Lab6 - Pthreads并行构造

实验要求

1. **构造基于Pthreads的并行for循环分解、分配、执行机制**

模仿OpenMP的omp_parallel_for构造基于Pthreads的并行for循环分解、分配及执行机制。此内容延续上次实验，在本次实验中完成。

问题描述：生成一个包含parallel_for函数的动态链接库（.so）文件，该函数创建多个Pthreads线程，并行执行parallel_for函数的参数所指定的内容。

函数参数：parallel_for函数的参数应当指明被并行循环的索引信息，循环中所需要执行的内容，并行构造等。以下为parallel_for函数的基础定义，实验实现应包括但不限于以下内容：

```
parallel_for(int start, int end, int inc,
    void *(*functor)( int,void*), void *arg, int num_threads)
```

- start, end, inc分别为循环的开始、结束及索引自增量；
- functor为函数指针，定义了每次循环所执行的内容；
- arg为functor的参数指针，给出了functor执行所需的数据；
- num_threads为期望产生的线程数量。
- 选做：除上述内容外，还可以考虑调度方式等额外参数。

2. **Parallel_for 并行应用**

使用此前构造的parallel_for并行结构，将heated_plate_openmp改造为基于Pthreads的并行应用。

hated plate问题描述：规则网格上的热传导模拟，其具体过程为每次循环中通过对邻域内热量平均模拟热传导过程，即：

$$w_{i,j}^{t+1} = \frac{1}{4}(w_{i-1,j-1}^{t} + w_{i-1,j+1}^{t} + w_{i+1,j-1}^{t} + w_{i+1,j+1}^{t})$$

其OpenMP实现见课程资料中的heated_plate_openmp.c

要求：使用此前构造的parallel_for并行结构，将heated_plate_openmp实现改造为基于Pthreads的并行应用。测试不同线程、调度方式下的程序并行性能，并与原始heated_plate_openmp.c实现对比。

实验过程

一、构造基于Pthreads的并行for循环分解、分配、执行机制

1. 实现思路

基于openmp矩阵乘法实验进行修改，定义一个头文件`parallel.h`包含`parallel_for`函数，其中再定义结构体存储计算任务参数以及用来划分的额外参数。执行文件`matrix.c`实现矩阵乘法函数以及调用`parallel_for`即可

2. 代码实现

① 生成动态链接库的源文件`parallel.h`

定义结构体用于`pthread_creat`传递参数

```
struct ThreadData {
    // start和end是parallel_for函数创建线程额外需要传递的变量
    int start;
    int end;
    void* arg;  // 指向计算任务存储参数的结构体
};
```

定义`parallel_for`函数，其中参数functor改为`void* (*functor)(void*)`，因为使用了上面结构体定义的start和end，没必要再添加一个int变量存储不同线程的划分区间。实现的思路与之前实验使用同样的划分方式确定start和end，逐个划分区间并将计算任务的参数arg传给`thread_data[i].arg`后即可创建线程。

```
void parallel_for(int start, int end, int inc, void* (*functor)(void*), void*
arg, int num_threads) {
    pthread_t threads[num_threads];
    struct ThreadData thread_data[num_threads];
    int m = end - start;
    int rows_per_process = m / num_threads;
    int remaining_rows = m % num_threads;
    int offset = 0;
    for (int i = 0; i < num_threads; i++) {
        thread_data[i].start = offset;
        offset += (i < remaining_rows) ? rows_per_process + 1 :
rows_per_process;
        thread_data[i].end = offset;
        thread_data[i].arg = arg;
        pthread_create(&threads[i], NULL, functor, (void*)&thread_data[i]);
    }

    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], NULL);
    }
}
```

② 矩阵乘法文件`matrix.c`

定义结构体存储计算任务有关参数

```
struct functor_args {
    double* A, * B, * C;
    int m, k, n;
};
```

主函数在上一次使用openmp矩阵乘法实验基础上修改：只需赋值参数到结构体args中并将矩阵乘法入口函数改为

```
parallel_for(0, m, 1, matrix_multiple_thread, (void*)&args, num_threads);
```

其中`matrix_multiple_thread`为定义的矩阵乘法函数

```
void* matrix_multiple_thread(void* args) {
    struct ThreadData* data = (struct ThreadData*)args;
    struct functor_args* arg = (struct functor_args*)data->arg;
    for (int B_i = 0; B_i < arg->n; B_i++) {
        for (int A_i = data->start; A_i < data->end; A_i++) { // start、end对A矩阵
的行划分
            arg->C[A_i * arg->n + B_i] = 0;
            for (int A_j = 0; A_j < arg->k; A_j++) {
                arg->C[A_i * arg->n + B_i] += arg->A[A_i * arg->k + A_j] * arg-
>B[A_j * arg->n + B_i];
            }
        }
    }
    pthread_exit(NULL);
}
```

3. 运行结果

```
gcc parallel.h -fPIC -shared -o libpa.so -lpthread
gcc -g -Wall matrix.c -L. -lpa -o mat -lpthread
./mat 4
```

使用`export LD_LIBRARY_PATH=$(pwd)`将动态链接库所在的文件添加到环境变量中，使用`ldd mat`查看opm.o文件是否成功链接到动态链接库

```
ehpc@61583b2ed2e2:~/data$ export LD_LIBRARY_PATH=$(pwd)
ehpc@61583b2ed2e2:~/data$ ldd mat
        linux-vdso.so.1 (0x00007ffc27a82000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f31c0676000
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f31c0285000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f31c0a98000)
```

执行结果

```
ehpc@61583b2ed2e2:~/data$ gcc parallel.h -fPIC -shared -o libpa.so -lpthread
ehpc@61583b2ed2e2:~/data$ gcc -g -Wall matrix.c -L. -lpa -o mat -lpthread
ehpc@61583b2ed2e2:~/data$ ./mat 4
Enter values for m, k, n (128-2048): 128
 Initializing data for matrix multiplication C=A*B for matrix
 A(128x128) and matrix B(128x128)
Time taken: 0.028970 seconds
average time taken: 0.028970
 Computations completed.
 Top left corner of matrix A:
         14          22          74           6          19          48
          7          90          91          24           8          91
         92          97           0          94          99           0
         13          78          28          18          83          93
         65          14          98          60          38          62
          5          41          80          87          69          17

 Top left corner of matrix B:
         77          19          45          53          85          64
         10          41          35          75          81          19
         83          78          73          77          50          91
         21           7          31           7          53          82
         46          65          64          79          38          73
         40          25          36          64          84          49

 Top left corner of matrix C:
 2.8332E+05  2.8782E+05  3.0264E+05  3.2122E+05  3.0641E+05  3.2417E+05
 2.8719E+05  2.8782E+05  2.8697E+05  3.3602E+05  3.2235E+05  3.1527E+05
 2.9265E+05  2.9758E+05  3.2972E+05  3.4045E+05  3.2847E+05  3.4729E+05
 2.3667E+05  2.4547E+05  2.6966E+05  2.9431E+05  2.7807E+05   2.786E+05
 3.2232E+05  2.9155E+05  3.0637E+05  3.5516E+05  3.1732E+05  3.2802E+05
 2.8698E+05  2.8413E+05    3.17E+05  3.4328E+05  3.0106E+05  3.2864E+05
```

二、Parallel_for 并行应用

1. 实现思路

定义结构体存储传递到线程参数，包括矩阵w、u(将w、u矩阵使用`(double*)malloc(M * N * sizeof(double))`初始化)，mean均值（在初始化过程每个线程通过互斥锁增加mean值），my_diff（同理通过互斥锁进行更新）。对于openmp中每个`# pragma omp parallel`的范围定义同等效果的Pthread线程函数替换

2. 代码实现

```
struct functor_args {
    double * w, * u;
    double mean;
    double my_diff;
};
```

实现线程函数，主要包括初始化的三个函数以及迭代过程的三个函数，在openmp版本的基础上进行修改

```
// 初始化过程
void* initial_value_1(void* args) {
    struct ThreadData* data = (struct ThreadData*)args;
    struct functor_args* arg = (struct functor_args*)data->arg;
    double sum = 0;
    for (int i = data->start + 1; i < data->end + 1; i++)
    {
        arg->w[i * N + 0] = 100.0;
```

```
        arg->w[i * N + N - 1] = 100.0;
        sum = sum + arg->w[i * N + 0] + arg->w[i * N + N - 1];
    }
    pthread_mutex_lock(&mutex); // 修改共享变量mean
    arg->mean += sum;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
void* initial_value_2(void* args) {
    struct ThreadData* data = (struct ThreadData*)args;
    struct functor_args* arg = (struct functor_args*)data->arg;
    double sum = 0;
    for (int j = data->start; j < data->end; j++)
    {
        arg->w[(M - 1) * N + j] = 100.0;
        arg->w[j] = 0.0;
        sum = sum + arg->w[(M - 1) * N + j] + arg->w[j];
    }
    pthread_mutex_lock(&mutex); // 修改共享变量mean
    arg->mean += sum;
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
void* initialize_solution(void* args) {
    struct ThreadData* data = (struct ThreadData*)args;
    struct functor_args* arg = (struct functor_args*)data->arg;
    for (int i = data->start+1; i < data->end+1; i++)
    {
        for (int j = 1; j < N - 1; j++)
        {
            arg->w[i * N + j] = arg->mean;  // 赋值w矩阵
        }
    }
    pthread_exit(NULL);
}
// 迭代过程
void* save_u(void* args) {
    struct ThreadData* data = (struct ThreadData*)args;
    struct functor_args* arg = (struct functor_args*)data->arg;
    for (int i = data->start; i < data->end; i++)
    {
        for (int j = 0; j < N; j++)
        {
            arg->u[i * N + j] = arg->w[i * N + j];  // 使用u矩阵存储w矩阵
        }
    }
    pthread_exit(NULL);
}
void* new_w(void* args) {
    struct ThreadData* data = (struct ThreadData*)args;
    struct functor_args* arg = (struct functor_args*)data->arg;
    for (int i = data->start + 1; i < data-> end + 1; i++)
    {
        for (int j = 1; j < N - 1; j++)
        {
```

```
            arg->w[i * N + j] = (arg->u[(i - 1) * N + j] + arg->u[(i + 1) * N +
j] + arg->u[i * N + j - 1] + arg->u[i * N + j + 1]) / 4.0;  // 使用u矩阵计算新的w矩
阵
        }
    }
    pthread_exit(NULL);
}
void* update_diff(void* args) {
    struct ThreadData* data = (struct ThreadData*)args;
    struct functor_args* arg = (struct functor_args*)data->arg;
    double tmp;
    for (int i = data->start + 1; i < data->end + 1; i++)
    {
        for (int j = 1; j < N - 1; j++)
        {
            tmp = fabs(arg->w[i * N + j] - arg->u[i * N + j]);
            pthread_mutex_lock(&mutex); // 判断是否更新my_diff共享变量
            if (arg->my_diff < tmp)
            {
                arg->my_diff = tmp;
            }
            pthread_mutex_unlock(&mutex);
        }
    }
    pthread_exit(NULL);
}
```

3. 运行结果

```
gcc parallel.h -fPIC -shared -o libpa.so -lpthread
gcc -g -Wall heated_plate_pthread.c -L. -lpa -o pth -lpthread
./pth 16
# 改动前
gcc -g -Wall -fopenmp -o oph heated_plate_openmp.c
./oph   #原文件默认使用最大线程数，即16
```

Pthread版本：

```
HEATED_PLATE_OPENMP
  C/PTHREAD version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 500 by 500 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of threads =              16

  MEAN = 74.949900

 Iteration  Change

         1  18.737475
         2  9.368737
         4  4.098823
         8  2.289577
        16  1.136604
        32  0.568201
        64  0.282805
       128  0.141777
       256  0.070808
       512  0.035427
      1024  0.017707
      2048  0.008856
      4096  0.004428
      8192  0.002210
     16384  0.001043

     16955  0.001000

  Error tolerance achieved.
  Wallclock time = 3281.930656

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

openmp版本：

```
  C/OpenMP version
  A program to solve for the steady state temperature distribution
  over a rectangular plate.

  Spatial grid of 500 by 500 points.
  The iteration will be repeated until the change is <= 1.000000e-03
  Number of processors available = 16
  Number of threads =              16

  MEAN = 74.949900

 Iteration  Change

        1  18.737475
        2  9.368737
        4  4.098823
        8  2.289577
       16  1.136604
       32  0.568201
       64  0.282805
      128  0.141777
      256  0.070808
      512  0.035427
     1024  0.017707
     2048  0.008856
     4096  0.004428
     8192  0.002210
    16384  0.001043

    16955  0.001000

  Error tolerance achieved.
  Wallclock time = 551.201924

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

4. 性能对比

使用Pthread版本，记录不同线程数量（1-16）下的时间开销

线程数	耗时(s)
1	24.698
2	84.294
4	141.107
8	182.523
16	294.196

随着线程数变多，耗时逐渐增加，多线程并行并没有起到多大作用，猜测是在线编程云主机只使用到了一个核实现多线程，导致只有一个线程时效率反而最高，而多个线程增加了调度时间开销，导致性能更低。

Pthread版本与openmp版本对比，如上文运行图片结果，Pthread版本的耗时更长，使用openmp的运行速度更快。