МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 "Компьютерные науки и прикладная математика"
Кафедра 806 "Вычислительная математика и программирование"

Лабораторная работа №2

По курсу «Операционные системы»

Студент: Теребаев К. Д.
Группа: М8О-203Б-22
Преподаватель: Миронов Е. С.
Дата: _____
Оценка: _____
Подпись: _____

Москва, 2024

Цель работы

Целью является приобретение практических навыков в:

- Управление потоками в ОС
- Обеспечение синхронизации между потоками

Задание

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы.

Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.

В отчете привести исследование зависимости ускорения и эффективности алгоритма от входных данных и количества потоков. Получившиеся результаты необходимо объяснить.

Вариант 10. Решить систему линейных уравнений методом Гаусса.

Общие сведения о программе

1. pthread_create (&mutex, NULL, start, &arg) – создает новый поток mutex, NULL - атрибуты потока по умолчанию, start - функция, которая будет выполняться в новом потоке, arg - аргументы, которые передаются этой функции. Возвращает 0 в случае успеха. В mutex сохраняется id потока;
2. pthread_join (tid, value_ptr) - откладывает выполнение вызывающего потока, до тех пор, пока не будет выполнен поток tid. Когда pthread_join выполнилась успешно, то она возвращает 0. Если поток явно вернул значение, то оно будет помещено в переменную value_ptr.

Общий алгоритм решения

Изначально из аргументов программы берем количество потоков. Далее открываем файл для ввода матриц и делаем проверку. Далее создаем массив для представления матрицы. Запускаю функцию для работы с потоками. В ней формирую массивы с данными, которые нужны будут для работы с потоками. После создаю потоки для решения матрицы. Изначально делю строку на диагональный элемент этой строки, а затем параллельно вычитаю из строк ниже эту строку, с которой сейчас работаю, умноженную на число ниже, так, чтобы снизу получился ноль. Нужная строка определяется посредством вычисления остатка при делении номера строки на общее количество потоков, если остаток равен идентификатору процесса, то это нужная нам строка. При количестве потоков равному 5 поток с номером 2 будет вычитать из строк с номером 2, 7, 12, 17 и т.д. Ответом для x_n будет являться последние столбцы n строки, из которых вычли все элементы этой строки, кроме n-го.

Основные файлы программы

main.cpp

```
#include <fcntl.h>
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>

#include <chrono>
#include <iostream>
```

```
using namespace std;

struct Args {
    int tid;
    int num_threads;
    float *matrix;
    int columns;
    pthread_barrier_t *barrier;
};

void print(float *matrix, int rows, int columns) {
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < columns; ++j) {
            cout << matrix[i * columns + j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

void *ge_parallel(void *args) {
    Args *local_args = (Args *)args;

    int tid = local_args->tid;
    int num_threads = local_args->num_threads;
    float *matrix = local_args->matrix;
    int columns = local_args->columns;
    pthread_barrier_t *barrier = local_args->barrier;

    for (int i = 0; i < columns - 1; i++) {
        if ((i % num_threads) == tid) {
            float pivot = matrix[i * columns + i];

            for (int j = i + 1; j < columns; j++) {
                matrix[i * columns + j] /= pivot;
            }

            matrix[i * columns + i] = 1;
        }

        pthread_barrier_wait(barrier);

        // print(matrix, columns - 1, columns);

        for (int j = i + 1; j < columns; j++) {
            if ((j % num_threads) == tid) {
                float scale = matrix[j * columns + i];

                for (int l = i + 1; l < columns; l++) {
                    matrix[j * columns + l] -= matrix[i * columns + l] * scale;
```

```
                }

                matrix[j * columns + i] = 0;
            }
        }
    }

    return 0;
}

void launch_threads(int num_threads, float *matrix, float *solve, int rows, int
columns) {
    pthread_t *threads = new pthread_t[num_threads];

    pthread_barrier_t barrier;
    pthread_barrier_init(&barrier, NULL, num_threads);

    Args *thread_args = new Args[num_threads];

    for (int i = 0; i < num_threads; i++) {
        thread_args[i].tid = i;
        thread_args[i].num_threads = num_threads;
        thread_args[i].matrix = matrix;
        thread_args[i].columns = columns;
        thread_args[i].barrier = &barrier;

        pthread_create(&threads[i], NULL, ge_parallel, (void *)&thread_args[i]);
    }

    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }

    solve[rows - 1] = matrix[rows * rows + rows - 1];
    for (int i = rows - 2; i >= 0; i--) {
        solve[i] = matrix[i * (rows + 1) + rows];
        for (int j = i + 1; j < rows; j++) {
            solve[i] -= matrix[i * (rows + 1) + j] * solve[j];
        }
    }

    delete[] threads;
    delete[] thread_args;
}

int main(int argc, char *argv[]) {
    int num_threads = stoi(argv[1]);
    if (argc < 1) {
        cerr << "At least 1 thread must exist" << endl;
        return 1;
    }
```

```
    int output_file_d = open("../InOutFiles/output.txt", O_CREAT | O_WRONLY |
O_TRUNC, S_IRWXU);
    int input_file_d = open("../InOutFiles/10.txt", O_RDONLY);
    // int input_file_d = open("../InOutFiles/2500.txt", O_RDONLY);
    // int input_file_d = open("../InOutFiles/3000.txt", O_RDONLY);

    if (input_file_d == -1 || output_file_d == -1) {
        cerr << "File error" << endl;
        return 1;
    }

    if (dup2(input_file_d, fileno(stdin)) == -1) {
        cerr << "Dup2 error" << endl;
        return 1;
    }

    int rows, columns;
    cin >> rows;
    columns = rows + 1;

    float *matrix;
    float *solve;

    matrix = new float[columns * columns];
    solve = new float[rows];

    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < columns; ++j) {
            cin >> matrix[i * columns + j];
        }
    }

    chrono::high_resolution_clock::time_point start;
    chrono::high_resolution_clock::time_point end;

    start = chrono::high_resolution_clock::now();
    launch_threads(num_threads, matrix, solve, rows, columns);
    end = chrono::high_resolution_clock::now();

    chrono::duration<float> elapsed =
chrono::duration_cast<chrono::duration<float>>(end - start);
    cout << "Time = " << elapsed.count() << " seconds" << endl;

    print(matrix, columns - 1, columns);

    if (dup2(output_file_d, fileno(stdout)) == -1) {
        cerr << "Dup2 error" << endl;
        return 1;
    }
```

```
    for (int i = 0; i < rows; i++) {
        cout << solve[i] << endl;
    }

    delete[] matrix;
    delete[] solve;
    close(input_file_d);
    close(output_file_d);

    return 0;
}
```

Пример работы

input.txt

```
10
13 12 14 10 13 14 13 10 12 14 13
13 11 14 11 14 15 11 12 14 11 12
14 13 11 13 12 10 13 14 14 11 14
13 14 15 12 10 12 11 12 15 13 11
12 14 10 14 14 13 10 11 13 10 15
13 14 13 15 15 13 12 12 14 14 14
13 12 13 11 15 10 15 13 15 14 11
14 14 11 10 14 11 11 12 13 10 13
11 14 13 15 14 13 13 12 12 12 11
14 12 13 14 13 14 13 14 14 12 12
```

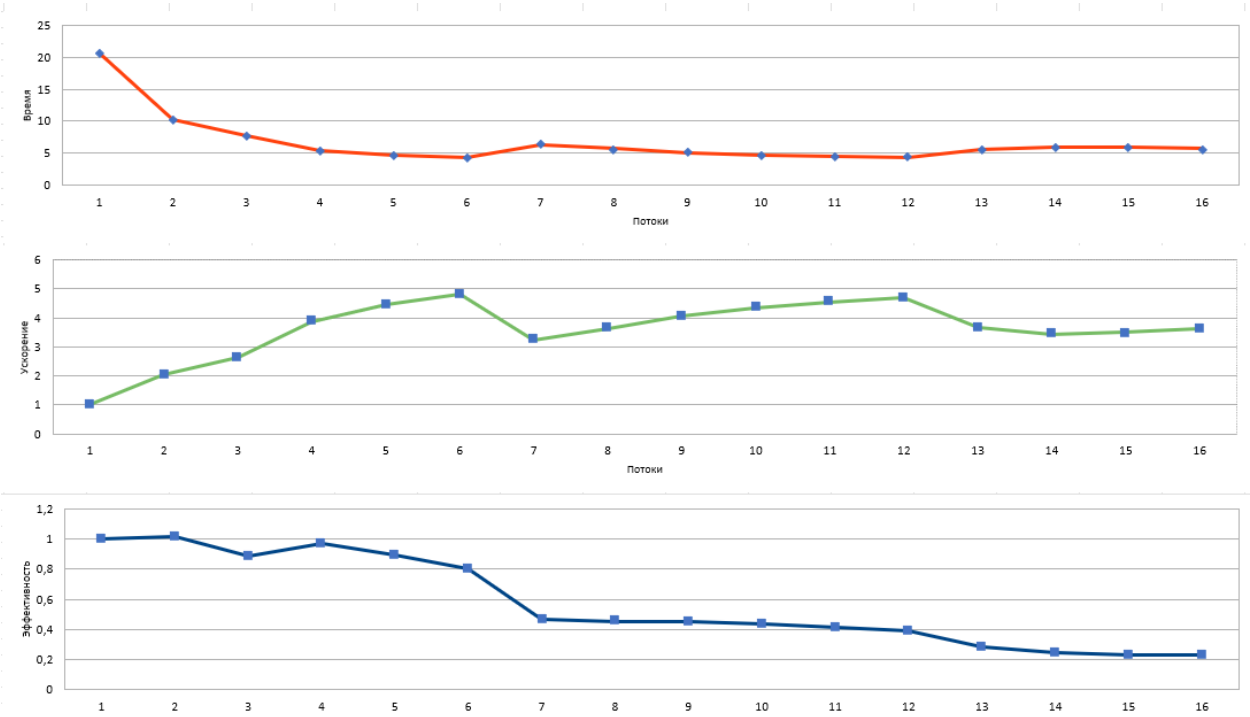
запуск

```
lw2/build [main●] » ./main 4
```

output

```
4.77141
-1.3921
2.02755
3.21891
0.39909
-1.84526
2.29108
-6.67633
1.35875
-3.68416
```

Графики



На графиках можно заметить, как с увеличением потоков время уменьшается. Так как у меня 6 ядер, то после 6 потоков эффективность падает, а время немного поднимается. Это можно обосновать тем, что после 6 потоков у них начинается, так называемая, «гонка» за данными. Можно сделать вывод, что оптимальным решением будет использовать только 6 потоков.

Вывод

В ходе лабораторной работы я научился производить параллельные вычисления, тем самым значительно ускоряя работу объемной программы. Основной проблемой было распределение задач для потоков так, чтобы между ними не возникало «гонки» за данными, то бишь выстроить работу таким образом, при котором разные потоки не изменяли одни и те же данные, так как в этом случае результат может быть непредсказуем. Так же я научился вычислять время работы программы и эффективность использования n-го количества потоков. Произведение таких вычислений помогает более эффективно использовать ресурсы компьютера. Стоит отметить, что на практике я повторил и укрепил знания о строении процессора. Эти умения определенно необходимы для работы программ с большим количеством данных.