Lab3-Parallel Query report

张越 522031910791

2024 年 5 月 29 日

1 背景介绍: 并行查询 HNSW

在 lab2 中，要求自主实现了一种基于图的 ANN 索引 HNSW 算法的整体框架以及 insert、query 功能，HNSW 是基于 NSW（Navigable Small World）进行优化的算法，旨在处理高维向量空间的最近邻搜索问题时，能够提供高效的查询性能，同时保持较低的空间复杂度。

NSW 算法的简介如下：将数据库中的向量与接近的向量相连，形成一个连通图。查询过程从这个连通图上的某个起始节点开始，不断跳到更靠近目标节点的邻居节点，直到无法再靠近目标节点为止，得到的终点节点即为查询结果。

HNSW 算法的简介如下，导航过程从入口节点（entry point）开始，在较高层级尽可能向目标节点靠近。如果无法继续靠近，则下降到下一层级的相同节点，直到最终下降到最底层（layer 0）并完成查询。

在实际应用场景中，向量数据库往往面临着大量的查询压力。简单的串行处理将带来用户不可忍受的时延，特别是在短时间内有大量用户同时发起请求的情况下；另一方面，现代服务器均是多核架构，且一般通过集群来更好地提升应用的性能，串行处理请求的　式无法利用拥有的计算资源，将造成极大的浪费。因此，在实际应用场景下一般采用并行处理查询请求的　式。

在这一个 Lab 中，要求使用多线程实现并行 HNSW 查询的功能，并且研究多线程并行对于查找性能的影响。

2 系统实现

简要介绍：在 Lab2 中我们已经实现了 HNSW 的大致功能，在本次 Lab 中对于查询功能而言，我们不需要考虑查询过程中可能出现的数据更新，例如插入和删除等操作，所以我分别使用了 1）使用线程池，2）预先确定用于服务查询请求的线程数量，依据线程数量为每个线程分配一定数量的查询请求这两种并行策略以高效的进行并行查询。

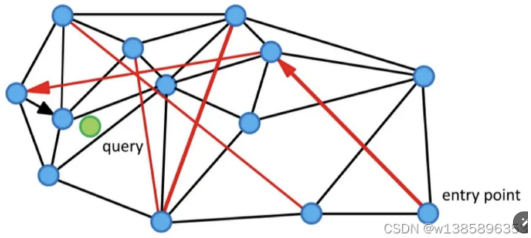

图 1: NSW 示意图

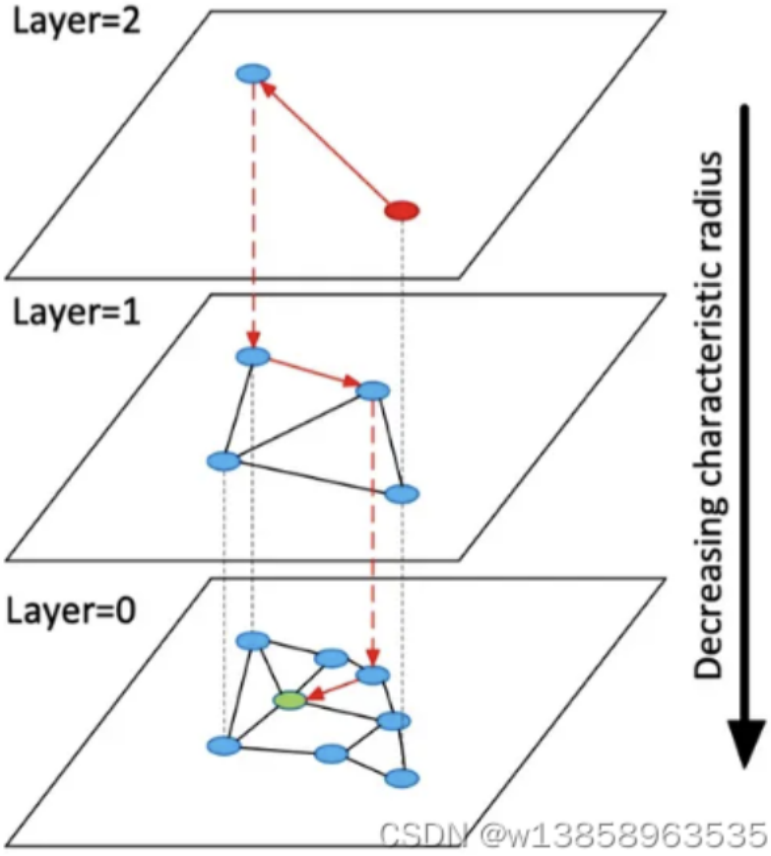

图 2: HNSW 示意图

3 测试

3.1 正确性测试

：可见在参数相同时、使用不同并行策略下查询结果与串行查询的召回率相同、可见并行查询并没有影响查询结果的正确性。

```cpp
class ThreadPool {
private:
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> tasks;

    std::mutex queue_mutex;
    std::condition_variable condition;
    bool stop;

public:
    ThreadPool(size_t threads) : stop(false) {
        for (size_t i = 0; i < threads; ++i) {
            workers.emplace_back([this] {
                while (true) {
                    std::function<void()> task;

                    {
                        std::unique_lock<std::mutex> lock(this->queue_mutex);
                        this->condition.wait(lock, [this] { return this->stop || !this->tasks.empty(); });
                        if (this->stop && this->tasks.empty()) return;
                        task = std::move(this->tasks.front());
                        this->tasks.pop();
                    }

                    task();
                }
            });
        }
    }
```

图 3: 线程池部分代码

```
querying
whole insert time for 10000 times 8969.7 ms
average recall: 0.975, total query time for hundred times 103.3 ms
average recall for Threadpool : 0.975, total query time for hundred times  37.1 ms
average recall for parallel : 0.975, total query time for hundred times  39.6 ms
```

图 4: version1-test

```
querying
whole insert time for 10000 times 9111.0 ms
average recall: 0.942, total query time for hundred times 111.7 ms
average recall for Threadpool : 0.942, total query time for hundred times  31.5 ms
average recall for parallel : 0.942, total query time for hundred times  34.8 ms
```

图 5: version2-test

3.2 性能测试

3.2.1 测试配置

测试对象：HNSW 不同并行线程数量、不同并行策略下的 insert 性能，即不同并行线程数量、并行策略对于 HNSW 算法查询性能的影响。

系统配置: Windows 系统，使用的 IDE 是 Clion , 语言：C++ 17。

机器配置：处理器: 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz

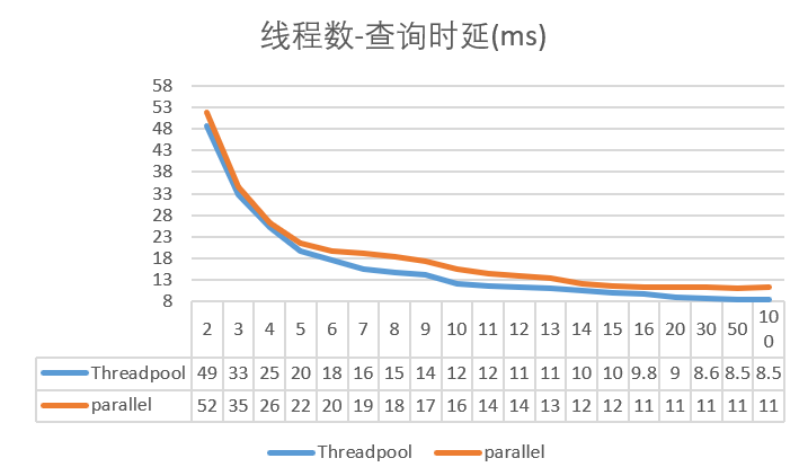

图 6: 并行线程数量-性能分析 (version1)

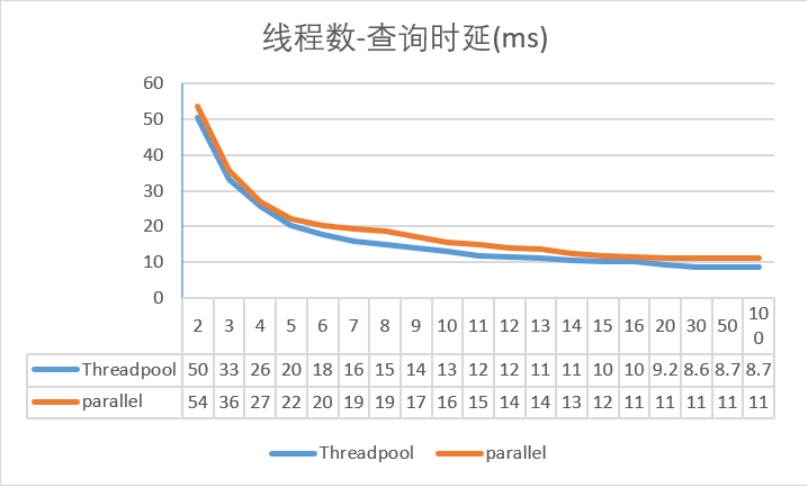

图 7: 并行线程数量-性能分析 (version2)

3.2.2 结果分析

首先研究在同种并行策略下并行线程数量与查询性能的关系，我发现查询性能随着并行线程数量增大而增加（平均时延减小），线程数量在小于 10 的情况下、并行的查询时延大约为串行时延除以并行线程数量，加速比大约为并行的线程数量，这说明并行的效果比较符合理想情况，但是当线程数量大于 16 时，并行线程数量增加无法增加效率，可能是因为我的电脑是 16 线程，超出最大线程优化幅度有限，符合理论。

再横向对比使用线程池和预先确定用于服务查询请求的线程数量，依据线程数量为每个线

程分配一定数量的查询请求这两种不同并行策略的查询性能，发现在同一并行线程数量下线程池的并行查询的性能大于后者的查询性能，这应该是线程切换的效率以及节省线程生成销毁的结果，符合理论。

最后横向分析两个版本的 HNSW 同层情况下的查询性能的差异、通过对比后我们发现在查询效率上、第二个版本的平均查找、插入时延 0.01ms-0.05ms(应该也是微乎其微的区别。。。)落后于第一个版本。。

总而言之，在相同的并行策略下，查询性能随着并行线程数量的增加而提升，在线程数量小于 10 的情况下，查询时延的减少与并行线程数量成反比，显示出理想的加速比。然而，当线程数量超过硬件资源的限制，如达到 16 线程时，进一步增加线程数量并不会带来性能上的提升，这表明存在硬件瓶颈。在比较不同的并行策略时，使用线程池的方法在查询性能上优于为每个线程预分配查询请求的方法，这归功于线程池减少了线程的创建和销毁开销，以及提高了线程切换的效率。

因此在面对大量用户同时的并行请求时，我们可以通过使用线程池很好的增加查询请求的效率，并行化查询操作也可以极好的优化用户的体验。

4 结论

在之前 Lab2 中，我自主完成了 HNSW 算法基本功能的实现，进行了正确性测试以及各种性能测试。而在这次 Lab3 中，我使用线程池和多线程并行优化了 Lab2 中的查询效率，两个 Lab 层层递进，逐步完善了 HNSW 的功能，这个 Lab 也巩固了我对于并行相关知识的掌握，实际应用多线程并行优化了程序的性能，也了解了部分线程池的相关知识。可谓受益匪浅！

5 建议

如果对 HNSW 算法进行优化，我觉得也可以对插入操作进行并行优化。对于查询，无需做额外的改动，因此直接采用并行编程方法将查询函数提交到线程池即可。对于建图的部分，我们可以为每个节点加一个锁，使得多线程并发的时候不会同时将一个节点多次进入构建状态，同时限制并发线程的数量，使得建图的过程相对有序。

附言：总体而言感谢这次 Lab 的经历以及助教老师相助，让我对于并行化编程以及线程池的认识有了进一步的了解。

——由于本人学疏才浅，做这个 Lab2 与这个并行化优化时已经绞尽脑汁、同时又由于最近事务繁多、被 LSM-KV Tree 以及各学科大作业的 ddl 所困、实属分身乏术，上述建议中提到进一步的研究恐怕已是无力为之，只得匆匆为之，若有纰漏之处，恳请助教与老师的谅解。

参考文献