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-212БВ-24

Студент: Головин В. П.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 17.10.25

Москва, 2025

Постановка задачи

Вариант 18.

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.

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

Общий метод и алгоритм решения

Использованные функции из библиотеки “pthread.h”:

pthread_create() - запускает новый поток с указанной функцией.
pthread_join() - приостанавливает основной поток до завершения указанного рабочего потока.

Использованные формулы для расчётов:

Ускорение: S = Ts / Tp, где
Ts – время последовательной реализации,
Tp – время параллельной реализации,
1 <= S <= p,
p – количество ядер
Показывает, во сколько раз параллельная версия быстрее последовательной.

Эффективность: X = S / p, где X < 1
Показывает, насколько эффективно каждое ядро используется.

Код программы

convolution.h

```
#ifndef CONVOLUTION_H

#define CONVOLUTION_H

typedef struct {
int width;
int height;
double** data;
} Matrix;

typedef struct {
int size;
double** kernel;
```

```
} ConvKernel;

Matrix* create_matrix(int width, int height);
void free_matrix(Matrix* matrix);
Matrix* copy_matrix(const Matrix* src);
void print_matrix(const Matrix* matrix);

ConvKernel* create_kernel(int size);
void free_kernel(ConvKernel* kernel);

void convolution(const Matrix* src, Matrix* dst, const ConvKernel* kernel);
double convolve_pixel(const Matrix* src, const ConvKernel* kernel, int x, int y);

#endif
```

utils.h

```
#ifndef UTILS_H
#define UTILS_H


#include <sys/time.h>
#include "convolution.h"

typedef struct {
struct timeval start_time;
struct timeval end_time;
} Timer;

void start_timer(Timer* timer);
void stop_timer(Timer* timer);
double get_spent_time(const Timer* timer);

void fill_matrix_random(Matrix* matrix);

#endif
```

convolution.c

```
#include <stdio.h>
#include <stdlib.h>
#include "convolution.h"

Matrix* create_matrix(int width, int height) {
Matrix* matrix = (Matrix*)malloc(sizeof(Matrix));
matrix->width = width;
matrix->height = height;
matrix->data = (double**)malloc(height * sizeof(double*));
for (int i = 0; i < height; i++) {
matrix->data[i] = (double*)malloc(width * sizeof(double));
}
```

```
return matrix;
}

void free_matrix(Matrix* matrix) {
if (matrix) {
for (int i = 0; i < matrix->height; i++) {
free(matrix->data[i]);
}
free(matrix->data);
free(matrix);
}
}

Matrix* copy_matrix(const Matrix* src) {
Matrix* dst = create_matrix(src->width, src->height);
for (int i = 0; i < src->height; i++) {
for (int j = 0; j < src->width; j++) {
dst->data[i][j] = src->data[i][j];
}
}
return dst;
}

void print_matrix(const Matrix* matrix) {
for (int i = 0; i < matrix->height; i++) {
for (int j = 0; j < matrix->width; j++) {
printf("%8.3f ", matrix->data[i][j]);
}
printf("\n");
}
}

ConvKernel* create_kernel(int size) {
ConvKernel* kernel = (ConvKernel*)malloc(sizeof(ConvKernel));
kernel->size = size;
kernel->kernel = (double**)malloc(size * sizeof(double*));
for (int i = 0; i < size; i++) {
kernel->kernel[i] = (double*)malloc(size * sizeof(double));
}
double value = 1.0 / (size * size);
for (int x = 0; x < size; x++) {
for (int y = 0; y < size; y++) {
kernel->kernel[x][y] = value;
}
}
return kernel;
}

void free_kernel(ConvKernel* kernel) {
if (kernel) {
for (int i = 0; i < kernel->size; i++) {
free(kernel->kernel[i]);
}
```

```
free(kernel->kernel);
free(kernel);
}
}

double convolve_pixel(const Matrix* src, const ConvKernel* kernel, int x, int y) {
int kernel_radius = kernel->size / 2;
double result = 0.0;
for (int i = -kernel_radius; i <= kernel_radius; i++) {
for (int j = -kernel_radius; j <= kernel_radius; j++) {
int src_x = x + i;
int src_y = y + j;
if (src_x < 0) src_x = -src_x;
if (src_y < 0) src_y = -src_y;
if (src_x >= src->height) src_x = 2 * src->height - src_x - 1;
if (src_y >= src->width) src_y = 2 * src->width - src_y - 1;
double pixel_value = src->data[src_x][src_y];
double kernel_value = kernel->kernel[i + kernel_radius][j + kernel_radius];
result += pixel_value * kernel_value;
}
}
return result;
}

void convolution(const Matrix* src, Matrix* dst, const ConvKernel* kernel) {
for (int i = 0; i < src->height; i++) {
for (int j = 0; j < src->width; j++) {
dst->data[i][j] = convolve_pixel(src, kernel, i, j);
}
}
}
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include "convolution.h"
#include "utils.h"

void sequential_convolution(Matrix* matrix, const ConvKernel* kernel, int iterations);
void parallel_convolution(Matrix* matrix, const ConvKernel* kernel, int iterations, int num_threads);

int main(int argc, char* argv[]) {
pid_t pid = getpid();
size_t n_threads = sysconf(_SC_NPROCESSORS_ONLN);
printf("Логических ядер: %ld\n", n_threads);
printf("%d\n", pid);
```

```
int width = 20, height = 20, iterations = 20, kernel_size = 5, max_threads = 4;
for (int i = 1; i < argc; i++) {
if (strcmp(argv[i], "-w") == 0 && i + 1 < argc) {
width = atoi(argv[++i]);
} else if (strcmp(argv[i], "-h") == 0 && i + 1 < argc) {
height = atoi(argv[++i]);
} else if (strcmp(argv[i], "-k") == 0 && i + 1 < argc) {
iterations = atoi(argv[++i]);
} else if (strcmp(argv[i], "-s") == 0 && i + 1 < argc) {
kernel_size = atoi(argv[++i]);
} else if (strcmp(argv[i], "-t") == 0 && i + 1 < argc) {
max_threads = atoi(argv[++i]);
}
}
Matrix* matrix = create_matrix(width, height);
fill_matrix_random(matrix);
//print_matrix(matrix);
//printf("\n");
ConvKernel* kernel = create_kernel(kernel_size);
Timer timer;
double sequential_time, parallel_time;
// Последовательная версия
Matrix* seq_matrix = copy_matrix(matrix);
start_timer(&timer);
sequential_convolution(seq_matrix, kernel, iterations);
stop_timer(&timer);

sequential_time = get_spent_time(&timer);
printf("Время выполнения: %.6f секунд\n", sequential_time);
// Параллельная версия
for (int num_threads = 1; num_threads <= max_threads; num_threads++) {
Matrix* par_matrix = copy_matrix(matrix);
start_timer(&timer);
parallel_convolution(par_matrix, kernel, iterations, num_threads);
stop_timer(&timer);
parallel_time = get_spent_time(&timer);
double speedup = sequential_time / parallel_time;
double efficiency = speedup / num_threads;
printf("Потоков: %d, Время: %.6f сек, Ускорение: %.2fx, Эффективность: %.3f\n",
num_threads, parallel_time, speedup, efficiency);
free_matrix(par_matrix);
}
free_matrix(matrix);
free_matrix(seq_matrix);
free_kernel(kernel);
return 0;
}
```

parallel.c

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <pthread.h>
#include "convolution.h"
#include "utils.h"

typedef struct {
Matrix* src;
Matrix* dst;
const ConvKernel* kernel;
int start_row;
int end_row;
} ThreadData;

void* process_rows(void* arg) {
ThreadData* data = (ThreadData*)arg;
for (int i = data->start_row; i < data->end_row; i++) {
for (int j = 0; j < data->src->width; j++) {
data->dst->data[i][j] = convolve_pixel(data->src, data->kernel, i, j);
}
}
pthread_exit(NULL);
}

void parallel_convolution(Matrix* matrix, const ConvKernel* kernel, int iterations, int num_threads) {
Matrix* temp = create_matrix(matrix->width, matrix->height);
Matrix* current_src = matrix;
Matrix* current_dst = temp;
for (int k = 0; k < iterations; k++) {
pthread_t threads[num_threads];
ThreadData thread_data[num_threads];
int rows_per_thread = matrix->height / num_threads;
int remaining_rows = matrix->height % num_threads;
int current_row = 0;
for (int t = 0; t < num_threads; t++) {
thread_data[t].src = current_src;
thread_data[t].dst = current_dst;
thread_data[t].kernel = kernel;
thread_data[t].start_row = current_row;
int thread_rows = rows_per_thread + (t < remaining_rows ? 1 : 0);
thread_data[t].end_row = current_row + thread_rows;
current_row += thread_rows;
pthread_create(&threads[t], NULL, process_rows, &thread_data[t]);
}
for (int t = 0; t < num_threads; t++) {
pthread_join(threads[t], NULL);
}
Matrix* swap_temp = current_src;
current_src = current_dst;
current_dst = swap_temp;
}
if (current_src == temp) {
for (int i = 0; i < matrix->height; i++) {
for (int j = 0; j < matrix->width; j++) {
matrix->data[i][j] = temp->data[i][j];
```

```
}
}
}
free_matrix(temp);
}
```

suquential.c

```
#include <stdio.h>
#include <stdlib.h>
#include "convolution.h"
#include "utils.h"

void sequential_convolution(Matrix* matrix, const ConvKernel* kernel, int iterations) {
Matrix* temp = create_matrix(matrix->width, matrix->height);
for (int k = 0; k < iterations; k++) {
convolution(matrix, temp, kernel);
for (int i = 0; i < matrix->height; i++) {
for (int j = 0; j < matrix->width; j++) {
matrix->data[i][j] = temp->data[i][j];
}
}
}
//print_matrix(matrix);
free_matrix(temp);
}
```

utils.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include "utils.h"

void start_timer(Timer* timer) {
gettimeofday(&timer->start_time, NULL);
}

void stop_timer(Timer* timer) {
gettimeofday(&timer->end_time, NULL);
}

double get_spent_time(const Timer* timer) {
double start = timer->start_time.tv_sec + timer->start_time.tv_usec / 1000000.0;
double end = timer->end_time.tv_sec + timer->end_time.tv_usec / 1000000.0;
return end - start;
}

void fill_matrix_random(Matrix* matrix) {
srand(time(NULL));
for (int i = 0; i < matrix->height; i++) {
```

```
for (int j = 0; j < matrix->width; j++) {
matrix->data[i][j] = (double)rand() / RAND_MAX * 100.0;
}
}
}
```

Протокол работы программы

```
spr0vay@spr0vayhost:~/Documents/Coding/Labs/OS_Labs/Lab2/build$ ./lab2 -w 80 -h 60 -k
20 -s 5 -t 24
Логических ядер: 12
4006529
Время выполнения: 0.008321 секунд
Потоков: 1, Время: 0.011531 сек, Ускорение: 0.72x, Эффективность: 0.722
Потоков: 2, Время: 0.007114 сек, Ускорение: 1.17x, Эффективность: 0.585
Потоков: 3, Время: 0.005479 сек, Ускорение: 1.52x, Эффективность: 0.506
Потоков: 4, Время: 0.006185 сек, Ускорение: 1.35x, Эффективность: 0.336
Потоков: 5, Время: 0.007337 сек, Ускорение: 1.13x, Эффективность: 0.227
Потоков: 6, Время: 0.008907 сек, Ускорение: 0.93x, Эффективность: 0.156
Потоков: 7, Время: 0.008494 сек, Ускорение: 0.98x, Эффективность: 0.140
Потоков: 8, Время: 0.009470 сек, Ускорение: 0.88x, Эффективность: 0.110
Потоков: 9, Время: 0.010112 сек, Ускорение: 0.82x, Эффективность: 0.091
Потоков: 10, Время: 0.010899 сек, Ускорение: 0.76x, Эффективность: 0.076
Потоков: 11, Время: 0.011221 сек, Ускорение: 0.74x, Эффективность: 0.067
Потоков: 12, Время: 0.011273 сек, Ускорение: 0.74x, Эффективность: 0.062
Потоков: 13, Время: 0.019469 сек, Ускорение: 0.43x, Эффективность: 0.033
Потоков: 14, Время: 0.019241 сек, Ускорение: 0.43x, Эффективность: 0.031
Потоков: 15, Время: 0.016765 сек, Ускорение: 0.50x, Эффективность: 0.033
Потоков: 16, Время: 0.017282 сек, Ускорение: 0.48x, Эффективность: 0.030
Потоков: 17, Время: 0.020597 сек, Ускорение: 0.40x, Эффективность: 0.024
Потоков: 18, Время: 0.027994 сек, Ускорение: 0.30x, Эффективность: 0.017
Потоков: 19, Время: 0.021780 сек, Ускорение: 0.38x, Эффективность: 0.020
Потоков: 20, Время: 0.038552 сек, Ускорение: 0.22x, Эффективность: 0.011
Потоков: 21, Время: 0.028855 сек, Ускорение: 0.29x, Эффективность: 0.014
Потоков: 22, Время: 0.028560 сек, Ускорение: 0.29x, Эффективность: 0.013
Потоков: 23, Время: 0.016139 сек, Ускорение: 0.52x, Эффективность: 0.022
Потоков: 24, Время: 0.016483 сек, Ускорение: 0.50x, Эффективность: 0.021
```

Вывод

В процессе выполнения лабораторной работы я составил программу на языке Си, обрабатывающую данные в многопоточном режиме. В процессе изучения результатов я на практике убедился, что увеличение числа потоков уменьшает время выполнения программы, поскольку мы используем сразу несколько ядер процессора для вычислений. Однако, при увеличении числа потока выше количества логических ядер выполнение замедляется, так как операционная система вынуждена выполнять множество операций управления ресурсами, что негативно отражается на эффективности решения основной задачи.