

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
Федеральное государственное автономное образовательное учреждение  
высшего образования «Санкт-Петербургский политехнический  
университет Петра Великого»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление: 02.03.01 «Математика и компьютерные науки»

## Отчет по научно-исследовательской работе

Технологии параллельного программирования в операционных  
системах Linux

Студент,

группы 5130201/20001

\_\_\_\_\_ Якунин Д. Д.

Преподаватель

\_\_\_\_\_ Чуватов. М. В.

«\_\_\_\_\_» \_\_\_\_\_ 2024г.

Санкт-Петербург, 2024

# Содержание

Термины и определения	3
Введение	5
<b>1 Основная часть</b>	<b>6</b>
1.1 Установка и настройка среды виртуализации . . . . .	6
1.1.1 Настройка hosts . . . . .	7
1.1.2 Настройка SSH соединения . . . . .	7
1.1.3 Буфер обмена с внешней системой . . . . .	8
1.1.4 Настройка OpenMP . . . . .	9
1.1.5 Настройка OpenMPI . . . . .	9
1.2 Параллелизация исходной программы . . . . .	11
1.2.1 Реализация OpenMP . . . . .	11
1.2.2 Реализация OpenMPI . . . . .	14
<b>2 Результаты работы программы</b>	<b>20</b>
Список литературы	22

# Термины и определения

1. Виртуальная машина - это программная среда, которая имитирует физический компьютер, позволяя запускать на нем операционную систему.
2. IP - это протокол сетевого уровня, который определяет формат и адресацию пакетов данных, передаваемых в компьютерных сетях.
3. IP-адрес - это уникальный идентификатор компьютера или сетевого устройства в сети.
4. DNS - это система, которая преобразует доменные имена в IP-адреса и обратно.
5. OpenMP (Open Multi-Processing) - открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Даёт описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.
6. OpenMPI (Open Message Passing Interface) - это библиотека и стандарт для параллельных вычислений на распределенных системах. Она предоставляет набор функций и инструментов для обмена сообщениями и координации вычислительных задач между процессами, работающими на разных узлах в кластере или распределенной сети. OpenMPI позволяет эффективно использовать мощности множества компьютеров для параллельных вычислений.
7. NAT-сеть (Network Address Translation) - это технология, которая позволяет переводить IP-адреса между локальной сетью и внешней сетью.
8. ОС (Операционная система) - это программное обеспечение, которое управляет аппаратными и программными ресурсами компьютера или устройства. ОС обеспечивает интерфейс между пользователем и компьютерной системой, управляет процессами, файлами, памятью, устройствами ввода-вывода и другими ресурсами, а также обеспечивает выполнение программ и поддержку различных функций.
9. SSH (Secure Shell) - это протокол сетевой безопасности, который обеспечивает защищенное удаленное подключение и обмен данными между компьютерами. SSH используется для удаленного управления компьютерами, передачи файлов и выполнения команд на удаленных машинах. Протокол SSH обеспечивает шифрование данных и аутентификацию, обеспечивая безопасность при удаленном доступе к системам.
10. DHCP (Dynamic Host Configuration Protocol) - это протокол сетевой конфигурации, который автоматически назначает IP-адреса, субнет-маски, шлюзы и другие сетевые параметры компьютерам в сети. DHCP позволяет упростить процесс настройки сети, позволяя компьютерам автоматически получать необходимую сетевую конфигурацию от DHCP-сервера.
11. RSA (Rivest-Shamitar-Adelman) — алгоритм, который использует пару ключей: один для шифрования (общедоступный), а другой — для дешифрования (приватный)

12. Алгоритм Лемпеля — Зива — Уэлча (Lempel-Ziv-Welch, LZW) — это универсальный алгоритм сжатия данных без потерь, созданный Авраамом Лемпелем, Яковом Зивом и Терри Велчем.
13. Кодирование длин серий (англ. run-length encoding, RLE) или кодирование повторов — алгоритм сжатия данных, заменяющий повторяющиеся символы (серии) на один символ и число его повторов.

# Введение

В рамках научно-исследовательской работы была поставлена задача ознакомиться с технологиями параллельного программирования в среде ОС на базе ядра Линукс, включая изучение основных принципов и методов параллельного программирования. В данной работе и полученном варианте было необходимо использовать дистрибутив Debian и технологий OpenMP и OpenMPI.

Для выполнения задач была необходима среда виртуализации. Одной из самых популярных на сегодняшний день является VirtualBox. С её помощью были созданы виртуальные машины и смоделирована работа простейшего суперкомпьютера с несколькими узлами.

Выполнение работы можно разбить на следующие подзадачи:

1. Установка и настройка VirtualBox, создание виртуальных машин.
2. Настройка локальной сети и подключения по SSH между машинами.
3. Установка и настройка ПО для работы с OpenMP и OpenMPI на виртуальных машинах.
4. Адаптация ранее написанной программы на параллельную работу.
5. Запуск программы на виртуальных машинах.

# 1 Основная часть

## 1.1 Установка и настройка среды виртуализации

В качестве программы для виртуализации была выбрана Oracle VirtualBox. Скачать её можно с [официального сайта](#).

Сначала настроим сеть для нашей системы. Для этого в VirtualBox в меню инструментов выбираем «Сеть», переходим в «Сети NAT» и создаем новую сеть: в полученном варианте IPv4 префикс 10.20.63.192/26, выключаем DHCP.

Для создания виртуальной машины необходимо выполнить следующие шаги:

1. В меню VirtualBox нажать «Создать»
2. Выбрать имя виртуальной машины, куда она будет установлена и выбрать заранее скачанный образ ОС, которую необходимо будет установить. В полученном варианте это Debian (64 бит).
3. Выделить ресурсы: так как при выполнении ресурсов было достаточно, то на каждую машину было выделено 2048 МБ оперативной памяти, 2 ЦП и 10 ГБ на виртуальный жесткий диск.
4. После создания виртуальной машины заходим в её настройки и в меню «Сеть» выбираем ранее созданную сеть NAT.

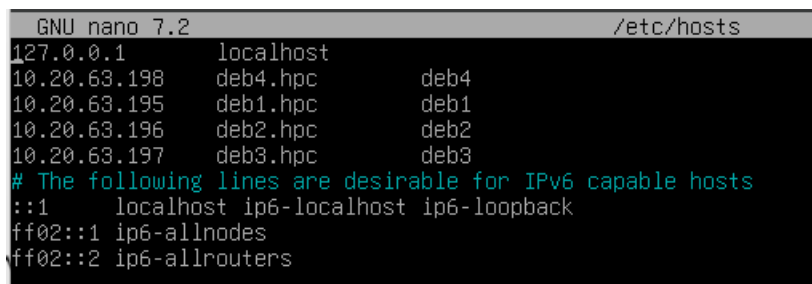
После этого будет необходимо запустить машину и установить саму ОС. Везде выбираем стандартные настройки, самостоятельно настраиваем лишь несколько вещей:

1. Так как ранее в настройках сети мы отключили DHCP, то сеть необходимо будет настроить вручную. Выбираем адрес машины: в полученном варианте сеть 10.20.63.192/26, поэтому, например, можно выбрать для первой машины адрес 10.20.63.195. Первые два адреса занимаются системой: 10.20.63.193 на шлюз, адрес 10.20.63.194 не подходит, так как потом будут проблемы с доступом извне с нашей основной ОС. У следующей настраиваемой машины адрес 10.20.63.196 и так далее.
2. Выбираем имя машины в сети и какой-либо домен, например, hpc.
3. Далее будет предложено ввести пароль суперпользователя, его можно не вводить, тогда все пользователи с помощью команды `sudo` смогут выполнять действия администратора. После этого создаем обычного пользователя и задаем пароль. Желательно везде выбрать одинаковое имя пользователя, так как в дальнейшем это упростит запуск программ.
4. Далее будет предложено выбрать ПО, оставляем только Стандартные утилиты и SSH-сервер.

После всего этого у нас будет виртуальная машина с установленной ОС и настроенной сетью.

### 1.1.1 Настройка hosts

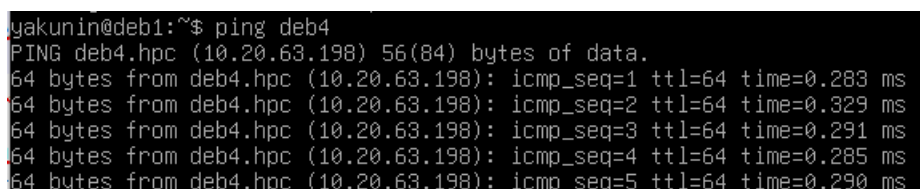
Стоит настроить разрешение имён узлов виртуальной сети, чтобы обращаться к ним по их именам вместо использования IP-адресов. Для этого были добавим соответствия имен и адресов в файле `/etc/hosts`, который позволяет разрешать имена узлов без использования DNS-сервера. После настройки, стало возможным обращаться к другим узлам сети по их именам, а не по IP-адресам. Открыть файл с конфигурацией можно командой **sudo nano /etc/hosts**.

A screenshot of a terminal window showing the `/etc/hosts` file being edited with `GNU nano 7.2`. The file contains entries for `localhost` and several nodes (`deb4`, `deb1`, `deb2`, `deb3`) with their corresponding IP addresses (`10.20.63.198` through `10.20.63.197`). It also includes IPv6 configuration lines for `localhost`, `ip6-localhost`, `ip6-loopback`, `ip6-allnodes`, and `ip6-allrouters`.

```
GNU nano 7.2 /etc/hosts
127.0.0.1 localhost
10.20.63.198 deb4.hpc deb4
10.20.63.195 deb1.hpc deb1
10.20.63.196 deb2.hpc deb2
10.20.63.197 deb3.hpc deb3
# The following lines are desirable for IPv6 capable hosts
::1 localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

Рис. 1. Настроенный файл `/etc/hosts`

Теперь проверить связь между узлами можно командой **ping host\_name**.

A screenshot of a terminal window showing the output of the `ping deb4` command. It displays five successful ping responses from `deb4.hpc (10.20.63.198)` with varying times.

```
yakunin@deb1:~$ ping deb4
PING deb4.hpc (10.20.63.198) 56(84) bytes of data.
64 bytes from deb4.hpc (10.20.63.198): icmp_seq=1 ttl=64 time=0.283 ms
64 bytes from deb4.hpc (10.20.63.198): icmp_seq=2 ttl=64 time=0.329 ms
64 bytes from deb4.hpc (10.20.63.198): icmp_seq=3 ttl=64 time=0.291 ms
64 bytes from deb4.hpc (10.20.63.198): icmp_seq=4 ttl=64 time=0.285 ms
64 bytes from deb4.hpc (10.20.63.198): icmp_seq=5 ttl=64 time=0.290 ms
```

Рис. 2. Проверка связи между узлами

### 1.1.2 Настройка SSH соединения

SSH (Secure Shell) обеспечивает безопасное удаленное соединение между двумя системами. С помощью этого криптографического протокола можно управлять машинами, копировать или перемещать файлы на удаленном сервере через зашифрованные каналы.

Существует два способа входа в удаленную систему через SSH - с использованием аутентификации по паролю или аутентификации с открытым ключом (вход SSH без пароля). Мы настроим вход без пароля.

Так как при установке ОС мы выбрали установку пакета SSH, то он уже есть на всех наших машинах. Проверить статус службы можно командой **sudo systemctl status sshd**, и если она не запущена, то запустим её командой **sudo systemctl start sshd**.

Для генерации ключевой пары на каждом узле необходимо выполнить: **ssh-keygen -t rsa**. По умолчанию ключевая пара будет сохранена в каталоге `/.ssh/`. Во время генерации ключевой пары можно задать пароль для закрытого ключа. В данной работе пароль для закрытого ключа не устанавливался. При генерации параметр `-t rsa` указывает тип ключа для создания, `rsa` означает, что будет создана пара ключей, использующая криптографический алгоритм RSA.

Далее необходимо разместить открытый ключ на удаленных узлах. Для этого введем команду **ssh-copy-id user@host\_name**. Если пользователь на всех узлах один, то его можно опустить и оставить просто **ssh-copy-id host\_name**.

Чтобы проверить, что все успешно, попробуем подключиться без пароля к какой-нибудь машине. Для этого с другой машины введем команду **ssh host\_name**. Если после этого мы подключились без пароля, то все настроено верно.

```
yakunin@deb1:~$ ssh deb3
Linux deb3 6.1.0-21-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.90-1 (2024-05-03) 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: Sat Jun 22 13:22:52 2024
yakunin@deb3:~$ _
```

Рис. 3. Успешное подключение по SSH

### 1.1.3 Буфер обмена с внешней системой

Чтобы была возможность отправлять файлы из нашей основной ОС на виртуальные машины, настроим проброс портов нашей NAT сети. В адресе гостя указываем тот, что указали при установке, его можно проверить на машине командой **ip -br a**. Порт гостя по умолчанию 22, так как мы нигде его не меняли, то его и пишем. Адрес хоста указываем 127.0.0.1, это стандартный локальный адрес. Порт хоста указываем больше, чем 4096, так как до него идут привелегированные хосты, используемые различными приложениями.

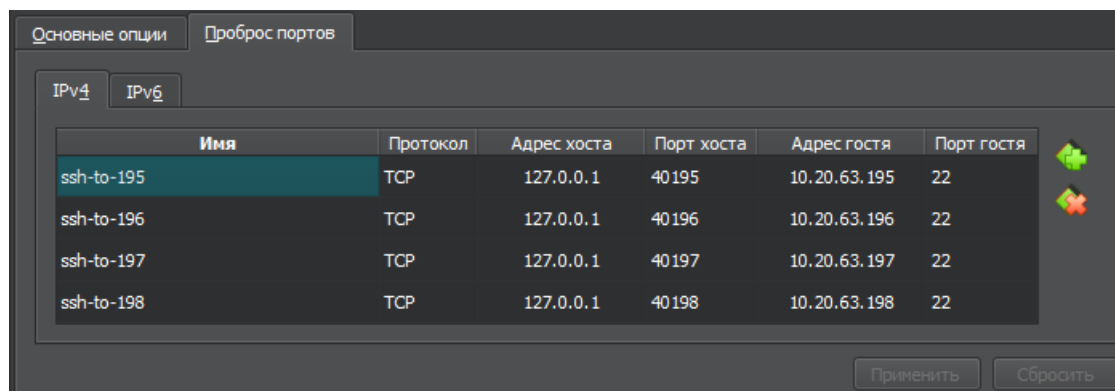


Рис. 4. Настроенный проброс портов

Для отправки файла с основной ОС на виртуальную, воспользуемся командой **scp -P host\_port source\_path user@localhost:destination\_path**.

```
PS C:\Users\dykun> scp -P 40195 -r C:\Users\dykun\OneDrive\Документы\coder-mpi yakunin@localhost:~
yakunin@localhost's password:
LZW_coder.cpp          100% 15KB   4.0MB/s   00:00
LZW_coder.h            100% 1376   27.2KB/s   00:00
main.cpp               100% 4123   2.5MB/s   00:00
RLE_coder.cpp          100% 10KB   6.4MB/s   00:00
RLE_coder.h            100% 974    616.8KB/s 00:00
PS C:\Users\dykun> |
```

Рис. 5. Пример отправки данных



### 1.1.4 Настройка OpenMP

Ключевые особенности OpenMP:

- Простота использования: OpenMP использует директивы препроцессора («прагмы»), которые вставляются в исходный код программ на C/C++ или Fortran. Эти директивы дают указания компилятору о том, какие части кода должны выполняться параллельно.
- Поддержка различных структур управления параллелизмом: OpenMP поддерживает параллелизм на уровне циклов (loop-level parallelism), секций (sections), и задач (tasks).
- Контроль над данными: OpenMP предлагает различные механизмы для управления доступом к данным в многопоточной среде, такие как private, shared, reduction и другие.
- Взаимодействие между потоками: OpenMP предоставляет функции и механизмы синхронизации, такие как барьеры, критические секции, замки и атомарные операции, что позволяет контролировать доступ к общим переменным и предотвращать условия гонки.

Для установки необходимых в работе пакетов необходимо воспользоваться встроенным пакетным менеджером apt. Установить пакеты, необходимые для разработки с поддержкой OpenMP можно командой **sudo apt install libomp-dev**.

Установить количество выполняемых одновременно потоков можно с помощью команды **export OMP\_NUM\_THREADS=N**, где N - количество потоков.

Для компиляции программы используется команда **g++ -fopenmp main.cpp name**, для запуска используется **./name**.

### 1.1.5 Настройка OpenMPI

Ключевые особенности OpenMPI:

- Масштабируемость: Open MPI предназначен для эффективной работы как на небольших кластерах, так и на системах с тысячами процессоров, поддерживая программы, масштабируемые до большого количества узлов.
- Высокопроизводительность: Библиотека настроена на использование оптимальных протоколов передачи данных для различных сетевых интерфейсов, что обеспечивает высокую производительность в параллельных вычислениях.
- Функциональные возможности MPI: Open MPI поддерживает все основные функции стандарта MPI, включая передачу сообщений, коллективные операции, группы и коммутаторы, синхронизацию процессов, и др.
- Динамичность: Open MPI поддерживает динамическое создание и управление процессами во время выполнения приложения, что позволяет оптимизировать и адаптировать поведение приложения под текущие условия выполнения.

Установить пакеты, необходимые для работы с OpenMPI можно командой **sudo apt install libopenmpi3 libopenmpi-dev**.

Для компиляции программы используется команда **mpic++ main.cpp name**, для запуска используется, например, **mpirun -host deb1:3,deb2 name** - эта команда запустит программу *name* с 4 процессами на узлах deb1 и deb2. Перед запуском программу нужно скомпилировать на всех узлах по одному пути.

## 1.2 Параллелизация исходной программы

В исходной программе была задача сгенерировать файл длиной в 10000 символов, а потом закодировать и декодировать его при помощи RLE и LZW и посчитать коэффициенты сжатия. Для параллелизации было решено сделать следующее:

- Генерацию файла разделить на части и потом объединить все части в один файл.
- При кодировании разбивать исходный файл на части и каждую часть кодировать обособленно от остальных. Полученные коды соединять и записывать в файл. Первой же строкой в этом файле будут размеры каждой закодированной части.
- При декодировании разбивать файл с кодом по метаданным из первой строки и декодировать каждую часть кода независимо от остальных. Декодированные данные объединяются и записываются в файл.

### 1.2.1 Реализация OpenMP

В функции генерации файла сначала создается вектор строк размером равный количеству потоков. Потом каждый поток генерирует строку длиной 10000/(количество потоков). Понятно, что редко будет возможно поделить поровну, поэтому остаток от деления добавляется к первому потоку и в итоге получится ровно 10000 символов. Каждый поток генерирует свою часть, после чего они все по очереди записываются в файл.

```
1 void generateFile(const std::vector<std::string>& init_dict, const std::string&
   ↪ file_name) {
2     std::ofstream file(file_name);
3     int alphabet_size = init_dict.size();
4     std::vector<std::string> local_buffers(omp_get_max_threads());
5
6     #pragma omp parallel
7     {
8         int thread_id = omp_get_thread_num();
9         std::string& buffer = local_buffers[thread_id];
10        int local_size = file_size / omp_get_num_threads();
11        if (thread_id == 0) {
12            local_size += file_size % omp_get_num_threads();
13        }
14        for (int i = 0; i < local_size; i++) {
15            buffer += init_dict[rand() % alphabet_size];
16        }
17
18    }
19
20    for (const std::string& buffer : local_buffers) {
21        file << buffer;
```

```

22     }
23
24     file.close();
25 }

```

Для разделения процесса кодирования была написана специальная функция LZW\_parallel. Она разбивает файл на количество потоков, после чего каждый поток кодирует свою часть независимо от остальных и кладет результат в свою ячейку вектора, чтобы они были в правильном порядке. Далее в первую строчку файла через пробел записываются размеры всех частей, после чего все части по очереди записываются во вторую строку.

```

1 void LZW_parallel(const std::vector<std::string>& init_dict, const std::string&
  ↪ fin_name, const std::string& fout_name) {
2     int threads_num = omp_get_max_threads();
3     std::vector<std::string> parts = split_file_into_parts(fin_name,
  ↪ threads_num);
4     std::vector<std::string> coded_parts(threads_num, "");
5
6     #pragma omp parallel
7     {
8         int tid = omp_get_thread_num();
9         //std::cout << tid;
10        // Работа каждого потока в своей части
11        coded_parts[tid] = LZW_encoder(init_dict, parts[tid]);
12    }
13
14    std::ofstream fout;
15    fout.open(fout_name);
16    for (const auto& coded_part : coded_parts)
17        fout << coded_part.size() << ' ';
18    fout << '\n';
19    for (const auto& coded_part : coded_parts)
20        fout << coded_part;
21 }

```

Для разделения процесса декодирования была написана специальная функция LZW\_parallel\_decode. Она считывает код с файла и разбивает его на те же части, что получились при кодировании благодаря первой строке с метаданными. Далее при помощи встроенных средств OpenMP декодирование разбивается по потокам и каждая декодированная часть кладется в свою часть вектора. После чего все части в правильном порядке записываются в файл.

```

1 void LZW_parallel_decode(const std::vector<std::string>& init_dict, const
  ↪ std::string& fin_name, const std::string& fout_name) {
2     std::vector<std::string> parts = split_file_decoder(fin_name);

```

```

3      std::vector<std::string> decoded_parts(parts.size(), "");
4
5      #pragma omp parallel
6      {
7          #pragma omp for
8          for (int i = 0; i < parts.size(); i++) {
9              decoded_parts[i] = LZW_decoder(init_dict, parts[i]);
10         }
11     }
12
13     std::ofstream fout;
14     fout.open(fout_name);
15     for (const auto& decoded_part : decoded_parts)
16         fout << decoded_part;
17 }

```

Для кодирования и декодирования RLE написаны аналогичные функции, кодирующие соответственно с помощью алгоритма RLE. Работают же они по точно такому же принципу.

Также в функции main есть проверка того, что декодирование прошло успешно и подсчет коэффициента сжатия файлов. Так как эти части кода не зависят друг от друга, то они были разделены на параллельные секции:

```

1      #pragma omp parallel
2      {
3          #pragma omp sections
4          {
5              // сравнение исходного файла и декодированных
6              #pragma omp section
7              {
8                  std::string tmp(std::string("\nLZW decoding was ") +
9                      ↪ ((compareFiles(file_name, "LZW_decoded.txt")) ? "correct"
10                      ↪ : "incorrect")
11                  + std::string(", thread ") +
12                      ↪ std::to_string(omp_get_thread_num()));
13                  std::cout << tmp;
14              }
15              #pragma omp section
16              {
17                  std::string tmp(std::string("\nRLE decoding was ") +
18                      ↪ ((compareFiles(file_name, "RLE_decoded.txt")) ? "correct"
19                      ↪ : "incorrect")
20                  + std::string(", thread ") +
21                      ↪ std::to_string(omp_get_thread_num()));
22                  std::cout << tmp;
23              }
24          }
25      }

```

```

18         // коэффициенты сжатия
19         #pragma omp section
20         {
21             std::string tmp(std::string("\n\nLZW compression ratio: ")
22             + std::to_string(float(file_size) /
23             ↪ (numberOfCharacters("LZW_coded.txt") / 8))
24             + std::string(", thread ") +
25             ↪ std::to_string(omp_get_thread_num())
26             + std::string("\nRLE compression ratio: ")
27             + std::to_string(float(file_size) /
28             ↪ (numberOfCharacters("RLE_coded.txt") / 8))
29             + std::string(", thread ") +
30             ↪ std::to_string(omp_get_thread_num()));
           std::cout << tmp;
        }
    }
}

```

## 1.2.2 Реализация OpenMPI

В реализации использовались функции и структуры MPI:

- MPI\_Init(&argc, &argv) - инициализация MPI.
- MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank) - функция для получения номера текущего процесса, значение будет положено в переменную rank.
- MPI\_Comm\_size(MPI\_COMM\_WORLD, &size) - аналогичная функция получения общего числа процессов.
- MPI\_Status - это структура, используемая для хранения информации о полученном сообщении. С её помощью можно получить идентификатор источника сообщения, тег сообщения и код ошибки.
- MPI\_Probe - используется для проверки, есть ли входящее сообщение, которое можно принять, не получая его. Она заполняет MPI\_Status информацией о входящем сообщении, которую затем можно использовать для подготовки к приему этого сообщения.
- MPI\_Get\_count - используется для получения количества элементов в полученном сообщении на основе информации, хранящейся в MPI\_Status.
- MPI\_Send - используется для отправки сообщения указанному получателю. Она принимает буфер, содержащий данные для отправки, а также идентификатор целевого процесса.
- MPI\_Recv - блокирует выполнение программы, пока не будет получено сообщение от указанного источника. Она заполняет буфер, указанный в аргументах, данными из полученного сообщения.

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

Для OpenMPI функция генерации файла работает по похожему образцу на OpenMP, с тем лишь отличием, что теперь все сгенерированные части файла собираются в нулевой процесс, который и записывает их вместе с метаданными в файл.

```

1 // генерация файла на основе алфавита
2 void generateFile(const std::vector<std::string>& init_dict, const std::string&
   ↪ file_name) {
3     int rank, size;
4     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5     MPI_Comm_size(MPI_COMM_WORLD, &size);
6
7     int alphabet_size = init_dict.size();
8     int local_size = file_size / size; // размер задачи на процесс
9     std::string local_buffer;
10
11     if (rank == 0) {
12         local_size += file_size % size;
13     }
14
15     // Генерация данных в каждом процессе
16     srand(time(NULL) + rank);
17     for (int i = 0; i < local_size; ++i) {
18         local_buffer += init_dict[rand() % alphabet_size];
19     }
20
21     // Сбор закодированных частей в процесс 0
22     if (rank == 0) {
23         std::vector<std::string> generated_parts(size);
24         generated_parts[0] = std::move(local_buffer);
25         for (int i = 1; i < size; i++) {
26             MPI_Status status;
27             MPI_Probe(i, 0, MPI_COMM_WORLD, &status);
28             int count;
29             MPI_Get_count(&status, MPI_CHAR, &count);
30             generated_parts[i].resize(count);
31             MPI_Recv(&generated_parts[i][0], count, MPI_CHAR, i, 0,
   ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
32         }
33
34         // Процесс 0 записывает собранные данные в файл
35         std::ofstream file(file_name);
36         for (const auto& gp : generated_parts) {
37             file << gp;

```

```

38         }
39         file.close();
40     }
41     else {
42         // Отправка сгенерированной части процессу 0
43         MPI_Send(local_buffer.data(), local_buffer.size(), MPI_CHAR, 0, 0,
44                 ↪ MPI_COMM_WORLD);
45     }
46 }

```

В функции LZW\_parallel нулевой процесс разделяет исходный файл на части и рассылает каждому процессу свою часть для кодирования. Далее все процессы выполняют кодирование и отправляют код нулевому процессу. Нулевой процесс же выполняет кодирование, собирает все части в правильном порядке и записывает в файл, предварительно записав метаданные по аналогичному OpenMP принципу.

```

1 void LZW_parallel(const std::vector<std::string>& init_dict, const std::string&
  ↪ fin_name, const std::string& fout_name) {
2     int rank, size;
3     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4     MPI_Comm_size(MPI_COMM_WORLD, &size);
5
6     std::vector<std::string> parts;
7     std::string part;
8     std::string coded_part;
9
10    if (rank == 0) {
11        // Разделить файл только в процессе с рангом 0
12        parts = split_file_into_parts(fin_name, size);
13
14        // Рассылка каждой части файла соответствующему процессу
15        for (int i = 1; i < size; i++) {
16            MPI_Send(parts[i].data(), parts[i].size(), MPI_CHAR, i, 0,
17                    ↪ MPI_COMM_WORLD);
18        }
19        part = parts[0]; // Часть для процесса 0
20    }
21    else {
22        MPI_Status status;
23        MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
24        int count;
25        MPI_Get_count(&status, MPI_CHAR, &count);
26        part.resize(count);
27        MPI_Recv(&part[0], count, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
28                ↪ MPI_STATUS_IGNORE);

```



```

27     }
28
29     // Непосредственно кодировка
30     coded_part = LZW_encoder(init_dict, part);
31
32     // Сбор закодированных частей обратно в процесс 0
33     if (rank == 0) {
34         std::vector<std::string> coded_parts(size);
35         coded_parts[0] = std::move(coded_part);
36         for (int i = 1; i < size; i++) {
37             MPI_Status status;
38             MPI_Probe(i, 0, MPI_COMM_WORLD, &status);
39             int count;
40             MPI_Get_count(&status, MPI_CHAR, &count);
41             coded_parts[i].resize(count);
42             MPI_Recv(&coded_parts[i][0], count, MPI_CHAR, i, 0,
43                 ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
44
45             }
46
47             // Сохранение результатов
48             std::ofstream fout(fout_name);
49             for (const auto& cp : coded_parts) {
50                 fout << cp.size() << ' ';
51             }
52             fout << '\n';
53             for (const auto& cp : coded_parts) {
54                 fout << cp;
55             }
56             fout.close();
57         }
58         else {
59             // Отправка закодированной части обратно процессу 0
60             MPI_Send(coded_part.data(), coded_part.size(), MPI_CHAR, 0, 0,
61                 ↪ MPI_COMM_WORLD);
62         }
63     }

```

Декодирование проходит по такой же схеме: нулевой процесс разбивает закодированный файл на части и рассылает эти части для декодирования на другие процессы, после чего принимает декодированный текст и записывает его в файл.

```

1 void LZW_parallel_decode(const std::vector<std::string>& init_dict, const
  ↪ std::string& fin_name, const std::string& fout_name) {
2     int rank, size;
3     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4     MPI_Comm_size(MPI_COMM_WORLD, &size);

```

```

5
6     std::vector<std::string> parts;
7     std::string part;
8     std::string decoded_part;
9
10    if (rank == 0) {
11        // Разделить файл только в процессе с рангом 0
12        std::vector<std::string> parts = split_file_decoder(fin_name);
13
14        // Рассылка каждой части файла соответствующему процессу
15        for (int i = 1; i < size; i++) {
16            MPI_Send(parts[i].data(), parts[i].size(), MPI_CHAR, i, 0,
17                    ↪ MPI_COMM_WORLD);
18        }
19        part = parts[0]; // Часть для процесса 0
20    }
21    else {
22        MPI_Status status;
23        MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
24        int count;
25        MPI_Get_count(&status, MPI_CHAR, &count);
26        part.resize(count);
27        MPI_Recv(&part[0], count, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
28                ↪ MPI_STATUS_IGNORE);
29    }
30
31    decoded_part = LZW_decoder(init_dict, part);
32
33    // Сбор декодированных частей обратно в процесс 0
34    if (rank == 0) {
35        std::vector<std::string> decoded_parts(size);
36        decoded_parts[0] = std::move(decoded_part);
37        for (int i = 1; i < size; i++) {
38            MPI_Status status;
39            MPI_Probe(i, 0, MPI_COMM_WORLD, &status);
40            int count;
41            MPI_Get_count(&status, MPI_CHAR, &count);
42            decoded_parts[i].resize(count);
43            MPI_Recv(&decoded_parts[i][0], count, MPI_CHAR, i, 0,
44                    ↪ MPI_COMM_WORLD, MPI_STATUS_IGNORE);
45        }
46
47        // Сохранение результатов
48        std::ofstream fout(fout_name, std::ios::binary);
49        for (const auto& dp : decoded_parts) {
50            fout << dp;

```

```
48         }
49         fout.close();
50     }
51     else {
52         // Отправка закодированной части обратно процессу 0
53         MPI_Send(decoded_part.data(), decoded_part.size(), MPI_CHAR, 0, 0,
54                 ↪ MPI_COMM_WORLD);
55     }
56 }
```

---

Кодирование и декодирование RLE происходит аналогичным образом. Единственное отличие этих функций - само кодирование, которое уже было написано до выполнения НИР, его реализация не рассматривается.

## 2 Результаты работы программы

При запуске программы она выводит сообщения об успешности или неудаче кодирования и декодирования, двуступенчатого кодирования и декодирования. Вот так выглядит вывод для программ на OpenMP и OpenMPI:

```
yakunin@deb1:~/coder-mp$ ./coder
Threads num: 3
RLE decoding was correct, thread 1
LZW decoding was correct, thread 0

LZW compression ratio: 2.891845, thread 2
RLE compression ratio: 1.148106, thread 2

RLE_LZW decoding was correct
LZW_RLE decoding was correct

RLE_LZW compression ratio: 1.45307
LZW_RLE compression ratio: 0.353457

Finishyakunin@deb1:~/coder-mp$ _
```

Рис. 6. Вывод OpenMP

```
yakunin@deb1:~/coder-mpi$ mpirun --host deb1,deb2,deb3,deb4 coder
LZW decoding was correct
RLE decoding was correct

LZW compression ratio: 2.84414
RLE compression ratio: 1.14705

RLE_LZW decoding was correct
LZW_RLE decoding was correct

RLE_LZW compression ratio: 1.41583
LZW_RLE compression ratio: 0.352634

Finish
Finish

Finish Finish yakunin@deb1:~/coder-mpi$
```

Рис. 7. Вывод OpenMPI

Как видно, программы успешно завершаются и выводятся сообщения о том, что декодированные файлы совпадают с исходным.

# Заключение

В ходе выполнения задания был освоен дистрибутив Linux Debian, а также базовые принципы параллельного программирования OpenMP и OpenMPI, получены навыки работы с виртуализацией в VirtualBox. Созданы виртуальные машины, между которыми налажена NAT-сеть и SSH-соединения. Также освоены ключевые функции MPI для обмена данными, и адаптирована на многопроцессорное выполнение ранее написанная программа.

Были получены полезные навыки, которые могут быть использованы в будущем при работе с многопроцессорными системами.

# Список литературы

- [1] Как настроить SSH вход без пароля: <https://wiki.merionet.ru/articles/kak-nastroit-ssh-vxod-bez-parolya> (дата обращения: 24.06.2024).
- [2] Хабр: Параллельные заметки №1 – технология OpenMP <https://habr.com/ru/companies/intel/articles/82486/6> (дата обращения: 24.06.2024).
- [3] OpenMP: Краткий обзор: [https://ssd.sccc.ru/sites/default/files/content/attach/343/lecture\\_openmp\\_2015.pdf](https://ssd.sccc.ru/sites/default/files/content/attach/343/lecture_openmp_2015.pdf) (дата обращения: 24.06.2024).
- [4] Документация OpenMPI: <https://www.lb.open-mpi.org/doc/v4.1/> (дата обращения: 24.06.2024).
- [5] GitHub: Hello World на OpenMPI: [https://github.com/mpitutorial/mpitutorial/blob/gh-pages/tutorials/mpi-hello-world/code/mpi\\_hello\\_world.c](https://github.com/mpitutorial/mpitutorial/blob/gh-pages/tutorials/mpi-hello-world/code/mpi_hello_world.c) (дата обращения: 24.06.2024).
- [6] GitHub: пример работы send и recv на OpenMPI: [https://github.com/pcaro90/mpi-examples/blob/master/03-send\\_recv\\_01.c](https://github.com/pcaro90/mpi-examples/blob/master/03-send_recv_01.c) (дата обращения: 24.06.2024).
- [7] Хабр: Серия статей про введение в OpenMPI: <https://habr.com/ru/articles/548266/> (дата обращения: 24.06.2024).
- [8] Для чего нужен каждый каталог в Linux: [https://itshaman.ru/articles/10/directory-linux#usr\\_src](https://itshaman.ru/articles/10/directory-linux#usr_src) (дата обращения: 24.06.2024).