Лабораторная работа №7:
Совместное применение OpenMP и MPI

ИКС ИБ
Б20-505
Барабанов Андрей

2022

Содержание

1 Рабочая среда

- Модель процессора: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
- Объём оперативной памяти: 16,0 ГБ
- Тип оперативной памяти: DDR4
- Версия операционной системы: ManjaroLinux 22.0.0
- Разрядность операционной системы: x86_64
- Среда разработки: GCC 12.2.0
- Версия OpenMPI: 4.1.4
- Версия OpenMP: 4.5

```
[bar1k@manjaro ~]$ lscpu
Архитектура:             x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Порядок байт:          Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
ID прроизводителя:       GenuineIntel
  Имя модели:            Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz
    Семейство ЦПУ:       6
    Модель:              165
    Thread(s) per core:  2
    Ядер на сокет:       6
    Сокетов:             1
    Степпинг:            2
    CPU(s) scaling MHz:  18%
    CPU max MHz:         5000,0000
    CPU min MHz:         800,0000
    BogoMIPS:            5202,65
```

```
[bar1k@manjaro ~]$ free -h
               total        used        free      shared  buff/cache   available
Mem:            15Gi       2,2Gi        11Gi       132Mi       2,2Gi        12Gi
Swap:             0B          0B          0B
```

```
[bar1k@manjaro ~]$ cat /etc/lsb-release
DISTRIB_ID=ManjaroLinux
DISTRIB_RELEASE=22.0.0
DISTRIB_CODENAME=Sikaris
DISTRIB_DESCRIPTION="Manjaro Linux"
```

```
[bar1k@manjaro ~]$ gcc --version
gcc (GCC) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
Это свободно распространяемое программное обеспечение. Условия копирования
приведены в исходных текстах.

  Без гарантии каких-либо качеств, включая
коммерческую ценность и применимость для каких-либо целей.
```

```
[bar1k@manjaro lab5]$ mpirun --version
mpirun (Open MPI) 4.1.4

Report bugs to http://www.open-mpi.org/community/help/
```

```
[bar1k@manjaro ~]$ echo | cpp -fopenmp -dM | grep -i open
#define _OPENMP 201511
```

2 Описание хода работы

2.1 Описание работы алгоритма

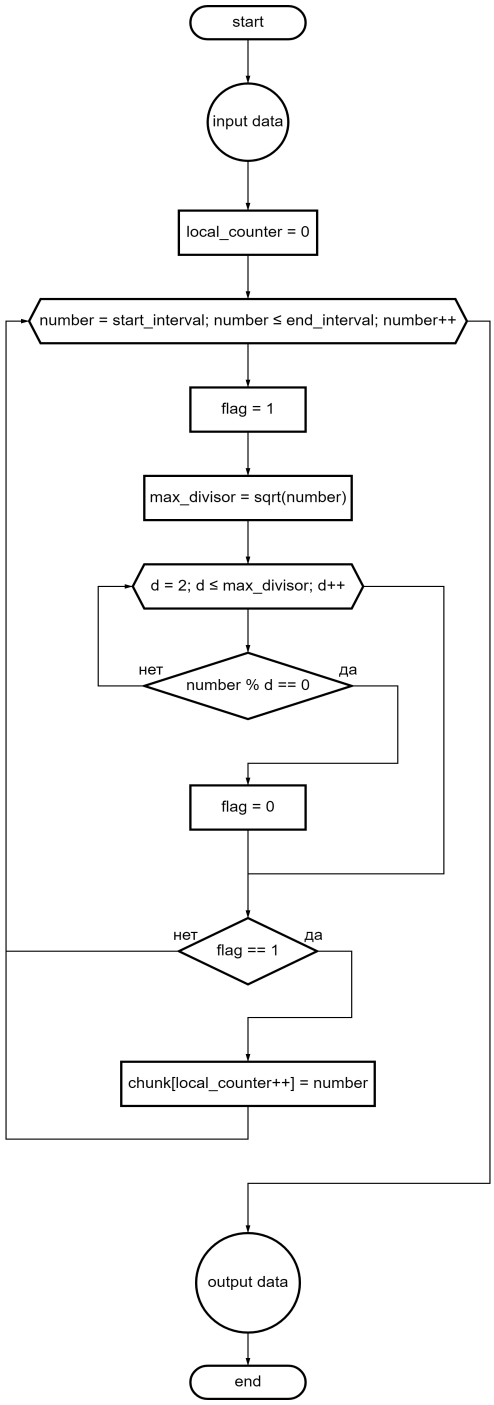
В данной лабораторной работе необходимо было разработать простейший алгоритм поиска простых чисел из заданного промежутка с помощью применения технологий MPI и OpenMP. Действия алгоритма состоят в следующем:

1. Первый этап: с помощью MPI разделяется работа по просмотру заданной области поиска, для этого определяются начальные и конечные числа диапазона для каждого потока, то есть:
   const int start_interval = rank * length_range / size + start_range
   const int end_interval = (rank + 1) * length_range / size + start_range - 1.

2. Второй этап: для кажого числа из заданного промежутка в кадом потоке MPI с помощью цикла перебираются его потенциальные делители. С помощью технологии OpenMP создаётся несколько процессов в одном потоке:
   #pragma omp parallel for num_threads(n) shared(flag, number) default(none).
   Простое число записывается в массив chunk, определённый для каждого потока.

3. Третий этап: собираем все найденные простые числа с кажого потока в корневой поток. Так как каждый массив может иметь разную длину необходимо воспользоваться опцией **MPI_Gatherv(chunk, local_counter, MPI_INTEGER, array, counters, offsets, MPI_INTEGER, 0, MPI_COMM_WORLD)**, а для этого ещё нужно сформировать массивы смещений и количества элементов в каждом потоке **offsets** и **counters**.

2.2 Блок-схема алгоритма



3 Данные

3.1 Входные данные

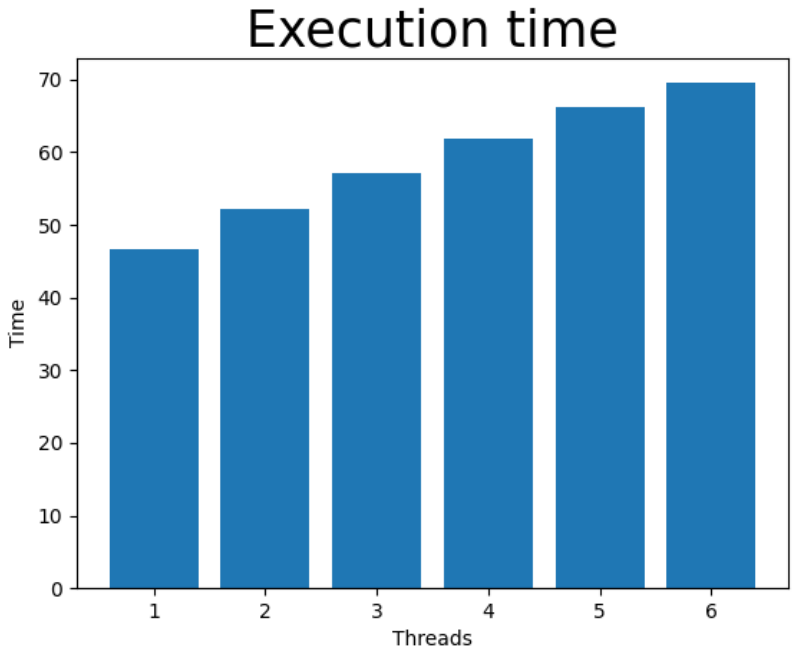
1. Число процессов MPI: 6
2. Число потоков OpenMP: 2
3. Размер рабочей области: [100000000, 300000000]
4. Принцип её разделения: заданный промежуток разбивается на количество потоков в порядке возрастания.

3.2 Экспериментальные данные

1. Время работы последовательного алгоритма: 322.704087 с.
2. Время работа параллельного алгоритма: 67.646763 с.
3. Ускорение параллельного алгоритма: 4,770429
4. Эффективность параллельного алгоритма: 0,795072

4 Графики

4.1 Гистограмма



5 Программные коды

5.1 build.sh

```
#!/usr/bin/zsh

N_START=100000000
N_END=300000000
THREADS_NUMBER=6
PROCESSORS_NUMBER=12

echo "Compiling..."
mpicc -o prog7 -lm src/lab7.c

echo "Working..."
mpirun -nq $THREADS_NUMBER ./prog7 $N_START $N_END $THREADS_NUMBER $PROCESSORS_NU

python3 src/plot.py

echo "Deleting..."
rm prog7 timing.txt
```

5.2 lab7.c

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


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

    // Initialization of thread rank and number of threads.
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Checking number of threads for MPI.
    int threads_number = atoi(argv[3]);
       if (!rank)
        if (threads_number != size) {
            fprintf(stderr, "Error! Invalid number of threads.\n");
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }

    // Checking   ondition   between number of threads and number of processes.
    int processors_number = atoi(argv[4]);
    if (!rank)
        if (!(processors_number > threads_number)) {
            fprintf(stderr, "Error! Invalid number of processors.\n");
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_OTHER);
        }

    // Initialization number of processes for OpenMP.
    int n = processors_number / threads_number;

    // Initialization starting and ending indexes for each thread.
    int start_range = atoi(argv[1]);
    int end_range = atoi(argv[2]);
    int length_range = end_range - start_range;
    const int start_interval = rank * length_range / size + start_range;
    const int end_interval = (rank + 1) * length_range / size + start_range - 1;


    // Allocation of memory for timing.
    double *timing;
```

```
    if (!rank) {
        timing = (double *) malloc(size * sizeof(double));
        if (!timing) {
            fprintf(stderr, "Error! Failed to allocate memory.\n");
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
        }
    }

    // Allocation of memory for array.
    int *array;
    if (!rank) {
        array = (int *) malloc(length_range * sizeof(int));
        if (!array) {
            fprintf(stderr, "Error! Failed to allocate memory.\n");
            MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
        }
    }

    // Allocation of memory for chunk of array.
    int *chunk = (int *) malloc(length_range * sizeof(int));
    if (!chunk) {
        fprintf(stderr, "Error! Failed to allocate memory.\n");
        MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
    }


    // Start of timing for each thread.
    double start_time = MPI_Wtime();

    int flag;
    int counter = 0;
    int local_counter = 0;

    // Finding prime numbers in chunk of array.
#pragma omp parallel for num_threads(n) shared(flag, number) default(none)
    for (int number = start_interval; number <= end_interval; number++) {
        flag = 1;
        int max_divisor = sqrt(number);

        for (int d = 2; d <= max_divisor; d++)
            if (number % d == 0) {
                flag = 0;
                break;
            }

        if (flag) {
```

```
            chunk[local_counter] = number;
            local_counter++;
        }
    }

    // End of timing for each thread.
    double end_time = MPI_Wtime();
    double period = end_time - start_time;

    // Collecting all periods for each thread into single array - timing.
    MPI_Gather(&period, 1, MPI_DOUBLE, timing, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD)

    // Allocation of memory for counters.
    int *counters = (int *) malloc(size * sizeof(int));
    if (!counters) {
        fprintf(stderr, "Error! Failed to allocate memory.\n");
        MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
    }

    // Collecting all local_counters into single array - counters.
    MPI_Gather(&local_counter, 1, MPI_INTEGER, counters, 1, MPI_INTEGER, 0, MPI_C

    // Sending counters to each thread.
    MPI_Bcast(counters, size, MPI_INTEGER, 0, MPI_COMM_WORLD);

    // Allocation of memory for offsets.
    int *offsets = (int *) malloc(size * sizeof(int));
    if (!offsets) {
        fprintf(stderr, "Error! Failed to allocate memory.\n");
        MPI_Abort(MPI_COMM_WORLD, MPI_ERR_NO_MEM);
    }

    // Initialization offsets for each chunk.
    int tmp = 0;
    for (int i = 0; i < size; i++) {
        offsets[i] = tmp;
        tmp += counters[i];
    }

    // Initialization of total number of prime numbers.
    MPI_Reduce(&local_counter, &counter, 1, MPI_INTEGER, MPI_SUM, 0, MPI_COMM_WO

    // Collecting all chunks of array into single array.
    MPI_Gatherv(chunk, local_counter, MPI_INTEGER, array, counters, offsets, MPI

    // Output of result.
```

```
    if (!rank) {
        printf("Range of numbers: [%d; %d).\n", start_range, end_range);

        printf("Found prime numbers: ");
        for (int i = 0; i < counter; i++)
            printf("%d ", array[i]);
        printf("\n");

        FILE *fd = fopen("timing.txt", "w");
        fprintf(fd, "%d\n", size);
        for (int i = 0; i < size; i++)
            fprintf(fd, "%lf ", timing[i]);
    }

    // Free memory.
    free(chunk);
    free(counters);
    free(offsets);
    if (!rank) {
        free(timing);
        free(array);
    }

    // Termination MPI.
    MPI_Finalize();

    return 0;
}
```

5.3 plot.py

```
import matplotlib.pyplot as plt


def read_file(filename):
    with open(filename) as file:
        threads = [x + 1 for x in range(int(file.readline()))]
        times = [float(x) for x in file.read().split()]
    return threads, times


def draw_plots(threads, times):
    plt.title('Execution time', fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Time')
    plt.bar(threads, times)
    plt.savefig('images/histogram/histogram.png')
    plt.close()


if __name__ == '__main__':
    data = read_file('timing.txt')
    draw_plots(data[0], data[1])
```

6 Заключение

В результате проделанной работы мы получили, что при помощи сочетания MPI и OpenMP можно также получить эффективные параллельные программы. Однако, результаты работы данного алгоритма можно улучшить, если изменить принцип разделения рабочей области для каждого потока. Последние потоки всегда будут работать медленнее чем первые, потому что они имеют боллее большие числа, значит, потенциальных делителей у них больше и время работы возрастатет. Чтобы избежать этого необходимо добавлять каждый n-ый элемент в определённый поток, где n - количество потоков.