南開大學
Nankai University

计算机学院

并行程序设计期末研究开题报告

倒排索引压缩及集合求交并行算法研究

姓名：张铭

学号：2211289

专业：计算机科学与技术

2024 年 4 月 5 日

摘要

信息检索是指从大规模数据集中检索出与用户查询相关的信息的过程。倒排索引（Inverted Index）是信息检索中常用的数据结构，用于加速查询过程。在信息检索中，当用户提交一个查询时，系统会首先将查询中的每个关键词在倒排列表中进行查找，然后将这些倒排列表进行合并，这个过程在算法问题中成为倒排列表求交（List Intersection）。为了加快查询效率，节省倒排列表的存储空间，对于算法优化和存储空间压缩，相关学者已经进行了大量研究，并取得了相当可观的成果。

本文 w 聚焦对前人部分工作的总结，并以 list-wise 和 element-wise 两种串行算法为基础，研究串行算法的优化策略，并通过 SIMD、多线程、MPI 和 GPU 等并行编程方法对算法进行并行化。小组二人明确分工，共同合作，完成实验并撰写实验报告。

关键字：倒排索引、倒排列表求交、并行优化

目录

1 引言

在当今信息时代，大规模数据的处理已成为许多领域中不可或缺的挑战之一。倒排索引作为一种重要的数据结构，广泛应用于信息检索、数据压缩和数据挖掘等领域。随着数据规模的不断增大，对倒排索引的高效压缩和快速查询需求也日益迫切。同时，集合操作如求交和求并在各种数据处理任务中占据重要地位，因此并行算法在此领域中的研究备受关注。本次期末实验中，我们将深入研究集合求交算法的串行优化和并行化设计，其中并行算法将基于 SIMD、Pthread、MPI 和 GPU 四种并行架构开展，探讨如何充分利用并行计算资源，提高求交操作的执行效率。

2 问题阐述

2.1 倒排索引

信息检索（Information Retrieval，IR）[2] 被认为是对大规模电子文本和其他人类语言数据进行表示、搜索和处理的技术。而信息检索系统和服务如今已经非常普遍，人们使用它们来进行各种商务、教育以及娱乐活动。倒排索引（Inverted Index），几乎已经是每一个信息检索系统的核心结构，常用于快速检索文档。最简单地，倒排索引提供了文档集中词项与其出现的位置之间的映射。

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。

倒排索引的相关概念可以由图 1 所示的示例解释，它给出了某篇小说文本上的一个索引。词典（Dictionary）列出了文档集的词汇表中包含的词项，每一个词项与一个表征它出现位置的位置信息列表，即倒排列表相关联。

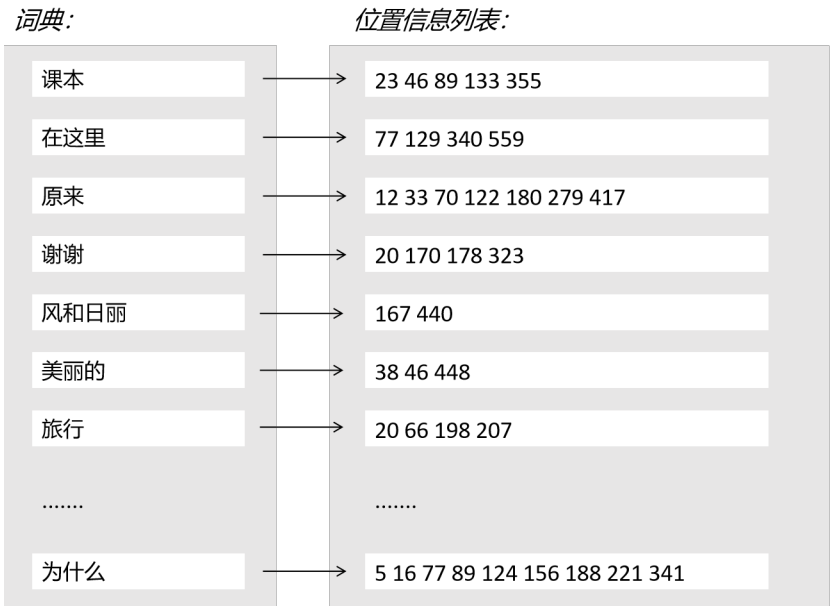

图 2.1: 某篇中文小说相关词项的倒排索引，这个词典建立了词项到其出现位置（页数）的映射

2.2 倒排列表求交

在获得倒排索引列表的基础上，我们考虑倒排列表求交（List Intersection）。它指的是将多个倒排列表（即文档或记录中某个特定项的出现位置列表，比如某个关键词在不同文档中的出现位置列表）进行交集操作，以找到它们共同出现的位置或文档。

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 $\cap_{1\leq i\leq k} l(t_i)$。

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：

$$l(2014) = (13, 16, 17, 40, 50) \tag{1}$$

$$l(NBA) = (4, 8, 11, 13, 14, 16, 17, 39, 40, 42, 50) \tag{2}$$

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

求交操作返回三个倒排列表的公共元素，即：

$$l(2014) \cap l(NBA) \cap l(Final) = (13, 16, 40, 50) \tag{4}$$

当用户输入多个关键词时，求取这些关键词的交集可以帮助提高搜索结果的相关性，因为只有包含所有关键词的文档才会被返回。因此，倒排列表求交的操作在信息检索、数据管理和数据分析等领域中具有重要的意义，可以帮助提高数据处理效率、改善搜索结果的质量以及发现数据之间的关联性。在大数据时代，对大规模数据集 (Massive Datasets) 的分析与应用已成为常态, 我们考虑以倒排列表求交的思想为基础，优化求交算法，并结合并行体系结构的相关知识，使数据处理的效率最大化。

3 前人的研究成果

3.1 求交的基本方法

倒排索引的核心思想是将文档中的每个术语（term）映射到包含该术语的文档列表。而这个概念的起源可以追溯到早期的信息检索系统，例如 20 世纪 60 年代末和 70 年代初的系统，这些系统开始尝试利用计算机技术来管理和检索大量的文档。在这些早期系统中，人们开始意识到将文档与其包含的术语之间建立关系的重要性，并尝试着构建相应的数据结构来实现这一目标。

早期的信息检索系统开始尝试使用倒排索引来加速文档检索，因为它能够有效地提供单词与文档之间的映射关系。随着信息检索领域的发展，研究者们意识到对多个倒排索引进行交集操作是非常常见且有用的需求，因此开始研究如何高效地实现倒排索引的交集操作。

最早被提出的倒排索引求交算法之一是经典的两个列表求交算法。这个算法的基本思想是遍历两个有序列表，从头到尾逐个比较它们的元素，并将相同的元素添加到交集中。

由于两个列表 S 和 T 是有序的，所以可以使用多种方法来高效地实现这一过程。假定两个有序列表分别为 S 和 T，其中 |S|=n_1，|T|=n_2，以下是在信息检索的发展中前人经过反复实践总结出来求交集的具体方法：

可以使用双指针法（Two Pointers）来同时遍历两个列表，从而减少比较的次数。具体来说，维护两个指针分别指向两个列表的起始位置，然后依次向后移动指针，比较两个指针所指向的元素，将相同的元素添加到交集中。该算法的时间复杂度为 O(n_1+n_2)。

基础的二分查找（Binary Search）适合于其中一个列表元素个数远大于另一个列表的情形。具体

来说，若 $n_1 << n_2$，对于 S 中每个元素，在 T 中进行二分查找，以 S 中的元素确定是否存在于 T 中。这种方法可以将查找的时间复杂度控制在 $O(n_1 log n_2)$ 的水平。

Baeza-Yate 算法 [1] 在此基础上使用分治策略，这种情况适于两个列表等长或相似的情况。首先在 S 中二分查找 T 的中间元素，不管查找是否成功，S 和 T 都被分割为左右两个部分，分割点分别是 S 中查找返回的位置和 T 的中间元素。这时 S 和 T 之间的求交问题，就转变为两者左右集合分别求交的子问题。进而可以通过递归方式继续处理，直到子问题不能再分割。

3.2 FOR 压缩技术

当文档总数 U 达到很大的规模时，需要开辟很大的存储空间来存储倒排列表的 DocID，为了节省存储空间和提高检索效率，常常需要对倒排列表进行压缩。部分学者提出 FOR（Frame Of Reference）压缩技术来解决这个问题，其核心是利用增量编码（Delta-encode）的方式来压缩整数的存储空间，提高空间利用率并提高检索效率。通常 FOR 技术会经历增量编码（Delta-encode）、分割成块（Split into blocks）和按需分配空间（Bit packing）这三个步骤。

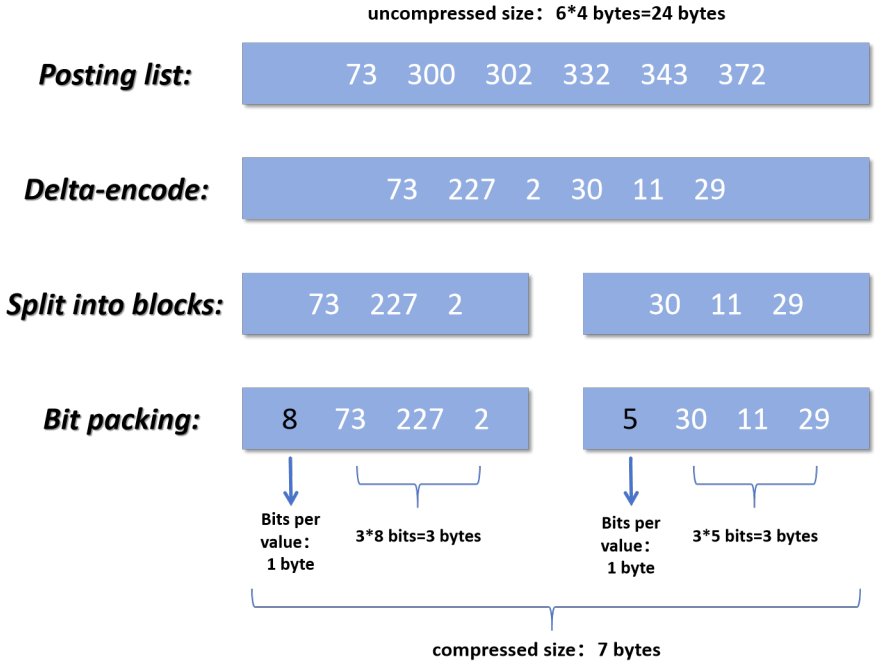

图 3.2: FOR（Frame Of Reference）压缩技术的步骤和流程

3.3 位图

上述方法均是在列表以数组或链表存储文档编号（DocID）的基础上设计的，这种简单的存储形式易于实现，但是在进行求交运算时缺乏灵活性，每个 term 所映射的倒排列表数组长度通常不是相同的，使得当一次搜索中 term 的值较多时求交运算变得较为复杂，不利于并行化算法的设计。由此，部分学者引入了位图（Bitmap）的概念。

位图是一种数据结构，用于表示一组二进制位的集合。在位图中，每个位都只能是 0 或 1，分别表示集合中的元素是否存在或者是否被标记。使用位图来存储倒排列表时，位图的长度通常等于文档总数，每个位表示一个 DocID 是否包含在该 term 对应的倒排列表中。假定文档总数 U=10，一次搜索中的词组为 “the boy first”。我们通过以下示例来展现位运算操作实现倒排列表求交的过程。

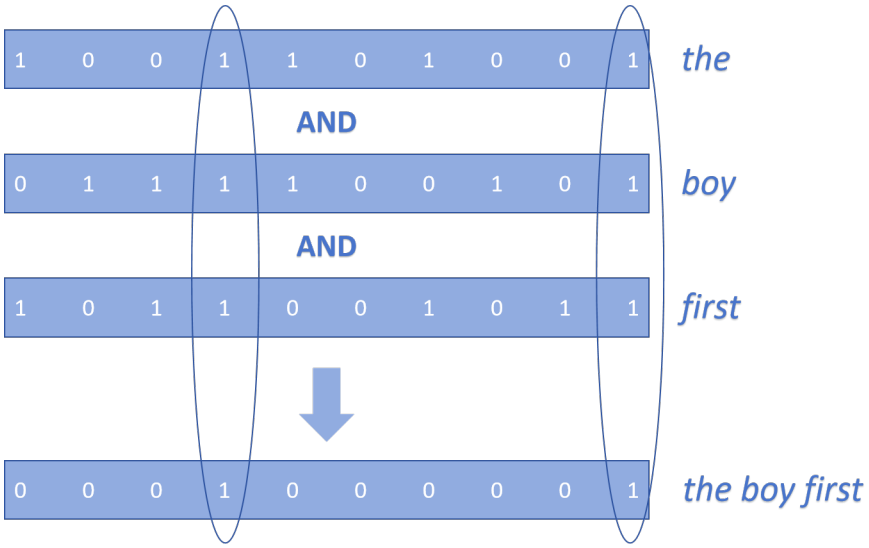

图 3.3: 位图存储的倒排列表通过 AND 位运算实现求交

不难发现，通过位运算的方式能够高效地完成倒排列表求交的操作，并且对这种存储方式易于进行并行体系结构的优化。

然而，位图也存在一些限制，例如对于非常大的文档集合 U，位图可能需要较大的存储空间。同时，对于需要频繁更新的动态倒排列表，其更新操作相对复杂。如插入新的 DocID 或删除现有的 DocID 时，因为更新可能需要对所有 term 的位图进行操作。此外，位图通常适用于密集的数据集合，即 DocID 的分布相对均匀且紧凑的情况下。如果 DocID 的分布非常稀疏或者不均匀时，可能会导致位图的空间利用率降低，从而影响查询操作的效率。

3.4 RBM 压缩方法

为了解决 bitmaps 压缩性较差的问题，许多研究都并取得了令人印象深刻的结果，但部分算法导致的代价是集合操作的性能不佳。其中一个重要的研究成果是由 Daniel Lemire 和 Owen Kase 提出的 RBM（Roaring Bitmap）方法 [3]，其通过哈希表（Hash Table）以及哈希函数建立倒排列表的二级索引（Secondary Index），旨在提供一种高效的方法来压缩由 bitmap 导致的大规模的整数集合，并且在实际应用中表现出了出色的性能和压缩比。

该策略采用的哈希函数定义为：$h(x) = \lfloor x/2^{k-p} \rfloor$。其中 k 为满足文档总数 $U \leq 2^k$ 的最小正整数，p 表示在同一个二级索引下的文档 DocID 的二进制数只有最低 p 位不同。这个哈希函数的本质，就是根据 DocID 的高 (k-p) 位划分倒排列表，实质是为整个倒排列表建立二级索引，划分后相同二级索引内的元素仍按升序排列。

假定文档总数 U=50000，此时 $2^{15} < U < 2^{16}$，即选取 k=16，设定 p=8，即建立 2^8 个二级索引，倒排索引的 8 个最高有效位（8 most significant bits）在二进制数 00000000~11111111 范围内变化。

如上表所示，每个二级索引下的 DocID 共享相同的 8 个最高有效位，Roaring Bitmaps 使用的划分方案可以确保一个 DocID 始终属于 2^8(或 256) 个连续整数所在的某个二级索引。由此，我们再根据某个 term 所建立的倒排列表中 DocID 的分布特征来设定每一个二级索引下 DocID 的存储方式。规定某一倒排列表的 DocID 总数为 N，第 i 个二级索引的 DocID 数量为 $N_i (1 \leq i \leq 2^8)$。当 $N_i \leq 2^6 = 64$ 时，称其为稀疏子列，其中的 DocID 以数组形式存储；当 $N_i > 2^6 = 64$ 时，称其为稠密子列，其中的 DocID 以位图形式存储。当某个子列表 DocID 较为稀疏时，将其转为普通数组或链表存储，这样可以大大减少由 bitmaps 导致的空间浪费，同时提高空间利用率。

8 most significant bits	Range Start(Decimal)	Range End(Decimal)
00000000	0	255
00000001	256	511
00000010	512	767
00000011	768	1023
...	...	...
11111111	65280	65535

表 1: 将 16 位整数划分为连续的二级索引

8 most significant bits	the number of DocID	storage type	storage
00000001	30	Array	(1,5,26,28,33,...,230,251)
00000010	128	Bitmap	(1,0,1,0,1,0,1,0,...,1,0)

表 2: 某个二级索引下的子列表存储方式

4 研究方案

4.1 开展研究的准备工作

4.1.1 并行架构的选择方案

基于并行课程设计规划，本学期将会接触到 SIMD，Pthread，MPI 以及 GPU 等多个并行优化架构，因此本次期末实验的研究将基于这几种并行编程工具开展，设计实现并行求交算法，在数据集上进行对比实验，并在算法实现的过程中设定性能评价指标（如 cache miss 次数，吞吐率等），根据性能评价指标给出算法的优化力度，对比分析总结其优劣势，在此基础上不断优化改进，提高并行优化力度。

4.1.2 实验平台的选择方案

基于对不同优化力度的对比分析以及课程的进阶要求，本次期末实验将同时在本机 x86 平台和 arm 平台（华为鲲鹏服务器）上进行算法实现与实验探究。我们将综合考虑平台的内存大小和各级缓存大小，由此来分析实验结果。根据得出的性能评价指标对比分析 x86 平台和 arm 平台对并行求交算法的性能表现差异。考虑到 x86 平台与 arm 平台之间的硬件差异，对比分析并不能做到绝对公平，得出的结论也就存在着误差。因此我们将考虑研究在不同平台下，部分性能评价指标随着问题规模等参数的变化趋势，并尝试分析其中的原因。

4.1.3 数据集的选择方案

本次期末实验给定的数据集是一个截取自 GOV2 数据集的子集，共含有两个文件：

- ExpIndex：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]....
- ExpQuery：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）

4.1.4 性能评价指标的选择方案

对于搜索引擎请求处理问题，主要考虑以下两个评价指标：

1. 响应延迟——单个 Query 处理所需时间
2. 吞吐率——每秒 Query 处理量

在此基础上我们还将通过并行加速比、效率函数等方式来量化并行效率，开展并行算法与串行算法的对比分析。

4.2 算法的求解与优化

4.2.1 基于串行算法的优化

基于对 list-wise 和 element-wise 两种思路的串行算法进行实现，在此基础上进行改进与优化。可行的方案包括：

1. 通过 cache 优化算法对倒排列表进行高速访存，减少 cache 未命中次数
2. 基于 list-wise 算法，优先将最短的两个表进行求交，得到中间结果，继续迭代，减少比较次数
3. 基于 element-wise 算法, 考虑建立一个散列表，记录每个文档编号在倒排列表中出现的位置，从而快速确定是否存在相同的文档编号，减少比较次数
4. 实现指令级并行（相邻指令无依赖）以利用超标量架构

4.2.2 基于 SIMD 的并行算法求解

考虑到升序列表求交操作本身不是标准的单指令流多数据流模式，很难直接进行向量化。因此，除了对倒排链表的存储格式及其上的串行求交算法直接进行 SIMD 并行化之外，我们将考虑使用位图存储方式——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。从而将求交运算转化为两个位向量的位与运算。

此外，考虑到倒排链表的稀疏性，我们将在此基础上建立二级索引——将位向量分块，每个块用一个 bit 的二级位向量表示其全 0（0）还是非全 0（1），只有二级位向量位与为 1 才进行底层位向量对应块的位与运算，这样做的好处是能够减少存储空间和串行计算时间，达到更好的并行效果。

4.2.3 基于 Pthread 的并行算法求解

多线程并行算法考虑以下并行两种方式：

- Query 间并行：不同线程处理不同 Query
- Query 内并行：将一个 Query 的处理进行任务划分。例如：基于 element-wise 算法，将扫描的链表进行划分，每个线程承担扫描一部分文档，最后再将结果汇总。

4.2.4 基于 MPI 的并行算法求解

MPI 算法实现过程与 Pthread 比较类似，但仍有以下几点需要注意：MPI 是一种分布式内存编程模型，由于需要在不同的节点之间传递消息，涉及网络通信和数据序列化等操作，相较于 Pthread，MPI 的通信开销较高，因此在设计 Query 内并行算法时，需要尽可能减少通信。可行的方法包括：

- 合理的任务划分和局部计算，尽量减少节点间需要交换的数据量
- 在每个节点上对局部数据进行聚合操作，减少全局通信的需求
- 在任务划分时考虑节点的负载情况，尽量使各节点的计算负载均衡

4.2.5 基于 GPU 的并行算法求解

线程块内参考 SIMD 设计思路，近似 SIMD 并行。线程块间参考多线程设计思路，近似多线程并行。同时考虑 GPU 特殊的架构特点进行调整。

5 分工明细

本次期末实验由张高和张铭同学合作开展，具体分工如下：

- 张铭同学主要负责基于并行课程设计的规划与要求，完成期末研究的准备工作并制定初步的算法设计方案; 基于 list-wise 思路实现串行算法优化和并行算法求解并在此基础上进行改进；
- 张高同学主要负责查阅与倒排索引求交以及索引压缩的相关文献和资料并进行提炼总结；基于 element-wise 思路实现串行算法优化和并行算法求解并在此基础上进行改进；
- 共同完成的任务：汇总实验数据，绘制图表等，分析得出结论并撰写期末实验报告。

参考文献

[1] Ricardo Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Combinatorial Pattern Matching: 15th Annual Symposium, CPM 2004, Istanbul, Turkey, July 5-7, 2004. Proceedings 15*, pages 400–408. Springer, 2004.

[2] Stefan Buttcher, Charles LA Clarke, and Gordon V Cormack. *Information retrieval: Implementing and evaluating search engines*. Mit Press, 2016.

[3] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience*, 48(4):867–895, 2018.