Ирин Сергей Георгиевич

M19-B01

Отчет по лабораторной работе №1.

Бизнес логика варианта 8:

Сформировать результирующий вектор как среднее по каждой строке исходной квадратной матрицы.

В данной работе рассматривается первый вариант исполнения, то есть последовательный.

Первоначальное исполнение поддерживает ввод-вывод через файлы, в дальнейшем планирую переделать на TCP.

Данные генерятся Python скриптом, который в качестве входных параметров принимает размер файла в мегабайтах и путь к нему (в случае отсутствия аргументов генерятся 5 файлов размером от 125Мб до 2Гб). Каждый файл содержит квадратную матрицу из чисел с плавающей точкой с восьмью цифрами после запятой. Код скрипта приведен на следующей странице.

```python
import sys, math
import numpy as np

# Random matrix file generator for
# Hybrid computing labs
# Usage:
# <script_name>.py [file_size_in_MegaBytes] [file_path]
# Produces file with square matrix of floats

def generate_five_files():
    '''
    Generated five files with
    matrices of sizes from 125Mb to 2Gb
    '''
    mega_bytes = 125
    while mega_bytes < 2001:
        generate_file_with_matrix([mega_bytes, str(mega_bytes)])
        mega_bytes *= 2
```

```python
def generate_file_with_matrix(sizeName):
    '''
    Function to generate file with
    random matrix of given size
    '''
    sizeInBytes = float(sizeName[0]) * (2**20) # 1 Mb = 2^20 bytes
    num_rows_and_cols = int(math.sqrt(sizeInBytes / 11)) # 11 byte chars
 for each number
    file_name = str(num_rows_and_cols)
    if (len(sizeName) > 1):
        file_name = sizeName[1]
    matrix = np.random.rand(num_rows_and_cols, num_rows_and_cols)
    np.savetxt(file_name + '.txt', matrix, fmt='%.8f', delimiter=' ')


if __name__ == "__main__":
    if len(sys.argv) > 1:
        generate_file_with_matrix(sys.argv[1:])
    else:
        generate_five_files()
```

Описание алгоритма выполнения бизнес-логики:

Программа, выполняющая бизнес-логику и производящая файл с результатом и временем его выполнения, написана на C.

Она состоит из трех вспомогательных функций:

1) double getMilliseconds() – возвращает текущее время в миллисекундах.
2) void trim_str(char* str) – обрезает все нечисельные символы.
3) long fsize(FILE* fp) – возвращает размер файла.

Верхнеуровневая бизнес-логика происходит в функции int process_file(char* file_path, FILE* out_file), которая принимает на вход путь к файлу, который надо обработать и пойнтер к файлу вывода. В ней открывается на чтение вводный файл, измеряется его размер и засекается время обработки файла и записи результата, вызывается функция, получающая результирующий вектор. Затем этот вектор и время исполнения записываются в выводной файл.

Логика получения результирующего вектора воплощена в функции char* get_file_vector(FILE* fp, long size). В ней построчно считываются строки матрицы из файла и рассчитываются средние значения для каждой строки, которые впоследствии склеиваются в строку, которая возвращается из этой функции.

Наконец, логика расчета среднего по строке воплощена в функции double get_line_avg(char* line). В ней строка, считанная из файла, разбивается на числа, которые суммируются, а затем делятся на их количество. Полученный результат возвращается из функции.

В данный момент в точке входа воплощена следующая логика: в качестве аргументов программа принимает путь к файлу ввода и путь к файлу вывода.

Код программы (ссылка на github, где его можно посмотреть https://github.com/RinSer/MephiHybridComputing/blob/master/Lab1/Lab1/main.cpp):

```
#include <cstdio>
#include <cstdlib>
#include <ctype.h>
#include <string.h>
#include <time.h>

int process_file(char* file_path, FILE* out_file);
double getMilliseconds();

int main(int argc, char* argv[])
{
    if (argc < 3)
    {
```

```
        printf("Need input file path and output file path arguments!");
        return -1;
    }

    FILE* out = fopen(argv[2], "w");

    double start = getMilliseconds();

    process_file(argv[1], out);

    double end = getMilliseconds();
    double total_execution_time_in_seconds = (double)(end - start);
    fprintf(out, "Total execution time: %.3f milliseconds",
total_execution_time_in_seconds);

    fclose(out);

    return 0;
}

double getMilliseconds() {
    return 1000.0 * clock() / CLOCKS_PER_SEC;
}

void trim_str(char* str)
{
    for (int i = 0; i < strlen(str); i++)
    {
        if (!isdigit(str[i]) && str[i] != '.')
        {
            str[i] = NULL;
            return;
        }
    }
}

long fsize(FILE* fp) {
    int initial = ftell(fp);

    fseek(fp, 0, SEEK_END);

    long size = ftell(fp);

    fseek(fp, initial, SEEK_SET);

    return size;
}

double get_line_avg(char* line)
{
    int count = 0;
    double sum = 0;
    char* number = strtok(line, " ");

    while (number != NULL)
    {
        trim_str(number);
        sum += atof(number);
        count++;
        number = strtok(NULL, " ");
    }
```

```
    return (double)sum / count;
}

char* get_file_vector(FILE* fp, long size)
{
    char* result = new char[size];
    char* line = NULL;
    size_t len = 0;

    while ((getline(&line, &len, fp)) != -1)
    {
        double avg = get_line_avg(line);
        char* avg_str = new char[size];
        sprintf(avg_str, "%.8f", avg);
        strcat(strcat(result, avg_str), " ");
    }

    return result;
}

int process_file(char* file_path, FILE* out_file)
{
    double start = getMilliseconds();

    FILE* fp = fopen(file_path, "r");
    if (fp == NULL)
        exit(EXIT_FAILURE);

    long file_size = fsize(fp);

    char* result = get_file_vector(fp, file_size);

    fclose(fp);

    fprintf(out_file, "%s\n", result);

    double end = getMilliseconds();
    double execution_time_in_seconds = (double)(end - start);
    printf("%d bytes in %.3f milliseconds\n", file_size,
execution_time_in_seconds);

    fprintf(out_file, "%d bytes in %.3f milliseconds\n", file_size,
execution_time_in_seconds);

    return 0;
}
```

График зависимости времени выполнения от размера данных (по горизонтали отложены байты, по вертикали – время исполнения в миллисекундах):