

МИНОБРНАУКИ РОССИИ  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
**«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ  
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»**

Институт Компьютерных наук и технологий  
Высшая школа искусственного интеллекта  
Направление 02.03.01 Математика и Компьютерные науки

Заключительный отчёт по дисциплине «Научно-исследовательская  
работа»

**«Технологии параллельного  
программирования в операционных  
системах Linux»**

Автор: \_\_\_\_\_

Черепанов Михаил Дмитриевич

Научный руководитель: \_\_\_\_\_

Чуватов Михаил Владимирович

«\_\_\_\_» \_\_\_\_\_ 20\_\_ г.

# Реферат

Объектом исследования является ОС на базе ядра Линукс, а именно: Debian, и технологии параллельного программирования в этой среде.

Цель данного исследования заключается в исследовании операционной системы на базе ядра Linux, конкретно Debian, а также в изучении технологий параллельного программирования в этой среде.

Основной задачей проекта является перенос приложения из лабораторной работы по теории графов, где происходит кодирование и декодирование сообщений из определенного алфавита с использованием алгоритма Хаффмана. Программа должна использовать технологию MPI, собираться и запускаться на одном из узлов, и в процессе работы использовать вычислительные ресурсы трех узлов (включая запускающий узел и два других). Все узлы должны работать на операционной системе Linux с дистрибутивом Debian.

Для достижения этой цели были выполнены следующие этапы: установка и настройка виртуальной среды VirtualBox, установка операционной системы и базовая настройка сети, изучение основных возможностей взаимодействия с ОС Linux, подготовка разработочной среды и перенос приложения из лабораторной работы по теории графов.

В результате проекта был написан код, использующий технологию MPI. Исходное сообщение для кодирования разделяется на равные части, которые обрабатываются несколькими процессами, что ускоряет работу программы, поскольку кодирование данных происходит параллельно на каждом процессе.

Основные преимущества и характеристики проекта включают повышение скорости передачи данных (максимальная скорость передачи данных в сети) и пропускной способности сети (максимальное количество данных, передаваемых через сеть за определенный период времени) благодаря использованию параллельного программирования. Надежность сети обеспечивается благодаря подключению трех узлов в локальную сеть, а обмен данными происходит безопасно благодаря настройке SSH-сервера для каждого узла.

Итоговое приложение демонстрирует эффективность в увеличении скорости передачи данных и пропускной способности сети, а также обеспечивает безопасность и надежность сети. Кроме того, оно может быть масштабировано путем увеличения числа узлов.

# Содержание

Термины и определения	4
Перечень сокращений и обозначений	5
Введение	5
<b>1 Основная часть</b>	<b>7</b>
1.1 Установка и настройка среды виртуализации VirtualBox . . . . .	7
1.2 Настройка локальной сети и подключения по SSH между машинами	7
1.3 Установка и настройка ПО для работы с MPI на виртуальных ма- шинах.Hello World . . . . .	9
1.4 Функции MPI, использованные в работе . . . . .	11
1.5 Портирование программы из лабораторной работы . . . . .	14
1.6 Результаты работы . . . . .	15
1.7 Заключение . . . . .	16
<b>Приложение</b>	<b>18</b>

## Термины и определения

В настоящем отчете о НИР применяют следующие термины с соответствующими определениями:

- Виртуальная машина - это программная среда, которая имитирует физический компьютер, позволяя запускать на нем операционную систему или несколько операционных систем одновременно на одном физическом компьютере.
- Сеть - это совокупность компьютеров и других устройств, которые связаны между собой для обмена данными и ресурсами.
- VirtualBox - это программное обеспечение для виртуализации, которое позволяет создавать и управлять виртуальными машинами на компьютере. Он может быть установлен на операционную систему хоста (физический компьютер) и предоставляет пользователю возможность создавать виртуальные машины, устанавливать на них операционные системы и приложения, а также настраивать их в соответствии с требованиями пользователя.
- IP - это протокол сетевого уровня, который определяет формат и адресацию пакетов данных, передаваемых в компьютерных сетях. IP является основным протоколом Интернета и обеспечивает маршрутизацию данных между узлами сети.
- IP-адрес - это уникальный идентификатор компьютера или сетевого устройства в сети, используемый для маршрутизации пакетов данных.
- DNS - это система, которая преобразует доменные имена в IP-адреса и обратно.
- MPI - это стандартное программное обеспечение для обмена сообщениями между процессами, работающими на разных узлах вычислительного кластера или на разных компьютерах в сети. MPI обеспечивает возможность синхронизации и координации вычислений между процессами, а также обмена данными между ними.
- MP - это технология, которая позволяет выполнению нескольких процессов одновременно на одном или нескольких процессорах компьютера. В многопроцессорной системе каждый процессор работает независимо и может выполнять свою задачу без вмешательства других процессоров.
- Клиент- это компьютер или программа, которые получают доступ к ресурсам или услугам, предоставляемым другим компьютером или программой (сервером).
- Сервер - это компьютер или программа, которые предоставляют ресурсы или услуги другому компьютеру или программе (клиенту).

- Параллельное программирование- это метод программирования, при котором задачи выполняются одновременно на нескольких процессорах или ядрах процессора, чтобы ускорить выполнение программы.
- SSH сервер - это программа, которая позволяет удаленно управлять компьютером или сервером через защищенное соединение. SSH является протоколом безопасной удаленной работы, который обеспечивает шифрование данных и аутентификацию пользователей для защиты от несанкционированного доступа.
- SSH-ключ - это файл, содержащий криптографические ключи, используемые для аутентификации пользователя на удаленном сервере по протоколу SSH.

## Перечень сокращений и обозначений

В настоящем отчете о НИР применяют следующие сокращения и обозначения:

- SSH - Secure Shell
- NAT - Network Address Translation
- DHCP - Dynamic Host Configuration Protocol
- DNS - Domain Name System
- MPI - Message Passing Interface
- MP - Multiprocessing
- IP - Internet Protocol
- ОС - операционная система
- ПО - программное обеспечение

## Введение

Операционная система (ОС) является необходимой составляющей любой компьютерной системы. Одной из самых популярных ОС является ОС на базе ядра Линукс. Она отличается высокой стабильностью, безопасностью и открытым исходным кодом, что позволяет свободно модифицировать и дорабатывать ее под нужды пользователя. В данной работе использовался дистрибутив CentOS steam. В рамках НИР была поставлена задача ознакомиться с технологиями параллельного программирования в среде ОС на базе ядра Линукс, включая изучение основных принципов и методов параллельного программирования, а также особенностей и

преимуществ использования ОС на базе ядра Линукс для выполнения параллельных задач. Выполнение работы можно разбить на следующие подзадачи:

- Установка и настройка VirtualBox, создание трех виртуальных машин.
- Настройка локальной сети и подключения по SSH между машинами.
- Установка и настройка ПО для работы с MPI на виртуальных машинах.
- Запуск программы «Hello World» на каждом из процессов.
- Портирование программы из лабораторной работы из теории графов.
- Запуск программы с использованием MPI на трех машинах.

# 1 Основная часть

## 1.1 Установка и настройка среды виртуализации VirtualBox

Для установки VirtualBox в ОС ubuntu достаточно написать в терминале команду

```
sudo apt install virtualbox virtualbox-ext-pack
```

установится актуальная версия VirtualBox.

Далее нужно создать три виртуальные машины с ОС Debian.

Важным пунктом в настройке VirtualBox является подключение гостевых дополнений (Guest Additions) для создания общего буфера обмена и возможности расширения окна машины на весь экран.

Для подключения Guest Additions нужно:

- Подключить контроллер VBoxGuestAdditions.iso
- Запустить внутри локальной сети скрипт VBoxLinuxAdditions.run.

На рис. 1 показано окно VirtualBox с тремя созданными машинами Debian.

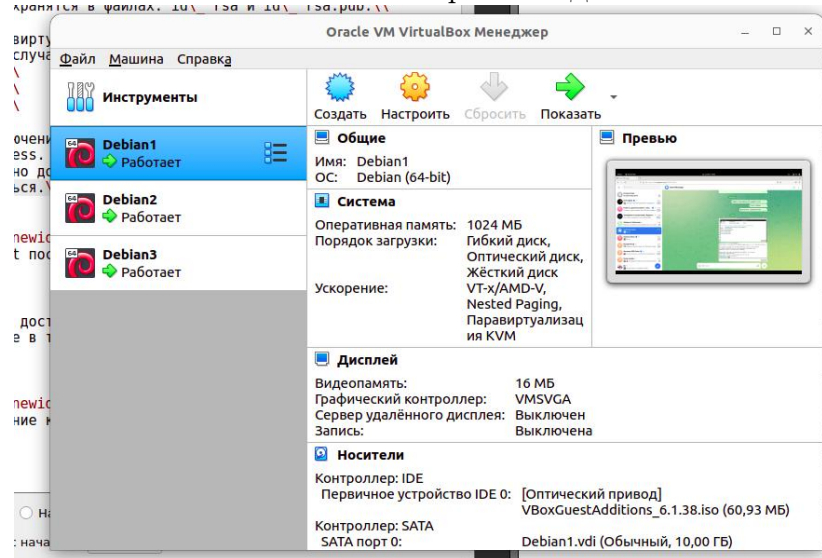


Рис. 1: VirtualBox.

## 1.2 Настройка локальной сети и подключения по SSH между машинами

Для настройки сети NAT необходимо: зайти во вкладку «Сети NAT», в поле IPv4 установить необходимый префикс и маску. В моем случае:

Адресс: 192.168.57.1, Маска:255.255.255.0.

Там же отключаем поддержку DHCP сервера и применяем получившиеся настройки.

Каждой машине в сети необходимо присвоить свой адрес, первые 3 цифры в котором обозначают адрес сети, а последняя цифра - адрес машины. Здесь нужно учесть, что первый адрес в сети занят встроенным коммутатором VirtualBox и его использовать нельзя.

Адреса моих машин: 172.16.56.10 ,172.16.56.20 ,172.16.56.30

#### Настройка подключения по SSH:

Генерируем ssh-ключи командой: `ssh-keygen -t rsa`.

Будут сгенерированы два ключа: приватный и публичный, которые сохранятся в файлах: `id_rsa` и `id_rsa.pub`.

Отправляем публичный ключ каждой виртуальной машине в сети командой: `ssh-copy-id username@host_address`. В моем случае это команды:

`copy-id mikhael@172.16.56.10`

`copy-id mikhael@172.16.56.20`

`copy-id mikhael@172.16.56.30`

После этого станет доступно подключение по SSH к любой машине в локальной сети с помощью команды: `ssh username@address`. Для удобства в файл `etc/hosts` можно добавить адреса машин в локальной сети и имена по которым к ним можно будет обращаться.



Рис. 2: `hosts.txt` после изменения.

На рис. 3 показан доступ из первой машины ко второй по SSH. После подключения все команды написанные в терминале будут исполняться на машине к которой осуществляется подключение.



```

mikhail@host1:~$ ssh mikhail@host2
Linux host2 6.1.0-9-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.27-1 (2023-05-08) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Jul 11 19:33:46 2023 from 172.16.56.10
mikhail@host2:~$ touch file1.txt
mikhail@host2:~$ █

```

Рис. 3: Подключение ко второй машине из первой.

### 1.3 Установка и настройка ПО для работы с MPI на виртуальных машинах. Hello World

Для работы с MPI необходимо установить компилятор для c++, в моем случае gcc, компилятор для работы с mpi, в моем случае mpic++, вместе с которыми установятся все необходимые библиотеки.

Установка mpic++: `sudo apt install mpic++`

Установка g++: `brew install gcc`

После установки запустим простейшую программу типа «Hello World» на виртуальных машинах параллельно на трех процессах. Код программы:

```

#include <iostream>
#include "mpi.h"
int main(int argc, char **argv) {
    int AMOUNT_FLOW, RANK_NUMBER;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &AMOUNT_FLOW);
    MPI_Comm_rank(MPI_COMM_WORLD, &RANK_NUMBER);
    std::cout << "Hello, World! From " << RANK_NUMBER << " process" << std::endl;
    MPI_Finalize();
    return 0;
}

```

---

Компиляция программы осуществляется с помощью команды: `mpic++ -o HelloWorld HelloWorld.cpp`

Запуск: `mpiexec -np 3 -host 172.16.56.10,172.16.56.20,172.16.56.30 ./HelloWorld`

На рис. 4 показан вывод программы.

```
Hello, World! From 2process  
Hello, World! From 1process  
Hello, World! From 0process  
mikhail@host1:~/Документы$
```

Рис. 4: Hello World.

## 1.4 Функции MPI, использованные в работе

Начнем с HelloWorld. Напомню код:

```
#include <iostream>
#include "mpi.h"
int main(int argc, char **argv) {
    int AMOUNT_FLOW, RANK_NUMBER;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &AMOUNT_FLOW);
    MPI_Comm_rank(MPI_COMM_WORLD, &RANK_NUMBER);
    std::cout << "Hello, World! From " << RANK_NUMBER << " process" << std::endl;
    MPI_Finalize();
    return 0;
}
```

---

В данном фрагменте используются следующие функции MPI:

*MPI\_Init*: Инициализирует MPI-окружение и запускает его перед использованием других функций MPI. Принимает указатели на аргументы командной строки *argc* и *argv* и инициализирует их значения.

*MPI\_Comm\_size*: Определяет количество процессов (потоков) в коммунитаторе. В данном случае, функция определяет количество процессов и сохраняет его значение в переменную *AMOUNT\_FLOW*.

*MPI\_Comm\_rank*: Определяет ранг (идентификатор) процесса в коммунитаторе. В данном случае, функция определяет ранг процесса и сохраняет его значение в переменную *RANK\_NUMBER*.

*MPI\_Finalize*: Завершает работу с MPI-окружением и освобождает все ресурсы. Вызов этой функции должен быть выполнен после завершения работы с другими функциями MPI.

Эти функции позволяют программе инициализировать MPI, определить количество процессов и ранг каждого процесса в коммунитаторе *MPI\_COMM\_WORLD*, а затем выводить сообщение "Hello, World! From" с номером процесса. В конце программы вызывается *MPI\_Finalize* для завершения работы с MPI.

Кроме того в работе используются функции:

- *MPI\_Send* используется для отправки сообщения из одного процесса в другой.

Параметры функции *MPI\_Send*:

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

buf: указатель на буфер, содержащий данные, которые будут отправлены.

count: количество элементов в буфере.

datatype: тип данных отправляемых элементов (например, *MPI\_INT*, *MPI\_FLOAT* и т.д.).

dest: идентификатор процесса-получателя (ранг процесса).

tag: числовая метка сообщения, используется для идентификации сообщения.

comm: коммуникатор, определяющий группу процессов, между которыми будет выполняться обмен сообщениями.

- *MPI\_Recv* используется для приема сообщения от другого процесса.

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

Параметры у функции такие же как у предыдущей, только вместо ранга получателя мы пишем ранг источника, и добавляется статус переданного сообщения. Если нет надобности использовать статус, можно передать *MPI\_Status\_Ignore*.

- *MPI\_Allgather* используется для сбора данных со всех процессов в коммуникаторе и распределения полученных данных обратно всем процессам. Это коллективная операция, которая гарантирует, что каждый процесс получит данные от всех остальных процессов.

Синтаксис функции *MPI\_Allgather* выглядит следующим образом:

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

Параметры функции:

sendbuf: указатель на буфер, содержащий данные, которые будут отправлены из каждого процесса.

sendcount: количество элементов в отправляемом буфере.

sendtype: тип данных отправляемых элементов.

recvbuf: указатель на буфер, в который будут сохранены принятые данные со всех процессов.

recvcount: количество элементов, которые каждый процесс ожидает получить от каждого другого процесса.

recvtype: тип данных принимаемых элементов (должен соответствовать типу данных, использованному при отправке данных).

comm: коммуникатор, определяющий группу процессов, между которыми выполняется обмен данными.

- *MPI\_Gather* используется для сбора данных со всех процессов в коммуникаторе.

каторе и их сбора на определенном процессе, который называется root процессом. Другие процессы отправляют свои данные root процессу, который собирает все данные в один буфер.

Синтаксис функции *MPI\_Allgather* выглядит следующим образом:

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

Параметры функции:

endbuf: указатель на буфер, содержащий данные, которые будут отправлены из каждого процесса.

sendcount: количество элементов в отправляемом буфере.

sendtype: тип данных отправляемых элементов.

recvbuf: указатель на буфер, в который будут сохранены принятые данные со всех процессов.

recvcount: количество элементов, которые каждый процесс ожидает получить от каждого другого процесса.

recvtype: тип данных принимаемых элементов (должен соответствовать типу данных, использованному при отправке данных).

root: идентификатор root процесса (ранг процесса), который будет собирать данные.

comm: коммуникатор, определяющий группу процессов, между которыми выполняется обмен данными.

## 1.5 Портирование программы из лабораторной работы

Для запуска лабораторной работы на нескольких процессах были распараллелены следующие части:

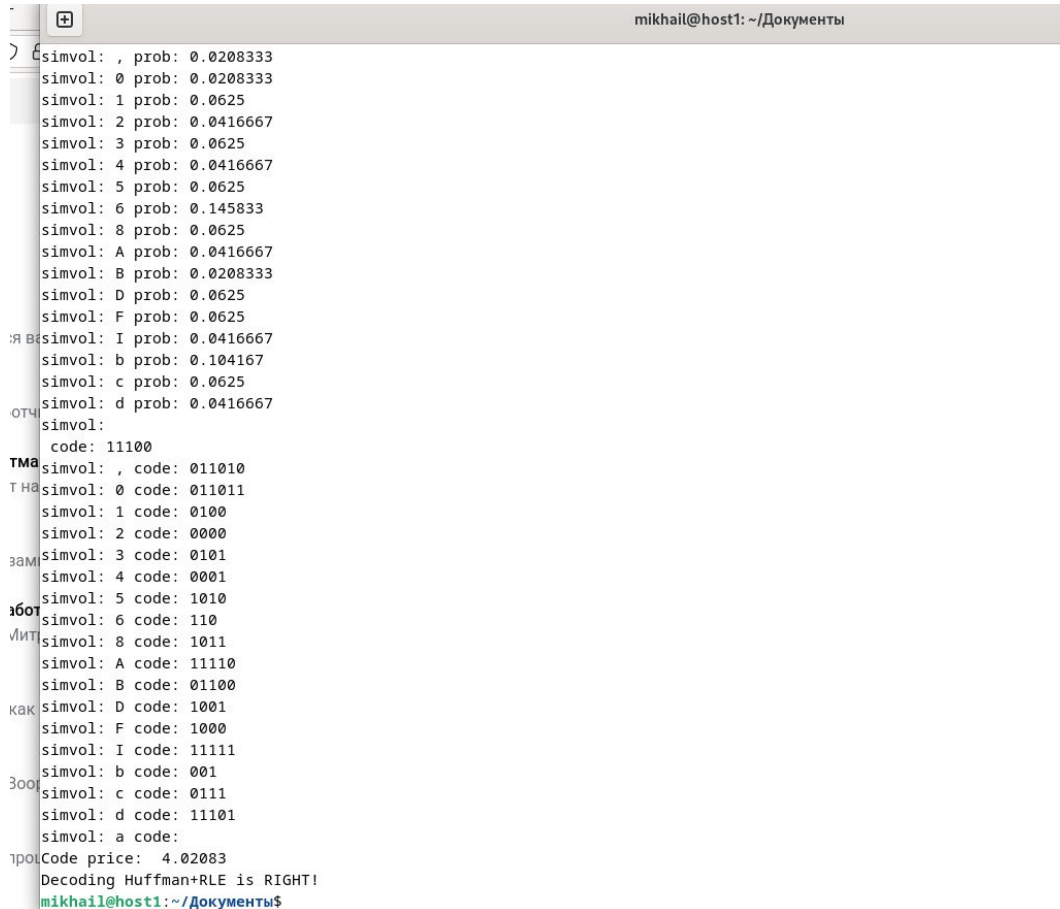
- Генерация строки заданной длины, состоящей из заданных символов. Каждый процесс генерирует массив случайных символов одинаковой длины, и затем, отправляет ее главному процессу с помощью функции *MPI\_Gather*. Главный процесс записывает строку в файл.
- Чтение строки из файла и получение «Таблицы» с вероятностью появления символа в строке. Главный процесс читает строку из файла, делит ее на равные части и отправляет каждому из процессов ее часть с помощью функции *MPI\_Send*. Каждый из процессов считает количество каждого из символов в строке, создает вектор структур с полями: символ, количество символов. Затем строка копируется в буфер для обмена с помощью функции *memcpy* и отправляется главному процессу функцией *MPI\_Allgather*. Основной процесс составляет общий вектор структур с вероятностью каждого символа.
- Генерация строки заданной длины, состоящей из заданных символов. Каждый процесс генерирует массив случайных символов одинаковой длины, и затем, отправляет ее главному процессу с помощью функции *MPI\_Gather*. Главный процесс записывает строку в файл.
- Главный процесс составляет дерево кодов символов и записывает их в свой вектор структур с полями: символ, код символа, копирует вектор в буфер для обмена, отправляет буффер всем остальным процессам и каждый из них кодирует свою часть строки. После этого закодированные части строки отправляются главному процессу и тот записывает их в файл.

Декодирование файла происходит в основном процессе.

При выполнении программы на нескольких виртуальных машинах стоит помнить, что файлы создаются только на той машине, откуда происходит запуск программы, и доступны только процессу с номером 0 (при запуске по одному процессу на каждой машине)

## 1.6 Результаты работы

Результатом работы является программа, переписанная лабораторная работа по теории графов, запущенная на трех виртуальных машинах.



```
mikhail@host1: ~/Документы
> C
simvol: , prob: 0.0208333
simvol: 0 prob: 0.0208333
simvol: 1 prob: 0.0625
simvol: 2 prob: 0.0416667
simvol: 3 prob: 0.0625
simvol: 4 prob: 0.0416667
simvol: 5 prob: 0.0625
simvol: 6 prob: 0.145833
simvol: 8 prob: 0.0625
simvol: A prob: 0.0416667
simvol: B prob: 0.0208333
simvol: D prob: 0.0625
simvol: F prob: 0.0625
simvol: I prob: 0.0416667
simvol: b prob: 0.104167
simvol: c prob: 0.0625
simvol: d prob: 0.0416667
simvol:
code: 11100
Tma simvol: , code: 011010
T Ha simvol: 0 code: 011011
simvol: 1 code: 0100
simvol: 2 code: 0000
зам simvol: 3 code: 0101
simvol: 4 code: 0001
simvol: 5 code: 1010
роб simvol: 6 code: 110
имт simvol: 8 code: 1011
simvol: A code: 11110
simvol: B code: 01100
как simvol: D code: 1001
simvol: F code: 1000
simvol: I code: 11111
Зод simvol: b code: 001
simvol: c code: 0111
simvol: d code: 11101
simvol: a code:
тро Code price: 4.02083
Decoding Huffman+RLE is RIGHT!
mikhail@host1: ~/Документы$
```

Рис. 5: Запуск программы на трех виртуальных машинах.

На 5 показан вывод программы. Мы можем увидеть код каждого символа, вероятность его появления, а так же сообщение о том, что кодирование и декодирование произошло верно.

## 1.7 Заключение

Во время выполнения работы было произведено знакомство с дистрибутивом Debian ОС линукс, основами параллельного программирования MPI, основами работы с VirtualBox. Были созданы три виртуальные машины, настроена сеть NAT между ними, настроено подключение по SSH.

Изучены основные функции MPI для отправки и приема данных. Было проведено портирование лабораторной работы по теории графов для запуска ее на нескольких процессах.

Результат НИР можно использовать как руководство по настройке среды виртуализации VirtualBox для задач распараллеливания вычислений.

Однако данная технология распараллеливания процессов на виртуальные машины на обычном пользовательском компьютере не всегда является экономически целесообразной, поскольку на обычном компьютере не всегда возможно выделить достаточно оперативной памяти каждой виртуальной машине, поэтому программа запущенная на нескольких процессах может работать медленнее, чем запущенная на одном.

Данная работа представляет из себя интерес в плане изучения технологий параллельного программирования и дальнейшего использования знаний на машинах с большими вычислительными мощностями.



## Список литературы

- [1] MPI Hands-On - C++  
<https://education.molssi.org/parallel-programming/04-distributed-examples.html>  
(дата обращения 07.07.2023)
- [2] Using MPI with C  
<https://curc.readthedocs.io/en/latest/programming/MPI-C.html> (дата обращения 06.07.2023)
- [3] habr Часть 1. MPI — Введение и первая программа  
<https://habr.com/ru/articles/548266/> (дата обращения 09.07.2023)
- [4] habr Часть 2. MPI - Учимся следить за процессами.  
<https://habr.com/ru/post/548418/> (дата обращения 09.07.2023)
- [5] habr Часть 3. MPI - Как процессы общаются? Сообщения типа точка-точка.  
<https://habr.com/ru/post/549312/> (дата обращения 09.07.2023)

## Приложение

```
#include <cstring> // memcpy
#include <cstdlib>
#include <iostream>
#include <time.h>
#include <fstream>
#include <map>
#include <locale>
#include <sstream>
#include <vector>
#include <algorithm>
#include <list>
#include "Tree.h"
#include "mpi.h"
#define FillRandom
using namespace std;
int SIZE_STR = 50;
int main(int argc, char **argv) {
    // setlocale(NULL, "Russian");
    // :

    int AMOUNT_FLOW, RANK_NUMBER;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &AMOUNT_FLOW); //
    MPI_Comm_rank(MPI_COMM_WORLD, &RANK_NUMBER); //
    ifstream file_in_first;
    ofstream file_encoding_Huffman;
    SIZE_STR=(SIZE_STR/AMOUNT_FLOW)*AMOUNT_FLOW; /// illegal cheating
    if (RANK_NUMBER == 0) {
        srand(time(0));

        file_encoding_Huffman.open("encodingHuffman.txt", ios::out);
        if (!file_encoding_Huffman.is_open()) {
            cout << "file error";
            return 0;
        }

        file_in_first = ifstream("Original.txt");
        if (!file_in_first) {
            cerr << "File error." << endl;
            return 1;
        }
    }
}
```

```

#ifdef FillRandom

char *sending_buff=new char [SIZE_STR / AMOUNT_FLOW];
char *getting_buff;
string returned_random = GetRandStr(SIZE_STR / AMOUNT_FLOW);
//cout<<"STR f 1:"<<returned_random<<endl;

for(int i=0;i<returned_random.size();i++){
sending_buff[i]=returned_random[i];
// cout<<sending_buff[i];
} //cout<<endl;
if(RANK_NUMBER==0){
getting_buff =new char [SIZE_STR+2];
}
MPI_Gather(sending_buff, SIZE_STR / AMOUNT_FLOW, MPI_CHAR,
getting_buff, SIZE_STR / AMOUNT_FLOW, MPI_CHAR, 0, MPI_COMM_WORLD);

if(RANK_NUMBER==0){
fstream file_original;
file_original.open("Original.txt", ios::out); //app -
if (!file_original.is_open()) {
cout << "file error";
return 0;
}
if(SIZE_STR % AMOUNT_FLOW==1){
string returned_random = GetRandStr(1);
getting_buff[SIZE_STR-1]=returned_random[0];
}
if(SIZE_STR % AMOUNT_FLOW==2){
string returned_random = GetRandStr(2);
getting_buff[SIZE_STR-2]=returned_random[0];
getting_buff[SIZE_STR-1]=returned_random[1];
}
for(int i=0;i<SIZE_STR;i++)
file_original<<getting_buff[i];
file_original.close();
// cout<<" ";
delete [] getting_buff;
}
delete [] sending_buff;
MPI_Barrier(MPI_COMM_WORLD);

#endif

```

```

////////////////////////////////////

```

```

char send0[] = "aaa", send1[] = "bbb", send2[] = "ccc";
int size_all_str;
string str_from_file((std::istreambuf_iterator<char>(file_in_first)),
std::istreambuf_iterator<char>());
if (RANK_NUMBER == 0) {
file_in_first = ifstream("Original.txt");
int size_0_1_str;
int size_2_str;
char** str_for_each_proc=new char*[AMOUNT_FLOW];
for(int i=0;i<AMOUNT_FLOW;i++){
str_for_each_proc[i]=new char[SIZE_STR / AMOUNT_FLOW + 5];
}
char *str_for_0proc = new char[str_from_file.size() / AMOUNT_FLOW + 5];
char *str_for_1proc = new char[str_from_file.size() / AMOUNT_FLOW + 5];
char *str_for_2proc = new char[str_from_file.size() / AMOUNT_FLOW + 5];
for (int i = 0; i < str_from_file.size() / AMOUNT_FLOW; i++) {
str_for_0proc[i] = str_from_file[i];
str_for_1proc[i] = str_from_file[i + str_from_file.size() / AMOUNT_FLOW];
str_for_2proc[i] = str_from_file[i + 2
* (str_from_file.size() / AMOUNT_FLOW)];
for(int j=0;j<AMOUNT_FLOW;j++){
str_for_each_proc[j][i]=str_from_file
[i+(str_from_file.size() / AMOUNT_FLOW)*j];
}
}
str_for_0proc[str_from_file.size() / AMOUNT_FLOW] = '\0';
str_for_1proc[str_from_file.size() / AMOUNT_FLOW] = '\0';
for(int j=0;j<AMOUNT_FLOW;j++){
str_for_each_proc[j][str_from_file.size() / AMOUNT_FLOW]='\0';
}

for(int i=0;i<AMOUNT_FLOW;i++){
MPI_Send(str_for_each_proc[i], str_from_file.size(
) / AMOUNT_FLOW + 1, MPI_CHAR, i, 0, MPI_COMM_WORLD);
}
size_0_1_str=str_from_file.size() / AMOUNT_FLOW;
// cout<<str_from_file.size()<<endl;
/* MPI_Send(&size_0_1_str, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
MPI_Send(&size_0_1_str, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&size_2_str, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);*/
size_all_str = str_from_file.size();
delete [] str_for_0proc;
delete [] str_for_1proc;
delete [] str_for_2proc;
}

```

```

int amount_simvols_in_str_each_proc=size_all_str / AMOUNT_FLOW+1;
char *str_each_proc = new char[size_all_str / AMOUNT_FLOW + size_all_str*2
MPI_Recv(str_each_proc, size_all_str /
AMOUNT_FLOW + 2 * SIZE_STR, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE); //

MPI_Barrier(MPI_COMM_WORLD);
////////////////////////////////////

map<char, double> probability;
int i = 0;
map<char, int> sim_and_amount;
while (str_each_proc[i] != '\0') {
sim_and_amount[str_each_proc[i]]++;
i++;
}
vector<SimvolAndAmount> vec_struct;
map<char, int>::iterator it = sim_and_amount.begin();
//cout << "rank: " << RANK_NUMBER;
for (; it != sim_and_amount.end(); it++) {
vec_struct.push_back(SimvolAndAmount(it->first, it->second));
// vec_struct.back().Print();
}
int buffer_size = vec_struct.size() * sizeof(SimvolAndAmount);
char *buffer_each = new char[buffer_size];
char *buffer_general;
if (RANK_NUMBER == 0) {
buffer_general = new char[100 * sizeof(SimvolAndAmount)];// <100
}

memcpy(buffer_each, vec_struct.data(), buffer_size);
vector<int> displs(AMOUNT_FLOW);
vector<int> sent_counts(AMOUNT_FLOW);
cout<<"buffer_size=" <<buffer_size<<endl;
MPI_Allgather(&buffer_size, 1, MPI_INT,
sent_counts.data(), 1, MPI_INT,
MPI_COMM_WORLD);
displs[0] = 0;
/* displs[1] = sent_counts[0];
displs[2] = (sent_counts[0] + sent_counts[1]);*/
for(int i=1;i<AMOUNT_FLOW;i++){
displs[i]=0;
for(int j=i;j>0;j--){
displs[i]+= sent_counts[j-1];
}
}

```

```

}
MPI_Gatherv(buffer_each, buffer_size, MPI_CHAR,
buffer_general, sent_counts.data(),
displs.data(), MPI_CHAR, 0, MPI_COMM_WORLD);
vector<int> sum_size(AMOUNT_FLOW);
MPI_Gather(&buffer_size, 1, MPI_INT,
sum_size.data(), 1, MPI_INT, 0, MPI_COMM_WORLD);
int help_me;
if (RANK_NUMBER == 0) {

// cout<<"after sending:"<<endl;
int general_size = 0;
for(int i=0;i<AMOUNT_FLOW;i++){
general_size+=sum_size[i];
}
cout<<" Gen size= "<<general_size<<endl;
vector<SimvolAndAmount> general_vec_struct
(general_size / sizeof(SimvolAndAmount));
cout<<general_vec_struct.size();
memcpy(general_vec_struct.data(), buffer_general, general_size);
cout<<"vec_struct_size= "<<general_vec_struct.size()<<endl;
/* general_vec_struct[0].Print();
general_vec_struct[1].Print();
general_vec_struct[1].Print();*/
help_me=0;

for (int i = 0; i < general_vec_struct.size(); i++) {
// general_vec_struct[i].Print();
probability[general_vec_struct[i].simvol]
+= general_vec_struct[i].amount;
}

map<char, double>::iterator it = probability.begin();
for (; it != probability.end(); it++) {
it->second /= SIZE_STR;
cout<<"simvol: "<<it->first<<" prob: "<<it->second<<endl;
}
}

delete buffer_each;
if (RANK_NUMBER==0)
delete buffer_general;
MPI_Barrier(MPI_COMM_WORLD);

```

```

// map<char , double> probability =
FillPropability(file_in_first , str_from_file);

Node *root;
map<string , vector<bool>> table_map;
int buffer_for_vec_struct_size;
vector<SimvolAndCodeSimvol> vec_struct_code;

if (RANK_NUMBER == 0) {
root = CreatTree(probability);
table_map = CreateTable(root); //////////

for_each(table_map.begin() ,
table_map.end() , [&vec_struct_code](
const pair <string , vector<bool>> &pair) { ///// map
SimvolAndCodeSimvol local;
local.simvol = pair.first[0];
for (int i = 0; i < pair.second.size(); i++) {
local.mass_code[i] = (int) pair.second[i];
}
local.mass_code[pair.second.size()] = 3;
vec_struct_code.push_back(local);
});
buffer_for_vec_struct_size=sizeof
(SimvolAndCodeSimvol)* vec_struct_code.size();
for(int i=0;i<AMOUNT_FLOW;i++)
MPI_Send(&buffer_for_vec_struct_size , 1,
MPI_INT, i , 0, MPI_COMM_WORLD);

}

MPI_Recv(&buffer_for_vec_struct_size , 1, MPI_INT, 0 , 0,
MPI_COMM_WORLD,MPI_STATUS_IGNORE);
char *buffer_for_vec_struct=new char
[buffer_for_vec_struct_size];

if (RANK_NUMBER==0) {
memcpy(buffer_for_vec_struct ,
vec_struct_code.data() , buffer_for_vec_struct_size);
for(int i=0;i<AMOUNT_FLOW;i++)
MPI_Send(buffer_for_vec_struct ,
buffer_for_vec_struct_size , MPI_CHAR,

```

```

    i, 0, MPI_COMM_WORLD);

}
MPI_Recv(buffer_for_vec_struct,
    buffer_for_vec_struct_size, MPI_INT, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
vec_struct_code.clear();
for(int i=0; i<(buffer_for_vec_struct_size/sizeof
(SimvolAndCodeSimvol))+1; i++){
vec_struct_code.push_back(SimvolAndCodeSimvol());
}

memcpy(vec_struct_code.data(),
    buffer_for_vec_struct, buffer_for_vec_struct_size);
////////// -
///
if(RANK_NUMBER==0)
for(int i=0; i<vec_struct_code.size(); i++){
vec_struct_code[i].Print();
}

//////////
vector<int> encoding_vec_every_proc;
for(int i=0; i<amount_simvols_in_str_each_proc; i++)
for(int j=0; j<vec_struct_code.size(); j++)
if(vec_struct_code[j].simvol==str_each_proc[i]) {
int k=0;
while (vec_struct_code[j].mass_code[k]!
=3&&k<vec_struct_code[j].amount_bit) {
encoding_vec_every_proc.push_back
(vec_struct_code[j].mass_code[k]);
k++;
}
}
int sending_size=encoding_vec_every_proc.size();

MPI_Allgather(&sending_size, 1, MPI_INT,
sent_counts.data(), 1, MPI_INT,
MPI_COMM_WORLD);
displs[0]=0;
for(int i=1; i<AMOUNT_FLOW; i++){
displs[i]=0;
for(int j=i; j>0; j--){

```



```

displs[i] += sent_counts[j-1];
}
}
int encoding_vec_general_size=0;
for(int i=0;i<AMOUNT_FLOW;i++)
encoding_vec_general_size+=sent_counts[i];
vector<int>encoding_vec_general(encoding_vec_general_size);
// cout<<"size: "<<sent_counts[0]+sent_counts[1]+sent_counts[2]<< " "<<am
MPI_Gatherv(encoding_vec_every_proc.data()
, sending_size, MPI_INT,
encoding_vec_general.data(), s
ent_counts.data(), displs.data(), MPI_INT, 0, MPI_COMM_WORLD);
if(RANK_NUMBER==0){
for(int i=0;i<encoding_vec_general.size();i++){
// cout<<encoding_vec_general[i];
file_encoding_Huffman<<encoding_vec_general[i];
}
// cout<<endl<<"Ok"<<endl;
}
MPI_Barrier(MPI_COMM_WORLD);// ,
// file_encoding_Huffman.close();
file_encoding_Huffman.close();
////////////////////////////////////

////////////////////////////////////
if(RANK_NUMBER==0) {
ifstream file_encodingHuffman_in("encodingHuffman.txt");
if (!file_encodingHuffman_in) {
cerr << "File error." << endl;
return 1;
}
string str_encoding;
if (!getline(file_encodingHuffman_in, str_encoding)) {
cout << "error reading" << endl;
return 0;
}
ofstream file_decodingHuffman;
file_decodingHuffman.open("decodingHuffman.txt", ios::out);
file_decodingHuffman << DecodingHuffman(str_encoding, root);
file_decodingHuffman.close();
file_encodingHuffman_in.close();

cout << "Code price: " << CodePrice(table_map, probability) << endl;
}

```

```

////////////////////////////////////// 2 RLE
if (RANK_NUMBER==0) {
    ofstream file_encoding_RLE;
    file_encoding_RLE.open("encodingRLE.txt", ios::out);
    file_in_first = ifstream("Original.txt");
    string str_from_file((std::istreambuf_iterator<char>(file_in_first)),
        std::istreambuf_iterator<char>());
    string strRLE = RLEGetStr(str_from_file);
    file_encoding_RLE << strRLE;
    file_encoding_RLE.close();
    ifstream file_encoding_RLE_from("encodingRLE.txt");
    if (!file_encoding_RLE_from) {
        cerr << "File error." << endl;
        return 1;
    }
    ofstream file_decoding_RLE;
    file_decoding_RLE.open("decodingRLE.txt", ios::out);

    stringstream all1;
    all1 << file_encoding_RLE_from.rdbuf();
    string all_str1;
    all_str1 = all1.str();

    file_decoding_RLE << DecodingRLE(all_str1);
    file_decoding_RLE.close();
    file_encoding_RLE_from.close();
}

////////////////////////////////////// 3 RLE+
////////////////////////////////////// RLE+ :
if (RANK_NUMBER==0) {
    ifstream file_encoding_RLE1("encodingRLE.txt");
    ofstream file_encoding_Huffman_RLE;
    file_encoding_Huffman_RLE.open("encodingHuffman_RLE.txt", ios::out);
    string str_from_file1;
    map<char, double> probability1 =
        FillPropability(file_encoding_RLE1, str_from_file1);
    Node *root1 = CreatTree(probability1);
    map<string, vector<bool>> table_map1 = CreateTable1(root1);

    stringstream str_stream1;
    for (int i = 0; i < str_from_file1.size(); i++) {
        for_each(table_map1[string(1, str_from_file1[i])).begin(),
            table_map1[string(1, str_from_file1[i])).end(),
            [&str_stream1](bool a) { str_stream1 << a; });
    }
}

```

```

}

string strRLE1 = str_stream1.str();
file_encoding_Huffman_RLE << strRLE1;
file_encoding_Huffman_RLE.close();

//////////////////////////////////// +RLE:
ifstream max_encoding_from("encodingHuffman_RLE.txt");
ofstream max_decoding_in("decodingHuffman_RLE.txt", ios::out);
stringstream all2;
all2 << max_encoding_from.rdbuf();
string all_str2;
all_str2 = all2.str();
string help_str;
help_str.reserve(100000);
help_str = DecodingRLE(all_str2);
string help_str1;
help_str1.reserve(100000);
help_str1 = DecodingHuffman(help_str, root1);
string help_str2;
help_str2.reserve(100000);
help_str2 = DecodingRLE(help_str1);
max_decoding_in << help_str2;
max_decoding_in.close();
max_encoding_from.close();
}
MPI_Barrier(MPI_COMM_WORLD);
if (RANK_NUMBER==0){
ifstream f1("decodingHuffman_RLE.txt");
ifstream f2("Original.txt");
string s1((std::istreambuf_iterator<char>(f1)),
std::istreambuf_iterator<char>());
string s2((std::istreambuf_iterator<char>(f2)),
std::istreambuf_iterator<char>());
bool flag=1;
for(int i=0;i<s2.size();i++){
if(s1[i]!=s2[i])
flag=0;
}
if(flag ==0) {
cout << "Decoding Huffman+RLE is WRONG!" << endl;
cout<<endl<<endl<<s1<<endl<<s2<<endl<<endl;
}
else

```

```

cout<<"Decoding Huffman+RLE is RIGHT!"<<endl;

}

delete [] str_each_proc;
delete [] buffer_for_vec_struct;

MPI_Barrier(MPI_COMM_WORLD);
cout<<"Process: "<<RANK_NUMBER<<" final ";
if (RANK_NUMBER==0){
file_encoding_Huffman.close();
// file_in_first.close();
}

MPI_Finalize();
}

```