Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №3 по курсу**

**«Операционные системы»**

**Тема работы**

Студент: Павлов Иван Дмитриевич

Группа: М8О-207Б-21

Вариант: 9

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2021

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Общий метод и алгоритм решения
5. Исходный код
6. Демонстрация работы программы
7. Выводы

**Репозиторий**

https://github.com/Pavloffff/MAI\_OS/tree/main/lab3

**Постановка задачи**

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение потоков должно быть задано ключом запуска вашей программы. Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.

**Вариант 9**: Рассчитать детерминант матрицы

**Общие сведения о программе**

Существует две программы под каждый алгоритм. Программы представляют из себя один файл main1.c и main2.c.

**Общий метод и алгоритм решения**

Существует 2 широко известных алгоритма подсчета детерминанта матрицы — рекурсивная формула и метод Гаусса. Алгоритм Гаусса быстрее (O(n³) по сравнению с O(n!), где n — размер матрицы, также его проще записать в многопоточном режиме. Принцип работы алгоритма: приводим матрицу к верхнетреугольному виду, затем перемножает элементы главной диагонали, при этом строки матрицы я распределил поровну между потоками (последний поток обрабатывает больше строк если есть остаток). Также я реализовал и рекурсивный вариант: разбивается на потоки только первый вызов рекурсии (первая строка), при этом каждый поток считает свой столбец.

**Исходный код**

**main1.c**

#include <stdio.h>

#include <pthread.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <unistd.h>

#include <time.h>

double \*\*matrix = NULL;

typedef struct

{

int rows;

int n;

int i;

} threadArgs;

void print(double \*\*matrix, int n)

{

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

printf("%.2lf ", matrix[i][j]);

}

printf("\n");

}

}

void clearMinor(double \*\*matrix, int n)

{

for (int i = 0; i < n; i++) {

free(matrix[i]);

matrix[i] = NULL;

}

free(matrix);

matrix = NULL;

}

void \*routine(void \*args)

{

threadArgs \*arg = (threadArgs \*) args;

int n = arg->n, i = arg->i, rows = arg->rows;

for (int k = 0; k < i + rows; k++) {

for (int j = k + 1; j < n; j++) {

double del = (-1) \* matrix[j][k] / matrix[k][k];

for (int x = 0; x < n; x++) {

matrix[j][x] += matrix[k][x] \* del;

}

}

}

free(args);

return NULL;

}

double det(double \*\*matrix, int n)

{

double res = 1;

for (int i = 0; i < n - 1; i++) {

for (int j = i + 1; j < n; j++) {

double del = (-1) \* matrix[j][i] / matrix[i][i];

for (int k = 0; k < n; k++) {

matrix[j][k] += matrix[i][k] \* del;

}

}

}

for (int i = 0; i < n; i++) {

res \*= matrix[i][i];

}

return res;

}

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

{

if (argc < 3 || argc > 4) {

printf("Syntax error: expected ./\*executable\_file\_name\* Square\_matrix\_dim Number\_of\_threads\n");

printf("or ./\*executable\_file\_name\* Square\_matrix\_dim Number\_of\_threads -t\n");

exit(1);

}

int n = atoi(argv[1]), cntOfThreads = atoi(argv[2]);

if (cntOfThreads > n - 1) {

printf("Error: Number\_of\_threads must be less then Square\_matrix\_dim\n");

exit(1);

}

pthread\_t \*threads = (pthread\_t \*) calloc(cntOfThreads, sizeof(double));

if (threads == NULL) {

printf("Allocation error: can't allocate array of threads\n");

exit(1);

}

matrix = malloc(sizeof(double \*) \* n);

for (int i = 0; i < n; i++) {

matrix[i] = malloc(sizeof(double) \* n);

}

if (matrix == NULL) {

printf("Allocation error: can't allocate exeptet matrix\n");

}

if (argc == 3) {

printf("Enter the square matrix dim of %d:\n", n);

}

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

scanf("%lf", &matrix[i][j]);

}

}

FILE \*timeTest;

clock\_t begin, end;

if (argc == 4) {

timeTest = fopen("../benchmark/outp1", "a");

begin = clock();

}

int rowsForThread = (n - 1) / cntOfThreads;

int rowsMod = (n - 1) % cntOfThreads;

for (int i = 0; i < cntOfThreads; i++) {

threadArgs \*args = malloc(sizeof(threadArgs));

args->n = n;

if (i == cntOfThreads - 1) {

args->rows = rowsForThread + rowsMod;

} else {

args->rows = rowsForThread;

}

args->i = i;

if (pthread\_create(threads + i, NULL, routine, args) != 0) {

printf("Thread creation error\n");

exit(1);

}

}

for (int i = 0; i < cntOfThreads; i++) {

if (pthread\_join(threads[i], NULL) != 0) {

printf("Thread join error\n");

exit(1);

}

}

double res = 1;

for (int i = 0; i < n; i++) {

res \*= matrix[i][i];

}

if (argc == 3) {

printf("%.2lf\n", res);

}

if (argc == 4) {

end = clock();

fprintf(timeTest, "%lf\n", (double)(end - begin) / CLOCKS\_PER\_SEC);

fclose(timeTest);

}

clearMinor(matrix, n);

free(threads);

threads = NULL;

return 0;

}

**main2.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include <unistd.h>

#include <pthread.h>

void print(double \*\*matrix, int n)

{

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

printf("%.2lf ", matrix[i][j]);

}

printf("\n");

}

}

double \*firstRow;

typedef struct

{

double \*\*matrix;

int n;

int cols;

int i;

pthread\_t \*threads;

} threadArgs;

void clearMinor(double \*\*matrix, int n)

{

for (int i = 0; i < n; i++) {

free(matrix[i]);

matrix[i] = NULL;

}

free(matrix);

matrix = NULL;

}

double getMinor(double \*\*matrix, int n)

{

if (n == 1) {

return matrix[0][0];

} else {

double res = 0, sign = 1;

for (int i = 0; i < n; i++) {

double \*\*mr = malloc(sizeof(double \*) \* (n - 1));

for (int k = 0; k < n - 1; k++) {

mr[k] = malloc(sizeof(double) \* (n - 1));

}

for (int j = 1; j < n; j++) {

for (int k = 0; k < n; k++) {

if (k == i) {

continue;

} else if (k < i) {

mr[j - 1][k] = matrix[j][k];

} else {

mr[j - 1][k - 1] = matrix[j][k];

}

}

}

res += sign \* matrix[0][i] \* getMinor(mr, n - 1);

sign \*= (-1);

clearMinor(mr, n - 1);

}

return res;

}

}

void \*routine(void \*args)

{

threadArgs \*arg = (threadArgs \*) args;

double \*\*matrix = arg->matrix;

pthread\_t \*threads = arg->threads;

int n = arg->n, i = arg->i, cols = arg->cols;

for (int j = i; j < i + cols && j < n; j++) {

double \*\*mr = malloc(sizeof(double \*) \* (n - 1));

for (int it1 = 0; it1 < n - 1; it1++) {

mr[it1] = malloc(sizeof(double) \* (n - 1));

}

for (int it1 = 1; it1 < n; it1++) {

for (int it2 = 0; it2 < n; it2++) {

if (it2 == j) {

continue;

} else if (it2 < j) {

mr[it1 - 1][it2] = matrix[it1][it2];

} else {

mr[it1 - 1][it2 - 1] = matrix[it1][it2];

}

}

}

// printf("\nminor of thread %d:\n", i);

// print(mr, n - 1);

firstRow[j] = getMinor(mr, n - 1);

clearMinor(mr, n - 1);

}

free(arg);

return NULL;

}

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

{

const long CORES = sysconf(\_SC\_NPROCESSORS\_ONLN);

if (argc < 3 || argc > 4) {

printf("Syntax error: expected ./\*executable\_file\_name\* Square\_matrix\_dim Number\_of\_threads\n");

printf("or ./\*executable\_file\_name\* Square\_matrix\_dim Number\_of\_threads -t\n");

exit(1);

}

int n = atoi(argv[1]), cntOfThreads = atoi(argv[2]);

if (n <= 1) {

printf("Math error: this is no matrix\n");

exit(1);

}

if (cntOfThreads > CORES) {

printf("Core error: in this device %ld logic cores\n", CORES);

exit(1);

}

firstRow = (double \*) calloc(n, sizeof(double));

pthread\_t \*threads = (pthread\_t \*) calloc(cntOfThreads, sizeof(double));

if (threads == NULL) {

printf("Allocation error: can't allocate array of threads\n");

exit(1);

}

double \*\*matrix = malloc(sizeof(double \*) \* n);

for (int i = 0; i < n; i++) {

matrix[i] = malloc(sizeof(double) \* n);

}

if (matrix == NULL) {

printf("Allocation error: can't allocate exeptet matrix\n");

}

if (argc == 3) {

printf("Enter the square matrix dim of %d:\n", n);

}

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

scanf("%lf", &matrix[i][j]);

}

}

if (cntOfThreads > n) {

printf("Error: Number\_of\_threads must be less or eqipual then Square\_matrix\_dim");

exit(1);

}

FILE \*timeTest;

clock\_t begin, end;

if (argc == 4) {

timeTest = fopen("../benchmark/outp2", "a");

begin = clock();

}

int colsForThread = n / cntOfThreads;

int colsMod = n % cntOfThreads;

for (int i = 0; i < cntOfThreads; i++) {

threadArgs \*args = malloc(sizeof(threadArgs));

args->matrix = matrix;

args->n = n;

if (i == cntOfThreads - 1) {

args->cols = colsForThread + colsMod;

} else {

args->cols = colsForThread;

}

args->i = i;

args->threads = threads;

if (pthread\_create(threads + i, NULL, routine, args) != 0) {

printf("Thread creation error\n");

exit(1);

}

}

for (int i = 0; i < cntOfThreads; i++) {

if (pthread\_join(threads[i], NULL) != 0) {

printf("Thread join error");

exit(1);

}

}

double det = 0;

for (int i = 0; i < n; i++) {

if (i % 2 != 0) {

firstRow[i] \*= -1;

}

det += firstRow[i] \* matrix[0][i];

}

if (argc == 3) {

printf("%.2lf\n", det);

}

if (argc == 4) {

end = clock();

fprintf(timeTest, "%lf\n", (double)(end - begin) / CLOCKS\_PER\_SEC);

fclose(timeTest);

}

clearMinor(matrix, n);

free(firstRow);

firstRow = NULL;

free(threads);

threads = NULL;

return 0;

}

**Демонстрация работы программы**

Ввод в консоль:

ggame@ggame:~/OS/ready/lab3/build$ cat ../benchmark/test.in

1 3 -2 0 0 1 7

2 0 2 5 3 -1 20

0 -3 6 2 -7 0 41

-3 6 -4 -5 0 2 -1

3 15 -4 2 -4 5 5

2 7 -7 -2 0 3 14

1 2 4 7 8 6 30

ggame@ggame:~/OS/ready/lab3/build$ ./main1 < ../benchmark/test.in 7 6

Enter the square matrix dim of 7:

-139440.00

ggame@ggame:~/OS/ready/lab3/build$ ./main2 < ../benchmark/test.in 7 6

Enter the square matrix dim of 7:

-139440.00

Запускаю обе программы считать детерминант матрицы размера 7 с 6 потоками.

Исследование зависимости ускорения и эффективности алгоритма от количества потоков. Я запустил алгоритм Гаусса на матрице 100 на 100 на 12 потоков (у меня 12 ядер в процессоре) и обнаружил, что время от количества потоков только увеличивается.

ggame@ggame:~/OS/ready/lab3/benchmark$ ./run.sh ../build/main1 ./test ./outp1 100 12

0.004184

0.006768

0.007545

0.009072

0.008474

0.010651

0.011957

0.017260

0.017375

0.020626

0.023384

ggame@ggame:~/OS/ready/lab3/benchmark$ ./run.sh ../build/main2 ./test ./outp2 12 12

45.209801

41.322582

45.722933

46.888112

48.256776

53.626621

51.253903

56.305299

63.967253

73.050157

82.098191

ggame@ggame:~/OS/ready/lab3/benchmark$

**Выводы**

Проделав лабораторную работу, я приобрёл практические навыки в управлении потоками в ОС и обеспечил синхронизацию между ними.