

## Phread 编程

姓名：郁万祥 学号：2013852

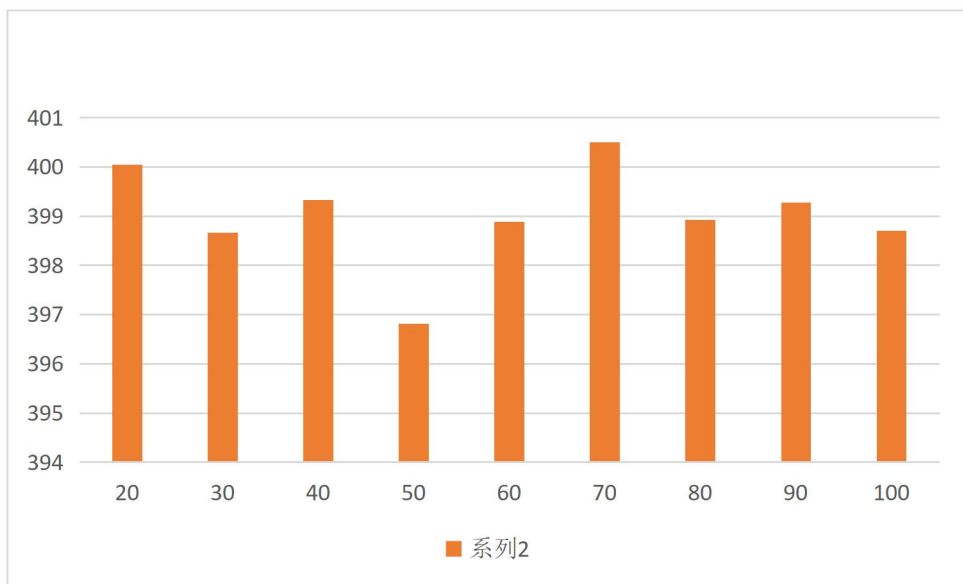
1、对于课件中“多个数组排序”的任务不均衡案例进行复现（规模可自己调整），并探索较优的方案。提示：可从任务分块的大小、线程数的多少、静态动态多线程结合等方面进行尝试，探索规律。

### 1.1 从任务分块的大小入手：

我们保持线程数为 4 不变，将任务块从 10 到 100 每次增加 10，观察运行效率，下面是实验结果：

```
C:\Workspace\CodeBlocksProjects\Pthread_arrays_sort\bin\Debug\Pthread_arrays_sort.exe
seg10
time:748.526ms
seg20
time:400.046ms
seg30
time:398.665ms
seg40
time:399.32ms
seg50
time:396.811ms
seg60
time:398.878ms
seg70
time:400.504ms
seg80
time:398.918ms
seg90
time:399.271ms
seg100
time:398.702ms

Process returned 0 (0x0)   execution time : 4.451 s
Press any key to continue.
```



可能是由于数据加载缓冲等影响，第一次的时间显然更长一些，出去第一次的结果，我们发现，**seg** 的改变对并行效率影响并不大，在本次问题当中，**seg=50** 时表现较好一些。

### 1.2 从线程数目少入手：

根据上述实验，我们保持 **seg=50** 不变，实验线程数目设置为单线程到 8 个线程，每个线程数目下运行 10 次，取平均程序运行总时间代价，下面是实验结果：


```
C:\Workspace\CodeBlocksProjects\Pthread_arrays_sort\bin\Debug\Pthread_arrays_sort.ex
Thread 0: 11652.897600ms.
Process returned 0 (0x0)   execution time : 12.552 s
Press any key to continue.

C:\Workspace\CodeBlocksProjects\Pthread_arrays_sort\bin\Debug\Pthre
Thread 1: 6497.298100ms.
Thread 0: 6741.449600ms.
Process returned 0 (0x0)   execution time : 7.657 s
Press any key to continue.
```

 C:\Workspace\CodeBlocksProjects\Pthread\_arrays\_sort\bin\Debug\Pthr

```
Thread 2: 5014.571500ms.  
Thread 1: 5311.119800ms.  
Thread 0: 5610.006400ms.
```

```
Process returned 0 (0x0)    execution time : 6.427 s  
Press any key to continue.
```

 C:\Workspace\CodeBlocksProjects\Pthread\_arrays\_sort\bin\Debug\Pthread\_arrays\_s

```
Thread 0: 3655.725800ms.  
Thread 1: 3659.866600ms.  
Thread 3: 3663.018400ms.  
Thread 2: 3666.434000ms.
```

```
Process returned 0 (0x0)    execution time : 4.584 s  
Press any key to continue.
```

 C:\Workspace\CodeBlocksProjects\Pthread\_arrays\_sort\bin\Debug\Pth

```
Thread 3: 2725.267700ms.  
Thread 4: 2758.337500ms.  
Thread 2: 3098.629900ms.  
Thread 1: 3194.432800ms.  
Thread 0: 3469.351100ms.
```

```
Process returned 0 (0x0)    execution time : 4.371 s  
Press any key to continue.
```

 C:\Workspace\CodeBlocksProjects\Pthread\_arrays\_sort\bin\Debug\Pth

```
Thread 3: 2279.704700ms.  
Thread 5: 2286.139800ms.  
Thread 4: 2333.781500ms.  
Thread 2: 2593.887700ms.  
Thread 1: 2686.763700ms.  
Thread 0: 2876.463200ms.
```

```
Process returned 0 (0x0)    execution time : 3.778 s  
Press any key to continue.
```

```
C:\Workspace\CodeBlocksProjects\Pthread_arrays_sort\bin\Debug\Ptl
Thread 5: 1987.385800ms.
Thread 6: 2001.799300ms.
Thread 3: 2031.067200ms.
Thread 4: 2050.376400ms.
Thread 2: 2437.565900ms.
Thread 1: 2441.124400ms.
Thread 0: 2606.752300ms.

Process returned 0 (0x0)    execution time : 3.504 s
Press any key to continue.
```

```
C:\Workspace\CodeBlocksProjects\Pthread_arrays_sort\bin\Debug\Pthread_arr
Thread 3: 1717.854800ms.
Thread 4: 1721.823500ms.
Thread 7: 1724.471000ms.
Thread 5: 1727.027300ms.
Thread 6: 1740.401500ms.
Thread 2: 1926.420200ms.
Thread 1: 2016.938100ms.
Thread 0: 2108.014900ms.

Process returned 0 (0x0)    execution time : 3.009 s
Press any key to continue.
```

结果分析：

经过计算，我们发现随着线程数目增加，所消耗的总时间，反而会增加，这可能是由于创建线程的本身比较费时，同时静态划分的方式，并不适合创建多线程，所以效率并不高。

### 1.3 从静态动态多线程结合入手：

对于多线程，反而效率不高的问题，我们改进动态任务划分的方式，进行多线程的操作，下面是实验结果：

```
C:\Workspace\CodeBlocksProjects\Pthread_arrays_sort\bin\Debug\Pthread_arrays_sc
Thread 0: 11273.545500ms.
Process returned 0 (0x0)   execution time : 12.012 s
Press any key to continue.

C:\Workspace\CodeBlocksProjects\Pthread_arrays_sort\bin\Debug\Pthre
Thread 3: 2037.236100ms.
Thread 2: 2303.106100ms.
Thread 1: 2454.448500ms.
Thread 0: 2562.146000ms.
Process returned 0 (0x0)   execution time : 3.126 s
Press any key to continue.
```

结果分析：

我们发现通过动态划分任务的方式，四个线程的效率会比单线程的效率。因此，最后发现，在目前的环境下，设置 `seg` 为 50，线程数为 4，动态任务划分的方式是效率比较高的方式。

2、实现高斯消去法解线性方程组的 Pthread 多线程编程，可与 SSE/AVX 编程结合，并探索优化任务分配方法。

实验过程：

首先，我们通过第一个实验发现，其实对于较小的矩阵规模，多线程的效果并不明显，会导致，无论任务划分如何，效率都还不错，因此对于任务划分的探索也比较局限，所以在进行高斯并行化实验的过程中，增加矩阵整体的大小，同时相应的，对于任务的分配也应该有所增加，因此实验数据如下：

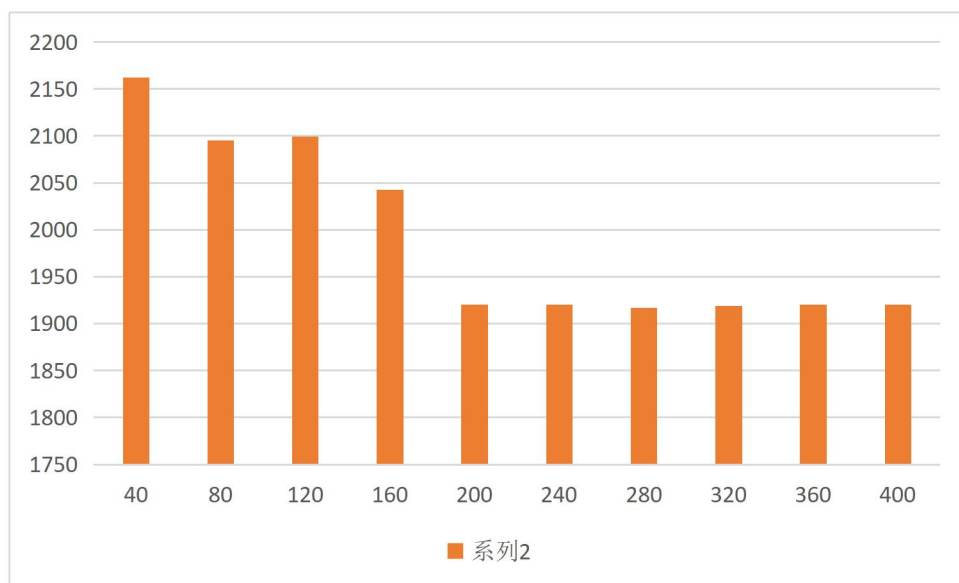
矩阵规模固定为 2048\*2048，线程数固定为 4，对于任务的划分，我们设定 `seg` 从 40 到 400，每次增加 40，来实现任务划分从每次 40 到每次 400 的目的，探索高斯消去中任务划分的优化。

实验结果：

```
C:\Workspace\CodeBlocksProjects\gause_Pthread\bin\Debug\gause_Pthread.exe
seg: 40
time: 2161.999000 ms
seg: 80
time: 2095.000000 ms
seg: 120
time: 2099.132000 ms
seg: 160
time: 2042.394000 ms
seg: 200
time: 1919.989000 ms
seg: 240
time: 1920.116000 ms
seg: 280
time: 1917.020000 ms
seg: 320
time: 1919.111000 ms
seg: 360
time: 1920.491000 ms
seg: 400
time: 1920.146000 ms

Process returned 0 (0x0)   execution time : 20.163 s
Press any key to continue.
```

结果分析：

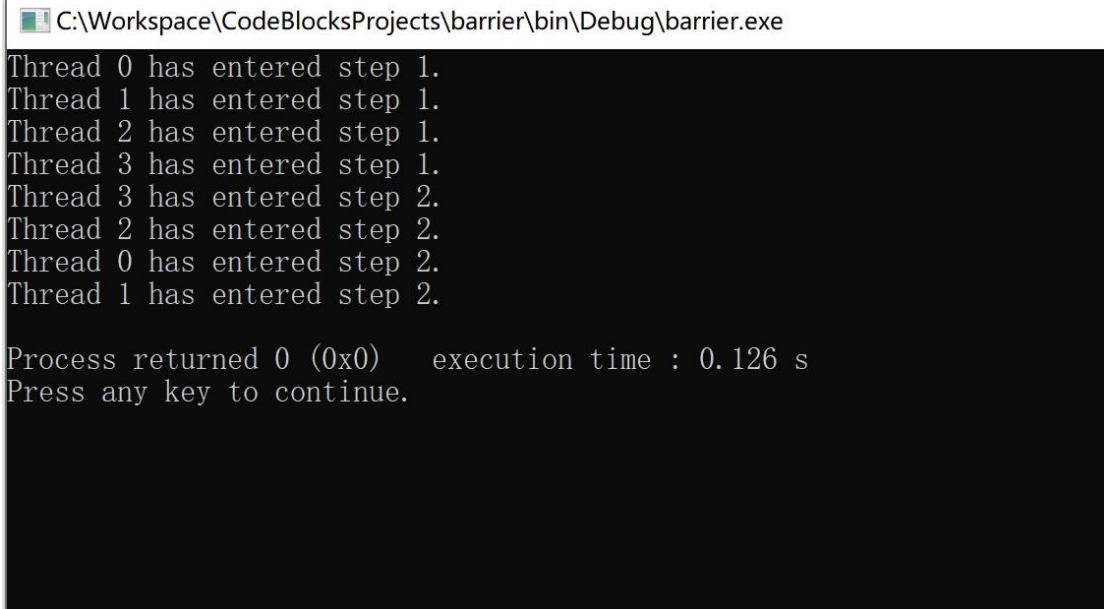


当前情况下，我们发现，当任务划分比较大的时候，运行的效率比较高，其中当任务划分为 280 时，本次实验的效率最高，所以，其实任务的合适划分需要根据问题的规模进行敲定，问题规模越大，任务划分的优化效果更加明显，达到效率最高所需要的任务划分一般也尽量比较大。



**3、附加题：**使用其他方式（如忙等待、互斥量、信号量等），自行实现不少于 2 种路障 Barrier 的功能，分析与 Pthread\_barrier 相关接口功能的异同。提示：可采用课件上路障部分的案例，用其他 2 种方式实现相同功能；也可自行设定场景，实现 2 种或以上 barrier 的功能，并进行效率、功能等方面的展示比较。

实现效果：



```
C:\Workspace\CodeBlocksProjects\barrier\bin\Debug\barrier.exe
Thread 0 has entered step 1.
Thread 1 has entered step 1.
Thread 2 has entered step 1.
Thread 3 has entered step 1.
Thread 3 has entered step 2.
Thread 2 has entered step 2.
Thread 0 has entered step 2.
Thread 1 has entered step 2.

Process returned 0 (0x0)    execution time : 0.126 s
Press any key to continue.
```

### 3.1 使用忙等待和互斥量实现路障

使用一个由互斥量保护的共享计数器。当计数器的值表明每个线程都已经进入临界区，所有线程就可以离开忙等待的状态了。

相同： 1、都是线程在执行一个任务后等待其他线程完成。

2、都是使得所有线程在某个地方同步。

不同：接口使用的是 **wait** 函数，每个线程完成任务主动调用 **wait**，然后等待同步，忙等待使用的是 **while** 循环，使得线程处于忙等待阶段，从而进行同步。

缺点： 1、线程处于忙等待循环时浪费了很多 CPU 周期，并且当程序中的线程数过多时，程序的性能有可能会下降的非常厉害。

2、通过这这种方式实现的路障，有多少个路障就要有多少个不同的共享计数器变量来进行计数，非常繁琐。

### 3.2 信号量实现路障

相同： 1、都是线程在执行一个任务后等待其他线程完成。

2、都是使得所有线程在某个地方同步。

在忙等待实现的路障中，使用了一个计数器 `counter` 来判断有多少线程抵达了路障。在这里，采用了两个信号量：`count_sem`，用于保护计数器；`barrier_sem`，用于阻塞已经进入路障的线程。

线程被阻塞在 `sem_wait` 不会消耗 CPU 周期，所以用信号量实现路障的方法比用忙等待实现的路障性能更佳。

如果想执行第二个路障，`counter` 和 `count_sem` 可以重用，但是重用 `barrier_sem` 会导致竞争条件。