МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО 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 о выполнении практической работы по дисциплине
«Научно-исследовательская работа»

«Технологии параллельного программирования в операционных системах Linux»

Группа: 5130201/20002

Обучающийся: _____ Шклярова Ксения Алексеевна

Руководитель: _____ Чуватов Михаил Владимирович

«_____»_____20___г.

Санкт-Петербург, 2024

Содержание

Основные термины и определения

1. Виртуальная машина - это программная среда, которая имитирует физический компьютер, позволяя запускать на нем операционную систему.
2. VirtualBox (Oracle VM VirtualBox) — программный продукт виртуализации для операционных систем Windows, Linux, FreeBSD, macOS, Solaris/OpenSolaris, ReactOS, DOS и других.
3. ОС (Операционная система) - программное обеспечение, управляющее компьютерами (включая микроконтроллеры) и позволяющее запускать на них прикладные программы. Предоставляет программный интерфейс для взаимодействия с компьютером, управляет прикладными программами и занимается распределением предоставляемых ресурсов, в том числе между прикладными программами.
4. NAT-сеть (Network Address Translation) — это технология, которая позволяет переводить IP-адреса между локальной сетью и внешней сетью.
5. SSH (Secure Shell) — сетевой протокол прикладного уровня, позволяющий производить удалённое управление операционной системой и туннелирование TCP-соединений (например, для передачи файлов).
6. 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.
7. OpenMPI (Open Message Passing Interface) — это высокопроизводительная реализация Message Passing Interface (MPI), которая широко используется для написания параллельных программ, работающих в распределенной вычислительной среде.
8. Дистрибутив - форма распространения программного обеспечения, обычно содержащая программу-установщик (для выбора режимов и параметров установки) и набор файлов, содержащих отдельные части программного средства.
9. DHCP (Dynamic Host Configuration Protocol) - это протокол динамической настройки сетевых параметров устройства. DHCP позволяет автоматически назначать IP-адреса, шлюзы, серверы DNS и другие параметры сети устройствам в локальной сети.
10. IP - это протокол сетевого уровня, который определяет формат и адресацию пакетов данных, передаваемых в компьютерных сетях.
11. IP-адрес — уникальный числовой идентификатор устройства в компьютерной сети, работающей по протоколу IP.

12. DNS - это система, которая преобразует доменные имена в IP-адреса и обратно.

13. Порт - это числовой идентификатор программы или процесса, который обслуживает сетевые соединения на заданном IP-адресе.

Введение

В отчете представлено описание выполнения практической работы по дисциплине «Научно-исследовательская работа». Работа была выполнена на основе лабораторной работы №0 по дисциплине «Теория графов», в ходе её выполнения было необходимо ознакомиться с работой среды виртуализации VirtualBox, с помощью которой были созданы виртуальные машины и симулирована работа простейшего суперкомпьютера с несколькими узлами. Также для выполнения работы было необходимо установить дистрибутив Debian операционной системы Linux, настроить сетевое взаимодействие и изучить работу технологий OpenMP и OpenMPI.

1 Постановка задачи

Задача: ознакомиться с технологиями параллельного программирования OpenMp и OpenMPI на основе выбранного дистрибутива операционной системы Linux в среде аиртуализации Oracle VM VirtualBox.

Выбранный вариант работы: дистрибутив - Debian, сеть - 172.27.128.192/26.

Выполнение работы можно разделить на подзадачи:

1. Установка среды виртуализации Oracle VM VirtualBox.
2. Установка дистрибутива Debian.
3. Создание и настройка четырех виртуальных машин.
4. Настройка локальной сети и подключение по SSH между виртуальными машинами.
5. Установка и настройка ПО для работы с OpenMP и OpenMPI на виртуальных машинах.
6. Изменение ранее написанной программы для параллельной работы.
7. Запуск программы на виртуальных машинах.

2 Подготовка среды виртуализации

2.1 Установка и настройка среды виртуализации

Установить VirtualBox можно с официального сайта. Далее нужно настроить сеть для системы. Для этого в VirtualBox в меню инструментов выбираем «Сеть», переходим в «Сети NAT» и создаем новую сеть, в соответствии с вариантом указываем в IPv4 префикс 172.27.128.192/26 и выключаем DHCP.

2.2 Создание виртуальной машины

1. Загружаем образ диска ОС Debian с официального сайта.
2. В меню VirtualBox выбираем «Создать».
3. Выбираем имя виртуальной машины, указываем тип ОС Linux и версию Debian(64-бит).
4. Выделяем для машины 2048 МБ оперативной памяти, 2 ЦП и 30 ГБ на виртуальный жесткий диск.
5. На вкладке «Накопители» добавить загруженный образ диска Debian 12 ISO в оптический диск виртуальной машины.
6. После создания виртуальной машины заходим в её настройки и в меню «Сеть» выбираем ранее мозданную сеть NAT.

2.3 Установка дистрибутива на виртуальные машины и его настройка

1. Запускаем машину и устанавливаем саму ОС, выбирая обычный установщик.
2. При настройках языка, местонахождения выбираем то, что удобней при работе, в данной работе был выбран русский язык и РФ.
3. Настраиваем сеть: выбираем адрес машины, в данном варианте сеть 172.27.128.192/26, адрес 172.27.128.193 используется в качестве шлюза, а 172.27.128.194 - следующий за шлюзом и его лучше не выбирать, поэтому адреса для машин будут начинаться с 172.27.128.195, у следующих машин адреса будут больше
4. Выбираем имя машины в сети и какой-либо домен, в данной работе в качестве имен машин были выбраны debian1, debian2, debian3 и debian4, а в качестве домена - hpc.
5. Создаем суперпользователя root, пароль можно не создавать, просто для администрирования системы необходимо ввести команду sudo.

6. Создаем обычного пользователя и задаем пароль, для удобства на всех машинах лучше установить одинакового пользователя и пароль.

7. В следующих пунктах выбираем стандартные настройки.

8. При выборе ПО оставляем только «Стандартные утилиты» и «SSH-сервер».

2.4 Настройка hosts

Настройка разрешения имён узлов виртуальной сети необходима для возможности обращения к ним по именам, а не по IP-адресам. Для этого в файле /etc/hosts были добавлены соответствия имен и адресов, данный файл был открыт при помощи команды sudo nano /etc/hoats. Проверить связь между узлами можно проверить при помощи команды ping имя_узла.

2.5 Настройка SSH соединения

1. Проверяем статус при помощи команды sudo systemctl status sshd и, если она не запущена, запустим её при помощи команды sudo systemctl start sshd.

2. Генерируем пару ключей SSH на машине при помощи команды ssh-keygen -t rsa.

3. Загружаем открытый ключ на удаленный сервер при помощи команды ssh-copy-id имя_пользователя@и
если пользователь везде один, то достаточно указать имя узла.

4. Проверяем, что мы можем подключиться к другой виртуальной машине без ввода пароля: ssh имя_узла.

2.6 Найстройка буфера обмена с внешней системой

Для возможности отправления файлов из основной ОС на виртуальные машины настроим проброс портов NAT-сети. Для этого переходим в раздел, где создавали нашу сеть NAT, там выбираем «Проброс портов». «Имя» указываем любое, «протокол» - TCP, «адрес хоста» - 127.0.0.1, это стандартный локальный адрес, в «порте хоста» указываем значение больше 4096, так как до него идут привелегированные хосты, в работе были выбраны 40195, 40196, 40197 и 40198, «адрес гостя» указываем в соответствии с IP-адресом виртуальной машины, «порт гостя» - 22.

Для отправления файла с основной ОС на виртуальную машину в консоли используем команду: scp -P порт_хоста пусть_к_файлу имя_пользователя@localhost:/tmp/

2.7 Настройка OpenMP

OpenMP представляет собой простой и эффективный способ добавления параллелизма в программы. Некоторые из ключевых особенностей OpenMP:

1. Простота использования: OpenMP позволяет добавлять параллельные возможности в программы с минимальной модификацией исходного кода благодаря простым директивам препроцессора.

2. Масштабируемость: OpenMP поддерживает создание программ, способных использовать все ядра процессора, обеспечивая улучшенную производительность при увеличении количества ядер.

3. Гибкость: OpenMP предоставляет различные опции контроля параллелизма, такие как распределение задач между потоками, управление синхронизацией и работу с разделяемой памятью.

4. Улучшенная производительность: параллельные программы, написанные с использованием OpenMP, могут реализовывать многопоточность для более быстрого выполнения вычислительных задач.

Для установки необходимых пакетов используется команда sudo apt install libomp-dev. Для изменения количества потоков используется команда export OMP_NUM_THREADS=n, где n-количество потоков. Для компиляции программы используется следуюая команда g++ file.cpp -o file -fopenmp. Для запуска программы используется команда ./file.

Набор средств:

1. Директивы компилятора (например, pragma omp...).

2. Библиотечные подпрограммы (например, get_num_threads()).

3. Переменные окружения (например, OMP_NUM_THREADS).

2.8 Настройка OpenMPI

Особенности OpenMPI:

1. Сообщения: OpenMPI предоставляет мощный набор функций для обмена сообщениями между процессами, что позволяет программам взаимодействовать и совместно решать задачи.

2. Масштабируемость: OpenMPI предназначен для работы на распределенных вычислительных кластерах с большим количеством узлов, что позволяет эффективно использовать ресурсы системы при увеличении числа процессов.

3. Гибкость: OpenMPI поддерживает различные режимы коммуникации (точка-точка, сборка данных, широковещательная рассылка и другие), а также обладает настройками для оптимизации производительности и контроля ресурсов.

Для установки необходимых пакетов используется команда sudo apt install liopenmpi3 libopenmpi-dev. Для компиляции программы используется следуюая команда mpic++ file.cpp -o file. Для запуска программы её сначала необходимо скомпилировать на всех узлах, а далее используется команда mpirun -host debian1:n,debian2 file, n-количество процессов, запущенных на узле, по умолчанию n = 1.

Основные функции и структуры:

1. MPI_Init(&argc, &argv) - процедура инициализации MPI.
2. MPI_Finalize(void) - процедура завершения использования MPI. Предназначена для корректного завершения всех параллельных операций и освобождения ресурсов, связанных с MPI.
3. MPI_Comm_size(MPI_Comm comm, intsize) - функция получения количества потоков, обрабатывающих программу.
4. MPI_Comm_rank(MPI_Comm comm, int* rank) - функция получения текущего процесса, исполняемого часть программы.
5. MPI_Status - это структура, используемая для хранения информации о полученном сообщении. С ее помощью можно получить идентификатор источника сообщения, тег и код ошибки.
6. MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int msgtag, MPI_Comm comm) - осуществляет отправку сообщения.
   (a) buf - указатель на буфер данных для отправки.
   (b) count - количество элементов, которые следует отправить.
   (c) datatype - тип данных элементов, которые отправляются.
   (d) dest - ранг (идентификатор) процесса-получателя.
   (e) msgtag - метка сообщения, которая используется для идентификации сообщения.
   (f) comm - коммуникатор, определяющий группу процессов, между которыми отправляется сообщение.
7. MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status) - блокирует выполнение до получения данных от указанного источника. Она заполняет буфер данными из полученного сообщения.
   (a) buf - указатель на буфер, в который будет сохранено полученное сообщение.
   (b) count - количество элементов, которые ожидаются к получению.

(c) datatype - тип данных элементов, которые будут приняты.

(d) source - ранг (ID) процесса, отправляющего сообщение.

(e) tag - метка сообщения, которое ожидается.

(f) comm - коммуникатор, определяющий группу процессов, участвующих в обмене.

(g) status - указатель на структуру статуса, которая содержит информацию о приеме (может быть NULL, если эта информация не нужна).

8. MPI_Get_count(MPI_Status* status, MPI_Datatype datatype, int* count) - функция, определяющая количество элементов пришедших данных определенного типа и записывающая это значение по адресу count.

   (a) status - указатель на структуру MPI_Status, содержащую информацию о сообщении.

   (b) datatype - тип данных элементов, которые были получены.

   (c) count - указатель на переменную, в которую будет сохранено количество полученных элементов данного типа данных.

9. MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Statusstatus) - позволяет проверить наличие входящих сообщений от конкретного процесса с определенным тегом в коммуникаторе, не принимая само сообщение.

   (a) source - индекс процесса-отправителя сообщения.

   (b) tag - метка, связанная с сообщением.

   (c) comm - коммуникатор MPI, через который ожидается приход сообщения.

   (d) status - указатель на структуру MPI_Status, в которую будет сохранена информация о проверяемом сообщении.

3 Реализация параллельной программы

Работа была реализована на основе лабораторной работы №0 по дисциплине «Теория графов», в которой было необходимо сгенерировать файл со строкой в 10000 символов, закодировать его при помощи алгоритмов RLE и LZW и после декодировать.

3.1 Реализация OpenMP

3.1.1 Генерация файла

В функции для генерации файла сначала создаеся вектор, который содержит в себе столько же элементов, сколько и потоков. Далее при помощи функции parts_size заполняем вектор значениями, которые равны длине строки, которую должен будет сгенерировать соответствующий поток. Далее в параллельном цикле for каждый поток генерирует указанное в векторе parts количество символов. После того как поток сгенерировал строку, она записывается в файл.

Реализацию данной функции см. приложение А.1.

3.1.2 Кодирование и декодирование LZW

Функция LZW_coder_parallel, реализующая параллельное кодирование LZW, реализована следующим образом: создается вектор, который содержит части строк для каждого потока, данные части определяются при помощи функции encode_parts, далее в параллельном цикле for каждый поток кодирует свою строку и помещает её в вектор с закодированными строками. После того, как все потоки закодировали свои части строки, в векторе my_cnts запоминаются длины закодированных строк, а сами эти строки записываются в файл.

Функция LZW_decoder_parallel, реализующая параллельное декодирование LZW, реализована следующим образом: создается вектор, который содержит части закодированных строк для каждого потока, данные части определяются при помощи функции decode_parts, далее в параллельном цикле for каждый поток декодирует свою строку и помещает её в вектор с декодированными строками. После того, как все потоки декодировали свои части строки, эти части записываются в файл.

Реализацию данных функций см. приложение А.1.

3.1.3 Кодирование и декодирование RLE

Функция RLE_coder_parallel, реализующая параллельное кодирование RLE, работает точно по тому же принципу что и параллельное кодирование LZW.

Функция RLE_decoder_parallel, реализующая параллельное декодирование RLE, работает точно по тому же принципу что и параллельное декодирование LZW.

Реализацию данных функций см. приложение А.1.

3.1.4 Сравнение изначального и декодированного файлов

В функции для сравнения изначального и декодированного файлов создаются 2 вектора, содержащие части файлов, которые были получения при помощи функции encode_parts. В параллельном цикле for каждый поток посимвольно сравнивает соответствующие части по одинаковому индексу из векторов. Если при проверке хотя бы один поток найдет несоответствие, то выведется сообщение о наличии ошибки.

Реализацию данной функции см. приложение А.1.

3.2 Реализация OpenMPI

3.2.1 Генерация файла

Функция параллельной генерации строки для OpenMPI реализована похожим образом, что и для OpenMP, только теперь все сгенерированные части строки отправляются в нулевой процесс, который записывает их в файл.

Реализацию данной функции см. приложение А.2.

3.2.2 Кодирование и декодирование LZW

Функция LZW_coder_parallel, реализующая параллельное кодирование LZW, работает следующим образом: создается вектор, который содержит части строк для каждого потока, данные части определяются при помощи функции encode_parts в нулевом потоке, а также нулевой поток рассылает части сообщения остальным потокам, далее каждый процесс кодирует свою часть строки. После в нулевом процессе происходит сбор закодированных частей и их сохранение в файл.

Функция LZW_decoder_parallel, реализующая параллельное декодирование LZW, работает следующим образом: создается вектор, который содержит части закодированных строк для каждого потока, данные части определяются при помощи функции decode_parts в нулевом потоке, а также нулевой поток рассылает части сообщения остальным потокам, далее каждый поток декодирует свою строку. После в нулевом процессе происходит сбор декодированных частей и их сохранение в файл.

Реализацию данных функций см. приложение А.2.

3.2.3 Кодирование и декодирование RLE

Функция RLE_coder_parallel, реализующая параллельное кодирование RLE, работает точно по тому же принципу что и параллельное кодирование LZW.

Функция RLE_decoder_parallel, реализующая параллельное декодирование RLE, работает точно по тому же принципу что и параллельное декодирование LZW.

Реализацию данных функций см. приложение А.2.

4 Результаты работы программы

При запуске программы выводится результат сравнения заколированного и декодированного файлов. На рис.1 - рис.3 продемонстрирован результат работы программы для OpenMP, а на рис.4 и рис.5 продемнострирован результат работы программы для OpenMPI.

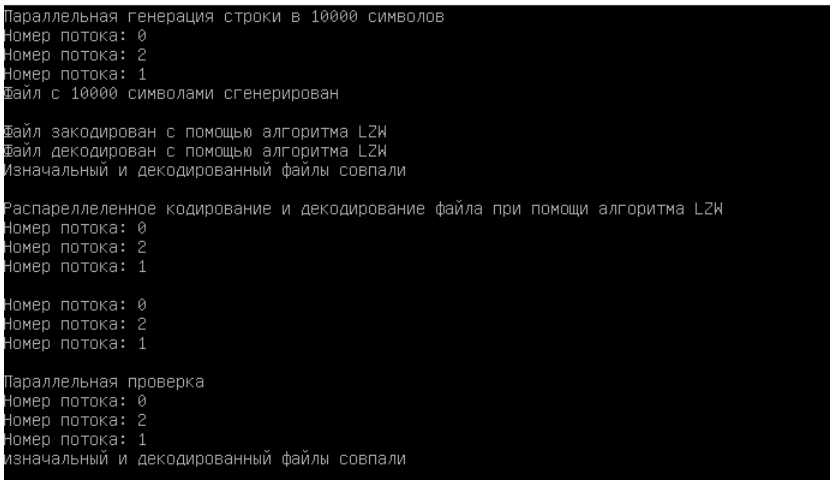

Рис. 1. Вывод OpenMP: параллельная генерация файла и кодирование-декодирование LZW

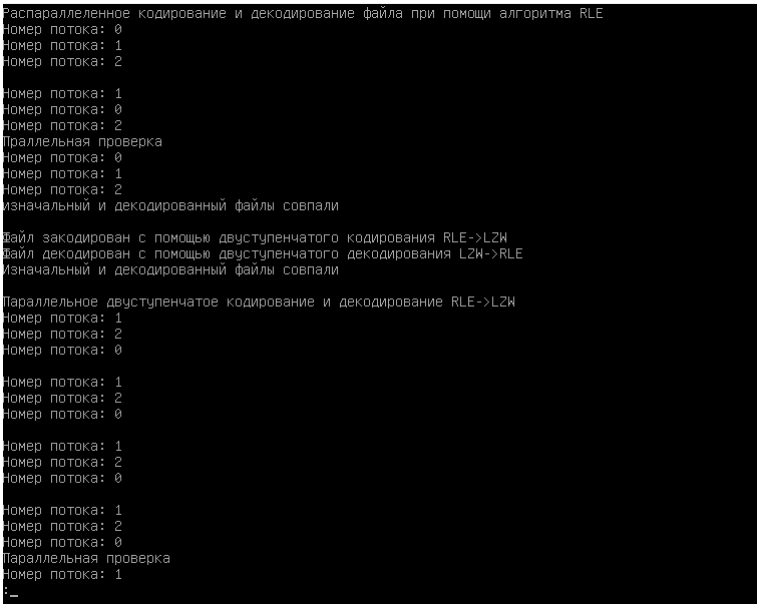

Рис. 2. Вывод OpenMP: кодирование-декодирование RLE и двуступенчатое кодирование-декодирование RLE_LZW

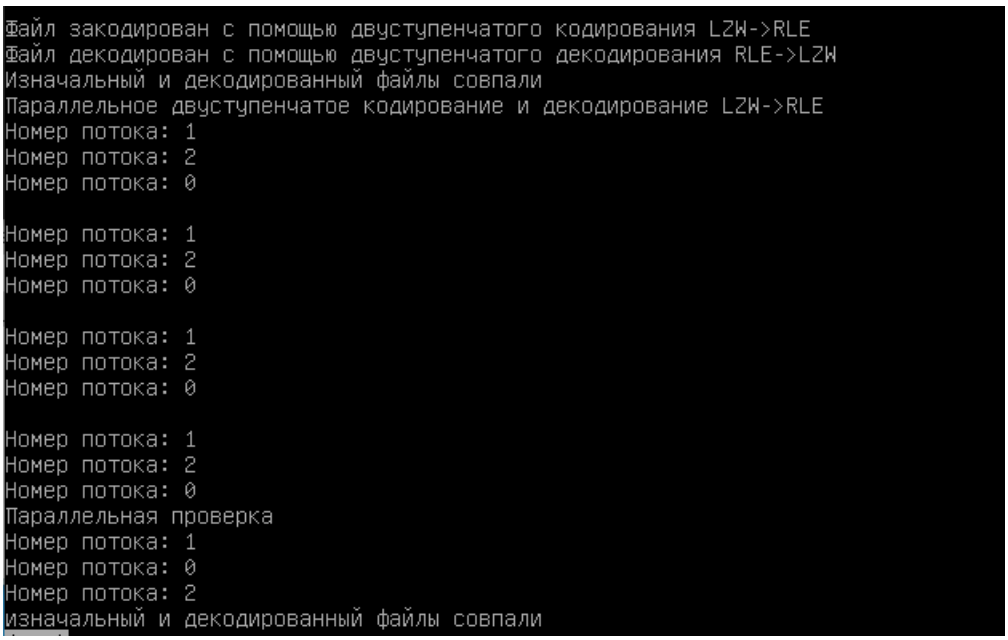

Рис. 3. Вывод OpenMP: двуступенчатое кодирование-декодирование LZW_RLE

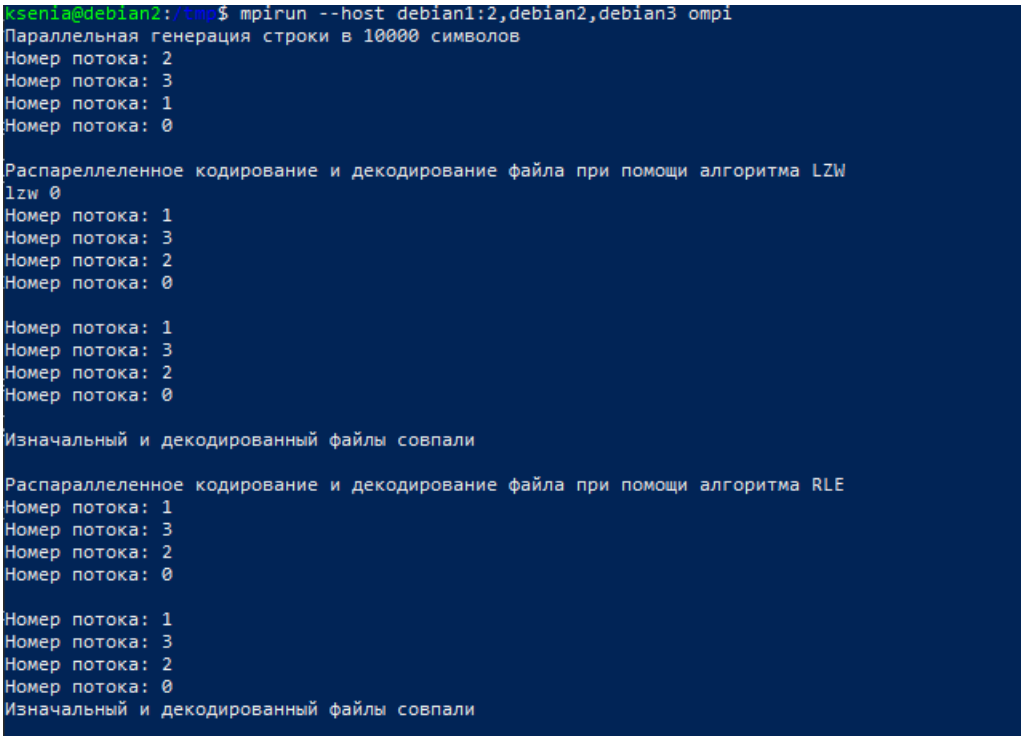

Рис. 4. Вывод OpenMPI: параллельная генерация файла и кодирование-декодирование LZW и RLE

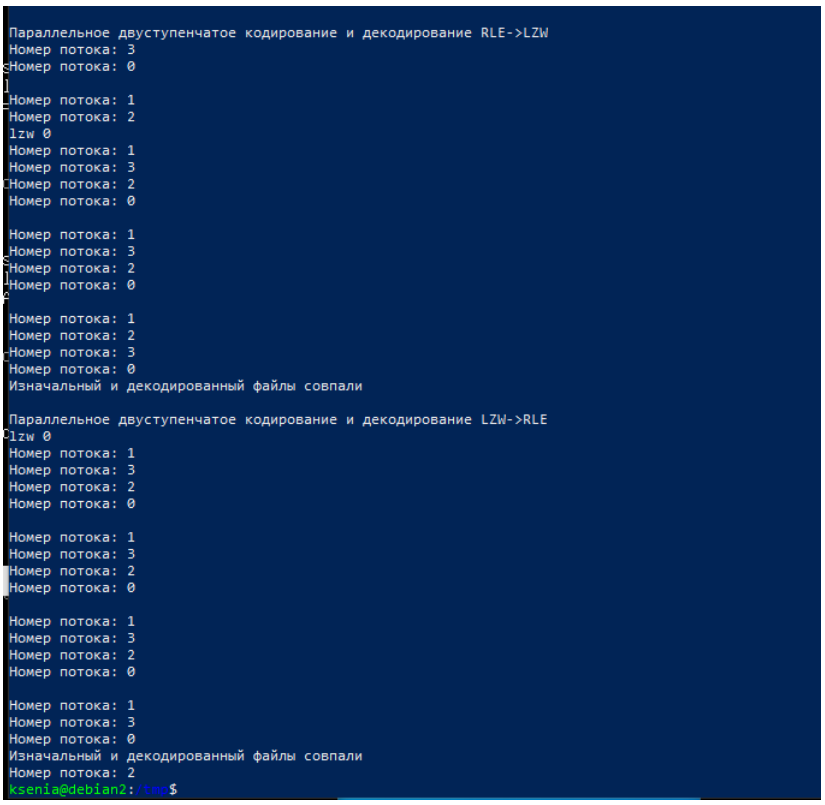

Рис. 5. Вывод OpenMPI: двуступенчатое кодирование-декодирование RLE_LZW и LZW_RLE

5 Заключение

В ходе выполнения практической работы были получены следующие навыки работы в среде виртуализации VirtualBox: создание и настройка виртуальной машины, настройка локальной сети, настройка сетевого взаимодействия, добавление файла с основной ОС на виртуальную машину, помимо этого были изучены работа дистрибутива Debian и работа параллельного программирования на основе технологий OpenMP и OpenMPI.

Корректность программы для OpenMP была проверена на разных количествах потоков, максимальное количество было равно 30, на большем количестве потоков программа не проверялась, так как это является нецелесообразным из-за длины строки, которую необходимо закодировать, потому что чем больше потоков, тем меньше длина строки для отдельного потока. При всех рассмотренных количествах потоков программа работала корректно.

Корректность программы для OpenMPI так же была проверена на разных количествах процессов, было рассмотренно несколько случаев: когда программа была запущена на 1 узле с несколькими проуессами, когда программа была запущена на нескольких узлах с одинаковым количеством процессов и когда программа была запущена на нескольких узлах с разным количеством процессов. Во всех рассмотренных случаях программа работала корректно.

6 Список литературы

1. «VirtualBox: Установка и настройка (для новичков). Создание виртуальной машины», https://youtu.be/j1FAZ0bUEvs?si=lQGhQKOTULifqMLn

   (дата обращения 18.06.2024)

2. «Как настроить SSH вход», https://wiki.merionet.ru/articles/kak-nastroit-ssh-vxod-bez-paroly

   (дата обращения 18.06.2024)

3. «OpenMP», https://www.openmp.org/

   (дата обращения 20.06.2024)

4. «OpenMPI:Open Source High Performance Computing», https://www.open-mpi.org/ (дата обращения 23.06.2024)

5. «Параллельное программирование: OpenMP», https://youtu.be/eAqHlX9TvCI?si=E-cP9Oth_bqyT8FK

   (дата обращения 20.06.2024)

6. «Параллельные процессы MPI», https://youtu.be/owkUOXZF-S4?si=h9zQEsL4ZFQtUAT5 (дата обращения 23.06.2024)

A.1 Реализация распараллеленной программы для OpenMP

```
void gen_File(vector<string> init_dict, int tid) {
    vector<int> parts;
    parts_size(tid, file_size, parts);
    ofstream file;
    file.open(File);
    if (file.is_open())
    {
        #pragma omp parallel for
        for (int z = 0; z < tid; z++) {
            srand(time(NULL));
            string str;
            for (int i = 0; i < parts[z]; i++) {
                str += alph[rand() % 25];
            }
            file << str;
            cout << "Номер потока: " << omp_get_thread_num() << endl;
        }
    }
    file.close();
}
```

```
void LZW_coder_parallel(vector<string> init_dict, int numb, string& fin1, string& fout1,
vector<int>& my_cnts) {
    vector<string> parts = encode_parts(fin1, numb);
    vector<string> coded(numb, "");
    my_cnts.clear();
    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        coded[tid] = LZW_code(init_dict, parts[tid]);
        cout << "Номер потока: " << tid << endl;
    }
```

```
    ofstream fout;
    fout.open(fout1);
    for (const auto& code : coded) {
        my_cnts.push_back(code.size());
    }

    for (const auto& code : coded) {
        fout << code;
    }


    void LZW_decoder_parallel(vector<string> init_dict, string& fin1, string& fout1,
    vector<int>& my_cnts) {
    vector<string> parts = decode_parts(fin1, my_cnts);
    vector<string> decode(parts.size(), "");

    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < parts.size(); i++) {
            decode[i] = LZW_decode(init_dict, parts[i]);
            cout << "Номер потока: " << omp_get_thread_num() << endl;
        }
    }

    ofstream fout;
    fout.open(fout1);
    for (const auto& part : decode) {
        fout << part;
    }
}
}


void RLE_coder_parallel(string& w, int numb, string& outt, vector<int>& my_cnts) {
```

```
    vector<string> parts = encode_parts(w, numb);
    vector<string> res(numb, "");
    my_cnts.clear();

    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < numb; i++) {
            res[i] = RLE_code(parts[i]);
            cout << "Номер потока: " << omp_get_thread_num() << endl;
        }
    }
    ofstream fout(outt);
    for (const auto& ress : res) {
        my_cnts.push_back(ress.size());
    }
    for (const auto& ress : res) {
        fout << ress;
    }
}


void RLE_decoder_parallel(string& fin, string& fout, vector<int>& my_cnts) {
    vector<string> parts = decode_parts(fin, my_cnts);
    vector<string> decoded(parts.size(), "");
#pragma omp parallel
    {
#pragma omp for
        for (int i = 0; i < parts.size(); i++) {
            decoded[i] = RLE_decode(parts[i]);
            cout << "Номер потока: " << omp_get_thread_num() << endl;
        }
    }

    ofstream fout1;
    fout1.open(fout);
```

```
    for (const auto& decode : decoded) {
        fout1 << decode;
    }
}

void check_parallel(string& file1, string& file2, int tid) {
    vector<string> str1 = encode_parts(file1, tid);
    vector<string> str2 = encode_parts(file2, tid);

    bool flag = true;
    #pragma omp parallel for
    for (int i = 0; i < tid; i++) {
        if (check_part(str1[i], str2[i]) == false) {
            flag = false;
            cout << "error" << endl;
            //  break;
        }
        cout << "Номер потока: " << omp_get_thread_num() << endl;
    }

    if (flag == true) {
        cout << "изначальный и декодированный файлы совпали" << endl;
    }
}
```

A.2 Реализация распараллеленной программы для OpenMPI

```
void gen_File(vector<string> init_dict) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int alphabet_size = init_dict.size();
    int local_size = file_size / size;
    string local_buffer;

    if (rank == 0) {
        local_size += file_size % size;
    }

    srand(time(NULL) + rank);
    for (int i = 0; i < local_size; ++i) {
        local_buffer += init_dict[rand() % alphabet_size];
    }
    if (rank == 0) {
        vector<string> generated_parts(size);
        generated_parts[0] = std::move(local_buffer);
        for (int i = 1; i < size; i++) {
            MPI_Status status;
            MPI_Probe(i, 0, MPI_COMM_WORLD, &status);
            int count;
            MPI_Get_count(&status, MPI_CHAR, &count);
            generated_parts[i].resize(count);
            MPI_Recv(&generated_parts[i][0], count, MPI_CHAR, i, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        }

        ofstream file(File);
        for (const auto& gp : generated_parts) {
            file << gp;
        }
```

```
        file.close();
    }

    else {
        MPI_Send(local_buffer.data(), local_buffer.size(), MPI_CHAR, 0, 0,
        MPI_COMM_WORLD);
    }
    printf("Номер потока: %d\n", rank);
}


LZW_coder_parallel(vector<string> init_dict, string& fin_name, string& fout_name,
vector<int>& my_cnts) {
    my_cnts.clear();
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);//
    MPI_Comm_size(MPI_COMM_WORLD, &size);//

    vector<string> parts;//
    string part;//
    string coded_part;//

    if (rank == 0) {
        parts = encode_parts(fin_name, size);

        for (int i = 1; i < size; i++) {
            MPI_Send(parts[i].data(), parts[i].size(), MPI_CHAR, i, 0, MPI_COMM_WORLD);
        }
        part = parts[0];  // Часть для процесса 0
    }

    else {
        MPI_Status status;
        MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
        int count;
        MPI_Get_count(&status, MPI_CHAR, &count);
```

```
        part.resize(count);
        MPI_Recv(&part[0], count, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    if (rank == 0)  cout << "lzw " << rank << endl;
    coded_part = LZW_code(init_dict, part);

    if (rank == 0) {
        vector<string> coded_parts(size);
        coded_parts[0] = std::move(coded_part);
        for (int i = 1; i < size; i++) {
            MPI_Status status;
            MPI_Probe(i, 0, MPI_COMM_WORLD, &status);
            int count;
            MPI_Get_count(&status, MPI_CHAR, &count);
            coded_parts[i].resize(count);
            MPI_Recv(&coded_parts[i][0], count, MPI_CHAR, i, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        }

        ofstream fout(fout_name);
        for (const auto& cp : coded_parts) {
            my_cnts.push_back(cp.size());
        }
        for (const auto& cp : coded_parts) {
            fout << cp;
        }
        fout.close();
        my_cnts[2] << endl;
    }

    else {
        MPI_Send(coded_part.data(), coded_part.size(), MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }
    printf("Номер потока: %d\n", rank);
}
```

```
void LZW_decoder_parallel(vector<string> init_dict, string& fin_name,
string& fout_name, vector<int>& my_cnts) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    vector<string> parts;
    string part;
    string decoded_part;

    if (rank == 0) {
        //std::vector<std::string> parts = split_file_decoder(fin_name);
        parts = decode_parts(fin_name, my_cnts);

        for (int i = 1; i < size; i++) {
            MPI_Send(parts[i].data(), parts[i].size(), MPI_CHAR, i, 0, MPI_COMM_WORLD);
        }
        part = parts[0];  // Часть для процесса 0
    }

    else {
        MPI_Status status;
        MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
        int count;
        MPI_Get_count(&status, MPI_CHAR, &count);
        part.resize(count);
        MPI_Recv(&part[0], count, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    decoded_part = LZW_decode(init_dict, part);

    if (rank == 0) {
        vector<string> decoded_parts(size);
        decoded_parts[0] = std::move(decoded_part);
        for (int i = 1; i < size; i++) {
```

```
            MPI_Status status;
            MPI_Probe(i, 0, MPI_COMM_WORLD, &status);
            int count;
            MPI_Get_count(&status, MPI_CHAR, &count);
            decoded_parts[i].resize(count);
            MPI_Recv(&decoded_parts[i][0], count, MPI_CHAR, i, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        }

        std::ofstream fout(fout_name, std::ios::binary);
        for (const auto& dp : decoded_parts) {
            fout << dp;
        }
        fout.close();
    }

    else {
        MPI_Send(decoded_part.data(), decoded_part.size(), MPI_CHAR, 0, 0,
        MPI_COMM_WORLD);
    }


    printf("Номер потока: %d\n", rank);
}


void RLE_coder_parallel(string& fin_name, string& fout_name, vector<int>& my_cnts) {
    my_cnts.clear();
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);//
    MPI_Comm_size(MPI_COMM_WORLD, &size);//

    vector<string> parts;
    string part;//
    string coded_part;
```

```
if (rank == 0) {
    parts = encode_parts(fin_name, size);
    for (int i = 1; i < size; i++) {
        MPI_Send(parts[i].data(), parts[i].size(), MPI_CHAR, i, 0, MPI_COMM_WORLD);
    }
    part = parts[0];
}

else {
    MPI_Status status;
    int count;
    MPI_Get_count(&status, MPI_CHAR, &count);
    part.resize(count);
    MPI_Recv(&part[0], count, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

coded_part = RLE_code(part);

if (rank == 0) {
    vector<string> coded_parts(size);
    coded_parts[0] = std::move(coded_part);
    for (int i = 1; i < size; i++) {
        MPI_Status status;
        MPI_Probe(i, 0, MPI_COMM_WORLD, &status);
        int count;
        MPI_Get_count(&status, MPI_CHAR, &count);
        coded_parts[i].resize(count);
        MPI_Recv(&coded_parts[i][0], count, MPI_CHAR, i, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
    }

    ofstream fout(fout_name);
    for (const auto& cp : coded_parts) {
        my_cnts.push_back(cp.size());
    }
```

```
        for (const auto& cp : coded_parts) {
            fout << cp;
        }
        fout.close();
    }

    else {
        MPI_Send(coded_part.data(), coded_part.size(), MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }

    printf("Номер потока: %d\n", rank);

}



void RLE_decoder_parallel(string& fin_name, string& fout_name, vector<int>& my_cnts) {
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    vector<string> parts;
    string part;
    string decoded_part;

    if (rank == 0) {
        parts = decode_parts(fin_name, my_cnts);

        for (int i = 1; i < size; i++) {
            MPI_Send(parts[i].data(), parts[i].size(), MPI_CHAR, i, 0, MPI_COMM_WORLD);
        }
        part = parts[0];
    }

    else {
        MPI_Status status;
```

```
        MPI_Probe(0, 0, MPI_COMM_WORLD, &status);
        int count;
        MPI_Get_count(&status, MPI_CHAR, &count);
        part.resize(count);
        MPI_Recv(&part[0], count, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    decoded_part = RLE_decode(part);

    if (rank == 0) {
        vector<string> decoded_parts(size);
        decoded_parts[0] = std::move(decoded_part);
        for (int i = 1; i < size; i++) {
            MPI_Status status;
            MPI_Probe(i, 0, MPI_COMM_WORLD, &status);
            int count;
            MPI_Get_count(&status, MPI_CHAR, &count);
            decoded_parts[i].resize(count);
            MPI_Recv(&decoded_parts[i][0], count, MPI_CHAR, i, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        }

        ofstream fout(fout_name, std::ios::binary);
        for (const auto& dp : decoded_parts) {
            fout << dp;
        }
        fout.close();
    }

    else {
        MPI_Send(decoded_part.data(), decoded_part.size(), MPI_CHAR, 0, 0,
        MPI_COMM_WORLD);
    }
    printf("Номер потока: %d\n", rank);
}
```