZZ	сохранить изменения и выйти
i	режим вставки
ESC	перейти в режим просмотра
I	добавление в начало строки
a	режим добавления
A	добавление в конец строки
dd	вырезать (удалить) строку
cc	удалить и начать редактирование
yy	копировать строку
:<№ строки>	перейти на строку №
:set number	включить нумерацию строк
:set nonumber	отключить нумерацию строк
u	отменить предыдущее изменение
U	восстановить первоначальный вид строки
p	вставить содержимое буфера или удаленные строки после текущей строки
P	поместить содержимое буфера или удаленные строки перед текущей строкой

Редактор **nano**

Вызов:

$ nano *<имя_файла>*

Команда	Описание
Ctrl-X	закрыть редактор
Ctrl-O	сохранить
Ctrl-C	номер строки\текущая позиция
Ctrl-W	поиск
Ctrl-W затем Ctrl-T	переход к строке №
Ctrl-K	вырезать строку
Ctrl-U	вставить из буфера
Alt-A	выделение текста: нажмите Alt+A (или Ctrl+6), затем установите курсор в конец текста, который нужно вырезать; отмеченный текст при этом выделяется
Alt-6	копировать в буфер

Приложение 2. Основные команды: Module и SLURM

Список основных команд для работы с пакетом Module

Команды Module	Описание
module avail	просмотреть список доступных модулей
module add <name of the module from the list>	загрузить модуль (добавить модуль в список подключенных)
module list	просмотреть список подключенных модулей
module rm <name of the module>	выгрузить модуль из списка подключенных

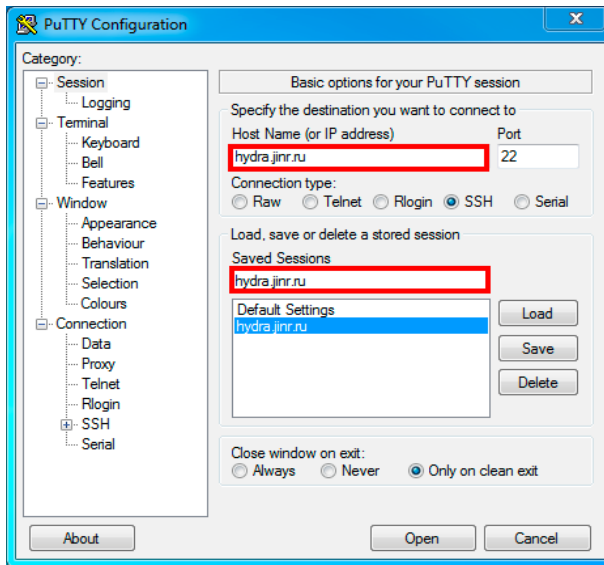
Список основных команд планировщика SLURM

Команды SLURM	Описание
sinfo	команда для просмотра состояния вычислительных узлов и очередей SLURM
squeue	команда для просмотра списка запущенных задач
sbatch < name of the submission script >	команда для запуска за в режиме очереди: отправляет задачу для последующего выполнения (**script** обычно содержит одну или несколько команд **srun** для запуска параллельных задач)
scancel <ID-job>	команда для удаления задачи из очереди

Приложение 3. Пять основных шагов запуска MPI-приложений на кластере HybriLIT

Для проведения расчетов на кластере можно выделить пять основных шагов (ниже приведена инструкция для запуска MPI-приложений):

1. Зайти на кластер. Для подключения к кластеру пользователей *Windows* необходимо использовать специальную программу – *SSH*-клиент, например, *PuTTY*.



2. Загрузить необходимый модуль (модули):

```
module add   intel/v2018.1.163-2
```

3. Откомпилировать программу (для программ, написанных на C):

```
mpiicc  start_mpi.c -o test1
```

4. Запустить задачу на счет:

```
sbatch script_mpi.sh
```

Вашей задаче будет присвоен уникальный номер:

```
Submitted batch job 59528
```

5. Посмотреть выходной файл (соответствует номеру задачи):

```
cat slurm-59528.out
```

Учебное издание

Башашин Максим Викторович
Земляная Елена Валериевна
Стрельцова Оксана Ивановна

Практическое введение в технологию MPI на кластере HybriLIT

УЧЕБНОЕ ПОСОБИЕ

Редактор Ю. С. Цепилова
Технический редактор Ю. С. Цепилова
Компьютерная верстка Ю. С. Цепилова
Корректор Ю. С. Цепилова

Подписано в печать 04.06.2019. Формат 60×84/16. Усл. печ. л. 2,90.
Тираж 24 экз. Заказ № 15.

ГБОУ ВО МО «Университет «Дубна»
141980, г. Дубна Московской обл., ул. Университетская, 19