

课 程 实 验 报 告

课程名称： 并行编程原理与实践

院 系： 计算机科学与技术

姓 名： 沈子旭

专业班级： 计算机科学与技术 202004

学 号： U202015396

指导教师： 金海

报告日期： 2022年 7 月 5 日

**计算机科学与技术学院**

目 录

[1 使用OpenMP进行并行矩阵乘法 1](#_Toc139477150)

[1.1 实验目的与要求 1](#_Toc139477151)

[1.2 算法描述 2](#_Toc139477152)

[1.2.1 基础算法 2](#_Toc139477153)

[1.2.2 增加OpenMP并行循环指令后的算法 2](#_Toc139477154)

[1.3 实验方案 3](#_Toc139477155)

[1.3.1 基础算法实现 3](#_Toc139477156)

[1.3.2 增加OpenMP并行循环指令后的算法实现 4](#_Toc139477157)

[1.4 实验结果与分析 5](#_Toc139477158)

[2 使用Pthreads实现并行文本搜索 6](#_Toc139477159)

[2.1 实验目的与要求 6](#_Toc139477160)

[2.2 算法描述 7](#_Toc139477161)

[2.3 实验方案 7](#_Toc139477162)

[2.4 实验结果与分析 10](#_Toc139477163)

[3 实验小结 12](#_Toc139477164)

# 1 使用OpenMP进行并行矩阵乘法

## 1.1 实验目的与要求

设计一个使用OpenMP的并行程序，用于对两个矩阵进行乘法运算。具体要求如下：

你的程序应该能够接受两个矩阵作为输入，并计算它们的乘积。

使用OpenMP将矩阵乘法操作并行化，以加快计算速度。

考虑如何将矩阵数据进行划分和分配给不同的线程，以实现并行计算。

考虑如何处理并行区域的同步，以避免竞态条件和数据一致性问题。

考虑如何利用OpenMP的并行循环和矩阵计算指令，以进一步提高并行效率。

## 1.2 算法描述

本次实验对educoder中提交的代码进行了优化，并提出了优化方法和优化比。

## 1.2.1 基础算法

简化矩阵乘法的基本原理是，从左矩阵中选择与目标值行相等的一行，从右矩阵中选择与目标值列相等的一列，然后逐个相乘并累加得到目标矩阵的元素。假设左矩阵为T1，右矩阵为T2，相乘后的目标矩阵为T3，则基本公式为：T3[i][j] += T1[i][k] \* T2[k][j]，其中k的取值范围是从1到左矩阵的列数（也是右矩阵的行数）。因此，该算法的步骤为初始化目标矩阵，然后使用三层循环来计算上述公式。

## 1.2.2 增加OpenMP并行循环指令后的算法

在基本算法中，每个计算步骤都是相互独立的，因此可以考虑使用OpenMP的并行循环指令#pragma omp parallel for来优化整个过程。根据您提供的信息，您的机器是具有6个核心和12个线程的CPU。因此，可以通过添加num\_threads(12)参数来指定使用最大线程数为12。此外，为了方便线程调度，可以采用schedule(dynamic, 1)的分配策略，即动态调整，每次分配一个迭代任务。

## 1.3 实验方案

## 1.3.1 基础算法实现

基础算法的实现，比较朴素和简单，即是构造三个循环，将左矩阵中的一行和右矩阵中的一列，先分别相乘再累加

void matrix\_multiply(int A[], int B[], int C[], int dim) {

// ======= Write your code below =======

// 使用OpenMP指令并行化计算过程

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

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

int sum=0;

for(int k=0;k<dim;k++) sum+=A[i\*dim+k]\*B[k\*dim+j];

C[i\*dim+j]=sum;

}

}

// ======= Write your code above =======

}

这个算法的时间复杂度可以通过推算得到。在最内层的循环中，每次迭代需要进行一次乘法和一次加法操作，因此每个循环的时间复杂度为O(1)。结合外层的两个循环，总的时间复杂度为O(n^3)，实际执行时间约为2n^3。

空间复杂度方面，只需要存储结果矩阵T3，因此空间复杂度为O(n^2)。

由于这是最基本的算法，并未进行任何优化，所以加速比为1，即并行化后的执行时间与串行执行时间相同。

## 1.3.2 增加OpenMP并行循环指令后的算法实现

该算法相对于基础算法只需要加上OpenMP的并行循环指令，并根据机器实际情况设定适当的参数即可。

实验环境：VS2019

CPU：Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz

核心与线程数：6核12线程

考虑到相关的参数，故将同时开的最大线程数设定为12，分配策略为动态分配，每1个为一组。

void matrix\_multiply(int A[], int B[], int C[], int dim) {

// ======= Write your code below =======

// 使用OpenMP指令并行化计算过程

#pragma omp parallel for schedule(dynamic,1) num\_threads(12)

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

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

int sum=0;

for(int k=0;k<dim;k++) sum+=A[i\*dim+k]\*B[k\*dim+j];

C[i\*dim+j]=sum;

}

}

// ======= Write your code above =======

}

该算法的时间复杂度可以理解为基本的时间复杂度除以所使用的线程数量。因此，可以将其表示为1/6n^3。整体而言，时间复杂度仍然为O(n^3)。空间复杂度仍然是O(n^2)。

当n较小时，优化效果是显著的。但是当n变得较大时，根据理论推测，优化效果应该在理论上接近十倍。然而，实际测量发现加速比只有约8左右，即只有理论上的80%左右。个人推测这可能是由于线程管理开销以及其他与访存相关的开销所导致的。

## 1.4 实验结果与分析

该实验的样例输入数据中两个矩阵都是2000x2000的大小，C[i][j] = i+ j。因为只有矩阵的量级达到103后才方便对时间进行测量，而1000×1000的矩阵观测效果依然不够明显。

为了对这次的实验进行效率上的分析，首先需要引入time库中的函数来测量单个函数的运行时间，具体代码如下：

clock\_t s\_time, e\_time;

s\_time = clock();

matrix\_multiply(A, B, D, dim);

e\_time = clock();

double l\_times = (double)(e\_time - s\_time) / CLOCKS\_PER\_SEC;

printf("linear time: %f seconds\n", l\_times);

clock\_t start\_time, end\_time;

start\_time = clock();

matrix\_multiply\_ex(A, B, C, dim);

end\_time = clock();

double p\_times = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

printf("parallel time: %f seconds\n", p\_times);

此外，为了验证优化后的准确性问题，还增加了矩阵验证的代码，具体代码如下：

int flag = 0;

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

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

if (C[i \* dim + j] != D[i \* dim + j]) {

flag = 1;

break;

}

}

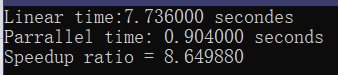
if (flag == 1) break;

}

if (flag == 1) printf("optimize wrong answer!\n");

else printf("speedup ratio = %f", l\_times / p\_times);

对于以上优化和检测，可得最后的优化比为8.65，该次所输入的两个矩阵都是2000x2000的大小，C[i][j] = I + j。其结果如下：



**图1.1增加了矩阵分块优化后的实验结果**

# 2 使用Pthreads实现并行文本搜索

## 2.1 实验目的与要求

设计一个多线程程序，使用Pthreads来实现并行文本搜索。

你需要实现一个函数，该函数接受一个目标字符串和一个包含多个文本文件的文件夹路径。

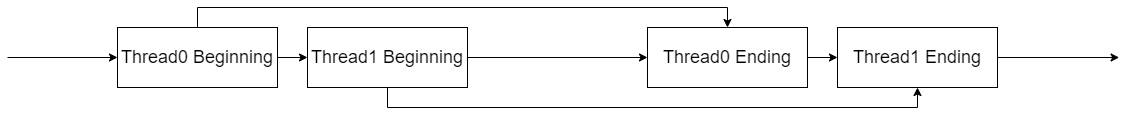
程序应该并行地搜索每个文本文件，查找包含目标字符串的行，并将匹配的行打印出来。

每个线程应该处理一个文件，你需要合理地分配文件给不同的线程。

确保你的程序是线程安全的，并正确处理多个线程之间的同步问题。

## 2.2 算法描述

本次实验需要使用Pthreads来实现并行文本搜索，Pthreads可以看成是一种可以手动编写线程的一种库，其共享内存，是一种分布式内存变成范例。每一个解决问题的Pthreads都离不开两个方法，分别是pthread\_create来创建线程以及pthread\_join来终止线程。其原理示意图如下：



**图2.1 pthread的原理示意图**

本关的具体算法就是在根据输入值定位到相应的文件目录，然后针对目录中的每个文件，创建一个pthread来执行寻找指定文本的操作。最后，等待所有线程完成，并找到所有匹配的文本。

关键问题是确保每个线程输出的结果是线程安全的，以避免可能出现的并发输出错误。为了实现这一点，可以使用锁机制对每个输出过程进行保护。具体而言，对于每个线程的输出操作，可以在开始之前加锁，以确保同一时间只有一个线程能够执行输出操作。输出完成后立即释放锁，以确保其他线程能够继续执行输出操作。这样可以保证整个流程的顺利进行。

## 2.3 实验方案

在完善主题函数逻辑之前，观察代码文件，发现我们需要补充一个threaddata的结构用以存放每一个线程的信息用以输出，考虑到逻辑需求，故给threaddata赋予两个属性：一个是该线程所遍历的文件名，还有一个是所要匹配的文本值，其代码如下：

typedef struct{

const char\* filename;

const char\* target\_string;

} ThreadData;

接着是pthread创建过程中所分配给线程的函数search\_files()的补充，主体逻辑已经给出，我们需要做的就是解析出相关的信息，并根据此创建线程，需要注意的是要对于线程建立失败的情况进行考虑并进行错误处理以及进行线程终结的操作，其具体代码如下：

void\* search\_file(void\* arg) {

ThreadData\* data = (ThreadData\*)arg;

char line[MAX\_LINE\_LENGTH];

FILE\* file = fopen(data->filename, "r");

if (file == NULL) {

perror("Error opening file");

pthread\_exit(NULL);

}

int line\_number = 1;

// ======= Write your code below =======

// 执行文本搜索

while(fgets(line, MAX\_LINE\_LENGTH,file)!=NULL){

if(strstr(line,data->target\_string)!=NULL){

printf("%s:%d: %s",data->filename,line\_number,line);

fflush(stdout);

}

line\_number++;

}

// 文件关闭&线程退出

fclose(file);

pthread\_exit(NULL);

// ======= write your code above =======

}

最后就是分配给每一个线程执行的函数search\_file()了，本次实验需要补充的也就是执行文本搜索的内容，唯一需要注意的就是注意保护线程安全，否则会出现线程冲突导致没有输出。其代码如下：

void search\_files(const char\* folder\_path, const char\* target\_string) {

DIR\* directory;

struct dirent\* entry;

directory = opendir(folder\_path);

if (directory == NULL) {

perror("Error opening directory");

return;

}

// ======= Write your code below =======

pthread\_t threads[256];

ThreadData thread\_data[256];

int num\_threads = 0;

// 执行搜索任务

while ((entry = readdir(directory)) != NULL) {

// 为每个线程分配任务

if(entry->d\_type==DT\_REG){

char filename[256];

snprintf(filename,sizeof(filename),"%s/%s",folder\_path,entry->d\_name);

thread\_data[num\_threads].filename=strdup(filename);

thread\_data[num\_threads].target\_string=target\_string;

if(pthread\_create(&threads[num\_threads],NULL,search\_file,&thread\_data[num\_threads])!=0){

fprintf(stderr, "Error creating thread for file :%s\n",filename);

}

else num\_threads++;

}

}

closedir(directory);

// 线程同步

for(int i=0; i<num\_threads;i++) pthread\_join(threads[i],NULL);

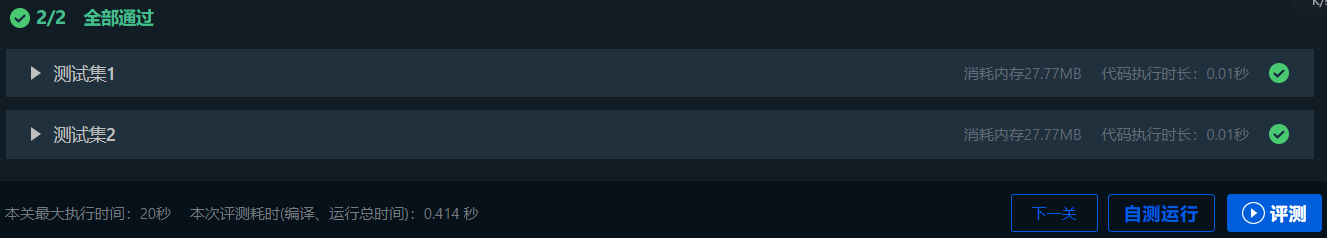
// ======= write your code above =======

}

本次实验所用算法的时间复杂度从理论上来讲应该是O(n)，因为每一个文档都配备了一个单独的线程去进行操作，并且操作的过程也仅是遍历。空间复杂度应该是2n，也是O(n)，因为只需要存储threaddata中的结构即可。

## 2.4 实验结果与分析

本次实验是直接在educoder上进行的，没有实验一一样的优化的空间，唯一需要注意的点就是线程安全性的问题，本次的实验结果已通过educoder的评测，结果如下：



**图2.2 pthread实现并行文本搜索的结果图**

**3 实验小结**

在本次实验中，我学习了两种经典的并行编程范例，分别是OpenMP和Pthread。通过实验的实践，我对这两种编程范例的理解更加深刻，对于将并行编程理论应用于实际问题也有了更深入的了解。

通过学习和探索OpenMP在矩阵乘法上的应用，我对利用并行编程框架对程序进行性能优化有了更深刻的认知。同时，我也进一步学习了OpenMP的标准和应用场景。此外，结合课堂上关于程序的地址连续性对并行化程序性能的影响的讨论，我对这一概念有了更加真切的了解，并希望能在未来的学习和工作中找到适合的场景来实践相关知识。

通过学习Pthread的相关应用，我对其相对灵活地管理线程的特性有了更深入的了解。相关实验也使我更加重视之前常常忽略的线程安全性问题。通过对比第一个实验和第二个实验，我对OpenMP和Pthread的区别和共同点有了更明确的认识，相信在解决相关问题时能够更好地做出最佳选择。

通过这次并行编程实验，我不仅在理论上学习了并行编程的相关知识，还在实践中进行了相应的尝试。通过查阅资料、与同学讨论并结合课堂所学，最终成功完成了实验，并有所收获。选修这门课让我对之前仅仅是会运用的工具和范例有了更深层次的认知。在此，我要感谢金老师、陆老师为我们开设这门课程并设计这次实验，也希望这门课能够越来越好！