LAB2

522031910213 朱涵

May 26, 2024

1 实验过程

1.1 HNSW部分

HNSW是一种用于多维特征的向量的近邻向量查询操作的数据结构，可以在电商的推荐系统等情景下广泛应用。HNSW的原理是基于无向图的查询操作，通过计算欧氏距离在近邻节点中不断导航跳跃直到获得一个局部最优解，牺牲一定的精确度来换取高效的时间性能。本人基于堆结构实现了一个简单的HNSW结构。

1.1.1 代码实现

HNSW部分主要实现了**insert**和**query**两个方法、**searchLayer**一个工具函数以及HNSW的基本结构。下面简单介绍一下我的实现思路（篇幅原因，只展示基本结构部分的代码，其余代码可自行查看**hnsw.hpp**）：

- **基本结构**：如下图。定义了一个结构体node作为无向图的节点，包含了向量、ID、next数组以及一个自定义比较函数用于建立**优先级队列**。HNSW只有两个属性，一个**enter_point**表示顶层的入口点以及**max_level**表示最高层。

```
struct node
{
    vect vec; // vector
    int id;    // label
    std::function<bool(node *, node *)> less = [this](node *a, node *b)
    {
        return l2distance(a->vec, this->vec, 128) < l2distance(b->vec, this->vec, 128);
    };
    std::vector<std::vector<node *>> next;                                                    // next[i]是当前节点在第i层的邻接节点集合（指针）
    node(vect v, int i, int level) : vec(v), id(i), next(level, std::vector<node *>()){}; // 初始化为level层，每层都是空的vector
};

class HNSW : public AlgorithmInterface
{
private:
    node *enter_point = nullptr; // 入口点
    int max_level;               // 最大层数
```

Figure 1: HNSW基本结构

- **查找近邻**：这个工具函数的实现和文档中基本一样，篇幅限制原因不再赘述。
- **插入**：插入操作和文档中的伪代码基本一样，分为三步进行，其中需要注意的是，伪代码中没有注明第一次插入节点时如何特判处理。在我的实现里，第一次插入节点会生成随机层数L，然后把L作为最高层并插入第一个节点到0——L层。
- **查询**：查询操作参考了文档的介绍，第一步和插入的第一步相似，在每一层寻找最近邻点并进入下一层。第二步为到0层之后开始查找对应数量的最近邻点并返回一个包含它们的数组。

1.1.2 优化细节

1. 本实验需要大量的“选取最近邻或最远邻点”的操作，因此本人使用了stl中的**优先级队列**结构以及make_heap等**堆函数**来优化时间复杂度：

   - 模仿了之前写的**跳表**结构，node结构里的next字段用于存储了一个节点在0——L层的邻接节点集合（即**next[i]是当前节点在第i层的邻接节点指针集合**）。在插入操作的第三步中，当一个新节点的插入导致了某些节点的连接数超过M_max后，需要删除离此节点最远的节点。此时就可以使用建堆函数进行优化，在next[lc]上建立大根堆，不断弹出最远的点，直至符合要求。
   - 同理，**插入操作**里的最近邻元素集合W也用了优先级队列进行优化，方便选出高层中的最近邻入口点，以及选出最近邻的M个点进行新节点的连接。
   - **searchLayer**方法中，候选节点集合C和最近邻元素集合W分别需要不断取出最近邻和最远点，因此分别建立为小根堆和大根堆来优化。

2. 在插入算法的第三步中，删去超过M_max限制的连接时，伪代码中并未删去对方和此节点的连接，而是**单向**进行了删除。本人并不知道是否是有意为之。如果需要删除双向的连接，则要耗费大量的时间（因为需要在邻接表里进行线性的查询），或者就要用邻接矩阵来存储边。两者都并非很好的实现方案，而且在本人的测试下，是否双向删除并不太会影响性能和召回率（甚至单向删除还更高，因为保留了更多连接，接近最优解的概率更大）。综上考虑，在实现时决定依照文档实现，只作**单向删除**。

3. 在实现HNSW结构时遇到了一些困难，一个是插入第一个节点的层数，一开始我使用了固定层数（比如20层），结果效果很差，后面改成了使用函数生成的层数；另一个是关于层数，在我的实现里，next[i]存储的是第i层的边，但是这个i是0-base的，而我把lc当成了1-base的，最终我在新建节点时创建L+1个数组，解决了正确性问题。

1.2 并行优化部分

当HNSW结构应用于实际场景时，串行带来的性能损失是致命的，并且也不符合用户的无阻塞需求。因此考虑对于查询操作进行多线程的**并行优化**，实现一个可以并发查询的HNSW结构。

1.2.1 代码实现

并行优化部分主要实现了对于查询操作的**多线程化**，下面简单介绍一下我的实现思路： 本人对课件上的多线程无锁优化程序进行了模仿，采用了预先确定线程数，并为每个线程平均分配任务的做法。如下图是每个线程的**任务函数**，及负责进行一部分向量的查询操作。

```
void query_tasks(int id)
{
    int start_index = id * query_n_vec / thread_num;
    int end_index = std::min((id + 1) * query_n_vec / thread_num, query_n_vec);
    for (int i = start_index; i < end_index; ++i)
        hnsw_parallel.query(query + i * query_vec_dim, query_num); // 执行查询操作
}
```

Figure 2: 线程任务函数

如下图是**主线程**里分配任务的代码，即开始计时并按照预先的线程数启动所有线程，等待所有线程回收后停止计时，最后计算出平均时延。

```
// 进行查询时间测试 - HNSW_Parallel
std::chrono::duration<double, std::micro> query_elapsed_parallel(0);
auto query_start_parallel = std::chrono::steady_clock::now();
for (int i = 0; i < thread_num; i++)
    threads[i] = std::thread(query_tasks, i);
for (int i = 0; i < thread_num; i++) // 等待所有线程结束
    threads[i].join();
auto query_end_parallel = std::chrono::steady_clock::now();
query_elapsed_parallel = query_end_parallel - query_start_parallel;
double average_query_time_parallel = query_elapsed_parallel.count() / query_n_vec;
// 输出结果
std::cout << "Average query time (HNSW_Parallel): " << average_query_time_parallel
```

Figure 3: 主线程部分

1.2.2 优化细节

基于此种方式的优化虽然带来的显著的提升，但是没有消除反复创建线程带来的时间损耗。在更复杂的查询情况下（如几万条查询操作），应该使用**线程池**，即预先启动所有线程，等待主线程分配任务，对等线程完成一个查询任务后主线程继续分配新的任务。

2 测试结果

2.1 正确性测试

如下图，在默认参数（M=30）下，召回率达到了**100%**，平均查询时延在**0.8ms**左右，说明HNSW的结构正确且性能优秀。 另外，并行部分的测试程序也经过了正确性验证，详见test.cpp文件，主要通过把查询结果存储到一个预分配好空间的数组里进行比较来计算召回率，在相同参数下召回率仍为**100%**。

```
average recall: 1.000, single query time 0.8 ms
```

Figure 4: 正确性测试结果

2.2 参数M的影响

参数**M以及M_max**会对HNSW的召回率以及性能产生显著影响。如下图是不同参数M下的性能曲线。容易看到，召回率在M>10之后就保持在了**100%**，说明本人实现的HNSW查询相似向量的准确率极高；而单次查询时延随着M的上升呈**线性趋势**，说明M越大，每次查询需要遍历的邻居节点就越多，也就耗时越高，这是符合预期的。

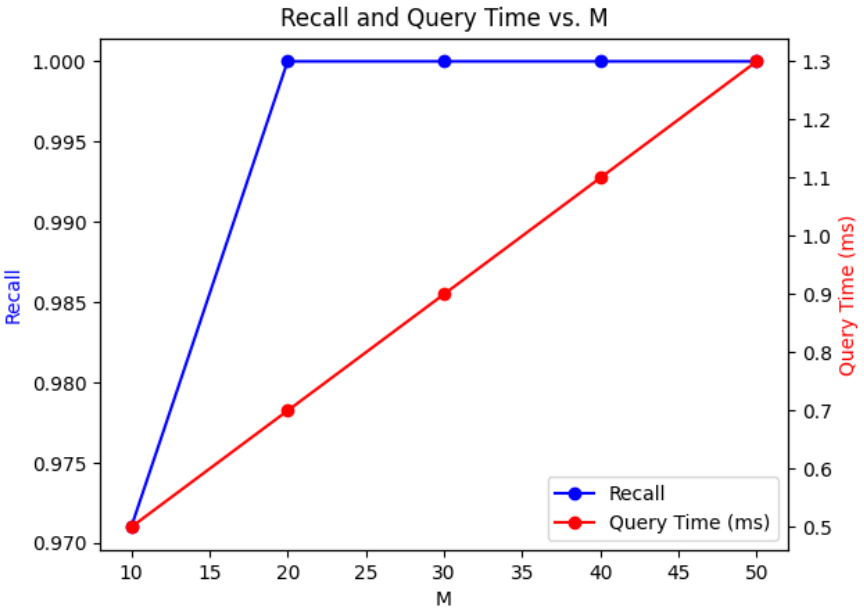

Figure 5: Enter Caption

2.3 串行及并行性能测试

进行并行优化后，HNSW的查询性能应该有显著提升。在test.cpp文件中编写了测试类，使用和第一部分一样的数据文件以及查询近邻数量（10）进行测试，分别对单线程的插入和查询、线程数从1——8、16——100的多线程查询进行了时延计算，下面是测试结果： 如下表是串行HNSW插入以及查询的时延。可以发现插入耗时比查询要低一些，可能是因为插入无需大量的寻找邻居节点。

单次插入耗时（μs）	617.6
单次查询耗时（μs）	862.5

Table 1: 单线程HNSW的性能

如下表和图是不同线程数下并行HNSW查询性能的测试结果。可以从图中看出，随着线程数增大到8，耗时呈反比例趋势下降；线程数从8增大到100，耗时则变化较小，甚至最终有所提高。

线程数	1	2	3	4	5	6	7
单次查询耗时（μs）	848.1	426.3	287.9	226.4	207.1	182.7	168.7
线程数	8	16	32	48	64	80	100
单次查询耗时（μs）	153.8	151.8	148.4	163.1	182.162	163.6	260.4

Table 2: 不同线程数下的并行HNSW性能表

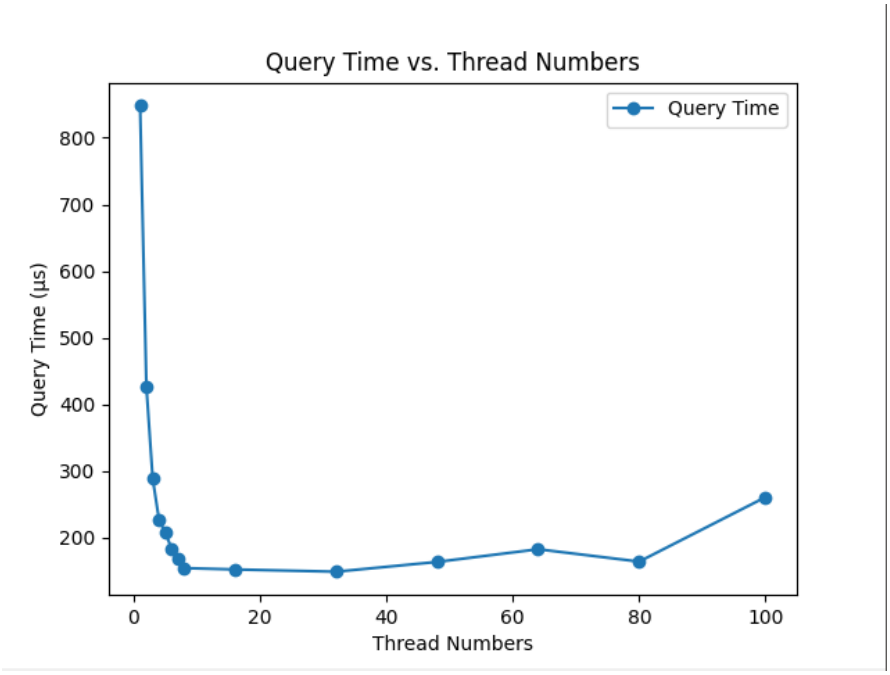

Figure 6: 不同线程数下的HNSW性能曲线

从多线程理论的角度来分析，由于我的运行环境（虚拟机，Ubuntu系统）设置了**8核**的处理逻辑，因此性能应该在8线程时达到最优，而继续增大线程数时则会因为线程创建的额外开销而导致性能反而降低。实验结果和预期分析是完全符合的，因此HNSW的并行优化部分是较为成功的。

3 总结

本次实验基于优先级队列和无向图实现了用于高效查询局部最相似向量的HNSW结构，并实现了多线程的并行优化，成功提升了HNSW的查询性能。后续实验可以继续优化HNSW的选取近邻向量算法以及并行优化的更优解（如线程池或是插入优化）。