МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра ВТ

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Архитектура параллельных вычислительных систем»
Тема: Сортировка на системах с общей памятью

Студенты гр. 0301 _____ Прохоров Б.В.
_____ Михайлов В.А.
_____ Козлов Т.В.
_____ Логунов О.Ю.
_____ Машенков И.А.
Преподаватель _____ Костичев С.В.

Санкт-Петербург
2024

Цель работы.

Изучить организацию параллельных вычислений для не вычислительных задач на системах с общей памятью.

Задание.

1. В зависимости от номера варианта задания разработать алгоритмы сортировки для последовательных и параллельных вычислений.

2. Написать и отладить программы на языке C++, реализующие разработанные алгоритмы последовательных и параллельных вычислений с использованием библиотек OpenMP и mpi.

3. Запустить программы для следующих значений размерности цепочек данных: 10, 100, 500, 1000, 5000, 10000.

4. Оценить размерность цепочек данных, при которых эффективнее использовать алгоритмы последовательного и параллельного вычислений для разного числа потоков (по крайней мере, для меньшего, равного и большего, чем число процессоров).

Вариант 3.

Сортировка выбором (SelectSort) с использованием MPI для параллельных вычислений.

Выполнение работы.

Программное и аппаратное окружение

Программное окружение при выполнении работы:

1. Операционная система: Windows 10 Pro 64bit.
2. Программа выполняется в среде WSL (Windows Subsystem for Linux), что позволяет запускать Linux-программы в Windows.
3. На WSL установлена версия дистрибутива Linux (Ubuntu 20.04).
4. Компилятор g++ (версии GCC), поддерживающий флаг -fopenmp для работы с OpenMP.
5. Библиотека MPI для распараллеливания вычислений (пакеты openmpi-bin openmpi-common libopenmpi-dev).

6. Python 3.11.4 (пакеты pandas и matplotlib).
7. IDE для разработки – Visual Studio Code с подключением к WSL.
8. Управление компиляцией и запуском программ осуществляется через командную строку WSL.

Аппаратное окружение:

1. Процессор 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz.
2. Установленная память (ОЗУ) 48 ГБ (47,7 ГБ доступно).
3. Тип системы 64-разрядная операционная система, процессор x64.

Описание метода снятия метрик производительности

Метод снятия метрик производительности для алгоритмов сортировки включает измерение времени выполнения последовательной и параллельной версий алгоритма сортировки выбором с использованием MPI (Message Passing Interface).

Для тестирования используется набор синтетических массивов разного размера со сгенерированными случайно целыми числами. Размеры массивов варьируются, что позволяет наблюдать, как изменяется время выполнения алгоритмов при увеличении объема данных. Массивы создаются с помощью функции generate_random_data, которая заполняет массив случайными числами в диапазоне от 1 до 10000.

Для каждого размера массива производится замер времени выполнения двух алгоритмов:

- Последовательная сортировка (sequential_selection_sort) – алгоритм сортировки выбором выполняется последовательно в одном потоке.
- Параллельная сортировка (parallel_selection_sort) – массив разделяется между несколькими потоками, которые работают параллельно с помощью MPI. Каждая часть массива сортируется отдельно, а затем минимальные элементы сравниваются и данные обновляются.

Измерение времени выполнения осуществляется с использованием высокоточного таймера std::chrono, который фиксирует время начала и окончания выполнения алгоритма. Время, затраченное на выполнение, рассчитывается как

разница между этими двумя моментами времени. Это выполняется внутри шаблонной функции measure_time.

Результаты измерений сохраняются в файл CSV. Каждый запуск программы с различным количеством потоков записывает следующие данные для каждого размера массива:

В файл с именем, зависящим от количества потоков (например, measurements_2.csv для двух потоков) записываются следующие данные:

Размер массива (количество элементов).

Время выполнения последовательного алгоритма.

Время выполнения параллельного алгоритма с текущим количеством потоков.

После выполнения всех вычислений данные из CSV-файлов обрабатываются с помощью Python-библиотеки pandas, и с помощью matplotlib строятся графики, на которых визуализируется зависимость времени выполнения от размера массива. Для каждого количества потоков строится отдельный график, где сравнивается время выполнения последовательной и параллельной сортировки. Графики сохраняются в формате PNG (см. рис. 1-4).

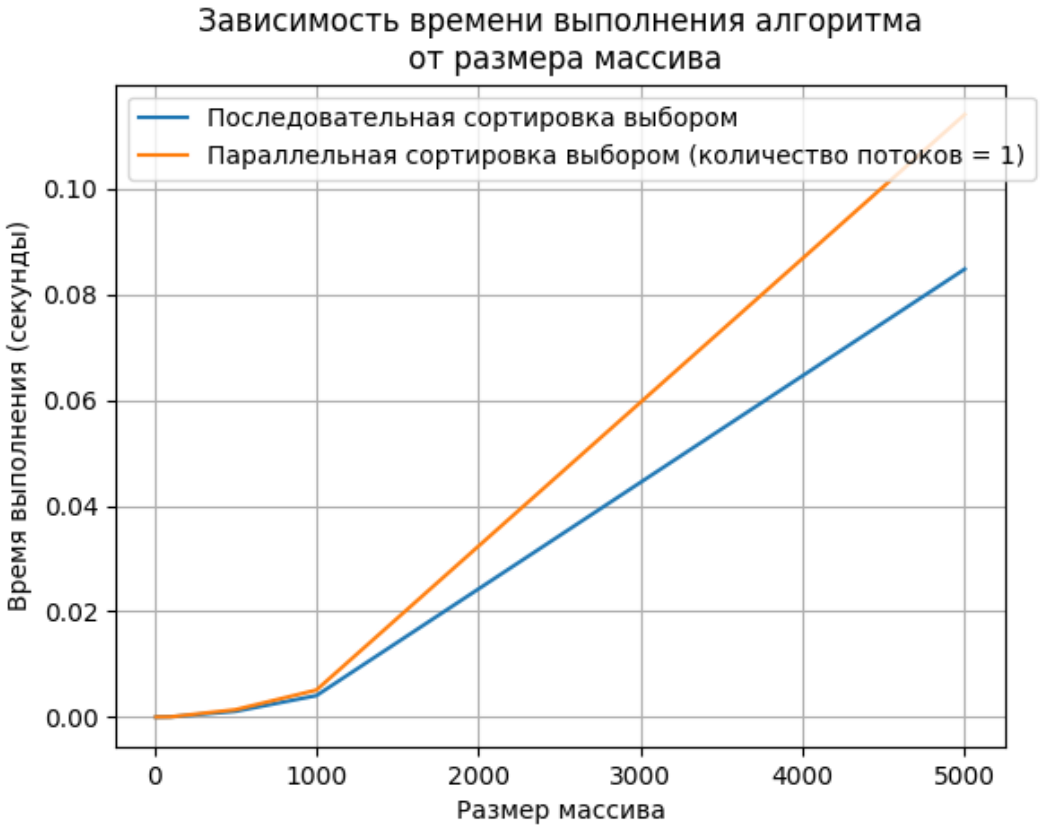

Рисунок 1 – График зависимости времени выполнения алгоритма от размерности цепочки данных. Количество потоков равно 1

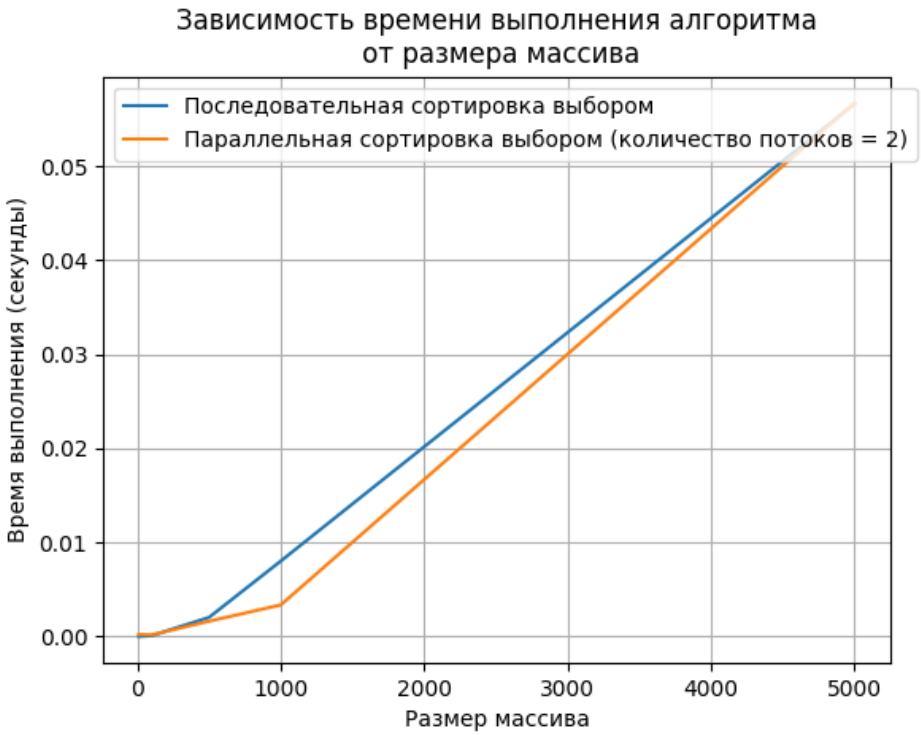

Рисунок 2 – График зависимости времени выполнения алгоритма от размерности цепочки данных. Количество потоков равно 2

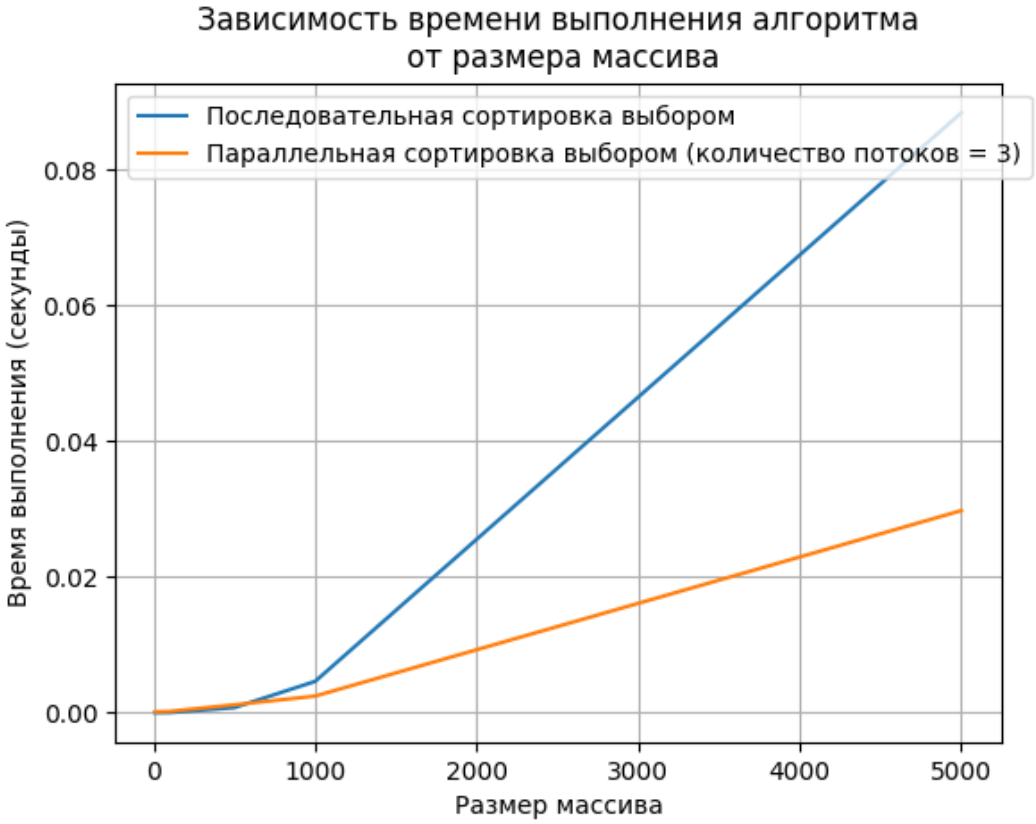

Рисунок 3 – График зависимости времени выполнения алгоритма от размерности цепочки данных. Количество потоков равно 3

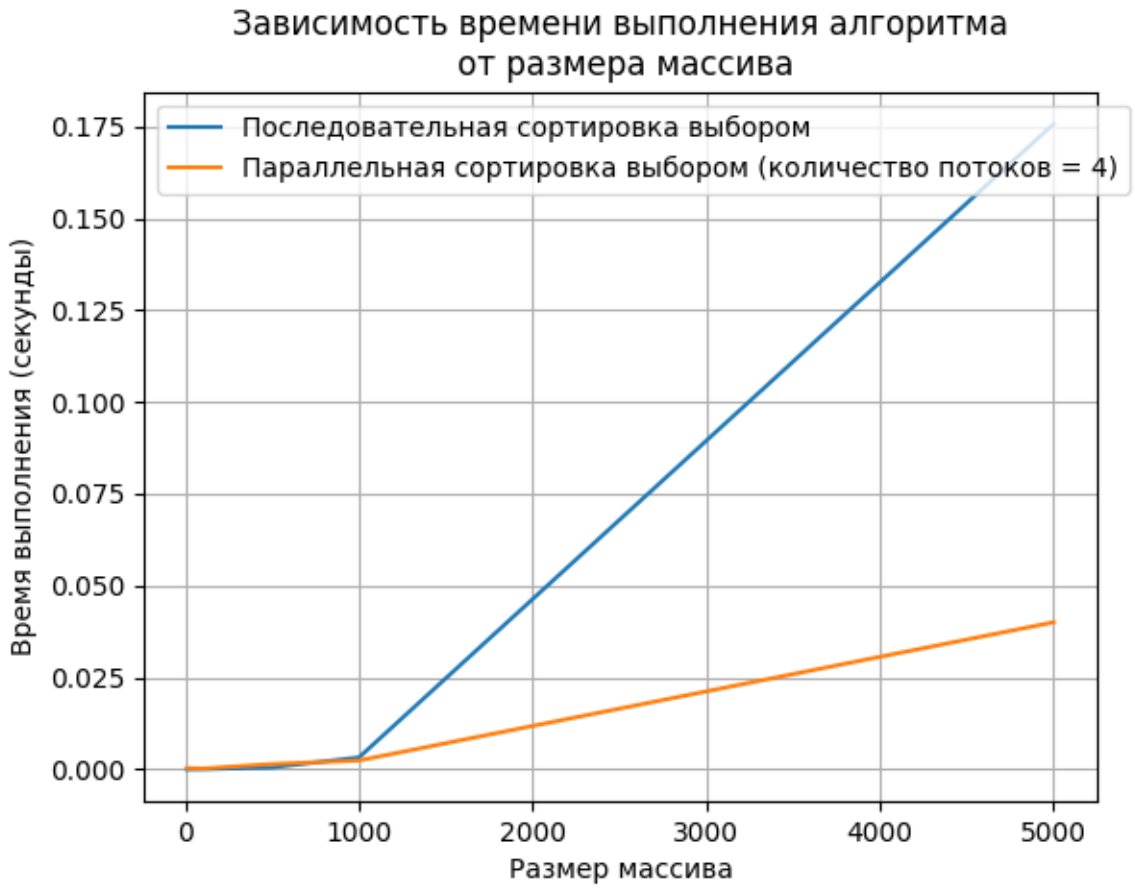

Рисунок 4 – График зависимости времени выполнения алгоритма от размерности цепочки данных. Количество потоков равно 4

Блок-схемы алгоритмов с пояснения

Блок-схема последовательного алгоритма сортировки выбором представлена на рис. 5.

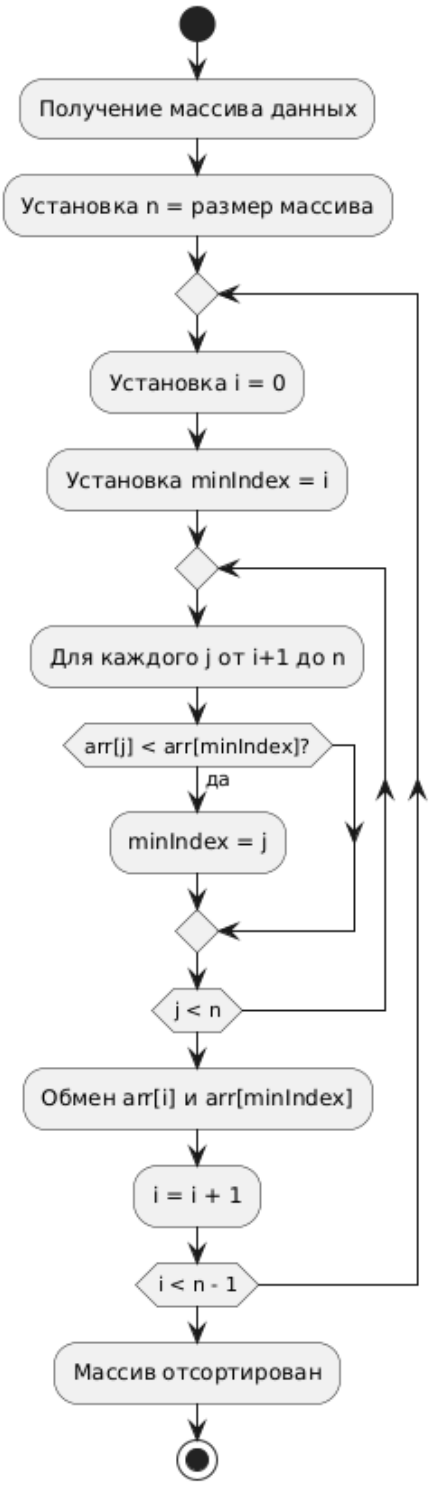

Рисунок 5 – Блок-схема последовательного алгоритма сортировки выбором

Последовательный алгоритм сортировки выбором проходит по массиву и на каждом шаге выбирает наименьший элемент из оставшейся части массива. Затем этот элемент меняется местами с текущим элементом в начале. Этот процесс повторяется для всех позиций массива до тех пор, пока весь массив не будет отсортирован.

Блок-схема параллельного алгоритма сортировки выбором представлена на рис. 6.

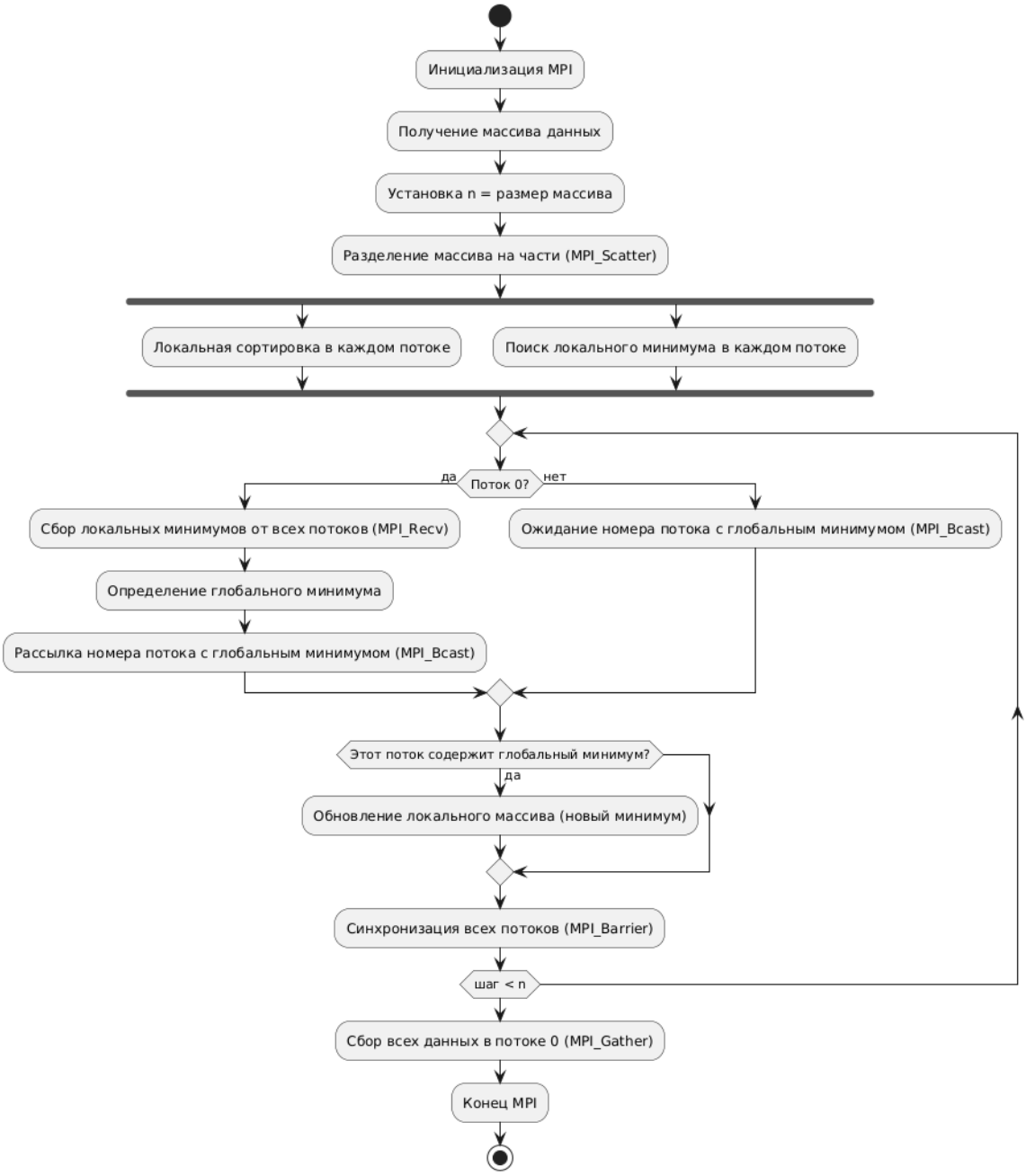

Рисунок 6 – Блок-схема параллельного алгоритма сортировки выбором

Параллельный алгоритм сортировки выбором с использованием MPI делит массив данных на части, каждая из которых обрабатывается отдельным потоком. Каждый поток сортирует свою часть данных и находит минимальный элемент. Затем поток с номером 0 собирает минимумы от всех потоков и определяет глобальный минимум. Поток с минимальным элементом обновляет свой локальный массив. Этот цикл повторяется до тех пор, пока весь массив не будет отсортирован.

Сравнительная оценка эффективности

Расчёт ускорения (Speedup) программы для параллельного алгоритма по сравнению с последовательным с помощью формулы $S = \frac{T_{serial}}{T_{parallel}}$.

Сравнительная таблица оценки эффективности программ для различных размерностей массивов (цепочек данных) при разном количестве потоков.

Таблица 1 – Сравнительная оценка эффективности программы для различных значений размерности матрицы и матрицы для разного числа потоков

Размер массива	Количество потоков	Время последовательного алгоритма	Время параллельного алгоритма	Ускорение
10	1	4.665e-06	2.5687e-05	0.18
10	2	1.866e-06	0.000216674	0.01
10	3	8.7e-07	0.000137978	0.01
10	4	1.426e-06	0.000312851	0.01
100	1	5.1871e-05	5.9298e-05	0.87
100	2	8.9468e-05	0.000179517	0.50
100	3	2.6545e-05	0.000200697	0.13
100	4	4.6048e-05	0.000199406	0.23
500	1	0.00109354	0.00141736	0.77

500	2	0.00203621	0.00162339	1.25
500	3	0.000754074	0.00116833	0.65
500	4	0.000518236	0.00141906	0.37
1000	1	0.00407977	0.00512949	0.79
1000	2	0.00802665	0.0033607	2.39
1000	3	0.00462092	0.0024528	1.88
1000	4	0.00322475	0.00239675	1.35
5000	1	0.084751	0.113995	0.74
5000	2	0.0566411	0.0567118	1.00
5000	3	0.0883696	0.0297732	2.97
5000	4	0.175675	0.0400248	4.39

Тестирование.

На рис. 7 представлен пример работы количества потоков 4.

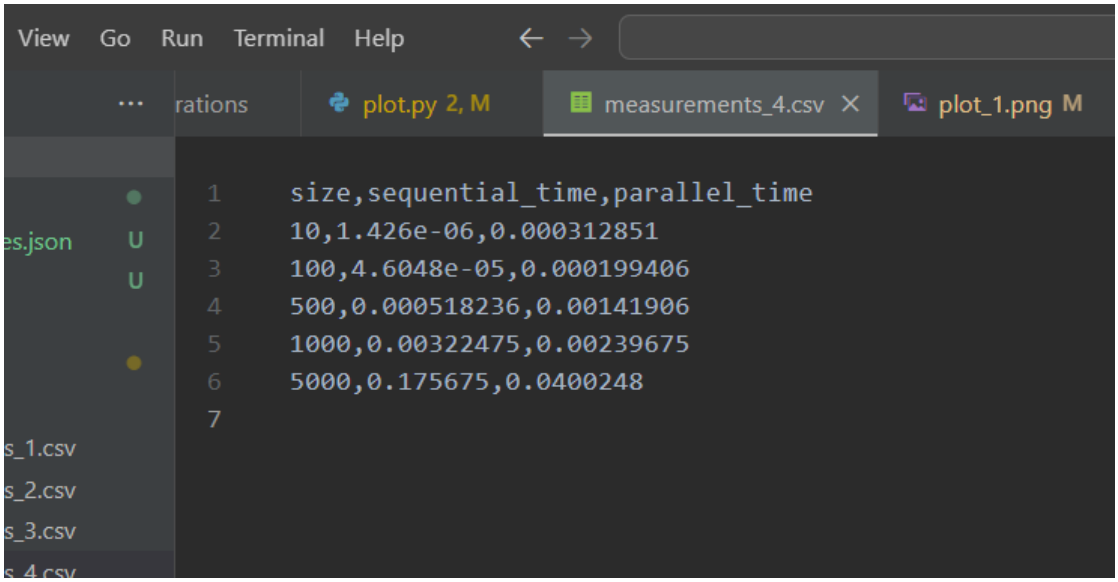

Рисунок 4 – Пример работы программы

Выводы.

Были получены знания об организации параллельных вычислений для не вычислительных задач на системах с общей памятью.

Для массивов небольшого размера (10 и 100) параллельная сортировка не эффективна. Speedup существенно ниже 1, указывая на то, что для небольших наборов данных параллелизация добавляет накладные расходы, которые перевешивают выгоду от распараллеливания.

Для массивов среднего размера (500 и 1000) параллельная сортировка начинает показывать заметное ускорение, особенно при 2 и более потоках.

Для массивов большого размера (5000) параллельная сортировка проявляет высокую эффективность, особенно при использовании 3 и 4 потоков, где достигается максимальное ускорение.

Можно сделать вывод о зависимости эффективности от количества потоков. С увеличением количества потоков до 3 и 4, параллельная сортировка начинает демонстрировать значительное улучшение по сравнению с последовательной для больших массивов, особенно при размере 5000 элементов, где ускорение превышает значение 4 для 4 потоков.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <mpi.h>
#include <algorithm>
#include <chrono>
#include <fstream>
#include <iostream>
#include <random>
#include <sstream>
#include <string>
#include <vector>

void sequential_selection_sort(std::vector<int>& arr) {
   int n = arr.size();
   for (int i = 0; i < n - 1; i++) {
      int minIndex = i;
      for (int j = i + 1; j < n; j++) {
         if (arr[j] < arr[minIndex]) minIndex = j;
      }
      std::swap(arr[i], arr[minIndex]);
   }
}

int get_local_minimum(int* arr, int start, int size) {
   int minIdx = start;
   for (int i = start + 1; i < size; i++) {
      if (arr[minIdx] > arr[i]) minIdx = i;
   }
   std::swap(arr[start], arr[minIdx]);
   return arr[start];
}

void parallel_selection_sort(std::vector<int>& arr, int rank, int
size,
                             int world_size) {
   int n = arr.size();
   int local_size = n / world_size;

   std::vector<int> local_data(local_size);

   MPI_Scatter(arr.data(), local_size, MPI_INT, local_data.data(),
local_size,
               MPI_INT, 0, MPI_COMM_WORLD);

   // Local sorting for each part of the array
   int local_min = get_local_minimum(local_data.data(), 0,
local_size);

   int global_min;
   int process_num;

   for (int step = 0; step < n; step++) {
```

```
            if (rank == 0) {
               global_min = local_min;
               process_num = 0;
            }

            for (int i = 1; i < world_size; i++) {
               if (rank == 0) {
                  int received_min;
                  MPI_Recv(&received_min, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                           MPI_STATUS_IGNORE);
                  if (received_min < global_min) {
                     global_min = received_min;
                     process_num = i;
                  }
               } else if (rank == i) {
                  MPI_Send(&local_min, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
               }
            }

            MPI_Bcast(&process_num, 1, MPI_INT, 0, MPI_COMM_WORLD);

            if (rank == process_num) {
               local_min = get_local_minimum(
                   local_data.data(), 1,
                   local_size);  // Updating the minimum element for this
process
            }

            MPI_Barrier(MPI_COMM_WORLD);
         }

         MPI_Gather(local_data.data(),  local_size,  MPI_INT,  arr.data(),
local_size,
                    MPI_INT, 0, MPI_COMM_WORLD);
      }

      template <typename Function>
      double measure_time(Function fn) {
         auto start = std::chrono::high_resolution_clock::now();
         fn();
         auto end = std::chrono::high_resolution_clock::now();
         std::chrono::duration<double> duration = end - start;
         return duration.count();
      }

      template <typename... Args>
      std::string format_string(const Args&... args) {
         std::ostringstream oss;
         (oss << ... << args);
         return oss.str();
      }

      std::vector<int> generate_random_data(int size) {
         std::vector<int> data(size);
         std::random_device rd;
         std::mt19937 gen(rd());
         std::uniform_int_distribution<> dis(1, 10000);
         for (int i = 0; i < size; i++) {
```

```
            data[i] = dis(gen);
        }
        return data;
    }

    int main(int argc, char** argv) {
        MPI_Init(&argc, &argv);

        int rank, world_size;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);

        std::vector<int> data_sizes = {10, 100, 500, 1000, 5000};

        std::string output_filename =
            format_string("measure111ments_", world_size, ".csv");
        std::string   output_path   =   format_string("./measurements/",
output_filename);

        if (rank == 0) {
            std::ofstream outfile(output_path);
            outfile << "size,sequential_time,parallel_time\n";
        }

        for (int n : data_sizes) {
            std::vector<int> data = generate_random_data(n);
            std::vector<int> parallel_data = data;

            // Measure the running time of the sequential algorithm
            double sequential_time =
                measure_time([&]() { sequential_selection_sort(data); });

            // Measure the running time of the parallel algorithm
            double parallel_time = measure_time([&]() {
                parallel_selection_sort(parallel_data, rank, n, world_size);
            });

            if (rank == 0) {
                std::ofstream outfile(output_path, std::ios_base::app);
                outfile  <<  n  <<  ","  <<  sequential_time  <<  ","  <<
parallel_time << "\n";
            }

            MPI_Barrier(MPI_COMM_WORLD);
        }

        MPI_Finalize();
        return 0;
    }
```

Название файла: plot.py

```
import pandas as pd
import matplotlib.pyplot as plt

process_counts = [1, 2, 3, 4]

for number in process_counts:
```

```
        df = pd.read_csv(f'./measurements/measurements_{number}.csv')

        plt.plot(df['size'],                          df['sequential_time'],
label='Последовательная сортировка выбором')
        plt.plot(df['size'],   df['parallel_time'],   label=f'Параллельная
сортировка выбором (количество потоков = {number})')


        plt.xlabel('Размер массива')
        plt.ylabel('Время выполнения (секунды)')
        plt.title(f'Зависимость времени выполнения алгоритма\n от размера
массива')
        plt.legend(loc='upper left')
        plt.grid(True)
        plt.savefig(f'./plots/plot_{number}.png')
        plt.close()
```