高性能计算程序设计基础 (3) 秋季2021

0. 构造MPI版本矩阵乘法加速比和并行效率表

> 参考下图，分别构造MPI版本的标准矩阵乘法和优化后矩阵乘法（例如：集合通信、create_struct）的加速比和并行效率表格。并分类讨论两种矩阵乘法分别在强扩展和弱扩展情况下的扩展性。

$S = \frac{T_{并行}}{T_{串行}}$

获得串行方法在矩阵规模为128、256、512、1024的运行时间：

MPI版本的标准矩阵乘法（点对点通信）的在线程为1、2、4、8、16的情况下分别计算矩阵规模为128、256、512、1024的运行时间：

（只截取了部分）

```
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_1 lab2_1.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_1 128 128 128
completed:
the number of process is 1, the time cost = 0.026098 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_1 256 256 256
completed:
the number of process is 1, the time cost = 0.263868 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_1 512 512 512
completed:
the number of process is 1, the time cost = 2.598716 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_1 1024 1024 1024
completed:
the number of process is 1, the time cost = 32.454763 s
```

MPI版本优化后（集合通信）的矩阵乘法在线程为1、2、4、8、16的情况下分别计算矩阵规模为128、256、512、1024的运行时间：

（只截取了部分）

```
wwwj@ubuntu:~/lab/lab2$ mpicc -o lab2_2_1 lab2_2_1.c
wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_2_1 128 128 128
completed:
the number of process is 1, the time cost = 0.035335 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_2_1 256 256 256
completed:
the number of process is 1, the time cost = 0.335989 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_2_1 512 512 512
completed:
the number of process is 1, the time cost = 2.424951 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 1 ./lab2_2_1 1024 1024 1024
completed:
the number of process is 1, the time cost = 34.347908 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 2 ./lab2_2_1 128 128 128
completed:
the number of process is 2, the time cost = 0.018792 s

wwwj@ubuntu:~/lab/lab2$ mpiexec -np 2 ./lab2_2_1 256 256 256
completed:
the number of process is 2, the time cost = 0.181072 s
```

1. MPI点对点通信和串行运算的加速比表格：

Comm_size (num of processes)	Order of Matrix (Speedups)			
	128	256	512	1024
1	0.784218276	1.517311176	1.660351657	2.61928406
2	0.746957541	1.254104252	0.996279614	1.71983302
4	0.685507377	0.685379949	0.639172992	1.021743082
8	1.672676463	0.885000431	0.480741266	0.617244385
16	1.92283422	1.031517208	0.598432748	0.501400567

2. MPI集合通信和串行运算的加速比表格：

Comm_size (num of processes)	Order of Matrix (Speedups)			
	128	256	512	1024
1	1.061780703	1.932026106	1.549331059	2.772071634
2	0.564680429	1.041212156	0.976396023	1.582181717
4	1.198894198	0.993605704	0.63741215	1.047294019
8	2.756543165	0.99070757	0.408045184	0.55031798
16	0.816280537	0.631959978	0.342034041	0.35228042

效率的计算：$E = \frac{S}{p}$

3. MPI点对点通信和串行运算的效率表格：

Comm_size (num of processes)	Order of Matrix (Speedups)			
	128	256	512	1024
1	0.784218276	1.517311176	1.660351657	2.61928406
2	0.37347877	0.627052126	0.498139807	0.85991651
4	0.171376844	0.171344987	0.159793248	0.25543577
8	0.209084558	0.110625054	0.060092658	0.077155548
16	0.120177139	0.064469825	0.037402047	0.031337535

4. MPI集合通信和串行运算的效率表格：

Comm_size (num of processes)	Order of Matrix (Speedups)			
	128	256	512	1024
1	1.061780703	1.932026106	1.549331059	2.772071634
2	0.282340215	0.520606078	0.488198012	0.791090858
4	0.299723549	0.248401426	0.159353037	0.261823505
8	0.344567896	0.123838446	0.051005648	0.068789748
16	0.051017534	0.039497499	0.021377128	0.022017526

分析扩展性

- 强可扩展性：如果程序在增加线程/进程的个数时，可以维持稳定的效率，却不增加问题的规模，那么这个程序就是强可扩展的
- 弱可扩展性：如果在增加线程/进程的同时，只能以相同倍率增加问题规模才能使效率值保持不变，那么这个程序就是弱可扩展的。

根据强可扩展性和若可扩展性的定义，从这两个效率表格上看，他们并不是强可扩展或者若可扩展的。在只增加进程的个数的时候，除了个别情况，他们的效率都是降低的，因此并不具有强可扩展性；在增加进程个数的同时增加问题规模，可能是算法的缺陷，它的效率也是降低的。

1. 通过 Pthreads实现通用矩阵乘法

> 通过Pthreads实现通用矩阵乘法（Lab1）的并行版本，Pthreads并行线程从1增加至8，矩阵规模从512增加至2048.
>
> 通用矩阵乘法（GEMM）通常定义为：
>
> $C = AB$
>
> $C_{m,n} = \sum_{n=1}^{N} A_{m,n} B_{n,k}$
>
> 输入：M , N, K三个整数（512 ~2048）
>
> 问题描述：随机生成M*N和N*K的两个矩阵A,B,对这两个矩阵做乘法得到矩阵C.
>
> 输出：A,B,C三个矩阵以及矩阵计算的时间

代码

> 完整代码见 lab3_1.c

1. 头文件以及一些全局变量

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<pthread.h>
4  #include<time.h>
5
6  int m,n,k,thread_num,partA;
7  struct bag{
8          double *A_;
9          double *B_;
10         double *C_;
11 };
```

- 第1-4行，是我们需要用到的一些头文件，其中`#include<pthread.h>`是我们利用`pthread`实现并行计算需要用到的头文件。
- 第6行，五个全局变量：m，n，k是矩阵A(m*n)、B(n*k)、C(m*k)的大小，thread_num是我们接下来要创建的线程个数,partA是我们将矩阵A划分成thread_num部分，每部分partA行，用来分给每个线程进行计算。
- 第7-8行，创建一个结构体，里面包含三个double类型的指针，这个结构体以及里面的数据是传递给不同线程的参数，单个线程用这个结构体里的数据进行计算。

2. 通过一个函数 `Get_args()` 来获得我们在终端输入运行指令时同时输入的参数m、n、k、thread_num，并通过计算获得需要将矩阵A划分成 `thread_num` 部分，每部分 `partA` 行。

```
void Get_args(int argc,char *argv[]){
    thread_num = strtol(argv[1], NULL, 10);
    m = strtol(argv[2], NULL, 10);
    n = strtol(argv[3], NULL, 10);
    k = strtol(argv[4], NULL, 10);
    partA = m/thread_num;
}
```

3. 一个用来输出矩阵的函数 `PRINT()`：

```
void PRINT(double *x,int m,int n){
    for(int i=0;i<m;++i){
        for(int j=0;j<n;++j) printf("%lf  ",x[i*n+j]);
        printf("\n");
    }
    printf("\n");
}
```

4. 创建 `pthread` 后调用的函数 `computer()`，计算分配给每个线程的$A_ * B_$，并把结果存储在$C_$中。

```
void *computer(void *arg){
    struct bag *part;
    part=(struct bag*)arg;
    for(int i=0; i<partA; ++i){
        for(int j=0; j<k ;++j){
            int temp=0;
            for(int a=0;a<n;++a) temp+=part->A_[i*n+a]*part->B_[a*k+j];
            part->C_[i*n+j]=temp;
        }
    }
}
```

5. 主函数中：
    - 首先调用函数获得矩阵尺寸，接着定义一些所需的变量：
        - 第5行，定义三个double类型的指针，并在第9-11行为他们分配空间；
        - 第6行，定义两个变量来记录时间；
        - 第7行，定义 `thread_num` 个数据类型为 `struct bag` 的变量，这里并不给结构体里的指针赋值是因为之后我们会将结构体里的指针直接指向与A、B、C指针指向的一些相关位置上：

```
int main(int argc,char *argv[]){
    Get_args(argc,argv);

    //定义和分配空间
    double *A,*B,*C;
    int starttime,finishtime;
    struct bag part[thread_num];

    A = (double*)malloc(sizeof(double)*m*n);
```

```
    B = (double*)malloc(sizeof(double)*n*k);
    C = (double*)malloc(sizeof(double)*m*k);

    ...
}
```

- 给矩阵A、B赋值50以内的随机数作为矩阵元素：

```
int main(int argc,char *argv[]){
    ...
    //给矩阵赋值
    srand(time(NULL));
    for(int i=0; i<m; ++i)
        for(int j=0; j<n; ++j) A[i*n+j]=rand()%50;
    for(int i=0; i<n; ++i)
        for(int j=0; j<k; ++j) B[i*k+j]=rand()%50;
    ...
}
```

- 给`thread_num`个结构体里的指针赋值：
  - 第4行，A_指向我们将A分成`thread_num`个部分的每一部分的首地址；
  - 第5行，B_指向指针B指向的位置；
  - 第6行，C_指向我们将部分A乘以B后获得的C_应该存放在C中的首地址；

```
int main(int argc,char *argv[]){
    ...
    for(int i=0; i<thread_num; ++i){
        part[i].A_ = A+i*partA*n;
        part[i].B_ = B;
        part[i].C_ = C+i*partA*k;
    }
    ...
}
```

- 利用pthread开始创建线程并计算：
  - 第3行，记录开始的时间；
  - 第4行，创建`thread_num`个线程标识符；
  - 第6-8行，创建`thread_num`个线程，每个线程都有对应的线程标识符，他们都调用`computer()`函数，并传入该线程计算需要用到的参数；
  - 第10-18行，由于我们的矩阵A可能并不能完整分成`thread_num`份，部分A可能并没有参与到计算中，因此这里需要计算剩下的A*B，并放入C对应的位置上；
  - 第20-22行，线程计算结束，释放线程标识符对应的线程占用的资源，本次计算没有返回值，参数为`NULL`；
  - 第24行，记录计算结束的时间。

```
int main(int argc,char *argv[]){
    ...
    starttime=clock();
    pthread_t pth[thread_num];

    for(int i=0 ;i<thread_num; ++i){
        pthread_create(&pth[i], NULL, computer, &part[i]);
    }
```

```
9
10      if(partA * thread_num < m){
11          for(int i=partA*thread_num ; i<m; ++i){
12              for(int j=0; j<k; ++j){
13                  int temp=0;
14                  for(int a=0; a<n; ++a) temp+=A[i*n+a]*B[a*k+j];
15                  C[i*k+j]=temp;
16              }
17          }
18      }
19
20      for(int i=0; i<thread_num; ++i){
21          pthread_join(pth[i],NULL);
22      }
23
24      finishtime=clock();
25      ...
26  }
```

- 结束运行：
  - 第3-4行，输出本次运算的线程数，以及花费的时间；
  - 第6-13行，输出矩阵A、B、C，如果是较大的矩阵运算可以把这部分注释掉；
  - 第15-17行，释放之前给A、B、C分配的空间。

```
1   int main(int argc,char *argv[]){
2       ...
3       printf("completed:\n");
4       printf("the number of thread is %d, the time cost = %lf s\n
    \n",thread_num,(double)(finishtime - starttime)/CLOCKS_PER_SEC);
5
6       //print
7           printf("\n");
8           printf("A:\n");
9           PRINT(A,m,n);
10          printf("B:\n");
11          PRINT(B,n,k);
12          printf("C=A*B:\n");
13          PRINT(C,m,k);
14
15      free(A);
16      free(B);
17      free(C);
18
19      return 0;
20
21  }
```

运行结果

- 下面是编译和运行这段代码的指令：

```
/* File:     lab3_1.c
 *
 * Purpose:  Use pthread to implement matrix multiplication
 *
 * Compile:  gcc -o lab3_1 lab3_1.c -lpthread
 *
 * Run:      ./lab3_1 <num> <m> <k> <n>
 *           num is the thread number that we will create
 *           m,k,n is size of matrix A,B,C
 *
 * Input:    none
 * Output:   the cost time. If you want ,you can also print matrix
A,B,C.
 */
```

- 我们首先验证代码计算矩阵乘法的正确性：
  计算两个5×5矩阵的乘法，左边是代码的计算结果，右边是用matlab计算的结果。

```
wwwj@ubuntu:~/lab/lab3$ gcc -o lab3_1 lab3_1.c -lpthread
wwwj@ubuntu:~/lab/lab3$ ./lab3_1 4 5 5 5

A:
33.000000  36.000000  27.000000  15.000000  43.000000
35.000000  36.000000  42.000000  49.000000  21.000000
12.000000  27.000000  40.000000  9.000000  13.000000
26.000000  40.000000  26.000000  22.000000  36.000000
11.000000  18.000000  17.000000  29.000000  32.000000

B:
30.000000  12.000000  23.000000  17.000000  35.000000
29.000000  2.000000  22.000000  8.000000  19.000000
17.000000  43.000000  6.000000  11.000000  42.000000
29.000000  23.000000  21.000000  19.000000  34.000000
37.000000  48.000000  24.000000  15.000000  20.000000

C=A*B:
4519.000000  4038.000000  3060.000000  2076.000000  4343.000000
5006.000000  4433.000000  3382.000000  2591.000000  5759.000000
2565.000000  2749.000000  1611.000000  1226.000000  3179.000000
4352.000000  3744.000000  2960.000000  2006.000000  4230.000000
3166.000000  3102.000000  2128.000000  1549.000000  3067.000000
```

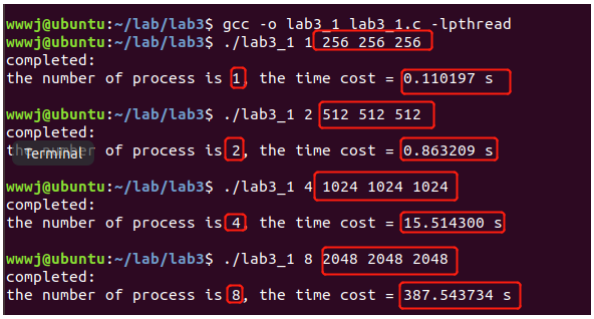
```
A=[33.000000, 36.000000, 27.000000, 15.000000, 43.000000;
35.000000, 36.000000, 42.000000, 49.000000, 21.000000;
12.000000, 27.000000, 40.000000, 9.000000 , 13.000000;
26.000000, 40.000000, 26.000000, 22.000000, 36.000000;
11.000000, 18.000000, 17.000000, 29.000000, 32.000000];

B=[30.000000, 12.000000, 23.000000, 17.000000, 35.000000;
29.000000, 2.000000 , 22.000000, 8.000000 , 19.000000;
17.000000, 43.000000, 6.000000 , 11.000000, 42.000000;
29.000000, 23.000000, 21.000000, 19.000000, 34.000000;
37.000000, 48.000000, 24.000000, 15.000000, 20.000000];

C=A*B;
```

令行窗口

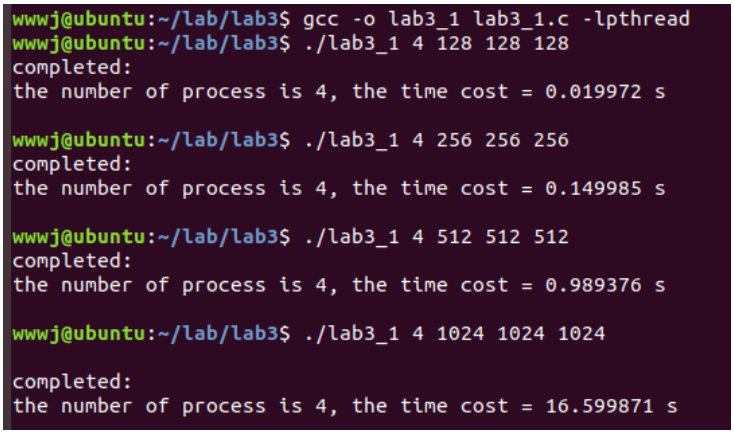
```
C =

        4519        4038        3060        2076        4343
        5006        4433        3382        2591        5759
        2565        2749        1611        1226        3179
        4352        3744        2960        2006        4230
        3166        3102        2128        1549        3067
```

计算结果正确。

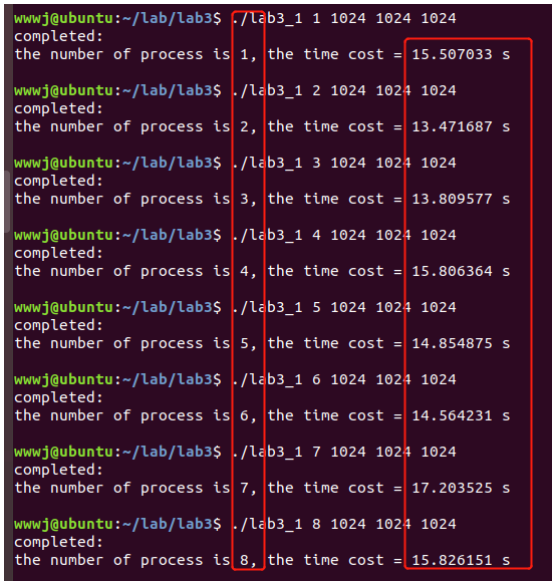
- Pthreads并行线程从1增加至8，矩阵规模从256增加至2048：



或者可以在线程4下，将矩阵规模从128增加到1024：



在矩阵规模为1024的情况下，将线程数从1增加到8：



运行时间在线程从1增加到2和3的时候减少最明显，其余情况下运行时间并没有减少。

遇到的问题

—

主要是在`pthread_create`中传递的函数的参数必须是`void *`类型的，如果直接将参数写成需要的`struct bag`然后引用：

```
void *computer(struct bag part){
    ...
}

pthread_create(&pth[i], NULL, computer, &part[i]);
```

那么就会在编译的时候出现警告：

```
wwwj@ubuntu:~/lab/lab3$ gcc -o lab3_1 lab3_1.c -lpthread
lab3_1.c: In function ‘main’:
lab3_1.c:55:39: warning: passing argument 3 of ‘pthread_create’ from incompatible pointer type [-Wincompatible-pointer-types]
         pthread_create(&pth[i], NULL, computer, (void*)&part[i]);
                                       ^~~~~~~~
In file included from lab3_1.c:3:0:
/usr/include/pthread.h:234:12: note: expected ‘void * (*)(void *)’ but argument is of type ‘void * (*)(struct bag *)’
 extern int pthread_create (pthread_t *__restrict __newthread,
            ^~~~~~~~~~~~~~
```

因此先将函数的参数类型写成`void *`然后再在函数中进行类型转换，然后引用就不会再有警告了：

```
void *computer(void *arg){
    struct bag *part;
    part=(struct bag*)arg;
    ...
}

pthread_create(&pth[i], NULL, computer, &part[i]);
```

二

刚开始写代码的时候，给结构体赋值，就是给他们分配空间，然后将A、B中的值“复制”过去，但是这样当矩阵非常大的时候也是非常耗时的，因此改用直接将指针复制过去，这样更加快捷：

```
int main(int argc,char *argv[]){
    ...
    for(int i=0; i<thread_num; ++i){
        part[i].A_ = A+i*partA*n;
        part[i].B_ = B;
        part[i].C_ = C+i*partA*k;
    }
    ...
}
```

还有需要注意的就是释放地址空间的时候，不要重复释放地址空间或者释放没有分配的地址空间（比如这里我并没有给结构体里的指针分配空间，如果释放空间）会出现“内存泄露”的问题：

```
free(): invalid pointer
Aborted (core dumped)
```

2. 基于Pthreads的数组求和

> 编写使用多个进程/线程对数组a[1000]求和的简单程序演示Pthreads的用法。创建n个线程，每个线程通过共享单元global_index获取a数组的下一个未加元素，注意不能在临界段外访问全局下标global_index
>
> 重写上面的例子，使得各进程可以一次最多提取10个连续的数，以组为单位进行求和，从而减少对下标的访问

(1) 对数组a[1000]求和

代码

> 完整代码见lab3_2_1.c

- 需要用到的一些头文件和全局变量

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<pthread.h>

int a[1000];
int global_index=0;//全局下标
int n;//thread_num
pthread_mutex_t mutex;//互斥锁，用来写临界区
```

- 创建`pthread`后调用的函数`computer()`：
  - 第2-3行，对传入的参数进行数据类型的转换，转换成int类型的指针；
  - 第4行，用来存储从数组a中获得的值，用来进行求和；
  - 第5-14行，一个`while`循环，直到全局下标变为1000就跳出循环，这说明数组内的所有值都已被计算；
    - 第6-9行，通过加锁和解锁进入临界区，获得a中下标为`global_index`的值，然后将`global_index`更新；
    - 第10行，将取得的a中的值加到指针`sum_`指向的值上。
    - 第12-13行，一个延迟，否则计算这1000个数字的和加起来很快，还没等其他几个线程创建就已经计算结束，这是为了体现所有线程都参与计算，并获得正确的计算结果；
  - 第15行，输出这个线程的计算结果。

```
void *computer(void *arg){
    int *sum_;
    sum_= (int*)arg;
    int x = 0;
    while(global_index != 1000){
        pthread_mutex_lock(&mutex);
            x = a[global_index];
            global_index++;
        pthread_mutex_unlock(&mutex);
        (*sum_)+=x;

         int delay = 0xfff;
         while (delay) --delay;
    }
    printf("the sum in a thread: %d\n",*sum_);
}
```

- 主函数：
  - 第1行，从运行的命令行获取线程数n；
  - 第4-6行，给数组a赋值，为了方便确认我们的计算结果，这里让`a[i]=i`，即计算从0加到999；
  - 第8-11行，定义一个有n个值的数组x，赋值为0；
  - 第13-14行，定义pthread线程标识符，初始化锁；

- 第16-18行，创建`n`个线程，每个线程都有对应的线程标识符，他们都调用`computer()`函数，并传入一个参数，用来存放每个线程的计算得到的和；
- 第20-24行，线程计算结束，释放线程标识符对应的线程占用的资源，本次计算没有返回值，参数为`NULL`，并把每个线程计算得到的和加起来得到数组a[1000]中所有值的和。

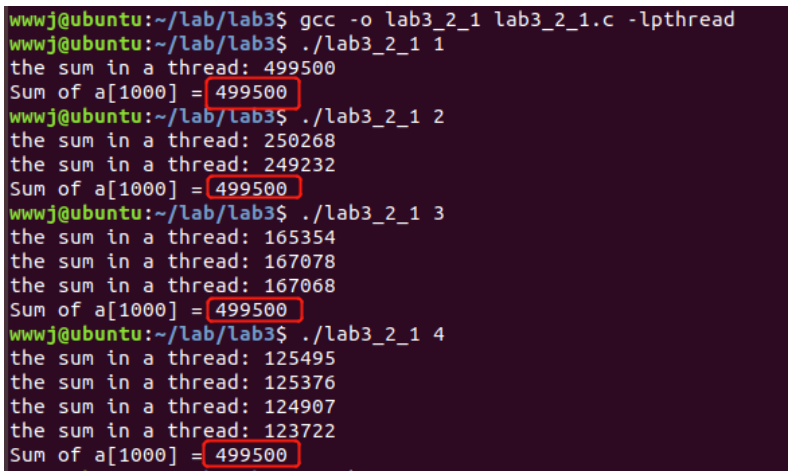
```
int main(int argc,char *argv[]){
    n = strtol(argv[1], NULL, 10);

    for(int i=0; i<1000; ++i){
        a[i]=i;
    }

    int x[n];
    for(int i=0; i<n ;++i){
        x[i] = 0;
    }

    pthread_t pth[n];
    pthread_mutex_init(&mutex, NULL);

    for(int i=0;i<n;++i){
        pthread_create(&pth[i], NULL, computer, &x[i]);
    }

    int sum = 0;
    for(int i=0; i<n; ++i){
        pthread_join(pth[i], NULL);
        sum+=x[i];
    }

    printf("Sum of a[1000] = %d\n",sum);
    return 0;
}
```

运行结果

- 下面是编译和运行这段代码的指令：

```
/* File:     lab3_2_1.c
 *
 * Purpose:  Use pthread to calculate the sum of all values in array a
 *
 * Compile:  gcc -o lab3_2_1 lab3_2_1.c -lpthread
 *
 * Run:      ./lab3_2_1 <num>
 *           num is the thread number that we will create
 *
 * Input:    none
 * Output:   A single thread and the sum of all threads
 */
```

- 将线程数从1加到4的运行结果，可以看到他们的计算结果正确：



(2) 重写例子

代码

> 完整代码见lab3_2_2.c

和上面的计算数字和的代码相比，只更改了函数`computer()`，这里只介绍更改了的部分，其余内容不再赘述:

- 第4行，定义一个整数值来存放临时的下标;
- 第5-15行，进入一个`while`循环，当`global_index >= 1000`时退出循环，或者像上面一样判断`global_index`是否为0，因为每次`global_index`更新都是加10，一定会到1000，如果这里更新的不是19，那么有可能`global_index`不会等于1000，而是直接大于1000;
    - 第6-9行，通过加锁和解锁进入临界区，获得全局下标`global_index`的值，然后将`global_index+10`更新;
    - 第10-11行，一个延迟，否则计算这1000个数字的和加起来很快，还没等其他几个线程创建就已经计算结束，这是为了体现所有线程都参与计算，并获得正确的计算结果;
    - 第12-13行，将获得的提取到的10个连续的数，以组为单位进行求和，注意要保证temp_index+i不大于等于1000;

```
1  void *computer(void *arg){
2      int *sum_;
3      sum_= (int*)arg;
4      int temp_index;
5      while(global_index < 1000){
6          pthread_mutex_lock(&mutex);
7              temp_index = global_index;
8              global_index+=10;
9          pthread_mutex_unlock(&mutex);
10              int delay = 0xffff;
11              while (delay) --delay;
12         for(int i=0; i<10 && temp_index+i<1000; ++i){
13             (*sum_)+=a[temp_index+i];
14         }
15     }
16     printf("the sum in a thread: %d\n",*sum_);
17 }
```

运行结果

- 下面是编译和运行这段代码的指令：

```
/* File:     lab3_2_2.c
 *
 * Purpose:  Use pthread to calculate the sum of all values in array a
 *
 * Compile:  gcc -o lab3_2_2 lab3_2_2.c -lpthread
 *
 * Run:      ./lab3_2_2 <num>
 *           num is the thread number that we will create
 *
 * Input:    none
 * Output:   A single thread and the sum of all threads
 */
```

- 将线程数从1加到4的运行结果，可以看到他们的计算结果正确：

```
wwwj@ubuntu:~/lab/lab3$ gcc -o lab3_2_2 lab3_2_2.c -lpthread
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_2 1
the sum in a thread: 499500
Sum of a[1000] = 499500
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_2 2
the sum in a thread: 248285
the sum in a thread: 251215
Sum of a[1000] = 499500
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_2 3
the sum in a thread: 165185
the sum in a thread: 167550
the sum in a thread: 166765
Sum of a[1000] = 499500
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_2 4
the sum in a thread: 139185
the sum in a thread: 129035
the sum in a thread: 115345
the sum in a thread: 115935
Sum of a[1000] = 499500
```

- 将线程数从1加到6的运行结果，可以看到他们的计算结果正确：

```
wwwj@ubuntu:~/lab/lab3$ gcc -o lab3_2_2 lab3_2_2.c -lpthread
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_2 1
the sum in a thread: 499500
Sum of a[1000] = 499500
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_2 2
the sum in a thread: 276575
the sum in a thread: 222925
Sum of a[1000] = 499500
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_2 3
the sum in a thread: 153795
the sum in a thread: 157395
the sum in a thread: 188310
Sum of a[1000] = 499500
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_2 4
the sum in a thread: 113335
the sum in a thread: 134015
the sum in a thread: 112890
the sum in a thread: 139260
Sum of a[1000] = 499500
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_2 5
the sum in a thread: 123225
the sum in a thread: 105545
the sum in a thread: 137115
the sum in a thread: 0
the sum in a thread: 133615
Sum of a[1000] = 499500
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_2 6
the sum in a thread: 117580
the sum in a thread: 0
the sum in a thread: 115380
the sum in a thread: 0
the sum in a thread: 123380
the sum in a thread: 143160
Sum of a[1000] = 499500
```

(3) 遇到的问题

这道题的代码实现并不难，但是我在这个过程中遇到了一个段错误：

```
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_1 8
Segmentation fault
```

然后在网上寻找解决方法，可以生成core文件然后利用gdb查找错误，即查看代码是在哪里宕掉的。

但是我没有生成core文件，查找解决方案，可以更改`core file size`，用指令`ulimit -a`查看，发现该值为0，然后将他设为无限制`ulimit -c unlimited`，更改后查看`core file size`的值变为`unlimited`：

```
wwwj@ubuntu:~/lab/lab3$ #ulimit -a
wwwj@ubuntu:~/lab/lab3$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 15435
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 15435
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
wwwj@ubuntu:~/lab/lab3$ ulimit -c unlimited
wwwj@ubuntu:~/lab/lab3$ gcc -o lab3_2_1 lab3_2_1.c -lpthread
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_1 8
this? global 0
Segmentation fault (core dumped)
wwwj@ubuntu:~/lab/lab3$ ulimit -a
core file size          (blocks, -c) unlimited
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 15435
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 15435
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

此时仍然没有生成core文件，根据一些博客找到了问题，是由于core文件产生的位置，采用指令`sudo bash -c "echo core > /proc/sys/kernel/core_pattern "`来进行写入，即指定程序所在目录为core文件生成目录，core文件名称为"core"。最后生成core文件：

```
wwwj@ubuntu:~/lab/lab3$ ls
core  lab3_1  lab3_1.c  lab3_2_1  lab3_2_1.c
wwwj@ubuntu:~/lab/lab3$
```

然后利用gdb查找错误：

```
wwwj@ubuntu:~/lab/lab3$ gdb lab3_2_1 core
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from lab3_2_1...(no debugging symbols found)...done.
[New LWP 24863]
[New LWP 24861]
[New LWP 24867]
[New LWP 24866]
[New LWP 24868]
[New LWP 24860]
[New LWP 24865]
[New LWP 24864]
[New LWP 24862]
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Core was generated by `./lab3_2_1 8'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x000055dde92c89a6 in computer ()
[Current thread is 1 (Thread 0x7fa8221ef700 (LWP 24863))]
(gdb) bt
#0  0x000055dde92c89a6 in computer ()
#1  0x00007fa8235ea6db in start_thread (arg=0x7fa8221ef700) at pthread_create.c:463
#2  0x00007fa82331371f in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95
(gdb) q
wwwj@ubuntu:~/lab/lab3$ ./lab3_2_1 8
this? global 0
Segmentation fault (core dumped)
```

根据gdb给出的信息，问题出在在函数`computer()`，最后找到错误，代码正确运行。

参考文章：

- linux下pthread简单编程实例及gdb调试（core dumped）*myshilion*的专栏-*CSDN*博客gdb pthread

  https://blog.csdn.net/myshilion/article/details/21447387
- linux下不产生core文件的原因_逐梦-CSDN博客

  https://blog.csdn.net/qq_35621436/article/details/120870746
- 我们是怎么发现C++异常从堆栈追踪中消失的原因的

  https://zhuanlan.zhihu.com/p/59554240

3. Pthreads求解二次方程组的根

> 编写一个多线程程序来求解二次方程组$ax^2+bx+c=0$的根，使用下面的公式
>
> $$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$
>
> 中间值被不同的线程计算，使用条件变量来识别何时所有的线程都完成了计算

实现思路

利用互斥锁和忙等待来实现路障。

将公式分成4步，即需要4个线程来完成：

(1) $\frac{-b}{a}$

(2) $\sqrt{b^2-4ac}$

(3) $\frac{(2)}{2a}$

(4) $(1)+/-(3)$

第(3)步需要等第(2)步完成；

第(4)步需要等第(1)、(3)步完成。

代码

> 完整代码见lab3_3.c

- 需要用到的头文件和一些全局变量：

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<pthread.h>

int two_to_three=0;//在第3步判断是否完成第2步的条件变量
int onethree_to_four=0;//在第4步判断是否完成第1、3步的条件变量
double a,b,c;//方程的常数项
double ans[5];//每一步的计算结果
pthread_mutex_t first,second;//两个锁，分别用来保护两个条件变量
```

- 第一个线程调用的函数`step1()`：
  - 第2行，计算 $\frac{-b}{a}$
  - 第3-4行，通过加锁解锁进入临界区，给条件变量`onethree_to_four+1`

```
void *step1(){
    ans[0]=(-b)/(2*a);
    pthread_mutex_lock(&second);
    onethree_to_four++;
    pthread_mutex_unlock(&second);
}
```

- 第二个线程调用的函数`step2()`：
  - 第2行，计算$\sqrt{b^2-4ac}$
  - 第3-4行，通过加锁解锁进入临界区，给条件变量`two_to_three+1`

```
void *step2(){
    ans[1] = sqrt(b*b - 4*a*c);
    pthread_mutex_lock(&first);
    two_to_three++;
    pthread_mutex_unlock(&first);
}
```

- 第三个线程调用的函数`step3()`：
  - 第2行，用`while`循环进入一个忙等待，直到`two_to_three`变为1，说明第2步已经完成计算，可以进行这一步了；
  - 第3行，计算$\frac{(2)}{2a}$。

```
} != 1);
    ans[2] = ans[1]/(2*a);

    pthread_mutex_lock(&second);
    ++;
    pthread_mutex_unlock(&second);
}
```

- 第四个线程调用的函数`step4()`：
  - 第2行，进入一个忙等待，直到`onethree_to_four`变为1，说明第1、3步已经完成计算，可以进行这一步了；
  - 第3-4行，计算$(1)+/-(3)$；

```
void *step4(){
    while(onethree_to_four != 2);
    ans[3] = ans[0] + ans[2];
    ans[4] = ans[0] - ans[2];
}
```

- 函数`Get_args()`来获得我们在终端输入运行指令时同时输入的参数a、b、c：

```
void Get_args(int argc,char *argv[]){
    a = strtol(argv[1], NULL, 10);
    b = strtol(argv[2], NULL, 10);
    c = strtol(argv[3], NULL, 10);
}
```

- 主函数：
  - 第2行，调用函数获得a、b、c的值；
  - 第3-6行，判断该方程是否有解，若无解，结束程序；
  - 第8行，创建4个线程标识符；
  - 第10-11行，给两个锁初始化；
  - 第13-16行，创建4个线程，每个线程都有对应的线程标识符，分别调用函数`step1()、step2()、step3()、step4()`，无参数传入；
  - 第18行，四个线程结束运行，释放他们占用的资源；
  - 第12行，输出解得的两个值。

```
int main(int argc,char *argv[]){
    Get_args(argc, argv);
    if((b*b - 4*a*c)<0){
        printf("There is no solution to this equation.\n");
        return 0;
    }

    pthread_t pth[4];

    pthread_mutex_init(&first, NULL);
    pthread_mutex_init(&second, NULL);

    pthread_create(&pth[0], NULL, step1, NULL);
    pthread_create(&pth[1], NULL, step2, NULL);
    pthread_create(&pth[2], NULL, step3, NULL);
    pthread_create(&pth[3], NULL, step4, NULL);
```

```
17
18      for(int i=0; i<4; ++i){
19          pthread_join(pth[i], NULL);
20      }
21
22      printf("when a = %lf, b = %lf, c = %lf, x1 = %lf and x2 =
    %lf\n",a,b,c,ans[3],ans[4]);
23      return 0;
24  }
```

运行结果

- 下面是编译和运行这段代码的指令：

```
1  /* File:     lab3_3.c
2   *
3   * Purpose:  Use multithreading to calculate the solution of a quadratic
    equation in one variable step by step.
4   *
5   * Compile:   gcc -o lab3_3 lab3_3.c -lpthread -lm
6   *
7   * Run:      ./lab3_3 <a> <b> <c>
8   *           a,b,c is constant variables of the equation.
9   *
10  * Input:    none
11  * Output:   Two solutions of the equation.
12  */
```

注：虽然我在开头加上了`#include<math.h>`的头文件，但是编译的时候还是无法识别开根号的函数，无法链接到这个库，因此的编译的时候直接加上`-lm`链接到`math`库。

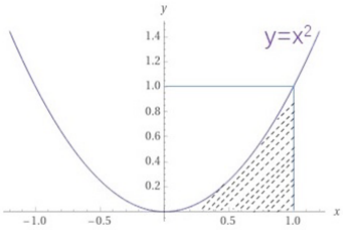
- 计算`a=1,b=1,c=1`和`a=2,b=5,c=3`情况下x的两个解

```
wwwj@ubuntu:~/lab/lab3$ gcc -o lab3_3 lab3_3.c -lpthread -lm
wwwj@ubuntu:~/lab/lab3$ ./lab3_3 1 1 1
There is no solution to this equation.
wwwj@ubuntu:~/lab/lab3$ ./lab3_3 2 5 3
when a = 2.000000, b = 5.000000, c = 3.000000, x1 = -1.000000 and x2 = -1.500000
wwwj@ubuntu:~/lab/lab3$
```

第一种情况无解，第二种情况的解为-1和-1.5，计算结果正确。

4. 编写一个多线程程序来估算面积

> Monte-carlo方法参考课本137页4.2题和本次实验作业的补充材料。
>
> 估算y=x^2曲线与x轴之间区域的面积，其中x的范围为[0,1]。



实现思路

利用"Monte-carlo方法"，用概率来估计：

在x的0-1上取n个随机数，在y的0-1上也取n个随机数；

获得n*n个点，加入这些点在阴影中的概率是$\frac{x}{n*n}$，即有x个点落在阴影里；

那么阴影的面积就是$S \times \frac{x}{n^2}$

代码

> 完整代码见lab3_4.c

- 需要用到的头文件和一些全局变量：
  - 第5行，n是总的投掷数，thread_num是线程数，part是将总点数按线程数划分后每个部分点的数量，shadowpoint是投掷结束后在阴影中的点数；
  - 第6行，mutex是创建的一个互斥锁，用来创造一个临界区更改shadowpoint值；
  - 第7行，用来记录所有点的坐标

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4  #include<pthread.h>
5
6  int n,thread_num,part,shadowpoint;
7  pthread_mutex_t mutex;
8  double *x,*y;
```

- 通过一个函数 `Get_args()` 来获得我们在终端输入运行指令时同时输入的参数n、thread_num，并通过计算获得part,将总点数按线程数划分后每个部分点的数量。

```
1  void Get_args(int argc,char *argv[]){
2      thread_num = strtol(argv[1], NULL, 10);
3      n= strtol(argv[2], NULL, 10);
4      part = n/thread_num;
5  }
```

- 创建 `pthread` 后调用的函数 `judge()`：
  - 第2-3行，进行数据类型的转换，i指向的是分配给这个线程part个点后是从所有点中第几个点开始计算的；
  - 第4-10行，进行part个点的判断：
    - 第5行，判断点$(x[(*i)+j], y[(*i)+j])$是否在阴影范围内；
    - 第6-9行，如果上一个条件成立的话，通过加锁和解锁进入临界区，给 `shadowpoint+1`。

```
void *judge(void *arg){
    int *i;
    i = (int*)arg;
    for(int j=0; j<part ; ++j){
        if(y[(*i)+j]<=x[(*i)+j]*x[(*i)+j]){
            pthread_mutex_lock(&mutex);
            shadowpoint++;
            pthread_mutex_unlock(&mutex);
        }
    }
}
```

- 主函数：
  - 第2行，调用函数`Get_args()`；
  - 第3-4行，创建`thread_num`个标识符，并给锁mutex初始化；
  - 第6-7行，给两个指针指向的地址分配空间；
  - 第9-13行，随机获得n个点，他们的范围在[0,1]；
  - 第15-19行，创建`thread_num`个线程，传入对应的标识符，调用的函数都为`judge()`，传入一个参数标识从第temp[i]个点开始计算；
  - 第21-23行，n个线程结束运行，释放他们占用的资源；
  - 第24-28行，输出总点数、在阴影里的点数，并且通过概率计算得到阴影面积大小并输出；

注：这里我们将n个点分成thread_num份后并不像之前计算a[1000]的和那样考虑没有被分配的点数。因为这里这些点并不是缺一不可的，之后计算比例，分母写的是`thread_num*part`，即分母也没有加上那些没有被分配的点，因此比例不变。假设我们输入的n是101，线程数为4，有1个点不能被分配到线程上去，那我们就只计算这100个点在阴影中的比例，对结果并不影响。

```
int main(int argc, char *argv[]){
    Get_args(argc,argv);
    pthread_t pth[thread_num];
    pthread_mutex_init(&mutex, NULL);

    x = (double*)malloc(n);
    y = (double*)malloc(n);

    srand(time(NULL));
    for(int i=0;i<n;++i){
        x[i]=((double)rand())/RAND_MAX;
        y[i]=((double)rand())/RAND_MAX;
    }

    int temp[thread_num];
    for(int i=0; i<thread_num; ++i){
        temp[i]=i*part;
        pthread_create(&pth[i], NULL, judge, &temp[i]);
    }

    for(int i=0; i<thread_num;++i){
        pthread_join(pth[i],NULL);
    }

    printf("Number of points = %d, The number of points in the shadow = %d\n",thread_num*part,shadowpoint);

    double area=(double)shadowpoint/(thread_num*part);
```

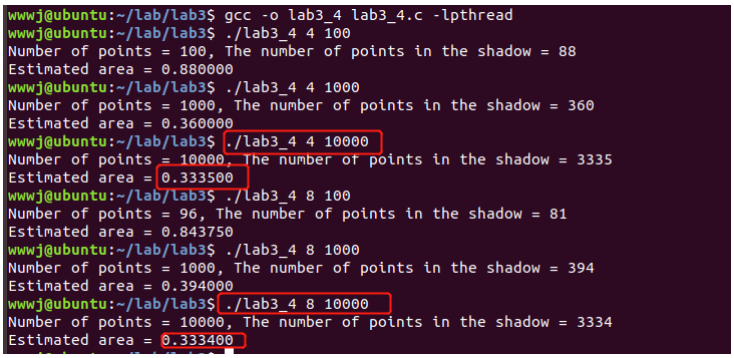
```
    printf("Estimated area = %lf\n",area);
}
```

运行结果

- 下面是编译和运行这段代码的指令：

```
/* File:     lab3_3.c
 *
 * Purpose:  利用Monte-carlo方法,估算y=x^2曲线与x轴之间区域的面积，其中x的范围为[0,1]。
 *
 * Compile:   gcc -o lab3_4 lab3_4.c -lpthread
 *
 * Run:      ./lab3_4 <n> <thread_num>
 *           n is the total number of throws
 *           thread_num is the number of thread that we will create;
 *
 * Input:    none
 * Output:   the number of points, The number of points in the shadow.
 *           Estimated area
 */
```

- 将点的数量从100、1000到1000，一直采用4个线程和8个线程计算：



根据积分计算可得y=x^2曲线与x轴之间区域的面积为$\frac{1}{3}$。

可以看到，投掷的点数越多，估算得到的面积越精确，在点数为10000的时候已经能得到0.3334的结果了。而仅仅采用100个点得到的结果和正确结果相比相差很多。