Лабораторная работа №5

"Технология *MPI*. Введение"

Выполнил студент группы Б20-505
Сорочан Илья

Московский Инженерно-Физический Интститут
Москва 2023

1 Рабочая среда

Технические характеристики:
CPU: *6-core AMD Ryzen 5 4500U*
Kernel: *5.15.85-1-MANJARO x86_64*
Mem: *7303.9 MiB*

Используется:
Компилятор: *GCC 12.2.0*
OpenMP: *4.5*
MPI: *4.1.4*

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:

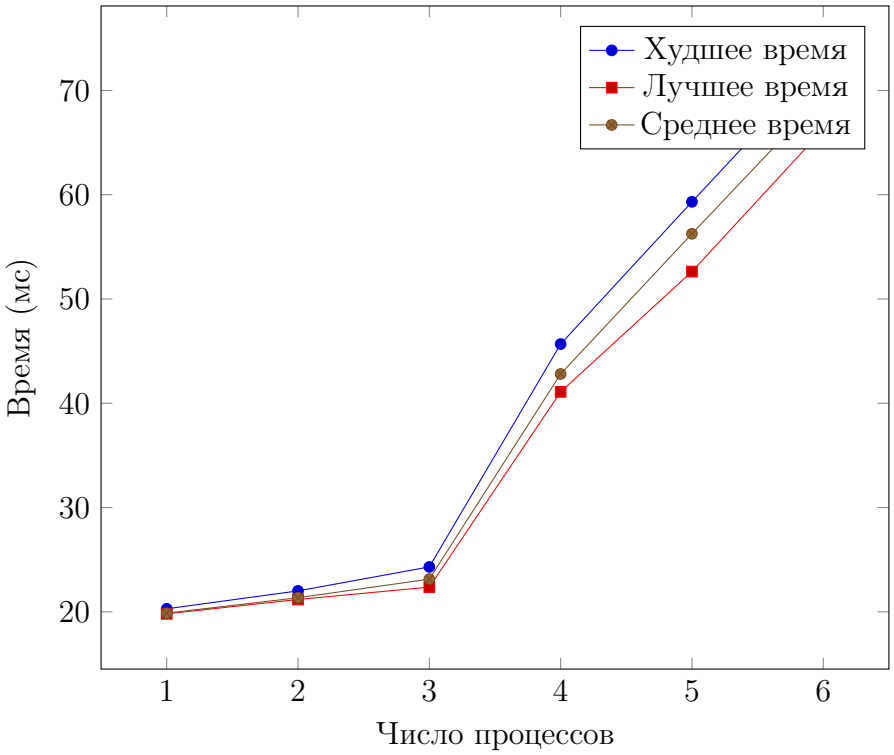
```
MPI Comm Size: 4;
MPI Comm Rank: 2;
MPI Comm Size: 4;
MPI Comm Rank: 3;
MPI Comm Size: 4;
MPI Comm Rank: 0;
Processor #0 has array: 788159773 2052308573 1377030627 1699618045 676203154 299802456 1767965774 1838448927 1686836254 1335355396
Processor #0 checks items 0 .. 1;
Processor #0 reports local 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 Экспериментальные данные

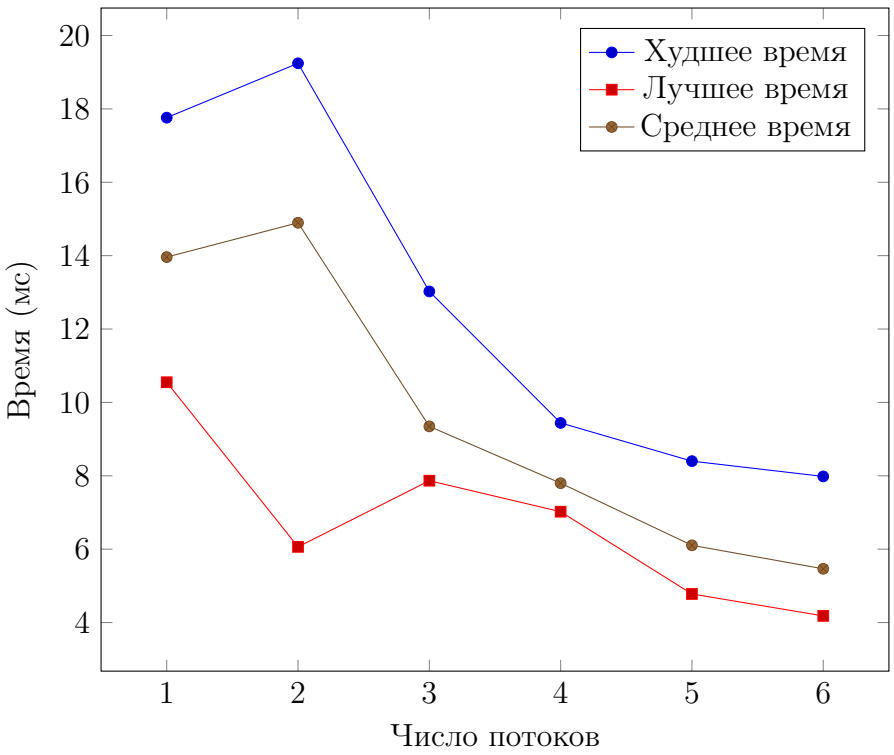
Код из первой лабораторной был немного изменен: было добавлено константное объявление значений сидов для генератора псевдослучайных чисел. Так же в программах использовалось 6 потоков/процессов.

3.1 Результаты выполнения

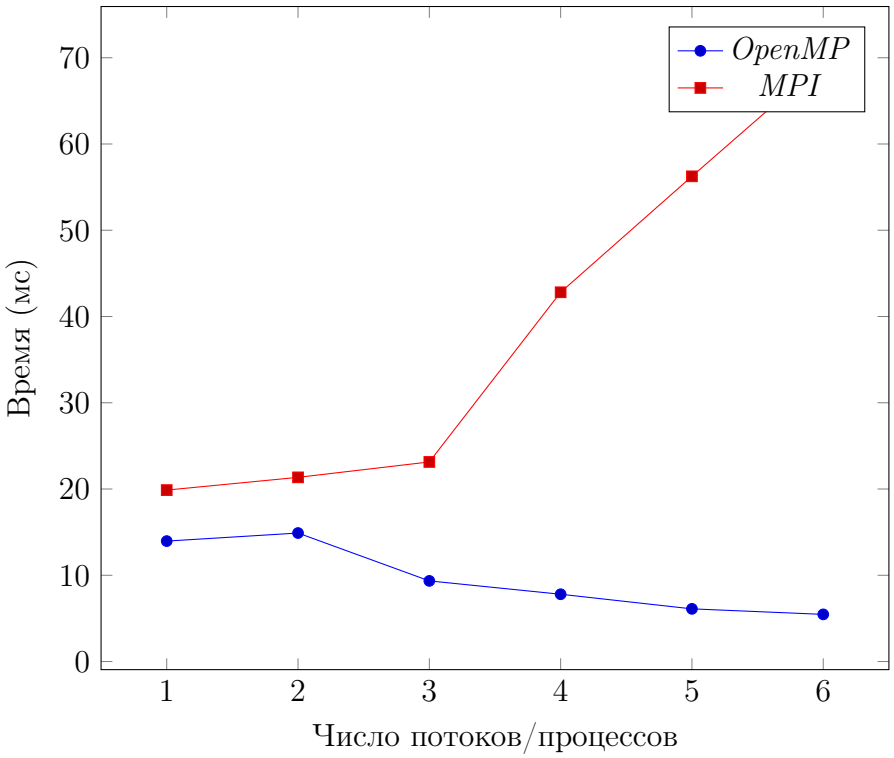
Рассмотрим результаты выполнения *MPI*:



Так же для чистоты эксперимента я решил обновить данные первой лабораторной работы:



Заметно ухудшение при использовании *MPI*:



4 Заключение

В данной работе было исследовано ускорение, получаемое при использовании технологии *MPI* в задании о поиске максимума. Была усовершенствована предоставленная программа и собраны данные. Так же был написан скрипт для сбора данных *MPI*. Оформлен отчет.

В ходе работы было выяснено, что применение *MPI* в данной задаче негативно сказывается на временных показателях. Могу предположить, что причиной является затратная по времени операция пересылки массива.

Приложение А

Использованные программные коды

Для проверки версии *OpenMP* использовался следующий код:

```
// Print openmp version

#include <stdio.h>

#if _OPENMP == 200505
#define _OPENMP_VERSION "2.5"
#elif _OPENMP == 200805
#define _OPENMP_VERSION "3.0"
#elif _OPENMP == 201107
#define _OPENMP_VERSION "3.1"
#elif _OPENMP == 201307
#define _OPENMP_VERSION "4.0"
#elif _OPENMP == 201511
#define _OPENMP_VERSION "4.5"
#elif _OPENMP == 201811
#define _OPENMP_VERSION "5.0"
#elif _OPENMP == 202011
#define _OPENMP_VERSION "5.1"
#else
#define _OPENMP_VERSION "unknown"
#endif

int main(int argc, char** argv) {
    printf("OpenMP Version: %s\n", _OPENMP_VERSION);

    return 0;
}
```

Код, использовавшийся для проверки функциональности *MPI*

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

        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);
        if (!rank) { printf("MPI Init returned (%d);\n", ret); }

        /* Determine our rankand processor count */
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        printf("MPI Comm Size: %d;\n", size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        printf("MPI Comm Rank: %d;\n", rank);

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

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

```
        }

        //printf("Processor #%d has array: ", rank);
        //for (int i = 0; i < count; i++) { printf("%d ", array[i]); }
        //printf("\n");

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

        printf("Processor #%d has array: ", rank);
        for (int i = 0; i < count; i++) { printf("%d ", array[i]); }
        printf("\n");

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

        printf("Processor #%d checks items %d .. %d;\n", rank, wstart, wend - 1);

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

        printf("Processor #%d reports local max = %d;\n", rank, lmax);

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

        ret = MPI_Finalize();
        if (!rank) {
                printf("\n*** Global Maximum is %d;\n", max);
        }
        printf("MPI Finalize returned (%d);\n", ret);

        return(0);
}
```

Для измерения времени исполнения программы с использованием *OpenMP* использовался следующий код(выводит *csv* в стандартный вывод):

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

#define RUNS_PER_THREAD 20

const int N = 10000000;
const int MAX_THREADS = 6;

const int SEED[RUNS_PER_THREAD] = {
    788159773,
    2052308573,
    1377030627,
    1699618045,
    676203154,
    299802456,
    1767965774,
    1838448927,
    1686836254,
    1335355396,
    1186224,
    74147217,
    898646163,
    106881286,
    10633766,
    1364600012,
    305070600,
    1539146084,
    822350517,
    875518628,
};

void randArr(int *array, int size) {
    for (int i = 0; i < size; ++i)
        array[i] = rand();
}

// run algo and return time elapsed
double run(const int threads, int *array, const int size) {
    double start = omp_get_wtime();
    int max = -1;
    #pragma omp parallel num_threads(threads) shared(array, size) reduction(max: max) default(none)
    {
        #pragma omp for
        for(int i = 0; i < size; ++i) {
            if(array[i] > max) {
                max = array[i];
            };
        }
    }
    double end = omp_get_wtime();
    return (end - start) * 1000;
}

int main(int argc, char **argv) {
    int *array = (int *)malloc(N * sizeof(int));

    puts("Threads,Worst (ms),Best (ms),Avg (ms)");

    for (int threads = 1; threads < MAX_THREADS + 1; ++threads) {
        double sum = 0, max_time = -1, min_time = 100000;
        for (int i = 0; i < RUNS_PER_THREAD; ++i) {
            // gen array with special seed
```

```
            srand(SEED[i]);
            randArr(array, N);

            // calc value
            double time = run(threads, array, N);
            if (time > max_time)
                max_time = time;
            if (time < min_time)
                min_time = time;
            sum += time;
        }

        printf("%d,%.3f,%.3f,%.3f\n", threads, max_time, min_time, sum / RUNS_PER_THREAD);
    }

    free(array);

    return 0;
}
```

Для измерения времени исполнения программы с использованием *MPI* использовался следующий код(выводит *csv* в стандартный вывод):

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

#define RUNS_PER_THREAD 20

const int N = 10000000;
const int MAX_THREADS = 6;

const int SEED[RUNS_PER_THREAD] = {
    788159773,
    2052308573,
    1377030627,
    1699618045,
    676203154,
    299802456,
    1767965774,
    1838448927,
    1686836254,
    1335355396,
    1186224,
    74147217,
    898646163,
    106881286,
    10633766,
    1364600012,
    305070600,
    1539146084,
    822350517,
    875518628,
};

void randArr(int *array, int size) {
    for (int i = 0; i < size; ++i)
        array[i] = rand();
}

// run algo and return time elapsed
double run(const int threads, int *array, const int size) {
    double start = omp_get_wtime();
    int max = -1;
    #pragma omp parallel num_threads(threads) shared(array, size) reduction(max: max) default(none)
    {
        #pragma omp for
        for(int i = 0; i < size; ++i) {
            if(array[i] > max) {
                max = array[i];
            };
        }
    }
    double end = omp_get_wtime();
    return (end - start) * 1000;
}

int main(int argc, char **argv) {
    int *array = (int *)malloc(N * sizeof(int));

    puts("Threads,Worst (ms),Best (ms),Avg (ms)");

    for (int threads = 1; threads < MAX_THREADS + 1; ++threads) {
        double sum = 0, max_time = -1, min_time = 100000;
        for (int i = 0; i < RUNS_PER_THREAD; ++i) {
            // gen array with special seed
            srand(SEED[i]);
            randArr(array, N);

            // calc value
            double time = run(threads, array, N);
            if (time > max_time)
                max_time = time;
            if (time < min_time)
                min_time = time;
            sum += time;
        }

        printf("%d,%.3f,%.3f,%.3f\n", threads, max_time, min_time, sum / RUNS_PER_THREAD);
    }

    free(array);

    return 0;
```

```
}
```

А так же для этой цели использовался скрипт:

```
import os, sys, subprocess

MAX_PROCS = 6
RUNS_PER_PROC = 20

def run(procs, run_id):
    proc = subprocess.run(["mpirun", "-c", str(procs), "-mca",\
    "opal_warn_on_missing_libcuda", "0", "main", str(run_id)], capture_output=True, text=True)
    return float(proc.stdout)

os.system("mpicc mpi_main.c -o main")

print("Threads,Worst (ms),Best (ms),Avg (ms)")

for procs in range(MAX_PROCS):
    sum = 0
    worst = 0
    best = 100000
    for run_id in range(RUNS_PER_PROC):
        time = run(procs + 1, run_id)
        if time > worst:
            worst = time
        if time < best:
            best = time
        sum += time

    print(procs + 1, worst, best, "{:.3f}".format(sum / RUNS_PER_PROC), sep=',')
```

Приложение Б

Таблицы с практическими результатами

OpenMP:

Threads	Worst (ms)	Best (ms)	Avg (ms)
1	17.76	10.55	13.96
2	19.24	6.06	14.9
3	13.03	7.87	9.35
4	9.44	7.02	7.8
5	8.4	4.78	6.11
6	7.98	4.18	5.47

MPI:

Threads	Worst (ms)	Best (ms)	Avg (ms)
1	20.29	19.81	19.87
2	22.01	21.19	21.35
3	24.31	22.37	23.14
4	45.68	41.09	42.81
5	59.32	52.63	56.25
6	72.8	66.27	69.53