中山大学计算机院本科生实验报告

（2023 学年春季学期）

课程名称：并行程序设计　　批改人：

实验	Openmp 矩阵乘法	专业（方向）	计算机科学与技术
学号	21307082	姓名	赖耿桂
Email	laigg@mail2.sysu.edu.cn	完成日期	2024.4.27

1. 实验目的（200 字以内）

使用 OpenMP 实现并行通用矩阵乘法，通过设置线程数量（1-16）、矩阵规模（128-2048）、调度方式(调度方式包括默认调度、静态调度、动态调度)并根据运行时间，分析程序的并行性能。

2. 实验过程和核心代码（600 字以内，图文并茂）

本次实验仍使用此前实验所用到的矩阵生成函数 random_matrix_generator(在 Generator.h 中)和打印矩阵函数 printMatrix。

在实验中使用 static(默认调度方式也是 static)、dynamic 两种调度方式进行多线程并行矩阵乘法，其实现如下，主要是预处理命令的不同：

预处理命令说明：

#pragma omp parallel for：openmp 并行 for 循环，在接下来的 for 循环中并行执行迭代；

num_threads(thread_num)：指定线程数量

schedule(static|dynamic)：指定调度方式为 static 或 dynamic

shared(A,B,C)：3 个矩阵被所有线程所共享，一般来说使用共享变量可能会造成并发访问，但由于每个线程写入 C 的位置不同，所以不存在安全性问题。

```
1.  // 使用静态调度方式进行并行矩阵乘法
2.  void matrix_multiply_static(vector<double> A,vector<double> B,vector<double>& C, int thread_num, int M, int N, int K){
3.      C.resize(M*K);   // 为矩阵 C 分配空间
4.      #pragma omp parallel for num_threads(thread_num) schedule(static) shared(A,B,C)
5.          for(int i = 0; i < M; ++i){
6.              for(int k = 0; k < K; ++k){
7.                  C[i*K+k] = 0;
```

```
                for(int j = 0; j < N; ++j){
                    C[i*K+k] += A[i*N+j] * B[j*K+k];
                }
            }
        }
}

// 使用动态调度方式进行并行矩阵乘法
void matrix_multiply_dynamic(vector<double> A,vector<double> B,vector<double>& C, int thread_num, int M, int N, int K){
    C.resize(M*K);  // 为矩阵C分配空间
   #pragma omp parallel for num_threads(thread_num) schedule(dynamic) shared(A,B,C)
        for(int i = 0; i < M; ++i){
            for(int k = 0; k < K; ++k){
                C[i*K+k] = 0;
                for(int j = 0; j < N; ++j){
                    C[i*K+k] += A[i*N+j] * B[j*K+k];
                }
            }
        }
}
```

在 **main** 函数中，实验所需参数的定义和前几次实验一致，可见注释，不再赘述：

```
int M, N, K, thread_num;    // 矩阵规模相关参数M、N、K(矩阵A大小：M*N；矩阵B大小：N*K)；线程数thread_num
double start_time, end_time, using_time;    // 用于计时的参数
```

然后使用 scanf 输入 M、N、K 和 thread_num 的值，在明确矩阵规模之后，对矩阵 A 和 B 进行初始化并打印它们：

```
printf("请输入矩阵规模参数M、N、K(矩阵A的大小为M*N，矩阵B的大小为N*K；在区间[128,2048]内的整数)：\n");
scanf("%d %d %d", &M, &N, &K);
printf("请输入线程数量(在区间[1,16]内的整数)：\n");
scanf("%d", &thread_num);

// 随机生成矩阵A和B
vector<double> A = random_matrix_generator(M, N);
vector<double> B = random_matrix_generator(N, K);
```

```
9.
10.  // 打印矩阵A 和B
11.  printf("Matrix A: \n");
12.  printMatrix(A, M, N);
13.  printf("Matrix B: \n");
14.  printMatrix(B, N, K);
```

声明矩阵 C：

```
1.   vector<double> C;
```

此处仅对 C 进行声明，不做定义，将 C 的定义放到矩阵乘法函数中(因为要比较不同调度方式的并行性能，故每个函数开始运算之前都先把 C 设为全 0 矩阵)。

接下来分别使用 2 种不同的调度方式进行并行矩阵乘法，记录不同调度方式下矩阵乘法所用时间(使用 openmp 的计时函数 omp_get_wtime())，并打印输出矩阵 C(只打印一次矩阵 C，后面使用 dynamic 方式算出来的结果一样，无需重复打印)：

```
1.   /*接下来使用静态调度方式进行并行矩阵乘法 * /
2.   start_time = omp_get_wtime(); // 开始计时
3.   matrix_multiply_static(A, B, C, thread_num, M, N, K);
4.   end_time = omp_get_wtime();          // 结束计时
5.   using_time = end_time - start_time; // 计算执行时间
6.   printf("Matrix C: \n");
7.   printMatrix(C, M, K); // 打印矩阵 C
8.   // 打印执行时间
9.   printf("在%d 个线程并行的情况下，使用 static 调度方式计算大小为%d*%d 的矩阵 A 乘%d*%d 的矩阵 B 所用时间为：%lfs\n", thread_num, M, N, N, K, using_time);
10.
11.  /*接下来使用动态调度方式进行并行矩阵乘法*/
12.  start_time = omp_get_wtime(); // 开始计时
13.  matrix_multiply_dynamic(A, B, C, thread_num, M, N, K);
14.  end_time = omp_get_wtime();          // 结束计时
15.  using_time = end_time - start_time; // 计算执行时间
16.  // 打印执行时间
17.  printf("在%d 个线程并行的情况下，使用 dynamic 调度方式计算大小为%d*%d 的矩阵 A 乘%d*%d 的矩阵 B 所用时间为：%lfs\n", thread_num, M, N, N, K, using_time);
```

3. 实验结果（500 字以内，图文并茂）

- 编译：

```
1.  g++ -fopenmp gemm_openmp.cpp -o gemm_openmp
```

- 运行：

```
1.  ./gemm_openmp
```

- 下为运行示例，仅用于展示程序正确性：

2 种调度方式下不同线程数下不同规模的矩阵所用时间：

使用 static 调度方式，不同线程数下计算不同规模矩阵的乘积所用时间(单位：秒)：

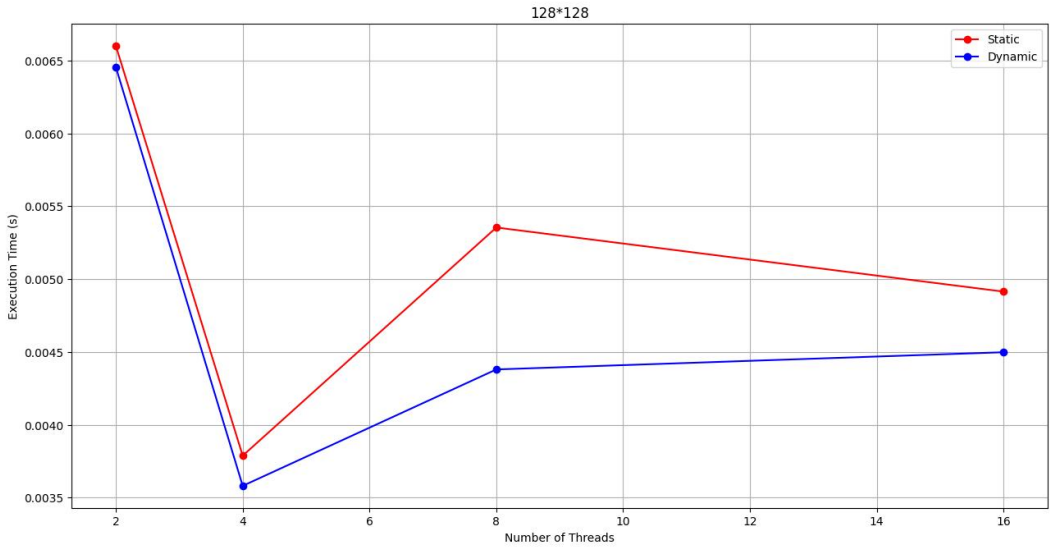
线程数	矩阵规模				
	128	256	512	1024	2048
1	0.013059	0.110208	1.488385	16.382537	237.557735
2	0.0066	0.057483	0.715432	8.770106	118.566468
4	0.003788	0.059848	0.377563	5.374012	64.717245
8	0.005354	0.046469	0.379783	4.556424	72.144375
16	0.004914	0.042713	0.404591	4.747665	71.690008

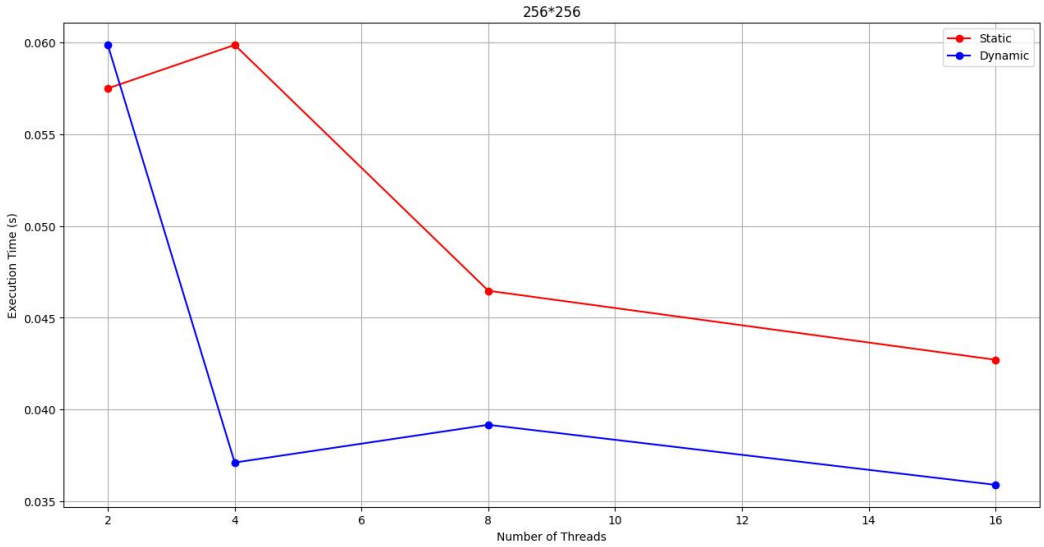
使用 dynamic 调度方式，不同线程数下计算不同规模矩阵的乘积所用时间(单位：秒)：

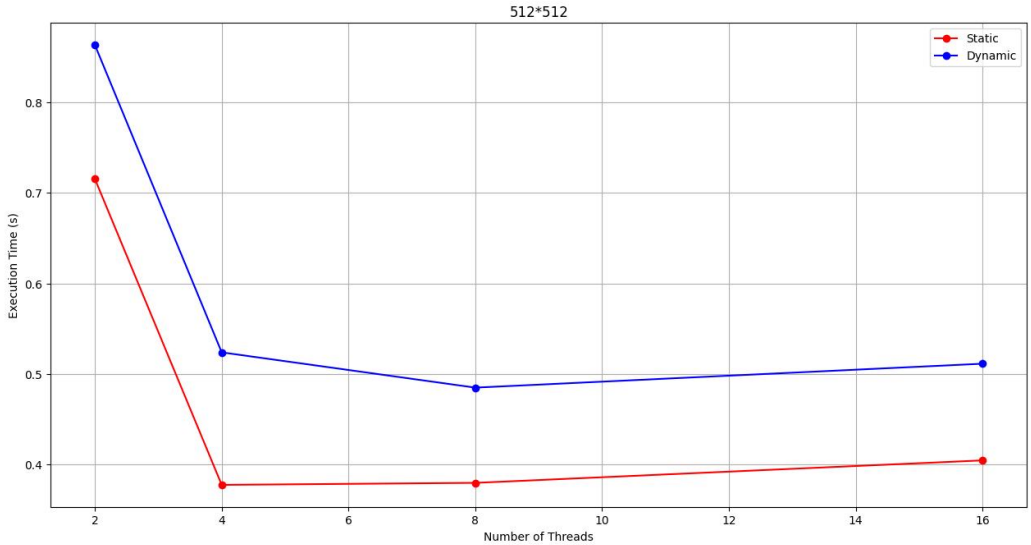
线程数	矩阵规模				
	128	256	512	1024	2048
1	0.013229	0.11383	1.47218	17.014815	238.298944
2	0.006454	0.059835	0.863286	8.803809	118.475931
4	0.003581	0.037121	0.523866	5.810303	66.232447
8	0.00438	0.039175	0.484915	4.941572	70.086717
16	0.004498	0.035906	0.511326	4.81754	78.790971

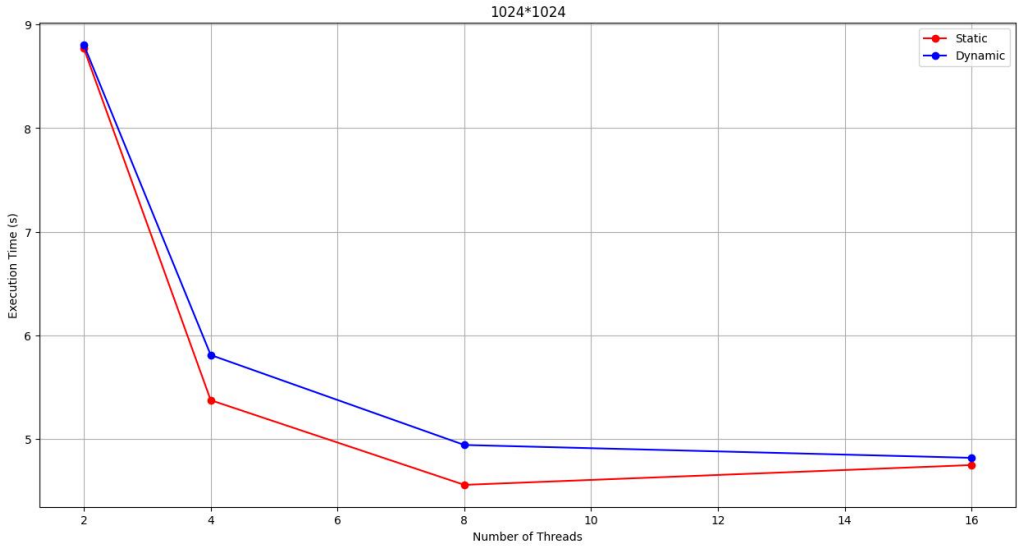
不同规模的矩阵在不同线程下 2 种调度方式的执行时间折线图：

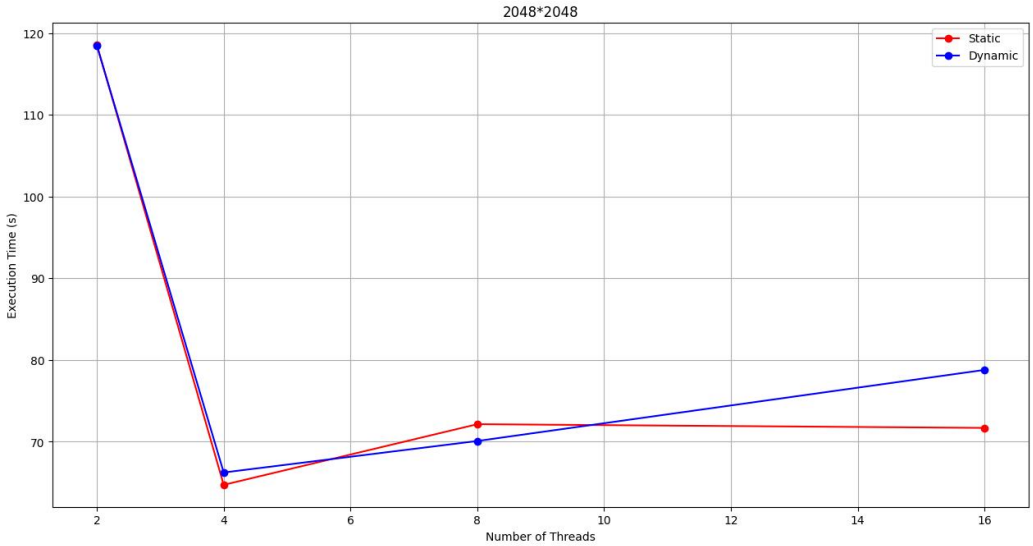
下列各图为使用 python 的 pandas 库读入 csv 文件并使用 matplotlib 库绘制产生的：




256*256
Static
Dynamic
Execution Time (s)
0.060
0.055
0.050
0.045
0.040
0.035
2
4
6
8
10
12
14
16
Number of Threads


512*512
Static
Dynamic
Execution Time (s)
0.8
0.7
0.6
0.5
0.4
2
4
6
8
10
12
14
16
Number of Threads





分析：

①首先从总体上来说，随着线程数的增加，程序的并行性能有所提高，即使可能出现线程数增加但是执行时间也增加的情况，但也比线程数较少时的性能要好，可能是线程数增加导致调度开销增加、线程数大于处理器数造成资源竞争的原因。

②当矩阵规模较小(128、256)时，调度开销对性能的影响会更大一些，特别是在静态调度中。由于静态调度会将迭代分配给线程，若循环迭代次数很少，线程可能会在分配的工作完成后空闲，导致负载不平衡和额外的调度开销。而动态调度则是主动请求分配任务，因此不会出现线程可能会在分配的工作完成后空闲的情况。

③当矩阵规模较大(512、1024、2048)时，静态调度方式将循环迭代静态地分配给线程，而且循环迭代数量很大时，调度开销相对于总计算量会减少。此外，由于迭代数量增加，负载更有可能是均

衡的，因此静态调度能够更有效地利用线程并行执行迭代。相较于动态调度，在大规模问题上，静态调度还少了请求开销。

4. 实验感想（200 字以内）

通过本次实验，我学习掌握了 openmp 并行编程，了解了 openmp 编程的基本语法，巩固了 openmp 相关的理论知识。

在实验过程中，随着矩阵规模的变化，static 和 dynamic 两种调度方式展示出了不同的性能，这使得我在分析实验结果时遇到了较大的困难。但是通过分析 openmp 并行编程中可能出现的开销、翻阅教材以及查阅相关资料之后，明白了两种调度方式的差异，对 openmp 并行编程有了更深入的理解。