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

1. 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：$\boldsymbol{m},\boldsymbol{n},\boldsymbol{k}$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$\boldsymbol{m}\times\boldsymbol{n}$的矩阵$\boldsymbol{A}$及$\boldsymbol{n}\times\boldsymbol{k}$的矩阵$\boldsymbol{B}$，并对这两个矩阵进行矩阵乘法运算，得到矩阵$\boldsymbol{C}$.

输出：$\boldsymbol{A},\boldsymbol{B},\boldsymbol{C}$三个矩阵，及矩阵计算所消耗的时间$\boldsymbol{t}$。

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（$\boldsymbol{C_i = A_i + B_i}$）。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。

```
// 随机生成矩阵
void generate_matrix(int rows, int cols, double *matrix) {
    for (int i = 0; i < rows * cols; ++i) {
        matrix[i] = (double)rand() / RAND_MAX;
    }
}
```

```
int m = 128, n = 128, k = 128; // 矩阵规模
double *A = (double *)malloc(m * n * sizeof(double));
double *B = (double *)malloc(n * k * sizeof(double));
double *C = (double *)malloc(m * k * sizeof(double));

// 设置随机数种子为当前时间
srand(time(NULL));

// 生成随机矩阵
generate_matrix(m, n, A);
generate_matrix(n, k, B);
```

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)。

```
// 矩阵乘法
void matrix_multiply(int m, int n, int k, double *A, double *B, double *C) {
    //选择调度方式
    //#pragma omp parallel for  schedule(static) num_threads(1)
    //#pragma omp parallel for  schedule(dynamic) num_threads(8)
    //#pragma omp parallel for  schedule(guided) num_threads(2)
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < k; ++j) {
            double sum = 0.0;
            for (int l = 0; l < n; ++l) {
                sum += A[i * n + l] * B[l * k + j];
            }
            // 使用 #pragma omp critical 声明临界区
            #pragma omp critical
            {
                C[i * k + j] = sum;
            }
        }

    }
}
```

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。

```
            // 使用 #pragma omp critical 声明临界区
            #pragma omp critical
            {
                C[i * k + j] = sum;
            }
```

④输出矩阵结果和矩阵乘法花费时间。

```
// 计时
double start_time = omp_get_wtime();

// 矩阵乘法
matrix_multiply(m, n, k, A, B, C);

double end_time = omp_get_wtime();
double elapsed_time = end_time - start_time;

   // 输出矩阵 C
   printf("Matrix C:\n");
   /for (int i = 0; i < m; ++i) {
       for (int j = 0; j < k; ++j) {
           printf("%.2f ", C[i * k + j]);
       }
       printf("\n");
   }

// 输出消耗时间
printf("Time elapsed: %.6f seconds\n", elapsed_time);
//printf("Max Number of threads: %d\n", omp_get_max_threads());
```

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// 定义结构体来传递参数给函数
struct parallel_args {
    void *functor_args; // 执行函数所需参数
    int start, end, inc;  // 开始位置 结束位置 增量
    pthread_mutex_t *mutex; // 互斥锁
};

// 打包成动态链接库的入口函数
void parallel_for(int start, int end, int inc,
                  void *(*functor)(void*), void *arg, int num_threads,pthread_mutex_t*mutex) {

    pthread_t threads[num_threads];
    struct parallel_args args_data[num_threads];

    int total_task = end-start;
    int per_task = total_task / num_threads;

    // 创建线程并执行并行for循环
    for (int i = 0; i < num_threads; ++i) {
        args_data[i].functor_args = arg;
        args_data[i].start = start + i * per_task;
        args_data[i].end = start + (i + 1) * per_task;
        args_data[i].inc = inc;
        args_data[i].mutex = mutex;
        pthread_create(&threads[i], NULL, functor, (void *)&args_data[i]);
    }

    // 等待所有线程结束
    for (int i = 0; i < num_threads; ++i) {
        pthread_join(threads[i], NULL);
    }
}
```

2.2.2.2 main.c

①函数参数结构体

```c
//函数参数的结构体
struct functor_args{
    double*A;
    double *B;
    double *C;
    int m;
    int n;
    int k;
};

struct parallel_args {
    void *functor_args;
    int start, end, inc;
    pthread_mutex_t *mutex;
};
```

②线程运行的函数

```
// 矩阵乘法
void *matrix_multiply(void *args) {
    struct parallel_args *para_args = (struct parallel_args *)args;
    int start = para_args->start;
    int end = para_args->end;
    int inc = para_args->inc;
    pthread_mutex_t *mutex = para_args->mutex;

    struct functor_args* func_args = (struct functor_args *)para_args->functor_args;
    double *A = func_args->A;
    double *B = func_args->B;
    double *C = func_args->C;
    int m = func_args->m;
    int n = func_args->n;
    int k = func_args->k;

    for (int i = start; i < end; i += inc) {
        for (int j = 0; j < k; j += inc) {
            double sum = 0.0;
            for (int l = 0; l < n; ++l) {
                sum += A[i * n + l] * B[l * k + j];
            }
            // 加锁
            pthread_mutex_lock(mutex);
            C[i * k + j] = sum;
            // 解锁
            pthread_mutex_unlock(mutex);
        }

    }
}
```

③函数参数赋值

```
struct functor_args *fun_args;
fun_args->A = A;
fun_args->B = B;
fun_args->C = C;
fun_args->m = m;
fun_args->n = n;
fun_args->k = k;
```

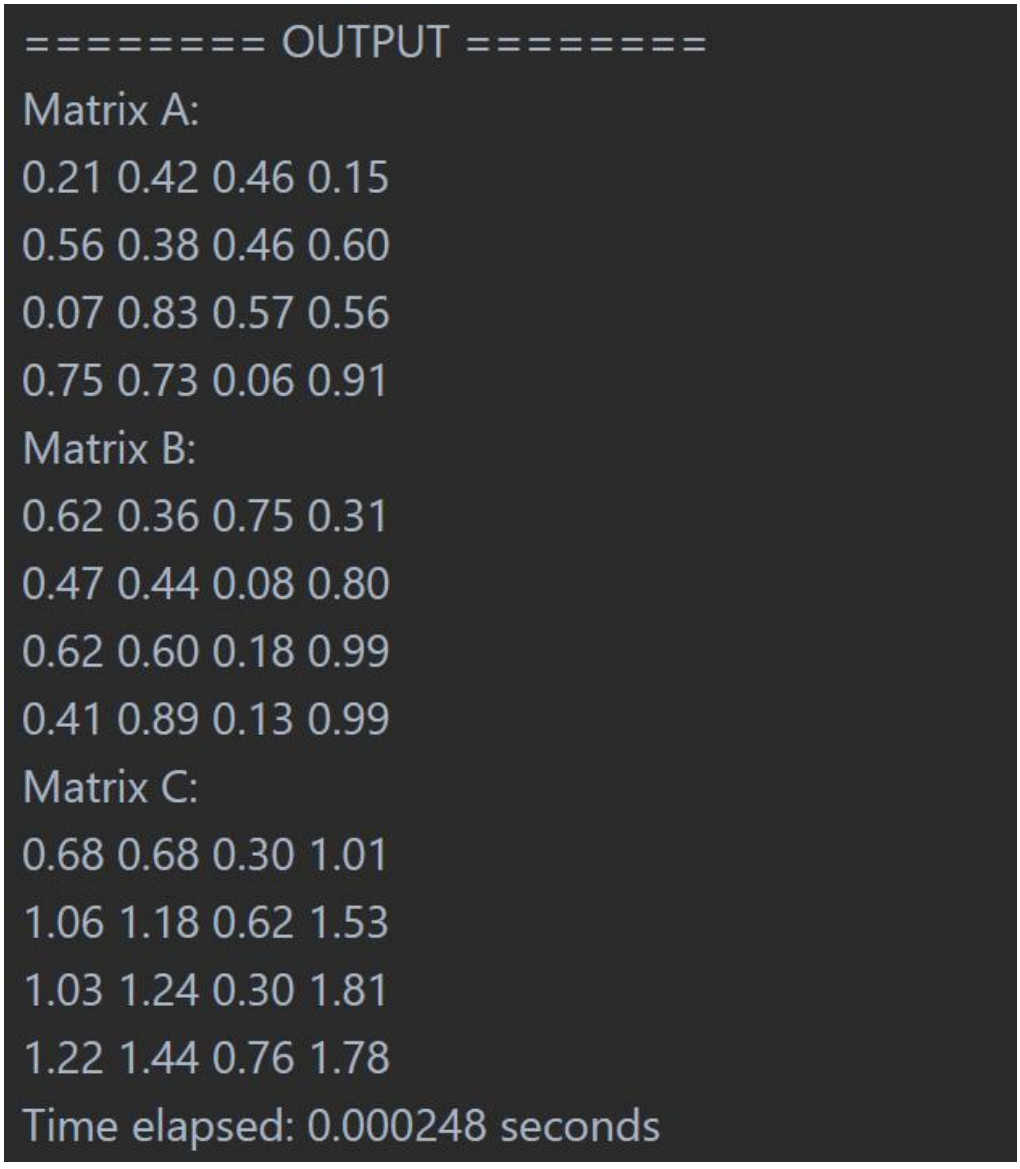
④多线程运行并输出结果

```
clock_t start_time = clock();

// 矩阵乘法
parallel_for(0,m,1,matrix_multiply,fun_args,thread_num);

// 获取结束时间
clock_t end_time = clock();

// 计算执行时间（以秒为单位）
double elapsed_time = (double)(end_time - start_time) / CLOCKS_PER_SEC;



// 输出矩阵 C
printf("Matrix C:\n");
for (int i = 0; i < m; ++i) {
    for (int j = 0; j < k; ++j) {
        printf("%.2f ", C[i * k + j]);
    }
    printf("\n");
}

// 输出消耗时间
printf("Time elapsed: %.6f seconds\n", elapsed_time);
//printf("Max Number of threads: %d\n", omp_get_max_threads());
```

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)：

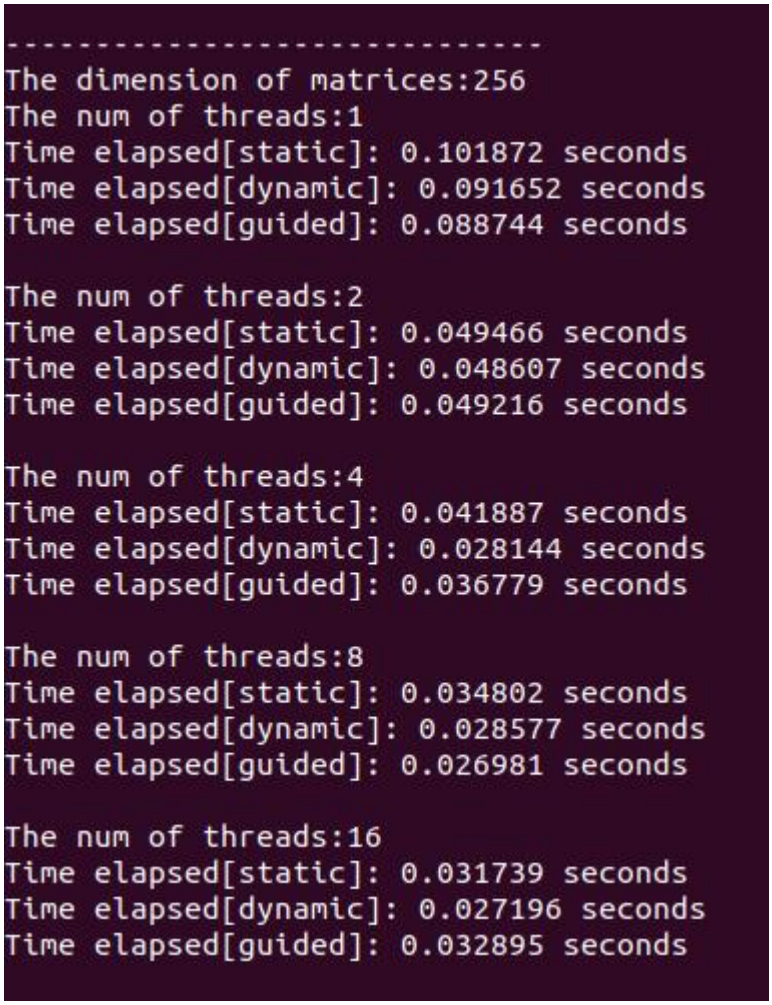


可以看到结果是正确的

①矩阵规模为 128

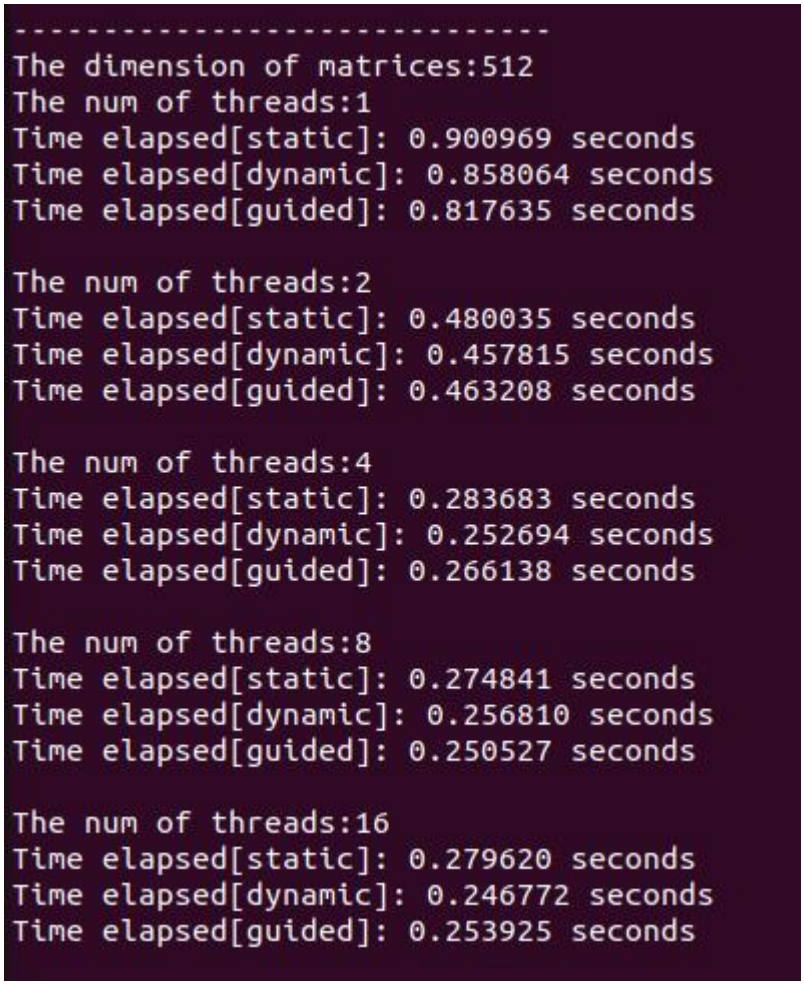
```
------------------------------
The dimension of matrices:128
The num of threads:1
Time elapsed[static]: 0.009866 seconds
Time elapsed[dynamic]: 0.010199 seconds
Time elapsed[guided]: 0.010997 seconds

The num of threads:2
Time elapsed[static]: 0.007252 seconds
Time elapsed[dynamic]: 0.006219 seconds
Time elapsed[guided]: 0.006793 seconds

The num of threads:4
Time elapsed[static]: 0.004341 seconds
Time elapsed[dynamic]: 0.003937 seconds
Time elapsed[guided]: 0.003636 seconds

The num of threads:8
Time elapsed[static]: 0.004447 seconds
Time elapsed[dynamic]: 0.004736 seconds
Time elapsed[guided]: 0.004770 seconds

The num of threads:16
Time elapsed[static]: 0.006597 seconds
Time elapsed[dynamic]: 0.004059 seconds
Time elapsed[guided]: 0.003846 seconds
```

②矩阵规模为 256



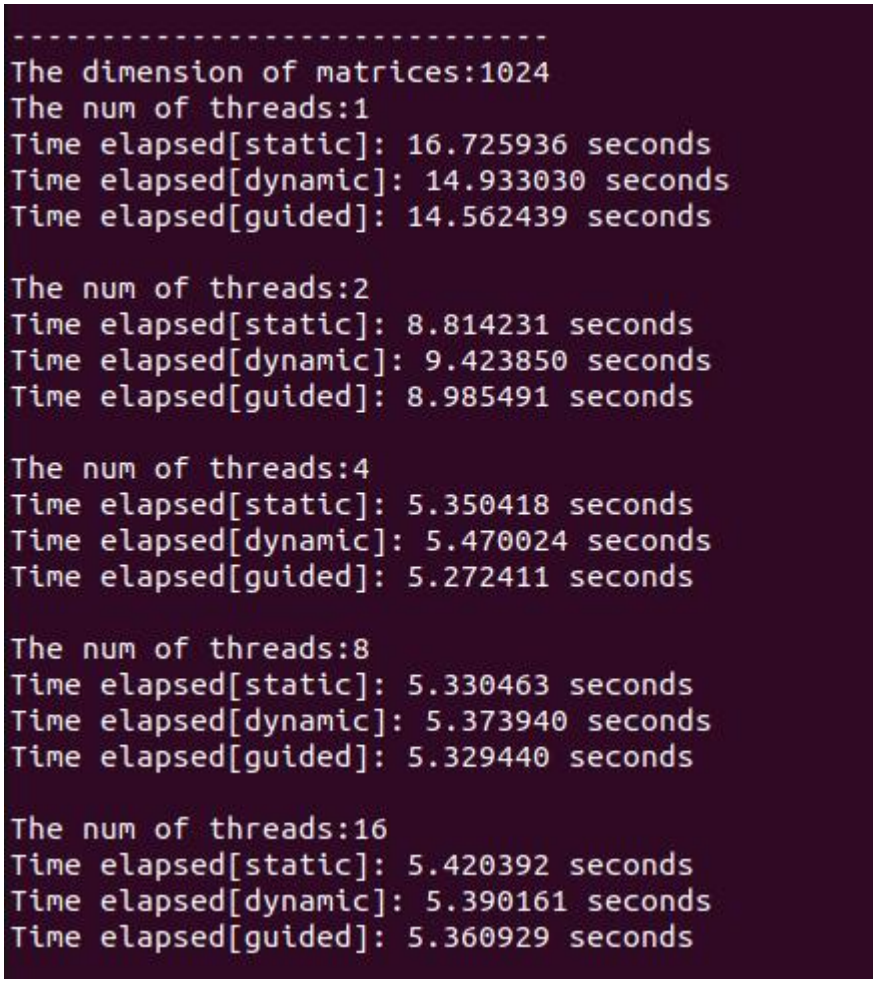
The dimension of matrices:256
The num of threads:1
Time elapsed[static]: 0.101872 seconds
Time elapsed[dynamic]: 0.091652 seconds
Time elapsed[guided]: 0.088744 seconds
The num of threads:2
Time elapsed[static]: 0.049466 seconds
Time elapsed[dynamic]: 0.048607 seconds
Time elapsed[guided]: 0.049216 seconds
The num of threads:4
Time elapsed[static]: 0.041887 seconds
Time elapsed[dynamic]: 0.028144 seconds
Time elapsed[guided]: 0.036779 seconds
The num of threads:8
Time elapsed[static]: 0.034802 seconds
Time elapsed[dynamic]: 0.028577 seconds
Time elapsed[guided]: 0.026981 seconds
The num of threads:16
Time elapsed[static]: 0.031739 seconds
Time elapsed[dynamic]: 0.027196 seconds
Time elapsed[guided]: 0.032895 seconds

③矩阵规模为 512



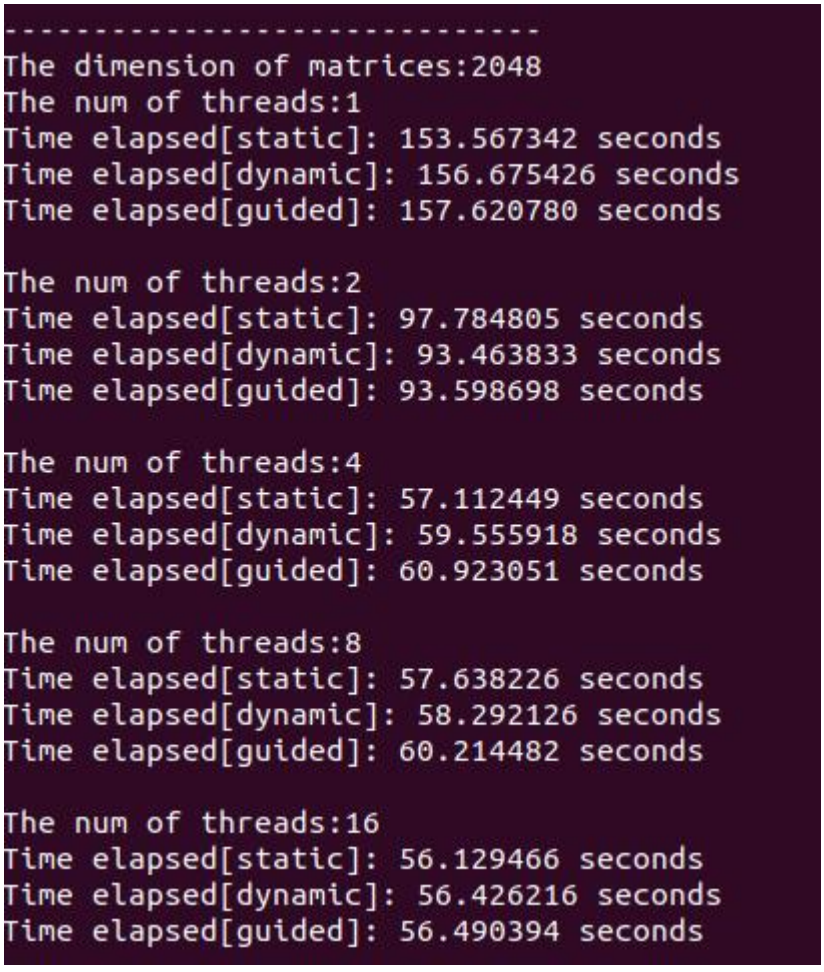
The dimension of matrices:512
The num of threads:1
Time elapsed[static]: 0.900969 seconds
Time elapsed[dynamic]: 0.858064 seconds
Time elapsed[guided]: 0.817635 seconds
The num of threads:2
Time elapsed[static]: 0.480035 seconds
Time elapsed[dynamic]: 0.457815 seconds
Time elapsed[guided]: 0.463208 seconds
The num of threads:4
Time elapsed[static]: 0.283683 seconds
Time elapsed[dynamic]: 0.252694 seconds
Time elapsed[guided]: 0.266138 seconds
The num of threads:8
Time elapsed[static]: 0.274841 seconds
Time elapsed[dynamic]: 0.256810 seconds
Time elapsed[guided]: 0.250527 seconds
The num of threads:16
Time elapsed[static]: 0.279620 seconds
Time elapsed[dynamic]: 0.246772 seconds
Time elapsed[guided]: 0.253925 seconds

④矩阵规模为 1024



The dimension of matrices:1024
The num of threads:1
Time elapsed[static]: 16.725936 seconds
Time elapsed[dynamic]: 14.933030 seconds
Time elapsed[guided]: 14.562439 seconds
The num of threads:2
Time elapsed[static]: 8.814231 seconds
Time elapsed[dynamic]: 9.423850 seconds
Time elapsed[guided]: 8.985491 seconds
The num of threads:4
Time elapsed[static]: 5.350418 seconds
Time elapsed[dynamic]: 5.470024 seconds
Time elapsed[guided]: 5.272411 seconds
The num of threads:8
Time elapsed[static]: 5.330463 seconds
Time elapsed[dynamic]: 5.373940 seconds
Time elapsed[guided]: 5.329440 seconds
The num of threads:16
Time elapsed[static]: 5.420392 seconds
Time elapsed[dynamic]: 5.390161 seconds
Time elapsed[guided]: 5.360929 seconds

⑤矩阵规模为 2048



⑥综合对比

Dimension of Matrices	Num of Threads: 1	Num of Threads: 2	Num of Threads: 4	Num of Threads: 8	Num of Threads: 16
128	Static: 0.009866 s	Static: 0.007252 s	Static: 0.004341 s	Static: 0.004447 s	Static: 0.006597 s
	Dynamic: 0.010199 s	Dynamic: 0.006219 s	Dynamic: 0.003937 s	Dynamic: 0.004736 s	Dynamic: 0.004059 s
	Guided: 0.010997 s	Guided: 0.006793 s	Guided: 0.003636 s	Guided: 0.004770 s	Guided: 0.003846 s
256	Static: 0.101872 s	Static: 0.049466 s	Static: 0.041887 s	Static: 0.034802 s	Static: 0.031739 s
	Dynamic: 0.091652 s	Dynamic: 0.048607 s	Dynamic: 0.028144 s	Dynamic: 0.028577 s	Dynamic: 0.027196 s
	Guided: 0.088744 s	Guided: 0.049216 s	Guided: 0.036779 s	Guided: 0.026981 s	Guided: 0.032895 s
512	Static: 0.900969 s	Static: 0.480035 s	Static: 0.283683 s	Static: 0.274841 s	Static: 0.279620 s
	Dynamic: 0.858064 s	Dynamic: 0.457815 s	Dynamic: 0.252694 s	Dynamic: 0.256810 s	Dynamic: 0.246772 s
	Guided: 0.817635 s	Guided: 0.463208 s	Guided: 0.266138 s	Guided: 0.250527 s	Guided: 0.253925 s
1024	Static: 16.725936 s	Static: 8.814231 s	Static: 5.350418 s	Static: 5.330463 s	Static: 5.420392 s
	Dynamic: 14.933030 s	Dynamic: 9.423850 s	Dynamic: 5.470024 s	Dynamic: 5.373940 s	Dynamic: 5.390161 s
	Guided: 14.562439 s	Guided: 8.985491 s	Guided: 5.272411 s	Guided: 5.329440 s	Guided: 5.360929 s
2048	Static: 153.567342 s	Static: 97.784805 s	Static: 57.112449 s	Static: 57.638226 s	Static: 56.129466 s
	Dynamic: 156.675426 s	Dynamic: 93.463833 s	Dynamic: 59.555918 s	Dynamic: 58.292126 s	Dynamic: 56.426216 s
	Guided: 157.620780 s	Guided: 93.598698 s	Guided: 60.923051 s	Guided: 60.214482 s	Guided: 56.490394 s

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

```
ljj@ljj-virtual-machine:~/parrallel_programming/Lab5$ gcc -shared -fpic -o libparallelfor.so parallel_for.c
ljj@ljj-virtual-machine:~/parrallel_programming/Lab5$ gcc -o matrix_mult main.c -L. -lparallelfor -lpthread
main.c: In function 'main':
main.c:101:24: warning: passing argument 4 of 'parallel_for' from incompatible pointer type [-Wincompatible-pointer-types]
  101 |     parallel_for(0,m,1,matrix_multiply,fun_args,thread_num);
      |                        ^~~~~~~~~~~~~~~
      |                        |
      |                        void * (*)(void *)
main.c:5:63: note: expected 'void (*)(int,  void *)' but argument is of type 'void * (*)(void *)'
    5 | extern void parallel_for(int start, int end, int step, void (*func)(int, void*), void* arg, int num_threads);
      |                                                         ~~~~~~~^~~~~~~~~~~~~~~~~~
ljj@ljj-virtual-machine:~/parrallel_programming/Lab5$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
ljj@ljj-virtual-machine:~/parrallel_programming/Lab5$ ./matrix_mult
Matrix A:
0.94 0.67 0.21 0.73
0.89 0.09 0.20 0.67
0.73 0.95 0.17 0.62
0.60 0.96 0.47 0.20
Matrix B:
0.80 0.18 0.27 0.18
0.26 0.54 0.88 0.01
0.95 0.81 0.96 0.91
0.95 0.31 0.51 0.89
Matrix C:
1.82 0.93 1.42 1.02
1.56 0.58 0.86 0.94
1.59 0.98 1.53 0.86
1.36 1.07 1.57 0.72
Time elapsed: 0.000543 seconds
```

可以看到计算结果是正确的

①矩阵规模为 128

```
ljj@ljj-virtual-machine:~/parrallel_programming/Lab5$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
ljj@ljj-virtual-machine:~/parrallel_programming/Lab5$ ./matrix_mult2
----------------------------
The dimension of matrices: 128
The num of threads: 1
Time elapsed: 0.014595 seconds

The num of threads: 2
Time elapsed: 0.010520 seconds

The num of threads: 4
Time elapsed: 0.013171 seconds

The num of threads: 8
Time elapsed: 0.009032 seconds

The num of threads: 16
Time elapsed: 0.008240 seconds
```

②矩阵规模为 256

```
--------------------------
The dimension of matrices: 256
The num of threads: 1
Time elapsed: 0.119692 seconds

The num of threads: 2
Time elapsed: 0.151434 seconds

The num of threads: 4
Time elapsed: 0.149183 seconds

The num of threads: 8
Time elapsed: 0.140268 seconds

The num of threads: 16
Time elapsed: 0.137160 seconds
```

③矩阵规模为 512

```
--------------------------
The dimension of matrices: 512
The num of threads: 1
Time elapsed: 1.015502 seconds

The num of threads: 2
Time elapsed: 1.151547 seconds

The num of threads: 4
Time elapsed: 1.198142 seconds

The num of threads: 8
Time elapsed: 1.195344 seconds

The num of threads: 16
Time elapsed: 1.169111 seconds
```

④矩阵规模为 1024

```
--------------------------
The dimension of matrices: 1024
The num of threads: 1
Time elapsed: 18.181878 seconds

The num of threads: 2
Time elapsed: 23.101471 seconds

The num of threads: 4
Time elapsed: 27.345289 seconds

The num of threads: 8
Time elapsed: 28.493221 seconds

The num of threads: 16
Time elapsed: 26.961267 seconds
```

⑤矩阵规模为 2048

```
--------------------------
The dimension of matrices: 2048
The num of threads: 1
Time elapsed: 223.219268 seconds

The num of threads: 2
Time elapsed: 270.178678 seconds

The num of threads: 4
Time elapsed: 329.654528 seconds

The num of threads: 8
Time elapsed: 324.716937 seconds

The num of threads: 16
Time elapsed: 331.491747 seconds
```

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（未解决）

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。

但随着线程数的增加性能不仅没有改善，反而下降了，这是后续需要解决的问题。