Московский авиационный институт
(Национальный исследовательский университет)

Институт: «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Операционные системы»

Лабораторная работа № 3

Студент: Марочкин И.А.
Группа: М8О-206Б-19
Преподаватель: Соколов А.А.
Дата: 24.04.2021
Оценка:

Москва, 2020

Цель работы

Приобретение практических навыков в:

- Управлении потоками в ОС
- Обеспечение синхронизации между потоками

Задание

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.

В отчете привести исследование зависимости ускорения и эффективности алгоритма от входящих данных и количества потоков. Получившиеся результаты необходимо объяснить.

Вариант:

Задается радиус окружности. Необходимо с помощью метода Монте-Карло рассчитать ее площадь.

Общие сведения о программе

Программа компилируется из файла laba.c. Для реализации поставленной задачи в программе используются следующие системные вызовы:

pthread_mutex_init - инициализация mutex.

pthread_mutex_destroy **-** уничтожение mutex.

pthread_mutex_lock - блокировка mutex.

pthread_mutex_unlock - разблокировка mutex.

pthread_create - создает поток.

pthread_join - блокирует вызывающий поток, пока указанный поток не завершится.

Листинг программы

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>

// Глобальные переменные:
long long bulls_eye = 0; // сумма попаданий в область окружности во всех потоках
pthread_mutex_t mutex;

// структура данных для потоковой функции:
typedef struct {
    long long tries_count;
    int rad;
    long long circle_points;
    int idx;
} pthread_Data;

// потоковая функция:
void* ThreadFunction(void* thread_data){
    srand(time(NULL));
    pthread_Data* data = (pthread_Data*) thread_data;
//    printf("Running thread number %d. Count of tries = %lld Wait...\n", data->idx + 1,
data->tries_count);
    double x, y;
    for (int i = 0; i < data->tries_count; ++i){
        x = (double)rand() / (double)RAND_MAX * (2*data->rad) - data->rad;
        y = (double)rand() / (double)RAND_MAX * (2*data->rad) - data->rad;
        if ((x*x + y*y) <= data->rad*data->rad){
            ++data->circle_points;
        }
    }
    pthread_mutex_lock(&mutex);
    bulls_eye += data->circle_points;
    pthread_mutex_unlock(&mutex);
    return NULL;
}

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

    if (argc != 3){
        perror("ERROR: invalid count of arguments");
        exit(-1);
    }

    long long thr_count = atoll(argv[1]);
    if (thr_count < 1){
        perror("ERROR: invalid first argument");
        exit(-2);
    }

    long long all_tries = atoll(argv[2]);
    if (all_tries < 1){
        perror("ERROR: invalid second argument");
        exit(-3);
    }

    printf("Enter radius: ");
    int rad;
    scanf("%d", &rad);
    if (rad < 1){
        perror("ERROR: invalid radius");
        exit(-4);
    }
```

```
    // выделяем память для массивов идентификаторов потоков и структур данных потоков:
    pthread_t* thread_id = (pthread_t*) malloc(thr_count * sizeof(pthread_t));
    pthread_Data* threadData = (pthread_Data*) malloc(thr_count * sizeof(pthread_Data));
    pthread_mutex_init(&mutex, NULL);

    clock_t begin_time = clock(); // запускаем таймер работы программы

    // присваиваем нужные значения данным и запускаем потоки:
    for (int i = 0; i < thr_count; ++i){
        threadData[i].rad = rad;
        threadData[i].idx = i;
        threadData[i].circle_points = 0;
        if (i == thr_count - 1){
            threadData[i].tries_count = all_tries/thr_count + all_tries%thr_count;
        }
        else{
            threadData[i].tries_count = all_tries/thr_count;
        }
        pthread_create(&thread_id[i], NULL, ThreadFunction, &threadData[i]);
    }

    // ждем завершения всех потоков:
    for (int i = 0; i < thr_count; ++i){
        pthread_join(thread_id[i], NULL);
    }

    clock_t end_time = clock(); // останавливаем таймер работы программы

    printf("S = %f\n", bulls_eye/((double)all_tries) * 2*rad * 2*rad); // выводим результат

    free(thread_id);
    free(threadData);
    pthread_mutex_destroy(&mutex);

    printf("Program running time = %f\n\n",
           (double)(end_time - begin_time) / CLOCKS_PER_SEC);
    return 0;
}
```

Исследование зависимости ускорения и эффективности алгоритма от входящих данных и количества потоков

Для исследования нам необходимо ввести некоторые формулы и обозначения:

$S_i = T_1/T_i$, где S_i - ускорение, T_i - время работы алгоритма, i - количество потоков.

$E_i = S_i / i$, где E_i – эффективность, S_i - ускорение, i - количество потоков.

100 млн. чисел:

$S_2 = 1{,}29$ $E_2 = 0{,}645$

$S_4 = 1{,}02$ $E_4 = 0{,}255$

$S_8 = 1{,}24$ $E_8 = 0{,}155$

$S_{16} = 1{,}34$ $E_{16} = 0{,}084$

$S_{32} = 1,34$ $E_{32} = 0,042$
$S_{64} = 1,31$ $E_{64} = 0,02$
На 100 млн. чисел лучшее ускорение показал алгоритм, использующий 32 потока (да, значение совпадает с 16 потоками, но если округлять менее грубо, то будет видно, что 32 потока сработали чуть быстрее). Что интересно, ускорение на 2 потоках больше, чем на 4 и 8. Однако, что касается эффективности… Скажем так, тут результаты не такие воодушевляющие, ведь лучшую эффективность показал алгоритм на 2 потоках, и с увеличением числа потоков эффективность падала, причем очень существенно.

1 млрд. чисел:

$S_2 = 1,25$ $E_2 = 0,627$
$S_4 = 1,16$ $E_4 = 0,3$
$S_8 = 1,23$ $E_8 = 0,15$
$S_{16} = 1,32$ $E_{16} = 0,08$
$S_{32} = 1,33$ $E_{32} = 0,04$
$S_{64} = 1,33$ $E_{64} = 0,02$
На 1 млрд. чисел лучшее ускорение показал алгоритм, использующий 64 потока. Что интересно, ускорение на 2 потоках снова больше, чем на 4 и 8. Эффективность снова не радует глаз, так как уменьшается пропорционально увеличению количества потоков.

Промежуточный вывод:

Для данной задачи нецелесообразно использовать более 2-х потоков. Да, лучшее ускорение показывает использование 32 и 64 потоков, но это ускорение очень мало относительно просадки в эффективности.
Немного пожертвовать эффективностью, чтобы ускорить алгоритм на 25-30% - это лучшее чего мы можем достичь, поэтому выбираем 2 потока.

Примеры работы

```
Vanya:Src ivan$ ./hundred_million 2 100000000
Enter radius: 1
S = 3.123101
Program running time = 6.378160

Vanya:Src ivan$ ./hundred_million 4 100000000
Enter radius: 1
S = 3.118362
Program running time = 8.078655

Vanya:Src ivan$ ./hundred_million 8 100000000
Enter radius: 1
S = 3.116150
Program running time = 6.643213

Vanya:Src ivan$ ./hundred_million 16 100000000
Enter radius: 1
S = 3.115381
Program running time = 6.189479

Vanya:Src ivan$ ./hundred_million 32 100000000
Enter radius: 1
S = 3.115838
Program running time = 6.140058

Vanya:Src ivan$ ./hundred_million 64 100000000
Enter radius: 1
S = 3.115826
Program running time = 6.294551
```

```
Vanya:Src ivan$ ./billion 2 1000000000
Enter radius: 1
S = 3.123456
Program running time = 63.387970

Vanya:Src ivan$ ./billion 4 1000000000
Enter radius: 1
S = 3.116538
Program running time = 68.368311

Vanya:Src ivan$ ./billion 8 1000000000
Enter radius: 1
S = 3.116246
Program running time = 64.611915

Vanya:Src ivan$ ./billion 16 1000000000
Enter radius: 1
S = 3.114893
Program running time = 60.312183

Vanya:Src ivan$ ./billion 32 1000000000
Enter radius: 1
S = 3.115006
Program running time = 59.774754

Vanya:Src ivan$ ./billion 64 1000000000
Enter radius: 1
S = 3.115135
Program running time = 59.426406
```

Вывод

На СИ можно писать программы, использующие распараллеливание. Ускорение и эффективность алгоритма при этом зависит от количества данных,количества потоков и характера задачи. На маленьких данных лучше работают алгоритмы использующие небольшое количество потоков, чем больше данных тем быстрее начинают работать алгоритмы с бОльшим числом потоков, однако улучшение не может наблюдаться бесконечно, поэтому в какой-то момент с увеличением числа потоков ускорение будет увеличиваться ничтожно мало или вовсе уменьшаться, а из-за этого эффективность будет падать. Главный вывод - для каждой задачи и каждого набора входных данных нужно подбирать лучшее кол-во потоков опытным путем.