中山大学计算机学院本科生实验报告

课程名称：并行程序设计与算法

实验	**Pthreads并行构造**	专业（方向）	计算机科学与技术
学号	21307035	姓名	邓栩瀛
Email	dengxy66@mail2.sysu.edu.cn	完成日期	2024.4.29

1、实验目的

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。此部分可在下次实验报告中提交。

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。

示例：给定functor及参数如下：

```
struct functor_args{
  float *A, *B, *C;
};
void functor(int idx, void* args){
  functor_args *args_data = (functor_args*) args;
  args_data->C[idx] = args_data->A[idx] + args_data->B[idx];
}
```

调用方式如下：

```
functor_args args = {A, B, C};
parallel_for(0, 10, 1, functor, (void*)&args, 2)
```

该调用方式应当能产生两个线程，并行执行functor完成数组求和（$C_i = A_i + B_i$）。当不考虑调度方式时，可由前一个线程执行任务{0,1,2,3,4}，后一个线程执行任务{5,6,7,8,9}。也可以实现对调度方式的定义。

要求：完成parallel_for函数实现并生成动态链接库文件，并以矩阵乘法为例，测试其实现的正确性及效率

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。

heated 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：

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

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。

2、实验过程和核心代码

1.构造基于Pthreads的并行for循环分解、分配、执行机制执行命令

执行命令

```
g++ -c -fPIC -o parallel_for.o parallel_for.cpp -lpthread
g++ -shared -o parallel_for.so parallel_for.o -lpthread
g++ -o main1 main1.cpp -L./ -lparallel_for -lpthread
```

相关结构体定义

```
struct for_index{
    void *args;
    int start;
    int end;
    int increment;
};
```

parallel_for函数的定义

```
void parallel_for(int start, int end, int increment, void *(*functor)(void *), void
*arg, int num_threads){
    //内存分配：在堆上分配存储线程标识符的数组threads和存储线程执行参数的数组index_arr
    pthread_t *threads = (pthread_t *)malloc(num_threads * sizeof(pthread_t));
    for_index *index_arr = (for_index *)malloc(num_threads * sizeof(for_index));
    //计算每个线程要处理的块的大小
    int block = (end - start) / num_threads;
    //循环创建线程
    for (int i = 0; i < num_threads; i++){
        index_arr[i].args = arg;
        index_arr[i].start = start + i * block;
        index_arr[i].end = index_arr[i].start + block;
        if (i == (num_threads - 1))
            index_arr[i].end = end;
        index_arr[i].increment = increment;
        pthread_create(&threads[i], NULL, functor, (void *)(index_arr + i));
    }
    //等待线程完成
    for (int thread = 0; thread < num_threads; thread++)
        pthread_join(threads[thread], NULL);
    //释放内存
    free(threads);
    free(index_arr);
}
```

2.parallel_for并行应用

执行命令

```
gcc -fopenmp heated_plate_openmp.c -o heated_plate_openmp
gcc -o main2 main2.c -lpthread
```

多线程并行热传导计算：通过多线程分解热传导计算任务，每个线程处理一个子区域的计算，并使用线程同步来确保正确性，每次迭代都根据周围点的温度计算新温度值，并更新到新的数组中，计算完成后将新数组的值复制回原数组。

```
void* parallel_heat_conduction(void* arg) {
    struct for_index* index = (struct for_index*)arg;
    int start = index->start;
    int end = index->end;
    int increment = index->increment;
    //热传导计算
```

```
    for (int t = 0; t < iterations; t++) {
        for (int i = start; i < end; i += increment) {
            for (int j = 1; j < SIZE - 1; j++) {
                new_plate[i][j] = 0.25 * (plate[i - 1][j - 1] + plate[i - 1][j + 1] +
plate[i + 1][j - 1] + plate[i + 1][j + 1]);
            }
        }
        pthread_barrier_wait(&barrier);
        for (int i = start; i < end; i += increment) {
            for (int j = 1; j < SIZE - 1; j++) {
                plate[i][j] = new_plate[i][j];
            }
        }
        pthread_barrier_wait(&barrier);
    }
    //线程执行完毕
    pthread_exit(NULL);
}
```

初始化

```
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            plate[i][j] = 0.0;
        }
    }
```

设置边界条件

```
    for (int i = 0; i < SIZE; i++) {
        plate[i][0] = 1.0;
        plate[i][SIZE - 1] = 1.0;
        plate[0][i] = 1.0;
        plate[SIZE - 1][i] = 1.0;
    }
```

调用parallel_for

```
parallel_for(1, SIZE - 1, 1, parallel_heat_conduction, NUM_THREADS);
```

3、实验结果

1.构造基于Pthreads的并行for循环分解、分配、执行机制

线程数	矩阵规模			
	128	256	512	1024
1	0.010448	0.09501	0.8778	11.5687
2	0.009779	0.121958	1.07026	12.9785
4	0.009124	0.127034	1.04925	12.6656
8	0.009769	0.118113	1.29845	11.8409
16	0.009734	0.122188	1.11006	12.2015

实验结果分析：

1. 随着线程数的增加，对于较小的矩阵规模，执行时间并没有明显减少，甚至在某些情况下稍微有所增加。
2. 对于较大的矩阵规模，随着线程数增加，执行时间呈现出先减少后增加的趋势。由此可见，增加线程数可以带来一定程度的性能提升，但在某个临界点之后，继续增加线程数可能导致性能下降。
3. 在某些情况下，增加线程数并没有明显改善执行时间，可能是因为任务的并行性受到了某些限制，无法充分利用增加的线程资源。
4. 随着矩阵规模的增加，执行时间也呈现出增加的趋势。

2.parallel_for并行应用

迭代次数为1024

对于`heated_plate_openmp.c`，线程数量为2

矩阵规模	**128**	**256**	**512**	**1024**
运行时间（秒）	0.144	0.467	2.055	segment fault

基于Pthreads

线程数	矩阵规模			
	128	256	512	1024
1	0.101	0.390	1.615	7.356
2	0.118	0.366	2.158	4.666
4	0.142	0.390	1.282	4.700
8	0.190	0.516	1.196	4.835
16	0.204	0.485	1.424	4.868

实验结果分析：

1.在OpenMP实现中，随着矩阵规模的增加，运行时间也呈现出明显的增加，同时，在1024x1024的规模下甚至导致了segment fault错误，可能是由于资源限制导致的。在基于Pthreads的实现中，随着矩阵规模的增加，运行时间也增加，但是在单线程和多线程的情况下都没有出现类似的错误。

2.在基于Pthreads实现中，随着线程数量的增加，总体上运行时间有所减少，但并不是呈现线性减少，这表明在某个点之后，增加更多的线程并不能进一步提高性能，可能会有性能的损耗。

3.当线程数相同的情况下，基于Pthreads的实现方式，在较小规模下问题的情况下，运行时间略短于OpenMP，而在较大规模问题的情况下，基于Pthreads的实现方式其运行时间略长于OpenMP，可能是因为Pthreads的线程管理开销和同步开销稍高于OpenMP，在小规模问题上这些开销表现得更加明显，而在大规模问题上这些开销相对较小。

4、实验感想

1. 并行编程涉及到多线程之间的协调和同步，以及资源管理等方面的挑战，通过多次实验的学习及实践，对并行程序设计有了更加深入的学习和理解。
2. 在选择并行实现方式时，需要权衡性能、稳定性和可移植性等因素，有时候简单的实现方式可能会牺牲一些性能，但能够提供更稳定的结果。
3. 线程的增加并不一定会带来性能的提升，在实际应用中，需要结合实际情况进行分析。
4. 对同一个问题，尝试多种并行实现方式，比较不同实现方式的优缺点和性能，从而找到最优的实现方式，体会并行实现方式对程序性能提升的重要性。