

Лабораторная работа №4

“Технология OpenMP. Особенности настройки”

Выполнил студент группы Б20-505

Сорочан Илья

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 данная версия компилятора поддерживает *OpenMP 5.0* (необходимо для сравнения с первой лабораторной)

2 Работа с *MPI*

Стоит отметить, что в используемой мной среде для компиляции программ, поддерживающих *MPI* обязательно использование специального компилятора *mpicc*.

Вывод программы в однопоточном режиме:

```
MPI Comm Size: 1;
MPI Comm Rank: 0;
Processor #0 has array: 788159773 2052308573 1377030627 1699618045 676203154 299802456 1767965774 1838448927 1686836254 1335355396
Processor #0 checks items 0 .. 9;
Processor #0 reports local max = 2052308573;

*** Global Maximum is 2052308573;
MPI Finalize returned (0);
```

Вывод программы в многопоточном режиме при запуске с 4-мя процессами:

```
MPI Comm Size: 4;
MPI Comm Rank: 2;
MPI Comm Size: 4;
MPI Comm Rank: 3;
MPI Comm Size: 4;
MPI Comm Rank: 0;
Processor #0 has array: 788159773 2052308573 1377030627 1699618045 676203154 299802456 1767965774 1838448927 1686836254 1335355396
Processor #0 checks items 0 .. 1;
Processor #0 reports local max = 2052308573;
Processor #3 has array: 788159773 2052308573 1377030627 1699618045 676203154 299802456 1767965774 1838448927 1686836254 1335355396
Processor #3 checks items 7 .. 9;
Processor #3 reports local max = 1838448927;
MPI Comm Size: 4;
MPI Comm Rank: 1;
Processor #1 has array: 788159773 2052308573 1377030627 1699618045 676203154 299802456 1767965774 1838448927 1686836254 1335355396
Processor #1 checks items 2 .. 4;
Processor #1 reports local max = 1699618045;
Processor #2 has array: 788159773 2052308573 1377030627 1699618045 676203154 299802456 1767965774 1838448927 1686836254 1335355396
Processor #2 checks items 5 .. 6;
Processor #2 reports local max = 1767965774;
MPI Finalize returned (0);

*** Global Maximum is 2052308573;
MPI Finalize returned (0);
MPI Finalize returned (0);
MPI Finalize returned (0);
```

3 Скорость *MPI*

3.1 Программные коды

Использовался код программы:

```

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

int main(int argc, char** argv)
{
    int ret = -1;    ///< For return values
    int size = -1;   ///< Total number of processors
    int rank = -1;   ///< This processor's number

    const int count = 1e1; ///< Number of array elements
    const int random_seed = 920215; ///< RNG seed
    clock_t start, end;

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

    /* Initialize the MPI */
    ret = MPI_Init(&argc, &argv);

    /* Determine our rank and processor count */
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* Allocate the array */
    array = (int*)malloc(count * sizeof(int));

    /* Master generates the array and starts the timer */
    if (!rank) {
        /* Initialize the RNG */
        srand(random_seed);
        /* Generate the random array */
        for (int i = 0; i < count; i++) { array[i] = rand(); }

        start = clock();
    }
    /* Send the array to all other processors */
    MPI_Bcast(array, count, MPI_INTEGER, 0, MPI_COMM_WORLD);

    const int wstart = (rank) * count / size;
    const int wend = (rank + 1) * count / size;

    for (int i = wstart;
         i < wend;
         i++)
    {
        if (array[i] > lmax) { lmax = array[i]; }
    }

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

    if (!rank) {
        end = clock();
        const double CLOCKS_PER_MS = (double)CLOCKS_PER_SEC / 1000;
        double total = (double)(end - start) / CLOCKS_PER_MS;
        printf("%.3f", total);
    }

    ret = MPI_Finalize();
}

```

```

        return 0;
    }

```

И небольшой скрипт для сбора данных в *.csv*:

```

# 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)

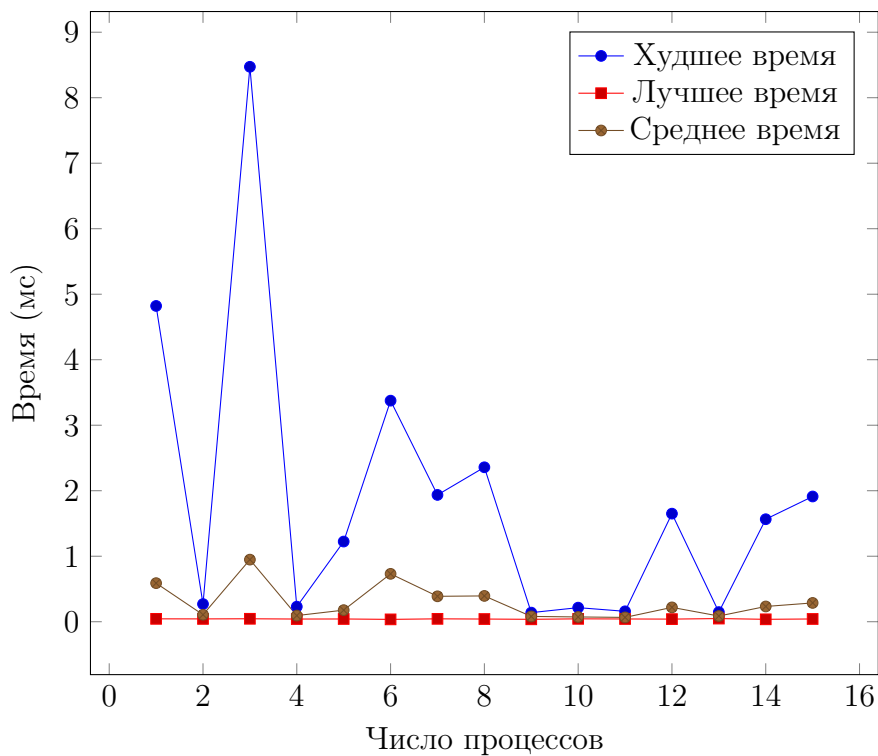
            writer.writerow([str(threads)] + \
                [ "{:.3f}".format(val) for val in run(threads)])

```

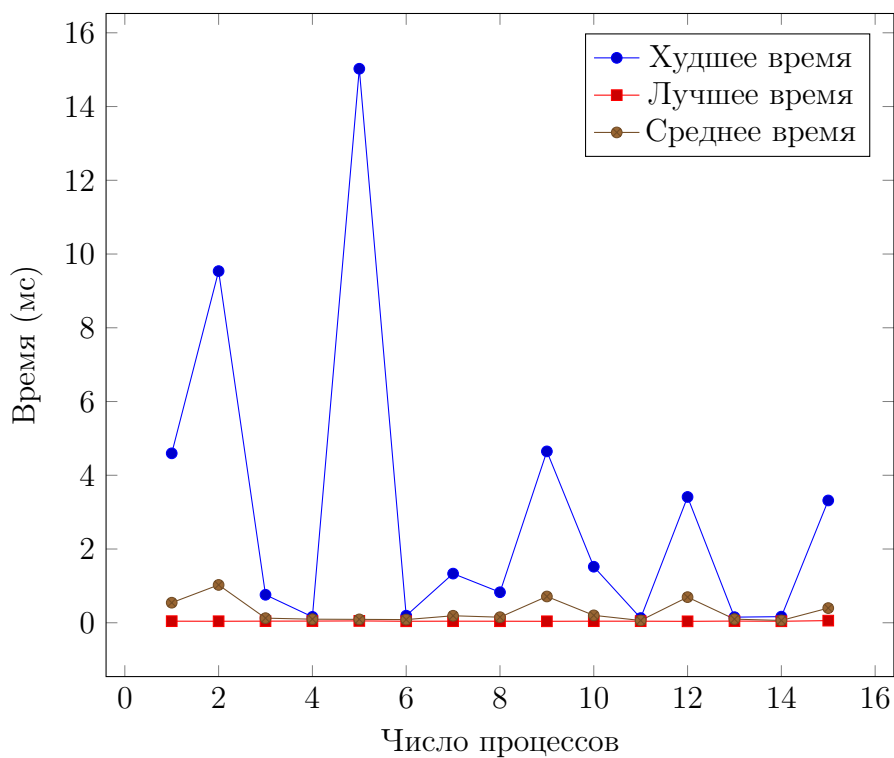
```
if __name__ == "__main__":
    main()
```

3.2 Экспериментальные данные

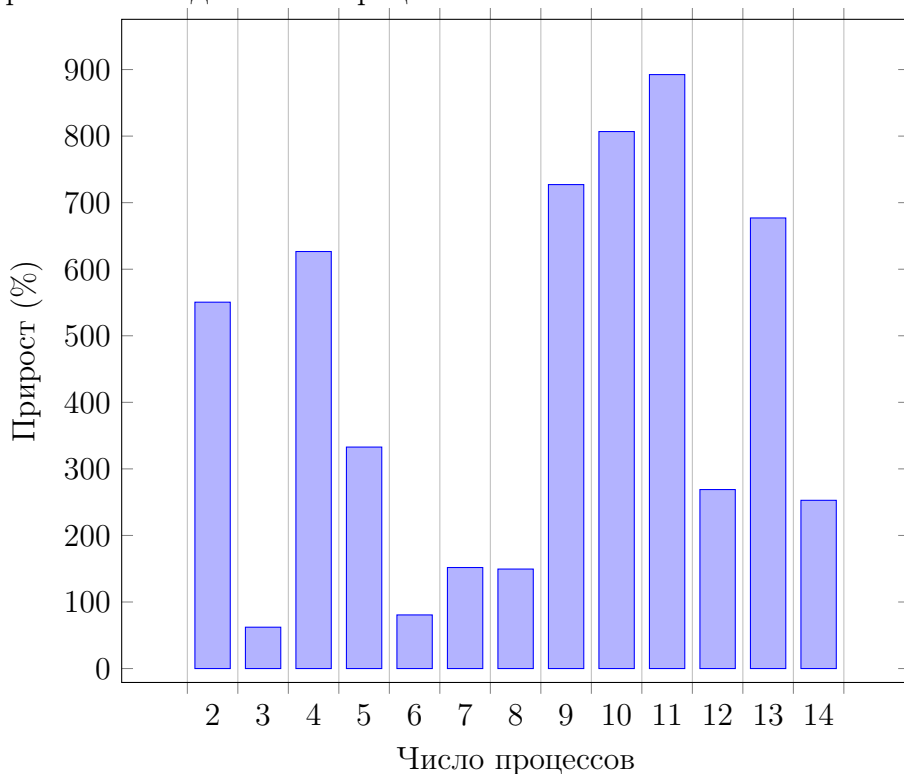
В программе так же использовалось по 10 запусков на количество процессов.



Видно, что в среднем программа работает быстрее, хоть и пик её производительности остается прежним. Рассмотрим случай с оптимизациями (аналогично первой лабораторной):

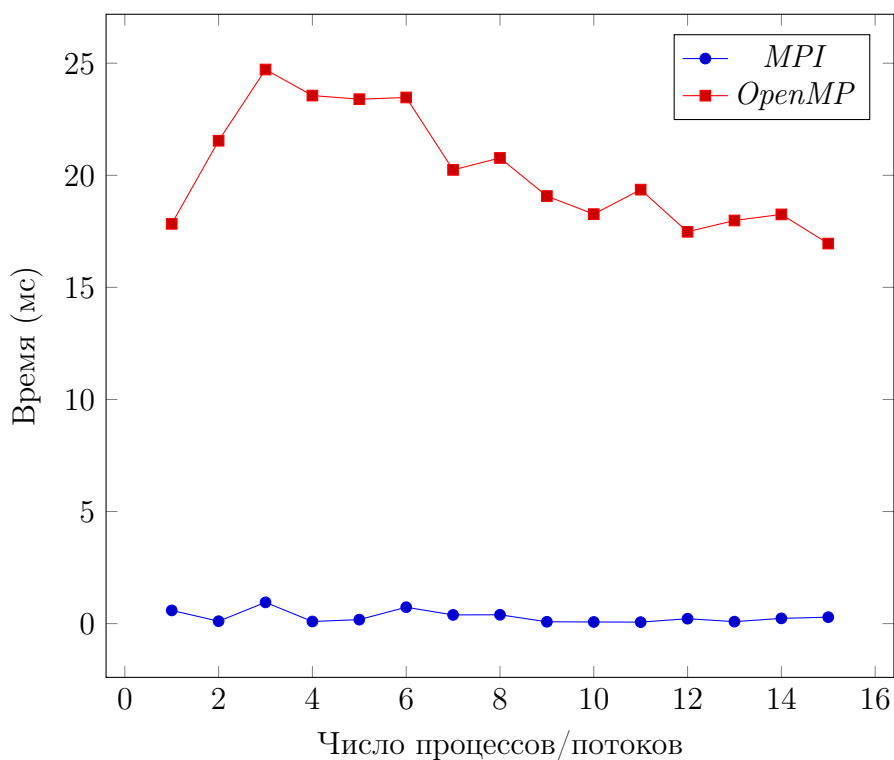


Прирост производительности в *MPI* в отличие от *OpenMP* положителен. Эффективность прироста за каждое число процессов относительно 1:



4 Сравнение *MPI* и *OpenMP*

Сравним время, для различного числа процессов/потоков:



Неоспоримое превосходство MPI (используются результаты из первой рабораторной)

5 Заключение

В данной работе была исследована работа с *MPI*. Был написан скрипт, собирающий информацию о времени исполнения программы.

В ходе работы было выяснено, что программа, написанная в первой лабораторной работе с применением *OpenMP* выполняется гораздо медленнее.

Несмотря на это в прошлой лабораторной мной были изучены типы распределения нагрузки, которые возможно помогли бы приблизить результаты *OpenMP* к результатам *MPI*.