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: М80-211Бв-24

Студент: Скворцов Ю.И.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата:26.12.25

Москва, 2025

Постановка задачи

Вариант 4.

Рассчитать детерминант матрицы (используя определение детерминанта)

Общий метод и алгоритм решения

Использованные системные вызовы:

- pthread_create() - создание потока
- pthread_join() - ожидание завершения потока
- pthread_mutex_init() / PTHREAD_MUTEX_INITIALIZER - инициализация мьютекса
- pthread_mutex_lock() - блокировка мьютекса
- pthread_mutex_unlock() - разблокировка мьютекса
- pthread_mutex_destroy() - уничтожение мьютекса
- clock() - измерение процессорного времени
- sysconf() - получение системной информации

Работа программы:

1. Инициализация
2. Создание и заполнение матрицы
3. Последовательное вычисление (замер времени)
4. Параллельные вычисления (серия экспериментов)
5. Вывод результатов
6. Очистка ресурсов

Код программы

deter.c

```c
#define _POSIX_C_SOURCE 199309L

#include <stdint.h>

#include <stdbool.h>

#include <limits.h>

#include <string.h>

#include <time.h>

#include <stdlib.h>

#include <stdio.h>

#include <unistd.h>

#include <pthread.h>
```

```c
typedef struct {
    size_t size;
    int **data;
} Matrix;


typedef struct {
    size_t thread_id;
    Matrix *matrix;
    int *result;
    size_t *current_index;
    pthread_mutex_t *mutex;
    int **minor_buffer;
} ThreadArgs;


Matrix* CreateMatrix(size_t size) {
    Matrix *matrix = malloc(sizeof(Matrix));
    matrix->size = size;
    matrix->data = malloc(size * sizeof(int*));
    for (size_t i = 0; i < size; i++) {
        matrix->data[i] = malloc(size * sizeof(int));
    }


    return matrix;
}


void FillMatrix(Matrix *matrix) {
    for (size_t i = 0; i < matrix->size; i++) {
        for (size_t j = 0; j < matrix->size; j++) {
            matrix->data[i][j] = rand() % 10 - 5;
```

```
        }
    }
}

void FreeMatrix(Matrix *matrix) {
    for (size_t i = 0; i < matrix->size; i++) {
        free(matrix->data[i]);
    }
    free(matrix->data);
    free(matrix);
}

Matrix* FindMinor(const Matrix *matrix, size_t row, size_t col, int **buffer) {
    Matrix *minor = malloc(sizeof(Matrix));
    minor->size = matrix->size - 1;
    minor->data = buffer;

    size_t minor_i = 0;
    for (size_t i = 0; i < matrix->size; i++) {
        if (i == row) continue;

        size_t minor_j = 0;
        for (size_t j = 0; j < matrix->size; j++) {
            if (j == col) continue;

            buffer[minor_i][minor_j] = matrix->data[i][j];
            minor_j++;
        }
        minor_i++;
```

```
    }

    return minor;
}

void free_minor(Matrix *minor) {
    free(minor);
}

int DeterminantSequential(const Matrix *matrix) {
    if (matrix->size == 1) {
        return matrix->data[0][0];
    }

    if (matrix->size == 2) {
        return matrix->data[0][0] * matrix->data[1][1] -
               matrix->data[0][1] * matrix->data[1][0];
    }

    int det = 0;
    int sign = 1;

    int **temp_buffer = malloc((matrix->size - 1) * sizeof(int*));
    for (size_t i = 0; i < matrix->size - 1; i++) {
        temp_buffer[i] = malloc((matrix->size - 1) * sizeof(int));
    }

    for (size_t j = 0; j < matrix->size; j++) {
        Matrix *minor = FindMinor(matrix, 0, j, temp_buffer);
```

```
        det += sign * matrix->data[0][j] * DeterminantSequential(minor);
        sign = -sign;
        free_minor(minor);
    }
    for (size_t i = 0; i < matrix->size - 1; i++) {
        free(temp_buffer[i]);
    }
    free(temp_buffer);

    return det;
}

static void *DetProc(void *_args) {
    ThreadArgs *args = (ThreadArgs*)_args;

    while (true) {
        pthread_mutex_lock(args->mutex);
        size_t current_index = *(args->current_index);
        if (current_index >= args->matrix->size) {
            pthread_mutex_unlock(args->mutex);
            break;
        }
        *(args->current_index) = current_index + 1;
        pthread_mutex_unlock(args->mutex);

        size_t j = current_index;
        int sign = (j % 2 == 0) ? 1 : -1;

        Matrix *minor = FindMinor(args->matrix, 0, j, args->minor_buffer);
```

```
        int minor_det = DeterminantSequential(minor);
        int contribution = sign * args->matrix->data[0][j] * minor_det;
        free_minor(minor);

        pthread_mutex_lock(args->mutex);
        *(args->result) += contribution;
        pthread_mutex_unlock(args->mutex);
    }

    return NULL;
}

int DetParallel(Matrix *matrix, size_t n_threads) {
    if (matrix->size == 1) {
        return matrix->data[0][0];
    }

    if (matrix->size == 2) {
        return matrix->data[0][0] * matrix->data[1][1] -
               matrix->data[0][1] * matrix->data[1][0];
    }

    if (n_threads > matrix->size) {
        n_threads = matrix->size;
    }

    int result = 0;
    size_t current_index = 0;
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_t *threads = malloc(n_threads * sizeof(pthread_t));
ThreadArgs *thread_args = malloc(n_threads * sizeof(ThreadArgs));

int ***thread_buffers = malloc(n_threads * sizeof(int**));
for (size_t i = 0; i < n_threads; i++) {
    thread_buffers[i] = malloc((matrix->size - 1) * sizeof(int*));
    for (size_t j = 0; j < matrix->size - 1; j++) {
        thread_buffers[i][j] = malloc((matrix->size - 1) * sizeof(int));
    }
}

for (size_t i = 0; i < n_threads; ++i) {
    thread_args[i] = (ThreadArgs){
        .thread_id = i,
        .matrix = matrix,
        .result = &result,
        .current_index = &current_index,
        .mutex = &mutex,
        .minor_buffer = thread_buffers[i]
    };

    pthread_create(&threads[i], NULL, DetProc, &thread_args[i]);
}

for (size_t i = 0; i < n_threads; ++i) {
    pthread_join(threads[i], NULL);
}
```

```
    for (size_t i = 0; i < n_threads; i++) {
        for (size_t j = 0; j < matrix->size - 1; j++) {
            free(thread_buffers[i][j]);
        }
        free(thread_buffers[i]);
    }
    free(thread_buffers);

    free(thread_args);
    free(threads);
    pthread_mutex_destroy(&mutex);

    return result;
}

double GetTime() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000.0 + ts.tv_nsec / 1000000.0;
}

size_t* ThreadProgression(size_t max_threads, size_t matrix_size, size_t
*num_experiments) {
    size_t *thread_counts = malloc(32 * sizeof(size_t));
    size_t count = 0;

    size_t logical_cores = sysconf(_SC_NPROCESSORS_ONLN);
    thread_counts[count++] = logical_cores;
    thread_counts[count++] = 1;
```

```
    size_t t = 2;
    while (t <= max_threads) {
        thread_counts[count++] = t;
        t *= 2;
    }

    *num_experiments = count;
    return thread_counts;
}

int main(int argc, char **argv) {
    if (argc != 3) {
        char buffer[256];
        snprintf(buffer, sizeof(buffer), "Usage: %s <matrix_size>
<max_threads>\n", argv[0]);
        fwrite(buffer, 1, strlen(buffer), stdout);
        return 1;
    }

    size_t matrix_size = atoi(argv[1]);
    size_t max_threads = atoi(argv[2]);

    if (matrix_size < 1 || max_threads < 1) {
        char buffer[256];
        snprintf(buffer, sizeof(buffer), "Matrix size and max threads must be
positive numbers\n");
        fwrite(buffer, 1, strlen(buffer), stdout);
        return 1;
    }
```

```
    srand(time(NULL));


    Matrix *matrix = CreateMatrix(matrix_size);
    FillMatrix(matrix);


    if (matrix_size <= 16) {
        for (size_t i = 0; i < matrix_size; i++) {
            char rowBuffer[1024] = "";
            for (size_t j = 0; j < matrix_size; j++) {
                char temp[16];
                snprintf(temp, sizeof(temp), "%4d ", matrix->data[i][j]);
                strcat(rowBuffer, temp);
            }
            strcat(rowBuffer, "\n");
            fwrite(rowBuffer, 1, strlen(rowBuffer), stdout);
        }
    }


    double start_time = GetTime();
    int det_seq = DeterminantSequential(matrix);
    double seq_time = GetTime() - start_time;


    char buffer[256];
    snprintf(buffer, sizeof(buffer), "\nSequential:\nDeterminant: %d\nTime: %.3f
ms\n", det_seq, seq_time);
    fwrite(buffer, 1, strlen(buffer), stdout);


    snprintf(buffer, sizeof(buffer), "\nParallel version:\n%-9s | %-9s | %-9s |
%-9s\n----------+-----------+-----------+-----------\n",
```

```
           "Threads", "Time (ms)", "Speedup", "Efficiency");
    fwrite(buffer, 1, strlen(buffer), stdout);



    size_t num_experiments;
    size_t *thread_counts = ThreadProgression(max_threads, matrix_size,
&num_experiments);


    for (size_t i = 0; i < num_experiments; i++) {
        size_t n_threads = thread_counts[i];


        double start_par = GetTime();
        int det_par = DetParallel(matrix, n_threads);
        double par_time = GetTime() - start_par;


        if (det_par != det_seq) {
            snprintf(buffer, sizeof(buffer), "ERROR: result mismatch (seq=%d,
par=%d)\n", det_seq, det_par);
            fwrite(buffer, 1, strlen(buffer), stdout);
            continue;
        }


        double speedup = seq_time / par_time;
        double efficiency = (n_threads > 1) ? (speedup / n_threads * 100) : 100.0;


        snprintf(buffer, sizeof(buffer), "%-9zu | %-9.3f | %-9.3f | %-9.1f%%\n",
               n_threads, par_time, speedup, efficiency);
        fwrite(buffer, 1, strlen(buffer), stdout);
    }
```

```
    size_t logical_cores = sysconf(_SC_NPROCESSORS_ONLN);

    snprintf(buffer, sizeof(buffer), "\nNumber of logical cores: %zu\n",
logical_cores);

    fwrite(buffer, 1, strlen(buffer), stdout);



    free(thread_counts);

    FreeMatrix(matrix);

    return 0;

}
```

Протокол работы программы

```
12x12 matrix:
Determinant: -963466520
Time: 45204.249 ms

Parallel version:
Threads   | Time (ms) | Speedup   | Efficiency
----------+-----------+-----------+-----------
12        | 5023.911  | 8.998     | 75.0     %
1         | 48404.459 | 0.934     | 100.0    %
2         | 24937.294 | 1.813     | 90.6     %
4         | 14511.508 | 3.115     | 77.9     %
8         | 9439.096  | 4.789     | 59.9     %
16        | 5511.196  | 8.202     | 51.3     %

Number of logical cores: 12
Максимальное ускорение: 12 потоков(совпадает с размерностью матрицы)
Дальше эффективность падает слишком сильно, как и скорость.

8x8 matrix:
Determinant: 679876
Time: 5.258 ms

Parallel version:
Threads   | Time (ms) | Speedup   | Efficiency
----------+-----------+-----------+-----------
12        | 2.537     | 2.072     | 17.3     %
1         | 6.265     | 0.839     | 100.0    %
2         | 3.564     | 1.475     | 73.8     %
4         | 2.325     | 2.261     | 56.5     %
8         | 2.046     | 2.570     | 32.1     %
```

```
16        | 2.312     | 2.274     | 14.2     %
32        | 1.779     | 2.956     | 9.2      %
64        | 2.205     | 2.385     | 3.7      %
128       | 2.384     | 2.206     | 1.7      %
256       | 2.440     | 2.155     | 0.8      %
512       | 2.081     | 2.527     | 0.5      %
1024      | 2.365     | 2.223     | 0.2      %

Number of logical cores: 12

8 потоков - хорошее ускорение. Если брать больше - эффективность слишком низкая.
```

Вывод

В ходе лабораторной работы была успешно реализована и исследована параллельная программа для вычисления определителя матрицы методом разложения по строке с использованием библиотеки POSIX Threads. Программа демонстрирует значительное ускорение вычислений при оптимальном выборе количества потоков, подтверждая эффективность многопоточного программирования для вычислительно сложных задач. Имеетсся прямая зависимость ускорения от размера задачи — чем больше матрица, тем выше потенциал параллелизма. Для каждой размерности матрицы существует оптимальное количество потоков. Использование потоков сверх разумного предела ухудшает производительность.