中山大学计算机院本科生实验报告

（2023 学年秋季学期）

课程名称：高性能计算程序设计　　**批改人：**

实验	**通用矩阵乘法**	专业（方向）	**信息与计算科学**
学号	21311359	姓名	**何凯迪**
Email	hekd@mail2.sysu.edu.cn	完成日期	2023/8/28

1. 实验目的

（1）通过 OpenMP 实现通用矩阵乘法

（2）基于 OpenMP 的通用矩阵乘法优化

（3）构造基于 Pthreads 的并行 for 循环分解、分配和执行机制

①基于 pthreads 的多线程库提供的基本函数，如线程创建、线程 join、线程同步等。构建 parallel_for 函数对循环分解、分配和执行机制，函数参数包括但不限于(int start, int end, int increment, void *(*functor)(void*), void *arg , int num_threads)；其中 start 为循环开始索引；end 为结束索引；increment 每次循环增加索引数；functor 为函数指针，指向的需要被并行执行循环程序块；arg 为 functor 的入口参数；num_threads 为并行线程数。

②在 Linux 系统中将 parallel_for 函数编译为.so 文件，由其他程序调用。

③将通用矩阵乘法的 for 循环，改造成基于 parallel_for 函数并行化的矩阵乘法，注意只改造**可被并行执行的 for 循环**（例如无 race condition、无数据依赖、无循环依赖等）。

由于通用矩阵乘法的核心部分与之前实验一致，且正确性也已经过验证，故本次报告中不再重复说明正确性，但经检验为正确的。

2. 实验过程和核心代码

（1）通过 OpenMP 实现通用矩阵乘法

```
// 矩阵乘法
Matrix matrixMultiply(const Matrix* A, const Matrix* B) {
        if (A->cols != B->rows) {
                fprintf(stderr, "Matrix dimensions are incompatible for multiplication\n");
                exit(1);
        }

        Matrix C;
        C.rows = A->rows;
        C.cols = B->cols;
        C.data = (float**)malloc(C.rows * sizeof(float*));

        #pragma omp parallel for
        for (int i = 0; i < C.rows; i++) {
                C.data[i] = (float*)calloc(C.cols, sizeof(float));
                for (int j = 0; j < C.cols; j++) {
                        for (int k = 0; k < A->cols; k++) {
                                C.data[i][j] += A->data[i][k] * B->data[k][j];
                        }
                }
        }

        return C;
}
```

#pragma omp parallel for 语句将矩阵乘法的 for 循环并行化，使其在多个线程上执行循环迭代。

（2）基于 OpenMP 的通用矩阵乘法优化

默认调度：

```
// 矩阵乘法
Matrix matrixMultiply(const Matrix* A, const Matrix* B) {
        if (A->cols != B->rows) {
                fprintf(stderr, "Matrix dimensions are incompatible for multiplication\n");
                exit(1);
        }

        Matrix C;
        C.rows = A->rows;
        C.cols = B->cols;
        C.data = (float**)malloc(C.rows * sizeof(float*));

        #pragma omp parallel for
        for (int i = 0; i < C.rows; i++) {
                C.data[i] = (float*)calloc(C.cols, sizeof(float));
                for (int j = 0; j < C.cols; j++) {
                        for (int k = 0; k < A->cols; k++) {
                                C.data[i][j] += A->data[i][k] * B->data[k][j];
                        }
                }
        }

        return C;
}
```

静态调度：

```
//static
Matrix matrixMultiply_sta(const Matrix* A, const Matrix* B) {
        if (A->cols != B->rows) {
                fprintf(stderr, "Matrix dimensions are incompatible for multiplication\n");
                exit(1);
        }

        Matrix C;
        C.rows = A->rows;
        C.cols = B->cols;
        C.data = (float**)malloc(C.rows * sizeof(float*));

        #pragma omp parallel for schedule(static, 1)
        for (int i = 0; i < C.rows; i++) {
                C.data[i] = (float*)calloc(C.cols, sizeof(float));
                for (int j = 0; j < C.cols; j++) {
                        for (int k = 0; k < A->cols; k++) {
                                C.data[i][j] += A->data[i][k] * B->data[k][j];
                        }
                }
        }

        return C;
}
```

动态调度：

```
//dynamic
Matrix matrixMultiply_dyn(const Matrix* A, const Matrix* B) {
        if (A->cols != B->rows) {
                fprintf(stderr, "Matrix dimensions are incompatible for multiplication\n");
                exit(1);
        }

        Matrix C;
        C.rows = A->rows;
        C.cols = B->cols;
        C.data = (float**)malloc(C.rows * sizeof(float*));

        #pragma omp parallel for schedule(dynamic, 1)
        for (int i = 0; i < C.rows; i++) {
                C.data[i] = (float*)calloc(C.cols, sizeof(float));
                for (int j = 0; j < C.cols; j++) {
                        for (int k = 0; k < A->cols; k++) {
                                C.data[i][j] += A->data[i][k] * B->data[k][j];
                        }
                }
        }

        return C;
}
```

（3）构造基于 Pthreads 的并行 for 循环分解、分配和执行机制

parallel_for.cpp

```
#include"parallel_for.h"

void parallel_for(int start, int end, int increment, void *(*functor)(void *), void *arg, int num_threads){
    pthread_t *threads = (pthread_t *)malloc(num_threads * sizeof(pthread_t));
    for_index *arr = (for_index *)malloc(num_threads * sizeof(for_index));
    int block = (end - start) / num_threads;
    for (int thread = 0; thread < num_threads; thread++){
        arr[thread].args = arg;
        arr[thread].start = start + thread * block;
        arr[thread].end = arr[thread].start + block;
        if (thread == (num_threads - 1))
            arr[thread].end = end;
        arr[thread].increment = increment;
        pthread_create(&threads[thread], NULL, functor, (void *)(arr+ thread));
    }
    for (int thread = 0; thread < num_threads; thread++)
        pthread_join(threads[thread], NULL);
    free(threads);
    free(arr);
}
```

pthread_t *threads=(pthread_t*)malloc(num_threads* sizeof(pthread_t)分配一个包含 num_threads 个线程标识符的数组，用于存储线程的信息。

for_index *arr = (for_index *)malloc(num_threads * sizeof(for_index))分配了一个包含 num_threads 个 for_index 结构体的数组，用于存储每个线程的任务参数。

int block = (end - start) / num_threads 用于计算每个线程负责处理的循环块大小。

for (int thread = 0; thread < num_threads; thread++)用于创建并启动多个线程执行任务。

arr[thread].args = arg 将任务参数 arg 存储在 arr 结构体中的 args 字段中，以便任务函数使用。

arr[thread].start = start + thread * block 计算当前线程负责处理的循环块的起始索引。

arr[thread].end = arr[thread].start + block 计算当前线程负责处理的循环块的结束索引。

if (thread == (num_threads - 1)) arr[thread].end = end 确保结束索引等于循环的最终结束索引 end。

arr[thread].increment = increment 将循环增量存储在 arr 结构体中的 increment 字段中。

pthread_create(&threads[thread], NULL, functor, (void *)(arr + thread))创建一个新线程，指定要执行的任务函数为 functor，并传递 arr 结构体的地址作为参数。

for (int thread = 0; thread < num_threads; thread++)用于等待每个线程完成它们的任务。

pthread_join(threads[thread], NULL)等待线程 thread 完成，然后继续执行。

free(threads)和 free(arr)释放分配的内存，以避免内存泄漏。

parallel_for.h

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

struct for_index{
    int start;
    int end;
    int increment;
    void *args;
};

void parallel_for(int start, int end, int increment, void *(*functor)(void *), void *arg, int num_threads);
```

按照实验要求进行相关定义。

example.c

```
struct args{
    int *A, *B, *C;
    int *m, *n, *k;
    args(int *tA, int *tB, int *tC, int *tm, int *tn, int *tk){
        A = tA;
        B = tB;
        C = tC;
        m = tm;
        n = tn;
        k = tk;
    }
};
```

struct args 用于传递矩阵乘法所需的参数，包括矩阵 A、B、C 的指针，以及它们的维度 m、n、k。

```
// 矩阵乘法
void* matrixMultiply(void *arg) {
        struct for_index *index = (struct for_index *)arg;
        struct args *true_arg = (struct args *)(index->args);
        for (int i = index->start; i < index->end; i = i + index->increment){
            for (int j = 0; j < *true_arg->k; ++j){
            int temp = 0;
            for (int z = 0; z < *true_arg->n; ++z)
                temp += true_arg->A[i * (*true_arg->n) + z] * true_arg->B[z * (*true_arg->k) + j];
            true_arg->C[i * (*true_arg->k) + j] = temp;
            }
        }

}
```

struct for_index *index = (struct for_index *)arg 从 arg 中获 for_index 结构体，该结构体包含了计算的索引信息，用于确定该线程负责计算哪一部分的矩阵。

struct args *true_arg = (struct args *)(index->args)从 for_index 中获取 args，其中包含了矩阵 A、B、C 的指针，以及它们的维度信息。

for (int i = index->start; i < index->end; i = i + index->increment)是外部循环，它迭代矩阵的行，根据 start、end 和 increment 确定了每个线程处理的行范围。

for (int j = 0; j < *true_arg->k; ++j)是矩阵乘法中的内部循环，迭代矩阵的列。

int temp = 0 初始化一个临时变量 temp，用于累加矩阵元素的乘积。

for (int z = 0; z < *true_arg->n; ++z)是另一个内部循环，用于迭代矩阵中的元素，计算对应元素的乘积。

temp += true_arg->A[i * (*true_arg->n) + z] * true_arg->B[z * (*true_arg->k) + j]计算两个矩阵中对应元素的乘积，并将结果累加到 temp 中。

true_arg->C[i * (*true_arg->k) + j] = temp 将计算的结果存储在结果矩阵 C 的适当位置。

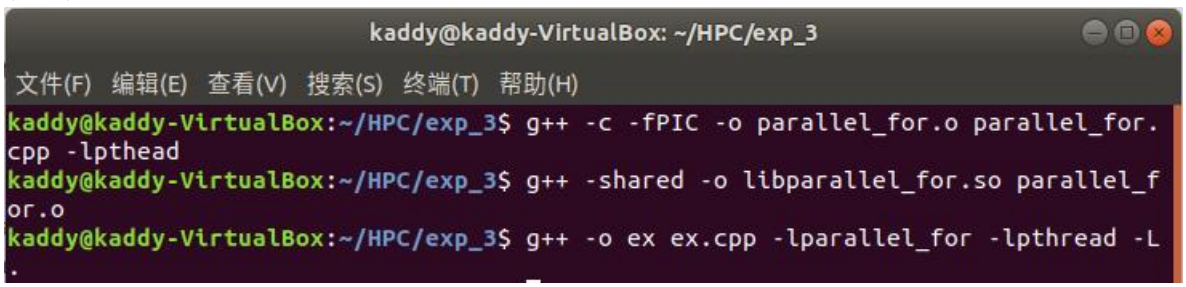
```
struct args *arg = new args(A, B, C, &M, &N, &K);
 start=clock();
 parallel_for(0, M, 1, matrixMultiply, arg, thread_count);
 end=clock();
 double endtime=(double)(end-start)/CLOCKS_PER_SEC;

printf("用时: %f ms.\n",endtime*1000);
```

创建 args 结构体指针 arg。args 包含了矩阵 A、B、C 的指针，以及它们的维度信息。

调用 parallel_for 函数。

编译运行：



生成了.so 文件 

并由 ex.cpp 调用。

3. 实验结果

（1）通过 OpenMP 实现通用矩阵乘法

128 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./matrix_mult 128 128 128 0
N=128, M=128, K=128
Matrix multiplication time: 0.001477 seconds

Matrix A:
0.625460 0.674859 ...
0.705981 0.660830 ...
...

Matrix B:
0.973198 0.202097 ...
0.354381 0.601607 ...
...

Matrix C (Result of A * B):
31.257132 32.437092 ...
34.264095 34.717716 ...
...
```

256 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./matrix_mult 256 256 256 0
N=256, M=256, K=256
Matrix multiplication time: 0.011160 seconds

Matrix A:
0.088739 0.505612 ...
0.389553 0.746698 ...
...

Matrix B:
0.474140 0.579326 ...
0.821535 0.989835 ...
...

Matrix C (Result of A * B):
64.033081 67.919037 ...
61.221306 66.772835 ...
...
```

512 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./matrix_mult 512 512 512 0
N=512, M=512, K=512
Matrix multiplication time: 0.100980 seconds

Matrix A:
0.977208 0.480031 ...
0.285582 0.732535 ...
...

Matrix B:
0.196138 0.776103 ...
0.560250 0.503254 ...
...

Matrix C (Result of A * B):
123.763153 126.014160 ...
125.781357 128.857864 ...
...
```

1024 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./matrix_mult 1024 1024 1024 0
N=1024, M=1024, K=1024
Matrix multiplication time: 0.746375 seconds

Matrix A:
0.522046 0.939339 ...
0.213397 0.735717 ...
...

Matrix B:
0.133136 0.348745 ...
0.168701 0.882200 ...
...

Matrix C (Result of A * B):
260.200409 260.962097 ...
261.298401 261.283691 ...
...
```

2048 阶：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./matrix_mult 2048 2048 2048 0
N=2048, M=2048, K=2048
Matrix multiplication time: 5.902022 seconds

Matrix A:
0.773325 0.831194 ...
0.272622 0.576293 ...
...

Matrix B:
0.218095 0.182332 ...
0.921665 0.025607 ...
...

Matrix C (Result of A * B):
514.904236 508.691681 ...
509.032257 510.098755 ...
...
```

（2）基于 OpenMP 的通用矩阵乘法优化（以 2048 阶矩阵为例测试）

默认调度：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ gcc -o matrix_mult -fopenmp matrix_mult.cpp
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./matrix_mult 2048 2048 2048 0
N=2048, M=2048, K=2048
Matrix multiplication time: 6.866332 seconds

Matrix A:
0.615391 0.009898 ...
0.166121 0.032017 ...
...

Matrix B:
0.084269 0.603472 ...
0.422122 0.223435 ...
...

Matrix C (Result of A * B):
493.014618 497.059448 ...
496.962402 504.343872 ...
...
```

静态调度：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./matrix_mult 2048 2048 2048 1
N=2048, M=2048, K=2048
Matrix multiplication time: 6.135722 seconds

Matrix A:
0.716046 0.422263 ...
0.220877 0.936669 ...
...

Matrix B:
0.494022 0.087288 ...
0.730271 0.932522 ...
...

Matrix C (Result of A * B):
516.777893 515.674988 ...
523.770447 533.632812 ...
...
```

动态调度：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./matrix_mult 2048 2048 2048 2
N=2048, M=2048, K=2048
Matrix multiplication time: 5.860649 seconds

Matrix A:
0.304141 0.556564 ...
0.933534 0.450853 ...
...

Matrix B:
0.926324 0.257323 ...
0.188801 0.152781 ...
...

Matrix C (Result of A * B):
518.486938 504.861176 ...
516.633240 510.232849 ...
...
```

（3）构造基于 Pthreads 的并行 for 循环分解、分配和执行机制（以 2048 阶矩阵乘法为例）

单线程：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./ex 2048 2048 2048 1
N=2048, M=2048, K=2048
用时: 50168.558000 ms.

Matrix A:
8 5 ...
5 7 ...
...

Matrix B:
8 0 ...
3 0 ...
...

Matrix C (Result of A * B):
42387 41785 ...
43345 41775 ...
...
```

双线程：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./ex 2048 2048 2048 2
N=2048, M=2048, K=2048
用时: 25743.618500 ms.

Matrix A:
3 5 ...
0 1 ...
...

Matrix B:
6 6 ...
4 0 ...
...

Matrix C (Result of A * B):
41844 41968 ...
42425 42381 ...
...
```

四线程：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./ex 2048 2048 2048 4
N=2048, M=2048, K=2048
用时: 13663.719250 ms.

Matrix A:
4 8 ...
0 3 ...
...

Matrix B:
8 0 ...
3 5 ...
...

Matrix C (Result of A * B):
41334 42470 ...
40179 40359 ...
...
```

八线程：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./ex 2048 2048 2048 8
N=2048, M=2048, K=2048
用时: 7495.998625 ms.

Matrix A:
3 5 ...
7 3 ...
...

Matrix B:
2 0 ...
6 8 ...
...

Matrix C (Result of A * B):
41027 41409 ...
41473 41307 ...
...
```

十六线程：

```
kaddy@kaddy-VirtualBox:~/HPC/exp_3$ ./ex 2048 2048 2048 16
N=2048, M=2048, K=2048
用时: 5146.368625 ms.

Matrix A:
3 2 ...
0 0 ...
...

Matrix B:
5 1 ...
3 7 ...
...

Matrix C (Result of A * B):
41637 41610 ...
41787 41217 ...
...
```

4. 实验感想

//可以写写过程中遇到的问题，你是怎么解决的。以及可以写你对此次实验的一些理解……

OpenMP 比 pthread 用起来更方便。第三小问还是蛮有难度的。从本次实验学到了许多。