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

Студент: В. М. Ватулин
Преподаватель: А. А. Соколов
Группа: М8О-206Б-19
Дата: 17.04.2021
Оценка:
Подпись:

Москва, 2021

1 Постановка задачи

Цель работы:

Приобретение практических навыков в:

- Управление потоками в ОС
- Обеспечение синхронизации между потокам

Задание (вариант 6):

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.

Произвести перемножение 2-ух матриц, содержащих комплексные числа

2 Общие сведения о программе

Матрицы хранятся в структуре Matrice. На вход программе подается размер первой матрицы, первая матрица, затем размер второй матрицы, вторая матрица. Умножение происходит по стандартной формуле умножения матриц. Ячейки новой матрицы для вычиления распределяются по потокам равномерно. После ввода двух матриц программа производит умножение и выводит результирующую матрицу.

3 Общий метод и алгоритм решения

Для реализации поставленной задачи необходимо:

1. Изучить принципы работы pthread.
2. Написать структуру для хранения матрицы
3. Написать функцию ввода матрицы
4. Написать функцию вывода матрицы
5. Написать функцию для умножения матриц в одном потоке (для бенчмарка)
6. Написать распределение ячеек новой матрицы по потокам
7. Написать функцию вычисления ячеек новой матрицы в отдельном потоке
8. Написать бенчмарк
9. Написать обработку ошибок
10. Написать тесты

4 Исходный код

lab3.c

```c

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

#define SECOND2NANO 1000000000

typedef struct {
    size_t width;
    size_t height;
    double complex **buff;
} Matrice;

typedef struct {
    Matrice *lhs;
    Matrice *rhs;
    Matrice *result;
    size_t linear_start;
    size_t linear_end;
} _Matrice_params;

void matrice_fill(Matrice *subj) {
    scanf("%zu %zu", &subj->height, &subj->width);
    subj->buff = malloc(subj->height * sizeof(double complex *));
    for (size_t i = 0; i < subj->height; ++i) {
        subj->buff[i] = malloc(subj->width * sizeof(double complex));
        for (size_t j = 0; j < subj->width; ++j) {
            double temp_real;
            double temp_imag;
            scanf("%lf;%lf", &temp_real, &temp_imag);
            subj->buff[i][j] = CMPLX(temp_real, temp_imag);
        }
    }
}

void matrice_print(const Matrice subj) {
    for (size_t i = 0; i < subj.height; ++i) {
        for (size_t j = 0; j < subj.width; ++j) {
            printf("%.2f;%.2f\t", creal(subj.buff[i][j]), cimag(subj.buff[i][j]));
        }
        printf("\n");
    }
}

```

```c
void matrice_free(Matrice subj) {
    for (size_t i = 0; i < subj.height; ++i) {
        free(subj.buff[i]);
    }
    free(subj.buff);
}

size_t _get_2d_x(const Matrice *matrice, size_t linear_coord) {
    return linear_coord % matrice->width;
}

size_t _get_2d_y(const Matrice *matrice, size_t linear_coord) {
    return linear_coord / matrice->width;
}

void *_matrice_indiv_thread(void *params_void) {
    _Matrice_params *params = (_Matrice_params *) params_void;
    for (size_t i = params->linear_start; i < params->linear_end; ++i) {
        size_t x = _get_2d_x(params->result, i);
        size_t y = _get_2d_y(params->result, i);
        params->result->buff[y][x] = CMPLX(0, 0);
        for (size_t j = 0; j < params->lhs->width; ++j) {
            params->result->buff[y][x] += params->lhs->buff[y][j] * params->rhs->buff[j
                ][x];
        }
    }
    return NULL;
}

Matrice matrice_mult_threads(Matrice lhs, Matrice rhs, unsigned int threads_limit) {
    if (lhs.width != rhs.height) {
        printf("inappropriate matrices' sizes\n");
        exit(1);
    }

    Matrice result;
    result.height = lhs.height;
    result.width = rhs.width;
    result.buff = malloc(result.height * sizeof(double complex *));
    for (size_t i = 0; i < result.height; ++i) {
        result.buff[i] = malloc(result.width * sizeof(double complex));
    }

    size_t linear_size = result.height * result.width;
    if (threads_limit > linear_size) {
        threads_limit = linear_size;
    }
    size_t quotient = linear_size / threads_limit;
    size_t remainder = linear_size % threads_limit;
```

```
95      size_t linear_iter = 0;
96      pthread_t threads[threads_limit];
97      _Matrice_params params[threads_limit];
98
99      for (unsigned int i = 0; i < threads_limit; ++i) {
100         params[i].lhs = &lhs;
101         params[i].rhs = &rhs;
102         params[i].result = &result;
103         params[i].linear_start = linear_iter;
104         linear_iter += quotient;
105         if (remainder > 0) {
106             ++linear_iter;
107             --remainder;
108         }
109         params[i].linear_end = linear_iter;
110         if (pthread_create(&threads[i], NULL, _matrice_indiv_thread, &params[i]) != 0)
                {
111             printf("error with thread creating occured\n");
112             exit(EXIT_FAILURE);
113         }
114     }
115     for (unsigned int i = 0; i < threads_limit; ++i) {
116         pthread_join(threads[i], NULL);
117     }
118     return result;
119 }
120
121 Matrice matrice_mult_casual(Matrice lhs, Matrice rhs) {
122     Matrice result;
123     result.height = lhs.height;
124     result.width = rhs.width;
125     result.buff = malloc(result.height * sizeof(double complex *));
126     for (size_t i = 0; i < result.height; ++i) {
127         result.buff[i] = malloc(result.width * sizeof(double complex));
128     }
129
130     for (size_t y = 0; y < result.height; ++y) {
131         for (size_t x = 0; x < result.width; ++x) {
132             result.buff[y][x] = 0;
133             for (size_t i = 0; i < lhs.width; ++i) {
134                 result.buff[y][x] += lhs.buff[y][i] * rhs.buff[i][x];
135             }
136         }
137     }
138
139     return result;
140 }
141
142 int main(int argc, char **argv) {
```

```
143     if (argc != 2 || atoi(argv[1]) == 0) {
144         printf("bad arguments\n");
145         exit(EXIT_FAILURE);
146     }
147     unsigned int threads_limit = atoi(argv[1]);
148     Matrice lhs;
149     matrice_fill(&lhs);
150     Matrice rhs;
151     matrice_fill(&rhs);
152     Matrice result;
153 
154     struct timespec casual_start, casual_end;
155     timespec_get(&casual_start, TIME_UTC);
156     result = matrice_mult_casual(lhs, rhs);
157     timespec_get(&casual_end, TIME_UTC);
158     matrice_free(result);
159     struct timespec thread_start, thread_end;
160     timespec_get(&thread_start, TIME_UTC);
161     result = matrice_mult_threads(lhs, rhs, threads_limit);
162     timespec_get(&thread_end, TIME_UTC);
163 
164     fprintf(stderr, "Casual multiplying: %lf\n", ((casual_end.tv_sec * SECOND2NANO +
            casual_end.tv_nsec) -
165                                                   (casual_start.tv_sec * SECOND2NANO +
                                                      casual_start.tv_nsec)) / (double)
                                                      SECOND2NANO);
166     fprintf(stderr, "Threading multiplying: %lf\n", ((thread_end.tv_sec * SECOND2NANO +
             thread_end.tv_nsec) -
167                                                      (thread_start.tv_sec * SECOND2NANO +
                                                         thread_start.tv_nsec)) / (double)
                                                         SECOND2NANO);
168     matrice_print(result);
169 
170     matrice_free(lhs);
171     matrice_free(rhs);
172     matrice_free(result);
173 
174     return 0;
175 }
```

5 Пример работы

Тест 1:

```
eri412@Eri-PC:~/Desktop/study/OS/OSlab3$ cat tests/test1
3 3
1;0 2;0 3;0
4;0 5;0 6;0
7;0 8;0 9;0
3 3
1;0 2;0 3;0
4;0 5;0 6;0
7;0 8;0 9;0

eri412@Eri-PC:~/Desktop/study/OS/OSlab3$ ./main <tests/test1 8
Casual multiplying: 0.000001
Threading multiplying: 0.000231
30.00;0.00 36.00;0.00 42.00;0.00
66.00;0.00 81.00;0.00 96.00;0.00
102.00;0.00 126.00;0.00 150.00;0.00
```

Тест 2:

```
eri412@Eri-PC:~/Desktop/study/OS/OSlab3$ cat tests/test2
5 1
1;0
2;0
3;0
4;0
5;0
1 5
1;0 2;0 3;0 4;0 5;0

eri412@Eri-PC:~/Desktop/study/OS/OSlab3$ ./main <tests/test2 8
Casual multiplying: 0.000001
Threading multiplying: 0.000181
1.00;0.00 2.00;0.00 3.00;0.00 4.00;0.00 5.00;0.00
2.00;0.00 4.00;0.00 6.00;0.00 8.00;0.00 10.00;0.00
```

```
3.00;0.00 6.00;0.00 9.00;0.00 12.00;0.00 15.00;0.00
4.00;0.00 8.00;0.00 12.00;0.00 16.00;0.00 20.00;0.00
5.00;0.00 10.00;0.00 15.00;0.00 20.00;0.00 25.00;0.00
```

Тест 3 представляет из себя две матрицы, размерность первой 100x200, размерность второй 200x300. Так как в отчет не вместить такие большие матрицы, я приведу лишь оценку времени выполнения:

```
Casual multiplying: 0.040350
Threading multiplying: 0.015814
```

Как видно из тестов, на маленьких матрицах программа с одним потоком выполняется быстрее, однако программа с множеством потоков выигрывает в скорости, если матрица большая. Это связано с тем, что создание потоков занимает некоторое время, и задача выполнится выстрее, если решить ее не создавая новые потоки. С увеличением же данных, создание новых потоков уже занимает лишь малую часть времени исполнения, соответственно это становится выгодно. Также стоит учитывать количество создаваемых потоков (в данном случае я взял оптимальное для моей системы число 8). Если это количество будет больше, чем может выполняться в системе параллельно, то программа лишь замедлится из-за того, что ей нужно переключать контекст выполняемого потока.

6 Вывод

В процессе работы над лабораторной я научился основам работы потоками в Си. В процессе работы не возникло потребности использования мьютексов, семафоров и т. п., но пришлось придумать, как распределить задачу по потокам равномерно. Мультипоточность мне кажется более удобной, чем мультипропроцессорность, потому что между потоками проще образовать связь за счет общей памяти процесса.