ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ "МИФИ"»

Институт Интеллектуальных Кибернетических Систем
Кафедра №42 "Криптология и кибербезопасность"
Дисциплина «Параллельное программирование»

Отчет к лабораторной работе № 5
«Коллективные операции в MPI»

Выполнил:
студент группы Б22-505
Титов Дмитрий Иванович

Принял:
Куприяшин Михаил Андреевич

Москва
2024 год

Цель работы

Приобрести навыки применения коллективных операций при разработке параллельных программ на основе технологии MPI.

1. Описание системы

Процессор - amd Ryzen 7 7840HS (2023) 8 ядер и 16 потоков
Оперативная память - 16GB DDR5
ОС - Ubuntu 24.04.1 LTS 64-бит
Среда разработки - VS Code, CMake 3.28.3, GCC 13.2.0
Версия OpenMP - 4.5 201511
mpirun (Open MPI) 4.1.6
Ноутбук был на зарядке

2. Выбор операций коллективного обмена

В представленном коде для параллельной сортировки используются механизмы коллективного обмена данными в MPI: **MPI_Scatter** и **MPI_Gather**. Эти операции позволяют эффективно распределить данные между процессами и собрать результаты обратно в процесс с рангом 0.

Описание коллективных операций:

- **MPI_Scatter** — разделяет данные на равные части и распределяет их среди всех процессов. В данном случае массив делится на несколько частей, которые обрабатываются разными процессами.
- **MPI_Gather** — собирает данные с различных процессов и объединяет их в массив в процессе с рангом 0.

Плюсы использования этих операций:

1. **Эффективность распределения данных:**
    - **MPI_Scatter** быстро и эффективно распределяет данные между всеми процессами. Каждый процесс получает ровно ту часть данных, которую он должен обрабатывать. Это позволяет избежать лишних копий данных, уменьшая накладные расходы на память и улучшая производительность.
2. **Легкость в использовании:**
    - Операции **MPI_Scatter** и **MPI_Gather** являются стандартными и обеспечивают упрощенный способ обмена данными между процессами. Это повышает читаемость и упрощает отладку кода, так как большинство трудностей уже решены в реализации MPI. Эти операции универсальны и могут использоваться для различных типов данных и алгоритмов.
3. **Параллелизм:**
    - Каждый процесс работает над своей частью данных, и операция сортировки выполняется параллельно, что значительно ускоряет выполнение программы по сравнению с последовательным методом.
4. **Меньше накладных расходов на синхронизацию:**
    - В данном примере синхронизация между процессами выполняется через операции обмена данными (распределение и сбор), что обычно быстрее, чем другие способы синхронизации.

Минусы использования этих операций:

1. **Проблемы с балансировкой нагрузки:**
    - Если размер массива не делится нацело на количество процессов, это может привести к неравномерному распределению данных, что ухудшит производительность. Например, при небольшом количестве данных или большом числе процессов, одна из частей может содержать значительно меньше данных, чем другая.

2. **Ограничения на количество процессов:**
    - Количество процессов должно быть делителем размера массива, иначе последние процессы могут не получить данных, что требует дополнительных проверок и компенсаций.
3. **Множество операций сбора и сортировки:**
    - После того как каждый процесс отсортировал свою часть данных, они должны быть собраны и объединены с помощью **MPI_Gather**. Это требует дополнительных операций на уровне процесса с рангом 0, и такие шаги могут занимать много времени, особенно на больших объемах данных. Несколько этапов слияния данных могут стать узким местом, особенно если процесс слияния не оптимизирован.

Сравнение с другими методами коллективного обмена данными:

- **MPI_Bcast** (широковещательная передача данных):
    - Эта операция полезна, когда нужно передать одну и ту же информацию всем процессам. Однако она не подходит для распределенной сортировки, так как она не предполагает разделение данных.
- **MPI_Allgather** (сбор всех данных у всех процессов):
    - В отличие от **MPI_Gather**, операция **MPI_Allgather** собирает все части данных на каждом процессе. Это может быть полезно для получения всех данных на каждом процессе, но увеличивает нагрузку на сеть, так как каждый процесс получает все данные. Это неэффективно для задачи сортировки, так как каждый процесс должен работать только со своей частью массива.
- **MPI_Reduce** и **MPI_Allreduce**:
    - Эти операции используются для агрегации данных (например, суммирования, нахождения минимума или максимума). Однако для сортировки данные нужно просто собрать и организовать правильно, а не агрегировать, поэтому такие операции здесь не подходят.

Вывод:

Использование **MPI_Scatter** и **MPI_Gather** — это вполне оптимальный и стандартный подход для распределенной сортировки, где данные должны быть разделены между процессами, а затем собраны обратно для дальнейшей обработки. Этот метод позволяет эффективно использовать ресурсы и поддерживает параллельную обработку данных. Однако при больших объемах данных важно оптимизировать операции слияния и минимизировать накладные расходы на обмен информацией между процессами.

3. Сравнение реализации параллеливания кодов

В данной лабораторной работе представлены два подхода для параллельной сортировки массива: использование OpenMP и MPI. Оба метода направлены на ускорение выполнения сортировки за счет параллельной обработки данных, однако их реализации значительно отличаются, что влияет на их производительность и области применения.

3.1. Параллельная сортировка с OpenMP

В первой реализации используется библиотека OpenMP для параллельного выполнения сортировки методом Шелла. Основная идея заключается в том, чтобы распараллелить внешнюю итерацию, при этом каждый поток обрабатывает часть массива с учетом определенного шага (gap).

Особенности реализации:

- Вся сортировка выполняется на одном процессе, но с использованием нескольких потоков.
- Каждый поток обрабатывает свою часть массива, и синхронизация между потоками осуществляется через директиву #pragma omp parallel for.
- Эффективность достигается благодаря параллельной обработке разных частей массива, что ускоряет выполнение на многоядерных процессорах.

Преимущества:

- Простой и быстрый способ параллелизации, который хорошо подходит для многозадачных сред, таких как многоядерные процессоры.
- Легкость в использовании — достаточно добавить директиву OpenMP и указать количество потоков.
- Подходит для использования на одном сервере или в рабочей станции с многими ядрами.

Недостатки:

- Требует, чтобы все данные находились в одном процессе, что ограничивает масштабируемость при увеличении размера данных.
- На больших объемах данных можно столкнуться с проблемами синхронизации и недостаточной эффективности из-за накладных расходов на обмен между потоками.

3.2. Параллельная сортировка с MPI

Вторая реализация использует MPI (Message Passing Interface), что позволяет разделить задачу сортировки между несколькими процессами, работающими на разных узлах в кластере или на одном многопроцессорном компьютере.

Особенности реализации:

- Массив делится на равные части, которые распределяются между процессами с помощью операции MPI_Scatter.
- Каждый процесс сортирует свою часть массива и возвращает результат с помощью операции MPI_Gather.
- После того как все процессы собрали свои отсортированные части массива, они сливаются с помощью функции слияния, что требует дополнительных вычислений.

Преимущества:

- Хорошо подходит для работы в распределенных вычислительных средах, где множество процессоров или узлов обрабатывают части задачи параллельно.
- Масштабируемость: код может эффективно использовать много процессов, что позволяет работать с большими объемами данных.
- Может быть использован на кластерах и суперкомпьютерах, что дает возможность обрабатывать очень большие массивы данных.

Недостатки:

- Более сложная реализация, требует настройки MPI и работы с распределенной памятью.
- Может быть менее эффективен на меньших объемах данных, так как накладные расходы на коммуникацию между процессами могут превысить выгоды от параллелизма.
- Меньшая гибкость в настройке по сравнению с OpenMP, поскольку работа с MPI требует явной передачи данных между процессами и их слияния.

3.3. Сравнение производительности

1. **Производительность на многопроцессорных системах:**
    - Для небольшой задачи с ограниченным количеством данных OpenMP может дать лучшие результаты, так как накладные расходы на синхронизацию потоков минимальны.
    - Для более крупных задач MPI обычно показывает лучшие результаты, так как позволяет распределить нагрузку на несколько физических машин и эффективно использовать ресурсы распределенной системы.
2. **Масштабируемость:**
    - OpenMP масштабируется только в пределах одного процесса (потока), что ограничивает его применение при увеличении данных или количества ядер.
    - MPI, напротив, масштабируется на много процессов и может использовать ресурсы кластера или распределенной системы, что делает его более подходящим для обработки больших массивов данных.
3. **Простота реализации:**
    - OpenMP проще для реализации и отладки, так как достаточно добавить несколько директив компилятора.
    - MPI требует больше усилий на настройку и управление процессами, включая синхронизацию и передачу данных между узлами.
4. **Эффективность и накладные расходы:**
    - OpenMP может быть ограничен накладными расходами на синхронизацию потоков, особенно на многозадачных системах с большим количеством потоков.
    - MPI требует дополнительных затрат на передачу данных между процессами, но для больших массивов данных эти затраты могут оправдать себя за счет масштабируемости.

3.4. Вывод

Обе реализации имеют свои преимущества и недостатки в зависимости от контекста использования. OpenMP является хорошим выбором для многозадачных вычислений на одном процессе или на многоядерной машине с умеренными размерами данных. MPI же является более подходящим вариантом для обработки больших объемов данных в распределенных вычислительных средах, таких как кластеры или суперкомпьютеры.

4. Сравнение результатов

4.1. Среднее время

OpenMP: Среднее время сортировки значительно уменьшается с увеличением числа потоков. Для 1 потока время составляет 39.16 секунд, а для 16 потоков — 6.37 секунд. Время сортировки в OpenMP сокращается до 6.37 секунд при максимальном числе потоков (16).
MPI: Среднее время также уменьшается с увеличением числа процессов. Для 1 процесса время составило 40.85 секунд, а для 16 процессов оно сократилось до 5.85 секунд. Время уменьшилось более стабильно по сравнению с OpenMP, особенно на меньших значениях числа процессов

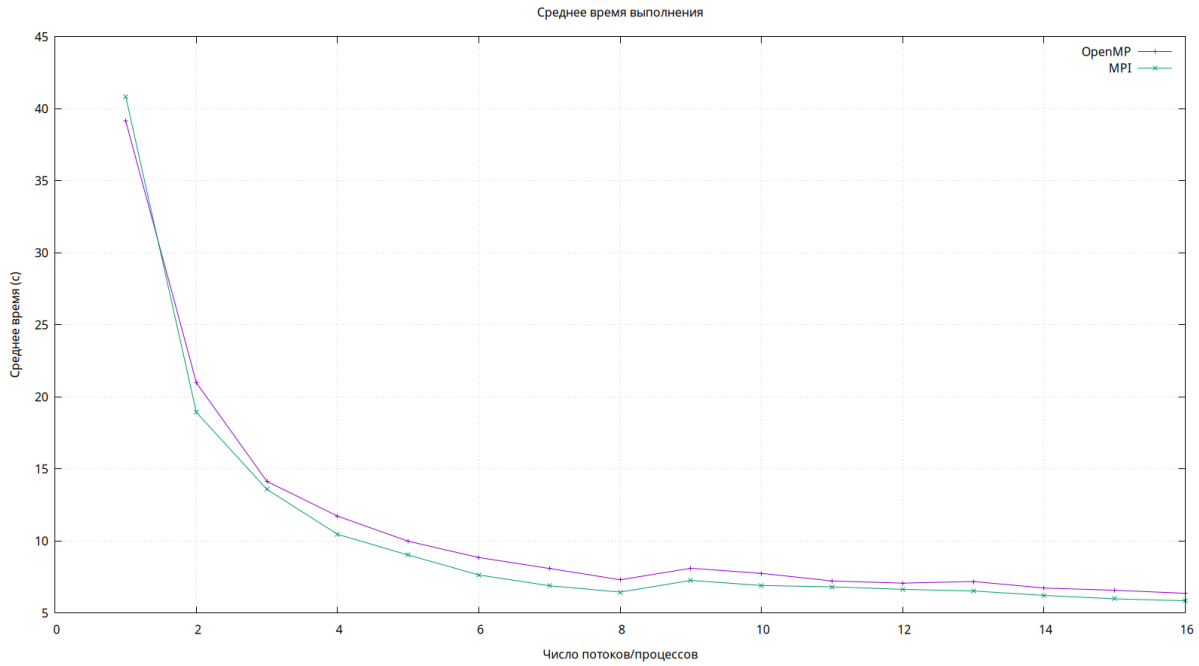

Рис. 4.1. График среднего времени от числа потоков

4.2. Ускорение

OpenMP: Ускорение растет с увеличением числа потоков до 8, после чего начинает замедляться. Максимальное ускорение достигается при 16 потоках и равно 6.15, что показывает хорошую производительность при многопоточности. Однако ускорение начинает падать после 8 потоков.
MPI: Ускорение в MPI продолжает расти почти линейно до 16 процессов, достигая максимума в 6.98. Однако ускорение стабилизируется на уровне около 5-6 с увеличением числа процессов, что связано с накладными расходами при увеличении числа процессов.

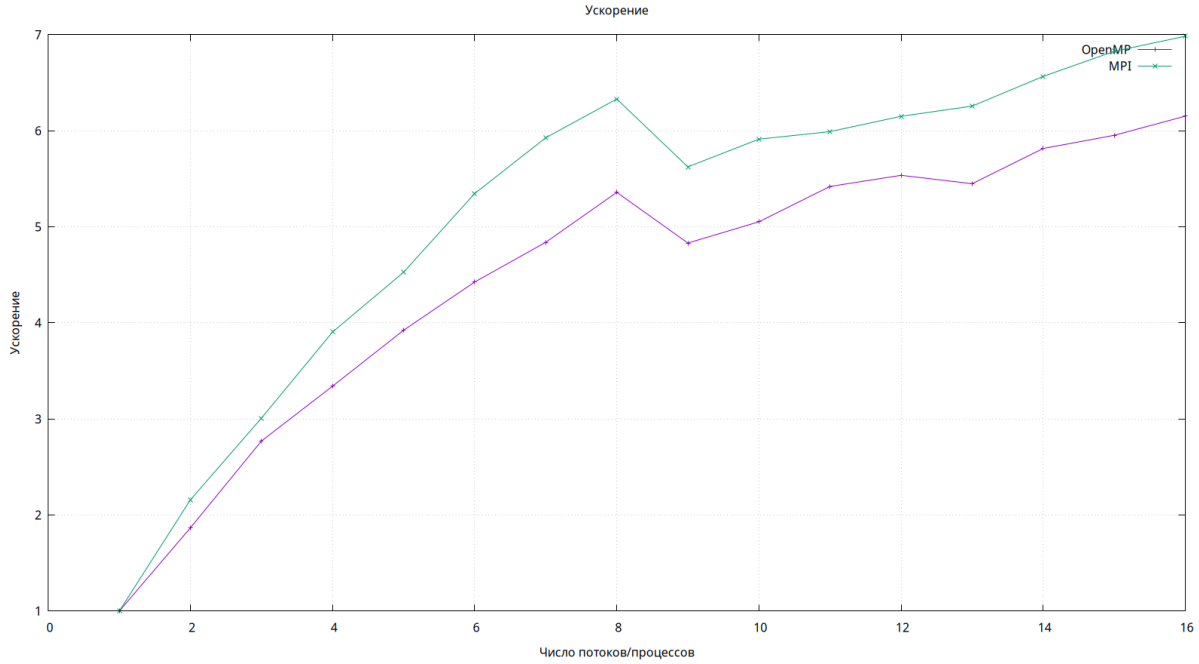

Рис. 4.2. График ускорения от числа потоков

4.3. Эффективность

OpenMP: Эффективность резко снижается с увеличением числа потоков. Для 1 потока эффективность равна 1, а при 16 потоках она падает до 0.384. Это говорит о значительных накладных расходах на управление потоками при большом числе потоков.
MPI: Эффективность в MPI остается более стабильной, чем в OpenMP, особенно при меньшем числе процессов. Например, для 8 процессов эффективность составляет 0.79, а для 16 процессов — 0.44. Несмотря на падение эффективности, MPI сохраняет более высокие значения, чем OpenMP при большом числе потоков/процессов.

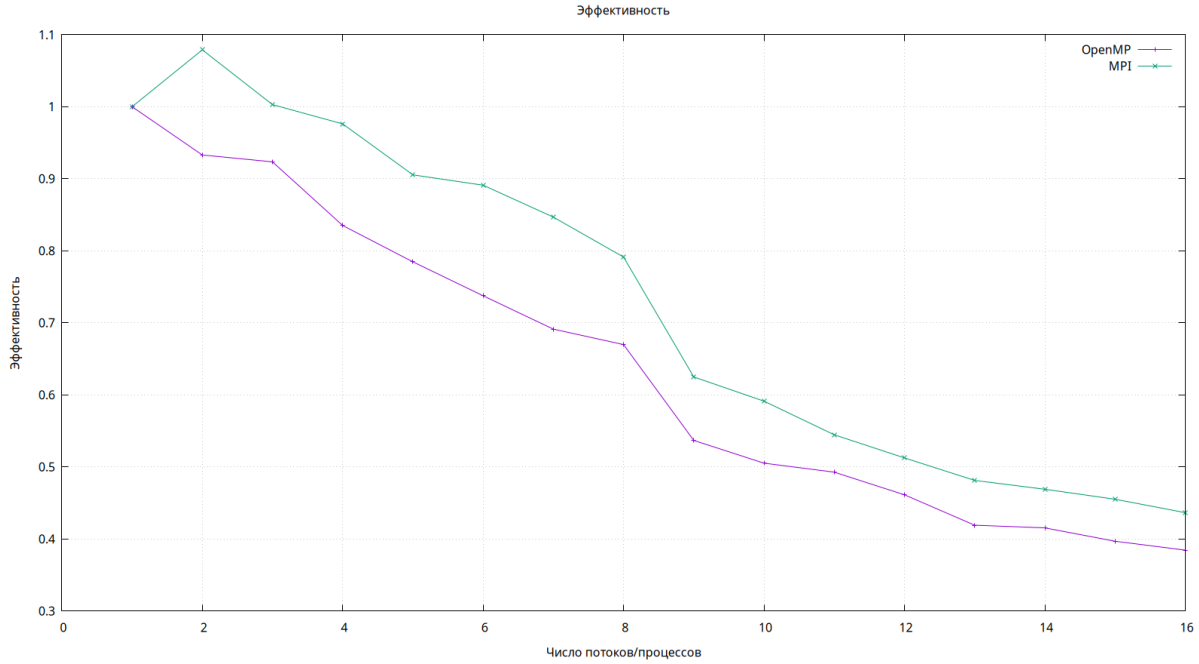

Рис. 4.3. График эффективности от числа потоков

4.4. Вывод

OpenMP демонстрирует хорошие результаты на небольших количествах потоков, но эффективность падает при увеличении числа потоков. Это может быть связано с накладными расходами на управление потоками и синхронизацию.
MPI показывает более стабильные результаты, особенно на меньших числах процессов, и лучше подходит для распределённых вычислений. Однако, при большом числе процессов (16) эффективность также снижается, но остается выше, чем у OpenMP на тех же значениях.
В целом, **MPI** более эффективно справляется с задачей при меньших значениях числа процессов и хорошо масштабируется на большом количестве процессов. **OpenMP**, с другой стороны, быстрее теряет эффективность при увеличении числа потоков, что ограничивает его использование при масштабировании на большое количество потоков.

Заключение

В ходе сравнения результатов сортировки с использованием OpenMP и MPI можно сделать несколько важных выводов относительно их производительности и эффективности.

OpenMP демонстрирует хороший прирост производительности при увеличении числа потоков, однако эффективность начинает снижаться после достижения определенного числа потоков (около 8). С увеличением числа потоков растет накладные расходы на управление потоками и синхронизацию, что приводит к ухудшению эффективности. Несмотря на это, OpenMP остаётся удобным инструментом для многозадачных вычислений на одном вычислительном узле, где накладные расходы на межпроцессное взаимодействие минимальны.

MPI в свою очередь показывает более стабильные результаты при увеличении числа процессов, особенно на меньших значениях числа процессов. Этот подход более эффективен для распределённых вычислений, где каждый процесс работает на отдельном узле, и накладные расходы на управление процессами более оправданы. Однако, несмотря на стабильность, эффективность также начинает снижаться при увеличении числа процессов, особенно на 16 процессах, но она остается выше, чем у OpenMP при аналогичном масштабировании.

Таким образом, **MPI** оказывается более подходящим для распределённых вычислений и может эффективно использоваться в средах с большим числом процессов. **OpenMP**, несмотря на свои ограничения по масштабируемости, остаётся хорошим решением для многозадачных вычислений на одном узле, особенно в случаях, когда необходимо быстрое параллельное выполнение с минимальными накладными расходами. Выбор между этими подходами зависит от конкретной задачи, архитектуры системы и требуемой производительности.

5. Приложение

5.1. Код OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>

void parallel_shell_sort(int *arr, size_t size, int threads) {
    for (size_t gap = size / 2; gap > 0; gap /= 2) {
        #pragma omp parallel for num_threads(threads)
        for (size_t start = 0; start < gap; start++) {
            for (size_t i = start + gap; i < size; i += gap) {
                int temp = arr[i];
                size_t j;
                for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                    arr[j] = arr[j - gap];
                }
                arr[j] = temp;
            }
        }
    }
}

int main() {
    const int count = 100000000;
    const int seed = 920214;

    int *array = (int *)malloc(count * sizeof(int));
    if (!array) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    srand(seed);
    for (size_t j = 0; j < count; j++) {
        array[j] = rand();
    }

    FILE *output = fopen("/home/dt/ParProg/lab6/omp_results.csv", "w");

    if (!output) {
        printf("Failed to open file for writing.\n");
        return 1;
    }

    fprintf(output, "# Threads\tAvgTime\tSpeedup\tEfficiency\n");
```

```
    double sequential_time = 0.0;

    for (int threads = 1; threads <= 16; threads++) {

        // Parallel sorting
        int *copy = (int *)malloc(count * sizeof(int));
        if (!copy) {
            printf("Memory allocation failed.\n");
            return 1;
        }
        memcpy(copy, array, count * sizeof(int));

        double start_time = omp_get_wtime();
        parallel_shell_sort(copy, count, threads);
        double end_time = omp_get_wtime();

        double time = end_time - start_time;

        free(copy);

        if (threads == 1) {
            sequential_time = time;
        }

        double speedup = sequential_time / time;
        double efficiency = speedup / threads;

        // Write results
        fprintf(output, "%d\t%.6f\t%.6f\t%.6f\n", threads, time, speedup, efficiency);
    }

    fclose(output);

    free(array);

    return 0;
}
```

Листинг 5.1. Файл lab6omp.c

5.2. Код MPI

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void shell_sort(int *arr, int size) {
    for (int gap = size / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < size; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

void merge(int *array, int start, int mid, int end) {
    int left_size = mid - start;
    int right_size = end - mid;

    int *left = (int *)malloc(left_size * sizeof(int));
    int *right = (int *)malloc(right_size * sizeof(int));

    for (int i = 0; i < left_size; i++) left[i] = array[start + i];
    for (int i = 0; i < right_size; i++) right[i] = array[mid + i];

    int i = 0, j = 0, k = start;
    while (i < left_size && j < right_size) {
        if (left[i] <= right[j]) {
            array[k++] = left[i++];
        } else {
            array[k++] = right[j++];
        }
    }

    while (i < left_size) array[k++] = left[i++];
    while (j < right_size) array[k++] = right[j++];

    free(left);
    free(right);
}

int main(int argc, char **argv) {
    const int count = 100000000; // Number of elements in the array
    const int seed = 920214;
```

```
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    int *array = NULL;
    if (rank == 0) {
        array = (int *)malloc(count * sizeof(int));
        srand(seed);
        for (int i = 0; i < count; i++) {
            array[i] = rand();
        }
    }

    int local_count = count / size;
    int *local_array = (int *)malloc(local_count * sizeof(int));

    double start_time = MPI_Wtime();

    MPI_Scatter(array, local_count, MPI_INT, local_array, local_count, MPI_INT, 0,
MPI_COMM_WORLD);

    // Each process sorts its portion of the array
    shell_sort(local_array, local_count);

    // Gather sorted subarrays at rank 0
    MPI_Gather(local_array, local_count, MPI_INT, array, local_count, MPI_INT, 0,
MPI_COMM_WORLD);

    if (rank == 0) {
        // Merge sorted subarrays
        for (int step = 1; step < size; step *= 2) {
            for (int i = 0; i < size; i += 2 * step) {
                int start = i * local_count;
                int mid = start + step * local_count;
                int end = mid + step * local_count;

                if (mid > count) mid = count;
                if (end > count) end = count;

                merge(array, start, mid, end);
            }
        }

        // Write results to a file
        FILE *file = fopen("/home/dt/ParProg/lab6/mpi_results.csv", "a");
        if (!file) {
            perror("Unable to open file");
            MPI_Abort(MPI_COMM_WORLD, 1);
```

```
        }

        double end_time = MPI_Wtime();
        fprintf(file, "%d\t%f\n", size, end_time - start_time);
        fclose(file);
        free(array);
    }

    free(local_array);
    MPI_Finalize();
    return 0;
}
```

Листинг 5.2. Файл lab6mpi.c

5.3. GNUPlot-script

```
set terminal pngcairo size 1600, 900

# Первый холст: Среднее время от числа потоков/процессов
set output 'average_time.png'
set title "Среднее время выполнения"
set xlabel "Число потоков/процессов"
set ylabel "Среднее время (c)"
set grid
plot \
    "omp_results.csv" using 1:2 with linespoints title "OpenMP", \
    "mpi_results_with_speedup.csv" using 1:2 with linespoints title "MPI"

# Второй холст: Ускорение от числа потоков/процессов
set output 'speedup.png'
set title "Ускорение"
set xlabel "Число потоков/процессов"
set ylabel "Ускорение"
set grid
plot \
    "omp_results.csv" using 1:3 with linespoints title "OpenMP", \
    "mpi_results_with_speedup.csv" using 1:3 with linespoints title "MPI"

# Третий холст: Эффективность от числа потоков/процессов
set output 'efficiency.png'
set title "Эффективность"
set xlabel "Число потоков/процессов"
set ylabel "Эффективность"
set grid
plot \
    "omp_results.csv" using 1:4 with linespoints title "OpenMP", \
    "mpi_results_with_speedup.csv" using 1:4 with linespoints title "MPI"
```

Листинг 5.3. Файл plot.gnuplot

5.4. Bash-файл для форматирования

```
#!/bin/bash

# Убедитесь, что файл mpi_results.csv существует, если нет - создадим
RESULTS_FILE="/home/dt/ParProg/lab6/mpi_results.csv"
if [ ! -f "$RESULTS_FILE" ]; then
    echo -e "# NumProcesses\tAvgTime" > "$RESULTS_FILE"
fi

# Запуск mpirun для разных значений np
for np in {1..8}  # для np от 1 до 8 без oversubscribe
do
    echo "Running with $np processes..."
    mpirun -np $np ./lab6mpi
done

for np in {9..16}  # для np от 9 до 16 с oversubscribe
do
    echo "Running with $np processes (oversubscribe)..."
    mpirun --oversubscribe -np $np ./lab6mpi
done

# Чтение и обработка данных
LINEAR_TIME=0
OUTPUT_FILE="/home/dt/ParProg/lab6/mpi_results_with_speedup.csv"

# Записываем заголовок в новый файл
echo -e "# NumProcesses\tAvgTime\tSpeedup\tEfficiency" > "$OUTPUT_FILE"

# Читаем файл и вычисляем ускорение и эффективность
while IFS=$'\t' read -r np avg_time
do
    if [ "$np" == "# NumProcesses" ]; then
        continue  # Пропустить заголовок
    fi

    # Убираем лишние пробелы и проверяем, что avg_time — это число
    avg_time=$(echo $avg_time | tr -d '[:space:]')

    # Если avg_time — это число, продолжаем
    if [[ "$avg_time" =~ ^[0-9]+(\.[0-9]+)?$ ]]; then
        # Для np=1, сохраняем время как линейное время
        if [ "$np" -eq 1 ]; then
            LINEAR_TIME=$avg_time
        fi

        # Если линейное время уже сохранено, рассчитываем ускорение и
эффективность
```

```
        if [ "$LINEAR_TIME" != "0" ]; then
            # Форматируем скорость и эффективность с 6 знаками после запятой
            SPEEDUP=$(echo "$LINEAR_TIME / $avg_time" | bc -l)
            EFFICIENCY=$(echo "$SPEEDUP / $np" | bc -l)

            # Используем printf для форматирования чисел с 6 знаками после запятой
            FORMATTED_SPEEDUP=$(printf "%.6f" $SPEEDUP)
            FORMATTED_EFFICIENCY=$(printf "%.6f" $EFFICIENCY)

            # Записываем данные в новый файл
            echo -e
"$np\t$avg_time\t$FORMATTED_SPEEDUP\t$FORMATTED_EFFICIENCY" >>
"$OUTPUT_FILE"
        fi
    else
        echo "Skipping invalid data: $np\t$avg_time"
    fi
done < "$RESULTS_FILE"

echo "Ускорение и эффективность добавлены в файл: $OUTPUT_FILE"
```

Листинг 5.4. Файл run_mpi_tests.sh

5.5. Данные для OpenMP

Thrds	AvgTime	Speedup	Efficiency
1	39.159324	1.000000	1.000000
2	20.984880	1.866073	0.933037
3	14.135074	2.770366	0.923455
4	11.722434	3.340545	0.835136
5	9.981230	3.923296	0.784659
6	8.848799	4.425383	0.737564
7	8.092685	4.838854	0.691265
8	7.307725	5.358620	0.669828
9	8.105750	4.831055	0.536784
10	7.750914	5.052220	0.505222
11	7.224696	5.420203	0.492746
12	7.073503	5.536058	0.461338
13	7.187087	5.448567	0.419121
14	6.733451	5.815639	0.415403
15	6.578488	5.952632	0.396842
16	6.365212	6.152085	0.384505

Таблица 5.5. Файл omp_results.csv

5.6. Данные для MPI

Thrds	AvgTime	Speedup	Efficiency
1	40.849538	1.000000	1.000000
2	18.930109	2.157914	1.078957
3	13.576919	3.008749	1.002916
4	10.461878	3.904609	0.976152
5	9.023271	4.527132	0.905426
6	7.640627	5.346359	0.891060
7	6.891845	5.927228	0.846747
8	6.452851	6.330464	0.791308
9	7.263191	5.624186	0.624910
10	6.908746	5.912728	0.591273
11	6.818406	5.991069	0.544643
12	6.641628	6.150531	0.512544
13	6.528913	6.256713	0.481286
14	6.221386	6.565987	0.468999
15	5.983399	6.827146	0.455143
16	5.848943	6.984089	0.436506

Таблица 5.6. Файл mpi_results_with_speedup.csv