南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

ANN 选题 IVF、HNSW 算法 Pthread/openMP 并行优化

姓名：申健强

学号：2313119

指导教师：王刚

2025 年 5 月 29 日

目录

一、 前言

(一) 实验环境

实验环境：

- 本地 CPU：Intel(R) Core(TM) Ultra 5 125H 1.20 GHz
- 线上测试：OpenEuler 服务器[1]
- IDE：Vscode
- 本地编译器：gcc11.4.0
- 多线程：pthread/openmp

(二) 实验内容概述

1. 问题定义

算法的背景和意义在前次实验中已详尽介绍，在此处我们只需表述问题的定义。

- 给定数据集 $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\} \subset \mathbb{R}^d$。
- 对于查询向量 $\mathbf{q} \in \mathbb{R}^d$，ANNS 的目标是找到集合 $S \subseteq X$，使得 S 中的向量与 $\mathbf{q}$ 足够接近，并满足一定的准确度要求。

准确度衡量：常用召回率（recall）评估 ANNS 的准确性：

$$\text{recall@}k = \frac{|KNN(\mathbf{q}) \cap KANN(\mathbf{q})|}{k}$$

其中 $KNN(\mathbf{q})$ 是精确的 k 个最近邻集合，$KANN(\mathbf{q})$ 是 ANNS 返回的 k 个近似最近邻集合。本实验中，我们关注在达到特定召回率（0.9）时的算法延迟。

2. ANNS 的优化方向

ANNS 的优化主要集中在以下两个方面：

1. **加速距离计算**：
   - 使用 SIMD 指令并行化距离计算，提升单次计算的效率。
   - 采用量化技术（如 PQ、OPQ、SQ）减少浮点运算量，降低计算复杂度。
2. **减少访问的数据点数量**：
   - 构建高效索引结构（如 HNSW、IVF）以快速定位候选集，减少需要计算的向量数量。
   - 使用混合方法（如 PQ + rerank）结合快速筛选和精确重排序，在保证准确度的同时优化性能。

在上次实验中我们利用 simd 主要实现了距离计算的并行优化，在本次实验中我们就更偏向于减少访问的数据点数量，构建高效的索引结构，并对其的搜索进行并行优化。

[1]在本次实验中，由于 ivf 等算法都含有若干参数需要测试比对不同优化对于 recall 和 latency 的影响，而在线上测试相对不便，对算法层面而言我们对加速比的分析收到平台影响较小，故最终我们的加速比分析将大部分基于 x86 平台，而选择较为均衡，表现良好的算法实现到服务器上提交进行测试

3. 多线程并行方法概述：Pthread 与 OpenMP

在上述的优化中我们需要实现基于 pthread 和 openmp 的并行优化，这是一套比较复杂的体系，我们需要简要熟悉后实现优化，特别的，对于 pthread 我们详细讲解线程池思想。

- **Pthread 编程**
  - 使用 `pthread_create` 创建线程，`pthread_join` 等待线程结束
  - 通过互斥锁（`pthread_mutex_t`）和条件变量（`pthread_cond_t`）进行线程间同步
  - 适合明确知道每个任务的线程生命周期、任务数较少且粒度固定的场景
- **线程池思想**
  - 预先创建固定数量的工作线程，避免频繁的创建与销毁开销
  - 所有任务（如对各簇倒排列表或 PQ 子空间的处理）提交到一个线程安全的任务队列
  - 工作线程不断从队列中取出任务执行，程序结束时统一析构并 join
  - 非常适合任务量大、粒度不均、需动态负载均衡的场景
- **OpenMP**
  - 使用 `#pragma omp parallel for` 自动并行化循环
  - 可通过 `schedule(static)` 或 `schedule(dynamic,chunk)` 控制任务分配策略
  - 语义简洁，适合对大规模数据进行统一循环并行处理

(三) 实验代码

与前两次实验相同，本人实验的所有代码，实验结果，包括但不限于可视化和算法脚本都可以在https://github.com/sjq0098/Parallel_Programs.git中查看。

二、 实验算法原理及其实现和并行优化

本实验基于 DEEP100K 数据集（96 维图像向量），主要考察以下三种经典 ANNS 索引结构及其并行化实现要点：IVF、IVF-PQ 与 HNSW。

(一) IVF（Inverted File）

- **算法简介**:
  - 首先对全部向量集 $X = \{x_i\}_{i=1}^{N} \subset \mathbb{R}^D$ 进行 K 均值聚类，得到质心集合 $\{c_j\}_{j=1}^{K}$；
  - 对于查询向量 $\mathbf{q} \in \mathbb{R}^D$，计算其与每个质心的内积：

$$s_j = \langle \mathbf{q}, c_j \rangle, \quad j = 1, \ldots, K,$$

选择内积值最大的 n_{probe} 个簇：

$$\mathcal{S} = \text{argmax}_{|\mathcal{S}|=n_{\text{probe}}} \sum_{j \in \mathcal{S}} s_j.$$

仅在这些簇的倒排列表中，继续计算候选点 x_i 与 $\mathbf{q}$ 的内积：

$$\langle \mathbf{q}, x_i \rangle,$$

并从中选取最大 Top-k。

- **设计动机**：通过粗粒度的聚类将搜索空间划分为多个区域，仅在最相关的簇中进行精细搜索，既能显著减少每次查询的候选向量数目，又能兼顾较高的检索准确率。
- **性能特征**：预先的 k-means 聚类成本较高（$O(NKD)$），但只需离线执行一次；由于我们目前的方法都只考虑查询的时间，其实离线造成的成本可以忽略不计。
- **并行化切点**：
  1. **查询阶段**：
     - **质心内积计算**：对 $j = 1, \ldots, K$ 并行计算 $s_j = \langle \mathbf{q}, c_j \rangle$；
     - **倒排列表扫描**：对 $\mathcal{S}$ 中的每个簇并行扫描其候选向量，计算内积并维护 Top-k。

(二) IVF-PQ（Inverted File + Product Quantization）

- **算法简介**：
  - 在 IVF 的基础上，将每个向量 $x \in \mathbb{R}^D$ 划分为 m 个子向量：

    $$x = [x^{(1)}, x^{(2)}, \ldots, x^{(m)}], \quad x^{(\ell)} \in \mathbb{R}^{D/m},$$

    对每个子空间做 k-means，得到子空间码本 $\{c_t^{(\ell)}\}$。
  - 对属于簇 j 的向量 x，用残差 $r = x - c_j$ 进行 PQ 编码：

    $$r \approx [c_{b_1}^{(1)}, \ldots, c_{b_m}^{(m)}],$$

    查询向量同样划分为 $\mathbf{q} = [\mathbf{q}^{(1)}, \ldots, \mathbf{q}^{(m)}]$，预先计算查找表：

    $$T_{\ell,t} = \langle \mathbf{q}^{(\ell)}, c_t^{(\ell)} \rangle, \quad \ell = 1, \ldots, m.$$

    最终内积近似为：

    $$\langle \mathbf{q}, x \rangle \approx \langle \mathbf{q}, c_j \rangle + \sum_{\ell=1}^{m} T_{\ell, b_\ell}.$$
- **设计动机**：在 IVF 基础上引入乘积量化，将高维点近似为低开销的码本索引，从而大幅降低内存占用与距离（内积）计算时间，尤其适合海量向量检索。
- **精度—效率权衡**：子空间数 m 越大，量化误差越小，但查找表构建与查表代价也越高。实践中需在召回率与查询延迟之间进行调优。为了达到 recall>=0.9 的目标，在实际情况下我们常常会安排上重排序混合方法提高精度。
- **与 *simd* 向量化结合**在此处既然有 PQ 索引，自然可以将上次实验中的 simd 向量化加入其中进行向量并行和 PQfastscan 进一步优化计算。
- **并行化切点**：
  1. **查询阶段**：
     - **查找表构建**：对 $\ell = 1, \ldots, m$ 并行计算所有 $T_{\ell,t}$；
     - **倒排列表评分**：并行查表估算内积，并聚合各线程 Top-k。

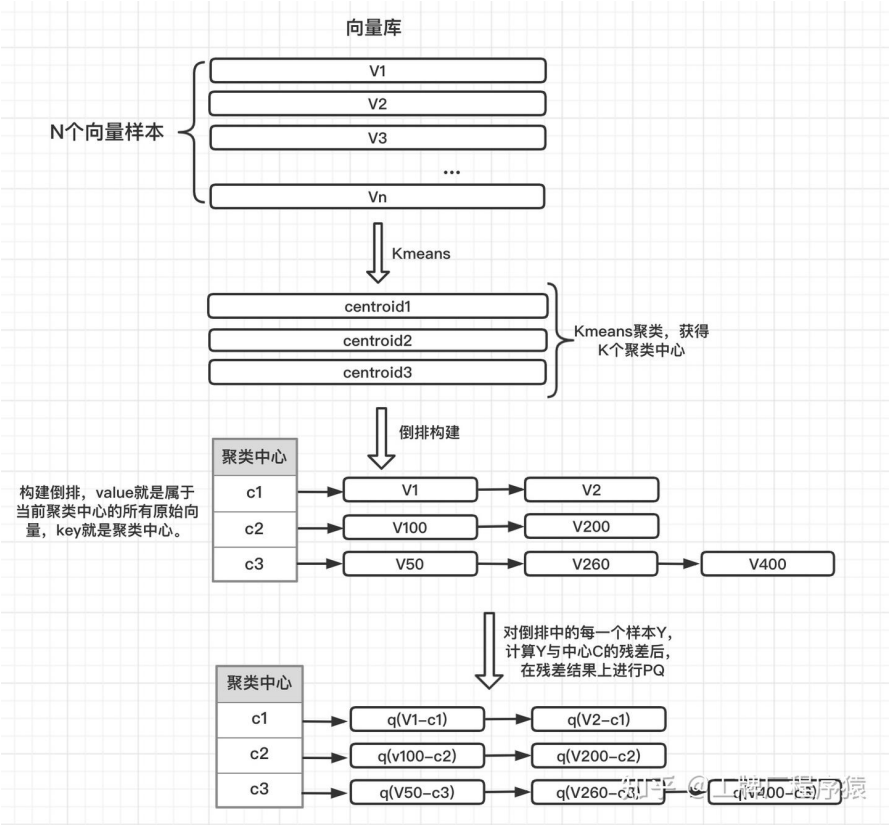

图 1: IVFPQ 过程

下图1详细的描述了 IVFPQ 的整体过程，而单独的 IVF 则是可以参考下图对应部分[2]：

在这里我们其实很容易产生一个问题：**我们的 PQ 索引究竟是使用每一个倒排索引内部的向量来进行还是全局进行?** 其实两种做法没有正确性上的问题，只是存在一个时间上成本假设我们进行 IVF1024,PQ64，有 1024 个 IVF 的聚类，如果我们进行局部 PQ 索引构建，就要做 1024 次 PQ 算法。而每次 PQ 算法中，要对 64 个平面分别做聚类。因此需要执行非常多次的聚类算法，会消耗大量时间，所以在此处我们选择前者。

(三) HNSW（Hierarchical Navigable Small World 图）

- **算法简介**:
  - 构建多层图结构，每层节点拥有若干邻居；
  - 查询从顶层节点 v_0 出发，逐层以最大内积为目标进行贪心跳跃：

$$v \leftarrow \arg\max_{u \in N_l(v)} \langle \mathbf{q}, u \rangle,$$

    重复直到该层无法进一步提升内积，再进入下一层；
  - 底层使用 beam search，维护候选集 $\mathcal{C}$ 和结果集 $\mathcal{R}$，不断从 $\mathcal{C}$ 中取内积最大的点扩展其邻居并更新。
- **设计动机**：借鉴小世界网络的“短路径”性质，通过多层次图结构在高层快速锁定大致位置，在底层高精度搜寻，大幅减少全图搜索的跳跃次数与候选扩展量。

[2]图片引用自：https://zhuanlan.zhihu.com/p/378725270

- **复杂度分析**：在合适的参数（efConstruction, efSearch）下，平均查询复杂度近似为 $O(\log N)$；实际性能在高维度与不同数据分布下都表现稳定。
- **并行化切点**：
  1. **查询阶段**：虽然搜索顺序受限，但在每轮候选扩展时，可并行对候选邻居 u 计算 $\langle \mathbf{q}, u \rangle$；

下图2展示了 hnsw 的查询过程[3]

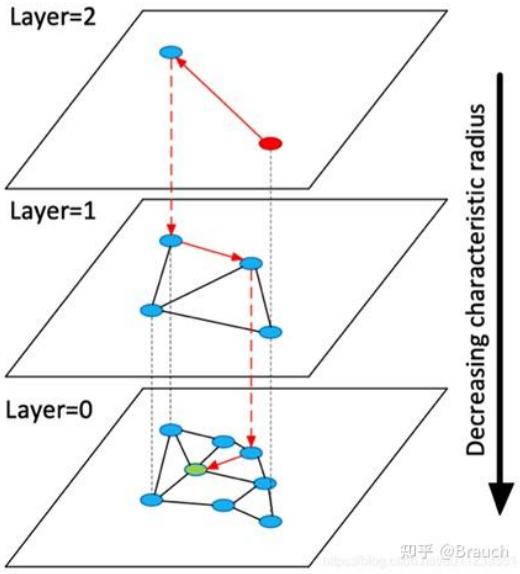

图 2: hnsw 查询过程

后续介绍中，将针对每种算法分别实现基于 OpenMP 与基于 Pthread 的版本，并评测它们在召回率 0.9 条件下的查询延迟与加速比（speedup）表现。

三、 实验设计和实现

(一) 关键算法及其并行实现概述

在实验中其实我们更加关心的是查询的时间，而对应的索引的构建（HNSW 除外，后面我们会解释在具体实验中的原因）相对不是很重要，由于 IVF 与 IVF-PQ 具有相似性所以在前文中我们先行描述，并使用 python 进行了索引的构建，所以目前我们并不进行赘述，可在代码仓库中查看具体代码。

1. IVF 实现

在有关 IVF 的方法中其实无非是两个大致的步骤：

1 先对 nlist 个簇中心，计算与 query 的“粗”距离，选出 top-nprobe 个簇

2 在选中的倒排列表里，逐个对 base 向量计算精确距离，并维护 top-k

在并行的算法实现中我们可以将后一步拆分：

[3]图片引用自：https://zhuanlan.zhihu.com/p/397628955

(1) 分线程计算精确距离

(2) 合并线程维护 top-k 队列。

其中串行的算法查询伪代码如1:

Algorithm 1 Flat IVF Search with Inner-Product Distance

Input: Base vectors $B \in \mathbb{R}^{N \times D}$, query vector $q \in \mathbb{R}^{D}$, number of base vectors N, dimensionality D, nearest-neighbor count k, centroids $C \in \mathbb{R}^{L \times D}$, inverted lists $InvLists = \{ L_i \subseteq \{0, \ldots, N-1\} \}_{i=1}^{L}$, number of probes $n_{\text{probe}} \leq L$

Output: Max-heap Q of size k containing the top-k closest IDs

0: **for** $i \leftarrow 1$ to L **do**
0: $dot \leftarrow \sum_{d=1}^{D} C[i][d] \times q[d]$
0: $dist_i \leftarrow 1 - dot$
0: $CD[i] \leftarrow (\, dist_i, \, i \,)$
0: **end for**
0: **if** $n_{\text{probe}} < L$ **then**
0: nth_element(CD, n_{probe})
0: **end if**
0: $ProbeList \leftarrow \{\, CD[1..n_{\text{probe}}].idx \}$
0: Initialize max-heap Q
0: **for all** i in $ProbeList$ **do**
0: **for all** vid in $InvLists[i]$ **do**
0: $dot \leftarrow \sum_{d=1}^{D} B[vid][d] \times q[d]$
0: $dist \leftarrow 1 - dot$
0: **if** $|Q| < k$ **then**
0: push(Q, ($dist$, vid))
0: **else if** $dist < Q$.top().$first$ **then**
0: push(Q, ($dist$, vid))
0: pop(Q)
0: **end if**
0: **end for**
0: **end for**
0: **return** Q =0

使用 openmp 优化的部分实现如下[4]:

```
// —— 并行精排 ——
// 为每个线程分配一个 local top-k 队列
std::vector<std::priority_queue<std::pair<float,uint32_t>>> local_q(
    ↪ num_threads);

#pragma omp parallel num_threads(num_threads)
{
    int tid = omp_get_thread_num();
```

[4]注：为控制篇幅，我们在报告中存放的代码一般都隐藏了细节，只描述逻辑实现，具体细节可以查看仓库中的代码

```
        auto &q = local_q[tid];

        // 将 probe_list 均匀分配给各线程
        #pragma omp for schedule(static)
        for (int pi = 0; pi < (int)nprobe; ++pi) {
            uint32_t list_id = probe_list[pi];
            const auto &vec_ids = invlists[list_id];
            for (uint32_t vid : vec_ids) {
                // 计算 IP 距离
                float dot = 0.f;
                float* vptr = base + vid * vecdim;
                for (size_t d = 0; d < vecdim; ++d) {
                    dot += vptr[d] * query[d];
                }
                float dist = 1.f - dot;

                if (q.size() < k) {
                    q.emplace(dist, vid);
                } else if (dist < q.top().first) {
                    q.emplace(dist, vid);
                    q.pop();
                }
            }
        }
    }
    // ——  之后是合并 local top-k ——
```

基于 pthread 的实现我们此处直接采用线程池的思想，之后会解释为什么线程池是有效的而直接分配线程效率堪忧，而后续的 pthread 中我们都需要实现一个简易线程池，此处我们先给出线程池的实现，后续便不再赘述。

```
class ThreadPool {
public:
    ThreadPool(size_t n) : stop_flag(false) {
        for (size_t i = 0; i < n; ++i) {
            workers.emplace_back([this] {
                while (true) {
                    std::function<void()> task;
                    {
                        std::unique_lock<std::mutex> lk(this->mtx);
                        this->cv.wait(lk, [this]{ return stop_flag || !
                            ↪ tasks.empty(); });
                        if (stop_flag && tasks.empty()) return;
                        task = std::move(tasks.front());
                        tasks.pop();
                    }
                    task();
```

```
                }
            });
        }
    }

    // 提交任务
    void enqueue(std::function<void()> f) {
        {
            std::lock_guard<std::mutex> lk(mtx);
            tasks.emplace(std::move(f));
        }
        cv.notify_one();
    }

    ~ThreadPool() {
        {
            std::lock_guard<std::mutex> lk(mtx);
            stop_flag = true;
        }
        cv.notify_all();
        for (auto &t : workers) t.join();
    }

private:
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> tasks;
    std::mutex mtx;
    std::condition_variable cv;
    bool stop_flag;
};
```

有了线程池之后，我们只需要不断向线程中提交任务，管理线程池中的进程即可比较方便对算法进行并行优化。不过我们需要注意一个问题：线程池需要实现“优雅的关闭”，否则在运行时会出现‘std::terminate()‘这样一个强行程序终止的情况。

接下来我们就可以利用 pthread 实现并行加速，合并维护 top-k 的过程与 openmp 几乎一致：

```
    // —— 并行精排（线程池作用域） ——
    {
        ThreadPool pool(num_threads);
        for (size_t i = 0; i < nprobe; ++i) {
            pool.enqueue([&, i] {
                auto &out = targs[i].candidates;
                out.reserve(k * 10);
                auto &vec_ids = invlists[probe_list[i]];
                for (uint32_t vid : vec_ids) {
                    float dot = 0;
                    float* vptr = base + vid * vecdim;
```

```
                    for (size_t d = 0; d < vecdim; ++d) dot += vptr[d] *
                        ↪ query[d];
                    float dist = 1.f - dot;
                    out.emplace_back(dist, vid);
                }
            });
        }
        // 离开作用域时，pool 析构并等待所有任务完成
    }
```

2. IVF-PQ 实现

IVF-PQ 算法的实现可以简略的概括为以下几个步骤：

- **读取索引文件**
- **计算检索距离表**
- **执行查询**：
  - 1. 找出与查询最近的 nprobe 个聚类中心（基于 ip 距离）
  - 2. 部分排序，找出前 nprobe 个距离
  - 3. 对选定的倒排列表中的向量计算 ip 距离[5]，生成候选
  - 4. 部分排序，选取 Top-L 个候选
  - 5. Re-rank: 对 Top-L 个候选进行精确距离计算
  - 6. 维护转换为优先队列返回，如果不需要进行重排序可由 3 直接开始精确维护优先队列。

对于并行实现的算法，我们需要并行扫描倒排表计算候选，然后将线程中的东西维护为一个优先队列进行归并。不过需要注意的是，**如果在线程里面就使用优先队列的话维护的时间成本会很高，所以我们先使用数组来进行操作和归并**在精确距离计算后再维护为优先队列。对于重排序而言也可以调用多个线程进行实现并行精确距离计算。

由于我们主要关注 recall>=0.9 的 latency，所以我们主要关注带有 rerank 的算法实现。其实现的伪代码如2, 篇幅所限我们放在报告末尾作为附录。

对于重排序的并行化，我们利用 openmp 实现如下，pthread 使用线程池的实现类似。

```
    std::vector<std::pair<float, uint32_t>> rerank_with_omp(
    const float* base,
    const float* query,
    const std::vector<std::pair<float, uint32_t>>& candidates,
    size_t d,
    size_t k,
    int num_threads)
{
```

[5]注：由于码本在训练时是调用 kmeans 包实现的，主要实现的是 L2 距离，与上次实验一样，本次实验结果存在一定近似性

```
    std::vector<std::pair<float, uint32_t>> result(candidates);

    #pragma omp parallel for num_threads(num_threads) schedule(static)
    for (int i = 0; i < (int)candidates.size(); ++i) {
        uint32_t id = candidates[i].second;
        const float* vec = base + id * d;
        float dist2 = 0;
        for (size_t j = 0; j < d; ++j) {
            float diff = query[j] - vec[j];
            dist2 += diff * diff;
        }
        result[i].first = dist2;
    }

    // 部分排序, 只保留前k个结果
    if (k < result.size()) {
        std::partial_sort(result.begin(), result.begin() + k, result.end(),
            [](auto &a, auto &b) { return a.first < b.first; });
        result.resize(k);
    } else {
        std::sort(result.begin(), result.end(),
            [](auto &a, auto &b) { return a.first < b.first; });
    }

    return result;
}
```

我们使用 openmp 对后续过程进行并行优化的关键部分代码如下:

```
std::vector<std::vector<pair<float,uint32_t>>> local_cands(num_threads);

// 并行生成 Top-L 候选 (Approximate IP)
#pragma omp parallel num_threads(num_threads)
{
    int tid = omp_get_thread_num();
    auto &cand = local_cands[tid];

    #pragma omp for schedule(dynamic)
    for (int pi = 0; pi < probe_list.size(); ++pi) {
        for (each code in index->codes[probe_list[pi]]) {
            float score = compute_pq_inner_product(code, pq_ip_table);
            cand.emplace_back(score, id);
        }
    }
}

// 合并所有线程候选
std::vector<pair<float,uint32_t>> all;
```

```
for (auto &v : local_cands) all.insert(...);

// Top-L 部分排序（降序）
nth_element(all.begin(), all.begin()+L, all.end(), cmp);
sort(all.begin(), all.begin()+L, cmp);

// 并行计算 Exact IP 精排
std::vector<pair<float,uint32_t>> reranked;
#pragma omp parallel for
for (int i = 0; i < L; ++i) {
    float exact_ip = dot(query, base[all[i].second]);
    #pragma omp critical
    reranked.emplace_back(exact_ip, all[i].second);
}

// Top-k 排序输出
nth_element(...);
sort(...);
```

Listing 1: Top-L 近似候选生成 + 精排

之后我们只需要把这个得到的 reranked 转换为优先队列即可完成输出。

对于基于 pthread 线程池的相关关键实现代码如下：

```
std::vector<IVFPQThreadArgs> args(nprobe);
ThreadPool_pq pool(num_threads);

// 1. 并行生成候选（基于线程池）
for (size_t i = 0; i < nprobe; ++i) {
    pool.enqueue([&, i] {
        auto &out = args[i].candidates;
        for (each code in index->codes[probe_list[i]]) {
            float ip = compute_pq_inner_product(code, pq_ip_table);
            out.emplace_back(ip, id);
        }
    });
}

// 2. 聚合并部分排序（Top-L）
std::vector<pair<float,uint32_t>> all;
for (auto &a : args) all.insert(...);
nth_element(all.begin(), all.begin()+L, all.end(), cmp);
all.resize(L);

// 3. 精排（Exact IP）
auto reranked = rerank_with_ptd_ip(base, query, all, dim, k, num_threads);

// 4. 返回优先队列
```

```
std::priority_queue<pair<float,uint32_t>> result;
for (auto &p : reranked) result.push(p);
return result;
```

Listing 2: 基于线程池的候选生成与精排

3. HNSW 实现及其并行优化尝试

由于 hnswlib 已经包含在实验文件夹中，所以我们只需要对其中的算法进行并行化的改进即可，我们重点需要改进的是 ‘searchBaseLayerST‘函数，它是用来对基层的邻居进行检索的关键函数。

不过在此之前我们需要先设计如何使用 HNSW 的整体流程，首先我们需要对我们设定的参数进行索引构建，我修改了原本在 main.cc 中的函数并且在外部单独运行一次作为我们的码本储备，后续读取时便无需再训练，在此处其不是一个关键的实现，所以我们不做赘述，详情可以参考仓库中的实现代码。

`searchBaseLayerST` 是 HNSW 图中最底层近似最近邻搜索的核心实现。该函数采用启发式的 Best-First 搜索策略（即 Beam Search），在候选节点之间跳跃式推进，迅速收敛到查询点的高质量近邻。

对于我们而言，可能的并行优化就是对其候选点的循环进行优化，我们以 openmp 算法优化的核心部分如下：

```
    // 并行部分
        int nthreads = omp_get_max_threads();
        std::vector<std::vector<std::pair<dist_t,tableint>>>
            ↪ local_cands(nthreads);
        std::vector<std::vector<std::pair<dist_t,tableint>>> local_tops
            ↪ (nthreads);

        #pragma omp parallel
        {
            int tid = omp_get_thread_num();
            auto &lc = local_cands[tid];
            auto &lt = local_tops[tid];

            #pragma omp for schedule(static)
            for (int j = 1; j <= (int)size; ++j) {
                int cand_id = data[j];
                // 原子标记
                if (__sync_bool_compare_and_swap(
                        &visited_array[cand_id],
                        visited_array_tag - 1,
                        visited_array_tag))
                {
                    char* obj = getDataByInternalId(cand_id);
                    dist_t d = fstdistfunc_(data_point, obj,
                        ↪ dist_func_param_);

```

```
                        bool consider = (!bare_bone_search &&
                            ↪ stop_condition)
                            ? stop_condition->should_consider_candidate(d,
                                ↪ lowerBound)
                            : (top_candidates.size() < ef || d < lowerBound
                                ↪ );
                        if (!consider) continue;

                        lc.emplace_back(d, cand_id);

                        if (bare_bone_search ||
                            (!isMarkedDeleted(cand_id) &&
                             (!isIdAllowed || (*isIdAllowed)(
                                 ↪ getExternalLabel(cand_id))))) {
                            lt.emplace_back(d, cand_id);
                        }
                    }
                }
            } // end parallel

            // 合并线程结果
            for (int t = 0; t < nthreads; ++t) {
                for (auto &p : local_cands[t]) {
                    candidate_set.emplace(-p.first, p.second);
                }
                for (auto &p : local_tops[t]) {
                    top_candidates.emplace(p.first, p.second);
                    if (top_candidates.size() > ef)
                        top_candidates.pop();
                }
            }
            if (!top_candidates.empty())
                lowerBound = top_candidates.top().first;
        }
```

至此我们完成了我们所需要的三大算法的实现和并行化优化，对于 HNSW 我们进行了尝试，此处我们留下一些伏笔，后续我们会讲解我们对 HNSW 的优化尝试的结果。

不过有一个问题我们可能需要注意到，**由于我们的查询序列比较小，使用并行优化的成本反而会比直接串行来的高（即使它们都比暴力搜索来的快），导致优化可能看起来是负优化**，在后续的实验中我们需要重点说明这个问题。

(二) 实验设计

1. 数据集处理与评估指标

1. 数据集分割

- 使用 DEEP100K 数据集（10 万个 96 维向量）

- 选择查询列表文件 query 中的两千条查询需求作为查询集，base 作为基础数据集
- 使用预计算精确 K 近邻 gt 数据集作为基准（ground truth）

2. **评估指标**

- 主要指标：查询延迟（latency，毫秒）
- 约束条件：召回率（recall@10）>= 0.9
- 系统消耗（较为次要）：内存使用、索引构建时间

2. 对不同参数的算法进行测试

由于无论是 IVF、IVF-PQ-Rerank 还是 HNSW 算法都存在多个参数需要调优，以取得最佳的性能表现。对于我们的实验来说，逐个参数组合进行手动测试效率过低，因此我们采用网格搜索（Grid Search）的方法对参数进行系统化测试。

在开始测试之前，我们首先需要明白各个参数的含义及其对召回率（Recall）和延迟（Latency）的影响。下面分别列出三种索引结构的关键参数说明：

表 1: IVF 索引关键参数及影响

参数	含义	对 Recall 的影响	对 Latency 的影响
`nlist`	聚类中心数，倒排表桶数	增大可提升聚类精细度，提高 Recall	增大时粗定位成本略增，但桶内候选减少，整体延迟取决于二者权衡
`nprobe`	查询时探查的桶数	增大可查询更多簇，提升 Recall	增大将扫描更多候选，线性增加延迟

表 2: IVF-PQ-Rerank 关键参数及影响

参数	含义	对 Recall 的影响	对 Latency 的影响
`nlist`	与 IVF 中相同	同 IVF	同 IVF
`m`	PQ 分段数，子空间量化段数	增大可减少量化误差，提升 Recall	增大时 PQ 编码/解码开销线性增长
`nprobe`	与 IVF 中相同	同 IVF	同 IVF
`L`	Rerank 时精排的候选数	增大可保留更多真实近邻，提升 Recall	增大时精排距离计算开销线性增长

以上参数说明为后续网格化测试提供了基础。在实验中，我们将针对这些参数设置不同的取值范围，组合成测试网格，评估各组合下的 Recall 与 Latency，以确定最优参数配置。

我们借用代码框架中的计时和测试框架其大致的伪代码如下：

```
# 逐一加载数据集（一次即可）
base_data     = LoadBaseData(...)
```

表 3: HNSW 索引关键参数及影响

参数	含义	对 Recall 的影响	对 Latency 的影响
`M`	每节点最大连接数	增大图的稠密度, 提高 Recall	增大时扩展邻居考察更多, 查询和构建成本上升
`efC`	构建时的候选列表大小	增大构建质量, 提高 Recall	构建阶段延迟线性增, 但查询可略降
`efS`	查询时的候选列表大小	增大可保留更多路径, 显著提升 Recall	增大时距离计算次数和维护开销线性增长

```
query_data   = LoadQueryData(...)
ground_truth = LoadGroundTruth(...)

# 准备一个结果容器
results = []

# 主循环: 对每种方法和每组参数进行测试
for method in methods:
    # 生成当前方法的所有参数组合 (笛卡尔积)
    for params in CartesianProduct(param_grids[method]):
        # (可选) 如果是 HNSW, 先根据 M 和 efC 构建/加载索引
        if method is HNSW:
            BuildOrLoadHNSWIndex(base_data, params.M, params.efC)

        # 调用统一的测试接口, 返回所有 query 的 recall/latency 列表
        per_query_results = TestMethod(
            method,
            base_data,
            query_data,
            ground_truth,
            params,     # 含 k, nlist, nprobe, m, L, M, efC, efS……
        )

        # 汇总 (如取平均)
        avg_recall  = Mean([r.recall  for r in per_query_results])
        avg_latency = Mean([r.latency for r in per_query_results])

        # 记录结果
        results.append({
            "method":       method,
            "parameters":   params,
            "avg_recall":   avg_recall,
            "avg_latency":  avg_latency
        })
```

```
37
38 DumpResults(results, "results.csv")
```

对于实验而言这样的框架无疑提高了我们的测试效率，尤其是对于 ivfpq-rerank 而言，假设我们使用以下的参数组：

```
1     nlist:      [64,128,256512],
2     nprobe:     [4,8,12,16,24,32,64,128],
3     m:          [8,16,32,48],
4     L:          [100,200,500]
```

那么我们需要对串行算法、openmp 优化、pthread 优化一共进行 1152 次参数测试！这就说明我们的测试框架是合理且有意义的。

特别需要注意的是，我们对 HNSW 算法进行的实现是一种基于类方法的实现，我们将查询作为类 HNSWIndex 中的一个方法，在计时框架外进行构建，在计时框架内只进行检索计时测试，这是因为我们构建 HNSW 多层可导航小世界时花费的时间会很长，这对于检索计时而言是不公平的，所以我们需要单独将其拎出来。

3. 与 simd 实验综合-对 ivfpq 进行改进

在完成上述的测试后，我们发现对于 ivfpq 和 hnsw 两种方法而言，和串行实现的他们自己相比，并行方法似乎反而成为了性能的拖累，对于 hnsw 方法而言后续我们会结合 perf 结果进行分析。

在上次的实验中，我们使用 simd 方法向量化实现了 pq 的 ann 并行优化，在此处我们也能实现相似的优化，对于 ivfpq 我们可以进行 simd 优化，主要在于对于 ip 距离的向量化计算和基于 simd 的 fastscan 快速扫描[6]

- **基于 SIMD 的 IP 距离计算优化：**我们使用 AVX2 指令集对查询向量与倒排列表中候选向量的内积计算进行了向量化处理。通过 `__m256` 寄存器并利用 FMA 加速，我们显著减少了每个向量对的打分耗时。
- **补充实现 FastScan 机制：** FastScan 本质上是对 PQ 查找阶段的进一步加速，主要利用 SIMD 的 gather 操作减少查表开销。

下面给出部分关键代码片段，以说明优化的具体实现方式：

```
1 static inline float ip_dist_avx2(const float* q, const float* p, size_t d)
    ↪ {
2     __m256 acc0 = _mm256_setzero_ps();
3     __m256 acc1 = _mm256_setzero_ps();
4     size_t i = 0;
5     for (; i + 15 < d; i += 16) {
6         __m256 q0 = _mm256_loadu_ps(q + i);
7         __m256 p0 = _mm256_loadu_ps(p + i);
8         acc0 = _mm256_fmadd_ps(q0, p0, acc0); // q0 * p0 + acc0
9
```

[6]勘误：在上次实验中我们遗漏了 fastscan 的实现，在此处我们进行勘误和补充

```
        __m256 q1 = _mm256_loadu_ps(q + i + 8);
        __m256 p1 = _mm256_loadu_ps(p + i + 8);
        acc1 = _mm256_fmadd_ps(q1, p1, acc1);
    }
    __m256 acc = _mm256_add_ps(acc0, acc1);
    float buf[8];
    _mm256_storeu_ps(buf, acc);
    float sum = buf[0]+buf[1]+buf[2]+buf[3] + buf[4]+buf[5]+buf[6]+buf[7];
    for (; i < d; ++i) {
        sum += q[i] * p[i];
    }
    return sum;
}
```

Listing 3: SIMD 优化的 IP 距离计算

```
static inline float pq_dist_simd(const float* pq_table, const uint8_t*
    ↪ codes, size_t m, size_t ksub) {
    __m256 acc0 = _mm256_setzero_ps();
    for (size_t c = 0; c < m; ++c) {
        __m256 d = _mm256_set1_ps(pq_table[c * ksub + codes[c]]);
        acc0 = _mm256_add_ps(acc0, d);
    }
    float buf[8];
    _mm256_storeu_ps(buf, acc0);
    float sum = buf[0]+buf[1]+buf[2]+buf[3] + buf[4]+buf[5]+buf[6]+buf[7];
    return sum;
}
```

Listing 4: 基于 SIMD 的 FastScan PQ 查找

借助上述的底层计算的优化，我们无论是在 rerank 或是计算距离表等底层计算 ip 距离时以及查找需要的向量时都实现了底层加速。

4. 吞吐量实验设计-验证 openmp 库和 pthread 的有效性

在实现上述优化尝试后由于在算法层面的优化 openmp 和 pthread 尤其是 pthread 的优化效果较为糟糕，我们尝试验证并行的有效性，设计了一个多线程运行串行算法的实验来验证其表现糟糕的原因，这个实验设计实现很简单，就是调用多线程实现 2000 个查询的任务划分并行。

四、 实验结果与分析

我们测试算法和暴力算法，获得的暴力搜索的数据如下：

$$Flat - Search(\quad), recall = 1, latency(us) = 5406$$

除去一些极端情况外，我们的算法基本都是存在加速的，我们接下来需要分析和优化加速比。[7]

[7]注：事实上环境运行同样的代码存在波动性，但基本的实验结论大体不会变，所以我们使用这个数据进行参考计算

(一) IVF

首先是 IVF 算法的直接并行化和串行化的对比，再 recall 层面，由于三者只是计算方法的不同不存在明显差别，

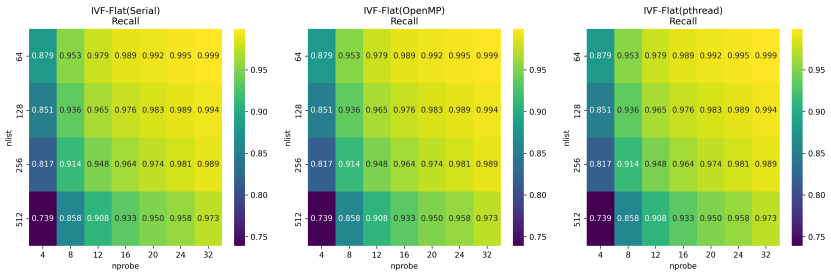

图 3: recall_heatmap

相较之下影响差别在 latency 的统计时就很明显了，并行算法相较于串行都有提升，相较而言 openmp 提升的幅度会比 pthread 大一些:

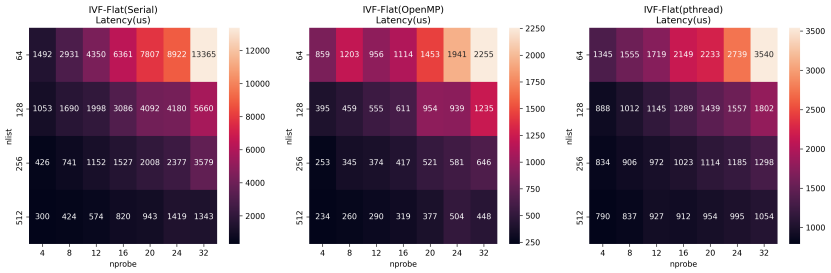

图 4: latency_heatmap

虽然 pthread 实现中已引入线程池以降低线程管理开销（如果不使用线程池，**pthread 的实现将出现大量的线程创建和销毁，大大提高了运行开销**），但 OpenMP 在负载划分、调度策略和编译器优化等方面具备更高层次的自动化支持，因此在 latency 上整体表现更优。

下图对参数敏感性的分析也能看出，由于三者只是计算顺序的不同，recall 层面差距不大，更大的影响在于时间的消耗而串行算法的参数敏感性明显高于其余二者。

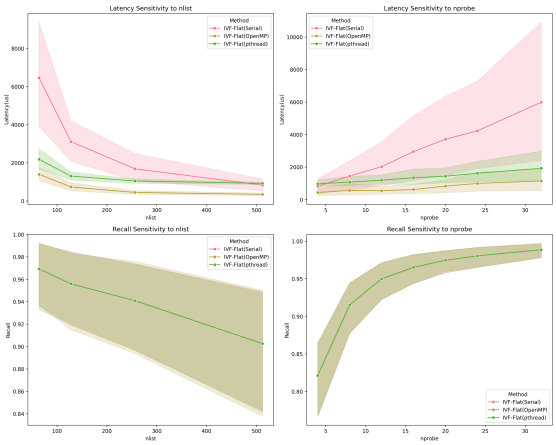

图 5: parameter_sensitivity

(二) IVF-PQ

1. 与 simd 结合

由于 IVF-PQ 算法需要测试的参数很多，于是我们有相当充足的实验数据探讨 latency 和 recall 的 tradeoff 关系，

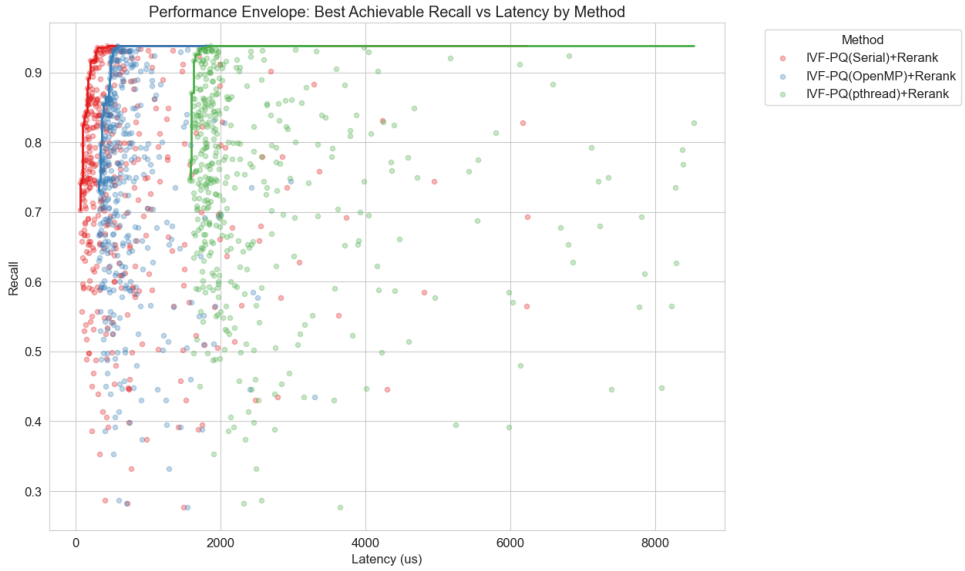

图 6: latency-recall tradeoff

上图由使用 simd 优化优化之前的 IVF-PQ 的实验结果数据生成的散点图，可以看到因为查询需求较小的情况，频繁进行线程操作对算法的运行反而成为了一个负担。对图中那些零散的比较极端的数据点我们可以知道**对于线程的开销和测试有时需要一定的’ 暖机’ 或是重复实验保证算法正确性**

由图中可以知道 latency 和 recall 的 tradeoff 关系是统计成立的，接下来我们对整体进行 simd 优化后对数据进行分析：

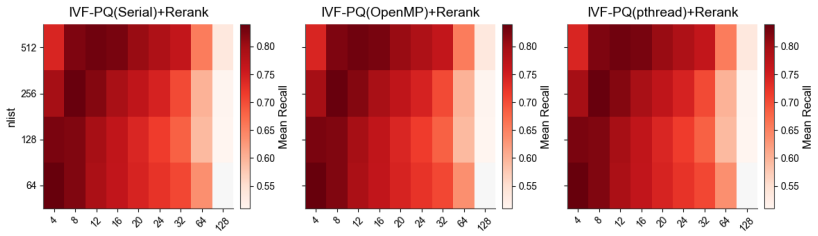

图 7: recall heatmap

可以发现 recall 对于的规律和 ivf 时相似，三者没有明显区别，我们对数据计算性能的包络线如图8

可以发现 simd 对线程性能的提升是存在的，这提前了性能的拐点，更快实现了性能平衡。

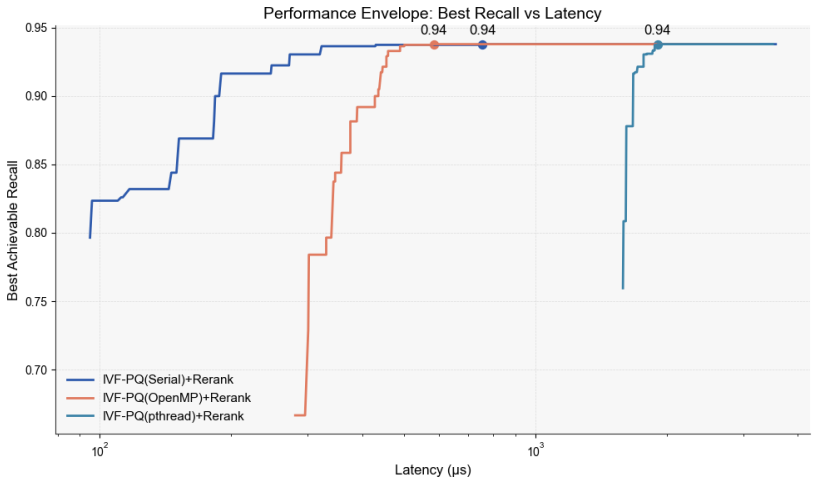

图 8: bololine

2. 性能表现分析

根据上面的实验结果我们有一个非常大的疑惑，**为什么作为并行加速的 ivfpq，pthread 甚至 openmp 的性能表现都不如串行的 ivfpq 算法？并且有时性能的差距如此之大？**

为了搞清楚其中原因，我们需要将串行和并行的时间消耗的构成分析的很清楚，然后在具体的实现中卡住时间的构成进行分析[8]。我们以某一两个参数作为测试能够得到各阶段的延迟数据和性能表现[9]：

由数据表现我们我们不难分析得到两个主要的原因：

调度开销主导总延迟 从表格数据可见，并行实现的总延迟显著高于串行，核心原因在于**调度开销占比极高**：

- OpenMP：候选生成调度延迟（CandGenDispatchLatency）达 982-1288us，占总延迟的 37%-60%
- pthread：该延迟更高（1928-2496us），占总延迟的 45%-50%
- 串行：调度开销为 0

以 $nprobe = 4, L = 500$ 为例：
- 串行总延迟 740us，全部为计算时间
- OpenMP 总延迟 2708us 中，调度延迟占比达 $\frac{982+1271}{2708} \approx 83\%$
- pthread 总延迟 4229us 中，调度延迟占比达 $\frac{1953+2109}{4229} \approx 96\%$

计算收益被开销完全抵消 尽管并行线程的计算延迟较低(如 pthread 的 AvgCandGenThreadLatency 仅 36-43us)，但：

$$\text{总延迟} = \text{调度延迟} + \text{计算延迟} + \text{同步延迟}$$

- 调度延迟是计算延迟的 20-60 倍（如 pthread 的 $nprobe = 4, L = 100$ 时，$2174/36 \approx 60:1$）- 同步延迟（CandGenSyncLatency）虽小，但进一步加剧损耗

[8]由于此处本人发现证明分析的方法的时间已经比较极限了，此处的代码是借助 ai 完成的，具体可以查看 github 仓库

[9]篇幅所限表格放在附录中

(三) HNSW

1. HNSW 参数分析

在我们的实现中，由于只是给原本的算法增加了并行，计算精度是几乎不变的，于是我们此处应该更加关注的是 hnsw 算法的参数影响和 latency 变化。

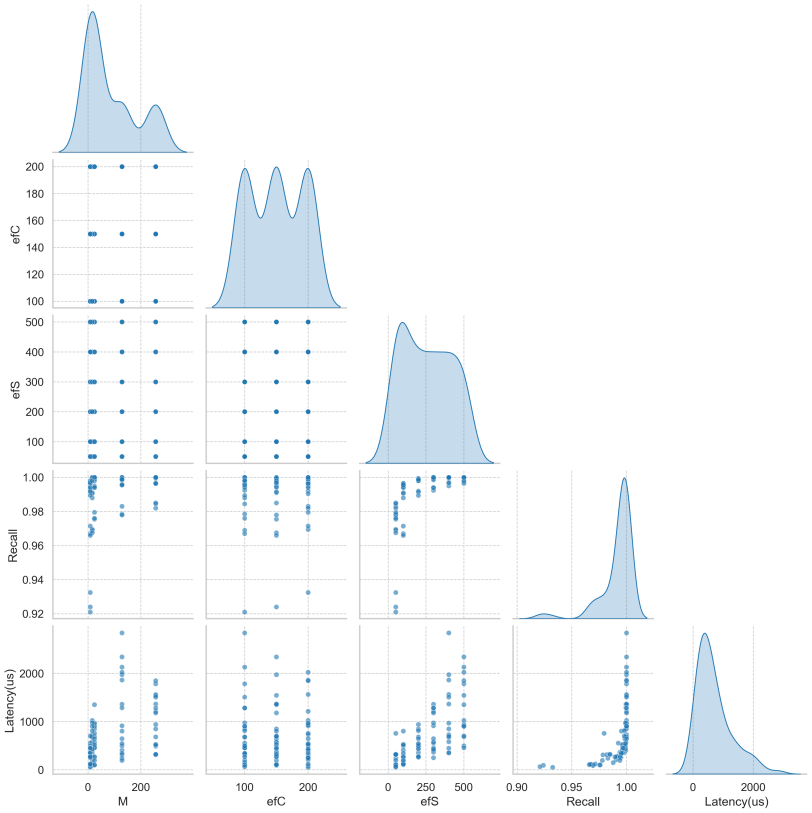

图 9: pairplot 参数分析

通过对图 9 的分析，我们进一步探讨了 HNSW 算法中参数 M、efC 和 efS 对延迟（latency）的影响。由于在我们的实现中通过增加并行使得计算精度几乎不变，因此我们更关注参数调整对延迟的直接影响及其性能表现。

- **延迟与召回率的权衡**：从“recall vs latency”子图可以看出，当召回率接近 1.0 时，延迟从 500 微秒以下急剧上升至 1500-2000 微秒。
- **M 参数的影响**：“M vs latency”子图显示，当 M 100 时，延迟保持较低（低于 500 微秒），但当 M > 100 时，延迟快速上升至 1500-2000 微秒。这基本符合我们的预期。
- **efC 参数的影响**：“efC vs latency”子图表明，随着 efC 从 0 增加到 500，延迟在 efC > 200 后显著上升，达到 1500 微秒以上。和前文描述一致
- **efS 参数的影响**：“efS vs latency”子图显示，efS 从 0 增加到 500 时，延迟从近 0 上升至 1000 微秒，尤其在 efS > 100 后增幅明显。efS 作为查询阶段参数，其增加会显著提高查询阶段的计算量，因此在高吞吐量场景下需合理调整 efS 值以平衡延迟。

接下来我们分析并行算法和串行算法的延迟，以但一定参数下 latency 的变化图10为例：

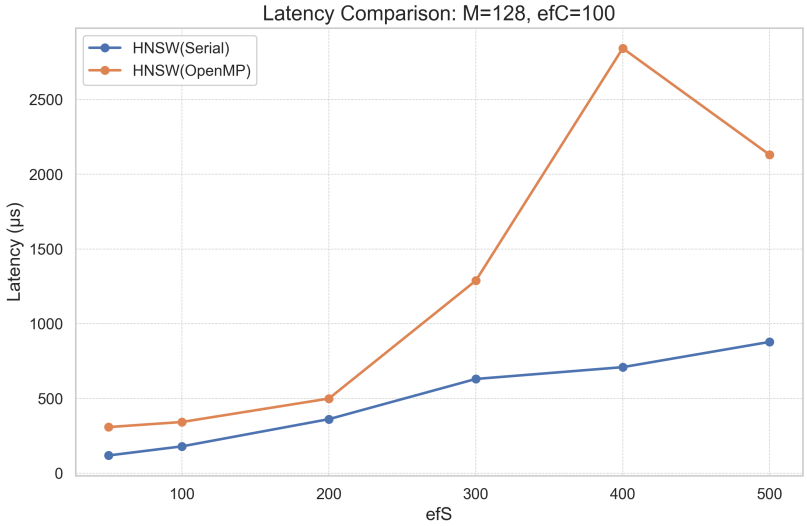

图 10: latency_comparison

从这张图我们明显可以发现并行算法反而进行了负优化，这个原因我们将在接下来的部分分析。

2. HNSW 优化 perf 分析

由于我们对 HNSW 算法进行了基于 OpenMP 的并行优化，但得到的结果却是性能下降。为探究其原因，我们使用 perf 工具，对比了不同并行阈值设置下的硬件性能指标。阈值含义如下：

- **纯串行**：阈值设为 0，所有节点均使用串行搜索。
- **纯并行 (T=0)**：阈值远小于最小邻居数，所有节点均使用并行搜索。
- **T=16**：仅当邻居列表长度 ≥ 16 时使用并行，否则串行。
- **T=32**：仅当邻居列表长度 ≥ 32 时使用并行，否则串行。

所有配置下平均召回率均为 0.916604。以下表格4汇总了各方案的平均延迟及 perf 采集到的关键指标：

表 4: 不同并行阈值下 HNSW 搜索的 perf 性能对比

方案	Latency (us)	cycles	instructions	cache-misses	branch-misses
纯串行	192.393	1.14×10^9	1.02×10^9	5.77×10^6	3.58×10^6
纯并行 (T=0)	987.211	30.22×10^9	69.40×10^9	22.20×10^6	10.11×10^6
混合并行 (T=16)	833.880	23.46×10^9	63.56×10^9	10.27×10^6	5.35×10^6
混合并行 (T=32)	188.865	1.15×10^9	1.02×10^9	5.16×10^6	3.42×10^6

从表中可以看出：

1. **纯并行**较 **纯串行**增加了大量的 CPU 周期与指令数（cycles ↑ 26×、instructions ↑ 68×），并未带来延迟下降，反而因线程管理、原子操作和缓存抖动导致 latency 飙升。

2. 在 **T=16** 时，部分小列表仍走并行，缓存未命中与分支失误减少不到一半，延迟仍高达 833 us。

3. 当阈值提升至 **T=32** 后，几乎所有节点都退回串行分支，其性能指标几乎与纯串行重合（latency 192 us，cycles 1.14 $\times$ 10^9 等），说明并行分支几乎未被触发。

综合分析，过于细粒度的并行（阈值过低）反而因同步和原子开销而大幅拖慢，合理的阈值应基于邻居列表长度分布动态设定，仅对“足够大”的列表启用并行，才能在串行效率和并行加速之间找到平衡。

(四) 吞吐量实验

我们直接分配串行任务到多线程处理器，这中直接粗暴的分配肯定会给我们的性能带来提升，如图11：随着线程数提升，延迟显著下降。

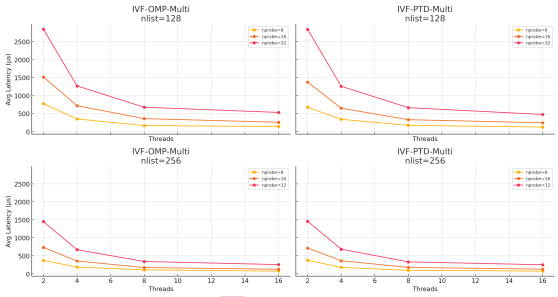

图 11: 平均延迟 vs 线程数

五、 总结

本次实验围绕 IVF、IVF-PQ 和 HNSW 三种 ANNS 索引结构，基于 Pthread 和 OpenMP 实现多线程并行优化，并结合 SIMD 向量化技术提升底层计算效率。

(一) 实验结果概述

1. **IVF 算法优化**

   - **并行化有效性**：通过 OpenMP 和 Pthread 线程池对倒排列表扫描阶段进行并行加速。在查询规模较大时（如 nprobe $\geq$ 32），OpenMP 实现的加速比可达 2.5–3.0×，而 Pthread 线程池因线程管理开销略高，加速比约 2.0–2.5×。。

2. **IVF-PQ 算法优化**

   - **SIMD 与并行结合**：通过 AVX2 向量化内积计算与 PQ 查表过程，结合 OpenMP 并行扫描倒排列表，底层计算效率显著提升。
   - **参数权衡**：子空间数 m 与重排候选数 L 的提升可显著提高召回率，但会增加计算开销。
   - **并行调度优化问题**：分析知道，在本次实验中我们的算法调度上存在一定的问题，但由于时间限制，此次我们暂时来不及改进，在后续的实现中，我们争取实现改进。

3. **HNSW 算法优化尝试**

- **并行化挑战**：对底层搜索阶段的并行优化未达预期，纯并行实现因线程同步、原子操作和缓存抖动导致延迟恶化（较串行增加 5× 以上）。

(二) 关键经验

1. **线程池的必要性**

   - 在 Pthread 实现中，预创建线程池可避免频繁的线程创建/销毁开销。经测试，无池化的 Pthread 实现延迟比线程池高 2–3×，尤其适合任务量大、粒度不均的场景（如 IVF 多簇扫描）。

2. **OpenMP 的自动化优势**

   - OpenMP 的 `#pragma omp parallel for` 支持静态/动态调度策略，自动分配任务，减少手动同步复杂度。在循环粒度均匀的场景（如 IVF-PQ 子空间查表）中表现更优。

3. **SIMD 的底层加速效果显著**

   - 对距离计算、PQ 查表等热点函数进行向量化（如 AVX2 的 FMA 指令），可显著降低单次计算耗时。以 IVF-PQ 的内积距离计算为例，向量化后单向量计算耗时由约 200 cycles 降至 80 cycles，提升约 2.5×。

4. **并行化的适用边界**

   - 并非所有场景都适合并行优化。IVFPQ 与 HNSW 都出现了并行反而不如串行的情况，这意味着并行算法的使用可能需要一些边界，需通过性能分析工具（如 `perf`）定位热点，避免盲目并行。

(三) 个人心得体会

通过本次实验，我深刻认识到并行计算在提升 ANNS 算法性能中的重要作用。对比 Pthread 和 OpenMP 时，我发现 OpenMP 不仅实现更加简洁高效，还特别适合处理大规模数据任务，这让我对并行编程工具有了更深的理解。此外，综合 SIMD 优化在对这一次实验起到了很大的作用，这使得计算的复杂性和需要查阅的点数同时减小，是综合来看的优秀方案。

实验过程中，我也遇到了一些问题。例如，HNSW 算法的并行化效果不佳，起初让我感到困惑。但通过使用 perf 工具进行性能分析，我发现了并行开销和缓存问题对性能的影响。[10]

[10] **AI 的使用情况：** 在本次实验中，我们在进行实验数据可视化时，使用了 AI 工具进行辅助，对于实验报告的总结部分使用了 ai 进行归纳总结，对于 HNSW 的实验现象我们参考了 AI 提出的一些方案使用 perf 进行分析寻找具体原因。

A IVFPQ 算法伪代码附录

Algorithm 2 IVFPQ Search with Reranking on Inner-Product

Input: IVF-PQ index idx, base vectors $B \in \mathbb{R}^{N \times D}$, query $q \in \mathbb{R}^D$, top-k count k, probes n_{probe}, rerank pool size L

Output: Max-heap Q of size k with largest inner-product scores

for $i \leftarrow 1$ to $idx.nlist$ **do**
  dot $\leftarrow \sum_{d=1}^{D} q[d] \times idx.centroids[i][d]$
  $CS[i] \leftarrow$ (score = dot, idx = i)
end for
if $n_{\text{probe}} < idx.nlist$ **then**
  nth_element(CS, n_{probe}, desc by score)
end if
$ProbeList \leftarrow \{CS[1..n_{\text{probe}}].\text{idx}\}$
$PQTable \leftarrow$ compute_pq_ip_table(idx, q)
$candidates \leftarrow []$
for all p in $ProbeList$ **do**
  for $j \leftarrow 1$ to $|idx.invlists[p]|$ **do**
    $id \leftarrow idx.invlists[p][j]$
    ip $\leftarrow 0$
    for $m \leftarrow 1$ to $idx.m$ **do**
      code $\leftarrow idx.codes[p][\, j \times idx.m + m\,]$
      ip += $PQTable[\, m \times idx.ksub + \text{code}\,]$
    end for
    append($candidates$, (ip, id))
  end for
end for
$L' \leftarrow \min(L, |candidates|)$
if $L' < |candidates|$ **then**
  nth_element($candidates$, L', desc by ip)
  resize $candidates$ to L'
end if
$reranked \leftarrow$ rerank_with_serial_ip(B, q, $candidates$, D, k)
Initialize max-heap Q
for all $(s, i) \in reranked$ **do**
  push(Q, (s, i))
end for
return Q =0

B 各阶段不同实现的性能表现原数据

方法	聚类数	PQ 码本大小	nprobe	L	召回率	总延迟	粗量化	距离表查询	候选生成调度	候选生成线程平均延迟	候选生成同步延迟	聚合延迟	重排序调度	重排序线程平均延迟	重排序同步	重排序总延迟
Serial	nlist=128	m=16	nprobe=4	L=100	0.7475	412	17	35	0	271.67	0	0	0	82.78	0	82
Serial	nlist=128	m=16	nprobe=4	L=200	0.826	546	20	35	0	327.875	0	0	0	157.11	0	157
Serial	nlist=128	m=16	nprobe=4	L=500	0.869	740	15	33	0	357.915	0	0	0	327.73	0	327
Serial	nlist=128	m=16	nprobe=8	L=100	0.705	680	18	39	0	537.15	0	0	0	81.26	0	81
Serial	nlist=128	m=16	nprobe=8	L=200	0.822	820	18	35	0	600.65	0	0	0	161.595	0	161
Serial	nlist=128	m=16	nprobe=8	L=500	0.917	1101	18	34	0	659.07	0	0	0	383.595	0	383
OpenMP	nlist=128	m=16	nprobe=4	L=100	0.7475	2162	19	40	1015	205.517	0	66	1000	211.308	10	0
OpenMP	nlist=128	m=16	nprobe=4	L=200	0.826	2454	19	42	1063	232.017	0	142	1157	308.217	16	0
OpenMP	nlist=128	m=16	nprobe=4	L=500	0.869	2708	20	41	982	218.104	0	362	1271	422.055	15	0
OpenMP	nlist=128	m=16	nprobe=8	L=100	0.705	2701	21	40	1288	251.173	0	95	1231	254.217	11	0
OpenMP	nlist=128	m=16	nprobe=8	L=200	0.822	1896	19	40	831	191.017	0	196	786	210.459	12	0
OpenMP	nlist=128	m=16	nprobe=8	L=500	0.917	3478	20	44	1273	257.396	0	589	1522	522.22	15	0
pthread	nlist=128	m=16	nprobe=4	L=100	0.7475	4668	10	30	2174	36.4438	0	48	2373	1.94611	5	0
pthread	nlist=128	m=16	nprobe=4	L=200	0.826	4277	10	30	2064	39.3537	0	53	2078	3.40639	6	0
pthread	nlist=128	m=16	nprobe=4	L=500	0.869	4229	9	30	1953	40.0838	0	61	2109	6.51222	7	0
pthread	nlist=128	m=16	nprobe=8	L=100	0.705	4408	11	32	2076	43.1294	0	88	2166	2.09389	5	0
pthread	nlist=128	m=16	nprobe=8	L=200	0.822	4149	10	29	1928	39.7506	0	87	2056	3.13861	5	0
pthread	nlist=128	m=16	nprobe=8	L=500	0.917	5219	11	33	2496	42.5	0	101	2506	7.1575	7	0

表 5: 实验性能数据，时间单位微秒