南開大學
Nankai University

《并行程序与设计》实验报告

(2023~2024 **学年第一学期**)

实验名称：Pthread 编程作业

学　　院：软件学院

姓　　名：陈高楠

学　　号：2112966

指导老师：孙永谦

2023 年 11 月 19 日

目录

1 “多个数组排序”的任务不均衡案例 …… 1
1 问题描述 …… 1
2 算法设计与分析 …… 1
2.1 分层分布的数组的初始化函数 …… 1
2.2 静态线程函数 …… 2
2.3 动态线程函数 …… 2
2.4 主线程派发线程 …… 3
3 结果统计与分析 …… 4
3.1 静态线程与动态线程的运行时间差别 …… 4
3.2 动态线程派发任务粒度的影响 …… 4
3.3 线程数的数量的的运行时间差别 …… 5
4 实验结论 …… 6
2 高斯消元 Pthread 实现 …… 7
1 问题描述 …… 7
2 算法设计与分析 …… 7
2.1 高斯消元法介绍 …… 7
2.2 串行算法 …… 8
2.3 普通 Pthread(动态分配线程） …… 9
2.4 Pthread 结合 SSE（动态分配线程） …… 11
2.5 Pthread 结合 AVX（动态分配线程） …… 14
2.6 AVX 结合信号量（静态分配线程） …… 16
2.7 AVX 结合屏障（静态分配线程，静态分配任务） …… 19
2.8 AVX 结合屏障（静态分配线程, 动态分配任务） …… 22
3 结果统计与分析 …… 25
3.1 不同方法运行时间比较 …… 25
3.2 对 Pthread 的改进 …… 26
3.3 不同线程数运行的比较 …… 28

4 分配任务动态和静态的比较 …… 29
5 实验结论 …… 30
3 附加题 …… 31
1 问题描述 …… 31
2 算法设计与分析 …… 31
2.1 路障 Barrier …… 31
2.2 信号量实现路障 Barrier 的功能 …… 32
2.3 忙等待和互斥量实现路障 Barrier 的功能 …… 34
3 结果分析 …… 35
3.1 信号量与 Barrier 的异同 …… 35
3.2 忙等待和互斥量与 Barrier 的异同 …… 35

实验一 “多个数组排序”的任务不均衡案例

1 问题描述

对于课件中“多个数组排序”的任务不均衡案例进行复现，并探索较优的方案。并从任务分块的大小、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。

2 算法设计与分析

2.1 分层分布的数组的初始化函数

当数组分层分布时 (即 1/4 升序，1/4 降序，1/4 一半升序一半降序，1/4 一半降序一半升序)，初始化函数如下：

```
1 void init_2(void)
2 {
3   int ratio;
4   srand(unsigned(time(NULL)));
5   for (int i = 0; i < ARR_NUM; i++) {
6     arr[i].resize(ARR_LEN);
7     if (i < seg) ratio = 0;
8     else if (i < seg * 2) ratio = 32;
9     else if (i < seg * 3) ratio = 64;
10    else ratio = 128;
11    if ((rand() & 127) < ratio)
12      for (int j = 0; j < ARR_LEN; j++)
13        arr[i][j] = ARR_LEN - j;
14    else
15      for (int j = 0; j < ARR_LEN; j++)
16        arr[i][j] = j;
17  }
18 }
```

2.2 静态线程函数

静态线程函数如下:

```
1 void* arr_sort(void* parm)
2 {
3   threadParm_t* p = (threadParm_t*)parm;
4   int r = p->threadId;
5   long long tail;
6   for (int i = r * seg; i < (r + 1) * seg; i++)
7     sort(arr[i].begin(), arr[i].end());
8   pthread_mutex_lock(&mutex);
9   QueryPerformanceCounter((LARGE_INTEGER*)&tail);
10  printf("Thread %d: %lfms.\n", r, (tail - head) * 1000.0 / freq);
11  pthread_mutex_unlock(&mutex);
12  pthread_exit(NULL);
13 }
```

2.3 动态线程函数

动态线程函数如下, 采用互斥访问 next_arr

```
1 void* arr_sort_fine(void* parm)
2 {
3   threadParm_t* p = (threadParm_t*)parm;
4   int r = p->threadId;
5   int task = 0;
6   long long tail;
7   while (1) {
8     pthread_mutex_lock(&mutex_task);
9     task = next_arr++;
10    pthread_mutex_unlock(&mutex_task);
11    if (task ≥ ARR_NUM) break;
```

```
12      stable_sort(arr[task].begin(), arr[task].end());
13    }
14    pthread_mutex_lock(&mutex);
15    QueryPerformanceCounter((LARGE_INTEGER*)&tail);
16    printf("Thread %d: %lfms.\n", r, (tail - head) * 1000.0 / freq);
17    pthread_mutex_unlock(&mutex);
18    pthread_exit(NULL);
19  }
```

2.4 主线程派发线程

主线程函数如下：

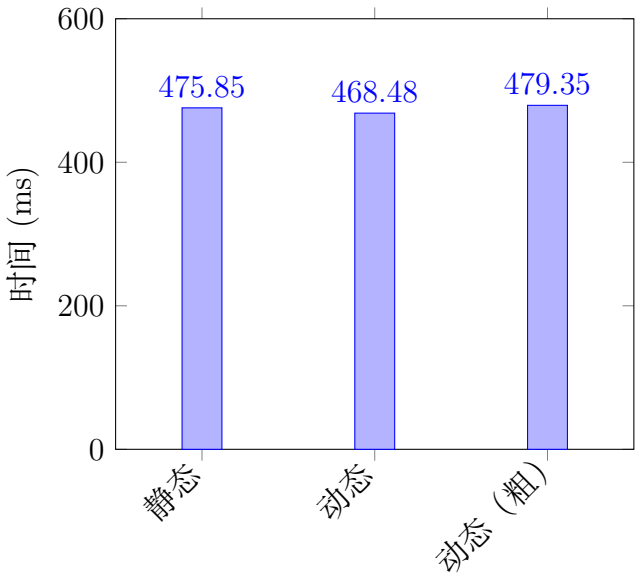
```
1  int current_block=0;
2  void* arr_sort_block(void* parm)
3  {
4    threadParm_t* p = (threadParm_t*)parm;
5    int r = p->threadId;
6    int task = 0;
7    int block2=-1;
8    long long tail;
9    while (1) {
10     pthread_mutex_lock(&mutex_task);
11     block2=-1;
12     if(current_block<TOTAL_BLOCKS)
13             {
14                 block2=current_block;
15             current_block++;}
16     pthread_mutex_unlock(&mutex_task);
17     if (block2==-1) break;
18     int start_task=(block2/(ARR_LEN/BLOCK_SIZE))*BLOCK_SIZE;
19     int start_task_col=(block2%(ARR_LEN/BLOCK_SIZE))*BLOCK_SIZE;
20         int end_task=start_task+BLOCK_SIZE;
21         int end_task_col=start_task_col+BLOCK_SIZE;
22     for (int i = start_task; i < end_task; i++) {
23             stable_sort(arr[i].begin()+start_task_col,arr[i].begin()+...
```

```
                    end_task_col);
24              }
25      }
26      pthread_mutex_lock(&mutex);
27      QueryPerformanceCounter((LARGE_INTEGER*)&tail);
28      printf("Thread %d: %lfs.\n", r, (tail - head)* 1000.0  / freq);
29      pthread_mutex_unlock(&mutex);
30      pthread_exit(NULL);
31
32  }
```

3 结果统计与分析

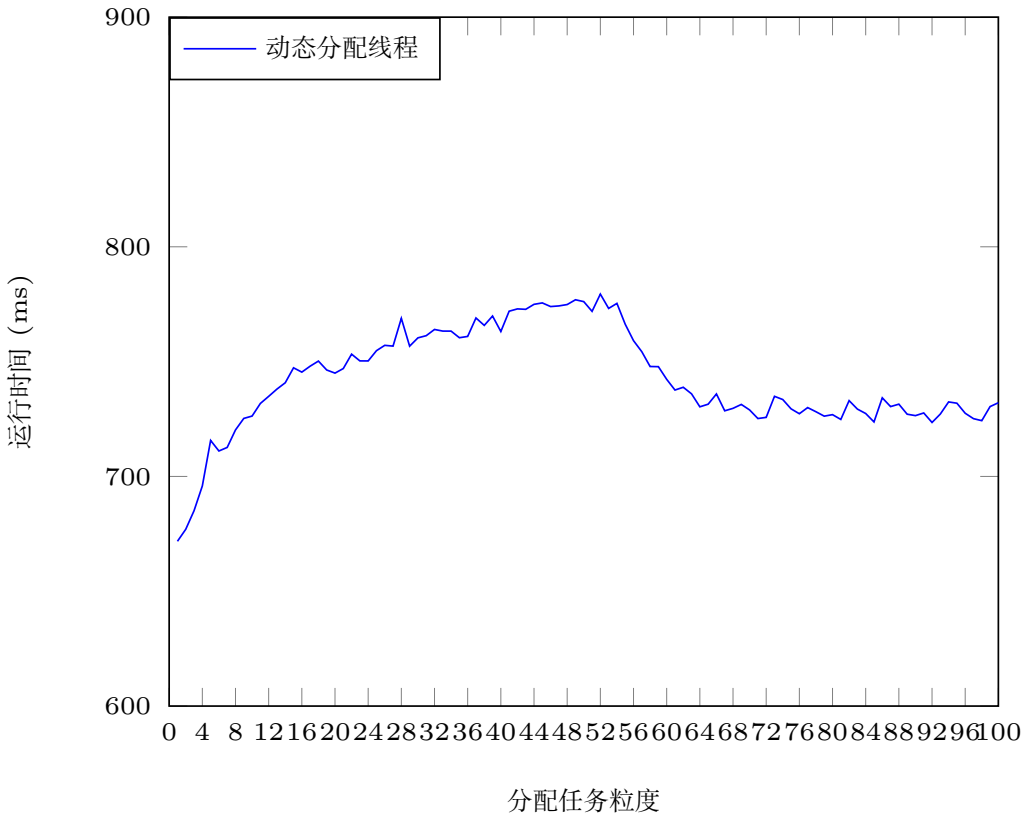
3.1 静态线程与动态线程的运行时间差别

当矩阵是有规律地随机时，(1/4 升序，1/4 降序，1/4 一半升序一半降序，1/4 一半降序一半升序) 不同线程之间存在难度的差距，因此可以体现出动态线程分配的优势。而此时静态线程分配因为总的时间受到分配的任务重的线程的影响，导致总运行时间上升。



3.2 动态线程派发任务粒度的影响

下面是动态线程派发任务不同粒度的运行时间情况曲线图。(采用 8 线程进行运算)



从上面曲线看出，分配任务的粒度大虽然能减少分配任务的次数，但是总的运行时间不一定好。这是因为分配任务大的时候，线程之间运行时间方差会越来越大，会导致总的运行时间被拖累。从上图可知，最好的效果还是每次分配 1 行进行计算。

3.3 线程数的数量的的运行时间差别

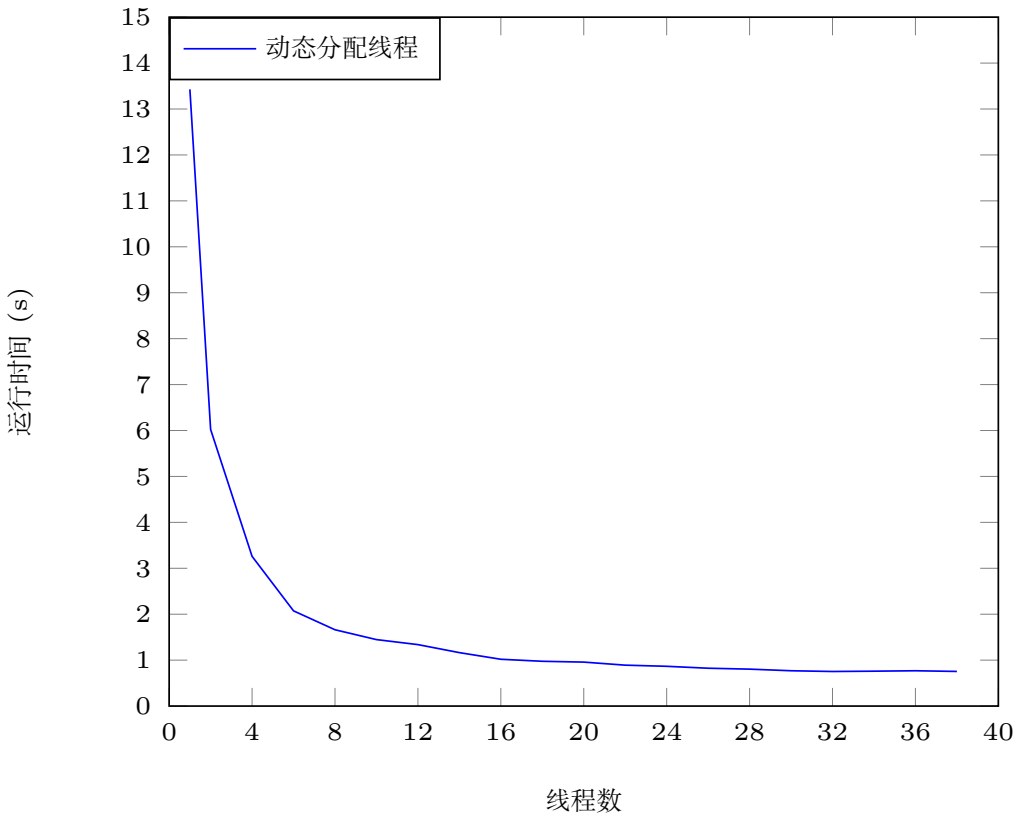
线程数多并行所需时间会缩短，但是线程数并不是越多越好，这跟 CPU 的核是有关系的。下面比较了在矩阵有规律的随机时，采用动态分配线程的运行时间差别。

1	2	4	6	8	10	12	14	16	18
13.4274	6.0228	3.2601	2.0729	1.6616	1.4473	1.3381	1.1636	1.0190	0.9751

表 1.1: 不同线程数的运行时间（一）

20	22	24	26	28	30	32	34	36	38
0.9573	0.8908	0.8658	0.8239	0.8041	0.7690	0.7529	0.7597	0.7688	0.7541

表 1.2: 不同线程数的运行时间（二）



从上图可以看出，一开始增加线程的时候，执行时间会缩短很多，但是后面加速越来越缓，这是因为线程数多了，调度线程也会产生一定的时间。可看出 32 线程是比较好的方案。

4 实验结论

结论:”多个数组排序“任务不均衡时，采取 32 线程，并采用平均分配来进行动态分配能达到一个比较好的效果。

实验二 高斯消元 Pthread 实现

1 问题描述

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。

2 算法设计与分析

2.1 高斯消元法介绍

消元过程：将 Ax=b 按照从上至下、从左至右的顺序化为上三角方程组，中间过程不对矩阵进行交换，主要步骤如下：

Step1: 将第 2 行至第 n 行，每行分别与第一行做运算，消掉每行第一个参数。公式如下：

$$a_{11}^{(1)} \neq 0, l_{i1} = \frac{a_{i1}^{(1)}}{a_{11}^{(1)}} \quad (i = 2:n), \text{第} i \text{行} + (-l_{i1}) \times \text{第 1 行} \quad (i = 2:n)$$

Step2: 从新矩阵的 a22 开始（a22 不能为 0），以第二行为基准，将第三行至第 n 行分别与第二行做运算，消掉每行第二个参数。公式如下：

$$l_{i2} = \frac{a_{i2}^{(2)}}{a_{22}^{(2)}} \quad (i = 3:n), \text{第} i \text{行加上} (-l_{i2}) \times \text{第 2 行} \quad (i = 3:n)$$

Step K: 按照上述方法，当第 k 步运算时，公式为：

$$a_{kk}^{(k)} \neq 0, l_{ik} = \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} \quad (i = k+1 \to n), \text{第} i \text{行加上} (-l_{ik}) \times \text{第} k \text{行} \quad (i = k+1 \to n)$$

Step n-1: 经过 n-1 步，方程组也就转化为了我们希望得到的上三角方程组，如下：

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ a_{22}^{(2)} & & \cdots & a_{2n}^{(2)} \\ \vdots & & \ddots & \vdots \\ a_{nn}^{(n)} & & & x_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(n)} \end{bmatrix}$$

回代过程：从第 n 行开始，倒序回代前面的行中，即可求解 x_1 到 x_n 的值。

2.2 串行算法

串行高斯消元的方法主要分成两步，第一步就是把要当被减行的行进行放大或缩小使得该行第一个不为 0 的数为 1（该步骤可以省去，只是后面乘的系数不一样），第二步便是遍历被减行（第 k 行）下面的所有行（k+1 行到 n 行），每一行都减去被减行与一个系数的乘积，即假设现在是 k+1 行 k+1 列，该数减去被减行对应列的值乘以一个系数的乘积，即第 k 行 k+1 列的值乘以 k+1 行 k 列的值，即为更新的值，以此类推。最后再把 k+1 行 k 列设置为 0。

如果省略第一步，那么系数则不为 k+1 行 k 列的值，而是 k+1 行 k 列的值除以 k 行 k 列的值。注意在矩阵遍历的时候，矩阵为 n* (n+1) 的规模，因为构成的矩阵为增广矩阵。由于循环最多嵌套三层，因此时间复杂度为 O(n) = n^3

具体代码如下：

```
1 double LU(int n, float a[][maxN]){
2     LARGE_INTEGER freq;
3    LARGE_INTEGER beginTime;
4    LARGE_INTEGER endTime;
5
6     QueryPerformanceFrequency(&freq);
7     QueryPerformanceCounter(&beginTime);
8     for (int i = 0; i < n - 1; i++) //对每一行
9         {
10            for(int j=i+1;j≤n;j++)//对这一行的每一个数
11            {
12                a[i][j]=a[i][j]/a[i][i];
13            }
14            a[i][i]=1.0;
```

```
15          for (int j = i + 1; j < n; j++)//每一行的下面的行
16           {
17               float tem = a[j][i] / a[i][i];
18               for (int k = i + 1; k ≤ n; k++) //这个是列，列是n+1个（有b）
19               {
20                   a[j][k] -= a[i][k] * tem;
21               }
22               a[j][i] = 0.00;
23           }
24      }
25      QueryPerformanceCounter(&endTime);
26
27      double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
        freq.QuadPart;
28      cout << "普通方法耗时: " << time << "s" << endl;
29      return time;
30
31
32 }
```

2.3 普通 Pthread(动态分配线程)

主要是两个函数，LU_pthread 函数是一个线程函数，执行高斯分解的一部分。LU 函数是主函数，它执行高斯分解的整个过程。在 LU 函数中，使用 pthread_create 函数创建了多个线程，并将 LU_pthread 函数作为线程执行的函数。每个线程获得一个包含执行信息的结构体作为参数。在所有线程完成它们的任务之后，使用 pthread_join 函数来等待所有线程完成。

此处的线程是静态线程，在派发时任务均匀分配，因为高斯消去的矩阵为随机生成，因此任务平均分配也比较均匀，于是使用静态线程。

```
1 void* LU_pthread(void* param) {
2       threadParam_t* p = (threadParam_t*)param;
3      int k = p->k;                //第k步
4      int t_id = p->t_id;          //第几个线程
5      int c_size=p->chunk_size;//一个线程执行几行
```

```
6       int start = k + t_id*c_size+1 ;    //获取任务
7       int n=p->n;
8       for(int j=start;(j<start+c_size)&&(j<n);j++)
9       {
10           for(int m=k+1;m≤n;m++)
11      {
12          a[j][m]-=a[k][m]*a[j][k];
13
14      }
15      a[j][k]=0.0;
16      }
17
18      pthread_exit(NULL);
19      return NULL;
20  }
21  double LU(int n, float a[][maxN]) {
22      LARGE_INTEGER freq;
23      LARGE_INTEGER beginTime;
24      LARGE_INTEGER endTime;
25
26      QueryPerformanceFrequency(&freq);
27      QueryPerformanceCounter(&beginTime);
28      for (int i = 0; i < n; i++) //对每一行
29      {
30          for (int j = i + 1;j ≤ n;j++)//对这一行的每一个数
31          {
32              a[i][j] = a[i][j] / a[i][i];
33          }
34          a[i][i] = 1.0;
35
36          int thread_cnt = THREAD_NUM;
37          int thread_size=(n-(i+1))/THREAD_NUM+1;
38          pthread_t* thread_handles = (pthread_t*)malloc(thread_cnt * sizeof(...
                pthread_t));
39          threadParam_t* param = (threadParam_t*)malloc(thread_cnt * sizeof(...
                threadParam_t));
40          for (int t_id = 0; t_id < thread_cnt; t_id++) {//分配任务
```

```
41              param[t_id].k = i;//到i步
42              param[t_id].t_id = t_id;//第几个线程
43              param[t_id].chunk_size=thread_size;//每个片分几个
44              param[t_id].n=n;
45          }
46
47          for (int t_id = 0; t_id < thread_cnt; t_id++) {
48              pthread_create(&thread_handles[t_id], NULL, LU_pthread, &param[...
                    t_id]);
49          }
50
51          for (int t_id = 0; t_id < thread_cnt; t_id++) {
52              pthread_join(thread_handles[t_id], NULL);
53          }
54          free(thread_handles);
55          free(param);
56      }
57
58
59      QueryPerformanceCounter(&endTime);
60
61      double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
            freq.QuadPart;
62      cout << "普通Pthread方法耗时: " << time << "s" << endl;
63      return time;
64
65
66  }
```

2.4 Pthread 结合 SSE（动态分配线程）

Pthread 结合 SSE 编程，其实就是在线程函数中使用 SSE 来加快运算，以达到提高运行效率的结果。代码原理其实和上面高斯消去只使用 Pthread 的方法相同。在 LU_SSE 中进行线程的派发，然后每一个线程都执行 LU_SSE_pthread 函数。这里采用的是静态分配任务，即在每一行向下消元时，每个线程的任务平均分配。在消元时使用 SSE 进行加速。

```
1 void* LU_SSE_pthread(void* param) {
2      threadParam_t* p = (threadParam_t*)param;
3     int k = p->k;             //第k步
4     int t_id = p->t_id;       //第几个线程
5     int c_size=p->chunk_size;//一个线程执行几行
6     int start = k + t_id*c_size+1 ;    //获取任务
7     int n=p->n;
8     __m128 t1, t2, sub, tem2;
9     for(int j=start;(j<start+c_size)&&(j<n);j++)
10    {    float tem = a[j][k];
11             tem2 = _mm_set1_ps(tem);
12          for(int m=k+1;m≤n;m+=4)
13    {
14         if (m + 3 > n)break;
15                  t1 = _mm_loadu_ps(a[k] + m);
16                  t2 = _mm_loadu_ps(a[j] + m);
17                  sub = _mm_sub_ps(t2, _mm_mul_ps(t1, tem2));
18                  _mm_storeu_ps(a[j] + m, sub);
19
20
21    }
22    for (int m = n - (n - k) % 4 + 1; m ≤ n; m += 1) {
23                  a[j][m]-=a[k][m]*a[j][k];
24             }
25
26    a[j][k]=0.0;
27    }
28
29    pthread_exit(NULL);
30    return NULL;
31 }
32 double LU_SSE(int n, float a[][maxN])
33 {
34     LARGE_INTEGER freq;
35    LARGE_INTEGER beginTime;
36    LARGE_INTEGER endTime;
37
```

```
38      QueryPerformanceFrequency(&freq);
39      QueryPerformanceCounter(&beginTime);
40      for (int i = 0; i < n; i++) //对每一行
41      {
42          for (int j = i + 1;j ≤ n;j++)//对这一行的每一个数
43          {
44              a[i][j] = a[i][j] / a[i][i];
45          }
46          a[i][i] = 1.0;
47
48          int thread_cnt = THREAD_NUM;
49          int thread_size=(n-(i+1))/THREAD_NUM+1;
50          pthread_t* thread_handles = (pthread_t*)malloc(thread_cnt * sizeof(...
                pthread_t));
51          threadParam_t* param = (threadParam_t*)malloc(thread_cnt * sizeof(...
                threadParam_t));
52          for (int t_id = 0; t_id < thread_cnt; t_id++) {//分配任务
53              param[t_id].k = i;//到i步
54              param[t_id].t_id = t_id;//第几个线程
55              param[t_id].chunk_size=thread_size;//每个片分几个
56              param[t_id].n=n;
57          }
58
59          for (int t_id = 0; t_id < thread_cnt; t_id++) {
60              pthread_create(&thread_handles[t_id], NULL, LU_SSE_pthread, &...
                    param[t_id]);
61          }
62
63          for (int t_id = 0; t_id < thread_cnt; t_id++) {
64              pthread_join(thread_handles[t_id], NULL);
65          }
66          free(thread_handles);
67          free(param);
68      }
69
70
71      QueryPerformanceCounter(&endTime);
```

```
72
73      double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
            freq.QuadPart;
74      cout << "SSE方法耗时: " << time << "s" << endl;
75      return time;
76
77  }
```

2.5 Pthread 结合 AVX（动态分配线程）

高斯消去法 Pthread 结合 AVX 编程的方法和高斯消去法 Pthread 结合 SSE 编程的方法大同小异，SSE 编程改为 AVX 编程即可。

```
1  void* LU_AVX_pthread(void* param) {
2       threadParam_t* p = (threadParam_t*)param;
3      int k = p->k;              //第k步
4      int t_id = p->t_id;        //第几个线程
5      int c_size=p->chunk_size;//一个线程执行几行
6      int start = k + t_id*c_size+1 ;    //获取任务
7      int n=p->n;
8      __m256 t1, t2, sub, tem2;
9      for(int j=start;(j<start+c_size)&&(j<n);j++)
10     {    float tem = a[j][k];
11              tem2 = _mm256_set1_ps(tem);
12          for(int m=k+1;m≤n;m+=8)
13     {
14          if (m + 7 > n)break;
15                  t1 = _mm256_loadu_ps(a[k] + m);
16                  t2 = _mm256_loadu_ps(a[j] + m);
17                  sub = _mm256_sub_ps(t2, _mm256_mul_ps(t1, tem2));
18                  _mm256_storeu_ps(a[j] + m, sub);
19
20
21     }
22     for (int m = n - (n - k) % 8 + 1; m ≤ n; m += 1) {
23                  a[j][m]-=a[k][m]*a[j][k];
```

```
24                 }
25
26         a[j][k]=0.0;
27         }
28
29         pthread_exit(NULL);
30         return NULL;
31 }
32 double LU_AVX(int n, float a[][maxN])
33 {
34          LARGE_INTEGER freq;
35         LARGE_INTEGER beginTime;
36         LARGE_INTEGER endTime;
37
38         QueryPerformanceFrequency(&freq);
39         QueryPerformanceCounter(&beginTime);
40         for (int i = 0; i < n; i++) //对每一行
41         {
42             for (int j = i + 1;j < n;j++)//对这一行的每一个数
43             {
44                 a[i][j] = a[i][j] / a[i][i];
45             }
46             a[i][i] = 1.0;
47
48             int thread_cnt = THREAD_NUM;
49             int thread_size=(n-(i+1))/THREAD_NUM+1;
50             pthread_t* thread_handles = (pthread_t*)malloc(thread_cnt * sizeof(...
                   pthread_t));
51             threadParam_t* param = (threadParam_t*)malloc(thread_cnt * sizeof(...
                   threadParam_t));
52             for (int t_id = 0; t_id < thread_cnt; t_id++) {//分配任务
53                 param[t_id].k = i;//到i步
54                 param[t_id].t_id = t_id;//第几个线程
55                 param[t_id].chunk_size=thread_size;//每个片分几个
56                 param[t_id].n=n;
57             }
58
```

```
59          for (int t_id = 0; t_id < thread_cnt; t_id++) {
60              pthread_create(&thread_handles[t_id], NULL, LU_AVX_pthread, &...
                    param[t_id]);
61          }
62
63          for (int t_id = 0; t_id < thread_cnt; t_id++) {
64              pthread_join(thread_handles[t_id], NULL);
65          }
66          free(thread_handles);
67          free(param);
68      }
69
70
71      QueryPerformanceCounter(&endTime);
72
73      double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
            freq.QuadPart;
74      cout << "AVX方法耗时: " << time << "s" << endl;
75      return time;
76
77 }
```

2.6 AVX 结合信号量（静态分配线程）

采用信号量，避免反复创建线程带来的时间开销。这里采用静态分配线程，静态分配任务。等主线程执行除法完成后，再唤起子线程进行消元，这里采用 AVX 编程加快运行速度。

```
1 void* LU_AVX_pthread_sem(void* param) {
2     threadParam_t* p = (threadParam_t*)param;
3     int t_id = p->t_id;
4     int n=p->n;
5     for (int k = 0; k < n; k++) {
6         sem_wait(&sem_workstart[t_id]);//阻塞，等待主线程除法完成
7
8         for (int i = k + 1 + t_id; i < n; i += THREAD_NUM) {
9             __m256 vaik = _mm256_set1_ps(a[i][k]);
```

```
10              int j;
11              for (int j = k + 1; j ≤ n; j += 8) {
12                      if(j+7>n)break;
13                  __m256 vakj = _mm256_loadu_ps(&a[k][j]);
14                  __m256 vaij = _mm256_loadu_ps(&a[i][j]);
15                  __m256 vx = _mm256_mul_ps(vakj, vaik);
16                  vaij = _mm256_sub_ps(vaij, vx);
17                  _mm256_storeu_ps(&a[i][j], vaij);
18              }
19              for (int j =n-(n-k)%8+1;j ≤n; j++) {
20                  a[i][j] =a[i][j] - a[i][k] * a[k][j];
21              }
22              a[i][k] = 0.0;
23          }
24          sem_post(&sem_main);          //唤醒主线程
25          sem_wait(&sem_workend[t_id]);  //阻塞，等待主线程唤醒进入下一轮
26
27      }
28      pthread_exit(NULL);
29      return NULL;
30 }
31 double LU_AVX_sem(int n, float a[][maxN])
32 {
33       LARGE_INTEGER freq;
34      LARGE_INTEGER beginTime;
35      LARGE_INTEGER endTime;
36      QueryPerformanceFrequency(&freq);
37      QueryPerformanceCounter(&beginTime);
38     sem_init(&sem_main, 0, 0); //初始化信号量
39      for (int i = 0; i < THREAD_NUM; i++) {
40          sem_init(&sem_workend[i], 0, 0);
41          sem_init(&sem_workstart[i], 0, 0);
42      }
43      pthread_t* handle = (pthread_t*)malloc(THREAD_NUM * sizeof(pthread_t));
44      threadParam_t* param = (threadParam_t*)malloc(THREAD_NUM * sizeof(...
            threadParam_t));
45      for (int t_id = 0; t_id < THREAD_NUM; t_id++) {
```

```
46          param[t_id].t_id = t_id;
47          param[t_id].k = 0;
48          param[t_id].n=n;
49          pthread_create(&handle[t_id], NULL, LU_AVX_pthread_sem, &param[t_id...
                ]);
50
51      }
52
53      for (int k = 0; k < n; k++) {
54
55          __m256 vt = _mm256_set1_ps(a[k][k]);
56
57          for (int j = k + 1; j  ≤ n; j += 8) {
58                  if(j+7>n)break;
59              __m256 va = _mm256_loadu_ps(&a[k][j]);
60              va = _mm256_div_ps(va, vt);
61              _mm256_storeu_ps(&a[k][j], va);
62          }
63          for (int j=n-(n-k)%8+1 ; j ≤ n; j++) {
64              a[k][j] = a[k][j] / a[k][k];
65          }
66         a[k][k] = 1.0;
67
68          for (int t_id = 0; t_id < THREAD_NUM; t_id++) {  //唤起子线程
69              sem_post(&sem_workstart[t_id]);
70          }
71
72          for (int t_id = 0; t_id < THREAD_NUM; t_id++) {  //主线程睡眠
73              sem_wait(&sem_main);
74          }
75
76          for (int t_id = 0; t_id < THREAD_NUM; t_id++) {  //再次唤起工作线...
                程，进入下一轮消去
77              sem_post(&sem_workend[t_id]);
78          }
79
80      }
```

```
81      for (int t_id = 0; t_id < THREAD_NUM; t_id++) {
82          pthread_join(handle[t_id], NULL);
83      }
84      sem_destroy(&sem_main);      //销毁线程
85      for (int t_id = 0; t_id < THREAD_NUM; t_id++)
86          sem_destroy(&sem_workstart[t_id]);
87      for (int t_id = 0; t_id < THREAD_NUM; t_id++)
88          sem_destroy(&sem_workend[t_id]);
89
90      QueryPerformanceCounter(&endTime);
91
92      double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
            freq.QuadPart;
93      cout << "AVX结合Pthread结合信号量方法耗时: " << time << "s" << endl;
94      return time;
95
96  }
```

2.7 AVX 结合屏障（静态分配线程，静态分配任务）

采用屏障，避免反复创建线程带来的时间开销。这里的线程 0 固定用为最开始的除法，等所有线程都到达之后便开始向下消元。这里采用的是静态分配的方法，每个线程执行的任务数平均。

```
1  void* LU_AVX_pthread_barrier(void* param) {
2      threadParam_t* p = (threadParam_t*)param;
3      int t_id = p->t_id;
4      int n=p->n;
5      for (int k = 0; k <n; k++) { //0号线程做除法
6          if (t_id == 0) {
7              __m256 vt = _mm256_set1_ps(a[k][k]);
8              for (int j = k + 1; j ≤ n; j += 8) {
9                  if(j+7>n)break;
10                 __m256 va = _mm256_loadu_ps(&a[k][j]);
11                 va = _mm256_div_ps(va, vt);
12                 _mm256_storeu_ps(&a[k][j], va);
```

```
13              }
14              for (int j=n-(n-k)%8+1 ; j ≤ n;  j++) {
15                  a[k][j] =a[k][j] / a[k][k];
16              }
17              a[k][k] = 1.0;
18          }
19
20          pthread_barrier_wait(&barrier_Division);//第一个同步点
21
22          for (int i = k + 1 + t_id; i < n; i += THREAD_NUM) {
23              __m256 vaik = _mm256_set1_ps(a[i][k]);
24
25              for (int j = k + 1; j ≤n ; j += 8) {
26                  if (j + 7 > n)break;
27                  __m256 vakj = _mm256_loadu_ps(&a[k][j]);
28                  __m256 vaij = _mm256_loadu_ps(&a[i][j]);
29                  __m256 vx = _mm256_mul_ps(vakj, vaik);
30                  vaij = _mm256_sub_ps(vaij, vx);
31                  _mm256_storeu_ps(&a[i][j], vaij);
32              }
33              for (int j  = n - (n - k) % 8 + 1; j ≤ n; j++) {
34                  a[i][j] = a[i][j] - a[i][k] * a[k][j];
35              }
36              a[i][k] = 0.0;
37          }
38
39          pthread_barrier_wait(&barrier_Elimination);//第二个同步点
40
41
42      }
43      pthread_exit(NULL);
44      return NULL;
45 }
46 double LU_AVX_barrier(int n, float a[][maxN])
47 {
48       LARGE_INTEGER freq;
49      LARGE_INTEGER beginTime;
```

```
50      LARGE_INTEGER endTime;
51      QueryPerformanceFrequency(&freq);
52      QueryPerformanceCounter(&beginTime);
53    pthread_barrier_init(&barrier_Division, NULL,THREAD_NUM);
54      pthread_barrier_init(&barrier_Elimination, NULL, THREAD_NUM);
55      pthread_t* handle = (pthread_t*)malloc(THREAD_NUM * sizeof(pthread_t));
56      threadParam_t* param = (threadParam_t*)malloc(THREAD_NUM * sizeof(...
            threadParam_t));
57      for (int t_id = 0; t_id < THREAD_NUM; t_id++) {
58          param[t_id].t_id = t_id;
59          param[t_id].k = 0;
60          param[t_id].n=n;
61          pthread_create(&handle[t_id], NULL,LU_AVX_pthread_barrier, &param[...
                t_id]);
62
63      }
64
65      for (int t_id = 0; t_id < THREAD_NUM; t_id++) {
66          pthread_join(handle[t_id], NULL);
67      }
68
69      pthread_barrier_destroy(&barrier_Division);
70      pthread_barrier_destroy(&barrier_Elimination);
71
72      free(handle);
73      free(param);
74      QueryPerformanceCounter(&endTime);
75
76      double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
            freq.QuadPart;
77      cout << "AVX, Pthread, 屏障结合方法耗时: " << time << "s" << endl;
78      return time;
79
80 }
```

2.8 AVX 结合屏障（静态分配线程，动态分配任务）

这里采用动态分配任务的方法，保证每个线程之间运行时间差异不会特别大。采用锁机制来保证线程运行的安全性。

```
1  void* LU_AVX_pthread_barrier2(void* param) {
2      threadParam_t* p = (threadParam_t*)param;
3      int t_id = p->t_id;
4      int n=p->n;
5      for (int k = 0; k < n; k++) { //0号线程做除法
6          if (t_id == 0) {
7              __m256 vt = _mm256_set1_ps(a[k][k]);
8              for (int j = k + 1; j ≤ n; j += 8) {
9                  if(j+7>n)break;
10                 __m256 va = _mm256_loadu_ps(&a[k][j]);
11                 va = _mm256_div_ps(va, vt);
12                 _mm256_storeu_ps(&a[k][j], va);
13             }
14             for (int j=n-(n-k)%8+1 ; j ≤n;  j++) {
15                 a[k][j] =a[k][j] / a[k][k];
16             }
17             a[k][k] = 1.0;
18         }
19         next_arr=k+1;
20         int task=0;
21         __m256 t1, t2, sub, tem2;
22         pthread_barrier_wait(&barrier_Division);//第一个同步点
23
24
25             while(1)
26             {
27                 pthread_mutex_lock(&mutex_task);
28                 task = next_arr;
29                 next_arr+=1;
30                 pthread_mutex_unlock(&mutex_task);
31                 if (task ≥ n) break;
```

```
32                  int end_task=task+1;
33                  if(end_task>n)end_task=n;
34                  for(int j = task; j < end_task; j++)
35                {    tem2 = _mm256_set1_ps(a[j][k]);
36                       for(int m=k+1;m≤n;m+=8)
37      {
38                  if (m + 7 > n)break;
39                  t1 = _mm256_loadu_ps(a[k] + m);
40                  t2 = _mm256_loadu_ps(a[j] + m);
41                  sub = _mm256_sub_ps(t2, _mm256_mul_ps(t1, tem2));
42                  _mm256_storeu_ps(a[j] + m, sub);
43
44
45      }
46      for (int m = n - (n - k) % 8 + 1; m ≤ n; m += 1) {
47                  a[j][m]-=a[k][m]*a[j][k];
48              }
49
50                  a[j][k]=0.0;
51
52                }
53              }
54
55          pthread_barrier_wait(&barrier_Elimination);//第二个同步点
56
57
58      }
59      pthread_exit(NULL);
60      return NULL;
61
62 }
63 double LU_AVX_barrier2(int n, float a[][maxN])
64 {
65       LARGE_INTEGER freq;
66      LARGE_INTEGER beginTime;
67      LARGE_INTEGER endTime;
68      QueryPerformanceFrequency(&freq);
```

```
69      QueryPerformanceCounter(&beginTime);
70    pthread_barrier_init(&barrier_Division, NULL,THREAD_NUM);
71      pthread_barrier_init(&barrier_Elimination, NULL, THREAD_NUM);
72      pthread_t* handle = (pthread_t*)malloc(THREAD_NUM * sizeof(pthread_t));
73      threadParam_t* param = (threadParam_t*)malloc(THREAD_NUM * sizeof(...
            threadParam_t));
74      for (int t_id = 0; t_id < THREAD_NUM; t_id++) {
75          param[t_id].t_id = t_id;
76          param[t_id].k = 0;
77          param[t_id].n=n;
78          pthread_create(&handle[t_id], NULL,LU_AVX_pthread_barrier2, &param[...
                t_id]);
79
80      }
81
82      for (int t_id = 0; t_id < THREAD_NUM; t_id++) {
83          pthread_join(handle[t_id], NULL);
84      }
85
86      pthread_barrier_destroy(&barrier_Division);
87      pthread_barrier_destroy(&barrier_Elimination);
88
89      free(handle);
90      free(param);
91      QueryPerformanceCounter(&endTime);
92
93      double time = (double)(endTime.QuadPart - beginTime.QuadPart) / (double)...
            freq.QuadPart;
94      cout << "AVX, Pthread, 屏障结合动态分配任务方法耗时: " << time << "s" <<...
             endl;
95      return time;
96
97  }
```

3 结果统计与分析

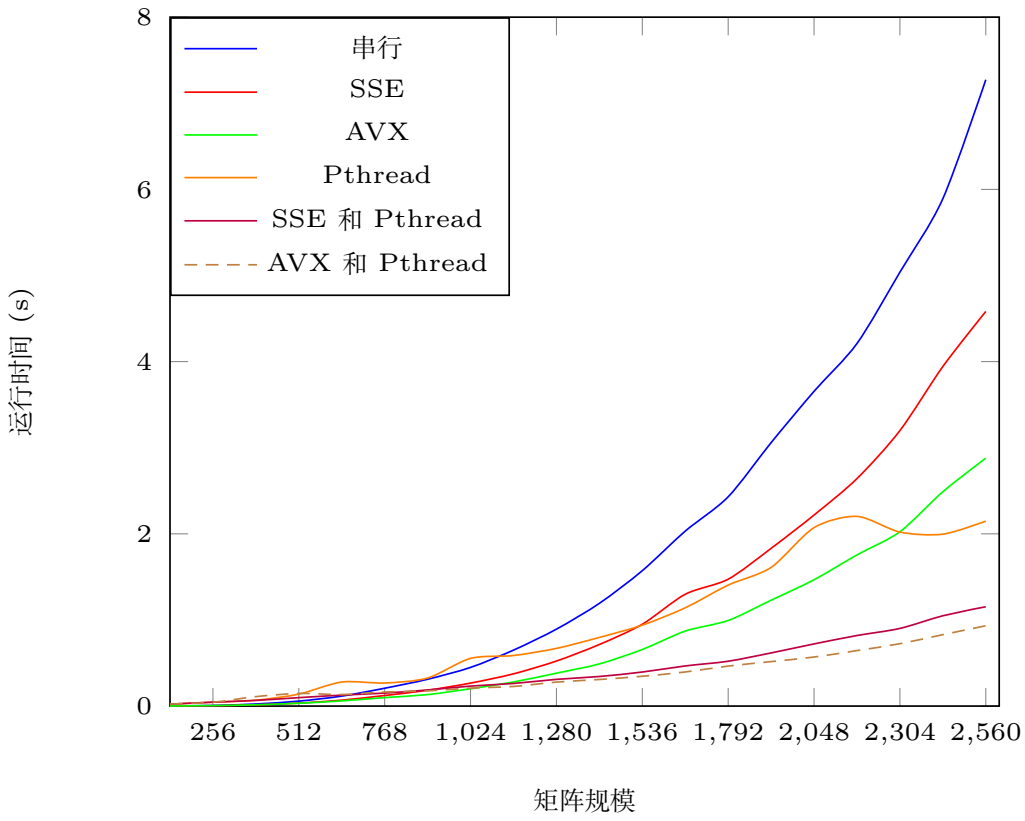
由于不同矩阵规模对实验结果有较大影响，这里以矩阵规模 2560*2560 为例探讨最优的方案。

3.1 不同方法运行时间比较

下面先比较普通的高斯消元方法, SSE 编程, AVX 编程, Pthread 高斯消元方法, Pthread 和 SSE 编程结合高斯消元方法，Pthread 和 AVX 编程结合高斯消元方法的运行时间。任务分配均采取静态分配，线程采取 8 线程。

矩阵规模	串行	SSE	AVX	Pthread	Pthread 和 SSE	Pthread 和 AVX
128*128	0.0009	0.0006	0.0011	0.0226	0.0224	0.0236
256*256	0.0071	0.0041	0.0055	0.0452	0.0455	0.0444
384*384	0.0242	0.0144	0.0152	0.0682	0.0660	0.1097
512*513	0.0574	0.0341	0.0328	0.1391	0.0979	0.1466
640*640	0.1156	0.0689	0.0601	0.2792	0.1272	0.1341
768*768	0.2072	0.1226	0.0985	0.2677	0.1506	0.1632
896*896	0.3115	0.1836	0.1332	0.3257	0.1869	0.1766
1024*1024	0.4490	0.2666	0.2013	0.5537	0.2322	0.2082
1152*1152	0.6496	0.3742	0.2798	0.5871	0.2625	0.2280
1280*1280	0.8931	0.5229	0.3821	0.6697	0.3102	0.2784
1408*1408	1.1929	0.7150	0.4896	0.7970	0.3435	0.3087
1536*1536	1.5722	0.9500	0.6571	0.9396	0.3953	0.3461
1664*1664	2.0317	1.2984	0.8708	1.1410	0.4659	0.3960
1792*1792	2.4301	1.4733	0.9940	1.4047	0.5207	0.4646
1920*1920	3.0591	1.8314	1.2291	1.6104	0.6171	0.5164
2048*2048	3.6550	2.2183	1.4668	2.0719	0.7217	0.5693
2176*2176	4.2105	2.6418	1.7552	2.2027	0.8191	0.6442
2304*2304	5.0389	3.1991	2.0243	2.0203	0.9017	0.7242
2432*2432	5.8919	3.9433	2.4891	1.9962	1.0486	0.8293
2560*2560	7.2735	4.5828	2.8792	2.1478	1.1538	0.9326

表 2.1: 不同优化方法下的运行时间



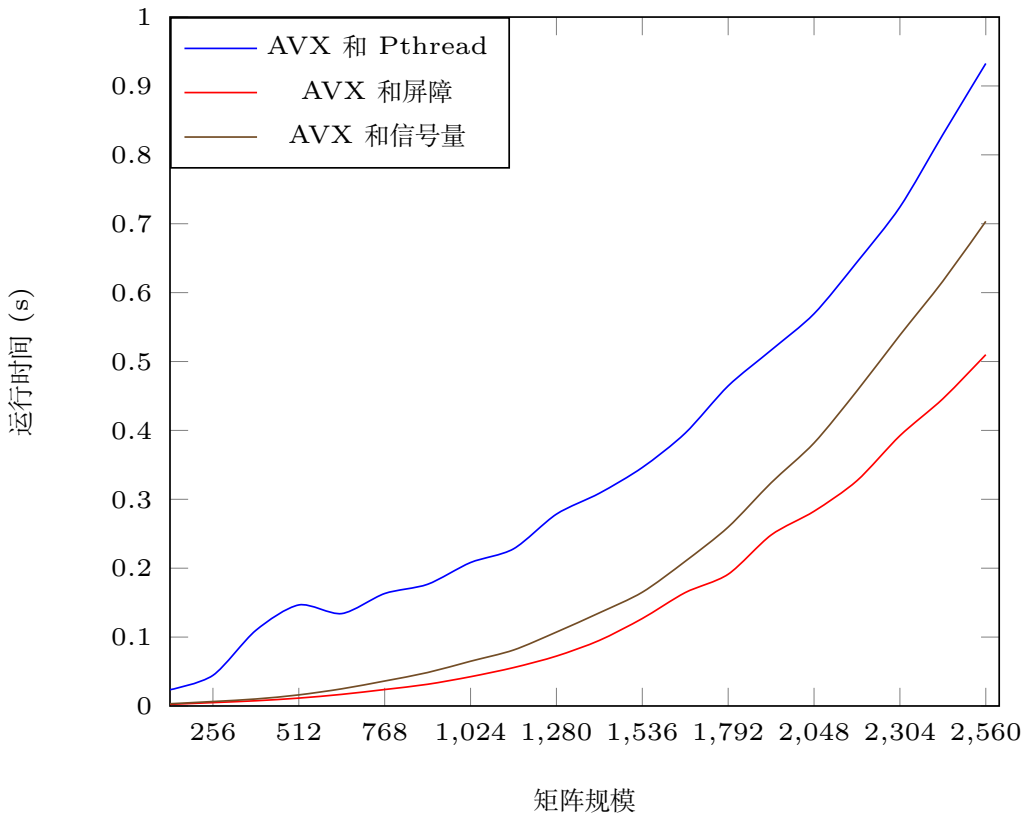
由图可知,当矩阵规模并不大的时,串行反而是效果比较好的。这是因为在在使用 Pthread 时,会产生额外的线程开销,当多线程降低的时间开销无法抵消开多个线程所带来的时间开销的时候，就没有必要使用多线程来进行加速。但是当矩阵规模变得越来越大的时候，Pthread 开始逐渐显示出它的优势，并且 Pthread 结合 AVX 能达到一个最好的效果。整体性能排行：Pthread 和 AVX>Pthread 和 SSE>Pthread>AVX>SSE> 串行。因此在这六种计算方式中，Pthread 和 AVX 编程是最好的选择。

3.2 对 Pthread 的改进

从上面的结论可以看出，AVX 与 Pthread 结合是最好的选择。但是，这里仍存在着可以避免的开销，也就是线程不断创建的动态开销。在上面的算法中，每一轮都要创建新的一组线程，再接着销毁，这个时间开销是比较大的。如何避免呢？我们可以结合信号量和屏障来进行计算，将动态分配线程改为静态分配线程下面是普通 Pthread 和 AVX，信号量和 AVX，屏障和 AVX 的运行时间表。(采取八线程进行计算)

矩阵规模	AVX 和 Pthread	AVX 和屏障	AVX 和信号量
128*128	0.0236	0.0024	0.0032
256*256	0.0444	0.0049	0.0065
384*384	0.1098	0.0076	0.0101
512*512	0.1467	0.0114	0.0161
640*640	0.1341	0.0169	0.0249
768*768	0.1632	0.0239	0.0362
896*896	0.1766	0.0315	0.0485
1024*1024	0.2082	0.0425	0.0649
1152*1152	0.2280	0.0557	0.0812
1280*1280	0.2785	0.0724	0.1072
1408*1408	0.3088	0.0952	0.1352
1536*1536	0.3462	0.1270	0.1651
1664*1664	0.3960	0.1646	0.2100
1792*1792	0.4646	0.1912	0.2597
1920*1920	0.5165	0.2481	0.3239
2048*2048	0.5693	0.2826	0.3818
2176*2176	0.6443	0.3270	0.4574
2304*2304	0.7242	0.3923	0.5383
2432*2432	0.8294	0.4459	0.6166
2560*2560	0.9326	0.5099	0.7035

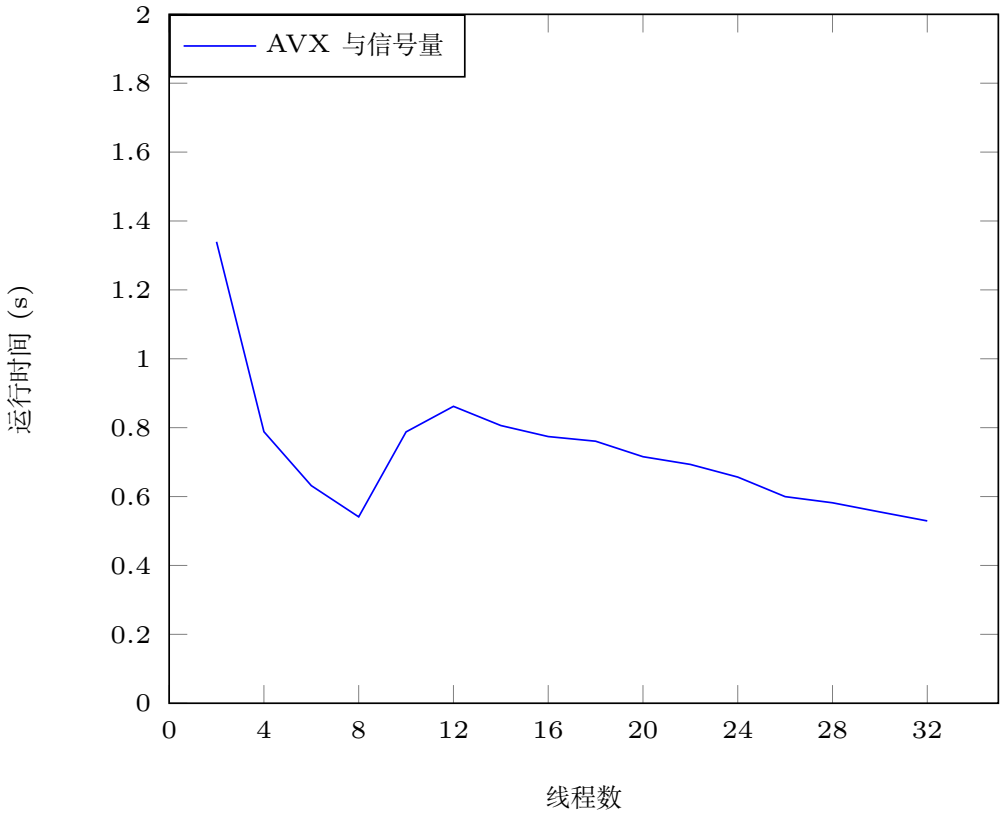
表 2.2: 不同优化方法下的运行时间



可以看出，当矩阵规模较大的时候，使用屏障和信号量减少了很多创建线程的额外开销，因此效果更好。同时可以很明显地看出，使用屏障的效果比使用信号量的效果要好一点。推测是因为屏障同步不需要多次调用信号函数进行资源分配，所以线程工作效率更高。

3.3 不同线程数运行的比较

上面我们得出 Pthread 结合屏障能得到最好的方案，下面我们来讨论线程数对运行时间的影响。线程数增加并不一定能减少运行的时间，因为线程数增加也就代表着额外的线程开销的增加。线程太少，会使得加速效果不明显，而线程太多，导致管理线程、分配任务的开销又增大，使得其加速效果下降，因此找到合适的线程数目极为重要。下面是当矩阵规模为 2560*2560 时，不同线程数运行时间的比较。



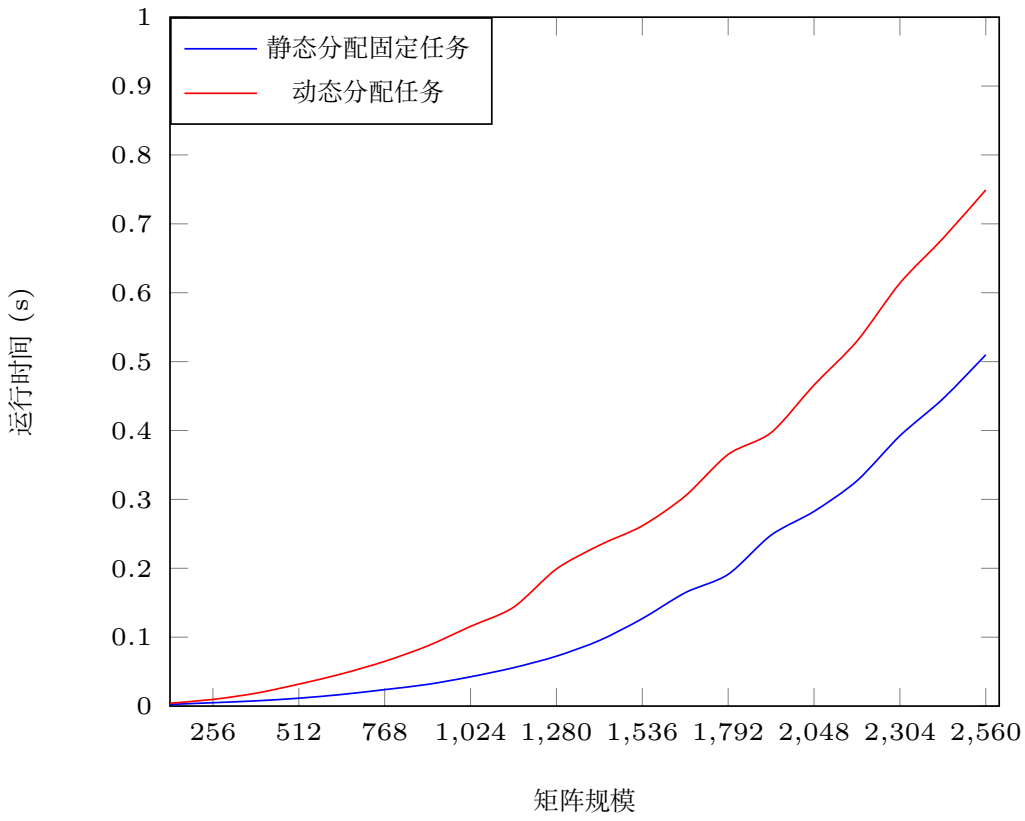
可以看到图中，当线程数为 8 时运行时间是最短的。所以当线程数为 8 时，是最好的选择。

4 分配任务动态和静态的比较

上面得出线程数为 8，AVX 与屏障结合是最好的方法。下面探讨任务分配动态和静态对运行时间的影响。

矩阵规模	静态分配固定任务	动态分配任务
128*128	0.0024	0.0041
256*256	0.0049	0.0096
384*384	0.0075	0.0186
512*512	0.0114	0.0318
640*640	0.0168	0.0467
768*768	0.0238	0.0648
896*896	0.0314	0.0873
1024*1024	0.0424	0.1156
1152*1152	0.0556	0.1433
1280*1280	0.0723	0.1990
1408*1408	0.0951	0.2334
1536*1536	0.1269	0.2616
1664*1664	0.1646	0.3048
1792*1792	0.1912	0.3653
1920*1920	0.2481	0.3969
2048*2048	0.2826	0.4660
2176*2176	0.3270	0.5297
2304*2304	0.3923	0.6139
2432*2432	0.4459	0.6786
2560*2560	0.5099	0.7492

表 2.3: 不同任务分配方式下的运行时间



可以看出，动态分配任务的效率不如静态分配。这是因为高斯消元的矩阵为随机生成，此时任务比较均衡，并没有必要通过平均分配来提高效率，分配任务的步骤反而增大了时间开销，不利于提高效率。

5 实验结论

综上所述，当计算 2560*2560 规模的矩阵时，采用 8 线程，AVX 与屏障结合，任务采取静态分配是最好的方法。当然，当矩阵规模不断变化时，答案也不固定。如果采取并行计算所减少的时间开销大于并行计算所带来的额外开销，那么这个并行计算就是有意义的。对比普通串行的高斯消元方法，该最佳方法加速比达到了惊人的 14.26。

实验三　附加题

1　问题描述

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。

2　算法设计与分析

2.1　路障 Barrier

原始路障 Barrier 的代码如下：

```
1 typedef struct{
2 int threadId;
3 }threadParm_t;
4 pthread_barrier_t barrier;
5 void *threadFunc(void *parm)
6 {
7 threadParm_t *p = (threadParm_t *) parm;
8 fprintf(stdout, "Thread %d has entered step 1.\n", p->threadId);
9 pthread_barrier_wait(&barrier);
10 fprintf(stdout, "Thread %d has entered step 2.\n", p->threadId);
11 pthread_exit(NULL);
12 }
13 int main(int argc, char *argv[])
14 {
15 pthread_barrier_init(&barrier, NULL, NUM_THREADS);
16 pthread_t thread[NUM_THREADS];
17 threadParm_t threadParm[NUM_THREADS];
```

```
18 int i;
19 for (i=0; i<NUM_THREADS; i++)
20 {
21 threadParm[i].threadId = i;
22 pthread_create(&thread[i], NULL, threadFunc, (void
23 *)&threadParm[i]);
24 }
25 for (i=0; i<NUM_THREADS; i++)
26 {
27 pthread_join(thread[i], NULL);
28 }
29 pthread_barrier_destroy(&barrier);
30 system("PAUSE");
31 return 0;
32 }
```

2.2 信号量实现路障 Barrier 的功能

采用一个变量来进行计数，当计数到了线程数之后，说明线程全运行完第一步了，则开始第二步，这样就起到了屏障的作用。

```
1 int count = 0;
2 typedef struct{
3 int threadId;
4 } threadParm_t;
5 sem_t sem_count;
6 sem_t sem_barrier;
7 void *threadFunc(void *parm)
8 {
9 threadParm_t *p = (threadParm_t *) parm;
10 fprintf(stdout,"Thread %d has entered step 1.\n", p->threadId);
11 sem_wait(&sem_count);
12 if (counter == NUM_THREADS - 1)
13     {
14         counter = 0;
15         sem_post(&sem_count);
```

```
16          for (int i = 0; i < NUM_THREADS - 1; i++) sem_post(&sem_barrier);
17      }
18      else
19      {
20          count++;
21          sem_post(&sem_count);
22          sem_wait(&sem_barrier);
23      }
24
25 fprintf(stdout, "Thread %d has entered step 2.\n", p->threadId);
26 pthread_exit(NULL);
27 }
28 int main(int argc, char *argv[])
29 {
30 sem_init(&sem_count, 0,1);
31 sem_init(&sem_barrier, 0, 0);
32 pthread_t thread[NUM_THREADS];
33 threadParm_t threadParm[NUM_THREADS]; //´«µÝ²ÎÊý
34 int i;
35 for (i=0; i<NUM_THREADS; i++)
36 {
37 threadParm[i].threadId = i;
38 pthread_create(&thread[i], NULL, threadFunc, (void
39 *)&threadParm[i]);
40 }
41 for (i=0; i<NUM_THREADS; i++)
42 {
43 pthread_join(thread[i], NULL);
44 }
45 sem_destroy(&sem_count);
46 sem_destroy(&sem_barrier);
47 return 0;
48 }
```

2.3 忙等待和互斥量实现路障 Barrier 的功能

这里同样采取一个变量来计数，并采用忙等待的方法来等所有线程执行完第一步。当计数器到线程数之后，忙等待停止等待开始输出第二步。

```
1 typedef struct {
2     int threadId;
3 } threadParm_t;
4 int count=0; //表示正在运行的线程数;
5
6 pthread_mutex_t barrier_mutex;
7
8 void *threadFunc(void *parm) {
9     threadParm_t *p = (threadParm_t *) parm;
10    fprintf(stdout, "Thread %d has entered step 1.\n", p->threadId);
11    pthread_mutex_lock(&barrier_mutex);
12    count++;
13    pthread_mutex_unlock(&barrier_mutex);
14    while (count < NUM_THREADS);
15    fprintf(stdout, "Thread %d has entered step 2.\n", p->threadId);
16    pthread_exit(NULL);
17 }
18
19 int main(int argc, char *argv[]) {
20    pthread_mutex_init(&barrier_mutex, NULL);
21    pthread_t thread[NUM_THREADS];
22    threadParm_t threadParm[NUM_THREADS];
23    int i;
24    for (i = 0; i < NUM_THREADS; i++) {
25        threadParm[i].threadId = i;
26        pthread_create(&thread[i], NULL, threadFunc, (void
27        *) &threadParm[i]);
28    }
29    for (i = 0; i < NUM_THREADS; i++) {
30        pthread_join(thread[i], NULL);
31    }
```

```
32      pthread_mutex_destroy(&barrier_mutex);
33      return 0;
34 }
```

3 结果分析

3.1 信号量与 Barrier 的异同

相同点：都是在线程执行完第一步的时候等待其他线程全完成再执行第二步，使得所有线程再某一时间段同步。

不同点：barrier 中 wait() 由每个线程主动调用，不同线程之间不会受到影响。信号量使用 wait 和 post 函数，需要等待其他线程调用公共资源。

3.2 忙等待和互斥量与 Barrier 的异同

相同点：都是在线程执行完第一步的时候等待其他线程全完成再执行第二步，使得所有线程再某一时间段同步。

不同点：barrier 中 wait() 由每个线程主动调用，不同线程之间不会受到影响。而，忙等待和互斥量会需要等待其他线程调用完公共资源，同时忙等待使 CPU 利用率低。