中山大学计算机学院

本科生实验报告

（2025学年春季学期）

课程名称: 并行程序设计

实验	parallel_for并行应用	专业(方向)	计算机科学与技术
学号	--	姓名	--
Email	--	完成日期	2025.5.10

实验目的

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

实验过程和核心代码

虽然前面的初始化过程并不计入运行用时, 但为了统一, 前面使用openmp并行的初始化过程也使用parallel_for来进行改造. 为了加快程序运行速度, 初始化的并行线程一律使用最大线程数 `MAX_THREAD`

初始化过程中注意用到了 `mean` 和 `w`, 由于 `mean` 参数还使用了归约操作, 所有线程同时操作mean会产生数据竞争. 所以参数结构体应包含各个线程自己的 `thrad_mean`, 在处理完各个线程的 `thread_mean` 后进行加和. 参数结构体如下:

```
struct Args {
    double* mean;
    double* thread_mean;
    double** w;
    Args(double* _mean, double _w[M][N]): mean(_mean){
        thread_mean = new double[MAX_THREAD];
        set_thread_mean_zero();
        w = new double*[M];
        for(int i = 0; i < M; ++i) {
            w[i] = _w[i];
        }
    }
    void set_thread_mean_zero() {
        for(int i = 0; i < MAX_THREAD; ++i) {
            thread_mean[i] = 0;
        }
    }
    void reduce_mean() {
        for(int i = 0; i < MAX_THREAD; ++i) {
            *mean += thread_mean[i];
        }
    }
    ~Args() {
        delete [] w;
        delete [] thread_mean;
    }
};
```

由于参数结构需要操作自己的数据, 而之前实现的parallel_for的函数指针只有两个参数: `(int i, void* arg)`, 我们还需要线程的id才能根据id去操作对应的数据, 所以parallel_for中的函数指针也需要添加一个参数, 变为 `(int thrad_id, int i, void* arg)`.

接下来列举一些初始化的改造示例

```
// 设置边界值
parallel_for(1, M - 1, 1, [](int thread_id, int i, void* arg) -> void* {
    Args* args = static_cast<Args*>(arg);
    args->w[i][0] = 100.0;
}, arg, MAX_THREAD);

// 计算边界平均值
parallel_for(1, M - 1, 1, [](int thread_id, int i, void* arg) -> void* {
    Args* args = static_cast<Args*>(arg);
    args->thread_mean[thread_id] = args->thread_mean[thread_id] + args->w[i][0] + args->w[i][N - 1];
}, arg, MAX_THREAD);

arg->reduce_mean();
arg->set_thread_mean_zero();

parallel_for(0, N, 1, [](int thread_id, int j, void* arg) -> void* {
    Args* args = static_cast<Args*>(arg);
    args->thread_mean[thread_id] = args->thread_mean[thread_id] + args->w[M - 1][j] + args->w[0][j];
}, arg, MAX_THREAD);

arg->reduce_mean();

mean = mean / (double)(2 * M + 2 * N - 4);

// 初始化内部值为平均值
parallel_for(1, M - 1, 1, [](int thread_id, int i, void* arg) -> void* {
    Args* args = static_cast<Args*>(arg);
    for(int j = 1; j < N - 1; j++) {
        args->w[i][j] = *(args->mean);
    }
}, arg, MAX_THREAD);
```

接下来是具体的计算部分, 由于使用的参数不同, 所以这里创建了一个新的参数结构体:

```
struct Args2 {
    double** u;
    double** w;
    double* diff;
    double* my_diff;
    int thread_nums;
    Args2(double _u[M][N], double _w[M][N], double* _diff, int _thread_nums) {
        w = new double*[M];
        for(int i = 0; i < M; ++i) {
            w[i] = _w[i];
        }
        u = new double*[M];
        for(int i = 0; i < M; ++i) {
            u[i] = _u[i];
        }
        diff = _diff;
        thread_nums = _thread_nums;
        my_diff = new double[thread_nums];
    }
    void set_my_diff_zero() {
        for(int i = 0; i < thread_nums; ++i) {
            my_diff[i] = 0;
        }
    }
    void set_max_diff() {
        for(int i = 0; i < thread_nums; ++i) {
            *diff = std::max(my_diff[i], *diff);
        }
    }
    ~Args2(){
        delete []u;
        delete []w;
        delete [] my_diff;
    }
};
```

这里使用的thread_nums为计算过程中使用的线程数, 在运行可执行文件时由第一个参数决定

```
if(argc >= 2) {
    threads_num = std::stoi(argv[1]);
}
```

在计算过程中, 第一步将旧值存在`u`中以及第二步使用`u`来计算新的w值都比较简单, 不涉及数据竞争的问题. 第三步计算`diff`的过程中要取最大的`diff`, 先由每个线程求出自己负责的数据最大`my_diff`, 最后根据各个线程的`my_diff`决定`diff`

```
diff = 0.0;
arg2->set_my_diff_zero();

parallel_for(1, M - 1, 1, [](int thread_id, int i, void* arg) -> void* {
    Args2* args = static_cast<Args2*>(arg);
    for(int j = 1; j < N - 1; ++j) {
        if(args->my_diff[thread_id] < fabs(args->w[i][j] - args->u[i][j])) {
            args->my_diff[thread_id] = fabs(args->w[i][j] - args->u[i][j]);
        }
    }
}, arg2, threads_num);

arg2->set_max_diff();
```

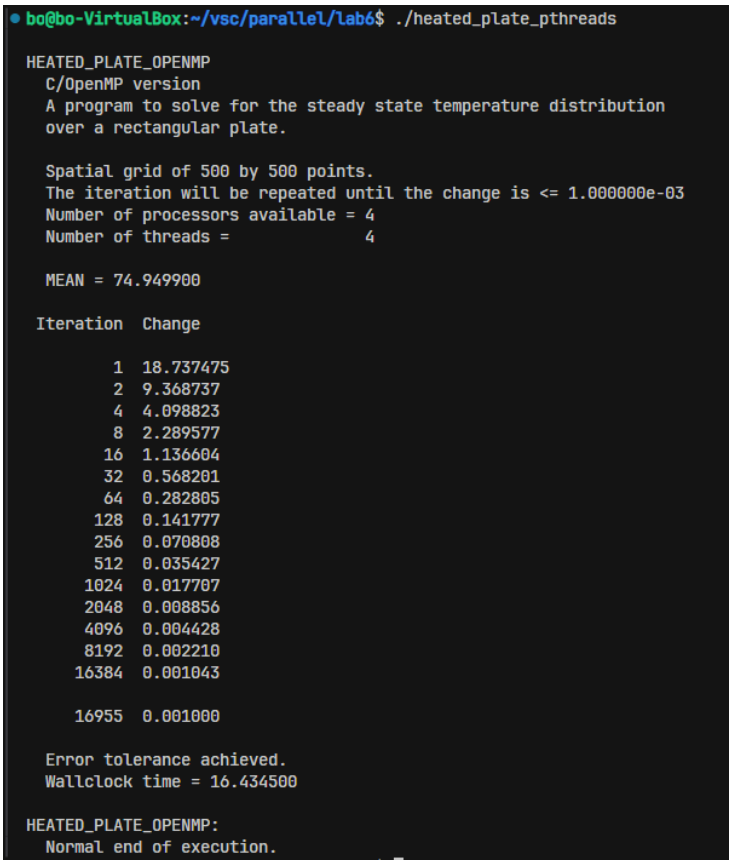
实验结果

虚拟机处理器个数: 4
时间单位: s

OpenMP

```
bo@bo-VirtualBox:~/vsc/parallel/lab6$ ./a.out

HEATED_PLATE_OPENMP
  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 = 4
  Number of threads =              4

  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 = 10.943649

HEATED_PLATE_OPENMP:
  Normal end of execution.
```

parallel_for



可以观察到使用`parallel_for`进行改造后运行的结果是正确的. 但是在相同的线程数下, 改造后程序的运行时间更久. 这是由于每一个使用`parallel_for`计算步骤都要有**线程的创建**开销, 而且**参数还需要进行多层的封装和转换**, 在具体的**任务执行时还需要通过函数指针来进行调用**, 有许多函数调用开销.

不同线程数的paralle_for运行结果

线程数	1	2	4	8	16
运行时间	42.5364	25.0455	16.4345	22.0013	35.3660

可以看到在线程数为虚拟机处理器个数时达到了最快的运行速度. 但是线程数小于等于4时加速比也没有接近理论值. 可能是由于每一个迭代计算过程分为了三个步骤, 每个步骤都要重新创建线程. 而且外层还有一个while循环, **导致线程创建的次数非常多**. 从而导致加速比较低. 而且随着线程数增多, 线程创建的开销更大, 导致8, 16个线程下运行的速度明显变慢.

实验感想

在本实验中, 我进一步加深了对parallel_for的理解, 并完善以使其能够处理线程私有数据. 此外, 本实验特殊之处在于线程的创建非常多, 加深了我对线程创建开销的了解. 我也了解到可以使用线程池来避免频繁创建线程的开销, 未来可以尝试线程池来进一步探索.