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:
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 Реализация сортировки Шелла с использованием *MPI*

2.1 Параллелизация алгоритма

Цикл по различным d будет выполнятся в главном процессе. При этом на каждой своей итерации он будет рассылать фрагменты массива другим потокам. Есть два способа делать это:

- Брать последовательные фрагменты массива;
- Брать d-е элементы относительно i.

2.2 Последовательные фрагменты

Суть данного способа – разделить массив на (почти) равные части между процессами и произвести сортировку на каждом. У данного способа есть 2 существенные проблемы:

1. Размер массива может ровно не делиться на количество процессов;
2. Фрагменты массива при получении главным потоком нужно дополнительно отсортировать. Простейшая сортировка слиянием.

Первая проблема нивелируется тем, что если число процессов невелико, то оставшиеся элементы можно отдать одному из них не потеряв сильно в производительности.

Вторая проблема гораздо существеннее. После параллельной сортировки сами фрагменты отсортированы, но не относительно друг-друга. В качестве решения можно произвести сортировку слиянием.

Её тоже можно сделать параллельно, но я затрудняюсь это реализовать.

2.3 d-е элементы

С помощью векторного типа *MPI* можно передавать d-е элементы. Тогда сортировка в подпроцессах превращается в обычную сортировку слиянием.

Однако этот подход так же не лишен проблем:

- Количество и длинна под-векторов, на которые необходимо разбивать меняется каждую итерацию внешнего цикла по d из-за чего произвести равномерное распределение сложнее;
- Если одному процессу передается несколько под-векторов, то не ясно в каком порядке он их вернет.

Вторую проблему можно решить путем задания определенного порядка отправления и возврата.

Первая проблема даже при хорошем распределении останется проблемой. К тому же пересылка большого объема данных звучит не очень хорошо.

2.4 Выбранный алгоритм

Учитывая вышеописанные минусы различных подходов я решил остановится на первом методе.

По своей сути второй метод совершает больше пересылок, когда как первый не использует все процессы в конце.

2.5 Исходный код

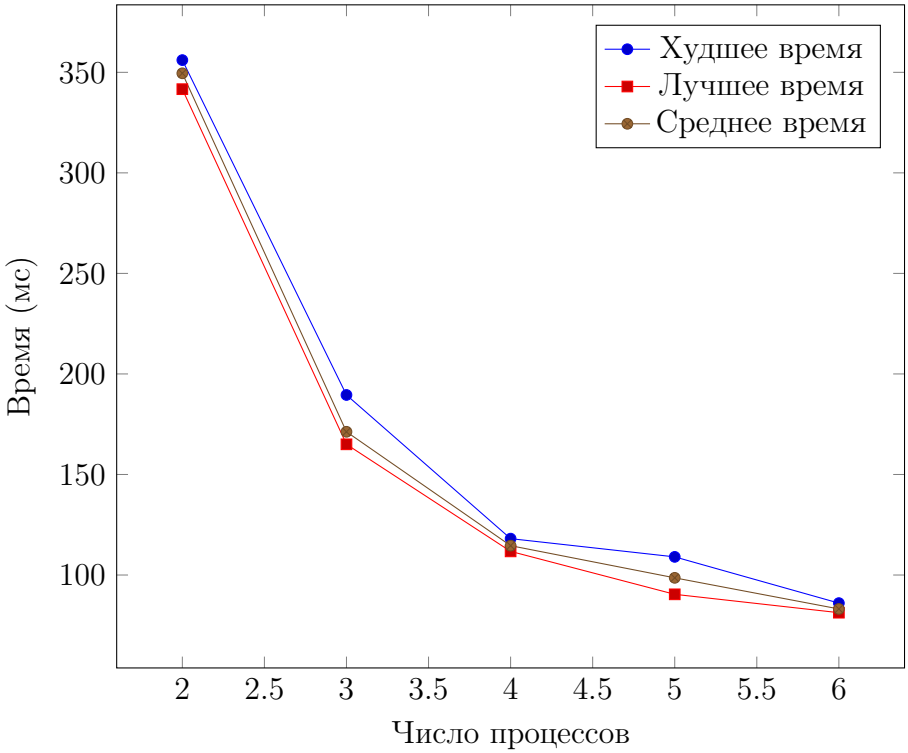
Полный исходный код предоставлен в приложении А.

3 Экспериментальные данные

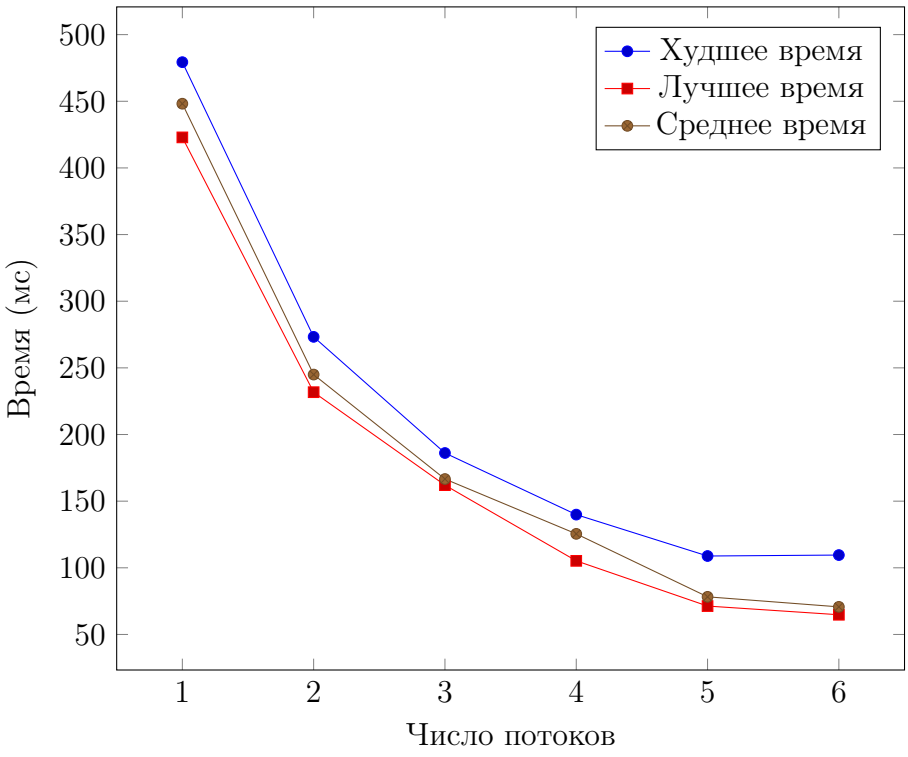
В программах использовалось до 6 потоков/процессов и 10 запусков на поток/процесс.

3.1 Результаты выполнения

Рассмотрим результаты выполнения *MPI*:

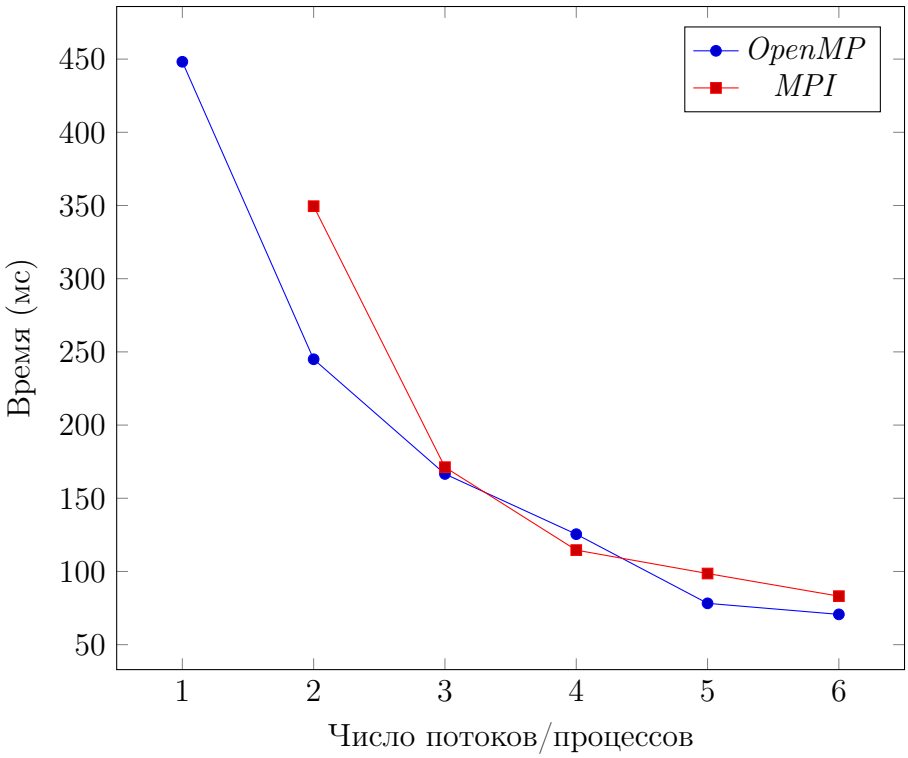


Так же для чистоты эксперимента я решил обновить данные первой лабораторной работы:



В отличии от предыдущей лабораторной положительные тенденции заметны не только у

OpenMP:



При 6-ти потоках их производительность в данной задаче можно считать одинаковой. Однако разработка программы на *MPI* заняла гораздо больше времени и ресурсов.

4 Заключение

В данной работе была реализована сортировка Шелла при помощи технологии *MPI*. Проведено сравнение с обновленными результатами *OpenMP*. Оформлен отчет.

В ходе работы было выяснено, что применение *MPI* в данной задаче оправдано. Впрочем из-за сложности данной технологии я все же предпочел бы *OpenMP*.

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;
}
```

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 10

const int N = 1000000;
const int MAX_THREADS = 6;

const int SEED[RUNS_PER_THREAD] = {
    788159773,
    2052308573,
    1377030627,
    1699618045,
    676203154,
    299802456,
    1767965774,
    1838448927,
    1686836254,
    1335355396,
};

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();
    for (int d = size / 2; d > 0; d /= 2) {
        const int cd = d;
        #pragma omp parallel for num_threads(threads) shared(array, size, cd) default(none)
        for (int i = 0; i < cd; ++i) {
```

```
            // insertion sort
            for (int j = cd + i; j < size; j += cd) {
                int key = array[j];
                int k = j - cd;

                while (k >= i && array[k] > key) {
                    array[k + cd] = array[k];
                    k -= cd;
                }
                array[k + cd] = key;
            }
        }
    }

    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 <limits.h>
#include <mpi.h>

const int N = 1000000;

#define RUNS_PER_PROC 10
const int SEED[RUNS_PER_PROC] = {
    788159773,
    2052308573,
    1377030627,
    1699618045,
    676203154,
    299802456,
    1767965774,
    1838448927,
    1686836254,
    1335355396,
};

int main(int argc, char** argv) {
    // Init MPI
    int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (argc != 2) {
        if (!rank)
            printf("Usage: %s [seed_id]\n", argv[0]);
        return MPI_Finalize();
    }

    int seed = atoi(argv[1]);
    if ((seed == 0 && argv[1][0] != '0') || seed < 0 || seed > RUNS_PER_PROC) {
        if (!rank)
            puts("Incorrect seed_id");
        return MPI_Finalize();
    }

    int *array;
    if (!rank) {
        array = malloc(N * sizeof(int));
        srand(SEED[seed]);
        for (int i = 0; i < N; ++i)
            array[i] = rand();
    }

    // if (!rank) {
    //     for (int i = 0; i < N; ++i)
    //         printf("%d ", array[i]);
    //     puts("");
    // }
```

```
    // set up proc slice length
    const int WORKERS = size - 1;
    int length = N / WORKERS;
    if (rank == size - 1)
            length += N % WORKERS;

    // local proc slice
    int *slice;
    if (rank) {
        slice = malloc(length * sizeof(int));
    }

    double start;
    if (!rank)
        start = MPI_Wtime();

    // parallel shell sort
    MPI_Status s;
    if (!rank) {
        // split array
        for (int p = 0; p < WORKERS - 1; ++p)
            MPI_Send(array + length * p, length, MPI_INTEGER, p + 1, 0, MPI_COMM_WORLD);
        const int pad = (WORKERS - 1) * length;
        MPI_Send(array + pad, N - pad, MPI_INTEGER, WORKERS, 0, MPI_COMM_WORLD);

        // collect array
        for (int p = 0; p < WORKERS - 1; ++p)
            MPI_Recv(array + length * p, length, MPI_INTEGER, p + 1, 0, MPI_COMM_WORLD, &s);
        MPI_Recv(array + pad, N - pad, MPI_INTEGER, WORKERS, 0, MPI_COMM_WORLD, &s);
    } else {
        // recv array
        MPI_Recv(slice, length, MPI_INTEGER, 0, 0, MPI_COMM_WORLD, &s);

        // sort
        for (int d = length / 2; d > 0; d /= 2)
            for (int i = d; i < length; ++i)
                for (int j = i - d; j >= 0 && slice[j] > slice[j + d]; j -= d) {
                    int temp = slice[j];
                    slice[j] = slice[j + d];
                    slice[j + d] = temp;
                }

        // send array
        MPI_Send(slice, length, MPI_INTEGER, 0, 0, MPI_COMM_WORLD);
    }

    // merge
    if (!rank) {
        int *idx = (int *)malloc(WORKERS * sizeof(int));
        for (int i = 0; i < WORKERS; ++i) idx[i] = 0;

        int *tmp = array;
        array = (int *)malloc(N * sizeof(int));

        for (int i = 0; i < N; ++i) {
            int pid = -1;
            int min = INT_MAX;

            // search all workers except last one
            for (int p = 0; p < WORKERS - 1; ++p) {
                if (idx[p] >= length)
                    continue;
                int val = tmp[idx[p] + length * p];
                if (val < min) {
                    min = val;
                    pid = p;
                }
            }

            // search last worker
            if (idx[WORKERS - 1] < length) {
                int val = tmp[idx[WORKERS - 1] + length * (WORKERS - 1)];
                if (val < min) {
                    min = val;
                    pid = WORKERS - 1;
                }
            }

            // set array to
            array[i] = min;
            idx[pid]++;
        }

        double end = MPI_Wtime();
        printf("%.3f", (end - start) * 1000);
        // for (int i = 0; i < N; ++i)
        //     printf("%d ", array[i]);
        // puts("");
    }

        return MPI_Finalize();
}
```

А так же для этой цели использовался скрипт:

```
import os, sys, subprocess

MAX_PROCS = 6
RUNS_PER_PROC = 10

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(1, 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	479.34	422.9	448.17
2	273.26	231.75	244.97
3	186.13	162.01	166.6
4	139.9	105.23	125.47
5	108.85	71.31	78.23
6	109.56	64.78	70.7

MPI:

Threads	Worst (ms)	Best (ms)	Avg (ms)
2	356.08	341.62	349.52
3	189.54	165	171.24
4	118.09	111.76	114.59
5	109.02	90.4	98.6
6	86.03	81.27	83.13