

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»

«Научно исследовательская работа»

«Технологии параллельного программирования
в операционных системах Linux»

Тищенко А. А, гр. 5130201/20002

Санкт-Петербург, 2024

Содержание

Введение	3
1 Основная часть	4
1.1 Настройка окружения	4
1.1.1 Настройка NAT сети в VirtualBox	4
1.1.2 Создание и настройка виртуальных машин	5
1.1.3 Установка и настройка ОС	6
1.1.4 Файлы hosts и hostname	8
1.1.5 Создание пользователя	8
1.1.6 Настройка SSH соединения между виртуальными машинам	9
1.1.7 Установка компилятора, OpenMP и MPI	9
1.2 Изменения в исходной программе	11
1.2.1 Генерация случайного файла	11
1.2.2 Кодирование файла по алгоритму Хаффмана	12
1.2.3 Сравнение файлов для проверки операций кодиро- вания и декодирования	16
1.2.4 RLE кодирование файла	19
1.2.5 RLE декодирование файла	22
Заключение	25
Список литературы	26

Введение

Объектом исследования в данном проекте является операционная система на базе ядра Linux, а именно дистрибутив CentOS Stream 9, и технологии параллельного программирования в этой среде. Цель данного исследования заключается в изучении операционной системы CentOS Stream 9 и использовании технологий параллельного программирования OpenMP и MPI (в реализации MPICH) для выполнения вычислительно интенсивных задач. Основной задачей проекта является перенос и распараллеливание программы из лабораторной работы по теории графов с использованием данных технологий.

Для достижения этой цели были выполнены следующие этапы: установка и настройка виртуальной среды VirtualBox, создание и конфигурация четырех виртуальных машин, настройка локальной сети и SSH-подключений между машинами, установка и настройка программного обеспечения для работы с OpenMP и MPI, портирование и адаптация программы из лабораторной работы по теории графов для параллельного выполнения. В процессе работы приложение использовало вычислительные ресурсы четырех виртуальных машин, каждая из которых была настроена на использование двух ядер.

1 Основная часть

1.1 Настройка окружения

1.1.1 Настройка NAT сети в VirtualBox

Для того чтобы виртуальные машины, созданные в VirtualBox, могли взаимодействовать друг с другом, необходимо было создать и настроить NAT сеть. Адрес сети задавался вариантами работы. В моём случае это 10.12.191.128/26, где /26 это маска подсети или, что то же самое, 255.255.255.192.

Для того, чтобы создать заданную NAT сеть, необходимо зайти в менеджер сетей VirtualBox (см. Рис. 1), указать там имя сети и её адрес, IPv6 и DHCP в этой работе не используются, поэтому их лучше выключить.

Эта же сеть была выбрана в настройках всех создаваемых в рамках этой работы виртуальных машин (см. Рис. 2).

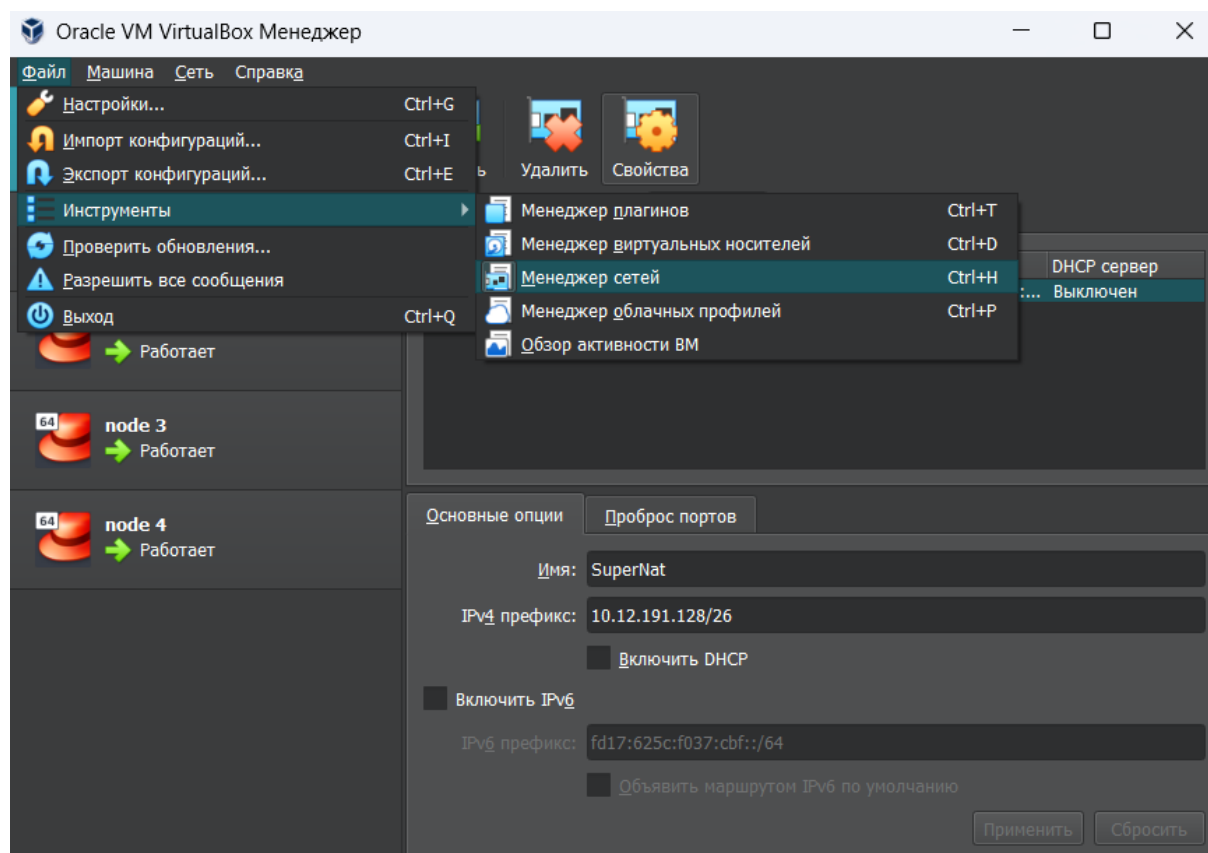


Рис. 1. Создание NAT сети в VirtualBox.

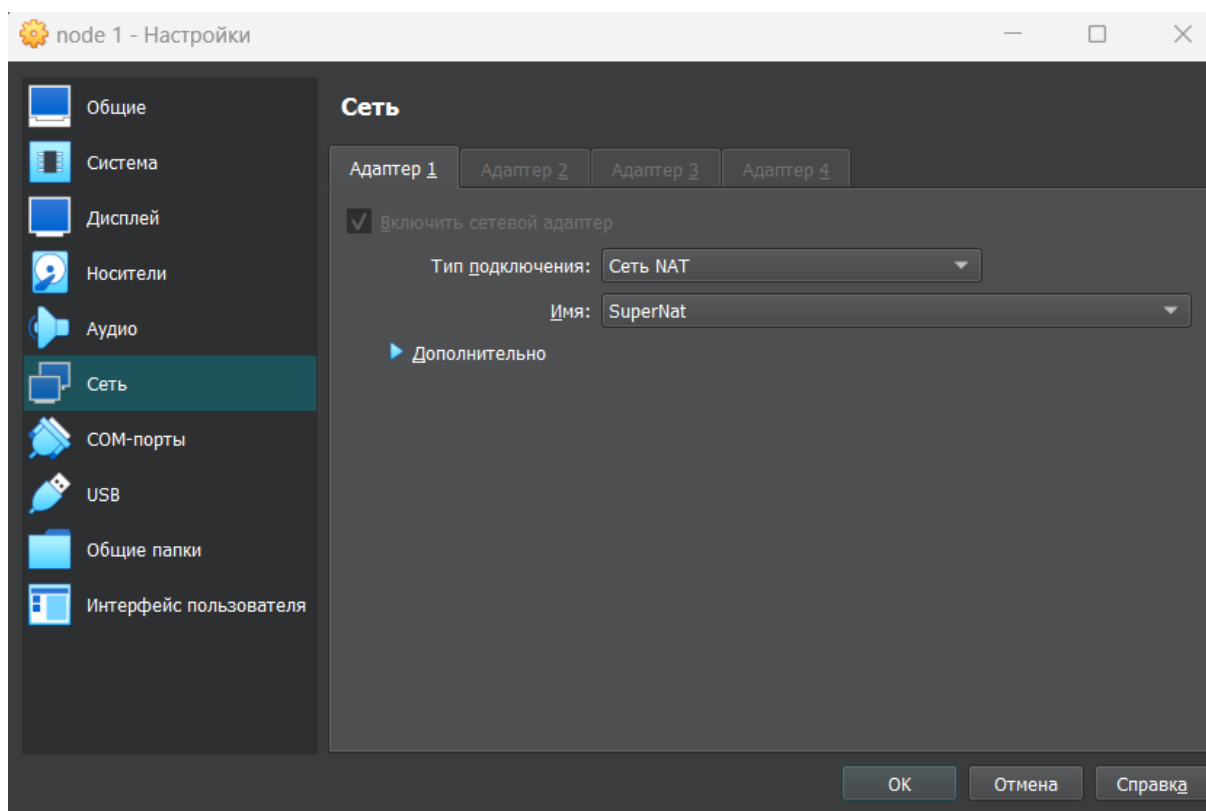


Рис. 2. Выбор сети NAT в настройках виртуальной машины.

1.1.2 Создание и настройка виртуальных машин

По варианту работы на виртуальные машины нужно было установить операционную систему CentOS Stream 9. Её образ доступен для скачивания на официальном сайте [1].

При создании виртуальной машины нужно указать её название, папку, в которой будут храниться её данные и путь до образа операционной системы. VirtualBox сам классифицирует систему как Linux версии Red Hat (64 bit) (см. Рис. 3). Этот вариант подходит, так как Cent OS действительно принадлежит семейству систем Red Hat.

Далее VirtualBox предложит определить размер выделяемых под виртуальную машину ресурсов. На моём компьютере установлено 16 гб оперативной памяти, а также процессор с 12 виртуальными ядрами. Я создал четыре виртуальные машины, поэтому для каждой выделил по 2 гб оперативной памяти и по 2 виртуальных ядра (см. Рис. 4).

Под жёсткий диск выделено по 40 гб, однако фактически каждая виртуальная машина с установленной операционной системой занимает около 3 гб.

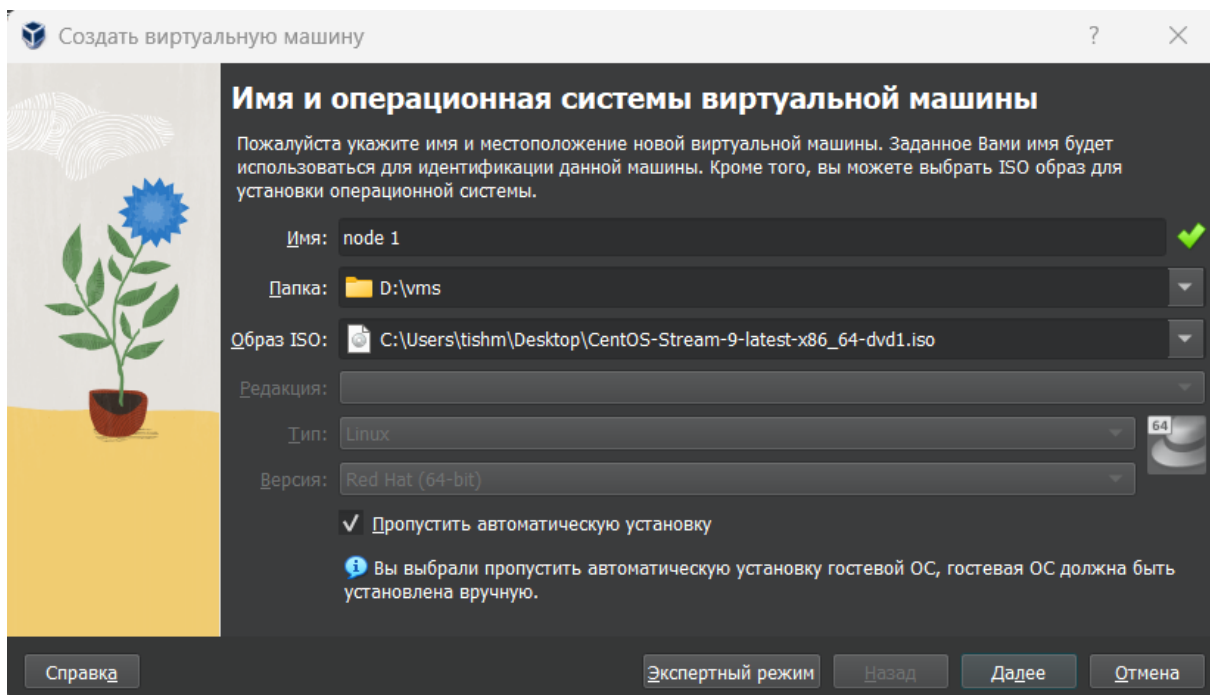


Рис. 3. Создание виртуальной машины в VirtualBox.

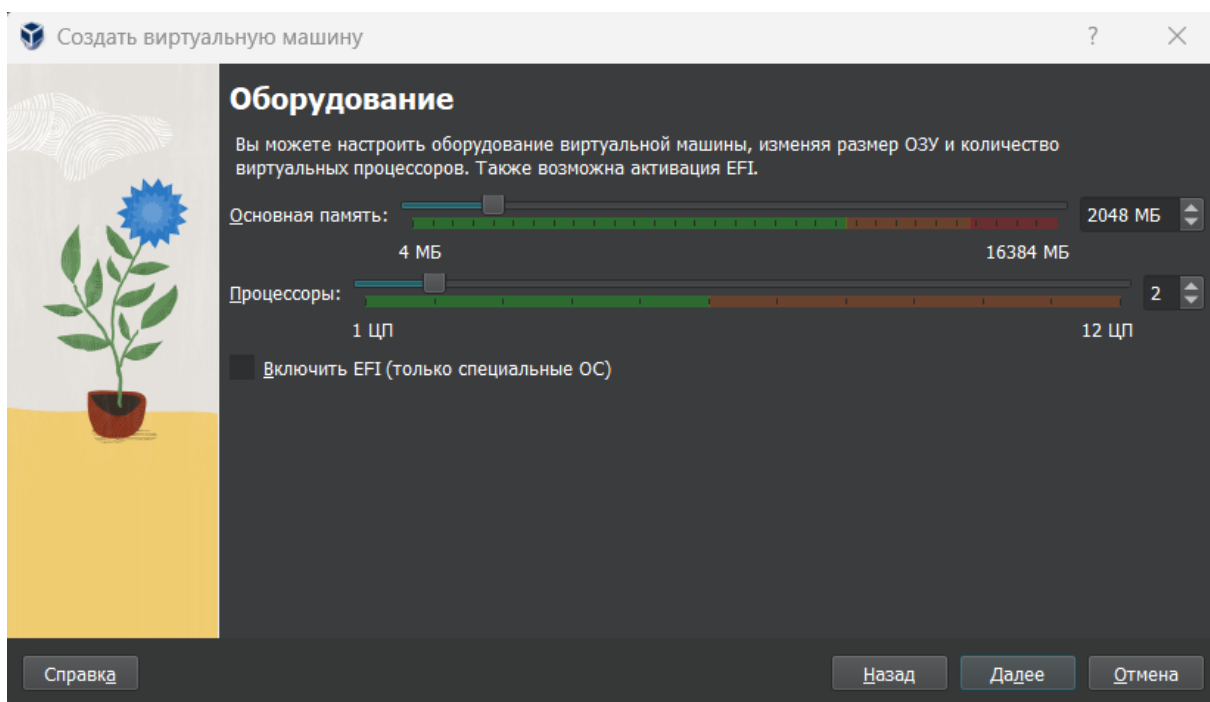


Рис. 4. Выделение ресурсов для виртуальной машины.

1.1.3 Установка и настройка ОС

При первом запуске виртуальной машины откроется установочник CentOS Stream. В настройках установочника для всех машин был выбран русский язык, а также устанавливаемый вариант операционной си-

стемы – **Server**, вместе **Server with GUI** по умолчанию. Одна из задач практики – симитировать работу на реальном суперкомпьютере. Разумеется GUI на реальном суперкомпьютере, как впрочем и почти на любом серверном оборудовании, не предоставляется.

Самые важные для работы настройки находятся в разделе «Имя сети и узла». Именно в нём настраивается подключение виртуальной машины к ранее созданной сети NAT. В подразделе «Параметры IPv4» необходимо добавить новый адрес для виртуальной машины, указать маску подсети и шлюз, а также сервер DNS и поисковый домен (см. Рис. 5).

В качестве адреса шлюза и сервера DNS был выбран первый доступный в сети адрес – 10.12.191.129, адрес перед ним – 10.12.191.128 – это адрес самой сети и использовать его нельзя. Также опытным путём было установлено, что нельзя использовать адрес, следующий после шлюза, поэтому первой машине был присвоен адрес 10.12.191.131, второй – 10.12.191.132 и так далее. Как поисковой домен была выбрана строчка **hpc**, аббревиатура от High Performance Computing.

После настройки сети необходимо задать пароль для пользователя **root** и запустить установку. Вспомогательного пользователя также можно создать на этом этапе, однако в этой работе пользователи создавались с помощью команд через терминал.

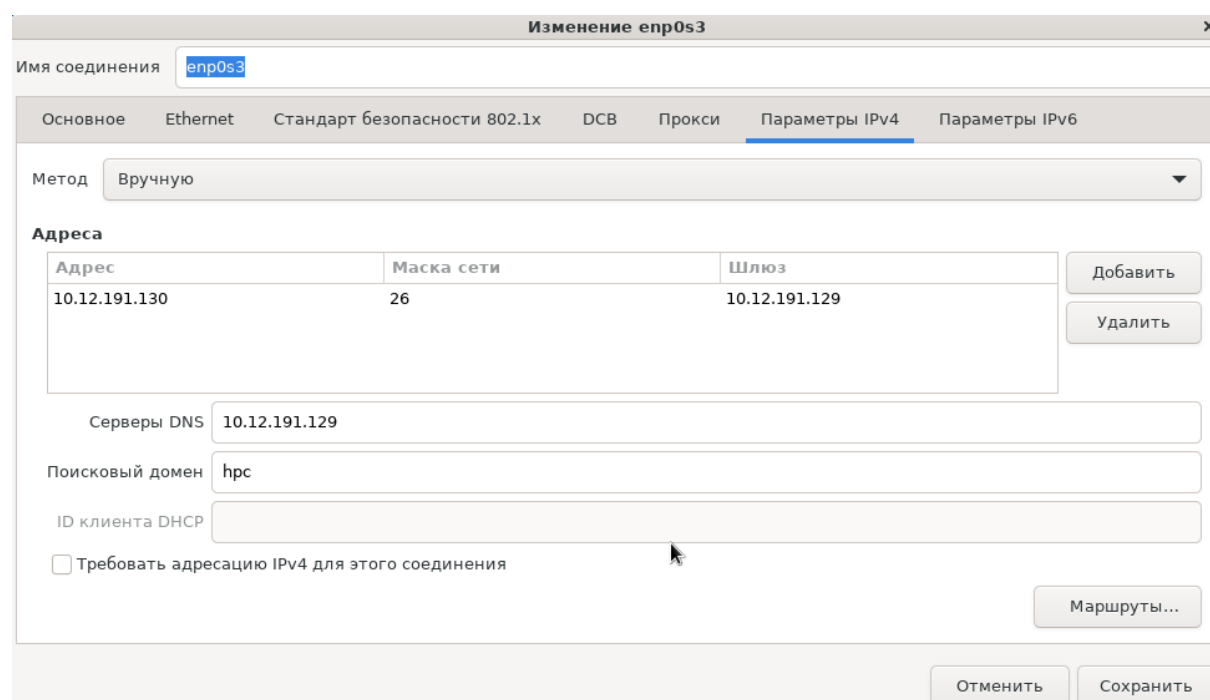
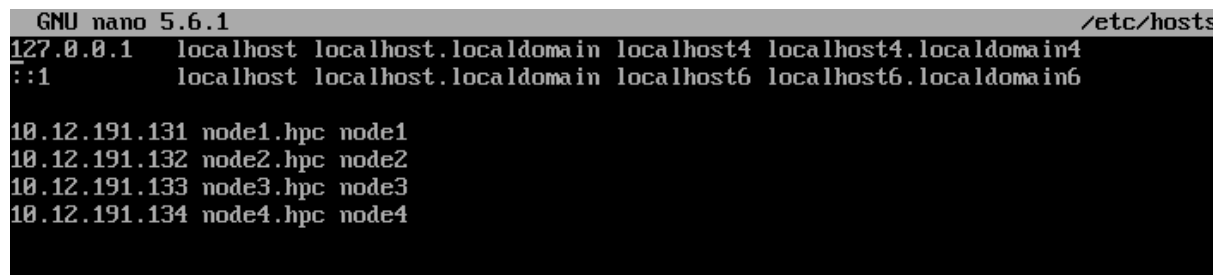


Рис. 5. Настройка адреса и сети во время установки CentOS Stream 9.

1.1.4 Файлы `hosts` и `hostname`

В первую очередь на всех машинах был заполнен файл `\etc\hosts`. С помощью встроенного редактора `nano`, на каждой машине в этом файле были перечислены адреса остальных машин (см. Рис. 6). После обновления этого файла, к другим машинами в сети можно обращаться по коротким именам, например `node1`, `node2` и так далее.



```
GNU nano 5.6.1 /etc/hosts
127.0.0.1    localhost localhost.localdomain localhost4 localhost4.localdomain4
::1        localhost localhost.localdomain localhost6 localhost6.localdomain6

10.12.191.131 node1.hpc node1
10.12.191.132 node2.hpc node2
10.12.191.133 node3.hpc node3
10.12.191.134 node4.hpc node4
```

Рис. 6. Заполнение файла `hosts`.

Также в файле `\etc \hostname` с помощью того же `nano` для каждой машины было указано её имя, те же `node1` для первой машины, `node2` для второй и так далее.

1.1.5 Создание пользователя

Есть множество причин, по которым не стоит запускать программы на `linux`, используя пользователя `root`. В общем случае это может повлечь за собой неисправимые последствия для операционной системы. Также `ssh` и `MPi` требуют дополнительной настройки для работы от имени пользователя `root`.

С помощью следующих трёх команд можно создать пользователя, задать его пароль и дать ему доступ к использованию `sudo`:

```
useradd arity
passwd arity
usermod -aG wheel arity
```

Переключиться на только что созданного пользователя можно с помощью команды `su - arity`.

1.1.6 Настройка SSH соединения между виртуальными машинами

Для настройки SSH соединения в первую очередь необходимо создать пару ключей с помощью команды `ssh-keygen`. Затем необходимо будет скопировать публичный ключ на остальные машины. Например, для первой машины (`node1`) необходимо было выполнить следующие команды:

```
ssh-copy-id arity@node2
ssh-copy-id arity@node3
ssh-copy-id arity@node4
```

Такую же последовательность команд необходимо было проделать и на других машинах. После их выполнения, стало возможным подключение по SSH с одной виртуальной машины на другую, например, с помощью команды `ssh arity@node2` – выполнит подключение ко второй машине. Если ключи созданы и скопированы верно, то при подключении вводить пароль не потребуется.

1.1.7 Установка компилятора, OpenMP и MPI

В CentOS Stream используется пакетный менеджер `dnf`. Компиляторы `gcc` и `g++` для языков C и C++ соответственно, устанавливаются с помощью команды `dnf install gcc gcc-c++`. Перед установкой рекомендуется обновить уже установленные пакеты с помощью команды `dnf upgrade`.

Поддержка OpenMP встроена в компилятор `g++` по умолчанию. Пример тестовой программы с использованием OpenMP представлен в листинге 1. При компиляции такой программы необходимо указать дополнительный флаг `-fopenmp`. Количество потоков можно указать с помощью переменной окружения `OMP_NUM_THREADS`. Пример команд для компиляции и запуска тестового файла:

```
export OMP_NUM_THREADS=4
g++ -fopenmp -o hello hello.cpp
./hello
```

Листинг 1. Минимальная программа для проверки работоспособности OpenMP.

```
1 #include <iostream>
2 #include <omp.h>
3
```

```

4 int main() {
5     #pragma omp parallel
6     {
7         std::cout << "Hello,_World_from_thread_" << omp_get_thread_num() << std::endl;
8     }
9     return 0;
10 }

```

В качестве реализации MPI была выбрана библиотека MPICH. Хотя изначально планировалось использование OpenMPI, в ходе экспериментов выяснилось, что MPICH лучше совместим с CentOS Stream. На всех машинах она была установлена с помощью команды `dnf install mpich mpich-devel`. После установки, необходимо было также добавить в PATH путь до неё. Для этого была построена команда

```
echo 'export PATH=/usr/lib64/mpich/bin:$PATH' >> ~/.bashrc
```

Она дописывает в настройки bash строчку, которая добавляет в PATH нужный путь. Это сделано для того, чтобы не добавлять путь вручную при каждом новом запуске bash.

После установки, можно попробовать запустить простой скрипт из листинга 2. Скомпилировать и запустить его можно с помощью команд:

```
mpicxx -o hello hello.cpp
mpiexec -n 2 ./hello
```

Листинг 2. Минимальная программа для проверки работоспособности MPI.

```

1 #include <mpi.h>
2 #include <iostream>
3
4 int main(int argc, char** argv) {
5     MPI_Init(&argc, &argv);
6
7     int world_size;
8     MPI_Comm_size(MPI_COMM_WORLD, &world_size);
9
10    int world_rank;
11    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
12
13    std::cout << "Hello,_world_from_rank_" << world_rank
14        << "_out_of_" << world_size << "_processors" << std::endl;
15
16    MPI_Finalize();
17    return 0;
18 }

```

Перед запуском скрипта на нескольких виртуальных машинах нужно убедиться, что MPICH и g++ установлены на всех них. Также скомпилированный исполняемый файл нужно скопировать в одно и то же место на всех машинах, например с помощью команды `scp`. Вот пример команд

для копирования программы с первой на все остальные машины:

```
scp /home/arity/shared/hello arity@node2:/home/arity/shared/hello
scp /home/arity/shared/hello arity@node3:/home/arity/shared/hello
scp /home/arity/shared/hello arity@node4:/home/arity/shared/hello
```

Затем можно запустить программу с помощью команды:

```
mpirun -n 4 -host node1,node2,node3,node4 /home/arity/shared/hello
```

1.2 Изменения в исходной программе

1.2.1 Генерация случайного файла

В листинге 3 представлен код функций, которые отвечают за генерацию файла, состоящего из случайной последовательности заданных символов.

Функция `generateFile` принимает на вход:

1. `const string &filename` – название файла для сохранения результата,
2. `const string &alphabet` – строку, задающую алфавит, то есть множество допустимых символов,
3. `int total_size` – количество символов, которое надо сгенерировать,
4. `int rank` – ранк, то есть номер текущего MPI процесса,
5. `int size` – общее количество процессов MPI.

Нулевой процесс считается корневым, в этой и других функциях, помимо непосредственно вычислений, он отвечает за разделение исходной задачи на части, разделение этих частей между процессами и сбор результатов. Отправка и сбор данных производятся с помощью функции `MPI_Gather`.

В `generateFile` также используется вспомогательная функция `generateChunk`, которая отвечает непосредственно за генерацию случайных символов. В ней используются директива OpenMP `#pragma omp parallel for` для дополнительной параллелизации генерации по потокам.

Листинг 3. Реализация параллельной версии функции generateFile.

```

1 void generateChunk(std::string &chunk, const std::string &alphabet, int chunkSize) {
2     int alphabetSize = alphabet.size();
3     std::random_device rd;
4     std::mt19937 generator(rd());
5     std::uniform_int_distribution<> distribution(0, alphabetSize - 1);
6
7     chunk.resize(chunkSize);
8
9     #pragma omp parallel for
10    for (int i = 0; i < chunkSize; ++i) {
11        char randomChar = alphabet[distribution(generator)];
12        chunk[i] = randomChar;
13    }
14 }
15
16 void generateFile(const string &filename, const string &alphabet, int total_size, int rank, int size)
17 {
18     int chunkSize = total_size / size;
19     string chunk;
20
21     generateChunk(chunk, alphabet, chunkSize);
22
23     if (rank == 0)
24     {
25         ofstream file(filename, ios::binary);
26         if (!file.is_open())
27         {
28             cout << "Не удалось открыть файл для записи." << endl;
29             MPI_Abort(MPI_COMM_WORLD, 1);
30             return;
31         }
32
33         vector<char> global_data(total_size);
34         MPI_Gather(chunk.data(), chunkSize, MPI_CHAR, global_data.data(), chunkSize,
35         MPI_CHAR, 0, MPI_COMM_WORLD);
36         file.write(global_data.data(), global_data.size());
37         file.close();
38         cout << "Файл_" << filename << "_успешно создан." << endl;
39     }
40     else
41     {
42         MPI_Gather(chunk.data(), chunkSize, MPI_CHAR, nullptr, 0, MPI_CHAR, 0,
43         MPI_COMM_WORLD);
44     }
45 }

```

1.2.2 Кодирование файла по алгоритму Хаффмана

В листинге 4 представлен код метода, который отвечает за кодирование файла по алгоритму Хаффмана.

Метод `encodeFileBin` принимает на вход:

1. `const string &inputFileName` – название файла с исходным текстом,
2. `const string &outputFileName` – название файла для сохранения результата.

Также он использует свойства объекта `Huffman`, которые задаются при его создании:

1. `int rank` – ранк, то есть номер текущего MPI процесса,
2. `int size` – общее количество процессов MPI.

Основная вычислительная сложность алгоритма Хаффмана заключается в подсчёте частот символов в тексте. Именно эта часть и была переписана так, чтобы подсчёт производился на нескольких машинах и потоках одновременно. Параллельная версия метода `countFrequencies` представлена в том же листинге 4. Функция принимает строку – текст, в котором необходимо подсчитать частоты символов. На нулевом узле функция разделяет текст на части и, с помощью `MPI_Send` и `MPI_Recv`, отправляет разным процессам. Затем идёт непосредственно подсчёт частот символов. Этот процесс также распределён по разным потокам с помощью OpenMP. В этом случае использовались три различные директивы:

1. `#pragma omp parallel` – начинает секцию, которая будет выполняться всеми потоками,
2. `#pragma omp for nowait` – выполняет итерации цикла `for` параллельно, при этом, из-за ключевого слова `nowait`, потоки не будут ждать синхронизации по окончании цикла, а продолжат выполнение программы,
3. `#pragma omp critical` – начинает секцию, код которой гарантировано не будет выполняться на двух потоках одновременно. В данном случае это необходимо, так как в секции изменяется разделяемый ресурс – переменная `chunkFreq`.

В конце функции `countFrequencies`, нулевой поток собирает результаты в одном месте, используя те же `MPI_Send` и `MPI_Recv`.

Листинг 4. Реализация параллельной версии метода `Huffman::encodeFileBin`.

```
1 void Huffman::countFrequencies(const std::string &text)
```

```

2  {
3      freq.clear();
4
5      int textSize = text.size();
6      int chunkSize = textSize / size;
7
8      std::string chunk;
9
10     if (rank == 0)
11     {
12         // Распределяем части текста по узлам
13         for (int i = 1; i < size; i++)
14         {
15             int startIndex = i * chunkSize;
16             int endIndex = (i == size - 1) ? textSize : startIndex + chunkSize;
17             std::string chunk = text.substr(startIndex, endIndex - startIndex);
18
19             int chunkLength = chunk.size();
20             MPI_Send(&chunkLength, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
21             MPI_Send(chunk.c_str(), chunkLength, MPI_CHAR, i, 0, MPI_COMM_WORLD);
22         }
23
24         // Для корневого узла
25         chunk = text.substr(0, chunkSize);
26     }
27     else
28     {
29         // Для других узлов
30         int chunkLength;
31         MPI_Recv(&chunkLength, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
32             MPI_STATUS_IGNORE);
33
34         chunk.resize(chunkLength);
35         MPI_Recv(&chunk[0], chunkLength, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
36             MPI_STATUS_IGNORE);
37
38         // Все машины считают частоты
39         std::map<char, int> chunkFreq;
40
41         #pragma omp parallel
42         {
43             std::map<char, int> localFreq;
44
45             // nowait — потоки не ждут друг друга и могут в разное время закончить цикл
46             #pragma omp for nowait
47             for (size_t i = 0; i < chunk.size(); ++i)
48             {
49                 char ch = chunk[i];
50                 localFreq[ch]++;
51             }
52
53             // А тут chunkFreq — разделяемый ресурс, только один поток одновременно
54             // может выполнять код из секции critical
55             #pragma omp critical
56             {

```

```

56         for (const auto &pair : localFreq)
57         {
58             chunkFreq[pair.first] += pair.second;
59         }
60     }
61 }
62
63 // Собираем результат на корневом узле
64 if (rank == 0)
65 {
66     for (int i = 1; i < size; i++)
67     {
68         int recv_size;
69         MPI_Recv(&recv_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
70         for (int j = 0; j < recv_size; j++)
71         {
72             char ch;
73             int count;
74             MPI_Recv(&ch, 1, MPI_CHAR, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
);
75             MPI_Recv(&count, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
76             freq[ch] += count;
77         }
78     }
79
80     // Не забываем добавить частоты, подсчитанные на корневом узле
81     for (const auto &pair : chunkFreq)
82     {
83         freq[pair.first] += pair.second;
84     }
85 }
86 else
87 {
88     // Отправляем подсчитанные частоты на корневой узел
89     int sendSize = chunkFreq.size();
90     MPI_Send(&sendSize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
91     for (const auto &pair : chunkFreq)
92     {
93         MPI_Send(&pair.first, 1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
94         MPI_Send(&pair.second, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
95     }
96 }
97 }
98
99 void Huffman::encodeFileBin(const string &inputFileName, const string &outputFileName)
100 {
101     if (rank != 0)
102     {
103         // Неосновные узлы будут ждать, когда корневой узел прочитает файл
104         // и раздаст всем текст
105         countFrequencies("");
106     }
107
108     if (rank == 0)

```

```

109     {
110         ifstream inputFile(inputFileName);
111         string text((istreambuf_iterator<char>(inputFile), istreambuf_iterator<char>()));
112         inputFile.close();
113
114         countFrequencies(text);
115
116         buildTree();
117         buildCodes(root, "");
118
119         ofstream outputFile(outputFileName, ios::binary);
120         unsigned char bitBuffer = 0;
121         int bitCount = 0;
122
123         for (char ch : text)
124         {
125             string code = huffmanCodes[ch];
126             for (char bit : code)
127             {
128                 bitBuffer = (bitBuffer << 1) | (bit == '1');
129                 bitCount++;
130                 if (bitCount == 8)
131                 {
132                     outputFile.write(reinterpret_cast<char*>(&bitBuffer), 1);
133                     bitBuffer = 0;
134                     bitCount = 0;
135                 }
136             }
137         }
138
139         // Обработка последнего байта, если он не полный
140         if (bitCount > 0)
141         {
142             bitBuffer <<= (8 - bitCount); // Сдвигаем оставшиеся биты влево
143             outputFile.write(reinterpret_cast<char*>(&bitBuffer), 1);
144         }
145
146         // Записываем количество значимых бит в последнем байте данных
147         unsigned char lastByteBits = bitCount > 0 ? bitCount : 8;
148         outputFile.write(reinterpret_cast<char*>(&lastByteBits), 1);
149
150         outputFile.close();
151         cout << "Файл_успешно_закодирован_и_сохранен_как_" << outputFileName << endl;
152     }
153 }

```

1.2.3 Сравнение файлов для проверки операций кодирования и декодирования

В листинге 5 представлен код функции `areFilesEqual`, которая отвечает за сравнение содержимого двух файлов.

Функция `areFilesEqual` принимает на вход:

1. `const string &filename1` – название первого файла для сравнения,
2. `const string &filename2` – название второго файла для сравнения,
3. `int rank` – ранк, то есть номер текущего MPI процесса,
4. `int size` – общее количество процессов MPI.

Нулевой процесс в этой функции отвечает за чтение файлов и разделение их на куски, отправку этих кусков остальным процессам и агрегацию результата. Для этих целей используются функции `MPI_Bcast`, `MPI_Send` и `MPI_Recv`. `MPI_Bcast` используется для того, чтобы сообщить процессам размер файлов, чтобы они могли самостоятельно вычислить размеры своих кусочков.

Также в функции используется директива `#pragma omp parallel for reduction(&&:isEqual)`. С её помощью итерации цикла распределяются по потокам, а затем результат агрегируется в переменную `isEqual`.

Листинг 5. Реализация параллельной версии функции `areFilesEqual`.

```
1  bool areFilesEqual(const std::string &filename1, const std::string &filename2, int rank, int size)
2  {
3      std::ifstream file1, file2;
4      std::vector<char> buffer1, buffer2;
5      int fileSize1 = 0, fileSize2 = 0;
6      bool stopEverything = false;
7
8      // Чтение файлов и проверка размеров на корневом узле
9      if (rank == 0)
10     {
11         file1.open(filename1, std::ios::binary);
12         file2.open(filename2, std::ios::binary);
13
14         if (!file1.is_open() || !file2.is_open())
15         {
16             std::cout << "Ошибка_при_открытии_файлов_для_сравнения." << std::endl;
17             stopEverything = true;
18         }
19
20         if (!stopEverything)
21         {
22             file1.seekg(0, std::ios::end);
23             file2.seekg(0, std::ios::end);
24             fileSize1 = file1.tellg();
25             fileSize2 = file2.tellg();
26
27             if (fileSize1 != fileSize2)
```

```

28         {
29             stopEverything = true;
30         }
31     }
32 }
33
34 MPI_Bcast(&stopEverything, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);
35 if (stopEverything)
36 {
37     return false;
38 }
39
40 // Сообщаем всем процессам размер файлов
41 MPI_Bcast(&fileSize1, 1, MPI_INT, 0, MPI_COMM_WORLD);
42
43 // Размеры чанков
44 int baseChunkSize = fileSize1 / size;
45 int remaining = fileSize1 % size;
46
47 // Размер чанка для текущего процесса
48 int start = rank * baseChunkSize;
49 int end = (rank == size - 1) ? (start + baseChunkSize + remaining) : (start + baseChunkSize);
50 int chunkSize = end - start;
51
52 buffer1.resize(chunkSize);
53 buffer2.resize(chunkSize);
54
55 // Корневой процесс распределяет чанки
56 if (rank == 0)
57 {
58     for (int i = 1; i < size; ++i)
59     {
60         int procStart = i * baseChunkSize;
61         int procEnd = (i == size - 1) ? (procStart + baseChunkSize + remaining) : (procStart +
baseChunkSize);
62         int procSize = procEnd - procStart;
63
64         std::vector<char> procBuffer1(procSize);
65         std::vector<char> procBuffer2(procSize);
66
67         file1.seekg(procStart, std::ios::beg);
68         file1.read(procBuffer1.data(), procSize);
69         file2.seekg(procStart, std::ios::beg);
70         file2.read(procBuffer2.data(), procSize);
71
72         MPI_Send(procBuffer1.data(), procSize, MPI_CHAR, i, 0, MPI_COMM_WORLD);
73         MPI_Send(procBuffer2.data(), procSize, MPI_CHAR, i, 0, MPI_COMM_WORLD);
74     }
75
76     // Корневой процесс тоже обрабатывает свой чанк
77     file1.seekg(0, std::ios::beg);
78     file1.read(buffer1.data(), chunkSize);
79     file2.seekg(0, std::ios::beg);
80     file2.read(buffer2.data(), chunkSize);
81 }
82 else

```

```

83     {
84         MPI_Recv(buffer1.data(), chunkSize, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
85         MPI_Recv(buffer2.data(), chunkSize, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
86     }
87
88     // Все машины производят операцию сравнения для своих чанков
89     bool isEqual = true;
90
91     #pragma omp parallel for reduction(&&:isEqual)
92     for (int i = 0; i < chunkSize; ++i) {
93         if (buffer1[i] != buffer2[i]) {
94             isEqual = false;
95         }
96     }
97
98     // Агрегируем результаты с помощью logical and
99     bool globalEqual;
100    MPI_Reduce(&isEqual, &globalEqual, 1, MPI_C_BOOL, MPI_LAND, 0,
MPI_COMM_WORLD);
101
102    if (rank == 0)
103    {
104        return globalEqual;
105    }
106
107    return true;
108 }

```

1.2.4 RLE кодирование файла

В листинге 6 представлен код метода `RLE::encodeFile`, который отвечает за RLE кодирование указанного файла.

Метод `encodeFile` принимает на вход:

1. `const string &inputFileName` – название исходного файла,
2. `const string &outputFileName` – название файла для сохранения результата,
3. `int rank` – ранк, то есть номер текущего MPI процесса,
4. `int size` – общее количество процессов MPI.

Нулевой процесс в этой функции отвечает за чтение входного файла и разделение его на куски, отправку этих кусков остальным процессам, агрегацию результата и его сохранение в файл. Для этих целей используются функции `MPI_Bcast`, `MPI_Send` и `MPI_Recv`. `MPI_Bcast` используется

для того, чтобы сообщить процессам размер файлов, чтобы они могли самостоятельно вычислить размеры своих кусочков. Также для сбора данных используются функции `MPI_Gather` и `MPI_Gatherv`. `MPI_Gatherv` отличается от `MPI_Gather` тем, что позволяет собирать куски данных разной длины от разных процессов.

Листинг 6. Реализация параллельной версии метода `RLE::encodeFile`.

```

1 void RLE::encodeFile(const std::string& inputFileName, const std::string& outputFileName, int size,
2   int rank) {
3   ifstream inputFile;
4   ofstream outputFile;
5
6   if (rank == 0) {
7     inputFile.open(inputFileName, ios::binary);
8     outputFile.open(outputFileName, ios::binary);
9
10    if (!inputFile.is_open() || !outputFile.is_open()) {
11      cout << "Ошибка_при_открытии_файлов." << endl;
12      return;
13    }
14
15    // Сообщаем всем размер файла
16    int fileSize;
17    if (rank == 0) {
18      inputFile.seekg(0, ios::end);
19      fileSize = inputFile.tellg();
20      inputFile.seekg(0, ios::beg);
21    }
22    MPI_Bcast(&fileSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
23
24    // Размеры чанков
25    int baseChunkSize = fileSize / size;
26    int remaining = fileSize % size;
27
28    // Определяем размер чанка для текущего процесса
29    int start = rank * baseChunkSize;
30    int end = (rank == size - 1) ? (start + baseChunkSize + remaining) : (start + baseChunkSize);
31    int chunkSize = end - start;
32
33    std::vector<char> buffer(chunkSize);
34
35    if (rank == 0) {
36      for (int i = 1; i < size; ++i) {
37        int procStart = i * baseChunkSize;
38        int procEnd = (i == size - 1) ? (procStart + baseChunkSize + remaining) : (procStart +
39          baseChunkSize);
40        int procSize = procEnd - procStart;
41
42        std::vector<char> procBuffer(procSize);
43        inputFile.seekg(procStart, ios::beg);
44        inputFile.read(procBuffer.data(), procSize);
45
46        MPI_Send(procBuffer.data(), procSize, MPI_CHAR, i, 0, MPI_COMM_WORLD);

```

```

46     }
47
48     // Корневой процесс тоже обрабатывает свой чанк
49     inputFile.seekg(start, ios::beg);
50     inputFile.read(buffer.data(), chunkSize);
51 } else {
52     MPI_Recv(buffer.data(), chunkSize, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
53     MPI_STATUS_IGNORE);
54 }
55
56 // Все обрабатывают свой чанк
57 std::vector<char> encodedChunk;
58 char currentChar;
59 char previousChar = '\0';
60 unsigned char count = 0;
61
62 for (int i = 0; i < chunkSize; ++i) {
63     currentChar = buffer[i];
64     if (currentChar == previousChar && count < 255) {
65         ++count;
66     } else {
67         if (count > 0) {
68             encodedChunk.push_back(count);
69             encodedChunk.push_back(previousChar);
70         }
71         previousChar = currentChar;
72         count = 1;
73     }
74 }
75
76 if (count > 0) {
77     encodedChunk.push_back(count);
78     encodedChunk.push_back(previousChar);
79 }
80
81 // Собираем размеры закодированных чанков
82 int EncodedChunkSize = encodedChunk.size();
83 std::vector<int> encodedSizes(size);
84 MPI_Gather(&EncodedChunkSize, 1, MPI_INT, encodedSizes.data(), 1, MPI_INT, 0,
85 MPI_COMM_WORLD);
86
87 // Собираем сами закодированные чанки
88 std::vector<char> globalEncodedData;
89 std::vector<int> displs(size);
90 if (rank == 0) {
91     int totalEncodedSize = 0;
92     for (int i = 0; i < size; ++i) {
93         displs[i] = totalEncodedSize;
94         totalEncodedSize += encodedSizes[i];
95     }
96     globalEncodedData.resize(totalEncodedSize);
97 }
98
99 // MPI_Gatherv отличается от MPI_Gather тем, что длина возвращаемых чанков может
100 // быть переменной
101 MPI_Gatherv(encodedChunk.data(), EncodedChunkSize, MPI_CHAR,
102             globalEncodedData.data(), encodedSizes.data(), displs.data(), MPI_CHAR, 0,

```

```

100     MPI_COMM_WORLD);
101     // Корневой процесс сохраняет результат в файл
102     if (rank == 0) {
103         outputFile.write(globalEncodedData.data(), globalEncodedData.size());
104         inputFile.close();
105         outputFile.close();
106         cout << "Файл_успешно_закодирован_и_сохранен_как_" << outputFileName << endl;
107     }
108 }

```

1.2.5 RLE декодирование файла

В листинге 7 представлен код метода `RLE::decodeFile`, который отвечает за RLE декодирование указанного файла.

Метод `decodeFile` принимает на вход:

1. `const string &inputFileName` – название файла, закодированного с помощью RLE,
2. `const string &outputFileName` – название файла для сохранения результата,
3. `int rank` – ранк, то есть номер текущего MPI процесса,
4. `int size` – общее количество процессов MPI.

Нулевой процесс в этой функции отвечает за чтение входного файла и разделение его на куски, отправку этих кусков остальным процессам, агрегацию результата и его сохранение в файл. При разделении исходного файла важно также учитывать структуру закодированных данных. В моей версии RLE кодирования, каждый чётный байт данных в файл это число от 0 до 255 в двоичном виде, за ним всегда следует байт, содержащий символ – `char`. Поэтому разделять файл можно только на куски чётной длины.

В `RLE::decodeFile` используются функции `MPI_Bcast`, `MPI_Send` и `MPI_Recv`. `MPI_Bcast` используется для того, чтобы сообщить процессам размер файлов, чтобы они могли самостоятельно вычислить размеры своих кусочков. Также для сбора данных используются функции `MPI_Gather` и `MPI_Gatherv`. `MPI_Gatherv` приходится использовать так как при декодировании RLE, мы не можем заранее узнать длину исходного текста, соответственно, у всех процессов она может быть разной и вычисляется динамически.

Листинг 7. Реализация параллельной версии метода RLE::decodeFile.

```

1 void RLE::decodeFile(const std::string& inputFileName, const std::string& outputFileName, int size,
2     int rank) {
3     ifstream inputFile;
4     ofstream outputFile;
5
6     if (rank == 0) {
7         inputFile.open(inputFileName, ios::binary);
8         outputFile.open(outputFileName, ios::binary);
9
10        if (!inputFile.is_open() || !outputFile.is_open()) {
11            cout << "Ошибка_при_открытии_файлов." << endl;
12            // Stop Everything
13            return;
14        }
15    }
16
17    int fileSize;
18    if (rank == 0) {
19        inputFile.seekg(0, ios::end);
20        fileSize = inputFile.tellg();
21        inputFile.seekg(0, ios::beg);
22    }
23
24    // Передаем размер файла всем процессам
25    MPI_Bcast(&fileSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
26
27    // Определяем размеры чанков, гарантируя, что они делятся на 2
28    int baseChunkSize = (fileSize / (2 * size)) * 2;
29    int remaining = fileSize % baseChunkSize;
30
31    // Определяем размер чанка для текущего процесса
32    int start = rank * baseChunkSize;
33    int end = (rank == size - 1) ? (start + baseChunkSize + remaining) : (start + baseChunkSize);
34    int chunkSize = end - start;
35
36    std::vector<char> encodedChunk(chunkSize);
37
38    if (rank == 0) {
39        for (int i = 1; i < size; ++i) {
40            int procStart = i * baseChunkSize;
41            int procEnd = (i == size - 1) ? (procStart + baseChunkSize + remaining) : (procStart +
42                baseChunkSize);
43            int procSize = procEnd - procStart;
44
45            std::vector<char> procBuffer(procSize);
46            inputFile.seekg(procStart, ios::beg);
47            inputFile.read(procBuffer.data(), procSize);
48
49            MPI_Send(procBuffer.data(), procSize, MPI_CHAR, i, 0, MPI_COMM_WORLD);
50        }
51
52        // Корневой процесс тоже обрабатывает свой чанк
53        inputFile.seekg(start, ios::beg);
54        inputFile.read(encodedChunk.data(), chunkSize);
55    } else {

```

```

54     MPI_Recv(encodedChunk.data(), chunkSize, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
55     MPI_STATUS_IGNORE);
56 }
57 // Каждый процесс декодирует свой чанк
58 std::vector<char> decodedChunk;
59 for (int i = 0; i < chunkSize; i += 2) {
60     unsigned char count = static_cast<unsigned char>(encodedChunk[i]);
61     char symbol = encodedChunk[i + 1];
62     decodedChunk.insert(decodedChunk.end(), count, symbol);
63 }
64
65 // Собираем размеры декодированных чанков
66 int decodedChunkSize = decodedChunk.size();
67 std::vector<int> decodedSizes(size);
68 MPI_Gather(&decodedChunkSize, 1, MPI_INT, decodedSizes.data(), 1, MPI_INT, 0,
69     MPI_COMM_WORLD);
70
71 // Собираем сами декодированные чанки
72 std::vector<char> globalDecodedData;
73 std::vector<int> displs(size);
74 if (rank == 0) {
75     int totalDecodedSize = 0;
76     for (int i = 0; i < size; ++i) {
77         displs[i] = totalDecodedSize;
78         totalDecodedSize += decodedSizes[i];
79     }
80     globalDecodedData.resize(totalDecodedSize);
81 }
82 MPI_Gatherv(decodedChunk.data(), decodedChunkSize, MPI_CHAR,
83     globalDecodedData.data(), decodedSizes.data(), displs.data(), MPI_CHAR, 0,
84     MPI_COMM_WORLD);
85
86 // Корневой процесс сохраняет результат в файл
87 if (rank == 0) {
88     outputFile.write(globalDecodedData.data(), globalDecodedData.size());
89     inputFile.close();
90     outputFile.close();
91     cout << "Файл_успешно_декодирован_и_сохранен_как_" << outputFileName << endl;
92 }

```


Заключение

В ходе этой работы были изучены основы взаимодействия с терминалом и семейством операционных систем Linux, на примере операционной системы CentOS Stream 9. Также удалось познакомиться с технологиями виртуализации на примере VirtualBox. Был получен практический опыт написания и отладки программ, выполняющихся параллельно. В работе были использованы основные директивы OpenMP и функции MPI. Помимо этого, в ходе работы были получены базовые представления о работе и настройке NAT сетей, а также начальные навыки работы с SSH.

В совокупности полученные навыки и опыт, может послужить хорошей практической основой для дальнейшего углубления в изучение суперкомпьютерных технологий.

Список литературы

- [1] CentOS Stream official download page // URL: <https://www.centos.org/centos-stream/#tab-3>
- [2] Официальный сайт с документацией MPICH MPI // URL: <https://www.mpich.org/documentation/guides/>
- [3] Официальный сайт с документацией OpenMP // URL: <https://www.openmp.org/resources/refguides/>