



南开大学  
Nankai University

计 算 机 学 院  
并行程序设计实验报告

---

MPI

---

张奥喆

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 6 月 27 日

## 摘要

本报告系统性地研究并实现了针对两种主流近似最近邻搜索算法——IVF 和 HNSW——的并行优化策略。核心并行化技术采用消息传递接口 (MPI)，并结合了单指令多数据 (SIMD) 指令集以及多线程 (OpenMP) 进行性能增强与对比分析。

实验首先将 MPI 应用于 IVF 算法，通过多进程间分散倒排列表的搜索任务，有效提升了查询性能。进一步地，通过将 MPI 的分布式并行与 SIMD 的数据级并行相结合，IVF 算法获得了高达 **8.1 倍的最高加速比**。

针对并行化难度更高的 HNSW 算法，报告探索了两种 MPI 并行策略：一种是结合 IVF 的混合索引方法 (IVF-HNSW)，通过 MPI 并行处理不同的数据簇，获得了约 1.16 倍的加速比；另一种是分区 HNSW 策略，该策略将数据集直接切分，在各分区上用 MPI 并行搜索。实验结果表明，分区 HNSW 策略不仅简化了索引构建，更在高召回率 ( $>0.95$ ) 场景下展现出比其他方法更强的性能和稳定性，**最高加速比可达 1.75 倍**。

实验也通过 VTune 等工具深入分析了不同并行模型的性能瓶颈，指出通信开销、负载均衡和缓存效率是影响加速比的关键因素。分析发现，不存在普适的“最佳”并行策略，最优解深度依赖于算法特性、硬件架构和性能目标。设计与并行模式协同的算法是实现可扩展、高性能计算的关键。

**关键字：**并行计算；MPI；ANN；IVF；HNSW

## 目录

一、 IVF	1
(一) MPI 优化	1
1. 原理	1
2. 关键代码	1
3. 并行加速理论分析	4
(二) MPI+OpenMP	5
1. 实现思路	5
2. 并行加速理论分析	5
(三) MPI+SIMD	6
1. 原理	6
2. 关键代码	6
3. 并行加速分析	7
(四) 实验	7
1. 查询延迟 & 加速比	7
2. recall-latency trade-off	8
3. VTune 分析	9
二、 HNSW	10
(一) IVF+HNSW	11
1. MPI 优化	11
2. MPI+SIMD	12
(二) 分区 HNSW	13
1. 核心思想	13
2. 数据集划分策略	13

3.	基于 MPI 的并行搜索 . . . . .	13
4.	与 IVF-HNSW 的对比优势 . . . . .	13
(三)	实验 . . . . .	13
1.	查询延迟 & 加速比 . . . . .	14
2.	recall-latency trade-off . . . . .	14
<b>三、</b>	<b>总结</b> . . . . .	<b>15</b>
(一)	核心结论 . . . . .	16
(二)	启发 . . . . .	16

## 一、 IVF

在多线程实验中，OpenMP 并行优化的 IVF 效果极好。本次实验计划把 MPI 应用到 IVF 的并行优化中，并尝试与多线程、simd 并行化进行结合。

### (一) MPI 优化

#### 1. 原理

IVF 搜索的核心思想是首先通过 K-Means 等聚类算法对原始数据集进行聚类，得到一组聚类中心和相应的倒排列表。每个倒排列表包含属于该聚类中心的数据点的索引。在查询阶段，首先找到与查询向量最近的  $nprobe$  个聚类中心，然后在这些聚类中心对应的倒排列表中进行穷举搜索，最终找出全局的  $k$  个最近邻。

MPI 优化主要体现在以下几个方面：

1. **倒排列表搜索的负载均衡与并行：**这是 MPI 优化的核心。在获取到  $nprobe$  个最近的聚类中心列表后，这些列表会被均匀地分配给不同的 MPI 进程。具体分配策略是，将第  $p$  个 probe 列表分配给进程  $p \pmod{\text{world\_size}}$ 。每个进程只负责搜索分配给它的倒排列表，从而将计算任务分散到各个处理器上。
2. **结果收集与全局 Top-K：**每个 MPI 进程在自己的倒排列表上执行搜索，并维护一个局部的 Top-K 结果（大顶堆）。搜索完成后，所有进程通过 MPI 的 `MPI_Gather` 操作将各自的局部 Top-K 结果收集到根进程（rank 0）。根进程负责将所有局部结果汇总，并从中选出最终的全局 Top-K 最近邻。

这种并行策略有效地利用了多处理器资源，将原本在一个进程上进行的  $nprobe$  个倒排列表的搜索任务分散开，显著缩短了查询时间。

#### 2. 关键代码

以下是实现 MPI 优化 IVF 搜索的关键代码段及其解释：

##### 1. MPI 环境封装 (ivf\_mpi::Env 和 init/finalize)

```

1 struct Env {
2     int rank = 0;
3     int size = 1;
4     bool we_init = false;
5 } env;
6
7 inline void init(int *argc, char ***argv)
8 {
9     int already;
10    MPI_Initialized(&already);
11    if (!already) {
12        MPI_Init(argc, argv);
13        env.we_init = true;
14    }
15    MPI_Comm_rank(MPI_COMM_WORLD, &env.rank);
16    MPI_Comm_size(MPI_COMM_WORLD, &env.size);
17 }
18
19 inline void finalize()

```

```

20 {
21     int done;
22     MPI_Finalized(&done);
23     if (!done && env.we_init)
24         MPI_Finalize();
25 }
26
27 inline int rank() { return env.rank; }
28 inline int size() { return env.size; }

```

Listing 1: MPI 环境封装

这部分代码提供了对 MPI 环境的轻量级封装。init 函数确保 MPI 环境在第一次调用时被初始化，并获取当前进程的 rank 和总进程数 size。finalize 函数则负责在进程结束时正确关闭 MPI 环境。rank() 和 size() 函数提供了便捷的方式来获取当前进程在通信组中的标识符及其总数。

## 2. 优先级队列到向量的转换 (pq\_to\_vec)

```

1 using DistIdx = std::pair<float,int>;
2
3 inline void pq_to_vec(std::priority_queue<DistIdx> &pq,
4                     std::vector<DistIdx> &vec,
5                     std::size_t need)
6 {
7     vec.clear();
8     while (!pq.empty()) {
9         vec.emplace_back(pq.top());
10        pq.pop();
11    }
12    std::reverse(vec.begin(), vec.end());
13    vec.resize(need, DistIdx(std::numeric_limits<float>::infinity(), -1));
14 }

```

Listing 2: 优先级队列到向量的转换

这是一个辅助函数，用于将 std::priority\_queue（大顶堆）中的元素按距离从近到远转换并填充到 std::vector 中。它首先将堆中的元素依次取出并放入向量，然后反转向量以实现升序排序，最后根据 need 参数调整向量大小，不足时用无限距离和无效索引 (-1) 填充。这保证了每个进程发送的局部结果向量长度一致，便于 MPI\_Gather 操作。

## 3. MPI 版本 IVF 搜索核心逻辑 (ivf\_search\_mpi)

```

1 inline std::priority_queue<DistIdx>
2 ivf_search_mpi(const float *base,
3               const float *query,
4               std::size_t base_number,
5               std::size_t dim,
6               std::size_t k,
7               int nprobe)
8 {
9     // 2.1 先各自找到 query 最近的 nprobe 个 centroids
10    std::vector<DistIdx> cent_dists; cent_dists.reserve(g_ivf_index.nlist);
11    for (int cid = 0; cid < g_ivf_index.nlist; ++cid) {
12        float d = l2_distance(query, g_ivf_index.centroids[cid].data(), dim);

```

```

13     cent_dists.emplace_back(d, cid);
14 }
15 std::partial_sort(cent_dists.begin(),
16                  cent_dists.begin() + std::min(nprobe, g_ivf_index.nlist),
17                  cent_dists.end());
18 // 2.2 按 “probe 序号 % world_size == rank” 把列表平均分给各进程
19 std::priority_queue<DistIdx> local_topk; // 大顶堆
20 for (int p = 0; p < nprobe && p < g_ivf_index.nlist; ++p) {
21     if (p % env.size != env.rank) continue; // 这一 probe 不归我管
22
23     int list_id = cent_dists[p].second;
24     const auto &invlist = g_ivf_index.invlists[list_id];
25
26     for (int idx : invlist) {
27         const float *v = base + std::size_t(idx) * dim;
28         float dist = l2_distance(query, v, static_cast<int>(dim));
29
30         if (local_topk.size() < k) {
31             local_topk.emplace(dist, idx);
32         } else if (dist < local_topk.top().first) {
33             local_topk.pop();
34             local_topk.emplace(dist, idx);
35         }
36     }
37 }

```

Listing 3: MPI 版本 IVF 搜索核心逻辑 - 局部搜索

这部分代码首先计算查询向量与所有聚类中心的距离，并找出最近的  $nprobe$  个。然后，通过  $p \% env.size != env.rank$  的条件，将  $nprobe$  个倒排列表平均分配给各个进程。每个进程只处理分配给自己的倒排列表，并在其中找出局部的 Top-K 结果，存储在 `local_topk` 中。

```

1 // 2.3 把每个进程的 k 个(或不足 k 个)结果 Gather 到 rank=0
2 std::vector<DistIdx> local_vec;
3 pq_to_vec(local_topk, local_vec, k); // 填充到 k 个
4 std::vector<float> send_d(k), recv_d;
5 std::vector<int> send_i(k), recv_i;
6 for (std::size_t t = 0; t < k; ++t) {
7     send_d[t] = local_vec[t].first;
8     send_i[t] = local_vec[t].second;
9 }
10 if (env.rank == 0) {
11     recv_d.resize(k * env.size);
12     recv_i.resize(k * env.size);
13 }
14
15 MPI_Gather(send_d.data(), k, MPI_FLOAT,
16           recv_d.data(), k, MPI_FLOAT,
17           0, MPI_COMM_WORLD);
18
19 MPI_Gather(send_i.data(), k, MPI_INT,
20           recv_i.data(), k, MPI_INT,
21           0, MPI_COMM_WORLD);

```

Listing 4: MPI 版本 IVF 搜索核心逻辑 - 结果收集

在每个进程完成局部搜索后，它们将各自的局部 Top-K 结果转换为固定大小的向量（利用 `pq_to_vec` 填充无效位）。然后，通过两次 `MPI_Gather` 调用，将所有进程的距离和索引数据分别收集到根进程（rank 0）的 `recv_d` 和 `recv_i` 向量中。`MPI_Gather` 确保了数据按照进程的 rank 顺序在接收缓冲区中排列。

```

1 // 2.4 rank-0 汇总 & 取全局 top-k
2 if (env.rank == 0) {
3     std::priority_queue<DistIdx> global_topk;
4
5     for (int r = 0; r < env.size; ++r)
6         for (std::size_t t = 0; t < k; ++t) {
7             float d = recv_d[r*k + t];
8             int i = recv_i[r*k + t];
9             if (i == -1) continue; // 被填充的无效位
10
11             if (global_topk.size() < k) {
12                 global_topk.emplace(d, i);
13             } else if (d < global_topk.top().first) {
14                 global_topk.pop();
15                 global_topk.emplace(d, i);
16             }
17         }
18
19     // 把距离改成负值以兼容原 main.cc 的写法
20     std::priority_queue<DistIdx> ret;
21     while (!global_topk.empty()) {
22         ret.emplace(-global_topk.top().first, global_topk.top().second);
23         global_topk.pop();
24     }
25     return ret;
26 }
27 else {
28     return {}; // 非 0 号进程返回空堆
29 }
30 } // namespace ivf_mpi

```

Listing 5: MPI 版本 IVF 搜索核心逻辑 - 根进程汇总

只有根进程 (rank 0) 会执行这部分代码。它遍历所有收集到的局部 Top-K 结果（来自所有进程），并从中找出最终的全局 Top-K 最近邻，同样使用一个大顶堆 `global_topk` 来维护。

### 3. 并行加速理论分析

总体来说 MPI 优化的思路和多线程类似，经过 IVF 搜索可以显著提升性能，但其加速比受到多种因素的影响：

#### 1. 可并行部分：

- **倒排列表搜索：**这是最主要的并行化部分。当 *nprobe* 足够大，且倒排列表的平均大小适中时，将 *nprobe* 个倒排列表的搜索任务分散到多个进程上，可以实现近乎线性的加速。每个进程独立地计算其负责的倒排列表内的 L2 距离。

#### 2. 串行部分和瓶颈：

- **聚类中心距离计算:** 查询向量与所有聚类中心的距离计算是每个进程独立进行的, 如果  $nlist$  非常大, 这部分会成为串行瓶颈。
- **MPI 通信开销:** MPI\_Gather 操作会引入通信延迟。所有进程都需要将各自的局部结果发送到根进程, 这涉及到数据的序列化、网络传输和根进程的接收缓冲。随着进程数的增加, 通信量也会增加, 可能会抵消一部分计算加速带来的收益。
- **根进程的合并:** 根进程需要将所有局部 Top-K 结果进行合并, 并从中选出最终的全局 Top-K。这个合并过程是串行的, 当  $world\_size \times k$  变得非常大时, 这部分可能会成为新的瓶颈。
- **负载不平衡:** 在 pthread 部分中我们已经讨论过了 IVF 算法容易产生负载不均的情况: 如果  $nprobe$  个倒排列表的大小差异很大, 可能会导致负载不平衡。

## (二) MPI+OpenMP

### 1. 实现思路

MPI 与 OpenMP 的结合旨在利用多节点 (通过 MPI) 和单节点内多核心 (通过 OpenMP) 的计算资源, 以期达到更优的并行加速效果。其核心实现思路如下:

- **簇间并行 (MPI):** 首先, 利用 MPI 实现进程级别的并行。在  $N$  个聚类中心被选出后, 这些中心对应的倒排列表的搜索任务被分配给  $P$  个 MPI 进程。具体分配策略为轮询方式, 即第  $i$  个探针任务分配给  $rank = i \pmod{P}$  的 MPI 进程。每个 MPI 进程独立负责搜索分配给它的那些簇。
- **簇内并行 (OpenMP):** 在每个 MPI 进程内部, 当处理分配给它的某个特定簇的倒排列表时, 利用 OpenMP 指令进行线程级别的并行。具体来说, 遍历该倒排列表中的所有向量, 计算查询向量与这些库向量之间的距离, 这一过程通过 OpenMP 的并行循环 (`#pragma omp parallel for`) 来加速。每个 OpenMP 线程计算一部分向量的距离, 并将结果汇总到共享结果容器中。
- **结果汇总:** 每个 MPI 进程首先通过 OpenMP 线程的协作, 得到其负责的簇中的局部 Top-K 结果。随后, 所有 MPI 进程的局部 Top-K 结果通过 MPI\_Gather 操作汇总到主进程 ( $rank = 0$ )。主进程最后对收集到的所有局部结果进行全局排序, 得到最终的全局 Top-K 结果。

### 2. 并行加速理论分析

理论上, MPI+OpenMP 的混合并行模型有望提供比单一并行策略 (纯 MPI 或纯 OpenMP) 更高的加速比。**两级并行带来的加速:** 假设有  $P$  个 MPI 进程, 每个进程内使用  $T$  个 OpenMP 线程。

理想情况下, 如果任务能够完美划分且无额外开销, 总加速比可以期望达到  $P \times T$ 。但是实际受限于并行开销, 这种理想情况无法达到:

- **过细的并行粒度与高昂的开销:** 对于簇内搜索, 如果倒排列表的平均长度不够大, 使用 OpenMP 进行并行化时, 每个线程分配到的计算任务可能过少。此时, OpenMP 线程创建、管理和同步的开销可能超过并行计算带来的收益。
- **MPI 与 OpenMP 资源竞争:** MPI 进程和 OpenMP 线程可能在 CPU 核心、内存带宽等资源上产生竞争, 导致整体效率下降。不当的线程/进程绑定策略也可能加剧此问题。

- **通信瓶颈加剧:** 引入 OpenMP 后, 如果每个 MPI 进程能更快地完成其计算部分, 可能会更早地到达 MPI\_Gather 点, 从而可能加剧对通信带宽的瞬时需求, 或者使得固有的通信延迟占比更高。

在本次实际测试中, MPI+OpenMP 的组合表现为**负优化**, 在后续的实验中我也会尝试对这一现象进行分析。

### (三) MPI+SIMD

考虑到 OpenMP 的加入对本次实验并没有正面的影响, 因此这一部分尝试将 SIMD 结合到 MPI 中。

#### 1. 原理

MPI 提供的是**分布式内存并行**, 它通过在多个独立的进程间通信来协调任务, 每个进程拥有自己的内存空间, 适用于跨节点或多核处理器间的任务分发。而 SIMD, 如 ARM NEON, 提供的是**数据级并行 (Data-Level Parallelism)**, 它允许单个 CPU 指令同时处理多个数据元素。

在 IVF 搜索中, **L2 距离计算**是内层循环中非常频繁的操作, 对性能影响巨大。一个典型的 L2 距离计算涉及多个浮点数的减法、乘法和累加。这个过程本质上是高度可并行的, 因为每个维度上的操作是独立的。通过使用 SIMD 指令, 可以将这些独立的操作打包, 在一个时钟周期内完成多个维度的计算, 从而显著减少计算 L2 距离所需的 CPU 周期 [1]。

当 MPI 将倒排列表的搜索任务分发到各个进程后, 每个进程在执行分配给自己的搜索任务时, 可以在**局部**利用 SIMD 来加速每个向量对的距离计算。这种结合方式旨在实现**粗粒度并行 (MPI)** 和 **细粒度并行 (SIMD)** 的优势叠加, 提高整体查询效率。

#### 2. 关键代码

结合 SIMD 的主要改动在于用 NEON Intrinsics 实现的 l2\_distance\_neon 函数替换了原有的标量 l2\_distance 函数。

```

1 inline float l2_distance_neon(const float *a, const float *b, std::size_t dim) {
2     float32x4_t sum_vec1 = vdupq_n_f32(0.0f);
3     float32x4_t sum_vec2 = vdupq_n_f32(0.0f);
4     std::size_t i = 0;
5
6     // 一次处理8个浮点数, 使用2个寄存器
7     for (; i + 7 < dim; i += 8) {
8         // Load first 4 elements
9         float32x4_t va1 = vld1q_f32(a + i);
10        float32x4_t vb1 = vld1q_f32(b + i);
11        float32x4_t diff1 = vsubq_f32(va1, vb1);
12        sum_vec1 = vmlaq_f32(sum_vec1, diff1, diff1); // sum_vec1 = sum_vec1 + (diff1
13        * diff1)
14
15        // Load next 4 elements
16        float32x4_t va2 = vld1q_f32(a + i + 4);
17        float32x4_t vb2 = vld1q_f32(b + i + 4);
18        float32x4_t diff2 = vsubq_f32(va2, vb2);
19        sum_vec2 = vmlaq_f32(sum_vec2, diff2, diff2); // sum_vec2 = sum_vec2 + (diff2
20        * diff2)
21    }
22    float32x4_t final_sum = vaddq_f32(sum_vec1, sum_vec2);

```

```

21
22 // 水平求和
23 float32x2_t pair_sum = vpad_f32(vget_low_f32(final_sum), vget_high_f32(final_sum)
24 );
25 float total_sum = vget_lane_f32(vpad_f32(pair_sum, pair_sum), 0);
26 //处理 剩余元素...
27 return total_sum;
28 }

```

Listing 6: NEON 加速的 L2 距离计算

### 3. 并行加速分析

将 SIMD 技术与 MPI 结合，可以在两个不同的并行层面上实现加速：

#### 1. 单核内效率提升 (SIMD)：

- `l2_distance_neon` 函数通过并行处理向量的多个维度，显著减少了计算单个 L2 距离所需的指令数和时间。对于高维数据，这能带来数倍的性能提升。
- 这直接加速了 IVF 搜索中的两个关键热点：
  - 查询向量与 `nlist` 个聚类中心的距离计算。
  - 在选定的 `nprobe` 个倒排列表中，查询向量与每个数据点的距离计算。

#### 2. 多进程间任务分发 (MPI)：

- MPI 继续负责将 `nprobe` 个倒排列表的搜索任务分发给不同的进程，实现计算的负载均衡。
- 每个进程在完成其分配的任务后，局部 Top-K 结果通过 `MPI_Gather` 收集到根进程进行最终汇总。

**协同效应：** MPI 和 SIMD 的结合是互补的。MPI 解决了如何在多个处理器之间分发大规模计算任务的问题，而 SIMD 则解决了单个处理器核心如何更高效地执行向量操作的问题。它们在不同的抽象层次上提供了并行性，并且通常不会相互干扰，能带来复合的性能提升。

## (四) 实验

在多线程实验中，我已经探索出聚类数量 `nlist=1024` 对这个数据集有较好的性能，此外，和之前的实验保持一致便于对比，设置 `recall=0.90`。

#### 1. 查询延迟 & 加速比

图 1 展示了不同索引方法的平均查询延迟及其相对于基准 IVF Flat 的加速比。

- **基准 IVF Flat：** 未进行任何并行优化的 IVF Flat 方法平均查询延迟约为 680 微秒 ( $\mu s$ )，作为后续所有优化的基准。
- **纯 MPI 优化：**
  - 使用 4 个 MPI 进程 (MPI:4) 后，平均查询延迟显著降低至约 300  $\mu s$ ，相对于 IVF Flat 实现了约 2.5 的加速比。这表明分布式并行化在降低延迟方面是有效的。

- 进一步增加 MPI 进程数到 8 个 (MPI:8) 时, 延迟继续降低至约  $250\ \mu s$ , 加速比提升至约 3.4。这说明在一定范围内, 增加 MPI 进程数可以持续提升性能。
- 然而, 当 MPI 进程数增加到 16 个 (MPI:16) 时, 查询延迟反而略微增加到约  $280\ \mu s$ , 加速比下降至约 2.6。并行开销 (如 MPI\_Gather 操作) 开始抵消并行计算带来的收益, 达到了并行化的瓶颈。

#### • MPI+SIMD 优化:

- 将 MPI 与 SIMD 技术结合后 (MPI+SIMD), 查询延迟取得了显著的突破, 降至约  $100\ \mu s$ 。
- 这一组合方案实现了相对于 IVF Flat 约 8.1 的最高加速比。这有力地证明了 MPI 提供的粗粒度并行 (任务分发) 与 SIMD 提供的细粒度并行 (单核内向量化计算) 可以有效协同, 显著提升整体性能。SIMD 极大地加速了每个进程内部的距离计算这一核心耗时操作, 从而进一步降低了总查询延迟。

总的来说, MPI 优化在降低查询延迟方面表现良好, 和多线程一样, 也存在最优进程数, 在真实环境下需要根据数据找到合适的参数。而 MPI 与 SIMD 的结合则展现出强大的潜力, 实现了最佳的查询性能。

※ 在实验中, 我实现了 MPI+OpenMP, 却是**负优化**, 并且负优化得... 很严重。反复检查代码后并没有发现不妥之处, 只能说按照目前的结果分析的话, 并行开销显著超过了并行带来的收益。在 MPI+OpenMP 这种混合并行模式中, 开销来源变得更加复杂, MPI 进程间通信、同步, OpenMP 线程创建/销毁、同步, 资源争夺, 内存瓶颈都可能是原因, 因此暂时没有展示 MPI+OpenMP 的效果, 之后有时间会再审视讨论一下这个问题。

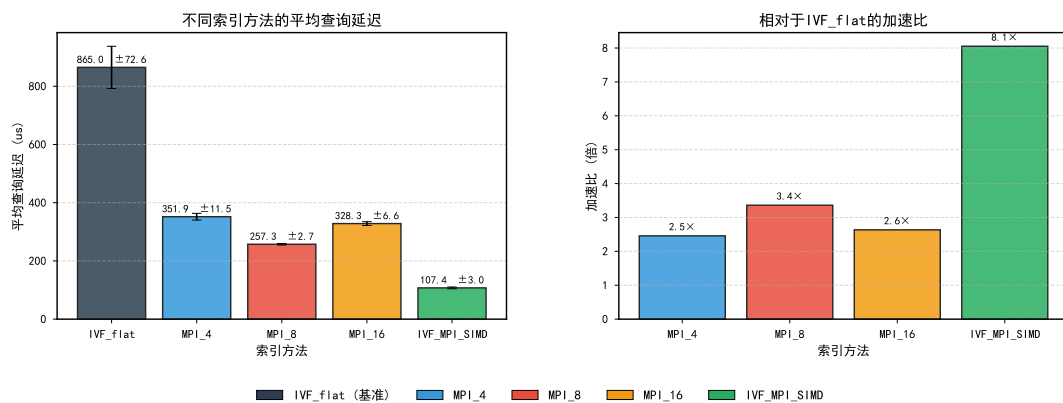


图 1: IVF 各并行优化效果

## 2. recall-latency trade-off

图 2 展示了不同并行优化算法在 Recall (召回率) 和 QPS (每秒查询数) 之间的权衡曲线。QPS 越高表示查询吞吐量越大, Recall 越高表示查询的准确性越好, 理想的算法应该尽量靠近右上角。

- **IVF (MPI\_8):** 用 8 个 MPI 进程的 IVF 优化方案, 其 QPS 显著高于 IVF-Flat。这再次验证了 MPI 在分布式环境下的高效并行能力。
- 随着 Recall 的提高, QPS 会有所下降, 并且**精度越高, 下降的速度越快**, 体现了精确度与速度之间的经典权衡。

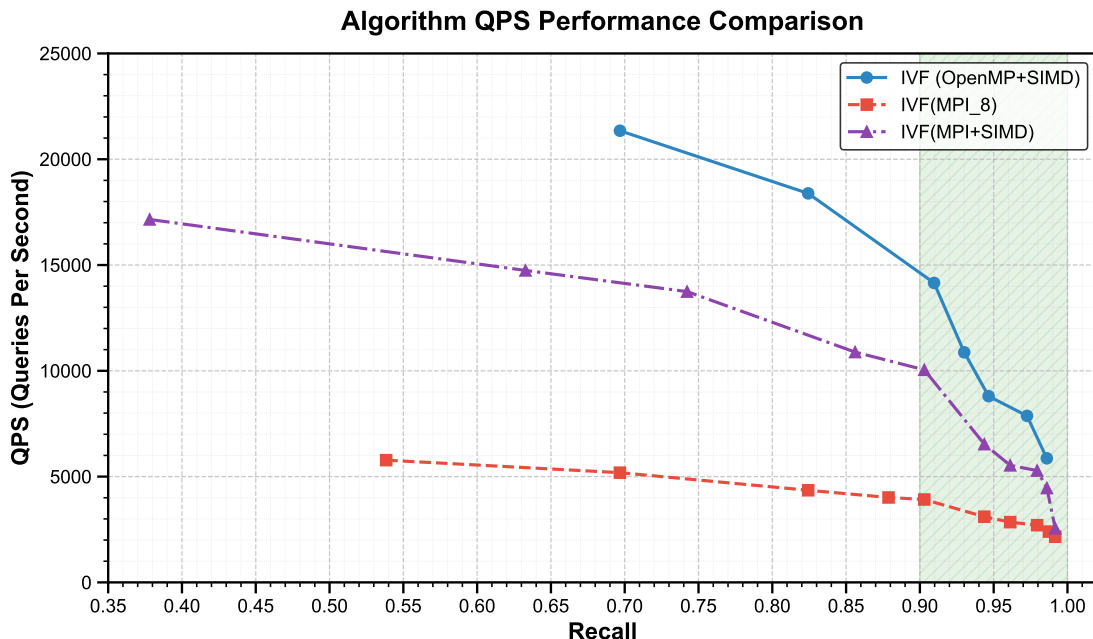


图 2: IVF

### 3. VTune 分析

在本实验环境中, OpenMP+SIMD 的曲线比 MPI+SIMD 的曲线更靠近右上角, 效果更好, 这是一个有意思的发现, MPI 实验中, 我申请的是一个节点 (MPI\_8)。这个现象本质上反映了两种并行模型在单节点多核环境下, 各自对计算资源、内存层次和通信开销的不同利用效率。为了探究其中可能的原因, 我将项目移植到 x86 上使用 VTune 进行分析。

如图3所示, 我比较了 IVF-Flat、IVF (OpenMP+SIMD) 以及 IVF (MPI+SIMD) 三种并行化方法的性能。从退休指令数 (Instructions Retired) 来看, MPI+SIMD 的指令数略多于 OpenMP+SIMD, 并且 MPI 的 CPI 也要略高一些。**这表明在这次的实验环境中, MPI 的并行开销比 OpenMP 更大。**

我进一步分析了 cache 性能, 如图4所示。从 (a) 子图的缓存命中率对比可以看出, OpenMP 版本在各级缓存上的表现均显著优于 MPI 版本, (b) 子图的绝对数量也印证了这一点。各级缓存都是 MPI 的访存次数更多, 且 MISS 比例高于 OpenMP。

MPI+SIMD 版本的分布式内存特性可能破坏了数据局部性, 导致了严重的缓存未命中问题。相比之下, OpenMP 作为共享内存并行模型, 能更好地利用节点内的缓存层次结构, 维持了较高的缓存命中率。

#### • 进程间通信 vs 线程内共享

MPI 的优势在于跨节点、跨机器的可伸缩性, 但对于这个数据集规模和实验环境下, 每个 MPI 进程拥有自己独立的地址空间, 必须通过内核态的消息传递来交换数据, 开销较大。如果数据规模更大, MPI 的加速效果可能更明显。

OpenMP 则是在同一进程内部启动多个线程, 所有线程共享同一块物理内存, 对候选向量或索引块的访问可以直接通过指针或共享缓冲区完成, 省去了进程边界的拷贝和同步开销, 内存带宽与缓存命中率的优势被放大 (并且这对 SIMD 也是有利的)。

- **负载均衡:** OpenMP 的动态调度策略 (schedule(dynamic)) 在倒排表长度差异较大时可以更灵活地负载均衡, MPI 暂时并没有实现相关特性。

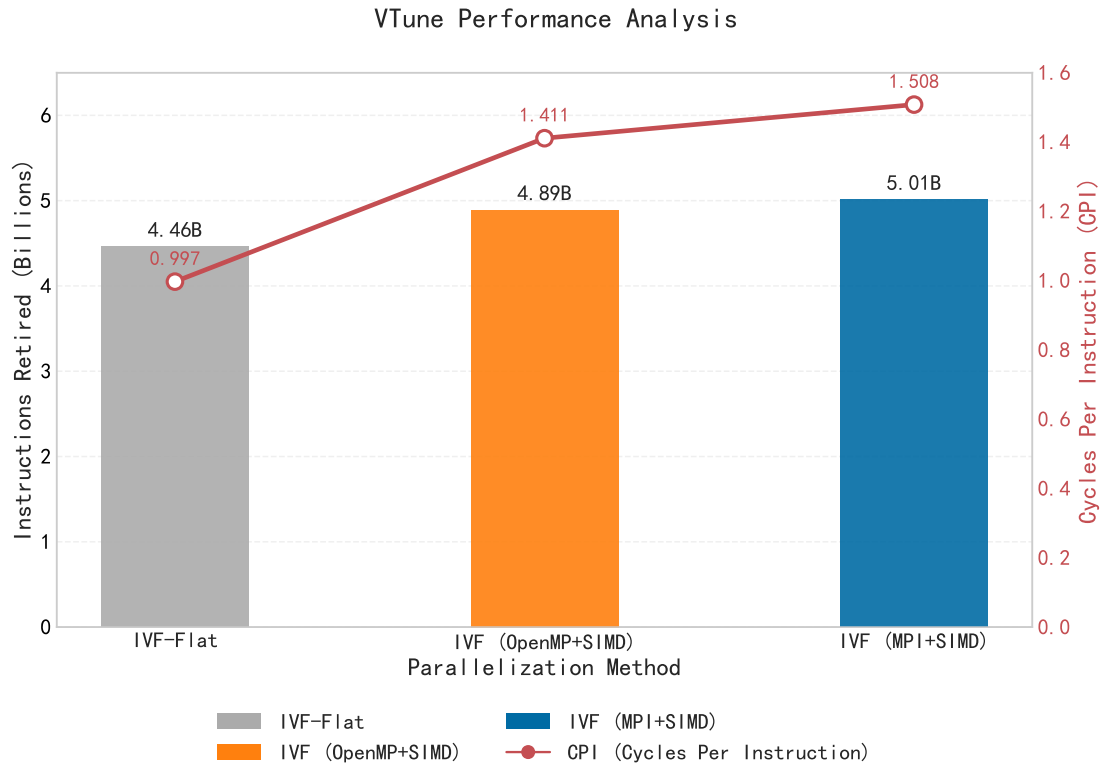


图 3: Instructions Retired &amp; CPI

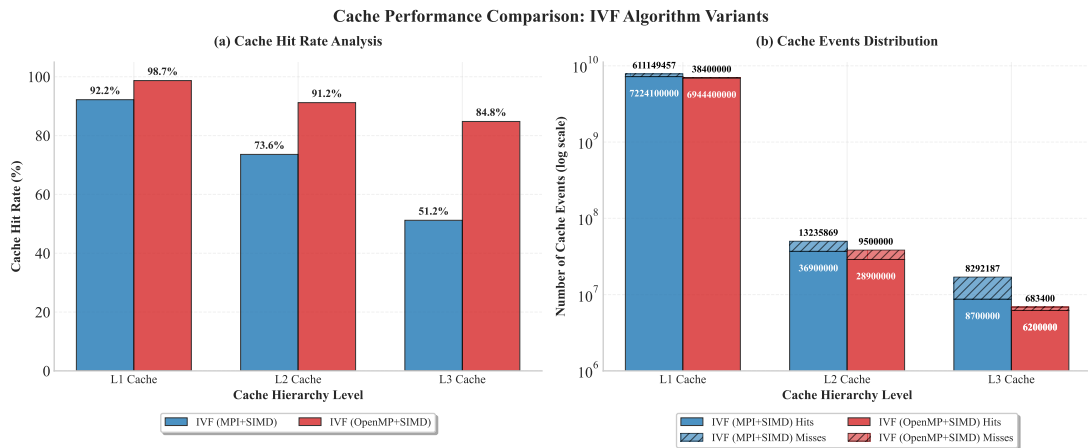


图 4: MPI 与 OpenMP 的缓存性能对比

## 二、 HNSW

在多线程实验中，我尝试了使用 OpenMP 并行化 HNSW [4]，确认使用简单的并行思路会带来负优化，也表明 HNSW 本身并行化的难度很高。于是转而取去实现 IVF+HNSW，并成功使用 OpenMP 实现了并行加速。在这一部分中，我先使用 MPI 对 IVF+HNSW 并行化。然后我探索了分区 HNSW 的并行策略。

## (一) IVF+HNSW

### 1. MPI 优化

IVF 和 HNSW 的结合是一种在精确性和速度之间取得良好平衡的近似最近邻搜索方法：MPI 优化粗粒度查找 (IVF)，而细粒度查找 (HNSW) 则保持串行不变。

1. IVF 索引预构建与 HNSW 簇内索引：数据集在索引构建阶段首先通过聚类被划分为 numClusters 个簇，每个簇拥有一个聚类中心。接着，在每个簇内部的数据点被独立地构建成一个 HNSW 图索引。这意味着每个簇都有一个独立的、优化的 HNSW 图结构用于高效的近邻搜索。这些 HNSW 索引在查询开始前已被构建并加载 [3]。
2. 局部 HNSW 索引加载：每个 MPI 进程会加载完整的聚类中心信息。在查询时，所有进程会计算查询向量与所有 numClusters 个聚类中心的距离，并找出最近的 nprobe 个候选簇。
3. 簇间并行搜索：查询被分发给不同的 MPI 进程。具体而言，程序通过 `if (i % size != rank) continue;` 语句，确保每个 MPI 进程只在其负责的候选簇内执行 HNSW 搜索。这实现了粗粒度的簇间并行。不同进程并行地在它们各自负责的 HNSW 索引中查找最近邻，充分利用了分布式计算资源。

```

1 // 每个进程处理分配给它的簇
2 for (size_t i = 0; i < search_clusters_indices.size(); i++) {
3     // 只有负责这个簇的进程才进行搜索
4     if (i % size != rank) continue;
5
6     size_t cluster_id = search_clusters_indices[i];
7     if (!indices[cluster_id] || indices[cluster_id]->getCurrentElementCount() ==
8         0) {
9         continue;
10    }
11
12    // 设置 HNSW 搜索参数并执行搜索
13    indices[cluster_id]->setEf(efSearch);
14    auto result_pq = indices[cluster_id]->searchKnn(query_point,
15        search_k_in_cluster);
16    // ... (结果收集) ...
17 }

```

Listing 7: 簇间并行搜索分配

4. 结果聚合：每个 MPI 进程完成其负责的簇内 HNSW 搜索后，会将找到的局部最佳近邻结果（距离和原始标签 `hnsplib::labeltype`）存储在 `all_results` 向量中。
5. 数据收集与全局合并：首先，所有进程通过 `MPI_Allgather` 交换各自 `all_results` 的数量。然后，使用 `MPI_Allgatherv` 将所有进程的局部结果收集到所有进程的 `final_results` 向量中。`MPI_BYTE` 作为数据类型和相应的字节偏移量，确保了复杂数据结构的正确传输。

```

1 // 使用 MPI_Allgather 收集所有进程的局部结果数量
2 std::vector<int> all_counts(size);
3 int local_count = local_results.size();
4 MPI_Allgather(&local_count, 1, MPI_INT, all_counts.data(), 1, MPI_INT,
5     MPI_COMM_WORLD);
6 // 计算偏移量并转换为字节计数

```

```

7  std::vector<int> displs(size);
8  std::vector<int> byte_counts(size);
9  std::vector<int> byte_displs(size);
10 int total_results = 0;
11 for (int i = 0; i < size; i++) {
12     displs[i] = total_results;
13     total_results += all_counts[i];
14     byte_counts[i] = all_counts[i] * sizeof(std::pair<float, hnswlib::labeltype>)
15     ;
16     byte_displs[i] = displs[i] * sizeof(std::pair<float, hnswlib::labeltype>);
17 }
18 // 收集所有结果到每个进程
19 std::vector<std::pair<float, hnswlib::labeltype>> final_results(total_results);
20 MPI_Allgather(local_results.data(), local_count * sizeof(std::pair<float,
21     hnswlib::labeltype>), MPI_BYTE,
22     final_results.data(), byte_counts.data(), byte_displs.data(),
23     MPI_BYTE, MPI_COMM_WORLD);

```

Listing 8: MPI 数据收集与合并

6. 使用堆的结构获取前  $k$  个结果：所有进程都拥有完整的合并结果后，使用堆的结构获取前  $k$  个结果，以获得最终的近似最近邻。

```

1 // 使用最大堆选择前k个最小距离的结果 - 比全排序更高效
2 std::vector<std::pair<float, hnswlib::labeltype>> result;
3 // 使用最大堆维护前k个最小值
4 std::priority_queue<std::pair<float, hnswlib::labeltype>> max_heap;
5 for (const auto& item : final_results) {
6     if (max_heap.size() < k) {
7         max_heap.push(item);
8     } else if (item.first < max_heap.top().first) {
9         max_heap.pop();
10        max_heap.push(item);
11    }
12 }
13 // 从堆中提取结果并按距离排序（此时堆中是最大的k个，从小到大排序）
14 result.reserve(k);
15 while (!max_heap.empty()) {
16     result.push_back(max_heap.top());
17     max_heap.pop();
18 }
19 // 最终按距离从小到大排序
20 std::sort(result.begin(), result.end());
21 return result;

```

Listing 9: 寻找 hnsw 最终结果 (最大堆优化)

## 2. MPI+SIMD

在计算内积距离的时候也可以使用 SIMD 进行数据上的并行，将距离计算的函数更换为 SIMD 版就行，此处和 IVF 部分一致，因此省略。

## (二) 分区 HNSW

除了之前讨论的 IVF-HNSW 混合索引方法, 另一种在分布式环境中实现高效近似最近邻搜索的策略是直接将数据集划分为多个独立的部分, 并对每个部分构建单独的 HNSW 索引 [2]。

### 1. 核心思想

分区 HNSW 的核心思想是将庞大的数据集直接水平地分割成  $P$  个较小的子集 (分片), 其中  $P$  通常可以根据可用的并行计算资源 (例如 MPI 进程数) 设定在 2 到 8 之间, 甚至更高。对于每个数据子集, 独立地构建一个完整的 HNSW 图索引。在执行查询时, 所有  $P$  个 HNSW 索引可以利用 MPI 进行并行搜索, 从而加速全局最近邻的查找。

### 2. 数据集划分策略

数据集的划分方式对最终的搜索性能和召回率有着重要影响:

1. **随机划分**: 这是最简单直接的划分方式, 将数据集中的点随机分配到  $P$  个分区中。
2. **启发式策略**: 这类策略旨在更智能地划分数据, 以优化某些性能指标。

临近期末时间有限, 本次实验我直接实现随机划分。

### 3. 基于 MPI 的并行搜索

在查询阶段, 查询向量会被发送到所有参与的 MPI 进程。每个 MPI 进程在其负责的数据分区对应的 HNSW 索引上执行局部近似最近邻搜索。每个进程会返回其局部子集中最接近查询向量的  $k'$  个结果。随后, 所有进程将各自的局部结果通过 MPI 通信机制 (例如 `MPI_Allgatherv`) 汇集起来, 形成一个全局的结果集。最后, 对这个全局结果集进行排序并截取前  $k$  个结果, 得到最终的全局近似最近邻。这些过程的细节和 IVF+HNSW 基本一致。

### 4. 与 IVF-HNSW 的对比优势

与之前介绍的 IVF-HNSW 混合索引方案相比, 分区 HNSW 具有以下几个潜在的优势:

- **索引构建简化**: 分区 HNSW 不需要执行 K-means 聚类这一耗时且迭代的预处理步骤, 索引构建过程更为直接, 且各分区索引的构建可以完全并行, 进一步缩短构建时间。
- **潜在的更高召回率**: 查询时对所有分区都执行搜索, 且每个分区内部的 HNSW 搜索质量足够高, 则这种方法有可能达到更高的召回率 (**上限更高**), 因为它本质上是在**搜索整个数据集**。
- **负载均衡潜力**: 通过合理的划分策略 (例如均匀的随机划分), 可以更好地平衡不同 MPI 进程的计算负载, 避免某些进程因处理过大或过于复杂的簇而成为瓶颈, 对 MPI 来说, IVF 带来的负载不均衡问题比较难解决。

## (三) 实验

无论是 HNSW 还是 IVF+HNSW, 都有相当多可以调整的参数, 经过前面的实验以及预实验可知, 对这个数据集来说, 数据量不算太大, 因此申请一个节点 (8 个核心) 比申请多个节点的效果更好, **节点之间的通信开销**在本次实验中体现的很明显, 但是如果测试更大的数据规模, MPI 的扩展性非常有用。

为了尽量控制变量，突出并行方法本身的效果，本次 MPI 实验仍然申请一个节点 8 个进程，控制 recall=0.90。

1. 对于分区 HNSW 来说，把数据集分为 8 个子集。
2. 对于 IVF+HNSW 来说，根据多线程实验的结论，聚类数量 nlist 设置为 64，查询数量 nprobe=8。在绘制 recall-latency 图时，如果发现 nprobe 的数量成为 recall 的瓶颈，则增加 nprobe，否则优先增加 efs（实验发现达到相同精度时，增加 efs 的延迟比增加 nprobe 的延迟低）。

总的来说，以上的一些要求都是尽量让算法的参数保持较好的状态，突出并行效果。

### 1. 查询延迟 & 加速比

实验结果如图5所示。

- **HNSW\_flat (基准)**: 作为未经特定优化或作为对比基准的实现,其查询延迟约为 153.5 us。
- **IVF-HNSW(MPI+SIMD)**: 此策略结合了倒排文件索引 (IVF)、消息传递接口 (MPI) 进行分布式计算以及单指令多数据 (SIMD) 指令集进行向量化加速。其查询延迟显著降低至约 132.2 us。相对于 HNSW\_flat，其加速比约为 1.16x。这意味着该策略的查询速度比基准快了 16%。
- **HNSW\_part(MPI)**: 此策略采用 HNSW 分区并行，并利用 MPI 进行分布式计算。其查询延迟为 140.1 us。其加速比约为 1.10x，表示查询速度比基准快了 10%。

本次并行化采用的策略也都取得了**正优化**，此外我还发现当要求的 recall 提高时，加速比也会有变化，这部分在接下来的 recall-latency trade-off 中重点讨论。

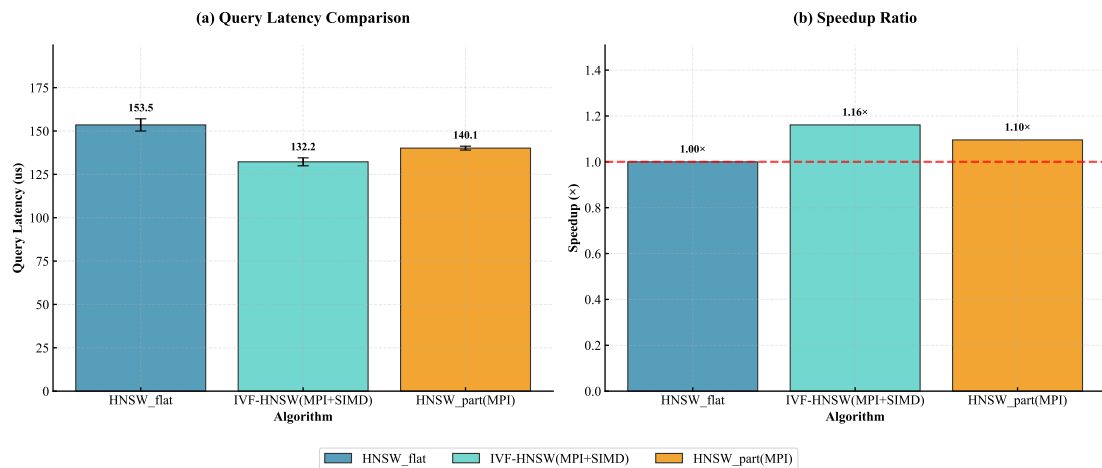


图 5: HNSW 各并行策略查询延迟 & 加速比

### 2. recall-latency trade-off

实验结果如图6所示，这张图展示了所有算法都呈现出召回率与 QPS 之间的反向关系：随着召回率的提高（从 0.60 到 1.00），QPS 普遍下降。这是因为为了达到更高的召回率，算法需要执行更全面的搜索，导致计算开销增加，从而降低了查询速度。

#### 1. 性能对比:

- **HNSW\_flat**: 作为基准, 其 QPS 性能在所有算法中最低, 但其曲线也相对平滑, 没有出现剧烈的性能骤降。
- **IVF-HNSW(OpenMP+SIMD)**: 在  $\text{recall}=0.93$  及之前, openMP 的加速表现出最高的 QPS 性能, 当召回率超过 0.92 后, 其 QPS 下降速度最快。
- **IVF-HNSW(MPI+SIMD)**:  $\text{recall}<0.92$  的范围内, MPI 优化的 QPS 优于串行 HNSW 和 MPI 分区 HNSW。但是在此之后, 性能又不如串行的 HNSW。
- **分区 HNSW\_MPI**: 该算法的 QPS 曲线相对平稳, 在整个召回率范围内保持了较好的稳定性。尤其是  $\text{recall}$  在 0.95 以上, 其他并行策略都出现了**负优化**的情况, 而 MPI 优化的分区 HNSW 仍然是**正优化**, 并且当  $\text{recall}=0.979$  时, **加速比达到 1.75**, **这个成绩比  $\text{recall}=0.90$  时更亮眼**, 表现出较强的鲁棒性。

## 2. 最优工作区域:

- 图中标注的绿色区域 ( $\text{recall}>0.90$ ) 表示高性能的区域, 其中多种算法的性能开始**波动或交叉**。在这个区域内, 不同的优化策略可能面临不同的挑战, 或者达到了各自的性能瓶颈。
- 选择何种算法取决于具体的应用需求: 如果对查询速度有极高要求, 且能接受略低的召回率, IVF-HNSW(OpenMP+SIMD) 在召回率 0.80-0.92 区间表现最佳; 如果需要更高的召回率且对速度也有要求, MPI 优化的分区 HNSW 更合适。

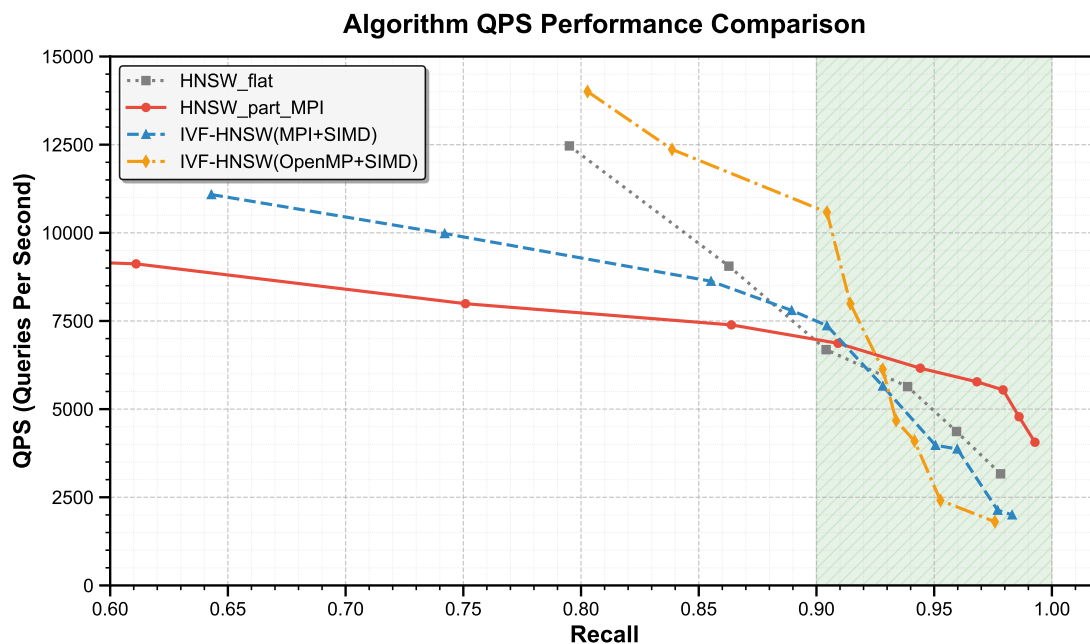


图 6: HNSW recall-latency trade-off

## 三、 总结

本次实验系统地探讨了针对 IVF 和 HNSW 两种近似最近邻搜索算法的并行化策略, 核心是用 MPI 进行优化, 同时结合了 SIMD 和多线程的技术进行分析和对比。

## (一) 核心结论

- **IVF 并行化**: 对 IVF 算法的 MPI 并行化研究表明, 使用 MPI 通过多进程间分配倒排列表搜索任务能有效提升性能, 但其加速效果会因进程数过多而受到通信开销和根进程合并瓶颈的限制而回落。此外, 结合 SIMD 可以进一步提高性能, **加速比达到 8.1x**。
- **HNSW 并行化**
  - **IVF-HNSW**: 通过 MPI 并行处理不同的数据簇, 此方法获得了约 1.16x 的加速比, 证明了该混合索引策略在分布式环境下的有效性。
  - **分区 HNSW**: 该策略直接将数据集分割, 并在各分区上并行搜索。此方法不仅简化了索引构建过程, 更在**高召回率 (recall > 0.95)** 的场景下展现出比其他并行策略更强的稳定性和性能优势, **加速比达到 1.75x**, 避免了其他方法在高精度要求下并行性能衰退甚至劣于串行的问题。

## (二) 启发

本次实验研究带来了以下几点深刻的启发:

1. **拥抱混合并行**: 无论是这次实验还是之前的多线程实验, SIMD 的加入都使性能更上一层楼。将不同层级的并行模型相结合是挖掘硬件潜力的关键, 通过粗粒度任务分发与细粒度计算加速的互补, 可以实现性能的叠加提升。
2. **根据实际情况选择合适的并行策略**: 不存在一种普适的“最佳”并行策略。最优解深度依赖于算法特性、硬件架构 (单节点多核 vs. 多节点集群)、数据规模和具体的性能目标 (如低延迟 vs. 高召回率)。例如, MPI 的优势在于跨节点的可扩展性, 上限更高, 而 OpenMP 在单节点内更具效率优势。
3. **算法与并行协同设计**: 与其在现有复杂算法上强行并行, 不如思考如何调整算法结构使其“并行友好”。分区 HNSW 的成功便是一个范例: 它通过简化数据划分策略, 天然地实现了更好的负载均衡和并行伸缩性, 尤其是在高精度场景下表现出众。这启发我们在设计算法之初就应将并行化考虑在内。

## 参考文献

- [1] ARM Limited. NEON Programmer's Guide. <https://developer.arm.com/documentation/102140/0201/Introduction-to-NEON>, 2023.
- [2] Chenying Du, Chenfei Zhou, Junyun Mai, Junlong Li, Yanfeng Ding, Zhiyuan Hua, Derong Kong, and Yifei Zhang. 并行程序设计实验指导书: MPI 编程. Technical report, May 2025. 以高斯消去为例的 MPI 编程实验指导.
- [3] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.
- [4] Yury A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2020.