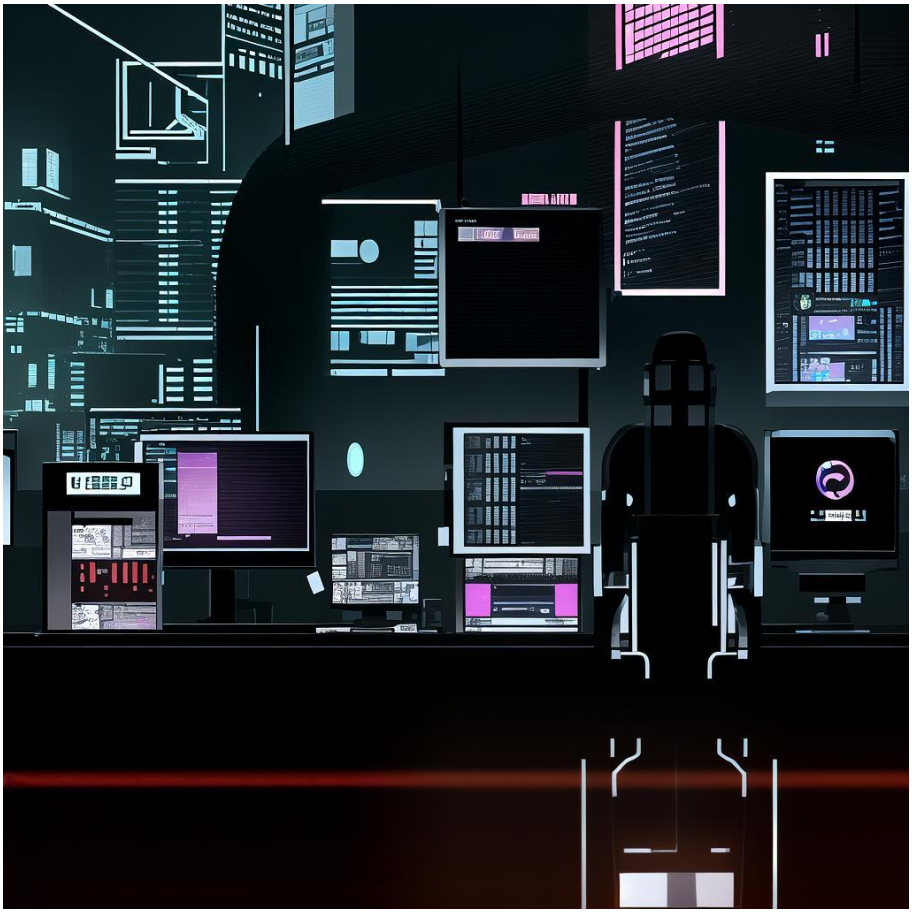
А. Б. БОРИСЕНКО, И. Л. КОРОБОВА, А. Ю. СВЕШНИКОВ

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ С ПРИМЕНЕНИЕМ MPI



Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2025

Министерство науки и высшего образования Российской Федерации

**Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Тамбовский государственный технический университет»**

А. Б. БОРИСЕНКО, И. Л. КОРОБОВА, А. Ю. СВЕШНИКОВ

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ С ПРИМЕНЕНИЕМ MPI

Утверждено Ученым советом университета
в качестве учебного пособия для студентов направлений подготовки
09.03.01 «Информатика и вычислительная техника»,
изучающих дисциплину «Разработка информационного обеспечения»,
09.03.03 «Прикладная информатика», изучающих дисциплину
«Распределенная и параллельная обработка информации»,
а также для студентов направления подготовки
09.04.01 «Информатика и вычислительная техника»,
изучающих дисциплину «Введение в большие данные
и анализ информации», очной и заочной форм обучения

Учебное электронное издание

**Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2025**

УДК 004.75
ББК 32.973.3
Б82

Рецензенты:

Кандидат технических наук, доцент, доцент кафедры
«Математическое моделирование и информационные технологии»
Института новых технологий и искусственного интеллекта
ФГБОУ ВО «ТГУ имени Г. Р. Державина»
Д. С. Соловьев

Кандидат технических наук, доцент, доцент кафедры
«Информационные процессы и управление» ФГБОУ ВО «ТГТУ»
А. А. Третьяков

Борисенко, А. Б.

Б82 Распределенные вычисления с применением MPI [Электронный ресурс] : учебное пособие / А. Б. Борисенко, И. Л. Коробова, А. Ю. Свешников. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2025. – 1 электрон. опт. диск (CD-ROM). – Системные требования : ПК не ниже Pentium IV ; CD-ROM-дисковод ; 1,7 Мб ; RAM ; Windows 95/98/XP ; мышь. – Загл. с экрана.

ISBN 978-5-8265-2871-6

Знакомит учащихся с основами распределенных вычислений и применением стандарта MPI (Message Passing Interface) для написания эффективных программ на языках C и Python. Приводятся примеры параллельных MPI-программ, а также варианты индивидуальных заданий для выполнения лабораторных работ. Представленные примеры демонстрируют, как с помощью MPI создавать масштабируемые приложения, которые можно легко адаптировать под различные распределенные вычислительные системы, начиная с небольших кластеров и заканчивая крупными суперкомпьютерами.

Предназначено для студентов направлений подготовки 09.03.01 «Информатика и вычислительная техника», изучающих дисциплину «Разработка информационного обеспечения», 09.03.03 «Прикладная информатика», изучающих дисциплину «Распределенная и параллельная обработка информации» 09.04.01, а также для студентов направления подготовки 09.04.01 «Информатика и вычислительная техника», изучающих дисциплину «Введение в большие данные и анализ информации», очной и заочной форм обучения.

УДК 004.75
ББК 32.973.3

ISBN 978-5-8265-2871-6

ВВЕДЕНИЕ

Решение многих прикладных задач требует значительных вычислительных ресурсов. В качестве примеров таких задач можно привести задачи тепло- и массопереноса, задачи аэро- и гидродинамики, задачи комбинаторной оптимизации и др. Одним из эффективных способов ускорения вычислений является распределение решения задачи между несколькими вычислительными узлами для параллельной обработки.

MPI (Message Passing Interface, [1]) является стандартом для создания параллельно выполняющихся программ, основанным на передаче сообщений между процессами (сами процессы при этом могут выполняться как на одном, так и на различных вычислительных узлах). Формально MPI-подход основан на включении в программные модули вызовов функций специальной библиотеки и загрузчика параллельно исполняемого кода на вычислительные узлы. Существует несколько популярных реализаций библиотеки MPI, перечисленные ниже.

MPICH (Message Passing Interface Chameleon) – это одна из самых старых и популярных реализаций MPI, разработанная для обеспечения высокой производительности и совместимости с различными операционными системами [2]. MPICH является референсной реализацией стандарта MPI и используется как в образовательных и исследовательских целях, так и в промышленных кластерах.

OpenMPI – это высокопроизводительная реализация библиотеки MPI с открытым исходным кодом [3]. Она разрабатывается как коллективный проект нескольких организаций и поддерживает большинство операционных систем и архитектур. OpenMPI отличается высокой производительностью и модульной архитектурой, что позволяет ей легко настраиваться и адаптироваться к требованиям различных вычислительных систем. Библиотека широко используется в академических и исследовательских кластерах.

Microsoft MPI (MS-MPI) – это реализация MPI для операционных систем семейства Windows от компании Microsoft [4].

Intel MPI – это коммерческая реализация MPI от компании Intel, оптимизированная для работы на кластерах на базе оборудования Intel [5]. Существуют реализации как под Windows, так и под Linux.

Несмотря на то, что каждая из этих реализаций библиотеки MPI имеет свои специфические параметры установки и настройки, для программиста их использование при написании и запуске программ практически одинаково, так как все они поддерживают один и тот же стандарт MPI. Например, независимо от того, используется MPICH, OpenMPI или Intel MPI, для компиляции программы с использованием MPI применяется команда mpicc, а для запуска – команда mpiexec. Кроме того, MPI подходит для создания приложений, которые легко масштабируются на кластерах с большим числом узлов. Благодаря поддержке низкоуровневого управления процессами и передаче данных, MPI позволяет создавать программы, которые можно запускать как на нескольких, так и на тысячах вычислительных узлов, эффективно распределяя нагрузку.

1. ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ И ПАРАЛЛЕЛЬНЫЕ ПРОГРАММЫ

MPI (Message Passing Interface) MPI – стандарт для обмена сообщениями между процессами в распределенных вычислительных системах. MPI-программа на этапе выполнения представляет собой множество параллельных взаимодействующих процессов. Каждый процесс идентифицируется уникальным номером (рангом), что позволяет варьировать логику работы программы. Независимо от того, на каком реальном вычислительном устройстве выполняется программа, технология MPI реализует модель распределенной памяти (distributed memory model): предполагается, что каждый процесс имеет свою локальную память. MPI используется для разработки параллельных приложений, работающих на кластерах и суперкомпьютерах.

1.1. КЛАССИФИКАЦИЯ ПО КОЛИЧЕСТВУ ПОТОКОВ КОМАНД И ДАННЫХ

Один из первых способов разделения архитектур по критерию множественности потоков команд (инструкций) и потоков данных был предложен Майклом Флинном (Michael J. Flynn). Поток (Stream) определяется как последовательность команд или данных, выполняемых или обрабатываемых процессором. С этой точки зрения программа предоставляет процессору поток команд (Instruction stream) для исполнения, при этом данные для обработки поступают также в виде потока (Data stream). При этом предполагается, что потоки команд и потоки данных независимы друг от друга. В связи с этим вычислительные системы разделяют на следующие классы (рис. 1).

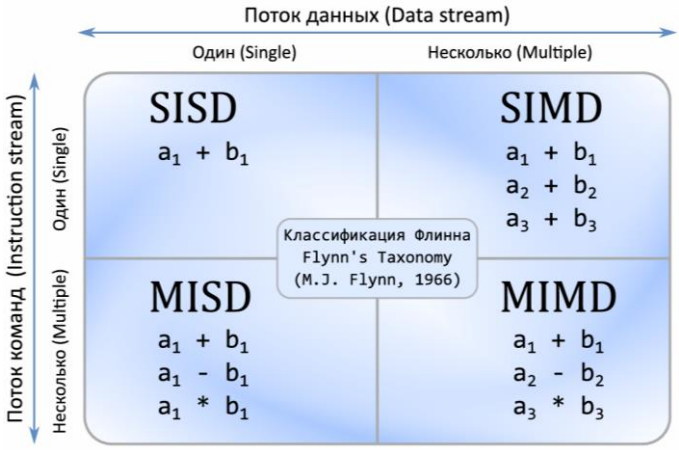


Рис. 1. Классификация Флинна

SISD – Single Instruction, Single Data

Вычислительная система SISD (один поток команд, один поток данных) представляет собой однопроцессорную машину, способную выполнять одну команду, работающую с одним потоком данных. Вычислительные системы, использующие эту модель, обычно называются последовательными. Такими характеристиками, например, обладали первые компьютеры с одноядерными процессорами.

SIMD – Single Instruction, Multiple Data

Вычислительная система SIMD (один поток команд, несколько потоков данных) – это многопроцессорная машина, способная выполнять одну и ту же команду на всех процессорах, но работающая с несколькими потоками данных. К данному классу относят, например, векторные вычислительные системы, в которых одна команда может выполняться над множеством элементов данных, графические процессоры. Сюда можно также отнести наборы векторных инструкций, например SSE, AVX и др.

MISD – Multiple Instruction, Single Data

Вычислительная система MISD (несколько потоков команд, один поток данных) представляет собой многопроцессорную машину, способную выполнять разные инструкции на разных процессорах, но все они работают с одним и тем же набором данных. Широкого практического применения системы MISD не нашли.

MIMD – Multiple Instruction, Multiple Data

Вычислительная система MIMD (несколько потоков команд, несколько потоков данных) – это многопроцессорная машина, способная выполнять несколько команд с разными наборами данных. Данный тип вычислительных систем представляет наиболее обширную и разнообразную группу в классификации Флинна. Большинство современных многопроцессорных вычислительных систем, в том числе вычислительные кластеры, относится именно к этому классу.

1.2. КЛАССИФИКАЦИЯ ПО ПРИНЦИПУ ОРГАНИЗАЦИИ ПАМЯТИ

Для целей параллельного программирования принципиальным моментом является принадлежность архитектуры используемой вычислительной системы к одному из следующих вариантов: системы с общей памятью, системы с распределенной памятью и гибридные системы, представляющие комбинацию общей и распределенной архитектур памяти.

В системах с общей памятью (рис. 2) все процессоры имеют доступ к общему адресному пространству, что позволяет процессорам обмениваться данными напрямую через общую память без необходимости явной передачи сообщений.

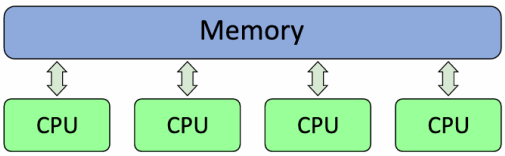


Рис. 2. Системы с общей памятью (Shared Memory Systems)

Общая память может быть реализована несколькими способами, включая симметричные мультипроцессорные системы (Symmetric MultiProcessing – SMP), многопроцессорные системы с неоднородным доступом к памяти (Non-uniform memory access – NUMA) и кеш-когерентные системы с распределенной разделяемой памятью (Cache coherent NUMA – ccNUMA).

В системах с распределенной памятью (рис. 3) каждый процессор имеет собственное локальное хранилище данных, которое недоступно напрямую другим процессорам. Каждый процесс обменивается с памятью других процессоров через коммуникационные каналы посредством передачи сообщений (Message Passing).

Системы с гибридной памятью (рис. 4) сочетают в себе свойства двух вышеописанных систем. Этот подход часто используется в кластерных системах, где каждый узел является самостоятельным компьютером с одним или несколькими многоядерными процессорами и собственной памятью.

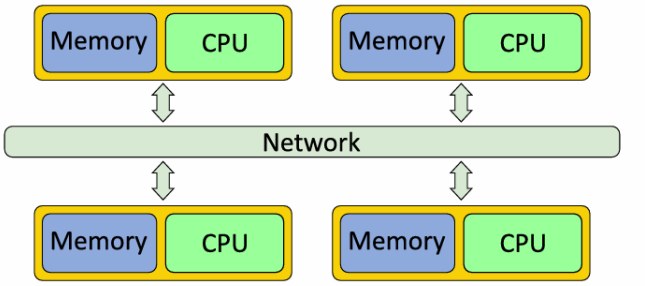


Рис. 3. Системы с распределенной памятью (Distributed Memory Systems)

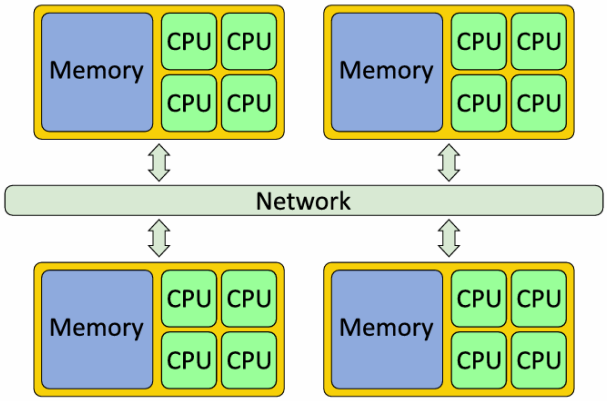


Рис. 4. Системы с гибридной памятью (Hybrid Memory Systems)

Гибридная архитектура памяти часто используется с гибридными моделями программирования, которые позволяют разработчикам комбинировать различные подходы к параллельному программированию, такие как общая память (например, с использованием OpenMP) и распределенная память (например, с использованием MPI).

1.3. КРИТЕРИИ ОЦЕНКИ РАБОТЫ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

***Время выполнения* (*Runtime*)**

Время выполнения задачи в контексте параллельных вычислений относится к времени, необходимому для решения определенной задачи или времени работы программы на параллельной вычислительной системе. Чем меньше время выполнения задачи, тем выше производительность системы.

***Ускорение* (*Speedup*)**

Ускорение параллельной программы – это показатель, отражающий насколько быстрее параллельная версия программы выполняется по сравнению с последовательной версией на одном процессоре или ядре. В общем виде ускорение определяется как отношение времени выполнения последовательной версии программы к времени выполнения параллельной версии на заданном количестве процессоров или ядер:

$$S_p(n) = \frac{T_s(n)}{T_p(n)},$$

где p – количество процессоров; $T_s(n)$ – время выполнения последовательной (sequential) программы на одном процессоре; $T_p(n)$ – время выполнения параллельной (parallel) программы на p процессорах; параметр n характеризует вычислительную сложность решаемой задачи и может трактоваться, например, как размерность входных данных задачи. Основные особенности, которые следует учитывать при интерпретации значения ускорения:

Линейное ускорение (Linear speedup). Если ускорение равно количеству использованных процессоров или ядер $S_p(n) = p$, то говорят о линейном ускорении.

Сверхлинейное ускорение (Superlinear speedup). Иногда ускорение может быть больше, чем количество использованных процессоров или ядер $S_p(n) > p$. Это может произойти, например, при использовании кэширования или других оптимизаций, которые становятся доступными при параллельном выполнении программы.

Ускорение меньше единицы (Sublinear speedup). Ускорение меньше единицы $S_p(n) < 1$ означает, что параллельная версия про-

граммы выполняется медленнее последовательной из-за накладных расходов на коммуникацию между процессорами или из-за других факторов.

***Эффективность* (*Efficiency*)**

Эта характеристика показывает, насколько эффективно используются вычислительные ресурсы параллельной системы для решения задачи. Эффективность обычно измеряется в виде отношения ускорения параллельной программы к количеству использованных процессоров или ядер:

$$E_p(n) = \frac{S_p(n)}{p} = \frac{T_s(n)}{p \cdot T_p(n)}.$$

При интерпретации значения эффективности следует учитывать следующие важные моменты:

Идеальная эффективность. Значение равно единице и достигается, когда параллельная программа демонстрирует линейное ускорение при увеличении числа используемых процессоров или ядер.

Убывающая эффективность. При увеличении количества процессоров или ядер эффективность программы снижается из-за увеличения накладных расходов на коммуникацию между процессорами, а также из-за конкуренции за общие ресурсы (например, доступ к памяти).

Сверхлинейная эффективность. Значение превышает единицу. Это означает, что ускорение программы при распараллеливании превышает ожидаемое значение, основанное на количестве доступных процессоров или ядер. Важно отметить, что сверхлинейная эффективность не является типичным явлением.

2. ОСНОВНЫЕ ПОНЯТИЯ MPI

К базовым понятиям MPI относятся процесс, группа процессов и коммуникатор. Процесс – это исполнение программы одним процессорным элементом, на котором загружен MPI. Процессы объединяются в группы с единой областью связи, внутри которой каждый процесс имеет свой уникальный номер в диапазоне от 0 до N–1, где N – количество процессов в группе. Для взаимодействия процессов в группе используется коммуникатор, который реализует передачу сообщений (обмен данными) между процессами и их синхронизацию. Коммуникатор осуществляет обмены только внутри области связи группы, с которой он ассоциируется.

Обычно MPI-программа для многопроцессорных вычислительных систем пишется на языке C, C++, Fortran или Python с использованием коммуникационных функций библиотеки MPI. Запуск на выполнение MPI-программы представляет собой одновременный запуск совокупности параллельных процессов, количество которых определяет пользователь при запуске. Параллельная часть программы начинается с инициализации MPI. При этом все активированные процессы объединяются в группу с коммуникатором, который имеет имя MPI_COMM_WORLD. Далее средствами MPI в программу передается число активированных процессов, каждому из них присваивается свой номер. Процессы, как правило, отличаются тем, что каждый отвечает за исполнение своей ветви параллельной MPI-программы.

2.1. СИНТАКСИС БАЗОВЫХ ФУНКЦИЙ MPI

Инициализация MPI:

```
int MPI_Init(int* argc, char*** argv)
```

Функция возвращает код ошибки. Нулевой код соответствует успешному выполнению.

Завершение MPI:

```
int MPI_Finalize(void)
```

Функция возвращает код ошибки. Нулевой код соответствует успешному выполнению.

Определение числа процессов в группе *comm*:

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

Функция возвращает количество процессов `size` в области связи с коммуникатором `comm`. Чаще в качестве параметра `comm` используют стандартный коммуникатор `MPI_COMM_WORLD`.

Определение номера процесса в группе comm:

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

Функция возвращает номер процесса, вызвавшего ее, из области связи с коммуникатором `comm`.

Посылка сообщения процессу:

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

Функция осуществляет блокирующую посылку сообщения `buf` с идентификатором `msgtag`, состоящего из `count` элементов типа `datatype`, процессу с номером `dest` в области связи с коммуникатором `comm`. Все параметры этой функции являются входными. Тип передаваемых элементов `datatype` должен указываться с помощью предопределенных констант типа (см. табл. 1). Параметр `msgtag` необходим для того, чтобы отличать сообщения от одного и того же процесса, либо сообщения от разных процессов, если принимающий процесс ожидает сообщение от любого процесса. В качестве параметров `source` и `msgtag` могут быть переданы константы `MPI_ANY_SOURCE` и `MPI_ANY_TAG` соответственно. Значение `MPI_ANY_SOURCE` параметра `dest` означает прием сообщения от любого процесса коммуникатора, а значение `MPI_ANY_TAG` означает прием сообщения с произвольным тэгом.

Прием сообщения от процесса:

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

Функция осуществляет прием сообщения `buf` с идентификатором `msgtag` от процесса `source` с блокировкой. Число элементов в принимаемом сообщении не должно превосходить значения `count`. Тип получаемых данных `datatype` должен указываться с помощью именованных констант MPI.

Библиотека MPI также содержит и неблокирующие функции передачи сообщений между двумя процессами: `MPI_Isend()`, `MPI_Irecv()`. В отличие от блокирующих, которые приостанавливают вызвавший их процесс до тех пор, пока операция передачи не будет завершена, неблокирующие подразумевают совмещение операций обмена с другими операциями.

В библиотеке MPI вводятся собственные типы. Это связано с тем, что языки программирования на разных машинных платформах для одних и тех же типов имеют различное представление в байтах. А поскольку MPI ориентирована на переносимость программных приложений, введение собственных типов дает возможность запуска процессов на различных платформах с автоматическим преобразованием данных при пересылках. В библиотеке MPI для всех стандартных типов данных определены соответствующие константы, позволяющие идентифицировать эти типы в функциях передачи и приема данных.

1. Соответствие между типами C и типами MPI

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

2.2. НЕКОТОРЫЕ ФУНКЦИИ КОЛЛЕКТИВНОГО ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ

Функция `MPI_Barrier` блокирует вызвавший ее процесс до тех пор, пока все остальные члены группы с коммуникатором `comm` не вызовут ее.

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

Возвращает код ошибки. Нулевой код соответствует успешному выполнению.

Функция `MPI_Bcast` рассылает сообщение `buf` процесса с номером `root` остальным процессам группы `comm`, в том числе и себе. Каждый процесс получает копию `count` элементов типа `datatype`. Возвращает код ошибки. Нулевой код соответствует успешному выполнению.

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

`MPI_Reduce` комбинирует элементы `sendbuf` каждого процесса в `recvbuf` процесса `root`. Комбинация подразумевает следующие операции: нахождение максимального (признак операции `MPI_MAX`), определение минимального (признак операции `MPI_MIN`), нахождение суммы (`MPI_SUM`), вычисление произведения

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

Здесь все параметры являются входными, за исключением recvbuf. В параметре recvbuf процесса root собирается результат операции op типа datatype, полученный в результате комбинации count элементов sendbuf каждого процесса группы с коммуникатором comm. Возвращает код ошибки. Нулевой код соответствует успешному выполнению.

2.3. РАБОТА СО ВРЕМЕНЕМ

При оценке эффективности и ускорения работы параллельной программы необходимо определить время, затрачиваемое на ее выполнение. Для этой цели в библиотеке MPI предусмотрена функция MPI_Wtime:

```
double MPI_Wtime(void)
```

Возвращает число секунд, прошедших с некоторого определенного момента в прошлом. Как правило, используется для засечки момента начала и окончания выполнения определенных участков программного кода.

```
double MPI_Wtick(void)
```

Возвращает разрешение таймера MPI в секундах и позволяет определить, насколько точно можно измерить время с помощью функции MPI_Wtime(). Разрешение таймера представляет собой минимальное время между двумя последовательными вызовами MPI_Wtime().

3. ПОДГОТОВКА К РАБОТЕ

3.1. СКАЧИВАНИЕ НЕОБХОДИМЫХ ПРОГРАММНЫХ СРЕДСТВ, НЕОБХОДИМЫХ ДЛЯ РАЗРАБОТКИ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ MPI

3.1.1. ОПЕРАЦИОННАЯ СИСТЕМА MICROSOFT WINDOWS

1. Microsoft MPI v10.1.3 (необходим в независимости от выбранных средств разработки):

https://www.microsoft.com/en-us/download/details.aspx?id=105289

Необходимо скачать и установить оба пакета: "msmpisdk.msi" и "msmpisetup.exe".

Microsoft MPI потребуется установить в любом случае, в независимости от того, что будет использоваться для разработки программ под Windows: C/C++/Fortran/Python.

2. **Если установлена полноценная среда Microsoft Visual Studio с установленной поддержкой проектов Visual C++, то выполнять этот пункт не нужно!** Microsoft C++ Build Tools (необходимый минимум, если разработка ведется на языках C/C++):

https://visualstudio.microsoft.com/ru/visual-cpp-build-tools

Используется как минимальный набор средств разработки на C/C++ под Microsoft Windows. Цитата: *Microsoft C++ Build Tools предоставляет наборы инструментов MSVC через автономный установщик с поддержкой сценариев без использования Visual Studio. Рекомендуется при создании библиотек и приложений C++, предназначенных для Windows, из командной строки.* Более совершенным инструментом является полноценная среда разработки Microsoft Visual Studio.

3. Для компиляции C-файла программы необходимо:

- Через «Пуск» запустить "x64 Native Tools Command Prompt for VS 2022". Откроется окно командной строки.
- Используя команду "cd", перейти в папку со скачанными примерами. Если примеры хранятся не на диске "c:", сначала сменить диск.
- Запустить командный файл "Compile4windows.bat"

Пример: Допустим, примеры скачаны в папку "d:\myproject\example1". Порядок ввода команд (выделены жирным шрифтом) в окне "x64 Native Tools Command Prompt for VS 2022":

C:\Program Files (x86)\Microsoft Visual Studio\2022\BuildTools> **d:**

D:\>**cd d:\myproject\example1**

D:\myproject\example1>**Compile4windows.bat**

В результате компиляции будет создан исполняемый файл "hello.exe".

4. Python (необходим, для разработки на языке Python):

https://www.python.org/downloads

Альтернативным способом является использование дистрибутива Anaconda (по желанию, назначение и установку изучить самостоятельно).

5. Библиотека MPI for Python (необходима для разработки на языке Python):

https://mpi4py.readthedocs.io/en/stable

Потребуется, если разработка параллельных MPI-программ будет осуществляться на языке Python. Устанавливается с помощью системы управления пакетами "pip": необходимо выполнить в командной строке команду: "pip install mpi4py".

3.1.2. СЕМЕЙСТВО ОПЕРАЦИОННЫХ СИСТЕМ LINUX

Установка всех необходимых средств изучается студентом самостоятельно с учетом используемой им версии Linux (Ubuntu, Fedora, Alt и пр.), а также менеджера пакетов (APT, YUM, Portage и др.). В любом случае потребуется установить реализацию библиотеки MPI (широкое распространение получили OpenMPI или MPICH), набор компиляторов GCC (GNU Compiler Collection) и актуальную версию Python 3. Также потребуется установить библиотеку "mpi4py" с помощью выполнения в терминале команды "pip install mpi4py".

3.2. ЗАПУСК ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Запуск параллельных MPI-программ осуществляется из командной строки. Команда "**mpiexec-n X program.exe**" является обычным способом для запуска параллельных заданий, где **X** является общим числом процессов для использования, а **program.exe** – скомпилированная MPI-программа. Допустим, что скомпилированный (с помощью, например, компилятора "mpicc") из исходного файла "hello.c" исполняемый файл называется "hello.exe". Тогда команда запуска на, допустим, четырех процессах (n = 4) будет выглядеть так:

mpiexec-n 4 hello.exe

Параллельная MPI-программа на языке Python, использующая библиотеку mpi4py и сохраненная в файле "hello.py", запускается аналогично, но с дополнительным использованием команды **python**:

mpiexec-n 4 **python** hello.py

Примечание: в некоторых системах вместо команды **mpiexec** используется команда **mpirun**.

4. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

4.1. ЛАБОРАТОРНАЯ РАБОТА № 1 «ПАРАЛЛЕЛЬНАЯ ПРОГРАММА HELLO WORLD!»

1. Скачайте и установите требуемые для работы программные пакеты (Microsoft MPI, Microsoft C++ Build Tools/Python + mpi4py). Необходимая для этого информация содержится в файле "downloadMPI.docx".

2. Скачайте папки с примерами "example1" и "example 3". Прочитайте пояснения к примерам в файлах "readme.txt". Изучите исходный код, будьте готовы отвечать на вопросы по коду программы.

3. На языке C или Python (по выбору студента) скомпилируйте (если программа на языке C) и запустите представленные примеры параллельных MPI-программ. Запуск осуществляйте с использованием "mpiexec" с соответствующими параметрами, как написано в файле "downloadMPI.docx".

4. Запустите примеры на разном количестве процессов, начиная с 1 ("mpiexec –n 1") до 10.

Отчет. Для отчета необходимо сохранить скриншоты окон с выводами результатов работы программ.

В процессе выполнения работы можно пользоваться следующими ресурсами.

1. Джанкарло Закконе. Книга рецептов параллельного программирования Python. 2-е изд.: http://onreader.mdl.ru/PythonParallelProgrammingCookbook.2nd/content/index.html

2. Программированию с использованием MPI посвящен четвертый раздел книги: http://onreader.mdl.ru/PythonParallelProgrammingCookbook.2nd/content/Ch04.html

3. Microsoft MPI: https://learn.microsoft.com/ru-ru/message-passing-interface/microsoft-mpi

4. Самостоятельный поиск ресурсов в Интернете, посвященных тематике данной лабораторной работы.

Пример 1: "Hello world"

В примере представлена программа, в которой каждый MPI-процесс определяет свой порядковый номер (rank – ранг, порядок), число процессов в группе (numproc) и выводит на экран эти значения.

Пример запуска параллельной программы на Python на четырех процессах:

```
mpiexec -n 4 python hello.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
numproc = comm.Get_size()
print ("Hello World! from process", rank, "of",
numproc)
```

Исходный текст параллельной программы на языке Си под Linux:

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

Исходный текст параллельной программы на языке Fortran под Linux:

```
mpif90 ./hello.f
      program Hello
      implicit none
      include 'mpif.h'
      integer rank, numproc, ret
```

```
      integer status(MPI_STATUS_SIZE)
      call MPI_Init(ret)
      call  MPI_COMM_SIZE(MPI_COMM_WORLD,  numproc,
ret)
      call   MPI_COMM_RANK(MPI_COMM_WORLD,   rank,
ret)
      write(6,*)  'Hello  World!  from  process',
rank, ' of ', numproc
      call MPI_FINALIZE(ret)
      end
```

Примечание. Если необходимо изменить имя исполняемого файла, то используется опция компилятора "-o". Например,

```
mpicc ./hello.c -o hello
mpiexec -n 4 hello
```

Пример 2:

Пример реализует простейшие коммуникационные операции типа «точка-точка» – блокирующие функции передачи и приема сообщений MPI_Send и MPI_Recv.

Процессы выполняют различные задания: все, кроме нулевого процесса *(rank != 0)*, производят посылку целого значения своего номера "rank" нулевому процессу.

Нулевой процесс *(rank == 0)* в цикле получает значения, отправленные другими процессами, выводит эти значения на экран и затем сообщает об окончании работы.

Процессы заканчивают свою работу не одновременно.

Первым свою работу закончит процесс с номером "1": он дождется приема "процессом "0" своего сообщения, и на этом его работа закончится.

Потом закончит работу процесс "2", затем процесс "3" и т.д.

Самым последним завершит работу процесс "0".

Исходный текст программы на языке Си:

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char** argv){
       int rank, numproc, rc, tag;
       MPI_Status status;
```

```
        int message;
        rc = MPI_Init(&argc, &argv);
        rc      =      MPI_Comm_size(MPI_COMM_WORLD,
&numproc);
        rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        tag = 50;
        if(rank == 0){
          for (int i = 1; i < numproc; i++){
              rc     =     MPI_Recv(&message,     1,
MPI_INTEGER, i, tag, MPI_COMM_WORLD, &status);
              printf("Process %d of %d is ready to
work.\n", message, numproc);
          }
          printf("Process %d have finished receiving
now.\n", rank);
        }
        else{
          rc = MPI_Send(&rank, 1, MPI_INTEGER, 0,
tag, MPI_COMM_WORLD);
        }
        rc = MPI_Finalize();
        return rc;
}
```

Исходный текст программы на языке Python:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
numproc = comm.Get_size()
if rank == 0:
        for i in range(1, numproc):
          message = comm.recv(source = i)
          print("Process", message, "of", numproc,
"is ready to work.");
        print("Process",   rank,   "have   finished
receiving now.");
else:
        comm.send(rank, dest = 0)
```

4.2. ЛАБОРАТОРНАЯ РАБОТА № 2 «ПАРАЛЛЕЛЬНОЕ ВЫЧИСЛЕНИЕ ЗНАЧЕНИЯ ОПРЕДЕЛЕННОГО ИНТЕГРАЛА МЕТОДОМ ПРЯМОУГОЛЬНИКОВ»

1. Скачайте и установите требуемые для работы программные пакеты (Microsoft MPI, Microsoft C++ Build Tools/Python + mpi4py). Этот пункт будет выполнен автоматически, если была успешно выполнена Лабораторная работа № 1

2. Изучите теоретический материал "Метод прямоугольников". Особенное внимание следует обратить на вычисление методом средних прямоугольников, так как представленные примеры используют именно его.

3. Скачайте папку с примерами "example5". Прочитайте пояснения к примеру в файле "readme.txt". Изучите исходный код, будьте готовы отвечать на вопросы по коду программы.

4. На языке C и Python скомпилируйте (если программа на языке C) и запустите представленный пример параллельной MPI-программы. Запуск осуществляйте с использованием "mpiexec" с соответствующими параметрами, как это осуществлялось при выполнении Лабораторной работы № 1.

5. Прочитать файл "Номера вариантов Распределенная обработка.docx", в котором представлены номера вариантов.

6. Осуществить вычисление определенного интеграла согласно своему варианту с использованием параллельной программы на языке программирования C и Python. В файле с заданиями "IntegralsVariants.pdf" представлены соответствующие подынтегральные функции f(x), интервалы интегрирования [a; b], а также результаты вычислений значений интегралов. Эти результаты можно использовать как «правильные ответы» для проверки полученных результатов. В качестве основы для программы использовать представленные примеры "defintegral0.c" или "defintegral0.py". В исходном тексте для выбранного языка программирования нужно поменять функцию (код функции double f(double x) или def f(x) соответственно) и пределы интегрирования (константы a и b) на свои.

7. В примерах задана точность eps = 1e-6. Запустите примеры, повысив точность (для чего нужно уменьшить значение константы eps). Значение константы eps уменьшить до таких значений, чтобы время выполнения программы стало достаточно значительным (не миллисекунды, а хотя бы несколько секунд, иначе программа

будет выполняться так быстро, что ускорения от параллелизации не будет заметно). Запустить несколько раз параллельные программы на C и Python на разном количестве процессов, начиная с 1 ("mpiexec -n 1") до 10. Заполнить в таблице 1 строки «Время работы алгоритма».

8. Рассчитать значения ускорений и эффективности и заполнить остальные ячейки таблицы.

9. Построить графики зависимости ускорения и эффективности от количества используемых процессоров.

10. Сделать выводы по полученным результатам.

Отчет. Для отчета необходимо сохранить скриншоты окон с выводами результатов работы параллельной программы.

Примечание. MPI позволяет запускать параллельные программы на количестве процессов (значение параметра запуска -n), превышающих количество физических ядер компьютера, например, *"mpiexec –n 50 python defintegral0.py"*. Однако реально это не ускорит, а наоборот, замедлит вычисления, так как запушенные процессы, утрируя, «не настоящие», и фактически ресурсы компьютера тратятся на переключения между этими процессами. Поэтому студент должен сам определить, на каком количестве процессов на конкретном используемом им оборудовании удается получить максимальное ускорение (минимальное время выполнения программы).

2. Расчет ускорения и эффективности параллельной MPI-программы

	Язык программирования	Количество процессоров					
		1	2	3	4	...	10
Время работы алгоритма, с	C						
	Python						
Ускорение (время выполнения для 1 процессора, деленное на время выполнения для *N*-процессоров)	C	1					
	Python	1					
Эффективность (значение ускорения для *N*-процессоров, деленное на число процессоров *N*)	C						
	Python						

Следует отметить, что интерфейс программирования MPI в реальной жизни чаще всего используется не на отдельных компьютерах, а на высокопроизводительных вычислительных кластерах, состоящих из тысяч вычислительных узлов, объединенных специализированной высокоскоростной вычислительной сетью (в частности, суперкомпьютер *Frontier*, возглавляющий рейтинг top500 от ноября 2023 г., состоит из 9400 вычислительных узлов, оснащенных 64-ядерными процессорами AMD EPYC 64C). На подобном оборудовании можно использовать значения параметра запуска `"-n"`, равные нескольким десяткам, а то и сотням тысяч, что позволяет значительно ускорить параллельные вычисления.

В процессе выполнения работы можно пользоваться следующими ресурсами.

1. Метод прямоугольников: http://mathprofi.ru/metod_prjamougolnikov.html
2. Джанкарло Закконе. Книга рецептов параллельного программирования Python. 2-е изд.: http://onreader.mdl.ru/PythonParallelProgrammingCookbook.2nd/content/index.html
3. Программированию с использованием MPI посвящен четвертый раздел книги: http://onreader.mdl.ru/PythonParallelProgrammingCookbook.2nd/content/Ch04.html
4. Microsoft MPI: https://learn.microsoft.com/ru-ru/message-passing-interface/microsoft-mpi
5. Frontier supercomputer: https://www.olcf.ornl.gov/frontier
6. Самостоятельный поиск ресурсов в Интернете, посвященных тематике данной лабораторной работы.

ВЫЧИСЛЕНИЕ ОПРЕДЕЛЕННЫХ ИНТЕГРАЛОВ. ВАРИАНТЫ ЗАДАНИЙ

Вариант 1

$$f(x) = \frac{e^{-x^2+0.38}}{\sin\left(\frac{1}{x^2+1.5}\right)+2.0}, a = 0.4, b = 1.0. \quad \int_a^b f(x)\mathrm{d}x = 0.2155$$

Вариант 2

$$f(x) = \frac{x^2 + \sin(3.25 \cdot x)}{e^{x^2} + 0.35}, a = 0.4, b = 0.6. \quad \int_a^b f(x)\mathrm{d}x = 0.1505$$

Вариант 3

$$f(x) = \frac{1.0 + e^{\frac{0.75}{x}}}{2.0 + x}, a = 1.0, b = 3.0. \quad \int_a^b f(x)\mathrm{d}x = 1.305$$

Вариант 4

$$f(x) = \frac{e^{-\tan(0.5 \cdot x)}}{0.25 + \cos(x)}, a = 0.0, b = 1.5. \quad \int_a^b f(x)\mathrm{d}x = 1.1608$$

Вариант 5

$$f(x) = \frac{atan(0.75 \cdot x)}{x + 1.48}, a = 0.2, b = 0.5. \quad \int_a^b f(x)\mathrm{d}x = 0.0415$$

Вариант 6

$$f(x) = \frac{x + \ln(x + 1.25)}{x + 0.1}, a = 0.0, b = 1.0. \quad \int_a^b f(x)\mathrm{d}x = 1.8024$$

Вариант 7

$$f(x) = \frac{e^x + \log(x + 1.0)}{\sin(x) + 3.75}, a = 0.3, b = 0.8. \quad \int_a^b f(x)\mathrm{d}x = 0.2265$$

Вариант 8

$$f(x) = \frac{\sqrt{x}}{e^{0.97 \cdot x} + 3.0}, a = 0.5, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 0.2498$$

Вариант 9

$$f(x) = \frac{\sqrt{x \cdot (3.75 - x)}}{x + 1.0}, a = 1.0, b = 1.2. \quad \int_a^b f(x)\mathrm{d}x = 0.1625$$

Вариант 10

$$f(x) = \frac{e^x + 1.0}{\sin(x^2 + 1.0) + 2.0}, a = 0.4, b = 1.0. \quad \int_a^b f(x)\mathrm{d}x = 0.6151$$

Вариант 11

$$f(x) = \frac{\sin(x + 2.0)}{\cos(x) + 0.4}, a = -1.0, b = 1.0. \quad \int_a^b f(x)\mathrm{d}x = 1.2246$$

Вариант 12

$$f(x) = \frac{\sqrt{x^3} + 1.0}{\cos^2(x) + 1.0}, a = 0.0, b = 1.0. \quad \int_a^b f(x)\mathrm{d}x = 0.8495$$

Вариант 13

$$f(x) = \frac{x^2 + \sin(x + 2.0)}{e^{x^3} - 0.75}, a = 0.4, b = 1.0. \quad \int_a^b f(x)\mathrm{d}x = 0.8836$$

Вариант 14

$$f(x) = \frac{x^3 \cdot asin(0.25 \cdot x)}{x - 0.95}, a = 1.0, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 2.7381$$

Вариант 15

$$f(x) = \frac{x^2 \cdot atan(0.5 \cdot x)}{x + 1.5}, a = 1.0, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 0.5023$$

Вариант 16

$$f(x) = \frac{\sqrt{x} + 5.25}{\cos^3(0.75 \cdot x)}, a = 0.1, b = 1.0. \quad \int_a^b f(x)\mathrm{d}x = 7.8412$$

Вариант 17

$$f(x) = \frac{x^4}{0.5 \cdot x^2 + x + 6.0}, a = 0.4, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 0.708$$

Вариант 18

$$f(x) = \frac{1.25}{x \cdot \sqrt{(0.2 \cdot x + 1.0)^3}}, a = 1.0, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 0.5946$$

Вариант 19

$$f(x) = \frac{0.95 \cdot x}{\sin^3(2.2 \cdot x)}, a = 0.1, b = 0.5. \quad \int_a^b f(x)\mathrm{d}x = 0.8146$$

Вариант 20

$$f(x) = \frac{3.33 \cdot x}{2.0 \cdot x^2 + \sqrt{x^3} + 3.62}, a = 1.0, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 0.4954$$

Вариант 21

$$f(x) = \frac{\ln(x + 1.0)}{e^x + 0.37 \cdot \sin(x)}, a = 0.0, b = 1.0. \quad \int_a^b f(x)\mathrm{d}x = 0.1891$$

Вариант 22

$$f(x) = \frac{0.75 \cdot x^2}{x \cdot \log(x) + 0.5}, a = 1.0, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 2.1762$$

Вариант 23

$$f(x) = \frac{(x + 1.9) \cdot \sin(0.4 \cdot x)}{x^2}, a = 1.0, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 0.8726$$

Вариант 24

$$f(x) = \frac{\ln(x + 5.75)}{x^2 + 0.71}, a = 2.0, b = 3.0. \quad \int_a^b f(x)\mathrm{d}x = 0.3117$$

Вариант 25

$$f(x) = \frac{3.0 \cdot \cos(x^3)}{2.0 \cdot x + 1.25}, a = 0.0, b = 1.0. \quad \int_a^b f(x)\mathrm{d}x = 1.3644$$

Вариант 26

$$f(x) = \frac{2.6 \cdot x^2 \cdot \ln(x)}{1.5 \cdot x + 1.75}, a = 1.0, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 0.6482$$

Вариант 27

$$f(x) = \frac{4.25 \cdot e^{0.25 \cdot x^2}}{5.25 \cdot x^3 + 1.0}, a = 1.0, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 0.4518$$

Вариант 28

$$f(x) = \frac{3.0 \cdot x + 5.0 \cdot \tan(x^3)}{x + 2.0}, a = 0.5, b = 1.0. \quad \int_a^b f(x)\mathrm{d}x = 0.8991$$

Вариант 29

$$f(x) = \frac{3.0 \cdot x^2 + \sin(x)}{(x + 0.75)^2}, a = 0.0, b = 1.5. \quad \int_a^b f(x)\mathrm{d}x = 1.433$$

Вариант 30

$$f(x) = \frac{\sqrt{1.0 - 0.25 \cdot \sin^2(x)}}{0.5 \cdot x + 3.0}, a = 0.0, b = 2.0. \quad \int_a^b f(x)\mathrm{d}x = 0.5324$$

Пример параллельного вычисления определенного интеграла с требуемой точностью методом средних прямоугольников:

Приведенные ниже примеры построены по следующему принципу: область интегрирования разбивалась на равные подобласти (отрезки) по числу используемых процессов, а исходный интеграл представлялся в виде суммы интегралов по таким частичным отрезкам.

Схема вычислений определенного интеграла состоит из трех этапов: во-первых, каждый процесс определяет свои границы интегрирования.

Во-вторых, зная число интервалов разбиения n, происходит вычисление шага h, после чего каждый процесс вычисляет интеграл на своем частичном отрезке.

На последнем, третьем этапе, происходит суммирование всех вычисленных значений нулевым процессом с получением значения интеграла.

После чего нулевой процесс сравнивает полученный результат со значением, полученным на предыдущей итерации.

Если разность delta между результатами, полученными на текущей и предыдущей итерации меньше требуемой точности eps, то вычисления останавливаются.

В противном случае число разбиений n увеличивается вдвое, и повторяются представленные выше три этапа вычисления значения определенного интеграла.

Кроме того, на нулевой процесс возлагается обязанность оценки времени, затраченного на выполнение программы, и вывода результатов расчета на экран.

Этот пример использует блокирующие функции пересылки сообщений MPI_Send и MPI_Recv.

Исходный текст программы на языке Си:

```
// Пример параллельного вычисления определенного
интеграла методом средних прямоугольников
#include <stdio.h>
#include <math.h>
#include <mpi.h>
// Подынтегральная функция
double f(double x);
int main (int argc, char** argv){
    int rank, numproc, rc, tag, n, nloc, count;
    MPI_Status status;
    double partsum, globalsum, start, finish, al,
bl, x, h, msg;
```

```
    // Пределы интегрирования
    const double a = 0.0, b = 1.0;
    rc = MPI_Init(&argc, &argv);
    rc = MPI_Comm_size(MPI_COMM_WORLD, &numproc);
    rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    tag = 50;
    //0-ой процесс засекает время
    if(rank == 0)
        start = MPI_Wtime();
    // Каждый процесс определяет свои пределы ин-
тегрирования
    al = a + (b - a) * rank / numproc;
    bl = al + (b - a) / numproc;
    // Начальное количество прямоугольников
    n = 10;
    double eps, I = 1.e10, I2 = 0.0, delta;
    // Требуемая точность
    eps = 1.0e-6;
    globalsum = 0.0;
    count  = 0;
    delta = fabs(I - I2);
    while(delta > eps){
        // шаг по значению x
        h = (bl - al) / n;
        // Каждый процесс определяет свою частич-
ную сумму ...
        partsum = 0.0;
        for(int i = 0; i < n; i++){
            x = al + h * (i + 0.5);
            partsum = partsum + f(x);
        }
        partsum = partsum * h;
```

```
        // 0-й процесс получает частичные суммы и
определяет результат
        // суммируя частичные суммы, полученные от
каждого процесса
        if(rank == 0){
            globalsum = partsum;
            for(int i = 1; i < numproc; i++){
                rc = MPI_Recv(&msg, 1,
MPI_DOUBLE, i, tag, MPI_COMM_WORLD, &status);
                globalsum = globalsum + msg;
            }
            I2 = I;
            I = globalsum;
            delta = fabs(I - I2);
            n = n * 2;
            count++;
        }
        else
            rc = MPI_Send(&partsum, 1,
MPI_DOUBLE, 0, tag, MPI_COMM_WORLD);
        // передать с нулевого процесса количество
интервалов n всем остальным процессам
        if(rank == 0)
            for(int i = 1; i < numproc; i++)
                rc = MPI_Send(&n, 1, MPI_INT, i,
tag, MPI_COMM_WORLD);
        else
            rc = MPI_Recv(&n, 1, MPI_INT, 0, tag,
MPI_COMM_WORLD, &status);
        // передать с нулевого процесса delta всем
остальным процессам
        if(rank == 0)
            for(int i = 1; i < numproc; i++)
```

```
                rc = MPI_Send(&delta, 1,
MPI_DOUBLE, i, tag, MPI_COMM_WORLD);
        else
            rc = MPI_Recv(&delta, 1, MPI_DOUBLE,
0, tag, MPI_COMM_WORLD, &status);

    }
    if(rank == 0){
        finish = MPI_Wtime();
        printf( "Result: %.16f Delta: %.16f Final
n: %d Count: %d Runtime: %f Num Proc: %d\n", glob-
alsum, fabs(globalsum - I2), n, count, finish -
start, numproc);
    }
    rc = MPI_Finalize();
    return rc;
}

double f(double x){
    return log(1/x);
}
```

Исходный текст программы на языке Python:

```
# Пример параллельного вычисления определенного
интеграла методом средних прямоугольников
import math
from mpi4py import MPI
# Подынтегральная функция
def f(x):
    return math.log(1/x)
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
numproc = comm.Get_size()
```

```
# 0-й процесс засекает начало времени выполнения
if rank==0:
    start = MPI.Wtime()
# Пределы интегрирования
a = 0.0
b = 1.0
# Каждый процесс определяет свои пределы интегри-
рования
# и число интервалов разбиения
al = a + (b - a) * rank / numproc
bl = al + (b - a) / numproc
# Начальное количество прямоугольников для каждого
процессора
n = 10
# Требуемая точность
eps = 1.0e-6
I = 1.e10
I2 = 0.0
globalsum = 0.0
count  = 0
delta = abs(I - I2)
while delta > eps :
    # шаг по значению x
    h = (bl - al) / n
    # Каждый процесс определяет свою частичную
сумму
    partsum = 0.0
    for i in range(0, n):
        x = al + h * (i + 0.5)
        partsum = partsum + f(x)
    partsum = partsum * h
    # 0-й процесс получает частичные суммы
и вычисляет конечный результат
```

```
    # суммируя частичные суммы, полученные
от каждого процесса
    if rank == 0:
        globalsum = partsum
        for i in range(1, numproc):
            msg = comm.recv(source = i)
            globalsum = globalsum + msg
        I2 = I
        I = globalsum
        delta = abs(I - I2)
        n = n * 2
        count = count + 1
    else:
        comm.send(partsum, dest = 0)
    # передать с нулевого процесса количество
интервалов n всем остальным процессам
    if rank == 0:
        for i in range(1, numproc):
            comm.send(n, dest = i)
    else:
        n = comm.recv(source = 0)
    # передать с нулевого процесса delta всем
остальным процессам
    if rank == 0:
        for i in range(1, numproc):
            comm.send(delta, dest = i)
    else:
        delta = comm.recv(source = 0)
if rank == 0:
    finish = MPI.Wtime()
    print(f"Result: {globalsum} Delta:{delta}
Final n: {n} Count: {count} Runtime: {finish -
start} Num Proc: {numproc}")
```

4.3. ЛАБОРАТОРНАЯ РАБОТА № 3 «ПАРАЛЛЕЛЬНОЕ ВЫЧИСЛЕНИЕ ДВОЙНОГО ИНТЕГРАЛА МЕТОДОМ МОНТЕ-КАРЛО»

1. Скачайте и установите требуемые для работы программные пакеты (Microsoft MPI, Microsoft C++ Build Tools/Python + mpi4py). Этот пункт будет выполнен автоматически, если была успешно выполнена Лабораторная работа № 1.

2. Изучите теоретический материал "Вычисление двойных интегралов".

3. Изучите теоретический материал из файла "MonteCarlo.pdf").

4. Скачайте папку с примерами "example7".

5. На языке C и Python скомпилируйте (если программа на языке C) и запустите представленный пример параллельной MPI-программы, вычисляющий значение двойного интеграла методом Монте-Карло. Запуск осуществляйте с использованием "mpiexec" с соответствующими параметрами, как это осуществлялось при выполнении Лабораторной работы № 1.

6. Прочитайте файл "Номера вариантов Распределенная обработка.docx", в котором представлены номера вариантов.

7. Осуществите вычисление определенного интеграла согласно своему варианту с использованием параллельной программы на языке программирования C и Python. В файле с заданиями *"DoubleIntegralsVariantsPar.pdf"* представлены соответствующие подынтегральные функции F(x,y), интервалы интегрирования [a; b], [c(x); d(x)], значения площади s, а также результаты вычислений значений интегралов. Эти результаты можно использовать как «правильные ответы» для проверки полученных результатов. В качестве основы для программы использовать представленные примеры *"doubleintegral.c"* или *"doubleintegral.py"*. В исходном тексте для выбранного языка программирования согласно

своего варианта нужно поменять функцию (код функции double F(double x, double y) или def F(x,y) соответственно) и пределы интегрирования (значения констант a и b, а также проверку на попадание в область, заданную кривыми c(x) и d(x)).

8. В примерах задано количество испытаний totalTosses = 1e8. Запустите примеры, повысив количество испытаний до значения totalTosses = 1e9, на разном количестве процессов, начиная с одного ("mpiexec –n 1") до 10. Сравните полученные результаты, в частности время выполнения программы. Заполните в таблице 1 строки «Время работы алгоритма».

9. Рассчитайте значения ускорений и эффективности и заполните остальные ячейки таблицы.

10. Постройте графики зависимости ускорения и эффективности от количества используемых процессоров.

11. Сделайте выводы по полученным результатам.

12. Обратите внимание, что в примерах используется функция редукции MPI_Reduce(). Изучите назначение функции MPI_Reduce(), передаваемые в функцию параметры. Также изучите листинги программ, умейте отвечать на вопросы по работе программы.

Отчет. Для отчета необходимо сохранить скриншоты окон с выводами результатов работы параллельной программы.

Примечание. MPI позволяет запускать параллельные программы на количестве процессов (значение параметра запуска -n), превышающих количество физических ядер компьютера, например "mpiexec –n 50 python doubleintegral.py". Однако реально это не ускорит, а наоборот, замедлит вычисления, так как запушенные процессы, утрируя, «не настоящие», и фактически ресурсы компьютера тратятся на переключения между этими процессами. Поэтому студент должен сам определить, на каком количестве процессов на конкретном используемом им оборудовании удается получить максимальное ускорение (минимальное время выполнения программы).

3. Расчет ускорения и эффективности параллельной MPI-программы

	Язык программирования	Количество процессоров					
		1	2	3	4	...	10
Время работы алгоритма, с	C						
	Python						
Ускорение (время выполнения для 1 процессора, деленное на время выполнения для *N*-процессоров)	C	1					
	Python	1					
Эффективность (значение ускорения для *N*-процессоров, деленное на число процессоров *N*)	C						
	Python						

В процессе выполнения работы можно пользоваться следующими ресурсами.

1. Двойные интегралы: http://mathprofi.ru/dvoinye_integraly_dlya_chainikov.html
2. Джанкарло Закконе. Книга рецептов параллельного программирования Python. 2-е изд.: http://onreader.mdl.ru/PythonParallelProgrammingCookbook.2nd/content/index.html
3. Программированию с использованием MPI посвящен четвертый раздел книги: http://onreader.mdl.ru/PythonParallelProgrammingCookbook.2nd/content/Ch04.html
4. Microsoft MPI: https://learn.microsoft.com/ru-ru/message-passing-interface/microsoft-mpi
5. Frontier supercomputer: https://www.olcf.ornl.gov/frontier
6. Самостоятельный поиск ресурсов в Интернете, посвященных тематике данной лабораторной работы.

ВЫЧИСЛЕНИЕ ДВОЙНЫХ ИНТЕГРАЛОВ. ВАРИАНТЫ ЗАДАНИЙ

Вариант 1

$$F(x,y)=\frac{5\cdot x}{(y+1)^2}, a=0.0, b=1.0, c(x)=0, d(x)=x,$$

$$\int_a^b\int_{c(x)}^{d(x)} F(x)\mathrm{d}y\mathrm{d}x=0.9657, s=\int_a^b d(x)\mathrm{d}x=0.5$$

Вариант 2

$$F(x,y)=e^{x-y}, a=-1.0, b=0.0, c(x)=0.0, d(x)=2.0-x,$$

$$\int_a^b\int_{c(x)}^{d(x)} F(x)\ \mathrm{d}y\mathrm{d}x=0.5736, s=\int_a^b d(x)\mathrm{d}x=2.5$$

Вариант 3

$$F(x,y)=e^{2\cdot x-2\cdot y}, a=0.0, b=1.0, c(x)=0.0, d(x)=1.0-x,$$

$$\int_a^b\int_{c(x)}^{d(x)} F(x)\ \mathrm{d}y\mathrm{d}x=0.6905, s=\int_a^b d(x)\mathrm{d}x=0.5$$

Вариант 4

$$F(x,y)=e^{2.0\cdot x-y}, a=0.0, b=1.0, c(x)=0.0, d(x)=x,$$

$$\int_a^b\int_{c(x)}^{d(x)} F(x)\ \mathrm{d}y\mathrm{d}x=1.4762, s=\int_a^b d(x)\mathrm{d}x=0.5$$

Вариант 5

$$F(x,y)=x\cdot y+y^2, a=0.0, b=2.0, c(x)=0.0, d(x)=x+1.0,$$

$$\int_a^b\int_{c(x)}^{d(x)} F(x)\ \mathrm{d}y\mathrm{d}x=12.3333, s=\int_a^b d(x)\mathrm{d}x=4.0$$

Вариант 6

$$F(x,y)=\sqrt{x+y}, a=0.0, b=1.0, c(x)=0, d(x)=x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 0.4876, s = \int_a^b d(x)\mathrm{d}x = 0.5$$

Вариант 7

$$F(x,y) = x \cdot y + \frac{y^2}{2}, a = -1.0, b = 1.0, c(x) = 0.0, d(x) = 3 - x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 8.0, s = \int_a^b d(x)\mathrm{d}x = 6.0$$

Вариант 8

$$F(x,y) = x^2 + 4.0 \cdot y, a = 0.0, b = 2.0, c(x) = 0.0, d(x) = x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 9.3333, s = \int_a^b d(x)\mathrm{d}x = 2.0$$

Вариант 9

$$F(x,y) = \frac{x^2}{y^2 + 1.0}, a = 0.0, b = 1.0, c(x) = 0, d(x) = x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 0.2107, s = \int_a^b d(x)\mathrm{d}x = 0.5$$

Вариант 10

$$F(x,y) = \frac{x^3}{(y+1)^2}, a = 1.0, b = 2.0, c(x) = 0, d(x) = x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 2.3221, s = \int_a^b d(x)\mathrm{d}x = 1.5$$

Вариант 11

$$F(x,y) = y^2 \cdot \sin^2(x), a = -\pi/2, b = \pi/2, c(x) = 0, d(x) = 3 \cdot \cos(x),$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 2.4, s = \int_a^b d(x)\mathrm{d}x = 6$$

Вариант 12

$$F(x,y) = \sqrt{-x^2 + y^2 + 1.0}, a = 0.0, b = 1.0, c(x) = 0, d(x) = x + 1,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 1.7303, s = \int_a^b d(x)\mathrm{d}x = 1.5$$

Вариант 13

$$F(x,y) = \frac{1.0}{\sqrt{(x^2 + y^2 + 1.0)^3}}, a = 0.0, b = 1.0, c(x) = 0, d(x) = x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 0.2618, s = \int_a^b d(x)\mathrm{d}x = 0.5$$

Вариант 14

$$F(x,y) = x^2 - 2.5 \cdot x + y^2, a = 0.0, b = 1.0, c(x) = 0, d(x) = 3 \cdot x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 0.5, s = \int_a^b d(x)\mathrm{d}x = 1.5$$

Вариант 15

$$F(x,y) = x \cdot \sin(y) + y \cdot \cos(x), a = 0.0, b = 1, c(x) = 0, d(x) = 3 \cdot x^2,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 1.0753, s = \int_a^b d(x)\mathrm{d}x = 1.0$$

Вариант 16

$$F(x,y) = \sqrt{x^3} + \sqrt{y^3} - 1.0, a = 0.0, b = 1.0, c(x) = 0, d(x) = x + 1.0,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 0.3644, s = \int_a^b d(x)\mathrm{d}x = 1.5$$

Вариант 17

$$F(x,y) = 2.0 \cdot y^2 + \ln(x^2 + 1.0), a = 0.0, b = 2.0, c(x) = 0, d(x) = 0.5 \cdot x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 1.3451, s = \int_a^b d(x)\mathrm{d}x = 1.0$$

Вариант 18

$$F(x,y) = \frac{y^2 + e^x}{x}, a = 1.0, b = 2.0, c(x) = 0, d(x) = x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 5.4486, s = \int_a^b d(x)\mathrm{d}x = 1.5$$

Вариант 19

$$F(x,y) = (x^3 - y^2)^2, a = 0.0, b = 1.0, c(x) = 0, d(x) = x + 1,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 1.3726, s = \int_a^b d(x)\mathrm{d}x = 1.5$$

Вариант 20

$$F(x,y) = |x \cdot y \cdot (x + y)|, a = 0.0, b = 1.0, c(x) = 0, d(x) = x + 1.0,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 1.3333, s = \int_a^b d(x)\mathrm{d}x = 1.5$$

Вариант 21

$$F(x,y) = y^2 \cdot (x^2 + 1.0), a = 0.0, b = 2.0, c(x) = 0, d(x) = x + 0.5,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 11.1167, s = \int_a^b d(x)\mathrm{d}x = 3.0$$

Вариант 22

$$F(x,y) = \frac{(x + y)^2}{x + 1.5}, a = 0.0, b = 2.0, c(x) = 0, d(x) = x + 1.0,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 8.416, s = \int_a^b d(x)\mathrm{d}x = 4.0$$

Вариант 23

$$F(x,y)=\frac{y^2\cdot\sin(x)+\cos(x)}{x+y}, a=0, b=1, c(x)=0, d(x)=2\cdot x,$$

$$\int_a^b\int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x=1.1697, s=\int_a^b d(x)\mathrm{d}x=1.0$$

Вариант 24

$$F(x,y)=\frac{x+y}{(x-y)^2}, a=3.0, b=4.0, c(x)=0, d(x)=x-1.0,$$

$$\int_a^b\int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x=3.7507, s=\int_a^b d(x)\mathrm{d}x=2.5$$

Вариант 25

$$F(x,y)=x^2-x\cdot y+2\cdot y^2, a=-1.0, b=1.0, c(x)=0, d(x)=1-x,$$

$$\int_a^b\int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x=4.0, s=\int_a^b d(x)\mathrm{d}x=2.0$$

Вариант 26

$$F(x,y)=\sqrt{-x^2+x+y^2}, a=0.0, b=1.0, c(x)=0, d(x)=x+1.0,$$

$$\int_a^b\int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x=1.3693, s=\int_a^b d(x)\mathrm{d}x=1.5$$

Вариант 27

$$F(x,y)=-x^4+y\cdot(x^2+y)^3, a=0.0, b=1.0, c(x)=0, d(x)=x+1.0,$$

$$\int_a^b\int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x=5.746, s=\int_a^b d(x)\mathrm{d}x=1.5$$

Вариант 28

$$F(x,y)=y\cdot e^{-x^2+y}, a=0.0, b=1.0, c(x)=0.0, d(x)=2.0\cdot x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 1.0586, s = \int_a^b d(x)\mathrm{d}x = 1.0$$

Вариант 29

$$F(x,y) = \frac{2.0 \cdot |x-y|}{(x+y)^2}, a = 0.0, b = 1.0, c(x) = 0, d(x) = 1.0 - x,$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 1.0, s = \int_a^b d(x)\mathrm{d}x = 0.5$$

Вариант 30

$$F(x,y) = x^2 \cdot y \cdot \cos^2(x), a = 0.0, b = \pi, c(x) = 0.0, d(x) = \sin(x),$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 0.6214, s = \int_a^b d(x)\mathrm{d}x = 2.0$$

Вариант 31

$$F(x,y) = 3 \cdot y^2 \cdot \sin^2(x), a = 0.0, b = \pi, c(x) = 0.0, d(x) = \sin(x),$$

$$\int_a^b \int_{c(x)}^{d(x)} F(x)\,\mathrm{d}y\mathrm{d}x = 1.0667, s = \int_a^b d(x)\mathrm{d}x = 2.0$$

Пример параллельного вычисления двойного интеграла методом Монте-Карло:

Исходный текст программы на языке Си:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <mpi.h>
#define   M_PI   3.14159265358979323846264643   //25-
digit-PI
//Подынтегральная функция
double F(double x, double y);
int main (int argc, char** argv){
```

```
    int rank, size, rc, tag, n;
    MPI_Status status;
    double partsum, globalsum, start, finish;
    double x, y, a, b, s, integralEstimate,
locElapsed, elapsed;
    long procTosses, totalPntIn, procPntIn;
    // общее количество испытаний
    long const totalTosses = 1e8;
    rc = MPI_Init(&argc, &argv);
    rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
    rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    tag = 50;
    // количество испытаний, приходящихся на один
процесс
    procTosses = totalTosses / size;
    // Задание площади интегрирования
    s = 2.0;
    // Пределы интегрирования по x
    a = 0.0;
    b = M_PI;
    start = MPI_Wtime();
    // инициализация генератора случайных чисел
        unsigned int seed = (unsigned) time(NULL);
    srand(seed + rank);
    // проведение испытаний и подсчет количества
"попавших" точек
    procPntIn = 0;
    partsum = 0.0;
    // Константа RAND_MAX является максимальным
значением, которое может возвращаться функцией
rand.
    // Отношение rand() / (double)RAND_MAX
соответствует случайной величине от 0 до 1
    for(long i = 0; i < procTosses; i++){
```

```
    // Генерация случайной величины в диапазоне от
a до  b
        x = a + rand() / (double)RAND_MAX * (b -
a);
        y = rand() / (double)RAND_MAX;
    // проверка попадания в площадь, ограниченную
кривыми c(x)=0, d(x)=sin(x)
        if(y <= sin(x)){
            procPntIn++;
            partsum += F(x,y);
        }
    }
    finish = MPI_Wtime();
    locElapsed = finish - start;
    // получаем максимальное время выполнения про-
цессов (операция MPI_MAX) и сохраняем его в пере-
менной elapsed
    rc  =  MPI_Reduce(&locElapsed,  &elapsed,  1,
MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
    // 0-й процесс (предпоследний аргумент root=0)
получает  глобальную  сумму  (операция  MPI_SUM)
частичных сумм (partsum)
    //  и  сохраняет  результат  в  переменной
globalsum
    // 0-й процесс (предпоследний аргумент root=0)
получает глобальную сумму (операция MPI_SUM)
    // получаем общее количество "попавших" точек
(сумму точек на каждом процессе)
    rc  =  MPI_Reduce(&procPntIn,  &totalPntIn,  1,
MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
        rc  =  MPI_Reduce(&partsum,  &globalsum,  1,
MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if(rank == 0){
```

```
        integralEstimate = s * globalsum / totalPntIn;
        printf( "Result: %.16f Proc: %d totalTosses: %ld Runtime: %f \n", integralEstimate, size, totalTosses, elapsed);
    }
    rc = MPI_Finalize();
    return rc;
}
double F(double x, double y){
    return 3 * y * y * sin(x) * sin(x);
}
```

Исходный текст программы на языке Python:

```
import random
import time
import math
from mpi4py import MPI
def F(x, y) -> float:
    return 3 * y * y * math.sin(x) * math.sin(x)
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
# общее количество испытаний
totalTosses = 1e8
# количество испытаний, приходящихся на один процесс
procTosses = int(totalTosses / size)
# Задание площади интегрирования
s = 2.0
# засекаем начало времени выполнения
start = MPI.Wtime()
# Пределы интегрирования по x
a = 0.0
b = math.pi
```

```
# инициализация генератора случайных чисел
seed = time.time()
random.seed(seed + rank)
# проведение испытаний и подсчет количества "попавших" в сектор точек
procPntIn = 0
partsum = 0.0
for i in range(0, procTosses):
    # Генерация случайной величины в диапазоне от a до  b
    x = a + random.random() * (b - a)
    y = random.random()
    # проверка попадания в площадь, ограниченную кривыми c(x)=0, d(x)=sin(x)
    if y <= math.sin(x):
        procPntIn = procPntIn + 1
        partsum = partsum + F(x, y)
totalPntIn = comm.reduce(procPntIn, op = MPI.SUM, root = 0)
globalsum = comm.reduce(partsum, op = MPI.SUM, root = 0)
finish = MPI.Wtime()
locElapsed = finish - start
# получаем максимальное время выполнения процессов (операция MPI_MAX) и сохраняем его в переменной elapsed
elapsed = comm.reduce(locElapsed, op = MPI.MAX, root = 0)
if rank == 0:
    integralEstimate = s * globalsum / totalPntIn
    print( "Result:", integralEstimate, "Proc:", size, "totalTosses:", totalTosses, "Runtime:", elapsed)
```

4.4. ЛАБОРАТОРНАЯ РАБОТА № 4
«ПАРАЛЛЕЛЬНОЕ УМНОЖЕНИЕ МАТРИЦ»

1. Скачайте и установите требуемые для работы программные пакеты (Microsoft MPI, Microsoft C++ Build Tools/Python + mpi4py). Этот пункт будет выполнен автоматически, если была успешно выполнена Лабораторная работа № 1.

2. Скачайте папки с примерами "example10" и "example14" параллельного умножения матриц. Каждый процесс имеет в распоряжении все необходимые данные, чтобы независимо рассчитать свою горизонтальную полосу матрицы C (горизонтальная полоса представляет собой несколько последовательных строк матрицы, которые будут обрабатываться соответствующим процессом).

Основные отличия примеров следующие. "example10":

1. Перемножаемые матрицы A и B целиком хранятся в памяти каждого процесса, результирующая матрица C хранится также в памяти каждого процесса (все три матрицы A, B и C хранятся в форме статических массивов). Перемножение матриц осуществляется поэлементно в форме трех вложенных циклов. "example14".

2. В памяти каждого процесса целиком хранится только матрица B. Матрицы A и C в процессе умножения хранятся в распределенном виде, т.е. в виде горизонтальных полос, при этом каждый процесс хранит часть строк матрицы A и матрицы C (строки матриц A и C как бы «размазаны» по всем процессам, при этом для хранения матриц используются динамические массивы). Для умножения матриц используются возможности специализированных библиотек (numpy для Python и OpenBLAS для C).

3. На языке C и Python скомпилируйте (если программа на языке C) и запустите представленные примеры параллельных MPI-программ, вычисляющих произведение матриц. Запуск осуществляйте с использованием "mpiexec" с соответствующими параметрами, как это осуществлялось при выполнении Лабораторной работы № 1. Перед компиляцией "example14" необходимо распаковать архив

"OpenBLAS-0.3.26-x64.zip", находящийся в папке "openblas" в саму эту же папку (после распаковки внутри папки "openblas" должны появиться папки "bin", "include" и "lib" с соответствующим содержимым).

4. Прочитайте файл "Номера вариантов Распределенная обработка.docx", в котором представлены номера вариантов.

5. В файле с заданиями "MatrixVariants.pdf" представлены формулы для инициализации матриц A и B размерностью 100×100 элементов (MSIZE=100), а также результаты вычислений следов результирующей матрицы C (matrix trace) для данной размерности. След матрицы – это сумма элементов главной диагонали. Эти результаты можно использовать как «правильные ответы» для проверки полученных результатов.

6. Осуществить перемножение согласно своему варианту с использованием параллельной программы на языке программирования C и Python двумя способами соответственно.

7. В примерах задана размерность матриц 100×100 элементов (MSIZE=100). Запустите программы, повысив размерность (изменив значение константы MSIZE) до 5000 на разном количестве процессов, начиная с одного ("mpiexec –n 1") до 10. Сравните полученные результаты, в частности время выполнения программы. Заполнить в таблице 1 строки «Время работы алгоритма».

8. Рассчитать значения ускорений и эффективности и заполнить остальные ячейки таблицы.

9. Построить графики зависимости ускорения и эффективности от количества используемых процессоров.

10. Сделать выводы по полученным результатам.

11. Обратите внимание, что в примерах используются функции MPI_Bcast(), MPI_Scatter(), MPI_Gather(). Изучить назначение функций, листинги программ, уметь отвечать на вопросы по работе программы.

Отчет. Для отчета необходимо сохранить скриншоты окон с выводами результатов работы параллельной программы.

4. Расчет ускорение и эффективности параллельной MPI-программы

	Вид умножения	Язык програм-мирования	Количество процессоров					
			1	2	3	4	...	10
Время выполнения алгоритма, с	example10 (поэлементное умножение)	C						
		Python						
	example14 (использование BLAS)	C						
		Python						
Ускорение (время выполнения для 1 процессора, деленное на время выполнения для N-процессоров)	example10 (поэлементное умножение)	C	1					
		Python	1					
	example14 (использование BLAS)	C	1					
		Python	1					
Эффективность (значение ускорения для *N*-процессоров, деленное на число процессоров *N*)	example10 (поэлементное умножение)	C						
		Python						
	example14 (использование BLAS)	C						
		Python						

В процессе выполнения работы можно пользоваться следующими ресурсами.

1. Пример использования процедуры xGEMM из библиотеки BLAS в программе на Си:
http://www.ccfit.nsu.ru/~kireev/lab4/lab4blas.htm

2. Linear algebra (numpy.linalg):
https://numpy.org/devdocs/reference/routines.linalg.html

3. Джанкарло Закконе. Книга рецептов параллельного программирования Python. 2-е изд.:
http://onreader.mdl.ru/PythonParallelProgrammingCookbook.2nd/content/index.html

4. Программированию с использованием MPI посвящен четвертый раздел книги:
http://onreader.mdl.ru/PythonParallelProgrammingCookbook.2nd/content/Ch04.html

5. Microsoft MPI:
https://learn.microsoft.com/ru-ru/message-passing-interface/microsoft-mpi

7. Самостоятельный поиск ресурсов в Интернете, посвященных тематике данной лабораторной работы.

ИНИЦИАЛИЗАЦИЯ МАТРИЦ
ВАРИАНТЫ ЗАДАНИЙ

Вариант 1

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \sin(i \cdot j), b_{i,j} = \sin(i + j), C = A \cdot B, \text{trace}(C) = 55.5103$$

Вариант 2

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \cos(i \cdot j), b_{i,j} = \cos(i + j), C = A \cdot B, \text{trace}(C) = 151.3205$$

Вариант 3

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = i + j, b_{i,j} = \sin(i \cdot j), C = A \cdot B, \text{trace}(C) = 15741.1469$$

Вариант 4

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \frac{1}{i + j + 1}, b_{i,j} = \frac{1}{|i - j| + 1}, C = A \cdot B, \text{trace}(C) = 15.0612$$

Вариант 5

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = i + j, b_{i,j} = \sqrt{\frac{1}{(i + 1) \cdot (j + 1)}}, C = A \cdot B, \text{trace}(C) = 24273.3136$$

Вариант 6

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \sqrt{i} + \sqrt{j}, b_{i,j} = \frac{1}{i + j + 1}, C = A \cdot B, \text{trace}(C) = 1473.0553$$

Вариант 7

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \cos(i \cdot j), b_{i,j} = \sin(i + j), C = A \cdot B, \text{trace}(C) = 81.3094$$

Вариант 8

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \frac{i + j}{|i - j| + 1}, b_{i,j} = \frac{|i - j|}{i + j + 1}, C = A \cdot B, \text{trace}(C) = 9129.0802$$

Вариант 9

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \frac{1}{(i + 1) \cdot (j + 1)}, b_{i,j} = |i - j|, C = A \cdot B, \text{trace}(C) = 647.8503$$

Вариант 10

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \frac{1}{j + 1} + \frac{1}{i + 1}, b_{i,j} = \frac{1}{(i + 1) \cdot (j + 1)}, C = A \cdot B, \text{trace}(C) = 16.9626$$

Вариант 11

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \ln(i + j + 1), b_{i,j} = e^{\frac{1}{i+j+1}}, C = A \cdot B, \text{trace}(C) = 45470.0525$$

Вариант 12

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = i \cdot \sin(j), b_{i,j} = j \cdot \cos(i), C = A \cdot B, \text{trace}(C) = 98715.9899$$

Вариант 13

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \sin(j) \cdot \cos(i), b_{i,j} = \tan(i + j), C = A \cdot B, \text{trace}(C) = 862.1602$$

Вариант 14

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = i \cdot j, b_{i,j} = \frac{1}{i^2 + j^2 + 1}, C = A \cdot B, \text{trace}(C) = 3428.0328$$

Вариант 15

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \sin(i \cdot j), b_{i,j} = \cos\left(\frac{i}{j + 1}\right), C = A \cdot B, \text{trace}(C) = 49.19$$

Вариант 16

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \sqrt{i^2 + j^2}, b_{i,j} = \frac{1}{\sqrt{i^2 + j^2} + 1}, C = A \cdot B, \text{trace}(C) = 9827.0296$$

Вариант 17

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \frac{1}{\sqrt{i^2 + j^2 + 1}}, b_{i,j} = e^{\frac{1}{i^2 + j^2 + 1}}, C = A \cdot B, \text{trace}(C) = 183.2191$$

Вариант 18

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \tan(i \cdot j), b_{i,j} = \frac{1}{\tan(i + j) + 1}, C = A \cdot B, \text{trace}(C) = 8410.6168$$

Вариант 19

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \tan(i + j), b_{i,j} = \sin(i \cdot j), C = A \cdot B, \text{trace}(C) = 995.8225$$

Вариант 20

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \tan(i \cdot j), b_{i,j} = \ln(i + j + 1), C = A \cdot B, \text{trace}(C) = 6402.015$$

Вариант 21

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \frac{1}{j+1} + \frac{1}{i+1}, b_{i,j} = \sin(i \cdot j), C = A \cdot B, \text{trace}(C) = 4.4226$$

Вариант 22

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \ln(i + j + 1), b_{i,j} = \cos(i + j), C = A \cdot B, \text{trace}(C) = 1.7153$$

Вариант 23

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \cos(i \cdot j), b_{i,j} = e^{\frac{1}{i^4 + j^2 + 1}}, C = A \cdot B, \text{trace}(C) = 193.8205$$

Вариант 24

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \sin(i) \cdot \cos(j), b_{i,j} = \frac{i + j}{|i - j| + 1}, C = A \cdot B, \text{trace}(C) = 78.7507$$

Вариант 25

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \cos(i + j), b_{i,j} = \tan\left(\frac{i}{j+1}\right), C = A \cdot B, \text{trace}(C) = 960.8621$$

Вариант 26

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \frac{|i-j|}{i+j+1}, b_{i,j} = \frac{1}{(i+1)\cdot(j+1)}, C = A \cdot B, \text{trace}(C) = 16.5572$$

Вариант 27

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \frac{1}{i+j+1}, b_{i,j} = \ln(i+j+1), C = A \cdot B, \text{trace}(C) = 548.7524$$

Вариант 28

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \frac{1}{j+1} + \frac{1}{i+1}, b_{i,j} = i \cdot \sin(j), C = A \cdot B, \text{trace}(C) = 2698.7739$$

Вариант 29

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \sin\left(\frac{i}{j+1}\right), b_{i,j} = \tan(i \cdot j), C = A \cdot B, \text{trace}(C) = 118.8327$$

Вариант 30

$$M_{SIZE} = 100, i \in [0;\ 100), j \in [0;\ 100), a_{i,j} = \frac{1}{i^3 + j^3 + 1}, b_{i,j} = i \cdot j, C = A \cdot B, \text{trace}(C) = 73.3144$$

Пример параллельного умножения матриц.

В данном примере для простоты предполагаем, что память экономить нет необходимости.

Поэтому перемножаемые матрицы A и B целиком хранятся в памяти каждого процесса, результирующая матрица C хранится также в памяти каждого процесса.

Таким образом, каждый процесс имеет в распоряжении все необходимые данные, чтобы рассчитать свою полосу (блочную строку) матрицы C.

Блочная строка представляет сбой несколько строк матрицы, которые будут обрабатываться соответствующим процессом.

Поэтому необходимо, чтобы количество строк исходных матриц без остатка делилось на количество процессов.

Пример.

Допустим, размерности матриц A, B и C 10×10. Номера строк и столбцов нумеруются с нуля до девяти.

Требуется разрезать эти матрицы на полосы (блочные строки) для параллельной обработки каждым процессом.

Для двух процессов (процесс 0 и процесс 1) ширина полоса будет 10 / 2 = 5:

нулевой процесс обрабатывает строки с нулевой по четвертую, второй с пятой по девятую.

Для пяти процессов (процесс 0, 1, ..., 4) ширина полоса будет 10 / 5 = 2:

первый процесс обрабатывает строки с нулевой по первую, второй со второй по третью, ..., пятый процесс с восьмой по девятую.

Для десяти процессов (процесс 0, 1, ..., 9) ширина полоса будет 10 / 10 = 1:

нулевой процесс обрабатывает нулевую строку, первый – первую, второй – вторую, ..., девятый процесс – девятую.

Использование трех, четырех, шести, семи, восьми и девяти процессов для рассматриваемого подхода невозможно, так как ширина обрабатываемой полосы должно быть целым числом.

Исходный текст программы на языке Си:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
// размерность матрицы
#define MSIZE 100
// для простоты используются статические массивы
double A[MSIZE][MSIZE], B[MSIZE][MSIZE],
C[MSIZE][MSIZE];

int main (int argc, char** argv){
    int rank, size, rc;
    MPI_Status status;
    int istart, iend, bandsize;
    double start, finish, locElapsed, elapsed;
    rc = MPI_Init(&argc, &argv);
    rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
    rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // каждый процесс обрабатывает свою полосу
из нескольких строк матрицы
    // проверка: делится ли размерность исходной
матрицы на количество процессов?
    if (MSIZE % size != 0) {
        if (rank==0)
            printf("Matrix size not divisible by
number of processors\n");
        MPI_Finalize();
        exit(-1);
    }
    // засекаем время
    start = MPI_Wtime();
    // 0-й процесс инициализирует матрицы A и B
```

```
    if (rank == 0) {
        for (int i = 0; i < MSIZE; i++)
            for(int j = 0; j < MSIZE; j++){
                A[i][j] = (i+1) * (j+1);
                B[i][j] = 1 / A[i][j];
                C[i][j] = 0.0;
            }
    }
    // Каждый процесс определяет первый
и последнюю строку своего расчета
    istart = rank * MSIZE / size;
    iend = (rank + 1) * MSIZE / size;

    // перемножаемые матрицы A и B хранятся
целиком в памяти каждого процесса
    // результирующая матрица C также хранится
целиком в памяти КАЖДОГО процесса
    // "0"-й процесс (аргумент root = 0) рассылает
всем матрицы A и B
    rc = MPI_Bcast(A, MSIZE * MSIZE, MPI_DOUBLE,
0, MPI_COMM_WORLD);
    rc = MPI_Bcast(B, MSIZE * MSIZE, MPI_DOUBLE,
0, MPI_COMM_WORLD);
    rc = MPI_Bcast(C, MSIZE * MSIZE, MPI_DOUBLE,
0, MPI_COMM_WORLD);
    // Каждый процесс ведет расчет свой полосы
матрицы C
    for(int i = istart; i < iend; i++)
        for(int j = 0; j < MSIZE; j++) {
            C[i][j] = 0.0;
            for (int k = 0; k < MSIZE; k++)
                C[i][j] = C[i][j] + A[i][k] *
B[k][j];
```

```
//                C[i][j] += A[i][k] * B[k][j];
        }
    // Каждый процесс производит рассылку своей
полосы остальным процессам
    double (* C_local)[MSIZE];
    for (int k = 0; k < size; k++){
    // Каждый процесс определяет первый
и последнюю строку своего расчета
        istart = k * MSIZE / size;
        iend = (k + 1) * MSIZE / size;
        // ширина полосы k-го процесса
        bandsize = iend - istart;
        C_local = C + istart;
        rc = MPI_Bcast(C_local, bandsize * MSIZE,
MPI_DOUBLE, k, MPI_COMM_WORLD);
    }

    finish = MPI_Wtime();
    locElapsed = finish - start;
    // получаем максимальное время выполнения
процессов (операция MPI_MAX) и сохраняем его
в переменной elapsed
//  rc = MPI_Reduce(&locElapsed, &elapsed, 1,
MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
//  if (rank == 0) {
        // считаем трек (сумму элементов главной
диагонали) итоговой матрицы C
        double track = 0.0;
        for (int i = 0; i < MSIZE; i++){
            track = track + C[i][i];
        }
```

```
        printf("C: MatrixSize: %d Size: %d Rank:
%d Runtime: %f Track: %f\n", MSIZE, size, rank,
locElapsed, track);
//  }
    rc = MPI_Finalize();
    return rc;
}
```

Исходный текст программы на языке Python:

```
import sys
import numpy as np
from mpi4py import MPI
# размерность матрицы
MSIZE = 100
# создание двумерного массива
A = np.empty(shape=(MSIZE, MSIZE),
dtype='float64')
B = np.empty(shape=(MSIZE, MSIZE),
dtype='float64')
C = np.empty(shape=(MSIZE, MSIZE),
dtype='float64')
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
# каждый процесс обрабатывает свою полосу
из нескольких строк матрицы
# проверка: делится ли размерность исходной
матрицы на количество процессов?
if (MSIZE % size != 0):
    if (rank == 0):
        print("Matrix size not divisible by number
of processors")
```

```
    sys.exit(-1)
# засекаем время
start = MPI.Wtime()
# 0-й процесс инициализирует матрицы A и B
if (rank == 0):
    for i in range(0, MSIZE):
        for j in range(0, MSIZE):
            A[i][j] = (i+1) * (j+1)
            B[i][j] = 1 / A[i][j]
            C[i][j] = 0.0
# перемножаемые матрицы A и B хранятся целиком
в памяти каждого процессорного элемента
# результирующая матрица C также хранится целиком
в памяти КАЖДОГО процессорного элемента
# 0-й процесс (аргумент root = 0) рассылает всем
матрицы A и B
A = comm.bcast(A, root = 0)
B = comm.bcast(B, root = 0)
C = comm.bcast(C, root = 0)
# Каждый процесс определяет первую и последнюю
строку своего расчета
istart = int(rank * MSIZE / size)
iend = int((rank + 1) * MSIZE / size)

for i in range(istart, iend):
    for j in range(0, MSIZE):
        C[i][j] = 0.0;
        for k in range(0, MSIZE):
            C[i][j] = C[i][j] + A[i][k] * B[k][j]
# Каждый процесс производит рассылку своей полосы
остальным процессам
for k in range(0, size):
```

```
    # Каждый процесс определяет первый и последнюю
строку своего расчета
    istart = int(k * MSIZE / size)
    iend = int((k + 1) * MSIZE / size)
    # ширина полосы k-го процесса
    bandsize = iend - istart
    C_local = C[istart:iend]
    C_local = comm.Bcast(C_local, root = k)
finish = MPI.Wtime()
locElapsed = finish - start
# получаем максимальное время выполнения процессов
(операция MPI.MAX) и сохраняем его в переменной
elapsed
# считаем трек (сумму элементов главной диагонали)
итоговой матрицы C
track = 0.0
for i in range(0, MSIZE):
    track = track + C[i][i]
print("Python: MatrixSize:", MSIZE, "Size:", size,
"Rank", rank, "Runtime:", locElapsed, "Track:",
track)
#print("numpy function: Track:", np.trace(C))
```

Пример параллельного умножения матриц.

В данном примере память для хранения матриц выделяется динамически.

В программе на языке C для хранения матриц используются динамические одномерные массивы размерностью M[ROWS * COLS] (где ROWS – количество строк в матрице,

COLS – количество столбцов в матрице), при этом доступ к этому одномерному массиву как к двумерному осуществляется по принципу M[i][j] = M[i * ROWS + j].

Кроме того, для умножения матриц используются возможности специализированных библиотек (numpy для Python и OpenBLAS для C).

Пример.

В отличие от предыдущего примера, в памяти каждого процесса целиком хранится только матрица B.

Матрицы A и C в процессе умножения хранятся в распределенном виде, т.е. в виде горизонтальных полос, при этом каждый процесс хранит часть строк матрицы A и матрицы C в соответствующих локальных массивах local_A и local_C.

Количество строк в каждой полосе равно размерности матрицы, деленной на количество используемых процессов, при этом результат деления должен быть целым числом (количество строк исходных матриц A и C без остатка делилось на количество используемых процессов).

Нулевой процесс инициализирует исходные матрицы A и B, затем делит матрицу A на полосы local_A соответственно количеству запущенных процессов и рассылает эти полосы local_A с помощью функции MPI_Scatter() и матрицу B целиком с помощью функции MPI_Bcast() всем остальным процессам, включая нулевой.

Затем каждый процесс, включая нулевой, проводит умножение local_C = local_A * B.

После выполнения умножения нулевой процесс с использованием функции MPI_Gather() собирает соответствующие полосы local_C от каждого процесса и сохраняет конечный результат в матрице C.

Результирующая матрица C хранится целиком только на нулевом процессе.

Исходный текст программы на языке Си:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <cblas.h>
// размерность матриц
#define MSIZE 100
int main(int argc, char *argv[]) {
    int rank, size, rc;
    double start, finish, locElapsed, elapsed;
    MPI_Status status;
    // будут использоваться динамические
одномерные массивы размерностью M[ROWS * COLS]
    // доступ как к двумерному массиву осуществля-
ется по принципу M[i][j] = M[i * ROWS + j]
    double *local_A, *B, *local_C, *A, *C;
    // локальная ширина полосы матрицы, обрабаты-
ваемая каждым процессом
    int BANDSIZE;
    rc = MPI_Init(&argc, &argv);
    rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
    rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    // каждый процесс обрабатывает свою полосу из
нескольких строк матрицы
    // проверка: делится ли размерность исходной
матрицы на количество процессов
    if (MSIZE % size != 0) {
        if (rank==0)
            printf("Matrix size not divisible by
number of processors\n");
        MPI_Finalize();
        exit(-1);
```

```
    }
    BANDSIZE = MSIZE / size;
    // динамическое выделение памяти
    local_A = (double *)malloc(BANDSIZE * MSIZE *
sizeof(double));
    B = (double *)malloc(MSIZE * MSIZE *
sizeof(double));
    local_C = (double *)malloc(BANDSIZE * MSIZE *
sizeof(double));
    // засекаем время
    start = MPI_Wtime();
    // 0-й процесс инициализирует матрицы A и B
    if (rank == 0) {
        A = (double *)malloc(MSIZE * MSIZE *
sizeof(double));
        C = (double *)malloc(MSIZE * MSIZE *
sizeof(double));
        for (int i = 0; i < MSIZE; i++)
            for(int j = 0; j < MSIZE; j++){
                A[i * MSIZE + j] = (i + 1) * (j +
1);
                B[i * MSIZE + j] = 1 / A[i *
MSIZE + j];
                C[i * MSIZE + j] = 0.0;
            }
        }
    else{
        A = NULL;
        C = NULL;
    }
        // Рассылка матрицы B всем процессам
```

```
        rc = MPI_Bcast(B, MSIZE * MSIZE,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
    // Рассылка частей полосы матрицы A каждому
процессу
    rc = MPI_Scatter(A, BANDSIZE * MSIZE,
MPI_DOUBLE, local_A, BANDSIZE * MSIZE, MPI_DOUBLE,
0, MPI_COMM_WORLD);
    // Каждый процесс перемножает свою часть поло-
сы матрицы A на матрицу B и сохраняет в локальной
матрице C
    // используем функцию из библиотеки BLAS, опи-
сание можно посмотреть, например, на
http://www.ccfit.nsu.ru/~kireev/lab4/lab4blas.htm
    cblas_dgemm(CblasRowMajor, CblasNoTrans,
CblasNoTrans, BANDSIZE, MSIZE, MSIZE, 1.0, lo-
cal_A, MSIZE, B, MSIZE, 0.0, local_C, MSIZE);
    // Нулевой процесс собирает результаты и со-
храняет в матрице C
    if (rank == 0) {
            C = (double *)malloc(MSIZE * MSIZE *
sizeof(double));
    }
    rc = MPI_Gather(local_C, BANDSIZE * MSIZE,
MPI_DOUBLE, C, BANDSIZE * MSIZE, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
    finish = MPI_Wtime();
    locElapsed = finish - start;
    // получаем максимальное время выполнения про-
цессов (операция MPI_MAX) и сохраняем его в пере-
менной elapsed
    rc = MPI_Reduce(&locElapsed, &elapsed, 1,
MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
```

```
    // Вывод результата только на нулевом процессе
    if (rank == 0) {
/*      for (int i = 0; i < MSIZE; i++){
            printf("\n| ");
            for (int j = 0; j < MSIZE; j++)
                printf("%.5f ", C[i * MSIZE +
j]);
            printf("|");
        }
*/
        // считаем трек (сумма элементов главной
диагонали) итоговой матрицы C
        double track = 0.0;
        for (int i = 0; i < MSIZE; i++){
            track = track + C[i * MSIZE + i];
        }
        printf("C: MatrixSize: %d, BandSize: %d,
Size (Proc Number): %d, Runtime: %f, Track: %f\n",
MSIZE, BANDSIZE, size, elapsed, track);
        free(A);
        free(C);
    }
    free(local_A);
    free(B);
    free(local_C);

    MPI_Finalize();
    return rc;
}
```

Исходный текст программы на языке Python:

```
    # https://numpy.org/devdocs/reference/
routines. linalg.html
    # The NumPy linear algebra functions rely on
BLAS and LAPACK to provide efficient low level
implementations of standard linear algebra algo-
rithms.
    # Метод show_config() возвращает свойства
библиотеки numpy, в частности версию библиотеки
OpenBLAS
    import numpy as np
    np.show_config()
    import sys
    import numpy as np
    from mpi4py import MPI
    # размерность матриц
    MSIZE = 100
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    numproc = comm.Get_size()
    # каждый процесс обрабатывает свою полосу
из нескольких строк матрицы
    # проверка: делится ли размерность исходной
матрицы на количество процессов
    if (MSIZE % numproc != 0):
        if (rank == 0):
            print("Matrix size not divisible by
number of processors")
        sys.exit(-1)
    # локальная ширина полосы матрицы, обрабатыва-
емая каждым процессом
    BANDSIZE = int(MSIZE // numproc)
```

```
    # Локальные массивы для хранения соответствую-
щей полосы матрицы на каждом процессе
    local_A = np.empty((BANDSIZE, MSIZE),
dtype='float')
    local_C = np.empty((BANDSIZE, MSIZE),
dtype='float')
    # засекаем время
    start = MPI.Wtime()
    # создание двумерного массива
    B = np.empty(shape=(MSIZE, MSIZE),
dtype='float')
    # 0-й процесс инициализирует матрицы A и B
    if (rank == 0):
        A = np.empty(shape=(MSIZE, MSIZE),
dtype='float')
        C = np.empty(shape=(MSIZE, MSIZE),
dtype='float')
        for i in range(0, MSIZE):
            for j in range(0, MSIZE):
                A[i][j] = (i+1) * (j+1)
                B[i][j] = 1 / A[i][j]
                C[i][j] = 0.0
    else:
        A = None
        C = None
    # Рассылка матрицы B всем процессам
    B = comm.bcast(B, root=0)
    # Рассылка частей полосы матрицы A каждому
процессу
    comm.Scatter(A, local_A, root=0)
```

```
    # Для перемножения матриц используется метод
numpy.dot.
    #
https://numpy.org/devdocs/reference/generated/nump
y.dot.html
    # Цитата: It uses an optimized BLAS library
when possible
    #
https://numpy.org/devdocs/reference/routines.linal
g.html
    # The NumPy linear algebra functions rely on
BLAS and LAPACK to provide efficient low level
implementations of standard linear algebra algo-
rithms.
    # Каждый процесс перемножает свою часть полосы
матрицы A на матрицу B и сохраняет в локальной
матрице C
    local_C = np.dot(local_A, B)
    # Нулевой процесс собирает результаты и сохра-
няет в матрице C
    comm.Gather(local_C, C, root=0)
    finish = MPI.Wtime()
    locElapsed = finish - start
    # получаем максимальное время выполнения про-
цессов (операция MPI.MAX) и сохраняем его в пере-
менной elapsed
    elapsed = comm.reduce(locElapsed, op =
MPI.MAX, root = 0)
    if rank == 0:
        track = 0.0;
```

```
        for i in range(0, MSIZE):
            track = track + C[i][i]
        print(f"Python: MatrixSize: {MSIZE},
BandSize: {BANDSIZE}, Numproc: {numproc}, Runtime:
{elapsed}, Track: {track}")
        print(f"numpy function: Track:
{np.trace(C)}")
```

4.5. ЛАБОРАТОРНАЯ РАБОТА № 5 «КОМПИЛЯЦИЯ И ЗАПУСК ПРИЛОЖЕНИЙ НА ВЫЧИСЛИТЕЛЬНОМ КЛАСТЕРЕ С ИСПОЛЬЗОВАНИЕМ SLURM»

В настоящий момент учебный кластер состоит из четырех узлов (каждый узел AMD Ryzen 5 5600X 6-Core Processor (6 ядер, 12 потоков), 16Gb RAM). IP-адрес сервера для удаленного подключения **192.168.154.12**, **пользователь mpiuser**, **пароль mpiuser**. В качестве операционной системы используется Linux Mint. Установлены необходимые средства разработки, в том числе библиотеки OpenMPI и OpenBlas.

Для работы на вычислительном кластере необходимы SSH (Secure Shell)-клиент для удаленного терминального доступа к узлам кластера, а также SCP (Secure Copy Protocol)-клиент для загрузки необходимых файлов с локального компьютера на узлы кластера. В составе Microsoft Windows последних версий есть консольные утилиты ssh и scp, однако, возможно использование и сторонних программ, например putty в качестве SSH-клиента и WinSCP в качестве SCP-клиента. Еще одной программой, которую можно использовать для передачи файлов на сервер является Far Manager.

1. Подключитесь к серверу, запустив консоль Windows и выполнив в ней команду:

> ssh -l mpiuser 192.168.154.12

После подключения смените папку на "cloud":

> cd ~/cloud

Следует отметить, что символ "~" является псевдонимом домашнего каталога пользователя, например "/home/mpiuser/". Например, если пользователь зашел под именем mpiuser, то перейти в папку cloud, находящуюся в его домашнем каталог, можно с помощью команды

> cd /home/mpiuser/cloud

Более короткий синтаксис:

> **cd ~/cloud**

При первом подключении создайте рабочую папку с именем, соответствующим фамилии студента, например Sidorov:

> mkdir Sidorov

Затем перейдите в нее:

> cd Sidorov

Эта папка будет рабочей папкой, в которой далее будет происходить компиляция и запуск программ.

В дальнейшем создавать повторно эту папку не нужно, достаточно перейти в нее сразу после подключения:

> cd ~/cloud/Sidorov

Для проверки текущей рабочей папки введите команду pwd (print working directory):

> pwd

В результате система должна напечатать что-то вроде:

/home/mpiuser/cloud/Sidorov

2. Далее необходимо скопировать с локального компьютера в эту папку требуемые файлы, например "multmatrix.c". Для этого, например, необходимо запустить консоль Windows, сменить локальную папку на ту, в которой находятся требуемые файлы, и затем выполнить команду:

> scp multmatrix.c mpiuser@192.168.154.12:~/cloud/Sidorov

Эта команда скопирует файл "multmatrix.c", находящийся в текущей папке локального компьютера, в папку "~/cloud/Sidorov" удаленного компьютера с адресом **192.168.154.12**.

Аналогично можно скопировать на кластер и другие файлы, например "run.sh", "multmatrix.py" и т.п.

3. Для компиляции С-программы под операционной системой Linux после подключения к кластеру необходимо перейти в Вашу рабочую папку (Sidorov в примере),

> cd ~/cloud/Sidorov

а затем выполнить в терминале команду:

> mpicc multmatrix.c

Если в программе используется библиотека "math.h", то добавить ключ "–lm":

> mpicc multmatrix.c –lm

Если в программе используются библиотеки "math.h" и "cblas.h", то добавить ключ "–lblas":

> mpicc multmatrix.c –lm – lblas

После успешной компиляции в рабочем каталоге появится исполняемый файл "**a.out**" (компилятор по-умолчанию выдает файл с именем a.out, если не используется опция -o).

4. Для того, чтобы запустить задачу на кластере, необходимо использовать систему управления заданиями Slurm. Для этого создается специальный скрипт и ставится в очередь с помощью команды sbatch. Пример такого скрипта с именем "run.sh" находится в папке с лабораторной работой "Lab5". Содержимое скрипта "run.sh" выглядит следующим образом:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#!/bin/bash
#SBATCH --job-name=borisenko          # Название задачи. Поменять на свою фамилию!
#SBATCH --error={task}-%j.err         # Файл для вывода ошибок
```

```
    #SBATCH --output={task}-%j.log       # Файл для
вывода результатов
    #SBATCH --time=01:00:00             # Максималь-
ное время выполнения
    #SBATCH --nodes=4                    # Требуемое
кол-во узлов
    #SBATCH --ntasks-per-node=6 # Требуемое количе-
ство процессов на одном узле
    orterun ./a.out
    #orterun python3 ./doubleintegral.py
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~
```

Перед тем, как загружать скрипт на сервер в рабочую папку, целесообразно поменять параметр "--job-name=borisenko" на свою фамилию, в данном примере "--job-name=sidorov". В дальнейшем этот скрипт можно открыть и редактировать прямо в рабочем каталоге на кластере с помощью, например, текстового редактора "nano":

> nano run.sh

Обратите внимание на параметры *"--nodes=4"* (количество узлов кластера, на данный момент их максимум 4) и *"--ntasks-per-node=6"* (количество процессов на одном узле, на используемом оборудование их максимум может быть 12). С их помощью можно управлять количеством задействованных при выполнении параллельной MPI-программы процессов. Для представленных значений общее количество процессов будет 6×4=24.

Данный файл предназначен для запуска программы с именем a.out (строка *"orterun ./a.out"*). Для запуска программы на Python необходимо закомментировать строку *"orterun ./a.out"*

поставив в начале строки знак комментария "#", и убрать знак комментария в начале строки *"orterun python3 ./doubleintegral.py"* (имя скрипта *"doubleintegral.py"* при необходимости заменить на соответствующий варианту).

Отправка задачи на кластер осуществляется с помощью команды "**sbatch**":

> sbatch run.sh

Просмотреть состояние задач в очереди можно с помощью команды "**squeue**":

Если по каким-то причинам задача так и не начала запускаться (например, запрошено слишком много ядер, или памяти), то удалить задачу из очереди можно с помощью команды "**scancel** <job_id>", где <job_id> – идентификатор здания Точно так же задачу можно удалить, если она уже выполняется (при этом она будет сразу завершена).

Команда "**sinfo**" используется для просмотра состояния вычислительных узлов и очередей SLURM.

5. Прочитайте файл "Номера вариантов БВТ201 Распределенная обработка.docx", в котором представлены номера вариантов.

6. Прочитайте задания для лабораторной работы, которые находятся в файле "Задания к лабораторной работе.docx".

7. Выполните задание согласно своему варианту с использованием параллельной программы на языке программирования C и Python. Количество используемых процессов следует задавать изменяя параметры *"--nodes"* и *"--ntasks-per-node"* в скрипте запуска *"run.sh"*. Например, для запуска программы на одном процессе параметры будут *"--nodes=1"* и *"--ntasks-per-node=1"*; для запуска программы на 48 процессах параметры будут *"--nodes=4"* и *"--ntasks-per-node=12"*. При этом следует заметить, что, запуск параллельной программы допустим на 24 процессах, можно

осуществить на двух узлах (*"--nodes=2"* и *"--ntasks-per-node=12"*), трех узлах (*"--nodes=3"* и *"--ntasks-per-node=8"*), четырех узлах (*"--nodes=4"* и *"--ntasks-per-node=6"*). Какой вариант запуска наиболее предпочтительный – вопрос дискуссионный.

8. Рассчитать значения ускорений и эффективности и заполнить остальные ячейки таблицы.

9. Построить графики зависимости ускорения и эффективности от количества используемых процессоров.

10. Сделать выводы по полученным результатам.

11. Обратите внимание, что в примерах используются функции MPI_Bcast(), MPI_Scatter(), MPI_Gather(). Изучить назначение функций, листинги программ, уметь отвечать на вопросы по работе программы.

Отчет. Для отчета необходимо сохранить скриншоты окон с выводами результатов работы параллельной программы.

5. Расчет ускорение и эффективности параллельной MPI-программы

	Язык программирования	Количество процессоров					
		1				...	*N*
Время выполнения алгоритма, с	C						
	Python						
Ускорение (время выполнения для 1 процессора, деленное на время выполнения для *N*-процессоров)	C	1					
	Python	1					
Эффективность (значение ускорения для *N*-процессоров, деленное на число процессоров *N*)	C						
	Python						

6. Базовые команды Linux

Команда	Описание
mkdir <каталог>	Создать новый каталог с именем <каталог>
ls	Отобразить список файлов в текущем каталоге
ls -la	Отобразить подробный список файлов в текущем каталоге, включая скрытые файлы
cd <путь>	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 файлы или переименовать файл
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

7. Команды редактора nano

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

8. Основные команды планировщика SLURM

Команда	Описание
sinfo	Команда для просмотра состояния вычислительных узлов и очередей SLURM
squeue	Команда для просмотра списка запущенных задач
sbatch < name of the sub-mission script >	Команда для запуска задачи в режиме очереди: отправляет задачу для последующего ее выполнения
scancel <job_id>	Команда для удаления задачи с идентификатором <job_id> из очереди

В процессе выполнения работы можно пользоваться следующими ресурсами.

1. Пример использования процедуры xGEMM из библиотеки BLAS в программе на Си:
http://www.ccfit.nsu.ru/~kireev/lab4/lab4blas.htm
2. Linear algebra (numpy.linalg):
https://numpy.org/devdocs/reference/routines.linalg.html
3. Джанкарло Закконе. Книга рецептов параллельного программирования Python. 2-е изд.:
http://onreader.mdl.ru/PythonParallelProgrammingCookbook.2nd/content/index.html
4. Программированию с использованием MPI посвящен четвертый раздел книги:
http://onreader.mdl.ru/PythonParallelProgrammingCookbook.2nd/content/Ch04.html
5. Microsoft MPI:
https://learn.microsoft.com/ru-ru/message-passing-interface/microsoft-mpi
6. Самостоятельный поиск ресурсов в Интернете, посвященных тематике данной лабораторной работы.

ЗАДАНИЯ К ЛАБОРАТОРНОЙ РАБОТЕ № 5

<table>
<tr><th>Вариант</th><th>Задание</th></tr>
<tr><td>Вариант 1
Вариант 4
Вариант 7
Вариант 10
Вариант 13
Вариант 16
Вариант 19
Вариант 22
Вариант 25
Вариант 28</td><td>Исследовать ускорение и эффективность работы параллельной программы вычисления двойного интеграла методом Монте-Карло. Использовать исходные тексты программ своего варианта из Лабораторной работы № 3 (example7). Количество испытаний не ниже totalTosses 10E9.
Примечание: Диапазона обычного (знакового – signed) long уже не хватает. Поэтому для проведения такого количества испытаний в исходном тексте программы на C необходимо произвести следующие изменения: изменить тип некоторых переменных с типа long на unsigned long:
unsigned long procTosses, totalPntIn, procPntIn;...
unsigned long const totalTosses = 1e10;..
for(unsigned long i = 0; i < procTosses; i++)...
printf("Result: %.16f Proc: %d totalTosses: %lu Runtime: %f \n", integralEstimate, size, totalTosses, elapsed);

Запуск программы осуществить на следующем количестве процессов:
1, 8, 16, 32, 48</td></tr>
<tr><td>Вариант 2
Вариант 5
Вариант 8
Вариант 11
Вариант 14
Вариант 17
Вариант 20
Вариант 23
Вариант 26
Вариант 29</td><td>Исследовать ускорение и эффективность работы параллельной программы параллельного умножения матриц с использованием трех вложенных циклов. Использовать исходные тексты программ своего варианта из Лабораторной работы № 4 (example10). Размерность матриц не ниже 2000 (MSIZE=2000).
Запуск программы осуществить на следующем количестве процессов:
1, 8, 16, 20, 40</td></tr>
<tr><td>Вариант 3
Вариант 6
Вариант 9
Вариант 12
Вариант 15
Вариант 18
Вариант 21
Вариант 24
Вариант 27
Вариант 30</td><td>Исследовать ускорение и эффективность работы параллельной программы параллельного умножения матриц с использованием возможностей специализированных библиотек (numpy для Python и OpenBLAS для C). Использовать исходные тексты программ своего варианта из Лабораторной работы № 4 (example14). Размерность матриц не ниже 4000 (MSIZE=4000).
Запуск программы осуществить на следующем количестве процессов:
1, 8, 16, 20, 40</td></tr>
</table>

ЗАКЛЮЧЕНИЕ

Учебное пособие знакомит учащихся с основами распределенных вычислений и применением стандарта MPI (Message Passing Interface) для написания эффективных программ на языках C и Python. В пособии приводятся примеры параллельных MPI-программ, а также варианты индивидуальных заданий для выполнения лабораторных работ. Представленные примеры демонстрируют, как с помощью MPI создавать масштабируемые приложения, которые можно легко адаптировать под различные распределенные вычислительные системы, начиная с небольших кластеров и заканчивая крупными суперкомпьютерами.

Пособие предназначено для студентов направления подготовки 09.03.01 «Информатика и вычислительная техника», изучающих дисциплину «Разработка информационного обеспечения», для студентов направления подготовки 09.03.03 «Прикладная информатика», изучающих дисциплину «Распределенные и параллельные вычисления», а также для студентов направления подготовки 09.04.01 «Информатика и вычислительная техника», изучающих дисциплину «Введение в большие данные и анализ информации» очной и заочной форм обучения.

СПИСОК ИСТОЧНИКОВ

1. https://www.mpi-forum.org/ – Message Passing Interface Forum
2. https://www.mpich.org/ – MPICH: High-Performance Portable MPI
3. https://www.open-mpi.org/ – Open MPI: Open Source High Performance Computing
4. https://learn.microsoft.com/en-us/message-passing-interface/microsoft-mpi – MS-MPI – Microsoft implementation of the Message Passing Interface standard
5. https://www.intel.com/content/www/us/en/developer/tools/oneapi/mpi-library.html – Intel MPI Library. Deliver flexible, efficient, and scalable cluster messaging.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3
1. ВЫЧИСЛИТЕЛЬНЫЕ СИСТЕМЫ И ПАРАЛЛЕЛЬНЫЕ ПРОГРАММЫ 4
   1.1. Классификация по количеству потоков команд и данных 4
   1.2. Классификация по принципу организации памяти 5
   1.3. Критерии оценки работы параллельных программ 7
2. ОСНОВНЫЕ ПОНЯТИЯ MPI 9
   2.1. Синтаксис базовых функций MPI 9
   2.2. Некоторые функции коллективного взаимодействия процессов 11
   2.3. Работа со временем 12
3. ПОДГОТОВКА К РАБОТЕ 13
   3.1. Скачивание необходимых программных средств, необходимых для разработки параллельных программ с использованием MPI 13
   3.2. Запуск параллельных программ 14
4. ЛАБОРАТОРНЫЙ ПРАКТИКУМ 15
   4.1. Лабораторная работа № 1. «Параллельная программа Hello World!» 15
   4.2. Лабораторная работа № 2. «Параллельное вычисление значения определенного интеграла методом прямоугольников» 19
   4.3. Лабораторная работа № 3. «Параллельное вычисление двойного интеграла методом Монте-Карло» 32
   4.4. Лабораторная работа № 4. «Параллельное умножение матриц» 45
   4.5. Лабораторная работа № 5. «Компиляция и запуск приложений на вычислительном кластере с использованием SLURM» 68
ЗАКЛЮЧЕНИЕ 77
СПИСОК ИСТОЧНИКОВ 78

Учебное электронное издание

БОРИСЕНКО Андрей Борисович
КОРОБОВА Ирина Львовна
СВЕШНИКОВ Артём Юрьевич

РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ С ПРИМЕНЕНИЕМ MPI

Учебное пособие

Редактирование И. В. Калистратовой
Графический и мультимедийный дизайнер Т. Ю. Зотова
Обложка, упаковка, тиражирование И. В. Калистратовой

ISBN 978-5-8265-2871-6

Подписано к использованию 14.02.2025.
Тираж 50 шт. Заказ № 22

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106/5, пом. 2, к. 14
Телефон 8(4752)63-81-08.
E-mail: izdatelstvo@tstu.ru