南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计——期末研究报告

倒排索引求交算法的并行优化

林雨豪 2012516

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

摘要

搜索引擎拥有庞大的数据集，为提高检索效率，需要建立由关键词到文档的映射，称为倒排索引。为保证搜索结果的质量，需要先筛选包含所有查找关键词的文档，该筛选过程实质上为倒排索引求交。倒排索引求交的关键词多，列表数量大；包含该关键词的文件多，列表中的元素数量大；求交查询需求多，对算法的吞吐率要求高，搜索引擎追求尽量短的响应延迟，需要提高单次求交的效率。本文对两种倒排索引求交算法：按列表求交和按元素求交做出分析并实现，对串行算法做出改进和优化，使用 Phtread、OpenMP 等并行方法实现程序并行的性能提高，额外实现了位向量法使程序具备 SIMD 的可能性并做出测试和分析。在不断的改进、测试、分析中加深对并行程序设计课程内容的理解和实际运用能力。

关键字：倒排索引求交、并行优化、Pthread、OpenMP、SIMD

目录

一、 引言

在搜索引擎等应用场景中，有根据输入关键词，检索数据集中包含该关键词的文档或网页的需求。对于一个庞大的文档或网页数据集，全文遍历的代价极大。为提高检索效率，需要建立文档或网页与其包含的关键词的映射。对于数据集中的每一个文档，经过特殊的文字提取、分词和去重后得到该文档包含的关键词集合，这种由文档到关键词的映射即为正排索引。

正排索引建立每个文档与其关键词的关系，但若需要根据关键词搜索对应文档，仍然需要遍历所有文档的关键词集合。搜索引擎会将正向索引重构，对于所有出现过的关键词，每个关键词保存包含该关键词的文档集合，这种由关键词到文档的映射称为倒排索引。

在搜索引擎中，需要根据用户输入处理提取得到多个关键词，根据一定的排序算法返回相关度最高的 k 个文档。一般情况下，包含所有关键词的文档更有可能是用户所需求的，每个关键词对应的文档集合的交集即为包含全部关键词的文档集合。所以将以倒排索引求交的方式作为对搜索结果排序的先导处理。

二、 倒排索引求交算法简述

(一) 倒排索引求交数学描述

此处忽略搜索引擎对检索结果的内容相关度排序，忽略关键词在文档中重复次数等因素，将问题抽象如下：

在数据集中共有 N 的文档（Document），为每个文档 d 设置一个唯一且固定的序列号 $i \in [1:N]$，每个文档 d 都看做是一组词（Term）的序列，获得文档到词的映射。对该映射进行处理，为每个出现过的词 t 建立与之对应的文档集合 $l(t)$，保存出现过该词的所有文档，并将该集合按照文档的序列号升序排列形成倒排列表。所有词的倒排列表集合即构成了数据集的倒排索引。

检索过程中，给定 k 个查询词 $t_1, \ldots, t_k$，进行倒排索引求交算法后返回 $\bigcap_{1 \le i \le k} l(t_i)$

(二) 倒排索引求交串行算法

倒排索引求交的主要算法有按表求交和按元素求交两种算法。

1. 按表求交法

对于 n 个关键词对应的列表 $l(t_1) \ldots l(t_n)$（关键词经过排序，满足 $|l(t_1)| \le \ldots \le |l(t_n)|$），令 $S = l(t_1)$。遍历列表 $l(t_2) \ldots l(t_n)$，对于每个 S 中的元素 d，如果在列表 $l(t_i)$ 未检索到 d，则将 d 从 S 中删去，最终 S 即为倒排索引求交结果。

该算法的伪代码表述如下：

Algorithm 1 按列表求交算法

Input: n 个关键词对应的列表 $l(t_1) \ldots l(t_n)$

Output: $\bigcap_{1 \le i \le n} l(t_i)$

```
1: S ← l(t_1)
2: for i = 2 to n do
3:     for each element e ∈ S do
4:         if find(e, l(t_i)) == false then
5:             Delete e from S
```

```
        end if
    end for
end for
return S
```

每一轮求交后 S 中元素变少，在之后的求交过程中计算量随之减少，求交效率加快。当列表中的元素很多但交集很较少时，该方法的效率较高。

2. 按元素求交法

对于 n 个关键词对应的升序倒排列表 $l(t_1)\dots l(t_n)$（关键词经过排序，满足 $|l(t_1)| \leq \dots \leq |l(t_n)|$）；列表文档也经过升序排列，满足 $l(t_i) = \{d_1 < \dots < d_k\}$），令 $S = \emptyset$，每次循环前将所有列表按照未访问的元素个数排列，第 1 个列表中的首个未访问元素记为 d，遍历剩下第 2-n 个列表，判断第 i 个列表中是否含有元素 d，并当前列表中小于等于 d 的元素全部置为已访问。如果第 i 个列表中不含有元素 d，将剩余列表中小于等于 d 的元素置为已访问（可选），继续下一轮循环；如果每个列表中都存在元素 d，则将 d 加入集合 S。当循环前的最短列表长度为 0 时，停止循环，S 为倒排索引求交的结果。

该算法的伪代码表述如下：

Algorithm 2 按元素求交算法

Input: n 个关键词对应的升序链表 $l(t_1)\dots l(t_n)$

Output: $\bigcap_{1\leq i\leq n} l(t_i)$

```
S ← ∅
while no completely visited list do
    l_1 = the first list in the set
    e = l_1.first_undetected_element
    for i = 2 to n do
        if find(e, l_i) == false then
            break
        end if
        if i == n then
            S.add(e)
        end if
    end for
    mark e' < e detected
    sort_lists_by_length
end while
return S
```

各个列表以链表的形式存储，使用迭代器对每个链表最小的元素开始访问，同时完成元素已访问标记和元素查找，优化了在列表中查找元素是否存在的效率。当一个链表走到尽头时完成求交运算，缩短了对链表进行扫描的过程。当某个列表中的元素个数较少时，该方法效率较高。

三、 算法分析

该算法的输入有：

- 数据集中的文档总数: D;
- 倒排索引列表个数: M;
- 输入关键词的数量: n;
- 倒排列表的最大长度: L（由于 $L \leq D$，可以用 D 替代, 且 L 与输入的关键词有关。但在实际应用中有 $L \ll D$，故还是加以区分）

我们的算法分析假设为每次只做一次查询。若为多次查询，时间复杂度在原有的基础上乘以查询次数。

(一) 按列表求交算法

每次查询，需要先找到所有关键词对应的列表，遍历所有的列表，时间复杂度为 $O(M)$。接下来是对所有得到的列表做求交集的操作。需要取出第一个列表作为初始的结果集合 S，然后遍历其余的 $n-1$ 个列表。对于每一个列表的遍历，需要查找 S 中的元素是否在该列表内，每一次查询的时间复杂度为 $O(L)$，如果未能查询到该元素，在集合 S 中删除该元素的时间复杂度为 $O(L)$。每遍历一个列表的时间复杂度为 $O(L^2)$, 列表共有 $n-1$ 个, 时间复杂度为 $O[(n-1)*L^2]$, 加之寻找列表的过程，该算法的总时间复杂度为:

$$O(M + n * L^2)$$

倒排索引列表是给定的数据集，不确定的只是每次查询的输入，即所有的倒排列表应当提前存储在数据库中，我们在进行单次查询的时候，取出需要的倒排列表存储在内存中，此步骤的空间复杂度为 $O(L*n)$。可以直接将第一个列表当做 S 使用，即创建结果集合 S 的过程不消耗额外的空间其余的查找、删除操作也不涉及额外的空间使用，该算法的总空间复杂度为:

$$O(L*n)$$

(二) 按元素求交算法

每次查询，也需要先找到所有关键词对应的列表，时间复杂度为 $O(M)$。进入迭代过程，每次迭代需要将所有的列表按照长度升序排列，无需改变这些表的物理内存位置，只需要改变访问顺序即可。列表共 n 个，时间复杂度为 $O(n\log n)$。每次迭代中的时间复杂度及迭代次数不能分别计算，我们按照链表的方式访问倒排列表，且列表经过升序排序，故每个列表中的元素在整个迭代过程中只会被访问到一次。查询操作和添加操作的总时间复杂度为 $O(L*n)$。迭代的次数不超过列表长度 L 次，排序过程的总时间复杂度为 $O(L*n\log n)$。该算法的总时间复杂度为:

$$O(M + L*n\log n)$$

取出需要的倒排列表存储在内存中，此步骤的空间复杂度为 $O(L*n)$。此外需要额外开辟一个数组用于储存结果集合 S，其他操作无额外的空间消耗。该算法的总空间复杂度为:

$$O(L*n)$$

(三) 比较和分析

1. 两种方法的时间复杂度和空间复杂度都在 $O(n^3)$ 数量级以内，有实际的可行性。

2. 按列表求交算法的时间复杂度大于按元素求交的算法。但是时间复杂度只是对算法粗略的描述，按列表求交算法每次迭代后结果集合 S 中的元素个数都会大幅降低，但时间复杂度分析时只能取后结果集合 S 中的元素个数为其上限 L，具体性能要取决于具体数据。

3. 对于按元素求交算法，从时间复杂度分析来看，不经过排序也能得到正确的结果，且时间复杂度为 $O(L*n)$。但从实际角度出发，一般输入的关键词个数 n 非常小，$O(L*n\log n)$ 与 $O(L*n)$ 差异极小。且排序的作用是降低迭代的次数，排序方法的最终迭代次数会小于未排序的方法。

四、 测试与优化

(一) 串行算法性能测试

按照伪代码描述实现了按列表求交和按元素求交两种算法，根据实验提供的数据集测试算法性能如表 1 所示：

	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量
按元素求交	1000	2816.76	2.816	355.1
按列表求交	1000	740558.25	740.558	1.3

表 1: 串行算法性能测试

按列表求交的性能明显差于按元素求交。数据集中每个倒排列表的长度最大可达到 30000，按列表求交会涉及大量的列表查找、删除操作，算法的效率很低。经观察，每个查询请求的关键词个数为 2-5，按列表求交算法的优势在于每次迭代后大幅缩减结果集合 S 中的元素个数，但实际每个请求中需要求交的次数较少，难以体现该算法的优势。

(二) 串行算法性能优化

1. 按列表求交

在应用场景中，对数据集的排序操作可以在处理请求之外进行的。经检查，数据集中的每个倒排列表已经按照升序排列完毕。在按列表求交中，应当充分利用数据集各列表已经完成排序的特征，将查找操作优化为二分查找的方式。每一次查询的时间复杂度为 $O(\log L)$，每遍历一个列表的时间复杂度为 $O(L*\log L)$，该算法的时间复杂度可以优化为 $O(M+n*L\log L)$

借鉴按元素求交中链表的访问查找思想，可以继续优化按列表求交的查找方式。对于每个列表，每次查询记录当前访问的位置，每次查询从上一次查询的位置开始，只需要遍历一次列表即可完成对该列表的所有查询操作。遍历一个列表的时间复杂度优化为 $O(L)$，算法的总时间复杂度可以优化为 $O(M+n*L)$。

该算法的伪代码表述如下：

Algorithm 3 按列表求交算法

Input: n 个关键词对应的列表 $l(t_1)\dots l(t_n)$

Output: $\bigcap_{1\le i\le n} l(t_i)$

1: $S \leftarrow l(t_1)$
2: **for** $i = 2$ to n **do**
3: $\quad detected \leftarrow 0$

```
    for each element e ∈ S do
        while dected < length of l(t_i) do
            if e == l(t_i)[dected] then
                found e
            end if
            if e < l(t_i)[dected] then
                not found e
                break
            end if
            detected++
        end while
        if not found then
            Delete e from S
        end if
    end for
end for
return S
```

实现以上改进程序，测试结果如下：

	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量
按列表（原版）	1000	740558.3	740.6	1.3
按列表（查找优化）	1000	26757.4	26.75	37.4
按元素（原版）	1000	2816.76	2.816	355.1

表 2: 按列表求交 查找方式优化

按列表求交的查找方式经过改进后，效率相较于原版有大幅的提升，但是仍然逊于按元素求交的方式。按列表求交的查找操作时间被大幅优化，但是当查询到列表中的元素不存在时，每次从结果集合 S 删除操作的时间复杂度仍为 O(L)，如果将列表以链表的形式存储，使用迭代器进行访问将可以近一步优化程序的性能。

2. 按元素求交

按元素求交中，需要依次访问不同列表一查询元素是否包含在所有列表中，在不同列表来回访问让我们注意到了 Cache 的问题。我们原本的实现方法是，使用 vector<unsigned int> 重新生成多个倒排列表的复制。我们希望多个列表之间尽量在内存中处于连续的位置，减少 cache 命中率低造成的性能下降。已知 C++ 中 vector 的元素连续排列，打印每个 vector 的首元素地址下标观察每个列表在内存中的位置（以请求 1 为例）：

	内存地址	列表元素个数
列表 1	23D967E7E80	268
列表 2	23D983E5B00	30000
列表 3	23D98403020	30000

可以看到，元素个数同为 30000 两个列表之间头元素之间相差 120,096 个字节，与 30000 个

整型变量的内存消耗相符，而列表 1 由于与其他两个列表长度差异较大，在动态分配内存时分配到了不同的内存块中，内存位置明显与其他两个不同。观察数据后发现，列表的最大元素个数为 30000，直接声明固定行列的二维数组可以保证每个列表之间的内存位置连续。

将创建临时列表的方式由 vector 拷贝改为二维数组依次赋值，测试结果如下：

	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量
按元素（原版）	1000	2816.76	2.816	355.1
按元素（Cache 改进）	1000	2962.44	2.962	337.6

表 3: 按元素求交 Cache 改进

实际上经过 cache 优化后的程序性能反而不如原版，可能是 vector 拷贝的效率要高于数组的依次访问、赋值。改变优化的角度，直接取消临时列表的创建、拷贝，改为添加 index 下标与列表排序结果的映射，直接访问 index 列表，牺牲 Cache 命中率以节省创建列表的时间，测试结果如下：

	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量
按元素（原版）	1000	2816.76	2.816	355.1
按元素（取消拷贝）	1000	2690.79	2.691	371.6

表 4: 按元素求交 列表访问改进

(三) Pthread 并行优化

我们在对该算法的性能进行评价时一般有两个方面：处理单条请求的平均时间和每秒可处理的请求数量。上面对串行算法的优化集中于提高单次查询的效率，Pthread 的优化也将从提高单次查询效率和多个请求的并行处理两个思路进行。

1. Pthread 请求分批处理

第一种并行的方式是将所有的请求做分批处理，每个线程完成一批请求，每个请求的计算过程独立。该方法只需每个线程完成属于自己的请求后进行一次同步，额外的通信开销小，编程方便。采用循环划分的方式安排每个线程的任务：计线程数为 n, 第 i 号线程负责编号为 $i + k * n$ 的所有请求。循环划分一定程度上保证了请求复杂度的均匀，在写入结果时不再使用 push_back 尾插，提前初始化结果的向量集合并通过下标访问进行修改储存。

对两种算法进行请求的多线程循环任务划分，使用不同数量的线程，测试结果如下：

线程数	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量	加速比
串行（按元素）	1000	2690.79	2.69	371.6	-
2	1000	1389.95	1.39	719.5	193.59%
4	1000	897.835	0.90	1113.8	299.70%
8	1000	560.79	0.56	1783.2	479.82%
10	1000	549.704	0.55	1819.2	489.50%
12	1000	494.036	0.49	2024.1	544.65%
14	1000	508.673	0.51	1965.9	528.98%
16	1000	516.233	0.52	1937.1	521.24%

表 5: 按元素求交 不同线程数量下的性能测试

线程数	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量	加速比
串行（按列表）	1000	26757.43	26.75	37.4	-
2	1000	17054.0	17.054	58.6	156.90%
3	1000	15952.1	15.9521	62.7	167.74%
4	1000	16830.1	16.8301	59.4	158.99%
6	1000	22615.8	22.6158	44.2	118.31%
8	1000	29113.40	29.1134	34.3	91.91%
10	1000	34556.1	34.5561	28.9	77.43%

表 6: 按列表求交 不同线程数量下的性能测试

对于按列表求交，多线程优化的效果较好. 最高在线程数设置为 12 时得到 540% 左右的加速效果。对于按元素求交，多线程在线程数小于 8 时有一定的加速效果，但是较为微弱，加速不超过 200%，且在线程数增加时性能反而大幅下降甚至不如原算法。

在反复测试后发现，删除按列表求交中 erase() 函数语句后（从列表中删除求交不存在的元素），程序运行大幅加快，且多线程算法开始出现明显的加速效果。可能该语句时影响多线程性能的原因之一。而且 vector 的删除时间复杂度较高，我们考虑去除删除操作，改用与 vector 等长的 bool 数组标记某个元素是否从列表中删除，每次遍历一个列表的“删除”操作总时间复杂度降低为 $O(L)$，并再次测试改进后的串行算法与多线程算法，结果如下：

线程数	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量	加速比
串行	1000	1811.37	1.81	552.1	-
2	1000	963.183	0.96	1038.2	188.06%
4	1000	535.805	0.54	1866.4	338.07%
8	1000	408.69	0.41	2446.8	443.21%
10	1000	344.496	0.34	2902.8	525.80%
12	1000	352.267	0.35	2838.8	514.20%
14	1000	338.269	0.34	2956.2	535.48%
16	1000	359.071	0.36	2785.0	504.46%

表 7: 按列表求交（改进删除操作）不同线程数量下的性能测试

改进后的串行算法相较于原串行算法有十倍以上的性能提升，甚至优于按元素求交的串行算法。多线程加速效果较好，在 12 线程时拥有最高的加速能力，与按元素求交的多线程优化能力相近。

2. Pthread 加速单次求交

第二种优化的思路是加速单次查询的效率。按列表求交的操作中，不适合做多个列表同时求交的多线程优化，违背了列表求交算法通过减少结果集合 S 中的元素个数，加快后续求交计算的初衷，而且关键词个数为 2-5，需要求交的次数为 1-4，这样的分配方式可能会造成线程的闲置。多个线程将并行处理两个列表求交的部分，通过循环划分的方式，每个线程探查结果集合 S 中元素下标为 $i+k*n$ 的列表是否存在，原本用于标记是否存在的 mark 数组改为全局变量，每个线程完成后无需同步，直接开始与下一个列表的求交（每个线程负责的下标是不变的，不涉及删除操作所以结果集合 S 中的元素下标不会变化），完成所有列表的求交后再进行一次同步。每次请求完成后，由 0 号线程负责求交结果的写入。测试结果如下：

线程数	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量	加速比
串行	1000	1811.37	1.81	552.1	-
2	1000	1477.88	1.48	676.6	122.57%
4	1000	1394.43	1.39	717.1	129.90%
6	1000	1501.86	1.50	665.8	120.61%
8	1000	1500.62	1.50	666.4	120.71%
12	1000	2440.27	2.44	409.8	74.23%
16	1000	4014.12	4.01	249.1	45.12%

表 8: 按列表求交 单次求交多线程并行

使用 pthread 加速单次查询的操作，在线程数较少时有部分性能的提升，但是相较于分批处理效果较差。分批处理只需在所有线程完成自己的全部计算时同步一次，而单次查询加速需要在每一次请求完成后同步一次，并且同步后所有线程必须等待 0 号线程进行结果的写入（结果写入期间不能进入下一个查询请求，否则会改变 mark 数组导致写入数据错误）。原本只需遍历一次列表就可以完成结果集合 S 中的元素是否在列表中的检查，多线程改进后有 n 个线程就会遍历列表 n 次，这也可能是单次查询改进的性能提升结果较差的原因。

对于按元素求交，不适于使用 pthread 进行单次查询效率改进。假定想要实现并行的部分为：通过循环划分的方式，使用多个线程共同查询某元素是否存在于列表中，实现时遇到的困难如下：

- 完成一个元素的求交后，需要对所有列表进行排序，列表访问次序改变。不同线程访问到列表的不同位置，排序结果可能不同，依赖一个线程对排序结果进行“广播”；
- 循环划分的方式分配每个线程查询的位置，如果该元素存在，也只会有一个线程获取到查询结果，其他线程需要获取到是否有线程成功找到该元素来决定是否继续遍历其他列表寻找该元素；
- 一个线程遍历列表一次就可以得到所有元素是否在该列表中，和 n 个线程遍历列表 n 次得到结果，本质上没有起到加速的作用，反而增加了线程同步的开销。

(四) openMP 并行优化

在 Pthread 测试中我们遇到了如下的问题：即使使用循环划分进行多线程的任务分批处理，各个线程的运行时间仍然有很大差异（见下表），且每次执行程序，用时最长和用时最短的线程并不确定，这可能不是任务分配不均匀而是程序执行时产生的误差。使用 Pthread 进行调度，使提前结束的线程去处理未完成的线程难度较大，考虑使用 openMP 实现更灵活的任务划分。

线程编号	执行时间
0	483.320
1	553.607
2	507.979
3	493.079
4	552.448
5	538.594
6	555.517
7	502.033
程序总用时	556.334
线程平均时间：	523.322

表 9: 按列表求交 8 线程分批处理 各线程执行时间（单次实验结果）

在依次处理请求的循环前添加编译指令 *pragma omp parrallel for ...*，在并行循环结束后添加 *pragma omp barrier*，即可完成并行区块的指定与同步。下面验证不同的任务划分与调度方式的优化性能（经过实验，在本地机器上若不使用其他预编译指令，以 *pragma omp parrallel for ...* 作为分批处理的任务划分编译指令，开启的线程数量为 12 个）

测试的编译指令有：

- *pragma omp parrallel for schedule*(*static*, 1)static 静态任务块划分，每块划分 1 个任务，与我们 pthread 中采用的循环划分方式等价；
- *pragma omp parrallel for schedule*(*dynamic*, 5) 队列式的任务划分，每块划分 5 个任务，一定程度上避免了线程执行完毕后等待的问题，任务划分较细，每个线程可能需要多次“领取任务”；
- *pragma omp parrallel for schedule*(*guided*, 20)guided 任务划分，一开始块的大小比较大，随着剩余工作量的减小，块的大小也随之变小，避免了线程执行完毕后等待的问题，减少了任务分配的次数。

划分方式	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量	加速比
串行	1000	1811.37	1.81	552.1	-
static	1000	338.763	0.34	2951.9	534.70%
dynamic	1000	310.132	0.31	3224.4	584.06%
guided	1000	326.791	0.33	3060.1	554.29%

表 10: 按列表求交 openMP 不同任务划分方式对比

划分方式	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量	加速比
串行	1000	2690.79	2.69	371.6	-
static	1000	481.259	0.48	2077.9	559.11%
dynamic	1000	449.373	0.45	2225.3	598.79%
guided	1000	478.902	0.48	2088.1	561.87%

表 11: 按元素求交 openMP 不同任务划分方式对比

动态队列式的划分明显优于静态的循环划分，相较于 Pthread，openMP 的任务划分更为灵活且简便。

（五） 位向量求交法与 SIMD

以上优化方法中均没有出现使用 SIMD 的方法。原因是计算大多数为数组遍历查找、删除，没有明显的向量化结构，也几乎不涉及基础运算，没有进行 SIMD 的空间。下面提出新的存储、计算方式以适应 SIMD 的使用。

经过测试，倒排列表中出现的最大 DocID 为 25205174。每个列表可以用 25205174 位构成的向量进行表示。该列表中存在的文档则其 ID 对应的位值为 1，否则为 0。按照此存储方法，两个向量之间求交的运算就变为了两个向量做按位与运算，将原本的求交操作转化为了向量之间的位运算，可以使用 SIMD 进行加速。

1. 可行性分析

经过测试，倒排列表共 1755 个，每个列表都按照位向量的方式存储，仅列表的空间开销为 5.14G，显然无法一次性存储在内存中。只能先按照原方式存储，在计算时将对应的列表转换为位向量。

每个倒排列表中，最大的文档数目为 30000 个，远小于最大的 DocID 值。故得到的位向量是非常稀疏的。可以考虑建立二级索引以确定某个区域内是否有文档。为了方便存储，我们选择 unsigned int 或者 unsigned long long int 作为存储单元。unsigned int 可存储 32 位，每个列表需要开辟大小为 787662 的 unsigned int 数组，为该数组建立二级索引，某一位为 0 代表其对应的 unsigned int 值为 0，为 1 则代表对应的 unsigned int 值不为 0（unsigned int 内存在 1），二级索引如用 unsigned int 表示，二级索引共需开辟大小为 24615 的 unsigned int 数组。如果使用 unsigned long long int 作为存储单元，列表数组大小为 393831，二级索引数组大小为 6154。两种方法没有优劣之分，区别主要在于二级索引对应的位数多少（unsigned int 二级索引每位对应 32 位，unsigned long long int 二级索引每位对应 64 位），需要在实际运行时确定二者差异。

每个列表转换为位向量的时间复杂度为 O(D)，两个向量按位与计算的时间复杂度为 O(D)（D 为文档最大 ID 值），单次请求中有 n 个关键词，算法的总时间复杂度为 O(n*D)。

2. 串行算法实现

求交运算函数被分为了三个部分，列表的位向量转化、位向量之间的按位与运算、由位向量转化为结果数组。(下面算法示例的以 ll int 存储为例，由位向量转化为结果数组实质上就是确定二级索引、位和位的下标三者的映射关系，此处省略）

Algorithm 4 列表的位向量转化

Input: 某个关键词对应的列表 $l(t_n)$

Output: 位向量（包含向量本身：$index[]$ 与二级索引：$Secondary[]$）

for each element $e \in l(t_n)$ **do**
    * 寻找元素对应位向量的位置 *
    $index\_pos \leftarrow e/64$ 数组下标
    $index\_bit \leftarrow e\%64$ 位
    $temp1 \leftarrow 1$
    $temp1 \leftarrow temp1 << index\_bit$
    $index[pos] \leftarrow index[pos] \mid temp1\_bit$ 按位或运算添加位
    * 寻找元素对应二级索引的位置 *
    $sec\_pos \leftarrow e/64$ 数组下标
    $sec\_bit \leftarrow e\%64$ 位
    $temp2 \leftarrow 1$
    $temp2 \leftarrow temp2 << sec\_bit$
    $Secondary[pos] \leftarrow Secondary[pos] \mid temp2\_bit$ 按位或运算添加位
end for
return $index[]$、$Secondary[]$

Algorithm 5 带二级索引的位向量按位与

Input: 两个列表 a、b 对应的位向量与二级索引
Output: 与运算后列表 a 对应的位向量与对应的二级索引

for $int\ i = 0; i < 6154; i++$ **do**
    $temp \leftarrow a.Secondary[i]\ \&\ b.Secondary[i]$
    if $temp == 0$ **then**
        $a.Secondary[i] = 0$ 二级索引与运算为 0，直接跳过
        $continue$
    end if
    * 存在与运算相同的位，开始寻找该位 *
    $new\_second = 0$
    $new\_second\_temp = 1$
    for $int\ j = 0; j < 64; j++$ **do**
        if $temp$ 第 j 位为 1 **then**
            位向量与运算
            $a.index[i*64+j] = a.index[i*64+j]\ \&\ b.index[i*64+j]$
            * 位向量存在相同的值，更新二级索引 *
            if $a.index[i*64+j]$ **then**
                $new\_second = new\_second \mid new\_second\_temp$ 按位或运算添加位
            end if
        end if
        $new\_second\_temp << 1$
    end for
    $a.Secondary[i] = new\_second$ 更新二级索引
end for
return $a.index[]$、$a.Secondary[]$

实现上述思路，按照 unsigned int 和 unsigned long long int 两种等级的二级索引进行测试，

得到的结果如下：

	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量
位向量 (unsigned int)	1000	14207.60	14.21	70.4
位向量 (ll int)	1000	9841.51	9.84	101.6

表 12: 位向量 不同等级索引效率对比

二级索引，每位对应 64 位的效率更高，以下的并行实验将以 64 位为参照。

3. SIMD 并行化

在实现串行算法后，发现实现 SIMD 并不方便。我们的意图是使用 AVX 指令集向量化处理向量之间的与操作。但实际上只有二级索引会按照连续的内存地址进行求交运算，而位向量只有在两个位向量对应的二级索引值均为 1 时才会进行与运算，大部分时候是地址不连续的，无法按照 AVX 指令集的方式从内存中连续取出（或者说取出时没有必要的）。二级索引的求交运算只占整个程序的一笑部分，如果只优化二级索引的求交运算反而会使后续对位向量的谭端更加麻烦。故我们再次改变索引的映射，二级索引一个位对应 256 个向量的位，而进行位向量的按位与时则可以使用 AVX256 进行向量化，然后使用 AVX256 与 openMP 结合（OpenMP 预编译语句：$pragma\ omp\ parrallel\ for\ schedule(dynamic,\ 5)$），得到程序性能测试结果如下：

	请求个数	平均总用时/ms	平均每条请求用时/ms	每秒吞吐量	加速比
位向量 (ll int)	1000	9841.51	9.84	101.6	-
AVX256	1000	10489.17	10.49	95.3	93.83%
openMP	1000	5300.47	5.30	188.7	185.67%
AVX+openMP	1000	5305.00	5.31	188.5	185.51%

表 13: 位向量 不同并行方式效率对比

SIMD 的改进并没有产生加速的效果。推测的原因如下：

- SIMD 的改进使二级索引的个数缩小到了 1/4，虽然可以同时进行四个 unsigned long long int 组成向量的按位与运算，但是二级索引与位向量的映射变得复杂，增加了计算量；
- 每次只将 4 个 ll int 加载为向量，并不会连续地取出，每一次取出向量都需要进行边界检查；
- 4 次按位与运算被改成了 2 次向量加载，1 次向量按位与，1 次存储指令，指令条数没有减少反而更加复杂。

五、 总结

本次实验对倒排索引求交算法进行了实现和分析。该算法的特点是程序过程中复杂计算较少甚至几乎没有计算，大部分由各种判断和分支语句构成，串行程序的优化空间很大。全程按照伪代码描述进行翻译整合得到的程序性能不佳，需要结合数据特点、应用场景等进行分析。如按列表求交算法的列表删除和元素查找，如果直接调用库函数进行实现，没有利用倒排列表数据有序的性质，得到的结果列表也没有必须内存连续的要求，运用这些函数过于浪费。

串行经过多次优化后的结果是难以实现并行化。如 Pthread 中欲对单次请求的效率进行优化，分析后发现函数内可以并行的部分很少，相互依赖的部分多，分支语句众多，想要安排多线程并行则要求大量的线程之间通信。在实现位向量按位与算法时也发现，虽然已经将列表向量化，但向量化的结果是仍然需要大量的按位操作；二级索引的建立大量减少了计算量，也使得位向量之间的求交预算变得分散零碎，不易于使用 SIMD 优化。其实只要稍对串行算法做出简化，比如删除二级索引直接进行向量的按位与，实现 SIMD 或是其他并行方式将会容易很多，但是这违背了并行提高程序性能的初衷，毕竟对于算法时间复杂度量级的优化比并行方法倍数的性能提高更加重要。

本次实验对多种并行方式进行了尝试和实现，即使并行的效果不佳、没有达到预期，还是如实将实验结果整理在了文档中。实验以提高程序的性能为目的，一步一步寻找是否有更好的并行解决方法，如使用 OpenMP 解决 Pthread 任务划分不均的问题，使用位向量法解决算法无法实现 SIMD 的问题（虽然结果不尽如人意），并不是一股脑将所有学习过的并行方法应用于算法中。体会到了很多程序分析、Debug 的乐趣，通过理论分析甚至逐行注释、添加输出语句找到程序的错误和影响程序性能的部分，增添了很多对程序设计的思考。

附：gitee 项目仓库:

https://gitee.com/shikimana/parallel1