Lab2+3: HNSW+并行编程实验报告

1. 实现时遇到的问题与难点

在实现HNSW的过程中，主要是要对伪代码进行代码的实现。

1.1 对于整个hnsw首先须明白，enterpoint和max_level是全局的，而在insert和query函数中，应该使用全局的enterpoint赋值给函数内部的ep，然后可以使用遍历layers来找到最大层不为空的layer即为max_level.

如下：

```
ep=enterpoint;
for (int lc =layers.size()-1; lc >= 0; --lc) {
      if(!(layers[lc].empty())){
           max_level=lc;
           break;
      }
}
```

而在我一开始的实验中ep和enterpoint混为一谈，导致实验结果不准确，尤其是伪代码的这一部分。

```
// 如果q更新了图的最高层数，则将之后插入或查询的入口节点设置为q
if L > maxL
    set enter point for hnsw to q
```

1.2 对于插入过程最初的节点处理，最开始的时候整个图为空，因此应该先进行初始化。

```
if (layers.empty()) {
            for (int lc = L; lc >= 0; --lc) {
            layers[lc][label] = {};
            }
        enterpoint=label;
        // std::cout << "Inserted first element at level " << L << " with label " <<
label << std::endl;
        return;
        }
```

1.3 对于search_layer函数，要返回C中的最近邻

在代码实现中，我的W，C均为最大化队列，因此为了使得C.top能返回我需要的结果，进入C的数据，为距离的负值，即可满足自动排序。

```
std::unordered_set<int> visited;
std::priority_queue<std::pair<double, int>> C;
std::priority_queue<std::pair<double, int>> W;
double initial_distance = euclidean_distance(q, nodes[ep][0]);
C.push({-initial_distance, ep});
W.push({initial_distance, ep});
```

1.4 并行查询的实现

```
TimeRecord query_record;

    std::vector<std::thread> threads;
    for (int i = 0; i < query_n_vec; ++i) {
        threads.emplace_back(query_thread, std::ref(hnsw), query + i * query_vec_dim,
query_vec_dim, gnd_vec_dim, std::ref(test_gnd_l), i);
    }

    for (auto& th : threads) {
        th.join();
    }

    single_query_time = query_record.get_elapsed_time_micro() / query_n_vec * 1e-3;
```

- 首先创建一个线程向量，用于存储所有查询线程。
- `threads.emplace_back(query_thread, std::ref(hnsw), query + i * query_vec_dim, query_vec_dim, gnd_vec_dim, std::ref(test_gnd_l), i);`：为每个查询创建一个线程，并将 `query_thread` 函数作为线程的执行函数。
- 然后在循环中确保主线程等待所有查询线程完成后再继续执行。

对于查询线程函数：

它接收 HNSW 实例、查询向量、向量维度、最近邻个数、结果容器和索引，并调用hnsw的query函数，将查询结果存储在结果容器中。

```
void query_thread(HNSW& hnsw, const int* query, int query_vec_dim, int k,
vector<vector<int>>& results, int idx) {
    results[idx] = hnsw.query(query, k);
}
```

2. 参数M的影响

修改parameter中的M和M_max的值，得到如下结果

M和M_max的值	查询召回率	查询时延
10	80.7%	0.0723ms
20	98.7%	0.1007ms
30	99.1%	0.1232ms
40	99.3%	0.1598ms
50	99.4%	0.1679ms

总体趋势：随着M的增加，查询召回率上升，查询时延上升，这是因为当M增大，每个节点的邻居节点数上升时，更有可能通过遍历查询得到争取的结果。同时查询时需要处理的节点也增多，查询时延增大。

从M=10到M=20查询召回率显著上升，随着M的上升查询召回率的提升幅度逐渐减小，趋于饱和。这可能是因为增加的邻居数量虽然能够带来更多的查询路径选择，但这些新增加的路径可能与已有的路径重合或仅带来少量新的节点访问。

而查询时间的延长较均匀。查询过程中使用优先队列来维护候选节点和最近邻节点。更多的邻居意味着更多的节点需要加入队列和从队列中移除，增加了优先队列操作的次数和复杂度。

选择 M 值时，需要在查询召回率和查询时延之间进行权衡。因此M=30或20是一个很好的选择，M再增大，查询召回率提升不大，但查询时延上升，代价较大。

3. **性能测试**

   串行查询和并行查询查询召回率几乎一致，查询时延对比如下。

M和M_max的值	串行查询平均单次	并行查询平均单次
10	0.0723ms	0.0225ms
20	0.1007ms	0.0277ms
30	0.1232ms	0.0326ms
40	0.1598ms	0.0369ms
50	0.1679ms	0.0410ms

M和M_max的值	插入平均单次查询
10	0.074ms
20	0.1112ms
30	0.1601ms
40	0.2195ms
50	0.2546ms

   整体而言，并行查询平均单次的时间小于串行查询，并行查询利用了CPU资源，利用多核处理器的优势，整体而言平均查询时间大大小于串行查询。