

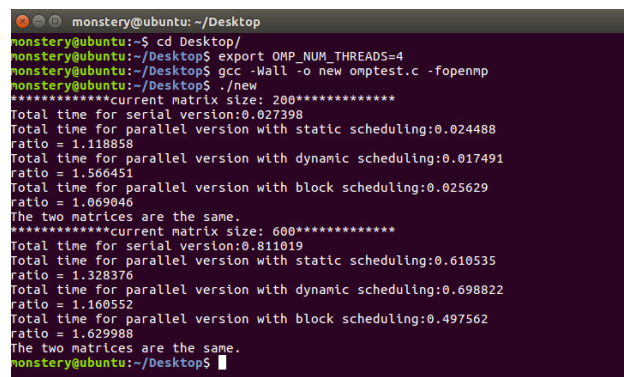
## Project1 实验报告

### 1. 并行思路

计算矩阵乘法  $C_{m \times k} = A_{m \times n} \times B_{n \times k}$  时, 严格按照公式我们可以知道对应  $A_{m \times n}$  中元素  $a_{i,j}$  和  $B_{n \times k}$  中元素  $b_{i,j}$ , 我们可以按照公式  $c_{i,j} = \sum_{q=1}^n a_{i,q} \cdot b_{q,j}$ , 进行计算, 所以内外一共会进行三轮的循环, 其中最内层循环用来计算  $c_{i,j}$ , 外面两层循环用来对  $C$  中的  $m \times k$  个元素分别进行循环。所以可以利用 OpenMP 中 `pragma omp for` 循环的编译命令进行并行化, 我们把大致可以得到下面的程序并行结构 (为方便起见, 我们实现的全部是方阵相乘, 所以输入时直接输入一个规模参数即可)。对于编译命令 `pragma omp for` 来说, 当不选择调度方式时默认为 `static`, 我们也可以选择修改命令为 `pragma omp for schedule(dynamic)`, 使得线程之间的调度是动态的, 理论上可以优化循环的效率。但是这种并行思路下, 当不同线程需要读取同一列或一行数组时, 因为 OpenMP 本质上是共享内存式的线程通信, 在矩阵规模较大的前提下可能存在 cache 频繁访问失效, 导致虽然外部已经并行化, 但是由于频繁访问造成的缓存失效导致的时间上的浪费却在增加, 可能造成最终并行效果并不如意。针对这种情况, 我们提出了进一步的并行思路, 即将矩阵先进行分块, 这样的话当每次取数组中的值时, 各个线程只需要到自己的内存范围内去找对应的值, 这样就提高了 cache 的命中率, 从而优化了并行的效果。具体实现即是依据 `omp_thread` 的数量定义划块的顺序, 在计算函数中利用 `omp_get_num_threads()` 来作为划分的标准 (具体实现见代码)。

### 2. 运行截图

程序的运行截图如下 (为演示方便所以这里只显示了当矩阵规模较小时的截图):



```

monstery@ubuntu: ~/Desktop
monstery@ubuntu:~$ cd Desktop/
monstery@ubuntu:~/Desktop$ export OMP_NUM_THREADS=4
monstery@ubuntu:~/Desktop$ gcc -Wall -o new ompstest.c -fopenmp
monstery@ubuntu:~/Desktop$ ./new
*****current matrix size: 200*****
Total time for serial version:0.027398
Total time for parallel version with static scheduling:0.024488
ratio = 1.118858
Total time for parallel version with dynamic scheduling:0.017491
ratio = 1.566451
Total time for parallel version with block scheduling:0.025629
ratio = 1.069046
The two matrices are the same.
*****current matrix size: 600*****
Total time for serial version:0.811019
Total time for parallel version with static scheduling:0.610535
ratio = 1.328376
Total time for parallel version with dynamic scheduling:0.698822
ratio = 1.160552
Total time for parallel version with block scheduling:0.497562
ratio = 1.629988
The two matrices are the same.
monstery@ubuntu:~/Desktop$
  
```

图 1: 运行过程截图

### 3. 结果分析

首先，我们在 `OMP_NUM_THREADS=4` 的条件下在不同矩阵大小下进行了实验，得到的运行时间和加速比的结果如图 2a 所示。

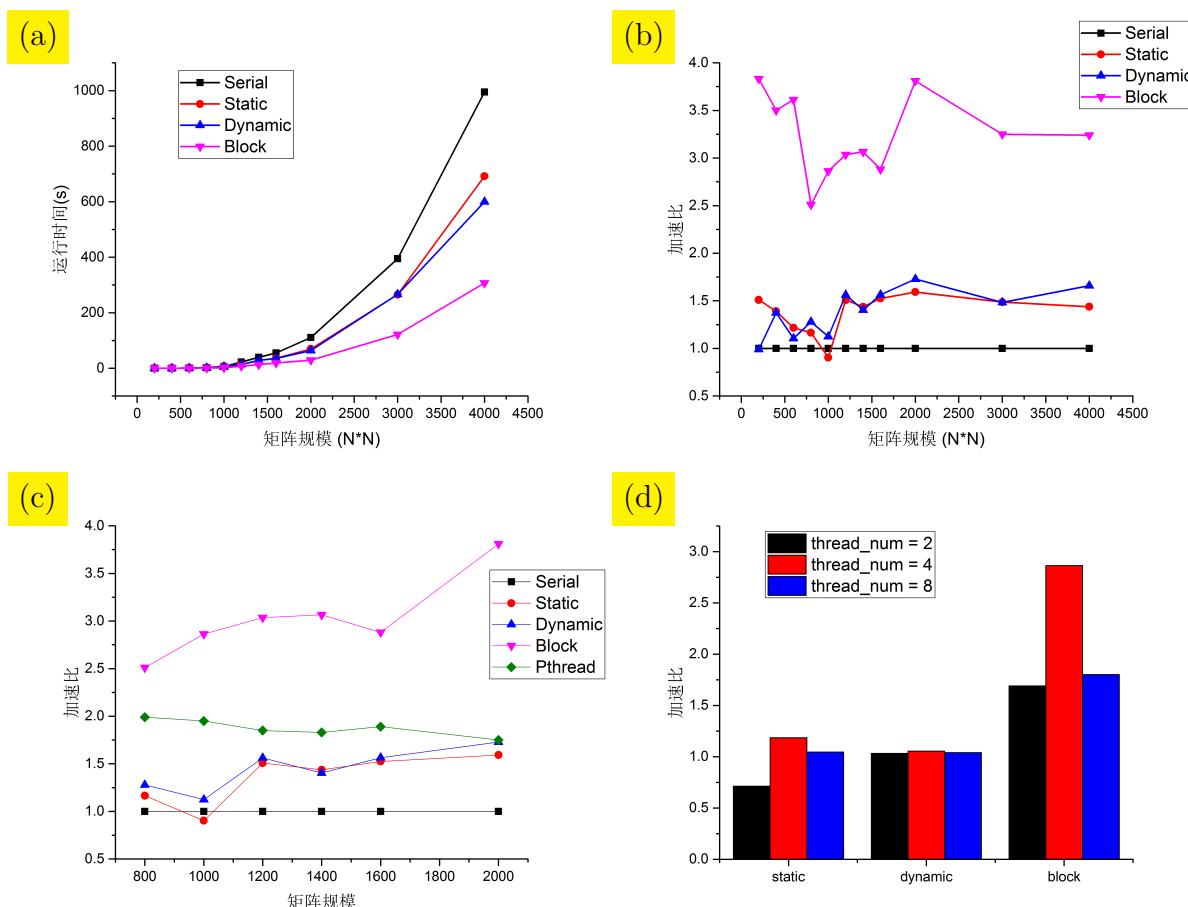


图 2: 运行结果

首先从图 2a 可以发现，随着矩阵规模的扩大，计算的时间迅速增长（接近 3 次方的增长速度），并且在三种并行化思路中，经过 cache 优化的 OpenMP 算法明显在时间上短于其他并行思路。同样从图 2b 可以看出，在矩阵规模较小时，简单并行的两种策略的加速比并不明显，有时甚至会意外地小于一，这应该是矩阵较小时多线程之间存在调度，导致计算开销不变的情况下总时间开销增加。意外地，我们可以看到 cache 优化的并行思路在一开始时加速比接近 4，但是当矩阵规模到 1000 附近后加速比突然掉到 2.5 左右，最后又重新上升到 3.5+，最后基本稳定在 3.2。我们分析原因是当规模较小时，cache 优化使得总的计算时间大大下降，而随着规模增加，cache 中一个 block 的大小已经小于了分给一个线程的数据量，导致加速比有所下降。随着规模进一步增加，并行加速的优势逐渐显现，导致加速比重新提升。最后，当规模已经到 3000 以上时，加速比受到 cache 失效和本身线程间通信的限制导致加速比无法进一步提升甚至有所下降。所以总的来说，我们认为加速比

本身受到并行思路，cache 大小和操作系统自身线程调度的影响。

接下来，我们又实现了 Pthread 和不同线程数下加速比的对比，结果如图 2c, 2d 所示。首先可以看出，在不考虑 cache 优化的情况下，pthread 的并行效果是要优于 OpenMP 的；另外，由于我们的实验平台是 2 核 4 线程的 i5 处理器，所以可以发现在提高线程数并不会增加加速比，反而由于线程调度的开销使得加速比下降。当然，线程从 2 加到 4 我们可以看出是有显著的效果的。

利用 Pthreads 实现矩阵乘法的关键是将求解结果矩阵  $res[M][M]$  所有元素的任务划分为  $number$ (线程数) 个互斥的子任务。可以采取按行划分的方法，参见 Figure 3。简而言之就是利用了一个结构体 `split` 来计算当前线程用于计算的起始行和用于计算的行之间的间距，，用于划分不同线程计算的区域 (`startrow` 可看作线程的编号，`gap` 即为用户定义的线程数)。

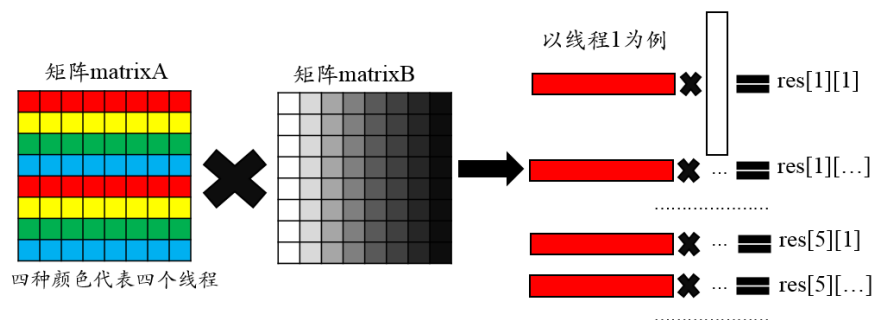


图 3: Pthread 原理图