南開大學
Nankai University

计算机学院

并行程序设计期末报告

倒排索引并行化探索

姓名：唐明昊 朱世豪

学号：2113927 2113713

专业：计算机科学与技术

2023 年 7 月 2 日

目录

1 选题介绍

1.1 项目背景

在传统的信息检索系统中，文档通常是按照文档编号或者时间等顺序排列，用户查询时需要逐一扫描文档库。这种方法随着数据量的增大，查询时间会逐渐变得无法接受。倒排索引就是为了解决这一问题而提出的。

倒排索引将每个单词映射到包含这个单词的所有文档的集合中，从而提高查找的效率。通过倒排索引，我们可以很快地找到包含任何查询单词的文档，这使得搜索引擎可以快速地响应用户的查询。然而网页文档等互联网资源的规模急剧膨胀，为了快速、准确地应答每秒数以千万计的用户查询请求，需要高效的基于倒排索引的请求处理算法。

1.2 问题描述

倒排索引（inverted index），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。

对于一个有 U 个网页或文档（Document）的数据集，若想将其整理成一个可索引的数据集，则可以为数据集中的每篇文档选取一个文档编号（DocID），使其范围在 [1,U] 中。其中的每一篇文档，都可以看做是一组词（Term）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。

索引压缩 随着互联网信息量的急剧膨胀，搜索引擎索引的网页文档数量已由最初的几千万增长到如今的上百亿，倒排索引集大小也由最初的 GB 级别增长到如今的 TB 级别。为了节省存储空间，降低 I/O 开销，提高查询处理效率，主流搜索引擎通常将倒排索引以压缩的形式存储在内存中。

倒排列表中的 docID 可能很大，需要非常多的位来编码。为了获得更好的压缩效果，搜索引擎通常先对 l_t 进行 d-gap 变换，即求出其中相邻 docID 的差值 d-gap，生成相应的 d-gap 列表。

例如，倒排列表 $l_t = \{9, 28, 33, 44, 123\}$ 的 d-gap 列表为

$$delta - l_t = \{9, 19, 5, 11, 79\}$$

通过**前缀和**操作，$delta - l_t$ 也可以转换成 l_t。也就是说，l_t 和 $delta - l_t$ 可以相互等价转换。因此，通过对 $delta - l_t$ 无损压缩，可以实现 l_t 的无损压缩。

倒排列表求交 也称表求交或者集合求交，当用户提交了一个 k 个词的查询，查询词分别是 t_1, t_2, ..., t_k，表求交算法返回 $\cap_{1 \le i \le k} l(t_i)$。

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：

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

例如查询 “2014 NBA Final”，搜索引擎首先在索引中找到 “2014”，“NBA”，“Final” 对应的倒排列表，并按照列表长度进行排序：

$$l(2014) = (13,\ 16,\ 17,\ 40,\ 50)$$

$$l\left(NBA\right)=(4,\ 8,\ 11,13,14,\ 16,17,39,40,42,50)$$

$$l\left(Final\right)=(1,\ 2,\ 3,\ 5,\ 9,\ 10,\ 13,\ 16,\ 18,\ 20,\ 40,\ 50)$$

求交操作返回三个倒排列表的公共元素，即：

$$l\left(2014\right)\cap l\left(NBA\right)\cap l\left(Final\right)=(13,\ 16,\ 40,\ 50)$$

1.3 前人工作

海量数据、海量查询、实时响应的应用需求对搜索引擎的存储和查询性能提出了很大挑战。

通过倒排索引压缩技术可以极大的降低倒排索引数据的磁盘存储空间开销，并使更多的数据可以被加载到内存中：通过结合高效的解压操作就能达到对海量磁盘压缩索引数据的快速查询访问。因此倒排索引压缩技术就成为提升海量倒排索引数据存储和查询性能的必要手段。以下是前人在压缩上做的一些工作：

1. S. W. Golomb 于 1966 年提出 Golomb 编码 [5]，通过基于游程长度的编码方式实现数据的高效压缩。
2. Zobel 和 Moffat 在 2006 年的文献中提出 Variable Byte Encoding[14]，通过可变长度的字节编码实现对文档 ID 列表的高效压缩
3. Anh 和 Moffat 在 2005 年提出通过结合前缀编码和布尔编码技术 [1]，利用对齐的二进制码对倒排列表进行高效压缩。
4. 2008 年 Zhang 和 Long 等人提出 PForDelta 压缩算法 [13]，通过使用 PForDelta 编码方法实现倒排索引的简单高效压缩。
5. Lemire、Kaser 和 Aouiche 在 2010 年的文献中提出 SIMD-based 压缩算法 [10]，通过利用 SIMD 指令集对字对齐的位图索引进行排序以提高压缩效率。

在实际应用中，需要对多个文档集合进行交集或并集操作，以获取相关文档的信息。这就需要使用倒排索引求交或求并的算法。随着数据规模的增大和计算机硬件的发展，倒排索引求交算法也在不断发展和优化，以提高计算效率和搜索精度。以下介绍一些集合求交的串行与并行算法：

1. Hwang 和 Lin 的算法 [6, 7]：利用两个指针分别遍历两个有序数组，比较指针所指的元素，如果相等则输出，否则移动较小元素的指针。该算法可以推广到 k 个有序数组的情况。
2. Demaine 的自适应算法：利用一个优先队列维护 k 个有序数组的当前元素，每次取出最小元素并将其所在数组的下一个元素入队。如果最小元素在所有数组中都出现，则输出。该算法可以根据数组长度和重复度自适应地调整遍历顺序 [2, 3]。
3. Tsirogiannis 的动态探测算法：利用 CPU 的多级 Cache，动态地选择最佳的遍历顺序，以减少 Cache 未命中和内存访问延迟。该算法还可以实现多处理器上的负载均衡 [11]。
4. Inoue 等人的 SSE 优化算法：利用 SSE 指令集，一次比较多个元素，并使用位操作来记录匹配结果，减少分支预测失败数。该算法可以提高数据吞吐量和计算效率 [8]。
5. Ding 等人的分段归并算法：利用 GPU 平台，将每个有序数组分为多个段，并在 GPU 上执行段与段之间的归并计算。该算法可以充分利用 GPU 的并行能力 [4]。

6. Zhang 等人的 Bloom Filter 算法：利用 GPU 平台，使用 Bloom Filter 来过滤掉不可能匹配的元素，并在 GPU 上执行剩余元素之间的求交计算 [12]。

2 工作说明

2.1 工作概览

本项目总结前人工作、查阅资料、学习他人代码思路，对倒排索引的压缩以及求交的并行化工作进行了细致的探索。

在平时报告中，我们重点针对**集合求交**问题运用学习到的并行化手段进行优化。

- SIMD 是第一个并行手段，该实验我们实现了调研到的 4 个串行算法，并根据算法特性实现了三种不同的 SIMD 并行优化方式。另外还探索了诸如对齐与不对齐的影响，不同平台性能对比等问题。
- 多线程并行使用了 PThread 与 OpenMP 实现，探索了 query 间与 query 内两种并行方式。并针对多线程并行的问题提出了任务池、加速大概率事件等优化手段。在这次实验还比较了两个多线程接口，使用 VTune 对两者性能表现做了详细分析。
- 多进程实验时，思路基本同多线程一致，但我们重点研究了 MPI 的通信机制，学习并运用了非阻塞通信、集合通信等。另外，此次实验我们还将多个并行手段混合起来，进行全过程并行化加速。
- GPU 实验是在期末周后仓促完成的，从粗粒度和细粒度两个角度尝试使用了 CUDA 进行并行加速，领略了 GPU 的强大计算能力。

本文力求**简明扼要**，对于集合求交部分省去了已在之前报告中详细介绍的算法描述，仅保留重点结论以及实现思想。

在本次期末报告中，我们额外补充了对**倒排索引压缩算法**的实现及其**解压算法并行化**的研究，希望对实际中倒排索引的压缩、解压与求交利用的过程进行整体优化。另外，我们使用了 OneAPI 对集合求交进行了并行化尝试，探索其并行化效果。（期末额外研究内容已在目录中用 * 标出）

对于倒排索引压缩部分，我们从两个方面对于倒排索引进行压缩：

- 变换压缩方面：我们实现了典型的数据变换压缩算法 d-gap。
- 整数编码压缩方面：我们实现了理论上压缩效果最好，但是**解压速度最慢**的整数编码压缩算法 OptPFD，它是经过优化过的 PForDelta 算法。

随后，针对两个算法的特点，我们从 OpenMP 并行化出发逐步添加上本学期所学习的 SIMD，MPI 等并行化策略对两种算法进行**多层并行化**，将构造倒排索引（读取压缩数据集并解压）的**效率提升了 5 倍以上**。另外，我们将 OptPFD 运用到经过 d-gap 变换过的数据上，从而最大化压缩比，使得**压缩比达到了 3.40**。

附唐明昊和朱世豪二人 GitHub 仓库链接。

2.2 实验环境

本项目一共在 4 个不同环境下进行过实验测试：

- arm 架构的鲲鹏服务器，配置如下：

  aarch64 架构

  L1 cache 64kB, L2 cache 512kB, L3 cache 49152kB

- amd 芯片的 x86 平台，配置如下：

  AMD Ryzen 5 5600H with Radeon Graphics, RAM 16.0 GB 22H2

  L1 cache 384kB, L2 cache 3.0MB, L3 cache 16.0MB

- intel 芯片的 x86 平台，配置如下：

  Windows 11 家庭中文版 64-bit (10.0, Build 22621)

  11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz (8 CPUs)，3.1GHz

  L1 cache 320KB，L2 cache 5MB，L3 cache 8MB

  内存容量：16384MB RAM

- DevCloud: Intel(R) UHD Graphics [0x9a60]

2.3 数据集描述

给定的数据集是一个截取自 GOV2 数据集的子集，格式如下：

1) ExpIndex 是二进制倒排索引文件，所有数据均为四字节无符号整数（小端）。格式为：[数组 1] 长度，[数组 1]，[数组 2] 长度，[数组 2]....

2) ExpQuery 是文本文件，文件内每一行为一条查询记录；行中的每个数字对应索引文件的数组下标（term 编号）。

通过对数据集分析得到，ExpIndex 中每个倒排链表平均长度为 19899.4，数据最大值为 25205174，这个数据在**构建 hash 表和 BitMap 位图**时，对参数的确定至关重要。查询数据集 ExpQuery 一共 1000 条查询，单次查询最多输入 5 个列表，可以据此设计测试函数。

另外，在给定的数据集以外，我们还自行设置了小数据集（见GitHub仓库）：每个倒排链表的长度在 50 至 100，生成 1000 个倒排链表，总计 300 次查询，每次查询 5 个单词。该数据集用于验证算法的正确性以及测试不同算法在小规模输入下的性能表现。

3 索引压缩 *

索引压缩算法通常从两个方向展开：

1. d-gap 变换与前缀和操作构成的数据变换方法以及文件重排技术等。
2. 整数编码算法：例如位对齐编码：Elias 编码、Golomb/rice 编码等；字节对齐编码：varint-SU、varint-GB 等；以及固定宽度编码：PForDelta、newPFD、OptPFD。

对于变换与重排方向上，我们选择对 d-gap 变换的优化进行研究。另外，对于编码算法，我们选择了固定编码中压缩比最高，但是解码速度较其他固定编码最慢的 OptPFor 作为终点并行化研究对象。

3.1 d-gap 变换算法

倒排列表 $l_t = \{d_0, d_1, ..., d_{n-1}\}$ 中的 docID 可能会很大，从而消耗更多存储位，如果我们对其进行变换 f，使得变换得到结果 $l'_t = \{d'_0, d'_1, ..., d'_{n-1}\}$ 中的 $d'_i < d_i$ 这样，我们就可以用较少的位来存储每一个 docID 项了，而如果变换 f 有逆变换 f^{-1} 那么我们就可以将 l'_t 无损还原至 l_t。

而对于已经有序的 docID 序列，有 $d_0 < d_1 < ... < d_{n-1}$，则该序列每个 docID 需要的存储位数由最大的 docID 也就是 d_{n-1} 所决定。若令 $l'_t = \{d'_0, d'_1 - d'_0, ..., d'_{n-1} - d'_{n-2}\}$ 则有

$$d'_i < \sum_{i=0}^{n-1} d'_i = d_{n-1}$$

这样，我们就能用更少的 bit 位来存储每个 docID 并且，只需要依次从第一项开始将前一项加至后一项，即可还原原来的 docID，因为这个变换是可逆的。

由此，得到串行算法如下：

d-gap 压缩算法（摘要）

```
// invertedIndex: 等待压缩dgap的倒排索引
// result: 压缩后的索引
// idx: 从第idx个bit开始压缩
void dgapCompress(const vector<unsigned>& invertedIndex, vector<unsigned>& result, int& idx){
    //...initialize here...
    // d-gap变换，后一项减去前一项,同时求最大间隔
    for (int i = 1; i < indexLen; i++){
        unsigned delta = invertedIndex[i] - invertedIndex[i - 1];
        deltaId.push_back(delta);
        if (delta > maxDelta)
            maxDelta = delta;//最大间隔
    }
    //...calculate indexLen, maxBitNum, bitCnt...
    writeBitData(result, idx, indexLen, 32);//写入索引的元素个数
    writeBitData(result, idx + 32, maxBitNum, 6);//写入每个docID需要的位数长度
    idx += 38;//从index+38位开始写压缩后的delta
    for (int i = 0; i < indexLen; i++){
        writeBitData(result, idx, deltaId[i], maxBitNum);
        idx += maxBitNum;
    }
}
```

在上述算法中，我们还需要使用额外 32+6 bit 的空间来存储整个倒排索引的长度以及进行 d-gap 变换后存储每个 docID 需要多少个 bit 位。这是因为每个 docID 最多 32 个 bit 位，而 32 在二进制表示下有 6 位，所以需要 6 位来存储。其实我们也可以用 5 位来存储用 11111b 来表示 32，而 00000b 表示 1，依次类推，但这里带来的压缩成本较小，我们不予考虑。

在解压时，我们对从文件中读出的 docID 作 d-gap 的逆变换，即可还原原来的 docID。

其中算法中向数组中读出与写入**非字节对齐位数据**（例如将 5bit 数据写入 unsigned 型的数组）的过程，同样较为关键，具体代码可见 GitHub 仓库链接。

3.2 OptPFD 算法

OptPFD 算法的基础思想是对于一个 chunk 的数列（例 128 个），认为其中占多数的 x% 数据（例 90%）占用较小空间，而剩余的少数 1-x%（例 10%）才是导致数字存储空间过大的异常值。因此，对 x% 的小数据统一使用较少的 b 个 bit 存储，剩下的 1-x% 数据单独存储。例如：有一串数列 23, 41, 8, 12, 30, 68, 18, 45, 21, 9, .. 如图3.1所示。取 b = 5，即认为 5 个 bit（32）能存储数列中大部分数字，剩下的超过 32（称为异常值 exception）的数字单独处理。可知超过 32 的数字只有 41, 68, 45。而这些数使得整个序列的 128 个 docID 都需要使用 7bit 的空间来存储，降低了内存的使用率。

图 3.1: OptPFD 存储思想

我们对这种情况进行改善，对于 90% 的数字，我们认为其为 “正常值”，可以统一使用这些正常值中最大的数字所需要的 bit 位数 b 来存储。而对于剩下 10% 的数字，我们首先用 b 位来存储它们的低位数字，然后在另外一块空间用 a 位来存储它们的高位，a 也是确定的，它的值等于是异常值中最大数值所需要的 bit 位数减去 b。同时，为了解压还原原序列，我们还要给每个高位前加上它们在原始序列中的索引 i。

如果将 OptPFD 算法与 d-gap 算法相结合，由于两者的压缩方式不同，能得到更大的压缩比，如3.4.1节所示，故实际中采用这种压缩方法。

由于篇幅原有限，这里只展示最终得到的文件的结构，如图3.2所示为正常数据部分的存储模式，而图3.3则展示了异常数据部分的存储模式。在文件中，两个部分是按 bit 位连续存储的，从而使得空间利用率最大化。

图 3.2: 正常数据部分存储结构

图 3.3: 异常数据部分存储结构

3.3 算法并行化

由于倒排索引集都是在离线期间进行一次压缩，然后在线查询处理过程进行多次解压。因此**解压速度**是衡量索引压缩算法好坏的重要指标，压缩速度则不是那么关键。为此我们对于倒排索引的解压算法进行并行化。

3.3.1 d-gap 解压算法

OpenMP 并行化 按照类似集合求交中使用的并行化技巧，可以从两个方面对 d-gap 解压算法进行并行化：

- 并行化外层循环，首先由主线程读出所有倒排索引的起始位置，然后依次将这些位置分配给多个子线程，示意让子线程解压一部分倒排索引，每个线程完成各自的压缩工作，最后进行汇总。
- 并行化内层循环，尝试将解压一个倒排索引列表的过程并行化。而 d-gap 的解压实际上是进行一个**前缀和操作**，将列表分为多个部分，每个线程进行部分前缀和，再多线程通信得到最终结果。

为了配合 MPI 多进程并行化，主要对内存循环即求前缀和的过程进行 OpenMP 并行化，而外层交由各个进程划分，以实现全过程并行。下面介绍通过 OpenMP 实现多线程前缀和。

1. 数组划分为多个段，每个线程负责数组一部分，进行局部前缀和。
2. 串行调整，将数组每段末尾加上前一段末尾的值，得到该位置正确的前缀和。
3. 再次启动多线程，每个线程对自己负责部分数组累加前一段末尾的值。该过程没有数据依赖，可以进行循环展开以及 SIMD 优化。
4. 串行处理数组末尾未被线程覆盖到的元素。

d-gap 多线程并行

```
vector<unsigned> dgapDecompressOMP(const vector<unsigned>& compressedLists, int& idx){
  // 预处理以及解决特殊情况
   ......
#pragma omp parallel num_threads(NUM_THREADS)
  {  // 分块求解前缀和
    unsigned delta = readBitData(compressedLists, localIdx, bitNum);// 找出该段第一个元素
      ......
    for (int j = 0; j < seq_num - 1; j++) {// 线程内进行前缀和
      delta = readBitData(compressedLists, localIdx, bitNum);
      localIdx += bitNum;
      result[tid * seq_num + j + 1] = delta + result[tid * seq_num + j];
    }
  }
#pragma omp single
  // 处理边界位置，方便后面并行使用
  for (int i = 2; i <= NUM_THREADS; i++)// 好像有隐式路障？
    result[i * seq_num - 1] += result[(i - 1) * seq_num - 1];
#pragma omp parallel num_threads(NUM_THREADS)
  {  // 加上其余线程计算结果
    int tid = omp_get_thread_num();
    if (tid != 0) // 0号线程不用做
      for (int j = 0; j < seq_num - 1; j++)// 其余线程每个元素加前面段的末尾元素
        result[tid * seq_num + j] += result[tid * seq_num - 1];
  }
#pragma single
  if (len % NUM_THREADS != 0) // 串行处理剩余元素
```

```
    ......
}
```

SIMD 并行化 前一部分在介绍多线程前缀和时已经提到，第二次多线程循环时，各个数据间没有数据依赖，很适合进行循环展开与 SIMD 优化。

思路上非常清晰，每个线程将使用 _mm_set1_epi32 前一段末尾元素填充到一个 128 位向量中；而后使用 _mm_loadu_si128 取出本段 4 个元素；再调用 _mm_add_epi32 即可一次完成四个元素求解；最后 _mm_storeu_si128 将 128 位向量放回原数组即完成了一次循环展开优化。

MPI 并行化 将解压算法进行 MPI 多进程并行化的思路同 OpenMP 大致相同，可以将所有倒排索引进行划分后分配给多个进程执行；或者优化内层循环，尝试将求前缀和的任务进行并行化。但是对于 MPI 而言，如果采用第二种方法进行并行化，由于每次将列表分块进行前缀求和后，每个进程都需要将本身所得到的最后一个中间值发送给下一个进程以实现全局累加，进程之间需要的通信成本较高，并不适合进行并行化的操作。

由此得到 d-gap 解压算法的 MPI 并行化代码如下：

d-gap 解压 MPI 并行化

```
void dgapDeMpi(const vector<unsigned>& compressedLists, vector<vector<unsigned>>& result,int
    rank, int proNum){
   //...读取每个链表的长度和每个docID所用的bit数...
   ...some codes here...
   // 分配任务，多进程开始工作
   int start = LIST_NUM / proNum * rank;
   int end = min(LIST_NUM / proNum * (rank + 1), LIST_NUM);
   for (int i = start; i < end; i++){
      vector<unsigned> curList;
      dgapDecompress(compressedLists, curList, listIndex[i]);
      result.push_back(curList);
   }
}
```

3.3.2 OptPFD 解压算法

OptPFD 解压算法大致可以分为三个步骤：

- 首先，读取正常部分的数据，从而得到每个数据的低位；
- 其次，读取异常部分的数据，以此记录下异常数据的索引，以及它们的高位
- 最后，利用异常数据索引值找到异常数据的低位，将低位与高位拼接，从而还原原始数据。

在实际压缩时，先使用 d-gap 变换，再使用 OptPFD 进行压缩，这样在解压时就需要加上第四步，也就是使用 d-gap 的逆变换还原每个 docID。下面我们对这四个步骤进行分析，尝试将其中的一些步骤并行化。

openMP 并行化 比较显而易见的是，在第四步中的 d-gap 逆变换过程中，依旧是同 d-gap 的多线程循环内并行优化相同，是一个求前缀和的过程，从而我们可以运用3.3.1节中求前缀和的策略进行优化。

另外，在读取正常/异常部分数据的过程中，每个线程仅涉及对正常数据的读取操作，而无其他复杂运算。故可将读取目标进行分段，随后分配给各个线程独立进行。

最后，对于利用异常数据索引值找到异常数据的低位中，由于每个异常值索引之间的**独立性**、**互斥性**，我们可以采用将这些索引分配给多个进程的方法来分配任务，让每个线程获取自己的索引后找到对应的异常值进行进行拼接。同时，利用任务池技术进行动态粒度任务划分，从而取得更好的并行化效果。

由上述的分析可以得到 OpenMP 部分算法如下：

OptPFD 解压 OpenMP 并行化

```
vector<unsigned> pfdDecompressOmp(const vector<unsigned>& compressedLists, int& idx){
    //---正常数据读取---
    ......
#pragma omp parallel num_threads(NUM_THREADS)
#pragma omp for
    for (int i = 0; i < u_normalLen; i++){//利用omp并行化
        int index = idx + i * u_normalBitNum;
        unsigned u_deltaId = readBitData(compressedLists, index , u_normalBitNum);
#pragma omp critical
        vec_u_deltaId.push_back(u_deltaId);//临界区，存入数组
    }
    idx = idx + u_normalLen * u_normalBitNum;
    //---异常数据读取---
    ......
    //读索引，读数据交替
#pragma omp parallel num_threads(NUM_THREADS)
#pragma omp for
    for (int i = 0; i < u_exceptionLen; i++)
        //读取索引u_exceptionIndex，异常高位u_exceptionHigh。normal位留出来，取或还原
        vec_u_deltaId[u_exceptionIndex] |= (u_exceptionHigh << u_normalBitNum);
    idx = idx + u_exceptionLen * (u_exceptionIndexBitNum + u_exceptionDataBitNum);
    //使用dgap逆变换还原docId
    dgapInverseTransform(vec_u_deltaId, result);//d-gap逆变换，使用omp
   ......
}
```

可见 OpenMP 的优化算法结合了 d-gap 解压中的前缀和优化方法，并且对于正常部分，异常部分数据的读取过程进行了多线程优化。

SIMD 并行化 在 openMP 优化的基础上，OptPFD 解压算法仍可继续使用 SIMD 并行化。可以从两个方面 SIMD 并行化，一是如同3.3.1节中的方法，对 d-gap 逆变换作并行化处理，这一点我们已经完成；二是对高位与低位的拼接过程进行并行化，利用 SIMD 中的指令一次左移多个异常值的高位，然后一次完成这些左移后的数与低位的或操作，从而完成拼接。由于这部分较为简单，代码不再展示，详细代码见 GitHub。

MPI 并行化 最后，我们还能再此基础上进一步利用**粗粒度划分**的手段，将倒排索引划分给多个进程完成。这一节的原理与3.3.1节中对任务的划分原理相同，故不再继续赘述。

3.4 实验结果

索引压缩算法的评价指标通常有如下两种:

第一个指标是**压缩效果**，更好的压缩效果意味着更少的存储空间和 I/O 开销，因而压缩效果是索引压缩算法的重要评价指标。

压缩效果可以通过压缩比 cr，即原始数据集大小与压缩数据集大小的比值来衡量；也可以通过 bits/int，即平均每个 docID 所需的位数来衡量。因为本文采用的索引集中的 docID 都是 32 位的，所以两者之间的关系为：bits/int = 32/cr。本文选择 bits/int 来衡量压缩效果。

另一个指标则是3.3节中提到的**解压速度**。

下面分别从压缩效果与解压速度两个方面来评价我们所实现的 d-gap 以及 OptPFD 算法。

3.4.1 压缩效果

各个算法的压缩效果如下表1所示。

压缩算法	Orignal	d-gap	OptPFD	d-gap+OptPFD
压缩大小	155471	109411	125107	45758
压缩比	1.00	1.42	1.24	3.40
bits/int	32.00	22.52	25.75	9.42

表 1: 不同算法下的压缩效果（大小单位：KB）

由表可知，单独使用 d-gap，和单独使用 OptPFD 算法时，d-gap 算法的压缩效果较好，bits/int 达到了 22.52，而如果同时使用两个算法进行压缩（先使用 d-gap 再使用 optPFD）压缩比以及 bits/int 比单独使用一种算法都要高，其中压缩比达到了 3.40，大于单独使用两种算法时的压缩比的乘积 1.42*1.24=1.7608，这是因为两者是从对于 docID 数据的变换以及整数本身的编码两个不同的角度出发进行压缩，从而最终达到了 **“1+1>2”** 的效果。

3.4.2 解压速度

d-gap d-gap 解压算法的并行优化效果如图3.4所示。其中横坐标表明了并行化方法的变换，”+=” 表示在左侧已有算法的基础上加入新并行化手段。纵坐标表示使用不同的优化策略后进行使用 d-gap 解压算法解压 2000 个倒排索引所需要的总时间。

可见当采用 MPI 与 OpenMP 相结合的算法时，并行化效果最好，解压速度最快。再次加入 simd 试图对该算法进一步优化时，算法运行速度反而变慢，这可能是由于支持 SIMD 的寄存器有限，细分过多需要等待系统调度而影响算法效率。

经过 MPI 与 OpenMP 的结合算法，d-gap 的读取压缩数据集并解压的速度较串行提高了 (5405.15-763.27)/763.27=6.08 倍。

OptPFD OptPFD 解压 2000 个倒排索引在不同并行化手段上的效果如图3.5所示。

由此可见对于 OptPFD 的解压算法，采用 mpi 与 omp 相结合的算法时，并行化效果最好，解压倒排索引速度最快，达到了 1000ms 以下。加入 SIMD 试图进一步优化反而略微变慢的原因可能与上一节相同。

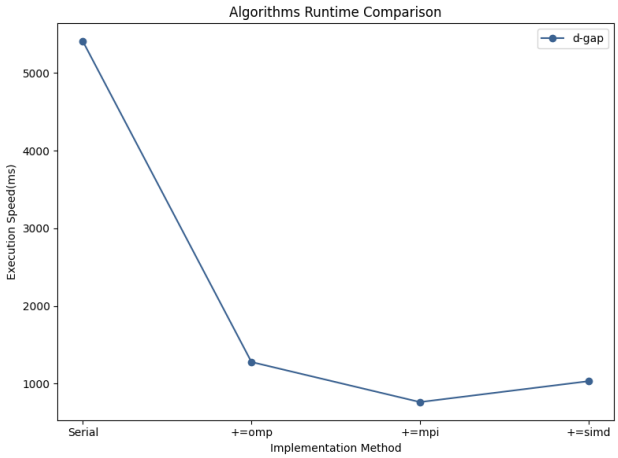

图 3.4: d-gap 解压算法的并行效果

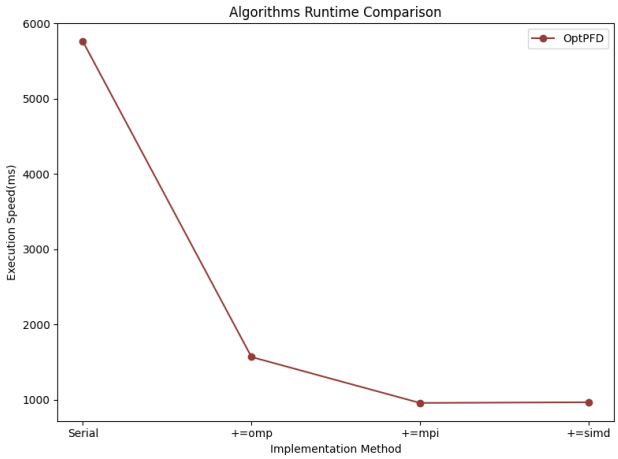

图 3.5: OptPFD 解压算法的并行效果

另外，还注意到，尽管经过了二次压缩，相较于 d-gap，OptPFD 解压串行算法的速度并没有明显下降，只是多使用了约 355ms。这可能是由于算法的压缩效果较好导致的。一方面，压缩比越高，所需要读取的文件的字节数就越少，从而在**文件读取**步骤使用时间更少；另一方面，压缩比越高，每个数据所需要的存储位数较少，而读出的向量使用 unsigned 类型进行单元的存储，所以向量中每个单元中所**包含的数据个数变多**，连续对数据进行访问时仅需访问一个 32 位单元即可，减轻计算资源的负担，提高 cache 的命中率，从而使得解压时间减少。

经过 MPI 与 OpenMP 的结合算法，OptFPD 算法的解压速度较串行提高了 (5760.32-956.20)/956.20=5.02 倍，结合上其优越的压缩比，算法整体效果很好。

4 集合求交

篇幅所限，串行算法代码以及大部分并行代码由于在之前报告中已有介绍，故期末报告中不再赘述，详见 GitHub 仓库。

4.1 算法描述

4.1.1 SVS 算法

SVS 算法即按表求交算法，先使用两个表进行求交，得到中间结果再和第三个表求交，依次类推直到求交结束。这样求交的好处是，每一轮求交之后的结果都将变少，因此在接下来的求交中，计算量也将更少。

在实现 SVS 算法过程中，为了最优化串行算法表现，结合了**拉链法**来加速求交过程：除了对列表按长度大小排序以外，在读取数据集构造倒排索引列表时，就对列表元素进行排序，每个索引列表都是有序的。在求交过程就可以结合拉链法，**发现大于关系即可跳出循环**。

4.1.2 ADP 算法

ADP 算法即按元素求交算法，在各个列表中寻找当前文档。当在所有列表中寻找完某个文档之后，查看每条列表的剩余的未扫描文档数量，只要其中一条列表无剩余元素，则本次求交结束。

同样的，为了加速串行算法执行效率，利用到初始将列表排序的技巧。另外，由于每次循环是拿出一个文档遍历所有列表，通过**记录每个列表上次访问位置**，下次再访问该列表越过前面无用文档，而不用去对已访问文档进行删除。

在实验过程中还发现，每次对剩余列表进行排序选出最短列表，**排序造成的时间消耗大于选择最短列表带来的收益**，于是选定最短列表后不再进行排序更改。

4.1.3 BitMap 算法

使用 BitMap 法即是使用 BitMap 结构来表示集合，将集合中的每个元素映射为一个唯一的二进制位，然后使用位运算来求解两个集合的交集储存结构如图4.6所示。

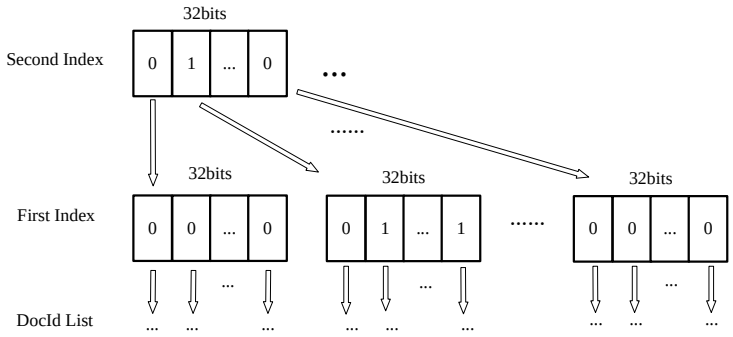

图 4.6: Bitmap 二级索引结构

在实现 BitMap 算法的过程中，发现可以沿用**跳指针法**的思路，从而减少在交集操作中不必要的比较次数。另外，在对 BitMap 作并行还对算法本身进行插桩检测，发现 BitMap 算法中最耗时间的部分在于对于 BitMap 结构进行复制的过程，而不在于 BitMap 算法求交的计算过程。由此，又进一步对其进行了优化，**从而将算法速度提升至 6 至 19 倍**。

4.1.4 Hash 算法

Hash 算法是受 bitmap 算法**用空间换时间的思想**启发，优化版本的 SVS 算法：为每个表构造 hash 表，将**该表的 DocID 依次映射到对应的 hash 桶里**，同一桶内元素按升序排列。在查找比较时，可以直接定位 hash 桶，再从 hash 桶找回比较的位置，在很短的范围内进行探查，不需要从头进行遍历。

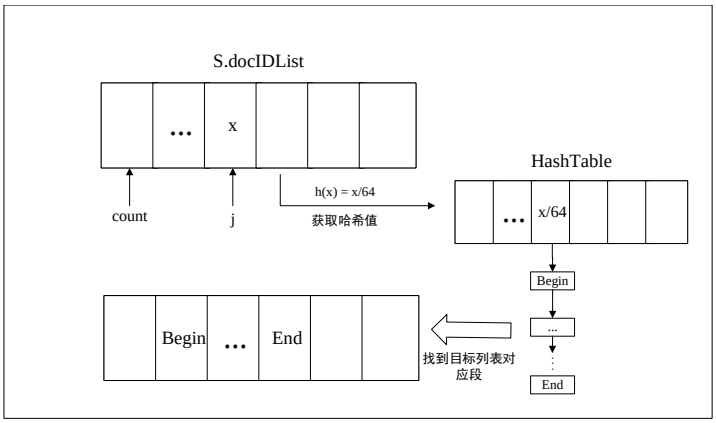

图 4.7: Hash 算法示意图

在算法编写过程中，对参数进行了多次调整，为了**最小化冲突**，即使得从 hash 桶定位回表中相应位置时，只需要很少的探查，最终构造 hash 函数：

$$h(x) = x/64$$

发现在该参数下，可以使 hash 桶内冲突尽量小，提高查询速率；同时不至于因为开辟空间过大导致内存爆炸，访问减慢等问题。

4.1.5 串行算法横向对比 *

对比在 arm 平台和 x86 平台下，SVS，ADP，Hash，BitMap 四个串行算法的执行效率。

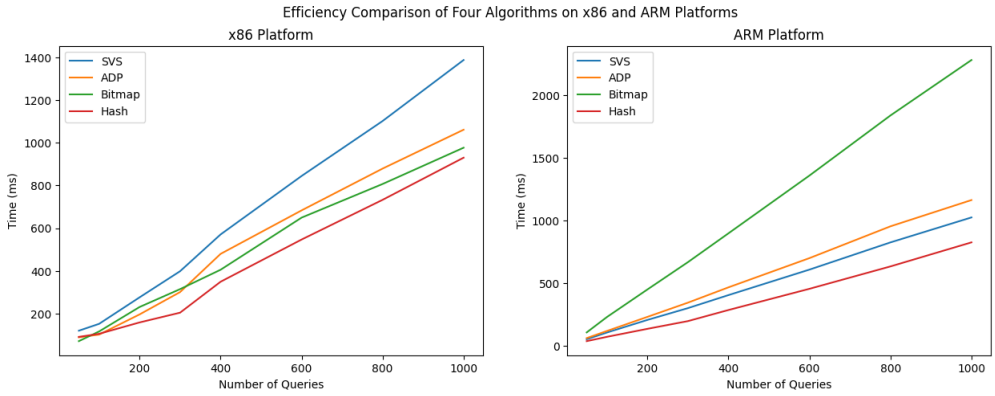

图 4.8: 各串行算法对比

可以看到 Hash 算法在 SVS 算法上进行优化，**有节制地用空间换时间**。求交时在 $o(1)$ 的时间复杂度就能定位到对应的 hash 桶，并且由于参数经过反复调试设置合理，hash 表内冲突少，极少的时间即可完成 hash 桶的遍历。时间复杂度接近 $o(n * l)$（n 为列表个数，l 为列表最长长度），于是运行效率最高。

SVS 和 ADP 算法二者效率接近，ADP 算法由于按元素求交，算法可以提前结束，不用遍历所有元素，而 SVS 算法虽然必须遍历所有列表，但**在经过拉链法优化以后**，同样有了提前结束的特性，二者时间复杂度都接近 $o(n*l^2)$。

BitMap 算法采用用空间换时间的思想，并且利用了**位图存储**，通过按位与即可完成一次求交操作，另外，朱世豪同学还对其融入了**跳表法**进行优化，时间复杂度为 $o(n*l)$。

另外**对比 arm 平台和 x86 平台**算法执行效率。除了 BitMap 算法，对于不同规模的测试数据集，均是 arm 平台相较于 x86 平台有着领先优势，这可能是因为 arm 架构的指令集更为简单，可以为算法停供更好的性能。

而 BitMap 算法在 arm 平台下性能落后于 x86 可能是由于其开辟了大量空间，鲲鹏的 cache 较小，会造成大量的 cache miss。当测试数据集规模不断增大时，BitMap 算法在两平台所表现出来的性能差距逐渐增大。

4.2 SIMD

4.2.1 并行实现

通用并行化 对 SVS，ADP 算法考虑对最内侧循环，两列表求交遍历元素进行并行化操作。为了加速列表遍历过程，DocID 一次比较当前列表四个元素：用 _mm_set1_epi32 指令将一个 DocID 填充向量四个位置，再从待比较列表取出四个元素放入另一向量，进行 128 位比较。

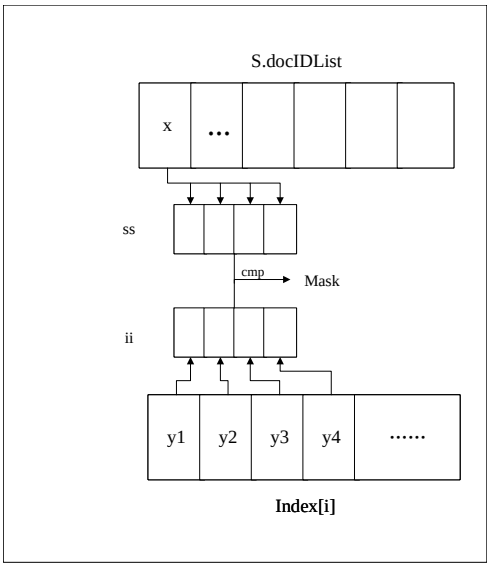

图 4.9: SIMD 通用并行示意图

通过对比较指令 _mm_cmpeq_epi32 的返回值的研究，再结合前期调研的 HIGHLY SCALABLE 博客 [9] 中的方法，发现并不需要去进行遍历比较结果，**使用 __mm__movemask__epi8 指令生成掩码**，可以加速求交。因为四个元素至多只能有一个元素与 DocID 相匹配，当发生匹配时，对应位置置 1，则最终生成的掩码不为 0，一个判断条件即可完成。

Hash 特殊并行化尝试 对于 Hash 优化算法，除了采用上述并行算法进行优化，还思考了从 hash 函数出发，修改 hash 函数，使得一个向量可以存储更多的元素。具体来说令

$$h(x) = x/65536$$

这样位于同一个 hash 桶内的元素，高 16 位值是相同的，区别仅在于低 16 位，进而可以使用一个 short 保存，于是一个 **128 位的向量即可以保存 8 个不同元素**，理论上可以提高算法执行效率。

Hash 尝试代码块

```
short* sArray = new short[L];
sArray[j]= (short) (index[i].docIdList[begin+j] & 0x0000FFFF);
__m128i ii = _mm_loadu_epi16(sArray + t);
result = _mm_cmpeq_epi16(ee, ii);
```

但经过实测发现，该算法虽然通过比较每个元素的低 16 位，增加向量中保存元素数目来提高并行效率；但由于增大了 hash 函数的参数，**使得 hash 表中的冲突增多**，即同一 hash 桶内元素增多，比较次数陡增，造成了最终效率远低于通用并行化算法。

BitMap 并行化 Bitmap 算法使用位图存储方式——每条链表用一个位向量表示，每个 bit 对应一个 DocID，某位为 1 表示该链表包含此 Doc、为 0 表示不包含。从而求交运算就变为两个位向量的位与运算，更适合 SIMD 并行化。

由于 BitMap 本身利用了以空间换时间的思想，使得数据之间依赖性被消除。所以在 BitMap 中的多级索引的结构中，比较容易对每层求交的过程进行并行化。一次对多个二进制位进行与操作，从而比较多个二进制位，达到比较多个 docID 的效果。

BitMap 算法 SIMD 并行化

```
//...最内层求交算法展示...
for (int l = (t+c) * 32; l < (t+c) * 32 ; l+=4){
    __m128i cIndex = _mm_loadu_epi32(&chosenListSSE.firstIndex[l]);//关键字比较
    __m128i cOpIndex = _mm_loadu_epi32(&bitmapQListsSSE[i].firstIndex[l]);
    __m128i comRes = _mm_and_si128(cIndex, cOpIndex);//按位与
    _mm_store_si128((__m128i*)&chosenListSSE.bits[l], comRes);
}
```

4.2.2 实验结果

由图4.10还可以看出，除了 SVS 算法以外，其余算法在两个指令集下表现差异并不大，除了受到数据集的影响以外，可能还因为 Hash 算法和 ADP 算法最内层循环每次迭代的次数已经优化到很少，每次处理 4 个或是 8 个元素对其性能影响不大，更多的会受到后续条件分支判断的制约。而 SVS 算法每次遍历一个完整的列表，迭代的次数较多，故一次处理 8 个元素可以使得其性能提升明显。而由于 Bitmap 算法开辟空间过大，占用内存过多，导致 cache miss 过多，导致使用 SSE 和 AVX 指令集时取数据 miss 次数多，抵消了一部分并行化带来的收益，对于这一点我们使用了 perf 进行了验证。

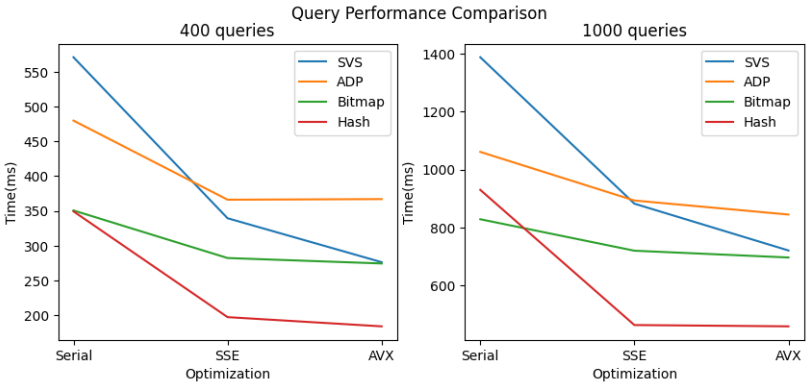

图 4.10: x86 下串行与 SSE、AVX 并行对比

4 个算法在各种 SIMD 指令集下并行加速效果如表2所示：

Events	NEON	SSE	AVX
SVS	20.69%	36.39%	48.02%
ADP	12.18%	15.80%	20.36%
Hash	42.07%	50.11%	50.60%
BitMap	5.57%	13.10%	15.19%

表 2: SIMD 并行加速比

4.3 PThread

多线程实验优化可以从 query 间与 query 内两个角度入手。

query 间即是在最外层循环进行循环划分，每个线程独自处理一部分 query。由于 query 与 query 间并没有很强的依赖性，所以**较为适合并行化**，并行化效果好。

query 内是对一个 query 求交过程中进行并行化。因为算法特性，该并行方法易造成负载不均，导致算法执行效率低下，具体描述见4.3.3节。可以利用加速大概率事件，多重循环 barrier，任务池等优化手段进行加速。

除了 query 间并行与 query 内并行，根据主线程将任务分配给子线程的位置不同，还可以分为**动态并行与静态并行**。

动态并行策略的优点是子线程随开随用，实现简单。子线程在执行完各自的任务后都被销毁，不会在请求空闲时占用系统的资源，不会造成资源的浪费。

静态并行策略的优点是子线程随时取用，无需开辟。子线程完成自己的任务后不会被销毁，而是进入**空闲等待**状态，以便于之后的随时取用。节省了开辟线程、销毁线程的时间开销，在请求拥挤时，能够节省大量时间。

4.3.1 query 间并行

不管是动态并行还是静态并行，均是由主线程将大量 query 平均分配给多个子线程，子线程调用相应的倒排索引求交算法，对于倒排索引求交问题进行计算。

下面以 query 间静态并行为例，介绍 query 间并行的思想。

在开始查询之前，开辟线程。线程没有任务时处于睡眠状态，主线程获取到需要处理的查询任务后，根据线程总数平均划分任务，再**使用信号量唤醒子线程**。子线程做完一轮工作后自动挂起，等待主线程下一次唤醒。

query 间静态并行-主线程

```
// 唤醒工作线程开始任务
for (int i = 0; i < NUM_THREADS; i++)
    sem_post(&sem_worker[i]);
// 主线程睡眠，等待子线程完成任务
for (int i = 0; i < NUM_THREADS; i++)
    sem_wait(&sem_main);
```

query 间静态并行-工作线程

```
while (true){
    sem_wait(&sem_worker[id]);// 阻塞，等待主线程唤醒
    if (staticFlag == false) // 总控制信号被主线程置为false即跳出
        break;
    int skip = queryCount / NUM_THREADS;
    int start = id * skip;
    int end = id + 1 == NUM_THREADS ? queryCount: (id + 1) * skip;
    for (int i = start; i < end; i++) // start到end个查询
        Call Function...
    sem_post(&sem_main);// 唤醒主线程
}
```

4.3.2 实验结果

query 间并行由于其良好的性质，任务划分相对均匀，故算法执行高效。各个算法实际执行时间对比如图4.11所示。不同规模下加速差异表现差距不大，加速效果均能达到 2.5 倍以上。

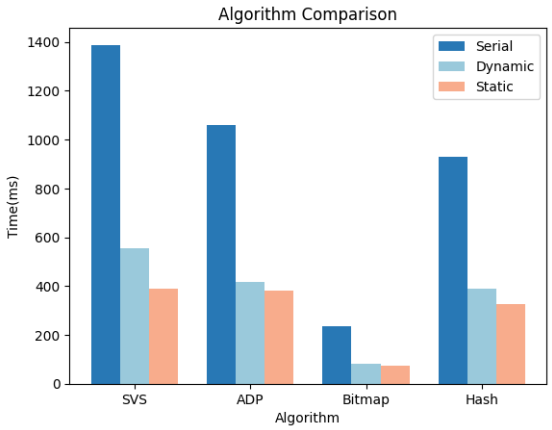

图 4.11: query 间并行加速效果

静态相较于动态，省去了繁杂的创建销毁线程操作，只需要通过信号量来唤醒工作线程即可。由上图可以看出，静态方法较动态都有一定提升。但均未能达到极致的三倍以上加速比，原因可能是负

载不均：虽请求的任务个数相同，但是 query 内的工作量的大小并不相同。4.4.1节对这一问题进行了详细描述，并在 OpenMP 中使用**任务池**策略对其进行优化。

4.3.3 query 内并行

query 内并行的整体思路是在一个 query 的求交过程中，每个线程从当前链表中取出一段子链表，用于将其与当前 query 下的其余几个链表进行求交。为使得每个线程的工作量相对均匀，每个子链表的长度为 len/NUM_THREADS。

在之前的实验报告中已经得出，动态版本开辟销毁线程的开销很大，且因为 query 内并行**循环嵌套深入**，动态开辟销毁线程的次数很多；而静态版本通过占用更多的系统资源，开辟线程后只需要用信号量进行唤醒与休眠。由图4.17也可以看出，两个版本性能差异巨大，故以下只介绍静态版本的 query 内并行。

query 内的静态并行与 query 间的实现类似，具体算法函数内每处理一个链表就使用信号量来唤醒工作线程，工作线程开始工作，做完以后保持休眠。只有当所有 query 任务做完以后，调整标志位，工作线程才会结束。

4.3.4 query 内并行优化

由于 SVS、ADP 算法为了提高表现，引入了拉链法，即链表前面的元素会帮助后面的元素移动求交链表中的指针，算法会**提前结束**。而在并行化分段以后，位于后面段的元素失去了前面段的信息，需要从头往后寻找匹配。

又因为链表都是升序排列的，故在本链表中位于后半部分的元素，**在另一链表中大概率也会位于后面**。这将导致部分线程遍历链表的长度很大，也即**工作量不均等**，最终性能低下。

加速大概率事件 使用加速大概率事件的思想进行预处理：根据当前线程的 t_id 调整在目标链表中遍历的起点。每个线程比较自己负责段起点元素与目标链表对应段元素，大于则直接从该位置开始遍历求交，小于则回退一个段的长度，重复判断，最坏情况下回到链表起点。

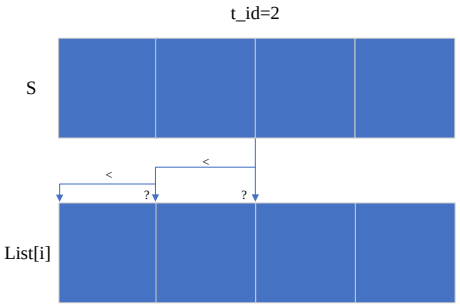

图 4.12: 加速大概率示意图

图4.17与图4.13中 PThread 的 query 内并行均为使用了加速大概率预处理的结果。

任务池优化 总的实现思路是声明一个全局变量 task，从原本的每个线程根据自己的 t_id 来取出自己的任务改为由当前的 task 值来取出当前的任务，并对 task 变量加锁以保证每个子线程取到的任务互不重叠。同时，更改子线程唤醒主线程的时机，子线程做完自己的一份工作后并不会立即向主线程汇报，而是等所有子线程均完成工作之后，再向主线程发出信号。

任务划分的方法尝试了多种方法，以保证各个线程负载均衡，并尽量减少任务池分配任务时的加锁解锁带来的额外开销。实际运行结果表明，如下划分方式最好：

当 L>300 时，划分 300 块任务给子线程，如果长度小于 300，认为工作量小，按照长度均分四块任务划分给子线程。

ADP_TaskPool 算法要点

```
int length = list[0].length>=300?list[0].length/ (list[0].length/300):
    list[0].length>4?list[0].length/4:list[0].length;//第三种划分策略
while (end < list[0].length){//让子线程做完求交的任务再向主线程报告
    pthread_mutex_lock(&mutex);
    //分配任务
    Assign Tasks codes...
    task++;
    pthread_mutex_unlock(&mutex);
    if (begin > list[0].length)//已经做完所有任务->报告
        break;
    //用Adp做求交工作
    ADP-work codes...
}
sem_post(&sem_main);// 唤醒主线程，报告所有线程已经完成
```

实验测试的结果如图4.13所示，可以看出任务池的优化有效，能够合理分配任务使得 query 内的负载更加均衡，从而减少 query 内静态算法的运行时间。

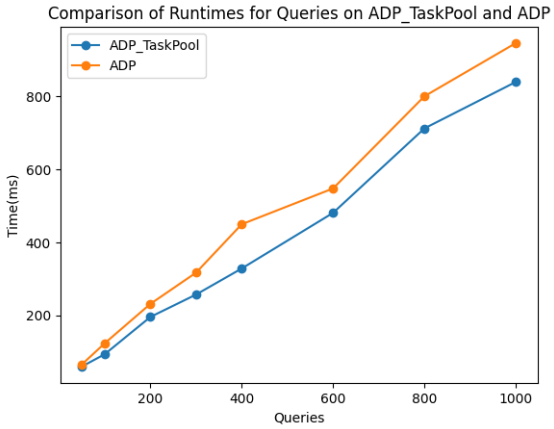

图 4.13: query 内静态任务池优化（使用加速大概率预处理）

BitMap 特殊优化 由于 BitMap 中耗时占比最大的实际为 BitMap 结构体的赋值过程，而不在于核心计算代码，我们进行多线程并行化的重心也应该放在优化 BitMap 的赋值过程。

由此采用**分段赋值**的方法对该赋值语句进行并行优化，每个子线程为减少资源负担，对于 chosenList 进行分段初始化。

BitMap 多线程优化 (Excerpt)

```
//根据t_id计算二级索引的赋值起始点和结束点
Calculate skipSec, startSec, endSec...
//根据t_id计算一级索引的赋值起始点和结束点
```

```
Calculate skipFir, startFir, endFir...
//根据t_id计算真正索引的赋值起始点和结束点
Calculate skipBit, startBit, endBit...
//根据t_id计算子线程迭代器的结束点
Calculate EndSec, EndFir, EndBit...
//子线程执行自己负责的copy复制
copy(bitmapList[qList[0]].secondIndex.begin()+startSec,EndSec, chosenList.secondIndex.begin() +
    startSec);
copy(bitmapList[qList[0]].firstIndex.begin()+startFir, EndFir, chosenList.firstIndex.begin() +
    startFir);
copy(bitmapList[qList[0]].bits.begin()+startBit, EndBit, chosenList.bits.begin() + startBit);
```

实验测试的结果如图4.14所示。BitMap 算法本身执行时间很短，动态开辟、销毁线程次的开销对算法本身造成不可忽视的影响，所以动态多线程效果差。静态版本略微领先与串行版本，这也是由于阿姆达尔定理，算法已经达到了很快的执行速度，优化效果边际递减。

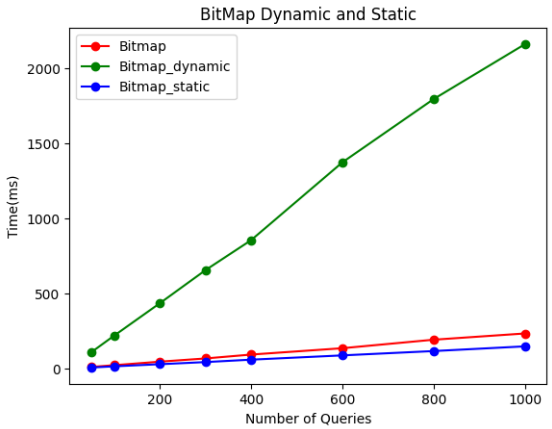

图 4.14: query 内 BitMap 算法效果对比

4.3.5 线程数影响探究

如前所述，query 内并行化尤其是动态开辟线程版本，执行效率低下。原因是 SVS 与 ADP 串行版本提前结束的特性，导致并行情况下有些子线程处理的数据段工作量庞大，负载不均。另一方面动态版本大量地开辟销毁线程，造成了严重的资源浪费。

为此，我们尝试**减少线程数**。一方面可以减轻线程间负载不均匀的现象，优化算法表现；另一方面可以探索线程开辟销毁对算法执行的具体影响。

静态并行版本测试数据如表3所示，可以看出该情况下 4 线程版本效率是领先于 2 线程版本的。证明在排除线程开辟销毁的影响后，负载不均带来的危害可以通过**多线程并发有效的弥补**。另一方面也说明了前述的**加速大概率**处理在一定程度上减轻了负载不均的影响。

而动态并行版本经过实测，两个线程的版本速度明显快于四线程。证明由于处于循环深处，动态开辟销毁线程带来的影响是巨大的。将线程数减半，算法的执行效率提升了 25% 以上。

Queries	SVS_thread4	SVS_thread2	ADP_thread4	ADP_thread2
100	148.097	163.704	92.7588	102.157
200	248.011	314.615	195.088	199.659
400	517.228	635.601	328.106	404.731
800	1067.64	1348.72	712.668	815.441
1000	1342.25	1668.56	840.149	1030.62

表 3: query 内静态对比（单位：ms）

4.4 OpenMP

4.4.1 query 间并行

openMP 实现思路整体与 PThread 类似，但具体实现上，借助 openMP 的相关 API，无需手动开辟、销毁线程，不需要单独写线程函数，可以很简单的实现相关功能。由下图可以看出，二者性能表现上基本一致。

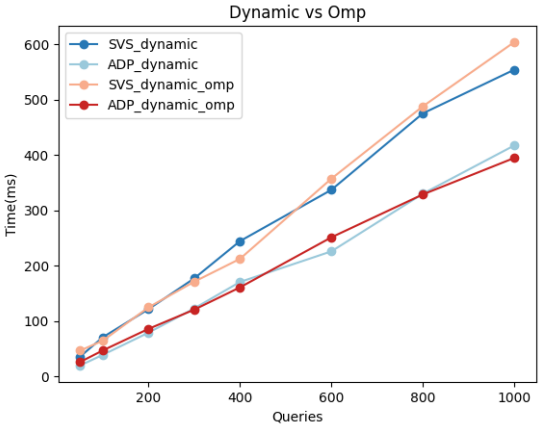

图 4.15: query 间并行 PThread 与 OMP 对比

任务池优化 虽然 query 间并行化结果高效，但并没有到此结束，因为并没有到达 3 倍以上的加速比，推测是由于数据集造成。

数据集 query 分布可能并不均匀，虽然分配的 query 总数相同，但不同 query 查询的项数并不一定相同；另外即使项数相同，不同链表组合求交工作量也不一定相等。以上原因都可能导致各个线程间负载不均，进而影响总的算法执行效率。

尝试**任务池优化**，在 #pragma omp for 后加上 **schedule(dynamic,1)** 子句，各个线程动态获取 query，保证线程间任务均等。

最终测试效果如图4.16所示，除 BitMap 算法外，其余算法均获得提升，**Hash 算法加速比突破 3**。

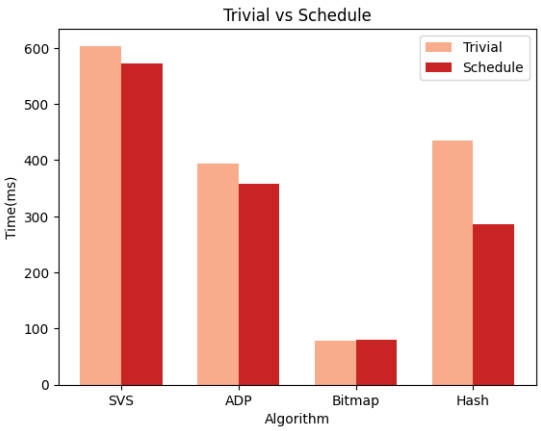

图 4.16: query 间任务池优化

BitMap 算法提升不明显的原因可能是 BitMap 将 DocId 转换成二进制位，每次求交只需要将几个链表做与运算即可，不同 query 执行时间差别不大。

4.4.2 query 内并行

使用 OpenMP 进行 query 内并行化，总体思路与 PThread 保持一致。在主函数中使用 OpenMP 开辟线程，将任务划分给子任务，同时利用 critical，single 关键字分别指明要进行加锁的部分和只需要执行一个线程执行的区域，从而将四个串行算法借助 PThread 的并行思路转化为简洁的 OpenMP 版本。

下面以 query 内并行的 hash 算法为例，展示使用 OpenMP 的多线程实现：

Hash_Inquery

```
#pragma omp parallel num_threads(NUM_THREADS),shared(count)
  for (int i = 1; i < num; i++) {// 与剩余列表求交
    // SVS的思想，将s挨个按表求交
      ......
#pragma omp for
    for (int j = 0; j < length; j++) {
       ......
       // 找到该hash值在当前待求交列表中对应的段
       int begin = hashBucket[queryList[i]][hashValue].begin;
       int end = hashBucket[queryList[i]][hashValue].end;
       // 遍历begin到end，查找
          Some Codes Here...
#pragma omp critical
          if (isFind) // 覆盖
             s.docIdList[count++] = s.docIdList[j];
    } }
#pragma omp single
    if (count < length)// 最后才做删除
       s.docIdList.erase(s.docIdList.begin() + count, s.docIdList.end());
  }
```

虽然 OpenMP 开辟线程的位置是内部循环，似乎与 PThread 动态开辟线程一致，但最终实验分析发现，其速度与静态线程更加相似。

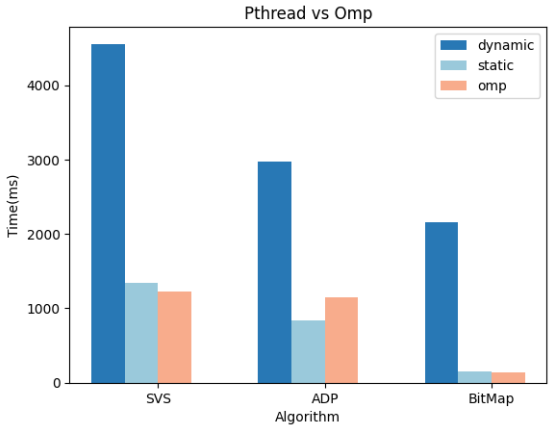

图 4.17: query 内 OMP 与 PThread 对比

经过 VTune 分析得知，OpenMP 在子线程结束后不会立即销毁子线程，而是自动将其挂起，节省了大量创建、销毁线程时间开销从而达到了 PThread 中静态线程的效果。

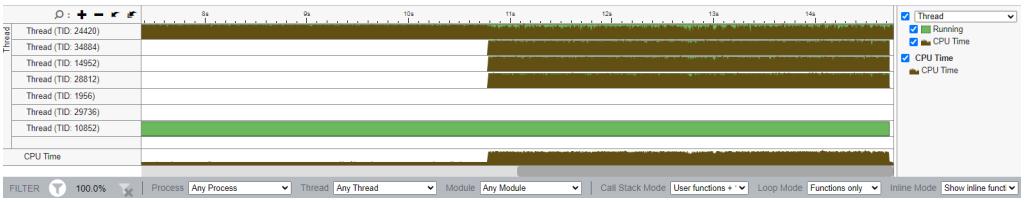

图 4.18: query 内 SVS 算法 VTune 分析结果

4.4.3 算法伸缩性分析

由伸缩性的定义，以并行加速比除以计算使用的计算资源总量，即可得到效率。以 ADP 为例对不同线程数下 OpenMP 优化的 query 间并行化伸缩性进行研究。

	ADP	ADP_2Threads	ADP_3Threads	ADP_4Threads
1000 Queries(ms)	1061.34	684.75	473.01	394.51
加速比	1	1.55	2.24	2.69
效率	1	0.77	0.75	0.67

表 4: 资源总量变化时的效率变化

从表4中的加速比的计算结果可以看出，随着线程数的增加，算法的加速比逐渐增大，但增长速度逐渐减慢。这可能是由于算法中存在一些串行部分或线程间的通信开销导致的。因此，从强伸缩性的角度来看，该算法的**伸缩性较好，但仍然存在一定的限制**。

另一方面，随着线程数的增加，算法的效率逐渐降低，这意味着每个线程所能贡献的性能逐渐减少。可能是由于线程间的竞争和调度开销增加导致的。**算法并没有完全利用系统资源来获得更好的性能提升**。

4.5 MPI

MPI 实验分别从 MPI 的 Query 间并行化与 MPI 的 Query 内并行化两个角度入手。

Query 间即是在最外层循环进行循环划分，并行化效果好。Query 内并行难度较大，是我们重点攻克的对象，实验过程中对其通信机制等优化做了许多尝试。

4.5.1 并行实现

query 间并行 在最外层循环，**粗粒度**地为每个进程划分工作任务，分配一定量的查询任务，每个进程调用相应的倒排索引求交算法。进程间依赖较少，并行化效果较好。(算法具体思路与4.3.1节介绍的类似，之前的实验报告也有描述，故不再赘述)

此外，Query 间 MPI 并行很适合结合 OMP 与 SIMD 等并行化手段，MPI 进行粗粒度并行，后者细粒度并行，做到全过程并行化。详细请参考4.6节。

query 内并行 算法的具体思路以及存在的问题在之前的报告中已详细说明，在4.3.3也有介绍，故不再重复说明。本文仅介绍算法的重点改进部分与通信机制的实现。

在算法求交结束以后，非 0 进程遍历自己的工作范围，找出求交成功的元素，放进 int 数组里，而后使用 MPI_Send 发送给 0 号进程，0 号进程收集汇总求交结果。

在发送数据时，为了避免缓冲区超限，采取两次发送，第一次发送求交成功元素的个数，接着再发送具体的数据。这个通信机制导致 0 号进程 MPI_Recv 时不得不按顺序，分 tag 的接收，否则可能出现长度与数据不匹配等问题。对通信机制的优化将在下一节介绍。

4.5.2 通信机制优化

精确负载 为了进一步解决各个线程之间负载不均的问题，提出了先二分查找、精确划分任务，再开始任务的方法。

当用于参考的首链表被划分给 4 个进程，每个进程在剩余倒排链表中依旧只能整体扫描一遍，而没有在对应能够得到有效结果的区域内进行扫描，这就会导致时间上的浪费。如果在正式开做求交算法之前先把 1 到 num-1 倒排链表遍历一遍，划分出来每个线程要做的部分。随后每个进程在这些部分中扫描求交，这样会使得负载均衡的程度大大提高，因为每个进程不再做“无用功”。

对于搜索方法，选择的是时间复杂度为 $O(logN)$ 的二分查找。

精确负载机制

```
// 计算每个进程的第一个倒排链表的开始位置和结束位置
......
// 找到剩余倒排链表的起始位置和结束位置
for (int i = 1; i < num; i++)// 二分查找，找到自己的开始位置start[i-1]
    ......
if (rank != proNum - 1){// 最后一个进程不需要知道自己何时结束，因为它直接扫描到最后
    // 从下一个进程接受该进程的起始位置
    MPI_Recv(end, num - 1, MPI_INT, rank + 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (int i = 1; i < num; i++)
        list[i].end = end[i - 1];// 下一个进程的开始正是这个进程的结束位置
}
else// 最后一个进程做到底
    for (int i = 1; i < num; i++)
```

```
        list[i].end = index[list[i].key].docIdList.size();
if (rank != 0) // 第一个进程不用发送信息，其余进程发送给前一个进程
    MPI_Send(start, num - 1, MPI_INT, rank - 1, 1, MPI_COMM_WORLD);
```

从表5可以看出，在加入精确负载机制后，各个进程之间的**负载不均问题明显得到改善**，但是算法的运行时间上总体却总体没有没有得到显著的提升，这可能是由于算法引入二分查找和进程之间的通信所带来的额外时间开销抵消了部分负载均衡所带来的收益。

	ADP_MPI2	ADP_MPI_Precis
进程 0	952.68	855.10
进程 1	952.63	855.07
进程 2	869.13	849.90
进程 3	912.14	849.88

表 5: 算法执行时间对比

减少通信 之前采取的通信机制需要 0 进程使用 for 循环依次接收其余线程发送的消息，存在一种可能，某个进程已经做完工作，而 rank 比它小的进程未做完工作，则该进程需要一直阻塞等待；另外，之前的通信需要先发送长度，再发送数据，分两次进行发送，同样可能造成时间上的浪费。

考虑使用 **MPI_ANY_SOURECE**，0 进程每次循环任意接收数据；再将长度放到数据数组的 0 号位置。以上，减少通信等待与通信次数，探索通信对算法执行效率的影响。

但该方法在发送消息前需增加 **MPI_Barrier** 同步各个进程，保证各进程发送消息时都在同一函数步内。否则可能出现某个非 0 线程工作过快提前进入下一函数步，抢先其他进程，多次发送消息。

由表6中实验结果可知，该通信机制优化最终并没有提升算法的执行效率，可能的原因如下：

- 两次通信对算法影响并没有那么大。
- 由于前述提到的负载不均的问题，rank 大的进程相对来说工作量更大，导致 rank 较大进程长时间等待 rank 比它小的进程出现频率较小。
- 即使使用了 MPI_ANY_Source 让进程可以到下一函数步内提前做一些工作，但 barrier 同步机制也在一定程度上让进程阻塞等待。

非阻塞通信 非阻塞通信是在前述减少通信基础上的进一步尝试。为了进一步提高进程利用率，减少阻塞等待时间，使用非阻塞通信，非 0 进程在 Isend 数据后即进入到下一个函数步去，减少两个进程拥挤等待的时间。

由表6可知，非阻塞通信有效的提升了减少通信的效率，基本与两次通信保持了一致。无法避免的 barrier 同步造成的时间消耗，与该通信机制对特殊情况阻塞优化获得的时间收益实现了对冲。

	Hash_divide	Hash_packed	Hash_isend
进程 0	1768.13	2066.91	1801.26
进程 1	1355.30	2045.47	1797.47
进程 2	1795.39	2053.40	1781.06
进程 3	973.91	2042.44	1798.89

表 6: 执行时间对比（hash 算法参数取 512）

集合通信 集合通信是对进程汇总过程进行优化的尝试。我们使用 MPI 中的 Gather 函数对线性的结果汇总过程进行优化，使得进程的平均等待时间减少，避免木桶效应，从而实现更好的通信机制。

由表7使用通信机制后有两个进程比其他进程快出很多，这是由于使用 gather 后进程能够更快的进入下一个函数步，然而**这一特点导致了负载不均衡**，从而在最后一行所展示的各进程运行算法所需的最长时间上，最终使用集合通信这一机制的算法的表现反而不如未使用该方法。

	SVS_MPI2	SVS_Commun	SVS_Precise	SVS_PreCom
进程 0	642.89	929.34	827.70	871.28
进程 1	953.69	1175.59	833.59	871.32
进程 2	1115.89	445.59	827.83	243.53
进程 3	1115.96	1175.56	833.69	220.14
Max	1115.96	1175.60	833.69	871.32

表 7: 集合通信

4.5.3 加速比异常探究

在实验过程中我们发现一个问题，鲲鹏平台并行加速异常的快，加速比已经超过进程数，这是超乎我们预料的。

我们实验探索后发现，在鲲鹏服务器上，只要使用 MPI 程序进行编译，即使没有对进程进行任务划分，即每个进程串行地都执行完整的 1000 个 Query，**算法执行时间都是有明显下降的**。然而如果是在自己的机器（x86）上实行同样的操作，程序的运行结果反而比串行所消耗的时间（1387.76ms）更多。详细数据见表8。

	SVS_ARM	SVS_MPI_ARM	SVS_x86	SVS_MPI_x86
进程 0	151.08	32.056	1948.52	468.47
进程 1	154.39	36.719	1991.06	508.33
进程 2	157.55	37.098	2047.03	515.29
进程 3	158.03	40.072	2012.91	495.19
加速比	-	3.94	-	3.97

表 8: 串行执行（使用 MPI）与 MPI 并行执行

如果**以开了 MPI 编译选项的串行数据做为串行执行时间**，那么 arm 平台下，MPI 并行化的加速比回归正常，x86 的加速比也变大，二者几乎一致。

造成该现象的原因,我们查阅资料认为可能与不同架构和系统对资源的调度有关,具体解释见 MPI 实验报告。

4.6 MPI+openMP+SIMD 混合并行

在 MPI 实验中，我们探索了将**各个层次的并行化算法结合**起来，最大化加速效果。

多进程与多线程的结合上，在外层循环即 query 间进行 MPI 并行，在 query 内进行 OpenMP 并行。具体来说，每个进程负责一部分查询，查询求交过程中进程内每个线程负责倒排索引链表的一部分，求交时进行共享内存的通信。

但由于 SVS 和 ADP 提前结束的算法特性，段与段之间有着**隐式依赖**，query 内任务划分很容易造成负载不均，这在4.3.3中已经提到过，实验证明该方法性能表现也确实低下。

故没有采用如上结合方式，转而在 Query 间 MPI 的基础上，配合使用 Query 间 OpenMP。每个进程获得一定量的查询任务，进程再将查询任务细分给 NUM_THREADS 个线程，最大化加速算法。

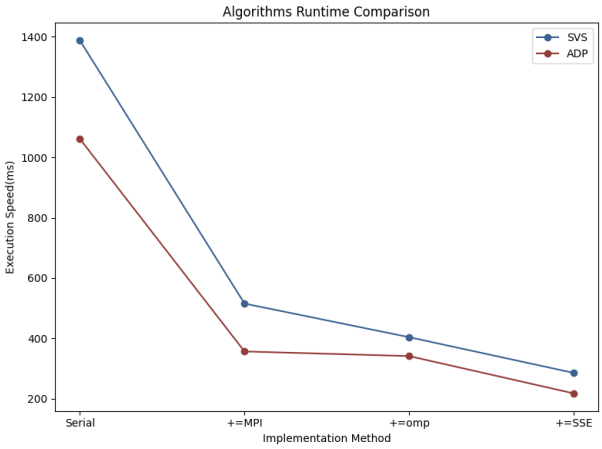

图 4.19: 各层次并行算法混合

可以看出，在增加了 OpenMP 多线程任务细化以后，算法的执行效率得到了进一步的提升。相较于单独 MPI 并行，增加了 OpenMP 以后，两个算法分别取得了 21.6% 和 4.3% 的性能提高。

外层循环已经进行了细致的任务划分，在内层求交时使用 SIMD 指令（一次比较多个数据，生成掩码）加速算法求交的比较过程。如图4.19，在增加了 SSE 以后，算法性能比普通 MPI 并行分别提高了 44.6% 和 38.9%，相较于串行算法加速比达到 4.86 和 4.88。

4.7 CUDA

CUDA 实验分别从**粗粒度**和**细粒度**两个角度入手。

4.7.1 粗粒度实现

粗粒度实现是使用类似于 pthread/openMP 并行化的方法，将 GPU 视为一个若干线程集合，每个线程取出当前链表的一部分（length/totalThreads）与当前 query 下的其他几个链表进行求交，求交结果放入数组 index 里。每个线程只会放入他所管理的那一段，所以并不会造成冲突的问题。

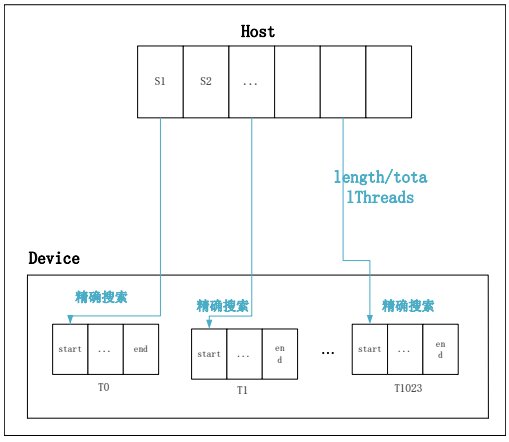

图 4.20: 粗粒度算法示意图

算法首先获取当前线程的索引，再计算总的线程数，据此计算出该线程所负责计算的链表部分。而后算法核心计算部分，每个线程在小范围内与其他链表进行求交。(算法计算核心逻辑前面实验已介绍，不再赘述)

该方法任务划分简单，host 端与 device 端**通信较少**，数据转移成本小。但由于 SVS 和 ADP 串行算法**提前结束**的特性（详见4.3.3），容易造成工作量不均等。实验过程中我们尝试了精确化搜索和块间通信进行优化。

4.7.2 细粒度实现

细粒度实现是使用类似于 **SIMD** 并行化的方法，将 GPU 视为一个**超大的向量**，一次进行比较若干个元素。

算法外层正常执行，无需并行分段（相应也不存在提前结束特性对算法效率造成的影响），当深入到函数内存循环比较部分时，调用核函数，一次比较若干个元素，加速求交过程。

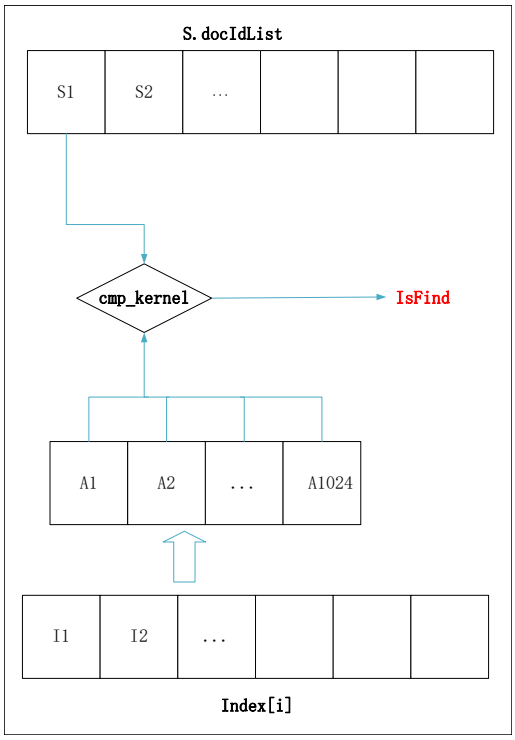

图 4.21: 细粒度算法示意图

定义核函数 cmp_kernel，负责将一个 docId 与待求交链表中 blockDim.x 个元素做比较，如果发生匹配，则将结果置为 true。因为若干个元素至多有一个元素与该 docId 相匹配，所以并不会发生冲突条件。

该方法直接在算法上运用 SIMD 的思想，不需要再考虑算法提前结束特性的影响。缺点是 host 端与 device 端**通信较多**，需要反复装载数据到 GPU，通信成本高。为此，我们尝试了使用统一虚拟内存等方法。

4.7.3 实验结果

如前所述，本次 GPU 实验共使用了两种并行手段：粗粒度和细粒度。两种并行手段加速效果对比如下：

	SVS	ADP	Hash(512)
串行时间	4379.99	3395.6	15422.8
粗粒度并行	199.27	180.74	504.01
加速比	21.98	18.78	30.60
细粒度并行	237.43	199.13	749.41
加速比	18.44	17.05	20.57

表 9: 两种并行方法对比

由表9可以看出，使用 GPU 并行加速取得了一定的优化效果，但并未达到 100 倍的理想加速比；另外粗粒度并行的加速效果整体略优于细粒度并行，在 Hash 算法上领先效果尤为突出。具体原因可能如下：

- 整个程序需要查询上千份 query，类似于 PThread 的**动态开辟线程**，查询每个 query 都需要重新对 device 端分配内存并从 host 端复制过去，造成了巨大的额外开销。依据阿姆达尔定律，算法对求交过程的加速效果需要平衡数据迁移、通信的成本，所以整体加速比并不理想。
- SVS 和 ADP 串行算法有提前结束的特性，容易造成**工作量不均等**，进而加速不理想。虽经过精确化搜索优化，但线程间**通信成本**也难免增大，只能是一定程度的效率提升。
- 细粒度实现参照了 SIMD 的思想，在算法最内层一次取出大量元素放进向量，使用核函数一次性比较。虽然没有了分段并行提前结束特性的影响，但由于核函数位于最内层，相较于分段并行，每次运行核函数还需要传输比较结果 isFind，下一次循环判断又需重新赋值通信，**通信开销成本高**。
- SVS 和 ADP 算法使用细粒度实现没有了提前结束的影响，但平衡上通信开销，两次并行加速效果基本保持一致。而 Hash 算法本身没有提前结束特性，在使用细粒度实现后，额外增加了多余的通信开销，于是**加速效果回落**到与另外两个算法一致。

4.8 oneAPI*

oneAPI 工具提供了更加灵活的共享内存机制（使用 malloc_shared API 接口），避免了 CUDA 实验中需要将倒排索引拷贝至 device 端所带来的开销，有望将 GPU 的加速比进一步提高。也正是由于这一点，我们决定额外对于 oneAPI 在倒排索引并行化中的应用进行探究。

4.8.1 并行实现

在 ADP 算法的基础上尝试将集合求交的过程使用 oneAPI 进行并行化，首先是要进行任务划分工作，利用 oneAPI 中方便的 API，我们便可以将 global_ndrange 设置为链表长度除以总的分组数量。这样便将 len/group 个子求交任务分配给了 GPU 中的一个线程。

接下来，就可以利用4.5.2节中的**精确负载思想**，让 GPU 的每个线程找出它们在 1 至 num-1 号链表的具体工作范围，便可使得每个线程的负载尽量保持均匀。

最后，我们就可以调用 ADP 算法，让每个线程分别对它们的工作项进行求交，以取得最终的整体求交结果。

由以上分析可编写 ADP 的 OneAPI 的算法代码如下：

ADP 的 oneAPI 优化

```
#define group 1024
double gpu_kernel(unsigned int* docId, int* lArr, sycl::queue& q){
    auto global_ndrange = range<1>(lArr[1]/group);//利用长度确定子任务项的大小
    auto local_ndrange = range<1>(5);//局部任务确定为5个
    auto e = q.submit([&](sycl::handler& h)
    {
        h.parallel_for<class k_name_t>(sycl::nd_range<1>(global_ndrange, local_ndrange),
            [=](sycl::nd_item<1> index)
        {
            int is = index.get_global_id() * group;//第0个链表的起始位置
            int ie = lArr[1]>is + group?is+group:lArr[1];//第0个链表的结束位置
            int rs[5], re[5];
            //利用二分搜索每个链表的求交起始位置rs[i]
            ...calculate rs, re here...
            //每个链表对自己工作范围内的元素进行求交
            ...ADP work here...
            if (isFind)
                docId[cnt++];
        });
    });
    e.wait();//等待GPU所有线程完成
    //计时得到算法执行时间
    duration += (e.get_profiling_info<info::event_profiling::command_end>() -
        e.get_profiling_info<info::event_profiling::command_start>()) / 1000.0f / 1000.0f;
    return duration;
}
```

4.8.2 实验结果

图4.22是 ADP 算法在不同 GPU 优化（CUDA 与 oneAPI）下的性能对比图。

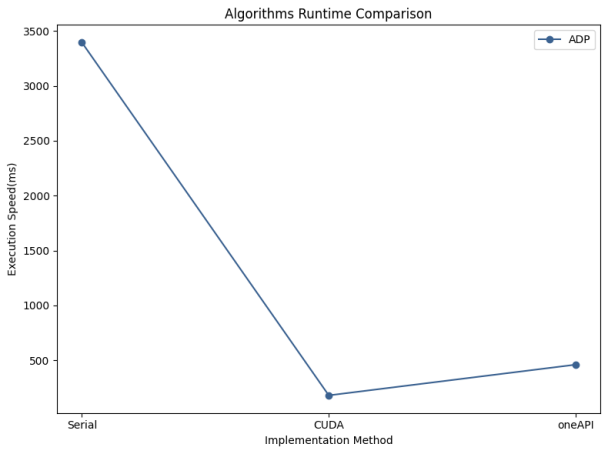

图 4.22: CUDA 实现与 oneAPI 实现对比

可以发现虽然 oneAPI 提供了共享内存机制，避免了 CUDA 从 host 端到 device 端的拷贝工作带来的开销，但是其优化程度依旧不能超过 CUDA。其原因可能有以下几点。

- 硬件适配性问题：由于测试时所使用的电脑显卡为 NVIDIA 的 GPU 而非 Intel 的 GPU，故 oneAPI 的适配性可能不如 NVIDIA 原生的 CUDA 好，这就导致了其性能的下降。
- 平台差异：CUDA 是 NVIDIA 的专有平台，而 oneAPI 旨在为多种硬件平台提供兼容的编程模型，可能会对其性能作出一定的牺牲。

5 项目总结

本项目完整地实现了**倒排索引压缩**、**解压**、**求交**的流程，并对每个步骤花费大量精力进行细节调优与并行优化，对实验过程中遇到的问题查阅资料并充分讨论，充分掌握了算法的各个细节以及丰富的并行化 API。

本文虽然力求简洁，尽量避免不必要叙述，但通篇下来仍然超过 30 页，是我们二人至今完成篇幅最大的一篇。这也是我们二人倾注大量心血的体现，整个项目从开题到结题，遇到无数困难：开题时的迷茫，算法探索的障碍，学期内繁重的课业..... 整个项目下来，我们二人也感受到巨大的进步，不论是对并行算法的掌握，编程风格的改进，多人协作开发的技巧，亦或是对项目整体规划的经验，都让我们觉得收获颇丰。

“艰难困苦，玉汝于成”，通过本学期并行程序设计课程的学习和实验的探索，我们二人掌握了一定的工程素养，并学会了如何规范的表达。在此向开设本课程的王刚老师和不厌其烦指导我们进行实验的李君龙学长等助教学长学姐表示衷心的感谢。

参考文献

[1] Vo Ngoc Anh and Alistair Moffat. Inverted index compression using word-aligned binary codes. *ACM Transactions on Information Systems (TOIS)*, 23(3):261–303, 2005.

[2] 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*, pages 743–752, 2000.

[3] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. Experiments on adaptive set intersections for text retrieval systems. In *ALENEX*, volume 1, pages 91–104. Springer, 2001.

[4] Shuai Ding, Jinru He, Hao Yan, and Torsten Suel. Using graphics processors for high performance ir query processing. In *Proceedings of the 18th international conference on World wide web*, pages 421–430, 2009.

[5] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, 1966.

[6] Frank K. Hwang and Shen Lin. Optimal merging of 2 elements with n elements. *Acta Informatica*, 1(2):145–158, 1971.

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

[8] Hiroshi Inoue, Moriyoshi Ohara, and Kenjiro Taura. Faster set intersection with simd instructions by reducing branch mispredictions. *Proceedings of the VLDB Endowment*, 8(3):293–304, 2014.

[9] ILYA KATSOV. Fast intersection of sorted lists using sse instructions, 2012. https://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/.

[10] Daniel Lemire, Owen Kaser, and Kamel Aouiche. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering*, 69(1):3–28, 2010.

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

[12] 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*, pages 825–826, 2011.

[13] Wei Zhang, Jianwen Long, and Yu Zhang. Pfordelta: Simple and efficient compressing of inverted indexes. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 107–114. ACM, 2008.

[14] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2):6, 2006.