

**Отчет по лабораторной работе №2.**

Бизнес логика варианта 8:

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

В данной работе рассматривается вариант распараллеливания через OpenMP.

Данные генерятся Python скриптом и передаются исполняющей программе по TCP. Результирующий вектор возвращается по сетевому протоколу процессу генератора. В нём проверяется правильность полученного результата, затем данные о времени исполнения выводятся в консоль. Скрипт принимает три параметра: размер генерируемых данных в мегабайтах, порт для отправки данных и порт для получения результирующего вектора со временем исполнения. Сам скрипт приведен ниже.

```
import sys, os, math, socket
import numpy as np

'''
Random matrix stream generator for
Hybrid computing labs
Usage:
<script_name>.py [stream_size_in_MegaBytes] [tcp_port_out] [tcp_port_in]
Produces stream with square matrix of floats
and sends it as byte stream to tcp_port_out
and returns result received from tcp_port_in
'''

NUM_AVERAGES_DISPLAYED = 3

def send_file_to_socket(port, file_path):
    '''
    Sends string as byte stream to an open tcp socket
    '''
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('localhost', int(port))
    sock.connect(server_address)

    with open(file_path, 'rb') as input_file:
        sock.sendfile(input_file)

    sock.close()
    os.remove(file_path)
```

```

def get_string_from_socket(port, check_vector):
    """
    Listens to tcp port until get
    all the stream and then prints it
    """
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('localhost', int(port))
    sock.bind(server_address)
    sock.listen(1)
    connection, _ = sock.accept()

    result = []
    data = connection.recv(256)
    while data:
        result.append(data)
        data = connection.recv(256)
    result_str = ''.join([part.decode() for part in result])
    lines = result_str.split('\n')
    execution_time = lines[-2]
    numbers = lines[0].split(' ')

    assert len(numbers)-1 == len(check_vector)
    for i in range(len(check_vector)):
        assert math.fabs(float(numbers[i]) - check_vector[i]) < 0.0000001

    print(len(numbers)-1)
    print(' '.join(numbers[:NUM_AVERAGES_DISPLAYED]))
    print(execution_time)

    connection.close()
    sock.close()

def generate_stream_with_matrix(sizePorts):
    """
    Function to generate file with
    random matrix of given size
    """
    sizeInBytes = float(sizePorts[0]) * (2**20) # 1 Mb = 2^20 bytes
    num_rows_and_cols = int(math.sqrt(sizeInBytes / 11)) # 11 byte chars for each
    number
    matrix = np.random.rand(num_rows_and_cols, num_rows_and_cols)
    print(len(matrix))
    avg_vector = [round(sum(vector) / len(vector), 8) for vector in matrix]
    print(' '.join([str(avg) for avg in avg_vector[:NUM_AVERAGES_DISPLAYED]]))
    matrix_file_path = str(num_rows_and_cols) + '.mtrx'
    np.savetxt(matrix_file_path, matrix, fmt='%.8f', delimiter=' ')
    # matrix_str = np.array2string(matrix, formatter={'float_kind':lambda x: "%.8
    f" % x}, separator=' ').replace('[','').replace(']',',')

    # print(matrix_str.split('\n')[0])

```

```

send_file_to_socket(sizePorts[1], matrix_file_path)
get_string_from_socket(sizePorts[2], avg_vector)

if __name__ == "__main__":
    if len(sys.argv) > 2:
        generate_stream_with_matrix(sys.argv[1:])
    else:
        print("Usage: <script_name>.py [stream_size_in_MegaBytes] [tcp_port_out]
[tcp_port_in]")

```

Описание алгоритма выполнения бизнес-логики:

Программа включает вспомогательные функции:

- 1) `int* listen_to_port(int port)` – открывает соединение для переданного порта и возвращает дескрипторы соединения и сокета.
- 2) `void close_connection(int connection, int sfd)` – закрывает соединение и сокет.
- 3) `void send_stream_to_port(int port, char* stream)` – отправляет массив символов на соединение, открытое по указанному порту.

Верхнеуровневая бизнес-логика происходит в функции `int process_stream(int input_port, int output_port)`, которая принимает в качестве параметров номера портов для получения данных и передачи результата. В ней открывается соединение для получения данных, засекается время исполнения, проводится вычисление результирующего вектора, который затем вместе со временем выполнения в виде строки отправляется по ТСП на порт вывода результата.

Логика получения по сети матрицы воплощена в функции `int get_stream_matrix(int connection, std::vector<std::vector<float>> &matrix)`. В ней передаваемая матрица считывается и записывается в двумерный массив.

Логика получения результирующего вектора воплощена в функции `std::vector<char> get_avg_vector(std::vector<std::vector<float>> matrix)`. Она получает матрицу в виде двумерного массива чисел, далее в цикле рассчитывает среднее для каждой строки, после чего превращает полученный вектор чисел с плавающей точкой в строку.

Наконец, логика расчета среднего по строке воплощена в функции `float get_line_avg(std::vector<float> line)`. В ней числа из передаваемого массива суммируются и делятся на размер массива. Полученный результат возвращается из функции.

В точке входа воплощена следующая логика: в качестве аргументов программа принимает порт для получения данных и порт для передачи результата, а также количество тредов, на которое будет распараллелено

вычисление вектора средних по строкам матрицы. Далее в бесконечной петле запускается функция `process_stream`, которая ждет данные для обработки и отправляет результат.

Логика распараллеливания заключается в следующем: в функции `get_avg_vector` прописана прагма `omp parallel for shared(result)`, которая распараллеливает исполнение цикла, вычисляющего среднее значение для каждой строки. Как упоминалось выше, количество нитей исполнения задается последним аргументом командной строки, который передается функции `omp_set_num_threads(num_threads)`. Время исполнения распараллеленных операций засекается функцией `omp_get_wtime()`.

В качестве эксперимента производились запуски программы на одном, двух и четырех ядрах с матрицами размером от одного мегабайта до шестнадцати. Наблюдается прирост производительности в среднем на 15.5% при исполнении на двух ядрах по сравнению с одним, и на 33% на четырех по сравнению с одним. Увеличение количества нитей с двух до четырех делает исполнение в среднем на 19.5% быстрее.

#### Код программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <string.h>
#include <time.h>
#include <vector>
#include <omp.h>
#define MAX_SIZE 12

int process_stream(int input_port, int output_port);

int main(int argc, char* argv[])
{
    if (argc < 4)
    {
        printf("Need input, output streams ports and number of threads as parameters!");
        return -1;
    }

    int input_port = atoi(argv[1]);
    int output_port = atoi(argv[2]);
    int num_threads = atoi(argv[3]);

    omp_set_num_threads(num_threads);

    for (;;) // forever and ever
        process_stream(input_port, output_port);

    return 0;
```

```

}

int* listen_to_port(int port)
{
    int sfd, connection;

    int* opts = new int[1]{ 1 };

    if ((sfd = socket(AF_INET, SOCK_STREAM, 0)) == 0
        || setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, opts,
sizeof(int)))
        exit(EXIT_FAILURE);

    struct sockaddr_in address;
    int addrlen = sizeof(address);
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons(port);

    if (bind(sfd, (struct sockaddr*)&address,
sizeof(address)) < 0 || listen(sfd, 3) < 0
        || (connection = accept(sfd, (struct sockaddr*)&address,
(socklen_t*)&addrlen)) < 0)
        exit(EXIT_FAILURE);

    return new int[2]{ connection, sfd };
}

void close_connection(int connection, int sfd)
{
    shutdown(connection, SHUT_RDWR);
    close(connection);
    shutdown(sfd, SHUT_RDWR);
    close(sfd);
}

void send_stream_to_port(int port, char* stream)
{
    int sock = 0;
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        exit(EXIT_FAILURE);

    struct sockaddr_in serv_addr;
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(port);

    if (connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr)) <
0)
        exit(EXIT_FAILURE);

    send(sock, stream, strlen(stream), 0);

    shutdown(sock, SHUT_WR);
    close(sock);
}

float get_line_avg(std::vector<float> line)
{
    int count = line.size();
    float sum = 0;

```

```

        //printf("%d\n", omp_get_thread_num());

        for (int i = 0; i < count; i++)
            sum += line[i] / count;

        return sum;
    }

std::vector<char> get_avg_vector(std::vector<std::vector<float>> matrix)
{
    std::vector<float> result;

    int matrix_size = matrix.size();

    result.reserve(matrix_size);

    #pragma omp parallel for shared(result)
    for (int i = 0; i < matrix_size; i++)
    {
        double avg = get_line_avg(matrix[i]);
        result[i] = avg;
    }

    std::vector<char> char_result;

    for (int i = 0; i < matrix_size; i++)
    {
        char* avg_str = new char[MAX_SIZE];
        sprintf(avg_str, "%.8f ", result[i]);
        char_result.insert(char_result.end(), avg_str, avg_str +
strlen(avg_str));
    }

    char_result.push_back('\n');

    return char_result;
}

int get_stream_matrix(int connection, std::vector<std::vector<float>>
&matrix)
{
    int buffer_size = 1;
    char* buffer = new char[buffer_size];

    std::vector<char> number;
    std::vector<float> line;
    int stream_size = 0;
    while (read(connection, buffer, buffer_size) > 0)
    {
        stream_size++;
        int num_bytes = 1;

        while (buffer[0] != '\n' && num_bytes > 0)
        {
            if (buffer[0] == ' ')
            {
                line.push_back(atof(&number[0]));
                number.clear();
            }
            else

```

```

        number.push_back(buffer[0]);

        num_bytes = read(connection, buffer, buffer_size);
        stream_size++;
    }
    line.push_back(atof(&number[0]));
    number.clear();

    matrix.push_back(line);

    line.clear();
}

return stream_size;
}

int process_stream(int input_port, int output_port)
{
    int* connection = listen_to_port(input_port);

    std::vector<std::vector<float>> matrix;

    int stream_size = get_stream_matrix(connection[0], matrix);

    close_connection(connection[0], connection[1]);

    double start = omp_get_wtime();

    std::vector<char> result = get_avg_vector(matrix);

    double end = omp_get_wtime();
    double execution_time_in_seconds = (double)(end - start) * 1000;

    char* buffer = new char[256];
    sprintf(buffer, "%d bytes in %.9f milliseconds\n", stream_size,
execution_time_in_seconds);

    result.insert(result.end(), buffer, buffer + strlen(buffer));
    result.push_back(NULL);

    send_stream_to_port(output_port, &result[0]);

    return 0;
}

```

График зависимости времени выполнения от размера данных (по горизонтали отложены байты, по вертикали – время исполнения в миллисекундах), синий график – последовательное исполнение, жёлтый – параллельное на двух тредах, красный – параллельное на четырех:

