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-211БВ-24

Студент: Бахшалиев М.А.

Преподаватель: Бахарев В.Д.

Оценка: _____

Дата: 29.10.25

Москва, 2025

Постановка задачи

Вариант 9.

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. Задание: Рассчитать детерминант матрицы (используя определение детерминанта)

Общий метод и алгоритм решения

Общая концепция подхода

Программа реализует гибридный алгоритм, сочетающий рекурсивное вычисление определителя с многопоточным параллелизмом. Основная идея заключается в распределении вычислительной нагрузки между несколькими потоками для ускорения обработки матриц значительного размера.

Архитектурный подход

Разделение ответственности:

- Главный поток управляет созданием рабочих потоков и сбором результатов
- Рабочие потоки независимо обрабатывают назначенные им части матрицы
- Общие ресурсы защищаются механизмами синхронизации

Модель данных:

- Матрица представляется в виде двумерного массива с метаинформацией о размере
- Для передачи параметров потокам используются специализированные структуры данных
- Временные миноры создаются и уничтожаются в контексте каждого потока

Алгоритмическая стратегия

Базовый вычислительный метод

В основе лежит классический алгоритм вычисления определителя через разложение по строке. Для матрицы большого размера выполняется рекурсивное разложение на миноры меньшего размера до достижения базовых случаев (матрицы 1×1 и 2×2).

Стратегия распараллеливания

Применяется модель "мастер-воркер", где:

- Мастер-поток инициализирует вычисления и управляет рабочими потоками
- Воркер-потоки динамически получают задачи из общего пула

- Распределение работы происходит через разделяемый счетчик строк

Механизм синхронизации

Используется мьютекс для защиты критических секций:

- Доступ к счетчику текущей обрабатываемой строки
- Обновление аккумулятора итогового результата
  Такая организация минимизирует время блокировок и снижает конкуренцию между потоками.

Процесс вычисления

Фаза инициализации

Программа создает матрицу заданного размера и заполняет ее случайными значениями. Затем определяет оптимальную последовательность количеств потоков для тестирования, обычно по степеням двойки.

Последовательное выполнение

Сначала выполняется эталонное вычисление определителя одним потоком для:

- Получения контрольного значения для проверки корректности
- Измерения базового времени выполнения
- Установки точки отсчета для расчета ускорения

Параллельное выполнение

Для каждого тестируемого количества потоков:

1. Создается пул потоков с общими параметрами
2. Потоки независимо обрабатывают назначенные строки матрицы
3. Результаты аккумулируются с соблюдением синхронизации
4. Измеряется время выполнения и сравнивается с последовательной версией

Методология оценки производительности

Ключевые метрики

- **Абсолютное время выполнения** - непосредственно измеренное время работы алгоритма
- **Ускорение** - отношение времени последовательного выполнения к параллельному
- **Эффективность** - процент использования вычислительной мощности потоков

Анализ результатов

Исследуются закономерности:

- Зависимость ускорения от количества потоков

- Влияние размера матрицы на эффективность параллелизма
- Пределы масштабируемости алгоритма

Особенности реализации

Управление памятью

Каждый поток работает с локальными копиями данных миноров, что исключает конфликты при доступе к памяти и уменьшает необходимость в синхронизации.

Балансировка нагрузки

Динамическое распределение строк между потоками обеспечивает равномерную загрузку даже при неоднородной сложности вычислений для разных строк.

Обеспечение корректности

Сравнение результатов параллельных вычислений с последовательной эталонной версией гарантирует правильность работы алгоритма при любом количестве потоков.

Область применения и ограничения

Данный подход наиболее эффективен для матриц среднего и большого размера, где вычислительная сложность достаточно высока для компенсации накладных расходов на создание потоков и синхронизацию. Для маленьких матриц последовательная версия может оказаться производительнее из-за отсутствия дополнительных затрат на управление параллелизмом.

Такой метод демонстрирует классический компромисс между простотой реализации, эффективностью использования ресурсов и масштабируемостью, характерный для задач с рекурсивной природой и независимыми подзадачами.

Код программы

main.c:

```
///usr/bin/cc -o /tmp/${0%%.c} -pthread $0 && exec /tmp/${0%%.c}
#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>

typedef struct {
    size_t size;
    int **data;
```

```
} Matrix;

typedef struct {
    size_t thread_id;
    size_t n_threads;
    Matrix *matrix;
    int *result;
    size_t *current_row;
    pthread_mutex_t *mutex;
} ThreadArgs;

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

Matrix* get_minor(const Matrix *matrix, size_t row, size_t col) {
    Matrix *minor = create_matrix(matrix->size - 1);

    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;

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

    return minor;
}

int determinant_sequential(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;

    for (size_t j = 0; j < matrix->size; j++) {
        Matrix *minor = get_minor(matrix, 0, j);
        det += sign * matrix->data[0][j] * determinant_sequential(minor);
        sign = -sign;
        free_matrix(minor);
    }

    return det;
}

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

    while (true) {
```

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

        int sign = (current_row % 2 == 0) ? 1 : -1;

        Matrix *minor = get_minor(args->matrix, current_row, 0);
        int minor_det = determinant_sequential(minor);
        int contribution = sign * args->matrix->data[current_row][0] * minor_det;
        free_matrix(minor);

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

    return NULL;
}

int determinant_parallel(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];
    }

    int result = 0;
    size_t current_row = 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));

    for (size_t i = 0; i < n_threads; ++i) {
        thread_args[i] = (ThreadArgs){
```

```
            .thread_id = i,
            .n_threads = n_threads,
            .matrix = matrix,
            .result = &result,
            .current_row = &current_row,
            .mutex = &mutex
        };

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

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

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

    return result;
}

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

size_t* generate_thread_progression(size_t max_threads, size_t *num_experiments) {
    size_t *thread_counts = malloc(32 * sizeof(size_t));
    size_t count = 0;
    size_t t = 1;

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

    if (count > 0 && thread_counts[count-1] != max_threads) {
        thread_counts[count++] = max_threads;
    }

    *num_experiments = count;
```

```
    return thread_counts;
}

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

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

    if (matrix_size < 1 || max_threads < 1) {
        printf("Matrix size and max threads must be positive numbers\n");
        return 1;
    }

    printf("=== Calculating determinant of %zux%zu matrix ===\n", matrix_size, matrix_size);
    printf("=== Testing thread progression up to %zu threads ===\n", max_threads);

    Matrix *matrix = create_matrix(matrix_size);
    fill_matrix_random(matrix);

    if (matrix_size <= 6) {
        printf("Matrix:\n");
        for (size_t i = 0; i < matrix_size; i++) {
            for (size_t j = 0; j < matrix_size; j++) {
                printf("%4d ", matrix->data[i][j]);
            }
            printf("\n");
        }
    }

    double start_time = get_time_ms();
    int det_seq = determinant_sequential(matrix);
    double seq_time = get_time_ms() - start_time;

    printf("\nSequential version:\n");
    printf("Determinant: %d\n", det_seq);
    printf("Time: %.3f ms\n", seq_time);

    printf("\nParallel version:\n");
    printf("%-12s | %-18s | %-10s | %-12s\n",
```

```
           "Threads", "Time (ms)", "Speedup", "Efficiency");
    printf("-------------+--------------------+-----------+-------------\n");

    size_t num_experiments;
    size_t *thread_counts = generate_thread_progression(max_threads, &num_experiments);

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

        double start_par = get_time_ms();
        int det_par = determinant_parallel(matrix, n_threads);
        double par_time = get_time_ms() - start_par;

        if (det_par != det_seq) {
            printf("Error: result mismatch! (seq=%d, par=%d)\n", det_seq, det_par);
        }

        double speedup = seq_time / par_time;
        double efficiency = speedup / n_threads * 100;

        printf("%-12zu | %-18.3f | %-10.3f | %-11.1f%%\n",
               n_threads, par_time, speedup, efficiency);
    }

    size_t logical_cores = sysconf(_SC_NPROCESSORS_ONLN);
    printf("\nNumber of logical cores: %zu\n", logical_cores);

    free(thread_counts);
    free_matrix(matrix);
    return 0;
}
```

Протокол работы программы

Тестирование 1:

```
./determinant 4 16

=== Calculating determinant of 4x4 matrix ===

=== Testing thread progression up to 16 threads ===

Matrix:

-2    1    2    0

-2    0    1   -3
```

```
  4   -4   -3    2
 -5    4   -2    1

Sequential version:
Determinant: 115
Time: 0.002 ms

Parallel version:
 Threads      | Time (ms)          | Speedup    | Efficiency
-------------+--------------------+------------+-------------
1             | 0.215              | 0.011      | 1.1        %
2             | 0.281              | 0.009      | 0.4        %
4             | 0.287              | 0.008      | 0.2        %
8             | 0.412              | 0.006      | 0.1        %
16            | 0.820              | 0.003      | 0.0        %

Number of logical cores: 12
```

Тестирование 2:

./determinant 12 64

=== Calculating determinant of 12x12 matrix ===

=== Testing thread progression up to 64 threads ===

Sequential version:

Determinant: -839352291

Time: 24455.792 ms

Parallel version:

Threads	Time (ms)	Speedup	Efficiency
1	27432.624	0.891	89.1 %
2	15151.588	1.614	80.7 %
4	9083.939	2.692	67.3 %
8	7247.534	3.374	42.2 %

16	6235.070	3.922	24.5 %
32	6003.104	4.074	12.7 %
64	5855.990	4.176	6.5 %

Number of logical cores: 12

Вывод

Многопоточная реализация неэффективна для малых матриц из-за преобладания накладных расходов над полезной работой. Алгоритм демонстрирует отрицательное масштабирование - добавление потоков ухудшает производительность.

Для малых матриц использовать последовательную версию; многопоточность целесообразна только для матриц большого размера, где объем вычислений компенсирует накладные расходы.