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
«Технология MPI. Введение»

Выполнил:
студент группы Б22-505
Титов Дмитрий Иванович

Принял:
Куприяшин Михаил Андреевич

Москва
2024 год

Цель работы

Изучить базовые особенности применения технлогии 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. Сравнение кодов параллельного поиска максимального элемента

Обе программы решают задачу поиска максимального значения в массиве случайных чисел, но используют разные подходы к параллелизации: **OpenMP** и **MPI**.

OpenMP

1. **Характеристики программы:**
    - Массив создаётся и обрабатывается в одном процессе.
    - Параллелизация достигается за счёт использования многопоточности.
    - Используется директива #pragma omp parallel для создания параллельных потоков.
    - Используется механизм **reduction** для агрегирования локальных максимумов из всех потоков в глобальный максимум.
2. **Преимущества:**
    - Простота реализации: OpenMP интегрирован с языком C, директивы легко добавляются в код.
    - Подходит для программ с разделяемой памятью (например, на многопроцессорных системах с общей памятью).
    - Меньше накладных расходов, так как не нужно передавать данные между процессами.
3. **Недостатки:**
    - Ограничена только системами с разделяемой памятью.
    - Параллелизация возможна только на уровне одного узла (сервер или рабочая станция).
    - Масштабируемость хуже, чем у MPI.
4. **Особенности реализации в программе:**
    - Код лаконичен: генерация массива и его обработка происходят в одном процессе.
    - Используются 16 потоков (оптимально для 16 ядер процессора).
    - Печать локального максимума из каждого потока может сбивать с толку, так как это промежуточные значения, а итоговый глобальный максимум вычисляется с использованием **reduction**.

MPI

1. **Характеристики программы:**
    - Массив создаётся только в одном процессе (корневой, rank == 0).
    - Используется распределённая память: массив разбивается на части и передаётся между процессами.
    - Используется функция MPI_Scatter для распределения данных по процессам и MPI_Reduce для объединения локальных максимумов в глобальный максимум.
2. **Преимущества:**
    - Подходит для систем с распределённой памятью (например, кластеры, суперкомпьютеры).
    - Хорошая масштабируемость: можно использовать большое количество узлов для обработки больших объёмов данных.
    - Эффективно для обработки задач, где данные изначально распределены.
3. **Недостатки:**
    - Более сложная реализация: требуется управление передачей данных между процессами.
    - Больше накладных расходов на коммуникацию между процессами.
    - Производительность может снижаться при большом количестве процессов из-за увеличения объёма коммуникаций.
4. **Особенности реализации в программе:**
    - Данные распределяются между процессами, что делает программу более сложной.
    - Максимум находится локально в каждом процессе, затем агрегируется в глобальный максимум с использованием MPI_Reduce.
    - Каждый процесс печатает свой локальный максимум, и только корневой процесс печатает глобальный максимум.

Сравнение OpenMP и MPI

Критерий	OpenMP	MPI
Тип параллелизма	Многопоточность (разделяемая память).	Многопроцессность (распределённая память).
Поддерживаемые системы	Узлы с общей памятью (например, рабочие станции).	Кластеры, распределённые системы, суперкомпьютеры.
Простота кода	Проще: директивы встроены в язык.	Сложнее: нужно управлять передачей данных.
Масштабируемость	Ограничена ядрами процессора в одном узле.	Масштабируется на тысячи узлов.
Накладные расходы	Низкие: потоки работают с общей памятью.	Высокие: требуется передача данных между процессами.
Производительность	Высока на одном узле.	Высока при распределённых вычислениях.
Программная модель	Однопроцессная с разделяемой памятью.	Многопроцессная с обменом сообщениями.

Выводы:

- **OpenMP** отлично подходит для задач, которые выполняются на системах с общей памятью, особенно если параллелизация ограничена одним узлом. Код проще, и накладные расходы минимальны.
- **MPI** более универсален и эффективен для распределённых систем, где ресурсы распределены по нескольким узлам. Он сложнее в реализации, но лучше масштабируется.

Если задача ограничена производительностью одного узла, OpenMP — лучший выбор. Если же нужно использовать кластер или распределённую систему, MPI будет предпочтительнее.

3. Сравнение вычислительных систем OpenMP и MPI

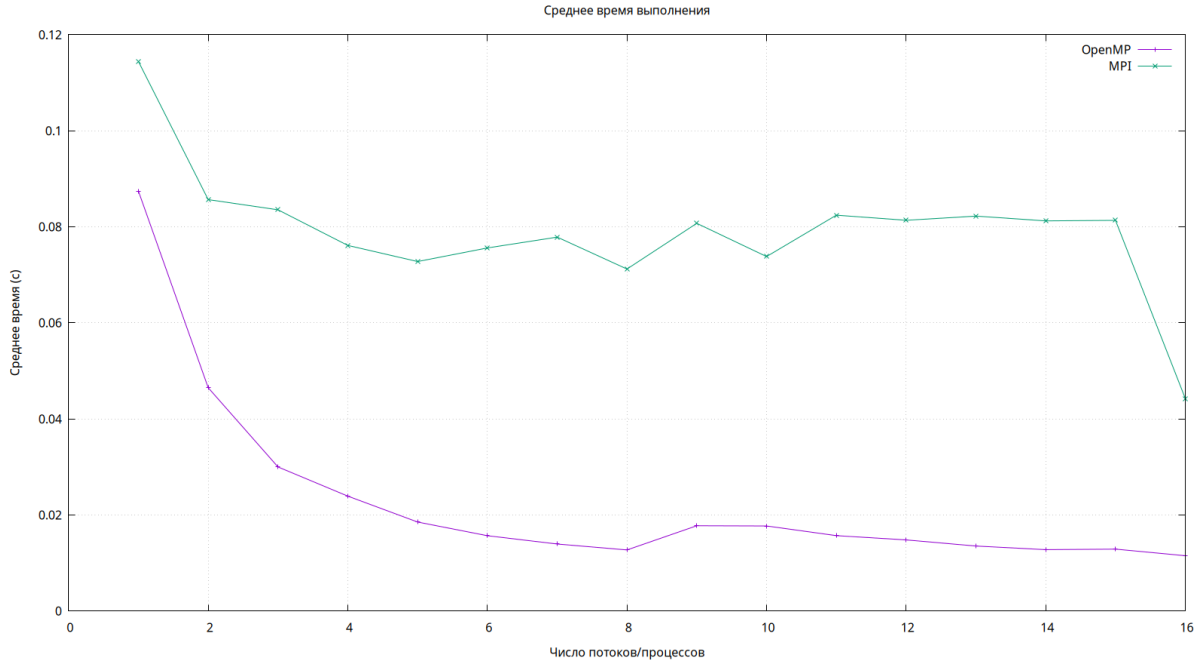

Рис. 3.1. Графики зависимости среднего времени от числа потоков

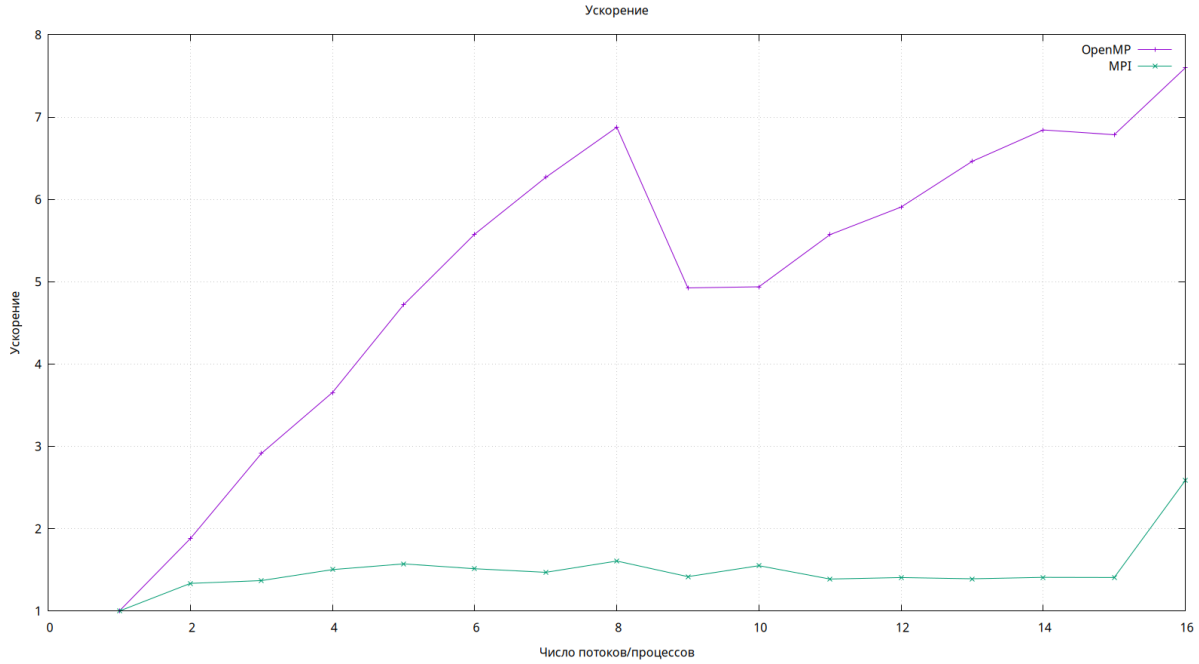

Рис. 3.2. Графики зависимости ускорения от числа потоков

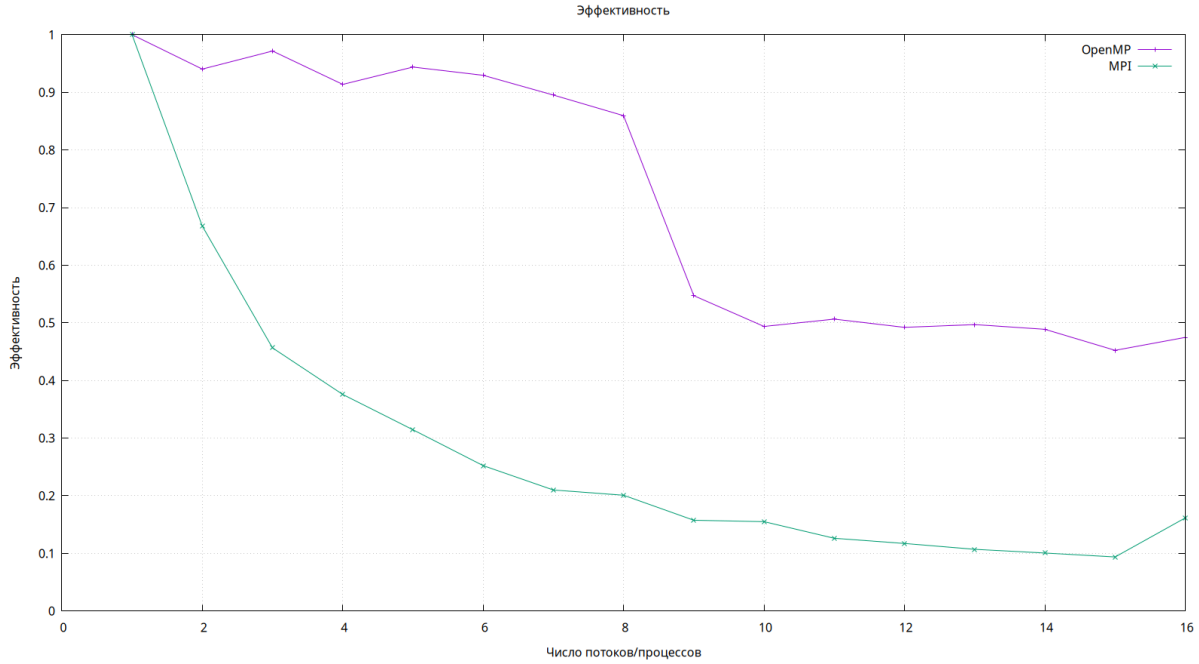

Рис. 3.3. Графики зависимости эффективности от числа потоков

Анализ результатов работы программ

OpenMP

Результаты программы на OpenMP показывают следующие ключевые моменты:

1. **Время выполнения и скорость**:
   - С увеличением количества потоков время выполнения задачи сокращается, что ожидаемо для параллельных вычислений.
   - Наибольший прирост скорости (speedup) наблюдается при переходе от 1 потока к 2 потокам, где скорость увеличивается почти в 2 раза (1.88). После этого прирост становится менее заметным.
   - На 16 потоках достигается максимальная скорость — около **7.6x** по сравнению с 1 потоком.
2. **Эффективность параллелизма**:
   - Эффективность параллельных вычислений (отношение speedup к количеству потоков) в начале (для 2–3 потоков) высока и превышает 0.9. Это указывает на хорошую эффективность параллелизма.
   - Однако начиная с 9 потоков эффективность значительно снижается, достигая 0.49 на 16 потоках. Это связано с увеличением накладных расходов на управление потоками и синхронизацию, которые становятся более выраженными при большом количестве потоков.
3. **Итоги**:
   - В целом, OpenMP демонстрирует хороший прирост скорости с увеличением числа потоков, однако эффективность начинает снижаться после 8 потоков.

- Это может быть связано с тем, что многозадачность на одном процессоре с 16 потоками достигает определённой предельной эффективности из-за накладных расходов на управление потоками и синхронизацию данных.

MPI

Результаты программы с использованием MPI показывают следующие ключевые моменты:

1. **Время выполнения и скорость**:
   - Для MPI также наблюдается снижение времени выполнения с увеличением числа процессов. Однако наибольший прирост скорости наблюдается при переходе от 1 процесса к 2 (speedup = 1.34), и дальнейший рост скорости происходит медленно.
   - При 16 процессах скорость выполнения увеличивается в 2.58 раза, что значительно меньше, чем для OpenMP (7.6x на 16 потоках).
2. **Эффективность параллелизма**:
   - Эффективность для MPI в начале (до 4 процессов) относительно высока, но с увеличением количества процессов эффективность начинает снижаться.
   - Например, на 16 процессах эффективность составляет **0.16**, что значительно ниже, чем у OpenMP. Это может быть связано с тем, что для MPI каждый процесс должен передавать данные между узлами, что создаёт дополнительные накладные расходы на коммуникацию.
3. **Итоги**:
   - MPI в основном показывает меньший прирост скорости по сравнению с OpenMP. Это может быть связано с высокой стоимостью операций по передаче данных между процессами, что ограничивает эффективность при большом количестве процессов.
   - Несмотря на это, на 16 процессах время выполнения значительно сокращается (до 0.044 с), что может быть полезно в крупных распределённых системах, где использование нескольких узлов оправдано.

Сравнение OpenMP и MPI

Критерий	OpenMP	MPI
Время выполнения	Снижается до 0.0115 с увеличением потоков.	Снижается до 0.0443 с увеличением процессов.
Скорость	Максимальный speedup — 7.6x (16 потоков).	Максимальный speedup — 2.6x (16 процессов).
Эффективность	Эффективность падает ниже 0.5 на 16 потоках.	Эффективность падает до 0.16 на 16 процессах.
Рекомендации	Лучше подходит для многозадачных систем с разделяемой памятью.	Лучше подходит для распределённых систем с несколькими узлами.

Выводы и рекомендации

1. **OpenMP**:
    - OpenMP показывает лучшие результаты для параллелизма с потоками на одном узле. Он обеспечивает хорошую эффективность при небольшом количестве потоков, но с увеличением числа потоков эффективность снижается, что связано с накладными расходами на управление потоками.
    - Для задач на одном сервере или рабочей станции OpenMP будет предпочтительнее, так как он эффективен для многозадачных систем с общей памятью.
2. **MPI**:
    - MPI подходит для распределённых вычислений, но накладные расходы на передачу данных между процессами ограничивают его эффективность. Он работает хорошо на кластерах и многозадачных системах, где данные должны быть распределены между узлами.
    - Несмотря на меньшую скорость на относительно небольшом количестве процессов, MPI может оказаться полезным в задачах, требующих параллельных вычислений на кластерах или суперкомпьютерах.
3. **Общие рекомендации**:
    - Для **задач с общими данными, где все вычисления выполняются на одном устройстве**, OpenMP — лучший выбор, так как он эффективен и прост в реализации.
    - Для **задач, требующих работы на распределённых системах или кластерах**, MPI будет более подходящим выбором, несмотря на больший накладной расход на коммуникацию.

Заключение

Из проведенного анализа можно сделать следующие ключевые выводы:

1. **OpenMP**: Этот подход наиболее эффективен для многозадачных вычислений на одном узле с общей памятью. Он демонстрирует значительное улучшение скорости с увеличением числа потоков, но эффективность начинает снижаться при использовании большого количества потоков из-за накладных расходов на управление и синхронизацию. Для задач, где требуется параллелизм на одном сервере, OpenMP является оптимальным выбором.
2. **MPI**: Этот подход более подходит для распределенных вычислений на нескольких узлах, однако накладные расходы на передачу данных между процессами значительно снижают его производительность при большом количестве процессов. Несмотря на меньшую скорость, MPI подходит для задач, требующих работы в распределенных системах или кластерах, где обработка данных осуществляется на нескольких машинах.
3. **Сравнение и рекомендации**:
    - Для **задач на одном узле** с разделяемой памятью OpenMP является более эффективным решением.
    - Для **распределенных вычислений** на кластерах или суперкомпьютерах MPI остается лучшим выбором, несмотря на большую стоимость коммуникации между процессами.

Таким образом, выбор между OpenMP и MPI зависит от типа задачи и используемой инфраструктуры. OpenMP предпочтителен для многозадачности на одном устройстве, тогда как MPI лучше справляется с параллельными вычислениями в распределенных системах.

4. Приложение

4.1. Листинг lab5OMP.c

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

int main(int argc, char** argv)
{
   const int count = 10000000;     ///< Number of array elements
   const int threads = 16;         ///< Number of parallel threads to use
   const int random_seed = 920215; ///< RNG seed

   int* array = 0;                 ///< The array we need to find the max in
   int  max   = -1;                ///< The maximal element

   /* Initialize the RNG */
   srand(random_seed);

   /* Determine the OpenMP support */
   printf("OpenMP: %d;\n======\n", _OPENMP);

   /* Generate the random array */
   array = (int*)malloc(count*sizeof(int));
   for(int i=0; i<count; i++) { array[i] = rand(); }

   /* Find the maximal element */
   #pragma omp parallel num_threads(threads) shared(array, count) reduction(max: max)
default(none)
   {
      #pragma omp for
      for(int i=0; i<count; i++)
      {
         if(array[i] > max) { max = array[i]; };
      }
      printf("-- My lmax is: %d;\n", max);
   }

   printf("======\nMax is: %d;\n", max);
   return(0);
}
```

Листинг 4.1. Код программы lab5OMP.c

4.2. Листинг lab5MPI.c

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

int main(int argc, char** argv)
{
   const int count = 10000000;     ///< Number of array elements
   const int random_seed = 920215; ///< RNG seed

   int *array = NULL;              ///< The array we need to find the max in
   int local_max = -1;             ///< The maximal element in the local process
   int global_max = -1;            ///< The global maximal element after reduction
   int rank, size;                 ///< MPI rank and size

   /* Initialize MPI */
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   /* Initialize the RNG (only for the root process) */
   if (rank == 0) {
      srand(random_seed);
      array = (int*)malloc(count * sizeof(int));

      /* Generate the random array (only on root process) */
      for (int i = 0; i < count; i++) {
         array[i] = rand();
      }
   }

   /* Divide the work among processes */
   int local_count = count / size;
   int *local_array = (int*)malloc(local_count * sizeof(int));

   /* Scatter the data to all processes */
   MPI_Scatter(array, local_count, MPI_INT, local_array, local_count, MPI_INT, 0,
MPI_COMM_WORLD);

   /* Find the local maximum */
   for (int i = 0; i < local_count; i++) {
      if (local_array[i] > local_max) {
         local_max = local_array[i];
```

```
        }
    }

    /* Print the local maximum for each process */
    printf("Process %d: Local max = %d\n", rank, local_max);

    /* Reduce all local maxima to global maximum */
    MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0,
MPI_COMM_WORLD);

    /* Only the root process prints the global maximum */
    if (rank == 0) {
        printf("Max is: %d;\n", global_max);
    }

    /* Cleanup */
    if (rank == 0) {
        free(array);
    }
    free(local_array);

    /* Finalize MPI */
    MPI_Finalize();
    return 0;
}
```

Листинг 4.2. Код программы lab5MPI.c

4.3. Листинг lab5_omp.c

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

int main(int argc, char** argv)
{
    const int count = 100000000;     // Number of array elements
    const int threads_max = 16;      // Maximum number of parallel threads to use
    const int num_arrays = 10;       // Number of experiments per thread count

    int** arrays;                    // Array of pointers for random number arrays (now dynamic)
    int  max;                        // Maximal element
    double start_time, end_time;     // Timing variables
    double total_time;               // Sum of times for averaging
```

```
double time_seq;                // Time for sequential equal for 1 thread
double avg_time;                // Average time for thread
double speedup, efficiency;     // Speedup and efficiency variables

/* Define 10 fixed random seeds */
const int seeds[10] = {123456, 789012, 345678, 901234, 567890,
                       112233, 445566, 778899, 990011, 223344};

/* Open file to write results for Gnuplot */
FILE *fp = fopen("/home/dt/ParProg/lab5/solve/omp_results.csv", "w");
if (!fp) {
   perror("Unable to open file");
   return 1;
}

fprintf(fp, "# threads \t Avg Time (s) \t Speedup \t Efficiency\n");

/* Allocate memory for the array of pointers dynamically */
arrays = (int**)malloc(num_arrays * sizeof(int*));

/* Generate 10 different arrays with fixed random seeds */
for (int arr = 0; arr < num_arrays; arr++) {
   /* Allocate memory for each array dynamically */
   arrays[arr] = (int*)malloc(count * sizeof(int));

   /* Set a fixed seed for each array */
   srand(seeds[arr]);

   /* Fill the array with random values */
   for(int i = 0; i < count; i++) {
      arrays[arr][i] = rand();
   }
}

/* Perform experiments for different number of threads */
for(int threads = 1; threads <= threads_max; threads++) {

   total_time = 0.0; // Reset total time for each thread count

   for (int arr = 0; arr < num_arrays; arr++) {

      max = -1; // Reset max value before each experiment

      /* Start timing */
```

```
        start_time = omp_get_wtime();

        /* Parallel computation using pre-generated array */
        #pragma omp parallel num_threads(threads) shared(arrays, count, arr) reduction(max:
max) default(none)
        {
           #pragma omp for
           for(int i = 0; i < count; i++) {
              if(arrays[arr][i] > max) { max = arrays[arr][i]; }
           }
        }

        /* End timing */
        end_time = omp_get_wtime();
        /* Add the current array's time */
        total_time += (end_time - start_time);
     }

     /* Compute and output the average time for the current threads */
     avg_time = total_time / num_arrays;
     printf("Average for threads %d: Avg Time = %f\n", threads, avg_time);

     if (threads == 1) {
        time_seq = avg_time;
     }

     speedup  = time_seq / avg_time;
     efficiency = speedup / threads;

     /* Write results to CSV file */
     fprintf(fp, "%d\t%f\t%f\t%f\n", threads, avg_time, speedup, efficiency);
  }

  /* Free all allocated arrays after all experiments */
  for (int arr = 0; arr < num_arrays; arr++) {
     free(arrays[arr]);
  }

  /* Free the array of pointers */
  free(arrays);

  /* Close the file */
  fclose(fp);
```

```
    return 0;
}
```

Листинг 4.3. Код программы lab5_omp.c

Код представленный в лабораторной номер 1, для подсчета времени, ускорения и эффективности программы на базе OpenMP.

4.4. Листинг lab5_mpi.c

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

int main(int argc, char** argv)
{
    const int count = 100000000;     ///< Number of array elements
    const int random_seed = 920215; ///< RNG seed
    const int num_experiments = 10; ///< Number of experiments

    int *array = NULL;              ///< The array we need to find the max in
    int local_max = -1;            ///< The maximal element in the local process
    int global_max = -1;           ///< The global maximal element after reduction
    int rank, size;               ///< MPI rank and size

    double total_time = 0.0;        ///< Total time for averaging
    double start_time, end_time;    ///< Timing variables
    double avg_time;               ///< Average time

    /* Initialize MPI */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* Initialize the RNG (only for the root process) */
    if (rank == 0) {
        srand(random_seed);
        array = (int*)malloc(count * sizeof(int));

        /* Generate the random array (only on root process) */
        for (int i = 0; i < count; i++) {
            array[i] = rand();
        }
    }
```

```
    /* Divide the work among processes */
    int local_count = count / size;
    int *local_array = (int*)malloc(local_count * sizeof(int));

    /* Repeat experiments */
    for (int exp = 0; exp < num_experiments; exp++) {
        local_max = -1; // Reset local max before each experiment

        /* Start timing */
        start_time = MPI_Wtime();

        /* Scatter the data to all processes */
        MPI_Scatter(array, local_count, MPI_INT, local_array, local_count, MPI_INT, 0,
MPI_COMM_WORLD);

        /* Find the local maximum */
        for (int i = 0; i < local_count; i++) {
            if (local_array[i] > local_max) {
                local_max = local_array[i];
            }
        }

        /* Reduce all local maxima to global maximum */
        MPI_Reduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, 0,
MPI_COMM_WORLD);

        /* End timing */
        end_time = MPI_Wtime();

        /* Add the current experiment time to the total time */
        total_time += (end_time - start_time);
    }

    /* Calculate and print the average time */
    avg_time = total_time / num_experiments;

    /* Open file to store results */
    FILE *file = NULL;
    if (rank == 0) {
        file = fopen("/home/dt/ParProg/lab5/solve/mpi_results.csv", "a");
        if (file == NULL) {
            perror("Unable to open file");
            MPI_Abort(MPI_COMM_WORLD, 1);
        }
```

```
        /* Write the results for the current experiment */
        fprintf(file, "%d\t%f\n", size, avg_time);
        fclose(file);
    }

    /* Cleanup */
    if (rank == 0) {
        free(array);
    }
    free(local_array);

    /* Finalize MPI */
    MPI_Finalize();
    return 0;
}
```

Листинг 4.4. Код программы lab5_mpi.c

Данный код нужен для получения времени для вычислений с помощью MPI

4.5. Листинг plot.gnuplot

```
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 4.5. Код программы plot.gnuplot

4.6. Листинг run_mpi_tests.sh

```
#!/bin/bash

# Убедитесь, что файл mpi_results.csv существует, если нет - создадим
RESULTS_FILE="/home/dt/ParProg/lab5/solve/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 ./lab5mpi
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 ./lab5mpi
done

# Заменить запятые на табуляцию в файле перед обработкой
sed -i 's/,/\t/g' "$RESULTS_FILE"

# Чтение и обработка данных
LINEAR_TIME=0
OUTPUT_FILE="/home/dt/ParProg/lab5/solve/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 4.6. Код программы run_mpi_tests.sh

Этот код нужен, чтобы протестировать lab5_mpi.c, тем что вызовет запуск программы, запишет результаты времени в файл mpi_results.csv, после чего посчитает ускорение и эффективность и сформирует файл mpi_results_with_speedup.csv, который используется для построения графиков.

4.7. Данные работы программ

# threads	Avg Time (s)	Speedup	Efficiency
1	0.087436	1.000000	1.000000
2	0.046492	1.880687	0.940343
3	0.029995	2.915022	0.971674
4	0.023921	3.655177	0.913794
5	0.018528	4.719232	0.943846
6	0.015678	5.577135	0.929522
7	0.013952	6.266973	0.895282
8	0.012717	6.875761	0.859470
9	0.017755	4.924695	0.547188
10	0.017707	4.937827	0.493783
11	0.015694	5.571358	0.506487
12	0.014804	5.906320	0.492193
13	0.013532	6.461288	0.497022
14	0.012777	6.843188	0.488799
15	0.012887	6.785097	0.452340
16	0.011510	7.596854	0.474803

Таблица 4.7.1. Содержимое файла omp_results.csv

# NumProcesses	AvgTime	Speedup	Efficiency
1	0.114431	1.000000	1.000000
2	0.085662	1.335843	0.667922
3	0.083539	1.369791	0.456597
4	0.076122	1.503258	0.375814
5	0.072768	1.572546	0.314509
6	0.075601	1.513618	0.252270
7	0.077852	1.469853	0.209979
8	0.071207	1.607019	0.200877
9	0.080723	1.417576	0.157508
10	0.073806	1.550430	0.155043
11	0.082421	1.388372	0.126216
12	0.081358	1.406512	0.117209
13	0.082229	1.391614	0.107047
14	0.081231	1.408711	0.100622
15	0.081341	1.406806	0.093787
16	0.044257	2.585602	0.161600

Таблица 4.7.2. Содержимое файла mpi_results_with_speedup.csv