

# Assignment2 实验报告

58121124 张博彦

2023 年 5 月 21 号

## 1 实验内容

In this assignment you have to write a version of matrix multiply that uses threads to divide up the work necessary to compute the product of two matrices. There are several ways to improve the performance using threads. You need to divide the product in row dimension among multiple threads in the computation.

Program must conform to the following prototype: `my matrix multiply -a a matrix file.txt -b b matrix file.txt -t thread count` where the `-a` and `-b` parameters specify input files containing matrices and thread count is the number of threads to use in your strip decomposition. The input matrix files are text files having the following format. The first line contains two integers: rows columns. Then each line is an element in row major order. Lines that begin with " #" should be considered comments and should be ignored.

Program will need to print out the result of  $A * B$  where  $A$  is contained in the file passed via the `-a` parameter and  $B$  is contained in the file passed via the `-b` parameter. It must print the product in the same format (with comments indicating rows) as the input matrix files: rows and columns on the first line, each element in row-major order on a separate line.

Your solution will also be timed. If you have implemented the threading correctly you should expect to see quite a bit of speed-up when the machine you are using has multiple processors. For example, on some machine you may witness

## 2 实验目的

Using multi-thread for data processing, familiar with the specific principles of multi-thread and code writing. Understand that multithreading is more efficient than single threading

## 3 设计思路和流程图

从文件中读取目标矩阵，创建多线程，分配计算至每个线程，得到最终的相乘矩阵的结果并输入 txt 文本中，同时在计算过程中使用 C 函数计时并输出，最终分别调用 1 个线程（即单线程），2 个线程（多线程）对比两者所用时间的差异。

## 4 主要数据结构及其说明

1. thread\_data: 用于传递线程参数并记录初始矩阵和最终相乘的目标矩阵。
2. matrix\_multiply: 运用线程进行矩阵乘法。

## 5 源程序

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

// 结构体用于传递线程参数
struct thread_data
{
    int thread_id;
    double *matrix_a;
    double *matrix_b;
```

```

    double *result;
    int rows;
    int cols;
    int thread_count;
};

// 矩阵乘法线程函数
void *matrix_multiply(void *arg)
{
    struct thread_data *data = (struct thread_data *)arg;
    int start_row = data->thread_id * (data->rows / data->thread_count);
    int end_row = (data->thread_id + 1) * (data->rows / data->thread_count);

    for (int i = start_row; i < end_row; i++)
    {
        for (int j = 0; j < data->cols; j++)
        {
            double sum = 0.0;
            for (int k = 0; k < data->cols; k++)
            {
                sum += data->matrix_a[i * data->cols + k] *
                    data->matrix_b[k * data->cols + j];
            }
            data->result[i * data->cols + j] = sum;
        }
    }

    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    if (argc != 7)

```

```

{
    printf("Usage: mymatrixmultiply -a <matrix_a_file>
    -b <matrix_b_file> -t <thread_count>\n");
    return 1;
}

char *matrix_a_file = NULL;
char *matrix_b_file = NULL;
int thread_count = 0;

// 解析命令行参数
for (int i = 1; i < argc; i += 2)
{
    if (strcmp(argv[i], "-a") == 0)
    {
        matrix_a_file = argv[i + 1];
    } else if (strcmp(argv[i], "-b") == 0)
    {
        matrix_b_file = argv[i + 1];
    } else if (strcmp(argv[i], "-t") == 0)
    {
        thread_count = atoi(argv[i + 1]);
    }
}

clock_t start, end;

start = clock();

// 打开输入文件并读取矩阵大小
FILE *file_a = fopen(matrix_a_file, "r");
FILE *file_b = fopen(matrix_b_file, "r");
int rows, cols;

```

```

if (file_a == NULL || file_b == NULL)
{
    printf("Error opening matrix file.\n");
    return 1;
}

fscanf(file_a, "%d %d", &rows, &cols);
fscanf(file_b, "%d %d", &rows, &cols);

// 分配内存以存储矩阵数据
double *matrix_a = malloc(rows * cols * sizeof(double));
double *matrix_b = malloc(rows * cols * sizeof(double));
double *result = malloc(rows * cols * sizeof(double));

// 从文件中读取矩阵数据
int current_row = 0;
double value;
char line[100];

while (fgets(line, sizeof(line), file_a) != NULL)
{
    if (line[0] == '#')
    {
        continue; // 忽略注释行
    }
    sscanf(line, "%lf", &value);
    matrix_a[current_row] = value;
    current_row++;
}

current_row = 0;

```

```

while (fgets(line , sizeof(line), file_b) != NULL)
{
    if (line[0] == '#')
    {
        continue; // 忽略注释行
    }
    sscanf(line , "%lf", &value);
    matrix_b[current_row] = value;
    current_row++;
}

// 关闭文件
fclose(file_a);
fclose(file_b);

// 创建线程并进行矩阵乘法计算
pthread_t *threads = malloc(thread_count * sizeof(pthread_t));
struct thread_data *thread_data_array = malloc(thread_count * sizeof(str

for (int i = 0; i < thread_count; i++)
{
    thread_data_array[i].thread_id = i;
    thread_data_array[i].matrix_a = matrix_a;
    thread_data_array[i].matrix_b = matrix_b;
    thread_data_array[i].result = result;
    thread_data_array[i].rows = rows;
    thread_data_array[i].cols = cols;
    thread_data_array[i].thread_count = thread_count;

    pthread_create(&threads[i], NULL, matrix_multiply, (void *)&thread_d

}

// 等待线程完成

```

```

    for (int i = 0; i < thread_count; i++)
    {
        pthread_join(threads[i], NULL);
    }

    end = clock();
    printf("time: %f\n", (double)(end - start) / CLOCKS_PER_SEC);

    FILE *file = fopen("result.txt", "w");
    fprintf(file, "%d %d\n", rows, cols);
    for (int i = 0; i < rows; i++)
    {
        fprintf(file, "# Line%d", i);
        for (int j = 0; j < cols; j++)
        {
            fprintf(file, "%.6f\n ", result[i * cols + j]);
        }
    }
    fclose(file);

    // 释放内存
    free(matrix_a);
    free(matrix_b);
    free(result);
    free(threads);
    free(thread_data_array);

    return 0;
}

```

## 6 程序运行结果及分析

运行结果如下调用 1 个线程运行时间如图 1

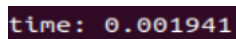
A terminal window with a dark background and light-colored text. The text reads "time: 0.001941".

图 1: 单线程

调用 2 个线程运行时间如图 2

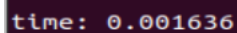
A terminal window with a dark background and light-colored text. The text reads "time: 0.001636".

图 2: 多线程

由于计时是从读取文件开始计时并且计算的两个矩阵大小都为  $50 * 50$ ，所以使用 2 个线程运行的时间并不是使用 1 个线程运行时间的  $1/2$ ，但随着所需计算矩阵的阶数增大，计算时间远大于读取文件时间时，使用  $n$  个线程的时间便会是单线程的  $1/n$  倍。

## 7 实验体会

通过这次实验，我对线程有了更进一步的了解，学会了使用 C 语言编写多线程控制计算的方法，同时通过实验结果了解到了相较于单线程，多线程在相关数据计算方面的效率极高。同时对与读写文件以及相关文件处理方式也有了更加深刻的认识。