南開大學
Nankai University

计算机学院

并行程序设计实验报告报告

高斯消去的 Pthread 优化

姓名： 高祎珂

学号：2011743

专业：计算机科学与技术

2022 年 5 月 6 日

目录

1 实验介绍

1.1 实验简介

对于高斯消去在 SIMD 里已经有过详细介绍，此处不再叙述，Pthread 实验就是在串行算法的基础上，进行多线程优化，我觉得这个优化原理和 SIMD 很像，因为高斯消去的二重循环和三重循环内部计算没有顺序影响，可以并行计算，所以可以进行多线程优化，而多线程与 SIMD 比较起来，我认为，其优势在于可以不同的线程执行不同的函数操作，但是再次实验中并没有体现出来，不过对于多线程我们可以自己指定线程数，而不用局限于 SSE 等语言所限定的固定向量化操作数。

1.2 实验设计

1. 首先生成后续进行高斯消去的矩阵，利用伪代码实现高斯消去串行算法。
2. 进行动态线程、静态线程 + 信号量同步、静态线程 + 信号量同步 + 三重循环全部纳入线程函数、静态线程 +barrier 同步算法的编程实现，并且在 ARM 平台上运行、分析，比较不同矩阵规模和不同线程数的影响。
3. 对于上述的四种算法中，加入 SIMD 操作，基于上次实验，三重循环向量化的效果比较好，所以此次仅进行三重循环的向量化，运行分析，perf 分析。
4. 在 X86 平台上进行同样操作，分析在 X86 平台上的优化表现，并且利用 VTune 工具进行更细致的分析，最后比较一下两平台。

1.3 理论分析

如果开辟出 n 个子线程，不考虑开辟销毁线程过程中的损失以及其他消耗，理论上应该能达到 n 倍的加速比，但是由于线程的开销，达不到这么高的优化。动态线程在不断的开辟和摧毁线程，这个过程中的消耗会比静态线程的大，至于线程开销对于最终结果有多大的影响，这个可以通过实验得出。另外加上 SIMD 优化，根据上次实验，可以大致推算能达到 3 倍左右的效率提升，接下来根据实验来验证假设。

2 ARM 平台

2.1 算法设计

完整代码上传到了github，这里进行核心代码的展示。

动态线程

```
void *thread_func(void *parm){
  thread_art*p=(thread_art*)parm;
  int k=p->k;
  int id=p->id;
  for(int i=k+id+1;i<n;i+=thread_count){
      for(int j=k+1;j<n;j++)
          B[i][j]-=B[i][k]*B[k][j];
      B[i][k]=0;
```

```
    }
    pthread_exit(nullptr);
  }
  for (k = 0; k<n; k++){
  for (int j = k + 1; j < n; j++)
    B[k][j] /= B[k][k];
  B[k][k] = 1.0;
    thread_art*param=new thread_art[thread_count];
    pthread_t*handles=new pthread_t[thread_count];
    for(int i=0;i<thread_count;i++){
        param[i].id=i;
        param[i].k=k;
    }
    for(int i=0;i<thread_count;i++)
        pthread_create(&handles[i],nullptr,thread_func,(void*)&param[i]);
    for(int i=0;i<thread_count;i++)
            pthread_join(handles[i],nullptr);
    }
```

静态线程 + 信号量同步

```
  void *thread_func(void *parm){
    thread_art*p=(thread_art*)parm;
    int id=p->t_id;
    for(int k=0;k<n;++k){
        sem_wait(&sem_workerstart[id]);
        for(int i=k+1+id;i<n;i+=thread_count){
            for(int j=k+1;j<n;j++)
                A[i][j]-=A[i][k]*A[k][j];
            A[i][k]=0.0;
        }
        sem_post(&sem_main);
        sem_wait(&sem_workerend[id]);
    }
    pthread_exit(nullptr);
  }
  int main()
  {
    int flag=sem_init(&sem_main,0,0);
    for(int i=0;i<thread_count;i++){
        int flag1=sem_init(&sem_workerstart[i],0,0);
        int flag2=sem_init(&sem_workerend[i],0,0);
    }
    pthread_t *handles=new pthread_t[thread_count];
    thread_art *param=new thread_art[thread_count];
    for(int i=0;i<thread_count;i++){
        param[i].t_id=i;
        pthread_create(&handles[i],nullptr,thread_func,(void*)&param[i]);
    }
```

```
    for(int k=0;k<n;++k){
        for(int j=k+1;j<n;j++)
            A[k][j]/=A[k][k];
        A[k][k]=1.0;
        for(int t_id=0;t_id<thread_count;t_id++)
            sem_post(&sem_workerstart[t_id]);
        for(int i=0;i<thread_count;i++)
            sem_wait(&sem_main);
        for(int i=0;i<thread_count;i++)
            sem_post(&sem_workerend[i]);
    }
    for(int i=0;i<thread_count;i++)
        pthread_join(handles[i],nullptr);
    for(int i=0;i<thread_count;i++)
        sem_destroy(&sem_workerstart[i]);
}
```

静态线程 + 信号量同步 + 三重循环全部纳入线程函数

```
sem_t sem_leader;
sem_t *sem_Division=new sem_t[thread_num-1];
sem_t *sem_Elimination=new sem_t[thread_num-1];
void *thread_func2(void *parm){
    thread_art*p=(thread_art*)parm;
    int id=p->t_id;
    for(int k=0;k<n;++k){
        if(id==0){
            for(int j=k+1;j<n;j++)
                B[k][j]/=B[k][k];
            B[k][k]=1.0;
        }
        else
            sem_wait(&sem_Division[id-1]);
        if(id==0){
            for(int i=0;i<thread_num-1;i++)
                sem_post(&sem_Division[i]);
        }
        for(int i=k+1+id;i<n;i+=thread_num){
            for(int j=k+1;j<n;j++)
                B[i][j]-=B[i][k]*B[k][j];
            B[i][k]=0.0;
        }
        if(id==0){
            for(int i=0;i<thread_num-1;i++)
                sem_wait(&sem_leader);
            for(int i=0;i<thread_num-1;i++)
                sem_post(&sem_Elimination[i]);
        }
        else{
```

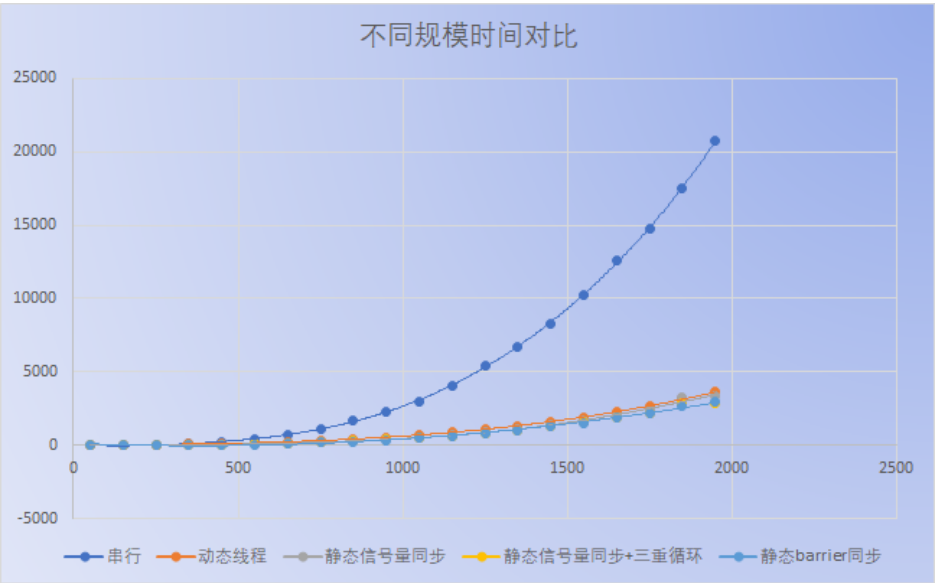
```
            sem_post(&sem_leader);
            sem_wait(&sem_Elimination[id-1]);
        }
    }
    pthread_exit(nullptr);
}
```

静态线程 +barrier 同步

```
pthread_barrier_t Divsion;
pthread_barrier_t Elimination;
void *thread_func3(void *parm){
  thread_art*p=(thread_art*)parm;
  int id=p->t_id;
  for(int k=0;k<n;++k){
      if(id==0){
          for(int j=k+1;j<n;j++)
              D[k][j]/=D[k][k];
          D[k][k]=1.0;
      }
      pthread_barrier_wait(&Divsion);
      for(int i=k+1+id;i<n;i+=thread_num){
          for(int j=k+1;j<n;j++)
              D[i][j]-=D[i][k]*D[k][j];
          D[i][k]=0.0;
      }
      pthread_barrier_wait(&Elimination);
  }
  pthread_exit(nullptr);
}
```

2.2 不同规模下结果分析

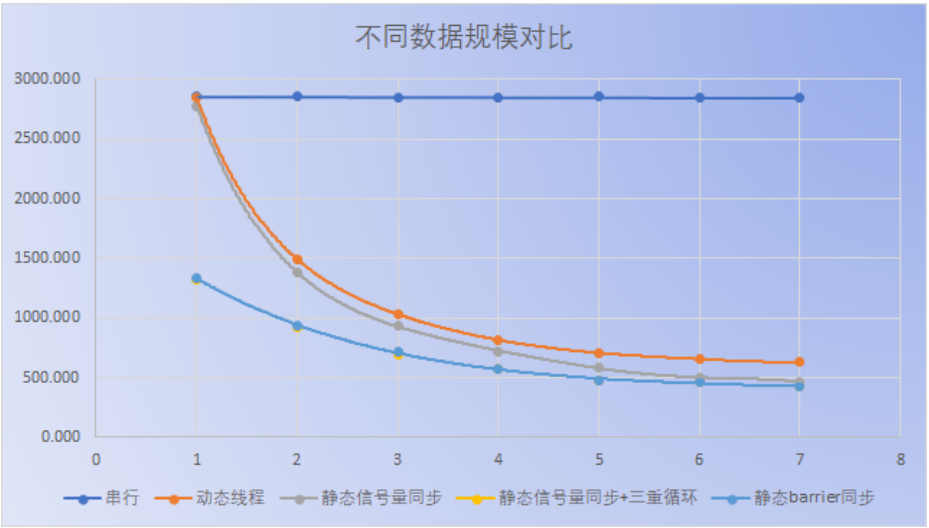
子线程数	串行/ms	动态线程/ms	静态线程 + 信号量同步/ms	静态信号量同步 + 三重循环/ms	静态 +barrier 同步/ms
50	0.32355	12.53304	1.75586	2.11456	1.32869
150	8.59709	37.65077	5.68981	5.97453	4.60569
250	39.68637	63.64156	15.16471	13.60887	11.71606
350	108.67461	94.70711	29.64727	28.65636	24.28334
450	234.61621	133.03383	54.73326	52.04141	46.40007
550	430.312	184.248	91.434	87.517	79.594
650	747.044	247.186	140.703	134.387	127.446
750	1139.917	327.727	205.389	195.683	190.136
850	1688.434	430.387	296.918	283.546	280.405
950	2268.845	555.202	410.340	383.587	378.116
1050	3010.639	731.662	538.677	515.863	490.242
1150	4077.728	896.713	723.199	618.416	615.427
1250	5464.151	1102.049	887.802	846.745	834.988
1350	6729.255	1346.343	1125.106	1045.116	1051.250
1450	8295.653	1615.194	1386.123	1289.918	1296.082
1550	10220.282	1950.732	1690.657	1559.431	1550.771
1650	12598.386	2296.560	2027.298	1882.345	1892.067
1750	14775.873	2696.450	2402.790	2237.714	2240.648
1850	17507.031	3146.776	3250.352	2932.486	2646.785
1950	20720.113	3634.407	3335.021	2924.776	2943.257



此结果是设置子线程数为 7 时得到的, 静态 +barrier 同步的优化效果较为显著，动态线程由于不断创建和销毁线程，倒是开销较大，优化效果没有后几个线程函数好，但是随着数据规模的增大，其优化效果也在逐渐向静态线程优化靠拢，观察数据可以看出，在鲲鹏上，线程消耗好像很小，在数据规模较大时，几乎达到了 7 倍，但是我考虑也有可能是因为直接在服务器上进行运行，没有生成测试节点（因为当时服务器出问题了），整体效果比较符合预期。下面看一下不同线程数的影响。

2.3 不同线程数下结果分析

子线程数	串行/ms	动态线程/ms	静态线程 + 信号量同步/ms	静态信号量同步 + 三重循环/ms	静态 +barrier 同步/ms
7	2846.796	634.933	466.125	429.673	426.017
6	2851.647	656.119	501.038	468.475	465.763
5	2852.371	707.819	584.973	477.778	476.056
4	2851.289	817.035	720.528	572.284	574.071
3	2851.090	1031.385	934.292	699.643	718.114
2	2852.742	1491.998	1379.223	929.951	933.498
1	2853.020	2844.810	2776.050	1321.745	1333.700



基于上述实验，这里选择了矩阵规模为 1000 进行实验测试，基于鲲鹏一个计算核心有 8 个线程，所以设置的子线程为从 7 开始递减，从实验数据可以看出，随着线程数的减少，pthread 的函数运行时间都在不断增加，，在子线程数 >=2 时，时间增加的缓慢，这是因为还存在一定程度的并行化，且减少线程可减小开辟和销毁线程的开销，这在动态线程函数上体现较为明显，在 n 为 7 时，动态线程效率比静态低很多，但是在线程数变小时，动态线程的效率几乎与静态线程相同，而较为反常的是后面列静态线程，在子线程数为 1 时，效率仍然几乎是串行的两倍，这是因为在子线程数为 1 时，二者和前面两个函数设置中，主线程和子线程互斥，只有一个子线程，相当于在串行执行，只不过两个线程执行不同的任务。而对于静态信号量同步 + 三重循环和静态 +barrier 同步的线程函数是主线程后续也会参与到消去运算，子线程为 1 时，相当于依旧是两个线程参与运算，所以最终结果仍然接近于串行的 2 倍。

2.4 Pthread+SIMD

几个算法的 SIMD 加入的算法设计基本相同，这里是一个 barrier 的 SIMD 修改。完整代码上传到了github

三重循环 SIMD

```
1  pthread_barrier_t Divsion;
2  pthread_barrier_t Elimination;
3  void *thread_func(void *parm){
```

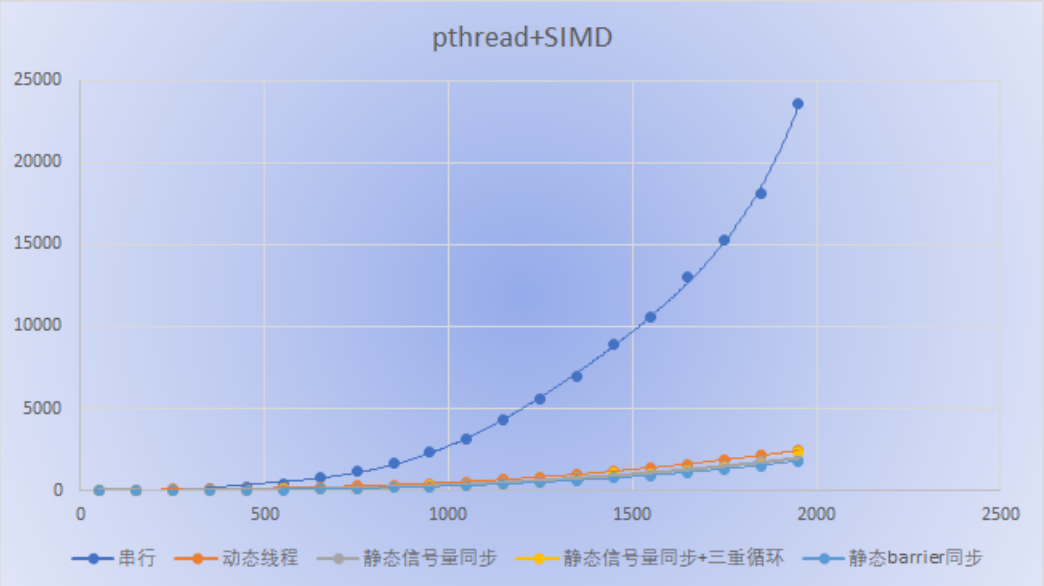
```
    thread_art*p=(thread_art*)parm;
    int id=p->t_id;
    for(int k=0;k<n;++k){
        if(id==0){
            for(int j=k+1;j<n;j++)
                D[k][j]/=D[k][k];
            D[k][k]=1.0;
        }
        pthread_barrier_wait(&Divsion);
        for(int i=k+1+id;i<n;i+=thread_num){
        //核心更改代码
            float32x4_t vaik=vdupq_n_f32(C[i][k]);
            int j=k+1;
            for(j=k+1;j+4<=n;j+=4)
            {
                float32x4_t vakj=vld1q_f32(&C[k][j]);
                float32x4_t vaij=vld1q_f32(&C[i][j]);
                float32x4_t vx=vmulq_f32(vakj,vaik);
                vaij=vsubq_f32(vaij,vx);

                vst1q_f32(&C[i][j], vaij);

            }
            while(j<n)
            {
                C[i][j]-=C[k][j]*C[i][k];
                j++;
            }
            C[i][k]=0;
        }
        pthread_barrier_wait(&Elimination);
    }
    pthread_exit(nullptr);
}
```

运行得到的结果如下:

子线程数	串行/ms	动态线程/ms	静态线程 + 信号量同步/ms	静态信号量同步 + 三重循环/ms	静态 +barrier 同步/ms
50	0.35306	11.81894	1.82841	1.58658	1.21287
150	8.93935	37.5277	6.51539	5.75251	4.20606
250	41.2419	63.5955	13.0156	12.2469	9.1904
350	113.6433	91.1695	24.2555	24.0892	18.0463
450	242.899	124.150	41.316	40.441	32.488
550	446.974	165.593	65.371	64.551	54.578
650	741.790	214.983	99.862	98.661	81.244
750	1147.143	276.383	146.764	141.109	124.104
850	1715.326	346.233	212.733	193.111	172.182
950	2354.903	435.391	292.250	262.242	235.539
1050	3186.095	543.044	374.156	339.780	311.280
1150	4307.748	677.621	466.184	427.788	399.940
1250	5581.832	821.917	601.465	528.520	505.750
1350	6953.246	995.451	749.082	660.108	620.107
1450	8880.361	1169.055	902.106	1092.554	761.281
1550	10597.035	1391.383	1086.211	959.630	926.065
1650	13049.665	1614.411	1281.720	1135.468	1100.236
1750	15243.612	1884.739	1511.981	1358.791	1303.117
1850	18059.260	2139.521	1770.976	1559.382	1516.547
1950	23560.579	2443.096	1906.976	2329.657	1813.476



这里线程数依旧设置为 7，加入 SIMD 整体变化规律与只进行 pthread 相同，下面看一下 pthread 和 pthread+SIMD 的对比。

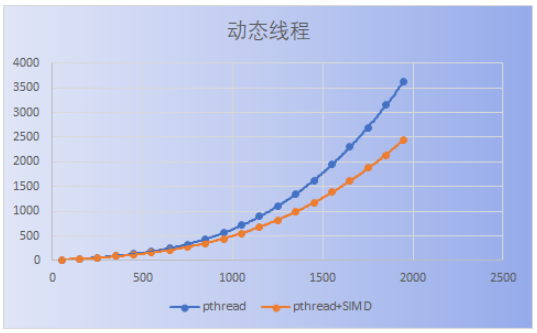


图 2.1: 动态线程

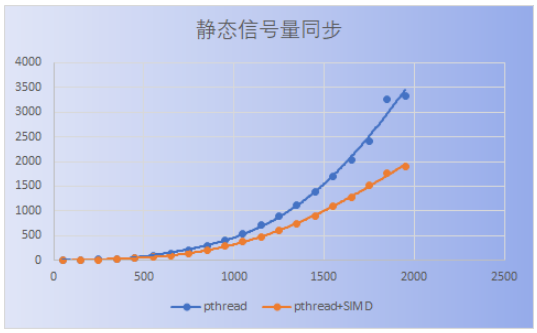


图 2.2: 静态信号量同步

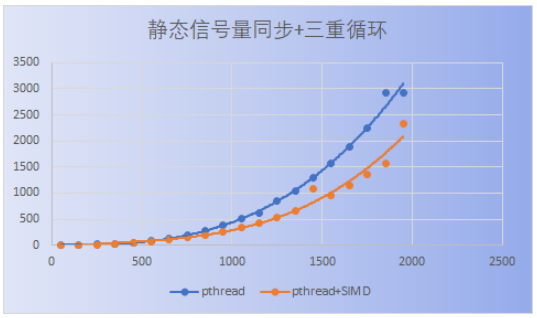


图 2.3: 静态信号量同步 + 三重循环

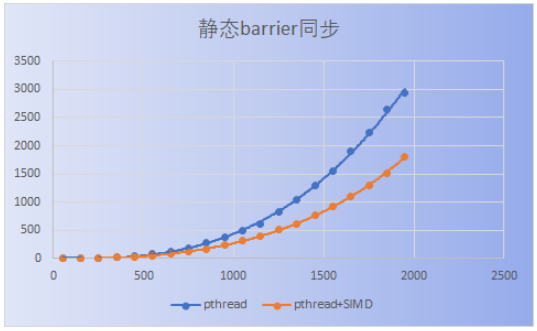


图 2.4: 静态 barrier 同步

SIMD 是进行了三重循环的 SIMD（基于上次实验，三重循环向量化的效果更显著），基于上次实验，可以得出能达到 2 倍左右的优化效果，但是对于此实验而言，似乎并没有如此大的优化效果，只有 1.5 倍左右的优化，我觉得可能是因为线程占用资源较大，导致本应该的向量化但其实只是进行了虚拟的向量化

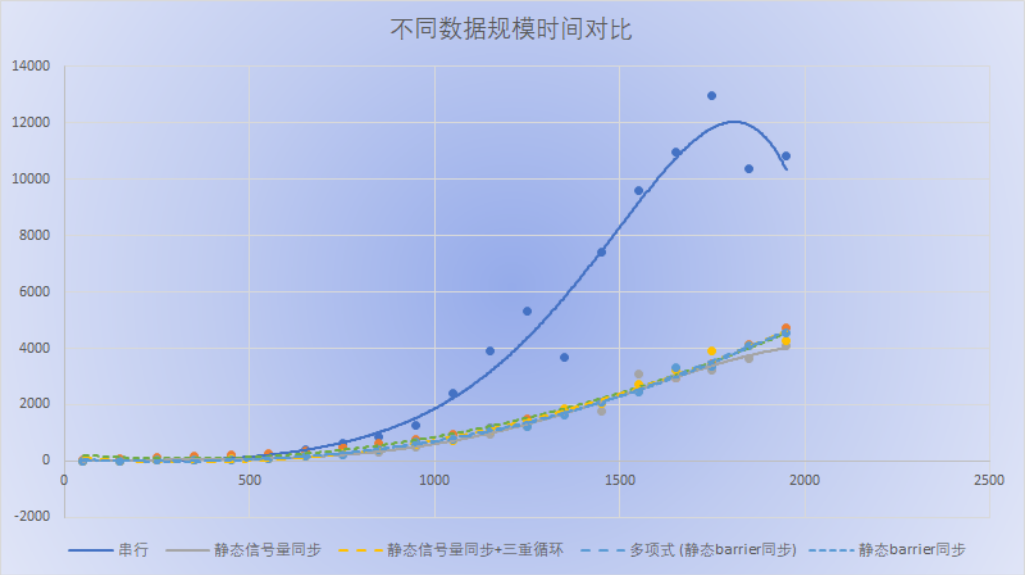
3 X86 平台

3.1 算法设计

与 ARM 平台的算法设计基本相同，只是在 SIMD 设计时更换一下语言，本次实验只用了 SSE 指令集，源码链接为github, 这里不再进行代码的展示。

3.2 不同规模下结果分析

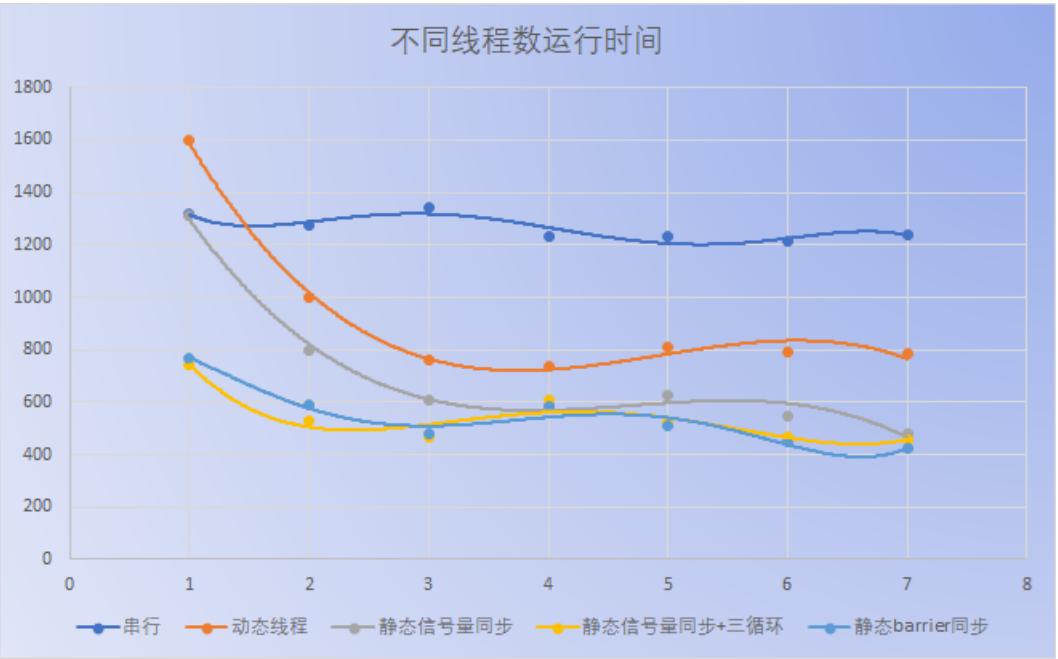
数据规模	串行/ms	动态线程/ms	静态线程 + 信号量同步/ms	静态信号量同步 + 三重循环/ms	静态 +barrier 同步/ms
50	0.1859	23.5839	4.5267	3.0091	2.9126
150	4.1902	63.4619	9.5239	7.6817	7.1481
250	22.5166	106.456	20.1851	16.0766	14.9369
350	57.1968	156.462	36.6676	30.1192	27.8713
450	124.635	219.393	62.8821	63.1699	57.0547
550	219.888	283.26	88.1868	103.133	97.825
650	390.812	371.8	162.696	152.066	158.469
750	619.526	483.268	231.353	242.997	235.536
850	846.255	610.092	325.209	383.964	387.407
950	1282.53	763.455	490.417	567.137	670.975
1050	2410.15	949.787	819.664	708.649	753.424
1150	3911.83	1184.34	925.774	1070.65	1197.04
1250	5305.54	1474.11	1208.12	1251.91	1204.81
1350	3699.28	1776.6	1702	1851.27	1646.68
1450	7396.16	2085.72	1776.81	2057.77	2095.92
1550	9605.84	2480.85	3086.35	2747.58	2469.38
1650	10950.9	2973.15	2947.12	3235.55	3307.69
1750	12973.1	3453.17	3243.38	3896.08	3354.61
1850	10355.7	4158.83	3620.12	4087.61	4096.61
1950	10841.9	4712.37	4102.43	4284.91	4548.63



此结果是设置子线程数为 7 时得到的，上图和上表中可以看到整体上大致符合预期，和 ARM 平台的整体规律基本一样，但是这个的优化效果明显没有 ARM 的好，而且我觉得这个优化效果是比较合理的（线程创建的消耗），但是由于动态线程不断地新建和销毁线程，会导致数据规模较小时，这个线程开销比并行化的效果影响要大，所以结果最差，矩阵规模小时，效率还不如串行，但是随着矩阵规模的增大，pthread 逐渐体现出了优势，矩阵规模在 1000 左右，静态优化效果达到了将近 4 倍。三重循环的优化效果在线程数较大时效果并不是很明显，看一下不同线程的结果。

3.3 不同线程数下结果分析

子线程数	串行/ms	动态线程/ms	静态线程 + 信号量同步/ms	静态信号量同步 + 三重循环/ms	静态 +barrier 同步/ms
7	1237.2	783.079	483.102	458.389	422.719
6	1214.94	790.426	548.668	468.538	451.767
5	1230.5	813.333	624.776	520.096	508.503
4	1233.22	736.134	587.486	605.553	586.207
3	1343.07	762.455	606.351	469.196	477.12
2	1278.22	997.933	799.641	528.604	589.821
1	1316.93	1598.02	1309.96	740.639	767.624



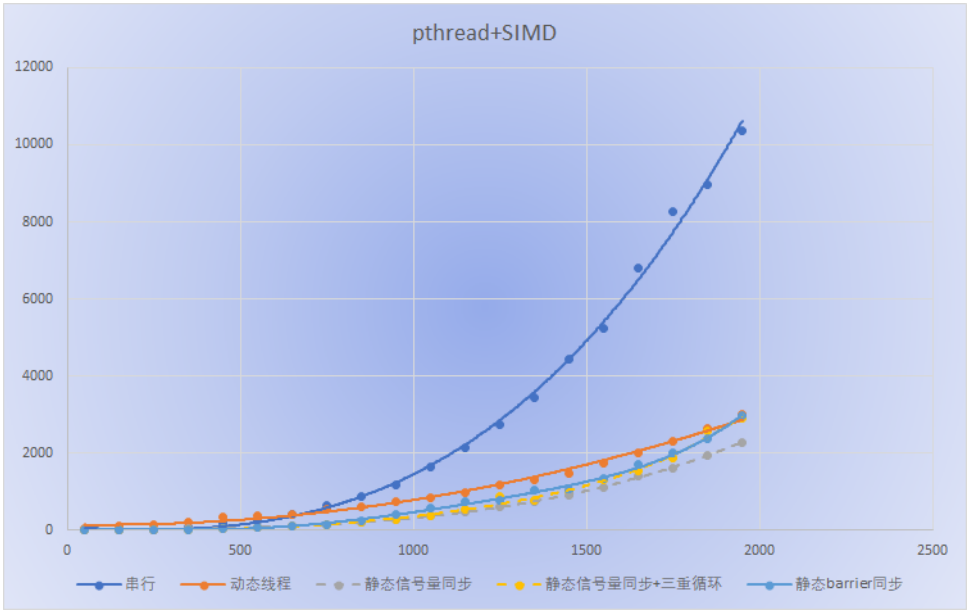
基于上述实验，这里也是选择了矩阵规模为 1000 进行实验测试，在子线程数 >=2 时，可以看到动态线程和静态线程 + 信号量同步都有优化，动态线程用时随着线程数的减少呈先上升后下降的趋势，这是因为线程数的减少虽然并行化程度减小了，但是线程的开销也减小了。在子线程数为 1 时，二者和串行结果基本一样，这是因为在这两个函数设置中，主线程和子线程互斥，只有一个子线程，相当于在串行执行，只不过两个线程执行不同的任务。

而对于静态信号量同步 + 三重循环和静态 +barrier 同步在子线程为 1 时仍有优化，且大致为串行的 2 倍，这是因为在函数设置过程中，这俩的线程函数是主线程后续也会参与到消去运算，子线程为 1 时，相当于依旧是两个线程参与运算，所以最终结果仍然接近于串行的 2 倍。

3.4 Pthread+SIMD

源码 github 链接戳这里。

数据规模	串行/ms	动态线程/ms	静态线程 + 信号量同步/ms	静态信号量同步 + 三重循环/ms	静态 +barrier 同步/ms
50	0.1919	44.312	5.0947	5.1509	6.4309
150	5.0867	109.013	10.4874	10.7405	13.7754
250	30.8042	147.324	19.2666	17.5857	18.3299
350	80.6914	210.636	33.638	31.2089	24.5391
450	191.67	352.091	53.028	46.8052	47.4387
550	244.85	387.914	82.3741	71.7293	76.0972
650	415.475	430.784	116.26	107.913	102.005
750	663.08	545.805	163.033	147.028	148.65
850	873.213	625.543	201.057	223.893	254.219
950	1184.91	737.635	288.387	296.05	413.726
1050	1638.6	863.598	371.458	380.661	595.967
1150	2150.7	995.81	482.413	545.581	746.887
1250	2755.17	1192.69	606.897	864.045	785.813
1350	3448.42	1310.43	738.845	776.182	1055.96
1450	4446.39	1488.78	924.216	1011.04	1030.77
1550	5237.15	1739.71	1116.71	1299	1334.99
1650	6805.92	2005.47	1423.43	1558.2	1702.21
1750	8278.36	2310.62	1620.51	1869.43	2005.39
1850	8970.66	2628.19	1956.69	2589.89	2384.47
1950	10362.3	3008.1	2288.85	2908.59	2991.52



可以看到加入了 SIMD，整体效率的变化规律和未加入之前基本一样，对比上述的表格可以发现 SIMD 有一定程度上的优化，但是优化效果并不如只进行 SIMD 的效果好，这和 ARM 平台保持一致，下面是二者对比分析的图表

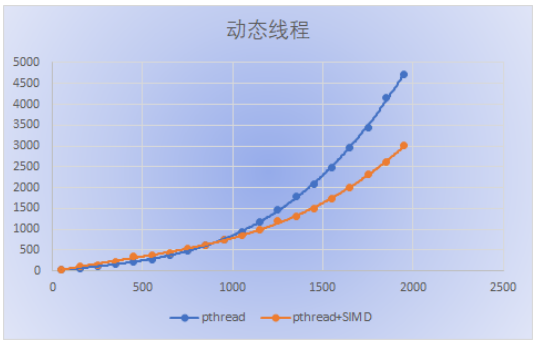


图 3.5: 动态线程

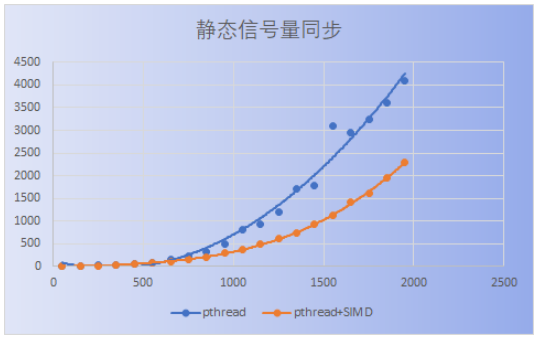


图 3.6: 静态信号量同步

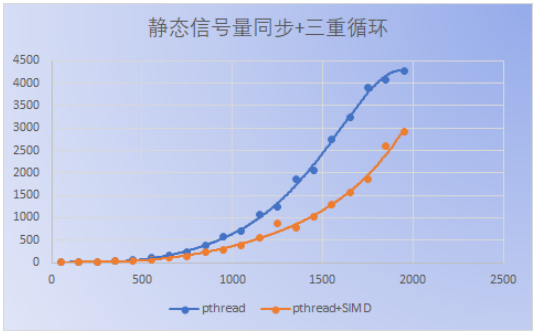


图 3.7: 静态信号量同步 + 三重循环

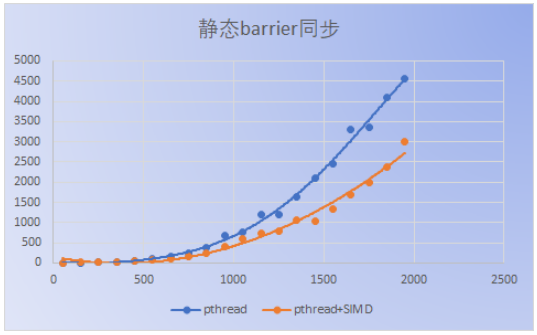


图 3.8: 静态 barrier 同步

3.5 VTune 性能剖析

在 VTune 中进行性能分析，得到结果如下图所示

Analysis Configuration | Collection Log | Summary | Bottom-up | Event Count | Platform

Grouping: Function / Thread / Logical Core / Call Stack

Function / Thread / Logical Core...	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate
thread_func3	137.406s	315,439,800,000	679,192,800,000	0.464
thread_func2	135.911s	313,677,000,000	679,393,000,000	0.462
thread_func	124.682s	291,525,000,000	678,688,400,000	0.430
thread_funcd	104.495s	268,268,000,000	647,574,200,000	0.414
m_reset	64.081s	198,278,600,000	779,370,800,000	0.254
chuanxing	55.430s	174,907,200,000	679,559,400,000	0.257

Function / Thread / Logical Core / Call Stack	CPU Time ▼	Clockticks	Instructions Retired	CPI Rate
thread_func3	137.406s	315,439,800,000	679,192,800,000	0.464
thread_func2	135.911s	313,677,000,000	679,393,000,000	0.462
Thread (TID: 34872)	3.349s	7,532,200,000	15,756,000,000	0.478
Thread (TID: 34932)	3.337s	7,443,800,000	15,901,600,000	0.468
Thread (TID: 16444)	3.304s	7,469,800,000	15,740,400,000	0.475
Thread (TID: 34892)	3.267s	7,196,800,000	15,771,600,000	0.456
Thread (TID: 40156)	3.255s	7,467,200,000	15,792,400,000	0.473
Thread (TID: 34908)	3.252s	7,459,400,000	15,737,800,000	0.474
Thread (TID: 34916)	3.227s	7,404,800,000	15,691,000,000	0.472
Thread (TID: 34884)	3.203s	7,324,200,000	15,701,400,000	0.466
Thread (TID: 32660)	2.873s	6,448,000,000	13,457,600,000	0.479
Thread (TID: 32664)	2.858s	6,323,200,000	13,449,800,000	0.470
Thread (TID: 32616)	2.845s	6,448,000,000	13,613,600,000	0.474
Thread (TID: 32648)	2.825s	6,377,800,000	13,483,600,000	0.473

thread_func2 是静态信号量同步 + 三重循环加入线程函数，可以看到创建出的不同线程运行时间都差不多，任务基本平均分配到了各个线程中，且从图中可以看出，各个线程之间有一定的顺序关系，这和预想相符。

4 总结

通过本次实验，验证了理论课上对 pthread 的学习，并且对于信号量和 barrier 的同步机制有了更深刻的认识。在实验中也遇到了很多问题，最开始由于对于信号量的操作不熟悉，导致结果一直不能输出正确，还有这个过程中 k 在不同线程中的取值，以及任务划分在不同循环中各个变量的临界，这些导致结果矩阵一直不正确。但是做完实验还有一些没解决的疑问，SIMD 为什么没有达到预期的效果，优化效果并不是很理想，目前理解可能与操作系统内部有关，这有待于后期继续学习。