Lab2+3: HNSW+ 并行编程实验报告

赵楷越 522031910803

2024 年 5 月 29 日

1 实现过程中遇到的难点和印象较深的细节

1.1 用于存储节点信息的数据结构的选择

在设计存储节点信息的数据结构上时、由于本项目结构并不复杂、为了进行简洁明晰的调用与存储、采取以向量的 label 为索引、将向量的实际数值和对应的 level 存入哈希表中进行使用。同时、将 neighbors 利用哈希表、映射成一个个集合进行存储、也有利于后续 SELECT_LAYER 和 SELECT_NEIGHBOR 函数的实现。这里也使用一个 int 类型对整个 HNSW 当前的入口点的 label 进行存储、方便 insert 过程中进行调用。具体代码如下图所示。

```
53      // 用unordered_map来存储label和对应的向量
54      std::unordered_map<int, const int *> vector_map;
55
56      // 用unordered_map来存储label和对应的level
57      std::unordered_map<int, int> level_map;
58
59      // 用unordered_map数组来存储在第lc层的label和对应所有邻居，lc不超过30，每一层的邻居集合用unordered_set来存储
60      std::unordered_map<int, std::unordered_map<int, std::unordered_set<int>>> neighbors;
61
62      // 入口点的label，-1表示没有入口点
63      int enter_point;
```

在实现 SELECT_LAYER 和 SELECT_NEIGHBOR 函数的过程中、由于需要利用到最大堆或是最小堆的性质对距离查询向量最远或最近的向量进行快速取出、原先考虑的利用 priority_queue 类型进行数据传递，但是在发现伪代码中、SELECT_NEIGHBOR 函数的第二个参数可能是 W 或者是 eConn 时、为了与 neighbors 的集合存储统一接口形式、统一使用了 unordered_set 作为返回值及第二个传递参数。有利于代码的撰写与调试。函数定义如下图所示。

```
77      // SEARCH_LAYER函数，q是查询的向量，ep是入口点的label，ef是搜索的最大个数，lc是搜索的层数
78      std::unordered_set<int> SEARCH_LAYER(const int *q, int ep, int ef, int lc);
79
80      // SELECT_NEIGHBORS函数，q是查询向量的label，W是当前层的所有候选邻居节点的label集合，M是最大邻居数，lc是搜索的层数
81      std::unordered_set<int> SELECT_NEIGHBORS(int q, std::unordered_set<int> W, int M, int lc);
```

1.2 最大（最小）堆的自定义比较函数

```
23      // 自定义最近邻比较函数
24      struct cmp_nearest
25      {
26          const int *item;
27          const std::unordered_map<int, const int *> &vector_map;
28
29          cmp_nearest(const int *item, const std::unordered_map<int, const int *> &vector_map)
30              : item(item), vector_map(vector_map) {}
31
32          bool operator()(int label1, int label2) const
33          {
34              return l2distance(item, vector_map.at(label1), 128) > l2distance(item, vector_map.at(label2), 128);
35          }
36      };
```

为了在 SELECT_LAYER 和 SELECT_NEIGHBOR 函数中利用最大堆或是最小堆的性质对距离查询向量最远或最近的向量进行快速取出，自定义了最大（最小）堆的自定义比较函数如上图所示，使用时、需要传入当前查询的向量与当前存储 HNSW 中的所有向量实际值的 vector_map。上图为自定义的最近邻比较函数，会将离查询向量最近的向量的 label 置于堆顶，便能利用 priority_queue 进行高效取值。

2 并行优化细节

这段并行优化代码的主要目的是使用多线程来处理查询请求。为每一个查询请求都分配了一个自己的线程。每个查询请求都在其自己的线程中处理，这样的并行处理能够提高程序的效率。

下面是对下图中关键代码的详细解释：

1. 第 58 行：创建了一个线程向量 threads，其中包含 gnd_n_vec 个线程。目的是使每个查询请求都将在其自己的线程中处理。
2. 第 60-64 行：在每次循环中创建一个新的线程，并将其添加到线程向量中。新线程的任务中调用 hnsw.query 进行查询，该函数的参数是查询向量和向量的维度。查询请求的结果存储在 test_gnd_l[index] 中。
3. 第 66-69 行：循环遍历线程向量中的每个线程，并等待它完成。这是通过调用 std::thread::join 函数实现的。join 函数会阻塞当前线程，直到被调用的线程完成。这样可以确保所有查询请求都处理完毕，然后再继续执行后续的代码。

```
54      vector<vector<int>> test_gnd_l(gnd_n_vec);
55      double single_query_time;
56      TimeRecord query_record;
57
58      std::vector<std::thread> threads(gnd_n_vec);
59
60      for (int i = 0; i < gnd_n_vec; i++)
61      {
62          threads[i] = std::thread([&](int index)
63                                   { test_gnd_l[index] = hnsw.query(query + index * query_vec_dim, gnd_vec_dim); }, i);
64      }
65
66      for (auto &thread : threads)
67      {
68          thread.join();
69      }
```

3 参数 M 的影响

整理不同的 M 取值与查询召回率和串行单次查询时延的关系如下表所示。首先、观察查询召回率的变化、发现仅当在 M=M_max=30 的情况下、该代码在我的机器上能跑出不太理想的 93.7% 的召回率、而在同学的机器上能跑出 99.1% 的召回率、而在 M=M_max=20、40、50 的情况下、该代码在我的机器上的查询召回率的表现均较为良好。询问助教老师后、分析得出结论：猜测可能是不同环境中随机数生成等问题导致的这一现象。

M=M_max	查询召回率	单次查询时延 (ms)
10	0.712	0.444
20	0.976	0.683
30	0.937	0.760
40	0.987	0.953
50	0.990	1.180

表 1: 不同的 M 取值与查询召回率和串行单次查询时延的关系

其余数据均与理论上的变化趋势相符、当 M=M_max 较小时、由于被插入节点需要与图中其他节点建立的连接数较少、向量间的关联度降低、因此查询召回率较低；同时、由于查询时需要处理的连接数变少、因此单次查询时延也较少。而当 M=M_max 较大时，由于向量间的连接数较多、向量间的关联度提升、因此最终的查询召回率也较高；但由于查询时需要处理的连接数变多、单次查询时延也变高。

4 性能测试

4.1 平均单次插入时延

在 test.cpp 中、首先对代码的平均单次插入时延进行了测试、如下表所示。发现随着 M=M_max 的提升、平均单次插入时延也相应地增加了。分析是因为每次需要建立的向量间的连接数增加了、因此所耗时间会对应增加。

M=M_max	平均单次插入时延
10	0.049
20	0.084
30	0.134
40	0.211
50	0.266

表 2: 不同的 M 取值与平均单次插入时延的关系

4.2 平均单次查询时延

首先、基于上文所述的朴素方法（给每一个查询任务分配一个线程，对应代码在 test.cpp 中）对不同 M=M_max 的情况下对并行的单次查询时延进行了测试。因为本次实验的查询请求只有 100 条，如果查询请求的数量达到数万甚至数十数百万，进一步思考后，一种可能更好的处理方法是利用线程池进行优化（对应测试代码在 threadpool_test.cpp 中）。测试得到不同的 M 取值与串行、朴素并行以及线程池并行时延的关系如下表所示：

M=M_max	串行方法单次查询时延 (ms)	朴素并行方法单次查询时延 (ms)	线程池单次查询时延 (ms)
10	0.444	0.132	0.101
20	0.683	0.157	0.105
30	0.760	0.249	0.183
40	0.953	0.241	0.190
50	1.180	0.343	0.302

表 3: 不同的 M 取值与串行、朴素并行以及线程池并行时延的关系

分析上表可知、经过了并行优化、并行情况下的平均单次查询时延均较串行方法下的时延得到了较大提升。原因是因为并行优化后、充分利用了多核处理器的并行计算优势。而利用线程池进一步对并行过程进行优化后、单次查询时延也得到了一定的提升。以下是对为什么利用线程池方法进行优化之后的效果更好的简单分析：

1. 线程池的资源利用率更高：线程池可以重复利用线程，减少了线程创建和销毁的开销。
2. 线程池能够进行动态调整：线程池可以根据实际需求动态调整线程数量，更加灵活和高效地处理不同数量的查询请求。因此，当查询请求数量较大时，使用线程池来处理查询请求是更好的选择，可以提高系统的性能和资源利用率。