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
PThread&OpenMP

姓名：唐明昊 朱世豪

学号：2113927 2113713

专业：计算机科学与技术

2023 年 5 月 14 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 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。

1.3 实验环境

本实验对 arm，x86 两种架构均做了实验测试，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

1.4 实验分工

本小组由唐明昊和朱世豪组成。唐明昊同学负责 SVS 算法和 ADP 算法的并行化实现，朱世豪同学负责对 BitMap 算法进行调优，并进行了相应的并行化实现。针对 query 内表现不佳的问题，唐明昊同学提出加速大概率事件，SVS 算法 barrier 优化等策略，朱世豪同学提出 ADP 算法任务池优化策略。另外实验数据的测量以及所有算法代码的封装主要由朱世豪同学完成。附唐明昊和朱世豪二人 GitHub 仓库链接。

1.5 实验概述

本次多线程实验分别从 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。普通算法 SVS 与 ADP 在是实现过程中用到了拉链法的思想，具有**提前结束**的特性，而分段并行处理后，前面元素做的工作无法为后面元素服务，会导致负载不均，即处理后面部分的线程工作量较大，导致最终算法执行效率低下。为了解决此问题，利用了加速大概率事件，多重循环 barrier，任务池等手段进行加速优化。

与 SVS 和 ADP 不同，我们的补充算法 Hash 和 BitMap 使用空间换时间，在实现逻辑上与前两者不同，不会产生数据依赖与负载不均等问题，query 内并行仍然有效果。

值得一提的是，在对 BitMap 算法进行 query 内多线程并行化的尝试中，使用了多种方法（动态，静态，任务池）而提升效果未达到预期，于是对串行算法本身进行优化，使得串行算法能够更好的并行。在优化时，发现能够节省大量数据复制的开销，从而大幅度优化串行算法，并在此基础上进一步编写相应的多线程优化算法。详情见2.4.2节。无特殊说明，在本文中使用的 BitMap 算法均为优化后的算法。

2 PThread

2.1 query 间并行

query 间并行是指主线程将大量 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的策略。根据主线程将特定 query 分配给子线程的位置不同，又可以将 query 间并行分为动态 query 间并行与静态 query 间并行两种。

2.1.1 动态实现

定义与分析 query 间的动态并行是指在多个请求被发出时，开辟动态线程，将多个 query 任务进行划分后分配给不同子线程的策略。这种方法开辟线程的过程在请求发出之后，并且所有子线程执行各自的任务后都会被销毁，等待下一次的查询请求到来重新分配工作、开辟相应的子线程对其进行处理。

query 间动态并行策略的优点是子线程**随开随用，实现简单**，并且由于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**，造成资源的浪费。该方法的缺点也很明显，也就是当**请求拥挤**时，反复的开辟线程与销毁线程，造成了额外的时间开销。

方法实现 整个程序会反复测试不同组规模的数据，每次选定一定规模的数据以后，开辟线程，为每个线程分配一定量的 query，调用相应的算法同步执行。

query 间动态并行

```
int skip = k / NUM_THREADS;
threadParamOut paramOut[NUM_THREADS];
for (int i = 0; i < NUM_THREADS; i++) // 创建线程，传入线程id以及处理的总query数
{
    paramOut[i].id = i;
    paramOut[i].start = i*skip;
    paramOut[i].end = i + 1 == NUM_THREADS ? k : (i + 1) * skip;
    paramOut[i].f = f;
    pthread_create(&thread[i], NULL, dynamicForTest, &paramOut[i]);
}
for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(thread[i], NULL);
```

测试结果 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。各个算法加速比如表1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。

Events	SVS	ADP	Hash	BitMap
400 规模加速比	2.34	2.81	2.65	2.77
400 规模加速效率	0.58	0.70	0.66	0.69
1000 规模加速比	2.51	2.54	2.38	2.85
1000 规模加速效率	0.63	0.64	0.60	0.71

表 1: query 间动态加速比

算法实际执行时间对比图2.1如下。

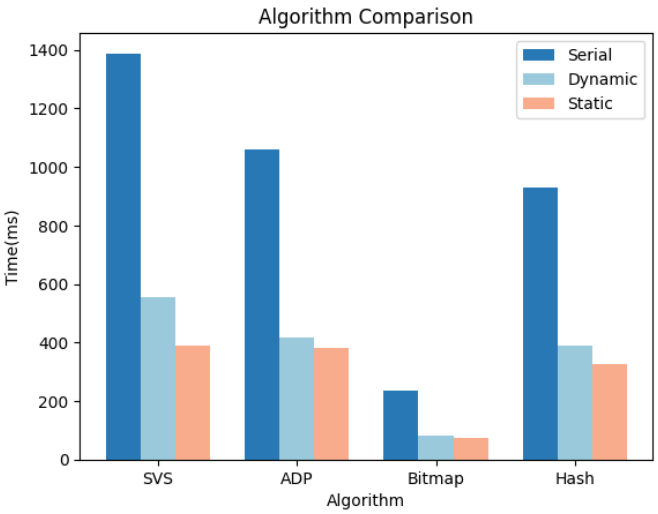

图 2.1: query 间并行加速效果

结合上 SIMD，多线程算法同样能够两倍以上的加速比。各个算法表现如图2.2所示。

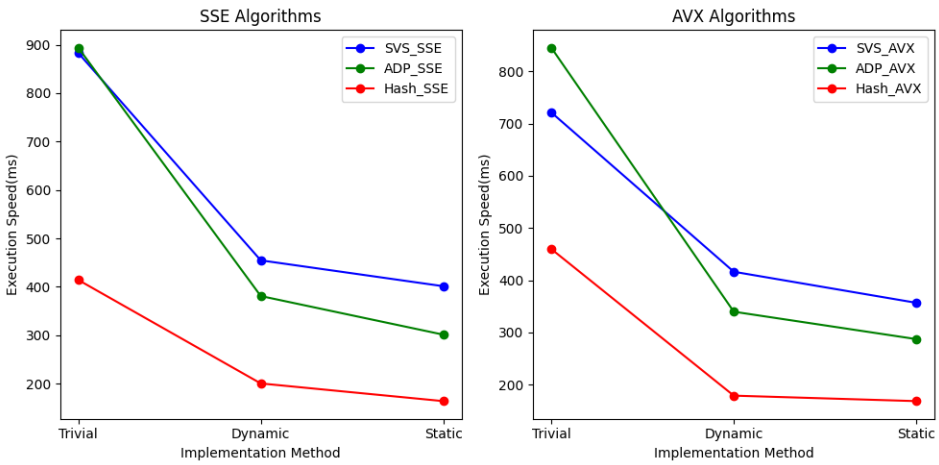

图 2.2: query 间并行 SIMD 加速效果

query 间并行加速并未达到理想的三倍以上，其原因可能是**负载不均**：请求的任务个数相同，而 query 内的工作量的大小并不相同。例如一个 query 的倒排链表个数可能不同，根据数据集的特点，我们所用的数据集中的单个 query 内部倒排链表数在 1 至 5 个之间不等，这可能会导致任务分配不均匀。针对这一问题，我们将在使用 omp 进行并行优化时使用**任务池**策略对此展开进一步探究。

2.1.2 静态实现

定义与分析 query 间的静态并行是指在多个请求被发出之前，开辟静态线程，等待任务的到来，当请求到来时才将任务分配给子线程的策略。这种方法在查询请求被提交时，唤醒已经开辟好的子线程，将这些 query 分配给不同子线程，当子线程完成自己各自的工作后，进入空闲等待状态而不被销毁，只有当总控制信号发出，指出所有静态线程停止工作时，才被销毁。

query 间静态并行策略的优点是子线程**随时取用，无需开辟**，这是由于每次子线程完成自己的任务后并未被销毁，而是进入空闲等待状态，以便于之后的随时取用。这样，相比动态线程，静态线程

能**节省开辟线程、销毁线程的时间开销**，在请求拥挤时，能够节省大量时间。但由于子线程在执行完任务后未被销毁，会造成资源的浪费。

方法实现 在测试数据之前，开辟线程。线程没有任务时处于睡眠状态，主线程获取到需要测试的数据规模并划分后，**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
}
```

测试结果 query 间静态相较于动态，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，由图2.1也可以看出，静态方法较动态有着或多或少的提升。未能达到极致的三倍以上加速比原因如前所述，不再赘述。

Events	SVS	ADP	Hash	BitMap
400 规模加速比	2.48	3.043	2.69	2.96
400 规模加速效率	0.62	0.76	0.67	0.74
1000 规模加速比	2.35	2.78	2.840	3.13
1000 规模加速效率	0.58	0.69	0.71	0.78

表 2: query 间静态加速比

query 间静态各算法执行时间比较如前图2.1与图2.2所示，由图可以直观地看出静态的加速效果以及其与动态的对比。

值得一提的是，静态相较于动态几乎可以说是全线领先，而在 Hash 算法的 avx 的并行化加速对比上，二者保持了一致，可能是算法已经探到了执行效率的上界。实际表现上，1000 个 query 任务重复查询 5 遍取均值，二者都将执行时间压缩到了 200ms 以内。

2.2 query 内动态并行

query 内意为对一个 query 中的几个倒排链表求交过程进行多线程并行化，动态即为动态得创建销毁线程。

2.2.1 SVS 算法

并行实现 算法动态开辟线程，每个线程取出当前链表的一部分（length/NUM_THREADS）与当前 query 下的其余几个链表进行求交。找到匹配的元素后，向后移动全局变量 countForSVS，放入当前元素，该过程需要使用互斥锁来进行同步。

SVS_dynamic

```
// s列表中的每个元素都拿出来比较
for (; begin < end; begin++) // 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 < (invertedLists)[i].docIdList.size(); t++) {
        // 遍历i列表中所有元素
        Some Codes Here...
    }
    if (isFind) {
        // 覆盖
        pthread_mutex_lock(&mutex);
        S_in_SVS.docIdList[countForSVS++] = S_in_SVS.docIdList[begin];
        pthread_mutex_unlock(&mutex);
    }
}
```

但由于 SVS 串行算法为了提高表现，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。

使用**加速大概率事件**的思想进行优化：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。

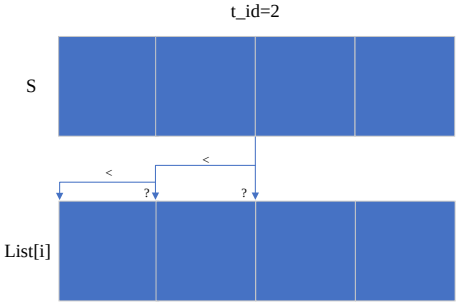

图 2.3: 加速大概率

加速大概率预处理

```
// 预处理加速大概率事件
t = t_id * ((invertedLists)[i].docIdList.size() / NUM_THREADS);
for (int j = t_id; j > 0; j--)
{
    if (S_in_SVS.docIdList[begin] < (invertedLists)[i].docIdList[t])
        t -= ((invertedLists)[i].docIdList.size() / NUM_THREADS);
    else break;
}
```

除了这种一段段回退的优化，还尝试过每次回退半段，二分查找等预处理手段，经实测效果大致相同。

测试结果 即使运用了加速大概率事件的手段，但由于 SVS 算法特性，仍然很难做到负载均衡。再加上动态线程反复开辟销毁线程额外开销，导致最终算法执行低效。

如下图2.4所示，并行后速度低于串行版本，随着规模增大，差距逐渐变大。原因正是前面提到，失去了提前结束的优势后，**规模越大，线程负载就愈发不均衡**。

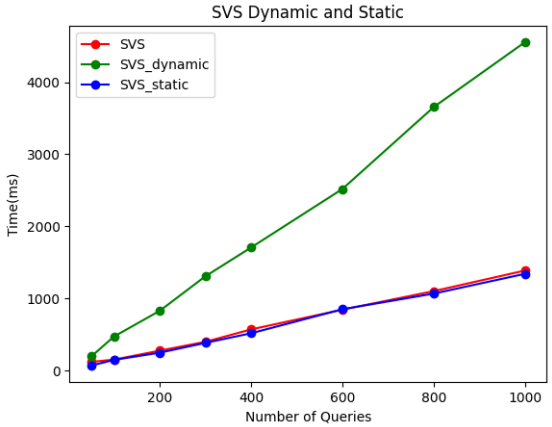

图 2.4: query 内 SVS

2.2.2 ADP 算法

并行实现 算法动态开辟线程，每个线程取出当前链表的一部分（length/NUM_THREADS）与当前 query 下的其余几个链表进行求交。找到匹配的元素后，将当前元素放入结果倒排链表 S_in_ADP 中，该过程需要使用互斥锁来进行同步。

ADP_dynamic

```
for (; begin < end; begin++) {
    bool isFind = true;
    unsigned int e = (invertedLists)[list[0].key].docIdList[begin];
    for (int s = 1; s != num; s++) {
        isFind = false;
        // 遍历检查s列表元素
        Some Codes Here...
        if (!isFind)break;
```

```
    }
    // 当前元素已被访问过
    if (isFind) {
        pthread_mutex_lock(&mutex);
        S_in_ADP.docIdList.push_back(e);
        pthread_mutex_unlock(&mutex);
    }
}
```

ADP 算法同 SVS 算法一样拥有提前结束机制，即当发现当前比较的 docID 不匹配时，视为比较的 docID 求交失败，移动该表的指针，从而节省下一元素在该表比较所花费的时间。而当多线程同时对倒排链表进行求交时，一个子线程无法有效地对其他子线程所保存的各表的指针位置信息进行利用。

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。这将导致分配到要对链表后 1/2 元素进行匹配的线程开销与原算法近似，性能较低。

因此，对应 ADP 算法我们也采用了类似于 SVS 算法加速大概率事件的策略进行优化，实际测试结果与 SVS 算法的优化程度近似相同。

测试结果 测试结果如图2.5所示，与预期结果相同。

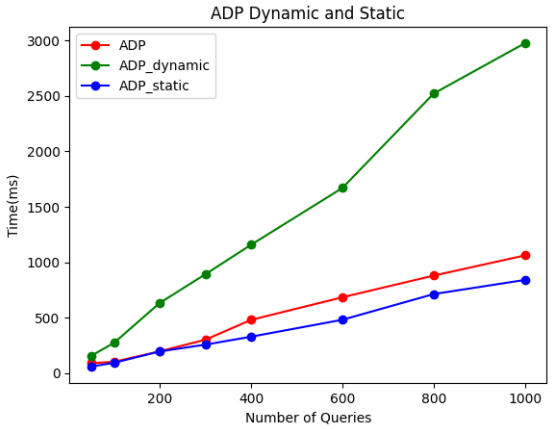

图 2.5: query 内 ADP

2.2.3 BitMap 算法

BitMap 的串行算法在2.4.2节中已经对串行算法进行了优化，得到了更好的串行算法速率。下面继续基于该优化算法进行多线程并行化。

并行实现 通过2.4.2节中的分析可知，BitMap 中耗时占比最大的实际为 BitMap 结构体的赋值过程（见2.4.2节代码段），而不在于核心计算代码，我们进行多线程并行化的重心应该放在优化 BitMap 的赋值过程，而不在于核心计算代码。实际上经过测量，一条赋值语句语句带来的开销是计算开销的 **9 倍**之多。

故采用**分段赋值**的方法对该赋值语句进行并行优化，每个子线程分段初始化 chosenList。

BitMap_dynamic(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);
```

测试结果 测试结果如图2.6所示。

由于动态开辟线程、销毁线程次数多，而 BitMap 算法本身执行时间很短，一次开辟的开销就可对算法本身造成不可忽视的影响，所以动态多线程效果差。

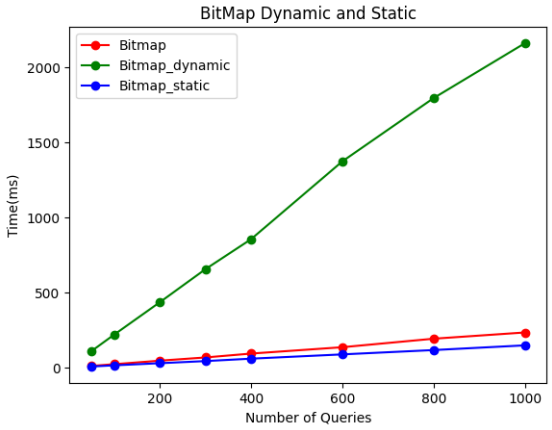

图 2.6: query 内 BitMap

2.3 query 内静态并行

2.3.1 SVS 算法

并行实现 静态并行与 query 间的实现类似，SVS 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。其余代码细节与动态保持一致。

SVS_static-工作线程

```
while (true) {// 静态多线程，一直工作
    sem_wait(&sem_worker[t_id]);// 阻塞，等待主线程唤醒
    if (staticFlag == false)// 所有任务完成，退出
        break;
    // 每次唤醒后获取工作范围
```

```
    int length = S_in_SVS.docIdList.size() / NUM_THREADS;
    Some Codes Here...
    if (length == 0)
        goto Out_SVS_static;
    // 预处理加速大概率事件
    Some Codes Here...
    // s列表中的每个元素都拿出来比较
    Some Codes Here...
Out_SVS_static:
    sem_post(&sem_main);// 唤醒主线程
}
```

测试结果 由图2.4可以看出，SVS 静态并行少去了动态开辟销毁线程的额外开销后，算法执行效率明显提高。但还是如前所述的算法特性原因，相比串行版本执行时间只是略有领先，在规模较大时，线程负载不均，差距逐渐变小。

2.3.2 ADP 算法

并行实现 主体架构与 SVS 保持一致，策略是由信号量来唤醒工作线程，将工作内容改为使用 ADP 算法求交。具体代码可参见 GitHub 仓库。

测试结果 如图2.5所示，静态线程的测试结果相较于普通串行算法只是略好，仍然是由于算法提前结束的特性造成。

2.3.3 BitMap 算法

并行实现 主体架构与其他静态算法保持一致，主线程使用信号量来唤醒工作线程。只是子线程执行的任务是是拷贝复制操作。具体代码可参见 GitHub 仓库。

测试结果 如图2.6所示，静态线程的测试结果相较于普通串行算法效率没有较大区别。原因可能是静态算法虽然开辟，销毁线程次数相比于动态版本较少，带来的开销较小，但是由于 BitMap 串行效率很高，这部分开销相较于其他算法已经不能忽略不计，另外，由于子线程调用的是 copy 函数对于 vector 进行拷贝，而 copy 函数对于不同长度的向量其相较于普通拷贝算法的优化力度不同，因为对于不同长度的向量，copy 函数会选择不同的拷贝方法，所以也就导致了测试结果与理论结果之间的差异。

2.4 query 内并行问题探究

2.4.1 不同线程数影响

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。

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）

而动态并行版本如图所示，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。

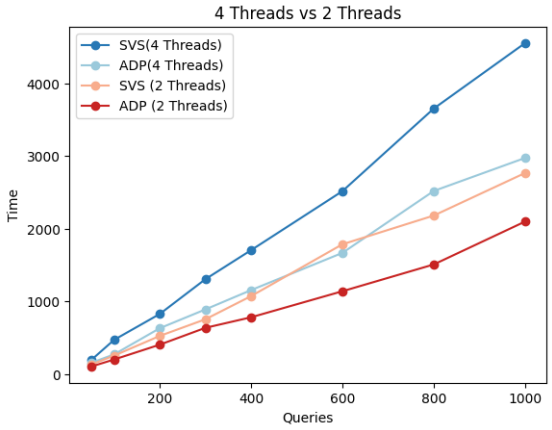

图 2.7: Query 内静态对比

2.4.2 BitMap 算法优化

尝试对于 BitMap 的三次与操作（二级索引，一级索引，文档 ID）进行多线程并行化，将二级索引的求交任务进行划分并派发给子线程来完成，子线程依次根据二级索引的求交结果来在一级索引和文档实际的 ID 进行求交。

由于二级索引的划分是不相交的，故根据二级索引找到的一级索引和文档实际 ID 也是不相交的，故该方法能保持数据不发生冲突。但对该方法进行测试，实际的优化效果较差。可见算法的优化力度

Queries	**BitMap**	**Static BitMap**	**New BitMap**
100	258.103	109.967	22.924
200	481.53	203.656	46.418
400	640.245	357.154	93.7959
800	1247.48	724.187	192.83
1000	1374.88	843.966	234.325

表 4: BitMap 的串行、静态和更新算法对比（单位：ms）

没有达到预期的四倍，我怀疑是算法的负载不均衡，于是设计了相应的任务池算法，但依旧没有达到预期效果。

于是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：

BitMap(Excerpt)

```
for (int i = 0; i < num; i++)
    bitmapQLists[i] = bitmapList[list[i]];//建立id对应bitmap
chosenList = bitmapQLists[0]; //选中的列表
```

而不在于算法进行与运算的核心部分。

对上述代码片段进行分析，可以发现，bitmapQLists 的对于 bitmapList 的复制并不是必需的，实际上这一步仅仅是为了算法之后便于利用 bitmapList 中的倒排索引。于是可以将这一步循环删去，将算法以后出现的 bitmapQLists[i] 改为 bitmapList[list[i]]，以减少赋值时带来的额外时间开销。

在表4中展现了优化的结果，可见这一改变对于程序性能的优化程度可达到 6 至 19 倍，优化很成功。这是 BitMap 的空间占用很大的缘由，一个二级索引的长度大约是 770，则 docIdList 的大小可达大约 3153920 字节，赋值拷贝时的时间代价很大，该优化省去了这一时间代价。

2.4.3 SVS 算法 barrier 优化

算法实现 原始 SVS 并行算法需要遍历 num 个链表，遍历一个即需要进入一次工作线程，工作线程结束后返回 SVS 主线程做一次删除操作后再进行循环。

但实际上外一层循环并不涉及繁杂的计算，于是考虑将**多层循环全部纳入工作线程**，只对内层做工作拆分。外层循环完成一次迭代后，使用 **barrier 进行线程间同步**，由一个线程来完成删除操作。

SVS_dynamic_barrier

```
// 与剩余列表求交
for (int i = 1; i < num; i++) {
    countForSVS = 0;// s从头往后遍历一遍
    // 获取工作范围
    int length = S_in_SVS.docIdList.size() / 4;
    Some Codes Here...
    if (length == 0)
        goto Out_SVS_dynamic;
    // 预处理加速大概率事件
    Some Codes Here...
    // s列表中的每个元素都拿出来比较
    Some Codes Here...
    // barrier
    if (t_id == 0) {
        // 线程0做删除操作，但需要等待其他线程做完
        for (int l = 1; l < NUM_THREADS; l++)
            sem_wait(&sem_barrier);
        if (countForSVS < S_in_SVS.docIdList.size())// 最后才做删除
            S_in_SVS.docIdList.erase(S_in_SVS.docIdList.begin() + countForSVS,
                S_in_SVS.docIdList.end());
    }
    else sem_post(&sem_barrier);
    pthread_barrier_wait(&barrier_wait);// 所有线程一起进入下一轮循环
}
```

测试结果 SVS 的动态并行算法在将多次循环全部纳入工作线程后，减少了开辟销毁线程的次数，只需要在工作线程内进行同步操作。由图2.8可以看出，在规模逐渐变大时，使用了 barrier 优化的 SVS 算法优势明显。

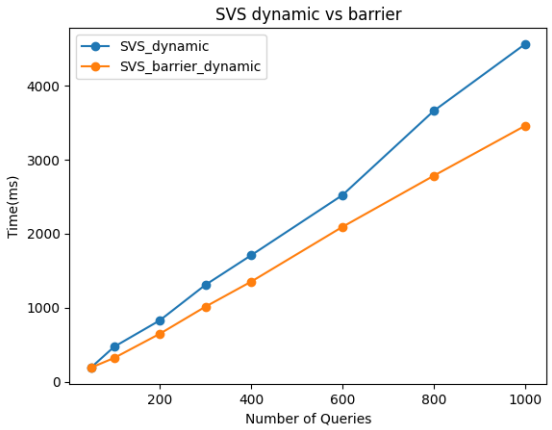

图 2.8: SVS_barrier

2.4.4 ADP 算法任务池优化

算法实现 为解决 ADP 算法执行时负载不均衡的问题，对 ADP 算法的任务划分进行了任务池优化，尝试多种划分方法，保证各个线程的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。

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如下几种（链表长度为 L）：

1. 按固定步长 300 划分任务。
2. 按照固定 300 块任务划分任务。
3. 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。

实际运行结果表明第三种任务划分方式最好。代码如下所示，仅列出要点，详细代码见 GitHub 仓库。

ADP_TaskPool

```
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++;
```

```
10      pthread_mutex_unlock(&mutex);
11      if (begin > list[0].length)//已经做完所有任务->报告
12          break;
13      //用Adp做求交工作
14      ADP-work codes...
15  }
16  sem_post(&sem_main);// 唤醒主线程，报告所有线程已经完成
```

测试结果 实验测试的结果如图2.9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。

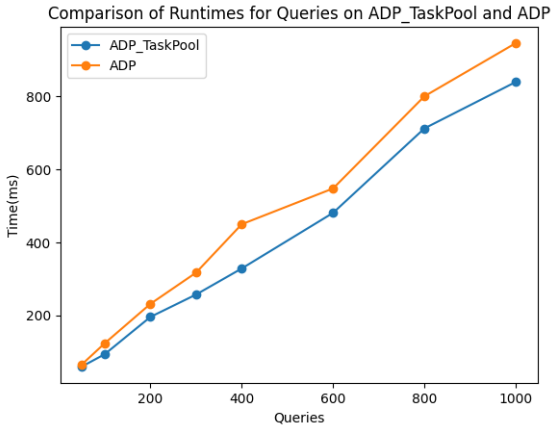

图 2.9: query 内静态原算法与任务池对比

2.5 结果分析

2.5.1 动态与静态对比

无论是 query 间并行还是 query 内并行，**静态的执行效率均高于动态版本**。

因为动态版本开辟销毁线程的开销过于严重，而静态版本占用更多的程序资源，开辟线程后只需要用信号量进行唤醒与休眠。

query 间是在对程序的外层循环做循环划分，动态开辟销毁线程的次数相对较少，故相对静态并行差距并不悬殊，平均动态版本较静态慢 10%。参考图2.2与图2.1。

而 query 内因为**循环嵌套深入**，动态开辟销毁线程的次数很多，由前面 query 内算法的分析以及图2.4与图2.5均可看出，两个版本性能差异巨大。

2.5.2 x86 与 arm 对比

比较 arm 平台和 x86 平台算法执行效率。

图2.10为 query 间各算法执行效率对比。可以看出，除 BitMap 算法以外，其余算法在 arm 平台均有着更好的表现，这可能是由于 arm 其优秀的架构和简单的指令集等。而 BitMap 算法表现差可能是由于其开辟了大量空间，而 arm 平台鲲鹏服务器 cache 较小，造成大量的 cache miss，影响了性能表现。

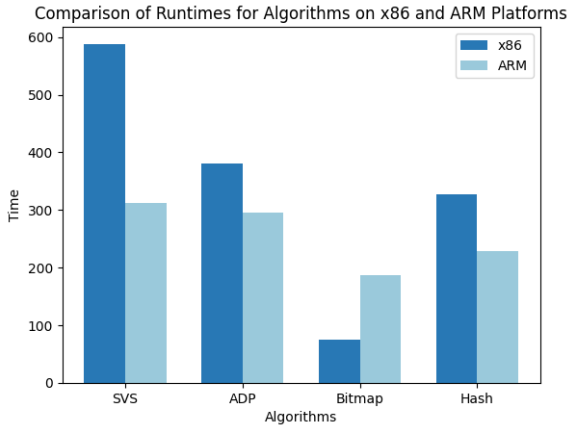

图 2.10: query 间并行对比

图2.11为 SVS 与 ADP 算法 query 内并行在两平台性能表现对比，x86 与 arm 各有领先。

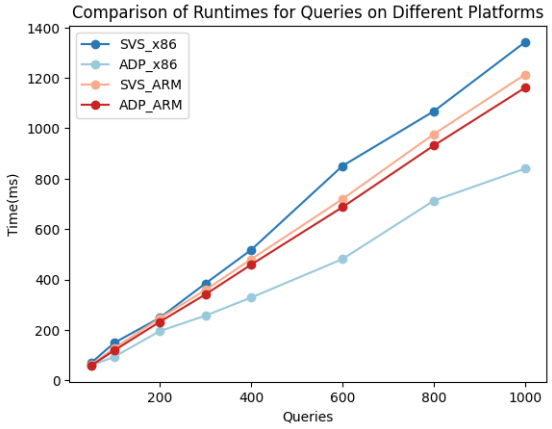

图 2.11: query 内并行对比

2.5.3 profiling

使用 VTune 对各线程进行 profiling，具体实例如下。

query 间 SVS 图2.12为 query 间 SVS 的分析结果。可以看出一共开辟了 5 个线程，主线程在读取完文件后进入睡眠状态，其余 4 个工作线程开始运行。从这个工作线程的 Running 以及 CPU Time 可以看出四个线程负载较为均匀，但也存在有一个线程工作量大，工作时间稍长的现象。问题主要出在数据集 query 的分布上，具体分析见3.3节。

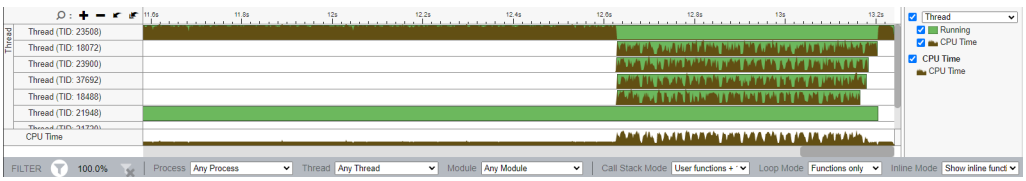

图 2.12: query 间 SVS 算法 VTune 分析结果

query 内 ADP_dynamic 图2.13为 query 内 ADP 动态算法使用 VTune 的分析结果，可以看到动态算法开辟了很多线程，而且各个线程之间负载并不均衡。从图中 Running 和 CPU Time 可以看出各个线程的 CPU 空闲很多，导致运行效率差，运行速度慢。

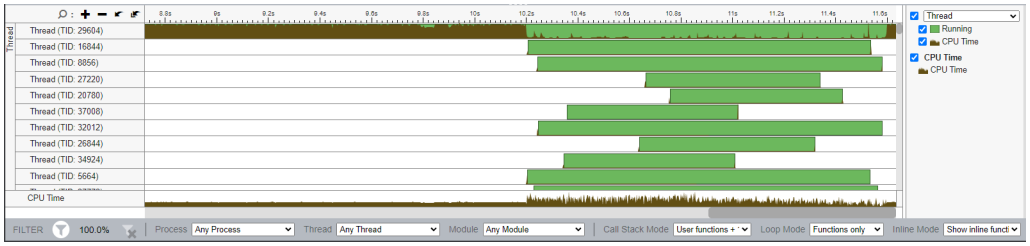

图 2.13: query 内 ADP 动态算法 VTune 分析结果

query 内 ADP_static 图2.14为 query 内 ADP 静态算法使用 VTune 的分析结果，可以看到各个线程的起始工作时间和结束工作时间大概相同，这是由于静态算法有同步的机制。但是，各个线程的工作量依然不平衡，因为从图中的 CPU Time 可知 T_id 为 25064 的工作量比其他线程略大，这主要是由 ADP 算法提前结束的特点导致的。

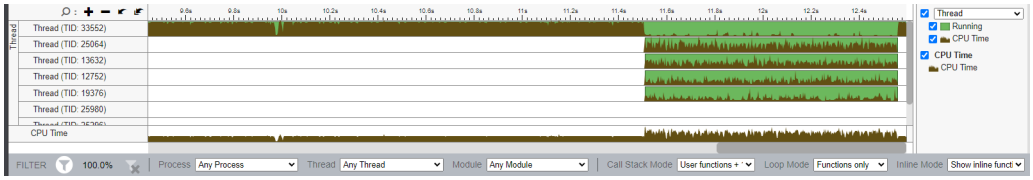

图 2.14: query 内 ADP 静态算法 VTune 分析结果

3 OpenMP

3.1 query 间并行

并行实现 使用 OpenMP 进行 query 间并行化，总体思路与 PThread 保持一致，在主线程将大量线程平均分配给多个子线程，子线程直接调用对应的求交算法。

OutQuery_OMP

```
#pragma omp parallel if(isParallel_out),num_threads(NUM_THREADS)
#pragma omp for
    for (int i = 0; i < k; i++) {// k个查询
        int num = 0;// 一个query查询的项数
        for (int j = 0; j < 5; j++)
            if (query[i][j] != 0)
                num++;
        int* list = new int[num];// 要传入的list
        for (int j = 0; j < num; j++)
            list[j] = query[i][j];
        sorted(list, (invertedLists), num);// 按长度排序
        f(list, invertedLists, num);// 根据选择调用函数
    }
```

可以看出，相比与 PThread 手动开辟线程、销毁线程，另外还需要单独写线程函数运算，OpenMP 代码实现上大幅简洁，大大减轻了工作量。

测试结果 query 间并行性质良好，**任务间没有依赖关系**，加速效果明显，除 Hash 算法外，均能达到 2.5 倍以上的加速比。

Events	SVS	ADP	Hash	BitMap
400 规模加速比	2.69	2.98	2.09	2.93
400 规模加速效率	0.67	0.75	0.52	0.73
1000 规模加速比	2.30	2.69	2.14	3.02
1000 规模加速效率	0.57	0.67	0.54	0.76

表 5: query 间加速比

算法实际执行时间对比图3.15如下。

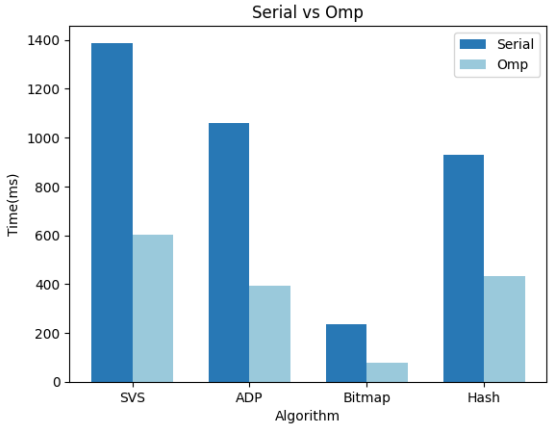

图 3.15: query 间并行对比

引入 SIMD，query 间并行同样取得了良好的效果，各个算法相比普通的 SIMD 算法加速比也能达到 2.5 倍左右。

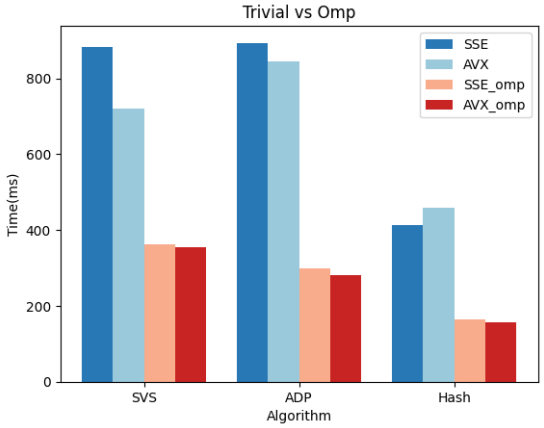

图 3.16: query 间并行 SIMD 加速效果

3.2 query 内并行

并行实现 使用 OpenMP 进行 query 内并行化，总体思路与 PThread 保持一致，利用在主函数中利用 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。

以 PThread 中没有实现的 Hash 算法为例。

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
    count = 0;
    int length = s.docIdList.size();
#pragma omp for
    for (int j = 0; j < length; j++) {
      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;
      // 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());
  }
```

测试结果 测试结果如图3.17所示，SVS 与 ADP 依然与 PThread 一致，由于这两个算法特性，并行化容易造成负载不均，所以执行效率低下。BitMap 同样与前保持一致，不再赘述。

而 Hash 算法虽然基于 SVS 算法改良，但没有了提前结束的依赖，每个线程直接去**访问 hash 桶**即可，很适合 query 内并行化，最终加速效果明显。

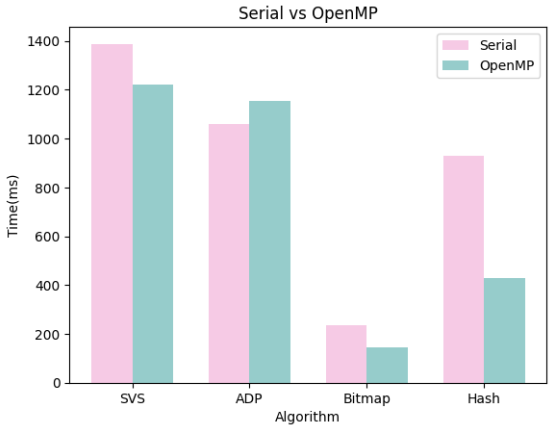

图 3.17: query 内并行对比

3.3 任务池尝试

在前述 query 间并行中，虽然算法执行高效，但并未达到 3 倍以上的加速比，推测可能是由于**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3.18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**。

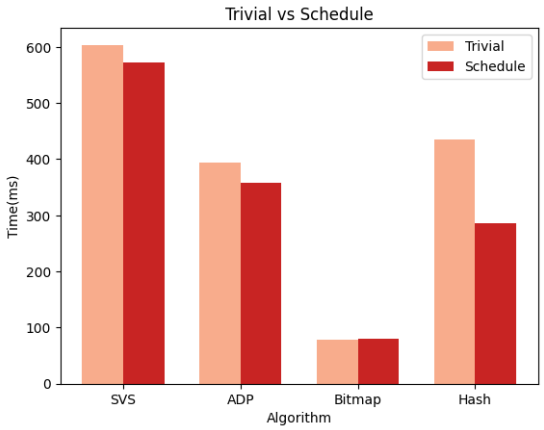

图 3.18: query 间任务池优化

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。

3.4 结果分析

3.4.1 x86 与 arm 对比

由图3.19对于 Query 间的 omp 优化进行对比，可以发现：

- BitMap 算法 ARM 平台较 x86 慢,这可能是由于 BitMap 开辟了大量的空间以储存文档的 docID，而 ARM 平台鲲鹏服务器的 cache 较小，这会导致 BitMap 结构不能完全装入 cache 之中，从而导致了大量的 cache miss，影响了算法性能的表现。
- 其他算法 ARM 平台较 x86 平台快，这可能得益于 ARM 平台优秀的架构和简单的指令集等，从而使得 ARM 平台更适合于多线程并行化。

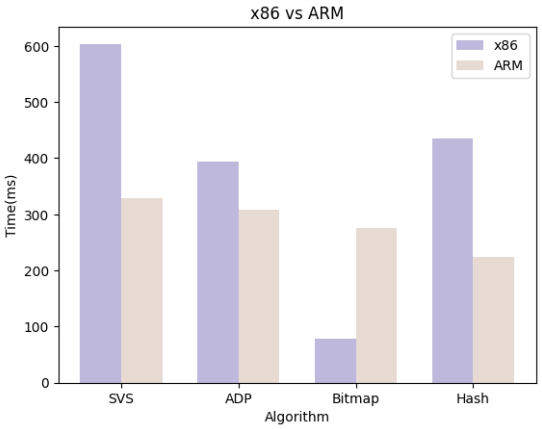

图 3.19: query 间 omp x86 与 arm 对比

3.4.2 profiling

query 间 SVS 由图3.20可以看出，OpenMP 只开辟了四个线程，充分运用上了主线程，**并没有让主线程空闲等待**。算法工作阶段，四个线程负载相对均衡，而且 OpenMP **自动实现同步操作**，工作线程同时完成任务退出，主线程完成收尾。

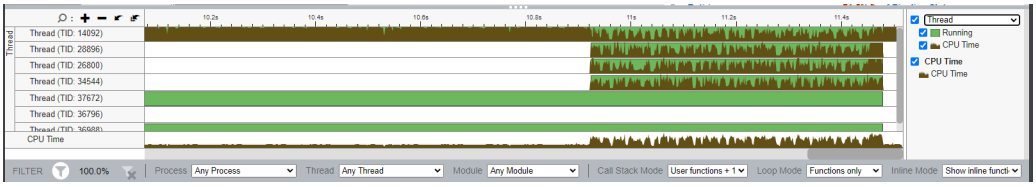

图 3.20: query 间 SVS 算法 VTune 分析结果

query 内 SVS 图3.21展示了 query 内 SVS 的分析结果。可以看出，OpenMP 会对主线程进行复用；另外尽管在 query 内，OpenMP 并不会动态开辟许多线程，而是类似于静态开辟线程，**一次开辟反复使用**，减少了额外开销；经优化以后，四个工作线程被**充分利用**，不会有空闲等待浪费资源。

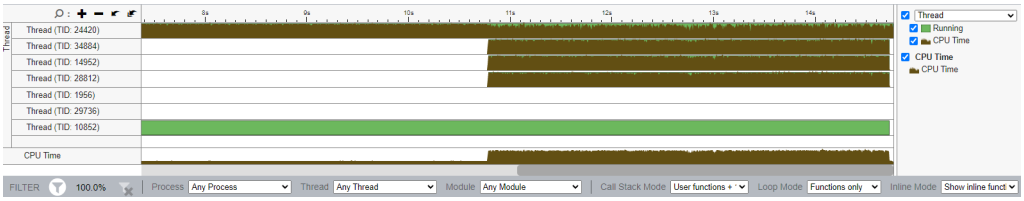

图 3.21: query 内 SVS 算法 VTune 分析结果

3.4.3 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	684.754	473.01	394.51
加速比	1	1.55	2.24	2.69
效率	1	0.77	0.75	0.67

表 6: 资源总量变化时的效率变化

从表6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。这意味着随着线程数的增加，算法的执行时间在减少，但减少的比例逐渐降低。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**。

从效率的计算结果可以看出，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 PThread vs OpenMP

query 间并行优化对比 由于使用 OpenMP 时开辟线程的位置与 query 间的动态算法相同，故将其与 query 间动态算法比较。

由图4.22可知, 对于 query 间的并行优化的相同算法, OpenMP 所带来的并行优化收益与 PThread 大致相同。

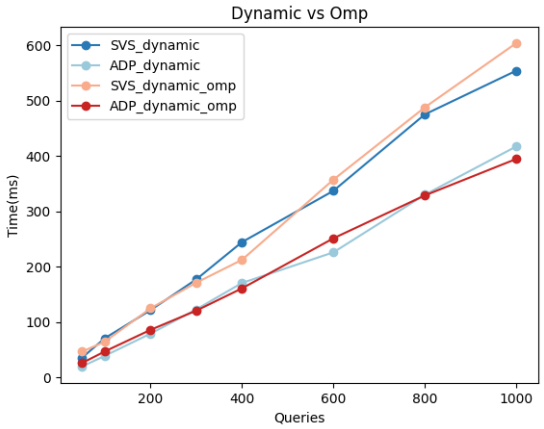

图 4.22: query 间动态 omp 与 Pthread 对比

query 内并行优化对比 由图4.23可知，对于 query 内的并行优化的相同算法，OpenMP 所带来的并行优化收益与 PThread 的静态算法大致相同。这与预期不同，因为在使用 openMP 开辟子线程时开辟的位置是内部循环，而非静态线程，创建、销毁线程的开销应与 PThread 的动态方法相一致，也就是运行时间应与其保持一致。经过实验得知，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。

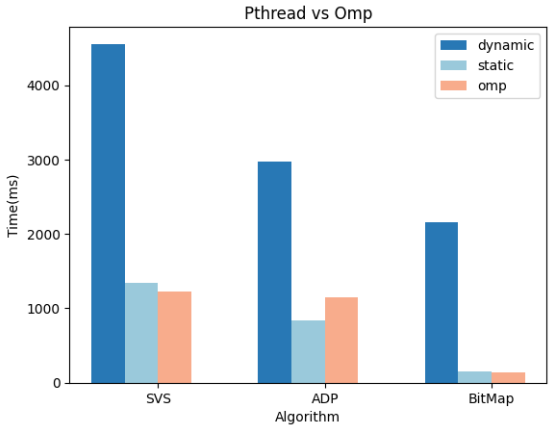

图 4.23: query 内 omp 与 PThread 对比

总结 总体上 OpenMP 与 PThread 对于串行算法的优化力度大约相同，但是 OpenMP 的 api 相对于 PThread 的封装性更好，更便于使用，使用几行代码即可完成 PThread 几十行代码才能完成的工作，而且对于不同的串行算法 PThread 还需要为他们各自编写不同的工作线程函数，而 OpenMP 则仅需在原串行算法上做微小的调整即可。故总体上 **OpenMP 相较 PThread 更简单易用**，而 PThread 则更接近底层，适合对算法的并行化做更加细致的调整。

另外由3.4.2节可知，OpenMP 对于多个线程的调度较 PThread 还有额外的优化，也就是**让空闲下来的主线程也去执行任务**，这样，主线程就可以少开辟一个子线程，减少了开辟线程的时间开销，而 PThread 要实现这一机制较为困难，造成了主线程的资源浪费。而 OpenMP 对于 query 内算法的的优化，既有 PThread 动态并行节省资源的优点，又达到了静态并行的执行效率。

5 实验总结

在本次 PThread&OpenMP 实验中，我们对四个算法分别做了 query 间并行与 query 内并行，在做 PThread 实验时又细分了静态并行与动态并行。

从实验结果来看，无论是 PThread 还是 OpenMP，query 间并行都能取得良好的效果。为了让其效果更好，我们还做了任务池的尝试，减少数据分布对程序的影响。

而 query 内并行由于负载不均的原因，只有当静态开辟线程时，才能取得一定的优化效果，动态版本下，线程开辟销毁浪费了大量时间。为了让 query 内并行有更好的表现，我们也尝试了许多优化的手段，也确取得了一定的作用。

通过本次实验，我们熟练掌握了 PThread 的运用，熟悉了 PThread 的如互斥锁，信号量，barrier 等各种操作。同时在用 OpenMP 并行化实验后，我们对二者进行比对，并通过 VTune 进行 profiling。认识到 OpenMP 相比于 PThread 简单易用，且拥有很好的优化机制，能够高效地利用线程。回顾整个实验过程，PThread 投入大量时间去实现与优化，而 OpenMP 只需要短短几行代码就能实现一样的效果，同时兼顾我们 PThread 各个版本的优点。在往后的学习过程中，我们会更多地探索 OpenMP 的使用，挖掘其更多价值。