Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа № 2 по курсу «Операционные системы»

Управление потоками в ОС.
Обеспечение синхронизации между потоками

Студент: Арусланов К. А.
Преподаватель: Миронов Е. С.
Группа: М8О-203Б-23
Дата:
Оценка:
Подпись:

Москва, 2024

Условие

Составить программу на языке C/C++, 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).

Метод решения

Реализована рекурсивная функция на C++ для вычисления определителя n×n-матрицы по формуле разложения по первой строке. Для ускорения использовалась многопоточность: при разбиении на миноры программа запускала потоки (через pthread), чтобы каждая часть вычислялась параллельно. Число одновременно работающих потоков ограничивалось специальным семафором или пулом заданий, чтобы избежать слишком большого распараллеливания и перегрузки системы.

Код программы

TDeterminantCalculator.h

```
#ifndef TDETERMINANT_CALCULATOR_H
#define TDETERMINANT_CALCULATOR_H

#include "TMatrix.h"
#include <queue>
#include <pthread.h>
#include <memory>

class TDeterminantCalculator {
public:
    TDeterminantCalculator(const TMatrix& Matrix, int numThreads);

    ~TDeterminantCalculator();

    // Метод для вычисления определителя
    double Compute();

private:
    // Задача для потока
```

```
    struct TTask {
        TMatrix Matrix;
        double element;

        TTask(const TMatrix& Mat, double e);
    };

    // Статическая функция-обёртка для потоков
    static void* WorkerHelper(void* arg);

    // Основная функция потока
    void* Worker();

    // Методы для QueueTask
    void PushTask(std::unique_ptr<TTask> Task);
    bool PopTask(std::unique_ptr<TTask>& Task);
    void Shutdown();

    TMatrix GetMinor(const TMatrix& Matrix, int row, int col);

    const TMatrix& Matrix_;
    int numThreads_;
    double globalDeterminant_;
    int activeTasks_;
    bool stop_;

    std::queue<std::unique_ptr<TTask>> taskQueue_;

    // Синхронизация
    pthread_mutex_t queueMutex_;
    pthread_cond_t queueCond_;
    pthread_mutex_t determinantMutex_;
    pthread_mutex_t activeTasksMutex_;
    pthread_cond_t activeTasksCond_;
};

#endif
```

TMatrix.h

```
#ifndef INPUT_TMATRIX_H
#define INPUT_TMATRIX_H

#include <vector>

using TMatrix = std::vector<std::vector<double>>;

TMatrix InputTMatrix();

#endif
```

TDeterminantCalculator.cpp

```
#include "TDeterminantCalculator.h"
#include <iostream>
#include <cstdlib>
#include <cstring>

TDeterminantCalculator::TTask::TTask(const TMatrix& Mat, double e)
    : Matrix(Mat), element(e) {}

TDeterminantCalculator::TDeterminantCalculator(const TMatrix& Matrix, int numThreads)
    : Matrix_(Matrix), numThreads_(numThreads), globalDeterminant_(0.0), activeTasks_(0),
stop_(false) {
    pthread_mutex_init(&queueMutex_, nullptr);
    pthread_cond_init(&queueCond_, nullptr);
    pthread_mutex_init(&determinantMutex_, nullptr);
    pthread_mutex_init(&activeTasksMutex_, nullptr);
    pthread_cond_init(&activeTasksCond_, nullptr);
}

TDeterminantCalculator::~TDeterminantCalculator() {
    pthread_mutex_destroy(&queueMutex_);
    pthread_cond_destroy(&queueCond_);
    pthread_mutex_destroy(&determinantMutex_);
    pthread_mutex_destroy(&activeTasksMutex_);
    pthread_cond_destroy(&activeTasksCond_);
}

void TDeterminantCalculator::PushTask(std::unique_ptr<TTask> Task) {
    pthread_mutex_lock(&queueMutex_);
    taskQueue_.push(std::move(Task));
    pthread_cond_signal(&queueCond_);
    pthread_mutex_unlock(&queueMutex_);
}

bool TDeterminantCalculator::PopTask(std::unique_ptr<TTask>& Task) {
    pthread_mutex_lock(&queueMutex_);
    while (taskQueue_.empty() && !stop_) {
        pthread_cond_wait(&queueCond_, &queueMutex_);
    }
    if (stop_ && taskQueue_.empty()) {
        pthread_mutex_unlock(&queueMutex_);
        return false;
    }
    Task = std::move(taskQueue_.front());
    taskQueue_.pop();
    pthread_mutex_unlock(&queueMutex_);
    return true;
}

void TDeterminantCalculator::Shutdown() {
```

```
    pthread_mutex_lock(&queueMutex_);
    stop_ = true;
    pthread_cond_broadcast(&queueCond_);
    pthread_mutex_unlock(&queueMutex_);
}

TMatrix TDeterminantCalculator::GetMinor(const TMatrix& Matrix, int row, int col) {
    TMatrix Minor;
    int n = Matrix.size();
    for (int i = 0; i < n; ++i) {
        if (i == row) {
            continue;
        }
        std::vector<double> minorRow;
        for (int j = 0; j < n; ++j) {
            if (j == col) {
                continue;
            }
            minorRow.push_back(Matrix[i][j]);
        }
        Minor.push_back(minorRow);
    }
    return Minor;
}

// Статическая функция-обёртка для потоков
void* TDeterminantCalculator::WorkerHelper(void* arg) {
    return ((TDeterminantCalculator*)arg)->Worker();
}

// Основная функция потока
void* TDeterminantCalculator::Worker() {
    while (true) {
        std::unique_ptr<TTask> Task;
        if (!PopTask(Task)) {
            break; // Очередь остановлена и задач больше нет
        }

        int n = Task->Matrix.size();

        if (n == 1) {
            // Базовый случай 1x1 матрица
            double det = Task->element * Task->Matrix[0][0];
            pthread_mutex_lock(&determinantMutex_);
            globalDeterminant_ += det;
            pthread_mutex_unlock(&determinantMutex_);
        }
        else if (n == 3) {
            // Базовый случайМ 3x3 матрица
            double a = Task->Matrix[0][0];
            double b = Task->Matrix[0][1];
            double c = Task->Matrix[0][2];
            double d = Task->Matrix[1][0];
```

```
            double e = Task->Matrix[1][1];
            double f = Task->Matrix[1][2];
            double g = Task->Matrix[2][0];
            double h = Task->Matrix[2][1];
            double i = Task->Matrix[2][2];

            // По методу звездочки
            double det = Task->element * (a*(e*i - f*h) - b*(d*i - f*g) + c*(d*h - e*g));

            pthread_mutex_lock(&determinantMutex_);
            globalDeterminant_ += det;
            pthread_mutex_unlock(&determinantMutex_);
        } else {
            // Создаем подзадачи
            int n_subtasks = n;
            pthread_mutex_lock(&activeTasksMutex_);
            activeTasks_ += n_subtasks;
            pthread_mutex_unlock(&activeTasksMutex_);

            for (int j = 0; j < n; ++j) {
                TMatrix Minor = GetMinor(Task->Matrix, 0, j); // Раскладываем по первой
строке
                double element = Task->Matrix[0][j];
                int sign = ((j % 2) == 0) ? 1 : -1;

                // Вычисляем новое накопленное значение элемента
                double newElement = Task->element * element * sign;

                // Создаем новую задачу и добавляем ее в очередь
                std::unique_ptr<TTask> newTask = std::make_unique<TTask>(Minor,
newElement);
                PushTask(std::move(newTask));
            }
        }

        pthread_mutex_lock(&activeTasksMutex_);
        activeTasks_--;
        int remaining_tasks = activeTasks_;
        if (remaining_tasks == 0) {
            // Уведомляем главный поток о завершении всех задач
            pthread_cond_signal(&activeTasksCond_);
        }
        pthread_mutex_unlock(&activeTasksMutex_);
    }
    return nullptr;
}

double TDeterminantCalculator::Compute() {
    globalDeterminant_ = 0.0;
    activeTasks_ = 0;
    stop_ = false;

    // Создание потоков
```

```
    std::vector<pthread_t> threads(numThreads_);
    for (int i = 0; i < numThreads_; ++i) {
        int ret = pthread_create(&threads[i], nullptr,
&TDeterminantCalculator::WorkerHelper, this);
        if (ret != 0) {
            std::cerr << "Ошибка создания потока: " << strerror(ret) << std::endl;
            exit(1);
        }
    }

    pthread_mutex_lock(&activeTasksMutex_);
    activeTasks_ = 1; // Корневая задача
    pthread_mutex_unlock(&activeTasksMutex_);

    std::unique_ptr<TTask> InitialTask = std::make_unique<TTask>(Matrix_, 1.0);
    PushTask(std::move(InitialTask));

    // Ожидаем завершения всех задач
    pthread_mutex_lock(&activeTasksMutex_);
    while (activeTasks_ > 0) {
        pthread_cond_wait(&activeTasksCond_, &activeTasksMutex_);
    }
    pthread_mutex_unlock(&activeTasksMutex_);

    Shutdown(); // Останавливаем очередь задач

    // Ожидаем завершения всех потоков
    for (int i = 0; i < numThreads_; ++i) {
        pthread_join(threads[i], nullptr);
    }

    return globalDeterminant_;
}
```

TMatrix.cpp

```
#include <iostream>
#include "TMatrix.h"

TMatrix InputTMatrix() {
    int n;
    std::cout << "Введите размер матрицы (n x n): ";
    std::cin >> n;

    if (n <= 0) {
        std::cerr << "Размер матрицы должен быть положительным." << std::endl;
        exit(1);
    }

    TMatrix Matrix(n, std::vector<double>(n));
    std::cout << "Введите элементы матрицы построчно:\n";
```

```
    for (int i = 0; i < n; ++i) {
        std::cout << "Строка " << i + 1 << ": ";
        for (int j = 0; j < n; ++j) {
            std::cin >> Matrix[i][j];
        }
    }

    return Matrix;
}
```

CMakeLists.txt

```
add_executable(lab2 main.cpp src/TDeterminantCalculator.cpp
include/TDeterminantCalculator.h src/TMatrix.cpp include/TMatrix.h)

target_include_directories(lab2 PRIVATE include)
```

main.cpp

```
#include <iostream>
#include "TDeterminantCalculator.h"
#include "TMatrix.h"

int main(int argc, char* argv[]) {
    // Обработка аргументов командной строки
    if (argc != 3 || std::string(argv[1]) != "-t") {
        std::cerr << "Использование: " << argv[0] << " -t <число_потоков>" << std::endl;
        return 1;
    }

    int numThreads;
    try {
        numThreads = std::stoi(argv[2]);
    } catch (const std::exception& e) {
        std::cerr << "Ошибка: неверное значение числа потоков." << std::endl;
        return 1;
    }

    if (numThreads <= 0) {
        std::cerr << "Ошибка: число потоков должно быть положительным." << std::endl;
        return 1;
    }

    // Ввод матрицы пользователем
    TMatrix Matrix = InputTMatrix();

    // Вычисляем определитель (разложение по первой строке)
    TDeterminantCalculator Calculator(Matrix, numThreads);
    double determinant = Calculator.Compute();
```

```
    std::cout << "Определитель: " << determinant << std::endl;

    return 0;
}
```

lab2_test.cpp

```
#include <TDeterminantCalculator.h>
#include <TMatrix.h>
#include <TDeterminantCalculator.h>
#include <TMatrix.h>
#include <gtest/gtest.h>
#include <chrono>

// Проверка корректности однопоточного варианта
TEST(DeterminantCalculatorTest, SingleThreadCorrectness) {
    // 1x1 матрица
    TMatrix Mat1 = {{5}};
    TDeterminantCalculator Calc1(Mat1, 1);
    double det1 = Calc1.Compute();
    EXPECT_DOUBLE_EQ(det1, 5.0);

    // 2x2 матрица
    TMatrix Mat2 = {
        {1, 2},
        {3, 4}
    };
    TDeterminantCalculator Calc2(Mat2, 1);
    double det2 = Calc2.Compute();
    EXPECT_DOUBLE_EQ(det2, -2.0);

    // 3x3 матрица
    TMatrix Mat3 = {
        {6, 1, 1},
        {4, -2, 5},
        {2, 8, 7}
    };
    TDeterminantCalculator Calc3(Mat3, 1);
    double det3 = Calc3.Compute();
    EXPECT_DOUBLE_EQ(det3, -306.0);

    // 4x4 матрица
    TMatrix Mat4 = {
        {3, 2, 0, 1},
        {4, 0, 1, 2},
        {3, 0, 2, 1},
        {9, 2, 3, 1}
    };
    TDeterminantCalculator Calc4(Mat4, 1);
    double det4 = Calc4.Compute();
```

```
    EXPECT_DOUBLE_EQ(det4, 24.0);
}

// Проверка соответствия результатов однопоточного и многопоточного вариантов
TEST(DeterminantCalculatorTest, SingleVsMultiThreadConsistency) {
    // 3x3 матрица
    TMatrix Mat = {
        {6, 1, 1},
        {4, -2, 5},
        {2, 8, 7}
    };

    // Однопоточный вариант
    TDeterminantCalculator CalcSingle(Mat, 1);
    double detSingle = CalcSingle.Compute();

    // Многопоточный вариант
    for (int i = 2; i <= 8; i += 2) {
        TDeterminantCalculator CalcMulti(Mat, i);
        double detMulti = CalcMulti.Compute();
        EXPECT_DOUBLE_EQ(detSingle, detMulti);
    }
}

// Проверка производительности многопоточного варианта
TEST(DeterminantCalculatorTest, MultiThreadPerformance) {
    // 11x11 матрица
    TMatrix Mat(11, std::vector<double>(11, 2.0));

    // Измеряем время для однопоточного варианта
    TDeterminantCalculator CalcSingle(Mat, 1);
    auto startSingle = std::chrono::high_resolution_clock::now();
    double detSingle = CalcSingle.Compute();
    auto endSingle = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> durationSingle = endSingle - startSingle;

    // Измеряем время для многопоточного варианта (например, 8 потоков)
    TDeterminantCalculator CalcMulti(Mat, 8);
    auto startMulti = std::chrono::high_resolution_clock::now();
    double detMulti = CalcMulti.Compute();
    auto endMulti = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> durationMulti = endMulti - startMulti;

    // Проверяем, что результаты совпадают
    EXPECT_DOUBLE_EQ(detSingle, detMulti);

    // Проверяем, что многопоточный вариант быстрее
    std::cout << "Однопоточный вариант: " << durationSingle.count() << " секунд\n";
    std::cout << "Многопоточный вариант: " << durationMulti.count() << " секунд\n";

    EXPECT_LT(durationMulti.count(), durationSingle.count());
}
```

```
int main(int argc, char **argv) {
    testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```

Выводы

Реализация многопоточного вычисления детерминанта позволила ускорить расчёты для больших матриц, но при слишком глубокой рекурсии и неразумном количестве потоков возникали накладные расходы. Контроль числа активных потоков и грамотная синхронизация (семафоры, мьютексы) стали ключевыми факторами стабильной и эффективной работы. Лабораторная работа показала, как многопоточность может существенно ускорить вычислительные задачи, но только при правильном управлении ресурсами.