南開大學
Nankai University

计算机学院

并行程序设计实验报告

倒排索引求交 SIMD 实验

姓名：唐明昊

学号：2113927

专业：计算机科学与技术

2023 年 4 月 16 日

目录

1 选题介绍

1.1 选题问题描述

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。

倒排列表求交（List Intersection）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 $\cap_{1\leq i\leq k} l(t_i)$。

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：

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

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：

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

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

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

求交操作返回三个倒排列表的公共元素，即：

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

链表求交有两种方式：按表求交（SVS 算法）和按元素求交（ADP 算法）。另外，我们还找到拉链法、跳表法、Bitmap 位图法、Hash 优化法、递归法等方法，针对 SIMD 编程的情景，我们选择了 Bitmap 位图法和 Hash 优化法作为研究的重点，同时综合两种基础算法：SVS 和 ADP 进行并行化研究。

1.2 实验分工

本小组由唐明昊和朱世豪组成。唐明昊负责 SVS 算法和 Hash 优化算法，并将拉链法融入其中；朱世豪负责 Bitmap 算法和跳表法；ADP 算法由二者共同讨论完成。在具体编写代码过程中，为了排除代码编写习惯带来实验数据误差，都有进行充分的讨论。另外由朱世豪同学负责进行鲲鹏服务器上的实验，主要数据由其账号运行获得。

2 算法实现

附GitHub仓库

2.1 串行算法

2.1.1 按表求交 SVS

SVS 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**，思考到对两集合求交可以**结合拉链法的思想**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**。

更进一步，发现开题报告中给出的伪代码不足之处：两个集合求交过程中，发现一个不合元素就进行删除，会严重拖慢算法执行时间。使用标志位，元素合适就移动标志位，不合适即不移动，到下一个合适元素即会覆盖掉当前不合元素，当两个表求交执行完成以后，**将标志位以后的元素一起删除**，可提高速率。

SVS 核心代码

```
// s列表中的每个元素都拿出来比较
for(int j=0;j<s.length;j++){// 所有元素都得访问一遍
    bool isFind = false;// 标志，判断当前count位是否能求交
    for (; t < index[queryList[i]].length; t++) {
        // 遍历i列表中所有元素
        if (s.docIdList[j] == index[queryList[i]].docIdList[t]) {
            isFind = true;
            break;
        }
        else if (s.docIdList[j] < index[queryList[i]].docIdList[t])// 升序排列
            break;
    }
    if (isFind)// 覆盖
        s.docIdList[count++] = s.docIdList[j];
}
if(count<s.length)// 最后才做删除
    s.docIdList.erase(s.docIdList.begin() + count, s.docIdList.end());
s.length = count;
```

2.1.2 按元素求交 ADP

ADP 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，只要其中一条列表走到尽头，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**记录列表访问位置**，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。

ADP 核心代码

```
while (list[0].cursor < list[0].length) {// 最短的列表非空
```

```
    bool isFind = true;
    int s = 1;
    unsigned int e = index[list[0].key].docIdList[list[0].cursor];
    while (s != num && isFind == true) {
      isFind = false;
      while (list[s].cursor < list[s].length) {// 检查s列表
        if (e == index[list[s].key].docIdList[list[s].cursor]) {
          isFind = true;
          break;
        }
        else if (e < index[list[s].key].docIdList[list[s].cursor])
          break;
        list[s].cursor++;// 当前访问过，且没找到合适的，往后移
      }
      s++;// 下一个链表
    }
    list[0].cursor++;// 当前元素已被访问过
    if (s == num && isFind) {
      S.docIdList.push_back(e);
      S.length++;
    }
    //sort(list , list + num);// 重排，将未探查元素少的列表前移
```

2.1.3 Hash 优化

受 bitmap 算法启发，**用空间换时间的思想**，优化 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，很少的步骤即可查找到，不需要从头进行遍历。

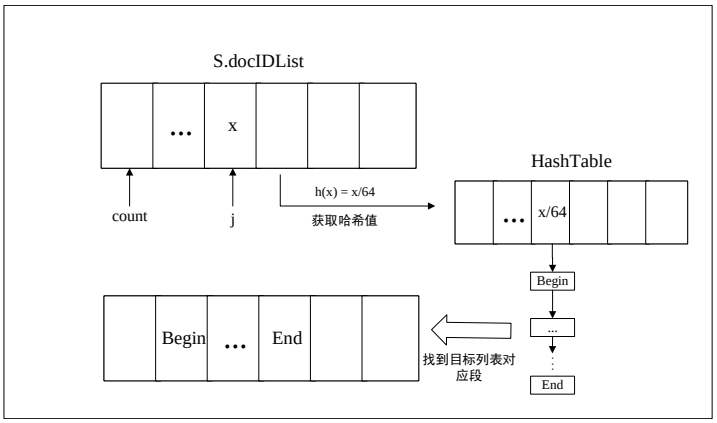

图 2.1: Hash 算法示意图

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：

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

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

hash 算法核心代码

```
    bool isFind = false;
    int hashValue = s.docIdList[j] / 64;
    // 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;
    if (begin==-1) {// 该值肯定找不到，不用往后做了
        continue;
    }
    else {
        for (begin; begin <= end; begin++) {
            if (s.docIdList[j] == index[queryList[i]].docIdList[begin]) {
                // 在该段中找到了当前值
                isFind = true;
                break;
            }
            else if (s.docIdList[j] < index[queryList[i]].docIdList[begin]) {
                break;
            }
        }
        if (isFind) {
            // 覆盖
            s.docIdList[count++] = s.docIdList[j];
        }
    }
```

2.1.4 拉链法

两个指针指向首元素，比较元素的大小：如果相同，放入结果集，随意移动一个指针；否则，移动值较小的一个指针，直到队尾。

Algorithm 1 拉链法 (Zipper)

Input: list A, list B

Output: Intersection Results C

function INTERSECTION(A, B)
  Sort(A)
  Sort(B)
  $pointerA \leftarrow 0$
  $pointerB \leftarrow 0$
  $C \leftarrow \{\}$
  while $pointerA < \text{len}(A)$ and $pointerB < \text{len}(B)$ **do**
    if $A[pointerA] == B[pointerB]$ **then**
      C.add($A[pointerA]$)

```
            pointerA ← pointerA + 1
        else if A[pointerA] < B[pointerB] then
            pointerA ← pointerA + 1
        else
            pointerB ← pointerB + 1
        end if
    end while
    return C
end function
```

本次实验并没有单独实现拉链法，**而是将拉链法融入到了 SvS 和 ADP 算法中**，以最大化串行算法的表现。

2.2 并行算法

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。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。bitmap 算法由朱世豪同学进行编写，本报告只讨论对其余算法的并行化。

2.2.1 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，下面以 SSE 为例介绍该思想。

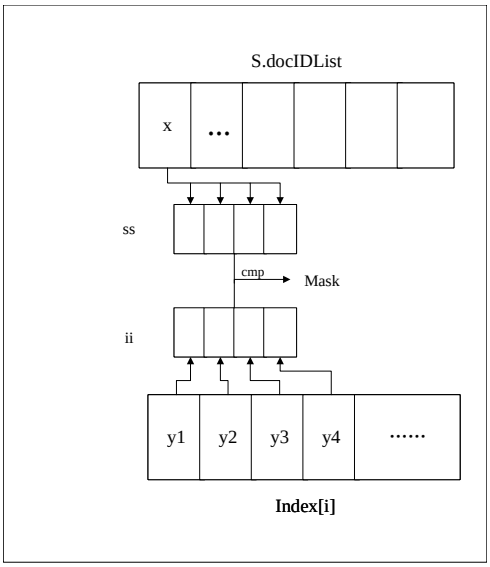

图 2.2: 并行算法示意图

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，进行四位比较。

通过对比较指令 _mm_cmpeq_epi32 的返回值的研究，再结合前期调研留存的 HIGHLY SCALABLE 博客 [1] 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。

SSE 代码块

```
unsigned int e = index[list[0].key].docIdList[list[0].cursor];
__m128i ee = _mm_set1_epi32(e);
// 构造向量
__m128i ii =
    _mm_load_si128((__m128i*)&index[list[s].key].docIdList[list[s].cursor]);
__m128i result = _mm_set1_epi32(0);
result = _mm_cmpeq_epi32(ee, ii);// 两个向量比较
int mask = _mm_movemask_epi8(result);// 得到掩码
if (mask != 0) {// 查看比较结果
    isFind = true;
    break;}
```

2.2.2 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];
short sEle = index[i].docIdList[begin+j] & 0x0000FFFF;
sArray[j]=sEle;
__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**，反而造成了最终效率远低于通用性并行算法。

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 50 100，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 实验环境

本实验对 arm，x86 两种架构及其上 Neon，SSE 和 AVX 指令均做了测试，均采用 g++ 编译器。

- arm 测试在鲲鹏平台运行，配置如下：

  aarch64 架构, L1 cache 64kB, L2 cache 512kB, L3 cache 49152kB

- x86 测试在本地电脑运行，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

4 实验结果分析

4.1 对齐与不对齐的影响

由于集合求交算法下标均是顺序移动，天然很进行好内存对齐，为了探究内存对齐对算法执行效率的影响，我们用 SVS 算法尝试，对其做了调整，让其首先处理两个 DocID，后续从下标 2 开始，每次增 4 的访问，故意构造不对齐。

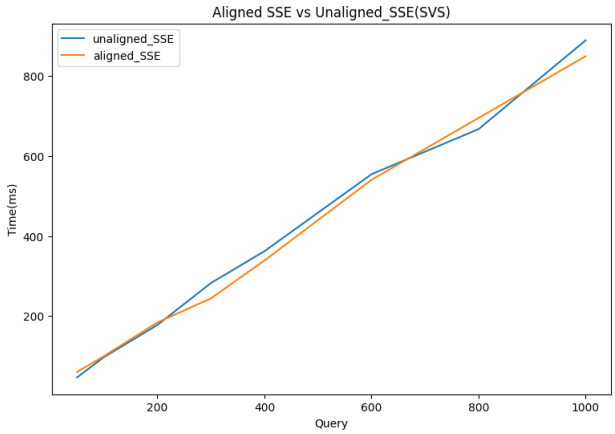

图 4.3: 对齐与不对齐性能比较

但从图2.1看到，不对齐的内存访问，**最终的执行效率并不比对齐的访问低下**，在测试过程甚至还会有略微的领先情况出现。

针对实验现象，我们查阅了资料，觉得可能有以下原因：

- 数据不是完全随机的，而是有一定的**重复性和局部性**。未对齐的内存访问可以更好地利用处理器内部的预取缓存机制，从而减少内存访问延迟，使得性能得到一定程度提升。
- 集合求交并行化算法只能对最内层循环进行并行化处理，过程中有一定的数据依赖性，使得并行化对性能提升不够极致。

4.2 串行算法纵向对比

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

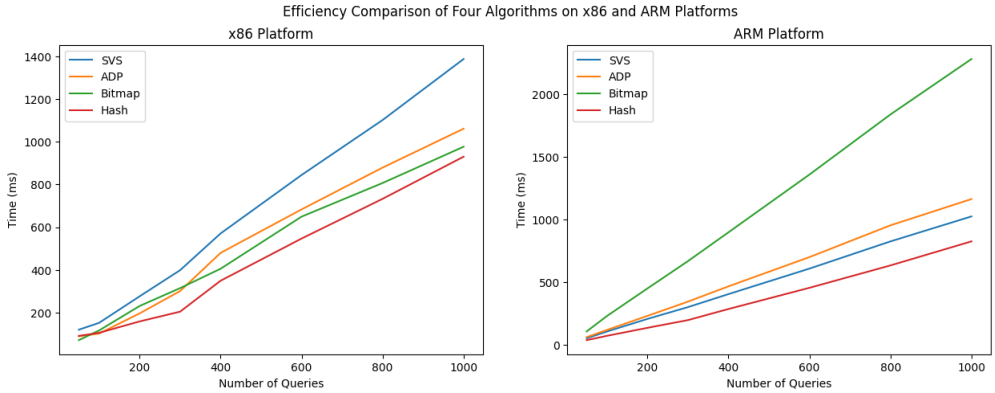

图 4.4: 各串行算法对比

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 表内冲突很少，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，在性能上可以比肩 ADP 算法，二者时间复杂度都接近 $o(n * l^2)$。

Events	ADP	SVS
L1 cache Hit	49,485,845,556	50,325,684,759
L1 cache Miss	28,880,919	29,250,534
L2 cache Hit	25,123,917	24,548,448
L2 cache Miss	3,670,242	4,458,333
L3 cache Hit	1,198,032	1,500,954
L3 cache Miss	2,158,350	2,366,961

表 1: SVS 与 ADP 各级 cache 命中

Events	ADP	SVS
Clockticks	94,519,650,186	97,021,375,308
Instructions Retired	168,621,112,530	167,372,712,364
CPI Rate	0.561	0.58

表 2: SVS 与 ADP CPI 大小

从二者的 VTune 分析结果可以看出，二者整体的 cache 命中相当，CPI 值上 ADP 略优于 SVS，故在 x86 架构下，ADP 算法对 SVS 算法有着微弱优势。

而由于 ADP 算法需要开辟链表记录各个列表中光标位置，读取 DocId 时需要查询光标，**空间局部性稍差**。在鲲鹏的 arm 平台下，cache 较小，cache 未命中增多，性能被 SVS 算法超越。

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)。虽然在 x86 平台 BitMap

算法性能表现优异，但在鲲鹏服务器下，因为 cache 较小，这种开辟大量空间换时间，且空间局部性较差的算法，性能大幅下降。

4.3 串行并行横向对比

本报告只讨论 SVS、ADP 和 Hash 的并行化，三者均采用在 2.1.1 提到通用性并行化手段进行并行。

对于 NEON 指令集，使用 vmovq_n_u32 指令将一个元素填充 128 位向量，再用 vld1q_u32 指令获取一次读取四个元素填充向量，最后使用 vceqq_u32 指令进行比较。但由于查询到的 vmovemaskq_u32 指令无法使用，退而求其次，将掩码保存到一个大小为 4 的数组中，用一个判断语句判断是否有匹配情况发生。arm 平台下串并行算法对比结果如图4.5所示

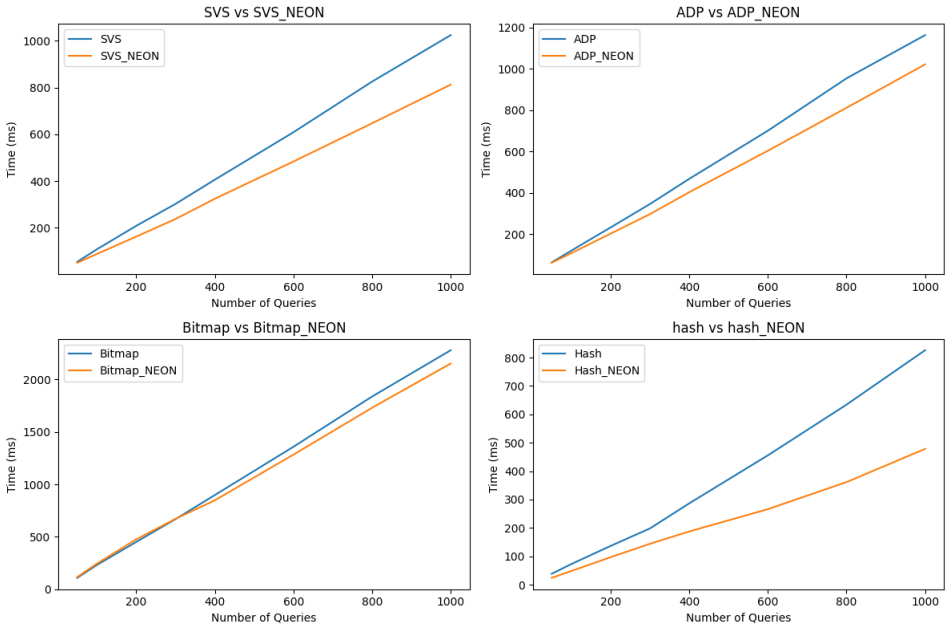

图 4.5: arm 下串行与 NEON 并行对比

SSE 指令和 AVX 指令思路一致，将 128 位向量修改为 256 位向量，从表4看，两种并行化相较于串行算法都有了明显的性能提升。VTune 分析如表3所示，并行算法虽然 CPI 略由提升，但因为单指令多数据操作，指令条数明显减少，故算法执行效率提高。

Events	ADP	ADP_SSE	SVS	SVS_SSE
Clockticks	94,519,650,186	87,590,486,776	97,021,375,308	107,283,481,760
Instructions Retired	168,621,112,530	153,102,989,468	167,372,712,364	166,925,089,084
CPI Rate	0.561	0.572	0.58	0.61

表 3: 串并行算法 CPI 对比

由图4.6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。

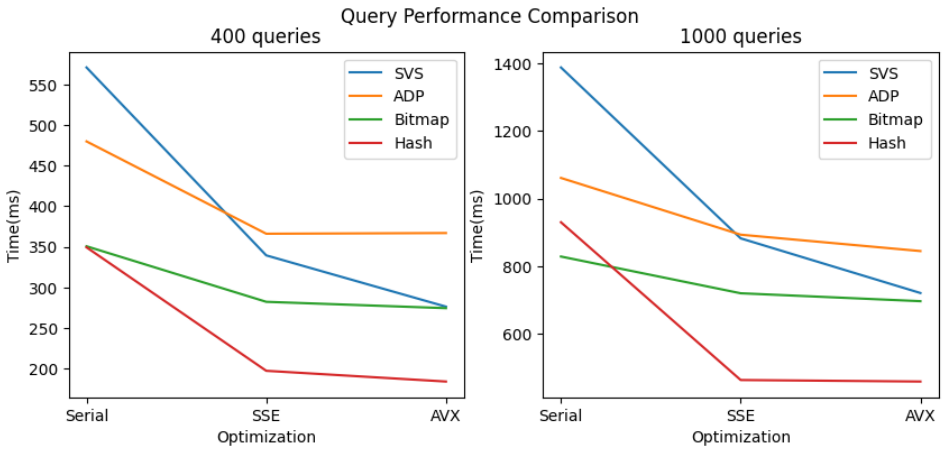

图 4.6: x86 下串行与 SSE、AVX 并行对比

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 4: 并行算法加速比

4.4 arm 与 x86 对比

对比 arm 平台和 x86 平台四个串行算法执行效率。由图4.5可以看到，除了 BitMap 算法，对于不同规模的测试数据集，其余算法均是 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。

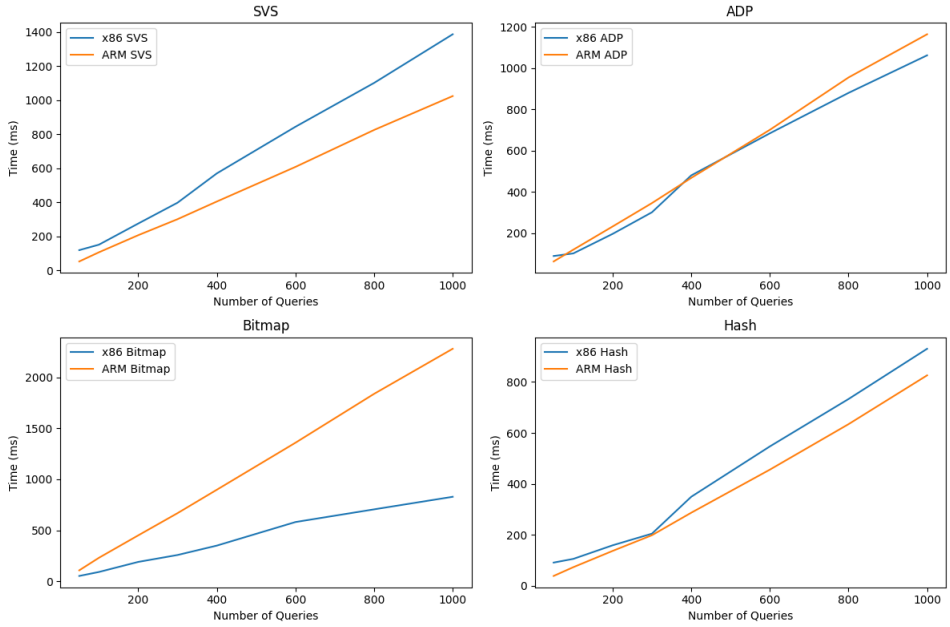

图 4.7: arm 与 x86 横向对比

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。这在图中可以看出，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。

5 实验总结

本次实验对于倒排索引求交这个问题，我们实现了 SVS、ADP、Hash、BitMap 四个算法，其中 SVS 融入了拉链法，BitMap 融入了跳表法，再用 NEON、SSE、AVX 指令集分别对各个算法做了并行的版本，分别测量了各算法的性能表现，进行了串行纵向对比，串并行横向对比，以及 arm x86 对比，另外还讨论了对齐与不对齐对算法的性能影响。

实验结果表明，Hash 优化有节制的用空间换时间使得算法在各个版本下性能表现最优；SVS 算法虽然串行版本表现稍差，由于其一次比较一条完整的列表，并行化一次比较多个元素对其性能提升明显；对于对齐与不对齐的比较，由于集合求交算法的数据有一定的重复性和局部性，于是不对齐的性能表现并不弱于对齐版本。

通过本次实验，我更清晰地理解了 SIMD 并行的机制，并学会使用 NEON/SSE/AVX 指令集写并行代码。但由于时间原因，指令学习稍浅，只学习了需要用到的部分 SIMD 指令，没有对更多的指令进行尝试，在后续的学习过程还需要继续拓展知识面。

另外在实验过程中，鲲鹏服务器的使用是一个很大的困难，为了能够在其上成功运行，反复咨询了助教，花费了很多的时间，但对其理解仍然片面，后续还需要更进一步学习。

参考文献

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