**中山大学计算机院本科生实验报告**

**（2024学年春季学期）**

**课程名称：并行程序设计 批改人：**

|  |  |  |  |
| --- | --- | --- | --- |
| **实验** | **5-基于OpenMP的并行矩阵乘法** | **专业（方向）** | **计算机科学与技术** |
| **学号** | **21307174** | **姓名** | **刘俊杰** |
| **Email** | **liujj255@mail2.sysu.edu.cn** | **完成日期** | **2024/4/29** |

# **实验目的**

**1.1 OpenMP通用矩阵乘法**

**使用OpenMP实现并行通用矩阵乘法，并通过实验分析不同进程数量、矩阵规模、调度机制时该实现的性能。**

**输入：三个整数，每个整数的取值范围均为[128, 2048]**

**问题描述：随机生成的矩阵及的矩阵，并对这两个矩阵进行矩阵乘法运算，得到矩阵.**

**输出：三个矩阵，及矩阵计算所消耗的时间。**

**要求：使用OpenMP多线程实现并行矩阵乘法，设置不同线程数量（1-16）、矩阵规模（128-2048）、调度模式（默认、静态、动态调度），通过实验分析程序的并行性能。**

**1.2 构造基于Pthreads的并行for循环分解、分配、执行机制**

**模仿OpenMP的omp\_parallel\_for构造基于Pthreads的并行for循环分解、分配及执行机制。**

**问题描述：生成一个包含parallel\_for函数的动态链接库（.so）文件，该函数创建多个Pthreads线程，并行执行parallel\_for函数的参数所指定的内容。**

**函数参数：parallel\_for函数的参数应当指明被并行循环的索引信息，循环中所需要执行的内容，并行构造等。以下为parallel\_for函数的基础定义，实验实现应包括但不限于以下内容：**

**parallel\_for(int start, int end, int inc,**

**void \*(\*functor)( int,void\*), void \*arg, int num\_threads)**

* **start, end, inc分别为循环的开始、结束及索引自增量；**
* **functor为函数指针，定义了每次循环所执行的内容；**
* **arg为functor的参数指针，给出了functor执行所需的数据；**
* **num\_threads为期望产生的线程数量。**
* **选做：除上述内容外，还可以考虑调度方式等额外参数。**

**示例：给定functor及参数如下：**

**struct functor\_args {**

**float \*A, \*B, \*C;**

**};**

**void \*functor(int idx, void\* args){**

**functor\_args \*args\_data = (functor\_args\*) args;**

**args\_data->C[idx] = args\_data->A[idx] + args\_data->B[idx];**

**}**

**调用方式如下：**

**functor\_args args = {A, B, C};**

**parallel\_for(0, 10, 1, functor, (void\*)&args, 2)**

**该调用方式应当能产生两个线程，并行执行functor完成数组求和（）。当不考虑调度方式时，可由前一个线程执行任务{0,1,2,3,4}，后一个线程执行任务{5,6,7,8,9}。也可以实现对调度方式的定义。**

**要求：完成parallel\_for函数实现并生成动态链接库文件，并以矩阵乘法为例，测试其实现的正确性及效率。**

# **2实验过程和核心代码**

**2.1 OpenMP通用矩阵乘法**

## **2.1.1 openMP调度方式**

**①静态调度static:**

**大部分编译器在没有使用schedule子句的时候，默认是static调度。static在编译的时候就已经确定了，那些循环由哪些线程执行。**

**当不使用size 时，将给每个线程分配┌N/t┐个迭代。当使用size时，将每次给线程分配size次迭代。**

**②动态调度dynamic:**

**动态调度依赖于运行时的状态动态确定线程所执行的迭代，也就是线程执行完已经分配的任务后，会去领取还有的任务。由于线程启动和执行完的时间不确定，所以迭代被分配到哪个线程是无法事先知道的。**

**当不使用size 时，是将迭代逐个地分配到各个线程。当使用size 时，逐个分配size个迭代给各个线程。**

**③启发式调度guided**

**采用启发式调度方法进行调度，每次分配给线程迭代次数不同，开始比较大，以后逐渐减小。**

**size表示每次分配的迭代次数的最小值，由于每次分配的迭代次数会逐渐减少，少到size时，将不再减少。如果不知道size的大小，那么默认size为1，即一直减少到1。具体采用哪一种启发式算法，需要参考具体的编译器和相关手册的信息。**

## **2.1.2 实验过程**

**①首先根据矩阵维度用随机数初始化矩阵A、B和C。**

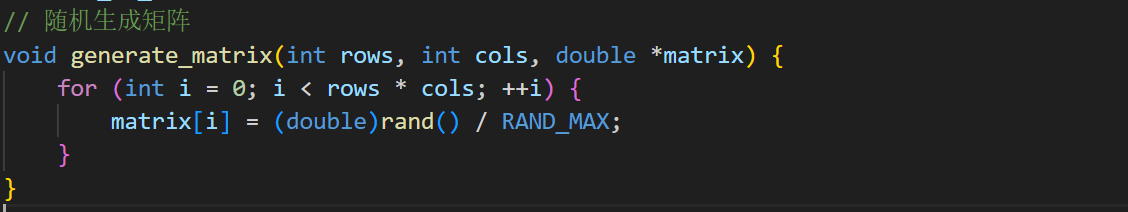
**②使用#pragma omp parallel for schedule(static) num\_threads(1)并行化矩阵乘法(可自选调度方式和线程个数)。**

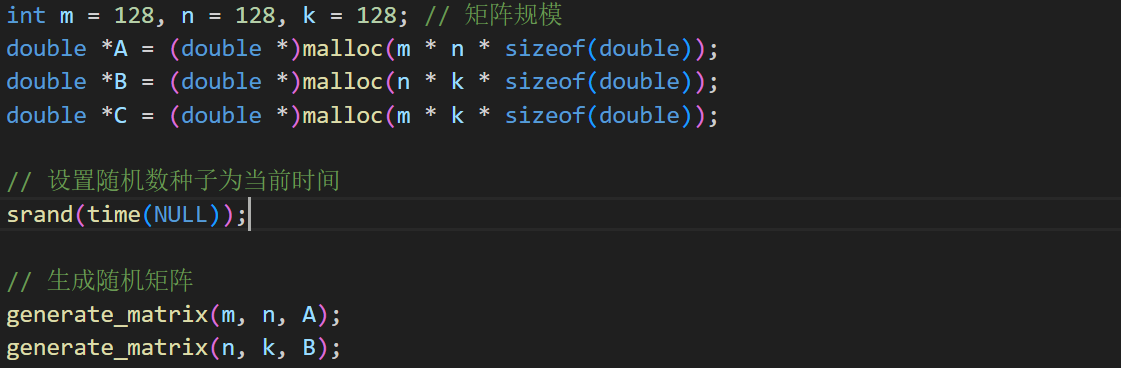
**③注意由于是多进程，访问矩阵C时可能会发生竞争关系，故需要使用#pragma omp critical 声明临界区(其实并不需要,每个进程不会访问C的同一个地方)。**

**④输出矩阵结果和矩阵乘法花费时间。**

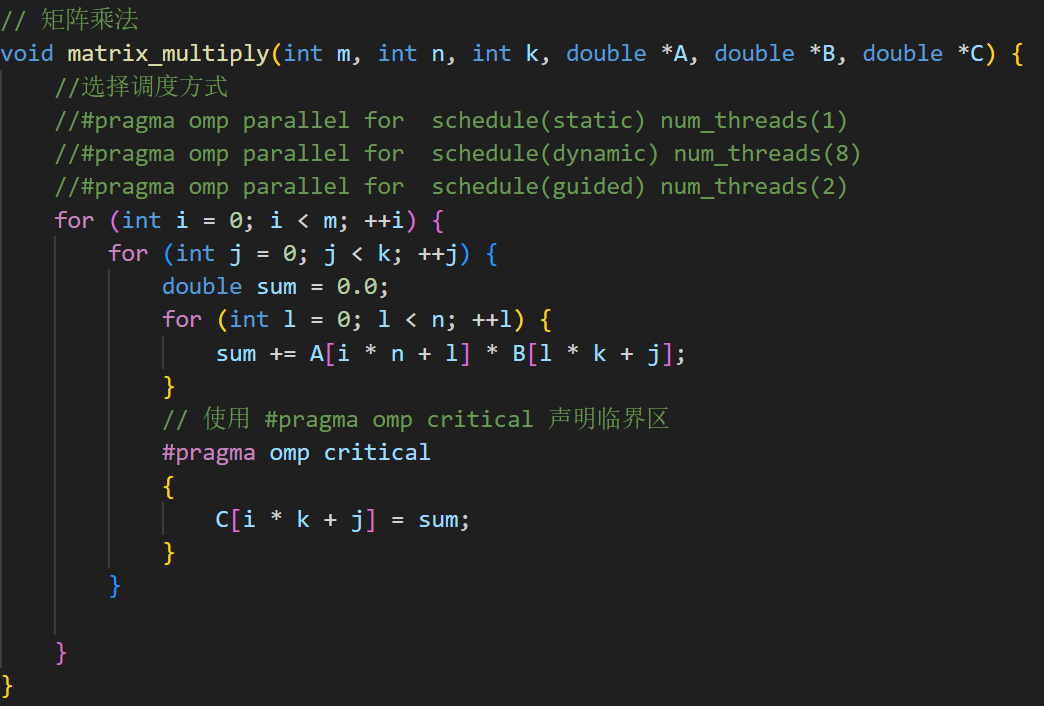
## **2.1.3 核心代码**

**①首先根据矩阵维度用随机数初始化矩阵A、B和C。**

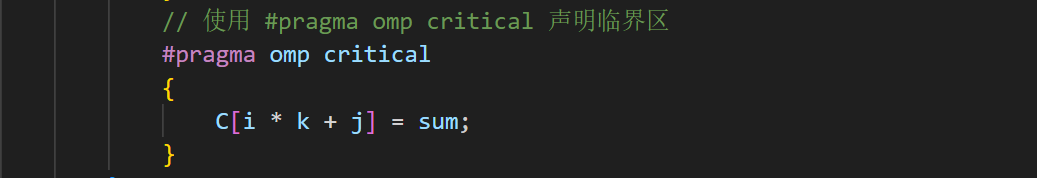
****

****

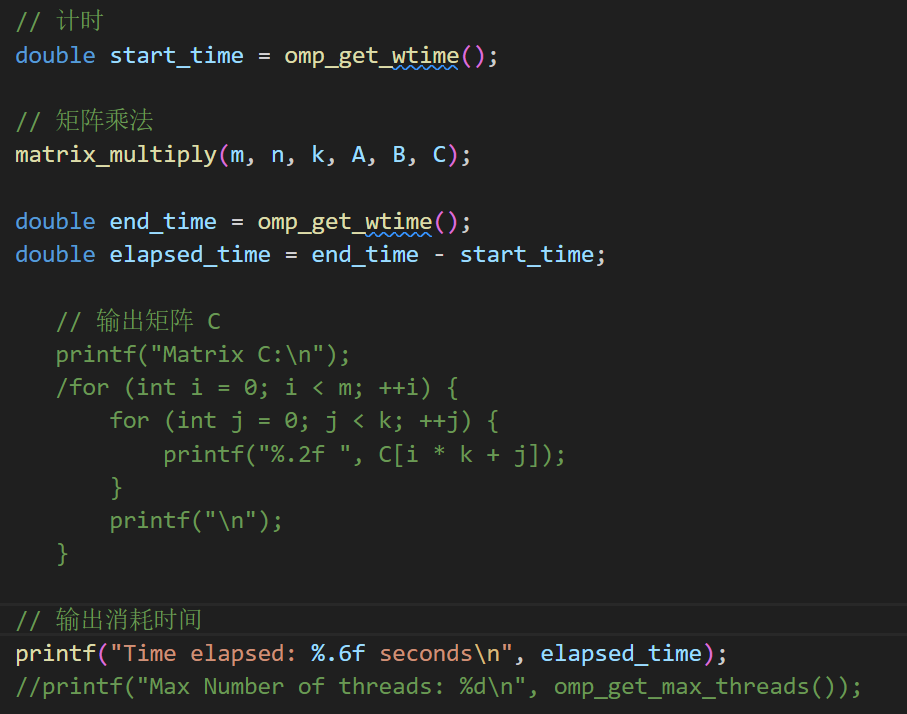
**②使用#pragma omp parallel for schedule(static) num\_threads(1)并行化矩阵乘法(可自选调度方式和线程个数)。**

****

**③注意由于是多进程，访问矩阵C时可能会发生竞争关系，故需要使用#pragma omp critical 声明临界区。**

****

**④输出矩阵结果和矩阵乘法花费时间。**

****

**2.2 构造基于Pthreads的并行for循环分解、分配、执行机制**

**2.2.1 实验思路**

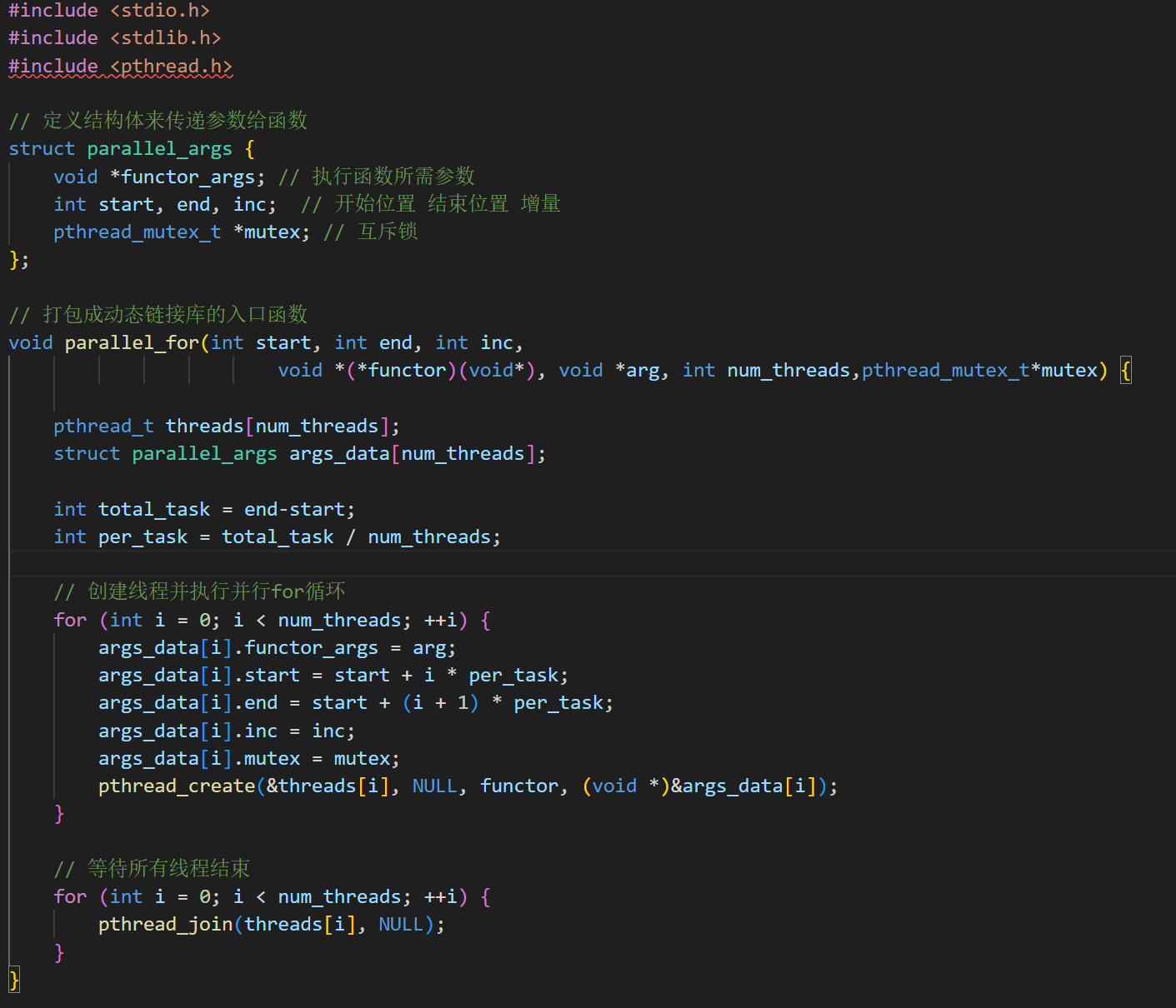
**①将函数参数传入给线程，线程根据矩阵A的维度和线程的个数将矩阵A的行划分给不同的线程计算。**

**②每个线程根据传入参数执行matrix\_multiply函数**

**③为避免竞争使用pthread\_mutex\_t mutex实现互斥(其实并不需要,每个进程不会访问C的同一个地方)。**

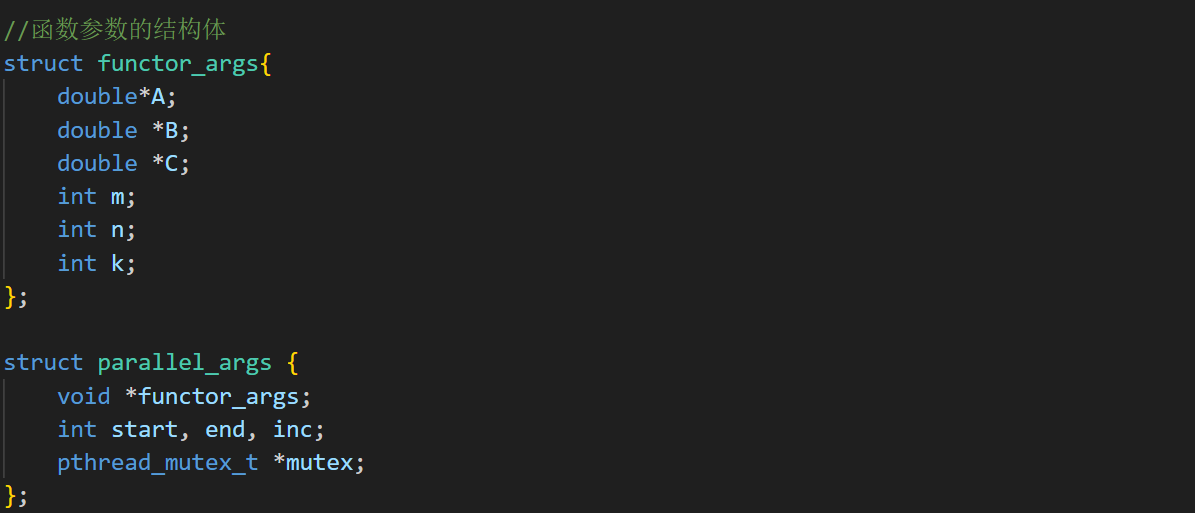
**2.2.2 核心代码**

**2.2.2.1 parallel.c**

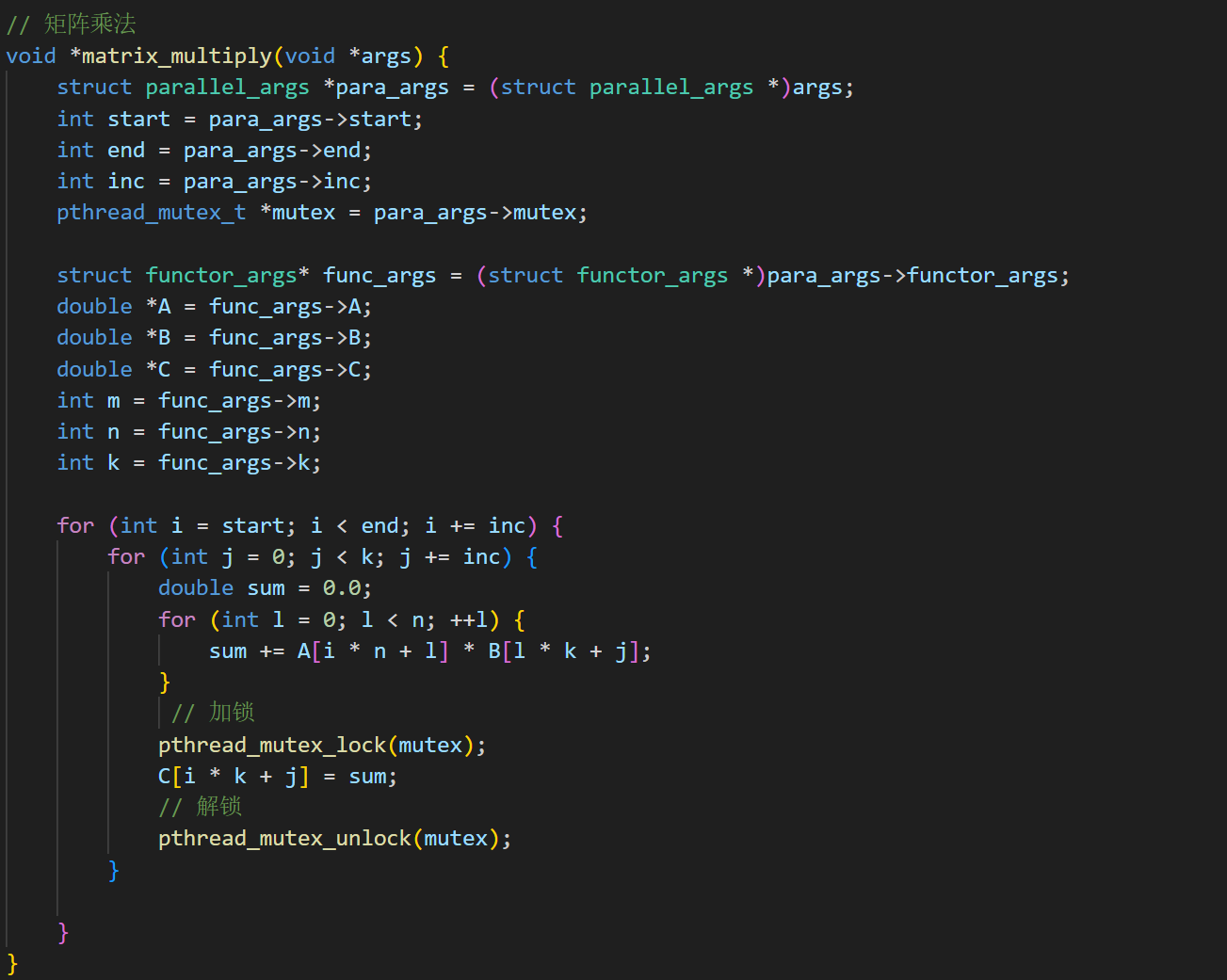
****

**2.2.2.2 main.c**

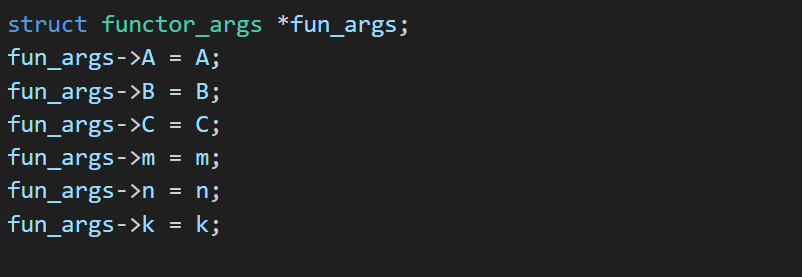
**①函数参数结构体**

****

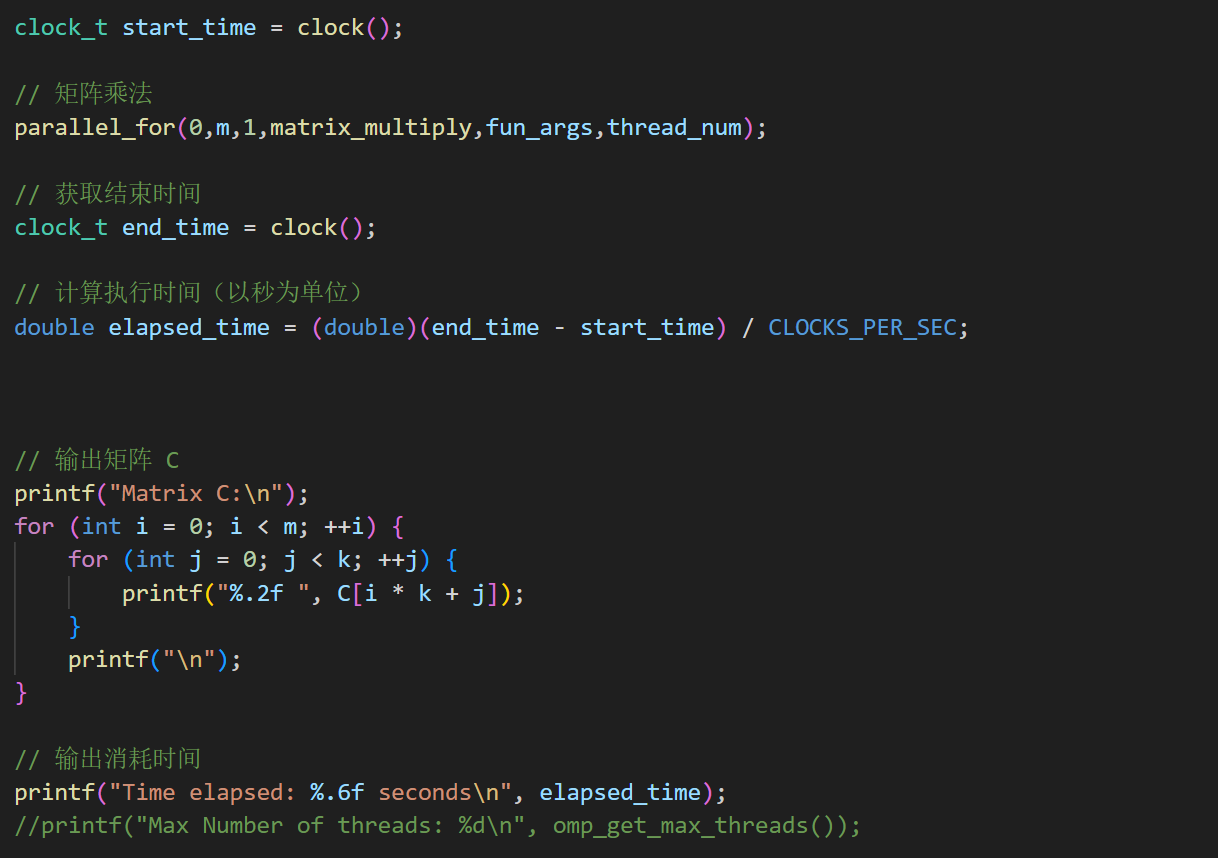
**②线程运行的函数**

****

**③函数参数赋值**

****

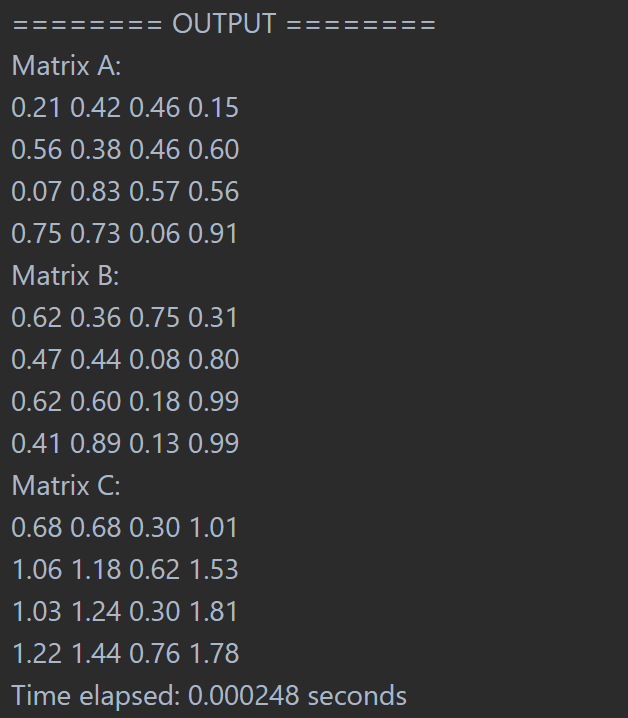
**④多线程运行并输出结果**

****

# **3.实验结果**

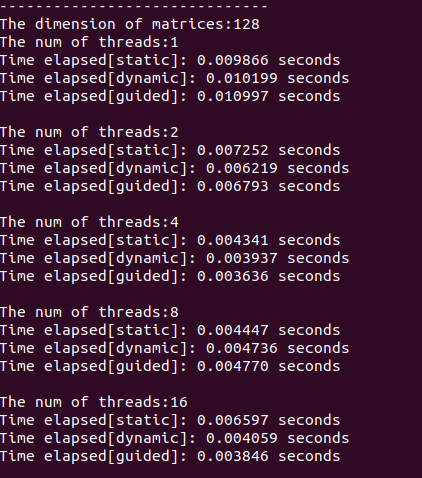
**3.1 OpenMP通用矩阵乘法**

**验证准确性(用4个线程检测4X4矩阵乘法):**

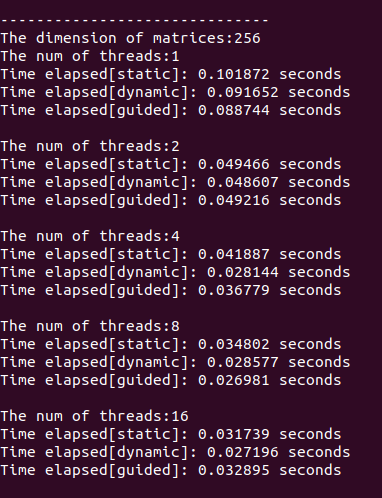
****

**可以看到结果是正确的**

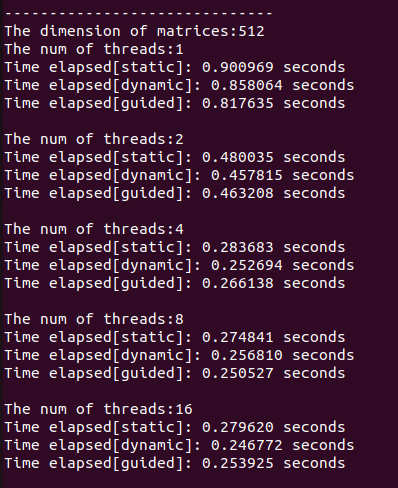
**①矩阵规模为128**

****

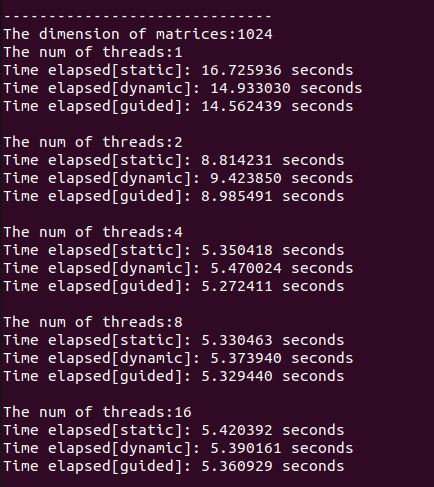
**②矩阵规模为256**

****

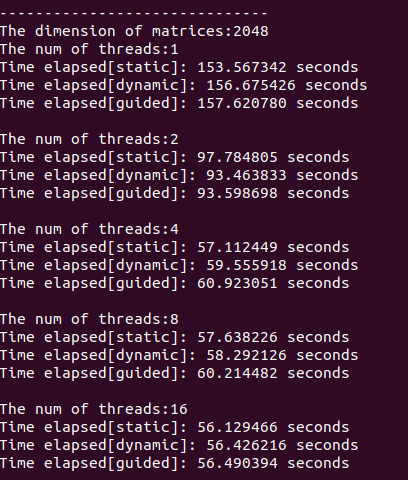
**③矩阵规模为512**

****

**④矩阵规模为1024**

****

**⑤矩阵规模为2048**

****

**⑥综合对比**

****

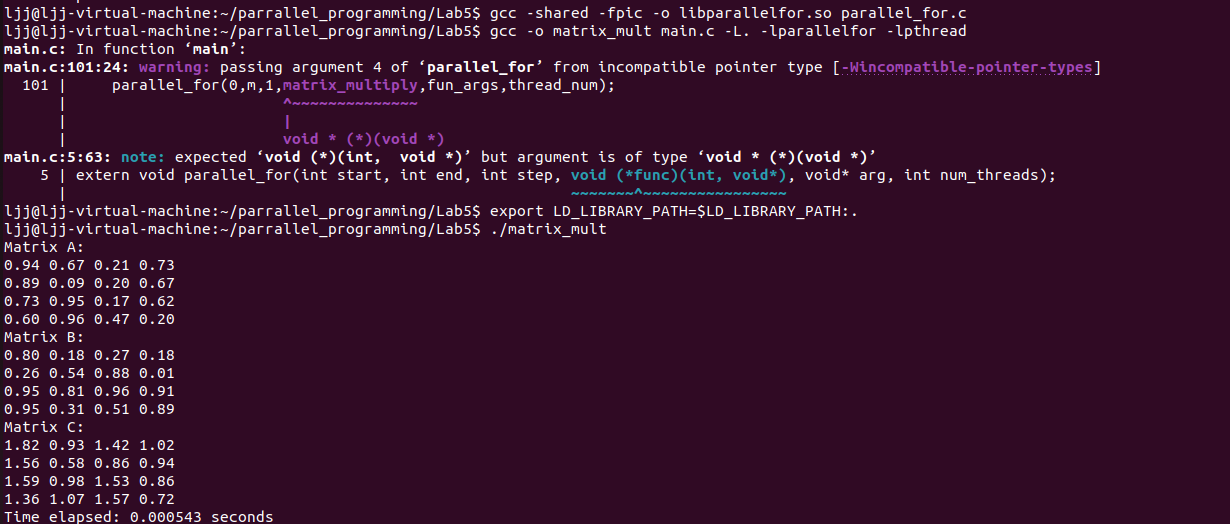
**由于虚拟机资源有限，但线程数目超过4个时线程的加速效果就不明显甚至有所倒退，故主要分析1到4个线程的数据。**

**可以看到随着线程数目的增多，加速效果比较明显，特别是当矩阵规模较大的时候计算时间大大减少了。**

**可以看到openMP的调度方式不同，对计算时间的影响也不同，总体上来看dynamic和guided的调度方式是比static更快的(但也不是绝对的)**

**3.2 构造基于Pthreads的并行for循环分解、分配、执行机制**

**验证计算准确性**

****

**可以看到计算结果是正确的**

**①矩阵规模为128**

****

**②矩阵规模为256**

****

**③矩阵规模为512**

****

**④矩阵规模为1024**

****

**⑤矩阵规模为2048**

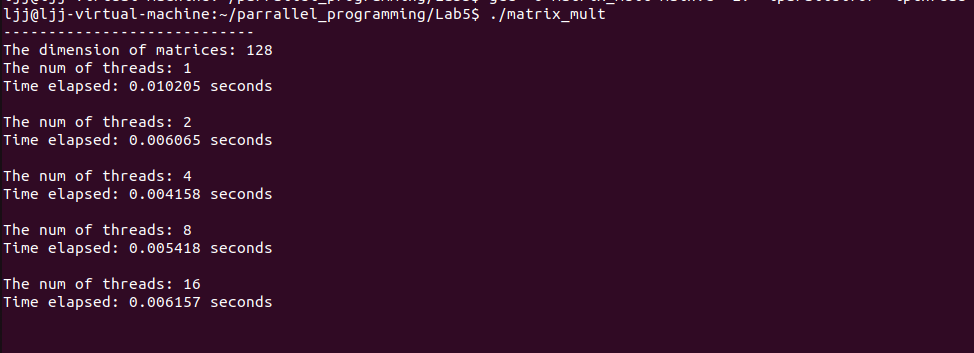
****

**问题：随着线程的增加，运行时间没有明显的改善 即使去掉锁也没有改善**

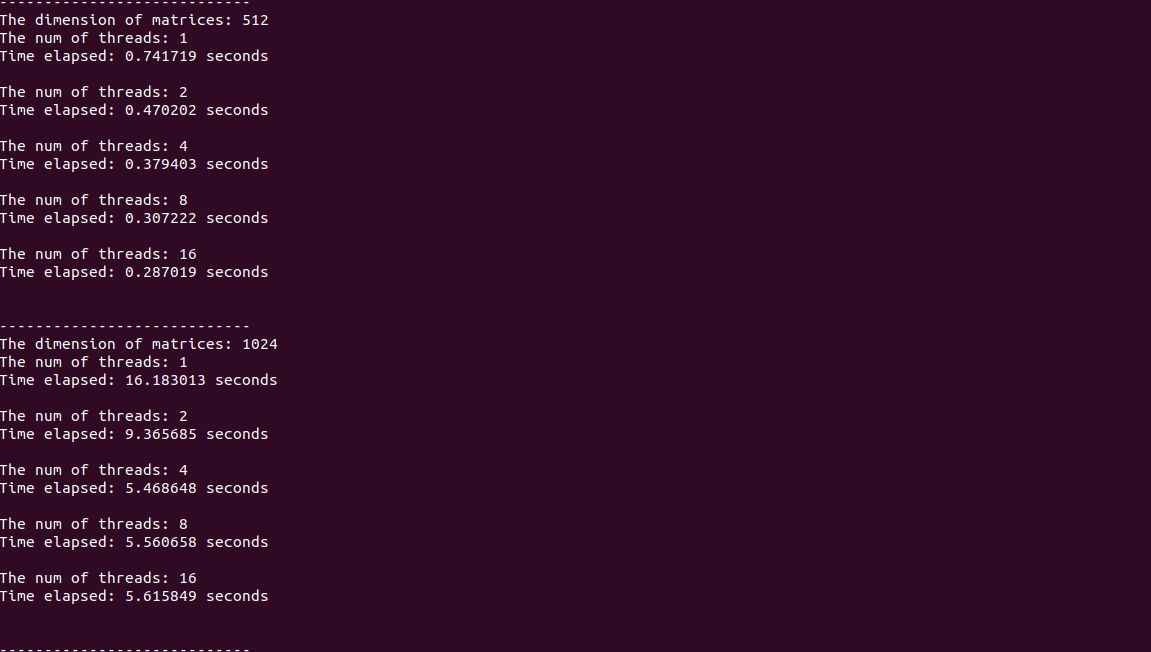
**解决:最后发现是clock计时的原因:**

**clock()函数的功能: 这个函数返回从“开启这个程序进程”到“程序中调用C++ clock()函数”时之间的CPU时钟计时单元（clock tick）数当程序单线程或者单核心机器运行时，这种时间的统计方法是正确的。但是如果要执行的代码多个线程并发执行时就会出问题，因为最终end-begin将会是多个核心总共执行的时钟嘀嗒数，因此造成时间偏大。**

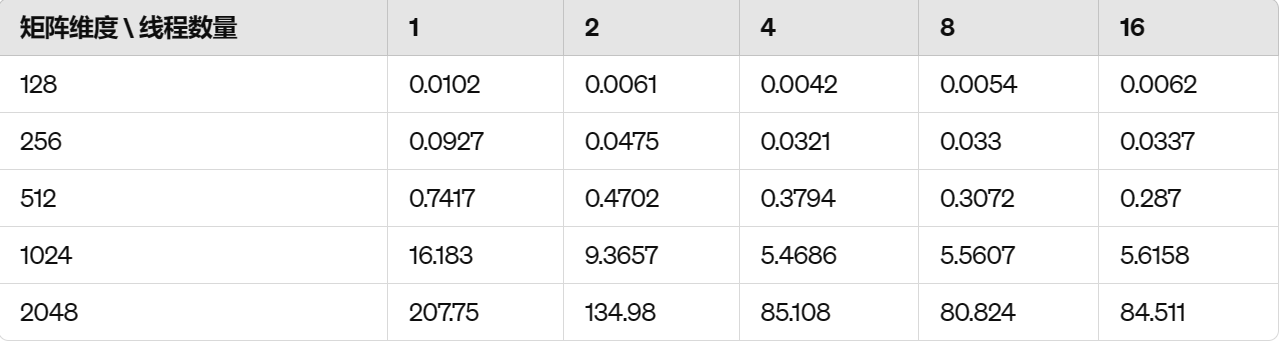
**使用clock\_gettime解决此问题最后实验结果:**











**4.实验感悟：**

**①在完成OpenMP通用矩阵乘法实现的过程中，我对并行计算的性能影响因素有了更深入的了解。增加线程数量可以提高并行计算的速度，但并非线程数量越多越好。在某个点之后，继续增加线程数量可能会带来性能的下降，这是因为线程间的竞争和同步造成的开销开始显现。**

**OpenMP提供了多种调度模式，包括默认、静态和动态等。不同的调度模式适用于不同的并行计算场景。在实验中，我发现对于较小规模的问题，默认的调度模式通常表现较好，而对于大规模问题，静态或动态调度可能会更有效。**

**②在完成基于Pthreads的并行for循环分解、分配、执行机制的实现过程中，我对多线程编程的原理和实践有了更深入的了解。在实现中，我学会了生成一个包含parallel\_for函数的动态链接库（.so）文件，该函数创建多个Pthreads线程，并行执行parallel\_for函数的参数所指定的内容。**

**多线程编程中最容易遇到的问题之一是并发访问共享资源的安全性问题。在实现中，我学会了使用互斥锁（mutex）来保护共享资源，以避免多个线程同时访问导致的数据竞争和不确定行为。**

**clock()函数的功能: 这个函数返回从“开启这个程序进程”到“程序中调用C++ clock()函数”时之间的CPU时钟计时单元（clock tick）数当程序单线程或者单核心机器运行时，这种时间的统计方法是正确的。但是如果要执行的代码多个线程并发执行时就会出问题，因为最终end-begin将会是多个核心总共执行的时钟嘀嗒数，因此造成时间偏大。需要使用clock\_gettime**