南開大學
Nankai University

计算机学院

并行程序设计实验报告

Pthread 和 OpenMP 编程实验
ANNS

姓名：杨宇翔

学号：2312506

专业：计算机科学与技术

2025 年 5 月 25 日

目录

1 前言

目前我已经实现了的算法有 sq,pq 和 ivfpq，其中 ivfpq 的效果是最好的。所以本次 pthread 和 OpenMP 编程将直接在 **IVF-PQ** 上进行实验。

2 pthread

2.1 算法设计

串行版本的 ivfpq 搜索步骤分为 3 个部分：

1. 计算与查询向量最近的 m 个粗簇
2. 计算查询向量的残差
3. 遍历 m 个粗簇查找最近的 top_k 个向量

串行第一部分

```
for(int i = 0; i < 256; i++) {
    float dis = get_query_dis(ivf_centers+i*vecdim, query, vecdim);
    if(ivf_q.size() < m) {
        ivf_q.push({dis, i});
    } else {
        if(dis < ivf_q.top().first) {
            ivf_q.push({dis, i});
            ivf_q.pop();
        }
    }
}
```

计算查询向量最近的 m 个粗簇需要 256（粗簇数量）次循环，由于计算查询向量与各个粗簇间的距离时彼此间没有依赖关系，所以可以直接并行化，各个线程维护自己的最大堆，最后归并。

串行第二部分

```
for(int i=0; i<m; i++){
    size[i] = inverted_kmeans[query_ivf_labels[i]].size();
    candidate_base[i] = inverted_kmeans[query_ivf_labels[i]].data();
    residual[i] = new float[vecdim];

    float* center_ptr = &ivf_centers[query_ivf_labels[i]*vecdim];
    float* residual_ptr = residual[i];

    for(int j=0; j<vecdim; j+=4){
        float32x4_t center = vld1q_f32(center_ptr + j);
        float32x4_t qvec = vld1q_f32(query + j);
        float32x4_t residual_vec = vsubq_f32(qvec, center);
```

```
            vst1q_f32(residual_ptr + j, residual_vec);
        }
    }
```

计算查询向量的残差只有 m 次循环。由于需要遍历的粗簇数量 m 一般很小，所以此处并行意义不大。但由于为了下一步的并行化，此处需要做一些额外的处理，在下面解释。

串行第三部分

```
    for(int i = 0; i < m; i++) {
        for(int j=0;j<size[i];j++){
            int candidate_query = candidate_base[i][j];
            float dis = 0;
            for(int k=0;k<4;k++){
                dis+=get_query_dis(residual[i]+k*vecdim/4,centers+k*256*vecdim/4
                +compressed_base[candidate_query*4+k]*vecdim/4, vecdim/4);
            }
            if(q.size() < k_top) {
                q.push({dis, candidate_query});
            } else {
                if(dis < q.top().first) {
                    q.push({dis, candidate_query});
                    q.pop();
                }
            }
        }
    }
```

遍历 m 个粗簇查找最近的 top_k 个向量，这里是整个搜索过程最耗时的地方。但在此处并行化存在二层循环的问题。可以看到循环最外层是遍历 m 个粗簇，而第二层是遍历每个粗簇下的 size[i] 个向量。每个粗簇下的向量数量是不确定的，且该数量依赖于外层循环的 i。如果线程数量大于或除不尽粗簇数量 m，就很难分配任务，即使强行划分也容易出现负载不平衡的情况。所以我们需要将这两层循环打开，提前将所有需要遍历的向量从粗簇中拿出来放在一维数组内，便于均匀分配任务。

此处不能直接并行化的原因还有 ivf 索引向量存储的问题。代码中的 candidate_base 为指针数组，存储每个粗簇下的向量数组的地址。如果不提前展开，此处访存就同时依赖外层循环的 i 和内层的 j，难以分配任务的同时，内存也不连续，所以必须将 candidate_base 展开。

并且由于用于计算距离的不是查询向量本身，而是在第二部分计算的查询向量与对应粗簇的残差，距离计算依赖于向量所属的粗簇编号，所以还需要知道所有位置上查询向量对应的粗簇编号。由于我们知道每个粗簇下的向量数量，且所有向量按粗簇编号顺序依次展开，所以可以根据各个向量处于一维数组中的位置得到自己所属的粗簇编号。

综上，第三部分的二层循环需要展开。而展开需要的一维向量数组和对应粗簇编号在第二部分时生成，供第三部分快速并行查询。

2.2 代码实现

1.0 版本里，我使用最简单的 pthread_create 和 pthread_join 方法做并行化，对各个任务做静态切分，并用结构体来给各个线程传递参数。

第一部分

```
    for(int thread_id=0;thread_id<thread_number;thread_id++){
        ivfdis_parm[thread_id] = {
            compressed_base,centers,ivf_centers,inverted_kmeans,query,base_number,vecdim,
            k_top,m,&amutex,thread_number,thread_id,&ivf_q,my_q+thread_id
        };
        pthread_create(&threads[thread_id],NULL,ivfpq_ivfdis_thread,&ivfdis_parm[thread_id]);
    }

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

    for(int i=0;i<thread_number;i++){
        while (my_q[i].size()) {
            if(ivf_q.size() < m) {
                ivf_q.push(my_q[i].top());
            } else {
                if(my_q[i].top().first < ivf_q.top().first) {
                    ivf_q.pop();
                    ivf_q.push(my_q[i].top());
                }
            }
            my_q[i].pop();
        }
    }
```

第二部分

```
int* ivf_size_flag = new int[m+1];
    ivf_size_flag[0] = 0;
    for(int i=0;i<m;i++){
        int size = inverted_kmeans[query_ivf_labels[i]].size();
        ivf_size_flag[i+1] = ivf_size_flag[i] + size;
    }
    int size_all = ivf_size_flag[m];

    int *candidate_queries = new int[size_all];
    int *residual_i = new int[size_all];
    float **residual = new float*[m];

    for(int i=0; i<m; i++) {
        int start = ivf_size_flag[i];
        int end = ivf_size_flag[i+1];
        std::vector<int> *candidate_base = &inverted_kmeans[query_ivf_labels[i]];
        for(int j=0; j<end-start; j++) {
            candidate_queries[start+j] = (*candidate_base)[j];
            residual_i[start+j] = i;
```

```
        }
        residual[i] = new float[vecdim];
        float* center_ptr = &ivf_centers[query_ivf_labels[i]*vecdim];
        float* residual_ptr = residual[i];
        for(int j=0; j<vecdim; j+=4){
            float32x4_t center = vld1q_f32(center_ptr + j);
            float32x4_t qvec = vld1q_f32(query + j);
            float32x4_t residual_vec = vsubq_f32(qvec, center);
            vst1q_f32(residual_ptr + j, residual_vec);
        }
    }
```

第三部分

```
    for(int thread_id=0;thread_id<thread_number;thread_id++){
        pq_parm[thread_id] = {
            compressed_base,centers,vecdim,k_top,size_all,&amutex,&main_q,
            candidate_queries,residual,residual_i,thread_number,thread_id,my_q+thread_id
        };
        pthread_create(&threads[thread_id],NULL,ivfpq_pq_thread,&pq_parm[thread_id]);
    }
    for(int j=0;j<thread_number;j++){
        pthread_join(threads[j],NULL);
    }

    for(int i=0;i<thread_number;i++){
        while (my_q[i].size()) {
            if(main_q.size() < k_top) {
                main_q.push(my_q[i].top());
            } else {
                if(my_q[i].top().first < main_q.top().first) {
                    main_q.pop();
                    main_q.push(my_q[i].top());
                }
            }
            my_q[i].pop();
        }
    }
```

虽然看似并行了，但实际运行后发现反而比串行慢了很多。首先就是第一部分，总共只循环 256 次，并行的开销原高于实际的效果，应该直接用串行版本。第二部分和第三部分在计时后发现也都比串行的慢，甚至线程开的越多跑的越慢。具体时间在实验部分有写。

2.3 实验分析

2.3.1 串行并行对比

上图是串行和并行的速度对比。两次实验的参数设置相同，粗簇数量 m 为 8，rerank 为 700，线程数为 8，所有数据均为 10 次实验的平均值。可以看到三个部分的速度都远远慢于串行版本。

```
成功读入inverted_kmeans.bin!
成功读入ivf_centers.bin!
成功读入ivf_base.bin!
成功读入ivfpq_kmeans_base.bin!
成功读入ivfpq_kmeans_centers.bin!
[0.902904,]
[602.566,]
IVF stage time: 12.3041 microseconds
candidates stage time: 0.509 microseconds
search stage time: 378.489 microseconds
```

图 2.1: 串行版本

```
成功读入inverted_kmeans.bin!
成功读入ivf_centers.bin!
成功读入ivf_base.bin!
成功读入ivfpq_kmeans_base.bin!
成功读入ivfpq_kmeans_centers.bin!
[0.902904,]
[1301.95,]
IVF stage time: 351.407 microseconds
candidates stage time: 22.8621 microseconds
search stage time: 667.519 microseconds
```

图 2.2: pthread 并行版本

第一部分可以在串行的图中看到这 256 次循环的耗时很短，只有 12ms，引入并行后飙升至 351，说明强行切分任务并行的开销过大。

第二部分由于串行版本只用循环 m 次，即计算 m 次残差，而并行版本需要提前遍历所有粗簇下的向量将其复制到一维数组中展开，所以多消耗了 22ms。但为了第三部分并行任务的均匀划分和内存地址连续，这点耗时是值得的。

第三部分则出乎意料的慢。明明已经将向量展开大小为 size_all 的一维数组中，且均匀地将所有向量划分到各个线程，在各个线程单独维护自己的最大堆，只在最终合并时上锁，性能却完全倒退。不过从第一部分就能看出 pthread_create 的开销太大了，可能对于 ivfpq 这样已经将计算规模从十万压缩到几千的算法来说，并行的开销远大于数据实际的规模，很难有好的效果。

2.3.2 线程数量对比

综合上面的分析，我在去除掉第一部分的并行，改回原本的串行版本后，单独更改线程数做了实验对比。rerank=700, m=8。

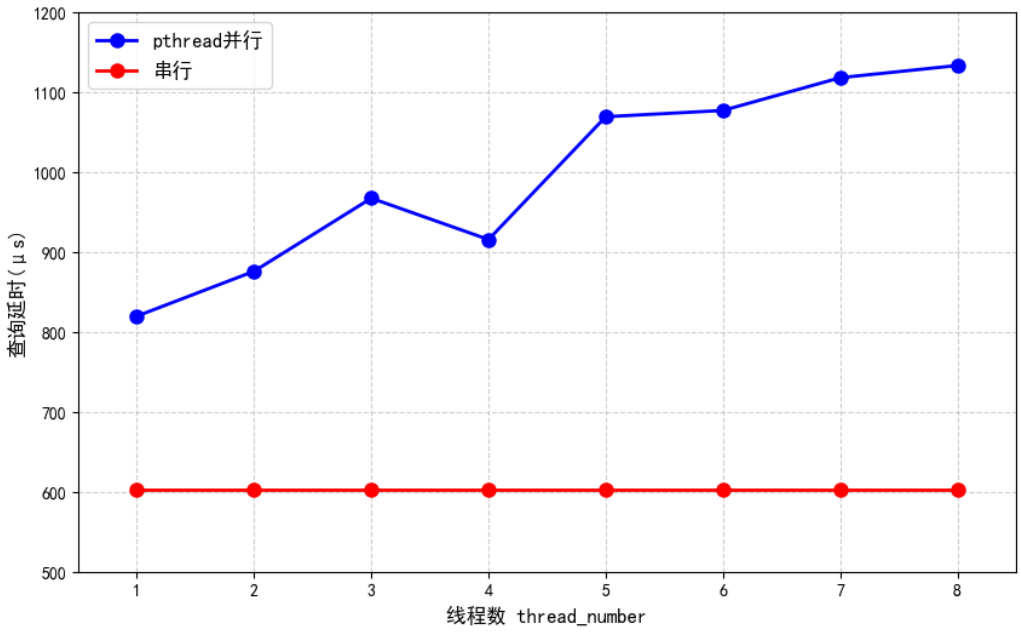

图 2.3: 线程数量对比

首先可以看到，去除了第一部分的并行后，查询延时降低了很多。其次从线程数量增加反而导致延时上升来看，并行线程的开销远大于其能带来的性能提升。

当然也不可能真把线程数降到 1，那就真无效并行了。综合一下考虑将线程数设置为 4，让后续实验在此基础上继续找办法优化。

2.3.3 rerank 和粗簇数量 m 的平衡

ivfpq 算法中，维护最大堆的开销主要取决于 rerank，而需要计算的距离数量取决于粗簇数量 m。在串行代码中，粗簇数量增加带来的计算开销大于维护最大堆的开销，所以平衡召回率时主要依靠提高 rerank 而不是增加需要遍历的粗簇数量 m。

```
成功读入inverted_kmeans.bin!
成功读入ivf_centers.bin!
成功读入ivf_base.bin!
成功读入ivfpq_kmeans_base.bin!
成功读入ivfpq_kmeans_centers.bin!
[0.902904,]
[602.566,]
IVF stage time: 12.3041 microseconds
candidates stage time: 0.509 microseconds
search stage time: 378.489 microseconds
```

图 2.4: rerank=700, m=8

```
成功读入inverted_kmeans.bin!
成功读入ivf_centers.bin!
成功读入ivf_base.bin!
成功读入ivfpq_kmeans_base.bin!
成功读入ivfpq_kmeans_centers.bin!
[0.903254,]
[654.104,]
IVF stage time: 15.2241 microseconds
candidates stage time: 1.32525 microseconds
search stage time: 526.848 microseconds
```

图 2.5: rerank=325, m=16

上图是 rerank=700, m=8 和 rerank=325, m=16 两次实验的串行代码对比，可以看到在维持召回率只相差 0.0003 的情况下，延时却高出了 50μs，三个部分的延时都提高了很多。后者 search stage time 比前者高出 148μs，但最终延时只相差 50μs 是后续根据 rerank 重排的耗时导致的。

但并行代码不同。并行算法主要优化了计算距离的部分，而 rerank 过高会导致每个线程各自维护的最大堆的开销增加，所以并行版本在平衡召回率时应该提高适当提高粗簇数量 m 而不是一味增加 rerank。这点在 pthread 这个反向优化里并不明显，因为此时粗簇数量带来的距离计算开销还是远高于维护堆的开销。但在后续的 OpenMP 的大力度优化下这点就很明显了。

```
成功读入inverted_kmeans.bin!
成功读入ivf_centers.bin!
成功读入ivf_base.bin!
成功读入ivfpq_kmeans_base.bin!
成功读入ivfpq_kmeans_centers.bin!
[0.902904,]
[857.531,]
IVF stage time: 13.7749 microseconds
candidates stage time: 24.8263 microseconds
search stage time: 582.729 microseconds
```

图 2.6: rerank=700, m=8

```
成功读入inverted_kmeans.bin!
成功读入ivf_centers.bin!
成功读入ivf_base.bin!
成功读入ivfpq_kmeans_base.bin!
成功读入ivfpq_kmeans_centers.bin!
[0.903254,]
[893.611,]
IVF stage time: 15.5167 microseconds
candidates stage time: 64.3101 microseconds
search stage time: 700.245 microseconds
```

图 2.7: rerank=325, m=16

上图是 pthread 并行版本下两种 rerank、m 平衡数值的实验对比。从上一个关于线程数量的实验得知 pthread=4 时效果比较好，所以这里已经将 pthread 调整为 4。图中可以看到后者虽然因为粗簇数量翻倍导致第二部分计算残差和展开数组的耗时大大提高，但第三部分的计算距离阶段耗时的增加幅度却比较小，说明我们的并行算法发力了。

2.3.4 总结

通过上面三个实验，我们可以得出以下结论：

首先由于 ivfpq 本身已经大大降低了数据规模，并行化的收益很低，甚至会出现反向优化的情况。对于原代码实现中冗余的第一部分和第二部分的并行应当舍弃，只保留耗时高的第三部分的并行优化。

然后由此我们应该适当降低线程数，过高的线程数只会增大开销。这点在后续的 OpenMP 中也有所体现。

最后是 rerank 和粗簇数量 m 的平衡中应当适当提高粗簇数量 m 的调整幅度，以配合并行优化的效果。

综上，将原始的 pthread 代码中第一部分的并行内容舍弃，保留第三部分，不对第二部分做并行优化。参数方面保留 rerank=700，m=8，将 thread_number 改为 4。此时在保证召回率不变的情况下，延时从 1301.95μs 降低到 857.531μs。

3 OpenMP

虽然我写的 pthread 是反向优化，但 OpenMP 是真的优化了，虽然幅度也很小。

3.1 算法设计

OpenMP 很好写，效果也很好，就是直接在我们 pthread 代码的基础上，把原有的 pthread_create 等并行结构替换成 OpenMP 的就可以了。唯一需要注意的是线程私有变量和共享变量的区分，和调度选项在 static 和 dynamic 间的选择。

第一部分由于耗时过短所以不用并行。虽然第二部分在 pthread 后续分析中选择了不做并行，但那是 pthread 开销过大的情况下的选择。从 pthread 的粗簇数量实验中可以知道，粗簇数量提升会导致第二部分的耗时飙升，但我们也需要提高粗簇数量来最大化利用我们的并行优化，所以在 OpenMP 中尝试加上并行优化。

最终版本设置的参数为 rerank=325，粗簇数量 m=16，线程数 thread_number=4，OpenMP 的调度方式选择 static。

3.2 代码实现

第二部分

```
    #pragma omp parallel num_threads(thread_number)
{
    #pragma omp for schedule(static)
    for(int i=0; i<m; i++) {
        int start = ivf_size_flag[i];
        int end = ivf_size_flag[i+1];
        std::vector<int> *candidate_base = &inverted_kmeans[query_ivf_labels[i]];
        for(int j=0; j<end-start; j++) {
            candidate_queries[start+j] = (*candidate_base)[j];
            residual_i[start+j] = i;
        }
        residual[i] = new float[vecdim];
        float* center_ptr = &ivf_centers[query_ivf_labels[i]*vecdim];
```

```
            float* residual_ptr = residual[i];
            for(int j=0; j<vecdim; j+=4){
                float32x4_t center = vld1q_f32(center_ptr + j);
                float32x4_t qvec = vld1q_f32(query + j);
                float32x4_t residual_vec = vsubq_f32(qvec, center);
                vst1q_f32(residual_ptr + j, residual_vec);
            }
        }
    }
```

第三部分

```
    #pragma omp parallel num_threads(thread_number)
    {
        std::priority_queue<std::pair<float, uint32_t> > local_q;

        #pragma omp for schedule(static)
        for(int i=0;i<size_all;i++){
            int candidate_query = candidate_queries[i];
            float dis = 0;

            for(int k=0;k<4;k++){
                dis+=get_query_dis(residual[residual_i[i]]+k*vecdim/4,
                centers+k*256*vecdim/4+compressed_base[candidate_query*4+k]*vecdim/4,
                    vecdim/4);
            }

            if(local_q.size() < k_top) {
                local_q.push({dis, candidate_query});
            } else if(dis < local_q.top().first) {
                local_q.push({dis, candidate_query});
                local_q.pop();
            }
        }

        #pragma omp critical
        {
            while(!local_q.empty()) {
                auto elem = local_q.top();
                if(q.size() < k_top) {
                    q.push(elem);
                } else if(elem.first < q.top().first) {
                    q.push(elem);
                    q.pop();
                }
                local_q.pop();
            }
        }
    }
```

3.3 实验分析

3.3.1 性能对比

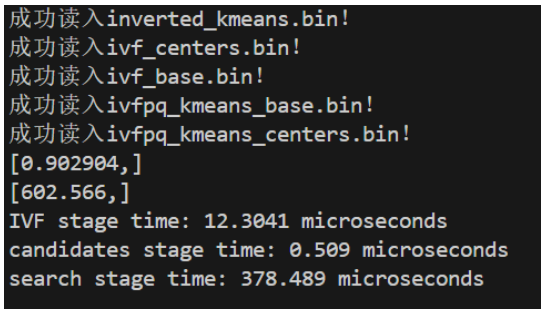

图 3.8: 串行版本

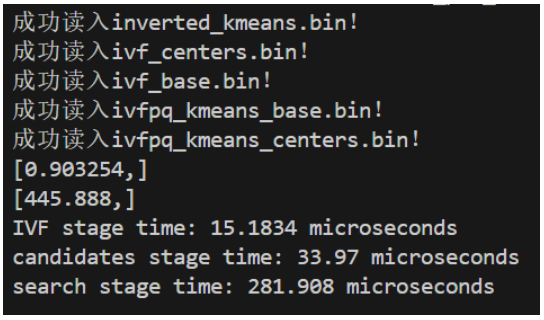

图 3.9: OpenMP 版本

上图是串行和 OpenMP 并行优化后的实验对比。串行的参数还是 rerank=700,m=8，而 OpenMP 的实验参数为 rerank=325,m=16。之所以后者改成 m=16，就是因为前面实验提到的并行优化后粗簇数量增加带来的开销会被大幅削弱。可以看到即使需要计算的距离数量翻倍了，搜索阶段（第三部分）的耗时却比原本串行的还要少。第二部分的耗时相较 pthread 没有做并行化的相同参数下 64.3101μs 的耗时也降低了很多。说明第二部分的并行化是发挥了作用的。

3.3.2 rerank 和粗簇数量 m 设置对比

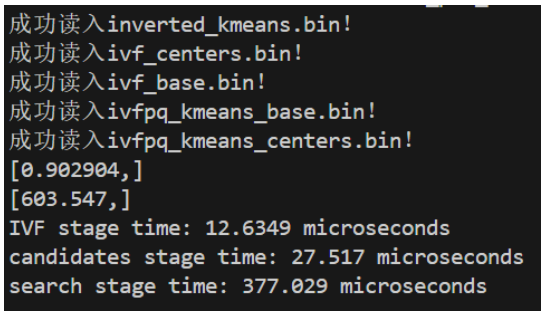

图 3.10: rerank=700, m=8

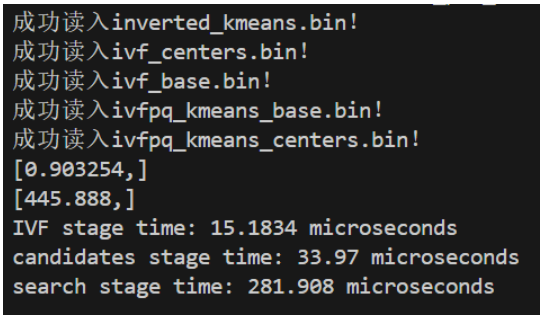

图 3.11: rerank=325, m=16

上面两图都是 OpenMP 并行优化后的实验，左图设置的参数为 rerank=700, m=8。右图设置的参数为 rerank=325, m=16。二者的 thread_number 都设置为 4。这个对比实验就是为了证明前面提到的并行化对粗簇数量 m 设置的影响。

可以看到和串行和 pthread 版本的结果相反，当 m 设置为 16 时，虽然第一部分和第二部分的耗时都因为粗簇数量增加而增加了，但第三部分的耗时反而降低了。原因就是之前分析的粗簇数量带来的计算量提升被并行化大幅削弱，rerank 提升带来的各个线程维护最大堆的开销则被放大了。所以在后续实验里选择设置参数为 rerank=325, m=16。

3.3.3 线程数量对比

这次实验参照前几次实验的结论，rerank=325, m=16，取 10 次实验的平均值。其实 10 次平均值有点少，但服务器一次只让跑这么久，改成 20 次就 killed 了。多跑几次 10 次也是个办法，但这种不连续的实验跑出来的数据往往也连续不起来，波动太大了。不太影响结论就讲究看罢。

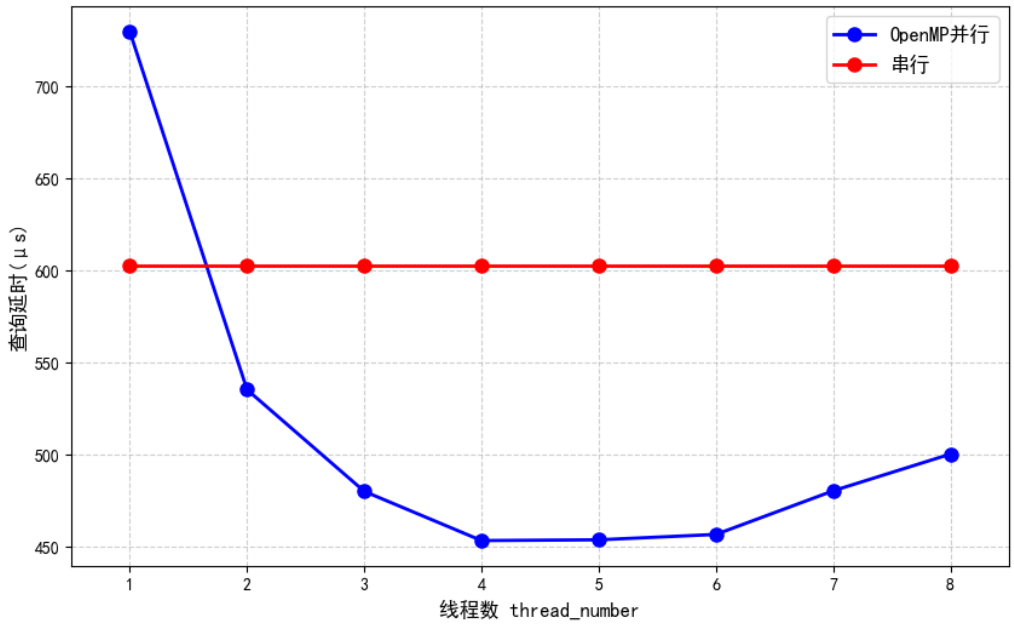

图 3.12: 线程数量对比

不得不说，这图比 pthread 的好看多了。可以看到这次多线程是真的起作用了。虽然线程数超过 4 后又开始上升，但这是数据规模太小导致的。跑了好几次 10 次的平均值，查询延时都和这张图上的基本一致，只是线程数等于 4 和 5 时的延时会有波动，有时候线程为 5 时延时更低。但考虑到线程划分任务，还是优先用 2 的指数，即线程数 4。

3.3.4 static 和 dynamic 对比

```
成功读入inverted_kmeans.bin!
成功读入ivf_centers.bin!
成功读入ivf_base.bin!
成功读入ivfpq_kmeans_base.bin!
成功读入ivfpq_kmeans_centers.bin!
[0.903254,]
[445.888,]
IVF stage time: 15.1834 microseconds
candidates stage time: 33.97 microseconds
search stage time: 281.908 microseconds
```

图 3.13: static

```
成功读入inverted_kmeans.bin!
成功读入ivf_centers.bin!
成功读入ivf_base.bin!
成功读入ivfpq_kmeans_base.bin!
成功读入ivfpq_kmeans_centers.bin!
[0.903254,]
[1270.97,]
IVF stage time: 14.4459 microseconds
candidates stage time: 37.6305 microseconds
search stage time: 1122.01 microseconds
```

图 3.14: dynamic

上图是分别使用 static 和 dynamic 的两次实验。前面的实验默认用的都是 static，这里单独展示 dynamic 的实验来做对比。参数设置都相同，rerank=325,m=16,thread_number=4。

可以看到 dynamic 的耗时惨不忍睹。前两个部分的耗时只是性能波动，基本一致。但第三部分的耗时直接飙升。性能差距这么大的原因也很好解释。由于我们已经在第二部分将原本需要遍历的向量从粗簇中拿出来，放在了一维数组里，且各个向量维度相同，计算量相同，就不存在负载不均衡的情况了。此时只用直接均分任务到各个线程即可，动态地分配任务反而会因为调度的开销大大增加延时。

从这次实验中也可以发现，ivfpq 算法的并行化受线程开销影响太大了。由于本身计算量就很小，并行化后线程的同步开销等会占据整个延时的很大一部分。所以前面 pthread 编程的效果不好应该也是意料中的结果（不会真是因为我写的太烂了吧）。

4 SQ 补充实验

前面的并行编程都是对 ivfpq 这个算法做优化，但优化的效果都不好，猜测是计算量太小导致的。于是我又对 SQ 这个简单暴力遍历的算法试着做 pthread 和 OpenMP 优化，来和 ivfpq 的做对比。实验结果相当的好：

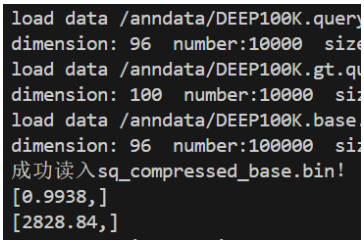

图 4.15: 串行 SQ

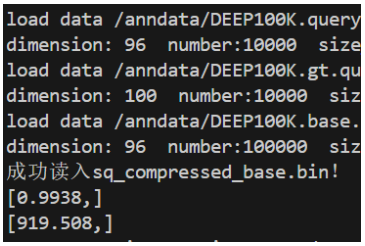

图 4.16: pthread

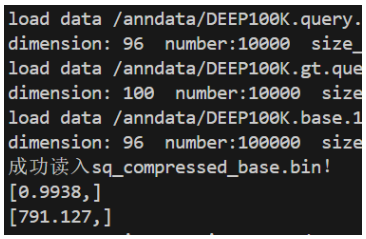

图 4.17: OpenMP

上图是经过参数调整后的三次实验结果。SQ 算法需要设置的参数只有线程数 thread_number 和 rerank。pthread 选择的线程数为 6，OpenMP 选择的线程数是 8。三次实验的 rerank 都设置为 200。结果取的 10 次实验平均值（直接在单次运行里循环 10 次实验，之前的实验也都是这样）。

可以看到结果非常的 amazing 啊！首先 SQ 算法由于是暴力遍历，正确率有着天然的巨大优势，在 SQ 算法上调整召回率和 ivfpq 的 0.90 对齐没有意义。调整 rerank 只会影响最大堆的开销，距离计算量是恒定的。于是在 SQ 保证召回率高达 0.99 的情况下，延时达到了三位数，已经和 ivfpq 算法的延时差距不大了。考虑到如此高的召回率，这点多出来的延时已经完全可以接受了。

通过 SQ 实验基本可以确定，ivfpq 算法的并行效果不好就是因为它本身已经大大降低了计算量，ivfpq 算法在串行的时候延时就已经降低到了 600，再并行化收益较低，甚至会面临并行开销大于优化效果的情况。当然这不是说 SQ 算法此时就比 ivfpq 算法好。虽然 SQ 算法现在的召回率达到了 0.99，但由于距离计算量恒定，即使通过调整 rerank 强行把召回率降到 0.90，延时也不会有明显的降低。而 ivfpq 则还有可以改进的空间。

关于 SQ 算法的 pthread 和 OpenMP 的线程数的设置，我又绘制了下面的图：

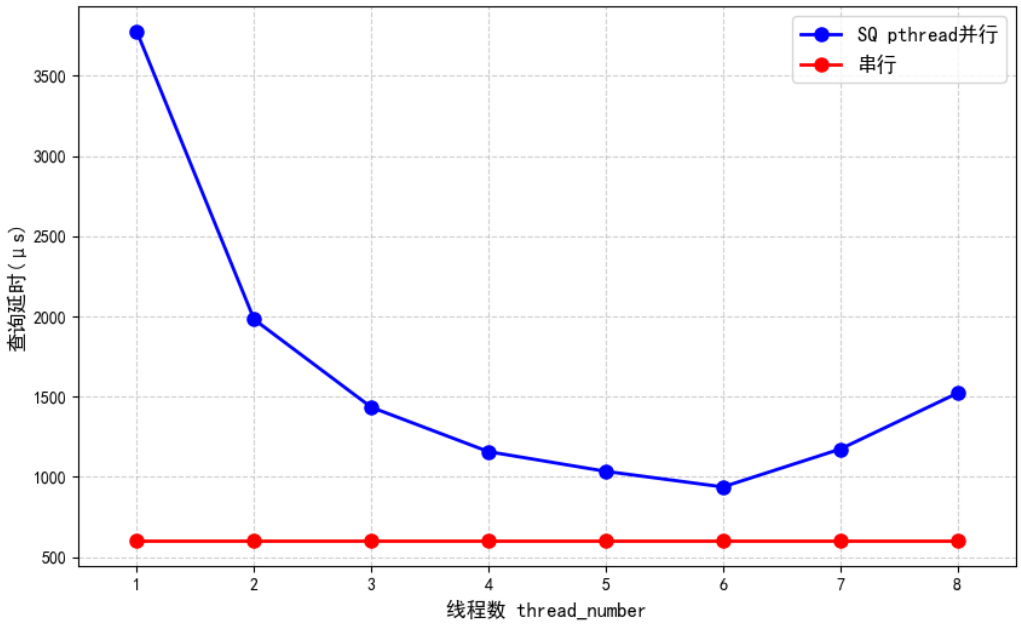

图 4.18: SQ 算法 pthread 线程数量对比

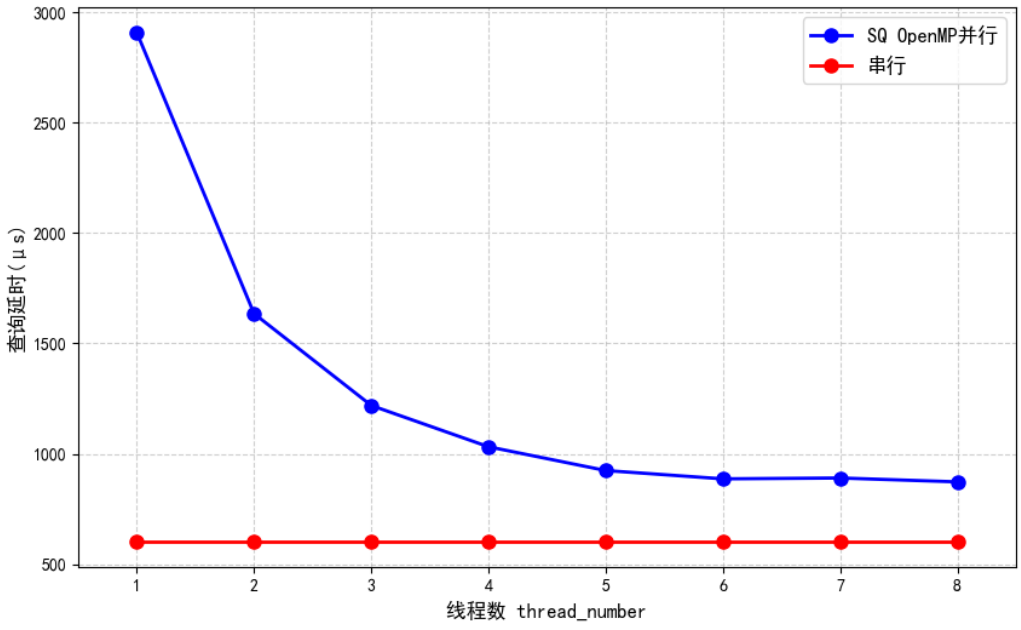

图 4.19: SQ 算法 OpenMP 线程数量对比

可以看到 pthread 线程数高后出现了开销大于优化的反弹，但 OpenMP 至少在 8 线程前保持着优化的趋势。据此在前面的性能测试中将线程数分别设置为了 6 和 8。从这两张图中还可以发现，线程数等于 1 时，pthread 的延时比 OpenMP 高了九百左右的延时，说明使用 pthread 的开销远大于 OpenMP。

5 总结

表 1: 各版本性能

指标	串行	pthread	OpenMP	SQ OpenMP
recall@10	0.902904	0.902904	0.903254	**0.9938**
latency (μs)	602.566	857.531	**445.888**	791.127

上表是四个版本的最终性能对比。ivfpq 的 pthread 的并行尝试最终反向优化，OpenMP 并行尝试最终略有提升。ivfpq 算法相较于 sq 和 pq 算法，已经大大降低了计算量。在此基础上做并行化不会有太好的效果。而 sq 则因为向量存储地址连续、缓存命中率高、任务划分简单等原因，并行效果非常显著。

关于前面一直没有提到的 SIMD 编程的内容，因为此次实验是直接在上次实验的基础上做的，所以所有实验都是已经经过 SIMD 加速计算后的。SIMD 加速主要在计算查询向量与粗簇的残差，以及计算欧式距离的部分体现。欧氏距离的计算被 get_query_dis() 包装起来了。

这次实验因为不清楚到底是我的代码写的有问题，还是 ivfpq 真的不适合并行化，反向优化实在是让人难蚌。但考虑到 SQ 的显著并行效果，应该不是我的并行思路有问题吧（?）

Git 项目链接：https://github.com/SheepSpaceFly/NKHW/tree/main/并行程序设计/pthread 和 OpenMP 编程实验