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-215Б-23

Студент: Лапенко К.А.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 21.02.24

Москва, 2024

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.

9. Рассчитать детерминант матрицы (используя определение детерминанта)

Общий метод и алгоритм решения

Использованные системные вызовы:

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg) – Создаёт новый поток. Поток начинает выполнение функции start_routine, в данном случае — calculate_determinant_thread. Аргументы передаются через структуру ThreadData.
- int pthread_join(pthread_t thread, void **retval) – Ожидает завершения указанного потока. Основной поток блокируется до завершения рабочих потоков, чтобы корректно собрать результаты.
- int sem_init(sem_t *sem, int pshared, unsigned int value) – Инициализирует семафор sem. В программе используется для ограничения количества одновременно работающих потоков (value = max_threads). Параметр pshared = 0указывает, что семафор доступен только внутри процесса.
- int sem_wait(sem_t *sem) – Уменьшает значение семафора на 1. Если значение становится отрицательным, поток блокируется до освобождения слота. В программе вызывается перед созданием потока, чтобы контролировать их количество.
- int sem_post(sem_t *sem) – Увеличивает значение семафора на 1. Вызывается потоком после завершения вычислений, чтобы разрешить запуск новых потоков.
- int sem_destroy(sem_t *sem) – Уничтожает семафор и освобождает связанные ресурсы. Вызывается в конце программы для корректного завершения работы с семафором.

Алгоритм решения:

1. Инициализация программы и обработка аргументов
   - Аргументы командной строки:
     Программа принимает два параметра:
     - Размер квадратной матрицы
     - Максимальное количество потоков для параллельных вычислений.
   - Проверка аргументов:
     - Если аргументы не указаны или их количество неверно, выводится сообщение об ошибке.
     - Проверяется корректность чисел (положительные значения).

2. Генерация матрицы

- Создание матрицы:
    - Матрица размером n x n заполняется случайными целыми числами от 1 до 1000.
    - Для генерации используется функция rand(), инициализированная текущим временем (srand(time(nullptr))).

3. Настройка многопоточности

- Инициализация семафора:
    - Создается семафор (sem_t), который ограничивает количество одновременно работающих потоков.
    - Начальное значение семафора равно max_threads.
    - Пример: если max_threads = 4, то одновременно могут выполняться 4 потока.

4. Распределение задач между потоками

- Определение числа потоков:
    - Фактическое число потоков num_threads выбирается как минимум из max_threads и n(размера матрицы).
    - Например, для матрицы 5x5 и max_threads = 10 будет создано 5 потоков.
- Распределение столбцов:
    - Столбцы матрицы делятся между потоками поровну.
    - Если n не делится нацело на num_threads, первые remaining_cols = n % num_threads потоков получают на один столбец больше.
- Структура ThreadData:
  Каждому потоку передается структура с данными:
    - start_col и end_col — диапазон столбцов для обработки.
    - matrix — исходная матрица.
    - result — частичный результат (вклад потока в детерминант).

5. Создание и запуск потоков

- Механизм семафора:
    - Перед созданием потока вызывается sem_wait(semaphore). Если счетчик семафора > 0, поток создается.
    - Если счетчик = 0, выполнение блокируется до освобождения слота.
- Создание потоков:
    - Для каждого потока:
        1. Выделяется диапазон столбцов.
        2. Инициализируется ThreadData.
        3. Вызывается pthread_create, который запускает функцию calculate_determinant_thread.

6. Вычисление детерминанта в потоках

- Функция calculate_determinant_thread:
    - Для каждого столбца в своём диапазоне поток:
        1. Создает подматрицу размером (n-1) * (n-1), исключая первую строку и текущий столбец.
        2. Вычисляет вклад элемента matrix[0][i] в детерминант
        3. Суммирует term в thread_data[i].result.
- Рекурсивная функция determinant:

    - Для подматриц размером >2x2 используется рекурсивное разложение по первой строке.
    - Для матриц 1x1 и 2x2 применяются частные случаи.
- Освобождение семафора:
  После завершения вычислений поток вызывает sem_post(semaphore), увеличивая счетчик семафора.

7. Сбор результатов
- Ожидание завершения потоков:
    - Основной поток вызывает pthread_join для каждого созданного потока, чтобы дождаться их завершения.
- Суммирование результатов:
    - Частичные результаты из thread_data[i].result суммируются в переменную det.

8. Вывод результата и завершение
- Измерение времени:
    - Время выполнения замеряется с помощью std::chrono.
- Вывод:
    - На экран выводится детерминант и время работы.
- Освобождение ресурсов:
    - Уничтожается семафор (sem_destroy).
    - Освобождается память, выделенная под потоки и данные.

Код программы

main.c

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <cstring>
#include <chrono>



// Структура для передачи данных в поток
struct ThreadData {
   int thread_id;
   int n;                          // размер матрицы
   std::vector<std::vector<int> > matrix;
   int start_col;                  // начальный столбец для потока
   int end_col;                    // конечный столбец для потока
   double result;                  // результат вычислений потока
};

sem_t* semaphore = nullptr;         // семафор для ограничения числа потоков
```

```
// Функция для нахождения детерминанта подматрицы
double determinant(const std::vector<std::vector<int> >& matrix, int n) {
    // Если размер матрицы 1x1, возвращаем единственный элемент
    if (n == 1) {
        return matrix[0][0];
    }

    // Если размер матрицы 2x2, используем формулу ad-bc
    if (n == 2) {
        return matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0];
    }

    double det = 0;
    int sign = 1;

    // Вычисляем детерминант через разложение по первой строке
    for (int i = 0; i < n; i++) {
        // Создаем подматрицу (n-1) x (n-1)
        std::vector<std::vector<int> > submatrix(n - 1, std::vector<int>(n - 1));

        // Заполняем подматрицу
        for (int j = 1; j < n; j++) {
            int col_idx = 0;
            for (int k = 0; k < n; k++) {
                if (k == i) continue;
                submatrix[j-1][col_idx++] = matrix[j][k];
            }
        }

        // Суммируем с учётом знака
        det += sign * matrix[0][i] * determinant(submatrix, n - 1);
        sign = -sign;
    }

    return det;
}

// Функция для потока
void* calculate_determinant_thread(void* arg) {
    ThreadData* data = (ThreadData*)arg;

    // Вычисляем детерминант для своего диапазона столбцов
    data->result = 0;
    int sign = (data->start_col % 2 == 0) ? 1 : -1;  // Корректируем знак для
начального столбца

    for (int i = data->start_col; i < data->end_col; i++) {
        // Создаем подматрицу (n-1) x (n-1)
        std::vector<std::vector<int> > submatrix(data->n - 1,
std::vector<int>(data->n - 1));
```

```
        // Заполняем подматрицу (исключаем первую строку и i-й столбец)
        for (int j = 1; j < data->n; j++) {
            int col_idx = 0;
            for (int k = 0; k < data->n; k++) {
                if (k == i) continue;
                submatrix[j-1][col_idx++] = data->matrix[j][k];
            }
        }

        // Вычисляем вклад текущего элемента в детерминант
        double term = sign * data->matrix[0][i] * determinant(submatrix, data->n -
1);
        data->result += term;

        sign = -sign;
    }

    // Освобождаем семафор, позволяя другому потоку запуститься
    sem_post(semaphore);

    return nullptr;
}

// Функция для вывода матрицы (для отладки)
    void print_matrix(const std::vector<std::vector<int> >& matrix, int n) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                std::cout << matrix[i][j] << "\t";
            }
            std::cout << std::endl;
        }
    }

int main(int argc, char* argv[]) {
    // Проверка аргументов командной строки
    if (argc != 3) {
        std::cerr << "Использование: " << argv[0] << " <размер_матрицы>
<макс_число_потоков>" << std::endl;
        return 1;
    }

    // Парсинг аргументов
    int n = atoi(argv[1]);             // размер матрицы
    int max_threads = atoi(argv[2]);   // максимальное число потоков

    // Проверка валидности аргументов
    if (n <= 0) {
        std::cerr << "Размер матрицы должен быть положительным числом" <<
std::endl;
        return 1;
```

```
    }
    if (max_threads <= 0) {
        std::cerr << "Максимальное число потоков должно быть положительным числом" << std::endl;
        return 1;
    }

    // Инициализация генератора случайных чисел
    srand(time(nullptr));

    // Создание и заполнение матрицы случайными числами от 1 до 1000
    std::vector<std::vector<int> > matrix(n, std::vector<int>(n));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = rand() % 1000 + 1;
        }
    }

    auto start = std::chrono::high_resolution_clock::now();
    // Инициализация семафора с нужным количеством потоков
    semaphore = new sem_t;
    sem_init(semaphore, 0, max_threads);

    // Специальная обработка для маленьких матриц
    if (n == 1) {
        std::cout << "Детерминант: " << matrix[0][0] << std::endl;
        sem_destroy(semaphore);
        delete semaphore;
        return 0;
    }

    // Определяем фактическое число потоков
    int num_threads = std::min(max_threads, n);  // Не больше, чем размер матрицы

    // Создаем потоки
    pthread_t* threads = new pthread_t[num_threads];
    ThreadData* thread_data = new ThreadData[num_threads];

    // Равномерно распределяем столбцы между потоками
    int cols_per_thread = n / num_threads;
    int remaining_cols = n % num_threads;

    int start_col = 0;
    for (int i = 0; i < num_threads; i++) {
        // Определяем диапазон столбцов для потока
        int cols = cols_per_thread + (i < remaining_cols ? 1 : 0);

        // Инициализируем данные для потока
        thread_data[i].thread_id = i;
        thread_data[i].n = n;
```

```
        thread_data[i].matrix = matrix;
        thread_data[i].start_col = start_col;
        thread_data[i].end_col = start_col + cols;
        thread_data[i].result = 0;

        // Ждем доступный слот (семафор)
        // Это уменьшает счетчик семафора. Если счетчик становится
        // отрицательным, поток блокируется до вызова sem_post
        sem_wait(semaphore);

        // Создаем поток
        pthread_create(&threads[i], nullptr, calculate_determinant_thread,
&thread_data[i]);

        // Обновляем начальный столбец для следующего потока
        start_col += cols;
    }

    // Ожидаем завершения всех потоков и суммируем результаты
    double det = 0;
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], nullptr);
        det += thread_data[i].result;
    }

    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;

    std::cout << "Заняло: " << duration.count() << " секунд\n";
    // Выводим результат
    std::cout << "Детерминант: " << det << std::endl;

    // Освобождаем ресурсы
    sem_destroy(semaphore);
    delete semaphore;
    delete[] threads;
    delete[] thread_data;

    return 0;
}
```

test_det.sh

```
#!/bin/bash

# Компилируем программу
g++ -o matrix_determinant main.cpp -pthread

# Проверяем, что компиляция прошла успешно
```

```
if [ $? -ne 0 ]; then
    echo "Ошибка компиляции."
    exit 1
fi

# Размеры матриц для тестирования
MATRIX_SIZES=(4 8 10 11 12)
# Количество потоков для тестирования
THREAD_COUNTS=(1 2 3 4 5 6 7 8 9 10 11 12)

# Заголовок таблицы результатов
echo "Размер_матрицы Макс_потоков Время_мс      Ускорение Эффективность"

# Функция для измерения времени, работающая на разных платформах
get_time_ms() {
    if [[ "$OSTYPE" == "darwin"* ]]; then
        # MacOS - используем perl для точного измерения времени
        perl -MTime::HiRes=time -e 'printf "%.6f\n", time'
    else
        # Linux
        date +%s.%N
    fi
}

# Для каждого размера матрицы
for size in "${MATRIX_SIZES[@]}"; do
    # Сохраняем время выполнения для одного потока, чтобы рассчитать ускорение
    single_thread_time=0

    # Для каждого количества потоков
    for threads in "${THREAD_COUNTS[@]}"; do
        # Если количество потоков превышает размер матрицы, пропускаем
        # (т.к. программа ограничивает фактическое число потоков размером матрицы)
        if [ $threads -gt $size ]; then
            continue
        fi

        # Запускаем программу и замеряем время
        start_time=$(get_time_ms)
        ./matrix_determinant $size $threads > /dev/null 2>&1
        end_time=$(get_time_ms)

        # Вычисляем время выполнения в миллисекундах
        elapsed_time=$(echo "($end_time - $start_time) * 1000" | bc)
        elapsed_ms=$(printf "%.2f" $elapsed_time)

        # Для одного потока сохраняем время как базовое
        if [ $threads -eq 1 ]; then
            single_thread_time=$elapsed_time
```

```
        fi

        # Вычисляем ускорение и эффективность
        speedup=$(echo "$single_thread_time / $elapsed_time" | bc -l)
        speedup=$(printf "%.2f" $speedup)

        efficiency=$(echo "$speedup / $threads" | bc -l)
        efficiency=$(printf "%.2f" $efficiency)

        # Выводим результаты
        printf "%-15s %-13s %-14s %-10.2f %-13.2f\n" "$size" "$threads"
"$elapsed_ms" "$speedup" "$efficiency"
    done
    echo "----------------------------------------------------------------"
done
```

monitor_threads.sh

```
#!/bin/bash

# Компилируем программу
g++ -pthread main.cpp -o matrix_determinant

# Запускаем программу в фоне
./matrix_determinant 11 6 &
PROG_PID=$!

echo "Program PID: $PROG_PID"
echo "Monitoring threads..."

# Мониторим каждую секунду
while kill -0 $PROG_PID 2>/dev/null; do
    # Используем ps для подсчета количества потоков
    thread_count=$(ps -M $PROG_PID | wc -l)
    # Вычитаем заголовок и сам процесс, чтобы получить только количество потоков
    thread_count=$((thread_count - 2))
    echo "$(date +%T%3N) - Number of threads: $thread_count"
    sleep 1
done

wait $PROG_PID
```

Протокол работы программы

Тестирование:

При тестировании программы при различных размерах матрицы и максимальном количестве потоков, были получены такие результаты:

Размер матрицы	Максимальное число потоков	Время (мс)	Ускорение	Эффективность

4	1	327.02	1.00	1.00
4	2	15.35	21.31	10.65
4	3	10.07	32.46	10.82
4	4	9.32	35.10	8.78
8	1	30.59	1.00	1.00
8	2	21.57	1.42	0.71
8	3	18.73	1.63	0.54
8	4	15.83	1.93	0.48
8	5	15.87	1.93	0.39
8	6	17.43	1.76	0.29
8	7	14.98	02.04	0.29
8	8	14.79	02.07	0.26
10	1	1939.04	1.00	1.00
10	2	1030.10	1.88	0.94
10	3	828.18	2.34	0.78
10	4	676.63	2.87	0.72
10	5	558.40	3.47	0.69
10	6	588.09	3.30	0.55
10	7	573.95	3.38	0.48
10	8	537.51	3.61	0.45
10	9	538.85	3.60	0.40
10	10	482.04	04.02	0.40
11	1	21078.84	1.00	1.00
11	2	12088.76	1.74	0.87
11	3	8530.75	2.47	0.82
11	4	6485.81	3.25	0.81
11	5	7388.22	2.85	0.57
11	6	6141.54	3.43	0.57
11	7	6095.71	3.46	0.49
11	8	6747.21	3.12	0.39
11	9	6856.67	03.07	0.34
11	10	7053.45	2.99	0.30
11	11	5992.70	3.52	0.32

12	1	252725.71	1.00	1.00
12	2	136794.51	1.85	0.93
12	3	107633.76	2.35	0.78
12	4	106191.63	2.38	0.59
12	5	96145.76	2.63	0.53
12	6	81098.92	3.12	0.52
12	7	84907.06	2.98	0.43
12	8	83860.40	03.01	0.38
12	9	82302.85	03.07	0.34
12	10	85831.04	2.94	0.29
12	11	90127.18	2.80	0.25
12	12	76194.00	3.32	0.28

Strace:

Вывод

Было интересно разобраться с потоками. Поняла, что важно внимательно следить за ними, во избежание непредвиденных ситуаций. Самым сложным было держать в голове для какого процесса я сейчас пишу код и как он может повлиять на другие процессы, которые будут этот код выполнять.