《Pthread 编程练习实验报告》

并行第三次实验作业

题　　目：	Pthread 编程练习实验报告
上课时间：	周一下午
授课教师：	孙永谦
姓　　名：	张怡桢
学　　号：	2013747
年　　级：	2020级本科生
日　　期：	2022/11/20

Pthread 编程练习实验报告

张怡桢，2013747

南开大学软件学院

1 实验内容

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。
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。
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。

2 实验环境

MacOs ARM架构环境完成多个数组排序的实验；

但是由于ARM架构无SSE，故在高斯结合SSE实验，使用Linux完成，基于X86架构。

3 多个数组排序

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。

3.1 问题描述

对ARR_NUM个长度为ARR_LEN的一维数组进行排序，使用Pthread多线程编程，每个线程处理一部分数组。若数组分布不均衡，如前二分之一的数组全部升序，后二分之一的数组全部逆序，则每个线程分得的任务负载可能也会不均衡。故本实验采取静态划分、动态划分、粗粒度动态划分三个角度，探索任务分块大小、线程数多少、静态动态多线程结合等方面因素对数组排序效率的影响。

3.2 实验初始化

3.2.1 均衡初始化

各个数组内数据顺序基本无差异。将数组长度设为 10000，生成 10000 个数组。

```
void init_1(void){
    srand(unsigned(time(nullptr)));
    for (int i = 0; i < ARR_NUM; i++) {
        arr[i].resize(ARR_LEN);
        for (int j = 0; j < ARR_LEN; j++){
            arr[i][j] = rand();
        }
    }
}
```

3.2.2 不均衡初始化

初始化 ARR_NUM 行，ARR_LEN 列的数组。

当行数 i 属于 0~2499 时, ratio = 0，此时 2500 行内完全升序排列；当

行数 i 属于 2500~4999 时, ratio = 32 此时随机有 2500 * 1/4 行升序， 2500 * 3/4 行降序;

当 行数 i 属于 5000~7499 时，ratio = 64，此时随机有 2500 * 1/2 行升序，2500*1/2 行降序;

当行数 i 属于 7500~9999 时，ratio = 128，此时 2500 行完全降序。此时数组可分为四块，数组负载不均衡。

```
const int seg = ARR_NUM / THREAD_NUM;
// 初始化待排序数组，使得
// 第一段：完全升序
// 第二段：1/4逆序，3/4升序
// 第三段：1/2逆序，1/2升序
// 第四段：完全逆序
void init_2(void){
    int ratio;
    srand(unsigned(time(nullptr)));
    for (int i = 0; i < ARR_NUM; i++){
        arr[i].resize(ARR_LEN);
        if(i < seg)ratio = 0;
        else if(i < seg * 2)ratio = 32;
        else if(i < seg * 3)ratio = 64;
        else ratio = 128;
        if((rand() & 127) < ratio){
```

```
            for(int j = 0; j < ARR_LEN; j++){
                arr[i][j] = ARR_LEN - j;
            }
        }
        else{
            for(int j = 0; j < ARR_LEN; j++){
                arr[i][j] = j;
            }
        }
    }
}
```

3.2.3 时间计算

使用mac系统，选择的计时器为gettimeofday()，头文件为#include <sys/time.h>，它的精度可以达到微妙，是C标准库的函数。

```
struct timeval tpstart,tpend;
gettimeofday(&tpstart,NULL);
gettimeofday(&tpend,NULL);
double timeuse = 1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
printf("used time:%fus\n",timeuse);
```

3.3 算法设计与实现

3.3.1 块划分

每个线程负责连续的 ARR_NUM / THREAD_NUM 个数组的排序。实现代码如下：

```
// 块划分：每个线程负责连续n/4个数组的排序
void *arr_sort_1(void *parm){
    threadParm_t *p = (threadParm_t *) parm;
    int r = p->threadId;
    long long tail;
    for (int i = r * seg; i < (r + 1) * seg; i++){
        stable_sort(arr[i].begin(), arr[i].end());
    }
    pthread_mutex_lock(&mutex_lock);
    gettimeofday(&tpend,NULL);
    result[0][p->threadId] = (1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-
tpstart.tv_usec)/1000;//注意，秒的读数和微秒的读数都应计算在内
    pthread_mutex_unlock(&mutex_lock);
    pthread_exit(nullptr);
```

```
}
```

3.3.2 动态划分

动态分配任务，每个线程每次分配到一个任务后执行数组排序，完成后继续领取新的任务， 直至所有数组均已排序完成。同时利用 mutex 互斥量加锁，保证任务分配时访问临界资源不出差。核心代码如下：

```
// 动态获取：加锁，每个线程动态获取任务
void *arr_sort_2(void *parm) {
    threadParm_t *p = (threadParm_t *) parm;
    int task = 0;
    while (1) {
        pthread_mutex_lock(&mutex_task);
        task = next_arr++;
        pthread_mutex_unlock(&mutex_task);
        if (task >= ARR_NUM) break;
        stable_sort(arr[task].begin(), arr[task].end());
    }
    pthread_mutex_lock(&mutex_lock);
    gettimeofday(&tpend,NULL);
    result[1][p->threadId] = (1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-
tpstart.tv_usec)/1000;//注意，秒的读数和微秒的读数都应计算在内


    pthread_mutex_unlock(&mutex_lock);
    pthread_exit(nullptr);
}
```

3.3.3 粗粒度划分

线程动态领取任务,线程一次性领取 taskNum 个数组进行排序。核心代码如下：

```
// 粗粒度动态获取：加锁，每个线程动态获取任务，一次性获取一部分任务
void *arr_sort_3(void *parm) {
    threadParm_t *p = (threadParm_t *) parm;
    int startTask = 0;
    int endTask = 0;
    long long tail;
    while (1) {
        pthread_mutex_lock(&mutex_task);
        startTask = next_arr;
        //每次分配100行
        endTask = next_arr + taskNum;
        next_arr = endTask;
        pthread_mutex_unlock(&mutex_task);
```

```
            if(startTask >= ARR_NUM) break;
            for(;startTask<endTask;startTask++){
                stable_sort(arr[startTask].begin(), arr[startTask].end());
            }
        }
        pthread_mutex_lock(&mutex_lock);
        gettimeofday(&tpend,NULL);
        result[2][p->threadId] = (1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-
tpstart.tv_usec)/1000;//注意，秒的读数和微秒的读数都应计算在内

        pthread_mutex_unlock(&mutex_lock);
        pthread_exit(nullptr);
}
```

3.4 实验设计与结果

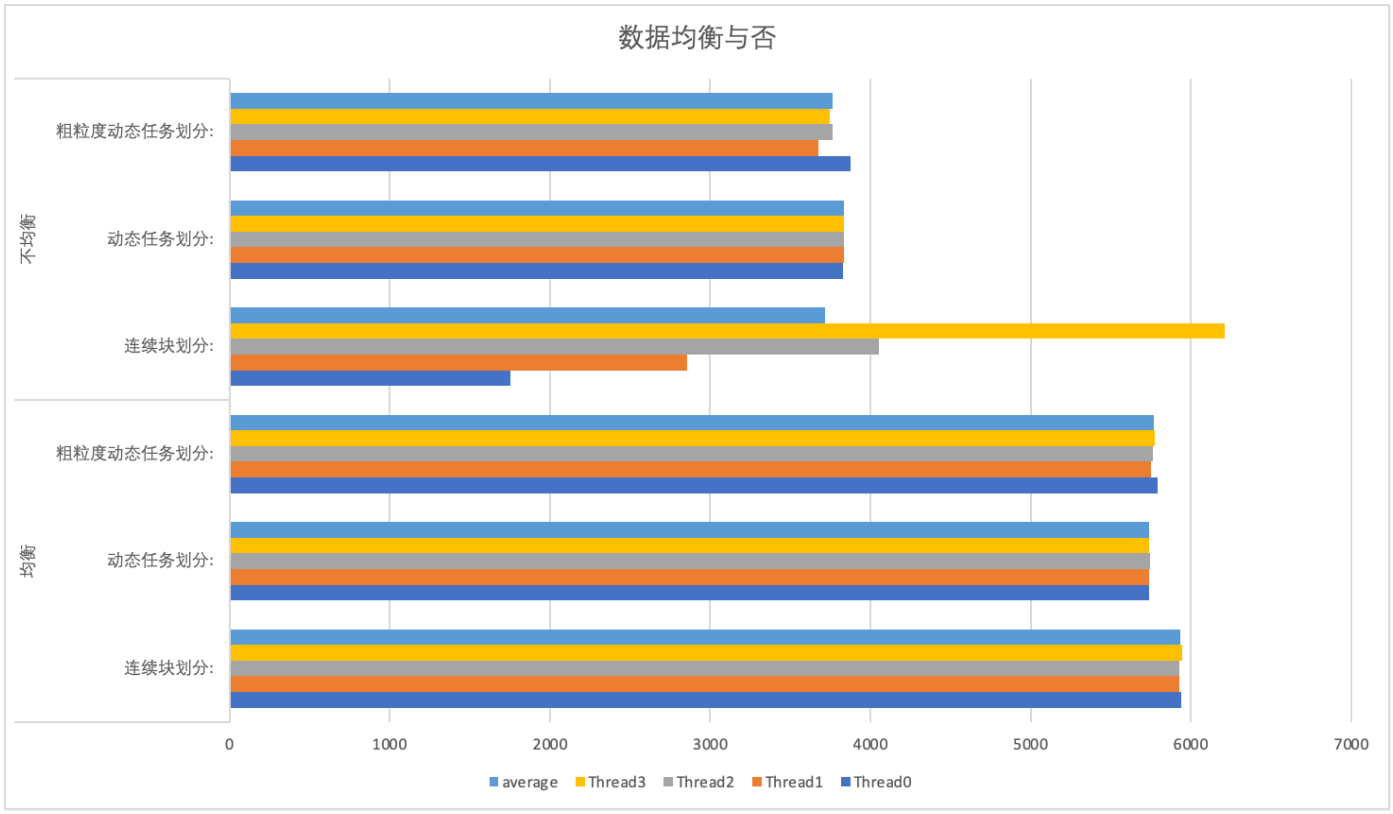
可从任务分块的大小、线程数的多少、静态动态多线程结合等方面进行尝试，探索规律。

3.4.1 数据是否均衡

数据的规模如下，有10000个数组，每个数组长度为10000，共4个线程。

```
const int ARR_NUM = 10000;
const int ARR_LEN = 10000;
const int THREAD_NUM = 4;
const int seg = ARR_NUM / THREAD_NUM;
```

	均衡			不均衡		
	连续块划分:	动态任务划分:	粗粒度动态任务划分:	连续块划分:	动态任务划分:	粗粒度动态任务划分:
Thread0	5942	5741	5794	1755	3832	3877
Thread1	5928	5742	5754	2856	3833	3674
Thread2	5929	5743	5766	4054	3834	3766
Thread3	5947	5742	5774	6213	3835	3746
average	5936.5	5742	5772	3719.5	3833.5	3765.75
sum	23746	22968	23088	14878	15334	15063



表中是均衡数据与不均衡数据跑三个算法得到的数据结构分析，时间单位为ms。

对于均衡划分的数据，三个算法的用时中：

1. 连续块划分用时较其他两者稍长，每个线程用时几乎一致。
2. 动态划分用时较其他两者稍小，每个线程用时几乎一致。
3. 粗粒度的动态划分介于二者之间，每个线程用时几乎一致。

对于不均衡划分的数据，三个算法中：

1. 连续块划分很明显可以看出每个线程不均匀：

   (a) Thread0处理的是前2500条数组，行内完全升序排列；
   (b) Thread1处理的是2500-4999条数组，此时随机有 2500 * 1/4 行升序， 2500 * 3/4 行降序;
   (c) Thread2处理的是 5000~7499条数组，此时随机有 2500 * 1/2 行升序，2500*1/2 行降序；
   (d) Thread3处理的是 7500~9999 数组，此 2500 行完全降序。

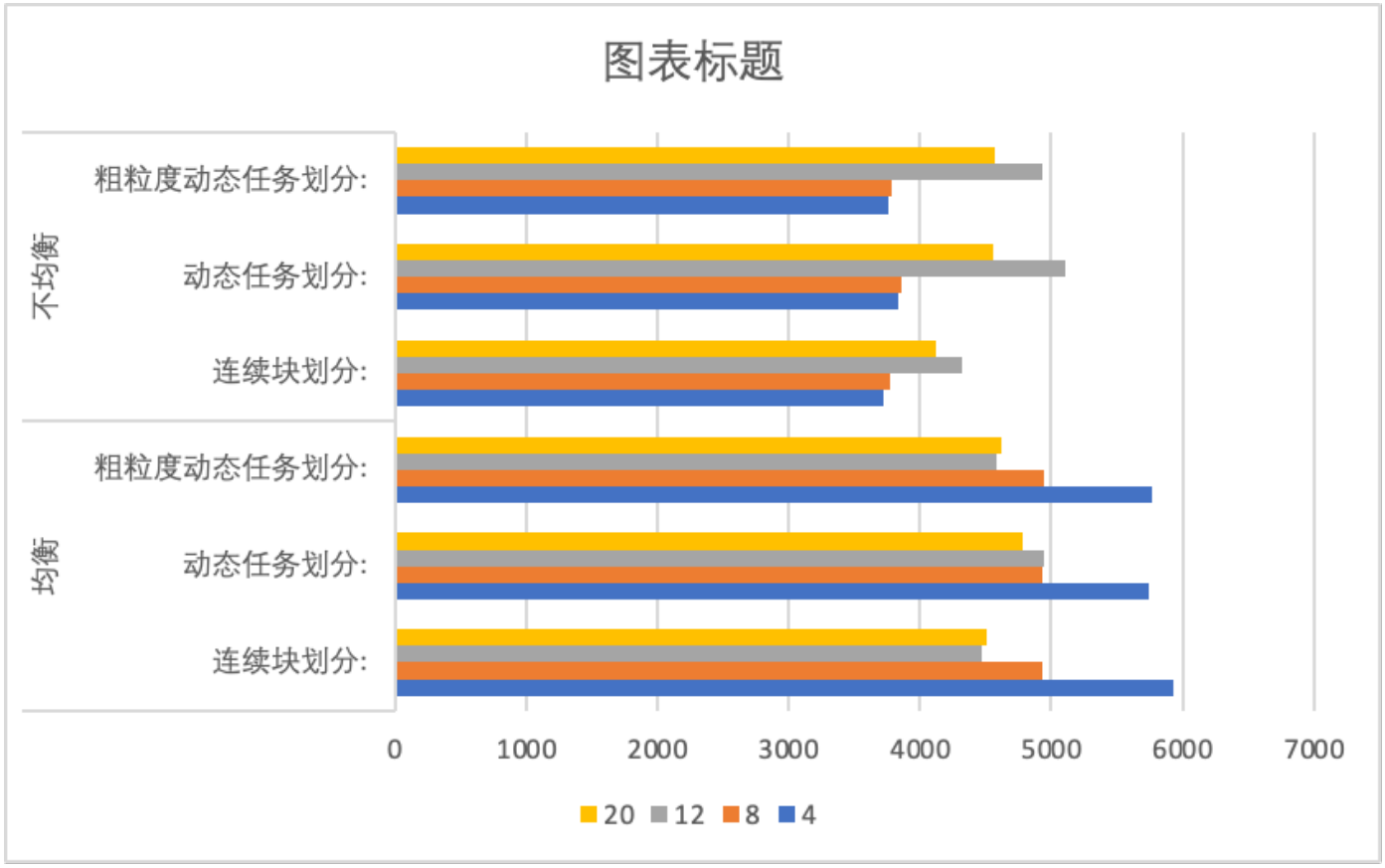
   故而从Thread0到Thread3分别是用时增长的递增不均匀用时。

2. 动态划分各线程用时均匀。
3. 粗粒度的动态划分较为均匀。

3.4.2 线程数

分别修改线程数为4，8，12，20，在均衡与不均衡的数据集上用3中不同的算法跑出结果，对线程数跑出的线程取平均值，绘制图表：

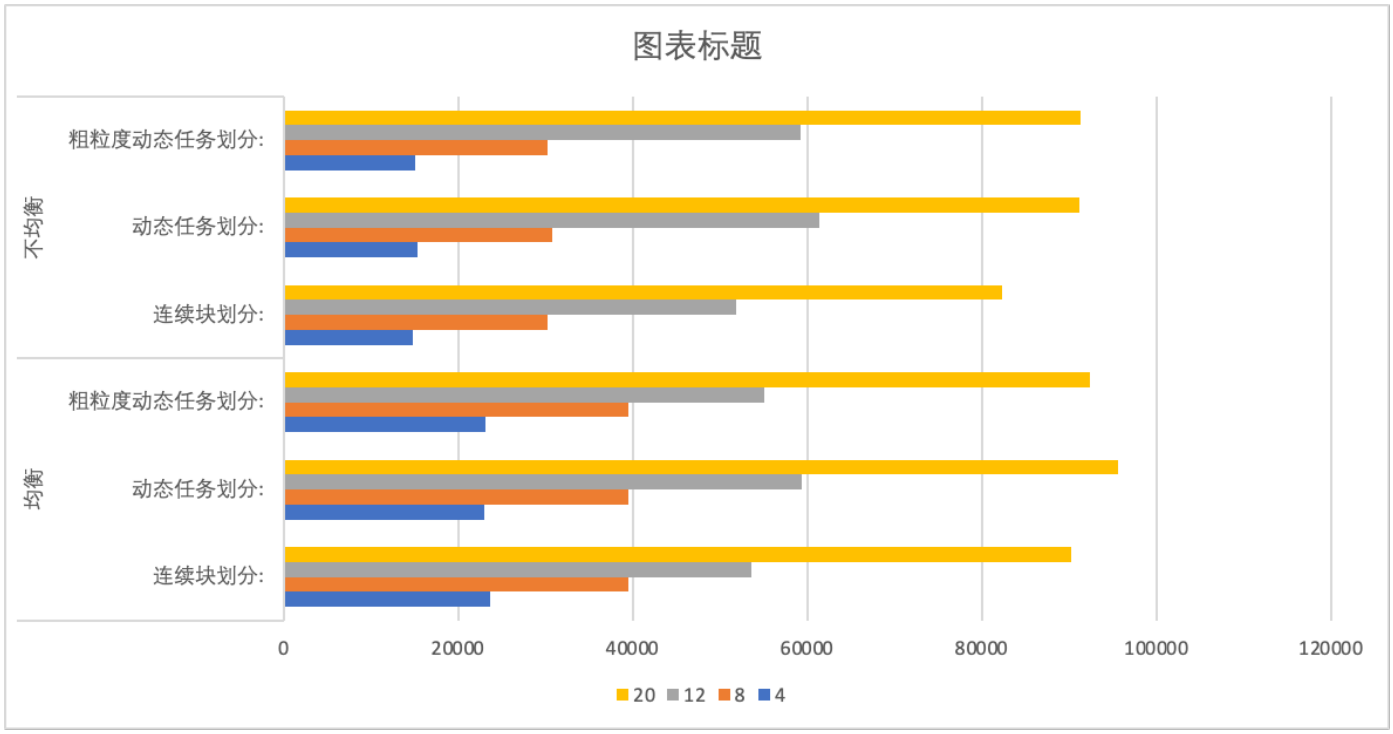
	均衡			不均衡		
线程数	连续块划分:	动态任务划分:	粗粒度动态任务划分:	连续块划分:	动态任务划分:	粗粒度动态任务划分:
4	5936.5	5742	5772	3719.5	3833.5	3765.75
8	4931.625	4934	4940.625	3777.125	3856.625	3787.875
12	4470.41667	4950.75	4587.08333	4323.5	5112.66667	4934.08333
20	4509.25	4783.45	4618.8	4116.55	4557.25	4566.25



1. 对于不均衡的数据而言，适当线程数会提高效率，但是，线程数划分的越多，意味着创建线程，销毁线程的消耗越大，可以看图中的平均线程用时，随着线程数的增大，平均的耗时也增大
2. 对于均衡的数据而言，可以看出，随着线程数变多，每个线程的用时在减少，但当线程数增加到一定的数量时，线程处理的开销将抵消多线程带来的效率提高，从而减低效率。

如下图，为跑出的所有线程时间加和的数据分析表：

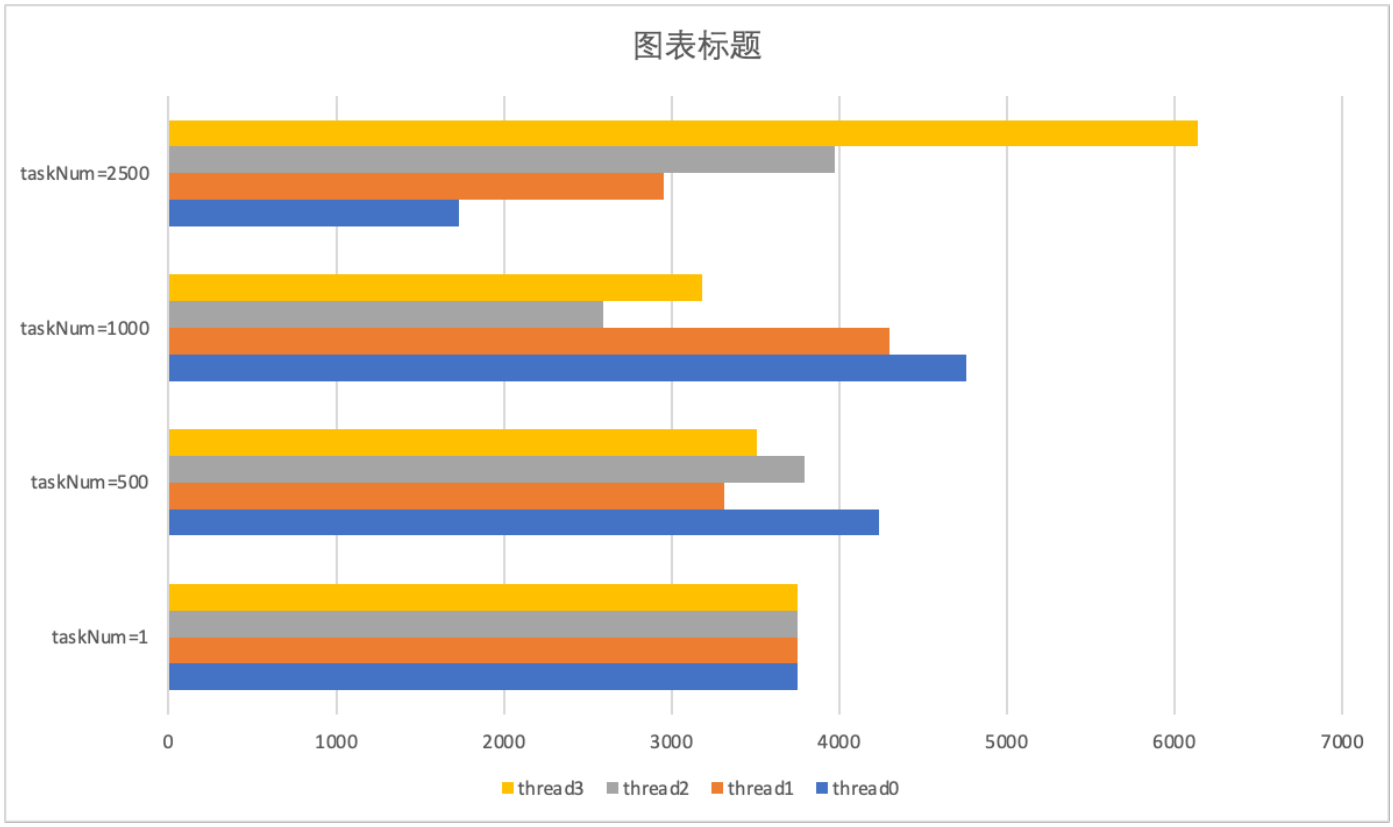
	均衡			不均衡		
线程数	连续块划分:	动态任务划分:	粗粒度动态任务划分:	连续块划分:	动态任务划分:	粗粒度动态任务划分:
4	23746	22968	23088	14878	15334	15063
8	39453	39472	39525	30217	30853	30303
12	53645	59409	55045	51882	61352	59209
20	90185	95669	92376	82331	91145	91325



可以看出，线程越多，总时间也越多，分析是因为创建于销毁线程的成本太高，所以，总时间必然增大。

3.4.3 动态划分的粒度大小

	taskNum=1	taskNum=500	taskNum=1000	taskNum=2500
thread0	3754	4242	4762	1733
thread1	3755	3315	4299	2954
thread2	3755	3793	2595	3976
thread3	3756	3512	3180	6139



对于粒度大小的划分，tasknum=1那么久等同于算法2动态划分，tasknum=2500就等同于算法1块划分；适当的粒度大小的选择，可以兼顾数据的均衡与任务的效率。

3.5 实验总结

以上三种对比与分析，分别从三个方面着重分析三种算法的表现，

1. 数据均衡与否：均衡的数据三种算法差不多，但是对于不均衡的数据而言，动态划分较好，而适当的降低粒度的精度也有利于效率的提高。
2. 线程的数量：线程数越大，线程的维护成本越高，线程数太小，则效率低下，适当的线程数，可以提高效率，但是线程越多，维护成本越高，不可滥用线程。
3. 粒度大小：对于粒度的选择而言，tasknum=1则为动态划分，tasknum=2500则为分块，适当的粒度选择可以提高算法效率。

4 高斯消去法解线性方程组

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。

4.1 问题描述

对于给定的方程组 Ax=b，使用 pthread 多线程编程，同时结合 sse 并行编程，完成对于线性方程组的高斯消元和回代求解。并探索多线程分配任务的最优分配方案。

4.2 实验初始化

1. 误差控制：生成 n 组矩阵 A 以及其对应的解 X 和 B。再创建线程，开始执行高斯消元和回代求解的过程。记录处理完 n组数据的总时间，再除以n，得到平均的处理时间，从而消除计时精度不够造成的误差。

2. 在执行程序的过程中，控制变量，分别改变动态划分过程中一次性分配给线程的任务数量、线程数量、矩阵规模，探索优化任务分配方法。
3. 时间：使用linux的gettimeofday()来计时，达到us级别的精度;

4.2.1 全局控制量

```
struct timeval tpstart,tpend;//计时

class Matrix;

// 规定矩阵最大规模
const int maxN = 1000;
// 测试矩阵的数量
const int maxCount = 50;
// 线程数
const int THREAD_NUM = 10;
// 矩阵实际规模
int n = 200;
// 动态划分线程任务
int nextTask = 1;
// 分配任务时上锁
pthread_mutex_t mutex_task;
pthread_mutex_t amutex;
// 一个线程一次分配处理的列数
const int columnGauss = 50;
const int columnBack = 4;
// 线程同步障碍
pthread_barrier_t barrier;
int flag;

typedef struct{
    int threadId;
} threadParm_t;

class Matrix{
public:
    float a[maxN][maxN];
    float atemp[maxN][maxN+1];
    float x[maxN];
    float b[maxN];
};
```

```
Matrix* dataset = new Matrix[maxCount];
```

4.2.2 矩阵初始化

```
// 初始化待测试矩阵
void init() {
    srand(static_cast <unsigned> (time(0)));
    for (int count = 0; count < maxCount; count++) {
        for (int j = 0; j < n; j++) {
            dataset[count].x[j] = static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / 1000));
            for (int k = 0; k < n; k++) {
                dataset[count].a[j][k] = static_cast <float> (rand()) / (static_cast
<float>(RAND_MAX / (1000 - 1)));
                dataset[count].atemp[j][k] = dataset[count].a[j][k];
            }
        }
        for (int j = 0; j < n; j++) {
            dataset[count].b[j] = 0;
            for (int k = 0; k < n; k++) {
                dataset[count].b[j] += dataset[count].a[j][k] * dataset[count].x[k];
            }
            dataset[count].atemp[j][n]=dataset[count].b[j];
        }
    }
}
```

4.2.3 转置矩阵

```
// 转置矩阵
void transpose(){
    for(int count=0; count<maxCount; count++){
        for(int i=0; i<n; i++){
            for(int j=i+1; j<n; j++){
                float temp=dataset[count].a[i][j];
                dataset[count].a[i][j]=dataset[count].a[j][i];
                dataset[count].a[j][i]=temp;
            }
        }
    }
}
```

4.2.4 计时器

使用linux系统，选择的计时器为gettimeofday()，头文件为#include <sys/time.h>，它的精度可以达到微妙，是C标准库的函数。

```
struct timeval tpstart,tpend;
gettimeofday(&tpstart,NULL);
gettimeofday(&tpend,NULL);
double timeuse = 1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
printf("used time:%fus\n",timeuse);
```

4.3 算法设计

4.3.1 高斯消元Pthread

```
void* pthread_elimination(void* parm){
    threadParm_t* p=(threadParm_t*) parm;
    int startJ;
    int endJ;
    for(int k=0; k<n; k++){
        pthread_mutex_lock(&amutex);
        if(k>=n){
            pthread_mutex_unlock(&amutex);
            pthread_exit(NULL);
        }
        pthread_mutex_unlock(&amutex);
        pthread_barrier_wait(&barrier);
        if(nextTask!=1)nextTask=1;
        pthread_barrier_wait(&barrier);
```

```
        while(1){
            pthread_mutex_lock(&mutex_task);
            startJ=k+nextTask;
            nextTask=nextTask+columnGauss;
            endJ=k+nextTask;
            pthread_mutex_unlock(&mutex_task);
            if(startJ>=n)break;
            for(int j=startJ; j<endJ; j++){
                for(int count=0; count<maxCount; count++){
                    dataset[count].atemp[k][j]=dataset[count].atemp[k]
[j]/dataset[count].atemp[k][k];
                    for(int i=k+1; i<n; i++){
                        dataset[count].atemp[i][j]=dataset[count].atemp[i][j]-
dataset[count].atemp[i][k]*dataset[count].atemp[k][j];
                    }
                }
            }
        }
    }
}
```

4.3.2 高斯消元结合SSE

结合sse的办法，一次并行处理4个数据。

1. Pthread 多线程编程：增广矩阵中每一行的工作都需要完成前面所有行的工作，因此不能按行分配工作给各个线程执行消元。但是，在一行中，可以并行处理每一列。故以第 k 行为 基准消元时，可以把 k+1 列到最后一列的任务，按照列数分配给多线程执行。
2. 各线程之间的同步通过barrier实现：通过使用 pthread_barrier 的接口设置障碍，保证所有线程均完成后进入下一轮循环。
3. 任务分配：通过互斥量 mutex 加锁，把每列待消元的任务动态分配给各线程；
4. 变量 columnGauss 控制动态分配时的粒度粗细。
5. 向量化计算，利用 sse 指令提高程序效率。

```
for(int j=startJ; j<endJ; j++){
                for(int count=0; count<maxCount; count++){
                    dataset[count].atemp[k][j]=dataset[count].atemp[k]
[j]/dataset[count].atemp[k][k];
                    // 先处理可以sse并行处理的部分
                    __m128 t1, t2, t3, t4;
                    int i;
                    for(i=k+1; i<n-3; i+=4){
                        float tmp1[4] = {dataset[count].atemp[k][j],
dataset[count].atemp[k][j], dataset[count].atemp[k][j], dataset[count].atemp[k][j]};
```

```
                        t1 = _mm_loadu_ps(tmp1);
                        float tmp2[4] = {dataset[count].a[i][k], dataset[count].a[i+1][k],
dataset[count].a[i+2][k], dataset[count].a[i+3][k]};
                        t2 = _mm_loadu_ps(tmp2);
                        float tmp3[4] = {dataset[count].a[i][j], dataset[count].a[i+1][j],
dataset[count].a[i+2][j], dataset[count].a[i+3][j]};
                        t3 = _mm_loadu_ps(tmp3);
                        t4 = _mm_sub_ps(t3, _mm_mul_ps(t1, t2));
                        _mm_storeu_ps(tmp1, t4);
                        dataset[count].a[i][j]=tmp1[0];
                        dataset[count].a[i+1][j]=tmp1[1];
                        dataset[count].a[i+2][j]=tmp1[2];
                        dataset[count].a[i+3][j]=tmp1[3];
                    }
                    // 再处理不能被4整除的部分
                    for(; i<n; i++){
                        dataset[count].a[i][j]=dataset[count].a[i][j]-dataset[count].a[i]
[k]*dataset[count].atemp[k][j];
                    }
                    if(endJ>=n+1){
                        for(i=k+1; i<n; i++){
                            dataset[count].b[i]=dataset[count].b[i]-dataset[count].a[i][k]*
(dataset[count].b[i]/dataset[count].a[k][k]);
                        }
                    }
                }
            }
```

4.3.3 回代求解 pthread 结合 sse 实现

回代求解的方法和具体细节，与上述高斯消元的过程类似。在并行处理之前对得到的增广矩阵作转置处理，从 cache 缓存的角度对回代求解的过程进一步优化。

```
void* pthread_sse_back(void* parm){
    threadParm_t* p=(threadParm_t*) parm;
    int startJ;
    int endJ;
    for(int count=0; count<maxCount; count++){
        for(int k=n-1; k>=0; k--){
            // 等待所有线程完成列计算，再进入下一个k
            pthread_barrier_wait(&barrier);
            if(flag!=1)flag=1;
            if(nextTask!=0)nextTask=0;
```

```
            pthread_mutex_lock(&amutex);
            if(k<0)pthread_exit(NULL);
            if(flag==1){
                dataset[count].x[k]=dataset[count].b[k]/dataset[count].a[k][k];
                flag=0;
            }
            pthread_mutex_unlock(&amutex);
            pthread_barrier_wait(&barrier);
            while(1){
                // 动态获取任务
                pthread_mutex_lock(&mutex_task);
                startJ=nextTask;
                endJ=nextTask+columnBack;
                nextTask=endJ;
                if(endJ>=k)endJ=k;
                pthread_mutex_unlock(&mutex_task);

                // 完成任务则退出循环进入下一个k
                if(startJ>=k)break;
                // 回代过程
                int j=startJ;
                // 先处理可以sse并行处理的部分
                __m128 t1, t2, t3, t4;
                for(; j<endJ-3; j+=4){
                    float tmp[4]=
{dataset[count].x[k],dataset[count].x[k],dataset[count].x[k],dataset[count].x[k]};
                    t1=_mm_loadu_ps(tmp);
                    t2=_mm_loadu_ps(dataset[count].a[k]+j);
                    t3=_mm_loadu_ps(dataset[count].b+j);
                    _mm_storeu_ps(tmp, _mm_sub_ps(t3, _mm_mul_ps(t1,t2)));
                    dataset[count].b[j]=tmp[0];
                    dataset[count].b[j+1]=tmp[1];
                    dataset[count].b[j+2]=tmp[2];
                    dataset[count].b[j+3]=tmp[3];
                }
                // 再处理不能被4整除的部分
                for(; j<endJ; j++){
                    dataset[count].b[j]=dataset[count].b[j]-
dataset[count].x[k]*dataset[count].a[k][j];
                }
            }
        }
    }
}
```

4.4 实验设计与结果

探索优化任务分配方法。

4.4.1 线程数

实验参数：

```
// 测试矩阵的数量
const int maxCount = 50;
// 线程数
const int THREAD_NUM = n;
// 矩阵实际规模
int n = 200;
// 一个线程一次分配处理的列数
const int columnGauss = 5;
```

线程	5	10	20	50	100
pthread 消元	1146ms	680ms	483ms	527ms	521ms
pthread+sse 消元	1197ms	708ms	493ms	503ms	520ms
pthread+sse 回代过程	15ms	19ms	19ms	18ms	20ms

如图，线程数的降低会一定的提高效率，在20的时候，消元达到一个较低的时间值，而对于回代而言，5的时候较低。

4.4.2 分配列数

实验其他参数：

```
// 测试矩阵的数量
const int maxCount = 50;
// 线程数
const int THREAD_NUM = 20;
// 矩阵实际规模
int n = 200;

// 一个线程一次分配处理的列数
const int columnGauss = n;
const int columnBack = n;
```

列数	1	5	10	15
pthread+sse 消元	373	485	754	1062
pthread+sse 回代过程	20	20	20	18

如表，分配的列数越大，对于消元不利，对于回代而言会降低。

4.4.3 矩阵规模

规模	50	100	200	500
pthread+sse 消元	87	312	1079	6336
pthread+sse 回代过程	2	8	17	150

如表，随着规模的增大，用时增多

4.5 分析总结

1. 随着线程数量的增大，程序的执行时间呈现下降的趋势而后回升——尽管并行的线程数增大了，但任务的总量不变，增加的线程可能并不会被分配到任务，反而可能会增加一些维护线程同步的额外开销。
2. 随着矩阵规模的增大，程序的执行时间上升较快。
3. 在实验动态分配粗细粒度中，我们可以看到对于回代而言，粗细度几乎不怎么变化，但是对于消元而言，随着粒度的变大，时间变大。

5 附加题：实现路障Barrier的功能

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。

5.1 使用忙等待实现

```
#define NUM_THREADS 5
typedef struct{
    int threadId;
}threadParm_t;

int flag = 0;

// 利用忙等待实现
```

```c
void *threadFuncBusywaiting(void *parm) {
    threadParm_t *p = (threadParm_t *) parm;
    fprintf(stdout, "Thread %d has entered step 1.\n", p->threadId);
    while(flag!=p->threadId)Sleep(0);
    flag++;
    while(flag!=NUM_THREADS)Sleep(0);
    fprintf(stdout, "Thread %d has entered step 2.\n", p->threadId);
    pthread_exit(NULL);
}

// 利用忙等待实现
int main(int argc, char *argv[]){
    pthread_t thread[NUM_THREADS];
    threadParm_t threadParm[NUM_THREADS];
    int i;
    for (i=0; i<NUM_THREADS; i++){
        threadParm[i].threadId = i;
        pthread_create(&thread[i], NULL, threadFuncBusywaiting, (void *)&threadParm[i]);
    }
    for (i=0; i<NUM_THREADS; i++){
        pthread_join(thread[i], NULL);
    }
    return 0;
}
```

```
zhangyizhen@linux-PowerEdge-R730:~/Program/CProgram/parallels_work/barrier-work$ ls
busywaiting.cpp  mutax.cpp  semaphore.cpp
zhangyizhen@linux-PowerEdge-R730:~/Program/CProgram/parallels_work/barrier-work$ g++ -o busywaiting busywaiting.cpp -lpthread
zhangyizhen@linux-PowerEdge-R730:~/Program/CProgram/parallels_work/barrier-work$ ./busywaiting
Thread 1 has entered step 1.
Thread 0 has entered step 1.
Thread 4 has entered step 1.
Thread 3 has entered step 1.
Thread 2 has entered step 1.
Thread 4 has entered step 2.
Thread 1 has entered step 2.
Thread 0 has entered step 2.
Thread 3 has entered step 2.
Thread 2 has entered step 2.
zhangyizhen@linux-PowerEdge-R730:~/Program/CProgram/parallels_work/barrier-work$
```

5.2 使用互斥量mutax实现

```c
#define NUM_THREADS 5
typedef struct{
    int threadId;
}threadParm_t;

int flag = 0;
pthread_mutex_t amutex;
pthread_cond_t cond;
```

```c
// 利用互斥量(条件变量)实现
void *threadFuncMutex(void *parm) {
    threadParm_t *p = (threadParm_t *) parm;
    fprintf(stdout, "Thread %d has entered step 1.\n", p->threadId);
    pthread_mutex_lock(&amutex);
    flag++;
    if(flag==NUM_THREADS){
        pthread_cond_broadcast(&cond);
    }
    else{
        while(pthread_cond_wait(&cond, &amutex)!=0);
    }
    pthread_mutex_unlock(&amutex);
    fprintf(stdout, "Thread %d has entered step 2.\n", p->threadId);
    pthread_exit(NULL);
}

// 利用互斥量(条件变量)实现
int main(int argc, char *argv[]){
    pthread_t thread[NUM_THREADS];
    threadParm_t threadParm[NUM_THREADS];
    pthread_mutex_init(&amutex, NULL);
    pthread_cond_init(&cond, NULL);
    int i;
    for (i=0; i<NUM_THREADS; i++){
        threadParm[i].threadId = i;
        pthread_create(&thread[i], NULL, threadFuncMutex, (void *)&threadParm[i]);
    }
    for (i=0; i<NUM_THREADS; i++){
        pthread_join(thread[i], NULL);
    }
    pthread_mutex_destroy(&amutex);
    pthread_cond_destroy(&cond);
    return 0;
}
```

```
zhangyizhen@linux-PowerEdge-R730:~/Program/CProgram/parallels_work/barrier-work$ g++ -o mutax mutax.cpp -lpthread
zhangyizhen@linux-PowerEdge-R730:~/Program/CProgram/parallels_work/barrier-work$ ./mutax
Thread 0 has entered step 1.
Thread 3 has entered step 1.
Thread 2 has entered step 1.
Thread 1 has entered step 1.
Thread 4 has entered step 1.
Thread 4 has entered step 2.
Thread 0 has entered step 2.
Thread 3 has entered step 2.
Thread 2 has entered step 2.
Thread 1 has entered step 2.
zhangyizhen@linux-PowerEdge-R730:~/Program/CProgram/parallels_work/barrier-work$
```

5.3 使用信号量实现

```
#define NUM_THREADS 5

typedef struct{
    int threadId;
}threadParm_t;

int sum=0;
sem_t counter;
sem_t barrier;

// 利用信号量实现
void *threadFuncSemaphore(void *parm) {
    threadParm_t *p = (threadParm_t *) parm;
    fprintf(stdout, "Thread %d has entered step 1.\n", p->threadId);
    sem_wait(&counter);
    if(sum==NUM_THREADS-1){
        for(int i=0;i<NUM_THREADS;i++){
            sem_post(&barrier);
        }
        sem_post(&counter);
    }
    else{
        sum++;
        sem_post(&counter);
    }
    sem_wait(&barrier);
    fprintf(stdout, "Thread %d has entered step 2.\n", p->threadId);
    pthread_exit(NULL);
}

// 利用信号量实现
int main(int argc, char *argv[]){
    pthread_t thread[NUM_THREADS];
    threadParm_t threadParm[NUM_THREADS];
    sem_init(&counter, 0, 1);
    sem_init(&barrier, 0, 0);
    int i;
    for (i=0; i<NUM_THREADS; i++){
        threadParm[i].threadId = i;
        pthread_create(&thread[i], NULL, threadFuncSemaphore, (void *)&threadParm[i]);
    }
    for (i=0; i<NUM_THREADS; i++){
        pthread_join(thread[i], NULL);
```

```
    }
    sem_destroy(&counter);
    sem_destroy(&barrier);
    return 0;
}
```