**基于指令级并行的快速求交算法**

宋省身1 杨岳湘1 江宇2

1（国防科学技术大学 计算机学院，湖南 长沙410000）

2（西北核技术研究所，陕西 西安 710024）

**摘要：**布尔查询中的求交操作被广泛应用于各种信息系统中，是进行文档检索的基本操作之一。其基本形式可以视作多个有序整数序列的交集问题。为了提高求交运算的效率相继有许多算法被提出，近几年学者开始使用并行指令集（SIMD）来加速传统的求交算法，并取得了良好的效果。然而，这些优化仅局限于使用在两两求交的场景中，传统的多序列求交算法难以与之匹配。本文首先回顾了传统的多序列求交算法，然后针对其核心的搜索算法提出了两种基于SIMD的优化方式。实验证明，优化后的算法相比未使用SIMD的情况下有了很大的提升，甚至优于SIMD优化后的两两相交算法，性能最高提升37.3%。

**关键词：**求交算法；倒排索引；性能评价；并行处理

**中图分类号：**TP391  **文献标识码：**A

#### Efficient Multiple Sets Intersection using SIMD Instructions

Xingshen Song1, Yuexiang Yang1 and Yu Jiang2  1(College of Computer, National University of Defense Technology, Changsha, Hunan 410000 ,China)

2(Northwest Institute of Nuclear Technology, Xi’an, Shaanxi 710024, China)

**Abstract :** Conjunctive Boolean query is one fundamental operation for document retrieval in many information systems and databases. In its most basic and popular form, a conjunctive query can be seen as the intersection problem of multiple sets of sorted integers. Various algorithms have been put up in terms of maximizing the query efficiency. In recent years, researcher began to exploit the parallel advantage of single-instruction-multiple-data (SIMD) instructions to accelerate the intersection procedure and achieved substantial gains over previous scalar algorithms. However, these works only focus on intersecting two sets at a time rather than the whole lot of them.

Missing from the literature is a thorough study that explores the combination of traditional multiple sets intersection algorithms and SIMD instructions. In this article, we first revise all the traditional algorithms, then propose two optimizations on the essential searching algorithms using SIMD instructions. Experiments show that the optimized algorithms performs much better than the traditional ones, even outperform the recent SIMD intersection algorithms，and the improvement is up to 37.3% at most.

**Key words:** Set Intersection, Performance Evaluation, Vectorized Processing

**1 引言**

布尔查询中的合取操作是信息检索中最常用的查询方式之一，其他复杂的查询方式如排序查询、模糊查询都建立在它的基础之上。合取操作，也就是对多个集合中的元素求取交集，常见于对文档内容的检索中[1]。在当前的信息检索系统中，倒排索引作为一种简单高效的数据结构，被广泛应用于网页数据的存储和查询中[2]。倒排索引的结构由两部分组成，其中，词典用于存储查询词，并将其映射到相应的倒排链表中，倒排链表则按序存储了查询词出现的文档号、频率和位置信息。通常，这些信息都是以整型数的格式进行操作，因此，处理查询词的过程也可以视作对其相应的个倒排链中的整型数求取交集的过程。当前的互联网数据和用户持续增长，为查询处理的效率带来了巨大的压力，为了解决这些问题，相继有许多算法被提出以提高求交操作的效率，最早的工作可以追溯到几十年前。

尽管算法众多，但很难有一个通用的方法能够适用于各种场景下的求交操作，参与的倒排链的数量，倒排链的长度比以及相交元素的数量分布都会影响到算法的效率。除此以外，不同的算法也拥有各自的特点和适用的情况。最直接的一种求交方法叫做Zipper（zip），通过对倒排链两两逐项求交的方式进行计算。另一个针对两两求交的算法叫做BaezaYates（bys），它采用一种分治的方法不断地二分查找出相交元素，其工作过程类似于快速排序。当倒排链表的数量超过两个时，另一类算法不再采用两两相交，而是把所有参与的倒排链视作一个整体同时求交，这类算法每次从某一倒排链中选出一个元素作为*eliminator*，之后在其他链表中进行查找。根据选定和查找方式的不同，这些算法可分为set\_vs\_set（SvS），Adaptive（adp），Sequential（sql）和MaxSuccessor（max）[3]。

除此以外，还有方法从不同的角度来解决这个问题。某些使用了辅助性的数据结构如树堆（treap）、小波树（wavelet tree）、跳表（skip list）和哈希表（hash table）等实现了对链表内任意元素的快速访问，并以此减少了许多不必要的比较；某些方法利用了当前硬件的特性如GPU和多核技术来实现并行计算；还有一些方法研究了如何使用哈希结构快速估计交集大小而不是精确计算交集的问题[4]。然而这些方法或者需要额外的预处理和存储空间，或者对输入倒排链存在特殊的要求。

近年来，研究开始使用并行指令集来加速倒排链求交操作的过程，通过使用代码级的并行指令和更大的寄存器空间，基于单指令多数据流（single instruction multiple data, SIMD）的方法能够一次比较4个甚至更多的整型元素，从而极大地提升传统算法的速度。相比于使用GPU和多核技术的优化算法，SIMD提供了更加简单易用的接口和灵活高效的实现方式，在倒排索引的压缩算法中已经得到了广泛的应用[5]。但在查询处理，尤其是布尔查询中，SIMD的应用还存在一些提升空间。当前的优化方法仅仅针对倒排链两两相交的情况，工作方式更类似于并行化的zip，使其适用范围局限在了长度相近并且交集较大的倒排链求交情景中。而传统的面向多个倒排链求交的算法并没有得到关注，并行指令集和多倒排链求交算法的结合在研究领域内尚属空白。为此，本文首先回顾分析了传统多倒排链表求交运算中具有代表性的方法以及其基于SIMD的优化，然后针对多倒排链求交算法提出了两种并行化的搜索算法，实验证明，这些算法能广泛适用在传统的多倒排链求交算法中，并且显著提高了它们的性能。

**2 相关工作**

倒排链求交问题一直是信息检索领域中的热门研究点，各种各样的算法和数据结构不断地被提出以提高这一问题的计算效率。除去某些特殊的研究点（如快速估计交集大小上限），本文仅考虑精确计算出交集的求交算法。同时，压缩状态下的倒排链或者增加了辅助数据结构的求交算法同样不在本文的研究范围，因为它们在便利了某一种计算场景的同时，也会限制算法在其他场景下的应用，本文所使用的倒排链均视作在压缩展开后的状态下进行求交。

**2. 1 多倒排链求交算法**

传统的串行求交算法主要关注如何减少不必要的比较。针对查询词，对应的倒排链为，其长度，算法的基本流程主要包含两个部分，一是从中选取*eliminator* ，二是从其他倒排链中使用特定的搜索算法*F-Search*找到不小于的元素。如果在所有倒排链中都被找到，那么称之为一次*full match*，否则是*mismatch*，无论如何，都会被更新，然后开始新一轮的搜索直至任一倒排链遍历完毕。按照处理倒排链的方式，可将这些算法分为两类，一类为两两相交，一类为整体相交。

其中，两两相交的通过每次选取两条最短的倒排链求取交集，再把结果与剩余倒排链中最短的重新求交，直至全部遍历完毕。zip和bys是两种典型算法。zip使用了线性查找（Linear search）作为*F-Search*，其时间复杂度为相交两倒排链长度之和，即。而bys则是基于二分查找（Binary search），不断地在长倒排链中寻找短倒排链的中值，如果找到则加入交集中；随后两个倒排链均被拆分为两部分，查找继续递归进行，直至任一方变为空集，其平均时间复杂度为。

整体相交方法同时搜索所有的倒排链，类似于排序查询中的Document-At-A-Time（DAAT）策略，其伪代码如图1所示。其中，最简单的策略叫做set\_vs\_set（SvS），每次都是从中选取，并在其他倒排链中二分查找。然而SvS没能利用倒排链中的数值分布特性，查找过程中其他倒排链很有可能取代成为最短倒排链，为此Demaine等人[6]提出了swapping\_SvS，在每次更新后，按照剩余倒排链长度进行重排序。adp和sql是应用较为广泛的算法，它们使用了一种新的*F-Search*，叫做Galloping search。即从初始步长1开始，不断倍增步长，直至找到不小于的元素位置，然后在当前步长内二分查找到具体位置。而如果发生了*full match*，从当前链的下一个位置选取，否则由引起*mismatch*的元素替代。sql和adp唯一的区别，就是在执行Galloping search时，adp每次循环只向前移动一次，而sql则是直至发现不小于的值才停止。尽管二者的时间复杂度相同，但实验显示adp在相交元素比较密集时表现出色，而后者更适用于交集较大的情况。结合Swapping\_SvS的特点，Demaine等人[7]又提出了small adp，在每次更新之前按照长度对倒排链进行重排序，以保证每次都从最短的倒排链中选取。Clupepper和Moffat[8]认为重排序会引入额外的时耗，为此他们提出了max来克服这一问题，即维持原来的访问顺序，而由的下一个元素和引起*mismatch*二者中的大值决定，以避免选择了无效的后引起的冗余比较。

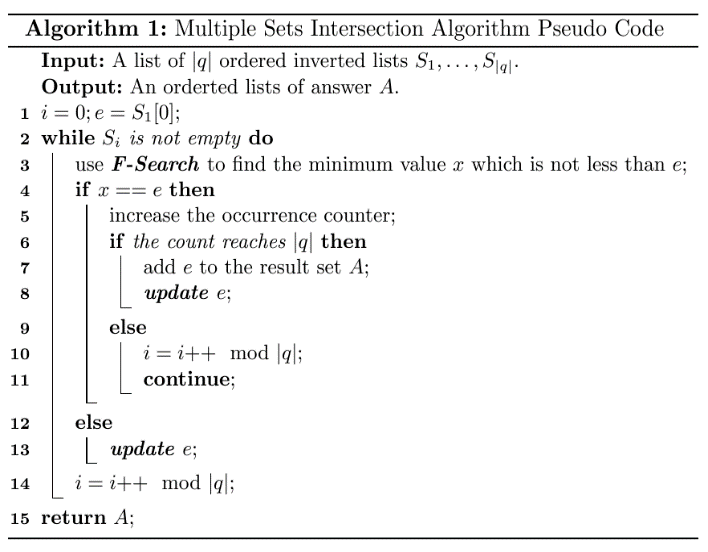


图1 多倒排链求交算法的伪代码

从图1的算法中，可以看到*F-Search*是算法中每一次时耗最长的部分，为此出现了很多搜索算法，除了之前提到的Linear search, Binary search和Galloping search，还有Golomb search, Fibonacci search以及Interpolation search[9]。

**2. 2 基于SIMD的求交算法**

近年来，研究者开始使用并行指令集来加速信息检索领域的传统算法。通过使用长度为128bit的寄存器（称为XMM寄存器），指令可以同时操作4个32bit的整型数，近年来还出现了256bit的YMM，甚至512bit的ZMM。目前主要针对索引压缩和求交两个方面，而并行指令的引入使效率得到了极大地提升。

Schlegel等人[10]使用了Intel SSE4.2中的STTNI指令集合，能够在一次调用中比较16个8-bit或者8个16-bit的整型数，这样在两个倒排链之间就能以块为单位进行比较。对于长度超过16bit的数字，他们设计了一种层次化结构来分开存储高低16bit部分，只有两部分完全相同时，才记为一次相交。该算法对两个长度相当并且值域较小的倒排链比较有效，但指令消耗的时钟周期相比于一般的SIMD指令要更高，因此提升效果有限。Lemire等人[11]选择每次从短倒排链中取出一个元素，然后同时与长倒排链的多个元素进行比较，这种算法可以视作并行化的Golomb search，长倒排链的指针每次移动一个或者多个分块的长度，直至找到可能相交的分块，并在分块内使用二分查找，但算法仅能返回相交位置的块头地址，想要获取精确地址就不得不引入块内的线性查找。为了适应不同长度的倒排链，他们提出了V1，V3和SIMD GALLOPING三种算法。Inoue等人[12]则是针对减少难以预测的*if\_greater*判断，重新组织了存放在寄存器中的元素，通过使用代价更小的*if\_equal*判断仅比较头部字节的情况下，排除掉大多数不相交元素。

上述三个算法尽管在效率上都取得了显著的提升，但仅考虑了两两相交的情况，并没有针对多倒排链相交做过验证，而传统多倒排链相交算法的优势并没有得到利用，这就制约了它们的适用范围。为此，我们介绍了两种并行优化的多倒排链求交算法。

**3 并行的多倒排链求交算法**

从图1中的算法可知，选择和*F-Search*是影响算法效率的重要因素，一个有效的可以避免很多无效的比较，而*F-Search*则是影响每次查询效率的关键步骤。但就目前的指令集来说，处理器仅能支持读取连续的数据块，而选取需要从多个倒排链中或者一个倒排链多个位置中选取最大值，支持这种操作的高级指令属于最新发布的AVX-512VL，目前尚未有处理器支持。为此，我们仅考虑从*F-Search*的角度优化算法。

**3. 1 基于SIMD的线性查找算法**



图2 并行的求交算法的示例

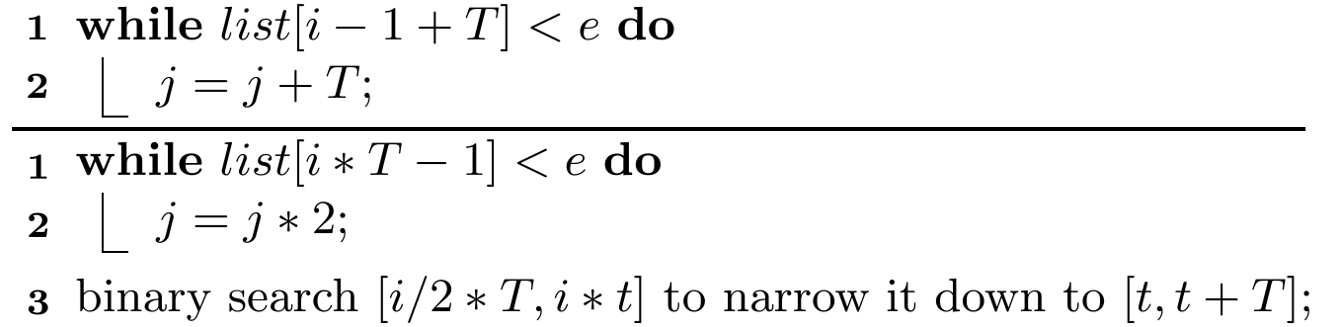
算法的基本思想与V1相近，当选定以后，针对每一个待比较倒排链，同时取出长度为个元素，以寄存器长度（128bit）为单位进行比较，然后返回第一个不小于的元素在块内的位置。具体步骤如下：

1. 把复制4份，载入寄存器中；
2. 从当前倒排链起始位置取出4个元素载入寄存器中，使用PCMPGTD指令对和以32bit为单位进行比较，返回长度为128bit的结果，若对应位置小于，则全部置1，否则置0；
3. 使用MOVMSKPS指令提取出的最高位，按顺序移入一个32bit整型数的高4位；
4. 继续读取当前倒排链的后四个元素，重复2，3步骤，将得到的整型数在原来的基础上偏移4位后求并；
5. 重复2，3，4步骤，直至把的所有bit位填满；
6. 统计第一个0出现的位置，即为当前块内第一个不小于的元素。

图2显示了算法的运行过程。值得一提的是，当前指令集仅支持比较带符号整型数，无符号数比较需要最新的AVX512指令集，因此当倒排链的值域超过了231，则需要使用PSUBD指令把，内的元素都减去231后才能进行比较。由于MOVMSKPS只能返回长度为32的整型数，这也是我们选择的原因。假设相交元素的平均距离为，易知，在未使用SIMD指令的情况下，线性查找需要进行次比较，而本算法则需要进行次比较，即每次都会相对增加次比较。并行指令集能够高效地执行这些操作，并且节省了多次的寻址操作，增加了缓存命中率，进而提高了求交算法的效率。同时，我们可以使用更小的（必须保证是4的整数倍，以对齐寄存器长度）来减少额外的时耗。

**3. 2 基于SIMD的跳跃式查找**

线性查找的劣势也很明显，它仅对比较小的倒排链有效，否则会产生很多无效的比较。而现实中的倒排链相交比例往往不会超过40%，因此加入指针跳跃，能够有效地提高算法效率。跳跃式的查找中，Fibonacci search和Interpolation search需要通过额外的计算来确定跳跃位置，研究中发现Golomb search和Galloping search这两种简单的跳查方式反而能够实现更高的效率。为此，我们同样选择这两种跳查方式作为并行优化的基础。



*Galloping search*

*Golomb search*

图3 两种搜索方法的代码片段

给定块长，二者都是比较块尾部元素与，若小于则继续向前移动，不同的是Golomb search每次都移动一个的长度；而Galloping search则是从开始，不断倍增，当移动到大于的停止间隔为时，在范围内二分查找到更细粒度的块位置。对于块内的查找，我们使用3.1节所提到的方法。对于倒排链长度不是整数倍的，多余部分使用zip进行求交。其伪代码如图3所示。

此时，对于平均距离，Golomb search需要次比较，Galloping search需要次比较，尽管可以通过求导获取最佳值以相对减少比较次数，但实际上小的值会引起处理器分支预测未命中（misprediction）的概率升高。相比于单条SIMD指令所节省的时钟周期（通常为2~3个），一次misprediction通常导致10~15个时钟周期的浪费。实验中发现，能取得更高的效率。而当值超过32时，需要再增加一层二分查找，把间隔的长度缩小至32以便使用之前的方法。

**3. 3 提高算法流畅性**

注意到3.1节为了返回相交的精确位置，使用了PCMPGTD指令获取寄存器长度内第一个不小于的偏移地址；当时，必须通过移位（MOVMSKPS）、求并（POR）、计数（POPCNT）获取块内的偏移地址；由于指令集的限制，还必须通过PSUBD指令取消符号数比较的限制；而Lemire提出的算法则是依赖于内的zip返回精确位置。然而这些操作会引入额外的代价，相比于直接粗暴地返回块头地址，甚至会更高。

SIMD指令集的特点即是在更短的时间内执行更多的指令，因此它对于比较次数有更高的容错性。早期基于SIMD的优化，相对于传统的求交算法，在比较次数上都要高出一个数量级，而运行时上却仍然能够得到较大的减少。由此可以看出，保证算法执行的流畅性，要远胜于返回结果的精确度。其中，*if\_greater*判断引起的misprediction是阻碍流畅性的重要因素。通过使用代价更小的*if\_equal*判断替换；以及移除元素的定位操作，取而代之仅返回相对粗糙的头部地址，可以节省很多步骤。这样做的代价，是在下一次的*F-Search*中，重复比较至多不超过长度个数的元素。具体步骤如下：

1. 把复制4份，载入寄存器中；
2. 从当前倒排链起始位置取出4个元素载入寄存器中，使用PCMPEQD指令对和以32bit为单位进行比较，返回长度为128bit的结果，若对应位置等于，则全部置1，否则置0；
3. 继续读取当前倒排链的后四个元素，重复2步骤，将得到的结果与上次的结果使用POR指令求并，直至遍历完内所有元素；
4. 对最后的结果，使用PTEST判断是否全部为0，若是标记未发现相交元素，否则标记发现。返回的位置始终是当前块的头部位置。

这样一来，仅需要使用PCMPEQD、POR和PTEST三种指令，而且指令数量也从下降到了。尽管返回的地址并不准确，很有可能取到一个小于的值，但针对平均距离情况下的跳查，只需要一次比较即可跳过。更重要的是，传统的多倒排链求交算法中，都考虑了对的筛选，如重排序和取最大值的方式，都能大大降低选中这个无效值的概率。

**4 实验结果与分析**

**4. 1 实验设置**

实验的目的是为了验证算法对多倒排链求交的效率，而倒排链中的数据是以排序整型数的形式存储的，为此，我们选择通过生成随机数来模拟排序倒排链，生成的方式按文献[11]所使用的ClusterData distribution。而真实的数据集与查询集通常会有一些偏置，不能像生成数据一样精确地控制长度、交集大小等信息。倒排链的值域设为[0,232-1]，参与的倒排链数量为2到10。其中，最短倒排链的长度从210倍增至214，与最长倒排链的长度比设为1，10，102，103，104，数量超过2时，中间的倒排链长度取二者间的随机值。同时，交集与最小倒排链的大小比例的取值有10%，50%，100%。

生成倒排链时，首先按照生成交集，然后以交集为基础，按照相交比例补充生成最短倒排链，之后按照长度比例补充生成其他参与求交的倒排链，以此控制相交比例。为了避免误差，每种情况均生成20个样例，结果取5次运行后的平均值。

实验机器是1台主频3.30GHz的4核Intel Core i5-4590，配有16G内存和6MB缓存的Linux服务器。为了避免与之前SIMD算法比较时，高阶指令集带来的性能误差，实验仅使用了128bit长度的寄存器。所有的算法均采用C++实现，由G++4.8.1使用“-O3”编译选项编译。我们将代码开源在了GitHub[[1]](#footnote-1)上。

**4. 2 实验结果**

首先，我们比较前文提到传统的多倒排链求交算法，以验证其在各种情况下的效率。参与比较的算法有zip、bys、SvS、swapping\_SvS、sql、small\_sql和max。由于操作和性能上的差别过小，adp，sql和small\_adp，small\_sql在图中位置几乎重叠，而sql，small\_sql始终略优于前者。为了便于观察，我们没有显示adp。另外，实验中发现交集大小的比例几乎对算法效率毫无影响，比较次数仅与倒排链长度相关，即无论元素是否真的相交，只能通过比较后才能判断，因此10%，50%和100%的相交比例耗时差别并不大。实验中三者的图形基本一致，为了节省空间，这里仅显示相交比例为50%的情况。

图4显示了各个算法在不同的长度比和倒排链数量下的运行时间。首先可以注意到，zip效率变化最大，在长度比小于10时，它是最优的方法，随后则越来越差，而其他算法的相对位置则基本不变。可知，潜在相交元素的距离决定了搜索方式的适用性，密集的排列反而使跳查命中率降低，进而影响到效率。事实上，决定算法运行时间的关键因素是最短长度的倒排链而非其他因素，当任一倒排链遍历完毕后，算法即会结束。由此也可以看出，如何选取最优的*eliminator*来尽快排除掉无效元素，使最短倒排链完成遍历，是提高效率的关键。

D:\WorkSpaceR\5_intersection\Rplot02.tiff

图4 传统多倒排链求交算法的效率比较

参与的倒排链数量与计算时间基本呈线性关系，随着数量的增加，计算时间也相应上升。具体到各个算法中，可以看到zip是效率最低的算法，因为它没有任何的跳转，仅通过顺序遍历所有倒排链求交；之后是sql，虽然使用了高效的Galloping search，它的效率相比其他使用了跳查的算法要慢很多，这主要是由于其糟糕的*eliminator*选取方法造成的，即不考虑运行时倒排链长度的变化。可以看到最简单的SvS通过每次仅从最短倒排链（未重排序）选取*eliminator*，同样可以达到很好的效果。而加入了运行时对倒排链长度的监控后，small\_sql的效率得到了极大的提升，仅次于最优的max，同时，也可以看出加入重排序后还是会略微降低效率，不涉及重排序的max始终保持第一的位置。尽管除了zip外，其他算法在图形中比较接近，但注意到纵轴是对时间以10为底对数化后的结果，为的是压缩图形，实际中还有一定差距的。随着长度比的增大，图形也越来越有分离的趋势，这就凸显了不同的*F-Search*在搜索时跳过无效元素的优劣。同样注意到bys作为两两相交的算法，其分而治之的方式同样达到了很好的效果。

其次，我们比较基于SIMD的两两相交算法在多倒排链求交的情况下的表现，同时加入之前效率最高的算法（长度比例小于100zip，大于100时max）作为参照系。其中Schlegel方法为了能够适用长度超过16bit的值，通过预处理构建了层次化结构，相当于对倒排链的元素按照高低位预先进行了一遍汇总。

图5显示了各个算法的运行时间。最为显著的一点是，Schlegel随着参与倒排链的数量上升，其效率反而也得到了提升。这要归功于其预处理过程合并了很多高位相同的元素，仅通过比较高位部分就可以滤除很多的无效元素，使运行时效率得到提升，但预处理同样需要时间和空间来存储这些数据，也限制了它在其他查询中的适用性。同时，该算法在查询词长度超过4时（长倒排链需要超过8）才能体现出优越性，这在实际查询中的适用性又将大打折扣。Inoue的效率在长度比为1时最优，随后越来越差；V1、V3、SIMD GALLOPING三个算法的差异在其跳查所使用的步长，可以看出，随着步长的增加，效率也越来越高。max作为参照系，其性能仅次于V3和SIMD GALLOPING，可见这些算法在处理多个倒排链相交时并没有充分利用到并行指令集的优势。接下来，我们展示本文所提出算法的性能。

D:\Program Files\RStudio\Rplot.tiff

图5 基于SIMD的两两相交算法的效率比较

定义返回准确位置的方法为exact，仅返回块头位置的方法称为rough，再与3.2节两种跳查方式结合，共有四中搜索方式：Golomb\_exact、Golomb\_rough、Gallop\_exact和Gallop\_rough。它们可以使用在几乎所有的多倒排链求交算法中。同样，我们以max为例，比较了这四种方法与未优化的时间效率，并加入了之前最优的算法（长度比1时的Inoue，以及其他情况下的SIMD GALLOPING）作为参照，如图6所示。从图中可以看出，使用SIMD优化后的max算法适用于长度比较大的情况，在长度比为1时略输于原始算法，而Inoue则远优于其他。而这四种方法中，可以看到rough的效率总是优于exact，这也验证了之前的观点，misprediction的造成的延迟要远大于重复比较。另一项观察是，随着长度比越来越大，Golomb的性能越来越差，注意到尽管其他算法在图形上越来越近，但纵坐标是经过对数化后的结果，Gallop\_rough始终能保持优势。这是由于潜在的相交元素变得更加稀疏，步长跨度大反而能更好地排除无效元素。注意到当长度比小于1000时，Golomb和Gallop的时耗都很小，而超过1000时二者的时耗上升了一个数量级，Gallop的提升效果更为明显。

D:\WorkSpaceR\5_intersection\Rplot05.tiff

图6 本文所提出算法的效率比价

表1 各算法在优化前后的平均计算时间（ms）

|  |  |  |  |
| --- | --- | --- | --- |
| Methods | without SIMD | Golomb\_rough | Gallop\_rough |
| SvS | 1.66 | 1.21(-27.1%) | 1.04(-37.3%) |
| swapping\_SvS | 1.65 | 1.26(-23.7%) | 1.10(-33.3%) |
| sql | 2.30 | 2.00(-13.0%) | 1.89(-17.8%) |
| s\_sql | 1.37 | 1.24(-9.49%) | 1.10(-19.7%) |
| max | **1.36** | **1.04(-23.5%)** | **0.92(-32.4.0%)** |

表1显示了所有的多倒排链求交算法在未使用SIMD、Golomb\_rough和Gallop\_rough三种情况下的平均计算时间（相交比例50%下所有参与的测试样例）。可以看到所有的算法都得到了不同程度的提升，其中SvS的性能提升最为明显，接近37.3%。这是因为不同的算法对*F-Search*依赖的程度不同，例如max本身已经可以通过筛选eliminator减少很多的比较了，而SvS则是由于涉及比较的次数最多，因而优化效果最明显。从性能上比较，max仍然是最优的算法。

**5 结束语**

本文主要研究了使用并行指令集对多倒排链求交算法的优化问题。首先，我们回顾了传统的多倒排链求交算法，以及近年来提出的基于SIMD的两两相交算法。在分析了影响求交算法效率的因素后，我们提出了两种基于SIMD的搜索算法，这些方法思路简易通用性好，在实验中也展示出了较大的性能提升和广泛的适用性。

目前的算法设计中，很多的参数如块长，并行指令与判断分支的数量比，不同长度比下搜索算法的选择仍然是靠经验性的设置。未来的研究中，将尝试构建指令代价的度量模型，通过回归分析选择最优的算法组合，来进一步提高就交算法的效率。

**参考文献**

1. Culpepper J S, Moffat A. Efficient set intersection for inverted indexing[J]. Acm Transactions on Information Systems, 2010, 29(1):1.
2. Zobel J, Moffat A. Inverted files for text search engines[J]. Acm Computing Surveys, 2006, 38(2):6.
3. Jérémy Barbay, Lu T, Salinger A. An experimental investigation of set intersection algorithms for text searching[J]. Journal of Experimental Algorithmics, 2009, 14.
4. Takuma D, Yanagisawa H. Faster upper bounding of intersection sizes[C]// International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM, 2013:703-712.
5. 闫宏飞, 张旭东, 单栋栋,等. 基于指令级并行的倒排索引压缩算法[J]. 计算机研究与发展, 2015, 52(5):995-1004.
6. Demaine E D, López-Ortiz A, Munro J I. Adaptive set intersections, unions, and differences[C]// Eleventh Acm-Siam Symposium on Discrete Algorithms, January 9-11, 2000, San Francisco, Ca, Usa. DBLP, 2000:743-752.
7. Demaine E D, López-Ortiz A, Munro J I. Experiments on Adaptive Set Intersections for Text Retrieval Systems[C]// Revised Papers from the Third International Workshop on Algorithm Engineering and Experimentation. Springer-Verlag, 2001:91-104.
8. Culpepper J S, Moffat A. Compact Set Representation for Information Retrieval[C]// String Processing and Information Retrieval, International Symposium, Spire 2007, Santiago, Chile, October 29-31, 2007, Proceedings. DBLP, 2007:137-148.
9. Barbay J, Pez-Ortiz A, Lu T. Faster adaptive set intersections for text searching[C]// Experimental Algorithms, International Workshop, Wea 2006, Cala Galdana, Menorca, Spain, May 24-27, 2006, Proceedings. DBLP, 2006:146-157.
10. Schlegel B, Gemulla R, Lehner W. Fast integer compression using SIMD instructions[C]// International Workshop on Data Management on New Hardware. ACM, 2010:34-40.
11. Lemire D, Boytsov L, Kurz N. SIMD compression and the intersection of sorted integers[J]. Software Practice & Experience, 2014, 46(6):723-749.
12. Inoue H, Ohara M, Taura K. Faster set intersection with SIMD instructions by reducing branch mispredictions[J]. Proceedings of the Vldb Endowment, 2014, 8(3):293-304.

1. https://github.com/Sparklexs/SIMDIntersections [↑](#footnote-ref-1)