中山大学计算机学院

并行程序设计

本科生实验报告

（2024学年春季学期）
课程名称：Parallel Programming
批改人：

实验	基于OpenMP的并行矩阵乘法	专业(方向)	计算机科学与技术
学号	21307185	姓名	张礼贤
Email	zhanglx78@mail2.sysu.edu.cn	完成日期	2024.4.22

1. 实验目的

1. 了解OpenMP的基本使用方法
2. 了解OpenMP的调度方式
3. 了解Pthreads对于OpenMp调度方式的实现

---

2. 实验过程和核心代码

2.1 实验环境

1. OpenMp 库
2. Pthreads 库
3. C 语言编译环境
4. Linux操作系统

2.2 实验步骤

2.2.1 OpenMP通用矩阵乘法

1. **生成随机数矩阵：**
   通过 `(double)rand() / RAND_MAX` 函数生成随机数初始化矩阵

2. **执行三种调度：**

   - **default：**
     OpenMP运行时环境决定如何分配循环的迭代。这种策略的具体行为可能因编译器和环境而异，但通常会尝试在所有线程之间均匀分配迭代。

   - **static：**
     循环的迭代在编译时就被分配给各个线程。如果没有指定调度块大小，那么迭代会被均匀分配。如果指定了块大小，那么每个线程会依次获得一个块，直到所有迭代都被分配。

   - **dynamic：**
     循环的迭代在运行时被动态分配给线程。当一个线程完成了它的迭代块后，它会从未分配的迭代中获取新的块。可以确保所有线程都忙碌，而不会有线程在等待其他线程完成长任务。

3. **计算时间：**
   通过 `omp_get_wtime()` 函数计算时间

2.2.2 构造基于Pthreads的并行for循环分解、分配、执行机制

1. **通过结构体定义任务参数：**
   结构体中包括任务的起始位置、结束位置、线程编号等信息

2. **编写线程执行函数：**
   接收一个 `parallel_for_args` 结构体作为参数，通过循环调用传入的任务函数指针

3. **编写parallel_for函数：**
   接收起始位置 start、结束位置 end、增量 inc、任务函数指针 functor 和参数 arg 以及线程数量 num_threads参数，启动线程分块执行任务

2.3 核心代码

2.3.1 OpenMp通用矩阵乘法

1. **生成随机数矩阵：**

```
matrix[i * n + j] = (double)rand() / RAND_MAX;
```

2. **执行三种调度与计时：**

```
#pragma omp parallel for
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        double sum = 0.0;
        for (int k = 0; k < n; k++)
        {
            sum += A[i * n + k] * B[k * n + j];
        }
        C[i * n + j] = sum;
    }
}

#paragma omp parallel for schedule(static)
for (int i = 0; i < n; i++)
{
    // 矩阵乘法
}

#paragma omp parallel for schedule(dynamic)
for (int i = 0; i < n; i++)
{
    // 矩阵乘法
}
```

2.3.2 构造基于Pthreads的并行for循环分解、分配、执行机制

1. **通过结构体定义任务参数：**

```
typedef struct parallel_for_args
{
    int start;  // 起始位置
    int end;    // 结束位置
    int inc;    // 增量
    void (*functor)(int, void *);   // 任务函数指针
    void *arg;  // 参数
    int thread_id;  // 线程编号
} parallel_for_args;
```

2. **编写线程执行函数：**
   通过循环调用任务函数，在划分的任务范围内通过functor函数指针调用任务函数

```
// 线程执行函数
void *parallel_for_thread(void *args)
{
    // 强制类型转换
    parallel_for_args *p_args = (parallel_for_args *)args;
    // 循环调用任务函数
    for (int i = p_args->start; i < p_args->end; i += 
p_args->inc)
    {
        p_args->functor(i, p_args->arg);
    }
    return NULL;
}
```

3. **编写parallel_for函数：**
   根据线程数量将任务范围分成多个块，并为每个块创建一个 parallel_for_args 结构体，然后启动对应数量的线程执行这些块的任务。

```
void parallel_for(int start, int end, int inc, void
(*functor)(int, void *), void *arg, int num_threads)
{
    pthread_t threads[num_threads]; // 线程数组
    parallel_for_args args[num_threads];    // 任务参数数
组
    // 创建线程
    for (int i = 0; i < num_threads; i++)
    {
        args[i].start = start + i * inc;
        args[i].end = end;
        args[i].inc = num_threads * inc;
        args[i].functor = functor;
        args[i].arg = arg;
        args[i].thread_id = i;
        pthread_create(&threads[i], NULL,
parallel_for_thread, &args[i]);
    }
    for (int i = 0; i < num_threads; i++)
    {
        pthread_join(threads[i], NULL);
    }
}
```

---

3. 实验结果分析

3.1 OpenMp通用矩阵乘法

- **部分实验截图：**

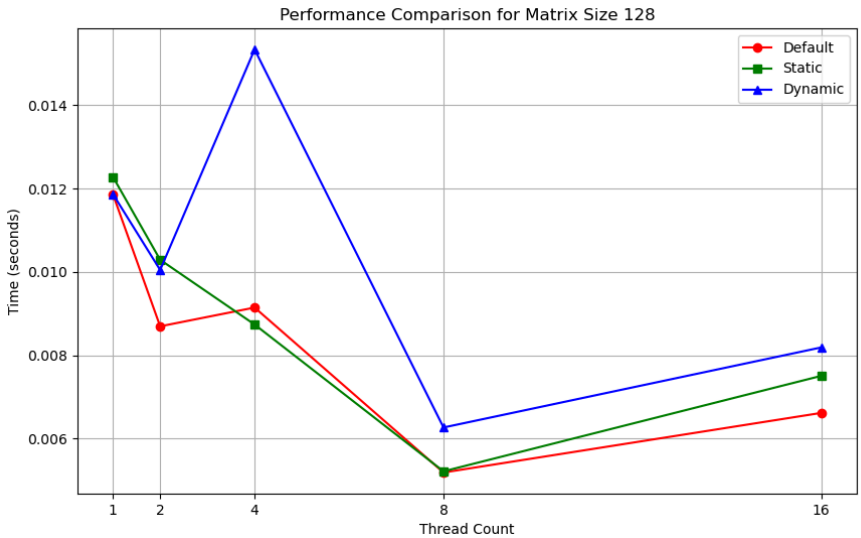
```
Threads: 16, Matrix Size: 256, Schedule: Dynamic, Elapsed Time: 0.066060 seconds
Threads: 16, Matrix Size: 512, Schedule: Default, Elapsed Time: 0.560676 seconds
Threads: 16, Matrix Size: 512, Schedule: Static, Elapsed Time: 0.509808 seconds
Threads: 16, Matrix Size: 512, Schedule: Dynamic, Elapsed Time: 0.469646 seconds
Threads: 16, Matrix Size: 1024, Schedule: Default, Elapsed Time: 8.776861 seconds
Threads: 16, Matrix Size: 1024, Schedule: Static, Elapsed Time: 9.718131 seconds
Threads: 16, Matrix Size: 1024, Schedule: Dynamic, Elapsed Time: 8.952377 seconds
Threads: 16, Matrix Size: 2048, Schedule: Default, Elapsed Time: 95.129859 seconds
Threads: 16, Matrix Size: 2048, Schedule: Static, Elapsed Time: 86.378478 seconds
Threads: 16, Matrix Size: 2048, Schedule: Dynamic, Elapsed Time: 85.901037 seconds
(base) www@www-ThinkPad-T440:~/桌面/Work_Space/MPI_Work/week5$
```

- **统计表格：**

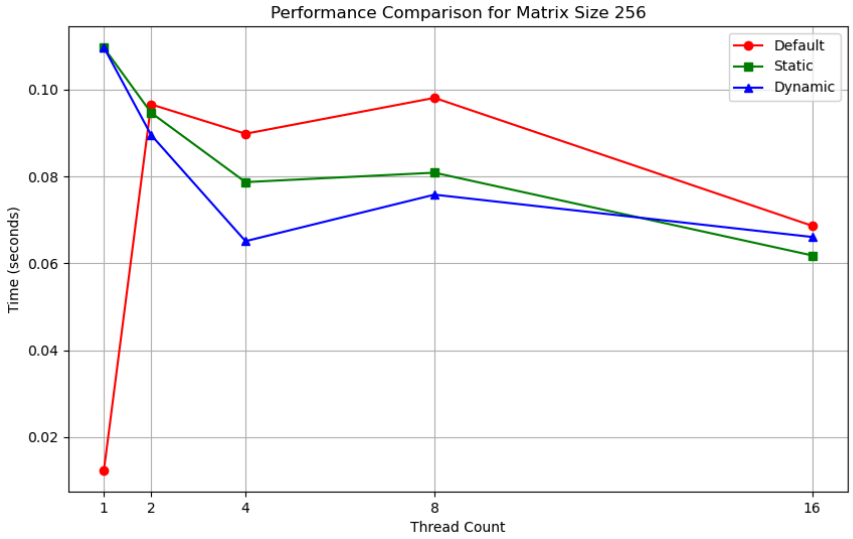
  统一单位为秒，并且单元格的三元组从上到下分别为default、static、dynamic

n \ thread_count	128	256	512	1024	2048
1	0.011864	0.114801	0.998369	19.350333	158.487825
	0.012282	0.109652	1.019061	19.338437	160.430407
	0.011869	0.109726	1.172297	19.383974	179.762714
2	0.008697	0.096607	0.557197	10.853303	102.270218
	0.010293	0.094684	0.607556	10.719360	116.219981
	0.010048	0.089556	0.922021	10.618772	148.299876
4	0.009147	0.089837	1.262526	14.572288	138.939475
	0.008740	0.078699	1.152207	14.836469	138.599989
	0.015339	0.065096	0.938071	13.290709	113.775346
8	0.005190	0.098107	1.466041	15.929347	132.063916
	0.005218	0.080871	1.214474	15.468089	113.519871
	0.006270	0.075811	1.156358	15.821204	107.482511
16	0.006620	0.068635	0.560676	8.776861	95.129859
	0.007509	0.061810	0.509808	9.718131	86.378478
	0.008191	0.066060	0.469646	8.952377	85.901037

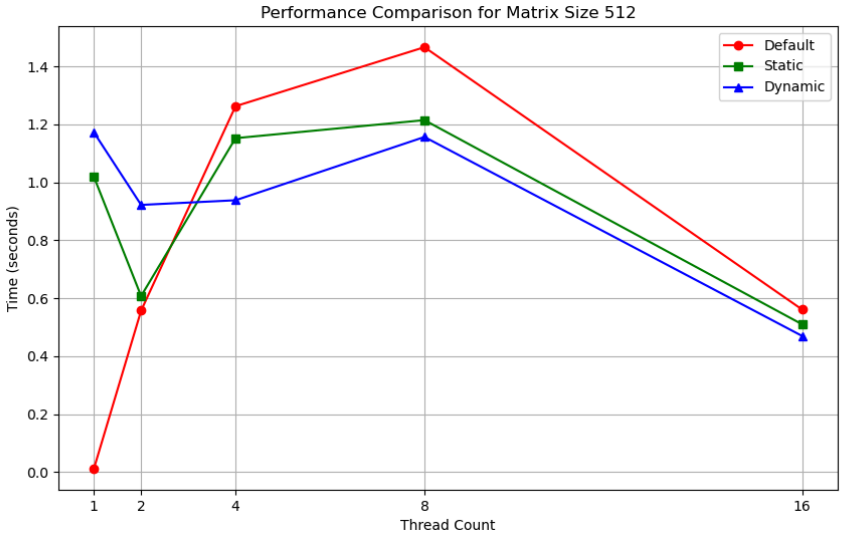
- **图形化展示：**
  - 128



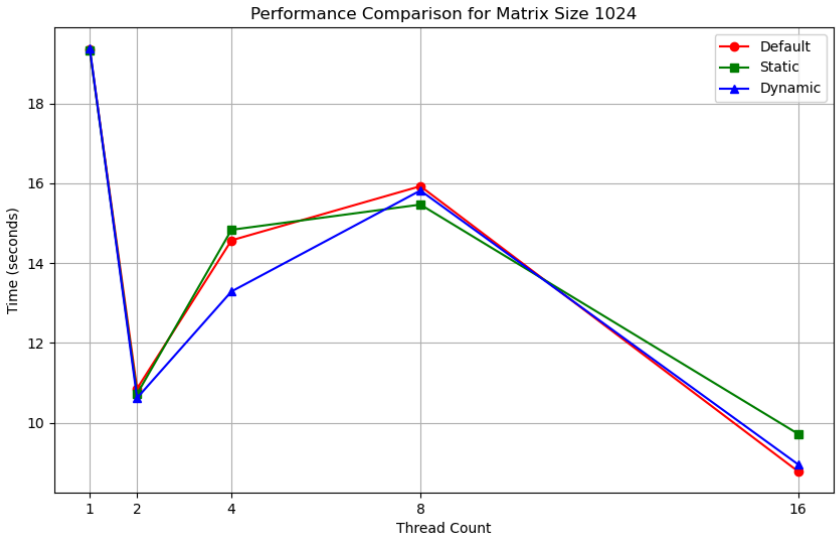
- 256


Performance Comparison for Matrix Size 256
Default
Static
Dynamic
Time (seconds)
0.10
0.08
0.06
0.04
0.02
1
2
4
8
16
Thread Count

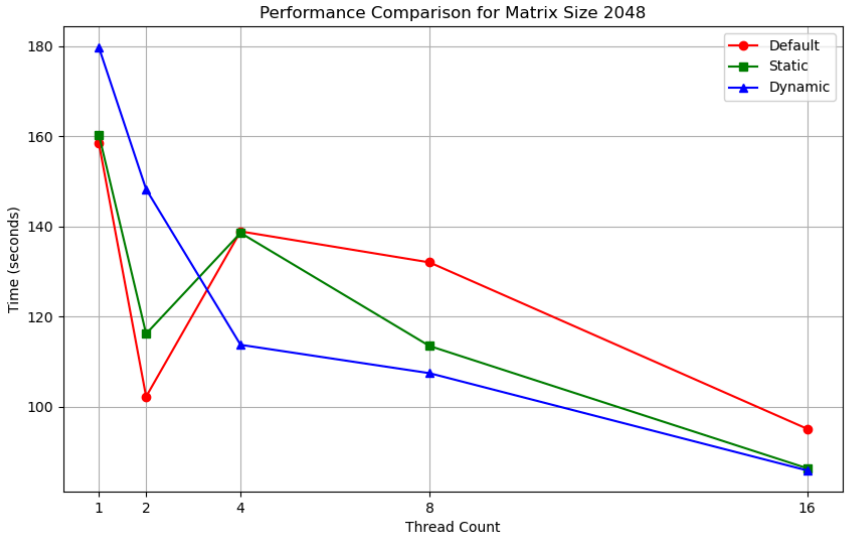
- 512


Performance Comparison for Matrix Size 512
Default
Static
Dynamic
Time (seconds)
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
1
2
4
8
16
Thread Count

- 1024


Performance Comparison for Matrix Size 1024
Default
Static
Dynamic
Time (seconds)
18
16
14
12
10
1
2
4
8
16
Thread Count

- 2048



- **结果分析：**
  - 在同线程数目下，通过表格可以看出计算时间随着n的增加而增加
  - 在给定矩阵规模的情况下：
    - 在矩阵规模为128时，default调度的时间最短，dynamic调度的时间最长，并且在8线程的时候三种调度方法时间最短。在小规模问题下，default调度能够更有效地管理任务分配，避免了线程间的竞争和通信开销，而dynamic调度由于需要动态地分配任务，导致了额外的开销。
    - 在矩阵规模为256以及更大时，dynamic调度的时间最短，default调度的时间最长，并且在16线程的时候三种调度方法时间最短。在较大规模问题下，dynamic调度能够更好地动态地分配任务，而default调度由于其静态分配的特性，在处理较大规模问题时可能会导致任务不均衡，从而增加了总体时间开销。
    - 由于电脑的性能限制，在运行的时候大致分配了两个核心，因此在1到2线程的时候执行时间明显下降，但是之后又有所上升，线程增加而核心有限，造成了伪并行的现象，反而增加了通信开销。

3.2 构造基于Pthreads的并行for循环分解、分配、执行机制

- **部分实验截图：**

```
Matrix_size: 1024, Num_thread: 2, Time_Spent: 13.110163 seconds
Matrix_size: 2048, Num_thread: 2, Time_Spent: 118.365790 seconds
Matrix_size: 128, Num_thread: 4, Time_Spent: 0.007454 seconds
Matrix_size: 256, Num_thread: 4, Time_Spent: 0.089706 seconds
Matrix_size: 512, Num_thread: 4, Time_Spent: 0.495962 seconds
Matrix_size: 1024, Num_thread: 4, Time_Spent: 11.206086 seconds
Matrix_size: 2048, Num_thread: 4, Time_Spent: 110.865360 seconds
Matrix_size: 128, Num_thread: 8, Time_Spent: 0.007403 seconds
Matrix_size: 256, Num_thread: 8, Time_Spent: 0.079884 seconds
Matrix_size: 512, Num_thread: 8, Time_Spent: 0.492105 seconds
Matrix_size: 1024, Num_thread: 8, Time_Spent: 9.463449 seconds
```

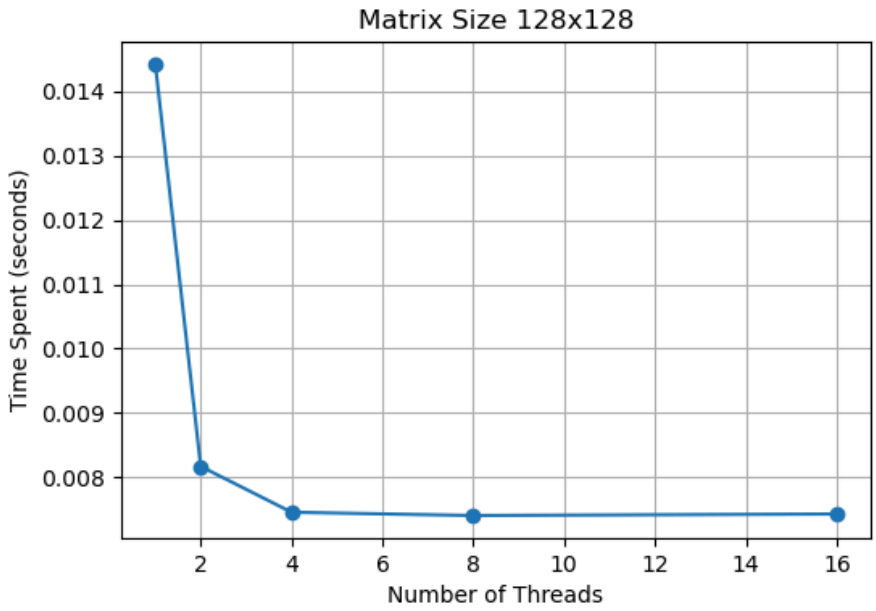
- **统计表格：**

matrix_size\n	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
128	0.014425	0.008167	0.007454	0.007403	0.007427
256	0.122783	0.095056	0.089706	0.079884	0.087455
512	0.999063	0.540252	0.495962	0.492105	0.504445
1024	21.046921	13.110163	11.206086	9.463449	9.388903
2048	206.909898	118.365790	110.865360	103.334304	107.905309

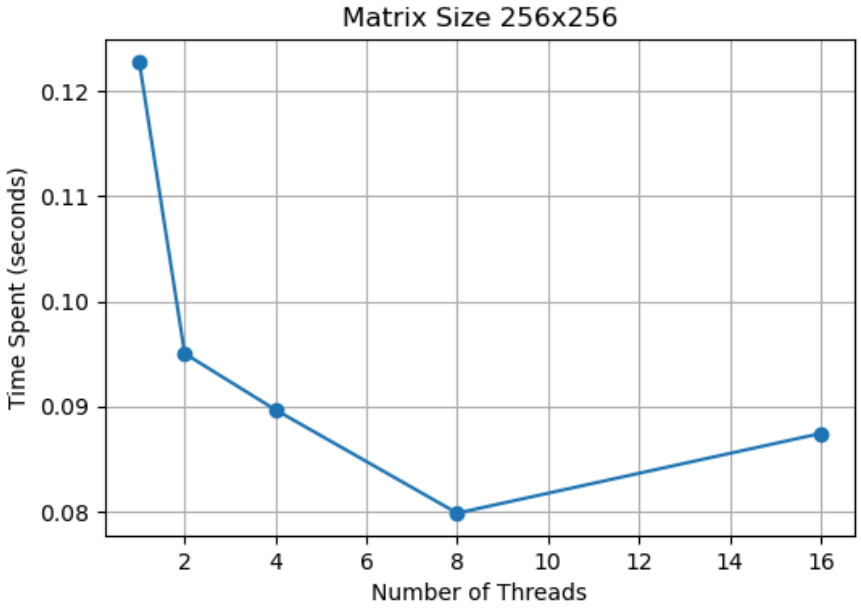
- **效率表格：**

	1	2	4	8	16
128	1.000	0.442	0.215	0.122	0.078
256	1.000	0.323	0.152	0.096	0.056
512	1.000	0.462	0.224	0.127	0.079
1024	1.000	0.401	0.209	0.139	0.090
2048	1.000	0.437	0.207	0.125	0.077

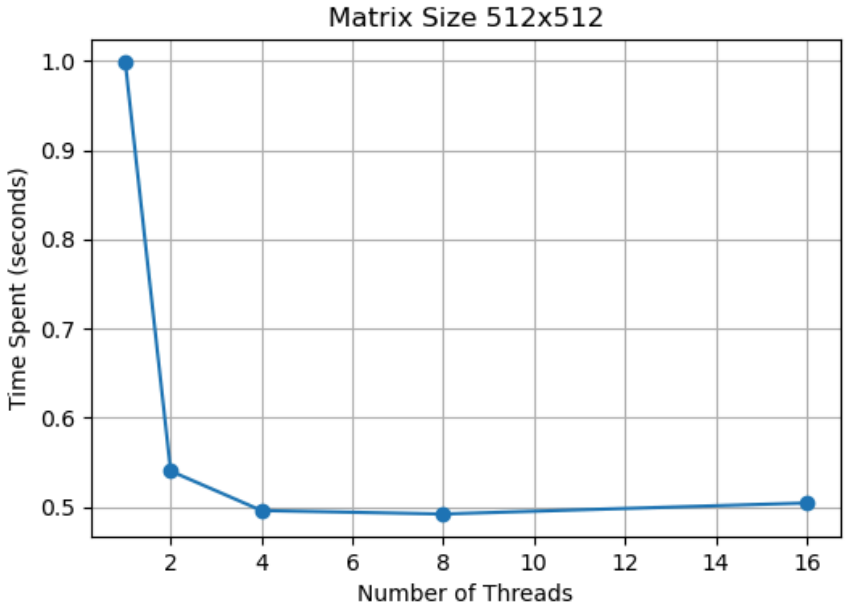
- **图形化展示：**
  - 128



- 256

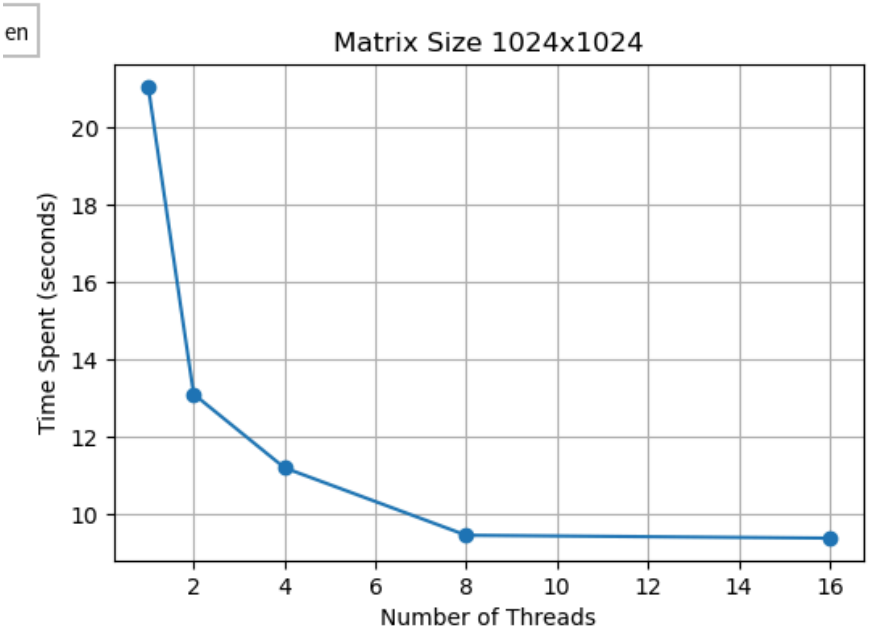

Matrix Size 256x256
Time Spent (seconds)
0.12
0.11
0.10
0.09
0.08
2
4
6
8
10
12
14
16
Number of Threads

- 512


Matrix Size 512x512
Time Spent (seconds)
1.0
0.9
0.8
0.7
0.6
0.5
2
4
6
8
10
12
14
16
Number of Threads

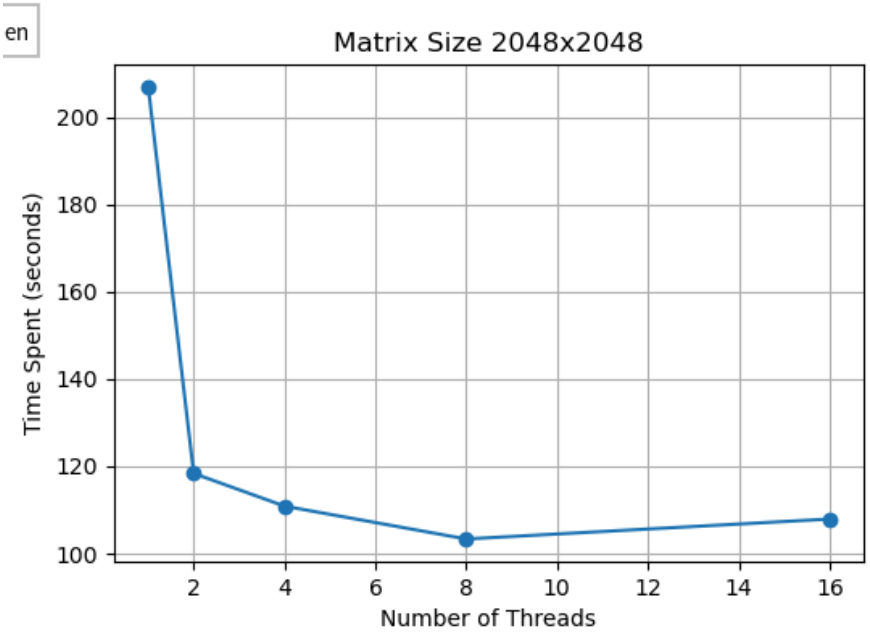
- 1024

en

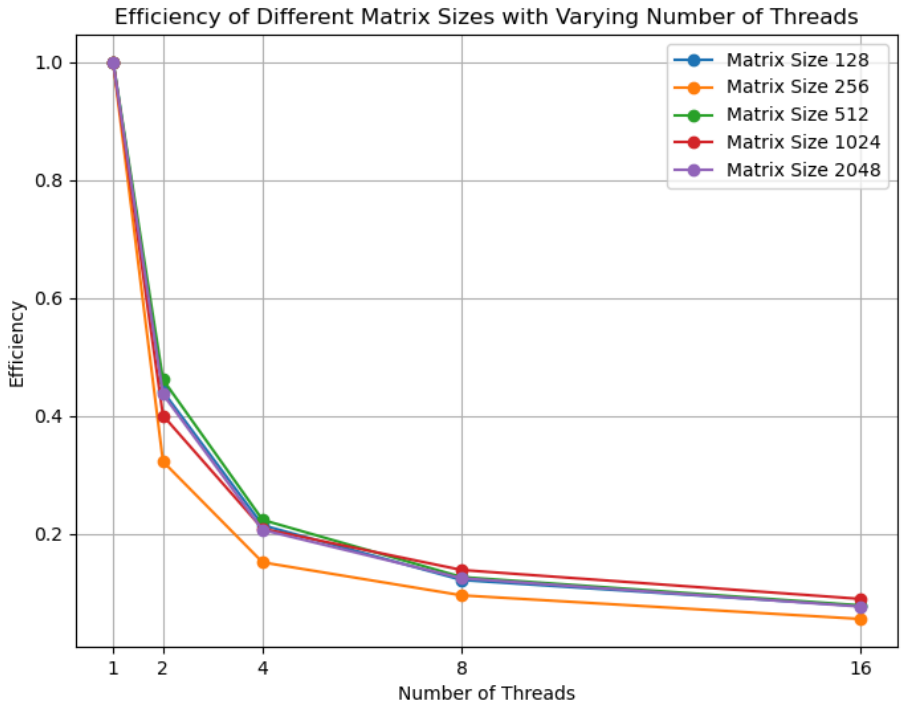


- 2048

en



- 效率：



- **结果分析：**
  - 利用Pthreads实现的并行for循环在矩阵规模一定的情况下，随着线程数的增加，计算时间逐渐减少，少数实验样本出现回升，但总体趋势是减少
  - 与Omp在2到4个线程后计算时间上升不同，通过Pthread构造的for循环实现了良好的任务分配，可以更精细地控制线程的创建和任务分配，更好地避免资源竞争，减少上下文切换，并且可以根据任务的特性更好地优化任务分配
  - 对于效率而言，线程数目增加，效率越低，通信开销增加，核心数量有限。

---

4. 选做：实现调度方式的设置

4.1 实验步骤

1. 实现的步骤与上面的基于Pthreads的并行for循环分解、分配、执行机制类似，为了实现动态调度增添了互斥锁
2. 各个线程不是通过循环划分任务，而是在一个循环中反复调用
3. 如果一个任务执行完成（代表锁被释放），则空闲的进程获取锁，并执行划定的任务分区，之后更新任务分区的起始和结束指针

4. 执行完毕之后释放锁，等待其他空闲的线程获取锁并执行任务分区，从而实现动态调度
5. 调用方式与前面的实验保持一致

4.2 核心代码

1. **定义互斥锁：**

```
pthread_mutex_t mutex;
```

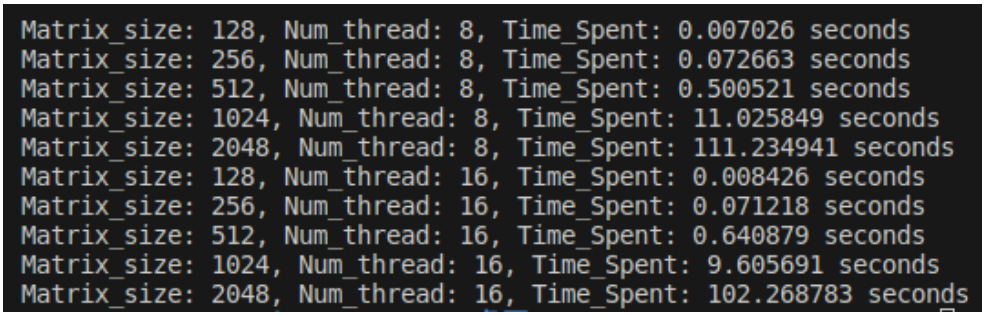
2. **线程执行函数：**

```
void *thread_function(void *args)
{
    struct parallel_for_args *pf_args = (struct
parallel_for_args *)args;
    void *(*functor)(int, void *) = pf_args->functor;
    void *arg = pf_args->arg;
    int i;

    while (1)
    {
        pthread_mutex_lock(&task_mutex); // 加锁
        i = pf_args->start; // 获取任务起始位置
        pf_args->start += pf_args->inc; // 更新任务起始位
置
        pthread_mutex_unlock(&task_mutex); // 解锁
        if (i >= pf_args->end) // 所有任务执行完毕，退出循
环
            break;
        (*functor)(i, arg); // 执行任务
    }
    return NULL;
}
```

4.3 实验结果

- **部分实验截图：**



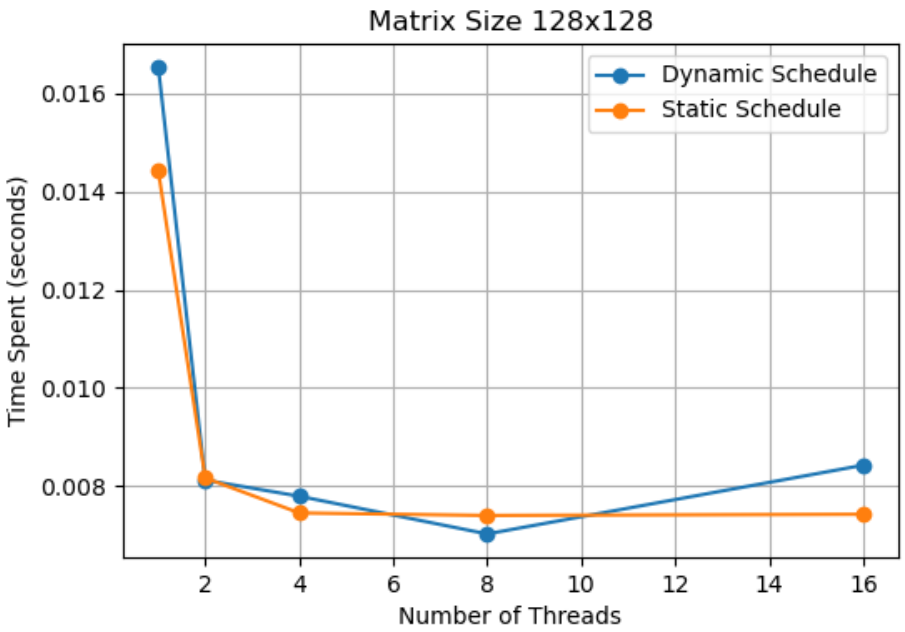
- **统计表格：**

matrix_size\n	1 Thread	2 Threads	4 Threads	8 Threads	16 Threads
128	0.016536	0.008122	0.007795	0.007026	0.008426
256	0.133756	0.082424	0.087185	0.072663	0.071218
512	0.952308	0.537093	0.498419	0.500521	0.640879
1024	19.940085	11.362789	10.685909	11.025849	9.605691
2048	201.097839	117.485825	109.637206	111.234941	102.268783

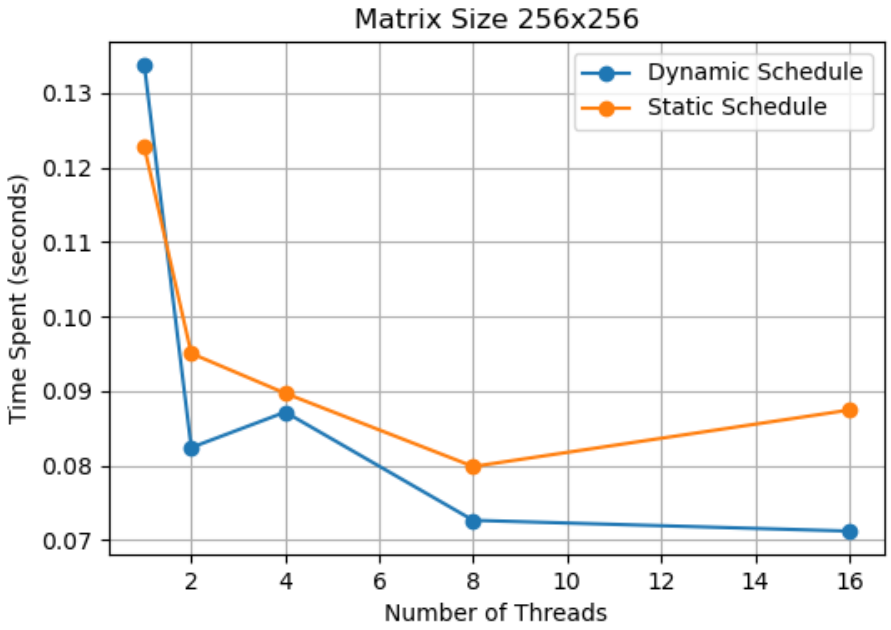
- **效率表格：**

	1	2	4	8	16
128	1.000	0.509	0.236	0.147	0.078
256	1.000	0.406	0.170	0.115	0.075
512	1.000	0.443	0.212	0.119	0.059
1024	1.000	0.439	0.207	0.113	0.083
2048	1.000	0.428	0.204	0.113	0.079

- **图形化展示：**
  - 128

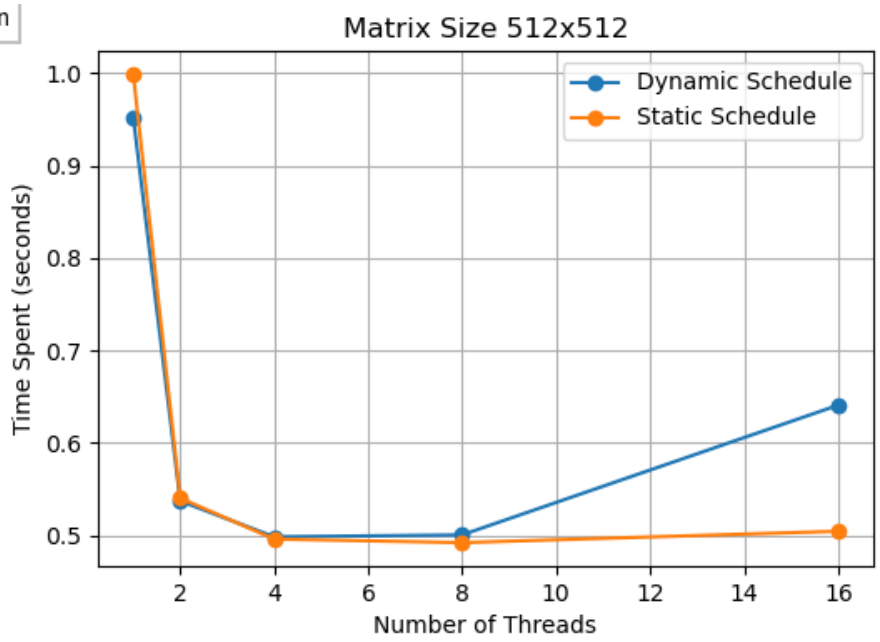


- 256

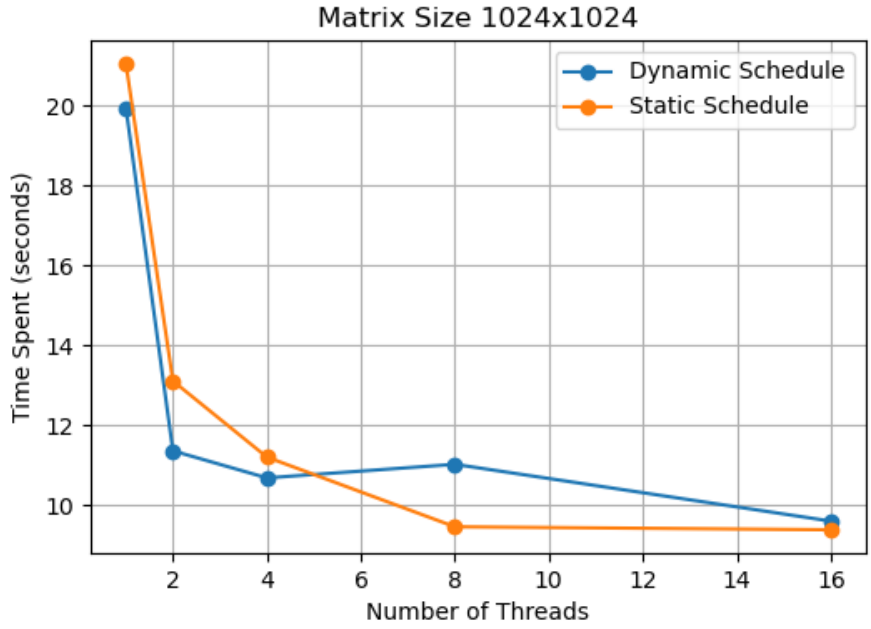

Matrix Size 256x256
Dynamic Schedule
Static Schedule
Time Spent (seconds)
0.13
0.12
0.11
0.10
0.09
0.08
0.07
2
4
6
8
10
12
14
16
Number of Threads

- 512

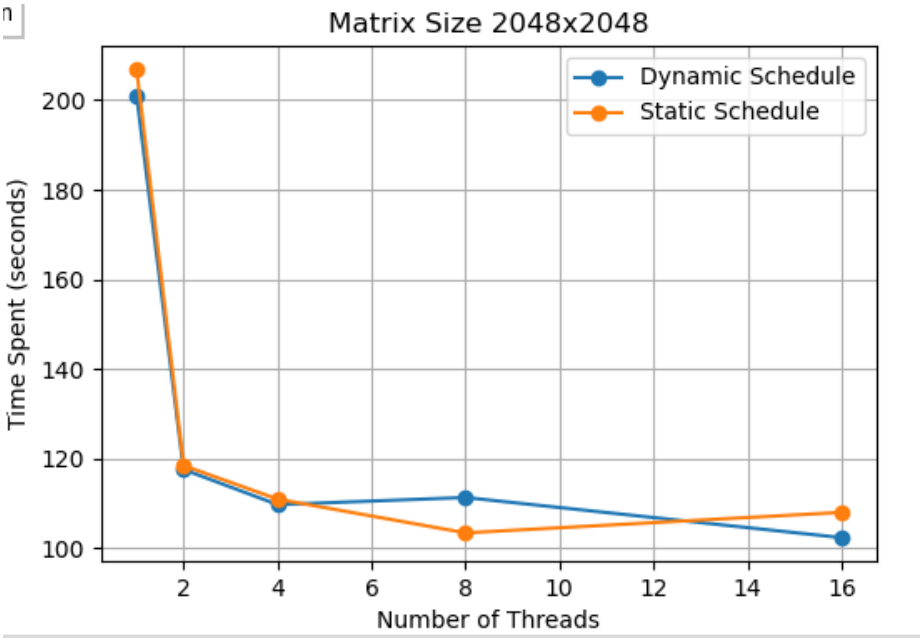
en


Matrix Size 512x512
Dynamic Schedule
Static Schedule
Time Spent (seconds)
1.0
0.9
0.8
0.7
0.6
0.5
2
4
6
8
10
12
14
16
Number of Threads

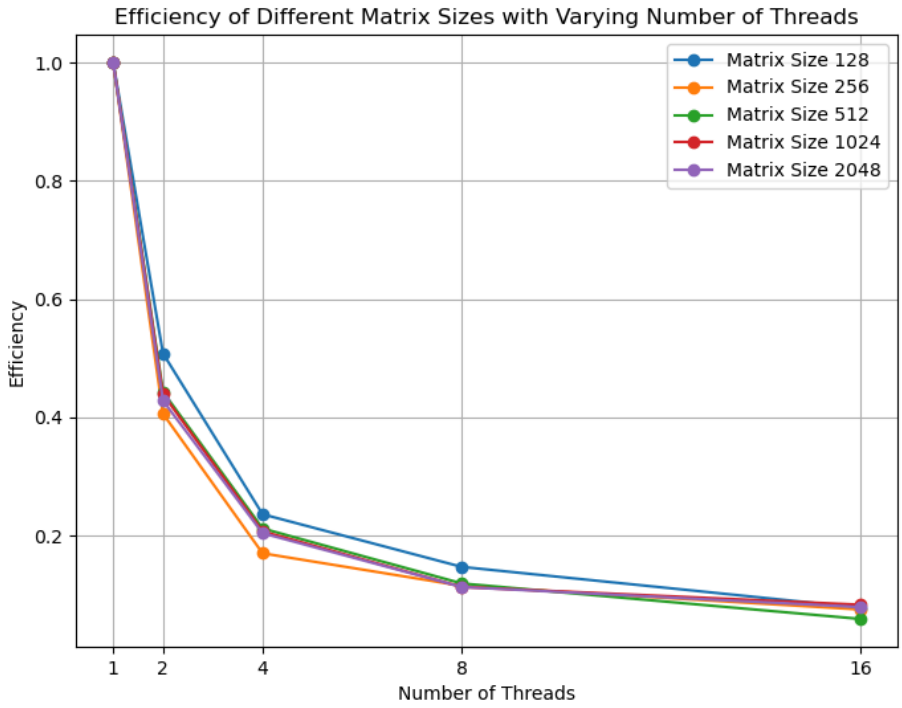
- 1024


Matrix Size 1024x1024
Dynamic Schedule
Static Schedule
Time Spent (seconds)
Number of Threads
20
18
16
14
12
10
2
4
6
8
10
12
14
16

- 2048


Matrix Size 2048x2048
Dynamic Schedule
Static Schedule
Time Spent (seconds)
Number of Threads
200
180
160
140
120
100
2
4
6
8
10
12
14
16

- 效率：



- **结果分析：**
  - 同等线程规模下，动态调度的计算时间随着矩阵规模的增大而增大
  - 同等矩阵规模下：
    - 在线程较少的情况下，dynamic的性能明显优于static默认调度
    - 但是在线程数目增加后，dynamic的性能逐渐下降，这是因为动态调度需要频繁地加锁和解锁，导致了额外的开销，而static调度由于静态分配的特性，能够更好地避免这种开销，因此在线程数目增加后，static调度的性能逐渐优于dynamic调度
  - 在效率方面，线程数目增加，效率逐渐下降，这是因为线程数目增加，通信开销增加，核心数量有限。

---

5. 实验感想

- 通过本次实验，我学会了如何使用OpenMP和实现并行for循环，以及如何通过不同的调度方式来优化任务分配，提高并行程序的性能
- 另外，我还利用了Pthread实现了OpenMp的静态和动态调度，通过互斥锁实现了动态调度，从而更好地理解了OpenMp的调度机制