Министерство науки и высшего образования
Российской Федерации
Государственный университет «Дубна»

Институт системного анализа и управления
Кафедра распределенных информационно-вычислительных систем

Е. В. Земляная, М. В. Башашин

Введение в параллельное программирование на основе технологий MPI и OpenMP

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
2023

УДК 004.42
ББК 32.973.14я73
З-532

Р е ц е н з е н т :
кандидат физико-математических наук *А. С. Айриян*

Земляная, Е. В.
З-532 Введение в параллельное программирование на основе технологий MPI и OpenMP : учебное пособие / Е. В. Земляная, М. В. Башашин. — Дубна : Гос. ун-т «Дубна», 2023. — 101, [1] с.

ISBN 978-5-89847-696-0

В учебном пособии собраны материалы по практическому освоению двух наиболее популярных технологий параллельного программирования – MPI и OpenMP, преподаваемых в университете «Дубна» в рамках дисциплины «Архитектура вычислительных систем». Пособие подготовлено с учетом многолетнего опыта проведения семинарских занятий по этому курсу, включая дистанционное и частично дистанционное обучение в период 2020–2021 гг. По каждому разделу даны подробные пояснения, инструкции и примеры, позволяющие самостоятельно освоить материал и проверить себя, выполняя предложенные самостоятельные задания.

Пособие может быть полезно студентам других курсов, самостоятельно осваивающих параллельную организацию вычислений.

УДК 004.42
ББК 32.973.14я73

ISBN 978-5-89847-696-0
© Государственный университет «Дубна», 2023
© Земляная Е. В., Башашин М. В., 2023

Содержание

Введение ... 5

Глава 1. Практическое введение в технологию MPI ... 7

Семинар 1. Технология MPI: общая характеристика, запуск простейшей программы ... 7

1. Порядок работы на кластере hydra.uni-dubna.ru ... 7
2. Запуск простейшей MPI-программы ... 9

Семинар 2. Технология MPI: функции для определения номера процесса и размера группы; распределение работы между процессами ... 12

1. Функции для определения номера процесса и размера группы ... 13
2. Примеры распределения работы между процессами ... 16

Семинар 3. Технология MPI: обмен данными между отдельными процессами. Функции блокирующего обмена ... 20

Семинар 4. Технология MPI: функции неблокирующего обмена данными между отдельными процессами ... 27

Семинар 5. Технология MPI: коллективное взаимодействие процессов ... 33

1. Функция MPI_Bcast для широковещательной рассылки данных ... 34
2. Функция MPI_Reduce для глобальных вычислительных операций ... 36

Семинар 6. Технология MPI: параллельная реализация циклов ... 40

1. Организация «блочного» распределения итераций цикла между MPI-процессами ... 41
2. Организация «циклического» распределения итераций цикла ... 45

Семинар 7. Применение технологии MPI для параллельной реализации вычисления суммы ряда ... 47

Семинар 8. Контрольная работа по итогам освоения технологии MPI ... 50

Глава 2. Практическое введение в технологию OpenMP ... 52

Семинар 9. Технология OpenMP – общая характеристика, организация параллельных областей ... 52

1. Особенности технологии OpenMP в сравнении с MPI ... 52
2. Организация параллельных областей ... 54
3. Варианты регулирования количества нитей в параллельных блоках ... 56

4. Функции для определения количества нитей и номера нити ... 58
Семинар 10. Технология OpenMP: работа с общими и локальными данными ... 60
Семинар 11. Технология OpenMP: параллельное выполнение циклов ... 66
Семинар 12. Технология OpenMP: параллельное выполнение независимых фрагментов ... 74
Семинар 13. Технология OpenMP: синхронизация нитей в параллельных областях ... 80
1. Однократное выполнение фрагментов в параллельных областях ... 81
2. Последовательное выполнение фрагментов в параллельных областях ... 83
Семинар 14. Технология OpenMP: операция редукции ... 86
Семинар 15. Контрольная работа по технологии OpenMP ... 91

Список рекомендованной литературы ... 93
Приложение 1: Основные команды Linux, редакторы vim и nano ... 94
Приложение 2: Основные команды: Module и SLURM ... 98
Приложение 3: Инструкция по работе на кластере hydra.uni-dubna.ru и запуску задач в интерактивном и пакетном режимах ... 99

ВВЕДЕНИЕ

Знания о параллельных вычислениях и практические навыки организации ресурсоемких расчетов в параллельном режиме с применением специализированных технологий программирования являются в настоящее время необходимым элементом образовательного багажа ИТ-специалистов, поскольку практически все современные вычислительные системы имеют параллельную архитектуру [1–3]. В государственном университете «Дубна» студенты начинают знакомство с технологиями параллельного программирования и организации параллельных вычислений уже на первом курсе, в рамках дисциплины «Архитектура вычислительных систем» (АВС).

Данное учебное пособие подготовлено по материалам семинарских занятий АВС и полностью охватывает изучаемый в рамках этой дисциплины объем знаний и практических навыков работы с двумя популярными технологиями параллельного программирования – MPI и OpenMP [4–6]. Цель авторов пособия – максимально облегчить студентам практическое освоение базовых конструкций этих технологий. Естественно, студенты получают лишь самые начальные знания о технологиях параллельного программирования и основах организации параллельных вычислений. Однако полученный объем теоретических и практических знаний достаточен для разработки параллельных приложений и является необходимым базисом для дальнейшего освоения методов и технологий организации параллельных вычислений.

Более 20 лет обучение технологиям параллельного программирования на разных курсах в университете «Дубна» проводилось на вычислительных ресурсах Лаборатории информационных технологий Объединенного института ядерных исследований (ЛИТ ОИЯИ), в том числе – на гетерогенной вычислительной платформе HybriLIT, которая является частью Многофункционального информационно-вычислительного комплекса (МИВК) Лаборатории информационных технологий ОИЯИ [7; 8]. Платформа HybriLIT состоит из суперкомпьютера «ГОВОРУН» и учебно-тестового полигона HybriLIT (кластер HybriLIT), имеющих единую программно-информационную среду.

С 2021 г. из-за возросшего количества студентов практические занятия по курсу АВС проходят на установленном для этой цели в университете «Дубна» вычислительном кластере. Университетский кластер, хотя и уступает по производительности и количеству вычислительных узлов, объему памяти и другим параметрам ресурсам платформы HybriLIT, позволяет студентам получить практический опыт работы на вычислительных системах с параллельной архитектурой в рамках информационно-программной среды, характерной для таких высокопроизводительных ресурсов, как суперкомпьютер «ГОВОРУН».

По каждому разделу пособия даны подробные пояснения, инструкции и примеры, позволяющие самостоятельно освоить материал и проверить себя, выполняя предложенные самостоятельные задания. Пособие может быть полезно студентам других курсов, самостоятельно осваивающих параллельную организацию вычислений.

ГЛАВА 1. ПРАКТИЧЕСКОЕ ВВЕДЕНИЕ В ТЕХНОЛОГИЮ MPI

Семинар 1. Технология MPI: общая характеристика, запуск простейшей программы

Работа на университетском кластере hydra.uni-dubna.ru осуществляется в удаленном режиме по SSH-протоколу. Пользователям Windows для выхода на удаленный кластер удобно пользоваться приложением PuTTY.

На кластере установлена операционная система CentOS Linux 7.8.2003. Для создания и редактирования файлов на кластере HybriLIT доступны редакторы nano и vim. Описание наиболее популярных Linux-команд и команд редакторов дано в Приложении 1.

1. Порядок работы на кластере hydra.uni-dubna.ru

Для получения доступа на кластер необходимо зарегистрироваться. Студенты, изучающие курс АВС, получают логин и пароль у преподавателя семинаров.

При соединении с кластером через приложение PuTTY необходимо указать имя хоста – hydra.uni-dubna.ru, а также номер порта SSH – 22. В случае появления всплывающего окна кликнуть «да». При появлении окна удаленного доступа набрать логин и пароль. При правильном вводе пользователь попадает в свою папку. Ее содержимое можно проверить с помощью Linux-команды ls. Другие популярные Linux-команды даны в Приложении 1.

Логика работы по созданию и запуску С-программ следующая. Вызывается редактор, например nano (при вызове редактора через пробел указать желаемое имя файла, с расширением «.с»). В редакторе набирается текст программы, сохраняется, осуществляется выход из редактора в командный режим. Мышка в режиме редактирования не срабатывает. Для движения курсора по экрану следует использовать стрелочки на клавиатуре.

После того как файл с программой подготовлен, в командном режиме вызывается команда компиляции. Если компилятор диагностировал ошибки, снова заходим в редактор, исправляем, сохраняем, компилируем.

Если в программе нет ошибок, то создается исполняемый файл, при этом никаких записей на экран не выводится. Имя исполняемого модуля по умолчанию – a.out.

В появлении файла a.out можно убедиться, просмотрев содержимое своей папки помощью Linux-команды ls (имя исполняемого файла подсвечивается на экране зеленым цветом).

Запуск осуществляется интерактивно либо в пакетном режиме.

Перед работой с компилятором необходимо (один раз за сеанс) загрузить соответствующий модуль. Список всех доступных модулей вызывается командой

```
module avail
```

Рекомендуется пользоваться модулем openmpi 4.1.2, его загрузка осуществляется командой

```
module add openmpi/v4.1.2
```

Список команд планировщика, пакета Module, а также подробные инструкции по подготовке и запуску программ в интерактивном и пакетном режиме даны в Приложениях 2, 3.

Попробуем отработать схему работы на удаленном кластере, запустив для начала простейшую (непараллельную) С-программу, которая печатает слово “HELLO”.

Вызываем редактор:

```
nano test1a.c
```

Набираем текст кода:

```
// c-test hello:
#include <stdio.h>
int main(){
printf("HELLO\n");
return 0;
}
```

Сохраняем текст программы с выходом из редактора – нажимаем Ctrl-X (см. подсказки внизу экрана). Теперь в папке есть файл с программой с именем test1a.c.

После выхода из редактора в командный режим вводим команды загрузки модуля и компиляции:

```
module add openmpi/v4.1.2
gcc test1a.c
```

Для интерактивного запуска вводим:

```
./a.out
```

В результате работы программы видим на экране слово "HELLO", т.е. запуск прошел успешно. Теперь перейдем к начальному ознакомлению с технологией MPI для создания параллельных программ.

2. Запуск простейшей MPI-программы

Технология Message Passing Interface (MPI, интерфейс передачи сообщений) представляет собой один из наиболее широко распространенных инструментариев для создания параллельных приложений. Стандарт MPI работает с языками программирования C, C++, Fortran, Java и другими путем добавления соответствующей библиотеки.

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.

Работа с технологией MPI начинается с необходимости подключить соответствующую библиотеку в коде программы.

Для языка C (C++) это библиотека mpi.h. Данная библиотека подключается при помощи команды:

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

Чтобы инициализировать доступ к библиотеке MPI, нужно воспользоваться функцией MPI_Init. В результате выполнения функции создается группа процессов, количество которых было указано при запуске задачи, а также область связи – коммуникатор с предопределенным именем MPI_COMM_WORLD. Функции MPI_Init передаются аргументы функции main, полученные из командной строки.

Функция MPI_Finalize закрывает все MPI-процессы и уничтожает все области связи. Использование функций MPI после MPI_Finalize и до MPI_Init невозможно.

Таким образом, общая схема MPI-программы имеет вид:

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

Попробуем теперь организовать запуск простейшей параллельной программы, где каждый из параллельных процессов печатает слово “HELLO”, т.е. мы увидим на экране столько слов “HELLO”, сколько процессов выполняют нашу задачу. Количество процессов указывается при запуске.

Снова вызываем редактор. Можно редактировать тот же самый файл, добавив три строки. Как уже сказано, нужно подключить библиотеку mpi, добавить функции MPI_Init и MPI_Finalize, проследить, чтобы параметры main соответствовали параметрам MPI_Init.

```
#include <stdio.h>
#include <mpi.h>
int main(int argc,char* argv[])
{
MPI_Init(&argc, &argv);
printf ("HELLO!\n");
MPI_Finalize();
return 0;
}
```

Команда компиляции для MPI-программ:

```
mpicc testmpi.c
```

Указываем имя файла с нужной программой, расширение должно быть «.c». При успешной компиляции образуется исполняемый модуль с предопределенным именем a.out.

Интерактивный запуск осуществляется командой

```
mpirun ./a.out
```

Если при запуске не указано количество параллельных процессов, то используется значение по умолчанию. На нашем кластере значение по умолчанию – 8.

Видим, что на экране каждый из восьми запущенных по умолчанию параллельных процессов напечатал свое слово "HELLO".

```
HELLO!
HELLO!
HELLO!
HELLO!
HELLO!
HELLO!
HELLO!
HELLO!
```

Количество процессов указывается с помощью ключа –n. Например, чтобы запустить нашу программу для четырех процессов, используем команду

```
mpirun -n 4 ./a.out

HELLO!
HELLO!
HELLO!
HELLO!
```

Видим, что на экране каждый из четырех запущенных параллельных процессов напечатал свое слово “HELLO”.

Для завершения сеанса работы на кластере нужно набрать команду “exit”:

```
exit
```

Задания:

1. Отработать уверенный самостоятельный вход-выход на кластер, работу с редактором, компиляцию и запуск программы.
2. Внести изменения в MPI-программе, чтобы вместо слова “HELLO” на экран выводилось слово “GOODBYE” (или любая другая запись по выбору). Выполнить компиляцию модифицированной программы и ее запуск для трех и семи параллельных процессов.

Семинар 2. Технология MPI: функции для определения номера процесса и размера группы; распределение работы между процессами

Технология MPI подразумевает модель распределенной памяти, независимо от того, на какой реальной архитектуре производится расчет. Это означает, в частности, что любая объявленная переменная появляется в локальной памяти каждого процесса, задействованного для решения данной задачи. При этом каждый задействованный MPI-процесс, независимо

от остальных процессов в группе, выполняет все операции в программе, но над своими собственными данными, расположенными в своей собственной локальной памяти. Таким образом, если не предпринимать специальных усилий, то каждый процесс выполнит все операции в программе. Мы это видели на примере простейшей программы “HELLO”.

В реальных задачах нам требуется не дублировать многократное идентичное выполнение кода всеми процессами, а организовать работу так, чтобы каждый процесс выполнял свой «кусочек» работы, чтобы эти кусочки не перекрывались (или перекрывались по минимуму) и чтобы в совокупности вся необходимая работа, ради которой мы составляли программу, оказалась выполнена.

За счет того, что каждому процессу приходится выполнять не всю работу, а только ее часть, можно ожидать уменьшения времени счета при одновременном использовании более чем одного MPI-процесса. Это называется ускорением вычислений.

Конечно же эффект ускорения вычислений будет виден на достаточно сложных ресурсоемких программах. Наша же задача – научиться на простейших примерах корректно распределять работу между процессами с использованием MPI-функций.

Для того чтобы «сказать» каждому процессу, что именно ему надо делать, нужно каким-то образом идентифицировать их. В MPI параллельные процессы идентифицируются по номерам. Нумерация MPI-процессов и определение внутри программы их количества осуществляются вызовом соответствующих функций.

1. Функции для определения номера процесса и размера группы

Как уже было сказано выше, при запуске система создает группу процессов, количество которых равно числу запрашиваемых команде запуска, а также создается коммуникатор с предопределенным именем MPI_COMM_WORLD, объединяющий все задействованные процессы.

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 имя коммуникатора группы, в которой состоит процесс, и указатель на целочисленную переменную, которой будет присвоен порядковый номер процесса.

Запрос количества процессов в группе (это обычно называют «размер группы») 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 – имя коммуникатора группы и указатель на целочисленную переменную, которой будет присвоено число процессов.

Пример

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; // myid – process number;
//numproc – group size
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
// Print size of group and number of current
//MPI process
printf("Numprocs is %d Hello from %d\n",
numprocs, myid);
MPI_Finalize();
return 0;
}
```

Здесь каждый процесс в своей локальной памяти объявляет целые переменные для записи номера процесса и количества процессов в группе.

```
int myid, numprocs;
```

Каждый процесс вызывает функции

```
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
```

В результате по адресу myid в каждом процессе записывается его номер, по адресу numprocs – количество процессов. Далее каждый процесс печатает свои собственные переменные myid, numprocs.

Номер процесса (myid) у каждого свой, нумерация идет от 0, при печати порядок номеров может нарушаться, поскольку процессы работают параллельно и необязательно процесс, скажем, с номером 0 успеет первым напечатать свои данные.

Переменная numprocs также у каждого процесса своя, но ее значения во всех процессах совпадают, что мы и видим на экране при запуске программы на пяти процессах:

```
Numprocs is 5 Hello from 1
Numprocs is 5 Hello from 2
Numprocs is 5 Hello from 3
Numprocs is 5 Hello from 4
Numprocs is 5 Hello from 0
```

Отметим, что в MPI существует целый набор функций для организации подгрупп в рамках имеющейся группы и для создания коммуникаторов. Поэтому в вышеописанных двух функциях и в большинстве остальных MPI-функций в качестве параметра фигурирует имя коммуникатора.

2. Примеры распределения работы между процессами

В рассмотренном примере, как и в примере на прошлом семинаре, каждый процесс выполнил все операции в программе.

Рассмотрим теперь, как, зная номер процесса и их общее количество, можно распределить работу между параллельными процессами, явно указав тем или иным образом, какие именно фрагменты кода выполняются процессами с определенными номерами. Простейший вариант – использование условного оператора if.

Модифицируем предыдущий пример:

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[ ]){
int myid, numprocs; // myid - process number;
//numproc - group size
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
// Print size of group and number of curren
//MPI process
printf("Hello from %d\n", myid);

if(myid==0)
printf("Numprocs is %d\n",numprocs);

if(myid==numprocs-1)
printf("HELLO!!! I am %d!\n",myid);

MPI_Finalize();
return 0;
}
```

Здесь каждый процесс печатает свой номер, а количество процессов (размер группы) печатает только процесс с номером 0. Также процесс, имеющий максимальный номер, печатает “HELLO”. Отметим, что максимальный номер на 1 меньше раз-

мера группы, поскольку нумерация процессов осуществляется с 0. Как видим, разные процессы выполнили различные действия:

```
Hello from 2
Hello from 0
Numprocs is 5
Hello from 1
Hello from 4
HELLO!!! I am 4!
Hello from 3
```

Рассмотрим теперь более общий пример – как можно в общем виде, без использования оператора if, реализовать параллельное выполнение цикла, в котором на каждой итерации значение целочисленной переменной A=0 увеличивается на 1.

Один из вариантов – «блочное» распределение расчета между параллельными процессами. Поставим задачу написать программу в общем виде, чтобы схема распределения работала при любом количестве процессов, не превышающем количество итераций цикла.

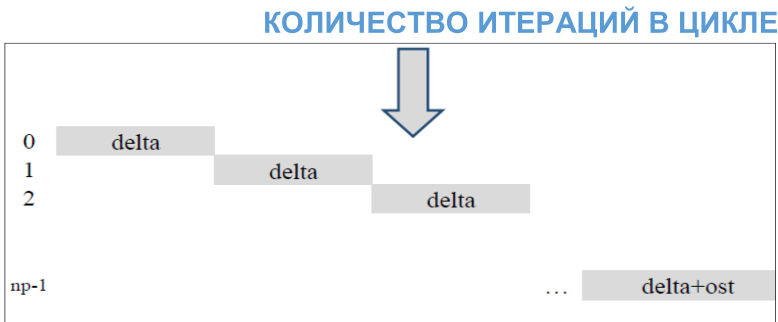


Рис. 1. Блочное распараллеливание

В предположении известной конечной длины цикла и известного количества процессов можно вычислить размер блока итераций, предназначенного для обработки каждым процессом, и позиции начального и конечного элементов массива.

Пусть мы находимся в нашем цикле N итераций и имеем np параллельных MPI-процессов. Для распределения итераций

цикла между задействованными процессами необходимо узнать размер блока (delta) итераций, выполняемого 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.

Здесь ost – остаток, который назначается в работу процессу с максимальным номером в случае, когда N не делится нацело на np.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[ ]){
int rank, np;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&np);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

int N=20; // number of iterations

int delta = N/np;
int ost = N%np;
int imin = rank*delta;
int imax = (rank+1)*delta;
if(rank== np-1) imax += ost;

int i,A=0;

for(i=imin;i<imax;i++)
{
A +=1 ;
printf("rank=%d, i=%d, A=%d\n", rank, i, A);
}
MPI_Finalize();
return 0;
}
```

При N=20 и np=4 каждый процесс с номером rank выполняет по del=N/np итераций, при этом переменная A в каждом процессе «пробегает» значения от 1 до del=5.

```
rank=1, i=5, A=1
rank=1, i=6, A=2
rank=1, i=7, A=3
rank=1, i=8, A=4
rank=1, i=9, A=5
rank=2, i=10, A=1
rank=2, i=11, A=2
rank=2, i=12, A=3
rank=2, i=13, A=4
rank=2, i=14, A=5
rank=3, i=15, A=1
rank=3, i=16, A=2
rank=3, i=17, A=3
rank=3, i=18, A=4
rank=3, i=19, A=5
rank=0, i=0, A=1
rank=0, i=1, A=2
rank=0, i=2, A=3
rank=0, i=3, A=4
rank=0, i=4, A=5
```

При запуске программы с другим количеством процессов переменная A в каждом процессе будет «пробегать» значения от 1 до del=N/np, за исключением последнего процесса, где A будет увеличиваться от 1 до del=N/np+ost. Предлагаем убедиться в этом самостоятельно.

Задания:

1. Воспроизвести на кластере HybriLIT рассмотренные примеры.
2. Составить и запустить программу, в которой процесс с номером 0 печатает слово “Hello”, процесс с номером 1 печатает слово “Goodbye”, процесс с номером 2 печатает количество процессов в группе, каждый процесс

с номерами >2 печатает свой номер. Запустить программу с количеством процессов 1, 2, 3, 4, 5, объяснить результат, который видим на экране.

Семинар 3. Технология MPI: обмен данными между отдельными процессами. Функции блокирующего обмена

Как мы уже обсуждали на предыдущих семинарах, технология MPI базируется на следующих принципах:

1) каждый процесс работает с данными, находящимися в собственной локальной памяти (модель распределенной памяти);

2) все задействованные для данной задачи процессы одновременно и независимо выполняют каждый оператор, написанный в программе (принцип «одна программа – много данных», SPMD, single program – multiple data).

В этих условиях распределение работы между процессами осуществляется с использованием их номеров. Этому мы учились делать на прошлом семинаре.

При этом в подавляющем большинстве реальных задач требуется обмен данными между процессами. Так, на стартовом этапе работы программы часто производится рассылка нужных входных данных разным процессам. Далее, на каждом этапе расчетов каждый процесс «знает» только свои собственные результаты, которые могут потребоваться другим процессам для их расчетов. На заключительном этапе вычислений часто требуется «сборка» окончательного результата из данных, находящихся в разных процессах.

Таким образом, помимо четырех уже известных нам функций MPI (`MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, `MPI_Comm_size`), нужны функции для организации обмена между процессами. MPI предоставляет широкие возможности для организации взаимодействия параллельных процессов. Соответствующие функции подразделяются на две группы:

- обмен между отдельными процессами (обмен типа «точка – точка»), когда в пересылках участвуют только два процесса: один посылает данные, другой их принимает;
- коллективные взаимодействия – в таких операциях задействованы все процессы группы (например, широковещательная рассылка).

Начнем наше рассмотрение с операций обмена «точка – точка», конкретно – с функций блокирующего приема-передачи. Для организации пересылки типа «точка – точка» необходим согласованный вызов процедуры отправки процессом-отправителем и процедуры приема процессом-получателем.

Логика использования таких функций следующая. Отправляющий процесс вызывает функцию отправки сообщения, в параметрах которой указан номер процесса, которому предназначено данное сообщение. Принимающий процесс вызывает функцию приема, параметры которой должны коррелировать с параметрами функции отправки, в частности, указывается номер процесса-отправителя, от которого принимается сообщение.

Процесс, который должен послать сообщение, может воспользоваться функцией блокирующей передачи `MPI_Send`. Данная функция формирует буфер сообщения и отправляет его процессу-адресату. Функция имеет следующий вид:

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

Параметры:

`buf` – адрес начала пересылаемых данных;

`count` – число пересылаемых элементов, расположенных подряд, начиная с адреса `buf`;

`datatype` – тип пересылаемых данных;

`dest` – номер процесса-получателя в группе, которой соответствует коммуникатор `comm`;

`tag` – идентификатор сообщения;

`comm` – коммуникатор, связанный с группой процессов, в которой идет данная пересылка.

Отметим особенности, характерные для процедур обмена в MPI:

– в MPI предполагается пересылка указанного количества расположенных подряд элементов указанного типа;

– данный порядок параметров является типичным для процедур обмена между отдельными процессами: указывается

начальная позиция массива данных, которые надо отправить (получить), их количество, тип, номер процесса, куда отсылаются данные (или откуда принимаются), идентификатор сообщения (тег) и имя коммуникатора;

– идентификатор сообщения должен совпадать в парных процедурах приема/передачи;

– все стандартные типы данных имеют MPI-аналоги (MPI_INT, MPI_DOUBLE и т. п.).

Прием сообщения с блокировкой осуществляется с помощью функции MPI_Recv, вызываемой процессом-адресатом. Функция имеет следующий вид:

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

Параметры:

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;

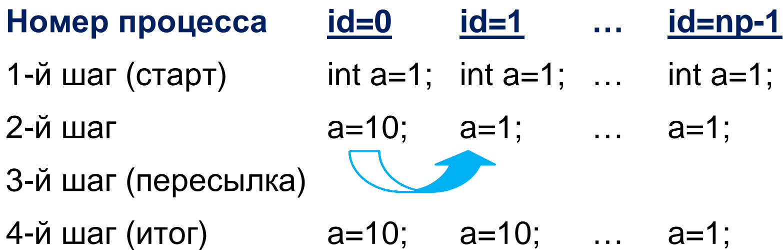
status – выходной параметр с атрибуты принятого сообщения. Для инициализации переменной status используется тип данных MPI_Status.

Функции MPI_Recv и MPI_Send называются блокирующими потому, что возврат из соответствующих процедур приема\передачи блокируется до возможности корректного доступа к данным буфера приема\передачи. В случае операции приема MPI_Recv это означает, что работа принимающего процесса блокируется до тех пор, пока сообщение не будет принято. Для процесса-отправителя выход из блокировки MPI_Send означает, что данные буфера передачи уже отправлены и можно безопасно пользоваться этой областью памяти.

Пример 1

Пусть в нашей программе, выполняемой группой np параллельных процессов, каждый из процессов объявляет целочисленную переменную a и присваивает ей значение 1. Далее, процесс с номером id=0 присваивает своей локальной переменной значение a=10 и пересылает это значение процессу id=1. В результате в процессах с номерами 0 и 1 будем иметь a=10, а в остальных процессах группы по-прежнему будет a=1.

Таблица 1. **Схема пересылки переменной**



Номер процесса	id=0	id=1	…	id=np-1
1-й шаг (старт)	int a=1;	int a=1;	…	int a=1;
2-й шаг	a=10;	a=1;	…	a=1;
3-й шаг (пересылка)				
4-й шаг (итог)	a=10;	a=10;	…	a=1;

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[ ]){
int id, np;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&np);
MPI_Comm_rank(MPI_COMM_WORLD,&id);
int a=1; // each process put a=1; step 1
if(id==0)
{
a=10; // 0-process put a=10; step 2
MPI_Send(&a,     1,     MPI_INT,     1,     55,
MPI_COMM_WORLD); // step 3
}
if(id==1)
{
MPI_Status status;
MPI_Recv(&a,1,MPI_INT,0,55,MPI_COMM_WORLD,
&status); // step 3
```

```
}
printf("id=%d, a=%d\n", id, a); // step 4
MPI_Finalize();
return 0;
}
```

В приведенном примере программы идентификатор сообщения равен 55.

Результат работы программы для четырех процессов имеет вид:

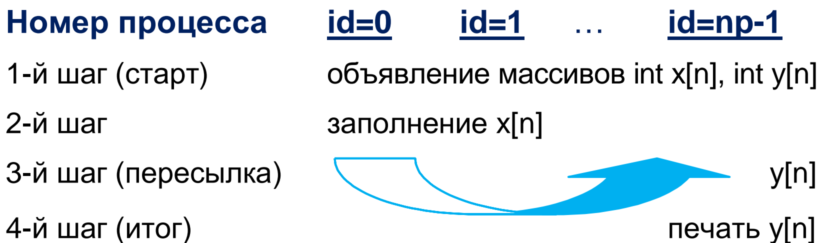
```
id=3, a=1
id=2, a=1
id=0, a=10
id=1, a=10
```

Таким образом, после пересылки процессы 0 и 1 имеют a=10, а в процессах 2 и 3 – a=1.

Пример 2

Рассмотрим теперь пересылку массивов. Пусть теперь в нашей программе, выполняемой группой np процессов, процесс 0 объявляет и заполняет целочисленный массив x длины n=5. Далее, процесс с номером id=0 отправляет содержимое массива x процессу с номером np-1 (максимальным в группе). Процесс с номером id=np-1 объявляет массив y длины n=5, записывает туда принятые из процесса 0 элементы массива x и выводит их на экран.

Таблица 2. **Схема пересылки массива**



Номер процесса	id=0	id=1	…	id=np-1
1-й шаг (старт)	объявление массивов int x[n], int y[n]			
2-й шаг	заполнение x[n]			
3-й шаг (пересылка)				y[n]
4-й шаг (итог)				печать y[n]

В программе идентификатор равен 33, элементам массива x присваиваются значения от 1 до 5. Ниже представлены программа и результат ее работы.

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

int main(int argc, char* argv[ ]){
int id, np;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&np);
MPI_Comm_rank(MPI_COMM_WORLD,&id);

int n=5; // length of array
int i=1; // index for loop

if(id==0) // 0-process // step 1
{
int *x; // int x[n];
x = (int*)malloc(n*sizeof(int));

for (i=0; i<n; i++)
x[i]=i+1;

MPI_Send(&x[0],n,MPI_INT,np-1,
33,MPI_COMM_WORLD); // step 2
free(x); // freeing the memory of pointer x
}

if(id==np-1) // Process with maximal number
{
MPI_Status status;

int *y; // int y[n];
y = (int*)malloc(n*sizeof(int));
```

```
MPI_Recv(&y[0],n,MPI_INT,0,33,MPI_COMM_WORLD,
&status); // step 2
printf("id=%d\n", id);

for(i=0; i<n; i++)
printf("i=%d, y[i]=%d\n", i, y[i]); // step 3
free(y); // freeing the memory of pointer y
}
MPI_Finalize();
return 0;
}
```

Результат работы программы при запуске на четырех процессах подтверждает, что процесс 3 успешно получил данные, отправленные ему процессом 0.

```
id=3
i=0, y[i]=1
i=1, y[i]=2
i=2, y[i]=3
i=3, y[i]=4
i=4, y[i]=5
```

Задания:

1. Воспроизвести на кластере HybriLIT рассмотренные примеры.
2. Модифицировать пример 1 так, чтобы присвоение a=100 происходило в процессе с максимальным номером в группе и пересылка этого значения осуществлялась в 0-процесс.
3. Модифицировать пример 2 так, чтобы пересылка массива y происходила из процесса с номером 1 в процесс с номером 0.

Семинар 4. Технология MPI: функции неблокирующего обмена данными между отдельными процессами

На предыдущих двух семинарах мы освоили шесть функций технологии MPI:

1) функции общего назначения (четыре функции):
   - `MPI_Init` – инициализация работы с MPI;
   - `MPI_Finalize` – завершение работы с MPI;
   - `MPI_Comm_rank` – определение номера процесса;
   - `MPI_Comm_size` – определение размера группы (количества процессов в группе);

2) функции для обмена между отдельными процессами (две функции):
   - `MPI_Send` – блокирующая отправка данных процессу с заданным номером;
   - `MPI_Recv` – блокирующий прием данных от процесса с заданным номером.

Этот набор функций можно считать «прожиточным минимумом», поскольку с их помощью можно организовать распараллеливание практически любой задачи. Остальные функции библиотеки MPI (их более 200) облегчают этот процесс – упрощают написание программы и (или) оптимизируют ресурсозатраты на обмен между параллельными процессами.

Наша следующая задача – познакомиться с функциями неблокирующего (асинхронного) обмена.

Неблокирующая пересылка 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. Таким образом, программа продолжает работу, в то время как данные передаются своим чередом.

С одной стороны, при таком обмене экономится время на ожидание завершения передачи данных. С другой стороны, существует опасность некорректной работы программы. Так, если в буфер, откуда отправляются данные, до завершения отправки будут записаны новые данные, произойдет искажение оправляемых данных и принимающий процесс получит не те данные, которые планировалось отправить. Аналогично, если буфер, куда записываются поступающие данные, будет исполь-

зован в расчетах до того, как данные реально приняты, это также приведет к ошибкам в работе программы. Поэтому для проверки завершения приема-передачи данных существуют специальные тестовые функции.

Итак, чтобы воспользоваться неблокирующей пересылкой, процесс-отправитель должен вызвать функцию **MPI_Isend**. Она имеет вид:

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

Параметры:

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». Это выходной параметр, который должен быть инициирован до обращения к функции, тип данных MPI_Request.

Отметим: параметры MPI_Isend совпадают с параметрами функции блокирующей передачи MPI_Send, отличие лишь в дополнительном параметре request, по которому осуществляется проверка завершения передачи данных.

Для организации неблокирующего приема используется функция **MPI_Irecv**, которую вызывает процесс-адресат. Она имеет следующий вид:

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

Параметры:

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».

Отметим, что функции блокирующего и неблокирующего приема/передачи полностью совместимы между собой и с другими MPI-функциями обмена «точка – точка», т.е. можно осуществить отправку данных с помощью блокирующей функции MPI_Send, а прием организовать с помощью неблокирующей функции MPI_Irecv, и наоборот.

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. Процедура имеет следующий вид:

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

Параметры:

request – «запрос обмена»;

status – атрибут сообщения.

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). Процедура вызывается следующим образом:

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

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

Пример

Рассмотрим использование неблокирующих функций приема/передачи на примере, разобранном на предыдущем семинаре. Итак, пусть в нашей программе, выполняемой группой из np параллельных процессов, каждый процесс объявляет целочисленную переменную a и присваивает ей значение 1. Далее, процесс с номером id=0 присваивает своей локальной переменной значение a=10 и пересылает это значение процессу id=1. В итоге, процессы с номерами 0 и 1 должны иметь a=10, а в остальных процессах группы по-прежнему будет a=1.

Для начала просто заменим в программе из примера 1 на предыдущем семинаре функции `MPI_Send`, `MPI_Recv` на неблокирующие функции `MPI_Isend`, `MPI_Irecv`. Также необходимо объявить параметр `request`.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[ ]){
int id, np;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&np);
MPI_Comm_rank(MPI_COMM_WORLD,&id);

int a=1; // each process put a=1

MPI_Request request;
MPI_Status status;

if(id==0){
a=10; // 0-process put a=10
MPI_Isend(&a,1,MPI_INT,1,55,MPI_COMM_WORLD,
&request);
}
```

```
if(id==1){
MPI_Irecv(&a,1,MPI_INT, 0, 55,MPI_COMM_WORLD,
&request);
}
printf("id=%d, a=%d\n", id, a);
MPI_Finalize();
return 0;
}
```

Результат запуска программы для четырех процессов:

```
id=0, a=10
id=1, a=1
id=2, a=1
id=3, a=1
```

Видно, что только в процессе с номером 0 имеем a=10. Процесс с номером 1 имеет a=1, поскольку прием происходит с помощью неблокирующей функции с немедленным возвратом в вызвавшую ее программу, т.е. печать переменной a произошла до того, как по этому адресу пришло новое значение 10.

Чтобы добиться корректной работы программы (т.е. печати нового значения a=10 в первом процессе), необходимо использовать функции тестирования. Сделаем это следующим образом: процесс 0 вызывает функцию `MPI_Test` и выводит на экран значение флага. Процесс 1 вызывает функцию `MPI_Wait`, обеспечивающую ожидание завершения обмена.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[ ]){
int id, np, flag0;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&np);
MPI_Comm_rank(MPI_COMM_WORLD,&id);

int a=1; // each process put a=1
MPI_Request request;
```

```
MPI_Status status;

if(id==0){
a=10; // 0-process put a=10
MPI_Isend(&a,1,MPI_INT,1,55,MPI_COMM_WORLD,
&request);
MPI_Test(&request,&flag0,&status);
}
if(id==1){
MPI_Irecv(&a,1,MPI_INT,0,55,MPI_COMM_WORLD,
&request);
MPI_Wait(&request,&status);
printf("id=%d, flag=%d\n", id, flag0);
}

printf("id=%d, a=%d\n", id, a);
MPI_Finalize();
return 0;
}
```

Видим, что после возврата из `MPI_Isend` флаг равен 1, т.е. отправка не завершена, но поскольку мы вызвали функцию `MPI_Wait` ожидания завершения обмена, то печать значений а происходит в процессе 1 после завершения приема, так что процесс номер 1 выводит на печать а=10.

```
id=0, flag=1
id=0, a=10
id=1, a=10
id=2, a=1
id=3, a=1
```

Задания:

1. Воспроизвести и запустить на кластере HybriLIT рассмотренный пример.
2. Модифицировать программу в рассмотренном примере так, чтобы отправка а=10 из 0-процесса осуществлялась с помощью блокирующей функции `MPI_Send`.

3. Модифицировать программу в рассмотренном примере так, чтобы прием a из 0-процесса осуществлялся с помощью блокирующей функции `MPI_Recv`.

Семинар 5. Технология MPI: коллективное взаимодействие процессов

Еще раз перечислим уже освоенные функции технологии MPI:

1) четыре функции общего назначения:
   – `MPI_Init` – инициализация работы с MPI;
   – `MPI_Finalize` – завершение работы с MPI;
   – `MPI_Comm_rank` – определение номера процесса;
   – `MPI_Comm_size` – определение размера группы (количества процессов в группе);

2) функции для организации обмена между отдельными процессами:
   – `MPI_Send` – блокирующая отправка данных процессу с заданным номером;
   – `MPI_Recv` – блокирующий прием данных от процесса с заданным номером;
   – `MPI_Isend` – неблокирующая отправка данных процессу с заданным номером;
   – `MPI_Irecv` – неблокирующий прием данных от процесса с заданным номером;
   – `MPI_Test` – вспомогательная функция для проверки завершения неблокироющего приема или передачи;
   – `MPI_Wait` – вспомогательная функция, обеспечивающая ожидание завершения приема или передачи.

Еще одно важное семейство операций – **операции коллективного обмена**.

Отличительные особенности коллективных операций:

- в таких операциях участвуют ВСЕ процессы группы;
- процедуры коллективного взаимодействия должны быть согласованно вызваны ВСЕМИ процессами группы;
- коллективные коммуникации НЕ взаимодействуют с процедурами типа «точка – точка»;
- возврат из процедуры коллективного обмена в каждом процессе происходит тогда, когда его участие в коллек-

тивной 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.

1. Функция MPI_Bcast для широковещательной рассылки данных

Рассылка информации от одного процесса всем остальным членам группы производится с помощью функции `MPI_Bcast`. Применение этой процедуры позволяет существенно сократить как объем кода, так и время выполнения рассылки по сравнению с использованием для этой цели процедур типа «точка – точка». Функция вызывается всеми процессами в группе и имеет следующие параметры:

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

Параметры:

`buf` – в рассылающем процессе: адрес начала расположения отправляемых данных; в остальных процессах: адрес начала буфера, где размещаются полученные данные;

`count` – количество отправляемых\получаемых данных;

`datatype` – тип отправляемых\получаемых данных;

`root` – номер процесса отправителя;

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

Графическая интерпретация функции `MPI_Bcast` демонстрирует схему рассылки данных от процесса `root` всем процессам в группе. Здесь `np` – количество процессов в группе, `count` – количество передаваемых элементов.

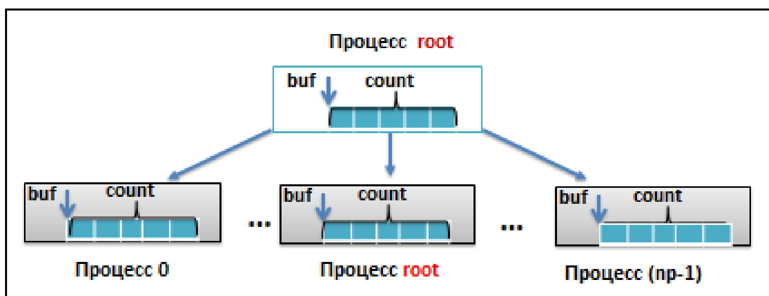


Рис. 2. Схема рассылки данных от одного процесса (root) всем процессам в группе: функция MPI_Bcast, np – количество процессов в группе

Фрагмент программы 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);
….
```

Пример 1. Широковещательная рассылка массива

Процесс с номером 0 рассылает массив a длины 5, заполненный целыми числами от 1 до 5, всем процессам в группе. После завершения пересылки каждый процесс печатает свой массив a.

```
#include <mpi.h>
#include <stdio.h>
#define comm MPI_COMM_WORLD
int main(int argc, char* argv[]){
 int rank, a[5];  int i;
 MPI_Init(&argc,&argv);
 MPI_Comm_rank(comm,&rank);
 if (rank==0)
```

```
 for(i=0;i<5;i++)
 a[i]=i+1;
 MPI_Bcast(&a,5,MPI_INT,0,comm);
 printf("rank=%d a=%d %d %d %d %d\n", rank,
 a[0], a[1], a[2], a[3] ,a[4]);
 MPI_Finalize();
 return 0;
}
```

Ниже показан результат запуска этой программы для четырех процессов. Видно, что все процессы корректно получили значения массива a, вычисленные в процессе с номером 0.

```
rank=0 a=1 2 3 4 5
rank=1 a=1 2 3 4 5
rank=2 a=1 2 3 4 5
rank=3 a=1 2 3 4 5
```

2. Функция MPI_Reduce для глобальных вычислительных операций

В параллельном программировании на базе MPI допускаются математические операции над блоками данных, распределенных между MPI-процессами. Такие операции называют глобальными операциями редукции. Подобные возможности в том или ином виде существуют и в других технологиях параллельного программирования.

Пусть параллельная программа на определенном этапе хранит в каждом процессе частичную сумму какой-либо величины. Формирование окончательного результата в принципе возможно путем сборки всех частичных сумм в один процесс и последующего суммирования полученных данных.

Однако применение для этой цели 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)
```

Параметры:

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.

Представленный ниже фрагмент программы иллюстрирует глобальное суммирование с помощью процедуры MPI_Reduce значений переменной a, находящихся в разных процессах переменных с типа double в переменную b, 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);
…
```

Пример 2. Глобальные операции с помощью MPI_Reduce и MPI_Allreduce

Пусть целочисленная переменная a в каждом процессе группы имеет значение, равное номеру процесса. Сумма всех значений a записывается по адресу S в процессе с номером 0, который печатает полученный результат.

```
#include <mpi.h>
#include <stdio.h>
#define comm MPI_COMM_WORLD
int main(int argc, char* argv[]){
int rank, a, S;
MPI_Init(&argc,&argv);
MPI_Comm_rank(comm,&rank);
a = rank;
MPI_Reduce(&a,&S,1,MPI_INT,MPI_SUM,0,comm);
if(rank==0)
printf("rank=%d, S=%d\n",rank,S);
MPI_Finalize();
return 0;
}
```

Результат запуска этой программы для четырех процессов:

```
 rank=0, S=6
```

Отметим, что помимо уже упомянутой операции `MPI_SUM` в MPI реализован целый ряд других глобальных операций, например, `MPI_MAX` – поиск максимума, `MPI_MIN` – поиск минимума, `MPI_PROD` – вычисление произведения и т. д.

Отметим также, что при применении глобальных операций к массивам данных в третьем параметре указывается длина массива. Результат такой глобальной операции также является массивом той же длины. В нулевой элемент массива-результата записывается результат глобальной операции над нулевыми элементами массивов-аргументов из разных процессов, в первый элемент – результат операции над первыми элементами

массивов-аргументов, во второй элемент – результат операции над вторыми элементами массивов-аргументов и т.д.

В случаях, когда результат глобальной операции должны «знать» все процессы группы, следует использовать функцию MPI_Allreduce. В отличие от MPI_Reduce, функция MPI_Allreduce возвращает результат редукции всем процессам, так что номер процесса-получателя не входит в список параметров, а буфер для записи результата должен быть объявлен во всех процессах группы. Функция имеет вид:

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

Параметры:

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.

Задания:

1. Воспроизвести и запустить на кластере HybriLIT рассмотренные примеры применения функций MPI_Bcast и MPI_Reduce.
2. Модифицировать программу из примера 1 для широковещательной рассылки целочисленной переменной X=100 из процесса, имеющего максимальный номер в группе.
3. Модифицировать программу из примера 2 для вычисления максимального значения всех а из разных процессов, с записью результата в процесс номер 1 и печатью этого результата.

4. Модифицировать программу из примера 2 для вычисления минимального значения всех a из разных процессов, с записью результата во все процессы в группе и с печатью полученных результатов. Использовать функцию `MPI_Allreduce`.

Семинар 6. Технология MPI: параллельная реализация циклов

На предыдущих семинарах мы освоили 12 необходимых и\или наиболее популярных функций технологии MPI. Хотя библиотека MPI содержит более 200 функций, полученных знаний вполне достаточно, чтобы начать составлять программы для эффективного выполнения задач в параллельном режиме.

Вопрос параллельной реализации циклов уже затрагивался ранее (см. семинар 2). Остановимся сейчас на этом более подробно. Наша задача – научиться писать программу в стиле SPMD (single program – multiple data), когда каждый процесс выполняет один и тот же фрагмент кода, обрабатывая при этом по единой схеме свои, назначенные ему данные. При этом работа между параллельными процессами должна быть распределена так, чтобы: 1) в совокупности работа должна быть выполнена полностью; 2) работы, назначенные к выполнению каждому процессу, не должны «перекрываться».

Рассмотрим задачу поэлементного суммирования двух массивов. Пусть определены значения элементов целочисленных массивов длины N: a[N] и b[N]. Тогда «классический» цикл по сложению элементов этих массивов с сохранением результата в массив с имеет вид:

```
for(i=0;i<N;i++)
c[i] = a[i]+b[i];
```

Рассмотрим варианты распределенного вычисления этого цикла. Основных вариантов два – «блочный» и «циклический», а также возможны различные варианты их комбинации.

1. Организация «блочного» распределения итераций цикла между MPI-процессами

Каждому процессу назначается к расчету фрагмент (блок) элементов массива с длины delta, начинающийся с позиции imin (значение imin – свое в каждом процессе). Размер блока delta вычисляется как delta=N/np. Если длина массива N делится нацело на количество процессов np, то работа между процессами с rank распределена поровну:

```
int delta = N/np;
imin = rank*delta;
imax = imin+delta;
for(i=imin;i<imax;i++)
c[i] = a[i]+b[i];
```

Если же N делится на np с остатком, то необходимо учесть это при распределении расчета, чтобы все элементы массива с были посчитаны. Один из вариантов – «назначить» расчет остающейся порции элементов процессу с номером np-1, как показано на схеме:

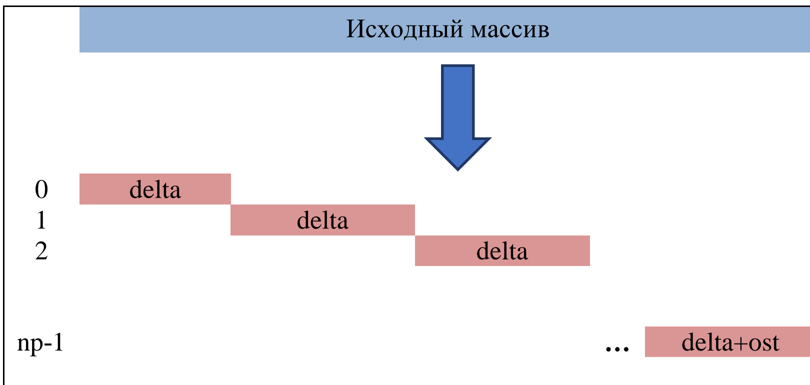


Рис. 3. Блочный тип распараллеливания

Учет остатка можно реализовать явным образом:

```
int ost = N%np;
imin = rank*delta;
```

```
imax = imin+delta;
if(rank==np-1) imax += ost;
for(i=imin;i<imax;i++)
c[i] = a[i]+b[i];
```

либо просто указать, что процесс с номером np-1 вычисляет элементы массива c, начиная с imin до последнего элемента массива, т.е. до N-1 включительно:

```
imin = rank*delta;
imax = imin+delta;
if(rank== np-1) imax = N;
for(i=imin;i<imax;i++)
c[i] = a[i]+b[i];
```

В результате выполнения любого из вышеприведенных фрагментов каждый процесс будет иметь в своей локальной памяти, в массиве c[N], блок рассчитанных элементов, в то время как остальные значения этого массива не определены. Для «сборки» блоков, посчитанных каждым процессом в единый массив в каком-либо одном назначенном процессе, можно использовать функции пересылки типа «точка – точка» либо функции коллективного взаимодействия типа `MPI_Gather`. В случае не слишком большой длины массивов сборку блоков в один процесс можно организовать с помощью уже известной нам процедуры `MPI_Reduce`.

В самом деле, если мы заранее положим все элементы массива c[N] равными 0 в каждом процессе, то получим после выполнения нашего распределенного цикла в каждом процессе массив c[N], в котором только рассчитанный блок содержит полезные данные, а остальные элементы равны 0. Так, для случая N=12, np=4 имеем:

rank=0											
c[0]	c[1]	c[2]	0	0	0	0	0	0	0	0	0
rank=1											
0	0	0	c[3]	c[4]	c[5]	0	0	0	0	0	0
rank=2											
0	0	0	0	0	0	c[6]	c[7]	c[8]	0	0	0
rank=3											
0	0	0	0	0	0	0	0	0	c[9]	c[10]	c[11]

Рис. 4. Распределение элементов при блочном типе распараллеливания

Применяя глобальное суммирование элементов массива из разных процессов с помощью процедуры `MPI_Reduce`, получим результирующий массив (обозначим его d[N]), содержащий результаты суммирования элементов массива a[N] и массива b[N].

Ниже показана программа, в которой объявляются массивы a[N], b[N], c[N], d[N]; с помощью «классического» цикла каждый процесс идентично определяет значения элементов массивов a[N], b[N], c[N]; затем выполняется «блочно»-распределенный цикл, в котором каждый процесс вычисляет свой фрагмент массива c[N]. Далее осуществляются сборка результатов в массив d[N] в процесс с номером 0. Заметим, что здесь для обозначения номера процесса используется id, а не rank, как в предыдущем тексте.

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char* argv[ ]){
int id, np;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&np);
MPI_Comm_rank(MPI_COMM_WORLD,&id);
int N=12;
int i;
int a[N], b[N], c[N], d[N];

for(i=0; i<N; i++){ // initial value of a,b,c
a[i]=i+1;
```

```
b[i]=i+2;
c[i]=0;
}
int delta = N/np;
int imin = id*delta;
int imax = imin+delta;
if(id == np-1) imax = N;

for(i=imin;i<imax;i++){ // parallel loop
c[i]=a[i]+b[i];
printf("id=%d,  i=%d,  c[i]=%d\n",  id,  i,
c[i]);
}
MPI_Reduce(&c[0], &d[0], N, MPI_INT, MPI_SUM,
0, MPI_COMM_WORLD);

if(id==0){
for(i=0;i<N; i++)
printf("final result: i=%d, d[i]=%d\n",  i,
d[i]);
}
MPI_Finalize();
return 0;
}
```

Ниже показан результат запуска данной программы для N=12, np=4. Каждый процесс печатает вычисленные им элементы массива c, затем 0-процесс печатает окончательный результат – массив d[N] целиком.

```
id=1, i=3, c[i]=9
id=1, i=4, c[i]=11
id=1, i=5, c[i]=13
id=0, i=0, c[i]=3
id=0, i=1, c[i]=5
id=0, i=2, c[i]=7
id=2, i=6, c[i]=15
id=2, i=7, c[i]=17
```

```
id=2, i=8, c[i]=19
id=3, i=9, c[i]=21
id=3, i=10, c[i]=23
id=3, i=11, c[i]=25
final result: i=0, d[i]=3
final result: i=1, d[i]=5
final result: i=2, d[i]=7
final result: i=3, d[i]=9
final result: i=4, d[i]=11
final result: i=5, d[i]=13
final result: i=6, d[i]=15
final result: i=7, d[i]=17
final result: i=8, d[i]=19
final result: i=9, d[i]=21
final result: i=10, d[i]=23
final result: i=11, d[i]=25
```

2. Организация «циклического» распределения итераций цикла

Другой вариант распределения расчета – «циклическое» распределение итераций цикла между параллельными процессами – заключается в том, что каждый процесс обрабатывает итерации цикла с индексами, начиная со своего порядкового номера, и с шагом, равным числу процессов в группе.

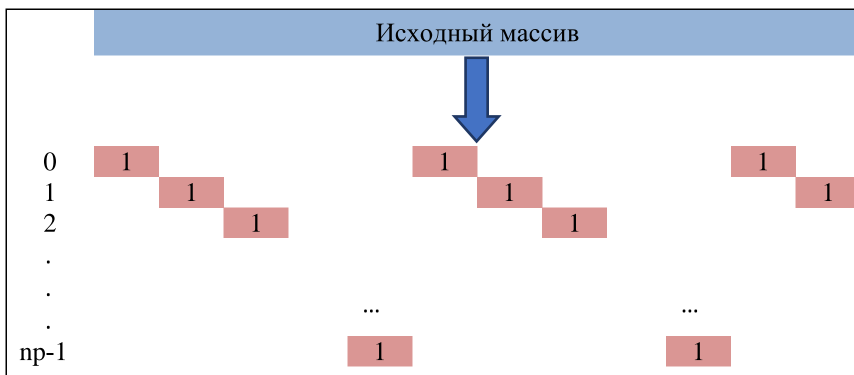


Рис. 5. Циклический тип распараллеливания

Реализация такого распределения на вышеприведенном примере имеет следующий вид:

```
for(i=rank;i<N;i+=np)
c[i]=a[i]+b[i];
```

В этом подходе операции расчета delta, imin, imax не нужны.

В результате выполнения этого цикла для N=12 и np=4 получаем следующую структуру массивов c[N] в каждом процессе.

rank=0											
c[0]	0	0	0	c[4]	0	0	0	c[8]	0	0	0
rank=1											
0	c[1]	0	0	0	c[5]	0	0	0	c[9]	0	0
rank=2											
0	0	c[2]	0	0	0	c[6]	0	0	0	c[10]	0
rank=3											
0	0	0	c[3]	0	0	0	c[7]	0	0	0	c[11]

Рис. 6. Распределение элементов при циклическом типе распараллеливания

Понятно, что применение процедуры `MPI_Reduce` также приведет к правильному формированию результирующего массива d[N].

Задания:

1. Воспроизвести и запустить на кластере HybriLIT рассмотренный пример по блочному распределению поэлементного суммирования двух массивов.
2. Модифицировать рассмотренную программу для случая «циклического» распараллеливания.
3. Модифицировать программу так, чтобы значения массивов a,b,c задавались в процессе с номером 0, а затем производилась рассылка с помощью процедуры `MPI_Bcast`.

Семинар 7. Применение технологии MPI для параллельной реализации вычисления суммы ряда

Применим наши знания, полученные на предыдущих семинарах, для параллельной реализации расчета суммы ряда

$$\mathrm{c} = \sum_{i=0}^{\infty} \frac{1}{x_i^2} + \frac{1}{x_i^4}, \qquad x_i = i + 1.$$

- Сумма этого ряда равна (если вспомнить математический анализ) $\pi^2/6+ \pi^4/90$.
- Естественно, вместо бесконечного числа членов ряда будем суммировать конечное число – n=10000000.
- Каждый процесс будет суммировать свои члены ряда – вычислять частичную сумму c.
- Для сложения всех частичных сумм из разных процессов применим процедуру `MPI_Reduce` с записью результата в c1.
- Будем измерять время счета с использованием функции `MPI_Wtime`

```
#include <mpi.h> // add MPI library
#include <stdio.h> //add input/output library
#include <stdlib.h> // add library for malloc
//and free functions
#define comm MPI_COMM_WORLD // redefined
//the communicator name
#define n 100000000 // number of series
//members
int main(int argc, char* argv[])
{
MPI_Init(&argc,&argv); // MPI initialization
int rank, np; // rank - process id, np -
//number of processes
MPI_Comm_rank(comm, &rank); // determined
//rank - process id
```

```
MPI_Comm_size(comm, &np); // determined np –
//number of processes

double c=0,c1; // c – part sum, c1 – final
//result

// declare an array x of length n for the
//members of our series
double *x;// double x[n];
x = (double*)malloc(n*sizeof(double));
// block distribution scheme for calculating
//the sum of a series
// calculation of minimum and maximum row
//numbers in each processint del = n/np;
int imin = rank*del;
int imax = (rank + 1) * del;
if (rank == np - 1)     imax = n;

int i; // loop index
double xx;
double time1 = MPI_Wtime(); // first time
//point
for (i = imin; i < imax; i++) // loop
//operator, block version
{
x[i] = 1.0+i;
xx = x[i]*x[i];
c += 1.0/xx + 1.0/(xx*xx); // partial sum
//calculated in each process
}
// global operation
// partial sums "c" are summed up
// with the result recorded in "c1" in rank=0
MPI_Reduce(&c,&c1, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
double time2 = MPI_Wtime(); // second time
//point.
// 0-process prints result and execution time
```

```
if (rank == 0)
{
printf ("Sum = %f; execution time=%f\n ", c1,
time2-time1);
}
MPI_Finalize();
free(x);
return 0;
}
```

Поясним, как работает процедура `MPI_Wtime`. Процедуру надо вызвать дважды – до и после участка программы, время выполнения которого необходимо измерить:

```
double time1=MPI_Wtime();
…
…
…
double time2= MPI_Wtime();
```

Разница между time1 и time2 дает время (в секундах) выполнения фрагмента между двумя вызовами `MPI_Wtime`.

Самостоятельные задания:

1. Довести до работающего состояния программу расчета суммы ряда. Проверить соответствие расчетной суммы ряда и аналитического значения.
2. Выполнить серию запусков с возрастающим количеством процессов. Найти для данной задачи оптимальное количество процессов, при котором время вычисления – минимальное.
3. Заменить в программе «блочную» схему параллельного выполнения цикла на «циклическую» и выполнить для этого случая пункт 2.

Семинар 8. Контрольная работа по итогам освоения технологии MPI

На предыдущих семинарах мы освоили 12 необходимых и\или наиболее популярных функций технологии MPI, научились организовывать параллельное выполнение циклов, а также получили опыт практического применения MPI для решения конкретной задачи – вычисления суммы ряда. Итак, мы знаем:

1) функции общего назначения:
   - `MPI_Init` – инициализация работы с MPI;
   - `MPI_Finalize` – завершение работы с MPI;
   - `MPI_Comm_rank` – определение номера процесса;
   - `MPI_Comm_size` – определение размера группы (количества процессов в группе);

2) функции для организации обмена между отдельными процессами:
   - `MPI_Send` – блокирующая отправка данных процессу с заданным номером;
   - `MPI_Recv` – блокирующий прием данных от процесса с заданным номером;
   - `MPI_Isend` – неблокирующая отправка данных процессу с заданным номером;
   - `MPI_Irecv` – неблокирующий прием данных от процесса с заданным номером;
   - `MPI_Test` – вспомогательная функция для проверки завершения неблокироющего приема или передачи;
   - `MPI_Wait` – вспомогательная функция, обеспечивающая ожидание завершения приема или передачи;

3) функции для организации коллективных взаимодействий между процессами:
   - `MPI_Bcast` – широковещательная рассылка данных всем процессам в группе;
   - `MPI_Reduce` (`MPI_Allreduce`) – глобальные операции над данными из разных процессов.

Контрольная работа состоит в самостоятельном написании, отладке и запуске параллельной программы по индивидуальному для каждого студента заданию с использованием пройденных на семинарах конструкций MPI.

Примеры заданий для контрольной работы по MPI

1. Написать MPI-программу, корректно работающую для трех и более MPI-процессов, выполняющую следующее.

Процесс номер 1 присваивает целой переменной a значение 555 и пересылает это значение процессу номер 0 с помощью процедуры `MPI_Send`; присваивает целой переменной b значение 666 и пересылает это значение процессу номер 2 с помощью процедуры `MPI_Send`.

Процесс номер 0 получает a от процесса номер 1 с помощью вызова процедуры `MPI_Recv` и выводит на дисплей. Процесс номер 2 получает b от процесса номер 1 с помощью вызова процедуры `MPI_Irecv` и выводит на дисплей b и количество MPI-процессов в группе.

2. Написать MPI-программу, корректно работающую для двух и более MPI-процессов и выполняющую следующее.

Процесс с номером 1 присваивает целой переменной a значение, равное количеству процессов в группе + 11. Осуществляется рассылка значения a всем процессам в группе с помощью процедуры `MPI_Bcast`. Каждый процесс печатает свой номер, число процессов в группе и значение a.

3. Написать MPI-программу, корректно работающую для трех и более MPI-процессов, выполняющую следующее.

Каждый процесс присваивает целой переменной a значение, равное 2. Осуществляется глобальное перемножение значений a из всех процессов в группе с помощью процедуры `MPI_Reduce` с записью результата в переменную s в процессе номер 2.

Процесс номер 2 выводит полученное значение s на дисплей. Процесс номер 0 выводит на дисплей свой номер и количество процессов в группе.

ГЛАВА 2. ПРАКТИЧЕСКОЕ ВВЕДЕНИЕ В ТЕХНОЛОГИЮ OpenMP

Семинар 9. Технология OpenMP – общая характеристика, организация параллельных областей

1. Особенности технологии OpenMP в сравнении с MPI

OpenMP (Open specifications for Multi-Processing) – один из популярных стандартов для создания параллельных компьютерных программ. В модели OpenMP программа представляется как набор параллельных потоков (нитей, threads), объединенных общей памятью. Технология OpenMP включает набор специальных директив компилятору, библиотечных функций и переменных окружения.

Директивы компилятору (прагмы) используются для обозначения областей в программе с возможностью параллельного выполнения. Компилятор, поддерживающий OpenMP, преобразует исходный код и вставляет соответствующие вызовы функций для параллельного выполнения этих областей. Компилятор, не поддерживающий OpenMP, директивы игнорирует. Тем самым открывается возможность создания единого кода для параллельного и последовательного выполнения одной и той же задачи, а также возможность относительно простой «пошаговой» параллельной оптимизации уже существующих последовательных программ.

В отличие от технологии MPI, реализующей модель распределенной памяти, особенностью технологии OpenMP является ориентация на компьютерные системы с общей памятью, в которых доступ к любой ячейке памяти имеют все доступные параллельные узлы.

Рассмотрим особенности в OpenMP в сравнении с технологией MPI, которую изучали в первой половине семестра.

Таблица 3. **Различия между технологиями MPI и OpenMP**

Свойство	MPI	OpenMP
Память	Каждый параллельный процесс имеет доступ только к своей локальной памяти	Взаимодействие между параллельными потоками (нитями, threads) осуществляется за счет общих переменных (shared)
Параллелизм	Явное распараллеливание	Помимо явного распараллеливания существуют также возможности высокоуровневого параллелизма за счет использования конструкций для автоматического распараллеливания циклов и независимых фрагментов
Реализация	Библиотека MPI-функций	Функции OpenMP и директивы (прагмы). В C\C++ все директивы начинаются с #pragma omp
Выполнение программы	Все операции компьютерного кода выполняют все задействованные параллельные процессы	Работу начинает одна нить (нить-мастер). Параллельные области организуются с помощью специальных OpenMP-директив

Отметим, что такие особенности OpenMP, как наличие общих переменных и возможности для высокоуровневого параллелизма делают технологию OpenMP удобной для создания параллельных приложений. Однако в данной технологии, как и в других, существуют свои «подводные камни», требующие специального внимания разработчика. В частности, при создании OpenMP-приложений следует учитывать затраты компьютерного времени на организацию параллельных секций и на работу с общими переменными.

2. Организация параллельных областей

Как уже сказано выше, в технологии OpenMP работу начинает одна нить (нить-мастер). Параллельные области организуются с помощью специальных OpenMP-директив. При входе в параллельную область (fork) порождаются дополнительные нити, каждая из которых получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все нити исполняют один и тот же код, соответствующий параллельной области. При выходе из параллельной области (join) происходит неявная синхронизация нитей, и дальнейшее выполнение программы продолжает только нить-мастер. Таким образом, программа разбивается на последовательные и параллельные области.

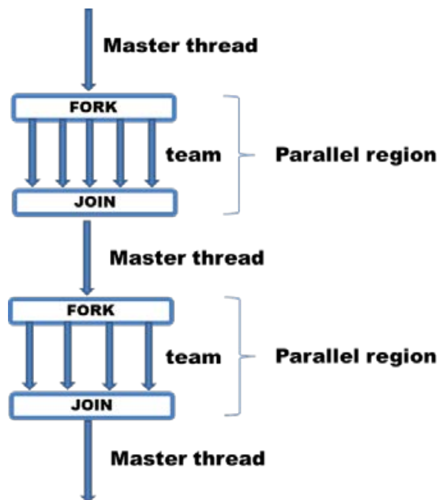


Рис. 7. Структура OpenMP-программы, состоящей из параллельных и последовательных областей

Параллельный блок организуется с помощью прагмы `#pragma omp parallel`:

```
#pragma omp parallel
{
…
}
```

Таким образом, все, что находится в фигурных скобках, выполняется набором нитей в параллельном режиме. Операторы за пределами параллельного блока выполняются одной нитью (потоком).

!!!ВАЖНО!!! язык программирования C/C++ подразумевает, что все параметры, относящиеся к прагме, должны быть написаны в коде в **ОДНУ СТРОКУ**. **Перенос на следующую строку не допускается.** Текстовый формат данного пособия не всегда позволяет соблюсти это правило.

Пример 1

В приведенной ниже программе печать “Parallel block” выполняется каждой нитью (шесть нитей при запуске в интерактивном режиме на кластере HybriLIT). Вывод на печать выражений перед параллельным блоком и после него выполняется единожды.

```
// OpenMP test: creating the parallel block
#include <stdio.h>
int main(int argc, char *argv[]){
printf("Only  Master-thread  starts  working
\n");
#pragma omp parallel
{printf("Parallel block\n");  }
printf("Only Master-thread continues working
\n");
return 0;
}
```

Перед компиляцией программы нужно загрузить соответствующий модуль. Компиляция осуществляется с помощью команды

```
gcc –fopenmp test.c
```

Здесь `fopenmp` – ключ, обеспечивающий поддержку технологии OpenMP (для gnu-компилятора).

Интерактивный запуск производится с помощью команды

```
./a.out
```

Результат выполнения программы на 16 OpenMP-нитях (значение по умолчанию для кластера hydra.uni-debna.ru) выглядит следующим образом:

```
Only Master-thread starts working
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Parallel block
Only Master-thread continues working
```

3. Варианты регулирования количества нитей в параллельных блоках

1. Использование команды export. За количество нитей в OpenMP-программах «отвечает» переменная окружения `OMP_NUM_THREADS`, значение которой можно установить в командном режиме, не меняя текст программы:

```
[@hydraOMP_examples]$export OMP_NUM_THREADS=4
[@hydra OMP_examples]$ ./a.out
Only Master-thread starts working
Parallel block
```

```
Parallel block
Parallel block
Parallel block
Only Master-thread continues working
```

Здесь установлено четыре нити, соответственно печать “Parallel block” выполняется четыре раза четырьмя параллельными нитями.

2. Использование функции `omp_set_num_threads` в программе. Все параллельные области после вызова этой функции будут выполняться установленным количеством нитей.

3. Использование опции `num_threads` прагмы parallel. Эта установка действует только на одну конкретную параллельную область в программе.

Пример 2

В приведенной программе организованы две параллельные области. Первая выполняется двумя нитями согласно установке с помощью функции `omp_set_num_threads`. Вторая – шестью нитями, как установлено с помощью опции `num_threads`.

```
//OpenMP: test 2.
// How to set number of threads in parallel
//secions

#include <stdio.h>
#include <omp.h>
int main()
{
// we set number of threads = 2
omp_set_num_threads(2);

// open first parallel block with two threads
#pragma omp parallel
{
printf("parallel  1 \n");
}
```

```
// open second parallel block with 6 threads
#pragma omp parallel num_threads(6)
{
printf("parallel 2 \n");
}
return 0;
}
```

В результате работы этой программы запись “parallel 1” выводится на дисплей четырьмя нитями, в то время как запись “parallel 2” – двумя.

```
parallel 1
parallel 1
parallel 2
parallel 2
parallel 2
parallel 2
parallel 2
parallel 2
```

4. Функции для определения количества нитей и номера нити

Функции `omp_get_thread_num` и `omp_get_num_threads` возвращают соответственно номер нити и количество задействованных в данной точке программы нитей. В примере 3 демонстрируется использование этих функций.

Пример 3

```
// OMP test3: getting number of threads and rank
//of thread

#include <omp.h>
#include <stdio.h>
int main(){

#pragma omp parallel
{
```

```
int id = omp_get_thread_num(); // rank ot thread
int nt = omp_get_num_threads(); // number of
//threads

printf("Hello from thread = %d\n", id);

if (id == 0){
printf("Number of threads = %d\n", nt);
}
}
return 0;
}
```

Отметим, что при объявлении переменных id, nt внутри параллельной области эти переменные являются локальными (т.е. каждая нить имеет свой экземпляр этих переменных) внутри области и не существуют вне этой области.

Отметим также, что, зная номер нити и их количество, можно организовать явным образом распределение работы между нитями. Так, в данном примере печать количества нитей осуществляется только нитью с номером 0:

```
Hello from thread = 3
Hello from thread = 1
Hello from thread = 0
Number of threads = 4
Hello from thread = 2
```

Задания:

1. Воспроизвести на кластере HybriLIT вышеприведенные примеры по организации параллельной области, установлению количества нитей и определению номера нити и их числа.
2. Подготовить и запустить программу, в которой организовать с помощью прагмы `parallel` и опции `num_threads` параллельную область с количеством нитей, равным 8. Нить с максимальным номером должна напечатать количество нитей в параллельном блоке.

Нить-мастер (с номером 0) должна напечатать слово “Hello” и свой номер.

Семинар 10. Технология OpenMP: работа с общими и локальными данными

По итогам предыдущего семинара мы знаем, что в OpenMP-программе работу начинает одна нить (нить-мастер), с помощью прагмы `omp parallel` организуются параллельные блоки, выполняемые набором потоков (нитей), количество которых можно регулировать, каждая из нитей имеет свой номер и нить-мастер имеет номер 0.

Взаимосвязь между нитями в параллельных секциях, а также между параллельными (многопоточными) и последовательными (однопоточными) участками кода поддерживается за счет различных типов данных.

В OpenMP существуют как общие, так и локальные переменные. Необходимо учитывать особенности работы с каждым типом, в том числе тот факт, что доступ у каждой нити к общей переменной и соответственно возможность изменить ее значение, в случае некорректного использования общих переменных может привести к неправильной работе программы.

В OpenMP определены следующие основные типы переменных, характеризующие доступ к ним из разных участков OpenMP-программы (эти типы объявляются в прагмах parallel; не путать со стандартными типами данных int, float и т.п.):

- `shared` – такие переменные являются общими для всех нитей, т.е. все нити во всех параллельных и последовательных областях кода будут обращаться к одной области памяти;
- `private` – каждый поток имеет свою копию переменной, значение которой обновляется независимо каждой нитью, не передается из последовательной области в параллельную и не сохраняется за пределами параллельной области;
- `firstprivate` – аналогична `private`, но переменные данного типа при входе в параллельный регион инициализируются значением, которое переменная имела перед открытием этого региона;

- `lastprivate` – аналогична `private`, но после закрытия параллельной области переменные сохраняют значение, полученное в ходе вычислений внутри параллельного региона. Используется в конструкциях `for` и `section`.

На кластере hydra.uni-debna.ru типом переменной по умолчанию является `shared`.

Пример 1

Задаем целочисленную переменную a – `private`, значение которой, как в таблице, задается в начале программы, когда работает только одна нить, нить-мастер.

Пусть a=199, как в таблице. Выводим на дисплей значения этой переменной:

- в мастер-нити;
- в начале параллельной секции,
- в конце параллельной секции после некоторых вычислений;
- после закрытия параллельной секции.

Будем следить, как изменится (или не изменится) значение a.

```
#include <omp.h>
#include <stdio.h>
int main() {
int a=199;
printf("in  master-thread  before   parallel
block: a=%d\n",a);

#pragma omp parallel private(a)
{
int id=omp_get_thread_num();
printf("begin   of   parallel   block:   a=%d,
id=%d\n",a,id);
a+=id;
printf("end   of    parallel   block:   a=%d,
id=%d\n",a,id);
}
```

```
printf("in master-thread after parallel
block: a=%d\n",a);
return 0;
}
```

При запуске этой программы с числом нитей 6 получаем следующий результат:

```
in master-thread before parallel block: a=199
begin of parallel block: a=0, id=0
end of parallel block: a=0, id=0
begin of parallel block: a=0, id=5
end of parallel block: a=5, id=5
begin of parallel block: a=0, id=4
end of parallel block: a=4, id=4
begin of parallel block: a=0, id=2
end of parallel block: a=2, id=2
begin of parallel block: a=0, id=3
end of parallel block: a=3, id=3
begin of parallel block: a=0, id=1
end of parallel block: a=1, id=1
in master-thread after parallel block: a=199
```

Пример 2

Объявим теперь в нашем предыдущем примере переменную a – `firstprivate`:

```
#pragma omp parallel firstprivate(a)
```

Снова проследим, как будет меняться (или не изменится) ее значение при открытии и закрытии параллельной области.

```
in master-thread before parallel block: a=199
begin of parallel block: a=199, id=0
end of parallel block: a=199, id=0
begin of parallel block: a=199, id=3
end of parallel block: a=202, id=3
begin of parallel block: a=199, id=2
```

```
end of parallel block: a=201, id=2
begin of parallel block: a=199, id=1
end of parallel block: a=200, id=1
begin of parallel block: a=199, id=5
end of parallel block: a=204, id=5
begin of parallel block: a=199, id=4
end of parallel block: a=203, id=4
in master-thread after parallel block: a=199
```

Видно, что при входе в параллельную область в каждой из шести нитей переменная a имеет значение a=199, заданное в мастер-нити до открытия параллельной области. Далее, каждая нить вычисляет свое собственное значение a, равное ее номеру, и выводит это значение на печать. Однако после закрытия параллельной области результаты этих расчетов не сохраняются. По-прежнему имеем в мастер-нити a=199, как и в предыдущем примере.

Пример 3

Теперь пусть наша переменная a – общая (`shared`):

```
#pragma omp parallel shared(a)
```

Снова проследим, как будет меняться (или не изменится) ее значение при открытии и закрытии параллельной области.

```
in master-thread before parallel block: a=199
begin of parallel block: a=199, id=2
end of parallel block: a=201, id=2
begin of parallel block: a=199, id=0
end of parallel block: a=201, id=0
begin of parallel block: a=199, id=1
end of parallel block: a=202, id=1
begin of parallel block: a=199, id=3
end of parallel block: a=205, id=3
begin of parallel block: a=199, id=5
end of parallel block: a=210, id=5
begin of parallel block: a=199, id=4
```

```
end of parallel block: a=214, id=4
in master-thread after parallel block: a=214
```

Поскольку переменная общая и все нити в параллельной области имеют к ней равноправный доступ – при входе в параллельную область каждая нить печатает a=199, далее каждая нить увеличивает значение общей для всех переменной на величину, равную своему номеру, так что на печати наблюдаем постепенное увеличение значения a. После закрытия параллельной области имеем: a=199+0+1+2+3+4+5=214.

Однако не все так просто! Все нити работают с единым адресом в памяти, т.е. каждая нить копирует значение переменной a в свой кэш, вычисляет новое значение и обновляет значение a в своем кэше и затем в общей памяти. Поскольку все нити работают параллельно, существует вероятность, что две (или более) нити реализуют запрос к памяти практически одновременно и скопируют в свой кэш одно и то же значение a, которое было на тот момент сохранено в памяти. Каждая их этих нитей увеличит скопированное значение a на свой номер и затем обновит содержимое общего адреса памяти. В результате окажется, что одна (или более) нитей записали свой результат «поверх» результата, записанного чуть ранее другой нитью, так что не все результаты вычислений окажутся учтены в итоговом значении a.

Из-за этого финальное значение переменной a может меняться от запуска к запуску. Для того чтобы убедиться в этом, запустим программу несколько раз подряд:

```
[@hydra codes]$ ./a.out
in master-thread before parallel block: a=199
in master-thread after parallel block: a=210
[@hydra codes]$ ./a.out
in master-thread before parallel block: a=199
in master-thread after parallel block: a=206
[@hydra codes]$ ./a.out
in master-thread before parallel block: a=199
in master-thread after parallel block: a=211
[@hydra codes]$ ./a.out
in master-thread before parallel block: a=199
```

```
in master-thread after parallel block: a=214
```

Видно, что в разных запусках финальное значение a после закрытия параллельной области разное: может получаться a=212, a=210 и т.п. Для наглядности в этой программе закомментирован вывод на печать внутри параллельной секции:

```
#include <omp.h>
#include <stdio.h>
int main() {
int a=199;
printf("in master-thread before parallel
block: a=%d\n",a);
#pragma omp parallel shared(a)
{
int id=omp_get_thread_num();
printf("begin of parallel block: a=%d,
id=%d\n",a,id);
a+=id;
/* printf("end of parallel block: a=%d,
id=%d\n", a, id); */
}
printf("in master-thread after parallel
block: a=%d\n",a);
return 0;
}
```

Таким образом, работа с общими данными требует внимательности и аккуратности. «Разруливать» ситуации, подобные представленной в примере 3, помогают директивы синхронизации нитей и опция редукции, которые будем изучать на следующих семинарах.

Задания:

1. Воспроизвести на кластере HybriLIT вышеприведенные примеры по работе с локальными и общими данными, убедившись, что объясненные выше закономерности поведения соответствующих переменных сохраняются.

2. Составить и запустить программу, выполняющую следующее. В мастер-нити объявляются целочисленные переменные a=1, b=10, c=100. Пусть c – `private`, b – `shared`, a – `firstprivate`. Пусть в параллельной области производятся следующие действия: c=a+id; b=b+a; a=22. Нужно вывести на печать значения a, b, c после закрытия параллельной области и объяснить полученный результат.

Семинар 11. Технология OpenMP: параллельное выполнение циклов

В OpenMP, наряду с поддержкой организации явного параллелизма путем использования нумерации нитей в параллельных блоках, существуют также возможности для автоматического распараллеливания на основе указаний компилятору с помощью соответствующих директив.

Директива `for` является указанием компилятору, что итерации данного цикла можно выполнить параллельно. Распределение итераций цикла между нитями возлагается на компилятор и зависит от количества нитей, выполняющих данный фрагмент кода, а также от количества итераций в цикле. В частности, применение директивы `for` в области, выполняемой одной нитью, не является ошибкой, однако в этом случае цикл будет выполнен в последовательном режиме нитью-мастером.

После директивы `for` обязательным условием является наличие в следующей строке соответствующего оператора цикла.

Варианты применения прагмы `omp for`:

1. Если параллельный регион создается только для распараллеливания цикла, можно совместить директивы `parallel` и `for` в одной строке:

```
#pragma omp parallel for
for(i=0; i<N;i++) {
//   begin   of   parallel   region   and   start
//parallel loop
…
} // end of loop and closing parallel region
```

2. Если в параллельной области, помимо цикла, присутствуют какие-либо еще вычисления, директива `for` применяется внутри параллельной области непосредственно перед оператором цикла:

```
#pragma omp parallel
{ // begin of parallel region
…
#pragma omp for
for (i=0; i<N;i++) { // start loop
…
} // end loop
…
} // closing parallel region
```

Пример 1

Рассмотрим применение прагмы `omp for` для поэлементного сложения двух массивов. В программе задаются значения элементов целочисленных массивов a и b длины N. Далее организуется параллельная область, в которой прагма `omp for` применяется для выполнения цикла. В цикле организуется на каждой итерации печать индекса i и номера нити, выполняющего эту итерацию. Соответственно, можно проследить, как компилятор распределил итерации между нитями.

```
// OMP: pragma for loop parallelization
#include <omp.h>
#include <stdio.h>
int main() {
int N = 20;
int a[N], b[N], c[N], i;

for (i = 0; i<N; i++){
// master-thread: initial values a and b
a[i] = i + 1;
b[i] = i - 1;
}
```

```
#pragma omp parallel firstprivate (a,b,N)
private (i)
{ // open parallel region

int id = omp_get_thread_num(); //id of thread

#pragma omp for
for (i = 0; i < N; i++){
c[i] = a[i] + b[i];
printf("Rank=%d calculates iteration i=%d
\n",id,i);
}
} // close parallel region

for (i = 0; i < N; i++){ // master-thread:
//final result
printf("i=%d, c[i]=%d \n", i, c[i]);
}
return 0;
}
```

Обращаем внимание, что в приведенном примере программы a, b, N определены в параллельной области как `firstprivate`, чтобы передать значения этих данных внутрь параллельной области. Можно было бы сделать эти переменные общими, но работа с локальными данными более «рентабельна» с точки зрения затрат компьютерного времени. Объявление `private(i)` дает возможность корректно использовать индекс i как в мастер-нити, так и в параллельной области. Массив с имеет тип `shared` по умолчанию.

Результат работы программы с числом нитей 6:

```
Rank=0 calculates iteration i=0
Rank=0 calculates iteration i=1
Rank=0 calculates iteration i=2
Rank=0 calculates iteration i=3
Rank=5 calculates iteration i=17
```

```
Rank=5 calculates iteration i=18
Rank=5 calculates iteration i=19
Rank=2 calculates iteration i=8
Rank=2 calculates iteration i=9
Rank=2 calculates iteration i=10
Rank=3 calculates iteration i=11
Rank=3 calculates iteration i=12
Rank=3 calculates iteration i=13
Rank=4 calculates iteration i=14
Rank=4 calculates iteration i=15
Rank=4 calculates iteration i=16
Rank=1 calculates iteration i=4
Rank=1 calculates iteration i=5
Rank=1 calculates iteration i=6
Rank=1 calculates iteration i=7
i=0, c[i]=0
i=1, c[i]=2
i=2, c[i]=4
i=3, c[i]=6
i=4, c[i]=8
i=5, c[i]=10
i=6, c[i]=12
i=7, c[i]=14
i=8, c[i]=16
i=9, c[i]=18
i=10, c[i]=20
i=11, c[i]=22
i=12, c[i]=24
i=13, c[i]=26
i=14, c[i]=28
i=15, c[i]=30
i=16, c[i]=32
i=17, c[i]=34
i=18, c[i]=36
i=19, c[i]=38
```

Видно, что нить с номером 0 выполнила итерации цикла для i=0, 1, 2, 3, нить с номером 1 посчитала элементы массива

c[N] с номерами 4, 5, 6, 7 и т.д. Таким образом, каждой нити достался в работу блок данных из трех или четырех элементов.

Если запустить программу с другим количеством нитей, то распределение итераций между нитями изменится, но конечный результат (значения массива c) не изменится.

Приведем некоторые полезные опции директивы `for`:

– `lastprivate` – если переменная объявлена `lastprivate`, внутри параллельной области она ведет себя как `private`, а после завершения цикла имеет значение, полученное на последней итерации;

– `nowait` – по завершении выполнения цикла позволяет нитям не ждать друг друга, а продолжать работу каждой нити сразу после выполнения назначенных ей итераций;

– `collapse` – позволяет ассоциировать с директивой `for` блок тесно-вложенных циклов, итерации которых в совокупности распределяются между нитями;

– `schedule` – регулирование распределения итераций цикла между нитями. Параметрами этой опции являются тип распределения (`static`, `dynamic`, `guided`, `auto` и др.) и количество итераций, назначаемых к выполнению каждой нити. Разберем подробнее работу этой опции.

Пример 2

Рассмотрим цикл из 12 итераций. Указываем число нитей 4 и распределение итераций типа `static`. Также в этом примере демонстрируется применение опции `lastprivate` относительно переменной x, куда сохраняется значение индекса i.

```
#include <omp.h>
#include <stdio.h>
int main(){
int N = 12;
int a[N], b[N], c[N], i, x;

for (i = 0; i < N; i++){//master-thread:
//initial values a and b
a[i] = i + 1;
b[i] = i - 1;
```

```
}
#pragma omp parallel firstprivate (a,b,N)
private (i) num_threads(4)
{
int id = omp_get_thread_num();

#pragma omp for lastprivate(x) schedule
(static)
for (i = 0; i < N; i++){
c[i] = a[i] + b[i];
x = i;
printf("Rank=%d calculates iteration i=%d
\n", id, i);
}
}
printf("x=%d\n", x);
return 0;
}
```

Из приведенного результата запуска программы видно, что каждая нить выполняет блок из трех итераций. При этом значение x=11 в конце программы соответствует номеру последней итерации цикла.

```
Rank=1 calculates iteration i=3
Rank=1 calculates iteration i=4
Rank=1 calculates iteration i=5
Rank=3 calculates iteration i=9
Rank=3 calculates iteration i=10
Rank=0 calculates iteration i=0
Rank=0 calculates iteration i=1
Rank=0 calculates iteration i=2
Rank=2 calculates iteration i=6
Rank=2 calculates iteration i=7
Rank=2 calculates iteration i=8
Rank=3 calculates iteration i=11
x=11
```

По умолчанию реализуется именно такое – статическое – распределение итераций между нитями (`static`), аналогичное «блочному» распределению итераций цикла, рассмотренному нами на семинарах по MPI.

Укажем теперь для распределения `static` порцию итераций, назначаемых каждой нити, равную 1:

```
#pragma omp for lastprivate(x) schedule (static,1)
```

Мы получили распределение итераций цикла между нитями, аналогичное рассмотренному нами ранее для MPI «циклическому» распараллеливанию. Действительно, нить с номером 0 выполняет расчеты для i=0, 4, 8; нить с номером 1 – для i=1, 5, 9 и т.д.

```
Rank=3 calculates iteration i=3
Rank=3 calculates iteration i=7
Rank=3 calculates iteration i=11
Rank=0 calculates iteration i=0
Rank=0 calculates iteration i=4
Rank=0 calculates iteration i=8
Rank=1 calculates iteration i=1
Rank=1 calculates iteration i=5
Rank=1 calculates iteration i=9
Rank=2 calculates iteration i=2
Rank=2 calculates iteration i=6
Rank=2 calculates iteration i=10
x=11
```

Укажем теперь:

```
#pragma omp for lastprivate(x) schedule (static,2)
```

В этом случае каждая нить выполняет по две итерации по возрастанию их номеров, затем еще по две и т.д. Таким образом, 0-нить выполняет расчеты для i=0, 1, затем для i=8, 9; 1-нить

выполняет итерации i=2, 3, затем i=10, 11; нитям с номерами 2 и 3 достаются соответственно итерации i=4, 5 и i=6, 7.

```
Rank=2 calculates iteration i=4
Rank=2 calculates iteration i=5
Rank=3 calculates iteration i=6
Rank=3 calculates iteration i=7
Rank=0 calculates iteration i=0
Rank=0 calculates iteration i=1
Rank=0 calculates iteration i=8
Rank=0 calculates iteration i=9
Rank=1 calculates iteration i=2
Rank=1 calculates iteration i=3
Rank=1 calculates iteration i=10
Rank=1 calculates iteration i=11
x=11
```

При использовании опций `dynamic` или `guided` каждая нить получает заданную порцию итераций и по мере их выполнения берет себе в работу следующие итерации по определенной схеме. При таком режиме количество итераций между нитями распределяется неравномерно и может оказаться, что большинство итераций успела выполнить одна нить (особенно, если число итераций невелико). При этом распределение итераций между нитями может меняться от запуска к запуску. Так, на приведенном здесь результате запуска нашей программы при использовании опции `dynamic`

```
#pragma omp for lastprivate(x) schedule (dynamic)
```

видно, что 9 из 12 итераций цикла выполнены нитью с номером 1:

```
Rank=1 calculates iteration i=3
Rank=1 calculates iteration i=4
Rank=1 calculates iteration i=5
Rank=1 calculates iteration i=6
Rank=1 calculates iteration i=7
Rank=1 calculates iteration i=8
```

```
Rank=1 calculates iteration i=9
Rank=1 calculates iteration i=10
Rank=1 calculates iteration i=11
Rank=2 calculates iteration i=0
Rank=0 calculates iteration i=2
Rank=3 calculates iteration i=1
x=11
```

Задания:

1. Воспроизвести на кластере HybriLIT вышеприведенные примеры по работе с прагмой `omp for`. Выполнить запуск программы из примера 1 с количеством нитей 4 и 5. Определить, сколько элементов массива c вычисляет каждая нить.
2. Составить и запустить OpenMP-программу, выполняющую следующее. Задается массив X из 15 целых чисел, каждое из которых равно 10. Организуется параллельная область, в которой с помощью прагмы `for` реализуется параллельное выполнение цикла. Внутри цикла к каждому элементу массива X прибавляется число, равное количеству параллельных нитей, и выводятся на печать номер этого элемента и номер нити. После завершения параллельного блока выводятся на экран результирующие значения массива X.

Семинар 12. Технология OpenMP: параллельное выполнение независимых фрагментов

Как мы уже выяснили на предыдущих семинарах, в OpenMP существуют, наряду с возможностями явного параллелизма с использованием номеров нитей в параллельных блоках, возможности для автоматического распараллеливания на основе указаний компилятору с помощью соответствующих директив.

Первая из таких возможностей – уже рассмотренная нами параллельная реализация циклов с помощью директивы `for`, которая указывает компилятору, что итерации данного цикла можно выполнить параллельно.

Вторая возможность – параллельное выполнение независимых фрагментов с использованием директивы `sections`. Особенности применения данной директивы – тема нынешнего семинара.

Возможности автоматического распараллеливания циклов и независимых фрагментов существенно упрощают параллельное программирование. При этом следует учитывать следующие обстоятельства:

- Компилятор не проверяет независимость итераций цикла или фрагментов кода, так что ответственность за такую проверку и корректное применение соответствующих директив полностью лежит на разработчике программы.
- Директивы `for` и `sections` нельзя применять совместно! Нельзя распараллеливать циклы внутри независимых фрагментов, определяемых прагмой `for`! Нельзя организовывать параллельное выполнение фрагментов с помощью прагмы `sections` внутри параллельного цикла!
- Чтобы добиться эффекта от параллелизма, вычислительная нагрузка на каждую OpenMP-нить должна быть высока. Поэтому в реальных программах не имеет смысла распараллеливать короткие циклы и\или короткие фрагменты с малым количеством вычислений.

Итак, директива `sections` используется для распределения независимых фрагментов (секций) программного кода между потоками, при этом каждый поток (нить) будет выполнять свою секцию. Выбор конкретной нити, которая будет выполнять данную секцию, осуществляется компилятором. Если секций больше, чем потоков, то несколько секций будут выполняться одной нитью по очереди.

Нижеприведенный фрагмент демонстрирует схему использования директивы `sections`. В параллельной области, организованной с помощью прагмы `parallel`, вся совокупность независимых фрагментов выделяется прагмой `sections`, а каждый из независимых фрагментов этой совокупности выделяется с помощью прагмы `section`.

```
#pragma omp parallel
{ //open parallel region
```

```
...
#pragma omp sections
{ // open sections block
#pragma omp section
{ // first section
 ...
 }
#pragma omp section
{ // second section
  ...
}
} // close sections pragma
...
} // close pragma parallel
```

Как и в случае директивы `for`, если параллельная область открывается только ради параллельного выполнения независимых секций, то прагмы `parallel` и `sections` можно применить совместно в одной строке:

```
#pragma omp parallel sections
```

Пример 1

Рассмотрим применение прагмы `omp sections` на примере параллельного выполнения трех коротких фрагментов. Пусть в программе задаются значения целочисленных переменных a=33 и b=55. В первом фрагменте вычисляем $c=a\cdot 10-b$, во втором – $d=b\cdot 2+a$, в третьем – $f=(a+b)/2$.

Каждый фрагмент выделяем прагмой `omp section`, а всю совокупность трех фрагментов – прагмой `omp sections`. Применяем совместно прагмы `parallel` и `sections` в одной строке. Переменные a и b объявляем `firstprivate`, переменные c, d, f – общие (`shared`) по умолчанию.

В каждом фрагменте печатаем номер нити, его выполняющей, и результат счета. После закрытия параллельной области мастер-нить печатает входные данные (a и b) и результаты (c, d, f), чтобы убедиться в правильности расчета и корректности пе-

редачи данных из параллельной области в нить-мастер. Программа выглядит следующим образом:

```
// OMP test: parallelization of independent
//fragments
#include <omp.h>
#include <stdio.h>
int main() {

 int a=33, b=55, c, d, f;

#pragma omp parallel sections firstprivate
(a,b)
{ // open parallel region and the
// sections-region

#pragma omp section
{ // open 1st section
int id = omp_get_thread_num();
c = a * 10 - b;
printf("id=%d; c=%d\n", id, c);
} // close 1st section

#pragma omp section
{ // open 2nd section
int id = omp_get_thread_num();
d = b * 2 + a;
printf("id=%d; d=%d\n", id, d);
} // close 2nd section

#pragma omp section
{ // open 3rd section
int id = omp_get_thread_num();
f = (a + b) / 2;
printf("id=%d; f=%d\n", id, f);
} // close 3rd section

}// close parallel region and sections-region
```

```
// print final result in the master thread
printf("master-thread:   a=%d,   b=%d,   c=%d,
d=%d, f=%d\n",a,b,c,d,f);
return 0;
}
```

Видно, что при запуске с количеством нитей 6 в расчете оказались задействованы три нити с номерами 2, 3, 4:

```
id=4; c=275
id=2; d=143
id=3; f=44
In the master-thread: a=33, b=55, c=275,
d=143, f=44
```

Номера нитей, выполняющих параллельные фрагменты, могут меняться от запуска к запуску. Если количество фрагментов больше, чем количество нитей, то компилятор группирует фрагменты при их назначении в работу имеющимся нитям.

Пример 2

Рассмотрим применение `lastprivate`-переменных в `sections`-областях. Такие переменные ведут себя в параллельной области как локальные. При этом после завершения действия прагмы `sections` им присваиваются значения, полученные в последней секции.

```
#include <omp.h>
#include <stdio.h>
int main() {
int x=0;
#pragma omp parallel
{
int id = omp_get_thread_num();
#pragma omp sections lastprivate (x)
{
```

```
#pragma omp section
{ // open 1st section
printf("Sect.1: id=%d\n",id);
x=id+1;
} // close 1st section
#pragma omp section
{ // open 2nd section
printf("Sect.2: id=%d\n",id);
x=id+1;
} // close 2nd section
#pragma omp section
{ // open 3rd section
printf("Sect.3: id=%d\n",id);
x=id+1;
} // close 3rd section
}
printf(" id=%d, x=%d\n", id,x);
}
printf("In the master-thread:  x=%d \n", x);
return 0;
}
```

В данной программе общей переменной x, имеющей значение 0 перед открытием параллельной области, в каждой из трех параллельных секций присваивается значение на 1 больше номера нити, выполняющей эту секцию. После закрытия `sections`-области во всех нитях x=1, что соответствует значению, присвоенному переменной x в последней (третьей) секции, выполненной нитью 0.

После закрытия параллельной области значение x=1 сохраняется в мастер-нити.

```
Sect.3: id=0
Sect.2: id=2
Sect.1: id=1
 id=0, x=1
 id=2, x=1
 id=5, x=1
```

```
 id=1, x=1
 id=4, x=1
 id=3, x=1
In the master-thread:  x=1
```

Задания:

1. Воспроизвести на кластере HybriLIT вышеприведенные примеры по работе с прагмой `omp sections`.
2. Написать OpenMP-программу, выполняющую следующее:
   - Задаются целые переменные a=1, b=10, c=1000. Организуется параллельная область с количеством нитей 3.
   - В параллельной области организуется `sections`-область, состоящая из пяти фрагментов: в первом фрагменте вычисляется x=a+b+c; во втором – y=(a+b)*c; в третьем – z=c/a–b; в четвертом – v=a*10+b*3; в пятом – w=a–b–c.
   - В каждом фрагменте напечатать номер нити, которая его выполняет, и результат расчета. После завершения параллельного блока напечатать значения x, y, z, v, w.

Семинар 13. Технология OpenMP: синхронизация нитей в параллельных областях

Директивы синхронизации обеспечивают возможность организации явной синхронизации работы нитей в параллельной области (прагма `omp barrier`), а также синхронизации путем назначения специального режима для выполнения фрагмента, выделенного внутри параллельной секции. Например, такой фрагмент может быть выполнен только одной нитью или всеми нитями, но по очереди.

Директивы данного класса позволяют обеспечить корректную работу с общими переменными, а также организовать однократное выполнение выделенного фрагмента без закрытия параллельной области.

1. Однократное выполнение фрагментов в параллельных областях

Директива `single` указывает на однократное выполнение выделенного внутри параллельной области блока произвольной нитью.

```
#pragma omp parallel
{ …
#pragma omp single
{ … }
…
}
```

Выделенный здесь `single`-фрагмент будет выполнен однократно, нитью с произвольным номером.

Пример 1

Рассмотрим пример применения прагмы `single` для однократного выполнения какой-либо нитью операции увеличения значения общей целочисленной переменной а на 10. Остальные нити пропускают выделенный фрагмент. Начальное значение переменной а=0. В параллельной области организована печать до прагмы `single`, внутри области, выделенной прагмой `single`, и после закрытия `single`-фрагмента.

```
//  OMP test: pragma single
#include <omp.h>
#include <stdio.h>
int main() {
int a=0;
#pragma omp parallel
{ // open parallel region
int id = omp_get_thread_num();
printf("Before  single:  id=%d;  a=%d\n",  id,
a);
#pragma omp single
{ // open single-section
a += 10;
```

```
printf("In  single-section:  id=%d;  a=%d\n",
id, a);
} // close single-section
printf("After single: id=%d; a=%d\n", id, a);
}  // close parallel region
// print final result in the master thread
printf("In the master-thread: a=%d \n", a);
return 0;
}
```

При выполнении программы увеличение общей переменной a на 10 выполняет только одна нить, так что в конечном итоге получаем a=10. Видно, что все нити печатают a=0 при открытии параллельной области (строки “before single…”), печать из `single`-секции осуществляется однократно (в нашем случае оказалось, что это нить с номером 5), затем после закрытия `single`-секции все нити печатают новое значение a=10 (строки (“after single…”):

```
Before single: id=5; a=0
In single-section: id=5; a=10
Before single: id=0; a=0
Before single: id=2; a=0
Before single: id=3; a=0
Before single: id=1; a=0
Before single: id=4; a=0
After single: id=5; a=10
After single: id=4; a=10
After single: id=1; a=10
After single: id=2; a=10
After single: id=0; a=10
After single: id=3; a=10
In the master-thread: a=10
```

Директива `master`, подобно `single`, также указывает на однократное выполнение блока, однако выделенный блок будет выполнен не произвольной нитью, как в случае примене-

ния директивы `single`, а мастер-нитью, имеющей внутри параллельной области номер 0.

```
#pragma omp parallel
{ …
#pragma omp master
{ … }
…
}
```

Выделенный фрагмент будет выполнен однократно, всегда нитью с номером 0. В этом можно убедиться, заменив в предыдущем примере `single` на `master`.

2. Последовательное выполнение фрагментов в параллельных областях

Директивы `critical` и `atomic` выделяют фрагмент в параллельной области, выполняемый всеми нитями, задействованными в данной параллельной области, однако в каждый конкретный момент критический фрагмент выполняется только одной нитью, т.е. когда одна нить выполняет содержимое критической секции, остальные нити, готовые к выполнению критической секции, находятся в режиме ожидания.

```
#pragma omp parallel
{ …
#pragma omp critical
{ … }
…
}
```

Выделенный критический фрагмент будет выполнен всеми нитями, но по очереди. Очередность выполнения нитями критической секции не определена и может отличаться от запуска к запуску. Заменив в предыдущем примере `single` на `critical`, можно на практике увидеть, как работает эта директива. В этом случае каждая нить прибавит 10 к общей переменной a, так что итоговое значение этой переменной будет не 10 (как при использовании директив `single` или `master`),

а 60 при запуске для шести параллельных нитей. Данная директива позволяет обеспечить корректную работу с общими данными при решении ряда задач.

Директива `atomic` работает аналогично директиве `critical`, только она относится к непосредственно следующему за ней оператору.

```
int count=0;
#pragma omp parallel
{
…
#pragma omp atomic
count++;
…
}
```

Здесь каждая нить увеличивает значение общей переменной count на 1, в результате переменная count примет значение, равное количеству нитей. Отметим, что в языках C и C++ имеются ограничения на операторы, которые могут использоваться в `atomic`-секции.

Директива `ordered` – еще одна директива для выделения критических секций, она используется только в параллельных циклах. В директиве `for` при этом должна быть указана опция `ordered`. Прагма `ordered` обеспечивает упорядоченное выполнение `ordered`-блока всеми нитями по очереди, по порядку их номеров. Внутри `ordered`-секции в каждый момент времени может находиться только один поток. Данная прагма может, например, использоваться для упорядочения печати в цикле.

Пример 2

Применение директивы `ordered`.

```
#include <omp.h>
#include <stdio.h>
int main(){
int N=10, i, id;
```

```
#pragma omp parallel private(id)
{
id = omp_get_thread_num();
#pragma omp for private(i) ordered
for(i=0; i<N; i++)
{
printf("myid=%d; i=%d\n", id, i);
#pragma omp ordered
{
printf("ordered myid=%d; i=%d\n", id, i);
}
}
}
return 0;
}
```

В приведенной программе с помощью прагмы `ordered` реализуется печать каждой нитью выполняемых ею номеров итераций цикла, в порядке возрастания номеров нитей. Прагма `for` имеет опцию `ordered`. В цикле присутствуют два оператора печати: вне `ordered`-секции и из `ordered`-секции (эта строка начинается со слова “`ordered`”). Видно, что печать вне `ordered`-секции – неупорядоченна, а строки, начинающиеся со слова “`ordered`”, выводятся на экран по порядку возрастания индекса i:

```
myid=5; i=9
myid=3; i=6
myid=2; i=4
myid=0; i=0
ordered myid=0; i=0
myid=0; i=1
ordered myid=0; i=1
myid=1; i=2
ordered myid=1; i=2
myid=1; i=3
ordered myid=1; i=3
ordered myid=2; i=4
```

```
myid=2; i=5
ordered myid=2; i=5
ordered myid=3; i=6
myid=3; i=7
ordered myid=3; i=7
myid=4; i=8
ordered myid=4; i=8
ordered myid=5; i=9
```

Примечание. К директивам синхронизации относятся также прагма барьерной синхронизации (`barrier`) и прагма `flush`, о них написано в учебном пособии по OpenMP.

Задания:

1. Воспроизвести на кластере HybriLIT вышеприведенные примеры по применению прагмы `omp single` и `ordered`.
2. Заменить в программе из Примера 1 директиву `omp single` на `omp master`. Прокомментировать результат запуска.
3. Заменить в этой же программе директиву `omp single` на `omp critical`. Прокомментировать результат запуска и получаемое конечное значение a.

Семинар 14. Технология OpenMP: операция редукции

Как мы уже знаем, директива синхронизации `critical` обеспечивает корректную работу с общими переменными в случае, если требуется, чтобы каждая нить выполнила какую-либо операцию над общими данными. Другой вариант организации корректной работы с общими данными – операции редукции.

Опция `reduction` прагмы `parallel` задает операцию редукции (сложение, умножение и т.п.) и список переменных редукции, над которыми указанная операция производится.

Операция редукции осуществляется при закрытии параллельной секции, над теми значениями переменных редукции, которые они имеют к моменту закрытия параллельной области в каждой нити. Рассмотрим, как работает редукция в OpenMP, на конкретном примере.

Пример 1. Пример применения редукции в OpenMP

```
...
int nCount=10;
#pragma omp parallel reduction (+ : nCount)
{
...
nCount+=1;
...
}
...
```

В данном фрагменте целочисленная переменная nCount объявлена переменной редукции, а операцией редукции является сложение.

Если переменная объявлена переменной редукции, то ее нельзя объявлять локальной или общей, это будет ошибкой. Логика работы с этой переменной следующая. При входе в параллельную область для каждой переменной редукции создаются локальные копии с тем же именем в каждой нити. Начальные значения этих локальных переменных в каждой нити определяются типом операции редукции и никак не связаны со значением переменной с этим именем до начала параллельной области. Так, для операции сложения начальное значение переменной редукции 0, для умножения – 1 и т.д.

В нашем примере, поскольку операцией редукции является сложение, начальные значения локальных переменных nCount в каждой нити параллельной области равны 0, несмотря на то что перед началом параллельной области мы имели nCount=10.

Итак, при входе в параллельную область каждая нить имеет начальное значение переменной редукции nCount=0. Далее, каждая нить увеличивает свою переменную nCount на единицу, т.е. при закрытии параллельной области каждая нить имеет nCount=1.

При закрытии параллельной области над всеми переменным nCount из каждой нити осуществляется операция редукции (в нашем случае – сложение). Все переменные nCount=1 из всех нитей складываются между собой, и эта сумма прибавляется

к тому значению nCount, которое эта переменная имела до объявления параллельной области, т.е. nCount=10.

Таким образом, после закрытия параллельной области будем иметь значение переменной nCount, равное 10+количество нитей, что мы можем видеть при запуске нижеследующего кода с количеством нитей, равным 6.

```
// OMP test. reduction option
#include <omp.h>
#include <stdio.h>
int main(){
int nCount=10;
#pragma omp parallel reduction (+: nCount)
{ // open parallel region
nCount++;
} // close parallel region
printf("after reduction: nCount=%d \n", nCount);
return 0;
}
```

Результат работы программы для 6 нитей:

```
after reduction: nCount=16
```

В необходимости использования опции редукции для корректной работы с общими переменными можно убедиться, если убрать эту опцию и задать достаточно большое количество нитей. В этой ситуации есть вероятность, что одна или более нитей обратится к общему для всех адресу nCount в момент, когда это значение еще не обновлено по результатам работы других нитей, так что результат окажется меньше, чем 10+количество нитей, и может меняться от запуска к запуску. Это хорошо видно на количестве нитей 40: при многократных запусках время от времени получаем nCount=49, nCount=47 и даже nCount=45 вместо nCount=50.

```
[@hydra OMP_examples]$ ./a.out
after reduction: nCount=50
```

```
[@hydra OMP_examples]$ ./a.out
after reduction: nCount=45
[@hydra OMP_examples]$ ./a.out
after reduction: nCount=50
[@hydra OMP_examples]$ ./a.out
after reduction: nCount=49
[@hydra OMP_examples]$ ./a.out
after reduction: nCount=50
[@hydra OMP_examples]$ ./a.out
after reduction: nCount=50
[@hydra OMP_examples]$ ./a.out
after reduction: nCount=50
[@hydra OMP_examples]$ ./a.out
after reduction: nCount=47
[@hydra OMP_examples]$ ./a.out
after reduction: nCount=50
```

Пример 2. Параллельная реализация вычисления суммы ряда

Применим полученные знания о технологии OpenMP для параллельного вычисления суммы ряда

$$c = \sum_{i=0}^{\infty} \frac{1}{x_i^2} + \frac{1}{x_i^4}, \qquad x_i = i + 1.$$

Эту задачу мы уже рассматривали на семинаре по MPI. OpenMP-программа имеет вид:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define n 1000000000
int main(int argc, char* argv[]){
double *x;
x = (double*)malloc(n*sizeof(double));
//initialize double x[n];

int i;
double c=0, xx;
```

```
double time1 = omp_get_wtime(); // first time
//point
#pragma omp parallel for reduction (+: c)
private (i)
for (i = 0; i < n; i++){ // loop operator
x[i] = 1.0+i; // calculation of i iteration
xx = x[i]*x[i];
c+=1.0/xx+1.0/(xx*xx);      //each      thread
//calculate partial sum
}
double  time2  =  omp_get_wtime();  //  second
//time point
// 0-thread prints result and execution time
printf ("Sum = %f; execution time=%f\n ", c,
time2-time1);
free(x);
return 0;
}
```

Видно, что общая структура программы сохраняется, но необходимо убрать все конструкции MPI, добавить в нужных местах конструкции OpenMP, а вместо явного распараллеливания основного цикла использовать прагму `parallel for` и опцию `reduction` относительно переменной "c" для корректного суммирования.

Для засечек времени (time1, time2) используется функция `omp_get_wtime()`.

При расчете на 10 нитях для количества членов ряда n=1000000000 имеем следующий результат работы программы (печатается вычисленная сумма ряда и время счета).

```
Sum = 2.727257; execution time=14.783209
```

Задания:

1. Воспроизвести на кластере HybriLIT вышеприведенные примеры по применению опции `reduction`.

2. Заменить в программе из примера 1 операцию сложения на операцию умножения. Прокомментировать результат запуска.
3. Для программы из примера 2 убедиться, что сумма ряда вычисляется корректно. Выполнить серию запусков с увеличивающимся количеством нитей с количеством нитей 1, 2, 3, 4, 5, 6… Найти оптимальное количество нитей, обеспечивающих наименьшее время счета для этой задачи.

Семинар 15. Контрольная работа по технологии OpenMP

На семинарах изучены конструкции технологии OpenMP:

– для организации параллельных областей;
– регулирования количества нитей в параллельных областях;
– определения номера нити и количества нитей в параллельной области;
– параллельной реализации выполнения циклов;
– параллельной реализации выполнения независимых фрагментов;
– синхронизации нитей в параллельных областях и реализации операций редукции.

Контрольная работа состоит в самостоятельном написании, отладке и запуске параллельной программы по индивидуальному для каждого студента заданию с использованием пройденных на семинарах конструкций OpenMP.

Примеры заданий для контрольной работы по OpenMP

1. Написать OpenMP-программу, выполняющую следующее.

Устанавливается число нитей в параллельных фрагментах, равное 4, с помощью функции `omp_set_num_threads`. Внутри параллельного блока, организованного с помощью прагмы `omp parallel`, каждая нить печатает свой номер и количество нитей.

Организуется `single`-секция с помощью прагмы `omp single`, внутри которой выводится на экран слово “Hello single” и номер нити, которая выполнила `single`-секцию.

2. Написать OpenMP-программу, выполняющую следующее.

Задается массив B из 30 целых чисел, каждое из которых равно 10. Устанавливается число нитей в параллельных фрагментах, равное 3, с помощью функции `omp_set_num_threads`.

Внутри параллельного блока с помощью прагмы `omp for` организуется параллельное выполнение цикла, в котором к каждому элементу массива B прибавляется число, равное количеству параллельных нитей, и выводится на печать номер этого элемента и номер нити.

После завершения параллельного блока выводятся на экран результирующие значения массива B.

3. Написать OpenMP-программу, выполняющую следующее.

Задается целая переменная b=50. Устанавливается число нитей в параллельных фрагментах, равное 3, с помощью функции `omp_set_num_threads`.

Внутри параллельного блока с использованием опции `reduction` организуется следующее: каждая нить уменьшает свое значение b на 10, и по завершении параллельного фрагмента все значения b складываются между собой. После завершения параллельного блока выводится на экран результирующее значение переменной b.

Список рекомендованной литературы

1. Степанов, А. Н. Архитектура вычислительных систем и компьютерных сетей. – Санкт-Петербург : Питер, 2007. – 509 с.

2. Столлингс, В. Структурная организация и архитектура компьютерных систем: Проектирование и производительность. – 5. изд. – Москва : Вильямс, 2002. – 892 с.

3. Таненбаум, Э. Архитектура компьютера / Э. Таненбаум, Т. Остин. – 6. изд. – Санкт-Петербург : Питер, 2022. – 816 с.

4. Воеводин, В. В. Параллельные вычисления : учебное пособие В. В. Воеводин, В. В. Воеводин. – Санкт-Петербург : Изд-во БХВ-Петербург, 2015. – 603 с.

5. Башашин, М. В. 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. – 52 с.

6. Башашин, М. В. Основы технологии OpenMP на кластере HybriLIT : учебное пособие / М. В. Башашин, Е. В. Земляная, О. И. Стрельцова. – Дубна : Гос. ун-т «Дубна», 2020. – 52 с.

7. Гетерогенная вычислительная платформа HybriLIT. – URL: http://hlit.jinr.ru/.

8. Adam Gh. et al. IT-ecosystem of the HybriLIT heterogeneous platform for high-performance computing and training of IT-specialists // CEUR Workshop Proceedings. – 2018. – V. 2267. – P. 638–644.

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

Окончание табл.

<table>
<tr><th>Команда</th><th>Описание</th></tr>
<tr><td><<i>имя_файла</i>> | grep < <i>фрагмент</i>></td><td>поиск фрагмента текста в <<i>имя_файла</i>></td></tr>
<tr><td>grep <<i>User_name</i>></td><td>отобразить все процессы, запущенные от <i>User_name</i></td></tr>
<tr><td>kill</td><td>принудительное завершение работы приложения (процесса)</td></tr>
<tr><td>killall <<i>имя_программы</i>></td><td>принудительное завершение работы всех процессов по имени программы</td></tr>
</table>

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 задач в режиме очереди: отправляет задачу для последующего выполнения (скрипт-файл обычно содержит одну или несколько команд srun для запуска параллельных задач)
scancel <ID-job>	команда для удаления задачи из очереди

Приложение 3: Инструкция по работе на кластере hydra.uni-dubna.ru и запуску задач в интерактивном и пакетном режимах

1. Открыть программу Putty и в поле host ввести адрес сервера, с которым вы хотите соединиться. В нашем случае это **hydra.uni-dubna.ru**.
2. Если вы соединяетесь с данной машиной впервые, согласитесь на добавление ключа в систему.
3. В поле “password” введите пароль вашей учетной записи. **ВАЖНО!!! Никаких признаков ввода пароля на экране отображаться не будет, но поверьте: ввод пароля будет происходить.** За логином\паролем следует обращаться к преподавателям семинаров.
4. Прежде чем приступить к компиляции программ, убедитесь, что подключен модуль, содержащий компилятор. Для этого воспользуйтесь командой **module list** и проверьте наличие модуля **openmpi/v4.1.2** для компиляции **MPI-**программ. Если модуль не подключен, подключите его с помощью команды:

   module add openmpi/v4.1.2

5. Для написания программ с помощью текстового редактора **nano** наберите команду

   nano <file.c>

   (здесь и ниже **<file.c>** – имя вашего файла с кодом программы на языке **C**).
6. Для компиляции **MPI**-программ, написанных на языке программирования **C**, воспользуйтесь командой

   mpicc <file.c>

   Для компиляции **MPI-**программ, написанных на языке **C++**, воспользуйтесь командой

   mpic++ <file.cpp>

   (здесь и ниже **<file.cpp>** имя вашего файла с кодом на **C++**). После компиляции убедитесь в наличии исполняемого файла с именем **a.out**.

7. Для компиляции **OpenMP**-программ (подключение дополнительного модуля не требуется), написанных на **C**, воспользуйтесь командой:

   gcc -fopenmp <file.c>

   Для компиляции **OpenMP-**программ, написанных на **С++**, воспользуйтесь командой

   g++ -fopenmp <file.cpp>

   После компиляции убедитесь в наличии файла с именем **a.out**.
8. Интерактивный запуск **MPI-**программ осуществляется командой

   mpirun -n <x> ./a.out

   (здесь <x> – число **MPI**-процессов, выполняющих вашу задачу). Интерактивный запуск **OpenMP-**программ осуществляется командой

   ./a.out
9. Для запуска программ в режиме очередей убедитесь в наличии файлов, содержащих скрипты для запуска соответствующей программы.

Скрипт для запуска MPI-программ:

```
#!/bin/sh
#SBATCH -p tut
#SBATCH -n 4
#SBATCH -t 5
mpirun ./a.out
```

Скрипт для OpenMP-программ:

```
#!/bin/sh
#SBATCH -p tut
#SBATCH -c 4
#SBATCH -t 5
export OMP_NUM_THREADS=4
export OMP_PLACES=cores
./a.out
```

Ключ **-p** в скриптах означает имя очереди. На кластере **hydra.uni-dubna.ru** для **MPI**- и **OpenMP-**задач предусмотрена очередь с именем “**tut**”. Ключ **-n** предназначен для указания количества **MPI-**процессов, ключ -**t** – для указания времени в (секундах), ключ **-c** устанавливает количество ядер для выполнения **OpenMP**-задач. Предпоследняя строка в **OpenMP-**скрипте устанавливает, что каждая нить выполняется на отдельном ядре.

ВАЖНО!!! Не рекомендуется задавать количество вычислительных ядер больше 8.
ВАЖНО!!! Не рекомендуется прямое копирование скриптов из данного файла в файл на кластере: из-за различий в кодировках OS LINUX и MS Word специальные и невидимые символы окажутся некорректными.

10. Для запуска скрипта воспользуйтесь командой

    sbatch <scriptname.sh>

    (здесь **<scriptname.sh>** – имя вашего скрипта). При корректном запуске вы получите идентификатор вашей задачи **<id>**.
11. С помощью команды

    squeue

    можно посмотреть список запущенных на счет задач.
12. С помощью команды

    scancel <id>

    можно при необходимости прервать выполнение задачи.
13. Результаты выполнения задачи в batch-режиме не выводятся на экран, а записываются в файл с именем **slurm-<id>.out**. Для просмотра результатов выполнения вашей задачи воспользуйтесь командой

    cat slurm-<id>.out

Учебное издание

Земляная Елена Валериевна
Башашин Максим Викторович

Введение в параллельное программирование на основе технологий MPI и OpenMP

УЧЕБНОЕ ПОСОБИЕ

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

Подписано в печать . Формат 60×84/16. Усл. печ. л. 6,12.
Тираж 24 экз. Заказ №.

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