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-209БВ-24

Студент: Забродин Р.У.

Преподаватель: Миронов Е.С

Оценка: _____

Дата 23.12.25

Москва, 2025

Постановка задачи

Вариант 10.

Общее задание. Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработке 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.

Общий метод и алгоритм решения

Использованные системные вызовы:

- **malloc** — выделяет память в куче для динамических массивов и структур.
- **free** — освобождает память, ранее выделенную с помощью malloc.
- **pthread_create** — создает новый поток и запускает выполнение функции в этом потоке.
- **pthread_join** — ожидает завершения выполнения указанного потока.
- **pthread_exit** — завершает выполнение текущего потока.
- **clock** — получает текущее время с точностью до микросекунд.
- **srand** — инициализирует генератор случайных чисел.
- **rand** — генерирует псевдослучайное число.
- **time** — возвращает текущее время в секундах (используется для иници- ализации генератора случайных чисел).
- **atoi** — преобразует строку в целое число (используется для преобразования аргумента командной строки в число потоков).

Алгоритм работы программы:

- Вводятся коэффициенты при неизвестных в линейной системе уравнений.
- Создаются объекты матрица, решение и вводится количество разрешенных потоков.
- Вызывается функция parallel_gauss_elimination, разделяющая матрицу на подблоки, значения для которых считаются в различных подпотоках, вызываемых с непересекающимися наборами данных
- После вычислений каждого потока результаты записываются в свою часть вектора solution, после чего при помощи pthread_join основной поток дожидается выполнения всех подпотоков, после чего результат выводится на экран.

Код программы

```
#include <iostream>
#include <vector>
#include <cmath>
#include <pthread.h>
#include <chrono>

struct Task {
```

```
    std::vector<std::vector<double>>* matrix;
    std::vector<double>* solution;
    int start_row;
    int end_row;
    int current_col;
    int total_size;
    bool is_forward;
};

int created_threads = 0;
int max_threads_allowed = 1;

void* forward_elimination(void* arg) {
    Task* t = (Task*)arg;
    auto& matrix = *(t->matrix);
    int n = t->total_size;
    int current_col = t->current_col;
    for (int i = t->start_row; i <= t->end_row; i++) {
        if (i <= current_col) continue;
        double factor = matrix[i][current_col] / matrix[current_col][current_col];
        for (int j = current_col; j <= n; j++) {
            matrix[i][j] -= factor * matrix[current_col][j];
        }
    }
    delete t;
    return nullptr;
}

void* backward_substitution(void* arg) {
    Task* t = (Task*)arg;
    auto& matrix = *(t->matrix);
    auto& solution = *(t->solution);
    int n = t->total_size;
    for (int i = t->start_row; i >= t->end_row; i--) {
        solution[i] = matrix[i][n];
        for (int j = i + 1; j < n; j++) {
            solution[i] -= matrix[i][j] * solution[j];
        }
        solution[i] /= matrix[i][i];
    }
    delete t;
    return nullptr;
}

int find_pivot(std::vector<std::vector<double>>& matrix, int col, int n) {
    int pivot_row = col;
    double max_val = fabs(matrix[col][col]);
    for (int i = col + 1; i < n; i++) {
        if (fabs(matrix[i][col]) > max_val) {
            max_val = fabs(matrix[i][col]);
            pivot_row = i;
        }
    }
    return pivot_row;
}
```

```
void swap_rows(std::vector<std::vector<double>>& matrix, int row1, int row2, int n) {
    for (int j = 0; j <= n; j++) {
        std::swap(matrix[row1][j], matrix[row2][j]);
    }
}

void parallel_gauss_elimination(std::vector<std::vector<double>>& matrix, std::vector<double>& solution, int
threads) {
    int n = matrix.size();
    if (n == 0) return;
    max_threads_allowed = threads;
    for (int i = 0; i < n; i++) {
        int pivot = find_pivot(matrix, i, n);
        if (pivot != i) {
            swap_rows(matrix, i, pivot, n);
        }
        if (fabs(matrix[i][i]) < 1e-10) {
            std::cerr << "Матрица вырождена или почти вырождена(из-за погрешности double)" << std::endl;
            return;
        }
        int rows_per_thread = std::max(1, (n - i - 1) / threads);
        pthread_t* thread_pool = new pthread_t[threads];
        int thread_count = 0;
        for (int t = 0; t < threads; t++) {
            int start_row = i + 1 + t * rows_per_thread;
            int end_row = (t == threads - 1) ? n - 1 : start_row + rows_per_thread - 1;
            if (start_row >= n) break;
            Task* task = new Task{&matrix, &solution, start_row, end_row, i, n, true};
            pthread_create(&thread_pool[thread_count], nullptr, forward_elimination, task);
            thread_count++;
        }
        for (int t = 0; t < thread_count; t++) {
            pthread_join(thread_pool[t], nullptr);
        }
        delete[] thread_pool;
    }
    solution.resize(n);
    int rows_per_thread = std::max(1, n / threads);
    pthread_t* thread_pool = new pthread_t[threads];
    int thread_count = 0;
    for (int t = 0; t < threads; t++) {
        int start_row = n - 1 - t * rows_per_thread;
        int end_row = (t == threads - 1) ? 0 : start_row - rows_per_thread + 1;
        if (start_row < 0) break;
        Task* task = new Task{&matrix, &solution, start_row, end_row, 0, n, false};
        pthread_create(&thread_pool[thread_count], nullptr, backward_substitution, task);
        thread_count++;
    }
    for (int t = 0; t < thread_count; t++) {
        pthread_join(thread_pool[t], nullptr);
    }
    delete[] thread_pool;
}
```

```cpp
int main() {
    int n, threads;
    std::cout << "Введите размер системы и количество потоков: ";
    std::cin >> n >> threads;
    std::vector<std::vector<double>> matrix(n, std::vector<double>(n + 1));
    std::vector<double> solution;
    std::cout << "Введите матрицу коэффициентов и правые части:\n";
    for (int i = 0; i < n; i++) {
        for (int j = 0; j <= n; j++) {
            std::cin >> matrix[i][j];
        }
    }
    parallel_gauss_elimination(matrix, solution, threads);
    for (int i = 0; i < n; i++) {
        std::cout << solution[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)
project(gauss)
set(CMAKE_CXX_STANDARD 20)

add_executable(main main.cpp)
```

Протокол работы программы

Тестирование:

./main
2 5 4 1 20
1 3 2 1 11
2 10 9 7 40
3 8 9 2 37

1 2 2 3.706e-15 (Здесь из-за типа double появляется небольшая погрешность)

Вывод

На основе тестирования программы с разным количеством потоков и объемом данных можно сделать следующие выводы:

1) Многопоточность значительно ускоряет выполнение программы при грамотном распределении нагрузки между потоками.

2) Для максимального ускорения следует выбирать оптимальное количество потоков, которое соответствует вычислительным возможностям компьютера (например, числу ядер процессора) и объему задачи.

3) Избыточное количество потоков может снижать эффективность работы из-за накладных расходов на управление потоками и синхронизацию.

Таким образом, многопоточность является эффективным инструментом для повышения производительности, если её правильно применять в зависимости от аппаратных характеристик и сложности задачи.