南開大學
Nankai University

计算机学院

并行程序设计作业报告

基于 Pthread 和 OpenMP 的 ANN 加速实验报告

姓名：禹相祐

学号：2312900

专业：计算机科学与技术

2025 年 5 月 23 日

目录

1 相关技术简介

1.1 Pthread 实验部分

1.1.1 近似最近邻搜索（Approximate Nearest Neighbor Search,ANNs）

近似最近邻搜索致力于在大规模向量集合中快速找到与查询向量最接近的若干结果，是图像检索、推荐系统等任务中的基础模块[1]。与精确搜索相比，ANNs 方法在保持较高精度的同时大幅降低了计算复杂度。主流方法一般基于量化、图结构或哈希等技术实现 ANNs。

1.1.2 乘积量化（Product Quantization，PQ）

乘积量化是一种高效的向量压缩与近似最近邻搜索（ANN）技术，广泛应用于大规模向量检索系统中。其核心思想是将高维向量划分为若干子空间，并对每个子空间独立进行量化，最终使用多个子量化码本组合来逼近原始向量。这种方法在保持较高检索精度的同时大大减少了存储开销和计算开销。

PQ 方法最早由 Jégou 等人提出[2]，并已成为众多向量索引方法的基础组件。其在图像检索、语义搜索和推荐系统中均有广泛应用。

1.1.3 倒排文件索引（Inverted File Index, IVF）

倒排文件索引（IVF）是一种将数据库向量划分到多个“桶”中进行组织的结构，常用于加速大规模向量搜索。在 IVF 中，通过聚类（如 k-means）将数据库划分为多个簇，然后仅在与查询向量最接近的几个簇中进行搜索，从而显著减少比较次数[3]。IVF 常与 PQ 结合使用（即 IVF+PQ）以进一步提升效率。

1.1.4 Pthreads（POSIX 线程）

Pthreads 是 POSIX 标准下的一套线程编程接口，提供了线程的创建、同步与管理功能，是 C/C++ 中多线程并发编程的基础工具之一。通过 Pthreads，开发者可以在多核处理器上实现并行任务，以提高程序性能[4]。

1.2 OpenMP 实验部分

1.2.1 OpenMP（开放多处理编程接口）

OpenMP 是一种支持共享内存多处理器编程的 API，适用于 C/C++ 和 Fortran。它使用编译指令、运行时库和环境变量相结合的方式，使得在现有串行代码中快速引入并行性变得简单[5]。OpenMP 在科学计算、图像处理和深度学习预处理任务中都有广泛应用。

2 框架中的原 flat_search 部分

此处部分同上篇 SIMD 实验报告的第二部分。为了方便此次实验的加速效果对比，此处保留了上次实验所测得的实验数据。

进行十次测试，平均运行时间为 15641.6us，召回率为 0.9999，具体数据如下表:

表 1: flat_search 方式十次测试运行时间以及召回率

测试序号	运行时间/us	召回率
1	16456.1	0.9999
2	15349.6	0.9999
3	17184.9	0.9999
4	15207.4	0.9999
5	15734.8	0.9999
6	15250.1	0.9999
7	15412.0	0.9999
8	15451.2	0.9999
9	15233.3	0.9999
10	15136.7	0.9999

3 IVF+PQ 的平凡加速实现方法

此次实验所需要的是利用 Pthread 和 OpenMP 的方式分别对 ANN 进行加速，且基于 IVF+PQ 的平凡加速方法。所以此处我们先对如何使用 IVF+PQ 进行加速进行讲解，后续讲解再加上 Pthreead 和 OpenMP 的单独优化部分即可。

3.1 PQ 和 IVF 的简要介绍

3.1.1 PQ 部分

Product Quantization（PQ，乘积量化）是一种常用的向量压缩与近似最近邻搜索技术。它的核心思想是将原始的高维向量空间划分为多个子空间，然后在每个子空间内分别进行以 K-Means 为代表性方法的聚类，将每个子空间的子向量表示为聚类中心的索引。这样一个原始向量就可以表示为由多个子空间中的聚类中心索引共同组成的编码，用这种方式就能实现对存储空间的极大减少。

在搜索阶段，PQ 使用一种称为 Asymmetric Distance Computation（ADC）的策略来计算查询向量与压缩向量之间的近似距离。具体而言，对于每个查询向量，首先将其拆分为多个子向量，并与各子空间的码本中的中心进行距离查找和预计算，从而快速获得与候选向量的近似距离。

PQ 的优点是压缩率高、距离计算快，并且对硬件要求不高，非常适合在类似 ANN 这样有大规模的向量检索需求的场景中应用。

3.1.2 IVF 部分

IVF（Inverted File Index，倒排索引）是一种用于 ANN 的空间划分方法。它通过对数据库中的所有向量进行聚类（例如使用 K-Means），然后生成一组中心点（称为“簇”或“cell”）。每个数据库向量都会被分配到与其距离最近的中心所代表的簇中。这样，整个数据库的向量就被划分为若干个倒排列表，方便后续查询。

在查询阶段，查询向量同样被分配到最近的几个中心（通过调整 nprobe 来控制），然后只在这些对应的倒排列表中进行近似搜索，从而大大减少了需要遍历的候选数量。

IVF 可以与 PQ 结合使用，在每个倒排列表中存储的不是原始向量而是其 PQ 编码，从而实现高效的存储和快速的 ANN 搜索。

3.1.3 PQ 与 IVF 的结合

PQ 和 IVF 本质上是两种互补的加速技术，能够协同使用以同时解决大规模近似向量搜索中的两大核心问题：搜索效率与存储开销。

具体而言，IVF 负责将整个高维的向量空间划分成多个簇（cell），从而在查询时只访问一小部分候选向量而不是全部的候选向量；而 PQ 则对每个 cell 中的向量进行压缩编码，使得即使当 nprobe 较大，即访问多个倒排列表时，也能合理地控制内存负担和距离计算的开销。

而将 PQ 与 IVF 结合的方式通常被称为 IVF+PQ，其具体的流程如下：

1. 训练阶段

- 使用 K-Means 聚类获取 IVF 的簇中心（*coarse quantizer*）
- 对残差向量（定义：$\boldsymbol{r} = \boldsymbol{x} - \boldsymbol{c}_i$，其中 $\boldsymbol{x}$ 为原始向量，$\boldsymbol{c}_i$ 为其所属簇中心）应用乘积量化（PQ）进行编码

2. 编码阶段

- 对数据库向量执行两阶段编码：
  (a) 将其分配到最近的 IVF 簇
  (b) 计算残差向量并通过 PQ 生成紧凑编码（形如 $code = PQ(\boldsymbol{r})$）

3. 查询阶段

- 查询向量 $\boldsymbol{q}$ 的最近邻搜索流程：
  (i) 定位到 $\boldsymbol{q}$ 最近的 `nprobe` 个 IVF 簇
  (ii) 仅在这些簇内使用非对称距离计算（**ADC**）方法：

$$d_{ADC}(\boldsymbol{q}, \boldsymbol{x}) = d(\boldsymbol{q}, \boldsymbol{c}_i + \mathrm{dec}(PQ(\boldsymbol{r})))$$

其中 dec(·) 表示 PQ 解码操作

具体的 IVF+PQ 流程可以用下图来展示：

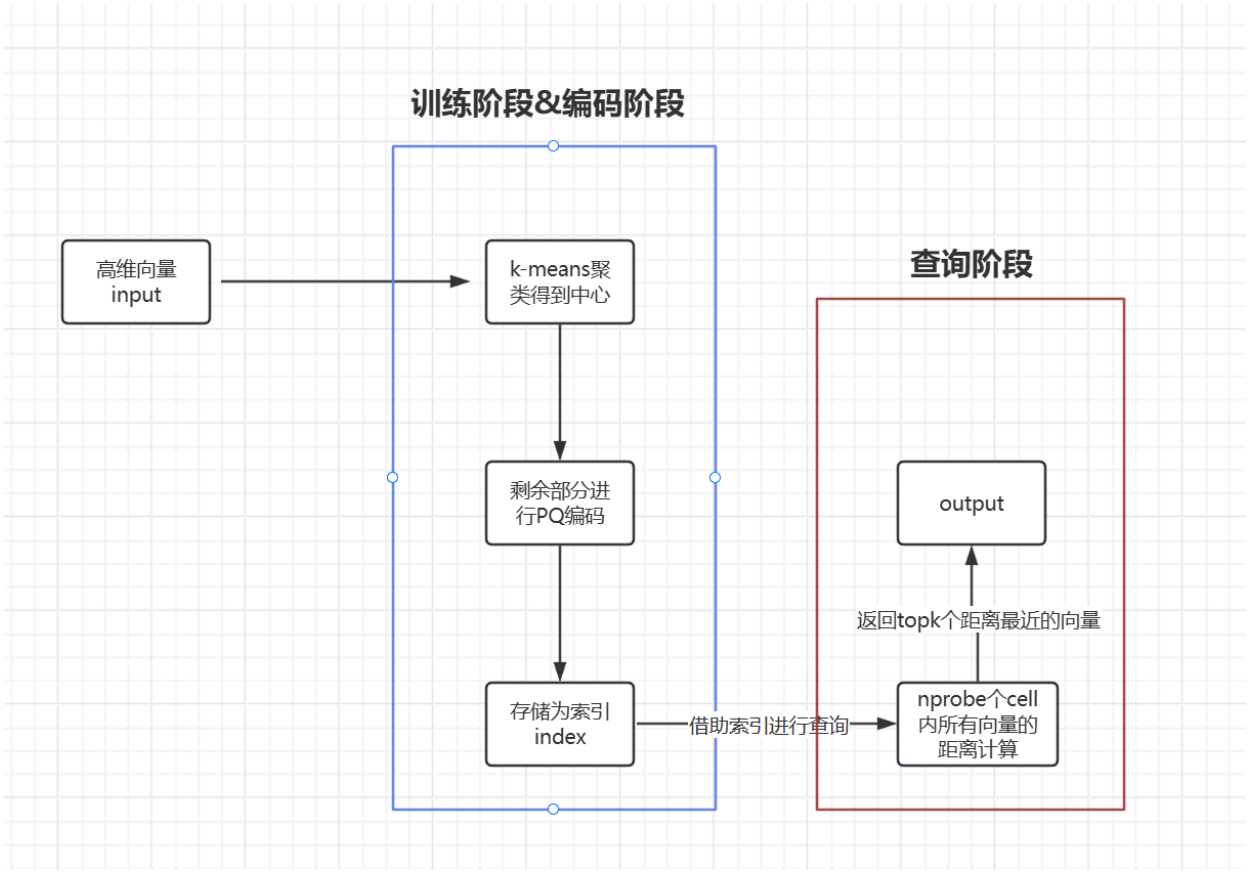

图 3.1: IVF+PQ 实现

3.2 代码讲解

3.2.1 训练阶段

训练阶段的目标是构建：

- coarse quantizer（粗量化器，用于倒排列表划分）
- PQ codebooks（每个子空间的编码字典）

欧式距离的计算函数:

```
// 距离计算函数 l2_distance_sq: Distance = (a_i-b_i)^2的求和
float l2_distance_sq(const std::vector<float>& a, const std::vector<float>& b) {
    float dist = 0;
    for (size_t i = 0; i < a.size(); ++i) {
        float diff = a[i] - b[i];
        dist += diff * diff;
    }
    return dist;
}
```

Kmeans 聚类实现：

```cpp
// KMeans 聚类：用于训练 coarse centroid 和 PQ 子码本
// k表示聚类k个cell，dim表示数据的维度，n_iter表示进行n_iter轮k-means
std::vector<std::vector<float>> kmeans(const std::vector<std::vector<float>>& data,
    int k, int dim, int n_iter = 20) {
    int n = data.size();
    // centroids为一个k*dim的二维的vector
    std::vector<std::vector<float>> centroids(k, std::vector<float>(dim));
    // 使用随机种子42初始化随机数的生成器，以确保每次运行的结果都是一样的
    std::mt19937 rng(42);
    std::uniform_int_distribution<int> uni(0, n - 1);

    // k-means最开始：为每个cell随机初始化中心点
    for (int i = 0; i < k; ++i) {
        centroids[i] = data[uni(rng)];
    }
    // assignments[i]表示第i个sample属于哪个cell
    std::vector<int> assignments(n);
    for (int iter = 0; iter < n_iter; ++iter) {
        // 1. 分配每个点到最近中心
        for (int i = 0; i < n; ++i) {
            float best_dist = std::numeric_limits<float>::max();
            int best_c = 0;
            for (int j = 0; j < k; ++j) {
                float dist = l2_distance_sq(data[i], centroids[j]);
                if (dist < best_dist) {
                    best_dist = dist;
                    best_c = j;
                }
            }
            // 把当前的vector认为其属于最近的那个cell
            assignments[i] = best_c;
        }
        // 2. 更新中心
        std::vector<std::vector<float>> new_centroids(k, std::vector<float>(dim, 0));
        // 保存每个cell具体有多少个vector
        std::vector<int> counts(k, 0);
        for (int i = 0; i < n; ++i) {
            int c = assignments[i];
            for (int d = 0; d < dim; ++d)
                new_centroids[c][d] += data[i][d];
            counts[c]++;
        }
        // 将每个cell内所有的vector相加并除以总数量，更新新的centroid
        for (int j = 0; j < k; ++j) {
            if (counts[j] > 0) {
                for (int d = 0; d < dim; ++d)
                    new_centroids[j][d] /= counts[j];
```

```
            }
        }
        // 将中心更新，进行下一轮k-means聚类
        centroids = new_centroids;
    }
    return centroids;
}
```

训练 PQ 编码器：

- 将每个原始向量分为 M 个子向量（维度为 $\frac{D}{M}$）
- 每个子空间分别做 K 均值聚类，生成码本

```
// 训练 PQ 子码本，每个子空间做一次独立的 KMeans
// base:原始向量 N*dim，M:将每个向量分为M块，Ks:每个子空间聚类的数量
std::vector<std::vector<std::vector<float>>> train_product_quantizer(const
    std::vector<std::vector<float>>& base, int M, int Ks, int dim) {
    // sub_dim: eg.输入dim=96且M=16 那么划分完后子向量的维数就是96/16=6
    int sub_dim = dim / M;
    // codebooks存储结果 大小为M*Ks*sub_dim
    std::vector<std::vector<std::vector<float>>> codebooks(M);
    // 遍历每个子空间进行k-means聚类
    for (int m = 0; m < M; ++m) {
        std::vector<std::vector<float>> sub_vectors;
        for (const auto& vec : base) {
            std::vector<float> sub(vec.begin() + m * sub_dim, vec.begin() + (m + 1) *
                sub_dim);
            sub_vectors.push_back(sub);
        }
        // 每个codebook都进行k-means聚类
        codebooks[m] = kmeans(sub_vectors, Ks, sub_dim);
    }
    return codebooks;
}
```

构建具体的索引：

```
// 每个 coarse centroid 的倒排表，保存属于该中心的 PQ 编码数据和对应向量编号
struct InvertedList {
    std::vector<std::vector<uint8_t>> pq_codes;  // 每个向量的 PQ 编码
    std::vector<int> ids;  // 原始向量在 base 中的编号
};

// IVF + PQ 索引结构体
struct IVFPQIndex {
    std::vector<std::vector<float>> coarse_centroids;       // coarse quantizer
        聚类中心 (nlist x dim)
```

```
    std::vector<std::vector<std::vector<float>>> pq_codebooks; // PQ 子空间的 codebook (M x Ks x sub_dim)
    std::unordered_map<int, InvertedList> inverted_lists;  // 倒排表: coarse centroid id -> 向量PQ码和id
    int M;       // 子空间个数
    int Ks;      // 每个子空间的聚类数 (一般为256)
    int dim;     // 原始向量维度
};

// 构建 IVF+PQ 索引: 包含 coarse quantizer 和 residual PQ 编码
IVFPQIndex build_ivf_pq_index(const std::vector<std::vector<float>>& base, int nlist, int M, int Ks, int dim) {
    IVFPQIndex index;
    index.dim = dim;
    index.M = M;
    index.Ks = Ks;
    // 首先将所有vector分成nlist个类
    index.coarse_centroids = kmeans(base, nlist, dim);
    // 利用PQ训练出子码本
    index.pq_codebooks = train_product_quantizer(base, M, Ks, dim);
    // 遍历所有vector构建IVF
    for (int i = 0; i < base.size(); ++i) {
        const auto& vec = base[i];

        // 1. 找到与当前向量最接近的 coarse centroid
        int best_cid = 0;
        float best_dist = std::numeric_limits<float>::max();
        for (int c = 0; c < nlist; ++c) {
            float dist = l2_distance_sq(vec, index.coarse_centroids[c]);
            if (dist < best_dist) {
                best_dist = dist;
                best_cid = c;
            }
        }

        // 2. 计算 residual (原始向量减去 coarse centroid)
        std::vector<float> residual(dim);
        for (int d = 0; d < dim; ++d) {
            residual[d] = vec[d] - index.coarse_centroids[best_cid][d];
        }

        // 3. 对 residual 向量编码为 PQ code
        std::vector<uint8_t> pq_code = encode_pq(residual, index.pq_codebooks, M, dim);

        // 4. 存入对应 coarse centroid 的倒排表
        index.inverted_lists[best_cid].pq_codes.push_back(pq_code);
        index.inverted_lists[best_cid].ids.push_back(i);
    }
```

```
    return index;
}
```

3.2.2 编码阶段

PQ 编码函数：将 residual 子向量与 PQ 码本比较，选取最近的索引值作为 PQ 编码。

```
// 对 residual 向量进行 PQ 编码
// vec:待编码的residual vector
std::vector<uint8_t> encode_pq(const std::vector<float>& vec, const
    std::vector<std::vector<std::vector<float>>>& pq_codebooks, int M, int dim) {
    int sub_dim = dim / M;
    // code:存储如何编码的向量
    std::vector<uint8_t> code(M);
    // 遍历所有的子空间
    for (int m = 0; m < M; ++m) {
        // sub用来存储将原先dim维度的vec拆分为M个dim/M维度的vector
        // eg.原来vec=[1,2,3,4]--->变为[[1,2],[3,4]]
        std::vector<float> sub(vec.begin() + m * sub_dim, vec.begin() + (m + 1) *
            sub_dim);
        // 找到dis最小的cell的中心 然后对其进行编码
        // eg. vec=[1,2,3,4]--->sub=[[1,2],[3,4]]
        // 假设在子空间内最近的中心编号分别为3,17
        // 那实际的编码结果code=[3,17] 至此编码完成
        float best_dist = std::numeric_limits<float>::max();
        int best_k = 0;
        for (int k = 0; k < pq_codebooks[m].size(); ++k) {
            float dist = l2_distance_sq(sub, pq_codebooks[m][k]);
            if (dist < best_dist) {
                best_dist = dist;
                best_k = k;
            }
        }
        code[m] = static_cast<uint8_t>(best_k);
    }
    return code;
}
```

3.2.3 索引保存和读取

索引的保存：

```
// 保存索引到文件
void save_index(const IVFPQIndex& index, const std::string& filename) {
    std::ofstream ofs(filename, std::ios::binary);
    if (!ofs) {
        throw std::runtime_error("无法打开文件写入");
```

```
    }

    // 写入基础参数
    ofs.write(reinterpret_cast<const char*>(&index.dim), sizeof(index.dim));
    ofs.write(reinterpret_cast<const char*>(&index.M), sizeof(index.M));
    ofs.write(reinterpret_cast<const char*>(&index.Ks), sizeof(index.Ks));

    // 写入 coarse_centroids
    size_t nlist = index.coarse_centroids.size();
    ofs.write(reinterpret_cast<const char*>(&nlist), sizeof(nlist));
    for (const auto& centroid : index.coarse_centroids) {
        ofs.write(reinterpret_cast<const char*>(centroid.data()), centroid.size() *
            sizeof(float));
    }

    // 写入 pq_codebooks
    size_t M = index.pq_codebooks.size();
    ofs.write(reinterpret_cast<const char*>(&M), sizeof(M));
    for (const auto& subspace_codebook : index.pq_codebooks) {
        size_t Ks = subspace_codebook.size();
        ofs.write(reinterpret_cast<const char*>(&Ks), sizeof(Ks));
        for (const auto& codeword : subspace_codebook) {
            ofs.write(reinterpret_cast<const char*>(codeword.data()), codeword.size()
                * sizeof(float));
        }
    }

    // 写入倒排表数量
    size_t inverted_list_count = index.inverted_lists.size();
    ofs.write(reinterpret_cast<const char*>(&inverted_list_count),
        sizeof(inverted_list_count));

    for (const auto& entry : index.inverted_lists) {
        int cid =entry.first;
        const auto& invlist = entry.second;
        // 写入倒排表对应的coarse centroid id
        ofs.write(reinterpret_cast<const char*>(&cid), sizeof(cid));

        // 写入 pq_codes 数量
        size_t pq_codes_num = invlist.pq_codes.size();
        ofs.write(reinterpret_cast<const char*>(&pq_codes_num), sizeof(pq_codes_num));

        for (const auto& code : invlist.pq_codes) {
            ofs.write(reinterpret_cast<const char*>(code.data()), code.size() *
                sizeof(uint8_t));
        }

        // 写入 ids 数量（应与 pq_codes_num 一致）
        size_t ids_num = invlist.ids.size();
```

```
        ofs.write(reinterpret_cast<const char*>(&ids_num), sizeof(ids_num));
        ofs.write(reinterpret_cast<const char*>(invlist.ids.data()), ids_num *
            sizeof(int));
    }

    ofs.close();
}
```

索引的读取:

```
// 从文件加载索引
IVFPQIndex load_index(const std::string& filename) {
    std::ifstream ifs(filename, std::ios::binary);
    if (!ifs) {
        throw std::runtime_error("无法打开索引文件");
    }

    IVFPQIndex index;

    // 读基础参数
    ifs.read(reinterpret_cast<char*>(&index.dim), sizeof(index.dim));
    ifs.read(reinterpret_cast<char*>(&index.M), sizeof(index.M));
    ifs.read(reinterpret_cast<char*>(&index.Ks), sizeof(index.Ks));

    // 读 coarse_centroids
    size_t nlist = 0;
    ifs.read(reinterpret_cast<char*>(&nlist), sizeof(nlist));
    index.coarse_centroids.resize(nlist, std::vector<float>(index.dim));
    for (auto& centroid : index.coarse_centroids) {
        ifs.read(reinterpret_cast<char*>(centroid.data()), centroid.size() *
            sizeof(float));
    }

    // 读 pq_codebooks
    size_t M = 0;
    ifs.read(reinterpret_cast<char*>(&M), sizeof(M));
    index.pq_codebooks.resize(M);
    int sub_dim = index.dim / index.M;
    for (auto& subspace_codebook : index.pq_codebooks) {
        size_t Ks = 0;
        ifs.read(reinterpret_cast<char*>(&Ks), sizeof(Ks));
        subspace_codebook.resize(Ks, std::vector<float>(sub_dim));
        for (auto& codeword : subspace_codebook) {
            ifs.read(reinterpret_cast<char*>(codeword.data()), codeword.size() *
                sizeof(float));
        }
    }

    // 读倒排表数量
```

```
    size_t inverted_list_count = 0;
    ifs.read(reinterpret_cast<char*>(&inverted_list_count),
        sizeof(inverted_list_count));

    for (size_t i = 0; i < inverted_list_count; ++i) {
        int cid = 0;
        ifs.read(reinterpret_cast<char*>(&cid), sizeof(cid));
        InvertedList invlist;

        size_t pq_codes_num = 0;
        ifs.read(reinterpret_cast<char*>(&pq_codes_num), sizeof(pq_codes_num));
        invlist.pq_codes.resize(pq_codes_num, std::vector<uint8_t>(index.M));
        for (auto& code : invlist.pq_codes) {
            ifs.read(reinterpret_cast<char*>(code.data()), code.size() *
                sizeof(uint8_t));
        }

        size_t ids_num = 0;
        ifs.read(reinterpret_cast<char*>(&ids_num), sizeof(ids_num));
        invlist.ids.resize(ids_num);
        ifs.read(reinterpret_cast<char*>(invlist.ids.data()), ids_num * sizeof(int));

        index.inverted_lists[cid] = std::move(invlist);
    }

    ifs.close();
    return index;
}
```

3.2.4 查询阶段

最终的查询函数:

```
// 此处int对应距离distance,float对应向量的id
struct MinHeapComparator {
    bool operator()(const std::pair<int, float>& a, const std::pair<int, float>& b)
        const {
        return a.first > b.first; // 最小的距离优先
    }
};

// IVF + PQ 搜索: 返回 min_heap
// index表示已经构建好了的ivf+pq的索引,query表示查询vector,
// topk表示返回最相似的k个, nprobe表示访问最近的nprobe个中心
std::priority_queue<std::pair<int, float>, std::vector<std::pair<int, float>>,
    MinHeapComparator>
ivf_pq_search(const IVFPQIndex& index, const std::vector<float>& query, int topk, int
    nprobe) {
```

```
    int dim = index.dim;
    int M = index.M;
    int Ks = index.Ks;
    int sub_dim = dim / M;

    // 1. 找最近的 nprobe 个 coarse center, int 对应 id , float 对应 distance
    std::vector<std::pair<int, float>> coarse_dists;
    for (int i = 0; i < index.coarse_centroids.size(); ++i) {
        float dist = l2_distance_sq(query, index.coarse_centroids[i]);
        coarse_dists.emplace_back(i, dist);
    }
    std::partial_sort(coarse_dists.begin(), coarse_dists.begin() + nprobe,
        coarse_dists.end(),
        [](const std::pair<int, float>& a, const std::pair<int, float>& b) {
            return a.second < b.second;
        });

    // 2. 计算 residual subqueries
    std::vector<std::vector<float>> residual_subqueries(M,
        std::vector<float>(sub_dim));

    // 3. 进行 PQ 近似计算并加入堆
    std::priority_queue<std::pair<int, float>, std::vector<std::pair<int, float>>,
        MinHeapComparator> result_heap;
    // 遍历每个coarse centroid, 获取id、中心vector以及ivf
    for (int i = 0; i < nprobe; ++i) {
        int cid = coarse_dists[i].first;
        float coarse_dist = coarse_dists[i].second;  // 新增
        const std::vector<float>& centroid = index.coarse_centroids[cid];
        const InvertedList& invlist = index.inverted_lists.at(cid);

        // 计算查询向量对于目前coarse centroid的residual向量, 方便计算距离dis
        for (int m = 0; m < M; ++m) {
            for (int d = 0; d < sub_dim; ++d) {
                residual_subqueries[m][d] = query[m * sub_dim + d] - centroid[m *
                    sub_dim + d];
            }
        }
        // 计算最终的距离dis, 根据距离dis维护min_heap
        for (int j = 0; j < invlist.pq_codes.size(); ++j) {
            float pq_dist = 0.0f;
            for (int m = 0; m < M; ++m) {
                uint8_t code = invlist.pq_codes[j][m];
                const std::vector<float>& codeword = index.pq_codebooks[m][code];
                pq_dist += l2_distance_sq(residual_subqueries[m], codeword);
            }
            // 将pq_dis和id加入最小堆, 维护topk的最小堆
            result_heap.emplace(static_cast<int>(pq_dist),invlist.ids[j]);
        }
```

```
    }
    return result_heap;
}
```

3.3 运行结果

IVF+PQ 方式召回率与 M、nprobe 的关系图如下：

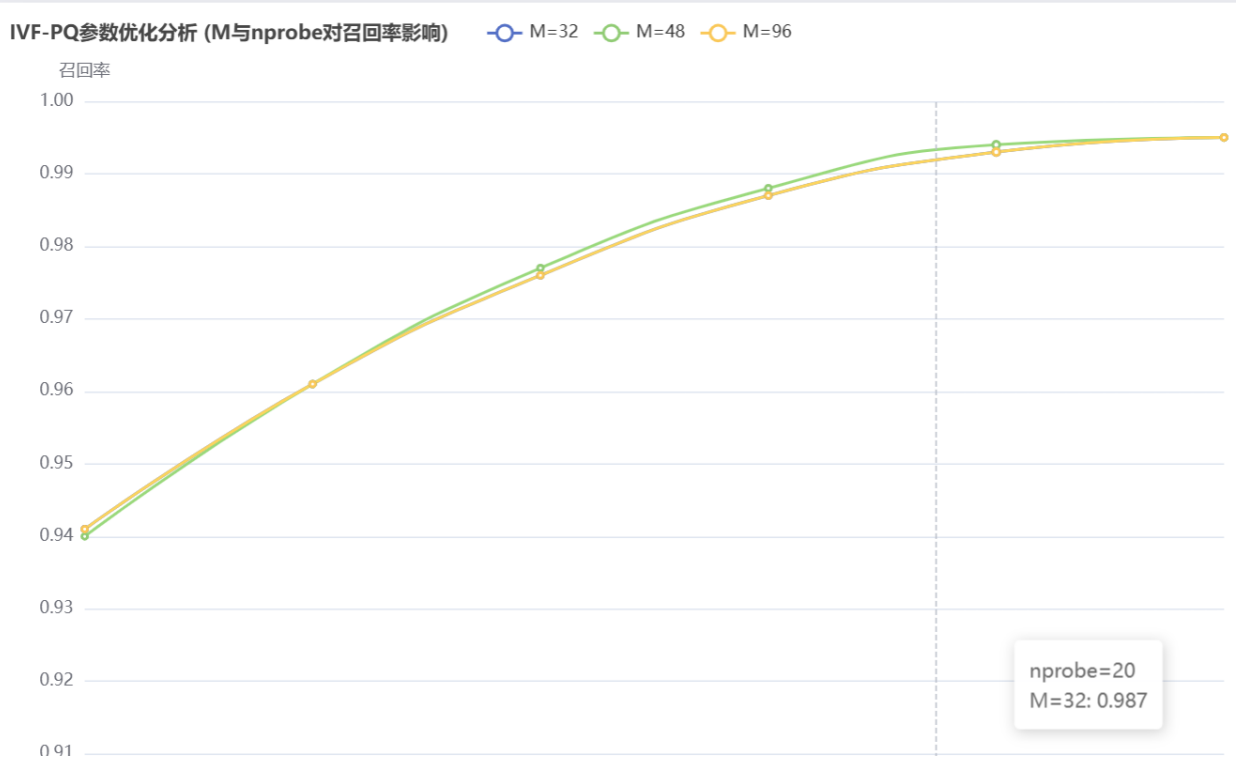

图 3.2: IVF+PQ 方式 Recall 与 M、nprobe 关系图

IVF+PQ 方式的 Latency 与 M、nprobe 的关系图如下：

图 3.3: IVF+PQ 方式 Latency 与 M、nprobe 关系图

最简单的 IVF+PQ 的方法的具体实验数据如下：

表 2: Recall、Latency 与 M、nprobe 的部分具体数据

M	nprobe	Recall	Latency(us)
32	10	0.961	3096
32	20	0.987	5935
32	30	0.995	10333
48	10	0.961	6834
48	20	0.988	13836
48	30	0.995	19594
96	10	0.961	13592
96	20	0.987	26510
96	30	0.95	35754

4 Pthread 实验部分

4.1 加速方法

在基于 IVF+PQ 结构的检索中，查询阶段的主要计算量集中于多个被探测（nprobe）的 coarse centroid 所对应的倒排表的近似距离计算。该部分属于高度可并行的计算，因此可以通过多线程并发

处理加速。POSIX 标准线程库 pthread 提供了低层次的线程接口，支持在多核 CPU 上实现任务级别的并行。

其基本加速原理如下：

- 将 n_{probe} 个 coarse centroid 分成若干小段，每个线程处理一段倒排表的 PQ 距离计算任务；
- 各线程并行维护自己的 top-k 局部最小堆；
- 主线程回收所有局部堆结果，合并得到最终的 top-k 最小距离向量结果。

4.2 代码实现

此次 Pthread 主要实现的就是在查询阶段的 Pthread 并行化，所以改的内容也主要是这一部分，具体改动从以下代码可见：

```
// 首先是为其加上对应的头文件
#include <pthread.h>

// 然后是定义一个线程数的常量以及构造对应的结构体
static const int NUM_THREADS = 8;

// 线程参数结构体
struct ThreadArg {
    const IVFPQIndex* index;
    const std::vector<float>* query;
    const std::vector<std::pair<int, float>>* coarse_dists;
    int topk;
    int start;
    int end;
    std::priority_queue<std::pair<float, int>, std::vector<std::pair<float, int>>,
        MinHeapComparator>* local_heap;
    //pthread_mutex_t* mutex;
};
```

search_worker 函数的作用是：在给定的 coarse centroid 区间内计算所有倒排表中的近似距离，维护本地 topk 最小堆，并返回结果。这样多个线程可以并行地处理 coarse centroid，从而显著加快 IVF+PQ 检索速度。

```
// 然后是多出来的一个函数
// 线程函数：处理 invlists[start_idx..end_idx)，本地维护 topk
    堆，最后一次性合并到全局堆
// void*类型是一个通用的指针类型，是pthread库的定义
static void* search_worker(void* arg) {
    // ThreadArg* 为自己上方定义的结构体 用于表示一个线程的必要信息
    ThreadArg* t = static_cast<ThreadArg*>(arg);
    const IVFPQIndex& idx = *t->index;
    int M = idx.M;
    int sub_dim = idx.dim / M;

```

```
    std::priority_queue<std::pair<float, int>, std::vector<std::pair<float, int>>,
        MinHeapComparator> local_heap;
    std::vector<std::vector<float>> residuals(M, std::vector<float>(sub_dim));

    for (int i = t->start; i < t->end; ++i) {
        int cid = (*t->coarse_dists)[i].first;
        const auto& centroid = idx.coarse_centroids[cid];
        for (int m = 0; m < M; ++m) {
            for (int d = 0; d < sub_dim; ++d) {
                residuals[m][d] = (*t->query)[m * sub_dim + d] - centroid[m * sub_dim
                    + d];
            }
        }
        const auto& inv = idx.inverted_lists.at(cid);
        for (size_t j = 0; j < inv.pq_codes.size(); ++j) {
            float dist = 0.0f;
            for (int m = 0; m < M; ++m) {
                uint8_t code = inv.pq_codes[j][m];
                dist += l2_distance_sq(residuals[m], idx.pq_codebooks[m][code]);
            }
            local_heap.emplace(dist,inv.ids[j]);
            if (static_cast<int>(local_heap.size()) > t->topk) {
                local_heap.pop();
            }
        }
    }
    *t->local_heap = std::move(local_heap);
    return nullptr;
}
```

然后是搜索函数的变化：先将 nprobe 个中心分给八个线程，然后每个线程都维护一个自己的 *local_min_heap*，然后再是合并所有的 *local_min_heap* 组成一个 *result_heap* 作为最终的结果。

```
// IVF + PQ 的Pthread搜索：返回最小堆
std::priority_queue<std::pair<float,int>, std::vector<std::pair<float,int>>,
    MinHeapComparator>
pthread_ivf_pq_search(const IVFPQIndex& index, const std::vector<float>& query, int
    topk, int nprobe) {
    std::vector<std::pair<int, float>> coarse_dists;
    std::priority_queue<std::pair<float, int>, std::vector<std::pair<float, int>>,
        MinHeapComparator> local_heaps[NUM_THREADS];

    for (int i = 0; i < static_cast<int>(index.coarse_centroids.size()); ++i) {
        float d = l2_distance_sq(query, index.coarse_centroids[i]);
        coarse_dists.emplace_back(i, d);
    }
    std::partial_sort(
        coarse_dists.begin(),
        coarse_dists.begin() + nprobe,
```

```
        coarse_dists.end(),
        [](const std::pair<int, float>& a, const std::pair<int, float>& b) {
            return a.second < b.second;
        }
    );

    pthread_mutex_t mutex;
    pthread_mutex_init(&mutex, nullptr);

    std::priority_queue<std::pair<float, int>, std::vector<std::pair<float, int>>,
        MinHeapComparator> result_heap;

    pthread_t threads[NUM_THREADS];
    ThreadArg args[NUM_THREADS];
    int per = (nprobe + NUM_THREADS - 1) / NUM_THREADS;
    for (int t = 0; t < NUM_THREADS; ++t) {
        int s = t * per;
        int e = std::min(s + per, nprobe);
        args[t] = ThreadArg{&index, &query, &coarse_dists, topk, s, e,
            &local_heaps[t]};

        pthread_create(&threads[t], nullptr, search_worker, &args[t]);
    }
    for (int t = 0; t < NUM_THREADS; ++t) {
        pthread_join(threads[t], nullptr);
    }
    for (int t = 0; t < NUM_THREADS; ++t) {
    while (!local_heaps[t].empty()) {
        result_heap.push(local_heaps[t].top());
        if ((int)result_heap.size() > topk) result_heap.pop();
        local_heaps[t].pop();
        }
    }
    pthread_mutex_destroy(&mutex);

    return result_heap;
}
```

4.3 运行结果

Pthread 方式召回率与 M、nprobe 的关系图如下：

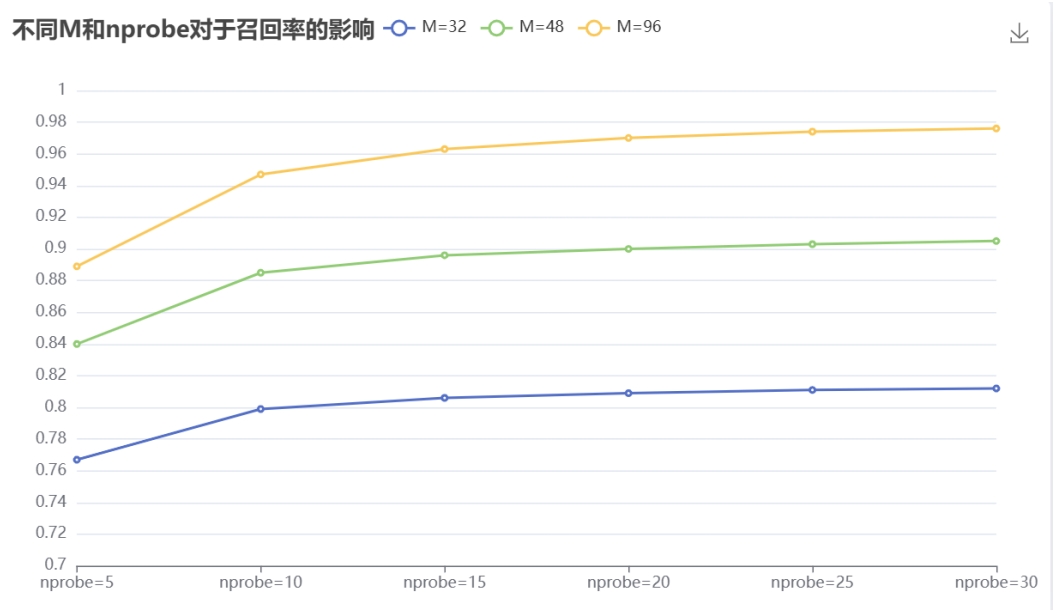

图 4.4: Pthread 方式 Recall 与 M、nprobe 关系图

Pthread 方式的 Latency 与 M、nprobe 的关系图如下：

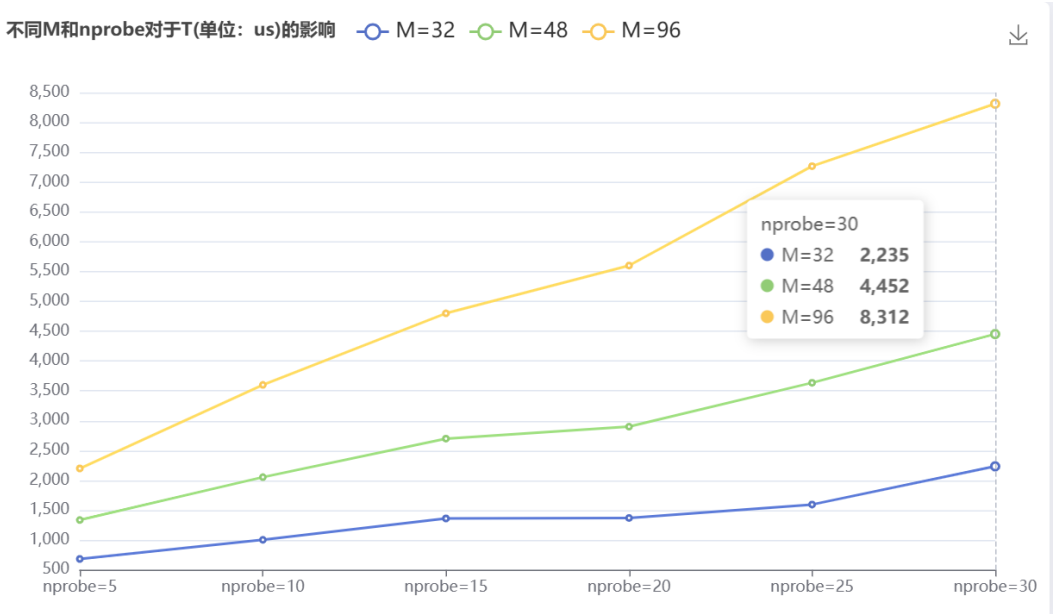

图 4.5: Pthread 方式 Latency 与 M、nprobe 关系图

此处给出折线图中的部分具体数据:

表 3: Pthread 中 Recall、Latency 与 M、nprobe 的部分具体数据

M	nprobe	Recall	Latency(us)
32	10	0.799	1003
32	20	0.809	1370
32	30	0.812	2235
48	10	0.885	2054
48	20	0.896	2700
48	30	0.905	4452
96	10	0.947	3600
96	20	0.970	5600
96	30	0.976	8312

5 OpenMP 实验部分

5.1 加速方法

在向量近似搜索中（如 IVF+PQ），最耗时的部分是对多个 coarse centroid 所对应倒排表中所有 PQ 向量与查询向量的近似距离计算。由于这些计算之间不存在依赖关系，可视为“完全独立任务”，非常适合使用 OpenMP 并行处理。

OpenMP 的并行加速过程如下:

1. 将查询向量与 n_{probe} 个 coarse centroid 的倒排表划分为多个循环迭代任务;
2. 使用 `#pragma omp parallel for` 对循环进行并行化;
3. 每个线程处理不同的 coarse centroid，计算 PQ 近似距离;
4. 各线程局部存储候选结果，再通过临界区 `critical` 合并回全局结果;
5. 最终对合并结果排序，选出 top-k 近似向量返回。

5.2 代码实现

OpenMP 所改动的只有加一个头文件以及 search 函数部分:

```
#include <omp.h>

std::priority_queue<std::pair<float, int>, std::vector<std::pair<float, int>>,
    MinHeapComparator>
openmp_ivf_pq_search(const IVFPQIndex& index, const std::vector<float>& query, int
    topk, int nprobe) {
    int dim = index.dim;
    int M = index.M;
    int Ks = index.Ks;
    int sub_dim = dim / M;

```

```
    // 1. 找最近的 nprobe 个 coarse center
    std::vector<std::pair<int, float>> coarse_dists;
    for (int i = 0; i < index.coarse_centroids.size(); ++i) {
        float dist = l2_distance_sq(query, index.coarse_centroids[i]);
        coarse_dists.emplace_back(i, dist);
    }
    std::partial_sort(
        coarse_dists.begin(),
        coarse_dists.begin() + nprobe,
        coarse_dists.end(),
        [](const std::pair<int, float>& a, const std::pair<int, float>& b) {
            return a.second < b.second;
        }
    );

    // 2. 并行处理每个候选中心
    std::vector<std::pair<float, int>> candidates;

    #pragma omp parallel
    {
        std::vector<std::pair<float, int>> local_candidates;

        #pragma omp for nowait
        for (int i = 0; i < nprobe; ++i) {
            int cid = coarse_dists[i].first;
            const auto& centroid = index.coarse_centroids[cid];
            const auto& invlist = index.inverted_lists.at(cid);

            // 预处理 residual 向量
            std::vector<std::vector<float>> residual_subqueries(M,
                std::vector<float>(sub_dim));
            for (int m = 0; m < M; ++m) {
                for (int d = 0; d < sub_dim; ++d) {
                    residual_subqueries[m][d] = query[m * sub_dim + d] - centroid[m *
                        sub_dim + d];
                }
            }

            // 计算 PQ 距离
            for (size_t j = 0; j < invlist.pq_codes.size(); ++j) {
                float pq_dist = 0.0f;
                for (int m = 0; m < M; ++m) {
                    uint8_t code = invlist.pq_codes[j][m];
                    const auto& codeword = index.pq_codebooks[m][code];
                    pq_dist += l2_distance_sq(residual_subqueries[m], codeword);
                }
                local_candidates.emplace_back(pq_dist, invlist.ids[j]); // 参数顺序为
                    (距离, id)
            }
```

```
56          }
57
58          // 合并到全局 candidates
59          #pragma omp critical
60          {
61          candidates.insert(candidates.end(), local_candidates.begin(),
                local_candidates.end());
62          }
63      }
64
65      // 3. 排序并取 topk
66      std::partial_sort(
67          candidates.begin(), candidates.begin() + std::min(topk,
                (int)candidates.size()),
68          candidates.end(), [](const std::pair<float, int>& a, const std::pair<float,
                int>& b) { return a.first < b.first; }
69
70      );
71      candidates.resize(std::min(topk, (int)candidates.size()));
72
73      // 4. 构建堆返回
74      std::priority_queue<std::pair<float, int>, std::vector<std::pair<float, int>>,
            MinHeapComparator> heap;
75      for (const auto& c : candidates) {
76          heap.emplace(c.first, c.second);
77      }
78      return heap;
79  }
```

5.3 运行结果

OpenMP 方式召回率与 M、nprobe 的关系图如下：

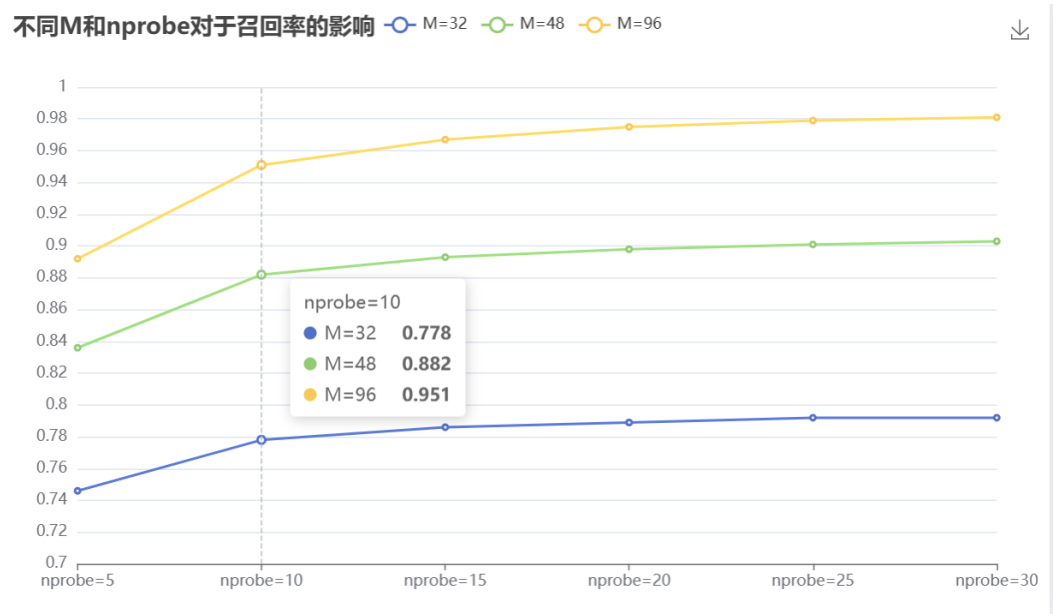

图 5.6: OpenMP 方式 Recall 与 M、nprobe 关系图

OpenMP 方式的 Latency 与 M、nprobe 的关系图如下：

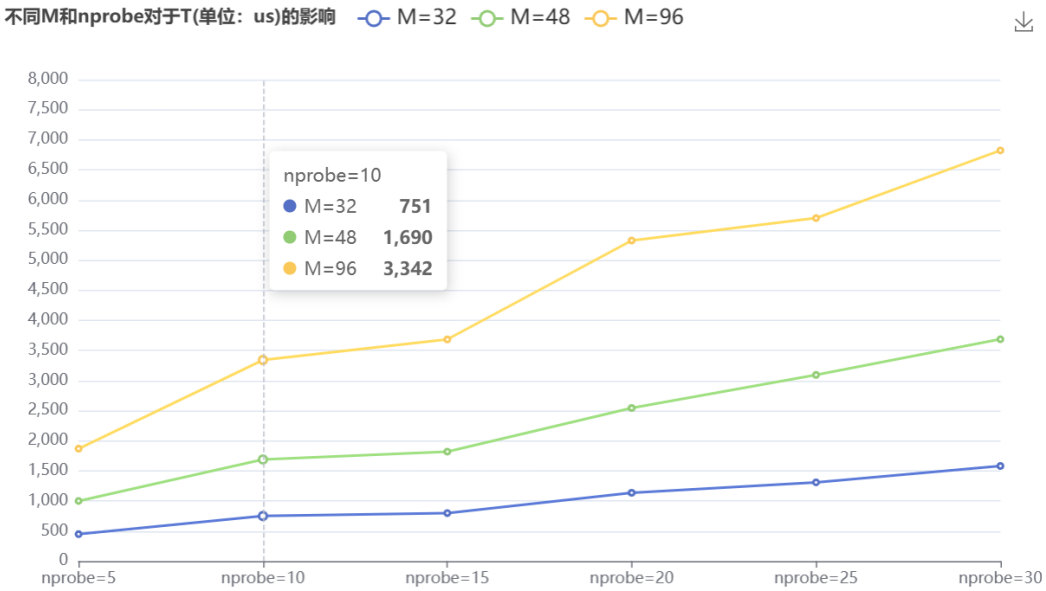

图 5.7: OpenMP 方式 Latency 与 M、nprobe 关系图

此处给出折线图中的部分具体数据:

表 4: OpenMP 中 Recall、Latency 与 M、nprobe 的部分具体数据

M	nprobe	Recall	Latency(us)
32	10	0.778	751
32	20	0.789	1137
32	30	0.792	1582
48	10	0.882	1690
48	20	0.898	2545
48	30	0.903	3687
96	10	0.951	3342
96	20	0.975	5327
96	30	0.981	6823

6 Pthread 和 OpenMP 的对比

表 5: 基于自己实现的 `pthread` 与 OpenMP 并行加速实现对比

比较项	**`pthread` 实现**	**OpenMP 实现**
并行机制	手动创建与回收线程（`pthread_create` / `join`）	使用 `#pragma omp parallel` 指令自动并行
任务划分	手动划分 coarse centroid 区间，传入线程函数	自动将循环分配给多个线程并发执行
实现形式	显式定义线程函数 `search_worker` 与参数结构体 `ThreadArg`	在循环外插入 `#pragma omp for` 实现并行
局部结果	每个线程维护一个 `local_heap` 小顶堆	每个线程维护一个 `local_candidates` 容器
合并方式	主线程统一回收各个线程维护的最小堆并且合并	使用 `omp critical` 合并局部候选结果
编程复杂度	高：需手动管理线程	低：一行指令并行化，自动实现线程合并
适用场景	适用于性能优化、线程调度需要精细控制的场合	适合结构简单、易于拆分的并行计算任务

7 总结

此次实验中，我实现了 Pthread 和 OpenMP 对于 IVF+PQ 的并行化，十分有收获，包括以下几点：

- **IVF+PQ 的基本原理**：IVF 能将聚类得到的中心进行分区，而 PQ 能将高维的向量进行聚类与压缩，二者相结合效果极佳，是用于加速 ANN 的主流方法；

- **Pthread 的基本使用**：首先先要构建线程的结构体，再是写出每个线程的处理函数，最后通过 `pthread_create` 创建多个线程，最后再统一回收所有线程；
- **OpenMP 的基本使用**：仅需在最开始加上一句 `#pragma omp parallel for` 就能实现对 coarse centroid 迭代的并行化处理；
- **编码能力的提升**：本次实验不仅让我掌握了 IVF+PQ 的构建流程，还让我通过学习索引的构建与读取等共同促进了我代码能力的进步。

参考文献

[1] MALKOV Y A, YASHUNIN D A. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs[C]//IEEE Transactions on Pattern Analysis and Machine Intelligence: vol. 42: 4. 2020: 824–836.

[2] JÉGOU H, DOUZE M, SCHMID C. Product quantization for nearest neighbor search[C]//IEEE Transactions on Pattern Analysis and Machine Intelligence: vol. 33: 1. IEEE, 2011: 117–128.

[3] JÉGOU H, DOUZE M, SCHMID C. Searching with quantization: approximate nearest neighbor search using short codes[C]//CVPR. 2011: 1610–1617.

[4] BUTENHOF D R. Programming with POSIX threads[M]. Addison-Wesley, 1997.

[5] DAGUM L, MENON R. OpenMP: An industry standard API for shared-memory programming [J]. IEEE Computational Science and Engineering, 1998, 5(1): 46–55.