
南開大學
Nankai University

计算机学院

并行程序设计实验报告

# 并行加速的倒排索引问题研究

马浩祎

学号：2213559

专业：计算机科学与技术

2024 年 7 月 1 日

# 目录

# 1 摘要

**Abstract**

本次实验是期末研究的综合实验。首先进行了串行的多种策略尝试，并结合早停技术进行优化。随后针对查询间并行思路进行多种并行技术的尝试，包括 **Pthread, OpenMP, MPI, SYCL, CUDA**，并对比分析其性能异同。类似的，我还实现了查询内并行思路的多线程多进程实验，并结合两种并行思路，实现查询间和查询内结合的单并行策略。然后我在新的位图存储形式下进行新的串行策略的尝试，并对其进行优化，在此基础上对其进行多种并行技术的实现，此处又包含了 SIMD 的技术。最终我融合多种并行技术，分别在**查询内，查询间，数据结构**三方面进行优化，得到目前加速比最好的程序。在性能分析方面，我采用多方面的分析方法，充分挖掘**规模走势，平均性能，汇编代码以及运行的动态数据**等信息，并对实验结果做出合理的解释和分析。

**关键词：**倒排索引，列表求交，并行加速，并行技术融合

# 2 课程学习回顾

本学期的并行程序设计课程，我在王老师的带领下系统了解了当前的业界前沿技术，学习实践了一些能够显著提高程序运行效率的并行化编程思想和技术，具体包括以下几个方面：

- **并行技术前沿**：我理解了并行计算机体系结构的基本知识，包括现代计算机体系结构的基本特点以及对程序性能的影响；了解了并行计算机核心的并行执行控制机制和互连网络拓扑结构；尤其是，了解现代超算的结构和其中的关键并行技术。
- **体系结构相关知识**：我深入了解了计算机体系结构的基础知识，尤其是缓存的内容。鉴于缓存的局部性特征，我学习到通过适当的更改循环策略，可以提高访问数据的效率。而且更为关键地，我初步接触了任务划分、负载均衡、通信优化和同步机制等并行算法设计原则。
- **SIMD 基本知识**：我学习了 SIMD 的基本概念和架构，熟悉了常见的 SIMD 指令集（如 Intel 的 SSE、AVX 和 ARM 的 NEON），并掌握了向量操作和数据对齐的技巧，掌握了编写相应优化程序的技能。
- **多线程基本知识**：我学习了 Pthread 的基本概念和创建、管理和终止线程的方法；掌握线程同步机制如互斥锁、条件变量和读写锁；理解线程间通信和数据共享的实现方法。对于 OpenMP 而言，我学习了 OpenMP 编译指令、运行时库函数和环境变量的使用；掌握并行区域、工作共享构造（如并行 for 循环）和同步构造（如临界区、屏障）的实现方法。以上两种多线程编程库，我也都掌握了相应的编程技巧，并自我探索了 OpenMP 库的调度特性。
- **MPI 基本知识**：我学习了 MPI 的基本概念和创建、管理和终止通信区域的方法；掌握进程间通信的机制和方法，并通过实际操作熟练了 MPI 的基本编程技术，自我探索了阻塞通信和非阻塞的机制差异。同时我也调研了解了分布式训练领域的组通信技术，了解了 HPC 的引入和算法，工具的发展。
- **GPU 基本知识**：我理解了 oneAPI 的统一编程模型及其跨架构支持，尤其是 SYCL 编程框架。在 Intel Devcloud 平台上我具体学习了 DPC++ 的语法和编程模型，掌握内存管理和性能优化工具如 Intel VTune Profiler；在课堂上，我还学习了 CUDA 编程模型和 GPU 架构，理解 GPU 和 CPU 的通信机制。

# 3 研究问题描述

## 3.1 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。

倒排索引，又名反向索引、置入文档等，多使用在全文搜索下，是一种通过映射来表示某个单词在一个文档或者一组文档中的存储位置的索引方法。在各种文档检索系统中，它是最常用的数据结构之一。

那么既然称其为倒排索引，反向索引，对应的，我们需要额外理解一下什么是正向索引。所谓正向索引，就是当用户发起查询时，搜索引擎会扫描索引库中的所有文档，找出所有包含关键词的文档，这样依次从文档中去查找是否含有关键词的方法。那很显然，这个搜索量是巨大的，算法效率低。

再来清楚的看一下两者的索引结构：

- 正向：文档-> 单词、单词、单词……
- 反向：单词-> 文档、文档、文档……

我们假设有一个数据集，包含 n 个网页或文档，现在我们想将其整理成一个可索引的数据集。我们当然可以按照正向索引，逐一遍历网页，为其内部的单词建立索引，但是正如前文所说，这样效率低下。具体地，如图3.1所示，利用倒排索引，我们可以为数据集中的每篇文档选取一个文档编号，使其范围在 [1,n] 中。其中的每一篇文档，都可以看做是一组词的序列。则对于文档中出现的任意一个词，都会有一个对应的文档序列集合，该集合通常按文档编号升序排列为一个升序列表，即称为倒排列表（Posting List）。所有词项的倒排列表组合起来就构成了整个数据集的倒排索引。这和我们的索引结构是相符的。

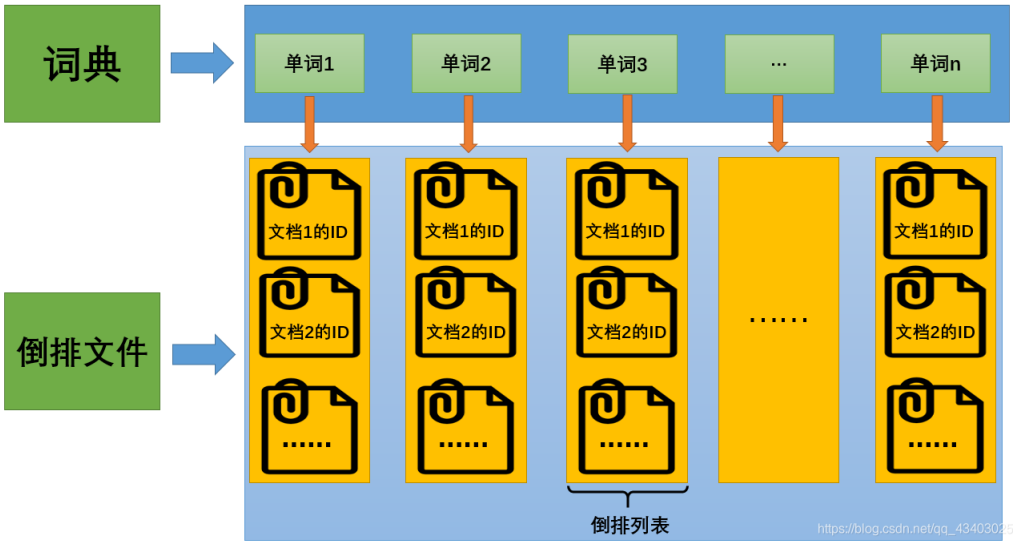


图 3.1: 倒排索引原理

## 3.2 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，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 $\cap_{1 \le i \le k} l(t_i)$。

举个例子：比如查询 “2024 Spring ”，搜索引擎会在索引中找到 “2024”，“Spring” 对应的倒排列表，假定为:

$$l(2024) = (13, 14, 15, 16, 17)$$

$$l(Spring) = (4, 8, 11, 13, 14, 16)$$

这是排好序的倒排列表，下面求交获得：

$$l(2024) \cap l(Spring) = (13, 14, 16)$$

但是这里只给出了输入和输出。如何进行求交操作并未提出，而如何设计出高效加速的并行求交算法正是我研究的问题关键。

## 3.3 研究历史简述

求解多个有序列表的交集是一个经典的问题。该问题的研究历史涉及面较广：串行，多 CPU 并行，GPU 并行，有序无序列表，哈希等工具的使用优化。早在 1970 年，Hwang 和 Lin[3] 提出了两个有序数组的求交串行算法，其时间复杂度为 $O(m+n)$，其中 $m$ 和 $n$ 分别为两个有序数组的元素个数。随后，Demaine[1] 将这一算法推广到多个有序数组，并提出了高效的自适应算法。下面我从两个并行方面介绍相关研究。

#### 3.3.1 多 CPU 并行算法

2009 年，Tatikonda[5] 提出了针对多 CPU 的两级并行算法，包括查询内并行化和查询间并行化。查询内并行化在单个查询中对索引列表求交进行并行计算，而查询间并行化在多个查询之间进行并行处理。同年，Tsirogiannis[6] 提出了动态探测算法，该算法在运行时利用以前的探测信息动态决定列表上的探测顺序，同时提出了一种基于哈希的算法来处理未排序列表。这些算法都利用了 CPU 的多级缓存，具有负载均衡的特点，降低了访问延迟。

2011 年，Schlegel 等人 [4] 利用 SSE 指令集和高效 SIMD 指令来优化并行算法，提出了针对不同整数数据类型的排序集交集算法，并通过定制数据布局进一步优化性能。

#### 3.3.2 GPU 并行算法

在 GPU 并行计算方面，2008 年，Ding[2] 提出了基于 CUDA 的查询处理架构，尽管该架构在小规模倒排索引上效果不佳，但在大规模查询中表现出色。2009 年，南开大学和百度联合实验室 [7] 提出了一种将查询分批提供给 GPU 的算法，通过输入预处理方法解决负载不平衡问题，提高了算法效率。2011 年，该团队 [8] 进一步提出了利用 Bloom Filter 进行 GPU 集合求交的方法，尽管存在一定的误判率，但在吞吐量上表现优异。

### 3.4 研究目的

#### 3.4.1 子问题系列研究

本小节主要简述先前四次并行子实验的研究内容。这些实验为本次的期末综合研究打下了坚实的基础，也提供了启发式的思路。

- SIMD 实验：该实验主要设计了两个基本串行算法，并给出性能分析，做出了 baseline 的选择。并考虑了适于 SIMD 指令集结合的算法改进模式，初步设计出可以嵌入 SSE,AVX 的 128 位分解的算法，真正实现 SIMD 的并行计算。
- Pthread&OpenMP 实验：该实验主要设计了多线程下两种并行化策略：查询间，查询内。实验分析了本机器环境下较佳的线程数量，能达到并行加速和线程开销的平衡，并重点探索了 OpenMP 的调度机制的差异。
- MPI 实验：该实验主要设计了多节点下两种并行化策略：查询间，查询内。实验分析了本机器环境下较佳的节点数量，能达到并行加速和通信开销的平衡，并重点探索了阻塞通信和非阻塞通信的机制差异。
- GPU 实验：该实验主要学习了平台下的 SYCL 的结构，内存，归约等重要机制，并设计了在 SYCL 框架下查询间并行化的策略，见识到了 Intel(R) Data Center GPU Max 1100 的高速运算性能。

#### 3.4.2 期末研究目的

基于 3.4.1 节的子实验研究，期末研究将采取统筹把握，整体优化的总方针！主要目的如下：

- 采用不同串行编程策略，并分别进行并行化尝试。
- 采用不同存储形式编程策略，进行并行化尝试。

- 针对查询间和查询内两种形式分别进行多种并行技术的研究。
- 实施双级查询，设计查询间和查询内同时并行化的策略研究。
- 全面整合学期所学，在最优基线上设计同时包含多种并行技术的高速策略。
- 全面掌握 SIMD，多线程，多进程，GPU 等不同并行技术的编程风格和实现特点。
- 强化并行思维，提高我利用多种并行技术编程解决实际问题的能力。

# 4 并行背景知识 & 实验相关

## 4.1 并行背景知识介绍

### 4.1.1 SIMD 介绍

SIMD（Single Instruction, Multiple Data）是一种并行计算架构，允许一条指令同时对多个数据进行操作。SIMD 广泛应用于多媒体、科学计算和图像处理等领域。典型的 SIMD 指令集包括 Intel 的 SSE（Streaming SIMD Extensions）和 AVX（Advanced Vector Extensions）以及 ARM 的 NEON。通过利用 SIMD 指令，程序可以在一个时钟周期内处理多个数据，从而显著提高计算效率。关键技术包括向量化编程、数据对齐和内存布局优化。

在搜索引擎的倒排索引求交问题中，SIMD 可以通过一次性对多个文档 ID 进行并行比较和处理，显著提高查询速度。例如，在求交操作中，可以利用 SIMD 指令集同时比较多个文档 ID 集合，快速筛选出共同出现的文档 ID。

接下来我简要介绍一下本研究所涉及到的 SSE 和 AVX 扩展的异同：

- **相同点**
  - 都是用于 SIMD 的指令集扩展，支持并行处理多个数据。
  - 都由英特尔开发并广泛应用于现代处理器中。
- **不同点**
  - **寄存器宽度：** SSE 的寄存器为 128 位，AVX 的寄存器为 256 位（AVX-512 为 512 位）。
  - **数据处理能力：** AVX 可以一次处理更多的数据，如更多的浮点数。
  - **指令集扩展：** AVX 引入了更多的指令和功能，提高了计算性能和效率。
  - **应用范围：** 由于寄存器更宽和指令集更丰富，AVX 更适用于高性能计算和数据密集型应用。

### 4.1.2 OpenMP&Pthread 介绍

OpenMP 和 Pthread 是两种常见的并行编程模型。OpenMP 是一种用于共享内存多处理器编程的 API，提供了简单易用的编译指令、库函数和环境变量，使得开发者可以轻松地将串行代码并行化。OpenMP 支持并行区域、工作共享构造和同步构造等特性。Pthread（POSIX 线程）是一种底层的线程库，允许开发者创建、管理和同步线程，适用于需要精细控制并行行为的场景。Pthread 提供了丰富的线程同步机制，如互斥锁、条件变量和读写锁。

在倒排索引求交问题中，OpenMP 可以通过简单的编译指令实现循环并行化，加快处理速度。Pthread 则可以创建多个线程并行处理不同的索引区间，提高计算效率。

下面我简要介绍一下 OpenMP 和 Pthread 库的异同：

- **相同点**
  - 都用于多线程并行编程，提升程序性能。
  - 都支持线程创建、控制和同步操作。
- **不同点**
  - **编程模型：** Pthread 采用线程级并行编程模型，需要手动管理线程；OpenMP 采用基于编译器指令的并行编程模型，更高层次。
  - **易用性：** Pthread 需要详细管理线程，复杂度高；OpenMP 通过简单的 pragma 指令并行化代码，更易用。
  - **可移植性：**Pthread 主要适用于 Unix/Linux 系统；OpenMP 适用于多种平台，包括 Unix/Linux 和 Windows。
  - **灵活性：** Pthread 提供更细粒度的控制，适合复杂并行任务；OpenMP 适合快速并行化、性能优化。
  - **开发效率：** Pthread 需要更多的代码和管理，开发效率较低；OpenMP 通过高层次抽象提高开发效率。

### 4.1.3 MPI 介绍

MPI（Message Passing Interface）是一种广泛应用于分布式内存并行计算的标准。它提供了一套函数库，用于在不同计算节点之间传递消息，从而实现并行计算，如图4.2。MPI 适用于大规模分布式系统和高性能计算（HPC）环境，支持点对点通信和集体通信操作。通过 MPI，开发者可以构建复杂的并行程序，实现高效的数据交换和同步。

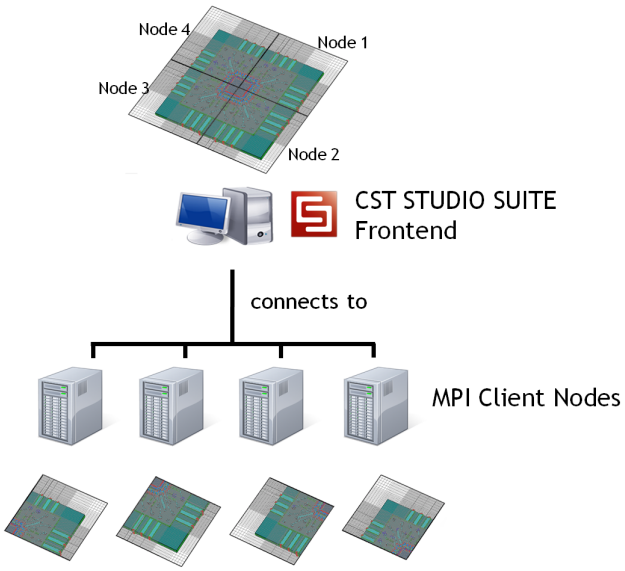


图 4.2: MPI 模拟流程

在倒排索引求交问题中，MPI 可以将索引数据分布到多个计算节点，并行处理不同的数据分区，通过消息传递汇总结果。这样可以充分利用分布式计算资源，加速求交过程。

下面我简要介绍一下 MPI 最为核心的通信机制：

- **点对点通信**
  - **MPI_Send 和 MPI_Recv：**基本的发送和接收函数，用于在两个进程之间传递消息。(这是阻塞通信)
  - **MPI_Isend 和 MPI_Irecv：**这是第一条两个函数的非阻塞通信版本。
  - **同步通信：**发送操作等待接收操作完成，确保数据传递的同步。
  - **异步通信：**发送和接收操作可以重叠执行，提高通信效率。
  - **缓冲通信：**通过中间缓冲区存储数据，发送操作不必等待接收操作完成。
- **集合通信**
  - **广播：**使用 MPI_Bcast 函数，将数据从一个进程发送到所有其他进程。
  - **散播：**使用 MPI_Scatter 函数，将数据从一个进程分发到所有其他进程。
  - **聚集：**使用 MPI_Gather 函数，将数据从所有进程收集到一个进程。
  - **规约：**使用 MPI_Reduce 函数，对所有进程的数据进行规约操作（如求和、求最小值等），并将结果发送到一个进程。
  - **全局规约：**使用 MPI_Allreduce 函数，对所有进程的数据进行规约操作，并将结果广播到所有进程。
- **同步和互斥**
  - **MPI_Barrier：**用于进程同步，使所有进程在此点上等待，直到所有进程都到达。
  - **MPI_Win_lock 和 MPI_Win_unlock：**用于一侧通信的锁定和解锁，确保数据一致性。

当然，MPI 还有更高级的功能如虚拟拓扑和并行 I/O 等等机制，这里限于篇幅便不再介绍，感兴趣的读者可以自行查阅。

#### 4.1.4 GPU 介绍

在 GPU 并行计算方面，oneAPI 和 CUDA 是两种主要的编程模型。oneAPI 提供了统一的编程模型，支持 CPU、GPU 和 FPGA 等多种架构，主要使用 Data Parallel C++ (DPC++) 进行开发。DPC++ 允许开发者编写跨平台的并行代码，并利用性能调优工具如 Intel VTune Profiler 进行优化。CUDA 是 NVIDIA 提供的并行计算平台和编程模型，通过使用 CUDA 编程模型，开发者可以在 NVIDIA GPU 上编写并行程序。CUDA 支持丰富的内核函数编写、内存管理和性能优化技术，如全局内存、共享内存、线程同步和原子操作。通过这些技术，CUDA 程序可以显著提升计算性能，适用于科学计算、图像处理和机器学习等领域。

在倒排索引求交问题中，GPU 可以通过 CUDA 或 OpenCL 等编程模型，将求交操作映射到大量的并行计算核心上，显著提升查询速度。例如，可以将倒排索引的求交操作分配给多个线程块，每个线程块负责处理一部分数据，快速完成并行计算。

下面我简要介绍一下 CUDA 和 SYCL 的异同：

- **相同点**
  - 都是用于并行计算的编程模型，旨在提高计算性能。

- 都支持大规模并行处理，适用于高性能计算任务。
- 都提供了丰富的 API 和库函数，支持科学计算、图像处理等领域。

- **不同点**
  - **平台依赖性：** CUDA 专用于 NVIDIA GPU，而 SYCL 基于 OpenCL，支持多种硬件平台，包括 CPU、GPU、FPGA 等。
  - **编程语言：** CUDA 主要扩展了 C/C++，而 SYCL 则与现代 C++ 标准完全兼容，支持更多 C++ 特性。
  - **开发复杂度：** CUDA 提供了详细的底层控制能力，适合需要高度优化的应用程序；SYCL 通过高层次抽象简化异构编程，适合更广泛的应用场景。
  - **生态系统：** CUDA 有广泛的生态系统支持，包括专门优化的库和工具链；SYCL 依托 OpenCL 生态系统，具有跨平台优势。
  - **单源编程：** SYCL 支持单源 C++ 编程，可以在同一个源文件中编写主机和设备代码，而 CUDA 通常将主机和设备代码分开。

## 4.2 实验相关介绍

### 4.2.1 研究流程图

如图4.3所示：

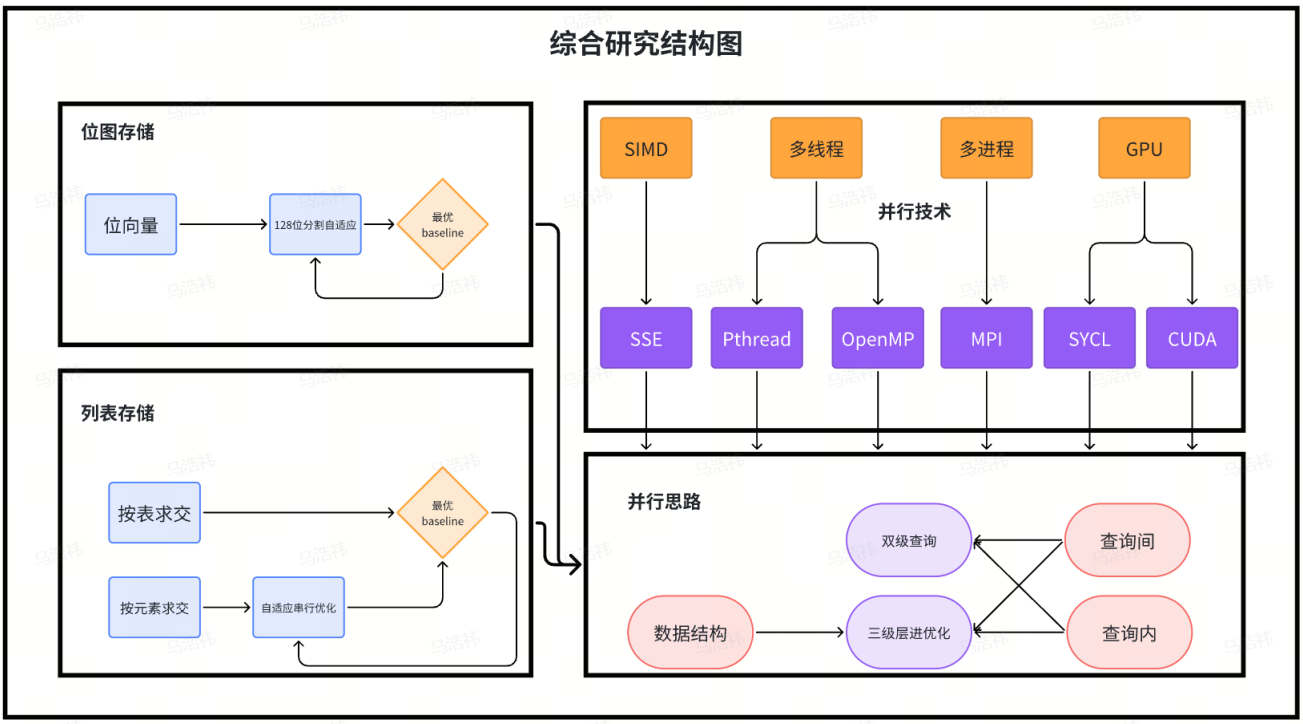


图 4.3: 研究流程图

### 4.2.2 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：

- 串行算法，Pthread&OpenMP 编程 IDE 为 CodeBlock，编译器为 GNU GCC Compiler。
- MPI 编程 IDE 为 Visual Studio，编译器内置，具体环境为 VS 下的 MSMPI。
- SYCL 编程平台为 DevCloud。GPU 为 Intel(R) Data Center GPU Max 1100。
- CUDA 为本机 NVIDIA 的 4060 显卡。
- 实验用到的汇编分析工具为 Godbolt 网站，动态分析工具为 Vtune。

### 4.2.3 实验数据集

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。

### 4.2.4 实验性能测试

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。时间测试方法如下：

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

### 4.2.5 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 Data 文件夹下 Result 文件里。

### 4.2.6 实验总仓库

本实验仓库位于 Github 下，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。仓库链接为：

https://github.com/Mhy166/parallel-programming.git

# 5 算法程序设计

对于所有的算法描述，我统一用代码进行解释，为了提高报告的信息密度，代码只选取关键部分，而且相应伪代码不再介绍。有兴趣的读者可以查阅仓库内的全部源码。

## 5.1 朴素串行算法

### 5.1.1 按表求交

本算法可以说是最容易想到，最简单的串行算法，其设计思想也较通俗易懂，如下：

- 初始情况，将第一个数组的所有元素插入结果集合 res 中。
- 随后，对于每一个后续数组，在结果集合 res 中也存在的元素保留，没出现在数组里的元素将被删除。
- 注意这里的设计有一个易错点，我用 unfound 去保存未出现的元素，在遍历后统一删除！而不是在原有数组直接删除，因为原有数组正在遍历，如果删除会出现意想不到的错误！

该算法时间复杂度大致为 $O(ln^2)$, 其中 $l$ 为列表数，$n$ 为列表规模。下面我们看一下本算法的关键代码以及必要的解释：

```
for (auto idx : queryIndices) {//遍历每一个列表
        if (flag) {//第一个列表的处理较特殊
            for (auto elem : arrays[idx]) {
                res.insert(elem);
                Flag.push_back(true);
            }
            flag = false;
        } else {
            vector<uint32_t> unfound;//保存未出现元素
            for (auto elem : res) {
                bool flag0=false;
                for (size_t i = 0; i < arrays[idx].size(); i++) {
                    if (elem==arrays[idx][i]) {
                        flag0 = true;//出现后及时停止
                        break;
                    }
                }
                if(!flag0){
                    unfound.push_back(elem);
                }
            }
            for (size_t i = 0; i < unfound.size(); i++) {//遍历后统一操作
                res.erase(unfound[i]);
```

```
            }
        }
    }
```

### 5.1.2 按元素求交

按元素求交算法会整体的处理所有的升序列表，充分利用列表的升序特点进行删减，每次得到全部倒排表中的一个交集元素。这类算法通常在 DAAT 查询下使用，以 Adaptive 算法为主，可以较好的应用提前停止技术。下面是该算法的设计思想：

- 降低存储容量。在寻找文档时，由于索引列表也是升序的，故可以将小于其的 DocID 删除，直到等于或者大于。
- 及时停止。当第一条链表走到尽头，则本次求交结束，这充分考虑了链表大小的数据特点。
- 整体处理。算法会处理每一个非第一链表，删减容量，直到元素出现后便停止处理该链表并计数，只有计数满了才意味着该元素是交集元素。

该算法时间复杂度大致为 $O(ln^2)$, 其中 $l$ 为列表数，$n$ 为列表规模。下面我们看一下本算法的关键代码以及必要的解释：

```
while (!lists[0].empty()) {//判定第一条链表，及时停止
    uint32_t tmp = lists[0].front();
    uint32_t cnt = 1;
    lists[0].pop_front();
    for (uint32_t i = 1; i < lists.size(); i++) {
        while (true) {
            if (lists[i].empty()) {//已经空了则退出
                break;
            }
            if (lists[i].front() < tmp) {//小于则删减
                lists[i].pop_front();
            }
            else if (lists[i].front() == tmp) {//等于则计数
                lists[i].pop_front();
                cnt++;
            }
            else {//大于也退出
                break;
            }
        }
    }
    if (cnt == lists.size()) {//计数满了则交集元素进入
        res.insert(tmp);
```

```
    }
}
```

### 5.1.3 自适应串行优化

我经过学习，发现按元素求交算法还有改进之处，可以利用其第一个列表判空结束的特点，更好地结合早停技术进行优化，本算法就是根据基础算法设计出的更优秀的串行算法，设计思想如下：

- 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第一个列表。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减少很多比较次数，有效提升实际性能，在结果部分我们会做分析。

下面我们看一下本算法的关键代码以及必要的解释：

```
//列表大小重排序函数
bool cmp(list<uint32_t>& a, list<uint32_t>& b) {
    return a.size() < b.size();
}
//处理循环
while (!lists[0].empty()) {
     uint32_t tmp = lists[0].front();
     uint32_t cnt = 1;
     bool flag = false;//用于早停检查的标志
     lists[0].pop_front();
     for (uint32_t i = 1; i < lists.size(); i++) {
         while (true) {
             if (lists[i].empty()) {//有列表空，直接停止处理
                 flag = true;
                 break;
             }
             //小于等于大于的处理部分同 5.1.2 节，不再赘述
         }
         if (flag) {//标志检查
             break;
         }
     }
     if (cnt == lists.size()) {
         res.insert(tmp);
```

```
        }
        sort(lists.begin(), lists.end(), cmp);//重排序，选择最短的列表作为下次迭代的第一个列表
    }
```

## 5.2 查询间优化算法

从本节开始，我将要介绍一系列的并行优化算法！这些算法均是在 5.1.3 节的自适应串行最优算法基础上设计的。

### 5.2.1 OpenMP 设计思路

针对查询间并行优化的思路，我们可以针对 OpenMP 的循环并行特性很方便地设计出高效适合并行化的算法，算法设计思想如下：

- 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。

为提高报告密度，只介绍算法的关键代码并给出必要解释：

```
        #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
            }
        }
```

### 5.2.2 Pthread 设计思路

类似地，我在 Pthread 框架下也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最优算法一致，算法设计难点落在如何分配查询给子线程上以及编写 Pthread 风格的代码框架。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;
    }
```

### 5.2.3 MPI 设计思路

接下来我们暂时对多线程告一段落,开始多节点多进程的并行设计!本小节我设计了查询间的 MPI 并行化策略。该算法的通信过程比较简单，但是 MPI 的具体编程细节也是很值得推敲的！本多进程优化算法的设计思想如下：

- 数据读取与广播：主进程负责读取输入数据文件 ExpIndex 和查询文件 ExpQuery，并将查询任务分发给各个子进程进行处理。
- 并行处理查询：每个查询任务由主进程分配给不同的子进程。每个子进程接收到查询任务后处理其分配的查询，并将结果返回给主进程。
- 结果收集与合并：主进程从所有子进程收集处理结果，并将结果合并和保存到输出文件中。
- 时间复杂度：仍为 $O(ln^2)$, 其中 $l$ 为列表数，$n$ 为列表规模。

我们同样看一下关键的 MPI 代码和必要的注释：

```
// 主进程：发送查询任务
int dest = queryID % (size - 1) + 1;
int querySize = queryIndices.size();
MPI_Send(&queryID, 1, MPI_INT, dest, 0, MPI_COMM_WORLD); // 发送查询 ID
MPI_Send(&querySize, 1, MPI_INT, dest, 0, MPI_COMM_WORLD); // 发送查询索引大小
MPI_Send(queryIndices.data(), querySize, MPI_UINT32_T, dest, 0, MPI_COMM_WORLD);
//主进程：发送结束信号
int endSignal = -1;
for (int i = 1; i < size; i++) {
    MPI_Send(&endSignal, 1, MPI_INT, i, 0, MPI_COMM_WORLD); // 发送结束信号
}
//主进程：接收处理结果
for (int i = 0; i < queryCount; i++) {
    int id;
    double elapsedSeconds;
    int resultSize;
    MPI_Recv(&id, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收查询 ID
    MPI_Recv(&elapsedSeconds, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // 接收查询时间
    MPI_Recv(&resultSize, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // 接收结果集大小
    results[id] = { id, elapsedSeconds };
    resultSizes[id] = resultSize;
}
// 子进程：接收查询总数 + 接收处理查询任务
MPI_Recv(&queryCount, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

```

```
28  while (true) {
29      int queryID;
30      MPI_Recv(&queryID, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收查询 ID
31      if (queryID == -1) break; // 检查结束信号
32      int querySize;
33      MPI_Recv(&querySize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收查询索引大小
34      vector<uint32_t> queryIndices(querySize);
35      MPI_Recv(queryIndices.data(), querySize, MPI_UINT32_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
36      // 接收查询索引数组
37      // 处理查询……
38      int resultSize = res.size();
39      MPI_Send(&queryID, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); // 发送查询 ID
40      MPI_Send(&elapsedSeconds, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD); // 发送查询时间
41      MPI_Send(&resultSize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD); // 发送结果集大小
42  }
```

### 5.2.4 SYCL 设计思路

本小节是本系列优化关于 GPU 的设计思路，该程序设计是基于 Intel Devcloud 下的 SYCL 框架的。在后面一节我们会看到同样是 GPU，CUDA 程序将会如何设计！下面介绍 SYCL 框架的设计思想：

- 数据并行处理：程序利用 SYCL 框架的并行处理能力，将查询的处理任务分配到多个计算单元上并行执行。这种设计大大提高了程序的执行效率，特别是面对大量数据和复杂计算时。
- 任务分解与独立执行：程序将查询处理任务分解成独立的子任务，每个查询的处理都是独立的。这种设计思想使得每个子任务可以在不同的计算单元上并行执行，充分利用计算资源，避免了任务之间的相互依赖，提高了计算效率。
- 缓冲区和访问器的使用：程序使用 SYCL 的 buffer 和 accessor 机制管理数据。这种设计保证了数据在主机和设备之间的高效传递和访问，并且能够在不同计算单元之间共享数据。此外，buffer 的生命周期管理和 accessor 的访问控制机制也提高了程序的健壮性和安全性。

下面我们看一下本问题的关键 SYCL 部分代码：

```
1       //在处理逻辑部分，首先将 queries 向量中的字符串转换为 C 风格的字符串数组。
2       //然后创建 SYCL 的 buffer，分别用于存储 arrays、queries 和 results。
3       results.resize(queries.size());
4       queue myQueue(default_selector_v);
5       vector<const char*> query_cstrs(queries.size());
6       for (size_t i = 0; i < queries.size(); ++i) {
7           query_cstrs[i] = queries[i].c_str();
8       }
9       buffer arrays_buffer(arrays.data(), range<1>(arrays.size()));
```

```
    buffer queries_buffer(query_cstrs.data(), range<1>(query_cstrs.size()));
    buffer results_buffer(results.data(), range<1>(results.size()));
    //通过 myQueue.submit 提交并行任务。任务处理函数中创建了三个 accessor,
    //分别用于访问 arrays_buffer、queries_buffer 和 results_buffer。
    //然后使用 parallel_for 函数并行处理每个查询。
    myQueue.submit([&](handler &h) {
        accessor arrays_accessor(arrays_buffer, h, read_only);
        accessor queries_accessor(queries_buffer, h, read_only);
        accessor results_accessor(results_buffer, h, write_only, no_init);
        h.parallel_for(range<1>(queries.size()), [=](id<1> i) {
            set<uint32_t> res;
            process_query(queries_accessor[i], res);
            results_accessor[i] = res;
        });
    }).wait();
```

### 5.2.5 CUDA 设计思路

可能是出于对 CUDA 的好奇，毕竟深度学习训练没少用 CUDA，我继续在本地安装了 CUDA 环境，进行了查询间的编程尝试。本小节的程序设计是基于 NVIDIA 下的 CUDA 框架的。下面介绍本框架下代码的设计思想：

- 设备内存分配与管理：程序使用 CUDA 的内存管理机制，在 GPU 上分配和管理设备内存。通过 cudaMalloc 和 cudaMemcpy 等函数，程序可以高效地在主机和设备之间传递数据，并确保设备内存的正确分配和释放。此外，合理的内存分配和管理也有助于提高程序的执行效率和稳定性。
- 核函数设计：程序通过编写 CUDA 核函数来定义并行计算的具体操作。核函数在 GPU 上执行，每个线程处理一个或多个数据元素。
- 同步处理：程序使用 CUDA 同步机制，如 cudaDeviceSynchronize，确保所有并行计算完成后再进行下一步操作。
- 数据并行处理：通过使用 CUDA 核函数，程序可以在大量 GPU 核心上同时执行相同的操作，大大提高了计算效率，特别是在处理大量数据和复杂计算时。

下面我们看一下关键的 CUDA 代码：

```
    //在 GPU 上分配内存并将主机上的数据传输到设备。
    uint32_t** d_lists;uint32_t* d_sizes;uint32_t* d_results;
    cudaMalloc((void**)&d_lists, lists.size() * sizeof(uint32_t*));
    cudaMalloc((void**)&d_sizes, lists.size() * sizeof(uint32_t));
    cudaMalloc((void**)&d_results, max_len * sizeof(uint32_t));
    for (size_t i = 0; i < lists.size(); i++) {
        uint32_t* d_list;
        cudaMalloc((void**)&d_list, lists[i].size() * sizeof(uint32_t));
```

```
        cudaMemcpy(d_list, h_lists[i], lists[i].size() * sizeof(uint32_t), cudaMemcpyHostToDevice);
        cudaMemcpy(d_lists + i, &d_list, sizeof(uint32_t*), cudaMemcpyHostToDevice);
    }
    cudaMemcpy(d_sizes, h_sizes, lists.size() * sizeof(uint32_t), cudaMemcpyHostToDevice);
    //CUDA 核函数 processQueries 在设备端运行，执行并行查询处理。
    __global__ void processQueries(uint32_t** d_lists, uint32_t* d_sizes, uint32_t* d_results, \
    uint32_t num_lists, uint32_t max_len) {...}//查询处理不再赘述
    //在主机端设置核函数调用的执行配置，指定线程块和线程数。并收回结果!
    int blockSize = 256;
    int numBlocks = (max_len + blockSize - 1) / blockSize;
    processQueries<<<numBlocks, blockSize>>>(d_lists, d_sizes, d_results, lists.size(), max_len);
    cudaDeviceSynchronize();
    cudaMemcpy(h_results, d_results, max_len * sizeof(uint32_t), cudaMemcpyDeviceToHost);
```

## 5.3 查询内优化算法

本节我将要介绍与先前查询间并行不同的查询内优化算法！理论上说，此节算法会显著影响查询的响应时间和总体耗时。

### 5.3.1 OpenMP 设计思路

由于 OpenMP 的适合 for 循环的编程风格，这里我对查询内形式的优化算法进行了许久思考，最终设计出一种 OpenMP 并行化算法，虽然说后续实验性能的时候效果不是很好，但是其程序设计过程的思考和同步、回写机制的熟练使用或许更为重要！我也想介绍一下我的尝试，下面是该算法的设计思想：

- 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;
            }
        }
    }
```

### 5.3.2 Pthread 设计思路

本小节，我设计出可以在查询内进行 Pthread 多线程并行化的算法，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，具体设计思想如下：

- 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);
    }
```

### 5.3.3 MPI 设计思路

再一次离开多线程，我们来介绍多进程 MPI 并行化的技术。其实和 Pthread 查询内差不多，算法的核心是将第一个列表进行分割，并将其他列表保持不变，分配给子进程进行并行处理。其设计思想如下：

- 数据读取和分发：主进程（rank 0）负责读取输入数据文件 ExpIndex 和查询文件 ExpQuery，并将数据分发给其他子进程进行处理。
- 并行查询处理：每个查询由主进程分解为多个部分，分配给不同的子进程处理。每个子进程处理其分配的查询部分，并将结果返回给主进程。
- 结果收集与合并：主进程从所有子进程收集部分结果，并将其合并为最终结果。
- 时间复杂度：仍为 $O(ln^2)$, 其中 $l$ 为列表数，$n$ 为列表规模。

限于篇幅原因，这里只给出关键的 MPI 代码和必要的通信流程的注释如下：

```
// 主进程：查询分发
int segmentSize = lists[0].size() / (size - 1); // 计算每个子进程分配到的查询片段大小
int startIdx = 0;
for (int i = 1; i < size; i++) {
    int endIdx = (i == size - 1) ? lists[0].size() : startIdx + segmentSize;
    // 计算每个子进程的结束索引
    MPI_Send(&startIdx, 1, MPI_INT, i, 0, MPI_COMM_WORLD); // 发送开始索引到子进程
    MPI_Send(&endIdx, 1, MPI_INT, i, 0, MPI_COMM_WORLD); // 发送结束索引到子进程
    int querySize = queryIndices.size();
    MPI_Send(&querySize, 1, MPI_INT, i, 0, MPI_COMM_WORLD); // 发送查询索引数组大小到子进程
    MPI_Send(queryIndices.data(), querySize, MPI_UINT32_T, i, 0, MPI_COMM_WORLD);
    // 发送查询索引数组数据到子进程
    for (uint32_t j = 0; j < lists.size(); j++) {
        int listSize = lists[j].size();
        MPI_Send(&listSize, 1, MPI_INT, i, 0, MPI_COMM_WORLD); // 发送当前列表大小到子进程
        vector<uint32_t> listData(lists[j].begin(), lists[j].end());
        MPI_Send(listData.data(), listSize, MPI_UINT32_T, i, 0, MPI_COMM_WORLD);
        // 发送当前列表数据到子进程
    }
    startIdx = endIdx; // 更新开始索引以供下一个子进程使用
}
// 主进程：结果收集合并
set<uint32_t> resultSet;
for (int i = 1; i < size; i++) {
    int resultSize;
    MPI_Recv(&resultSize, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // 从子进程接收结果集大小
    vector<uint32_t> localRes(resultSize);
    MPI_Recv(localRes.data(), resultSize, MPI_UINT32_T, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // 从子进程接收结果集数据
    resultSet.insert(localRes.begin(), localRes.end());
    // 将子进程的结果集数据合并到主进程的结果集中
}
// 子进程：接收分配的查询数据
while (true) {
    int startIdx, endIdx, querySize;
    MPI_Recv(&startIdx, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收开始索引
    MPI_Recv(&endIdx, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE); // 接收结束索引
    MPI_Recv(&querySize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // 接收查询索引数组大小
```

```
41      vector<uint32_t> queryIndices(querySize);
42      MPI_Recv(queryIndices.data(), querySize, MPI_UINT32_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
43      // 接收查询索引数组数据
44      vector<list<uint32_t>> lists(querySize);
45      for (uint32_t i = 0; i < querySize; i++) {
46          int listSize;
47          MPI_Recv(&listSize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
48          // 接收当前列表大小
49          vector<uint32_t> listData(listSize);
50          MPI_Recv(listData.data(), listSize, MPI_UINT32_T, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
51          // 接收当前列表数据
52          lists[i] = list<uint32_t>(listData.begin(), listData.end());
53          // 将接收到的数据转换为列表
54      }
55      set<uint32_t> localRes;
56      processQueryPart(startIdx, endIdx, queryIndices, lists, localRes);
57      // 处理查询数据并得到局部结果集
58      int resultSize = localRes.size();
59      vector<uint32_t> localResVec(localRes.begin(), localRes.end());
60      MPI_Send(&resultSize, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
61      // 发送局部结果集大小到主进程
62      MPI_Send(localResVec.data(), resultSize, MPI_UINT32_T, 0, 0,MPI_COMM_WORLD);
63      // 发送局部结果集数据到主进程
64  }
```

## 5.4 双级查询优化算法

本小节将融合 5.2 和 5.3 的查询间和查询内的设计思路，进一步研究二者结合的算法，本节思路也将拓展到 5.6 节的多并行技术融合研究。其最大区别是：本节的双级均以多线程为工具进行，而 5.6 节的双级并行甚至三级并行都是采用不同并行技术进行的！

### 5.4.1 OpenMP 设计思路

以 OpenMP 为基准，我设计了双级并行算法，其核心思想如下：

- **查询间并行**
  - 程序使用 `#pragma omp parallel for` 指令实现查询间并行。
  - 通过设置线程数 `num_threads`，将多个查询任务分配给不同的线程并行处理，每个线程负责处理一个查询。
- **查询内并行**
  - 在每个查询的处理过程中，使用 `#pragma omp parallel for` 实现查询内并行。
  - 对于每个查询，首先将查询索引按照列表大小排序，以提高效率。

- 使用多个线程并行处理查询内中的每个列表，确保高效的并行化。

- **结果集的更新与同步**
  - 使用 `#pragma omp critical` 确保对共享变量（如结果集）的安全访问，避免数据竞争。
  - 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 `#pragma omp 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。

有了前面两大节的铺垫，这里的代码也比较简单，简述如下：

```
#pragma omp parallel for num_threads(numThreads) schedule(dynamic)//查询间
    for (int i = 0; i < queryLines.size(); i++) {
        //前置工作不再赘述
        while (!lists[0].empty()) {//开始处理单条查询
            //前置工作不再赘述
            //查询内进行并行，同时采用 shared 共享机制
            #pragma omp parallel for num_threads(lists.size() - 1) shared(flag, cnt)
            for (uint32_t j = 1; j < lists.size(); j++) {
                while (true) {
                    //也设立临界区和实现刷新缓存机制，和查询内类似，这里不再赘述。
                }
            }
            //后置求结果大小，排序部分不再赘述
        }
    }
```

### 5.4.2 Pthread 设计思路

以 Pthread 为基准，我同样结合上述思路设计了双级并行算法，其核心思想如下：

- **查询间并行**
  - 使用 `pthread_create` 和 `pthread_join` 实现查询间并行。
  - 通过创建多个外层线程，每个线程负责处理一部分查询任务，实现查询任务的并行处理。

- **查询内并行**
  - 对于每个查询，通过创建多个内层线程来处理查询内的并行任务。
  - 内层线程负责处理查询中的子任务，每个子任务对应于查询中的一个子列表。
  - 将每个查询的索引按照数组大小进行排序，以提高处理效率。

- **结果集的更新与同步**
  - 每个内层线程独立处理自己的子任务，并将结果存储在局部结果集中。
  - 使用 `pthread_mutex` 实现对全局结果集的安全访问，避免数据竞争。
  - 内层线程完成处理后，将局部结果集合并到全局结果集中。

与 OpenMP 一样，在前面两大节的铺垫上，这里的 Pthread 代码简述如下：

```
//查询内处理函数
void* processSublist(void* arg) {
    // 处理子列表，不再赘述
    // 使用互斥锁保护结果集更新
    pthread_mutex_lock(&res_mutex);
    results.push_back(local_res);
    pthread_mutex_unlock(&res_mutex);
    return nullptr;
}
//查询间处理函数，函数内再分工查询内处理
void* processQueries(void* arg) {
    //前置工作不再赘述
    //拆分子列表，准备查询内并行
    //创建子任务并启动线程
    vector<SubTask> tasks(inner_thread_num);
    for (int i = 0; i < inner_thread_num; i++) {
        tasks[i].lists = lists;
        tasks[i].lists[0] = sublists[i]; // 只使用子列表作为第一个列表
    }
    vector<pthread_t> threads(inner_thread_num);
    //启动查询内并行
    for (int i = 0; i < inner_thread_num; i++) {
        pthread_create(&threads[i], nullptr, processSublist, (void*)&tasks[i]);
    }
    //聚合，不再赘述
}
//主函数内分工查询间处理
int main() {
    // 读取数据和查询
    // 初始化互斥锁
    pthread_mutex_init(&res_mutex, nullptr);
    int outer_thread_num = 4; // 外部线程数
    pthread_t threads[outer_thread_num];
    int thread_ids[outer_thread_num];
    //启动查询间并行
    for (int i = 0; i < outer_thread_num; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], nullptr, processQueries, (void*)&thread_ids[i]);
    }
    //聚合，输出结果和清理资源，不再赘述
```

```
41          return 0;
42      }
```

## 5.5 存储优化算法

从本节开始，我将给出另一种利用位图存储的方式进行查询处理的高效算法策略！并自行设计出更兼容后续层进式结合的算法。

### 5.5.1 朴素串行设计思路

这里我们直接介绍位向量化的存储优化算法：

算法较简单，但是需要提前将存储格式转换成 bitset，由于搜索引擎是可以离线进行这部分工作的，故在介绍和计时中我们均不考虑，直接关心查询处理的性能。

而对于设计思想来说，bitset 数据结构提供了简单的按位与运算符，故直接对每个 bit 列表进行运算即可，不再分点陈述。这里我们直接介绍查询处理的代码：

```
1   for (auto idx : queryIndices) {
2       if (flag) {
3           res = bitmaps[idx];
4           flag = false;
5       }
6       else {
7           res &= bitmaps[idx];
8       }
9   }
```

### 5.5.2 128 位分割自适应设计思路

然而，由于我之前分析过数据，得出最大 DocID 为 25205174，也就是说位向量 bitset 的大小会高达 25205200，然而索引长度最多 30000，意味着这些位当中最多只有 30000 位是 1，可见空间利用效率非常低，因此本节我经过思考，给出了一种优化算法——128 位分割自适应算法，下面我们看它的设计思路：

- 利用自适应思想改变排序方式。选择具有最小的最大 DocId 的列表作为 L1 位向量，其后的列表的 DocID 不管多大，都可以不必考虑，只需保存 L1 位向量的大小的位数即可。如在查询 1 里，可以将位向量从 2500 万优化到 62 万位左右。
- 存储格式按照 128 位位图为单位去分割存放，这样可方便未来的 SIMD 优化。但是需要额外注意总体位向量的位数也应该扩展到 128 位的倍数。

接下来简要看一下 128 位排序优化的存储和计算部分的代码：

```
1       sort(queryIndices.begin(), queryIndices.end(), compareBySize);//按照最大 DocID 排升序
2       vector<vector<bitset<128>>> rebit(queryIndices.size());
```

```
    uint32_t id = queryIndices[0];
    for (uint32_t i = 0; i < arrays[id][arrays[id].size() - 1]; i += 128) {
        bitset<128> tmp;
        for (uint32_t j = 0; j < 128; j++) {
            tmp[j] = bitmaps[queryIndices[0]][j + i];
        }
        rebit[0].push_back(tmp);
    }
    for (uint32_t i = 1; i < queryIndices.size(); i++) {
         for (uint32_t k = 0; k < arrays[id][arrays[id].size() - 1]; k += 128) {
            //存储同上，保持大小是排序后第一个位向量的最大 DocID
         }
    }
    //计算，结果直接就存在第一个位向量的 128 位子向量里面
    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];
        }
    }
```

### 5.5.3 SIMD 设计思路

接下来我将 128 位分割自适应算法当中的 128 位的位图改成 SSE 支持的 128 位向量即可。由于思想和先前差不多，这里主要介绍所用到的 SSE 指令及其对 SIMD 加速的影响：

- $\_mm\_and\_si128$：这个函数执行两个 128 位整数寄存器之间的按位与操作。这种按位与操作可以同时处理多个数据元素。这种并行计算的能力是 SIMD 加速的核心，它允许一次性处理多个数据，从而提高了计算效率。
- $\_mm\_store\_si128$：这个函数将一个 128 位整数寄存器的内容存储到内存中的指定位置。在 SIMD 加速中，经常需要将计算结果存储到内存中以便后续使用。这个函数提供了一种高效的方式来处理大量数据。此外，存储操作也可以减少数据在寄存器和内存之间的频繁移动，从而提高了计算效率。
- $\_mm\_load\_si128$：这个函数从内存中加载一个 128 位整数到一个寄存器中。在 SIMD 加速中，加载操作通常用于将数据准备好，以便后续的并行计算。这个函数允许在单个指令中获取多个数据元素，从而提高了加载数据的效率。加载操作还可以减少数据在内存和寄存器之间的传输时间，从而减少了内存访问造成的延迟。

下面给出 SIMD 优化的关键部分，即存取数据，计算处理的代码：

```
    vector<vector<__m128i>> rebit(queryIndices.size());
    //存入数据（第一个大向量的，其他类似）
    for (uint32_t i = 0; i < arrays[id][arrays[id].size() - 1]; i += 128) {
```

```
        bitset<128> tmp;
        for (uint32_t j = 0; j < 128; j++) {
            tmp[j] = bitmaps[queryIndices[0]][j + i];
        }
        const uint32_t* bitmapPtr = reinterpret_cast<const uint32_t*>(&tmp);
        __m128i bitmap = _mm_load_si128((__m128i*)bitmapPtr);
        rebit[0].push_back(bitmap);
    }
    //计算按位与（这是循环体，循环部分省略）。
    rebit[0][j] = _mm_and_si128(rebit[0][j], rebit[i][j]);
    //取回数据
    for (uint32_t i = 0; i < rebit[0].size(); i++) {
        bitset<128>tmp;
        _mm_store_si128((__m128i*) & tmp, rebit[0][i]);
        result.push_back(tmp);
    }
```

### 5.5.4 OpenMP 设计思路

进一步采用多线程，我针对 OpenMP 循环并行为特性的风格，设计出下面可以更高效适合并行化的算法，同时还可以对预存储部分并行化。这其实是查询内的方向，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];
        }
    }
```

### 5.5.5 MPI 设计思路

同样地，我们继续多进程的并行思路尝试新策略。这里采用查询间的优化思路，主进程和子进程的工作和 3.3 节的基本一样，这里不再赘述。而且主进程分配任务的方式和接收结果的方式也近乎一样，这里也不再赘述，有兴趣的读者可以查看源代码了解细节。我们直接看子进程部分的差别较大的 MPI 代码（必要的注释已经给出）：

```
//接收并处理查询任务
//接收查询大小，查询索引，代码不再赘述
int resultSize;
double elapsedSeconds;
processQuery(queryID, queryIndices, resultSize, elapsedSeconds); // 处理查询
//子进程的查询处理函数（关键部分）
void processQuery(int queryID, const vector<uint32_t>& queryIndices,\
int& resultSize, double& elapsedSeconds) {
// ... 构建 rebit 数据结构，同 128 位存储串行算法 ...
// ... 进行查询处理，同 128 位存储串行算法 ...
//构造结果集大小（位图中 1 的个数）
    resultSize = 0;
    for (uint32_t i = 0; i < rebit[0].size(); i++) {
        resultSize += rebit[0][i].count();
    }
```

```
16  }
17  //发送查询 ID，发送查询时间，发送结果集大小，代码不再赘述
```

## 5.6 多并行技术结合算法

算法设计的最后一节，我将利用多种并行技术，同时参照 5.4 节的双级思路进行结合。在这之前，我先介绍一下为什么需要考虑多种并行技术的结合：

- **性能提升**：多并行技术结合可以显著提升算法的性能。
- **资源利用**：不同的并行技术擅长处理不同类型的任务。结合多种并行技术可以更充分地利用系统资源，提高计算效率。例如，SIMD 适合处理特定存储结构有关的任务，OpenMP 适合处理循环无依赖的任务，MPI 适合处理分布式大规模的任务。
- **扩展性**：多并行技术结合可以提高系统的扩展性和灵活性。通过组合不同的并行技术，可以适应不同规模的计算任务，灵活应对计算需求的变化。
- **负载均衡**：结合多种并行技术可以实现更好的负载均衡，避免单一技术的瓶颈。通过合理分配任务，可以更均匀地利用系统资源，避免资源浪费和性能下降。
- **算法优化**：结合多种并行技术可以更好地优化算法。不同的并行技术可以互相补充，通过协调优化，可以更高效地解决复杂问题。

### 5.6.1 OpenMP+MPI 双级设计

本小节将采用两种并行技术，结合两种本问题经典的并行思路得出如下核心思想：

- MPI 负责查询间的节点通信
- OpenMP 负责查询内的循环优化

并行算法是在 5.5.1 节位向量化的基础上进行的，因为其代码结构特点适合这两种并行技术。这里直接介绍代码，为提高信息密度，与 MPI 查询间类似的代码均不赘述：

```
1   //主进程发送，子进程接收查询任务：查询大小，查询索引等等
2   int resultSize;
3   double elapsedSeconds;
4   processQuery(queryID, queryIndices, resultSize, elapsedSeconds); /子进程处理查询
5
6   //子进程的查询处理函数（关键部分）
7   void processQuery(int queryID, const vector<uint32_t>& queryIndices,\
8   int& resultSize, double& elapsedSeconds) {
9       //进行查询处理，子进程处理查询仅需要一行即可实现优化！第二层 for 是无依赖的！
10      for (int i = 1; i < static_cast<int>(queryIndices.size()); i++) {
11  #pragma omp parallel for num_threads(4)
12          for (int j = 0; j < static_cast<int>(rebit[i].size()); j++) {
13              rebit[0][j] &= rebit[i][j];
```

```
            }
        }
    }
    //子进程发送查询 ID，发送查询时间，发送结果集大小，主进程接收
```

### 5.6.2 SIMD+OpenMP+MPI 三级设计

本小节将进一步加上 SIMD 的向量化，采用三种并行技术。算法核心思想如下：

- MPI 负责查询间的节点通信
- OpenMP 负责查询内的循环优化
- SIMD 负责查询内数据结构和运算的高效化

SIMD 的结合是在 5.6.2 节上进行的，因此这里直接介绍体现 SIMD 差异的代码：

```
//SSE 向量存储部分
vector<vector<__m128i>> rebit(queryIndices.size());
for (uint32_t i = 0; i < arrays[id][arrays[id].size() - 1]; i += 128) {
    bitset<128> tmp;
    const uint32_t* bitmapPtr = reinterpret_cast<const uint32_t*>(&tmp);
    __m128i bitmap = _mm_load_si128((__m128i*)bitmapPtr);//存入 SSE 向量
    rebit[0].push_back(bitmap);
}//其他列表处理过程一样，不再赘述
//查询处理部分：
for (int i = 1; i < static_cast<int>(queryIndices.size()); i++) {
#pragma omp parallel for num_threads(4)
    for (int j = 0; j < static_cast<int>(rebit[i].size()); j++) {
        rebit[0][j] = _mm_and_si128(rebit[0][j], rebit[i][j]);//SSE 向量的位与运算
    }
}
```

# 6 实验结果分析

本大节将介绍第 5 大节所介绍的编程实验结果，以小节为单位，结果展示分两部分：部分数据表，体现科学性和严谨性，所有实验结果均保留 7 位小数，以 s 为单位，同时为了提高信息密度，选取 6 个在规模上具有代表性的查询。数据图表，直观体现性能对比和规模趋势的影响，这里是用全部查询数据进行绘制的，这里展示两种图，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，这里前 4 节我按照列表平均大小 5000 为单位分组，后 2 节由于涉及位向量存储的转换，采用最小位向量的大小 500 万为单位分组，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。

## 6.1 朴素串行结果分析

本小节针对 5.1 节的朴素串行不同策略 + 优化实验组进行分析，列出各规模下具有代表性的单条查询响应时间和 1000 条查询总时间（吞吐量），下面我们看相应数据表1和图6.4,6.5.

| 查询 ID | 列表平均长度 | 按表求交 | 按元素求交 | 自适应串行优化 |
|---|---|---|---|---|
| 741 | 134 | 0.0000765 | 0.0000112 | 0.0000105 |
| 497 | 1133 | 0.0023466 | 0.0001222 | 0.0001084 |
| 734 | 5181 | 0.0475786 | 0.0007366 | 0.0007568 |
| 282 | 10091 | 0.6228810 | 0.0006807 | 0.0011879 |
| 1 | 20089 | 0.0214816 | 0.0017623 | 0.0007465 |
| 666 | 30000 | 5.9538800 | 0.0045552 | 0.0070833 |
| Total | : | 3375.3500000 | 4.5033300 | 4.0874000 |

表 1: 朴素串行结果数据表

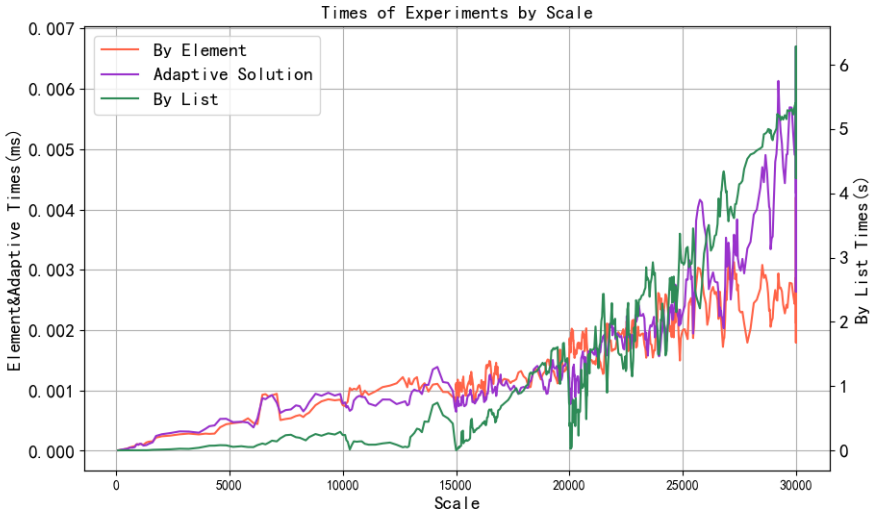


图 6.4: 朴素串行结果折线图

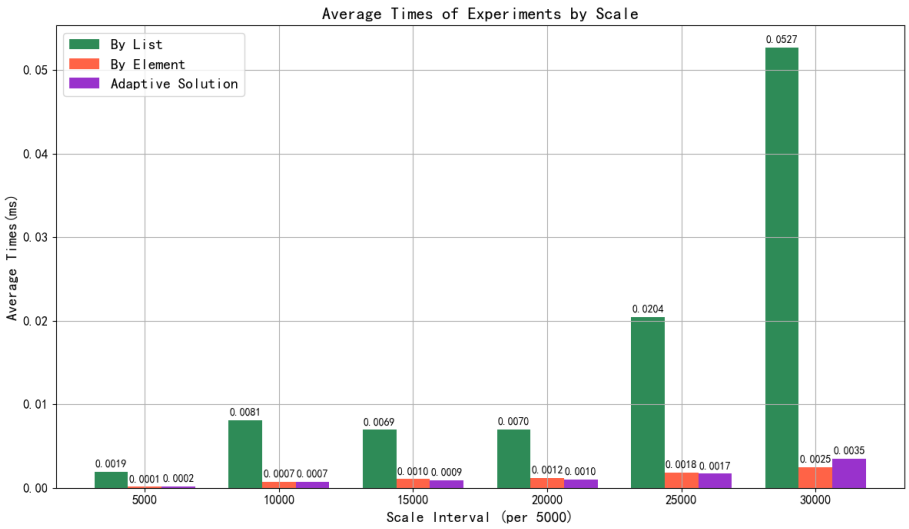


图 6.5: 朴素串行结果平均柱状图

我根据结果数据作出如下解释和分析（鉴于报告面向并行加速，这里的串行分析较为简略）:

- 不管从响应时间还是总吞吐量来看，按表求交都是最慢的，而且随着规模增加，该算法的增长量级也大大超过了其他两种策略。这是因为按表求交策略逐一对每个表两两求交，是以每个列表的元素为粒度的，而不考虑数据的特性，比如说，如果第一个表最大元素比第二个表最小元素还要小，那么本算法仍旧会做比较，而我们知道这种情况是可以直接返回 0 的，按元素求交合自适应策略则都考虑了每个列表数据的特性，得到了优秀的性能。
- 自适应串行优化策略作为早停技术的结合策略，可以看到其性能在部分查询上有着优越性，而且在总时间上看，也较优于按元素求交策略，这证实了早停技术的积极影响。这两者都是以公共元素为粒度进行比较的，并及时删除列表元素，其不仅在时间上有着良好的性能，空间上也有一定的优势!

## 6.2 查询间优化结果分析

本小节针对 5.2 节的不同并行技术的查询间实验组进行分析，列出各规模下具有代表性的单条查询响应时间和 1000 条查询总时间（吞吐量），下面我们看相应数据表2和图6.6,6.7.

| 查询 ID | 列表平均长度 | 自适应串行优化 | OpenMP | Pthread | MPI | SYCL | CUDA |
|---|---|---|---|---|---|---|---|
| 741 | 134 | 0.0000105 | 0.0000114 | 0.0000112 | 0.0000922 | 0.0000040 | 0.0001239 |
| 497 | 1133 | 0.0001084 | 0.0001151 | 0.0001228 | 0.0005693 | 0.0000549 | 0.0000208 |
| 734 | 5181 | 0.0007568 | 0.0007316 | 0.0007412 | 0.0044388 | 0.0002460 | 0.0056604 |
| 282 | 10091 | 0.0011879 | 0.0012819 | 0.0013158 | 0.0084204 | 0.0004380 | 0.0047788 |
| 1 | 20089 | 0.0007465 | 0.0009013 | 0.0009246 | 0.0033118 | 0.0003457 | 0.0386860 |
| 666 | 30000 | 0.0070833 | 0.0083327 | 0.0084417 | 0.0621340 | 0.0025623 | 0.0470095 |
| Total | : | 4.0874000 | 1.5865900 | 1.7103400 | 4.9437100 | 3.5218600 | 36.2845000 |

表 2: 查询间优化结果数据表

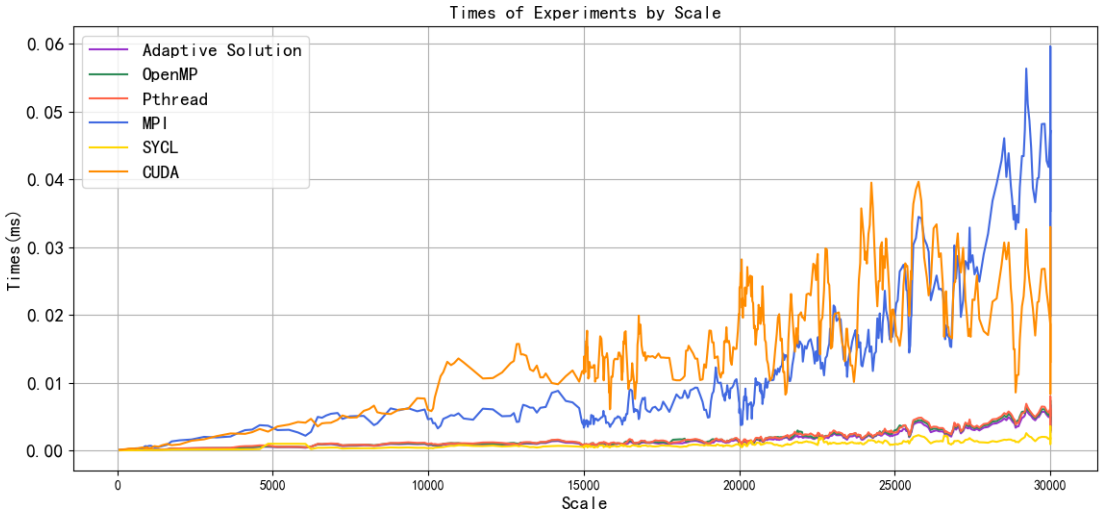


图 6.6: 查询间优化结果折线图

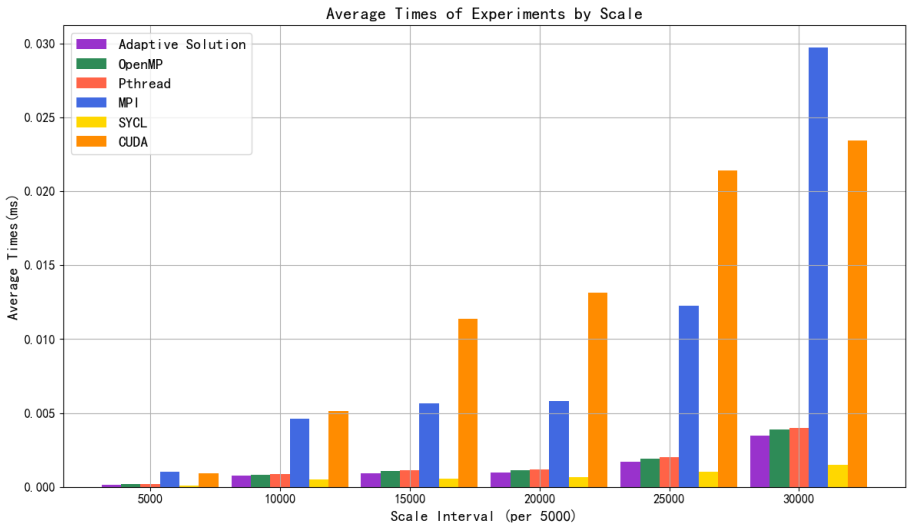


图 6.7: 查询间优化结果平均柱状图

我根据性能结果作出如下解释和分析，包含了本课程所学的最全并行技术的应用及特性分析：

- 从总时间我们可以看出，多线程 OpenMP 加速比最大，为 2.58，Pthread 也表现出良好的性能，SYCL 由于在 Dev cloud 的平台下，也表现出了一定的加速，这意味着其能高效利用异构资源，但是 MPI 反而恶化了性能，考虑是其通信环境的创建，阻塞通信的延迟等因素所致，因为小规模下它的时间恶化的更多（通信部分影响占比更大），故上述解释具备了一定的合理性，然而 CUDA 表现出更为恶化的性能，由于这是在本机 GPU 上运行的，我认为是环境下机器未能充分调度任务，亦或是本任务的这种存储求交策略不能很好的发挥 GPU 并行的优势，就比如 OpenMP 适合循环调度一样。
- 然后我们进行每种并行技术细致的分析：
  - OpenMP：作为多线程的一种高效自动库，它实现了很多对程序员透明的步骤，使得我们能

方便，又高效地实现并行化，本问题下可以看到其的高效性，由于是查询间并行，单条查询的响应时间影响不大，但可以看到有略微的增加，这是 OpenMP 多线程调度的一个开销体现，在先前的实验中我特地比较了单线程的性能差异，是可以证明这一点的，具体详见仓库。

- Pthread：作为多线程的一种大改代码结构的库，对于初学多线程的我来说有一定的难度，实现的效果也相对 OpenMP 较差。同样的，响应时间没有受多大影响，其性能较 OpenMP 略差估计是因为线程函数设计和同步机制的影响，毕竟这些是需要我实现的。
- MPI：作为多进程的高效通信库，不同于前面两个技术，其响应时间受影响较大！显然是通信环境的，以及数据广播发送和接收之间的开销所致。而且随着规模增大，其广播量也会正比增加，这是 MPI 在处理这样的任务的一个待改进之处。考虑到最近研究的 RingAllReduce 技术，这可以作为一种可尝试的改进手段！我们知道它将线性增长的通信量优化到了常数级别！
- SYCL：作为云平台上的 GPU 加速技术，它通过优化数据传输、同步和通信、硬件利用率，再加上 **Intel(R) Data Center GPU Max 1100** 的高速运算能力，提高了程序的性能，达到更好的加速效果。这很好的体现在响应时间上了，虽然我们没有进行查询内的并行化，但这正是 GPU 较 CPU 处理的高效之处。
- CUDA：作为英伟达的 GPU 加速技术，其性能不尽如人意，这可能是因为：在大规模计算中，主机和 GPU 之间的数据传输开销显著增加，影响整体性能。数据传输速度相比 GPU 的计算速度较慢，成为性能瓶颈。而且 CUDA 程序在执行过程中需要频繁的同步操作，这在大规模计算中会导致显著的时间开销。还有可能的影响因素在于 GPU 的架构和硬件限制（如最大线程数、最大并发块数）对性能的直接影响。（本机还是不如云平台提供的好）

## 6.3 查询内优化结果分析

本小节针对 5.3 节的不同并行技术的查询内实验组进行分析，列出各规模下具有代表性的单条查询响应时间和 1000 条查询总时间（吞吐量），下面我们看相应数据表3和图6.8,6.9.

| 查询 ID | 列表平均长度 | 自适应串行优化 | OpenMP | Pthread | MPI |
|---|---|---|---|---|---|
| 741 | 134 | 0.0000105 | 0.0002055 | 0.0004014 | 0.0005601 |
| 497 | 1133 | 0.0001084 | 0.0006513 | 0.0002138 | 0.0041000 |
| 734 | 5181 | 0.0007568 | 0.0146901 | 0.0009484 | 0.0188099 |
| 282 | 10091 | 0.0011879 | 0.0159186 | 0.0007470 | 0.0278463 |
| 1 | 20089 | 0.0007465 | 0.0019560 | 0.0014592 | 0.0341572 |
| 666 | 30000 | 0.0070833 | 0.6315290 | 0.0078008 | 0.2181430 |
| Total | : | 4.0874000 | 147.3740000 | 21.0001000 | 100.2830000 |

表 3: 查询内优化结果数据表

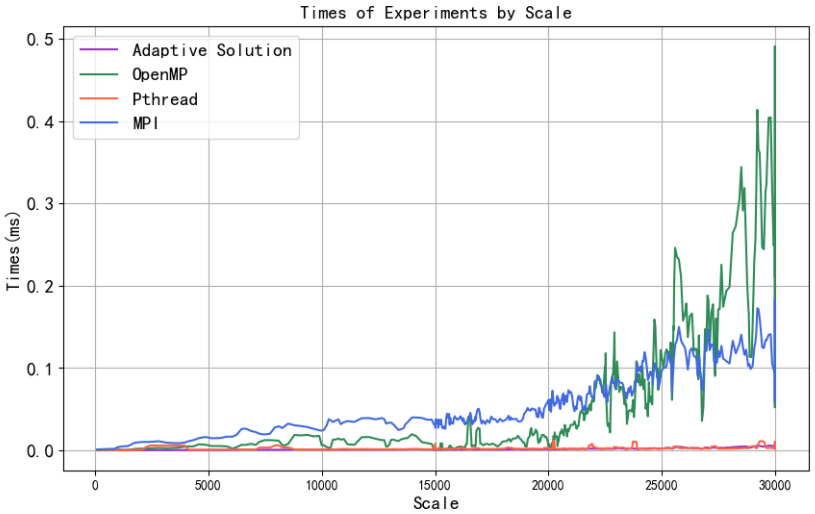


图 6.8: 查询内优化结果折线图

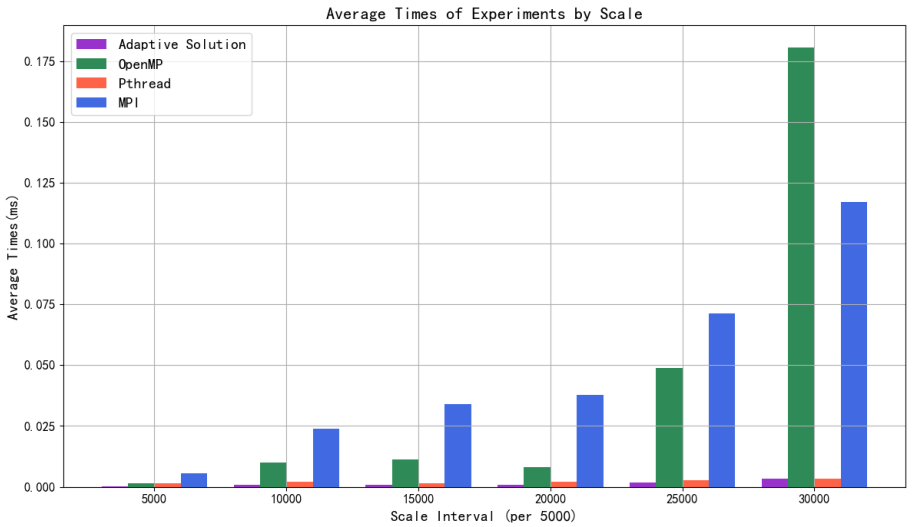


图 6.9: 查询内优化结果平均柱状图

我根据性能结果作出如下解释和分析:

- 从总时间我们可以看出，由于第一个列表的分割策略不是很适合并行调度优化，几种技术都无法很好的进行加速,本实验组很好的让我们知道了并行加速需要算法和工具相匹配,比如说 OpenMP 适合并行循环，而列表存储方式事实上是不适合并行的，强制并行并不会带来我们预期的加速效果，正如查询间 OpenMP 位于第一，而本实验下它的性能倒数第一了。
- 然后我们进行每种并行技术细致的分析:
  - OpenMP：我们知道查询内具体影响吞吐率和响应时间，但是这里受算法影响，每个线程需要保存其对应处理的列表，空间消耗增大，致使线程管理调度的开销剧增，而且硬件资源使用也到达了瓶颈，这是不合理应用循环并行技术的缺点。

- Pthread：为了同步，我们引入了互斥锁，这必然是时间开销较大的机制，因为涉及到等待阻塞的开销。但是其性能较 OpenMP 还有不少提升，显然，Pthread 并不是单纯为循环并行设计的库，因此其算法设计会相对较合适，但是也并不是完全合适，空间消耗较 OpenMP 会更大，因为它需要保存所有列表的副本。
- MPI：算法和 Pthread 差不多，仍然需要保存全部列表的副本，增大空间开销。但是值得注意的是，性能较 Pthread 恶化了不少，根据所学，这是由于其通信环境下各个进程在资源分配后涉及到了大量空间使用，导致调度不灵活以及进程运行速度受限，当然，主进程和子进程的通信开销仍然很大，这也是影响因素之一。

那么查询内优化策略就没法加速了吗？当然不是，在后面我们将介绍新存储思路下其大显身手的策略变化！

## 6.4 双级查询优化结果分析

本小节针对 5.4 节的不同并行技术的双级查询间 + 查询内实验组进行分析，列出各规模下具有代表性的单条查询响应时间和 1000 条查询总时间（吞吐量），下面我们看相应数据表4和图6.10,6.11.

| 查询 ID | 列表平均长度 | 自适应串行优化 | Pt 查询间 | Pt 查询内 | Pt 双级查询 | Op 双级查询 |
|---|---|---|---|---|---|---|
| 741 | 134 | 0.0000105 | 0.0000112 | 0.0004014 | 0.0002286 | 0.0001173 |
| 497 | 1133 | 0.0001084 | 0.0001228 | 0.0002138 | 0.0031084 | 0.0010329 |
| 734 | 5181 | 0.0007568 | 0.0007412 | 0.0009484 | 0.0033328 | 0.0060213 |
| 282 | 10091 | 0.0011879 | 0.0013158 | 0.0007470 | 0.0014336 | 0.0219750 |
| 1 | 20089 | 0.0007465 | 0.0009246 | 0.0014592 | 0.0021451 | 0.0009807 |
| 666 | 30000 | 0.0070833 | 0.0084417 | 0.0078008 | 0.0042592 | 0.0823411 |
| Total | : | 4.0874000 | 1.7103400 | 21.0001000 | 6.0413700 | 12.5358000 |

表 4: 双级查询优化结果数据表

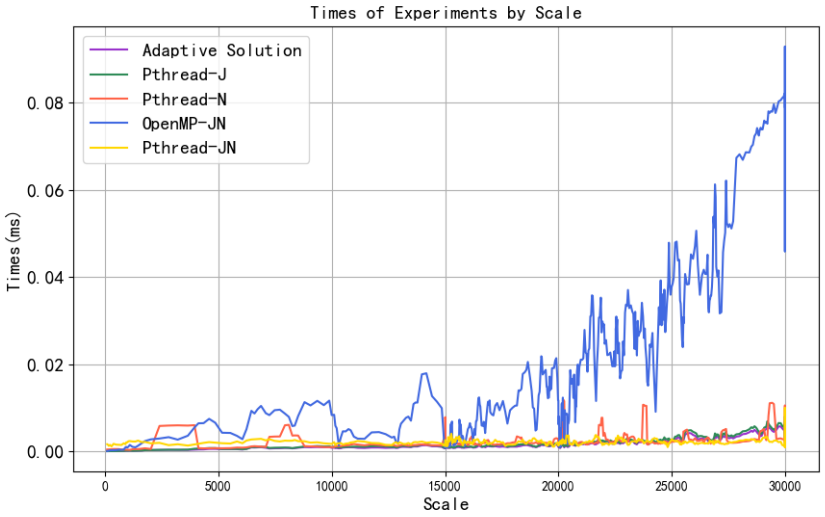


图 6.10: 双级查询优化结果折线图

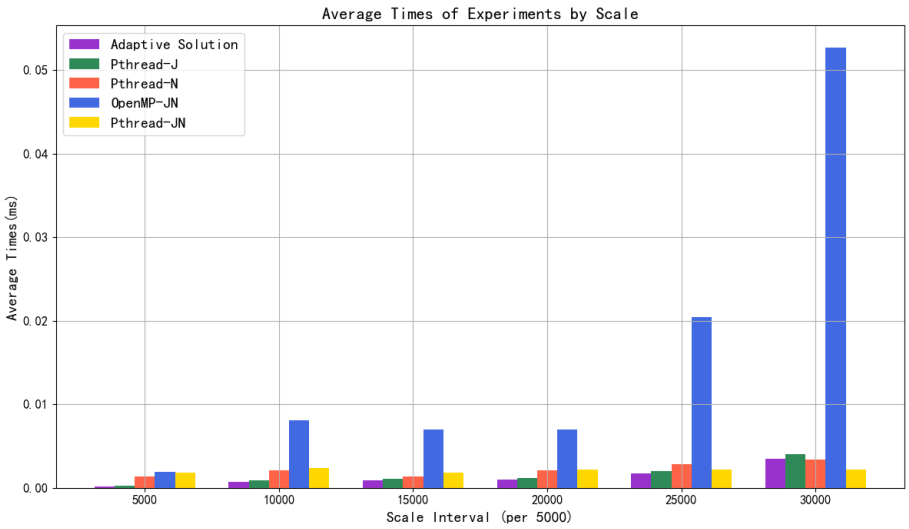


图 6.11: 双级查询优化结果平均柱状图

我根据性能结果作出如下解释和分析:

- 先看总时间，这里列出了前两节的 Pthread 的数据以供对比，我们发现确实，在查询间和查询内均进行了并行化后性能得到了折中的表现，更倾向于查询间，这意味着查询间的负载分配是较为均衡的，把查询内的性能的不利影响进行了平均化，使得综合结果还可以，当然也给出 OpenMP 的结果，也可以发现体现了这一点，查询内和查询间的综合对查询内的影响进行了一定的消除。
- 对于响应时间来说，受规模影响，OpenMP 的双级查询性能恶化严重，这体现了 6.3 节分析的列表策略下其并行化循环风格的不适用性，Pthread 受影响较小，甚至在柱状图最大规模平均下，我们发现其响应时间超越了最初的串行算法！这得益于其查询内的调度优势，而且是在多线程同步查询下的进一步多线程分配，这会对我们的调度起到一定的积极作用！我们能更充分利用资源和数据。
- 涉及到 Pthread 查询内，查询间的单个对比数据，前面两节已经做了分析，这里不再赘述。

关于列表存储模式下的并行优化策略就到此告一段落了，在这里我们展示了查询间，查询内，两者结合的实验结果，并对多种并行技术进行了分析，最终还是由于查询内的存储不适用性导致了性能的恶化，但是我们也看到了两种多线程，多进程 MPI，两种 GPU 的性能特点和其加速效果，接下来峰回路转，我们继续介绍新的实验结果，并分析更为高效的多并行技术融合的方法!

## 6.5 存储优化结果分析

本小节针对 5.5 节的不同并行技术的位向量存储实验组进行分析，列出各规模下具有代表性的单条查询响应时间和 1000 条查询总时间（吞吐量），本节将使用新的规模：最小位向量的大小。下面我们看相应数据表5和图6.12,6.13.

| 查询 ID | 位向量最小长度 | 自适应串行优化 | 位向量 | 128 分割法 | SSE | OpenMP | MPI |
|---|---|---|---|---|---|---|---|
| 322 | 66881 | 0.0047891 | 0.0019355 | 0.0000133 | 0.0000068 | 0.0000844 | 0.0000490 |
| 673 | 511741 | 0.0006794 | 0.0015274 | 0.0000502 | 0.0000294 | 0.0000923 | 0.0001954 |
| 468 | 5159616 | 0.0007583 | 0.0012320 | 0.0007011 | 0.0002522 | 0.0003584 | 0.0012840 |
| 97 | 10002537 | 0.0019727 | 0.0010757 | 0.0008914 | 0.0005184 | 0.0006134 | 0.0024922 |
| 48 | 20205038 | 0.0042865 | 0.0014751 | 0.0019286 | 0.0009868 | 0.0010640 | 0.0050748 |
| 270 | 25203594 | 0.0010502 | 0.0012716 | 0.0047895 | 0.0015525 | 0.0012239 | 0.0064000 |
| Total | : | 4.0874000 | 1.6141915 | 0.7859992 | 0.5540468 | 0.5922711 | 1.7182846 |

表 5: 存储优化结果数据表

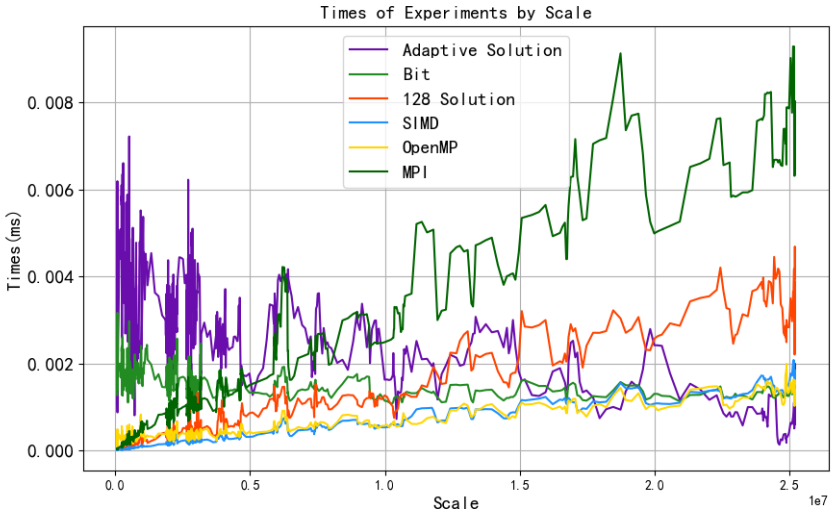


图 6.12: 存储优化结果折线图

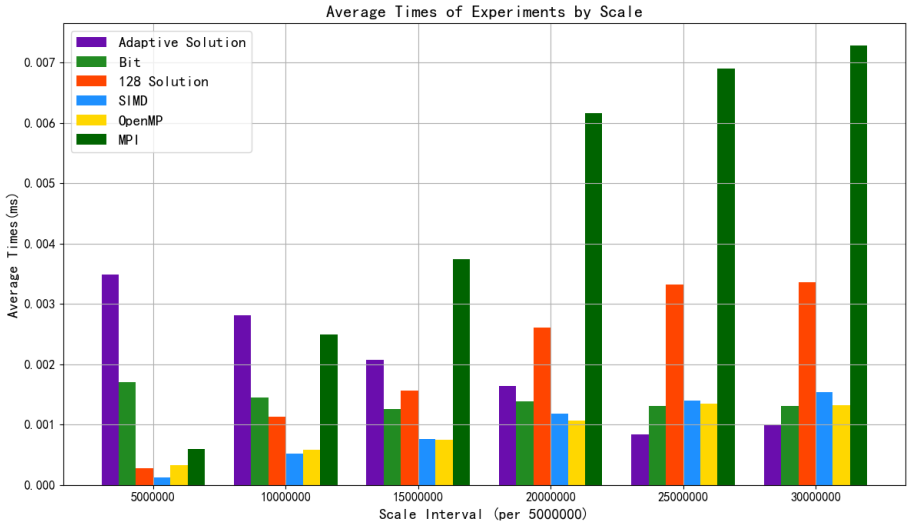


图 6.13: 存储优化结果平均柱状图

根据性能结果，我们可以作出如下分析：

- 从总时间我们可以看出，当进行了位向量化后，新的编程策略性能显著高于原有串行列表策略，

这体现了位向量在求交运算时直接可以进行与运算的高效性。而且在 128 位分割后我们也能看出性能更上一层楼，这是因为其再次利用自适应技术，以最小长度的位向量为基准进行与运算，这能很好的利用数据的特点。随后的 SIMD 优化则在此 128 分割基础上得到 1.4 的加速比，这体现了 SSE256 向量化的高效性。当再一次考虑多线程 OpenMP 和多进程 MPI 的时候，我们发现 OpenMP 仍然有加速效果，但是 MPI 显著差于原有 baseline 了。

- 接下来我们对每种并行技术进行细致的分析：
  - SSE：作为 SIMD 的一个典型方法，我们可以很方便的进行存储的转换然后几个数据同时进行运算。这主要影响了查询内的延迟，可以认为其有效地加速了程序，我们观察数据表图表可以证实这一点。
  - OpenMP：前面几节分析到 OpenMP 不适合列表的存储策略，现在的位向量存储策略则可以让其高效地发挥作用，我们可以看到在规模较小的时候，其线程管理调度开销占比过大导致优化不明显，但是规模增大后其并行效果便越发优秀。
  - MPI：其响应时间受通信环境影响依然很大。我们知道 MPI 会创建通信环境并维护，在主进程和子进程间进行数据的广播发送和接收，这些都会造成巨大的开销。而且本来与运算的时间就很少，开销的占比影响进一步增大，而且随着规模增大，传输数据量增大，其开销会正比增加。当然也由于我们的任务量很小，MPI 未能充分发挥作用，MPI 在大规模任务中还是十分重要的，尤其是最近调研的深度学习结合分布式训练领域。
- 我们应当注意到，原有的自适应串行优化的图中走势其实没有参考价值，因为我们的规模选取和其走势无关，两种串行策略所利用的数据特点是不同的，因此在6.12中它的走势没有参考价值，在6.13中拿它进行对比也没有意义，这里仅是进行一个展示。
- 事实上，位向量的性能和 ID 关系不大，和列表数量关系大，因为其大小均为最大值 2500 万左右，其随规模走势不明显，但是利用最短基准的 128 位分割则再次体现规模效应，其能利用数据特点进行进一步的优化。

## 6.6 层进式结合结果分析

本小节针对 5.6 节的最终多并行技术融合实验组进行分析，列出各规模下具有代表性的单条查询响应时间和 1000 条查询总时间（吞吐量），本节使用规模：最小位向量的大小。下面我们看相应数据表6和图6.14,6.15.

| 查询 ID | 位向量最小长度 | 128 分割法 | MPI | O+M | S+O+M |
|---|---|---|---|---|---|
| 322 | 66881 | 0.0000133 | 0.0000490 | 0.0000158 | 0.0000071 |
| 673 | 511741 | 0.0000502 | 0.0001954 | 0.0000922 | 0.0000420 |
| 468 | 5159616 | 0.0007011 | 0.0012840 | 0.0009077 | 0.0004276 |
| 97 | 10002537 | 0.0008914 | 0.0024922 | 0.0016961 | 0.0007968 |
| 48 | 20205038 | 0.0019286 | 0.0050748 | 0.0031696 | 0.0016154 |
| 270 | 25203594 | 0.0047895 | 0.0064000 | 0.0044322 | 0.0022306 |
| Total | : | 0.7859992 | 1.7182846 | 0.9518989 | 0.4717589 |

表 6: 层进式融合结果数据表

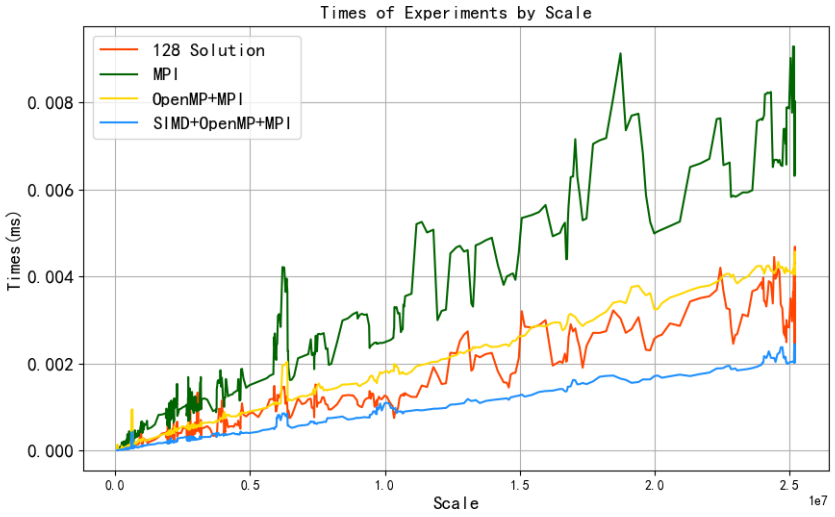


图 6.14: 层进式融合结果折线图

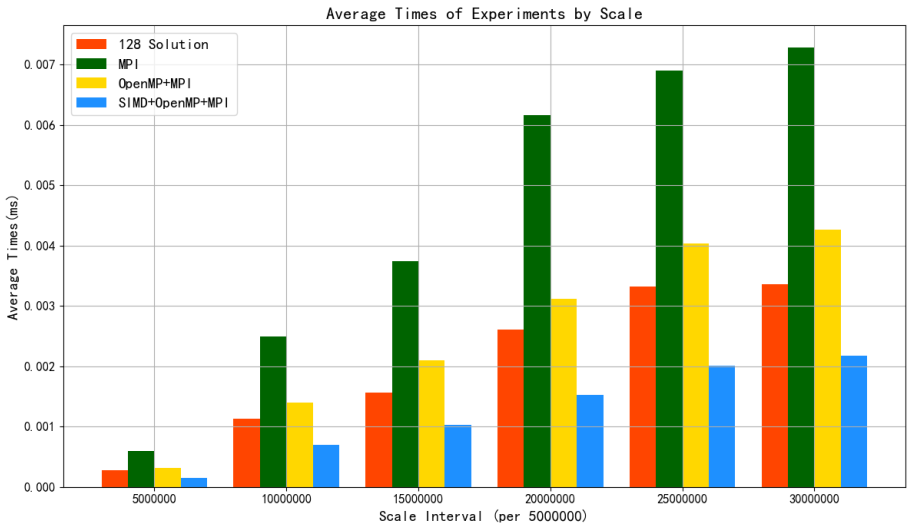


图 6.15: 层进式融合结果平均柱状图

这里根据性能结果作出如下分析:

- 先看总时间，我们发现 MPI 在单独作用时会因为通信环境开销影响实验性能，但是在结合了 OpenMP 后其效果有所好转，这是查询间和查询内分别采用不同并行技术优化的结果，随后进一步采用 SSE 优化存储结构后，其性能再次提升了一倍左右，这是在双级查询基础上又一级存储形式优化的结果，可以看到多种并行技术能在一个程序中的不同结构发挥各自的优势，共同促进整体性能的优化!
- 对于响应时间来说，四者在规模影响下均提升平稳，而且任凭规模大小，三级结合的并行程序性能始终是最好的，在上一节我们看到它们单打独斗时的性能除了 SSE 都差强人意，本节将它们结合后，在各个进程下的各个线程内，我们可以更高效的利用 SSE 资源进行同步计算，这对于性能的提高至关重要，而单独 SSE 时，不难理解，其利用效率不高，我们无法获得更好的性能。

到此，所有的实验结果就分析完毕了，我进行了：

- 列表存储下：串行优化，查询间，查询内，单并行技术下查询间 + 查询内的结合策略。
- 位图存储下：串行优化，多进程，多线程，SIMD 优化以及多并行技术融合的策略。

事实上，从结果分析还不够，接下来我将进行汇编层级的分析以及运行过程中的信息分析，以更深入认识多种并行技术。

# 7 Profiling 分析

下面我们将要从多方面进一步分析并行技术的加速性能成因，主要从汇编级别和运行级别上进行探索。

## 7.1 GodBolt 汇编分析

Godblot 是一个线上生成汇编代码的网站，可以直观地看出高级语言代码对应的汇编代码，以获得深入的并行程序理解。下面我们将分别展示 3 种并行技术的汇编示例。

```
596         call    std::vector<long long __vector(2),
597         movdqa  xmm0, XMMWORD PTR [rax]
598         movaps  XMMWORD PTR [rbp-144], xmm0
599         movdqa  xmm2, XMMWORD PTR [rbp-2480]
600         movaps  XMMWORD PTR [rbp-160], xmm2
601         movdqa  xmm1, XMMWORD PTR [rbp-144]
602         movdqa  xmm0, XMMWORD PTR [rbp-160]
603         pand    xmm0, xmm1
604         movaps  XMMWORD PTR [rbp-2480], xmm0
```

图 7.16: SSE 汇编

如图7.16，是 SIMD 下的 SSE 部分汇编代码，我们可以看到图中 call 在调用一个存储函数 long long ___vector, 指针的类型也与我们熟知的 DWORD 不同，是 XMMWORD，这体现了汇编级别存储的差异。也体现出 SIMD 的向量寄存器的使用。在指令层级，SSE 优化的指令是专属的，用 movdqa 来存储，movaps 来对齐，这很好的体现了 SIMD 中专属指令的使用。在与运算的处理部分，可以看到 SSE 优化用专属指令 pand 来对 xmm0 和 xmm1 两个向量器进行运算，直接进行 SIMD 的专属指令的快速运算。这应该是 SSE 性能好的一方面原因。


图 7.17: Pthread 汇编


图 7.18: OpenMP 汇编

如图7.17和7.18, 我们进行对比的分析:

- 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，这是合乎我们认知的。
- 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。

## 7.2 Vtune 运行分析

Vtune 是 Intel 下的程序分析工具，可以实时分析程序运行时的信息和数据，能帮助我们很好的观察特定机器下运行程序的性能，见表7:

| 指标 | 自适应串行 | Pthread | OpenMP | MPI |
|---|---|---|---|---|
| CPI | 0.324 | 0.320 | 0.300 | 0.280 |
| Instructions | 124200000 | 98000000 | 92000000 | 87000000 |
| L1 Hit | 6000003 | 29000000 | 28000000 | 27000000 |
| CPU Time | 0.01 | 0.009 | 0.0092 | 0.0095 |
| TopDown.Slots | 8000003 | 90000000 | 85000000 | 80000000 |

表 7: Vtune 动态数据表

下面我们针对数据进行简要分析:

- CPI: 平均每条指令的周期数，一般情况下越低代表执行效率越高。由于 Pthread 和 OpenMP 能够在一定程度上优化并行计算的性能，因此其 CPI 相对较低。MPI 进一步利用多进程优化了并行计算的性能，所以 CPI 更低。
- Instructions: 执行指令数，一般情况下越低代表效率越高。Pthread 和 OpenMP 的指令数较少，因为它们可以更好地利用多线程。MPI 进一步减少了指令数，因为它们能够更高效地并行化计算。
- L1 Hit：L1 缓存的命中次数，可以有效表示程序访问 cache 的次数。Pthread 和 OpenMP 的命中量相对较高，这一方面是其利用资源更加充分，另一方面也是由于多线程并行计算能发挥缓存的作用。MPI 的命中较低，因为它在多进程缓存冲突的管理可能不是很高效。
- CPU Time：CPU 执行时间，越低越好。Pthread ,OpenMP, MPI 均得到较低的 CPU 时间，这意味着这些并行技术可以充分 CPU 的计算能力。
- TopDown.Slots：指的是处理器资源的插槽，能一定程度上表示程序利用 CPU 的效率，越高，代表利用越充分。Pthread，OpenMP 和 MPI 的指标均高于串行，而且高了一个数量级，这意味着这些并行技术能更好地利用 CPU 资源。

# 8 总结与反思

## 8.1 实验总结

针对期末的研究大实验，我进行如下总结:

- 进行了串行算法的多种策略尝试，并尝试优化，取得了优秀的性能。
- 针对查询间并行思路进行多种并行技术的尝试，并对比分析其性能异同。
- 针对查询内并行思路进行编程实验，着重分析其性能恶化原因。
- 探索实现查询间和查询内结合的单并行策略，得到了折中的性能并进行了分析。
- 在新的位图存储形式下进行新的串行策略的尝试，并对其进行优化，在此基础上对其进行多种并行技术的实现。
- 融合多种并行技术，相辅相成，查询内 + 查询间 + 数据结构，三级层进得到目前加速比最好的程序性能。
- 给出多方面的性能分析，包含规模走势，平均性能，汇编代码以及运行的动态数据等信息。

## 8.2 课程收获

到此为止，我的期末大报告就临近尾声了，这不仅意味着课程的学习告一段落，而且意味着一段精彩纷呈的实验旅程即将结束。这一学期的实验中，我经历过配置环境，调试 Debug，思路受限的困难，但也体验到运行成功并得到了合理结果的惊喜感和成就感，当然要感谢王老师和助教团队们，在先前的每一次报告中，你们的评语让我更加有动力有自信走完这段路途，你们的帮助让我相对顺利学完了并行课程并获得了很多知识。还记得第一节课我忧心忡忡地问老师：这门课程用什么软件呢？那时的我还不清楚并行是什么东西，关于并行的技术有什么，以及并行带来的问题又有哪些。然而现在，我想我能给出自己的答案了！

总的来说，课程带给我如下收获！

- 认识并行体系结构，掌握各种常见并行技术的特点和编程实现，提高编程能力。
- 熟练使用 Latex 写报告，掌握排版的技巧，文献的引用规范，增强学术规范意识。
- 熟练编写计时脚本，并掌握利用 Python 进行多种图表的绘制。
- 提高实验结果的呈现以及分析能力，熟练使用各种工具进行多方面的性能挖掘并给出自己的分析。
- 培养自力更生，艰苦奋斗的学习态度，树立认真负责的学习精神。

**最后，再次感谢老师的授课和助教团队的帮助，愿老师和助教们健康快乐，事业顺利！**

# 参考文献

[1] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, page 743–752, USA, 2000. Society for Industrial and Applied Mathematics.

[2] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. Using graphics processors for high-performance ir query processing. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, page 1213–1214, New York, NY, USA, 2008. Association for Computing Machinery.

[3] F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered sets. *SIAM Journal on Computing*, 1(1):31–39, 1972.

[4] Benjamin Schlegel, Thomas Willhalm, and Wolfgang Lehner. Fast sorted-set intersection using simd instructions. In *ADMS@VLDB*, 2011.

[5] Shirish Tatikonda, Flavio Junqueira, B. Barla Cambazoglu, and Vassilis Plachouras. On efficient posting list intersection with multicore processors. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '09, page 738–739, New York, NY, USA, 2009. Association for Computing Machinery.

[6] Dimitris Tsirogiannis, Sudipto Guha, and Nick Koudas. Improving the performance of list intersection. *Proc. VLDB Endow.*, 2(1):838–849, aug 2009.

[7] Di Wu, Fan Zhang, Naiyong Ao, Fang Wang, Xiaoguang Liu, and Gang Wang. A batched gpu algorithm for set intersection. In *2009 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, pages 752–756, 2009.

[8] Fan Zhang, Di Wu, Naiyong Ao, Gang Wang, Xiaoguang Liu, and Jing Liu. Fast lists intersection with bloom filter using graphics processing units. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, page 825–826, New York, NY, USA, 2011. Association for Computing Machinery.