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 MPI 多进程优化实验报告

姓名：申健强

学号：2313119

专业：计算机科学与技术

指导教师：王刚

2025 年 6 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
- 本地操作系统：WSL[1]
- IDE：Vscode
- 本地编译器：gcc11.4.0、mpicxx
- 多进程：mpi，多线程：openmp

(二) 实验内容概述

1. 问题定义

算法的背景和意义在前两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。

[1]考虑到我们需要进行较多次的实验和调整而在服务器上相对不变，我们大部分情况仍然使用本地环境进行测试，挑选表现最好的算法到服务器上进行测试。

在上上次实验中我们利用 simd 主要实现了距离计算的并行优化，在上次实验中我们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。而在本次实验中我们利用多进程编程 mpi 方法进一步探索索引算法的可能性。

相较于 openmp，MPI 适合分布式环境，可扩展到多节点，拥有更强的可扩展性和内存隔离优势，但通信开销和编程复杂度更高。根据问题规模、硬件环境和团队经验，选择纯 OpenMP、纯 MPI 或混合方案。若需突破单机资源或在集群上运行，就需要多进程 MPI，其优势在于分布式扩展、内存隔离和灵活的通信控制。

(三) 实验代码

与前三次实验相同，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

在上次实验中，我们实现了基于多线程优化的普通 IVF 算法和 IVF+PQ 的算法以及图索引 HNSW 算法, 其算法原理不作赘述，在本次实验我我们将借助 mpi 进行进一步的探索

(一) IVF+PQ 勘误与补充

在上次实验中，我们粗浅地认为“IVF+PQ”即“先用 IVF 粗排，再对簇内向量做 PQ 量化”。实际上，PQ 和 IVF 的组合顺序也会显著影响量化误差和查询效率。因此，本次实验中，我们补充并实现了两种不同路径，并对比它们在同一数据集、同样参数下的表现。

- **PQ → IVF（先 PQ，后 IVF）**
  1. **全局 PQ 码本训练**：在全集上将 d 维空间等分为 M 个子空间，对每个子空间使用 K-means++ 训练全局质心，共得到 $M \times K$ 个中心点，形成一套共享的 PQ 码本。
  2. **IVF 聚类指派**：仍以原始浮点向量为输入，使用 k-means++ 在全集上聚类得到 n_{list} 个粗簇中心，建立倒排文件（Inverted File）。
  3. **数据写入**：每个向量既记录其所属簇 ID，又计算并存储其 M 字节的全局 PQ 码。
  4. **查询阶段**
     (a) **粗排（IVF）**：对查询向量 q 与所有簇中心 c_i 计算标准化内积距离
     $$d_{\text{coarse}}(q, c_i) = 1 - \langle q, c_i \rangle ,$$
     并选出前 n_{probe} 个簇。
     (b) **距离表生成**：仅对 q 计算一次全局 PQ 的 $M \times K$ 子空间距离表。
     (c) **精排（PQ）**：遍历所选簇的倒排文件，对每条 PQ 码查表累加 M 段子距离，维护一个大小为 ef 的小顶堆，保留候选。
- **IVF → PQ（先 IVF，后 PQ）**
  1. **IVF 聚类指派**：同上，对全集向量聚成 n_{list} 个粗簇，并建立倒排文件。
  2. **簇内 PQ 码本训练**：在每个簇内，仍将高维向量拆分为 M 个子空间，并对每个子空间独立运行 K-means++，得到 $M \times K$ 个本地质心，形成该簇专属的 PQ 码本。

3. **数据写入**：每个簇内向量使用其对应簇的局部 PQ 码本计算并存储 PQ 码。
4. **查询阶段**
   (a) 粗排同上，选出 n_{probe} 个簇。
   (b) 对于每个选中簇，使用该簇的本地 PQ 码本生成子空间距离表（共 n_{probe} 次）。
   (c) 在各簇倒排文件中查表累加子距离，合并所有簇的候选至大小为 ef 的堆。
   (d) 对堆中前 k 个候选做一次原始浮点向量的精确内积重排，以提高 Top-k 准确率。

(二) IVF+HNSW 混合索引

在该方案中，我们将图索引 HNSW 嵌入到 IVF 的“精排”阶段，以提升检索精度和速度：

1. **IVF 构建**：对全集先执行 n_{list} 聚类并建立倒排列表。
2. **簇内 HNSW 构建**：对每个倒排簇中包含的向量单独构建一座 HNSW 图，图节点表示向量，图层与长边保证高效跳跃搜索。
3. **MPI 跨簇并行**：查询时，主进程将 n_{probe} 个簇 ID 分发给不同 MPI 进程，各进程并行执行 HNSW 搜索。
4. **多线程簇内并行**：若单个 HNSW 查询实现多线程（如多起点初始化、并行邻居探测），则在每个 MPI 进程内部进一步利用多核加速。
5. **结果合并**：各进程返回局部 Top-k 候选，主进程收集并在全局堆中合并，必要时再做精确重排。

我们绘制如下过程图1来表示算法的大概过程：[2]：

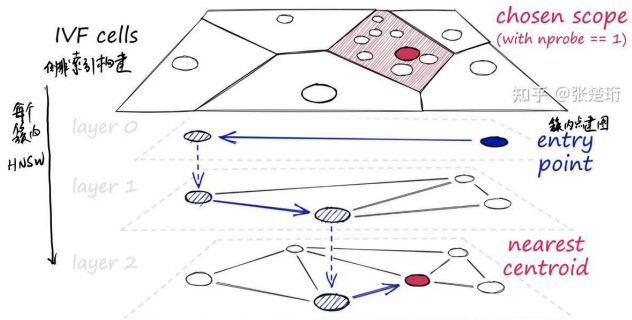


图 1: IVF+HNSW 算法流程示意

(三) IVF 算法的综合并行尝试

启发自计算机组成原理课程的流水线设计，针对纯 IVF 框架，我们设计了如下多级并行与硬件加速方案：

[2]原图来自：https://zhuanlan.zhihu.com/p/595249861, 我们对其进行了改造来表述我们需要的算法过程

1. **MPI 簇间并行**：将 n_{list} 个倒排簇等分到不同进程／节点，实现粗排和倒排查询的分布式执行。
2. **多线程簇内并行**：每个进程内部使用线程池，按批次并行读取倒排列表、解码 PQ、累加距离与堆更新。
3. **SIMD 加速距离计算**：在向量与查询向量的内积距离计算中，利用 AVX/NEON 指令集进行数据对齐与批量计算，提升单核计算带宽。
4. **流水线并行**：将粗排、解码、累加、维护堆四个阶段设计为生产者—消费者流水线，减少线程阻塞与锁竞争。

在这里我们的算法构成了一个四级的流水线（pipeline），将计算与通信交错重叠：

1. **IVF 粗排（主线程，OpenMP 并行）**：SIMD+OpenMP 计算查询向量与所有簇中心的内积，选出 n_{probe} 个簇；
2. **簇内异步搜索（工作线程）**：为每个目标簇启动异步线程，使用 SIMD 批量计算簇内向量与查询的内积，将结果写入对应的 `AsyncSearchResult` 并标记 `ready`；
3. **本地候选收集（主线程）**：轮询各 `AsyncSearchResult` 的 `ready` 标志，边到达边收集，将本地 $(\text{score}, \text{id})$ 打包为紧凑数组，为 MPI 通信做准备；
4. **非阻塞 MPI 汇总（主线程）**：发起 `MPI_Iallgather` 和 `MPI_Iallgatherv`，在通信进行时可继续处理已到达的候选，最后合并并去重，再做全局 Top-k 排序。

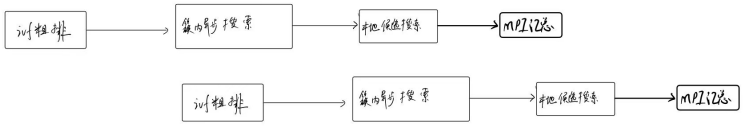


图 2: 流水线流程手绘

（四） 划分 P 块 HNSW 算法的 MPI 并行

此方法直接将数据拆分为 P 个独立分区，并在每个分区上构建 HNSW：

1. **数据分块**：将全集随机或基于启发式（如聚类、哈希划分）分为 P 个部分，每部分大小约为 N/P。
2. **并行构建**：启动 P 个 MPI 进程，各自在其分区上独立构建 HNSW 索引。
3. **并行查询**：查询时，各进程在本地 HNSW 上并行搜索 Top-k，并将结果与距离返回主进程。
4. **全局合并**：主进程在收到 P 份候选后维护全局小顶堆，得到最终的 Top-k 结果。

三、 实验算法实现

相比 OpenMP 的共享内存模型，MPI 属于分布式内存编程，使用显式消息传递进行进程间通信，因此在编写 MPI 程序时需特别注意数据的划分与分布、消息配对的正确性、通信顺序和死锁风险、进程同步以及通信开销优化（如合并小消息、使用非阻塞通信、合理利用集体操作）。我们在实现中需要格外注意这一点。

(一) IVF 算法 MPI 并行迁移

利用 MPI 我们将 IVF 算法从单独的多线程迁移到多进程：

1. **系统初始化与聚类中心广播**：构造函数中获取 MPI 进程编号（`rank`）与总进程数（`size`），并预分配倒排列表结构。通过 `load_centroids` 从 Rank 0 加载训练好的聚类中心，并使用 `MPI_Bcast` 广播到所有节点。

2. **索引构建阶段（Build Phase）**：调用 `build_index()` 对所有向量进行聚类与索引构建：

   - 使用 **数据划分策略**，按行将原始向量均匀划分到各进程处理；
   - 每个进程内部使用 OpenMP 多线程遍历本地数据，计算其到所有聚类中心的 L2 距离，并归入最邻近簇；
   - 为避免线程冲突，每个线程向本地簇结构中使用 `#pragma omp critical` 添加向量；
   - 所有本地倒排结构通过一次性打包和 `MPI_Allgatherv` 实现高效聚合，包括：
     - 所有向量 ID 和数据构建为扁平化数组；
     - 使用每个进程对每个簇的偏移量重建全局结构；

   最终，全体进程将协作完成所有簇的向量重建和同步，构建出统一的分布式索引结构。

3. **并行搜索阶段（Search Phase）**：使用 `mpi_search()` 函数实现完整的近似查询过程：

   - **簇选择**：首先通过 OpenMP 并行计算查询向量到所有簇中心的距离，并选出 Top-n_{probe} 个候选簇；
   - **簇划分与多线程搜索**：候选簇按进程号轮流分配到各进程，各进程内部使用线程池对所负责簇执行内积计算，获得局部候选；
   - **本地合并与裁剪**：线程级候选结果合并为进程级候选集，并进行 Top-$2k$ 的预筛选；
   - **跨进程合并**:使用 `MPI_Allgatherv` 汇总所有候选 $(\texttt{dist}, \texttt{id})$ 对,并通过 `unordered_set` 去重；
   - **最终排序**：按相似度排序后，选出最终 Top-k 结果，返回至查询调用方。

(二) PQ→IVF（先 PQ，后 IVF）

该方法在**全局**范围训练一套 PQ 码本后，再进行 IVF 聚类和倒排存储。流程如下：

1. 启动 P 个 MPI 进程（P 可等于节点数或按负载分配簇数），每个进程加载完整的 PQ 码本与聚类中心列表；

2. 各进程并行对簇中心进行距离计算并选出目标簇；

3. 进程内部生成一次 PQ 距离查表，并并行遍历自己负责的簇中的倒排文件，累加每段子空间距离；

4. 本地进程用小顶堆保留前 ef 个候选，最后所有进程将候选发送至主进程合并并返回 Top-k。

Algorithm 1 构建：PQ → IVF

Input: 原始向量集合 $\mathcal{X}$, 子空间数 M, 每子空间质心数 K, 聚类数 n_{list}
Output: 全局 PQ 码本、聚类中心列表、倒排表

// 全局 PQ 码本训练
for $m = 1$ to M **do**
    将全集向量投影到第 m 个子空间，并运行 K-means++ 得到该子空间的所有质心
end for
// IVF 聚类指派
在全集上运行 k-means++ 得到所有粗簇中心
for 每个向量 $x \in \mathcal{X}$ **do**
    找到该向量最近的簇中心索引
    计算该向量在全局 PQ 码本下的编码（各子空间对应最近质心的索引）
    将编码和向量 ID 插入对应簇的倒排列表
end for

Algorithm 2 查询：PQ → IVF（MPI 并行）

Input: 查询向量 q, PQ 码本、聚类中心、倒排表, 参数 n_{probe}, ef
Output: Top-k 近邻列表

// 粗排：并行计算查询向量与所有簇中心的距离，并选出最靠近的 n_{probe} 个簇
// 查表：生成一次 PQ 距离查表
// 精排：每个进程并行遍历自己负责的簇中的所有 PQ 码，累加对应子空间距离，维护本地小顶堆
// 合并：各进程将本地堆结果发送至主进程，由主进程合并为最终 Top-k

(三) IVF→PQ（先 IVF，后 PQ）

1. 启动 P 个 MPI 进程，将所有粗簇平均分配到各进程；

2. 每个进程加载自己负责簇的局部 PQ 码本；

3. 查询时在本进程内完成：

   - 对所有簇中心并行计算距离并选出目标簇；
   - 为每个目标簇生成对应的 PQ 距离查表；
   - 遍历倒排列表，累加子空间距离，维护本地小顶堆；

4. 最后将本地堆结果发送主进程合并，并可选地对最终候选做一次原始向量内积重排。

Algorithm 3 构建：IVF → PQ

Input: 原始向量集合、子空间数、质心数、簇数

Output: 每簇的局部 PQ 码本、聚类中心列表、倒排表

在全集上运行 k-means++ 得到所有粗簇中心，并分配向量到各簇
for 每个簇 **do**
    在该簇内分别对每个子空间运行 K-means++ 得到本地质心
    用本地质心生成所有向量的 PQ 编码，并插入倒排表
end for

Algorithm 4 查询：IVF → PQ（MPI 并行）

Input: 查询向量、聚类中心、各簇局部 PQ 码本、倒排表, 参数 n_{probe}, ef

Output: Top-k 近邻列表

// 本进程：并行计算与自身负责的簇中心的距离并选出目标簇
// 本进程：为每个目标簇生成 PQ 距离查表
// 本进程：遍历本簇倒排列表，累加距离，维护本地小顶堆
// 合并：各进程将堆结果发送主进程，由主进程合并并可选地做重排

(四) IVF+HNSW 混合索引

该方案在簇内使用 HNSW 图进行精排，多级并行逻辑如下：

- MPI 簇间并行：启动 P 进程，划分目标簇到各进程；
- 进程内部多线程并行：在各自簇的 HNSW 上并行图搜索；
- 主进程合并：收集所有进程返回的候选 Top-k 并合并。

Algorithm 5 构建：IVF + HNSW

Input: 原始向量集合、聚类数

Output: 聚类中心列表、各簇倒排列表、各簇 HNSW 索引

在全集上聚类并分配向量到簇，填充倒排列表
for 每个簇 **do**
    在该簇向量上构建 HNSW 图
end for

Algorithm 6 查询：IVF + HNSW（MPI + 线程）

Input: 查询向量、聚类中心列表、各簇 HNSW 索引, 参数 n_{probe}, k

Output: Top-k 近邻列表

并行计算与所有簇中心的距离并选出目标簇集合
MPI 将目标簇分配给各进程
for all 进程并行 **do**
    多线程并行在各自簇的 HNSW 上执行图搜索，得到本地 Top-k
    将本地 Top-k 发送主进程
end for
主进程合并所有本地结果，输出最终 Top-k

（五） IVF 算法的综合并行尝试

按簇间 MPI + 簇内多线程 + SIMD 三层并行：

1. MPI 将目标簇分派至各进程；
2. 各进程线程池流水线执行：读取倒排列表、批量计算向量距离（SIMD）、更新本地堆；
3. 所有进程将本地堆发送主进程合并，输出最终 Top-k。

Algorithm 7 查询：MPI + 多线程 + SIMD 加速 IVF

Input: 查询向量、聚类中心、倒排表, 参数 n_{probe}, ef
Output: Top-k 近邻列表

MPI 并行分派目标簇到各进程
for all 进程并行 **do**
    使用线程池流水线并行地：
        1. 读取倒排列表分块
        2. 批量向量距离计算（SIMD 加速）
        3. 更新本地小顶堆
    将本地堆结果发送主进程
end for
主进程合并所有进程结果，输出最终 Top-k

（六） 划分 P 块 HNSW 算法的 MPI 并行

每个进程只对一块数据进行 HNSW 构建与查询：

1. 数据按随机或启发式分为 P 块；
2. 启动 P 进程，独立构建各自块上的 HNSW；
3. 查询时，各进程并行在本地 HNSW 上执行搜索并返回 Top-k；
4. 主进程合并并输出最终 Top-k。

Algorithm 8 构建：P 块 HNSW

Input: 原始向量集合、分块数 P
Output: 每块 HNSW 索引

将向量集合分为 P 块
for all 块并行 MPI **do**
    在该块向量上构建 HNSW 索引
end for

Algorithm 9 查询：P 块 HNSW MPI 并行

Input: 查询向量、各块 HNSW 索引, 参数 k

Output: Top-k 近邻列表

1: **for all** 进程并行 **do**
2: 在本地 HNSW 上执行搜索，得到本地 Top-k
3: 将本地结果发送主进程
4: **end for**
5: 主进程合并所有本地结果，输出最终 Top-k

四、 实验设计

(一) 数据集处理与评估指标

本实验使用 DEEP100K 数据集，实现并测试不同 MPI 并行优化方法在 ANN 中的性能表现。以下是详细的实验设计与实现：

1. **数据集分割**
   - 使用 DEEP100K 数据集（100,000 个 96 维向量）。
   - 从 query 文件中选取前 2000 条作为查询集；base 文件作为基础数据集。
   - 使用预先计算好的精确 K 近邻（GT）数据作为基准。
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。

(二) 预先计算所需要的索引

在正式实验前，需要先上完成以下索引构建并保存或将索引构建的部分放在计时框架之外：

- **IVFPQ**：训练全局 PQ 码本，执行 IVF 聚类及倒排列表构建。
- **PQ→IVF**：同上，但按全局 PQ → IVF 路径构建并存储。
- **IVF→PQ**：IVF 聚类后在每个簇内训练局部 PQ 码本，并构建簇内倒排列表。
- **IVF+HNSW**：在每个 IVF 簇内构建 HNSW 图索引。
- **分片 HNSW**：将数据集划分为 P 块（与 MPI 进程数一致），分别构建每块 HNSW。

(三) 实验流程与参数设置

1. **暖机与试运行**每种方法先对前 100 条查询执行暖机，以预热缓存与索引结构，不纳入正式统计。
2. **正式测试**
   (a) 对每种方法、每组参数组合（P, n_{probe}, ef）进行独立测试。
   (b) 每组配置重复运行 $R = 5$ 次，记录并统计：

- 平均查询延迟 μ_{latency} 与标准差 σ_{latency}。
- 平均召回率 $\mu_{\text{recall@10}}$ 与标准差 $\sigma_{\text{recall@10}}$。

(c) 单次测试流程：

i. 记录测试开始时间；

ii. 对 2000 条查询逐条执行 MPI 分发 + 本地并行检索；

iii. 计时并计算 recall@10；

iv. 记录测试结束时间，并将结果追加到日志文件。

(四) 调用测试框架进行测试

注意由于硬件需要暖机，我们在正式统计前进行暖机；正式测试时，采用多次重复取平均值的方法以消除偶发抖动。

五、 结果分析

(一) MPI 并行化 IVF 结果分析

我们以 1 进程与 2 进程实现的 IVF 算法进行实验结果分析，其实验结果大致如下图3所示，可以发现相比于纯 openmp 多线程，MPI 多线程对朴素 IVF 算法存在算法速率上较为清晰的优化，而参数表现上，二者表现类似，所以不做赘述。

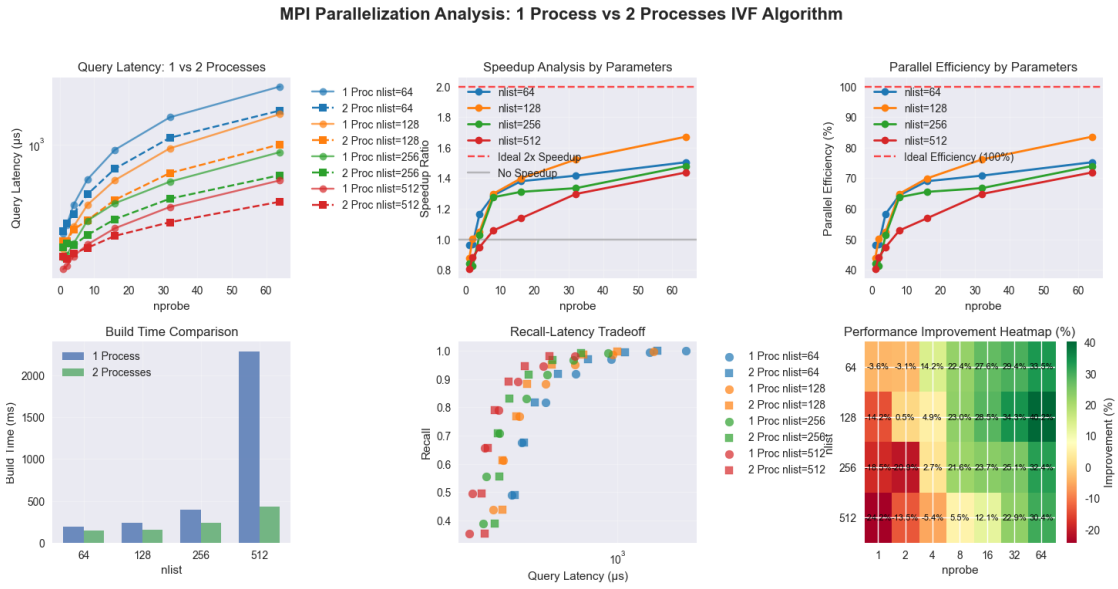


图 3: MPI 并行化 IVF 结果

(二) PQ 与 IVF 混合的两种方法对比分析

我们以 4 进程带重排序的固定参数的结果为例进行分析，我们可以得到如下结果如图4所示：通过上面的实验，我们可以对比分析如下：

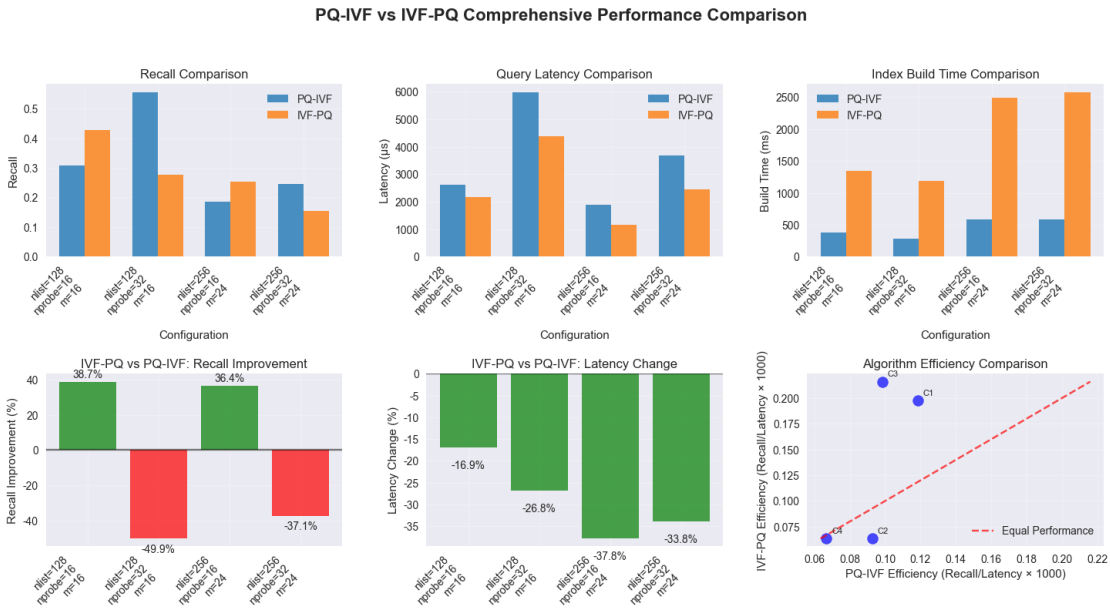


图 4: 两种策略的对比

1. IVF-PQ 混合策略对比分析

IVF 与 PQ 的混合使用主要存在两种策略，其核心区别在于**码本训练的先后顺序、码本的作用域（全局 vs. 簇内）**以及由此带来的**编码精度、存储开销**和**查询效率**上的差异。下面分别从几个维度进行分析：

1. **码本训练顺序与作用域**

   - **PQ→IVF（先 PQ 后 IVF）**：在全量数据上一次性训练出一个全局 PQ 码本（$M \times K$ 个中心），随后执行 k-means 聚类构建 n_{list} 个粗簇中心，将原始向量落入最近簇中，并使用全局 PQ 对所有数据统一编码。
   - **IVF→PQ（先 IVF 后 PQ）**：先对原始数据进行粗聚类得到 n_{list} 个簇中心，随后在每个簇内独立训练一套 PQ 码本，最终使用该簇的局部 PQ 对向量进行编码，共有 n_{list} 套码本。

2. **量化误差与编码精度**

   - **全局 PQ（PQ→IVF）**：训练和存储开销较小，但由于全局码本对所有向量平均建模，不能适配簇内分布差异，量化误差相对较大。
   - **局部 PQ（IVF→PQ）**：每个簇独立训练码本，能够更好拟合局部结构，具有更低的量化误差，但训练时间和存储需求显著提升。

3. **存储结构与内存开销**

   - **PQ→IVF**:仅需存储一套全局 PQ 码本及簇中心,内存总开销为 `sizeof(cluster centers)`+ $M \cdot K \cdot d$。
   - **IVF→PQ**：需要存储 n_{list} 套 PQ 码本，内存开销为 `sizeof(cluster centers)` + $n_{\text{list}} \cdot M \cdot K \cdot d$，当 n_{list} 较大时负担显著。

4. **查询流程与效率**

- 两者均采用 coarse-to-fine 策略：先通过粗排选出 n_{probe} 个候选簇，再在簇内执行 PQ 解码并查表匹配。
- **PQ→IVF**：使用一套全局查表结构，查询时仅需生成一次距离查找表，计算开销较小。
- **IVF→PQ**：每访问一个簇需使用其专属 PQ 码本，需多次生成查找表（n_{probe} 次），计算略慢但准确率更高。

5. **重排（Re-ranking）策略**

- 两者都可对 ef 个候选执行精确点积重排获取最终 Top-k；
- IVF→PQ 因为初选召回更准，重排收益更大。

综上，我们总结两者的区别如下表1

表 1: PQ→IVF 与 IVF→PQ 对比

特性	**PQ→IVF**	**IVF→PQ**
码本训练方式	全局一次性训练	每簇独立训练（n_{list} 次）
量化误差	较大	较小
存储开销	较低（1 套码本）	较高（n_{list} 套码本）
查询查表次数	1 次	n_{probe} 次
查询速度	快	略慢
检索准确率	一般	更高
训练时间	较短	较长

(三) IVF+HNSW 混合方式结果分析

我们以 4 进程环境下的 IVF+HNSW 混合索引为例, 从算法两个阶段——IVF 粗筛与 HNSW 精排——分别讨论它们对最终召回率和查询延迟的影响，借助参数的变化，我们可以分析两种混合的算法的作用，下图5是其直观化表示。

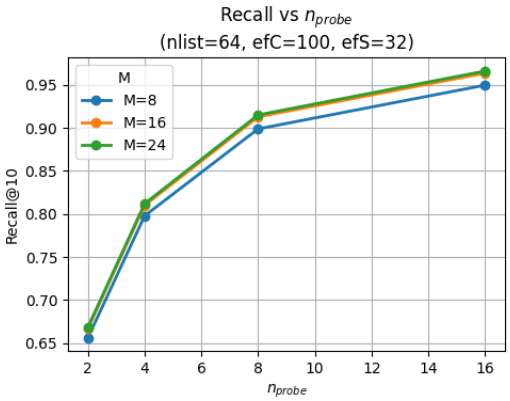
1. IVF 粗筛阶段的作用 IVF 将数据分为 $n_{\text{list}} = 64$ 个簇，每次查询仅探测 n_{probe} 个最相近簇，因而将 HNSW 搜索空间从全量 100k 向量缩减到约

$$\frac{n_{\text{probe}}}{n_{\text{list}}} \times 100\,000$$

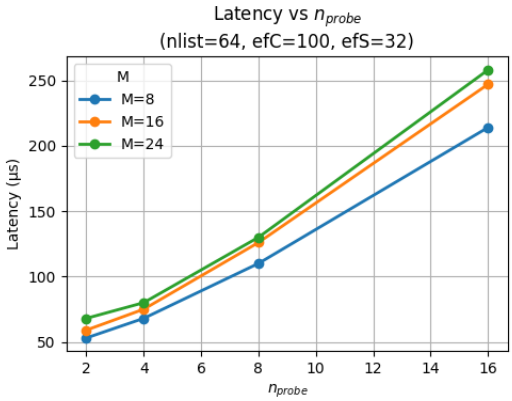
大幅降低后续图搜索成本。实验中随着 n_{probe} 从 2 增加到 16，查询延迟由 ∼50us 缓慢上升至 ∼200us，而召回率则从 0.65 快速跃升至 0.95 以上。可见，IVF 粗筛对召回率的提升具有“断崖式”效果——当 n_{probe} 较小时，大量近邻落在未探测簇中；当 $n_{\text{probe}} \gtrsim 8$ 时，主要近邻均被包含，召回迅速逼近 0.9。

2. HNSW 精排阶段的作用 在每个被选中的粗簇内部，HNSW 用 beam search（搜索深度参数 $efSearch$）对局部候选作精细排序。实验数据显示：

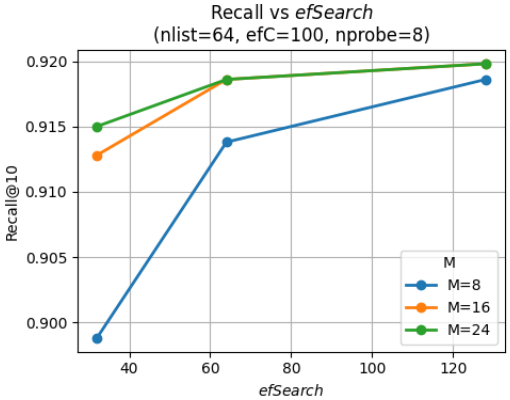
- 当 $efSearch$ 从 32 提升到 128 时，召回率在低 n_{probe} 情况下有明显补漏效果，可再提升约 2–3%；



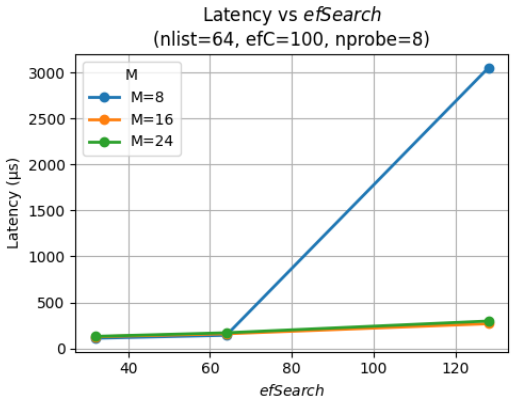
(a) 召回率 vs. n_{probe}（$efSearch = 32$）



(b) 延迟 vs. n_{probe}（$efSearch = 32$）



(c) 召回率 vs. $efSearch$（$n_{\text{probe}} = 8$）



(d) 延迟 vs. $efSearch$（$n_{\text{probe}} = 8$）

图 5: IVF+HNSW 混合索引不同参数下的性能折线图。(a)(b) 中展示了随着 n_{probe} 增大，召回率的“断崖式”提升与延迟的线性增长；(c)(d) 则展示了在中等粗筛强度下，HNSW 精排深度 $efSearch$ 对召回率补漏与延迟代价的折中。

- 进一步增大至 256，则召回提升边际递减，而延迟却呈近线性增长。

这说明：在 IV F 已完成大部分近邻筛选后，HNSW 主要负责“补漏”——在少量簇内深入搜索，提高整体召回率。

3. 整体权衡与参数调整 混合方式将两者优势结合：IVF 快速排除绝大多数远邻，HNSW 在小范围内做高质量搜索。基于上述实验结果，我们给出以下参数推荐：

- **低延迟场景**（latency $< 100\,\mu$s）：$n_{\text{probe}} = 4$–8，$efSearch = 32$，可达召回 0.80–0.92；
- **高召回场景**（recall ≥ 0.95）：$n_{\text{probe}} = 8$–16，$efSearch = 64$–128，召回可达 0.95–0.97，延迟约 120–300us；
- **极致召回**（recall ≥ 0.98）：需 $n_{\text{probe}} \geq 16$ 且 $efSearch \geq 128$，延迟 300–500us，可根据资源预算选择最优组合。

综上所述，我们可以知道，IVF 阶段决定了召回的“下限”——通过调整 n_{probe} 快速控制召回率门槛；HNSW 阶段则负责在这一基础上做“微调”，进一步提升召回率但带来额外延迟。混

合索引通过将粗排与精排有机结合，能够在召回与延迟之间实现更优折中，特别适合需要在百微秒级内实现高召回的在线检索场景。

（四） 流水线多级并行的 IVF 算法尝试结果分析

我们以两线程下多级并行 IVF 算法为例进行结果分析，为控制变量，我们将其与普通 IVF 的 MPI 并行在同样的参数下进行对比。

两种算法均采用相同的硬件配置：2 个 MPI 进程，每个进程 4 个 OpenMP 线程，使用 AVX2 SIMD 指令集加速。测试参数包括不同的聚类数量（nlist：64、128、256、512）和探查聚类数（nprobe：1、2、4、8、16、32、64），以评估算法在不同负载下的性能表现，来验证我们的尝试是否有效，两种算法的对比分析图如6

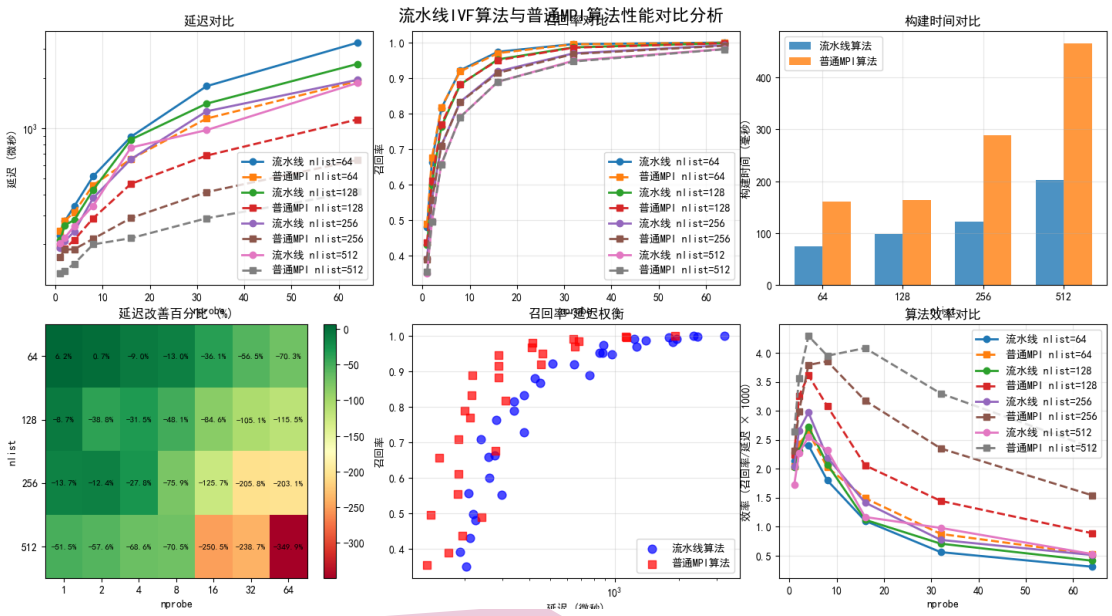


图 6: 流水线多级并行与朴素 IVF 的 MPI 算法实现对比

1. 查询延迟性能分析

实验数据显示，流水线算法在查询延迟方面面临意外的问题：

- 平均延迟表现：流水线算法延迟平均比普通 MPI 算法高 84.35%；
- 最好情况：仅在少数配置下有 6.25% 的延迟改善；
- 最差情况：部分配置下延迟增加高达 **349.88%**。

延迟增加原因分析：流水线算法延迟增加的主要原因包括：

1. **线程管理开销：**多线程异步任务调度引入了额外的线程创建、同步和销毁开销，在计算量相对较小的场景下，这些开销可能超过并行计算带来的收益；
2. **内存竞争加剧：**多个工作线程同时访问内存和缓存，可能导致缓存失效和内存带宽竞争，特别是在 SIMD 批量处理时；

3. **流水线阶段不平衡：**四级流水线的各个阶段处理时间可能不均衡，导致某些阶段成为瓶颈，整体性能受限于最慢的阶段；

4. **MPI 通信开销：**非阻塞 MPI 通信虽然理论上可以与计算重叠，但在实际环境中可能由于网络延迟、进程同步等因素导致通信开销增加。

2. 召回率保持分析

通过对比分析，流水线算法在召回率方面与普通 MPI 算法基本保持一致，证明了算法优化并未损失精度：

- 两种算法的召回率曲线几乎重合，差异在 0.1% 以内；
- 在相同参数配置下，流水线算法能够达到与基准算法相同的搜索质量；
- 召回率随 nprobe 增加的趋势完全一致，验证了算法正确性。

3. 构建时间性能分析

值得注意的是，索引构建阶段的性能对比显示流水线算法具有明显优势：

表 2: 索引构建时间对比（毫秒）

nlist	流水线算法	普通 MPI 算法	改善百分比
64	74	160	53.8%
128	98	164	40.2%
256	122	288	57.6%
512	202	466	56.7%
平均	**124.0**	**269.5**	**53.99%**

构建时间改善 53.99% 是流水线算法的唯一优势，主要得益于：

1. **并行索引构建：**在索引构建阶段，多线程并行处理向量数据和聚类分配，有效提升了构建速度；

2. **批量数据处理：**SIMD 批量处理在大规模数据构建时效果明显，减少了单次操作的开销；

3. **内存访问模式优化：**构建阶段的顺序访问模式更适合流水线设计，减少了缓存失效；

4. **MPI 通信优化：**在构建阶段，MPI 进程间的数据分发和同步相对简单，通信开销较小。

4. 系统效率综合评估

定义算法效率为召回率与延迟的比值（不妨定义效率 = 召回率/延迟 × 1000，下同），综合评估两种算法的性能：

效率表现总结：

- 流水线算法平均效率：1.5936
- 普通 MPI 算法平均效率：2.5993
- 整体效率下降：38.69%

效率下降可能原因：

1. **并行化开销过大：**在当前数据规模和硬件配置下，多级并行引入的管理开销超过了并行计算带来的收益；
2. **资源竞争严重：**2 个 MPI 进程每个 4 线程的配置可能导致 CPU 核心和内存带宽的过度竞争，特别是在 SIMD 指令密集执行时；
3. **流水线设计不匹配：**四级流水线的设计可能不适合当前的计算任务特征，各阶段负载不均衡导致整体性能受限；
4. **同步开销增加：**虽然设计为异步执行，但实际运行中线程间的同步和数据传递开销可能比预期更高。

5. 扩展性分析

流水线算法在扩展性方面暴露出设计上的局限性：

聚类数量扩展性：随着 nlist 从 64 增加到 512，流水线算法的延迟劣势在某些配置下更加明显，说明多级并行在大规模参数下可能面临更严重的资源竞争。

探查深度扩展性：当 nprobe 增加时，流水线算法的性能劣势并未随计算量增加而改善，表明当前的流水线设计在不同负载规模下都存在效率问题。

流水线多级并行 IVF 算法在理论设计上具有一定先进性，但实验结果表明在具体实施中需要更加细致的参数调优和硬件适配，才能真正发挥其性能潜力。但本部分由于临近期末考试的进行未能深化探究，在后续的期末报告中我们将深入解决这个问题找出其中真实的原因。

（五） 划分 P 块 HNSW 算法结果分析

在这个方案中，我们采用了四种分块策略进行尝试，分别是随机分块、轮询分块 (即取模分块)、哈希分块和 KMEANS 分块将数据分为同样数量的块（这里是 4）到进程中进行测试，将测试序列中的向量分到不同的进程中进行 HNSW 查询，其中随机分块与轮询分块不考虑向量内部情况，哈希与 KMEANS 尝试利用向量中的一部分信息让每一块中的向量分布更有规律性从而加速查询。

我们得到在不同参数条件下暖机后的实验结果数据并对其进行分析，用以下可视化方式表示实验结果图7展示了在不同分片方法下的性能表现，可以看到由于数据分布由于其一般性，我们对其进行的人为聚类反而造成了一定的性能损失，但相比于不分片的情况性能有所提升。

而参数表现上，我们固定分片方式等，发现在参数上的表现分片 HNSW 算法与普通 HNSW 表现类似如图8所示，故不做赘述。

（六） 进程数对实验结果的影响分析

我们以朴素 IVF 算法（不含 PQ 与 HNSW）在不同进程下的并行表现为例分析多进程进程数对算法效率的影响。实验采用 1 进程、2 进程和 4 进程三种配置，每种配置下均使用 4 个 OpenMP 线程，在相同的硬件环境和数据集上进行对比测试。

整体性能表现分析 实验结果显示，不同进程数配置下的性能表现存在显著差异：

整体如下图9所示：

从整体性能表现可以看出：

- **2 进程配置表现最优：**在查询延迟、构建时间和算法效率三个关键指标上均优于其他配置；

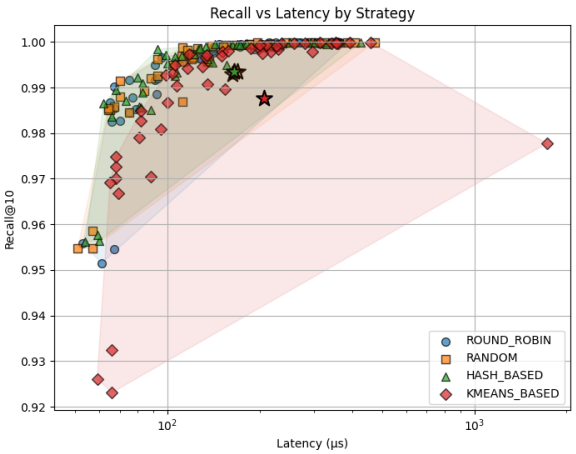


图 7: 不同分片方法的性能

表 3: 不同 MPI 进程数配置的性能对比

进程数	平均延迟 (s)	平均构建时间 (ms)	平均召回率	算法效率
1	532.1	779.8	0.7849	1.4749
2	399.0	241.5	0.7849	1.9669
4	640.9	560.8	0.7849	1.2246

- **4 进程配置性能下降：**相比单进程配置，4 进程在查询延迟和算法效率上都出现了一定倒退；
- **召回率保持稳定：**所有配置下的召回率完全一致，验证了算法正确性。

扩展性规律总结 通过对不同进程数配置的分析，我们发现了 IVF 算法 MPI 并行化的扩展性规律:

1. **存在最优进程数：**并非进程数越多性能越好，需要根据具体的硬件和数据特征确定最优配置；
2. **构建与查询阶段特性不同：**构建阶段更适合并行化，查询阶段对通信延迟更敏感；
3. **数据规模影响扩展性：**不同的 nlist 设置下，最优进程数可能有所不同；
4. **硬件架构制约：**当前的硬件配置下，2 进程是通信开销与并行收益的最佳平衡点。

在本部分我们发现了并行度并非越高越好的情况，在此由于上述类似的的原因我们将在期末对其进行细化分析。

六、 总结

本次实验基于 MPI 分布式并行编程框架，针对 ANN 搜索问题设计并实现了多种优化方案，包括 IVF 算法的 MPI 并行迁移、PQ 与 IVF 的两种组合方式（PQ→IVF 和 IVF→PQ）、IVF+HNSW 混合索引、流水线多级并行 IVF 以及分片 HNSW 算法。通过在 DEEP100K 数据

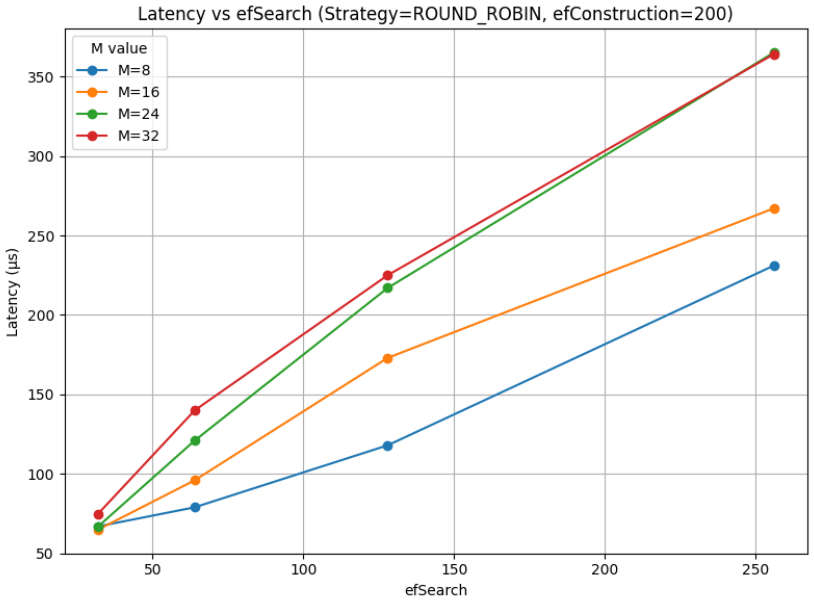


图 8: 参数表现

集上的测试，我们系统地评估了这些方案在查询延迟、召回率、索引构建时间及扩展性等方面的表现：

1. **IVF 算法的 MPI 并行化**：通过簇间 MPI 并行与簇内多线程结合，成功将 IVF 算法扩展到分布式环境。实验表明，2 进程配置在查询延迟（399.0 μs）和构建时间（241.5 ms）上表现最优，优于 1 进程和 4 进程配置，揭示了并行度并非越高越好的规律，需根据硬件和数据规模选择最优进程数。

2. **PQ 与 IVF 组合策略**：PQ→IVF 和 IVF→PQ 两种方式在量化误差、存储开销和查询效率上存在显著差异。PQ→IVF 因全局码本训练具有较低的存储和计算开销，适合低延迟场景；IVF→PQ 因簇内码本更精准，召回率更高但查询稍慢，适合高精度需求场景。两者的权衡为后续优化提供了重要参考。

3. **IVF+HNSW 混合索引**：通过将 IVF 的粗筛与 HNSW 的精排结合，该方案在召回率和延迟间取得了良好折中。实验表明，$n_{\text{probe}} = 8$ 至 16、$efSearch = 64$ 至 128 可在 120–300 μs 内实现 0.95 以上召回率，适合高召回在线检索场景。

4. **流水线多级并行 IVF**：尽管设计上引入了四级流水线以重叠计算与通信，但在当前硬件配置下，查询延迟平均增加 84.35%，整体效率下降 38.69%，主要因线程管理和内存竞争开销过大。然而，索引构建时间改善 53.99%，显示出流水线在批量处理场景的潜力。未来我们可以进一步尝试优化流水线阶段平衡性和硬件适配。

5. **分片 HNSW 算法**：通过随机、轮询、哈希和 K-means 四种分片策略测试发现，随机和轮询分片性能优于人为聚类分片，表明数据分布的通用性可能削弱聚类分片的加速效果。相比不分片 HNSW，分片方案在多进程下性能有所提升，参数表现与普通 HNSW 类似。

6. **进程数影响**：实验验证了进程数对性能的非线性影响，2 进程配置在查询延迟和构建时间上均优于 1 进程和 4 进程，表明通信开销与并行收益需在特定硬件环境下平衡。构建阶段对并行化更友好，而查询阶段对通信延迟更敏感。

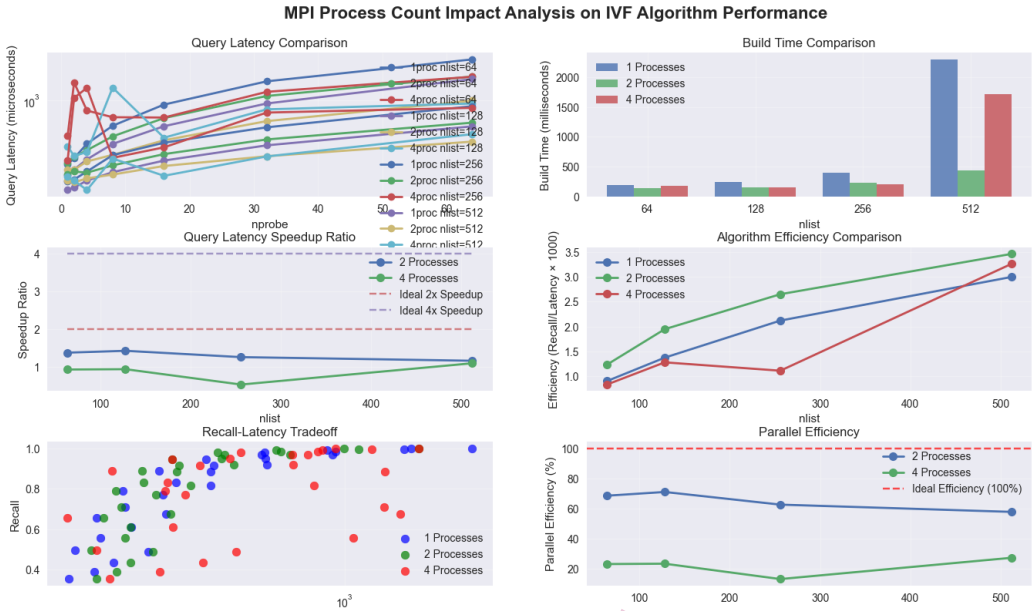


图 9: IVF 算法不同进程数下的性能表现对比

未来改进:

1. 优化流水线算法的阶段负载平衡，减少线程管理和内存竞争开销，可能通过动态任务分配或更细粒度的并行控制实现。
2. 针对分片 HNSW，探索基于数据分布特性的自适应分片方法，结合聚类与空间划分提高分片效率。
3. 在更高性能的集群环境下测试更多进程数（如 8、16 进程），进一步验证算法的扩展性。
4. 引入更高效的通信优化（如异步通信压缩、拓扑优化）以降低 MPI 通信开销。
5. 结合 GPU 或其他硬件加速器，探索异构并行方案以进一步提升查询性能。

综上，本次实验我们通过 MPI 框架深入探索了 ANN 搜索的分布式并行优化，验证了多种方案的可行性与局限性，为后续研究提供了宝贵的经验和数据支持。未来将在算法设计、参数调优和硬件适配上进一步深化，以实现更高效的并行 ANN 搜索。[3]

[3]**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，在一些拓展方向的分析上参考了 ai 的分析方案