并行快速排序算法实验报告

一、 实验目的

1. **掌握多线程编程技术**：通过使用 OpenMP 库，深入理解多线程环境下的并行程序设计方法。
2. **理解分治算法的并行化**：分析快速排序这一典型分治算法的并行化策略，特别是任务并行模式的应用。
3. **性能分析与优化**：探究在不同数据规模和不同线程数下的加速比，理解并行计算中的开销、负载均衡以及加速比瓶颈。

二、 算法设计与实现

2.1 核心思想

快速排序是一种典型的分治算法。并行化的核心思路是在进行划分后，将两个子区间的递归排序任务分配给不同的线程并行执行。

本实验采用 **OpenMP 的 Task 机制**。传统的 #pragma omp parallel for 不适合快速排序这种动态递归结构，而 task 指令允许动态创建任务并由运行时调度器分配给空闲线程，非常适合不平衡的递归树。

2.2 关键代码实现

2.2.1 基础串行排序与优化

为了防止递归过深导致栈溢出，并减少小数组排序时的函数调用开销，当数据量小于阈值（cutoff = 32）时，切换为插入排序。

```
// 插入排序：处理小规模数据
static inline void insertion_sort(vector<int>& a, int l, int r) {
    for (int i = l + 1; i <= r; ++i) {
        int x = a[i], j = i - 1;
        while (j >= l && a[j] > x) { a[j + 1] = a[j]; --j; }
        a[j + 1] = x;
    }
}
```

2.2.2 并行任务划分 (Task Parallelism)

这是并行化的核心部分。为了避免产生过多微小任务导致调度开销大于计算收益，引入了 depth_left 控制。只有在递归层级较浅时才创建新任务，深层递归转为串行执行。

```
void quicksort_task(vector<int>& a, int l, int r, int depth_left, int cutoff) {
    if (l >= r) return;
    // 优化1：小数据量使用插入排序
    if (r - l + 1 <= cutoff) { insertion_sort(a, l, r); return; }

    int m = hoare_partition(a, l, r); // Hoare划分

    // 优化2：控制任务粒度，仅在前几层递归使用并行Task
    if (depth_left > 0) {
        #pragma omp task default(none) firstprivate(l, m, depth_left, cutoff) shared(a)
        { quicksort_task(a, l, m, depth_left - 1, cutoff); }

        #pragma omp task default(none) firstprivate(r, m, depth_left, cutoff) shared(a)
        { quicksort_task(a, m + 1, r, depth_left - 1, cutoff); }

        #pragma omp taskwait // 等待子任务完成
    } else {
        // 超过深度限制，转为串行递归
        quicksort_serial(a, l, m, cutoff);
        quicksort_serial(a, m + 1, r, cutoff);
    }
}
```

2.2.3 计时策略（小规模放大）

由于小数据量（如1K）的排序时间极短（微秒级），直接计时误差极大。实验设计了自动放大策略 choose_loops，通过重复运行多次取平均值来获得精确时间。

```
static inline size_t choose_loops(size_t n) {
    if (n <= 1000)   return 4000;   // 1K数据循环4000次
    if (n <= 5000)   return 1000;
    // ...
    return 1;
}
```

三、 实验结果

实验分别在数据量 N=1000,5000,10000,100000以及线程数 T=1,2,4,8的情况下进行了测试。时间单位为微秒（us），加速比计算公式为$S = T_{serial}/T_{parallel}$

编译运行方法：

```
g++ -O3 -std=c++17 -fopenmp .\parallel_quicksort_omp.cpp -o pq.exe
.\pq.exe
```

```
PS C:\Users\IScream\Desktop\study\并行分布式计算> .\pq.exe
# Parallel QuickSort (OpenMP tasks)
# reps=3 cutoff=32
N         threads   serial(us)      parallel(us)    speedup
1000      1         6.303           10.524          0.599
1000      2         10.168          28.053          0.362
1000      4         7.889           58.462          0.135
1000      8         7.831           138.062         0.057
5000      1         144.215         143.267         1.007
5000      2         139.788         125.287         1.116
5000      4         146.008         128.311         1.138
5000      8         143.334         203.813         0.703
10000     1         388.393         369.057         1.052
10000     2         356.690         221.050         1.614
10000     4         371.427         208.733         1.779
10000     8         361.423         261.060         1.384
100000    1         4208.100        4899.850        0.859
100000    2         4335.900        3408.100        1.272
100000    4         4380.600        2889.200        1.516
100000    8         4952.750        1798.350        2.754
```

3.1 测试结果数据表

数据规模 (N)	线程数	串行耗时 (us)	并行耗时 (us)	加速比
1,000	1	6.303	10.524	**0.60**
	2	10.168	28.053	0.36
	4	7.889	58.462	0.14
	8	7.831	138.062	0.06
5,000	1	144.215	143.267	1.01

数据规模 (N)	线程数	串行耗时 (us)	并行耗时 (us)	加速比
	2	139.788	125.287	1.12
	4	146.008	128.311	**1.14**
	8	143.334	203.813	0.70
10,000	1	388.393	369.057	1.05
	2	356.690	221.050	1.61
	4	371.427	208.733	**1.78**
	8	361.423	261.060	1.38
100,000	1	4208.100	4899.850	0.86
	2	4335.900	3408.100	1.27
	4	4380.600	2889.200	1.52
	8	4952.750	1798.350	**2.75**

3.2 结果分析

- **小规模数据 (N=1K, 5K)**：
  - **现象**：随着线程数增加，加速比反而下降，甚至远低于1（即并行比串行慢）。
  - **原因**：并行开销占主导地位。创建线程、分配任务、上下文切换以及线程同步所消耗的时间远大于排序这几千个整数所需的计算时间。此时并行化是得不偿失的。
- **中规模数据 (N=10K)**：
  - **现象**：在 2线程和 4线程时获得了约 1.6x~1.7x 的加速比，但在 8线程时性能回落。
  - **原因**：计算量开始增加，能够抵消部分开销。但数据量仍不足以填满 8 个线程的负载，过多的线程竞争反而导致了调度拥堵和缓存一致性问题。
- **大规模数据 (N=100K)**：
  - **现象**：加速比随线程数增加而显著提升。在 8线程下达到最大加速比 **2.75**。
  - **原因**：计算密集度足够高，分治算法能够有效地将负载分散到各个核心。每个线程处理的数据块足够大，使得“计算时间”远超“调度时间”。

四、 实验总结

本次实验成功使用 C++ 和 OpenMP 实现了并行快速排序算法，并通过不同维度的测试验证了其性能。

1. 数据规模是并行的关键：实验数据清晰表明，并行计算仅在数据量达到一定规模（N>10,000）时才能体现优势。对于小规模数据，串行算法通常更快。

2. OpenMP Task 模型：Task 模型非常适合处理非结构化的递归并行问题，比单纯的 parallel for 更加灵活。
3. 加速比限制：受限于内存带宽和算法本身的特性，简单的并行快排很难达到线性加速比，但在 10 万数据量级下，8线程获得 2.75倍的加速是一个合理的优化结果。