Лабораторная работа №6

"Коллективные операции в MPI"

Выполнил студент группы Б20-505
Сорочан Илья

Московский Инженерно-Физический Интститут
Москва 2023

1 Рабочая среда

Технические характеристики (вывод *inxi*):

```
CPU: 6-core AMD Ryzen 5 4500U with Radeon Graphics (-MCP-)
speed/min/max: 1396/1400/2375 MHz Kernel: 5.15.85-1-MANJARO x86_64 Up: 46m
Mem: 2689.5/7303.9 MiB (36.8%) Storage: 238.47 GiB (12.6% used) Procs: 238
Shell: Zsh inxi: 3.3.24
```

Используемый компилятор:

```
gcc (GCC) 12.2.0
```

Версия MPI:

```
Open MPI 4.1.4
```

Согласно официальной документации даная версия компилятора поддерживает *OpenMP 5.0* (необходимо для сравнения с первой лабораторной)

2 Реализация сортирвки Шелла с использованием *MPI*

Решить данную задачу я решил следующим образом:

1. Инициализация MPI и прочих необходимых значений;
2. Распределение частей массива между процессами;
3. Сортировка шеллом каждого фрагмента;
4. Объединение всех фрагментов;

Если с инициализацией и сортировкой все приблизительно понятно, то с другими этапами нет. Их я уточну далее.

2.1 Распределение массива

Для того что бы распределить массив равномерно между процессами был использовал *MPI_ Scatterv*. Она учитывает случаи, когда наш массив не делится на число процессов ровно, однако сложнее в применении.

После распределения фрагментов вызывается сортировка.

После сортировки фрагменты собираются во временный массив *tmp*. Он будет использоваться на следующем этапе. Непосредственно для сборки применялся *MPI_ Gatherv*.

2.2 Объединение фрагментов

Теперь все фрагменты объеденины в одном массиве, длинна которого равна начальному. Каждый фрагмент отсортирован, поэтому достаточно брать элементы с их “верхов“.

Тоесть по сути это одна итерация сортировки слиянием.

Из-за того, что число фрагментов является так же числом процессов, их тяжело объединять параллельно. Если каждому процессу назначить два фрагмента, то половина всех процессов будет простаивать.

Может показатся, что в качестве решения стоит делить каждый фрагмент пополам, однако таким образом фрагменты, которые нужно объединить лишь множатся.

Поэтому мной было принято решение произвести слияние в главном процессе.

2.3 Исходный код

Код программы:

```
/*
    The strategy:
    - Distribute array between every process
    - Shell sort it
    - Merge everything in one thread
*/
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include <time.h>
#include <limits.h>

void shellsort(int *array, int length) {
    for (int d = length / 2; d > 0; d /= 2)
        for (int i = d; i < length; ++i)
            for (int j = i - d; j >= 0 && array[j] > array[j + d]; j -= d) {
                int tmp = array[j];
                array[j] = array[j + d];
                array[j + d] = tmp;
            }
}

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

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

    // Init and clocks
    clock_t start, end;
    const int length = 1000000;
    int *array;

    if (!rank) {
        srand(920214);
        array = (int *)malloc(length * sizeof(int));
        for (int i = 0; i < length; ++i) array[i] = rand();
        start = clock();
    }

    // shellsort scatter
    int pad[size], len[size];
    int per_proc = length / size;
    int extra = length % size;

    int padding = 0;
    for (int pid = 0; pid < size; ++pid) {
        pad[pid] = padding;
        len[pid] = (pid < extra) ? per_proc + 1 : per_proc;
        padding += len[pid];
    }

    const int local_len = len[rank];
    int local_array[local_len];

    MPI_Scatterv(array, len, pad, MPI_INTEGER,\
    local_array, local_len, MPI_INTEGER, 0, MPI_COMM_WORLD);

    shellsort(local_array, local_len);

    int *tmp;
    if (!rank) {
        tmp = (int *)malloc(length * sizeof(int));
    }
    MPI_Gatherv(local_array, local_len, MPI_INTEGER,\
    tmp, len, pad, MPI_INTEGER, 0, MPI_COMM_WORLD);

    // merge everything in one proc
    if (!rank) {
        int idx[size];
        for (int i = 0; i < size; ++i) idx[i] = 0;
        int i = 0;
```

```
        while (1) {
            int pid = -1;
            int min = INT_MAX;
            for (int p = 0; p < size; ++p) {
                if (idx[p] >= len[p])
                    continue;
                int val = tmp[pad[p] + idx[p]];
                if (val < min) {
                    min = val;
                    pid = p;
                }
            }

            if (pid == -1)
                break;

            idx[pid]++;
            array[i] = min;
            i++;
        }

        free(tmp);
    }

    // print result
    if (!rank) {
        end = clock();

        // for (int i = 0; i < length; ++i)
        //     printf("%d ", array[i]);
        // puts("");

        const double CLOCKS_PER_MS = (double)CLOCKS_PER_SEC / 1000;
        double total = (double)(end - start) / CLOCKS_PER_MS;
        printf("%.3f", total);

        free(array);
    }

    MPI_Finalize();

    return 0;
}
```

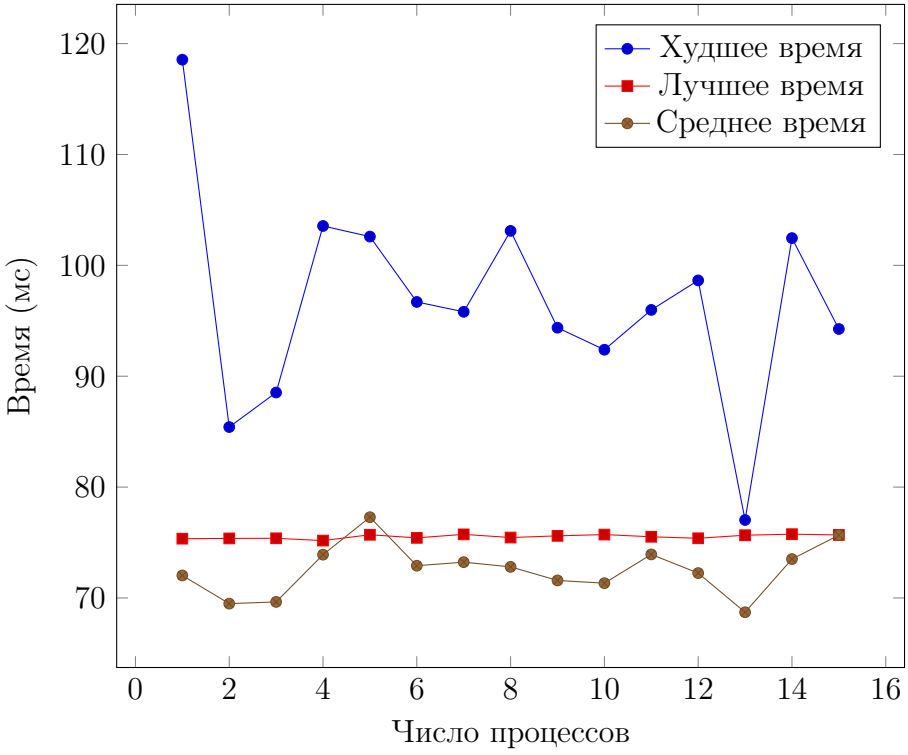
Дополнительный скрипт:

```
# This script compiles main.c with different number of threads
# and collects data to data.csv file
# format: worst,best,average
import os
import subprocess
import csv
import sys

# important constants
RUNS_PER_THREADS = 10
THREADS_LIMIT = 16

# compile with threads
def compile():
    os.system("mpicc main.c -o main")

def compile_opt():
    os.system("mpicc main.c -O3 -o main")

# capture worst, best and average
def run(threads):
    data = []
    for _ in range(RUNS_PER_THREADS):
        proc = subprocess.run(["mpirun", "main", "-c", str(threads), "-mca",\
            "opal_warn_on_missing_libcuda", "0"], capture_output=True, text=True)
        data.append(float(proc.stdout))

    worst = data[0]
    best = data[0]
    s = 0
    for val in data[1:]:
        if val > worst:
```

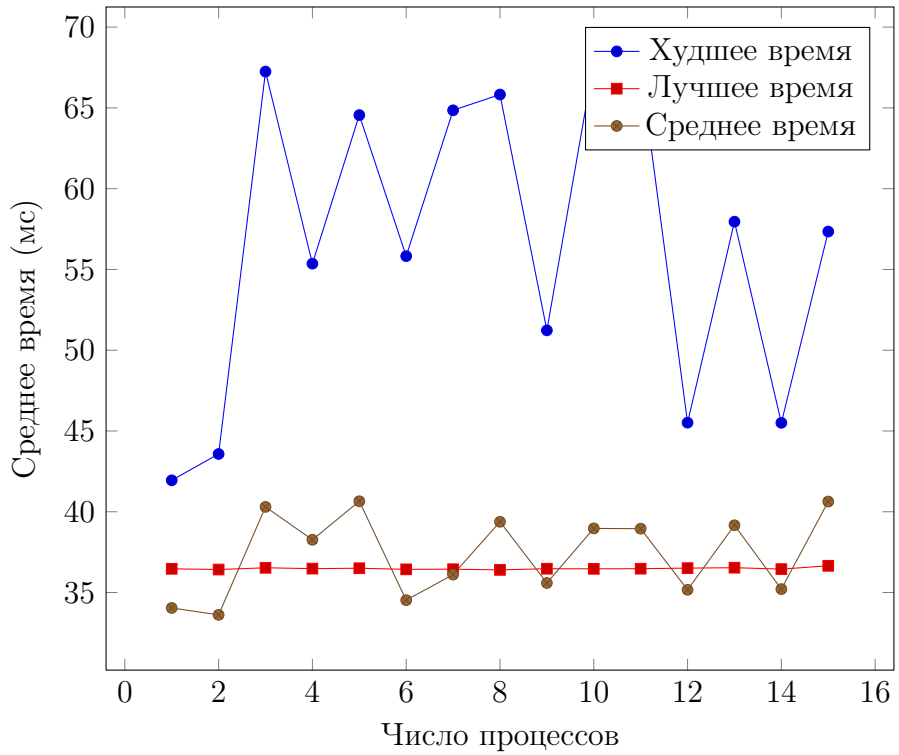
```
            worst = val
        if val < best:
            best = val
        s += val
    return (worst, best, s / len(data))

def main():
    if len(sys.argv) < 2 or sys.argv[1] != 'opt':
        comp = compile
    else:
        comp = compile_opt
    comp()

    if len(sys.argv) >= 3:
        file = sys.argv[2]
    else:
        file = "data.csv"

    with open(file, "w") as data:
        writer = csv.writer(data)
        writer.writerow(["Threads", "Worst (ms)", "Best (ms)", "Average (ms)"])
        for threads in range(1, THREADS_LIMIT):
            print("Testing threads =", threads)

            writer.writerow([str(threads)] + \
                ["{:.3f}".format(val) for val in run(threads)])

if __name__ == "__main__":
    main()
```

3 Экспериментальные данные

Во всех измерениях бралось 10 запусков на поток.

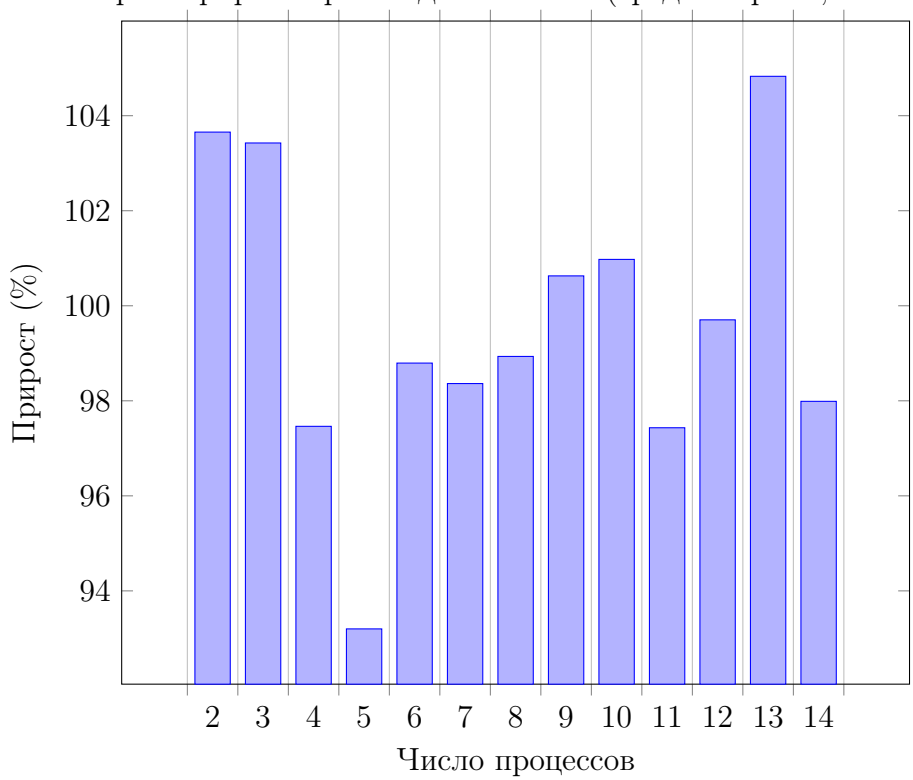


Из графика видно, что *MPI* достигает ускорения за счет хорошей производительности в худших случаях. Лучшее время у них почти одинаково. Взглянем на графики с оптимизациями:



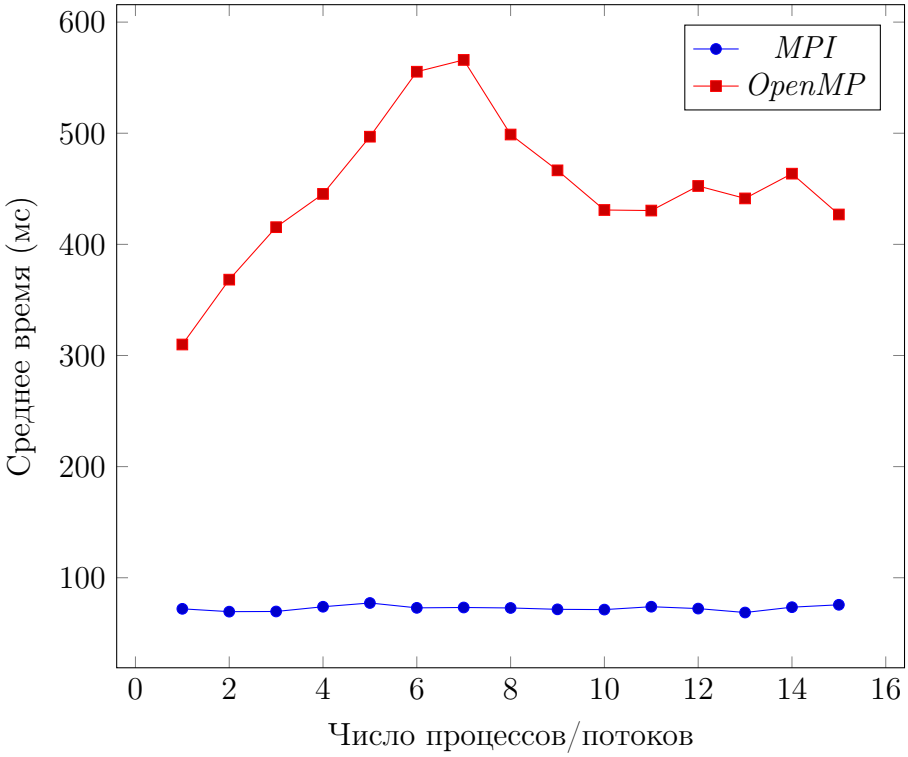
Хочу обратить внимание, что с оптимизациями *MPI* проигрывает однопоточному запуску. Возможно алгоритм все еще слишком прост, как было в предыдущих лабораторных.

Рассмотрим прирост производительности (среднее время; без оптимизаций):



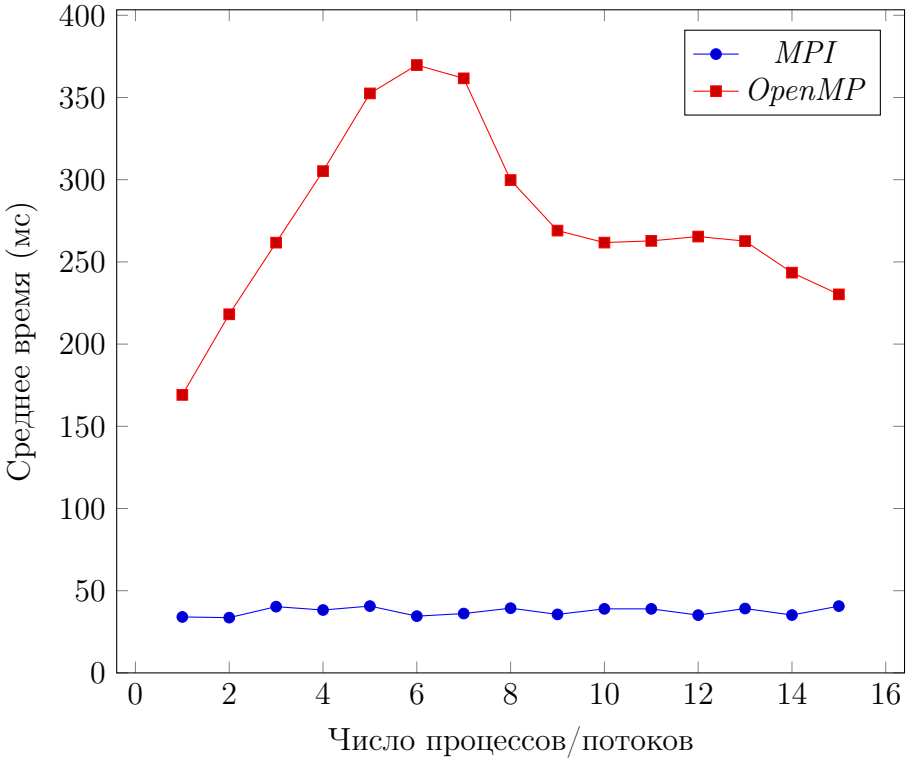
4 Сравнение с *OpenMP*

Сравним *MPI* и *OpenMP*:



MPI выигрывает с большим отрывом и при этом, практически не теряет в производительности при увеличении числа процессов. Хоть разработка программы на *MPI* была сложнее и заняла больше времени она оказалась производительнее.

Самое удивительное это то, что ситуция не меняется с применением оптимизаций:



5 Заключение

В данной работе была разработана параллельноя версия сортировки Шелла с использованием *MPI*. Был написан специальный скрипт для сбора данных. Осуществлено сравнение скорости исполнения версии программы использующей *MPI* и её аналогом из третьей лабораторной работы, использующий *OpenMP*.

В ходе работы было выяснено, что разработка с использованием *MPI* занимает гораздо больше времени, но является эффективнее. Автор так же допускает возможность неэффективной параллелизации - что еще может повысить результаты *MPI*.

С другой стороны следует отметить, что в третьей лабораторной работе не устанавливалось распределение нагрузки (флаг *schedule*), что может подкорректировать результаты *OpenMP* в лучшую сторону.