Лабораторная работа №4: Технология OpenMP. Особенности настройки

ИИКС ИБ
Б20-505
Барабанов Андрей

2022

Содержание

1 Рабочая среда **3**

2 Описание хода работы **4**

3 Графики **5**
3.1 Время 5
3.2 Ускорения 6
3.3 Эффективность 7

4 Программные коды **8**
4.1 build.sh 8
4.2 lab5.c 9
4.3 creating.c 11
4.4 lab1.c 13
4.5 main.py 15

5 Заключение **17**

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 MPI: 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

В данной лабораторной работе необходимо было разработать параллельный алгоритм поиска максимума в одномерном массиве с помощью технологии **MPI** и сравнить результаты с лабораторной работой №1, где этот алгоритм разрабатывался с помощью технологии **OpenMP**.

Алгоритм поиска максимума в одномерном массиве с помощью технологии **MPI** заключается в том, что массив длины N разбивается на равные промежутки, количество которых определяется заданным числом работающих потоков. В каждой подпоследовательности массива находится локальный максимум и с помощью опции **MPI_Reduce(&local_max, &max, 1, MPI_INTEGER, MPI_MAX, 0, MPI_COMM_WORLD** выбирается главный максимум.

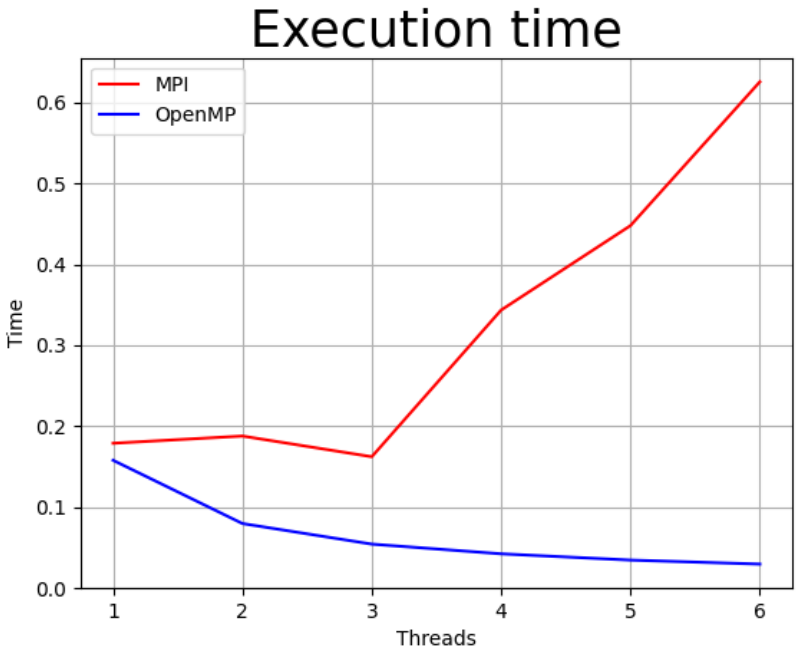
Так как число потоков, которые будут обрабатывать массив определяется на этапе выполнения команды **mpirun -np n -q ./prog5**, где **n** — количество потоков, то для получения экспериментальных результатов был написан bash-скрипт с циклами.

Для усреднения полученных результатов было расссмотрено 10 случайных массивов, которые сохрнялись файл для получения справедливых резуллътатов.

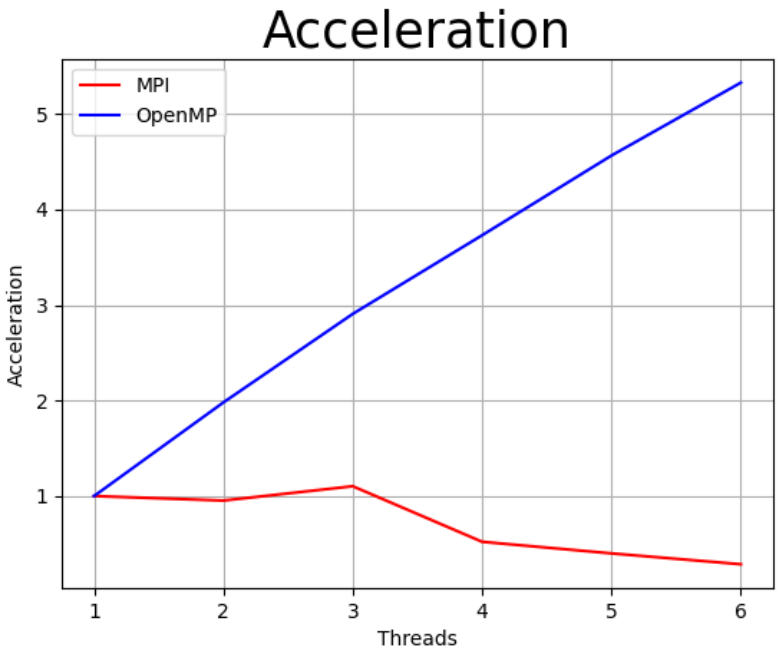
Графики сравнения времени, ускорения и эффективности двух различных технологий параллеьного программирования были построены с помощью библиотеки Python **matplotlib**.

3 Графики

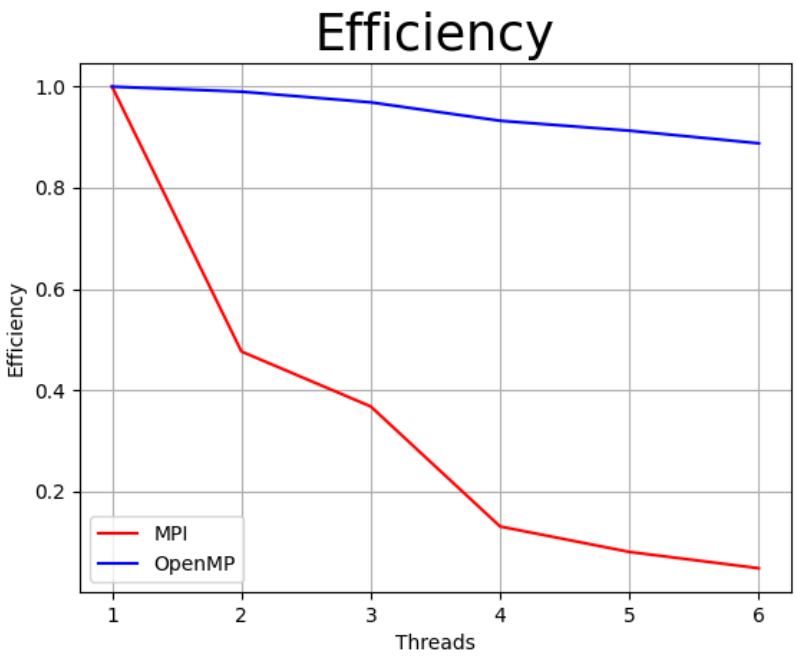
3.1 Время



3.2 Ускорения



3.3 Эффективность



4 Программные коды

4.1 build.sh

```
#!/usr/bin/zsh

echo "Compiling"

gcc -fopenmp -o prog1 src/lab1.c
gcc -o creating src/creating.c
mpicc -o prog5 src/lab5.c

echo "Working OpenMP..."
./prog1

echo "Working MPI..."
for ((i = 1; i <= 10; i++))
do
    ./creating 0
    for ((j = 1; j <= 6; j++))
        do
        mpirun -np $j -q ./prog5
        done
done
./creating 1

echo "Deleting"
rm prog1 creating prog5
```

4.2 lab5.c

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

#define ARRAY_SIZE 10000000

int main(int argc, char **argv) {
    int size = -1;
    int rank = -1;

    MPI_Init(&argc, &argv);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    int *array = (int *) malloc(ARRAY_SIZE * sizeof(int));

    if (!rank) {
        FILE *fd = fopen("array.txt", "r");
        for (int i = 0; i < ARRAY_SIZE; i++)
            fscanf(fd, "%d", &array[i]);
        fclose(fd);
    }

    MPI_Bcast(array, ARRAY_SIZE, MPI_INTEGER, 0, MPI_COMM_WORLD);

    const int start_index = rank * ARRAY_SIZE / size;
    const int end_index = (rank + 1) * ARRAY_SIZE / size;
    int max = -1;
    int local_max = -1;


    clock_t start_time = clock();

    for (int i = start_index; i < end_index; i++)
        if (array[i] > local_max)
                local_max = array[i];

    MPI_Reduce(&local_max, &max, 1, MPI_INTEGER, MPI_MAX, 0, MPI_COMM_WORLD);

    clock_t end_time = clock();

    free(array);
```

```
    MPI_Finalize();

    if (!rank) {
        FILE *fd = fopen("timing_MPI.txt", "a+t");

        fprintf(fd, "%lf ", (double)(end_time - start_time) / CLOCKS_PER_SEC);
        fclose(fd);
    }
}
```

4.3 creating.c

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

#define ARRAY_SIZE 10000000
#define TESTS_NUM 10
#define MAX_THREADS_NUM 6

int main(int argc, char **argv) {
    int option = atoi(argv[1]);
    switch (option)
    {
       case 0:
            srand(time(NULL));
            FILE *fd1 = fopen("array.txt", "w");
            for (int i = 0; i < ARRAY_SIZE; i++)
                fprintf(fd1, "%d ", rand() % ARRAY_SIZE);
            fclose(fd1);
            break;

        case 1:
            double *timing = (double *) malloc(sizeof(double) * MAX_THREADS_NUM
            FILE *fd2 = fopen("timing_MPI.txt", "r");
            //fread(timing, sizeof(double), MAX_THREADS_NUM * TESTS_NUM, fd2);
            fseek(fd2, 0, SEEK_SET);
            for (int i = 0; i < MAX_THREADS_NUM * TESTS_NUM; i++) {
                fscanf(fd2, "%lf", &timing[i]);
            }
            fclose(fd2);

            FILE *fd3 = fopen("timing_MPI.txt", "w");
            fprintf(fd3, "%d\n", MAX_THREADS_NUM);
            fprintf(fd3, "%d\n", TESTS_NUM);
            for (int i = 0; i < MAX_THREADS_NUM; i++) {
                for (int j = 0; j < TESTS_NUM; j++)
                    fprintf(fd3, "%lf\t", timing[6*j + i]);
                fprintf(fd3, "\n");
            }
            fclose(fd3);
            free(timing);
            break;
    }

    return 0;
```

}

4.4 lab1.c

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

#define MAX_THREADS_NUM 6
#define ARRAY_SIZE 10000000
#define TESTS_NUM 10

void func(int* array, int n) {
    int max = -1;
#pragma omp parallel num_threads(n) shared(array) reduction(max : max)
    {
#pragma omp for
        for(int i = 0; i < ARRAY_SIZE; i++)
            if (array[i] > max)
                max = array[i];
    }
}

int main() {
    double *timing = (double *) malloc(sizeof(double) * MAX_THREADS_NUM * TESTS_
    int *array = (int *) malloc(sizeof(int) * ARRAY_SIZE);
    for (int i = 0; i < TESTS_NUM; i++) {
        srand(time(NULL));
        for (int j = 0; j < ARRAY_SIZE; j++)
            array[j] = rand() % 10;

        for (int n = 0; n < MAX_THREADS_NUM; n++) {
            double start_time = omp_get_wtime();
            func(array, n + 1);
            double end_time = omp_get_wtime();
            timing[MAX_THREADS_NUM * i + n] = end_time - start_time;
        }
    }

    FILE *fd = fopen("timing_OpenMP.txt", "w");
    fprintf(fd, "%d\n", MAX_THREADS_NUM);
    fprintf(fd, "%d\n", TESTS_NUM);
    for (int i = 0; i < MAX_THREADS_NUM; i++) {
        for (int j = 0; j < TESTS_NUM; j++)
            fprintf(fd, "%lf\t", timing[MAX_THREADS_NUM * j + i]);
        fprintf(fd, "\n");
    }
```

```
    free(timing);
    free(array);
}
```

4.5 main.py

```
import matplotlib.pyplot as plt
from prettytable.colortable import ColorTable


def read_file(filename):
    with open(filename) as file:
        threads = [x + 1 for x in range(int(file.readline()))]
        tests = int(file.readline())
        times = [round((sum([float(x) for x in file.readline().split()]) / tests

    return threads, times

def draw_plots(data1, data2):
    plt.title("Execution time", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Time')
    plt.grid(True)
    plt.plot(data1[0], data1[1], 'r')
    plt.plot(data2[0], data2[1], 'b')
    plt.legend(['MPI', 'OpenMP'])
    plt.show()

    experimental_acceleration1 = [data1[1][0] / x for x in data1[1]]
    experimental_acceleration2 = [data2[1][0] / x for x in data2[1]]
    plt.title("Acceleration", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Acceleration')
    plt.grid(True)
    plt.plot(data1[0], experimental_acceleration1, 'r')
    plt.plot(data2[0], experimental_acceleration2, 'b')
    plt.legend(['MPI', 'OpenMP'])
    plt.show()

    experimental_efficiency1 = [experimental_acceleration1[x] / data1[0][x] for
    experimental_efficiency2 = [experimental_acceleration2[x] / data2[0][x] for
    plt.title("Efficiency", fontsize=25)
    plt.xlabel('Threads')
    plt.ylabel('Efficiency')
    plt.grid(True)
    plt.plot(data1[0], experimental_efficiency1, 'r')
    plt.plot(data1[0], experimental_efficiency2, 'b')
    plt.legend(['MPI', 'OpenMP'])
    plt.show()
```

```
if __name__ == '__main__':
    data1 = read_file("/home/bar1k/                    /lab5/timing_MPI.txt")
    data2 = read_file("/home/bar1k/                    /lab5/timing_OpenMP.tx
    draw_plots(data1, data2)
```

5 Заключение

В результате проделанной лабораторной работы мы получили, что OpenMP работает эффективнее чем MPI. Однако, чтоит отметить, что сравнивать технологии OpenMP и MPI не является разумной идеей, потому что при работе с MPI затрачивается огромное количество времни на пересылку массива, к тому же некоторые потоки в MPI могут быть заняты выполнением других приложений и программ, из-за этого невозможно эффективно расчитать время при работе на одной машине.