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»

Группа: М8О-213Б-23

Студент: Чувилов А.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 15.11.24

Москва, 2024

Постановка задачи

Вариант 13.

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.

Наложить K раз фильтр, использующий матрицу свертки, на матрицу, состоящую из вещественных чисел. Размер окна задается пользователем.

Общий метод и алгоритм решения

Использованные системные вызовы:

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg) - создание нового потока выполнения.
- int pthread_join(pthread_t thread, void **retval) - ожидание завершения работы потока.
- int open(const char *pathname, int flags, mode_t mode) - открытие файла
- int close(int fd) - закрытие файла
- ssize_t read(int fd, void *buf, size_t count) - чтение данных из файла
- ssize_t write(int fd, const void *buf, size_t count) - запись данных в файл

Программа выполняет многопоточную свёртку матрицы с использованием заданного ядра. На вход передаются количество итераций (K), размер ядра (kernelSize) и количество потоков (numThreads). Матрица считывается из файла, создаются потоки, каждый из которых обрабатывает свой диапазон строк, выполняя свёртку: для каждого элемента строки суммируется произведение значений окна ядра и соответствующих элементов входной матрицы. После завершения работы всех потоков выходная матрица обновляется для следующей итерации. Итоговая матрица записывается в файл output_matrix.txt.

Код программы

main.c

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

#define EPS 0.0001

pthread_mutex_t printMutex = PTHREAD_MUTEX_INITIALIZER;

typedef struct Matrix {
   float** matrix;
   int rows;
   int columns;
} Matrix;

typedef struct ThreadData {
   float** input;
   float** kernel;
   float** output;
   int startRow;
   int endRow;
   int rows;
   int cols;
   int kernelSize;
   int id;
} ThreadData;

float** AllocateMatrix(int rows, int cols) {
   float** matrix = (float**)malloc(rows * sizeof(float*));
   for (int i = 0; i < rows; i++) {
      matrix[i] = (float*)calloc(cols, sizeof(float));
   }
   return matrix;
}

Matrix* CreateMatrix(int rows, int cols) {
   Matrix* m = (Matrix*)malloc(sizeof(Matrix));
   m->matrix = AllocateMatrix(rows + 2, cols + 2);
   m->rows = rows + 2;
   m->columns = cols + 2;
   return m;
}

void FreeMatrix(Matrix* m) {
   for (int i = 0; i < m->rows; i++) {
      free(m->matrix[i]);
   }
   free(m->matrix);
   free(m);
}

void FreeRawMatrix(float** matrix, int rows) {
   for (int i = 0; i < rows; i++) {
      free(matrix[i]);
   }
   free(matrix);
}
```

```
int ParseInt(const char* str, int* index) {
    int num = 0;
    while (str[*index] >= '0' && str[*index] <= '9') {
        num = num * 10 + (str[*index] - '0');
        (*index)++;
    }
    (*index)++;
    return num;
}

float ParseFloat(const char* str, int* index) {
    float num = 0;
    int sign = 1;
    if (str[*index] == '-') {
        sign = -1;
        (*index)++;
    }
    while (str[*index] >= '0' && str[*index] <= '9') {
        num = num * 10 + (str[*index] - '0');
        (*index)++;
    }
    if (str[*index] == '.') {
        (*index)++;
        float factor = 0.1;
        while (str[*index] >= '0' && str[*index] <= '9' && (factor > EPS)) {
            num += (str[*index] - '0') * factor;
            factor *= 0.1;
            (*index)++;
        }
    }
    (*index)++;
    return sign * num;
}

Matrix* ProcessFile(const char* filename) {
    int file = open(filename, O_RDONLY);
    if (file < 0) {
        char msg[] = "Ошибка: не удалось открыть файл.\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return NULL;
    }

    char buffer[100];
    int len = read(file, buffer, sizeof(buffer) - 1);
    if (len <= 0) {
        close(file);
        return NULL;
    }
    buffer[len] = '\0';

    int index = 0;
    int rows = ParseInt(buffer, &index);
    int cols = ParseInt(buffer, &index);

    Matrix* m = CreateMatrix(rows, cols);

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            while (buffer[index] == ' ' || buffer[index] == '\n') {
                index++;
            }
```

```
            m->matrix[i + 1][j + 1] = ParseFloat(buffer, &index);
        }
    }
    close(file);
    return m;
}

void* ApplyConvolutionThread(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    int offset = data->kernelSize / 2;

    for (int i = data->startRow; i < data->endRow; i++) {
        for (int j = offset; j < data->cols - offset; j++) {
            float sum = 0.0;
            int rowOffset = i - offset;
            int colOffset = j - offset;
            for (int ki = 0; ki < data->kernelSize; ki++) {
                for (int kj = 0; kj < data->kernelSize; kj++) {
                    sum += data->input[rowOffset + ki][colOffset + kj] * data->kernel[ki][kj];
                }
            }
            data->output[i][j] = sum;
        }
    }

    return NULL;
}

void FloatToStr(float num, char* buffer, int precision) {
    int i = 0;
    int integerPart = (int)num;
    float fractionalPart = num - integerPart;

    if (integerPart == 0) {
        buffer[i++] = '0';
    } else {
        if (integerPart < 0) {
            buffer[i++] = '-';
            integerPart = -integerPart;
            fractionalPart = -fractionalPart;
        }
        int start = i;
        while (integerPart > 0) {
            buffer[i++] = '0' + (integerPart % 10);
            integerPart /= 10;
        }
        for (int j = start; j < (i + start) / 2; j++) {
            char temp = buffer[j];
            buffer[j] = buffer[i - 1 - (j - start)];
            buffer[i - 1 - (j - start)] = temp;
        }
    }

    buffer[i++] = '.';

    for (int p = 0; p < precision; p++) {
        fractionalPart *= 10;
        int digit = (int)fractionalPart;
        buffer[i++] = '0' + digit;
        fractionalPart -= digit;
    }

    buffer[i] = '\0';
```

```
}

int SaveMatrixToFile(float** matrix, int rows, int cols, const char* filename) {
    int file = open(filename, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (file < 0) {
        char msg[] = "Ошибка: не удалось сохранить файл.\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return 0;
    }

    char buffer[50];
    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++) {
            FloatToStr(matrix[i][j], buffer, 2);
            int len = 0;
            while (buffer[len] != '\0') len++;
            buffer[len++] = ' ';
            write(file, buffer, len);
        }
        write(file, "\n", 1);
    }
    close(file);
    return 1;
}

void ApplyConvolution(float** input, float** kernel, float** output, int rows, int cols, int kernelSize, int
numThreads) {
    pthread_t threads[numThreads];
    ThreadData threadData[numThreads];

    int offset = kernelSize / 2;
    int rowsPerThread = (rows - 2 * offset) / numThreads;
    int extraRows = (rows - 2 * offset) % numThreads;

    for (int t = 0; t < numThreads; t++) {
        threadData[t].id = t;
        threadData[t].input = input;
        threadData[t].kernel = kernel;
        threadData[t].output = output;
        threadData[t].rows = rows;
        threadData[t].cols = cols;
        threadData[t].kernelSize = kernelSize;
        threadData[t].startRow = t * rowsPerThread + offset;
        threadData[t].endRow = (t + 1) * rowsPerThread + offset;

        if (t == numThreads - 1) {
            threadData[t].endRow += extraRows;
        }

        pthread_create(&threads[t], NULL, ApplyConvolutionThread, &threadData[t]);
    }

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

int main(int argc, char* argv[]) {
    if (argc != 4) {
        char msg[] = "Usage: ./main <count> <kernel size> <max_threads>\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        return 1;
    }
```

```
    int K = atoi(argv[1]);
    int kernelSize = atoi(argv[2]);
    int numThreads = atoi(argv[3]);

    Matrix* m = ProcessFile("gen.txt");
    if (kernelSize > m->rows || kernelSize > m->columns || kernelSize % 2 == 0) {
        char msg[] = "Некорректный размер окна свёртки.\n";
        write(STDERR_FILENO, msg, sizeof(msg) - 1);
        FreeMatrix(m);
        return -1;
    }

    float** kernel = AllocateMatrix(kernelSize, kernelSize);
    for (int i = 0; i < kernelSize; i++) {
        for (int j = 0; j < kernelSize; j++) {
            kernel[i][j] = 1.0;
        }
    }

    float** output = AllocateMatrix(m->rows, m->columns);

    for (int k = 0; k < K; k++) {
        ApplyConvolution(m->matrix, kernel, output, m->rows, m->columns, kernelSize, numThreads);

        float** temp = m->matrix;
        m->matrix = output;
        output = temp;
    }

    if (!SaveMatrixToFile(m->matrix, m->rows, m->columns, "output_matrix.txt")) {
        FreeMatrix(m);
        FreeRawMatrix(output, m->rows);
        FreeRawMatrix(kernel, kernelSize);
        return EXIT_FAILURE;
    }

    FreeMatrix(m);
    FreeRawMatrix(output, m->rows);
    FreeRawMatrix(kernel, kernelSize);

    return EXIT_SUCCESS;
}
```

Протокол работы программы

Тестирование:

1) $ ./a.out 3 3 4 2)$ ./a.out 3 3 5 3)$ ./a.out 3 3 10

50.253158 28.302659 3.141448

Производительность на примере теста 1:

Количество потоков	Среднее время выполнения, с
4	0,001374
5	0,001690
10	0,002086

Вывод

В ходе выполнения работы была разработана программа на языке Си, обрабатывающая матрицу вещественных чисел в многопоточном режиме. Для реализации использовались стандартные средства создания потоков операционной системы Unix: pthread_create для создания потоков и pthread_join для их синхронизации. Программа обеспечивает обработку данных с наложением фильтра свёртки, который применяется к матрице заданное пользователем количество раз (K). Размер окна фильтра также задаётся пользователем. Реализовано равномерное распределение строк матрицы между потоками, что позволяет эффективно использовать ресурсы системы. Максимальное количество одновременно работающих потоков регулируется параметром запуска программы. Итоговая матрица, полученная после выполнения всех итераций свёртки, сохраняется в файл. Программа корректно обрабатывает данные и демонстрирует эффективное использование многопоточности для выполнения ресурсоёмких операций.