南開大學
Nankai University

计算机学院

并行程序设计实验报告

OpenMP & Pthread 实验报告

马浩祎

学号：2213559

专业：计算机科学与技术

2024 年 5 月 23 日

目录

1 摘要

Abstract

本次实验进行了多平台多框架的多线程编程实验，并进行了多样的探究和全方面的性能分析。首先针对 Pthread 框架，设计了查询内和查询间两种并行化思路的程序并给出对比分析，然后同样在 OpenMP 框架下进行了这两种并行化的尝试，特别地，在查询内部分设计出更适合 OpenMP 循环优化风格的位存储式算法，此外，还探究了 OpenMP 的调度机制的影响，以上实验都利用图表进行了形象的对比并给出分析。而且，上述算法均在 Windows 平台上实验，我还迁移到 Linux 平台进行查询间实验结果的平台差异分析。而针对所有的多线程化实验，我重点探究了线程数所带来的管理调度的开销影响，特别是设置了单线程组，来重点观察与串行算法的性能差异。最终，为了进一步理解 Pthread 和 OpenMP 框架下的编程的底层差异，我利用 Godbolt 进行了汇编代码层级的观察分析。

关键词：多线程编程，Pthread，OpenMP，倒排索引，列表求交

2 实验相关介绍

2.1 问题描述

2.1.1 倒排索引

在当代的数据库里，索引是检索数据最有效率的方式。但考虑搜索引擎的如下特点：

- 搜索引擎面对的是海量数据。像 Google，百度这样大型的商业搜索引擎索引都是亿级甚至百亿级的网页数量
- 搜索引擎使用的数据操作简单。一般而言，只需要增、删、改、查几个功能，而且数据都有特定的格式，可以针对这些应用设计出简单高效的应用程序。
- 搜索引擎面临大量的用户检索需求。这要求搜索引擎在检索程序的设计上要分秒必争，尽可能的将大运算量的工作在索引建立时完成，使检索运算尽量的少。

我们就不能去构建简单的索引了，早在 1958 年，IBM 就在一次会议上展示了一台“自动索引机器”。在当今的搜索引擎里也经常能看到它的身影，它就是——倒排索引。倒排索引又叫反向索引，它是一种逆向思维运算，是现代信息检索领域里面最有效的一种索引结构。

为满足用户需求，顺应信息时代快速获取信息的趋势，开发者们在进行搜索引擎开发时对这些信息数据进行了逆向运算，开发出“关键词——文档”形式的映射结构，实现了通过了物品属性信息对物品进行映射，可以帮助用户快速定位到目标信息，极大地降低了信息获取难度。

2.1.2 索引求交

有了倒排索引后，搜索引擎在查询处理过程中，需要对这些索引集合进行求交操作，这个也是运算时间占比非常大的部分。假定用户提交一个 k 个词的查询，查询词分别是 $t_1, t_2, \ldots, t_k$，这 k 个关键词对应的倒排列表为 $\ell_{t_1}, \ell_{t_2}, \ldots, \ell_{t_k}$ 求交算法返回 $\cap_{1 \le i \le k} l(t_i)$。

首先求交会按照倒排列表的长度对列表进行升序排序，具体原因后面会介绍，使得：

$$|l(t_1)| \le |l(t_2)| \le \cdots \le |l(t_k)|$$

然后求交操作返回 k 个倒排列表的公共元素 $\cap_{1 \le i \le k} l(t_i)$。

2.2 Pthread 简介

Pthreads（POSIX Threads）是一种用于多线程编程的标准 API，通常用于 Unix 和类 Unix 操作系统（如 Linux）。它定义了一组线程创建、同步、销毁等操作的函数，使得程序员能够在应用程序中创建和管理多个线程。Pthreads 库提供了一种在多线程环境中进行并发编程的方式，开发者可以使用它来编写并行化的应用程序。与其他多线程库相比，Pthreads 是一种较低级别的 API，它提供了更多的灵活性和控制，但同时也需要程序员自行处理线程同步、互斥和条件等复杂问题。

对于 Windows 环境，有一些第三方库可以在 Windows 平台上实现 POSIX 线程的功能，其中最常用的是"Pthreads-win32"。这个库提供了一组函数和数据结构，允许在 Windows 操作系统上创建、同步和管理线程，与在 Unix/Linux 系统上使用 Pthreads 类似。

经查阅资料，我总结在进行倒排索引问题的 Pthread 编程时需要注意的问题如下：

- 线程安全：Pthreads 并不会自动保证程序的线程安全性。因此，需要我自行确保在多线程环境中正确地使用互斥锁、条件变量等同步机制来保护共享资源，以防止竞争条件和数据竞争。
- 死锁：死锁是多线程编程中常见的问题，特别是在使用互斥锁和条件变量时容易发生。为避免死锁，我需要仔细设计和管理线程之间的同步和互斥关系，确保线程获取锁的顺序是一致的，避免相互等待对方释放锁的情况。
- 性能优化：在编写多线程程序时，需要考虑到线程的创建、同步和销毁等开销，以及线程之间的负载均衡等问题，以优化程序的性能。避免创建过多的线程或者频繁地进行线程创建和销毁，可以提高程序的效率。

同时，Pthread 对于同步机制并不像 OpenMP 会自动处理，需要我自行选择相关机制并进行比较，结合上课所讲，Pthread 的同步机制有：

- 互斥锁（Mutex）：互斥锁用于保护临界区，一次只允许一个线程进入临界区执行，其他线程需要等待。通过对共享资源加锁和解锁，可以确保在任何时候只有一个线程能够修改共享资源，从而避免竞争条件和数据竞争。
- 条件变量（Condition Variables）：条件变量用于线程之间的通信和同步，允许一个线程等待某个条件成立后才继续执行。条件变量通常与互斥锁一起使用，当某个条件不满足时，线程会进入等待状态，直到其他线程发出信号通知条件已经满足。
- 信号量（Semaphores）：信号量是一种计数器，用于控制对共享资源的访问。通过对信号量进行 P（等待）和 V（释放）操作，可以实现对临界资源的访问控制和同步。
- 读写锁（Read-Write Locks）：读写锁允许多个线程同时读取共享资源，但只允许一个线程写入共享资源。这种锁适用于读操作远远多于写操作的场景，可以提高程序的并发性能。
- 自旋锁（Spinlocks）：自旋锁是一种简单的锁，当线程尝试获取锁时，如果锁已经被其他线程持有，它会一直在一个循环中等待直到获取到锁。自旋锁适用于锁被持有时间很短的情况，避免了线程上下文切换的开销。

2.3 OpenMP 简介

OpenMP 是一种并行编程的 API（应用程序编程接口），用于编写多线程应用程序。它允许开发者在 C、C++ 等语言中添加并行化指令，以便利用多核处理器和共享内存体系结构的性能。OpenMP

提供了一组指令和库函数，用于将任务并行化，并管理线程之间的通信和同步。通过在代码中插入特定的指令，开发者可以将程序中的某些部分标记为可以并行执行的，并且 OpenMP 运行时库会负责在多个线程之间自动分配工作并处理同步问题。这使得开发者能够更轻松地编写高性能并行程序，而无需深入了解底层的线程管理和同步机制。

针对 OpenMP 实验，结合其使用特点，同样需要注意一些问题：

- 数据共享与数据私有性：OpenMP 并行化的一个重要概念是共享内存模型，即多个线程可以访问相同的内存地址。因此，需要确保共享数据的正确性。可以使用 OpenMP 的数据范围指定符（例如 shared 和 private 来控制变量的共享性和私有性，以避免竞争条件和数据竞争。
- 循环并行化优化：OpenMP 最常用的方式是并行化循环。但并不是所有循环都适合并行化，并行化可能会引入额外的开销。因此，在并行化循环之前，需要仔细评估循环的迭代次数、循环体内的计算量以及数据依赖关系等因素，以确保并行化能够带来性能的提升。
- 负载均衡：在使用 OpenMP 并行化程序时，需要确保各个线程之间的负载均衡，即每个线程执行的工作量大致相等。如果某些线程的工作量过大或过小，可能会导致性能下降。可以通过调整任务分配的方法（例如 dynamic 任务分配策略等）来实现负载均衡。
- 线程同步：OpenMP 提供了一些同步机制，如 critical、atomic、barrier 等，用于控制多个线程之间的同步和互斥。需要注意使用这些同步机制时的性能开销和正确性，避免死锁和竞争条件。
- fork-join 机制：如图2.1，OpenMP 会在代码的并行区域开始时创建线程、结束时销毁线程，如果并行区域设计不当，就可能造成巨大开销，对性能有较大影响。

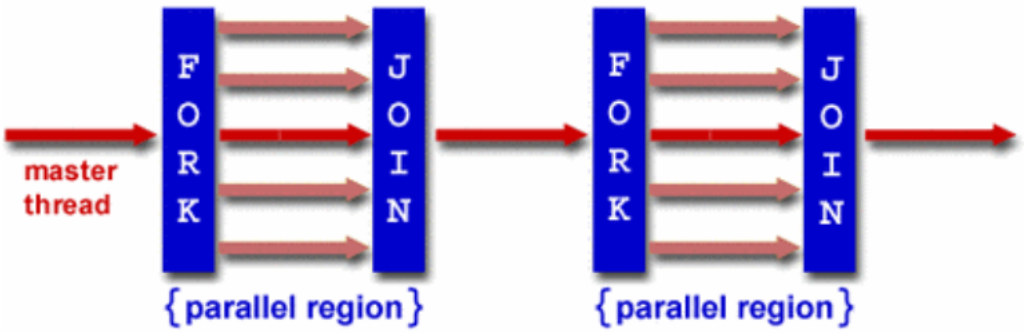


图 2.1: OpenMP 机制

相对 Pthread，OpenMP 其实在同步机制上不用我太多关注，但也是可以进行相关的设置来进行分析理解的！

2.4 实验目的

先前，我进行了 SIMD 并行化的实验，提出了适合 SIMD 并行化的存储形式和算法设计。

在本次多线程实验中，我将分别利用 OpenMP 和 Pthread 库进行实验，考虑 Query 间并行和 Query 内并行两种方式。前者是指每个 Query 还是由单线程处理，不同线程处理不同 Query，优点是

额外开销少，有利于高吞吐率，但不能优化响应延迟；后者是将一个 Query 的处理进行任务划分，由多个线程共同完成，优点是能优化大 Query 的响应延迟，但有额外开销，不利于高吞吐率。

当然，我也会考虑位图存储方式的多线程优化进一步提高我的并行程序设计能力，同时我也将进行以下的实验对比工作并给出分析：

- 在 Windows 下和 Linux 双平台下进行部分实验结果的对比
- Pthread 和 OpenMP 的对比
- 查询间和查询内两种优化方式的对比
- OpenMP 下调度方式的对比
- 多线程数量的对比
- 线程空间和串行空间的对比

总之，针对本次多线程实验，实验目的总结如下：

- 理解并行编程的基本概念
- 掌握多线程编程技术
- 比较不同库的特点和性能极其编程风格
- 培养并行思维和解决问题的能力

2.5 实验环境

Windows 平台下，硬件：CPU：Intel(R) Core(TM)i9-14900HX,24 核，基准速度 2.2GHz，内存容量 16GB，Cache L1: 2.1MB;L2: 32.0MB;L3: 36.0MB。软件：编程 IDE 为 CodeBlocks，编译器为 GNU GCC Compiler，实验用到的汇编分析工具为 Godbolt 网站。

2.6 有关数据/性能

2.6.1 数据集

本问题研究所采用的数据集是给定好的 1000 条查询及 1756 条倒排索引列表，经过初步的数据分析，这些列表包含 DocID 最多 30000 个，最大 DocID 达 25205174，具体详见 github 仓库 src 文件夹下生成的各种数据信息。

2.6.2 性能测试方法

本问题研究的性能测试主要以时间为主，单位均为秒，profileing 部分可能会涉及其他指标，如资源插槽数等。Windows 下时间测试方法如下（Linux 时间测试代码会在后面修改内容部分介绍）：

```
1  LARGE_INTEGER freq, start, end0;
2  QueryPerformanceFrequency(&freq);
3  ...Initial...
4  QueryPerformanceCounter(&start);
```

```
...Function...
QueryPerformanceCounter(&end0);
double elapsedSeconds = static_cast<double>(end0.QuadPart - start.QuadPart) / freq.QuadPart;
```

关于 baseline，我遵循串行最优性能原则，选择按元素求交算法的时间性能，具体见 SIMD 实验报告。

2.6.3 实验正确性

由于并行优化的前提是保证结果的正确性，所以我在问题研究中的相关实验都保证结果是正确的。同时我在 github 仓库中公开了所有本问题研究过程中用到的代码项目，可下载验证结果。而正确结果的生成是由 stl 库 set_intersection 函数生成的标准结果，并保存在 github 仓库 src 文件夹下。

2.6.4 GitHub 仓库

仓库中包含实验涉及的所有程序，所有查询的信息和性能数据，以及整理出来的数据表。
https://github.com/Mhy166/parallel-programming.git

3 Pthread 算法设计

3.1 平凡串行算法

列表求交主要有两种方式: 按表求交，按元素求交。在先前的实验中，我经过性能比较，发现按元素求交算法远胜于按表求交，因此考虑按元素求交的多线程化。同时我根据基础算法设计出更优秀的串行算法，设计特点如下:

- 自适应重排序。算法在每轮迭代后都将针对列表大小进行重排序，选择最短的列表作为基准列表。这样可以在算法进行中动态调整策略，选择较快速的方式。
- 空表提前停止。利用列表升序特点，算法在进行中会逐一删除元素，并且不断检查列表为空的情况，一旦发现则可以直接结束算法，提前停止有利于加速算法执行时间。
- 时间复杂度大致为 $O(ln^2)$, 其中 l 为列表数，n 为列表规模。该复杂度仅作为规模的参考，事实上在实际算法中应用的上述优势可以有效提升实际性能，在结果部分我们会做分析。

下面是本算法的伪代码:

Algorithm 1 元素求交串行算法

Input: $\boldsymbol{l}(\boldsymbol{t}_1), \boldsymbol{l}(\boldsymbol{t}_2), \ldots, \boldsymbol{l}(\boldsymbol{t}_k)$, Sorted by $|\boldsymbol{l}(\boldsymbol{t}_1)| \leq |\boldsymbol{l}(\boldsymbol{t}_2)| \leq \cdots \leq |\boldsymbol{l}(\boldsymbol{t}_k)|$

Output: $\cap_{1 \leq i \leq k} l(t_i)$

1: **function** FUNCTION($\boldsymbol{l}(\boldsymbol{t}_1), \boldsymbol{l}(\boldsymbol{t}_2), \ldots, \boldsymbol{l}(\boldsymbol{t}_k)$)
2: **Input:** 多个列表 `lists`
3: **Output:** 交集元素集合 `res`
4: **while** lists[0] 不是空的 **do**
5: $tmp \leftarrow lists[0].front()$
6: $lists[0].pop\_front()$
7: $cnt \leftarrow 1$

$flag \leftarrow$ false
for $i \leftarrow 1$ to lists.size() $- 1$ **do**
逐一排查其他列表，从表头开始，如果元素小于等于 tmp 则删去，直到大于 tmp
如果有列表为空，$flag \leftarrow$ true
if $flag$ **then**
直接停止算法
end if
end for
if $cnt == lists.size()$ **then**
$res.insert(tmp)$
end if
$lists$ 重排序，始终将最短列表置于第一个列表 $list[0]$
end while
end function

3.2 多线程优化算法——查询内

我针对 Pthread 库的编程特点设计出可以在查询内进行并行化的算法，该算法将最短列表元素划分给各个子线程，同时各个子线程自有一份其他列表的完整备份，随后将结果进行合并，这里的同步机制采用了 Pthread 的互斥锁机制，具体设计特点如下：

- 多线程并行处理。该程序使用 Pthread 库来创建多个线程并行处理任务。每个线程独立处理子任务，提高了处理速度。
- 任务划分均衡。将待处理的最短列表分割为多个子列表，每个线程负责处理一段更短列表和其他列表的求交。这种划分方式有助于均衡负载，避免单个线程负担过重。同时可以做到处理的进一步加速。
- 互斥锁机制保护共享资源。使用 Pthread 互斥锁来保护共享资源，防止多个线程同时访问和修改共享资源全局结果集合，确保数据一致性。processSublist 函数中，在更新全局结果集合时使用 pthread_mutex_lock 和 pthread_mutex_unlock 函数进行锁定和解锁。
- 同步组合逻辑的正确性。每个线程在处理子任务时，会遍历各个列表，找出在所有列表中都存在的元素，并将这些元素加入本地结果集合中。最后，将本地结果集合合并到全局结果集合中。

算法时间复杂度仍然为 $O(ln^2)$, 其中 l 为列表数，n 为列表规模。这里给出多线程算法的关键代码及其详解：

```
//任务划分
list<uint32_t>& first_list = lists[0];
vector<list<uint32_t>> sublists(thread_num);
auto it = first_list.begin();
size_t sublist_size = first_list.size() / thread_num;
//根据线程数进行划分，划分原则是一小段连续的最短子列表
for (int i = 0; i < thread_num; i++) {
```

```
        auto sublist_start = next(it, i * sublist_size);
        auto sublist_end = (i == thread_num - 1) ? first_list.end() : next(it, (i + 1) * sublist_size);
        sublists[i] = list<uint32_t>(sublist_start, sublist_end);
    }
    vector<SubTask> tasks(thread_num);
    for (int i = 0; i < thread_num; i++) {
        tasks[i].lists = lists;
        tasks[i].lists[0] = sublists[i]; // Use only the sublist for the first list
    }
    //线程创建
    vector<pthread_t> threads(thread_num);
    for (int i = 0; i < thread_num; i++) {
        pthread_create(&threads[i], nullptr, processSublist, (void*)&tasks[i]);
    }
    //线程内执行逻辑，执行部分和串行算法类似，这里介绍额外同步部分
    pthread_mutex_lock(&res_mutex);
    //本地结果在互斥机制的保证下安全加到全局结果中
    global_res.insert(local_res.begin(), local_res.end());
    pthread_mutex_unlock(&res_mutex);
    //线程的聚合
    for (int i = 0; i < thread_num; i++) {
        pthread_join(threads[i], nullptr);
    }
```

3.3 多线程优化算法——查询间

类似的，查询内算法可以加快单个查询的响应时间，我还在 Pthread 框架下设计出查询间并行化的算法，该算法不会影响到单个查询的响应时间，但是可以增加总体的吞吐量，子线程内具体执行逻辑和串行算法一致，算法设计难点落在如何分配查询给子线程上。具体设计特点如下：

- 使用多线程处理多个查询，改进了原有的单线程处理单个查询的方法。每个线程负责处理若干个查询，从而提高整体处理效率。在 main 函数中，算法通过循环创建多个线程，每个线程执行 processQueries 函数。该函数也是 Pthread 框架下需要额外编写的一个部分。
- 任务分配：每个线程根据其线程 ID 处理特定范围内的查询，这样可以确保所有查询均匀地分配到各个线程，避免线程间负载不均。在 processQueries 函数中，通过循环条件 i += thread_num 实现线程间任务的分配，每个线程处理间隔为 thread_num 的查询。
- 同步机制：这里的同步机制较为简单，处理列表不需要有额外的开销，比如加锁解锁的时间消耗。这是因为查询间本身就有较好的独立性，在各自备份的列表中进行相关操作而不相互影响，这也是这种并行化思路的一大优点！
- 正确性保证：处理列表前进行所需列表的备份，在其副本上进行操作即可，这样不会影响全局总列表的数据！从而用额外的空间保证了算法正确性。

算法时间复杂度仍然为 $O(ln^2)$, 其中 l 为列表数，n 为列表规模。这里给出算法的关键代码及其详解：

```
    //线程创建
    for (int i = 0; i < thread_num; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], nullptr, processQueries, (void*)&thread_ids[i]);
    }
    //线程聚合
    for (int i = 0; i < thread_num; i++) {
        pthread_join(threads[i], nullptr);
    }
    //子线程处理函数
    void* processQueries(void* arg) {
        int thread_id = *(int*)arg;    //线程 id 传参，以划分任务
        for (int i = thread_id; i < queries.size(); i += thread_num) {//查询分配间隔
            string line = queries[i];   //各自独立负责相应的查询
            vector<list<uint32_t>> lists(queryIndices.size());  //各自备份相应的列表
            for (uint32_t i = 0; i < queryIndices.size(); i++) {
                for (uint32_t j = 0; j < arrays[queryIndices[i]].size(); j++) {
                    lists[i].push_back(arrays[queryIndices[i]][j]);
                }
            }
        //执行逻辑，和串行算法相同，这里不再赘述
        }
        return nullptr;
    }
```

3.4 Linux 平台 Pthread 算法设计

这里针对上述的 Windows 查询间算法进行修改，以适应鲲鹏 Linux 服务器的运行，因为只修改了少量内容，故不再赘述代码，直接介绍修改内容：

- 移除 <bits/stdc++.h> 和 <Windows.h> 头文件, 改用 Linux 平台下的标准库头文件。
- 添加 <sys/time.h> 头文件，替换 Windows 特定的计时方法，使用 gettimeofday() 获取起始和结束时间。
- 创建 getElapsedTime 函数, 用于计算两个 timeval 结构体之间的时间差，以秒为单位返回。
- 其他部分均可保留，在 Linux 和 Windows 是跨平台兼容的。

4 OpenMP 算法设计

4.1 128 位存储串行算法设计

基于 OpenMP 的并行思路特殊性——多在 for 循环处进行并行化。我的串行程序需要有一定修改，对于先前的串行算法，查询间并行仍可沿用，查询内的任务划分则不适合循环并行的处理，因为前述的串行算法求交的代码前后依赖性很强。故考虑 128 位位向量优化，从而构造出适合循环并行的算法逻辑，下面是新的串行算法的设计特点：

- 数据结构的使用：vector<bitset<25205248» bitmaps：存储每个数组的位图表示。这种位图数据结构的使用可以加速按位与的速度，同时大大减少空间的占用。
- 查询处理：每次读取查询文件中的一行，并将其转换为查询索引的向量 queryIndices。对 queryIndices 进行排序，确保较小的集合在前，之后均以其为计算标准，其没出现的更大位为 0，按位与仍然是 0，可以省去不必要的计算，有助于优化后续的位图与操作。
- 128 位分块：将每个位图按 128 位进行分块存储，以便后续进行循环并行操作。同时这样规划还可以为以后的 SIMD 结合研究做好框架基础。

该算法时间复杂度为 $O(ln)$, 其中 l 为列表数，n 为位图规模。我们看一下关键代码：

```
1     vector<bitset<25205248>>bitmaps;
2     //数组存储格式转换为位图存储格式
3     for (uint32_t i = 0; i < arrays.size(); i++) {
4         for (uint32_t j = 0; j < arrays[i].size(); j++) {
5             bitmaps[i][arrays[i][j]] = 1;
6         }
7     }
8     //按位向量长短排序，即列表最后一个元素的数值
9     bool compareBySize(uint32_t& a, uint32_t& b) {
10        return arrays[a][arrays[a].size()-1] != arrays[b][arrays[b].size() - 1] ? arrays[a][arrays[a
11    }
12    sort(queryIndices.begin(), queryIndices.end(), compareBySize);
13    //保存副本
14    vector<vector<bitset<128>>> rebit(queryIndices.size());
15    uint32_t id = queryIndices[0];  //最短列表 id
16    //128 位副本形式保存——第 1 个列表
17    for (uint32_t i = 0; i < arrays[id][arrays[id].size() - 1]; i += 128) {
18        bitset<128> tmp;
19        for (uint32_t j = 0; j < 128; j++) {
20            tmp[j] = bitmaps[queryIndices[0]][j + i];
21        }
22        rebit[0].push_back(tmp);
23    }
24    for (uint32_t i = 1; i < queryIndices.size(); i++) {
```

```
        //其他列表同样进行存储。不再赘述
    }
    //查询执行，即按位与操作，第二层循环可以在未来无依赖式并行
    for (uint32_t i = 1; i < queryIndices.size(); i++) {
        for (uint32_t j = 0; j < rebit[i].size(); j++) {
            rebit[0][j] &= rebit[i][j];
        }
    }
```

4.2 多线程优化算法——查询内

4.2.1 原始串行算法设计

这里我针对最开始的列表式求交操作查询内并行进行了相关思考，设计出一种 OpenMP 风格并行化的设计思路：

- 任务分配：每个线程负责一个列表对应的执行操作。这对于多列表查询来说会有明显的效率提升，但是本数据集最多的一条查询包含列表数为 4，因此可能效果不是很好。所能用的线程数也很有限。
- 同步机制：这里需要用到共享变量，因此在 OpenMP 机制里，我采用 critical 临界区方法来保证区内只有一个进程可以执行，从而保证了线程安全。
- 回写机制：由于 flag 需要全部线程可看，在一个线程进行更改后，需要采用 OpenMP 的 flush 刷新缓存机制同步这个变化。以保证及时停止技术的正确应用。

算法时间复杂度同原始串行算法，仍然为 $O(ln^2)$，其中 l 为列表数，n 为列表规模。核心代码解释如下：

```
    //存储列表的副本，具体存储代码不再赘述
    vector<list<uint32_t>> lists(queryIndices.size());
    //OpenMP 并行化，利用 shared 设置共享变量
    #pragma omp parallel for num_threads(lists.size()-1) shared(flag, cnt)
    for (uint32_t i = 1; i < lists.size(); i++) {
        while (true) {
            if (flag) break; // 如果 flag 为真，说明其他线程负责列表已经执行到空，终止循环
            if (lists[i].empty()) { // 如果当前列表为空
                #pragma omp critical //针对共享变量，保证其不会被同时修改
                flag = true;
                #pragma omp flush(flag) // 刷新 flag 的缓存，确保所有线程都能看到这个变化
                break;
            }
            if (lists[i].front() < tmp) {
                lists[i].pop_front();
            } else if (lists[i].front() == tmp) {
```

```
                lists[i].pop_front();
                #pragma omp critical  //针对共享变量，保证其不会被同时修改
                cnt++;
                break;
            } else {
                break;
            }
        }
    }
```

4.2.2 128 位存储式算法设计

进一步，针对 OpenMP 以循环并行为特性的风格，我设计出 4.1 节的可以更高效适合并行化的算法，因此在 OpenMP 的更改下也比较简单了，同时还可以对预存储部分并行化。实验设计思路如下：

- 位图与操作：使用 OpenMP 并行处理对每个 128 位块进行与操作。通过 OpenMP 的 parallel for 和 schedule(dynamic) 实现动态调度，加速计算过程。
- 任务分配：重点比较线程的数量和 OpenMP 的多线程调度方式，尤其是 dynamic 动态调度和静态调度。
- 并行高效存储转换：还可以在存储格式转换过程进行 OpenMP 并行化，加速离线工作速度。
- 同步机制：OpenMP 给我们自动实现了同步机制，不必由程序员负责，但是其性能影响究竟如何，后面我会进行分析。

算法时间复杂度同位优化算法 $O(ln)$。下面我们看一下 OpenMP 并行化后的关键代码：

```
    int num_th=4;
    vector<bitset<25205248>>bitmaps;
    //数组存储格式转换为位图存储格式，由于无依赖，可以并行化加速。
    for (uint32_t i = 0; i < arrays.size(); i++) {
    #pragma omp parallel for num_threads(num_th),schedule(dynamic)
        for (uint32_t j = 0; j < arrays[i].size(); j++) {
            bitmaps[i][arrays[i][j]] = 1;
        }
    }
    //按位向量长短排序，此部分同上，不赘述
    //保存副本
    vector<vector<bitset<128>>> rebit(queryIndices.size());
    uint32_t id = queryIndices[0];  //最短列表 id
    //128 位副本形式保存——第 1 个列表
    for (uint32_t i = 0; i < arrays[id][arrays[id].size() - 1]; i += 128) {
    //存储形式转换工作也可以循环无依赖并行化
    #pragma omp parallel for num_threads(num_th),schedule(dynamic)
```

```
        for (uint32_t j = 0; j < 128; j++) {
            tmp[j] = bitmaps[queryIndices[0]][j + i];
        }
    }
    //其他列表同样可存储并行化。不再赘述
    //查询执行，即按位与操作，第二层循环可以无依赖并行化
    for (uint32_t i = 1; i < queryIndices.size(); i++) {
    #pragma omp parallel for num_threads(num_th),schedule(dynamic)
        for (uint32_t j = 0; j < rebit[i].size(); j++) {
            rebit[0][j] &= rebit[i][j];
        }
    }
```

4.3 多线程优化算法——查询间

针对查询间并行思路，可以简要针对 OpenMP 的循环并行特性设计出高效适合并行化的算法，实验设计思路如下：

- 任务分配：每个线程负责一个查询对应的执行操作。这对于大量查询来说会有明显的效率提升，本数据集查询数为 1000，因此估计会有不错的效果。所能用的线程数也可以适度增长，对比性能。
- 调度机制：由于每条查询的时间消耗差异较大，因此考虑 OpenMP 特有的 dynamic 机制，该机制可以动态调度查询安排，可以有效保证负载均衡，同时我后面会对比分析不同机制，包括 guided 等的实际性能差异。
- 同步机制：这里用到共享结果，在 OpenMP 机制里，我采用 critical 临界区方法来保证区内只有一个进程可以执行，从而保证了线程安全。

算法的关键代码如下：

```
    #pragma omp parallel for num_threads(numThreads),schedule(dynamic)
    //所有查询并行化
    for (int i = 0; i < queryLines.size(); i++) {
        //保存副本，同 3.1，省略大部分代码
        vector<list<uint32_t>> lists(queryIndices.size());
        while (!lists[0].empty()) {
            //执行流程，同 3.1，这里不再赘述
        }
        #pragma omp critical
        {
            //结果保存
        }
    }
```

4.4 Linux 平台 OpenMP 算法设计

这里同样针对上述的 Windows 查询间算法进行修改，以适应鲲鹏 Linux 服务器的运行，和 Pthread 类似，直接介绍修改内容：

- 移除 <bits/stdc++.h> 和 <Windows.h> 头文件, 改用 Linux 平台下的标准库头文件。
- 添加 <sys/time.h> 头文件，替换 Windows 特定的计时方法，使用 gettimeofday() 获取起始和结束时间。
- 创建 getElapsedTime 函数, 用于计算两个 timeval 结构体之间的时间差，以秒为单位返回。
- 调整 OpenMP 并行部分，确保并行处理与计时操作的正确性。

5 实验结果

关于数据表：所有实验结果均均保留 7 位小数，以 ms 为单位，且仅列举具有规模代表性的部分查询，并给出查询 ID，以确保数据真实性和严谨性。关于数据图，我均以两种形式来呈现，折线图体现性能随规模变化的趋势（为方便观察，制图采用平滑处理，平滑区间为 5）；柱状图体现规模区间内平均性能的对比，这里我按照 5000 为单位分组，一定程度上减少了实验误差的影响。全部相关数据和程序均在 Github 仓库内，读者可以查阅相关程序和查询数据。

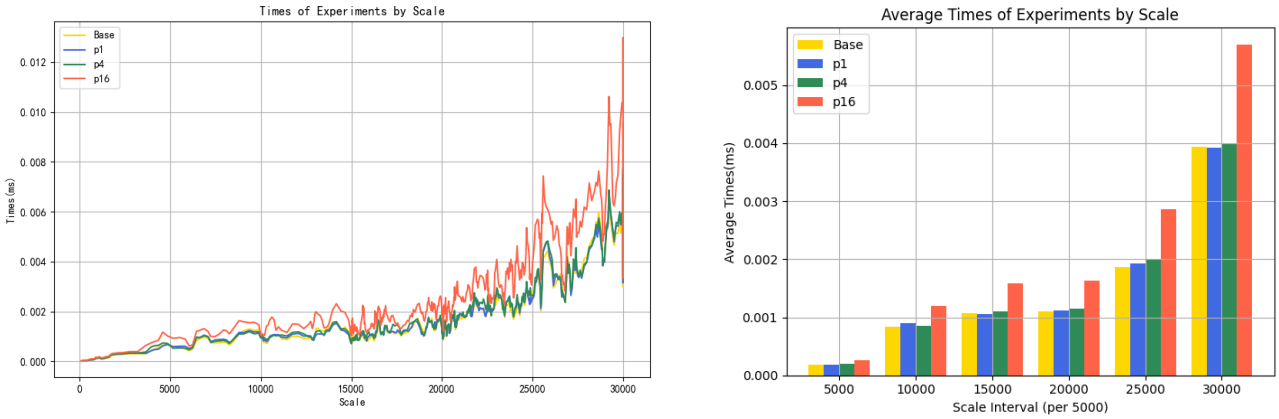
5.1 查询间——综合结果

5.1.1 Pthread 对比分析

本小节主要针对 Windows 平台上的 Pthread 查询间并行化实验进行分析，重点关注线程数的影响。同时以列表平均长度为规模，我测量了查询响应时间和全部查询耗时。由于是查询间并行，预估前者不会有明显差异，后者会有所加速。下面我们看相应数据表1和图5.2.

查询 ID	列表平均长度	串行平凡	p 查询间 1	p 查询间 4	p 查询间 16
648	17	0.0000013	0.0000014	0.0000015	0.0000028
217	595	0.0000364	0.0000355	0.0000350	0.0000548
268	1103	0.0000666	0.0000735	0.0000769	0.0000857
52	5813	0.0002712	0.0002698	0.0003041	0.0003086
504	10049	0.0007197	0.0006913	0.0006733	0.0013567
940	15003	0.0002922	0.0002764	0.0003156	0.0003369
403	20269	0.0008622	0.0009438	0.0008843	0.0018588
312	25096	0.0026108	0.0025390	0.0028068	0.0047374
39	30000	0.0035165	0.0035999	0.0038821	0.0042079
732	30000	0.0055183	0.0062824	0.0060427	0.0067536
Total	:	6.0314200	6.2435400	1.7103400	0.6927290

表 1: Pthread 查询间数据表



(a) Pthread 线程数规模折线图

(b) Pthread 线程数规模平均柱状图

图 5.2: Pthread 查询间实验图表

我根据实验性能数据作出如下解释和分析：

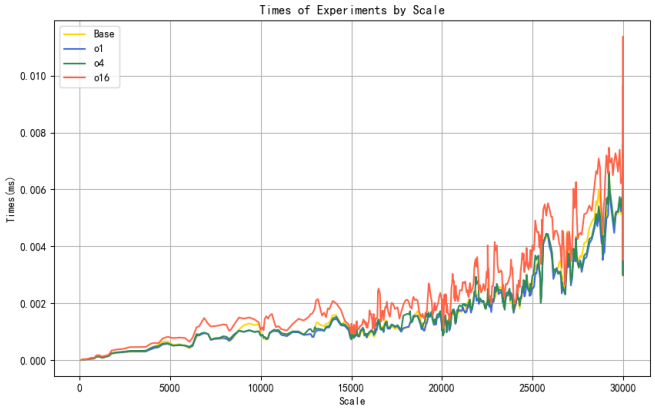
- 一般来说，确实查询间的并行不会影响单个查询的响应时间，从表 1 的总时间也可以看出，4 线程并行的加速比可以达到 3.52，16 线程加速比则为 8.69。可以看到这并不是完全的 4 和 16，甚至后者差距还比较大，这应该是由于随着线程数的增加，程序上下文切换的开销会变大；资源竞争会更加激烈；对于缓存而言，也可能会访问相同缓存从而导致额外开销；16 个线程的管理和调度开销显然是大于 4 线程的。因此导致了上述的加速比结果。
- 这里增加了单线程和串行的对比，结果表明，单线程性能会略微慢一点，这应该是因为使用了 Pthread 库，程序借助它，势必要对这一个线程进行管理和调度，这证明了单线程的时间也并不是和串行完全相等，虽然我们以前可能会这么认为。
- 值得注意的是，单线程和 4 线程的单个查询响应时间增多并不明显，然而 16 线程在查询间并行化的时候严重拖慢了单条查询的响应时间，这也值得我分析思考，我认为这是因为大量的线程，程序为了管理每个线程，其分配到的资源和运行效率必然不如线程少的时候，就像父母看管双胞胎的认真程度必然不如独生子精致。具体来说，可能由于系统硬件配置的限制，使得 4 线程还没有明显对每个线程进行限制，16 线程则不作限制会超过硬件负载能力。
- 由两图，查询时间随规模上升这一点显然容易理解，毕竟需要处理的数据变多了，这里不再过多赘述。

5.1.2 OpenMP 对比分析

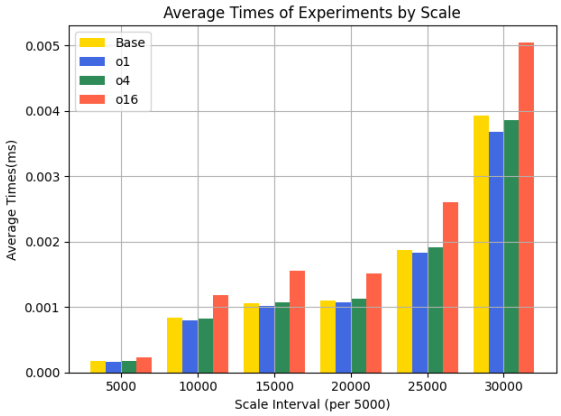
本小节主要针对 Windows 平台上的 OpenMP 查询间并行化实验进行分析，跟 Pthread 分析相似，这里仍然重点关注线程数的影响。同时以列表平均长度为规模，我测量了查询响应时间和全部查询耗时。由于是查询间并行，预估前者不会有明显差异，后者会有所加速。下面我们看相应数据表2和图5.3.

查询 ID	列表平均长度	串行平凡	o 查询间 1	o 查询间 4	o 查询间 16
648	17	0.0000013	0.0000013	0.0000014	0.0000024
217	595	0.0000364	0.0000328	0.0000362	0.0000404
268	1103	0.0000666	0.0000786	0.0000785	0.0001277
52	5813	0.0002712	0.0003071	0.0003046	0.0006261
504	10049	0.0007197	0.0007167	0.0006869	0.0007697
940	15003	0.0002922	0.0002883	0.0003098	0.0003573
403	20269	0.0008622	0.0007764	0.0008794	0.0015797
312	25096	0.0026108	0.0024931	0.0027191	0.0029229
39	30000	0.0035165	0.0034063	0.0035812	0.0039821
732	30000	0.0055183	0.0057386	0.0057420	0.0091781
Total	:	6.0314200	5.8517500	1.5865900	0.6306090

表 2: OpenMP 查询间数据表



(a) OpenMP 线程数规模折线图



(b) OpenMP 线程数规模平均柱状图

图 5.3: OpenMP 查询间实验图表

我根据实验性能数据作出如下解释和分析：

- 同样的，查询间的并行不会影响单个查询的响应时间。从表 2 的总时间可以看出，4 线程并行的加速比可以达到 3.79，16 线程加速比则为 9.52。可以看到这也不是完全的 4 和 16，和 Pthread 一样，这应该是由于随着线程数的增加，程序上下文切换的开销等均会受到影响，16 个线程的管理和调度开销显然是大于 4 线程的。数据也证实了这一点。

- 这里同样观察单线程和串行的对比，我们发现，OpenMP 的单线程性能居然会快于串行算法，这可能是因为使用 OpenMP 时，编译器往往会对代码进行更多的优化。例如，编译器可能会更积极地进行循环展开、矢量化等优化，以提高性能。即使是在单线程的情况下，这些优化也可能带来性能的提升；同时 OpenMP 也会内联一些小的函数调用，减少函数调用的开销，这在单线程的执行过程中也能带来性能提升，而且 OpenMP 的机制大多不用程序员手动操控，其并行化的效果不会受程序员的水平而降低。可以说是一辆高速的自动挡跑车。

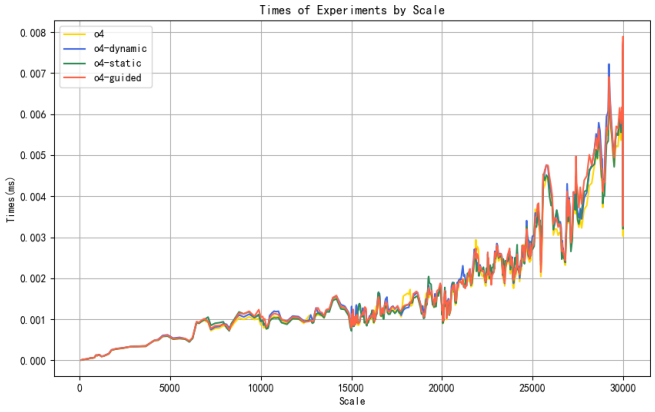
- 我们再来看一下 16 线程的响应时间的增多现象。这确实不罕见了，因为 OpenMP 在管理大量线程的时候也会类似于 Pthread，硬件资源是不会变的，如果超载，必然会进行线程使用资源的限制，从而影响每个线程负责的单查询的响应时间。

5.1.3 OpenMP 调度机制分析

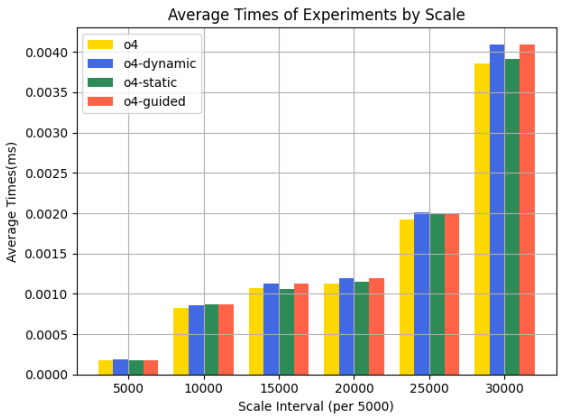
本小节主要针对 Windows 平台上的 OpenMP 特有的几种调度机制进行分析，线程数均设置为 4，因为由前述分析，该设置能较好查询间并行化，同时受硬件资源的限制较小。同时重点关注调度类型的影响，我测量了查询响应时间和全部查询耗时。下面我们看相应数据表3和图5.4.

查询 ID	列表平均长度	o 查询 4	dynamic	static	guided
648	17	0.0000014	0.0000014	0.0000014	0.0000015
217	595	0.0000362	0.0000363	0.0000336	0.0000365
268	1103	0.0000785	0.0000784	0.0000734	0.0000763
52	5813	0.0003046	0.0003031	0.0002912	0.0002895
504	10049	0.0006869	0.0006744	0.0008414	0.0007011
940	15003	0.0003098	0.0003114	0.0002918	0.0004101
403	20269	0.0008794	0.0008931	0.0008509	0.0008700
312	25096	0.0027191	0.0028381	0.0028117	0.0028592
39	30000	0.0035812	0.0038043	0.0050241	0.0037356
732	30000	0.0057420	0.0059923	0.0057723	0.0073153
Total	:	1.5865900	1.5886500	1.6094500	1.5938200

表 3: OpenMP 调度机制数据表



(a) OpenMP 调度机制规模折线图



(b) OpenMP 调度机制规模平均柱状图

图 5.4: OpenMP 调度机制实验图表

我根据实验数据作出如下解释和分析：

- 针对 Total 时间：dynamic < guided < static，可以在如下方面进行分析：
  - 调度机制的负载均衡效果。根据所学，我认为 Dynamic 机制每个线程在完成分配的任务后，可以动态地从剩余任务中获取新的任务，从而更好地平衡负载。这意味着线程几乎没有空闲时间，最大化了并行执行的效率，因此总时间最少。Guided 调度机制开始时分配较大的任务块，随着任务的完成，任务块的大小逐渐减小。这种方法使得任务粒度由粗变细，处理性能有所改善。Static 调度机制在开始时将任务块静态地分配给线程，如果任务的工作量不均匀，某些线程可能在完成任务后空闲，而其他线程仍在工作，这会导致较高的总时间。
  - 调度开销。Dynamic 调度机制通过调度最为频繁，Static 最少，但是根据结果表明，负载不均衡所带来的影响是大于调度开销的影响的。

  - 任务粒度与线程利用率。Dynamic 调度机制分配较小的任务块，线程利用率最高，因为每个线程都能快速获取新任务，总时间最短。Static 调度机制分配固定的任务块，任务块大小不能动态调整，线程利用率最低，总时间最长。Guided 调度机制正如第一条分析，介于二者之间，粒度由粗变细，因此时间介于二者之间。

- 针对单条查询响应时间：static < guided < dynamic。可以看到这刚好与 Total 相反，这应该是因为调度开销方面，Static 调度机制由于没有运行时的调度开销，因此单条查询响应时间最短。Guided 调度机制有一些运行时的调度开销，单条查询响应时间中等。Dynamic 调度机制的频繁任务分配带来较高的调度开销，单条查询的执行经常被调度动作打断，导致单条查询响应时间最长。

- 实验中还有一组数据是 o 查询 4，其为默认调度方式，从柱状图可看出其应该为 static，但是其 Total 时间有所不同，这应该是因为指定调度的 Static 和默认调度仍然有所区别。OpenMP 的编译器在默认 Static 调度时可优化的空间更大，相对指定 Static，可以更好的调度线程，针对特定程序做出最优的微小变化，达到总时间和响应时间的性能最优化。

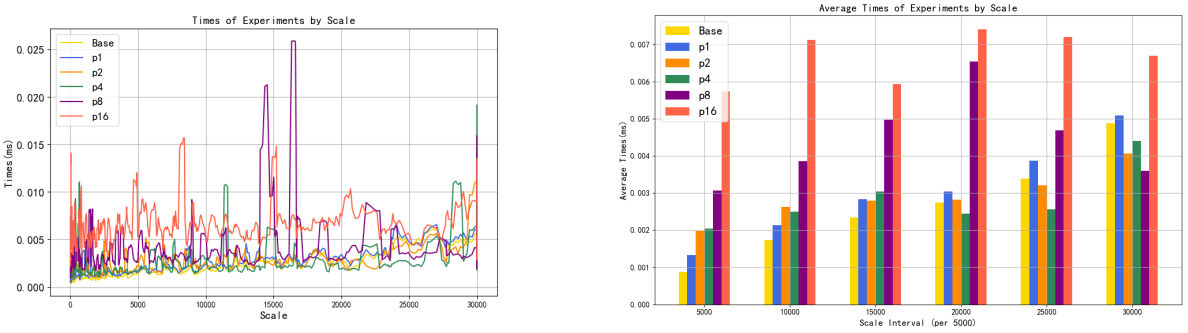
5.2 查询内——综合结果

5.2.1 Pthread 对比分析

在本小节中，由于 Pthread 查询内并行化算法主要针对第一个列表 L1 进行任务划分，因此此处规模标准选取为 L1 列表长度。我仍然在 Windows 平台上进行实验，重点关注线程数的影响，测量了 5 种线程数下的查询响应时间和全部查询耗时。由于是查询内并行，预估前者会有差异，后者也仍然会随之有所加速。下面我们看相应数据表4和图5.5。

查询 ID	L1 长度	串行平凡	p 查询内 1	p 查询内 2	p 查询内 4	p 查询内 8	p 查询内 16
491	21	0.0007788	0.0010431	0.0012420	0.0015329	0.0026160	0.0064642
492	513	0.0008781	0.0012161	0.0011072	0.0014875	0.0018677	0.0067977
20	1077	0.0008255	0.0011185	0.0009497	0.0011115	0.0015316	0.0043803
119	5146	0.0021061	0.0023567	0.0019683	0.0019594	0.0029394	0.0047410
658	10078	0.0011986	0.0016838	0.0023811	0.0015860	0.0021019	0.0053410
891	15291	0.0008319	0.0008731	0.0023628	0.0012801	0.0042634	0.0083365
269	20037	0.0043592	0.0043327	0.0024341	0.0021133	0.0039774	0.0038167
495	25145	0.0046733	0.0053014	0.0032247	0.0023397	0.0033239	0.0058784
12	30000	0.0065814	0.0067887	0.0045761	0.0031788	0.0042750	0.0053642
487	30000	0.0058451	0.0063589	0.0041895	0.0031151	0.0039886	0.0098284
Total	:	6.0314200	11.6971000	20.0715000	21.0001000	35.7401000	122.5510000

表 4: Pthread 查询内数据表



(a) Pthread 线程数规模折线图

(b) Pthread 线程数规模平均柱状图

图 5.5: Pthread 查询内实验图表

观察实验数据，我作出如下解释和分析：

- 从图中可以看出，随着 L1 列表长度的增加，各算法时间也在增加。这是合理的，因为更长的 L1 列表需要在任务划分后子线程的 L1 列表也更长，这需要更多的处理时间。
- 注意到 L1 规模较小时，任何线程数量的并行化查询响应时间都大于串行 base，这是因为 Pthread 在管理线程：线程创建和销毁，线程的调度，缓存友好性等方面都要付出额外的开销。针对设计的算法的同步机制来看，每个线程需要把局部结果同步到全局结果，这部分势必也要造成开销。因此我们看到对比下，单线程在任何规模下都要差于串行算法，原因就在于管理线程的其他开销。这也正是设置单线程组实验的意义。
- 我们注意到当 L1 列表规模增大后，2，4，8 线程数组的响应时间均少于串行算法，可以说达到了并行优化的目的。对此，我认为这是由于 L1 列表规模对响应时间的影响大于了多线程带来的影响。L1 列表规模划分后会变小，如果说本来长度就很小，再分成子段其实没有什么意义，在大规模下我们就看到了划分后所带来的性能提升，尽管多线程的额外开销依然存在。正如图 b 所示，规模再大后，最优的线程数从 4 增为 8，这是验证了上述分析的。更多线程的性能表现会随着规模增大而更得到体现。但是也观察到，16 线程的调度开销实在过大，导致其响应时间普遍高于其他组，但由于本数据集列表规模最大 30000，或许在更大规模的应用场景下，其也有不错的表现。

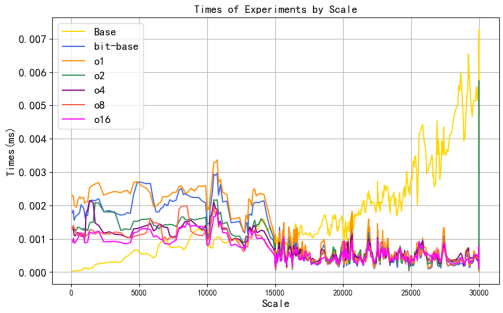
针对算法而言，多线程开销加上主线程的对子线程的副本生成操作的开销，总时间消耗是逐级增长的，但也能看到 2，4 线程之间差别不大，这意味着 4 线程其实是一个良好的折中了。这其实启发我思考更好的并行算法设计来尽可能减少额外操作的影响，我将在期末研究中进一步完善这个子问题。

5.2.2 OpenMP 对比分析

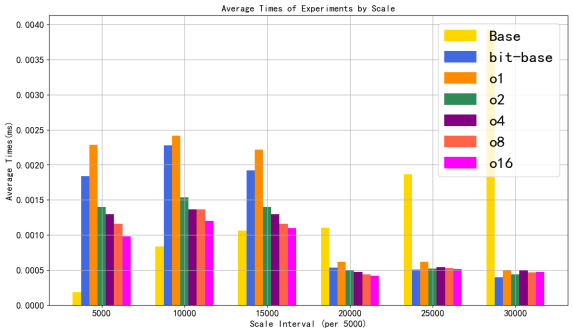
本节实验结果包含 OpenMP 的 4.2 节并行化实验结果和对应串行算法 4.1 的结果。由于 OpenMP 并行风格和 Pthread 差别较大，这里规模标准换回平均列表长度。由于 OpenMP 擅长并行化循环操作，我对原串行算法采取对列表个数进行循环并行处理，这是一组实验。随后我受 SIMD 并行化的启发，在更适合 OpenMP 优化的 128 位存储改良算法上进行循环并行处理，这又是一组实验，这两组实验的 base 性能有不同，我将分别进行介绍，同时用动态调度方式看一下性能影响。

查询 ID	列表平均长度	bit-base	bit-1	bit-2	bit-4	bit-8	bit-16
648	17	0.0012765	0.0016366	0.0011383	0.0011459	0.0007780	0.0005140
217	595	0.0016894	0.0019691	0.0011736	0.0012275	0.0008563	0.0010190
268	1103	0.0014729	0.0020459	0.0011278	0.0008191	0.0009557	0.0008898
52	5813	0.0019593	0.0019112	0.0016410	0.0013504	0.0009843	0.0012243
504	10049	0.0015666	0.0021437	0.0013111	0.0008802	0.0009892	0.0009609
940	15003	0.0008269	0.0009160	0.0008017	0.0005707	0.0005593	0.0003695
403	20269	0.0000680	0.0000838	0.0000942	0.0001070	0.0001651	0.0001918
312	25096	0.0002441	0.0002578	0.0002690	0.0002140	0.0002045	0.0003864
39	30000	0.0000718	0.0000751	0.0002616	0.0001660	0.0002068	0.0001606
732	30000	0.0004461	0.0005119	0.0003690	0.0006219	0.0003813	0.0008605
Total	:	0.4846631	0.5765380	0.5036834	0.4922711	0.4761031	0.4610397

表 5: 存储优化查询内数据表



(a) OpenMP 线程数规模折线图

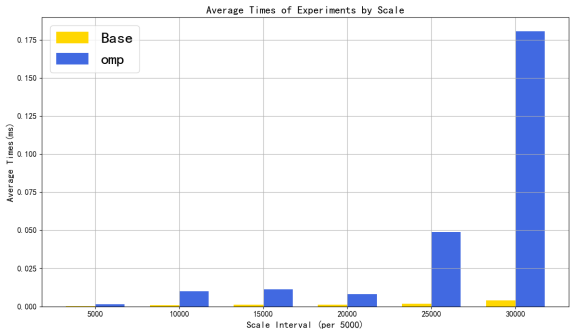


(b) OpenMP 线程数规模平均柱状图

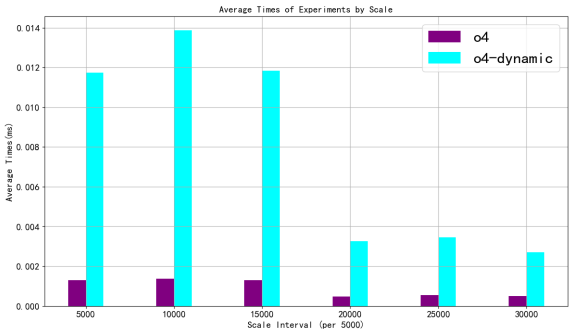
图 5.6: OpenMP 线程数实验图表

从表5和图5.6可以看出，随着规模增长，位存储优化的算法越发优于原始串行算法，这意味着位运算起到了至关重要的作用。就并行线程数的影响而言，OpenMP 的实验结果几乎保持一样，在规模小的适合呈现正相关，规模稍大则几乎持平，但考虑到多线程带来的开销，我们可以认为并行化的性能正向影响弥补了部分开销。仍然要注意到单线程的性能依然大于新串行算法，具体的解释在前文有所介绍，这里不再赘述。

下面我们来简单看一下原始串行算法优化的结果和在调整为 OpenMP 特有的动态调度机制后的实验结果，即图5.7和表6：



(a) 原始串行并行化平均性能图



(b) 动态调度平均性能图

图 5.7: OpenMP 原始并行化 + 动态调度对比图

查询 ID	列表平均长度	串行平凡	omp	bit-4	bit-4-dy
648	17	0.0000013	0.0000087	0.0011459	0.0083104
217	595	0.0000364	0.0001015	0.0012275	0.0098479
268	1103	0.0000666	0.0002175	0.0008191	0.0124589
52	5813	0.0002712	0.0002917	0.0013504	0.0089567
504	10049	0.0007197	0.0032053	0.0008802	0.0132281
940	15003	0.0002922	0.0005325	0.0005707	0.0034747
403	20269	0.0008622	0.0011307	0.0001070	0.0004997
312	25096	0.0026108	0.0386100	0.0002140	0.0010666
39	30000	0.0035165	0.0574667	0.0001660	0.0006439
732	30000	0.0055183	0.3657890	0.0006219	0.0036179
Total	:	6.0314200	143.7269940	0.4922711	3.1513376

表 6: 原始并行化 + 动态调度数据表

显而易见，我们可以做出如下分析：

- 原始串行算法本身不适合 OpenMP 的并行优化风格，即时对列表个数循环并行后，算法中仍然不得不出现很多共享变量和临界区，这导致了开销的异常增大，同时数据集里列表数不多于 4，这意味着并行线程数不会多于 3，并行能力受到极强的限制。因此其性能远差于 Baseline，总时间都是 Baseline 的 20 倍以上了。
- 换用 dynamic 后发现性能显著变差，仔细思考算法，这便很好解释：对于位存储算法的查询执行过程不过是以 128 位为步长的循环，负载本身很好的均衡了，当选择 dynamic 机制后，开销的增大却换来很小的效果，这确实在图 b 中表现为得不偿失。同时注意到随规模增大，dynamic 的性能还有所好转，这可能是因为较大的任务可能更好地利用缓存，因为较长时间的计算可以更有效地利用数据局部性，减少缓存失效带来的性能损失。

总之，这两组对比实验告诉我们，并行算法的设计需要考虑并行框架的风格，这是没有绝对的适配的；对于某些框架，任务的调度方式的选择需要因算法而异，这也是没有绝对的完美的。

5.3 鲲鹏 Linux 平台综合结果

本节介绍我在鲲鹏下的查询间并行化实验结果，这里只选取前 10 个查询进行分析，可以证明实验数据正确性的内容均以图片形式存在仓库中。这里我给出数据表7和图5.8：

查询 ID	列表平均长度	Linux-base	p1	p4	p16	o1	o4	o16
1	20089	0.001876	0.001254	0.00165	0.001689	0.001635	0.001376	0.001388
2	25584	8.04011	7.71275	9.6824	8.96855	8.17854	8.46842	10.6041
3	30000	7.02053	6.45913	6.66894	6.76398	6.86832	6.98185	7.14555
4	30000	17.0333	16.7455	17.8667	17.0868	16.9275	18.2611	19.4129
5	22645	1.34108	1.2768	1.25335	1.19838	1.30836	1.24319	1.17933
6	20285	0.015731	0.015456	0.014101	0.016355	0.015285	0.014024	0.015189
7	15631	0.039819	0.040368	0.03761	0.036347	0.03979	0.036425	0.041517
8	30000	1.21957	1.16544	0.979031	1.04569	1.19429	0.86269	1.20234
9	9962	0.298931	0.282141	0.244033	0.259403	0.291747	0.246344	0.245191
10	30000	7.10728	6.98095	5.21149	5.27237	6.64108	5.6357	5.33518
Total	:	42.1471	40.7105	18.8594	17.0974	41.5035	19.5289	19.4247

表 7: Linux 综合实验结果

这是 Pthread 和 OpenMP 整体的数据展示，简要观察可以得到如下分析：

- Pthread 和 OpenMP 的并行加速效果：经过计算（Total），以 4 线程为例，Pthread 为 2.23，OpenMP 为 2.15。16 进程并没有多大变化。没达到预期的 4 或者 16，应该是因为多线程的管理开销较大，同时硬件资源分配受到限制，因为单线程和串行的对比我们可以看出前者性能都有微妙的提升，说明其能更好地利用硬件的资源。当然 16 进制的效果提升不显著，这可能是因为服务器的总线程管理有一定的限制，当然，最客观的我认为还是大量线程导致有关上下文切换的开销，缓存利用不充分的缘故。
- 观察响应时间图，我们也可以看出，因为查询间并行不会显著影响单条查询的响应时间，但是还是有些许差异。显然，Linux 平台上，对比之下 OpenMP 在更大量线程的处理上差于 Pthread，经过学习，我认为这可能是两者的抽象层次不同。OpenMP 提供更高层次的抽象，通过编译器指令来自动管理线程的创建、同步和销毁。这种高层次的抽象虽然简化了编程，但也限制了对线程的精细控制。Pthread 是低层次的线程库，提供了更多的控制和灵活性，允许程序员手动管理线程的生命周期和同步机制。因此，Pthread 更适合需要精细控制线程行为的应用。

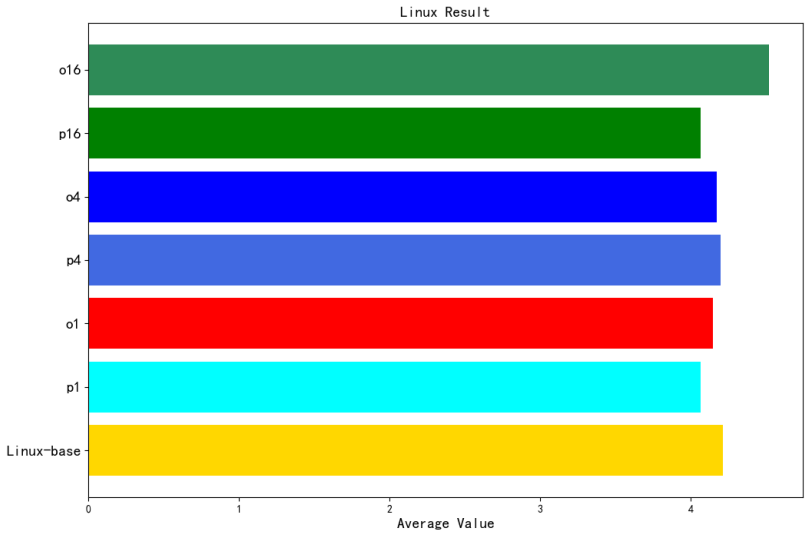


图 5.8: 响应时间性能图

5.4 综合对比分析

5.4.1 多编程库分析

根据上述实验结果，这里给出 Pthread&OpenMP 的对比分析，我们以 Windows 上查询间的实验为例，因为两者算法差异不大，较能体现不同编程库的差异，而且更能通过响应时间体现子线程内部资源受限的情况，见图5.9. 可以得出：

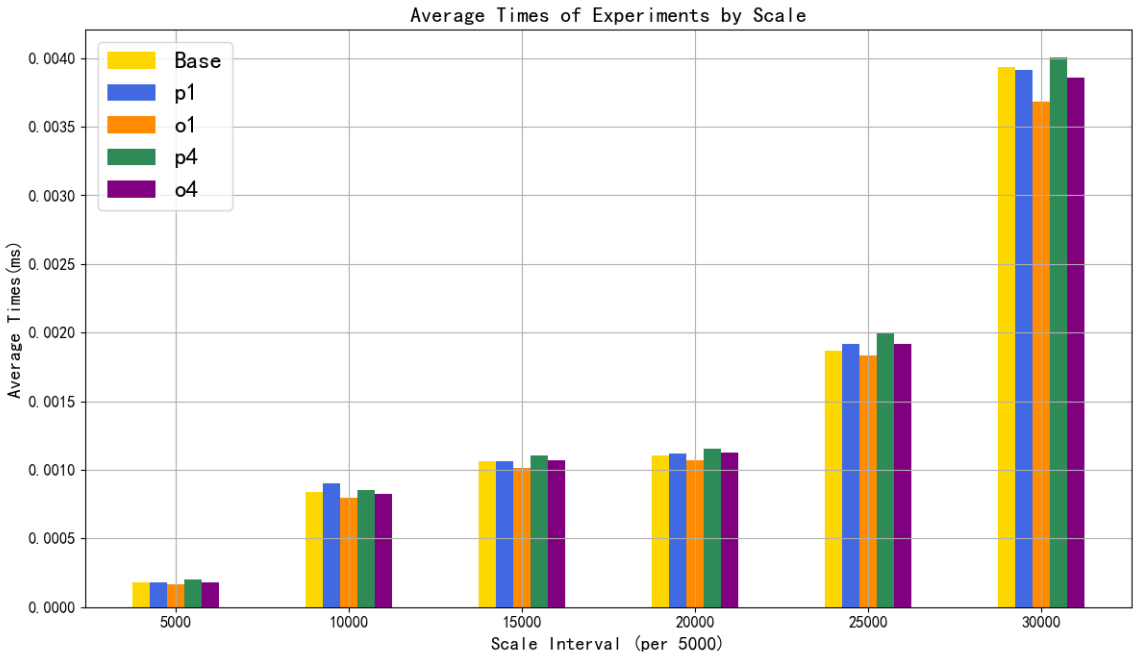


图 5.9: Pthread&OpenMP 性能图

- 根据前文的加速比，两者加速比均为 3.5 左右，这意味着两者都可以创建和管理多线程程序，以提高应用程序的并行度和性能。且本次实验也进行了跨平台编程，修改很少，这也意味着两者都支持多种操作系统和硬件平台，具有良好的跨平台兼容性。
- 随规模增大，可以发现二者管理多线程开销都会增大，这在前文有所分析不再赘述，值得分析的是，Pthread 的内部线程的响应时间要略大于 OpenMP：
  - Pthread 提供了对线程的低级控制，允许开发者手动管理线程的创建、同步、通信和销毁，因此其很灵活，适合于需要精细控制线程行为和资源管理的高性能计算场景。但是它复杂性也高：由于需要手动管理线程和同步，编程相对复杂，容易出现竞争条件问题。
  - OpenMP 采用编译指令的方式，它可以自动管理线程的创建和同步。易于使用，降低开发难度，适合快速实现并行化的应用。同时它运行时系统负责任务调度和负载均衡，我无需手动管理线程，但这也限制了对线程的精细控制。
  - 于本问题而言，由于数据集规模限制和算法的简便易行，这并不是一个高性能计算应用，因此 Pthread 造成的同步复杂性略大于 OpenMP。我们较少受益于 Pthread 灵活性的优点，但其复杂性使得子线程更受系统资源的限制，我手动同步等的操作比 OpenMP 自动同步要低效一点；也可能因为 OpenMP 对共享缓存的调度利用要优于 Pthread，因此造成了图中的微小响应时间差异。

5.4.2 多平台对比分析

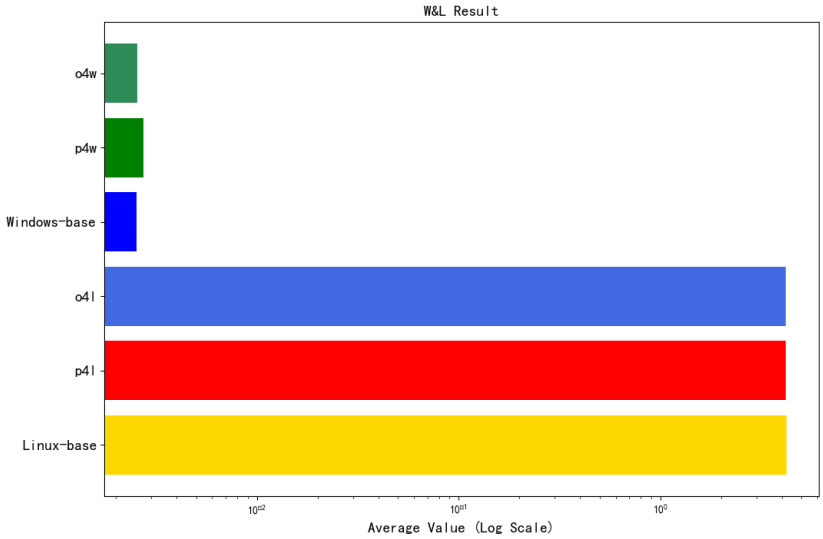


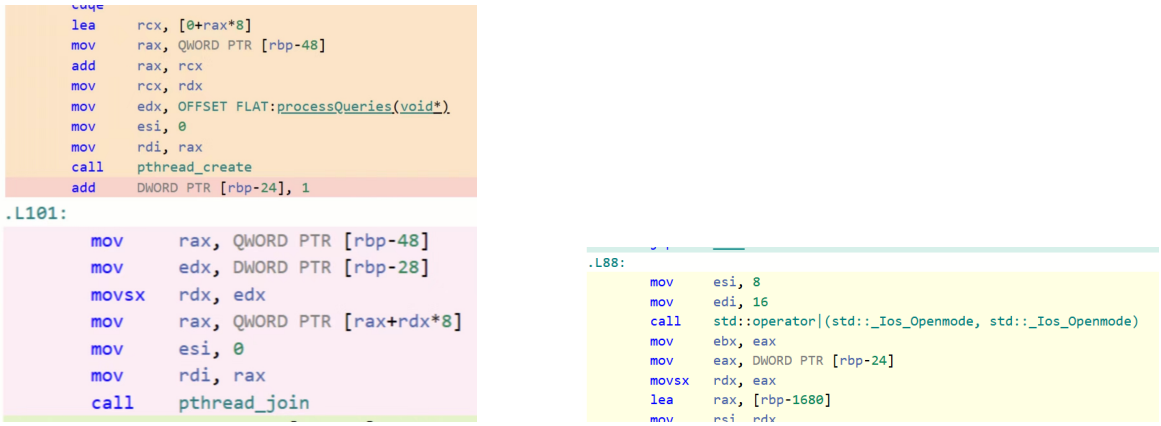
图 5.10: Windows&Linux 性能图

平台内的具体差异在前文已经有所分析，这里观察到 Windows 平台性能显然优于 Linux 平台性能，下面尝试对此进行理解和分析，见图5.10：

- 平台内核与系统优化的差异。我们知道 Windows 和 Linux 的内核调度策略不同。在本任务场景中，Windows 的调度策略可能更优，从而表现出更好的性能。再考虑到某些系统参数的配置，Linux 系统需要针对特定算法进行优化配置。例如，文件系统参数、内存管理参数等。如果未进行针对性优化，可能导致性能不如 Windows。
- 文件系统差异。Linux 和 Windows 使用不同的文件系统（如 Linux 使用 ext4、btrfs，Windows 使用 NTFS）。不同文件系统在特定操作下的性能表现不同，可能导致 Linux 在本问题这种 I/O 密集型任务中性能不如 Windows。
- I/O 性能差异。Linux 和 Windows 在处理 I/O 请求时采用不同的调度策略和缓存策略，这也会影响性能表现。
- 系统调用函数的差异。Linux 和 Windows 的系统调用机制不同，某些系统调用在 Linux 上可能比 Windows 更耗时，特别是在频繁调用系统资源的情况下。
- 内存管理的差异：Linux 和 Windows 的内存管理策略存在差异，在特定负载下，Linux 的内存管理机制可能表现不如 Windows。同样的，高速缓存的差异也可能会影响到其性能。

6 Profiling 汇编分析

接下来为探求更硬件层次的多线程汇编代码的差异，我针对 Windows 的 GCC 编译器下 Pthread 和 OpenMP 的汇编代码进行查看并得出分析，见图6.11：



(a) Pthread 关键汇编代码 (b) OpenMP 关键汇编代码

图 6.11: 不同框架的汇编代码对比图

- Pthread 底层汇编进行了线程创建与管理的显式调用，如图中 pthread_create 函数，pthread_join 函数。Pthread 编程结构下，汇编层级是会将参数压入堆栈，然后调用相应库函数的。但是由于其并行编程的风格，需要额外写每个线程处理查询函数的代码，因此其压参数的时候我们也看到了对应的执行函数 processQueries，这是符合理论的。
- OpenMP 底层汇编则未出现显式的调用，转而将 OpenMP 指令在编译时转换为相应的并行结构，即在代码整体结构上进行并行化，当然这应该是图中那句 std::operator|(std::_Ios_OpenMode，std::_Ios_OpenMode) 所起的作用，在该调用函数内部做了相关工作而使每个线程都执行下面对应的代码。

7 总结

本次实验是倒排索引问题的多线程优化实验，也作为期末研究的一个子实验，主要有以下收获：

- 实现了两种优化方式：查询内和查询间。增强了我在不同编程风格下的程序撰写能力。
- 实现了 Windows 和 Linux 双平台下的查询间实验结果的对比，并给出几个致使性能差异的平台因素。
- 进行存储格式的转换，实现适于 OpenMP 循环并行的位向量查询算法，并对其开销和优化能力进行了分析。
- 探究了 OpenMP 调度方式的差异，尤其是动态调度方式，并给出了默认调度和指定三种调度方式的性能分析。
- 探究了多线程下线程数所带来的管理调度的开销影响，特别是设置了单线程组，来重点观察与串行算法的性能差异。
- 在 Pthread 和 OpenMP 两种多线程并行框架下进行对照实验，熟练使用各自特有的机制，并能尝试在多层面上分析原因，包括汇编代码层级。

总之，这次实验帮我打好了未来子问题研究和期末研究的基础，比如 MPI 编程中可沿用多线程的并行化思路。让我掌握了不同多线程库的编程风格和性能影响，增强了我在多平台多框架下的并行编程能力。