Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №3 по курсу**

**«Операционные системы»**

**Управление потоками в ОС. Обеспечение синхронизации между потоками.**

Студент: Епанешников Владислав Сергеевич

Группа: М80 – 206Б-19

Вариант: 9

Преподаватель: Миронов Евгений Сергеевич

Дата: 21.04.2021

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2021

1. **Постановка задачи**

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработке использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение потоков должно быть задано ключом запуска программы.

Необходимо уметь продемонстрировать количество потоков, используемых программой, с помощью стандартных средств операционной системы.

Привести исследование зависимости ускорения и эффективности алгоритма от входящих данных и количества потоков. Объяснить получившиеся результаты.

Вариант заданий: рассчитать детерминант матрицы.

1. **Общие сведения о программе**

Для запуска программы в качестве аргумента командной строки необходимо указать количество потоков, которые могут быть использованы программой.

Программа содержит две структуры: квадратную матрицу и аргументы для каждого потока.

Программа включает в себя потоковую функцию void\* \_matrix(void \*Args), которая вычисляет миноры, причём у каждого потока свой диапазон, в котором он работает. Так как все потоки программы работают в одном и том же пространстве памяти, аргументы для передачи потоковой функции хранятся по разным адресам (в массиве, размер которого равен количеству потоков).

В программе предусмотрена проверка на системные ошибки – ошибки выделения памяти, ошибки запуска.

1. **Общий метод и алгоритм решения**

Для реализации поставленной задачи необходимо:

* Изучить принципы работы read, write, pthread\_create, pthread\_join.
* Написать считывание матрицы в память.
* Написать функцию расчёта детерминанта матрицы с использованием потоков.
* Отладить программу и протестировать на тестах.

**Исследование искорения и эффективности от количества потоков.**

Si – ускорение, где i = количество потоков.

Si = T1/Ti

Ei – эффективность, где i = количество потоков.

Ei = Si / i

Ti – время работы алгоритма, где i = количество потоков.

1. **Матрица 11 порядка:**

S2 = 1.74 E2 = 0.87

S4 = 3.28 E4 = 0.82

S8 = 3,39 E8 = 0.42

S11 = 3.67 E16 = 0.33

На матрице 11 порядка лучшее ускорение и лучшую эффективность показал алгоритм на 2 потоках.

2. **Матрица 12 порядка:**

S2 = 2.30 E2 = 1.15

S4 = 3.00 E4 = 0.75

S8 = 2.68 E8 = 0.335

S12 = 2.59 E16 = 0.16

На матрице 12 порядка лучшее ускорение показал алгоритм на 4 потоках, причем от 2 до 4 потоков ускорение росло, от 4 до 12, что говорит о том, что не всегда, чем больше потоков, тем лучше. Лучшую эффективность показал алгоритм на 2 потоках.

1. **Основные файлы программы**

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <time.h>

#include <math.h>

#define SEC 1000000000

typedef long long int ll;

typedef struct {

int \*\*mat;

size\_t order;

} Matrix;

typedef struct {

size\_t begin;

size\_t end;

Matrix \*mat;

ll \*result;

} Args;

void m\_init(Matrix \*m, size\_t size) {

m->order = size;

m->mat = (int\*\*) malloc(sizeof(int\*) \* m->order);

for (size\_t i = 0; i < m->order; ++i) {

m->mat[i] = (int\*) malloc(sizeof(int) \* m->order);

}

}

void m\_input(Matrix \*m) {

int value;

scanf("%zd", &m->order);

m->mat = (int\*\*) malloc(sizeof(int\*) \* m->order);

for (size\_t i = 0; i < m->order; ++i) {

m->mat[i] = (int\*) malloc(sizeof(int) \* m->order);

for (size\_t j = 0; j < m->order; ++j) {

scanf("%d", &value);

m->mat[i][j] = value;

}

}

}

void m\_print(Matrix \*m) {

for (size\_t i = 0; i < m->order; ++i) {

for (size\_t j = 0; j < m->order; ++j) {

printf("%d ", m->mat[i][j]);

}

printf("\n");

}

}

void m\_delete(Matrix \*m) {

for (size\_t i = 0; i < m->order; ++i) {

free(m->mat[i]);

}

free(m->mat);

}

void getMatrix(Matrix \*matrix, size\_t row, size\_t col, Matrix \*newMatrix) {

size\_t offsetRow = 0;

size\_t offsetCol = 0;

for (size\_t i = 0; i < newMatrix->order; ++i) {

if (i == row) {

offsetRow = 1;

}

offsetCol = 0;

for (size\_t j = 0; j < newMatrix->order; ++j) {

if (j == col) {

offsetCol = 1;

}

newMatrix->mat[i][j] = matrix->mat[i + offsetRow][j + offsetCol];

}

}

}

ll determinant(Matrix \*m) {

ll det = 0;

int degree = 1;

if (m->order == 1) {

return m->mat[0][0];

}

else if (m->order == 2) {

return m->mat[0][0] \* m->mat[1][1] - m->mat[0][1] \* m->mat[1][0];

}

else {

Matrix newMatrix;

m\_init(&newMatrix, m->order - 1);

for (size\_t j = 0; j < m->order; ++j) {

getMatrix(m, 0, j, &newMatrix);

det += degree \* m->mat[0][j] \* determinant(&newMatrix);

degree \*= -1;

}

m\_delete(&newMatrix);

}

return det;

}

void \*\_matrix(void \*args) {

Args \*params = (Args\*) args;

Matrix newMatrix;

m\_init(&newMatrix, params->mat->order - 1);

ll temp = 0;

for (size\_t i = params->begin; i < params->end; ++i) {

getMatrix(params->mat, 0, i, &newMatrix);

temp += pow(-1, i) \* params->mat->mat[0][i] \* determinant(&newMatrix);

}

\*params->result = temp;

m\_delete(&newMatrix);

return NULL;

}

ll determiant\_threads(Matrix \*m, size\_t threads\_limit) {

size\_t quotient = m->order / threads\_limit;

size\_t remainder = m->order % threads\_limit;

pthread\_t threads[threads\_limit];

size\_t it = 0;

Args data[threads\_limit];

ll results[threads\_limit];

for (size\_t i = 0; i < threads\_limit; ++i) {

data[i].result = &results[i];

data[i].begin = it;

data[i].mat = m;

it += quotient;

if (remainder > 0) {

++it;

--remainder;

}

data[i].end = it;

if (pthread\_create(&threads[i], NULL, \_matrix, &data[i]) != 0) {

printf("error with thread creating occured\n");

exit(-2);

}

}

ll result = 0;

for (size\_t i = 0; i < threads\_limit; ++i) {

pthread\_join(threads[i], NULL);

result += results[i];

}

return result;

}

int main(int argc, char \*argv[]) {

if (argc != 2 || atoi(argv[1]) == 0) {

printf("Bad arguments\n");

exit(-1);

}

Matrix m;

m\_input(&m);

size\_t threads\_limit = atoi(argv[1]);

struct timespec start, end;

if (m.order < threads\_limit) {

threads\_limit = m.order;

}

timespec\_get(&start, TIME\_UTC);

ll result = determiant\_threads(&m, threads\_limit);

timespec\_get(&end, TIME\_UTC);

printf("Time with %zu threads: %lf\nResult: %lld\n", threads\_limit, ((end.tv\_sec \* SEC + end.tv\_nsec) -

(start.tv\_sec \* SEC + start.tv\_nsec)) / (double) SEC, result);

timespec\_get(&start, TIME\_UTC);

result = determinant(&m);

timespec\_get(&end, TIME\_UTC);

printf("Time without threads: %lf\nResult: %lld\n", ((end.tv\_sec \* SEC + end.tv\_nsec) -

(start.tv\_sec \* SEC + start.tv\_nsec)) / (double) SEC, result);

m\_delete(&m);

return 0;

}

1. **Демонстрация работы программы**

MacBook:src vladislove$ cat ../tests/matrix12

12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

MacBook:src vladislove$ ./a.out 12 < ../tests/matrix12

Time with 12 threads: 18.938294

Result: 0

Time without threads: 52.415567

Result: 0

MacBook:src vladislove$ cat ../tests/matrix3

3

9 2 3

4 5 6

7 8 9

MacBook:src vladislove$ ./a.out 3 < ../tests/matrix3

Time with 3 threads: 0.000275

Result: -24

Time without threads: 0.000002

Result: -24

1. **Выводы**

Данная лабораторная работа направлена на изучение потоков в языке СИ. Для работы с потоками необходимо подключить библиотеку pthread.h.

Преимущества потоков перед процессами:

1. Создать поток быстрее, чем создать процесс.

2. Потоки используют одну область памяти, следовательно, с помощью них можно ускорить вычисление каких-либо данных, отводя каждому потоку какое-то определенное действие.

На СИ можно писать программы использующие распараллеливание, ускорение и эффективность алгоритма зависит от количества данных и количества потоков, на маленьких данных лучше работают алгоритмы использующие небольшое количество потоков, чем больше данных тем лучше начинают работать алгоритмы с большим числом потоков, однако улучшение не может наблюдаться бесконечно, поэтому в какой-то момент с увеличением числа потоков ускорение не увеличится, а эффективность будет падать.