



# 中山大学数据科学与计算机学院本科生实验报告

(2020 秋季学期)

课程名称：高性能程序设计 任课老师：黄聘 批改人：

年级 + 班级	18 级计算机	年级 (方向)	计算机科学
学号	18340236	姓名	朱煜
Email	zhuy85@mail2.sysu.edu.cn	完成日期	2020.10.30

## 1 实验目的

### 1.1 通过 OpenMP 实现通用矩阵乘法

通过 OpenMP 实现通用矩阵乘法 (Lab1) 的并行版本, OpenMP 并行线程从 1 增加至 8, 矩阵规模从 512 增加至 2048。

通用矩阵乘法 **GEMM** 通常定义为:

$$C = AB$$
$$C_{m,n} = \sum_{k=1}^N A_{m,k} B_{k,n}$$

输入: M, N, K 三个整数 (512~2048)

问题描述: 随机生成  $M * N$  和  $N * K$  的两个矩阵 A, B, 对这两个矩阵做乘法得到矩阵 C

输出: A, B, C 三个矩阵以及矩阵计算的时间

### 1.2 基于 OpenMP 的通用矩阵乘法优化

分别采用 OpenMP 的默认任务调度机制、静态调度 `schedule(static, 1)` 和动态调度 `schedule(dynamic,1)` 的性能, 实现 `#pragma omp for`, 并比较其性能。

### 1.3 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制。

- 1) 基于 pthreads 的多线程库提供的基本函数, 如线程创建、线程 join、线程同步等。构建 `parallel_for` 函数对循环分解、分配和执行机制, 函数参数包括但不限于 (int start, int end, int increment, void \*(\*functor)(void\*), void \*arg, int num\_threads); 其中 start 为循环开始索引; end 为结束索引; increment 每次循环增加索引数; functor 为函数指针, 指向的需要被并行执行循环程序块; arg 为 functor 的入口参数; num\_threads 为并行线程数。



- 2) 在 Linux 系统中将 parallel\_for 函数编译为.so 文件，由其他程序调用。
- 3) 将基于 OpenMP 的通用矩阵乘法的 omp parallel for 并行，改造成基于 parallel\_for 函数并行化的矩阵乘法，注意只改造**可被并行执行的 for 循环**（例如无 race condition、无数据依赖、无循环依赖等）。

举例说明：将串行代码：

---

```
1 for ( int i = 0; i < 10; i++ ){
2     A[i] = B[i] * x + C[i]
3 }
```

---

替换为——>

---

```
1 parallel_for(0, 10, 1, functor, NULL, 2);
2 struct for_index {
3     int start;
4     int end;
5     int increment;
6 }
7 void * functor (void * args){
8     struct for_index * index = (struct for_index *) args;
9     for (int i = index->start; i < index->end; i = i + index->increment){
10         A[i] = B[i] * x + C[i];
11     }
12 }
```

---

编译后执行阶段：多线程执行

在两个线程情况下：

Thread0: start 和 end 分别为 0, 5

Thread1: start 和 end 分别为 5, 10

---

```
1 void * functor(void * arg){
2     int start = my_rank * (10/2)
3     int end = start + 10/2;
4     for(int j = start, j < end, j++)
5         A[j] = B[j] * x + C[j];
6 };
```

---



## 2 实验过程

### 2.1 通过 OpenMP 实现通用矩阵乘法

为了方便数据传输，使用一维数组保存矩阵，矩阵元素使用 `int` 型，随机初始化矩阵，代码如下

---

```
1 void FillMatrix(int *matrix, int row, int col)
2 {
3     for (int i = 0; i < row; ++i)
4         for (int j = 0; j < col; ++j)
5             matrix[i * col + j] = random(0, 9);
6 }
```

---

根据 **GEMM** 实现简单的矩阵乘法，代码如下

---

```
1 void MatrixMul(int *A, int *B, int *C, int m, int n, int k)
2 {
3     for (int i = 0; i < m; ++i)
4     {
5         for (int j = 0; j < k; ++j)
6         {
7             int temp = 0;
8             for (int z = 0; z < n; ++z)
9                 temp += A[i * n + z] * B[z * k + j];
10            C[i * k + j] = temp;
11        }
12    }
13 }
```

---

将以上函数的第一层循环采用 `parallel for` 指令进行并行，调度方式使用缺省调度，并行后的代码如下

---

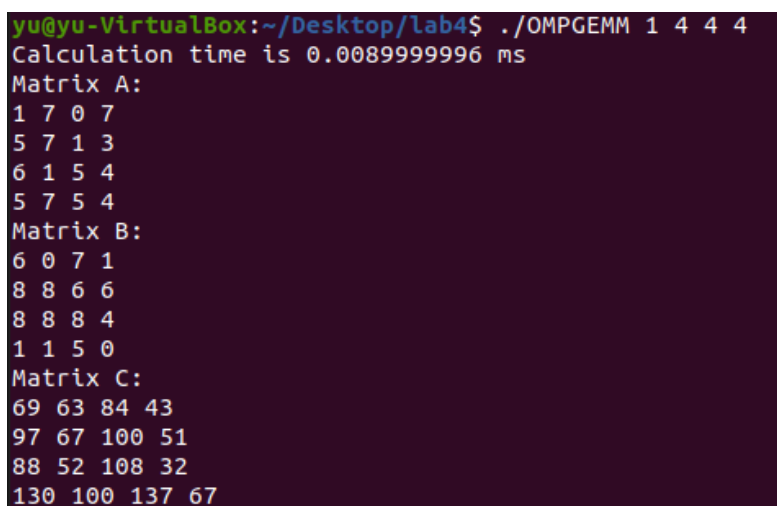
```
1 void MatrixMul(int *A, int *B, int *C, int m, int n, int k)
2 {
3     //缺省调度
4     # pragma omp parallel for num_threads(thread_count)
```

---

```
5     for (int i = 0; i < m; ++i)
6     {
7         for (int j = 0; j < k; ++j)
8         {
9             int temp = 0;
10            for (int z = 0; z < n; ++z)
11                temp += A[i * n + z] * B[z * k + j];
12            C[i * k + j] = temp;
13        }
14    }
15 }
```

主线程时间计算，矩阵打印代码实现与之前 lab3 类似。

为验证正确性，使用小的矩阵维度进行矩阵的打印，当矩阵规模大时，为了方便调试，将矩阵打印的取消，运行结果如图 1，则运算正确。调节线程数与矩阵规模进行输入，其运



```
yu@yu-VirtualBox:~/Desktop/lab4$ ./OMPGEMM 1 4 4 4
Calculation time is 0.0089999996 ms
Matrix A:
1 7 0 7
5 7 1 3
6 1 5 4
5 7 5 4
Matrix B:
6 0 7 1
8 8 6 6
8 8 8 4
1 1 5 0
Matrix C:
69 63 84 43
97 67 100 51
88 52 108 32
130 100 137 67
```

图 1: 4\*4 矩阵运算结果

算时间结果（单位：ms）如表 1，具体分析见实验结果

## 2.2 基于 OpenMP 的通用矩阵乘法优化

简单地将 parallel for 指令进行修改，实现静态调度 schedule(static, 1) 和动态调度 schedule(dynamic,1)，实现代码分别如下

表 1: 运算时间与线程数与矩阵规模的关系

线程数 \ 矩阵规模	512*512*512	1024*1024*1024	2048*2048*2048
1	745.58	8574.94	102651.67
2	328.74	4582.50	59908.74
4	311.78	3480.11	44266.90
6	230.28	3043.68	43468.89
8	244.95	3120.85	46069.71

---

```

1 void MatrixMul(int *A, int *B, int *C, int m, int n, int k)
2 {
3     // 静态调度
4     # pragma omp parallel for num_threads(thread_count)\
5       schedule(static,1)
6     for (int i = 0; i < m; ++i)
7     {
8         for (int j = 0; j < k; ++j)
9         {
10             int temp = 0;
11             for (int z = 0; z < n; ++z)
12                 temp += A[i * n + z] * B[z * k + j];
13             C[i * k + j] = temp;
14         }
15     }
16 }

```

---

```

1 void MatrixMul(int *A, int *B, int *C, int m, int n, int k)
2 {
3     //动态调度
4     # pragma omp parallel for num_threads(thread_count)\
5       schedule(dynamic,1)
6     for (int i = 0; i < m; ++i)
7     {
8         for (int j = 0; j < k; ++j)

```

---

```

9      {
10         int temp = 0;
11         for (int z = 0; z < n; ++z)
12             temp += A[i * n + z] * B[z * k + j];
13         C[i * k + j] = temp;
14     }
15 }
16 }

```

调节线程数与矩阵规模进行输入，**静态调度**运算时间结果（单位：ms）如表 2，具体分析见实验结果

表 2: **静态调度**运算时间与线程数与矩阵规模的关系

线程数 \ 矩阵规模	512*512*512	1024*1024*1024	2048*2048*2048
1	536.80	8933.34	91806.23
2	263.54	3195.48	56125.66
4	138.28	1528.65	34590.65
6	150.64	2111.69	36286.58
8	144.27	2036.26	35296.63

调节线程数与矩阵规模进行输入，**动态调度**运算时间结果（单位：ms）如表 3，具体分析见实验结果

表 3: **动态调度**运算时间与线程数与矩阵规模的关系

线程数 \ 矩阵规模	512*512*512	1024*1024*1024	2048*2048*2048
1	510.42	7948.09	91906.23
2	261.74	3564.91	53544.74
4	130.99	1711.37	32390.79
6	171.49	2327.36	32808.36
8	174.19	2225.35	33792.03

## 2.3 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制。

### 2.3.1 parallel\_for 函数实现

例子中采用 `for_index_arg` 结构体传入每个划分循环块的开始、结束与递增量，考虑到 `pthread_create` 函数需要传入并行函数的参数，因此在结构体中增加 `void` 型指针来为并行函数传参。以给的例子为例，`functor` 函数中有 `A`、`B`、`C`、`x` 四个使用到的变量，当不采用全局变量时需要通过 `args` 进行参数的传递，因此需要结构体保存 `A`、`B`、`C`、`x` 的内容，该结构体指针作为 `for_index_arg` 中的 `void` 型指针进行传递，以实现参数从 `parallel_for` 函数传递给 `pthread_create` 函数，结构体的定义代码如下

---

```
1 struct for_index_arg
2 {
3     int start;        //划分后循环的开始
4     int end;          //划分后循环的结束
5     int increment;    //循环递增量
6     void *args;       //循环函数参数
7 };
8 // void 型指针指向的结构体
9 struct args
10 {
11     int *A;
12     int *B;
13     int *C;
14     int *x;
15     args(int *tA, int *tB, int *tC, int *tx)
16     {
17         A = tA;
18         B = tB;
19         C = tC;
20         x = tx;
21     }
22 };
```

---

对于需要并行的循环，简单地使用与 OpenMP 相同的缺省调度进行划分，相当于 `schedule(static, total_iterations/thread_count)`。当无法整除时，需要最后一个线程进行剩下次数的计算。

```
1 void parallel_for(int start, int end, int crement, void *(*functor)(void
↪ *), void *arg, int num_threads)
2 {
3     // 线程数
4     int thread_count = num_threads;
5     pthread_t *thread_handles = (pthread_t *)malloc(thread_count *
↪ sizeof(pthread_t));
6     // 每个线程对应的 for_index_arg 结构体
7     for_index_arg *for_index_arg_a = (for_index_arg *)malloc(thread_count *
↪ sizeof(for_index_arg));
8     // 每个线程分到的循环次数
9     int block = (end - start) / thread_count;
10    // 对每个循环的 for_index_arg 结构体赋值
11    for (int thread = 0; thread < thread_count; thread++)
12    {
13        for_index_arg_a[thread].args = arg;
14        for_index_arg_a[thread].start = start + thread * block;
15        for_index_arg_a[thread].end = for_index_arg_a[thread].start +
↪ block;
16        // 对于最后一个线程 需要保证循环被完整划分
17        if (thread == (thread_count - 1))
18            for_index_arg_a[thread].end = end;
19        for_index_arg_a[thread].increment = crement;
20        pthread_create(&thread_handles[thread], NULL, functor, (void
↪ *) (for_index_arg_a + thread));
21    }
22    for (int thread = 0; thread < thread_count; thread++)
23        pthread_join(thread_handles[thread], NULL);
24    free(thread_handles);
25    free(for_index_arg_a);
26    return;
27 }
```

修改例子实现 functor 函数的并行，使用 void 型指针将所需要的 A、B、C、x 四个变量传入，实现代码如下

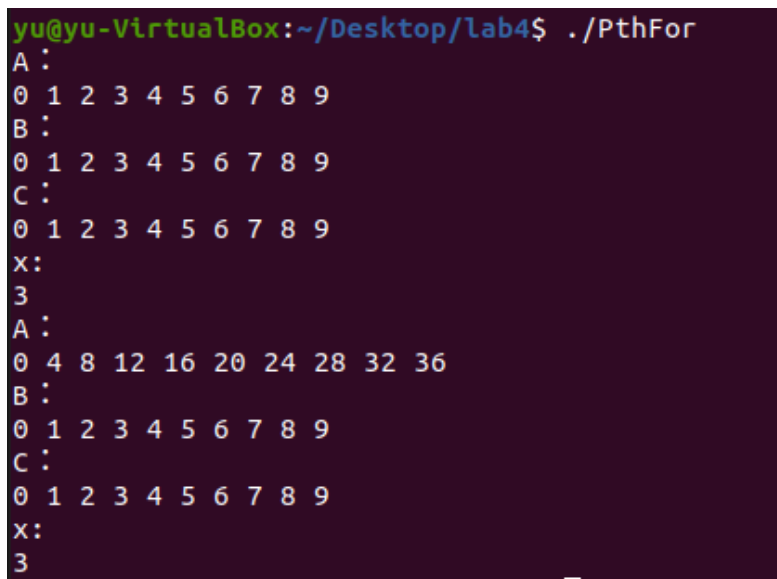


---

```
1 void *functor(void *arg)
2 {
3     struct for_index_arg *index = (struct for_index_arg *)arg;
4     struct args *true_arg = (struct args *)(index->args);
5     for (int i = index->start; i < index->end; i = i + index->increment)
6     {
7         (true_arg->A)[i] = (true_arg->B)[i] * (*(true_arg->x)) +
            ↪ (true_arg->C)[i];
8     }
9     return NULL;
10 }
```

---

程序编译运行，对 A、B、C、x 进行简单的初始化，再使用 parallel\_for 进行并行，运行结果如图 2，可见正确运行



```
yu@yu-VirtualBox:~/Desktop/lab4$ ./PthFor
A :
0 1 2 3 4 5 6 7 8 9
B :
0 1 2 3 4 5 6 7 8 9
C :
0 1 2 3 4 5 6 7 8 9
x:
3
A :
0 4 8 12 16 20 24 28 32 36
B :
0 1 2 3 4 5 6 7 8 9
C :
0 1 2 3 4 5 6 7 8 9
x:
3
```

图 2: 例子运行结果

### 2.3.2 parallel\_for 编译成.so 文件

本次实现与 lab2 中实现方式相同，将 parallel\_for 函数改写为 parallel\_forLib.cpp 源文件与 parallel\_forLib.h 头文件，代码见提交文件，此处省略，再进行编译生成.so 文件。使用的编译命令如下：

```
g++ -ggdb -Wall -shared -fpic -o libPF.so ParallelForLib.cpp
```

为了能让其他程序调用函数，需要将生成的.so 文件到 ld 的链接库当中，此处省略过程。

### 2.3.3 parallel\_for 编译成.so 文件

与例子的实现方式类似，对 args 结构体进行修改，在 GEMM 中我们需要使用到矩阵 A、矩阵 B、矩阵 C 与传入的矩阵规模参数 m,n,k, 结构体定义代码如下：

---

```
1 struct args
2 {
3     int *A;
4     int *B;
5     int *C;
6     int *m;
7     int *n;
8     int *k;
9     args(int *tA, int *tB, int *tC, int *tm, int *tn, int *tk)
10    {
11        A = tA;
12        B = tB;
13        C = tC;
14        m = tm;
15        n = tn;
16        k = tk;
17    }
18 };
```

---

按照 functor 函数的实现形式，对 MatrixMul 函数进行修改，则最终矩阵乘法根据行进行划分，由于无 race condition、无数据依赖、无循环依赖，因此可以使用实现的 parallel\_for 函数，并行化后的 GEMM 代码如下

---

```
1 void *MatrixMul(void *arg)
2 {
3     struct for_index_arg *index = (struct for_index_arg *)arg;
4     struct args *true_arg = (struct args *)(index->args);
5     // 按行进行划分
6     for (int i = index->start; i < index->end; i = i + index->increment)
```

---

```
7     {  
8         for (int j = 0; j < *true_arg->k; ++j)  
9         {  
10            int temp = 0;  
11            for (int z = 0; z < *true_arg->n; ++z)  
12                temp += true_arg->A[i * (*true_arg->n) + z] * true_arg->B[z  
13                    ↪ * (*true_arg->k) + j];  
14            true_arg->C[i * (*true_arg->k) + j] = temp;  
15        }  
16    }  
17    return NULL;  
18 }
```

主线程时间计算，矩阵打印代码实现与之前 lab3 类似。

为验证正确性，使用小的矩阵维度进行矩阵的打印，当矩阵规模大时，为了方便调试，将矩阵打印的取消，运行结果如图 3，则运算正确。调节线程数与矩阵规模进行输入，其运

```
yu@yu-VirtualBox:~/Desktop/lab4$ ./PFGEMM 4 4 4 4  
Calculation time is 0.5410000086 ms  
Matrix A:  
1 7 0 7  
5 7 1 3  
6 1 5 4  
5 7 5 4  
Matrix B:  
6 0 7 1  
8 8 6 6  
8 8 8 4  
1 1 5 0  
Matrix C:  
69 63 84 43  
97 67 100 51  
88 52 108 32  
130 100 137 67
```

图 3: 4\*4 矩阵运算结果

算时间结果（单位：ms）如表 4，可

表 4: 运算时间与线程数与矩阵规模的关系

线程数 \ 矩阵规模	512*512*512	1024*1024*1024	2048*2048*2048
1	1021.72	14651.25	164830.31
2	569.51	6385.73	86179.96
4	3251.46	3247.17	55876.62
6	285.74	4157.53	56529.86
8	245.37	3416.22	53598.48

### 3 实验结果

#### 3.1 通过 OpenMP 实现通用矩阵乘法

OpenMP 实现矩阵乘法的运算时间与矩阵的规模关系如图 4，运算时间随着矩阵规模大致成线性增长，而当矩阵规模达到 2048 左右时，运算时间增长了几倍。这是由于矩阵规模增大超过缓存界限，使得多线程在同时进行计算时，L3 缓存发生了大量 miss，增加了大量的矩阵读取时间，影响了运行效率。由于采用的虚拟机环境为 4 核 4 线程，因此在线程数大于 4 时，并不能充分地进行并行，而频繁的线程切换花费了时间，提高了运算时间。

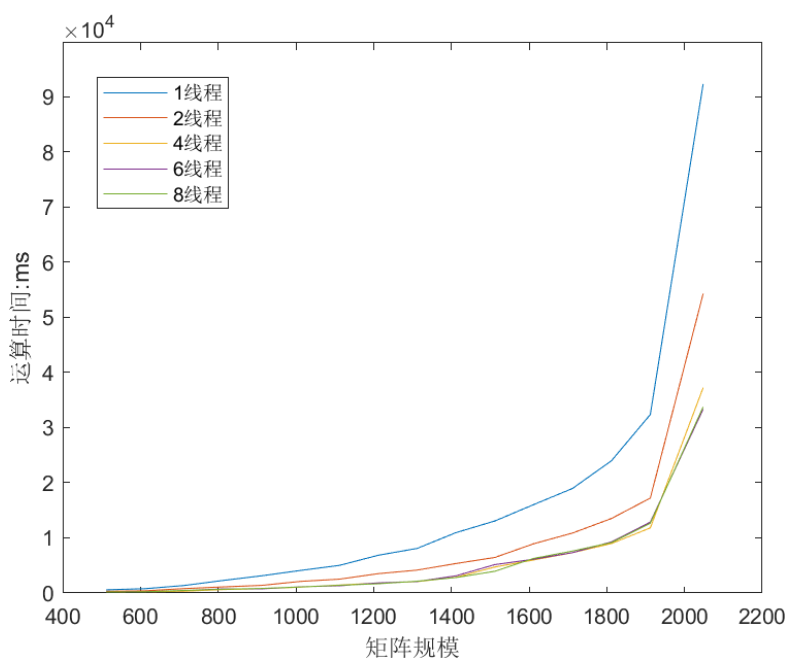


图 4: 矩阵乘法的运算时间与矩阵的规模关系

OpenMP 实现矩阵乘法的加速比如图 5，在矩阵规模为  $512 * 512 * 512$  时，近似达到了线性加速比。而当矩阵规模增大，进程数的增加会提高缓存的 miss，降低了运算效率，因此加速比增长缓慢。由于计算的上限仍为 4 进程，在进程数大于 4 时出现的加速比下降现象原因可能是进程之间的频繁切换花费了时间。

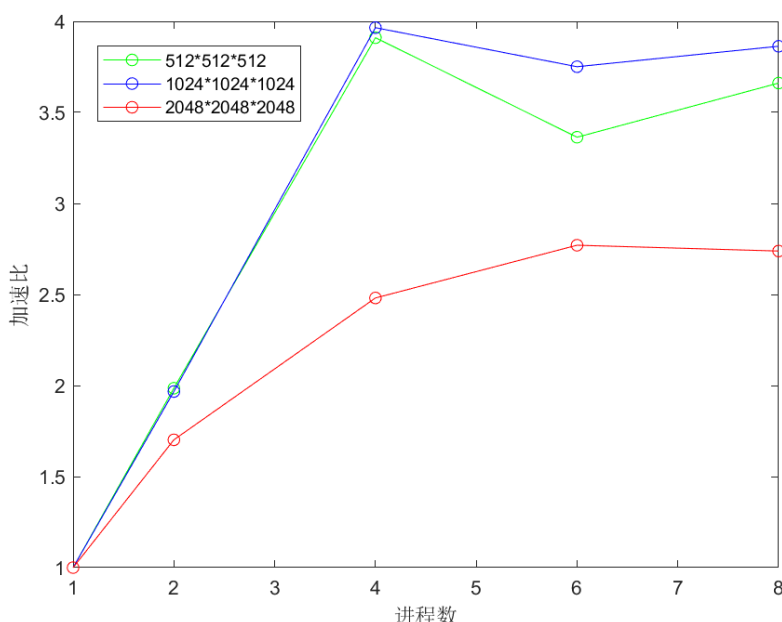


图 5: 矩阵乘法的加速比

### 3.2 基于 OpenMP 的通用矩阵乘法优化

为比较 3 种调度策略，选择矩阵规模为  $512 * 512 * 512$  与  $1024 * 1024 * 1024$  两种矩阵分别采用不同的调度策略进行计算，得到运算时间与调度策略的关系如图 6，可知在矩阵规模较小，为  $512 * 512 * 512$  时，缺省调度与静态调度的运算时间大致相同，而动态调度稍差；在矩阵规模增加到  $1024 * 1024 * 1024$  时，缺省调度的运算时间较短，而静态调度与动态调度的效果较差。综合比较，在缺省调度时的并行加速效果较好，而采取的矩阵规模数较少，实验存在一定偶然性。

### 3.3 构造基于 Pthreads 的并行 for 循环分解、分配和执行机制。

调节线程数与矩阵规模进行输入，使用 `parallel_for` 函数并行矩阵运算时间结果（单位：ms）如图 7。与之前实现的矩阵运算时间比较，可见其并行效果并不好，主要原因在于传入参数采用的是指针，因此在访问参数的时候进行了指针的寻址，降低了运算时间。考虑对内存进行优化，因该将变量重新保存，减少指针寻址的次数以减少访存次数。

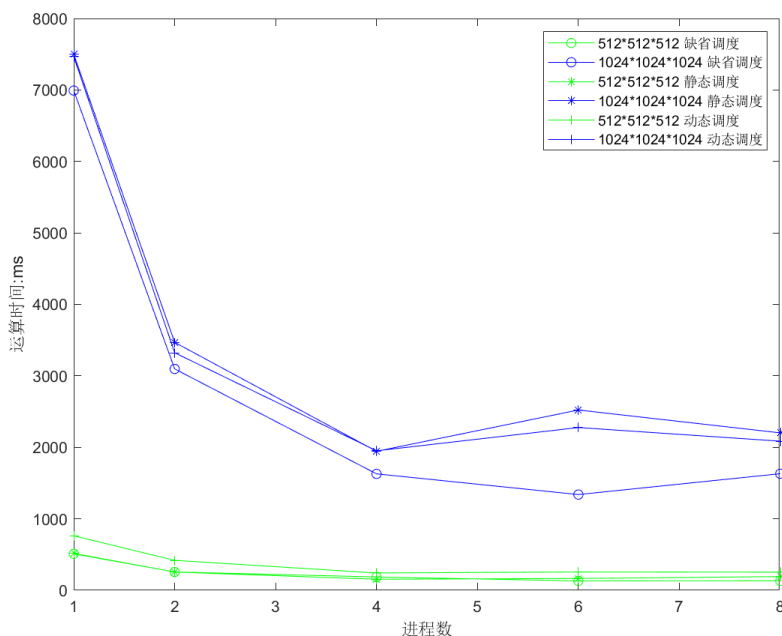


图 6: 调度策略比较

调节线程数与矩阵规模进行输入，使用 `parallel_for` 函数并行加速比结果如图 8。由于计算的上限为 4 进程，在进程数大于 4 时出现的加速比下降现象原因可能是进程之间的频繁切换花费了时间。根据加速比可知并行效果较好，与其余方法实现的并行效果类似，主要原因仍是指针寻址，在进行内存优化后可预想到其加速效果的提高。

## 4 实验感想

- 本次实验来看，基于 OpenMP 的通用矩阵乘法与之前的矩阵乘法相同，线程在运行过程中会共享相同的缓存，多线程在运行过程中因为时序的问题，导致内存优化更加困难，但为了不出现像实验中矩阵规模在 2000 左右出现的性能锐减，保证多线程的高效工作，多线程需要更加完善的内存优化策略。
- 本次实验比较了 3 种调度策略，由于实验存在一定偶然性，缺省调度不一定为最优调度，而最优调度是由线程的个数和矩阵的规模共同决定的，因此在每一次更改矩阵规模与线程时，我们都应该采用大量的试验以得到在当前条件下最优的调度策略。根据课本，当循环的每次迭代需要几乎相同的计算量，那默认的调度方式能提供最好的性能，也符合本次实验的结果。
- 本次实验由于考虑欠缺，导致 `parallel_for` 函数的并行计算效果较差，没有充分利用程序的局部性原理，代码还有改进空间，这是之后在写代码的过程中需要考虑的。

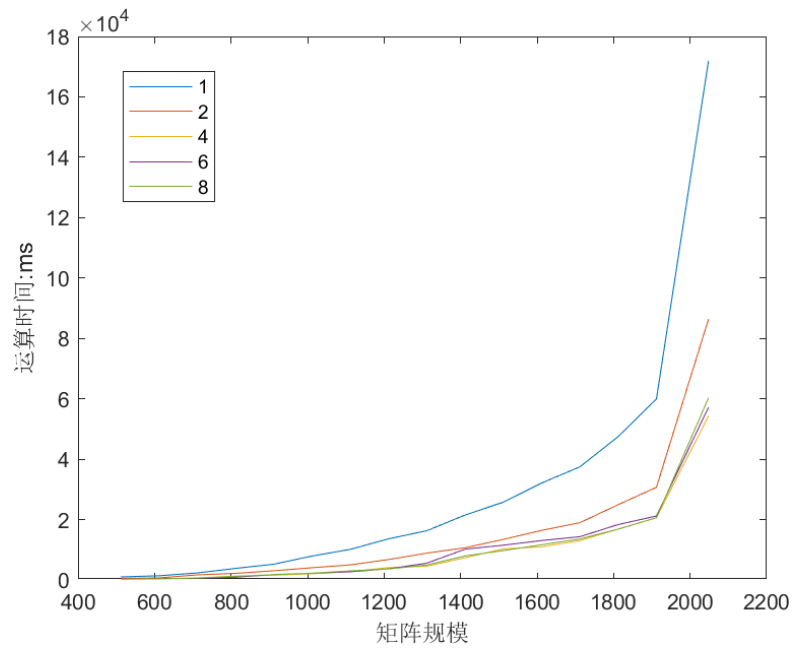


图 7: 矩阵乘法的运算时间与矩阵的规模关系

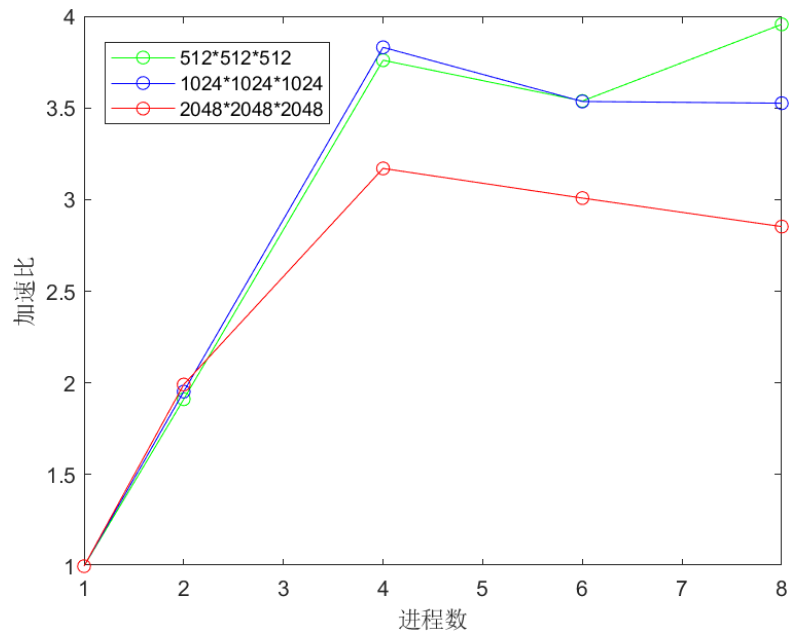


图 8: 矩阵乘法的加速比