南開大學
Nankai University

计算机学院

并行程序设计期末报告

并行加速的倒排索引求交算法

姓名：王俊杰 刘家骥

学号：2212323 2211437

专业：计算机科学与技术

2025 年 3 月 23 日

目录

1 相关信息说明

本报告是由刘家骥（2211437）和王俊杰（2212323）共同完成的并行期末研究报告。

本研究的代码已上传至 Github，网址为 https://github.com/2212323/ParallelFinalProject。

本报告系算法研究类报告，选题来源于《期末研究报告选题示例》中的“选题示例三——倒排索引求交（算法研究类）”。

1.1 作业要求

- 在章节3中，我们对此问题前人工作的文献综述，特别是并行求解工作的综述。
- 在章节2中，我们准确描述了所研究的问题，并详细、清晰地描述了并行算法设计，包括算法复杂度和解的质量分析。
- 在章节10中，我们对几种算法进行了对比研究，不只是孤立研究问题求解的一种算法，而是几种算法的比较与分析。

1.2 工作量说明

本研究为两人合作解决更大规模问题，工作量对应更大。平时作业经删减整理内容篇幅大约为 75 页，加入新内容后总报告页数已超过 110 页，新增任务量超过了原来的 30%，符合要求。

1.3 新旧内容及其分工

旧的内容中：

- 王俊杰负责位图算法的建立，list-wise 串行算法的实现、SIMD、多线程、MPI 中以 list-wise 算法为例的全部内容。
- 刘家骥负责鲲鹏服务器的使用，element-wise 串行算法的实现、SIMD、多线程、MPI 中以 element-wise 算法为例的全部内容。

新的内容包括：

- 过去工作的整理并形成文章（王俊杰负责）。
- GPU 并行的有关内容（刘家骥负责）。
- 压缩算法的有关内容（王俊杰负责）。
- 对比研究的有关内容（王俊杰负责）。
- 对比研究多线程部分（刘家骥负责）。
- 全篇实验报告的总结（刘家骥负责）。

通过以上说明，我们希望明确本研究报告的工作量及分工情况，确保各项内容的规范与完整。

2 问题描述

倒排索引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。倒排列表求交是当用户提交了一个 k 个词的查询，表求交算法返回各个倒排索引的交集。本实验拟实现倒排索引求交的串行算法，并采用并行化的方法，对算法进行优化和测试。

2.1 期末研究问题

倒排索引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。倒排列表求交是当用户提交了一个 k 个词的查询，表求交算法返回各个倒排索引的交集。本实验拟实现倒排索引求交的串行算法，并采用并行化的方法，对算法进行优化和测试。了解并实现倒排索引求交的两种算法，先实现 list-wise 和 element-wise 两种思路的串行算法，然后从多个角度研究相应并行算法的实现，如从减少 cache 未命中次数，减少比较次数、实现指令级并行（相邻指令无依赖）以利用超标量架构等等方面进行尝试，并且比较优化的性能。

3 文献综述

3.1 研究历史

在知网上检索中文文献，可以看到蓝色的倒排索引主题文献很多，集合求交的文献对比下很少，从 2005 年开始两个问题的研究都开始增加。说明两个主题之间具有很强的相关性。

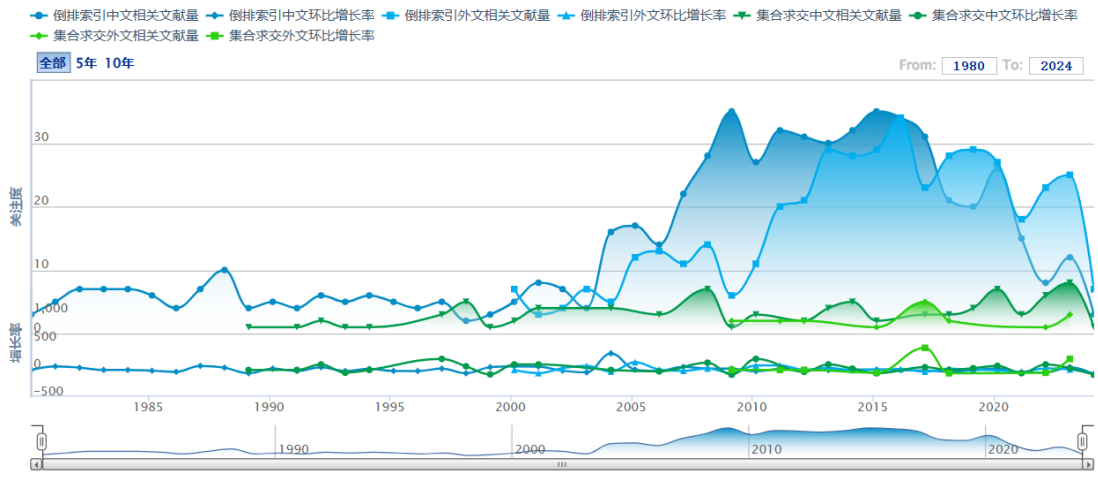


图 3.1: 中文文献

从 SCI 中检索”List Intersection” 精准匹配结果可以看到，2016 年来，研究主题的出版物和被引频次大幅度增加。

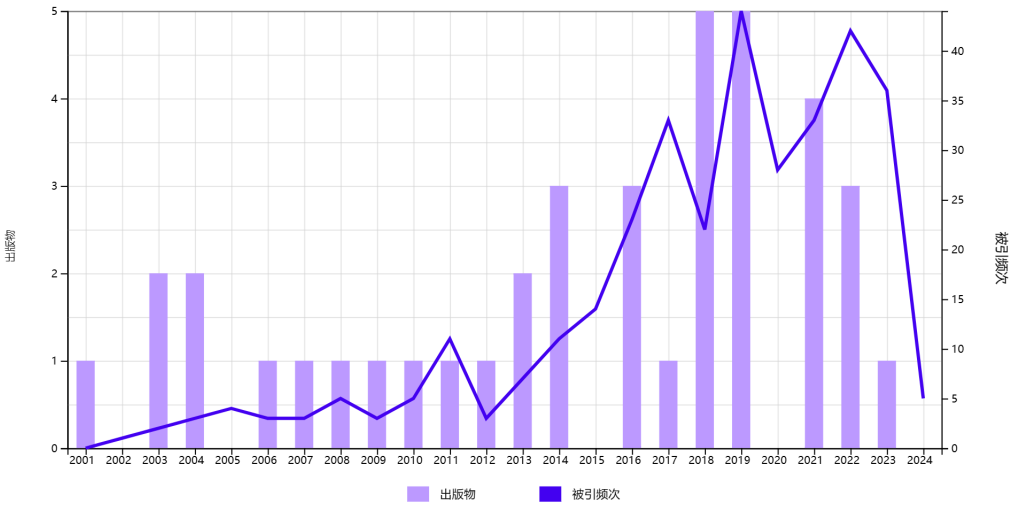


图 3.2: 外文文献

经过检索，发现所有文献中被引次数最多的 *Mining frequent itemsets using the N-list and subsume concepts*[4] 就出自研究主题大幅度增加的 2016 年，对于 N-list intersection 进行了改进研究，说明求交算法对于实际应用方面很重要。对于并行算法的改进，2011 年的 *Posting List Intersection on Multicore Architectures*[3] 使用了最新多核系统的计算能力来提高发布列表交集的性能，将与给定查询相关的工作划分为许多小而独立的任务。采用了多线程的方式。

3.2 研究现状

在最近的研究当中，对于并行化集合求交算法的并不多，集中在图形学，三角形有关运算的并行算法。[2][1] 关于倒排索引的研究很多，研究方向遍布各个领域。

图 3.3: 近期倒排索引研究领域

3.3 研究工作对比

求交算法很多，应用领域很广，在多个领域都存在对于求交算法的应用。在不同的算法当中采用的方式不同。在 *Mining frequent itemsets using the N-list and subsume concepts*[4] 中，采用了 N-list 和 Subsume 的信息检索优化技术，可以加速查询速度，N-list 可以有效地加速查询速度，特别是在倒排索引中存在大量倒排列表的情况下，通过将倒排列表划分成多个较小的子列表，可以减少查询的范围，提高查询效率，还可以简化查询处理减少了查询优化和计算逻辑的复杂性，有利于系统的维护和管理。但是对比并行算法，它适用性受限，N-list 主要针对查询范围较大的情况，对于查询范围较小或者查询条件较为简单的情况，可能会引入额外的计算和存储开销而无法带来明显的性能提升。而并行算法的适用性更为广泛，可以针对不同规模和复杂度的计算任务进行优化。

4 实验平台

4.1 刘家骥硬件平台

4.1.1 硬件平台

- CPU：12th Gen Intel(R)Core(TM)i7-12700H
- CPU 核心数：14
- 指令集架构：x86 架构
- GPU：NVIDIA GeForce RTX 3060 Laptop
- 内存容量：16GB
- L1 cache: 1.2MB
- L2 cache: 11.5MB
- L3 cache: 24MB

4.1.2 软件平台

x86

- 操作系统及版本：Windows 11 家庭中文版
- 编译器及版本：GNU GCC Compiler
- 编译选项：Have g++ follow the C++17 GNU C++ language standard(ISO)
  Optimize even more(for speed) [-O2] 加入-fopenmp

4.2 王俊杰实验平台

4.2.1 硬件平台

- x86 平台
  - 处理器：AMD Ryzen 7 5700U with Radeon Graphics 1.80 GHz

- 内存：16.0 GB (13.9 GB 可用)
- 系统类型：64 位操作系统, 基于 x64 的处理器

4.2.2 软件平台

- 操作系统：Windows 11 家庭中文版 23H2 版本 22631.3296
- 编译器：基于 GNC GCC 的 TDM-GCC-64
- 编译优化选项：
  - g++ 编译器选项：-std=c++17 -O2 -msse4.1 -pthread -fopenmp
  - 目标架构：x86_64 (64 位)
  - 优化等级：O2

4.3 鲲鹏 ARM 硬件平台

- 华为鲲鹏服务器
- CPU 核心数：96
- 指令集架构：aarch64
- L1d cache: 64K
- L1i cache: 64K
- L2 cache: 512K
- L3 cache: 49152K

5 串行算法设计

5.1 数据集的平凡导入

首先需要把二进制数据集整体导入程序当中，采用了二维向量的方式存储数据集，其中每个一维向量代表一个文档的倒排索引，每个元素代表一个单词的索引。以这样的方式存储数据集，可以方便地进行后续的处理。

Algorithm 1 导入 ExpIndex 文件

Input: 二进制文件"ExpIndex"

Output: 二维向量 indexData 存储数据

1: 打开文件"ExpIndex"，以二进制模式
2: 创建一个空的二维向量 indexData
3: **if** 文件成功打开 **then**
4: **while** 未到达文件末尾 **do**
5: 读取数组的长度 arrayLength
6: 创建一个长度为 arrayLength 的一维向量 arrayData
7: **for** 每个元素的索引 i，从 0 到 arrayLength-1 **do**

读取元素的值 value
将 value 存入 arrayData 的第 i 个位置
end for
将一维向量 arrayData 存入二维向量 indexData
end while
关闭文件
end if
return indexData

每个查询里面的向量形成一组，便于后续的处理。

Algorithm 2 导入 ExpQuery 文件

Input: 文件"ExpQuery"
Output: 三维向量 queryData 存储查询结果

打开文件"ExpQuery"
创建一个空的三维向量 queryData
if 文件成功打开 **then**
while 未到达文件末尾 **do**
读取查询的数量 queryCount
创建一个空的二维向量 queryIndex
for 每个查询的索引 j，从 0 到 queryCount-1 **do**
读取数组的长度 arrayLength
创建一个长度为 arrayLength 的一维向量 arrayData
for 每个元素的索引 i，从 0 到 arrayLength-1 **do**
读取元素的值 value
将 value 存入 arrayData 的第 i 个位置
end for
将一维向量 arrayData 存入二维向量 queryIndex
end for
将二维向量 queryIndex 存入三维向量 queryData
end while
关闭文件
end if
return queryData

5.2 list-wise 串行算法

按表求交基本思想是：先使用两个表进行求交，得到中间结果再和第三条表求交，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。

list-wise 串行算法

```
for (const auto& result : queryData) {
  // 创建一个一维向量 intersection 来存储求交结果
  vector<unsigned int> intersection = result[0];
```

```
4
5       // 对于每一个查询结果里面的每一个向量
6       for (const auto& value : result) {
7           // 创建一个一维向量 tempIntersection 来存储当前向量与 intersection 的交集
8           vector<unsigned int> tempIntersection;
9
10          // 对于当前向量中的每一个元素
11          for (const auto& element : value) {
12              // 如果当前元素也存在于 intersection 中，则将其添加到 tempIntersection
13              if (find(intersection.begin(), intersection.end(), element) !=
                    intersection.end()) {
14                  tempIntersection.push_back(element);
15              }
16          }
17
18          // 将 tempIntersection 更新为新的 intersection
19          intersection = tempIntersection;
20      }
21  }
```

5.3 element-wise 串行算法

5.3.1 算法设计思路

对倒排索引以元素求交，首先对读取数据集，按照 ExpQuery 中的要求将 ExpIndex 中的各倒排索引列表读出，建立一个三维的 vector，最低维度只存储一个倒排索引列表，所以可以利用两个维度存储 ExpQuery 中一组需要进行求交的倒排索引列表，第三个维度用来存储每一组需要进行求交的列表。

求交的算法思路为，先找出每一组中最短的那个列表并设为 S，因为求交后的列表不可能大于此列表，之后以此列表为基准遍历其他列表，如果其他列表中的元素在 S 中存在，则说明存在交集，若不存在，则从 S 中删除此元素，依次遍历所有列表。最终得到的 S 即为求交后的结果。

5.3.2 编程实现

关键代码如下，首先选出最小列表 S，之后循环每个列表与其比较，在 S 中删除没有交集的数。

其中使用 QueryPerformance 进行精确计时，将数据分为一百组，每一百组记录一次时间，存入数组，最后全部输出。

逐列访问平凡算法

```
1   vector<vector<unsigned int>> intersectionResults;
2   size_t queryDataSize = queryData.size();//组数，需要查询的个数，也是查询结果的个数
3
4   cout<<"queryDataSize: "<<queryDataSize<<endl;
5
6   size_t times=0;
7   size_t index=0;
8   size_t step=100;//每多少组数据测试一次时间
```

```
LARGE_INTEGER frequency;        // ticks per second
LARGE_INTEGER t1, t2;           // ticks
vector<double> elapsedTime(queryDataSize/step);

// get ticks per second
QueryPerformanceFrequency(&frequency);

QueryPerformanceCounter(&t1); // start timer at the beginning of the loop

for(size_t i=0;i<queryDataSize;i++)
{
    size_t minSize = queryData[i][0].size();
    size_t minIndex = 0;//最短列表的数组下标
        for (size_t j = 0; j < queryData[i].size(); j++)
            //queryData[i].size()为每组列表的个数
        {
            if (queryData[i][j].size() < minSize)
            {
                minSize = queryData[i][j].size();
                minIndex = j;
            }
        }
    vector<unsigned int> S = queryData[i][minIndex];//最短的列表
    for (size_t k = 0; k < queryData[i].size(); k++) //检查
    {
        auto it = S.begin();
        while (it != S.end())
        {
            if (find(queryData[i][k].begin(), queryData[i][k].end(), *it) ==
                queryData[i][k].end())    //
                std::find函数没有找到等于*it的元素，它会返回lists[i].end()
            {
                it = S.erase(it);//未找到即无交集，删除它
            }
            else
            {
                it++;
            }
        }
        cout<<i<<" k:"<<k<<endl;
    }
    intersectionResults.push_back(S);//S即为每组的交集
    times++;
    if(times%step==0)
    {
        // stop timer
    QueryPerformanceCounter(&t2);

    // compute and print the elapsed time in millisec
```

```
55        elapsedTime[index] = (t2.QuadPart - t1.QuadPart) * 1000.0 / frequency.QuadPart;
56        std::cout << "Elapsed time for 100 iterations: " << elapsedTime[index] << "
              ms.\n";
57        index++;
58        QueryPerformanceCounter(&t1); // reset the start timer for the next 100
              iterations
59        }
60    }
```

5.3.3 性能测试

比较在鲲鹏服务器及 arm 架构上运行此算法与 x86 架构上运行此算法的时间。

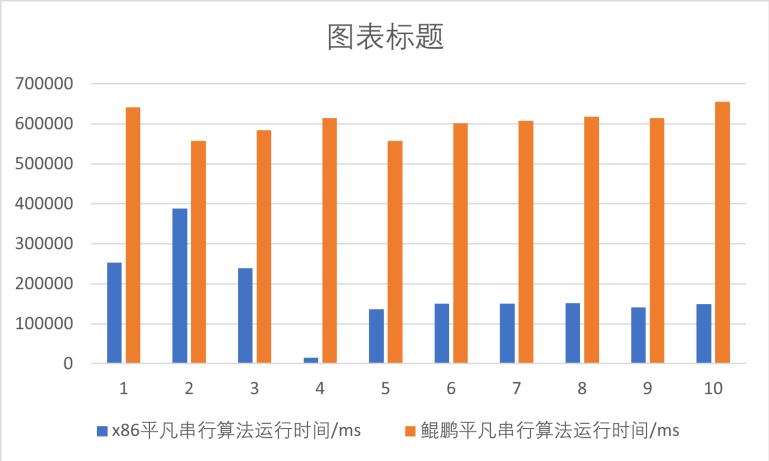


图 5.4: 平凡串行算法时间比较

6 SIMD 编程设计

6.1 位图索引

位图索引是一种用于加速数据库查询的技术，它通过将每个文档的倒排索引表示为一个位图，其中每个位代表一个单词的索引，从而实现了对文档的快速查询。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。

在进行位图索引的建立时，需要注意的是，每个文档的倒排索引的长度可能不同，因此需要在堆上动态分配内存来存储位图。在设置位图的最大长度时，需要设置为所有文档中最大的倒排索引的长度，以避免内存溢出。但是同时如果设置的位图长度过长，会导致内存的浪费，在栈上分配内存的方式也会导致栈溢出，因此需要在堆上动态分配内存。同时在堆里如果分配的内存过多，内存消耗会非常大，程序会崩溃。经过不断的调试，最终设置了一个合适的位图长度 40000000，使得程序能够正常运行。

Algorithm 3 通过位图索引读取 ExpIndex

Input: 二进制文件”ExpIndex”

Output: 二维位图 indexData 存储数据

1: 打开二进制文件”ExpIndex”

```
2: 创建一个二维位图 indexData 来存储数据
3: if 文件成功打开 then
4:     输出"indexFile opened"
5:     while 不是文件末尾 do
6:         读取数组的长度
7:         在堆上创建一个长度为 MAX_SIZE 的位图 arrayData
8:         for 每个元素 do
9:             读取元素的值
10:            将元素的值对应的位设置为 1
11:        end for
12:        将位图 arrayData 存入二维位图 indexData
13:        释放动态分配的内存
14:    end while
15:    关闭文件
16: end if
```

接下来的查询结果也同样需要进行位图索引的建立，以便于后续的处理。

Algorithm 4 通过位图索引读取 ExpQuery

Input: 打开查询文件"ExpQuery"

```
1: 创建一个三维位图 queryData 来存储查询结果
2: if 文件成功打开 then
3:     输出"queryFile opened"
4:     while 循环读取文件直到文件末尾 do
5:         创建一个二维位图 queryResult 来存储查询结果
6:         将行中的每个数字转换为索引文件的数组下标，并查询对应的位图
7:         创建一个字符串流 ss 并将行存入其中
8:         创建一个整数变量 index
9:         while 从 ss 中读取 index do
10:            查询对应的位图并将其存入 queryResult
11:        end while
12:        将二维位图 queryResult 存入三维位图 queryData
13:    end while
14:    关闭文件
15: end if
```

6.2 二级索引

位图索引太大，内存消耗太大，因此需要对位图索引进行压缩，采用二级索引的方式。二级索引的基本思想是：将位图索引分为多个块，每个块的大小为一个固定值，然后对每个块进行压缩，将每个块中的位图索引进行压缩，然后将压缩后的结果存储在二级索引中。具体操作上，将位向量分块，每个块用一个 bit 表示其全 0（0）还是非全 0（1），这些 bit 构成二级位向量。

6.2.1 由一级索引生成二级索引

在算法3的基础上，我们可以通过一级索引生成二级索引。

Algorithm 5 通过位图索引生成二级索引

Input: 一级索引数据 $indexData$

Output: 二级索引数据 $secondaryIndexData$

```
1: 创建一个空的二级索引数据 secondaryIndexData
2: for 每个一级索引 Onebitset in indexData do
3:     创建一个长度为 MAX_SIZE/BLOCK_SIZE 的位图 secondaryIndex
4:     for 每个块 i in Onebitset do
5:         创建一个空的位图 block
6:         for 每个位 j in i do
7:             if Onebitset[i + j] 为真 then
8:                 将 block 的第 i/BLOCK_SIZE 位设置为真
9:                 跳出内层循环
10:            end if
11:        end for
12:        将 block 存入 secondaryIndex
13:    end for
14:    将 secondaryIndex 存入 secondaryIndexData
15: end for
16: return secondaryIndexData
```

在这个过程中，有很大的内存消耗和时间复杂度，要对每一个位图都进行遍历，位图很大，所需要的时间和空间都很大，经过测试，一个位图的转化都需要很久很久，根本无法接受。因此，需要对这个过程进行优化，减少时间和空间的消耗。

6.2.2 同步构造二级索引

我们可以放弃整体把一级索引的位图转换为二级索引的位图，而是利用哈希表的方式，在构建一级索引的同时，利用哈希函数构造二级索引，由此省略了一次遍历的过程。

哈希函数如下

$$H\ (\text{value})\ = \lfloor \text{value}\ / BLOCK\_\text{SIZE}\ \rfloor$$

Algorithm 6 在生成位图索引生成二级索引

Input: 一级索引数据 $indexData$

Output: 二级索引数据 $secondaryIndexData$

```
1: 打开二进制文件"ExpIndex"
2: 创建一个空的一维位图列表 indexData
3: 创建一个空的一维位图列表 secondaryIndexData
4: if 文件成功打开 then
5:     while 未到达文件末尾 do
6:         读取数组的长度 arrayLength
7:         创建一个长度为 MAX_SIZE 的位图 arrayData
8:         创建一个长度为 MAX_SIZE/BLOCK_SIZE 的位图 secondaryIndex
```

```
        for i 从 0 到 arrayLength do
            读取元素的值 value
            将 value 对应的位设置为 1 在 arrayData 中
            将 value/BLOCK_SIZE 对应的位设置为 1 在 secondaryIndex 中
        end for
        将位图 arrayData 存入一维位图列表 indexData
        将位图 secondaryIndex 存入一维位图列表 secondaryIndexData
    end while
    关闭文件
end if
return secondaryIndexData
```

6.3 list-wise 基于二级索引位图的 SIMD 算法实现

6.3.1 基于二级索引位图的串行算法

在二级索引位图的基础上，我们可以先实现串行算法。在串行算法中，我们可以通过位运算来实现位图的求交。首先遍历二级索引，发现 1 的位置，然后在一级索引中找到对应的位置，然后进行位运算，得到最终的位图，然后再把位图转换为文档 ID。如此对于每个查询结果按表迭代求交，得到最后的结果。

Algorithm 7 基于二级索引位图的串行算法

Input: 二级索引位图 $secondaryqueryData$，一级索引位图 $BasequeryData$

Output: 交集结果 $intersectionData$

```
for i = 0 to DoNum do
    创建一维位图 intersection 并初始化为 BasequeryData[i][0]
    创建一维位图 secondaryintersection 并初始化为 secondaryqueryData[i][0]
    for j = 0 to BasequeryData[i].size() do
        创建一维位图 tempintersection
        创建一维位图 tempsecondaryintersection
        for k = 0 to MAX_SIZE/BLOCK_SIZE do
            if secondaryintersection[k]&secondaryqueryData[i][j][k] ≠ 0 两个二级索引位图均为 1 then
                for l = 0 to BLOCK_SIZE do
                    if intersection[k×BLOCK_SIZE+l]&BasequeryData[i][j][k×BLOCK_SIZE+l] ≠ 0 两个一级索引位图均为 1 then
                        tempintersection[k × BLOCK_SIZE + l] ← 1
                        tempsecondaryintersection[k] ← 1
                        if j = BasequeryData[i].size() − 1 then
                            将 k × BLOCK_SIZE + l 添加到 intersectionData[i]
                        end if
                    end if
                end for
            end if
```

19: **end for**
20: 更新 *intersection* 为 *tempintersection*
21: 更新 *secondaryintersection* 为 *tempsecondaryintersection*
22: **end for**
23: **end for**

6.3.2 基于二级索引位图的 SSE 算法实现

在 SSE 算法中，我们可以通过 SSE 指令来实现位图的求交。首先遍历二级索引，发现 1 的位置，然后在一级索引中找到对应的位置，然后通过 SSE 指令来实现位图的求交，得到最终的位图，然后再把位图转换为文档 ID。如此对于每个查询结果按表迭代求交，得到最后的结果。

基于二级索引位图的 SSE 算法

```
1
2
3                   // 如果当前位图的二级索引与 intersection 的二级索引有交集
4                   if ((*secondaryintersection)[k] & (*secondaryqueryData)[i][j][k])
5                   {
6                       bitset<128> *smallintersection=new bitset<128>() ;//
                            创建一个新的bitset
7                       bitset<128> *smallquery=new bitset<128>() ;// 创建一个新的bitset
8                       // 将大bitset中的特定范围的位复制到小bitset中
9                       for (int l = 0; l < BLOCK_SIZE; l++) {
10                          (*smallintersection)[l] = (*intersection)[k*BLOCK_SIZE + l];
11                          (*smallquery)[l] = (*BasequeryData)[i][j][k*BLOCK_SIZE + l];
12                      }
13                      // 使用_mm_loadu_si128函数将两个bitset转换为__m128i
14                      __m128i vec1 = bitset_to_m128i(*smallintersection);
15                      __m128i vec2 = bitset_to_m128i(*smallquery);
16                      delete smallintersection;
17                      delete smallquery;
18                      // 使用_mm_and_si128函数执行按位与操作
19                      __m128i andResult = _mm_and_si128(vec1, vec2);
20                      // 将__m128i转换为两个unsigned long long
21                      unsigned long long lower = _mm_cvtsi128_si64(andResult);
22                      unsigned long long upper =
                            _mm_cvtsi128_si64(_mm_srli_si128(andResult, 8));
23                      // 将两个unsigned long long值写入bitset
24                      for (int l = 0; l < 64; ++l) {
25                          (*tempintersection)[l+k*BLOCK_SIZE] = (lower >> l) & 1;
26                          (*tempintersection)[l+k*BLOCK_SIZE + 64] = (upper >> l) & 1;
27                          if(j==(*BasequeryData)[i].size()-1){
28                              if((*tempintersection)[l+k*BLOCK_SIZE] == 1){
29                                  intersectionData[i].push_back(l+k*BLOCK_SIZE);
30                              }
31                              if((*tempintersection)[l+k*BLOCK_SIZE + 64] == 1){
32                                  intersectionData[i].push_back(l+k*BLOCK_SIZE + 64);
33                              }
```

```
34                          }
35                      }
36                      // 如果当前位图的一级索引与 intersection
                            的一级索引有交集,则更新二级索引
37                      if(lower || upper)
38                          tempsecondaryintersection->set(k);
39                  }
40              }
41              // 更新 intersection 为 tempintersection
42              //......  }
```

通过 SSE 指令，可以大大提高位图的求交速度，减少了位图的求交时间，提高了程序的效率。经过并行 SIMD 化设计，可以大大提高程序的效率，减少了程序的运行时间。

6.4 list-wise 实验结果

测试集中包含了 1000 个查询，每个查询包含了若干个文档。对此我们进行了分组，分成十组，每组包含了 100 个查询。实验共测量三种算法，分别为按表求交的串行算法，基于二级索引位图的串行算法，基于二级索引位图的 SSE 算法。利用刘家骥提供的时间测量方法，分别对这三种算法进行了测试，得到了如下的结果。

表 1: 实验结果

分组	平凡算法	二级索引	SSE
0	16209.5	2717.35	1606.99
1	13518.2	1622.96	2279.28
2	13635.3	1542.06	1560.9
3	15290.8	1455.23	1754.48
4	13940.2	1714.09	2067.27
5	14312.9	1652.66	1601.21
6	14652.7	2281.62	2120.46
7	14586.6	1794.87	2936.28
8	15076.2	1444.54	1949.74
9	15134.5	1461.34	1504.01

6.5 list-wise 实验分析

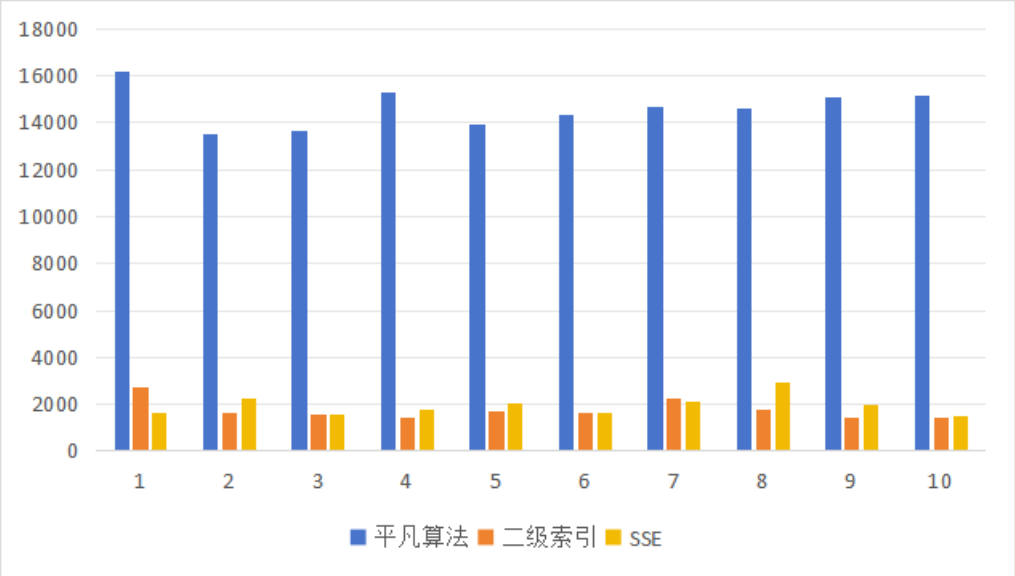


图 6.5: 三种算法用时比较

可以看出，二级索引位图的优化十分明显，比按表求交的串行算法快了很多，而 SSE 算法的优化效果不佳，有些组甚至还出现了用时变长的情况。

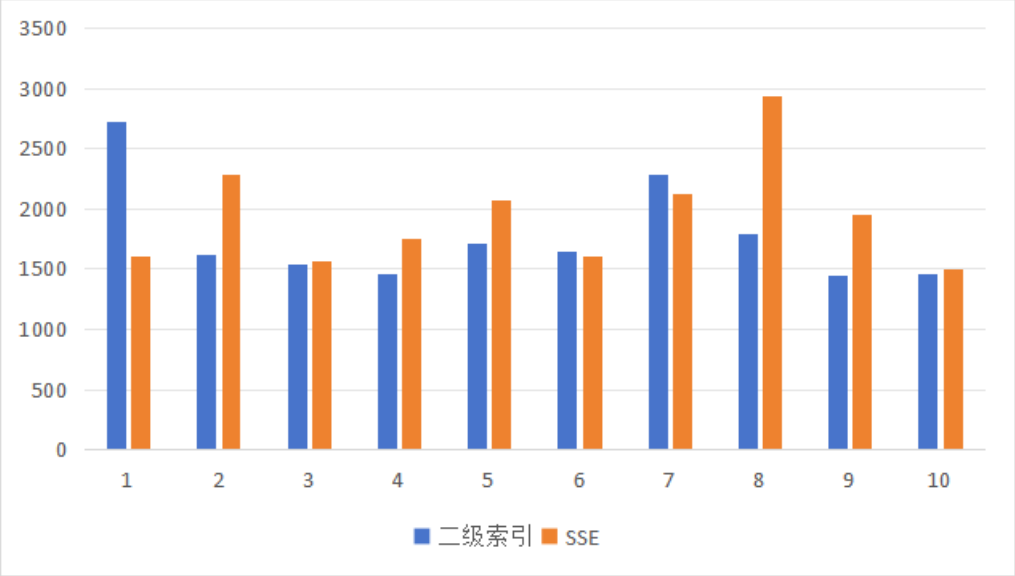


图 6.6: 两种算法用时比较

可以看到，并行优化的效果并不明显。因为算法只对位运算进行优化，而位运算的多了很多其他的转换步骤，因此优化效果不好，还有很多进步的空间。

6.6 list-wise 总结

二级索引位图：在数据库系统中，通常使用位图索引来加速查询操作。位图索引将每个可能的取值都映射到一个位图，位图的每一位表示对应的行是否包含该取值。二级索引位图是在此基础上建立

的优化方法。它使用两级位图来表示索引信息，第一级位图用于标识哪些块（Block）中可能包含某个值，第二级位图则用于标识确切的行号。并行化优化：为了提高查询性能，可以利用并行计算的特性，将任务分配给多个处理单元同时执行。在二级索引位图的优化中，可以将位图的操作并行化，加速查询过程。SIMD 化优化：SIMD 是一种并行计算的技术，通过一条指令同时处理多个数据，提高了计算效率。在二级索引位图的优化中，可以利用 SIMD 指令集来同时处理多个位图，加速位图的合并、交集等操作。

未来的工作：在二级索引位图的优化中，可以进一步提高查询性能。可以考虑以下几个方面的工作：

并行加载位图：在处理大规模数据时，可以将位图的加载操作并行化，利用多个处理单元同时加载多个位图，减少加载时间。

并行位图操作：针对位图的合并、交集等操作，可以将这些操作分解成多个子任务，并行执行。利用 SIMD 指令集，可以在每个处理单元上同时处理多个位图，提高操作效率。

任务调度与负载均衡：合理的任务调度和负载均衡策略可以确保各个处理单元的工作量均衡，避免资源浪费和性能下降。

6.7 element-wise 基于二级索引位图的串行算法

6.7.1 算法设计思路

求交算法思路与之前相同，还是先找出每一组中最短列表并设为 S，并遍历其他列表与之比较，删除无交集的元素。但是由于使用了位图存储方式，由于采用位图存储方式会有巨大的空间上的浪费（即 0 极多，1 极少），故添加了二级索引，算法部分也应有相应变化。

首先运用王俊杰同学写出的位图及二级索引相关代码，将文件读出，并使用 vector+bitset 类型进行存储。在搜索相交元素时，先搜索二级索引，再搜索位图中具体的信息，并对 S 进行更新。注意 S 应也有一个二级索引 S_second，以便在后续将位图翻译为整形数组时加快查找速度。

将整型变为 bitset 类型，并且以 BLOCK_SIZE 为大小进行二级索引的建立。

6.7.2 编程实现

实现位图及二级索引的伪代码如下，二级索引将位图分为很多个块，每个块的大小为 BLOCK_SIZE，本次实验 BLOCK_SIZE 取值位 64。

我对实现位图及二级索引的代码做了一部分改动以适应 element-wise 算法，即在读取数据时，将各列表长度存储到容器 query_Lengths 中，并在后续存储查询结果时，按查询结果的顺序将各列表长度存储到 query_Lengths_searched 中。以便在后续选择最短列表时，不用再求各位图中 1 的个数，可以节省大量时间。

Algorithm 8 二级索引下 element-wise 串行算法

```
输出 MAX_SIZE / BLOCK_SIZE
打开二进制文件"ExpIndex"
在堆上创建一个二维位图 indexData 来存储数据
创建一个二级索引 secondaryIndexData
创建一个数组 query_Lengths
if 文件成功打开 then
    输出"indexFile opened"
    while 文件未到末尾 do
        读取数组的长度 arrayLength
        将 arrayLength 存入 query_Lengths
        在堆上创建一个长度为 MAX_SIZE 的位图 arrayData
        在堆上创建一个长度为 MAX_SIZE / BLOCK_SIZE 的位图 secondaryIndex
        for i = 0 to arrayLength do
            读取元素的值 value
            将 arrayData 和 secondaryIndex 的对应位设置为 1
        end for
        将 arrayData 和 secondaryIndex 存入 indexData 和 secondaryIndexData
        释放 arrayData 和 secondaryIndex 的内存
    end while
    关闭文件
end if
打开查询文件"ExpQuery"
创建一个三维位图 BasequeryData 来存储查询一级索引结果
创建一个三维位图 secondaryqueryData 来存储查询二级索引结果
创建一个二维数组 query_Lengths_searched
if 文件成功打开 then
    输出"queryFile opened"
    初始化计数器 count
    while 文件未到末尾 do
        读取一行数据 line
        创建二维位图 BasequeryResult 和 secondaryqueryResult
        for line 中的每个数字 index do
            查询 index 对应的位图并存入 BasequeryResult 和 secondaryqueryResult
            输出"index:" 和 index
            将 query_Lengths[index] 存入 query_Lengths_searched[count]
        end for
        count 加一
        输出"count:" 和 count
        将 BasequeryResult 和 secondaryqueryResult 存入 BasequeryData 和 secondaryqueryData
    end while
    关闭文件
end if
```

对于求交算法的实现，我们对 S 的二级索引 S_second 和所比较二级索引按位进行与运算即可，如果按位与结果为 0，对 S 即 S_second 进行 reset 置零操作。但是注意此处如果按位与结果均为 1，并不一定说明存在相交元素，还应该进入位图中继续逐位进行与运算，若此时按位与运算结果为 1，说明此处确实存在相交元素，这也是在编程过程中我遇到的一个找了很久的 bug。

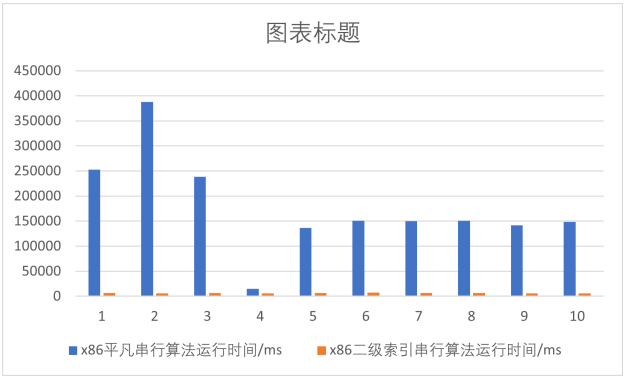
关键代码如下，前面获取最短列表与平凡算法相同，下文仅展示了关键的按位与计算的步骤。

```
1
```

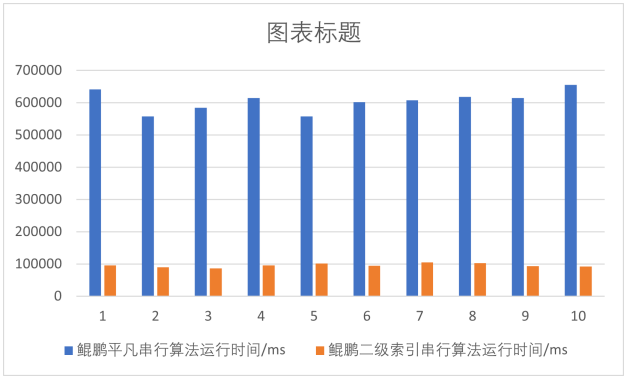
```
    for (size_t k = 0; k < query_Lengths_searched[i].size(); k++) // 检查
    {
      if (k == minIndex) // 跳过最短列表
      {
        continue;
      }
      for (size_t l = 0; l < MAX_SIZE / BLOCK_SIZE; l++) {
        bool flag_second = false;
        if ((*S_second)[l] &
            (*secondaryqueryData)
                [i][k][l]) // 如果按位与结果为1,进去检查一遍, 有可能两数不同
        {
          for (size_t m = l * BLOCK_SIZE; m < l * BLOCK_SIZE + BLOCK_SIZE;
               m++) // 检查范围
          {
            if (!((*S)[m] & (*BasequeryData)[i][k][m])) {
              S->reset(m);
            } else { // 这组中存在相交的元素,flag置为true, 表示二级索引此处应为1
              flag_second = true;
            }
          }
          if (!flag_second) {
            S_second->reset(l); // 此处为假1
          }
        } else {

          S_second->reset(l); // 第1个BLOCK_SIZE位
          // 进入一级索引置零
          for (size_t m = l * BLOCK_SIZE; m < l * BLOCK_SIZE + BLOCK_SIZE;
               m++) // 检查范围
          {
            S->reset(m);
          }
        }
      } // 进行求交算法

      cout << i << " k:" << k << endl;//进度条
    }
    intersectionResults->push_back(*S);
    intersectionResults_second->push_back(*S_second);
  }
```

最后将结果分别存入 intersectionResults 和 intersectionResults_second，构成新的位图及其二级索引。最后再将位图翻译为数组，与平凡串行算法进行对比，得出的结果一样，说明算法正确。

6.7.3 性能测试



(a) x86 二级索引串行算法比较



(b) 鲲鹏二级索引串行算法比较

图 6.7: 不同平台串行优化算法的执行时间对比

6.8 element-wise 算法 neon 化算法实现

6.8.1 neon 化算法实现

代码使用 vdupq_n_u32 指令将当前要检查的元素复制到一个 4 元素的向量中,然后使用 vld1q_u32 指令加载当前索引数据的 4 个元素到另一个向量中, 再对两向量进行比较。

在多次进行尝试后，NEON 并行化并没有起到优化效果，因为**按元素求交算法**这个思路在理论上并不适合进行 NEON 并行化，因为其运行逻辑是嵌套循环在两个向量中查找元素，需要逐个比较进行查找，若进行 NEON 化，在连续读取为四个元素组成的向量后，无法对结果一起处理，依旧需要逐个进行比较，失去了 NEON 化的意义。

尝试优化后发现性能不如原串行算法。

	arm 平凡倒排索引求交	armneon 倒排索引求交
0	450000	500000
100	390000	397000
200	408000	452000
300	431000	462000
400	390000	386000
500	421000	436000
600	426000	423000
700	432000	458000
800	431000	427000
900	459000	463800
TOTAL TIME	4238000	4404800

6.9 elemeng-wise 算法在二级索引及位图存储下的 NEON 并行化

6.9.1 算法设计思路

在上述串行算法的基础上，尝试对其实现 NEON 并行化。

主要思路集中在按位与运算上，串行算法中，对于与运算的实现是通过循环，一位一位的进行与运算，浪费了大量的时间。我们考虑将需要进行按位与运算的目标 bitset，即 S_second 进行向量化，

与化为向量后的原二级索引的 bitset 进行两个向量之间的按位与运算，得到结果向量，并在 S_second 中更新结果。

6.9.2 编程实现

关键代码如下，简化了进行与运算的步骤，将一位一位运算，变为了 128 位一起运算。

Algorithm 9 位图并行与操作

```
将 S 的地址转换为 size_t 类型并赋值给 data1
将 S_second 的地址转换为 size_t 类型并赋值给 data2
将 BasequeryData 的地址转换为 size_t 类型并赋值给 rdata1
将 secondaryqueryData 的地址转换为 size_t 类型并赋值给 rdata2
for t = 0 to MAX_SIZE / BLOCK_SIZE / 128 do
    从 rdata2 + 4*t 的地址加载一个 uint32x4_t 类型的值到 secondaryqueryData_bits
    从 data2 + 4*t 的地址加载一个 uint32x4_t 类型的值到 S_second_bits
    对 secondaryqueryData_bits 和 S_second_bits 进行并行与操作，结果存入 and_result
    将 and_result 存回 data2 + 4*t 的地址
end for
```

后续按照二级索引串行算法对 S_second 操作编写代码即可，同样需要注意二级索引中为 1 的情况需要检查原始位图。

6.9.3 性能测试

可以看出，在并行化后，性能并没有明显的提升，可能是因为在按元素求交算法的执行过程中，每一步的运算结果都依赖于前一步，所以并行空间有限，而只将位运算向量化后并不能明显的提升性能，所以总体性能表现平凡，下一步可以尝试在数据读取过程中或其他地方继续进行优化。

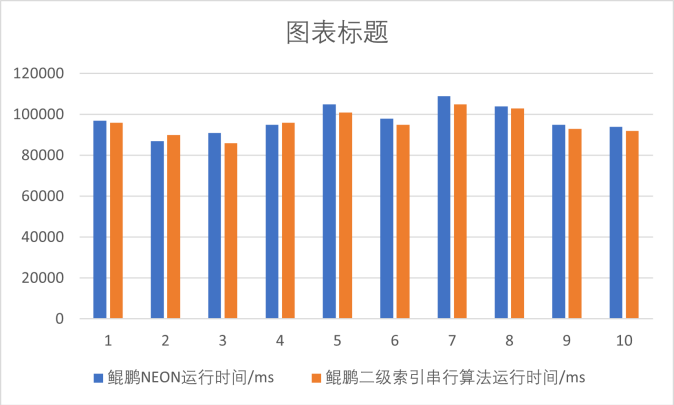


图 6.8: 并行算法与串行算法优化比较

6.10 element-wise 实验结果分析

实验原数据如下

问题分组 n	1	2	3	4	5	6	7	8
x86 平凡串行算法运行时间/ms	252797	387947	238552	147973	136184	150352	149748	1509
x86 二级索引串行算法运行时间/ms	6594.15	6110.65	6228.01	6059.47	6523.4	7639.22	6832.79	6447
鲲鹏平凡串行算法运行时间/ms	641000	557000	584000	615000	557000	602000	608000	6180
鲲鹏二级索引串行算法运行时间/ms	96000	90000	86000	96000	101000	95000	105000	1030
鲲鹏 NEON 运行时间/ms	97000	87000	91000	95000	105000	98000	109000	1040

如图??所示，我将整个查询数据集分为了 10 份，每份含一百组数据。

将所有的计算时间进行对比后，可以明显看出，加入二级索引的串行算法运行速度明显高于其平凡串行算法。而对于并行算法，与使用二级索引的串行算法不相上下。

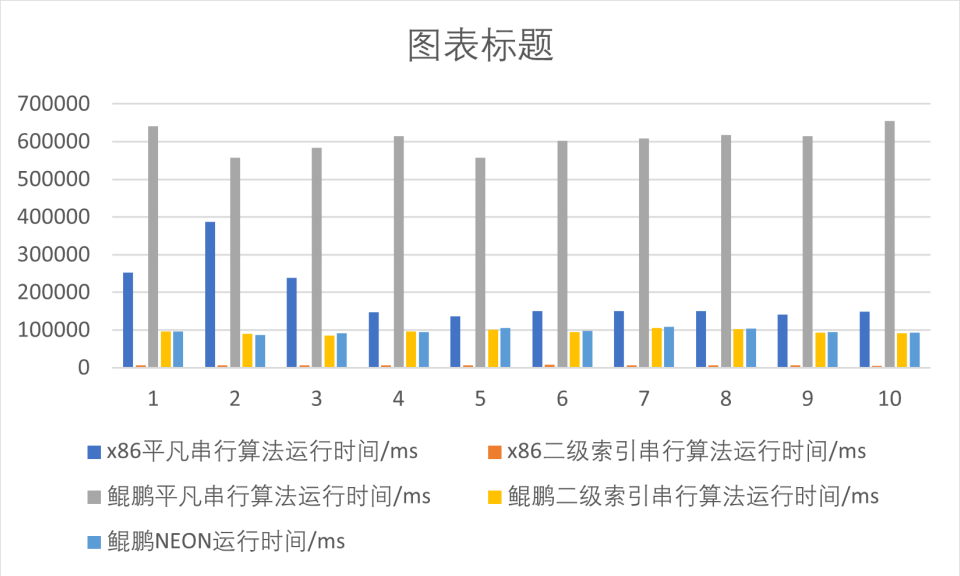


图 6.9: 各算法不同架构时间对比

如图8.57所示，经过比较，在 x86 平台上，使用二级索引的串行算法的性能提升十分显著。可能的原因是，在我的测试设备上，x86 的每级缓存更大，因此在处理倒排索引求交运算这种可能涉及大量内存访问的任务时，这个优势可能会更明显。

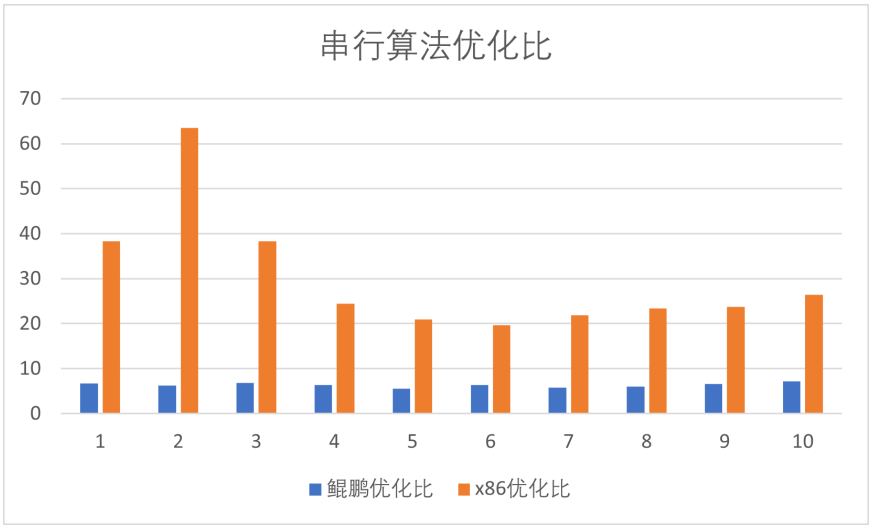


图 6.10: 串行算法优化比

6.11 element-wise 总结

与 SSE 和 AVX 不同，ARM 架构采用的是 NEON 指令集，需要注意其数据宽度的差别。在进行并行操作时，SSE 指令集最大的并行度是 128 位，NEON 最大的并行度也是 128 位，而在 AVX 和 AVX2 中，这个宽度扩大到了 256 位。在进行并行计算时，需要注意设置的并行位宽。

下一步应继续优化二级索引的存储方式，现在这个算法对于存储空间还是有些浪费，后续将考虑使用压缩技术来减少存储空间。例如，可以使用位图索引、B+ 树、哈希索引、可扩展哈希表等数据结构来存储索引，减少空间的浪费，同时提高访问效率。

同时对于并行化的实现，可考虑使用 NEON 的加载指令（如 vld1q_u32）来并行加载数据。还可以优化串行算法代码本身，减少不必要的访问，比如现在的串行算法代码中会有重复置零的操作，可以通过修改算法来减少不必要的比较和赋值。

7 多线程编程设计

7.1 list-wise:Query 间并行 pthread 算法

Query 间并行是指对于每个 Query，都可以开启一个线程来处理，这样可以充分利用多核的优势，提高并行度。在这里，我们采用 pthread 库来实现 Query 间并行算法。首先使用动态线程方式，设置好线程数目，然后对于每个 Query，开启一个线程，线程的执行函数是一个函数指针，指向一个函数，这个函数的参数是 Query 的编号，然后在这个函数中，对于这个 Query，进行处理。

[H] **Algorithm 10** Query 间并行 pthread 算法动态线程方式

Input: 工作线程数量 $worker\_count$，查询数据大小 $queryDataSize$，索引数据 $indexData$，查询数据 $queryData$

Output: 无

function THREADFUNC($param$)
    $t\_id \leftarrow param.t\_id$ ▷ 线程编号
    $t\_begin \leftarrow t\_id \times \frac{queryDataSize}{worker\_count}$ ▷ 计算任务的起始位置
    $t\_end \leftarrow (t\_id + 1) \times \frac{queryDataSize}{worker\_count}$ ▷ 计算任务的结束位置
    初始化线程计时器
    for $Qi \leftarrow t\_begin$ to t_end **do**
        $result \leftarrow queryData[Qi]$
        处理查询结果
    end for
    结束线程计时器
end function

function MAIN
    初始化数据
    创建工作线程数组 $handles$
    创建线程参数数组 $param$
    初始化总计时器
    for $t\_id \leftarrow 0$ to $worker\_count$ **do**
        $param[t\_id].t\_id \leftarrow t\_id$

end for
for $t\_id \leftarrow 0$ to $worker\_count$ **do**
    创建线程 $handles[t\_id]$
end for
for $t\_id \leftarrow 0$ to $worker\_count$ **do**
    等待线程 $handles[t\_id]$ 完成
end for
结束总计时器
end function

该算法的线程分配图示如下：

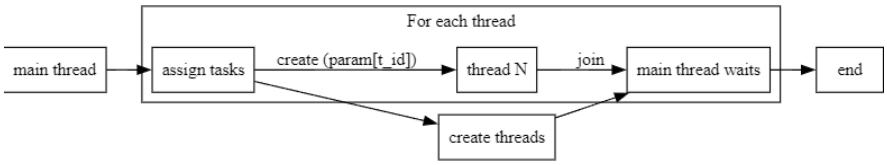


图 7.11: Query 间并行 pthread 算法线程分配

7.2 list-wise:Query 间并行 OpenMP 算法

openMP 也是一种多线程编程的方式，它可以很方便的实现多线程编程。从串行算法到并行算法的转换，只需要在需要并行的地方加上一句注解，就可以实现并行化，这样可以大大减少代码量，提高开发效率。对于 Query 间并行 OpenMP 算法，我们只需要在 for 循环的前面加上一句 #pragma omp parallel for，就可以实现并行化。

Query 间 openMP 编程

```
int worker_count = 100; //工作线程数量
//读取数据
......
//开启总计时器
#pragma omp parallel  num_threads(worker_count)
{
    #pragma omp for
for (int Qi = 0; Qi < queryDataSize; ++Qi) {
    const auto& result = queryData[Qi];
    //开启线程计时器
    //处理查询结果
    ......
    //关闭线程计时器
  }
}
//关闭总计时器
```

7.3 list-wise:Query 内并行 pthread 算法

Query 内并行是指对于每个 Query 内部进行多线程的处理，这样可以充分利用多核的优势，提高并行度。对于按表求交算法，在 Query 内部的多个表进行求交的过程中，可以采用多个线程分别两两求交得到结果，然后再合并结果。采用树状结构进行合并，可以减少合并的时间复杂度。

然而，对实际情况进行考察，发现在 1000 个 Query 中，有 563 个 Query 有两个表，340 个 Query 有三个表，83 个 Query 有四个表，14 个 Query 有五个表，两个和三个表的情况没有多线程的意义，因为只需要一次或者两次求交操作，

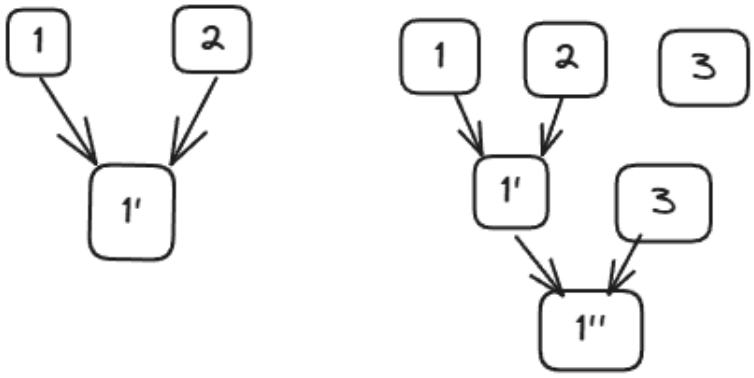


图 7.12: 两个和三个表求交的情况

所以我们只对四个表和五个表的 Query 进行多线程处理，在这两种情况中理论上可以实现一次并行来减短运行时间。

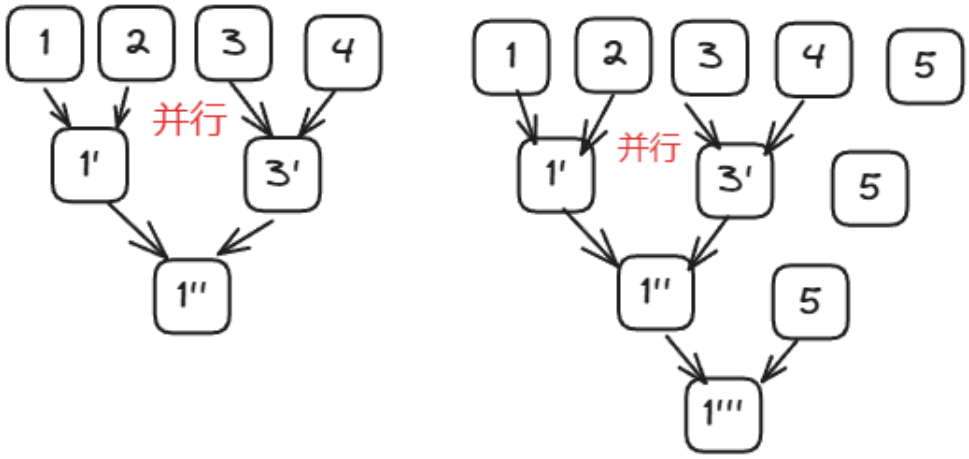


图 7.13: 四个表和五个表求交的情况

因为只有这两种情况需要并行，所以简化代码，仅对于这两种情况进行多线程处理。

[H] **Algorithm 11** Query 内并行 pthread 算法

Input: 工作线程数量 *worker_count*，查询数据大小 *queryDataSize*，索引数据 *indexData*，查询数据 *queryData*

Output: 查询结果 *intersection*

1: **function** QUERY 内并行 PTHREAD 算法 (worker_count, queryDataSize, indexData, queryData)

创建工作线程句柄数组 $handles$，大小为 4
创建线程参数结构体数组 $param$，大小为 4
for Qi in 0 to $queryDataSize - 1$ **do**
    设置 $param[0].QueryIndex$ 为 Qi
    设置 $param[1].QueryIndex$ 为 Qi
    设置 $param[2].QueryIndex$ 为 Qi
    设置 $param[3].QueryIndex$ 为 Qi
    设置 $param[0].r1$ 为 0
    设置 $param[0].r2$ 为 1
    设置 $param[1].r1$ 为 2
    设置 $param[1].r2$ 为 3
    设置 $param[2].r1$ 为 0
    设置 $param[2].r2$ 为 2
    设置 $param[3].r1$ 为 0
    设置 $param[3].r2$ 为 4
    创建线程 $handles[0]$，执行函数为 $threadFunc$，参数为 $param[0]$
    创建线程 $handles[1]$，执行函数为 $threadFunc$，参数为 $param[1]$
    等待线程 $handles[0]$ 结束
    等待线程 $handles[1]$ 结束
    创建线程 $handles[0]$，执行函数为 $threadFunc$，参数为 $param[2]$
    等待线程 $handles[0]$ 结束
    if 查询结果大小为 5 **then**
        创建线程 $handles[0]$，执行函数为 $threadFunc$，参数为 $param[3]$
        等待线程 $handles[0]$ 结束
    end if
    将查询结果 $result[0]$ 赋值给 $intersection$
end for
end function

线程的变化图示如下：

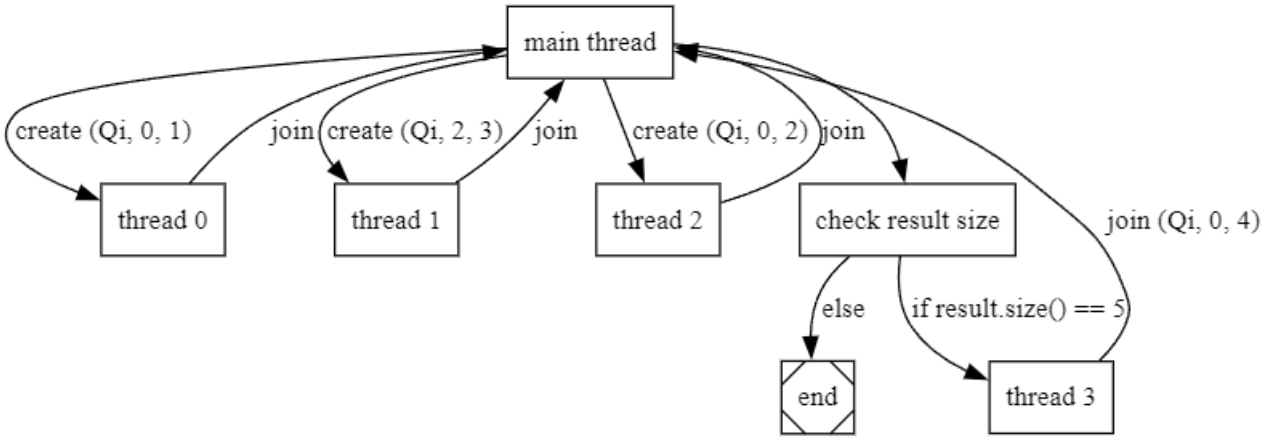


图 7.14: Query 内并行 pthread 算法线程分配

然而可以看到这样的处理线程创建销毁的操作很频繁，开销很大，而且优化的范围很小，可能达不到理想的效果。

考虑到进一步的 openMP，因为对于按表求交来说，存在很大的数据依赖，所以只能采取树状结构进行求交，但是又因为规模小，只有手动安排线程来进行逐一操作，这与 openMP 的自动化并行化相违背，所以不适合使用 openMP 进行优化。对于 Query 内并行 openMP 的优化，会在后面进阶要求使用位图来实现。

7.4 list-wise: 实验步骤与结果

注意：本节下文所有时间单位均为 ms

共有 1000 个 Query, 为了更细致的体现不同并行策略的情况，我们将数据按照顺序分为十组，每组 100 个 Query。最平凡的按表求交算法，我们将其作为基准算法，并且可以看出在无优化的情况下数据集的情况分布。

0	1	2	3	4	5	6	7	8	9	SUM
22639	21487.3	21539	24025.9	22216.6	22958.6	23448.4	22440.2	26365.7	24028.1	231148.8

表 2: 最平凡算法每一组分布

实验要求与 SIMD 结合，于是将其中的求交操作使用 SSE 改写之后，得到如下数据:

0	1	2	3	4	5	6	7	8	9	SUM
15173.7	13952	14420.9	15982.1	14934.5	15485.7	15249.5	15115.7	15572	16091.3	151977.4

表 3: SSE 改写之后的每一组分布

对于 SSE 改写之后的代码进行进一步优化，将其空间分配改写成更适合 SIMD 的写法，实验数据如下:

0	1	2	3	4	5	6	7	8	9	SUM
14721	13625.7	14004.1	15632.8	14561.8	15156.2	15219.9	15001.5	15333	15500.7	148756.7

表 4: 内存分配优化之后 SSE 的每一组分布

之后为了实现优化的效果最大化，使用上面这个 SIMD 化的算法作为基准进行多线程的优化。

在 Query 间并行 pthread 算法中，我们将数据分为 10 组，每组 100 个 Query，然后对于每个 Query，开启一个线程进行处理，线程的数量为 100，实验数据如下:

0	1	2	3	4	5	6	7	8	9	TOTAL
30935.3	25731.6	27202.1	31152.5	30650.5	29144.7	29380.5	30230.8	30427.7	30239.8	31154

表 5: Pthread 算法十个线程分布

对于 Query 内并行 pthread 算法，我们只对四个表和五个表的 Query 进行多线程处理，使用十个线程，实验数据如下:

0	1	2	3	4	5	6	7	8	9	TOTAL
17228.4	15894.4	16841.2	17979.4	15353.8	17470.8	15298.5	18028.3	17471.1	17599.8	169165.7

表 6: Query 内并行 pthread 算法十个线程分布

对于不同线程数目的影响，我们将 Query 内的多线程编程线程数目改变，记录数据如下表；

线程数	Pthread 总用时	openMP 总用时
2	78992.9	39258.1
4	56430.4	23403.7
5	38583.8	18359.1
8	38098.2	\
10	27948.3	16538.7
20	29392.9	12636.8
25	29527.6	12438.9
40	\	12327.5
50	\	12283.8
100	\	12224.9
200	\	12039.2
500	\	11975.7
750	\	11977.5
800	\	11938.6
825	\	11837.2
850	\	11844.3
900	\	11915.8
1000	\	11962.6

表 7: 线程数目对时间的影响

7.5 list-wise 下各种并行效果对比

7.5.1 list-wise: 并行和串行对比

问题规模方面把各个并行化策略和串行程序运行时间整理在一起，如下表：

	最平凡	平凡 SIMD	预分配 SIMD	间十个线程	内 pthread
0	22639	15173.7	14721	30935.3	17228.4
1	21487.3	13952	13625.7	25731.6	15894.4
2	21539	14420.9	14004.1	27202.1	16841.2
3	24025.9	15982.1	15632.8	31152.5	17979.4
4	22216.6	14934.5	14561.8	30650.5	15353.8
5	22958.6	15485.7	15156.2	29144.7	17470.8
6	23448.4	15249.5	15219.9	29380.5	15298.5
7	22440.2	15115.7	15001.5	30230.8	18028.3
8	26365.7	15572	15333	30427.7	17471.1
9	24028.1	16091.3	15500.7	30239.8	17599.8
SUM	231148.8	151977.4	148756.7	31154	169165.7

表 8

串行算法进行 SIMD 化之后，可以看出 SIMD 实现了 1.52 和 1.55 的优化，效果显著。

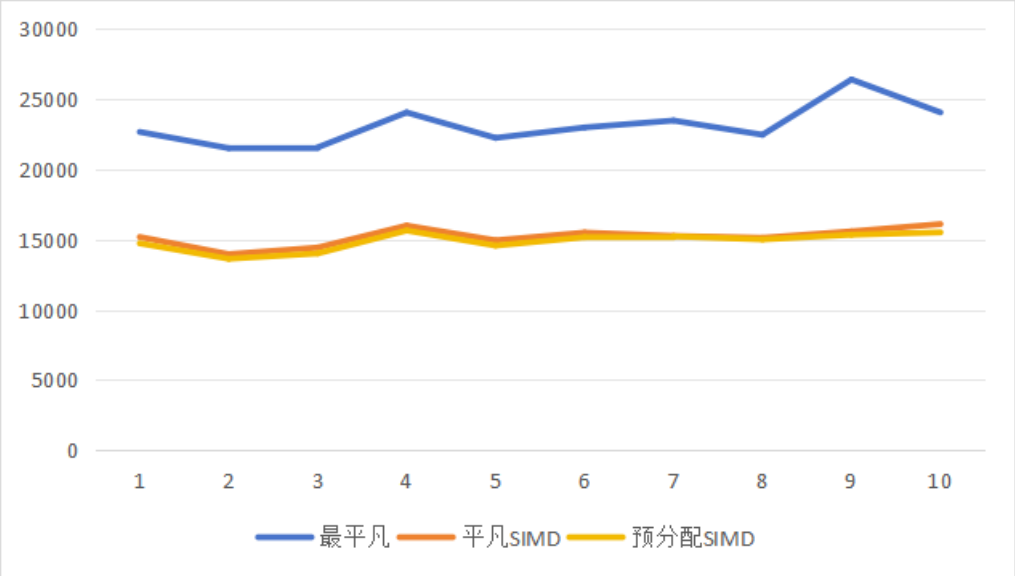


图 7.15: SIMD 优化效果

可以看到蓝色的线表示最平凡算法，也表示数据集的分布，可以看到 SIMD 优化后的曲线起伏基本持平，说明 SIMD 对于各种的 Query 都有着同样的优化效果，和理论上相符合。具体展开来看，对比每组的加速比如下：

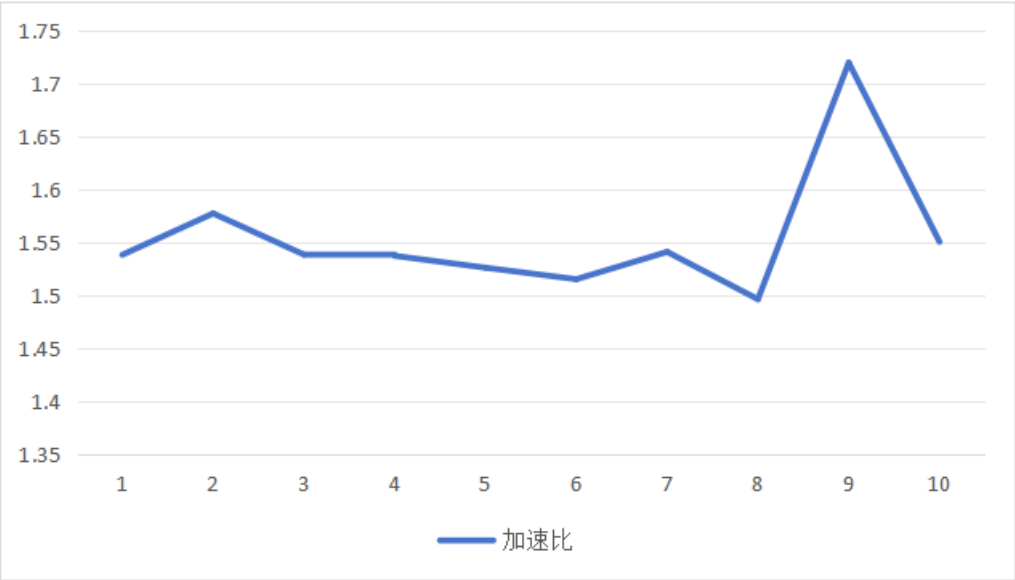


图 7.16: SIMD 加速比

可以看到，第九组有着突出的加速比，在问题规模差不多的情况下，这是因为第九组的数据分布和其他组有着很大的差异，SIMD 优化后的算法对于这种数据分布有着很好的优化效果。

为了和分组对比，我们将数据分为十组，每组 100 个 Query，然后对于每个 Query，开启一个线程进行处理，线程的数量为 100，实验数据如下:

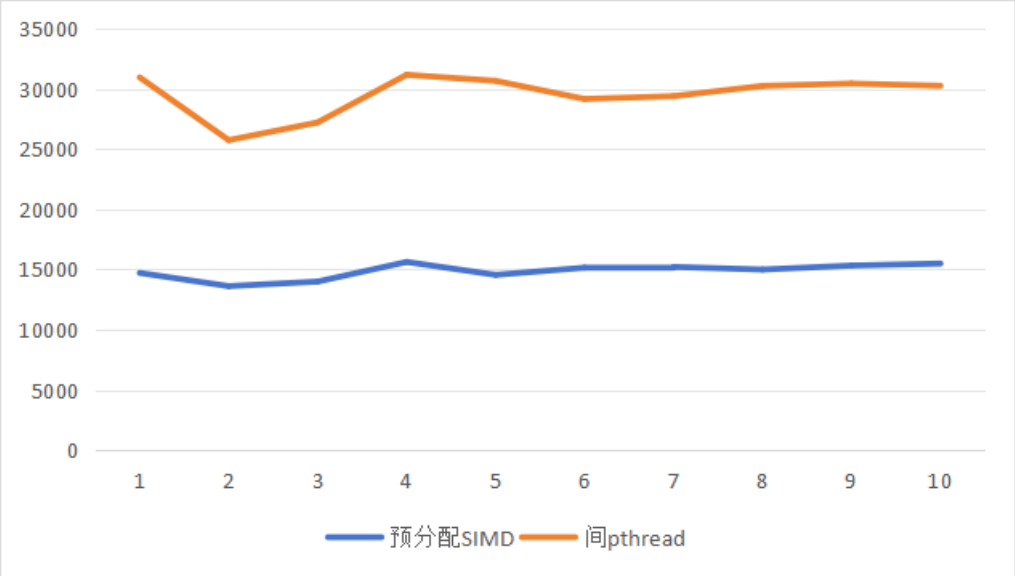


图 7.17: 每一个线程用时

我们可以看到，对于每一个线程来说，用时都有着显著的增加。这是因为每个子线程的处理能力有限，激素速度要低于主线程串行执行。

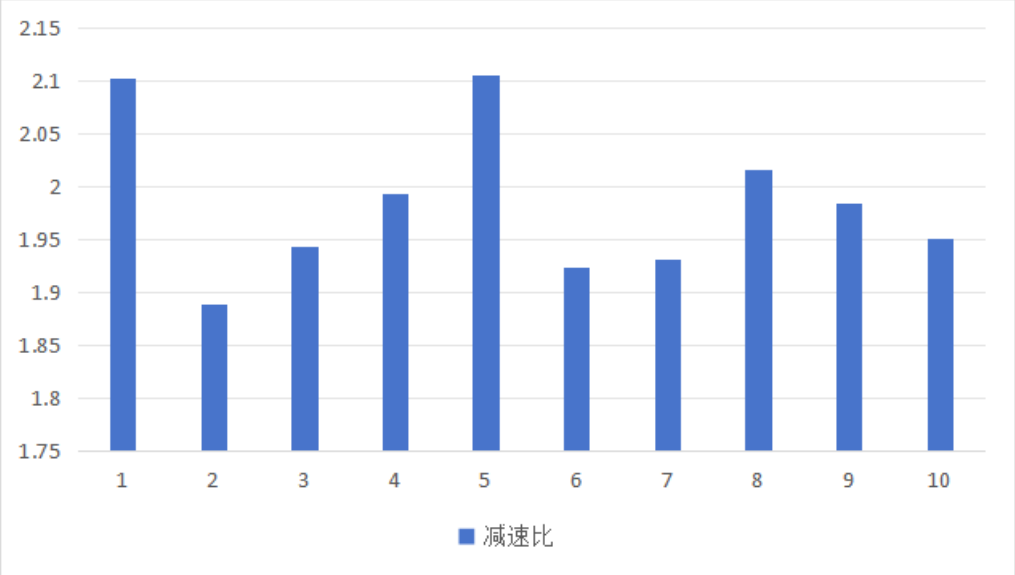


图 7.18: 每一个线程减速效果

线程数目方面为了实现探究线程数的对于优化效果的影响，我们进行了各种线程数目的实验，如下表:

线程数	2	4	5	8	10	20	25
总用时	78992.9	56430.4	38583.8	38098.2	27948.3	29392.9	29527.6

表 9: Pthread 线程数的影响

可以看到，各个线程数目会对于最后的用时产生影响。绘制出线程数目和用时的关系图如下:

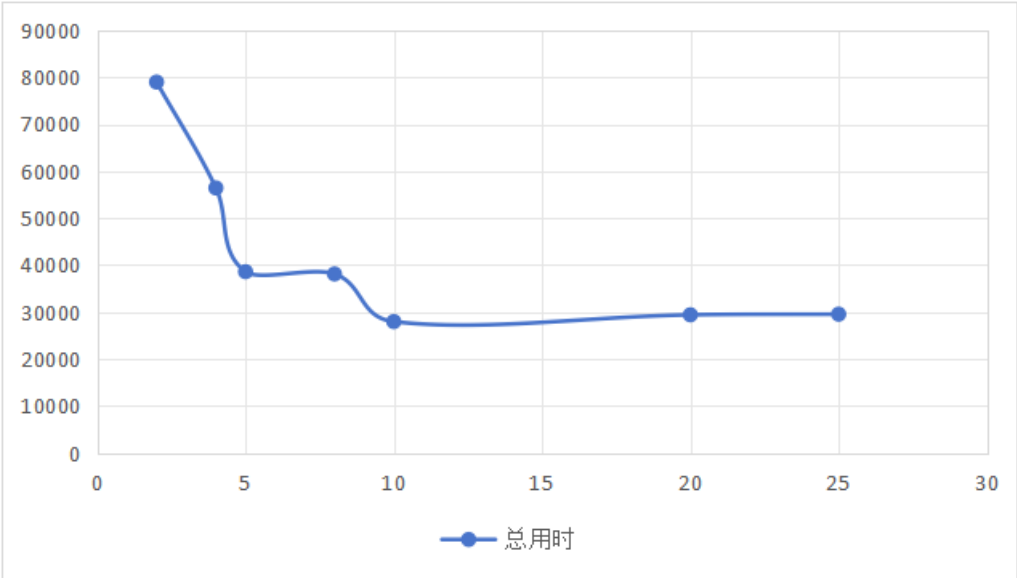


图 7.19: Pthread 用时随线程数变化

可以看到，随着线程数的增加，用时先减少后增加。可以看到当线程数小于 10 的时候，随着线程数目的增加速度快速提升，而过了 10 之后速度不再有太大的变化，而且有所减慢。

计算加速比如下：

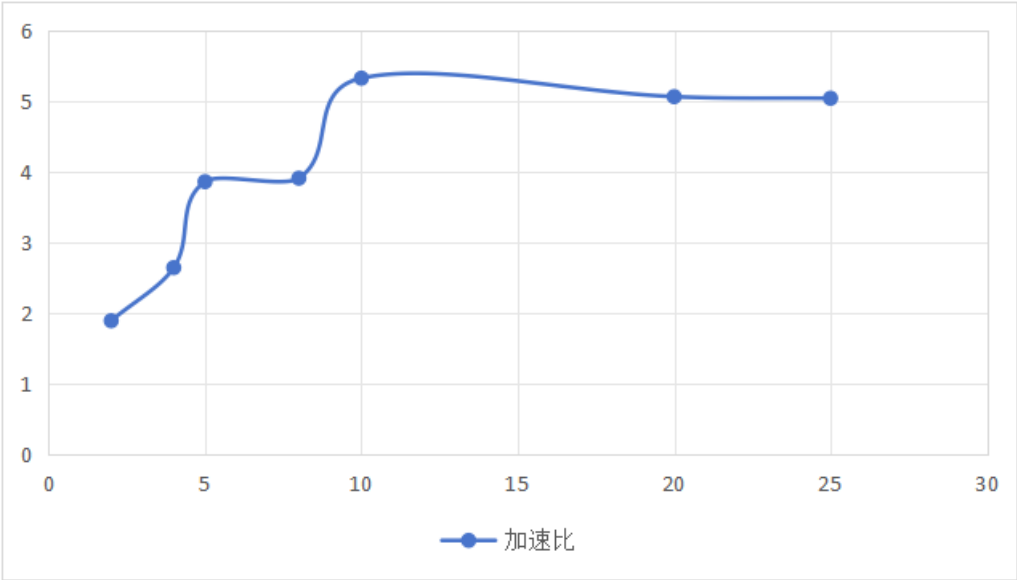


图 7.20: Pthread 加速比随线程数变化

可以看到，随着线程数的增加，加速比稳定在 5 左右。探索是因为线程管理代价和每个线程的负载不均导致的。

使用 openMP 进行多线程实验，得到运行结果和线程数目的关系如下表：

5	**10**	**20**	**25**	**40**	**50**
18359.1	**16538.7**	**12636.8**	**12438.9**	**12327.5**	**12283.8**

表 10: openMP 线程数 100 以内的耗时统计

100	200	500	750	800	825	850	900	1000
12224.9	12039.2	11975.7	11977.5	11938.6	11837.2	11844.3	11915.8	11962.6

表 11: openMP 线程数 100 以上的耗时统计

我们把两个数量级的图像绘制出来如下：

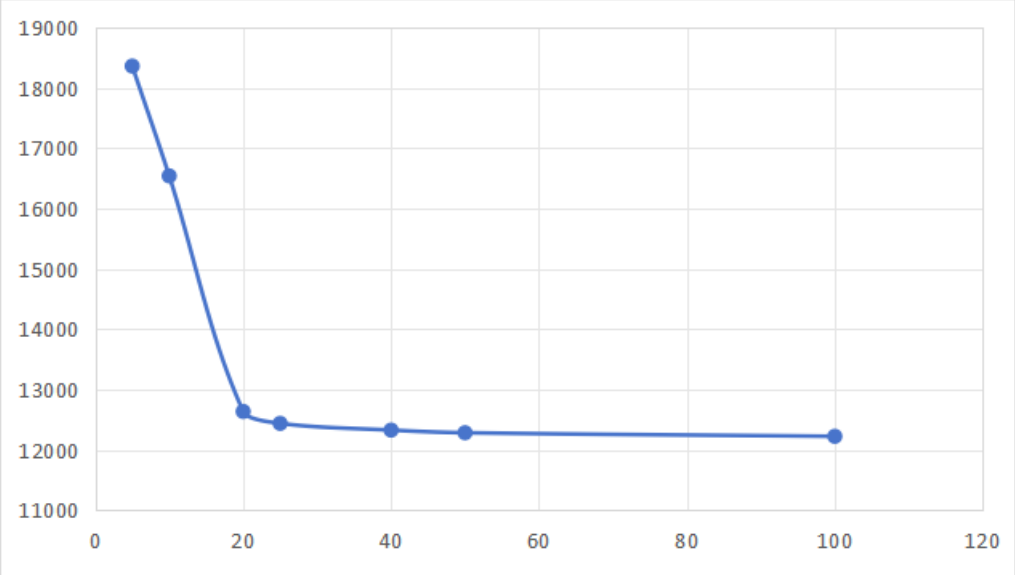


图 7.21: openMP 线程数 100 以内的变化图像

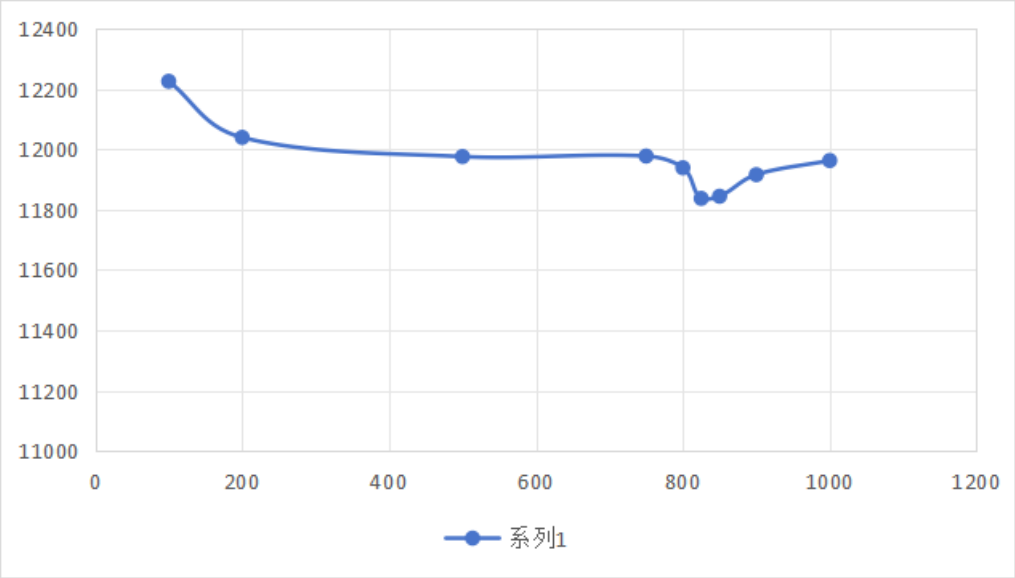


图 7.22: openMP 线程数 100 以上的变化图像

通过图像可以看出随着线程数目的增加，用时呈下降趋势。可以看到当线程数小于 100 的时候，随着线程数目的增加速度快速提升，而过了 100 之后速度不再有太大的变化，依旧在减慢，在 800 左右的时候，速度减慢到了最低点，之后有所上升。

对于 800-900 的数据进行细致测量，发现线程数 825 左右是最优的线程数目。这是多种因素共同作用的结果，是线程管理代价和并行优化的平衡点。

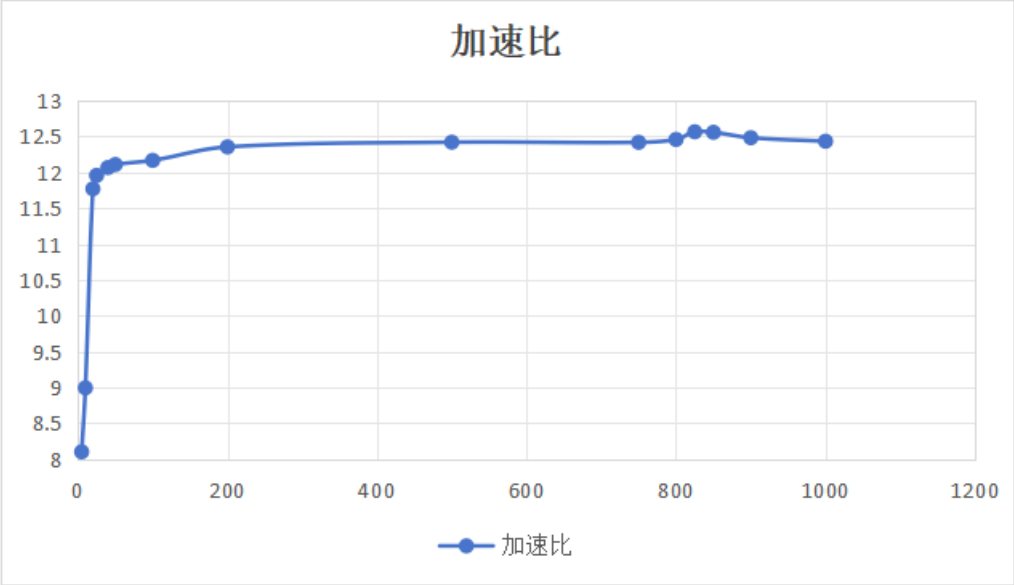


图 7.23: openMP 随加速比变化

可以看到，随着线程的增加，加速比最高能达到 12.56，已经是一个非常高的加速比了。最后随着线程数目的增加，加速比稳定在 12 左右，是这个算法的稳定加速比。

7.5.2 list-wise:Query 间和 Query 内并行的对比

不使用多线程，使用 Query 间并行和 Query 内并行的对比如下：

串行	**间 pthread**	**内 pthread**
14721	**30935.3**	**17228.4**
13625.7	25731.6	15894.4
14004.1	27202.1	16841.2
15632.8	31152.5	17979.4
14561.8	30650.5	15353.8
15156.2	29144.7	17470.8
15219.9	29380.5	15298.5
15001.5	30230.8	18028.3
15333	30427.7	17471.1
15500.7	30239.8	17599.8

表 12: 不同多线程策略的性能对比

其中，Query 间并行的使用十个线程分组进行，Query 内并行是串行进行每个 Query 但是每个 Query 内部进行多线程处理。Query 间并行的性能已经探索过，下面探索 Query 内并行的性能。

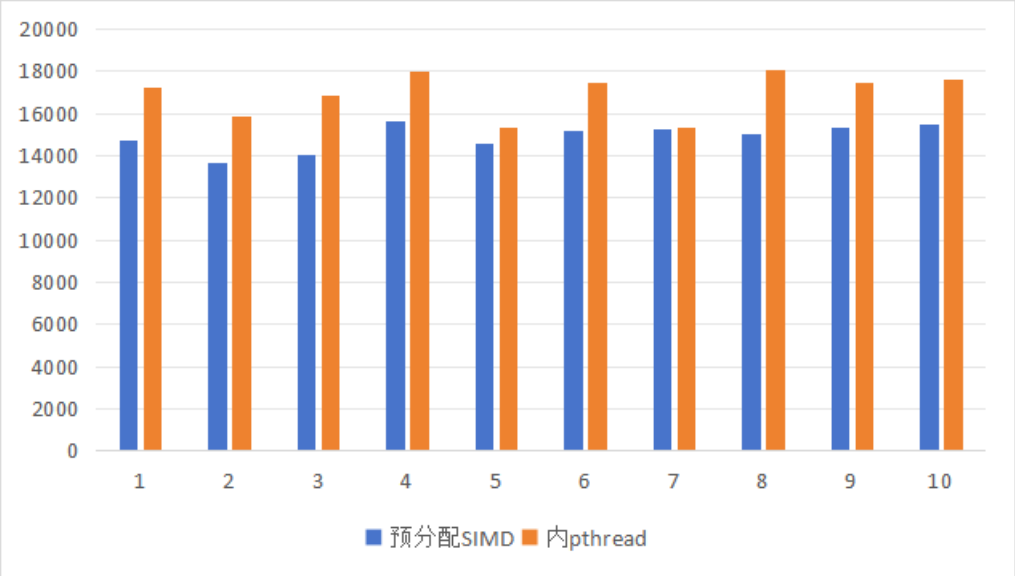


图 7.24: 每一组用时

我们惊讶的看到，每一个在主线程里面进行多线程处理的 Query 内并行的性能都差于普通的平凡算法。总体进行计算，加速比达到了 0.898，不但没有实现优化反而还更加慢了。

分析原因，就是对于内部多线程的处理太过繁琐。可优化的部分不多，而且频繁的线程创建销毁操作很频繁，开销很大，而且优化的范围很小，可能达不到理想的效果。对于可以进行多线程的四个或者五个向量，多线程的操作反而降低了性能增长了时间；对于两个或者三个向量，多线程的操作也没有优化效果，因为只需要一次或者两次求交操作。因此对于 Query 内并行按表求交的多线程操作，不适合使用多线程进行优化。

究其原因，是因为按表求交存在很大程度上的数据依赖。只有完成一个复杂的遍历求交运算之后产生新的变量才可以进行下一步，这样的算法让多线程的等待和通信开销很大，所以无法进行优化。

7.5.3 list-wise:Pthread 和 OpenMP 的对比

控制线程数相同，我们将 Pthread 和 OpenMP 的性能进行对比，如下表:

线程数	4	5	10	20	25
Pthread	56430.4	38583.8	27948.3	29392.9	29527.6
openMP	23403.7	18359.1	16538.7	12636.8	12438.9

表 13

根据这样的数据，我们可以绘制如下两条曲线来进行对比:

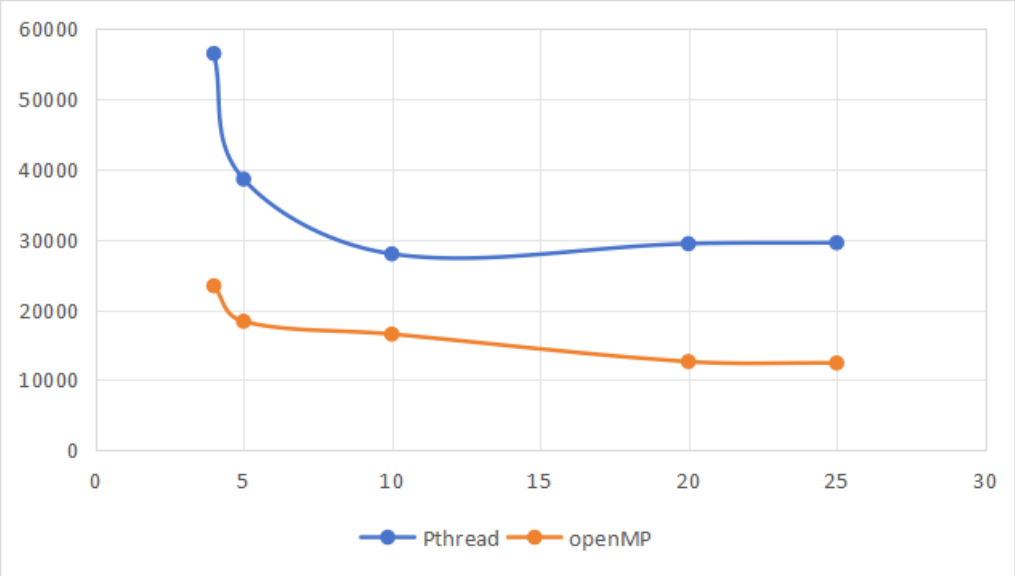


图 7.25: Pthread 和 OpenMP 的对比

可以看到，openMP 相较于 Pthread 有着显著的性能提升，两条曲线走向基本一致，计算加速比如下：

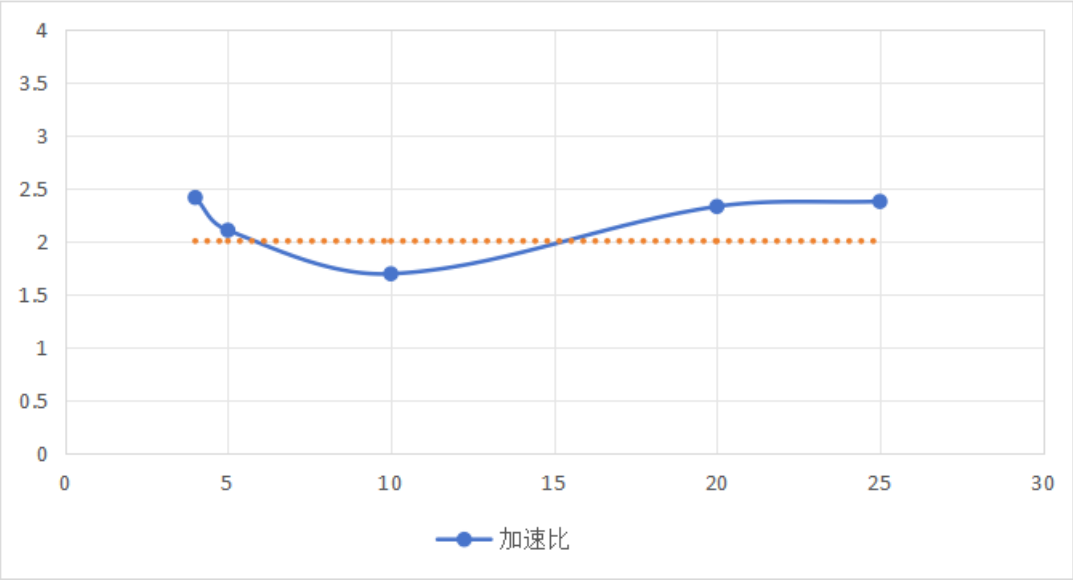


图 7.26: OpenMP 相对于 Pthread 的加速比

可以看到,OpenMP 相对于 Pthread 的加速比在 2 左右浮动，基本上与线程数目无关，分析是因为 openMP 实现的是线程的自动分配和销毁，对于多线程的优化与线程数目关系不大。

7.6 list-wise：二级索引位图

之前在 SIMD 优化的时候，我们使用了位使用二级索引位图来进行优化。优化效果十分显著。现在我们再次使用二级索引位图来实现优化。并且使用位图的存储方式，可以让 Query 内并行的有效优化成为可能。

7.6.1 二级索引 +Query 间 openMP 优化

首先我们使用上次实验的二级索引平凡算法和这次实验的 Query 间 openMP 进行结合:

二级索引加 omp

```
//初始化二级索引位图
#pragma omp parallel  num_threads(worker_count)
{
#pragma omp for
for(int i=0;i<DoNum;i++){
  //进行二级索引求交查询
}
}
```

改变线程数目 *worker_count*, 我们得到如下数据:

线程数目	omp+ 二级索引
10	9217.49
100	8308.83
200	8317.3
400	8414.4
825	8135.06
1000	8533.88

表 14

根据数据绘制图像如下;

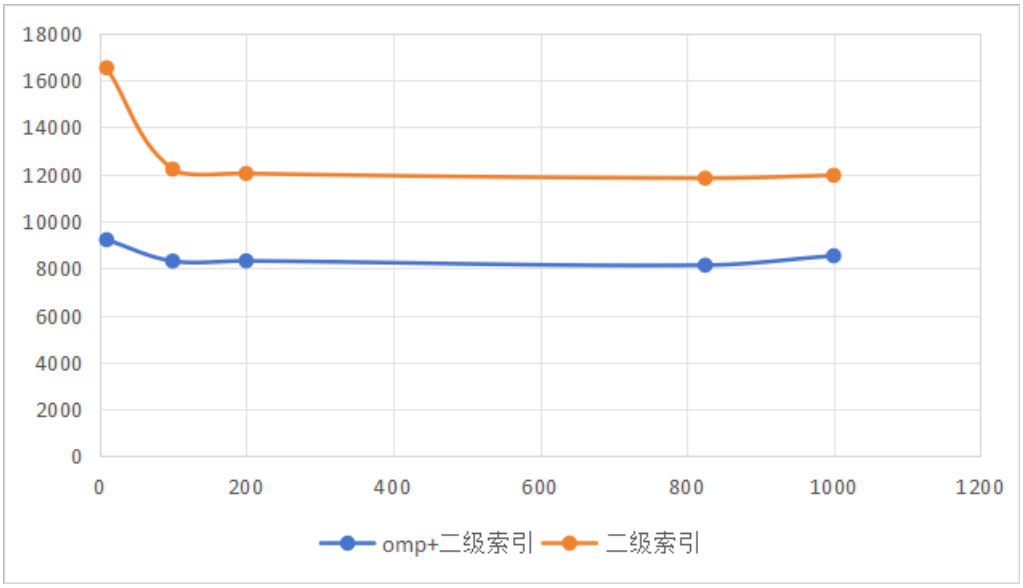


图 7.27: 二级索引位图的加速比

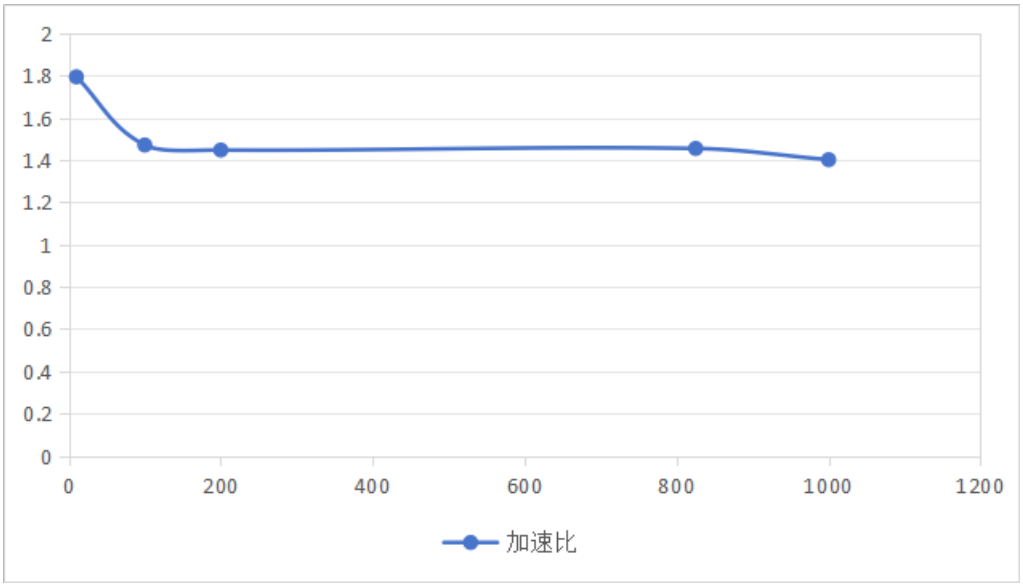


图 7.28: 二级索引位图的加速比

经过计算, 二级索引随着线程数增大，带来的加速比稳定在 1.4 左右。然后和上次 SIMD 优化中二级索引接近 7 的加速比来说有了很大的缩水。

究其原因，是因为线程数目增加，导致了线程管理的开销增加，而且二级索引位图的优化效果是对于每个查询而言的，对于每个查询的优化逐渐小于了线程管理的开销，导致了加速比的下降。最后到 1.4 达到平衡。不过和最平凡的算法来说总的加速比达到 27, 还是有着相当大的优化效果。

7.6.2 二级索引实现 Query 内多线程

二级索引本身为每个 Query 提供了一个独立的优化空间，我们可以使用 Query 内多线程的方式来进行优化。二级索引本身将位图分块，每个块可以独立进行处理，多线程的方式有了使用空间。

二级索引 +Query 内多线程

```
//初始化二级索引位图
#pragma omp parallel  num_threads(worker_count)
{
  for(int i=0;i<DoNum;i++){
    for (int j = 0; j < (*BasequeryData)[i].size(); ++j) {
      tempintersection = new bitset<MAX_SIZE>();
      tempsecondaryintersection = new bitset<MAX_SIZE/BLOCK_SIZE>();
      #pragma omp for
      for(int k=0;k<MAX_SIZE/ BLOCK_SIZE;k++){
        //进行二级索引求交查询

      }
    }
  }
}
```

利用二级索引，我们实现了 Query 内多线程的优化方式。

7.7 list-wise: 多线程的不同策略

在多线程的实现中，对于线程的管理和分配有着不同的策略。我们可以使用不同的策略来进行优化。openMP 可以自动分配线程，而 Pthread 需要手动分配线程，我们可以使用不同的线程分配策略来进行优化。

7.7.1 静态线程原理

我们之前使用的是动态线程的方式，分配任务、创建线程、完成任务，销毁线程。

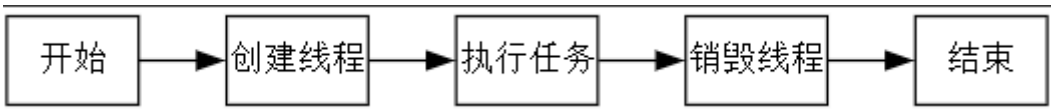


图 7.29: 二级索引位图的加速比

现在我们来尝试静态线程的方式，创建线程，分配任务，完成任务，销毁线程。这样的方式可以减少线程的创建和销毁的开销，提高线程的利用率。

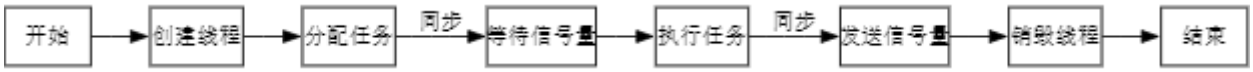


图 7.30: 二级索引位图的加速比

我们还使用了同步信号量的方式，来进行线程的同步，这样可以保证线程的同步，避免了线程的冲突，提高了线程的利用率。

7.7.2 静态线程代码实现

我们静态线程编程加上信号量的同步具体实现如下：

[H] **Algorithm 12** 静态线程信号量同步代码实现

$NUM\_THREADS \leftarrow 10$ ▷ 工作线程数量
$queryDataSize \leftarrow 1000$
$indexData \leftarrow$ 二维数组
$queryData \leftarrow$ 三维数组
$elapsedTime \leftarrow$ 一维数组，长度为 $queryDataSize/NUM\_THREADS$
$index \leftarrow 0$
$mtx \leftarrow$ 互斥锁

procedure THREADFUNC($param$)
    $t\_id \leftarrow param.t\_id$ ▷ 线程 id
    $t\_begin \leftarrow t\_id \times queryDataSize/NUM\_THREADS$ ▷ 计算任务的起始位置
    $t\_end \leftarrow (t\_id + 1) \times queryDataSize/NUM\_THREADS$ ▷ 计算任务的结束位置
    SEM_WAIT(sem_workerstart[t_id]) ▷ 等待工作线程开始信号
    进行任务的计算
    SEM_POST(sem_main) ▷ 发送主线程信号
    SEM_WAIT(sem_workerend[t_id]) ▷ 等待工作线程结束信号

```
    pthread_exit(NULL)
end procedure

procedure main
    sem_init(sem_main, 0, 0)                                  ▷ 初始化主线程信号量为 0
    for i ← 0 to NUM_THREADS − 1 do
        sem_init(sem_workerstart[i], 0, 0)                    ▷ 初始化工作线程开始信号量为 0
        sem_init(sem_workerend[i], 0, 0)                      ▷ 初始化工作线程结束信号量为 0
    end for
    handles ← 线程句柄数组，长度为 NUM_THREADS
    param ← 线程参数数组，长度为 NUM_THREADS
    for t_id ← 0 to NUM_THREADS − 1 do
        param[t_id].t_id ← t_id                               ▷ 设置线程参数的 t_id 成员
        pthread_create(handles[t_id], NULL, threadFunc, &param[t_id])   ▷ 创建线程并执行 threadFunc 函数
    end for
    for t_id ← 0 to NUM_THREADS − 1 do
        sem_post(sem_workerstart[t_id])                       ▷ 发送工作线程开始信号
    end for
    for t_id ← 0 to NUM_THREADS − 1 do
        sem_wait(sem_main)                                    ▷ 等待主线程信号
    end for
    for t_id ← 0 to NUM_THREADS − 1 do
        sem_post(sem_workerend[t_id])                         ▷ 发送工作线程结束信号
    end for
    for t_id ← 0 to NUM_THREADS − 1 do
        pthread_join(handles[t_id], NULL)                     ▷ 等待线程结束
    end for
    sem_destroy(sem_main)                                     ▷ 销毁主线程信号量
    for i ← 0 to NUM_THREADS − 1 do
        sem_destroy(sem_workerstart[i])                       ▷ 销毁工作线程开始信号量
        sem_destroy(sem_workerend[i])                         ▷ 销毁工作线程结束信号量
    end for
    return 0
end procedure
```

7.7.3 静态线程实现的性能对比

我们使用十个线程，与动态线程的方式进行对比，得到如下数据：统计成图像如下：

分组	动态	静态
0	30935.3	29288.4
1	25731.6	27032.5
2	27202.1	27069.6
3	31152.5	30328
4	30650.5	28370.7
5	29144.7	30221.1
6	29380.5	30684.2
7	30230.8	30669.5
8	30427.7	30587.9
9	30239.8	30887
TOTAL	31154	30888.5

表 15

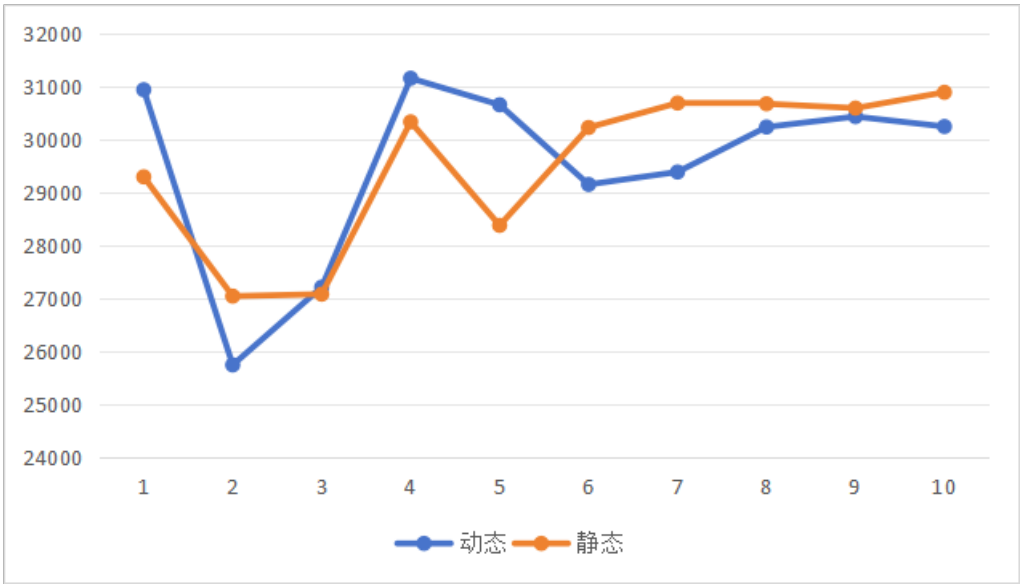


图 7.31: 动态和静态线程的对比

我们可以看到，两种并行策略不相上下。当然，静态线程的方式更加稳定，因为线程的创建和销毁开销更小，所以性能更加稳定。静态线程的负载更加均衡，最终相较于动态实现了一定的优化。

7.8 list-wise: 跨平台性能测试

不同的操作系统，指令集也很对于程序的性能产生影响。我们可以使用不同的操作系统和不同的指令集来进行性能测试。

7.8.1 硬件平台

x86 平台同上，ARM 平台采用如下配置：

- 华为鲲鹏服务器
- CPU 核心数：96
- 指令集架构：aarch64

- L1d cache: 64K
- L1i cache: 64K
- L2 cache: 512K
- L3 cache: 49152K

7.8.2 实验结果和分析

进行实验，得到如下数据：

	X86	ARM
0	30935.3	704434
1	25731.6	646447
2	27202.1	662234
3	31152.5	732710
4	30650.5	675752
5	29144.7	693618
6	29380.5	694131
7	30230.8	690083
8	30427.7	716946
9	30239.8	727831
TOTAL	31154	732710

表 16

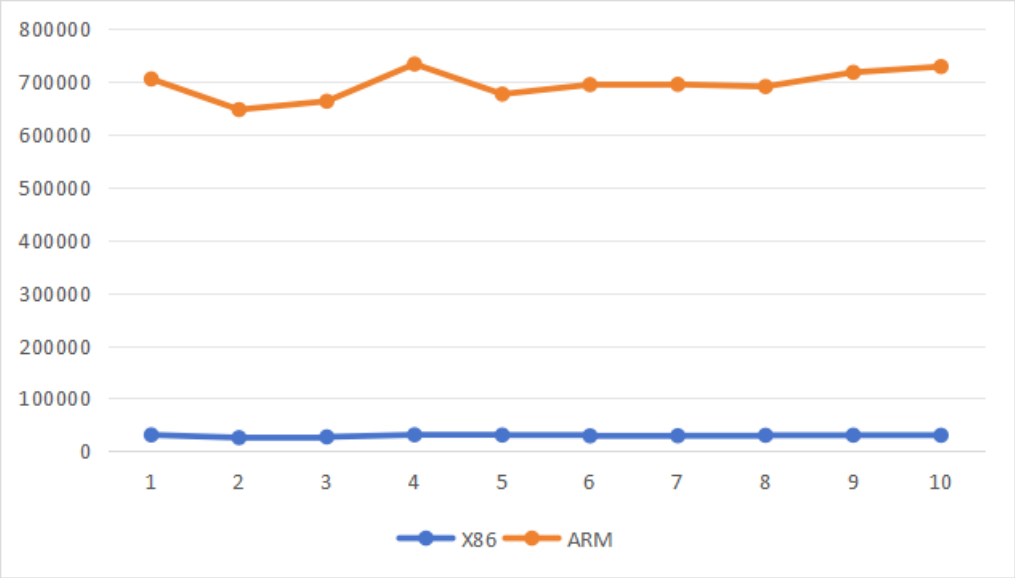


图 7.32: x86 和 ARM 的对比

可以看到，数据差的十分悬殊，ARM 平台的性能远远不如 x86 平台。这是因为 ARM 平台的指令集和 x86 平台有着很大的差异，ARM 平台的指令集更加简单，对于复杂的计算有着很大的影响。

7.9 element-wise 下 Pthread 的 Query 间动态并行算法

7.9.1 设计思路

由于倒排索引的查询文件中有一千条查询，而对于每个查询，其内部算法实现都是一样的，都是通过找出每个查询中的最短向量，进而与其他的向量进行求交运算。

所以我考虑了可以在每个查询间进行并行任务，将整个查询计划分组，每组分配一个线程进行查询。本次实验中有一千条查询，所以此处将查询分为十组，每组内有一百条查询，即开辟十条线程同时进行查询，每个线程负责其中一百条查询。

7.9.2 算法实现

算法在 x86 平台和 arm 平台的实现基本一致，只有在时间测量方面所用函数不同。如图7.33所示

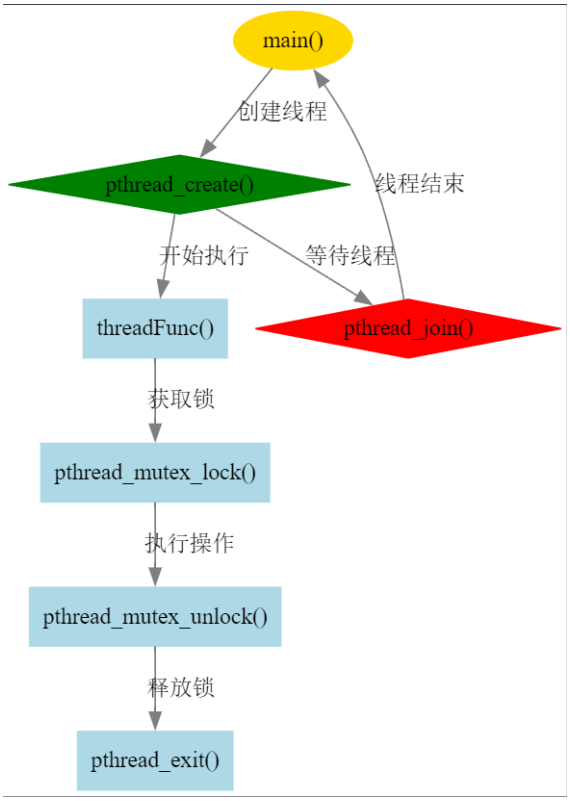


图 7.33: Query 间 Pthread 并行算法

7.9.3 时间测量

在 x86 平台上，采用 windows.h 中的 QueryPerformance 测量时间。在测量时间时，在每个线程运行内部记录运行时间，并根据其线程 id（t_id）将其存入向量 elapsedTime。并在主函数中测量全部线程运行的时间。

在 arm 平台上，采用 sys/time.h 头文件来进行计时，计时逻辑与在 x86 平台上一致，故不在阐述。

对于问题规模，由于本实验的查询数目是固定的，所以将其分为十组，即每组一百个查询。对于串行算法来说，每计算一百个记录一次时间；对于并行算法来说，给每组查询分配一个线程，即一共十个线程并行，分别记录每个线程运行的时间。这样就可以在问题规模这一维度上进行比较。

7.9.4 实验结果

本实验分别对比了 x86 和 arm 下两算法的性能，还对比了不同线程数下 pthread 并行算法的性能变化。

直接从数据即可看出，并行的效果非常显著，性能提升十分明显。此次实验 pthread 并行是以 10 个线程运行，用于和串行算法进行对比。

	x86 倒排索引求交串行/ms	x86 倒排索引求交 pthread 并行/ms
0	14580.1	17881.2
1	12348.2	15881.1
2	13367.5	16547.3
3	13317.3	17253.9
4	11774.5	15880.9
5	12883.4	16980.5
6	13084	17286.6
7	13228.1	17673.2
8	13349.9	17440.9
9	14319.8	18383
TOTAL TIME	132249	18384.2

表 17: x86

	arm 平凡倒排索引求交/ms	arm 平凡倒排索引求交 pthread 并行/ms
0	450000	449391
100	390000	390112
200	408000	408610
300	431000	430604
400	390000	390059
500	421000	421663
600	426000	425778
700	432000	432255
800	431000	430984
900	459000	458000
TOTAL TIME	4238000	458635

表 18: arm

7.9.5 实验结果比较及分析

当线程数过多时，x86。。。。。。。。

由于 pthread 并行时 10 个线程平均分配 1000 个查询，所以每个线程分配了一百个查询，正好可以与串行算法中每进行一百次查询记录的时间进行比较，可以比较出串行程序执行和多线程下每个线程的执行速度之间的差异。

图中横坐标表示 1000 组查询数据平均分为 10 组后的编号，100 则表示第 1-100 个查询，200 表示 101-200 个查询，以此类推。纵坐标表示这一组查询结束所用时间，以 ms 为单位。

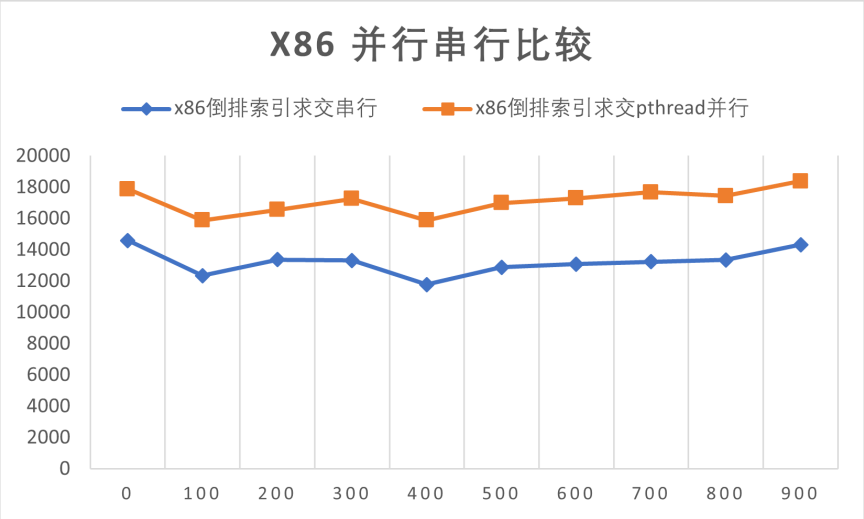


图 7.34: x86 并行串行比较

显然如图8.52所示，在 pthread 多线程并行时，对于每个线程自己分配的任务，要略慢于串行算法对应的部分。所以由上述表格可以看出，pthread 并行算法运行的总时长是 18384.2ms，而串行算法运行的总时长是 132249ms，加速比为 132249/18384.2 = 7.19362，并不到十倍。

但是在 arm 平台下多线程每个线程自己运行的时间与串行算法相应部分运行时间几乎一致，由于 arm 平台下二者太过于接近，故使用直方图来展示数据。

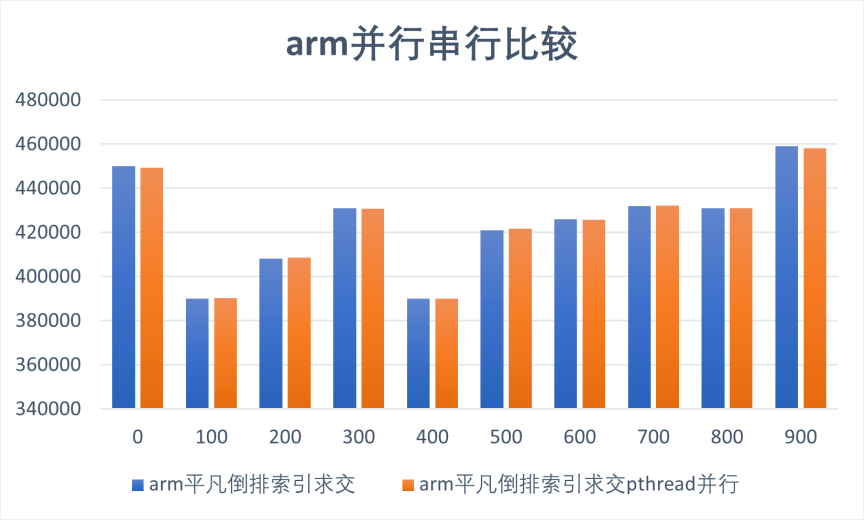
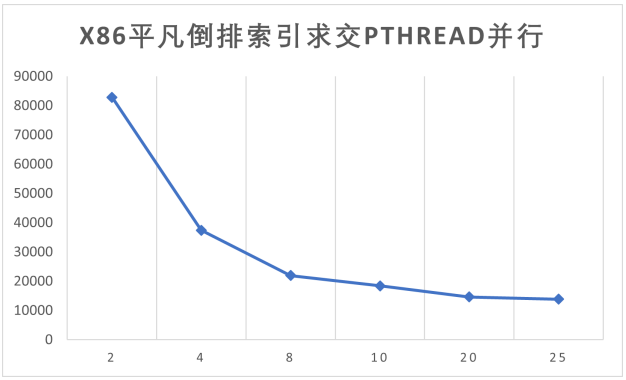


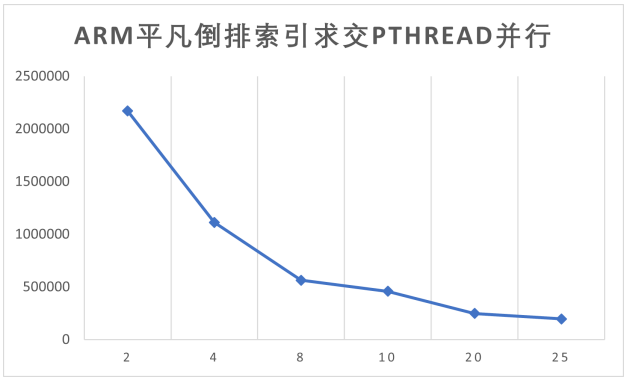
图 7.35: arm 并行串行比较

如下图对比，横坐标为开辟的线程数，纵坐标为运行的总时间（ms）。由于线程数需为 querydatasize 的因数，才能保证负载均衡，所以横坐标的变化并不均匀。

但是不难看出，随着线程数的增多，运行的总时间在逐渐下降，但是显然优化的力度在不断减弱。



(a) x86



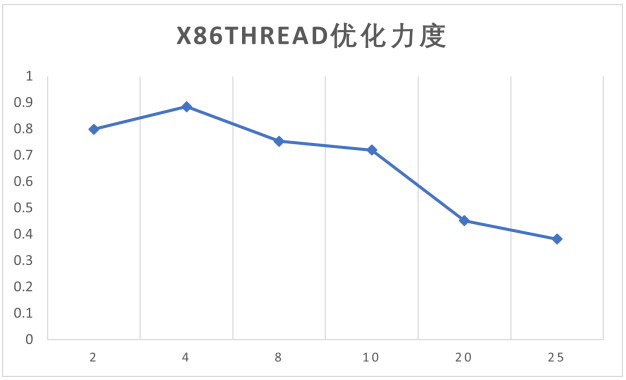
(b) arm

图 7.36: 不同平台串行优化算法的执行时间对比

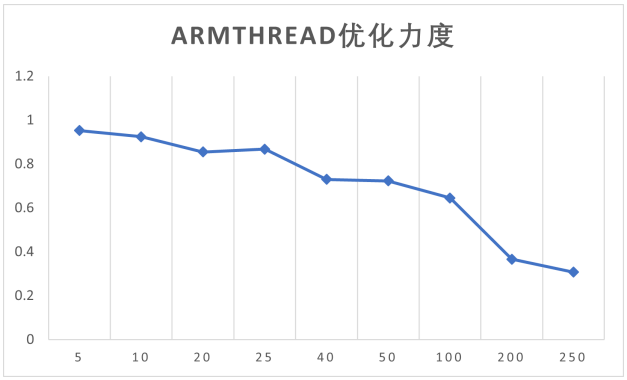
从理论上来说，线程数增加一倍，则总运行时间应该减少一倍，但是从上述不同线程的运行时间来看，显然不符合。所以为了直观看出线程数增加对程序性能优化真正的影响，引入：

$$\text{优化力度} = \frac{\text{串行算法运行时间}}{\text{并行算法运行时间} \times \text{线程数}}$$

显然优化力度越接近 1 表示情况越理想。可画出如下图



(a) x86



(b) arm

图 7.37: 不同平台 pthread 并行线程数

显然如图**??**和7.37所示，随着线程数的增多，多线程并行的优化效率在下降，这是因为线程创建、管理和同步都需要消耗一定的系统资源。当线程数增多到一定程度时，这些开销可能会超过并行处理带来的性能提升，导致总体运行速度下降。此外，互斥锁也会成为一个瓶颈。当一个线程在等待锁时，它不能做任何工作，显然会降低并行性。

其次硬件资源（例如 CPU 核心数）也会限制并行性。如果线程数超过 CPU 核心数，那么这些线程必须在核心之间进行切换，这也会带来一定的开销。

并且还可以注意到，在线程数小于 25 时，在 arm 平台上的优化力度均大于 x86 平台上的优化力度。而 ARM 通常使用更简单的指令集和更高效的能耗管理，这可能使其在某些并行任务上表现得更好。

7.10 element-wise 下 Pthread 的 Query 内动态并行算法

7.10.1 设计思路

在利用按元素求交的倒排索引求交串行算法时，我们首先是选出一个最短向量 S，接着让其他向量依次与 S 进行逐元素的比较，若未找到相同元素，则在 S 中删除相应元素，最终得到的 S 即为求交的结果。

由于在每组查询中有 2 到 5 个向量不等，我们可以不采用“依次”比较的方式，而是采用并行的方式。在每次外层循环进入当前的查询组时，计算查询组内部含有多少个查询向量，并为其分配等数目的线程数，让每个向量同时与 S 进行比较，并在删除 S 中元素时添加互斥锁，以防访问冲突。

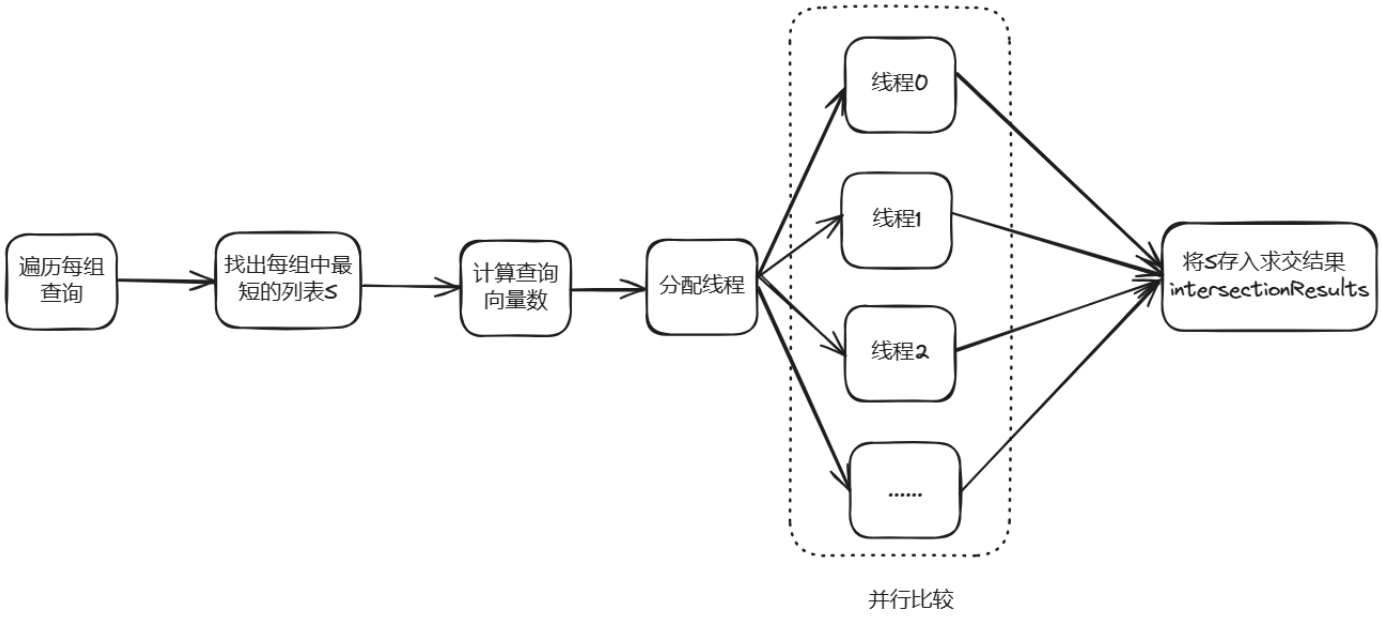


图 7.38: Query 内查询逻辑图

7.10.2 算法实现

注意若未找到共同元素，在 S 中删除相应元素时，应添加线程锁，以防不同线程访问冲突。并且将 S 存储求交结果向量时应为所有线程都结束后再存入。

```
1  void *threadFunc(void *param) {
2      threadParam_t *p = (threadParam_t*)param;
3      int t_id = p->t_id; //线程编号
4      vector<unsigned int> S=(*(p->S));
5      int i=p->i;
6      int k=p->k;
7              auto it = S.begin();
8              while (it != S.end())
9              {
10                 if (find(queryData[i][k].begin(), queryData[i][k].end(), *it) ==
                       queryData[i][k].end())    //
                       std::find函数没有找到等于*it的元素，它会返回lists[i].end()
11                 {
12                     pthread_mutex_lock(&mtx);
```

```
                    it = S.erase(it);//未找到即无交集，删除它
                    pthread_mutex_unlock(&mtx);
                }
                else
                {
                    it++;
                }
            }
            cout << i<<" k:"<<k<<endl;

        pthread_exit(NULL);
        return nullptr;
}
```

7.10.3 时间测量

利用 <chrono> 头文件来进行时间测量，在 arm 平台和 x86 平台上都可以用，将时间设置为以 ms 为单位。

```
std::chrono::duration<double, std::milli> TotalTime = end-start;
```

由于在 Query 内进行并行，线程数是由查询计划中当前组决定的，并不固定，所以使用数组或容器来存储每个线程运行所花费的时间没有任何意义。所以在这里我依旧选择了将大小为 1000 的查询计划分为十组，每组一百个查询，记录每一百组查询的用时，这样即可在问题规模上进行比较。

7.10.4 实验结果

	x86 倒排索引求交串行/ms	x86pthreadQuery 内并行/ms
0	14580.1	23851.6
100	12348.2	26682.6
200	13367.5	21373
300	13317.3	22821.6
400	11774.5	25715.8
500	12883.4	24720.1
600	13084	24643.4
700	13228.1	32704.2
800	13349.9	28723.9
900	14319.8	26432.6
TOTAL TIME	132249	274567

表 19: x86

	arm 倒排索引求交串行/ms	armpthreadQuery 内并行/ms
0	450000	596764
100	390000	631174
200	408000	530439
300	431000	556210
400	390000	602875
500	421000	569952
600	426000	580112
700	432000	678330
800	431000	633333
900	459000	612724
TOTAL TIME	4238000	6030590

表 20: arm

直接观察数据可以看出，在运行总时间上，pthread 在 query 内并行的算法的执行时间甚至超过了串行算法的执行时间，完全没有起到优化的作用。由于运行速度过慢，故不在与 pthread 在 query 间并行算法进行比较。

7.10.5 实验结果比较及分析

画折线图如下，可以更直观的看出随问题规模的变化以及运行时间的长短。

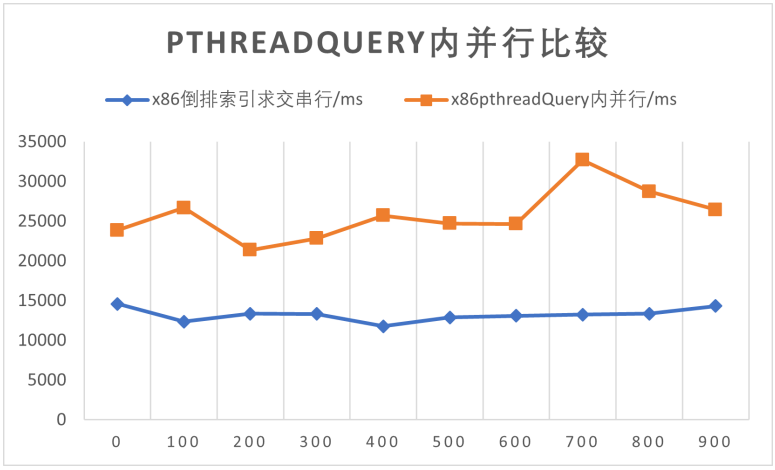


图 7.39: x86pthreadQuery 内并行比较

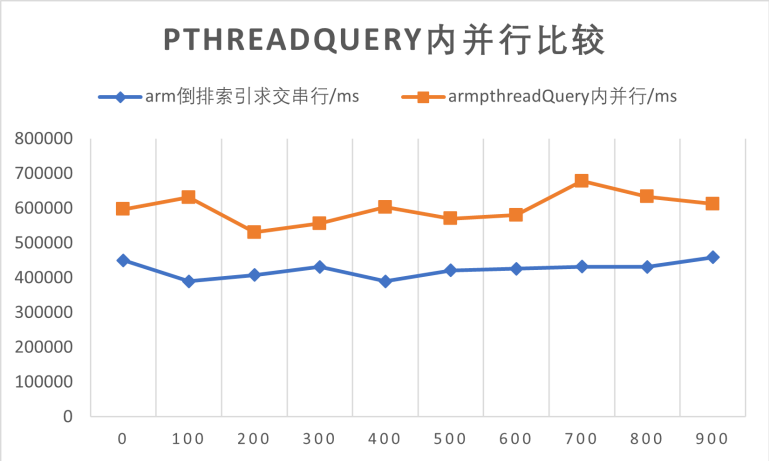


图 7.40: armpthreadQuery 内并行比较

观察图8.55和图8.53可得，在所有问题规模即 1000 条查询分为的 10 组上，并行算法运行时间均高于串行算法的运行时间。

究其原因，对于此次实现的代码，很重要的一点就是线程开销问题。并行计算需要创建和管理多个线程，这本身就有一定的开销。如果任务本身很小，那么这种开销可能会超过并行带来的性能提升。

在 query 内进行并行，每次只能对 2 到 5 个向量不等，来进行并行计算，并且对于每组查询，在开始前都会计算并行线程的数量，再依此开辟线程，进行多线程计算，所以线程创建和销毁的开销很大，极大程度上影响了并行的效率，甚至效率还不如串行算法。

还存在同步问题，在并行计算中，线程之间可能需要进行数据同步。最短向量 S 作为一个共享向量可能会被不同向量同时进行访问，所以必须在向量访问时添加线程锁，这样可能就会造成线程等待的问题，也会影响运行效率。

7.11 element-wise 下 Pthread 的 Query 间动态并行算法

7.11.1 设计思路

OpenMP 并行化实现较为简单。只需要在代码中添加几个指令，就可以实现并行执行，编程简单也是 OpenMP 的特点。

利用 OpenMP 进行并行化与 Pthread 并行化类似，都可以其分为 Query 间并行和 Query 内并行。

7.11.2 算法实现

只需在遍历全部列表之前加上以下代码，则编译器和系统会自动为其中的代码分配线程数，进行并行操作，worker_count 表示要为其分配的线程数量。

#pragma omp parallel 后跟 for 循环即可。

```
1 #include <omp.h>
2 void main() {
3 #pragma omp parallel  num_threads(worker_count)
4 {
5     #pragma omp parallel
6     {
```

```
        // 这里的代码将由多个线程并行执行
    }
}
}
```

7.11.3 实验结果

以下为不同线程数下，代码完成倒排索引求交操作所用时间

	x86 倒排索引求交 openMP 并行	arm 倒排索引求交 openMP 并行
50	13750.6	117311
100	13692.5	65090
150	13537.8	62927.7
200	13430	59457.6

以下是以 100 个线程为基准，OpenMP 并行算法、pthread 并行算法、串行算法在不同平台上运行的时间比较

	x86 倒排索引串行	x86 倒排索引 openMP 并行	x86 倒排索引 pthread 并行
TOTAL TIME/ms	132249	13692.5	18384.2

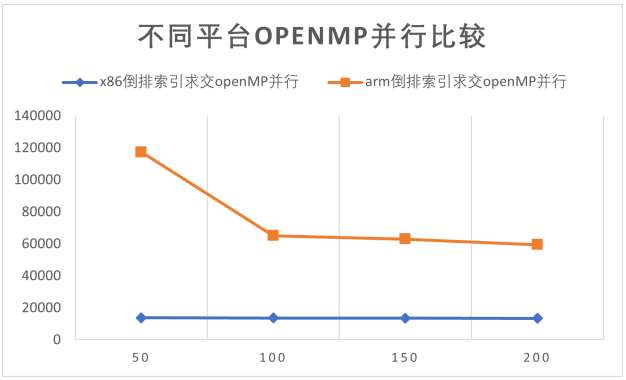
	arm 倒排索引求交	arm 倒排索引 openMP 并行	x86 倒排索引 pthread 并行
TOTAL TIME/ms	4238000	65090	458635

7.11.4 实验结果比较及分析

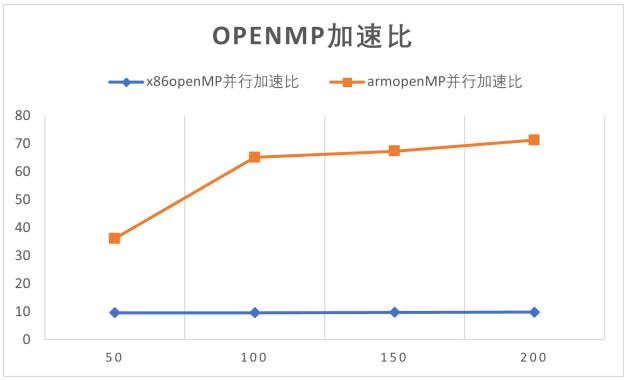
我们将线程数作为横坐标，运行时间作为纵坐标，画出如图所示的折线图。

可以明显看出，在 x86 平台上，随着线程数的增加，程序的执行时间基本上维持不变，并没有因为并行的线程数增加而运行速度变快。

而在 arm 平台上，随着线程数的增加，可以看出运行时间是逐渐递减的，说明并行起到了相应的效果。但是随着线程数的增加，下降的速率在减小，也就是优化的效果在减弱。



(a) OpenMP 加速比



(b) OpenMP 加速比

加速比为串行算法运行时间/并行算法运行时间，我们可以看出 x86 平台上的加速比非常低，始终只比串行算法优化了 0.1 倍。但是我们从图中看不出线程对运行时间的影响，所以我们依旧使用优化力度来进行探讨

$$\text{优化力度} = \frac{\text{串行算法运行时间}}{\text{并行算法运行时间} \times \text{线程数}}$$

如图7.41, 可以看出，随着线程数的增加，arm 平台上和 x86 平台上的优化效率都在逐渐降低，这与之前 pthread 并行时的原因应该是相同的：当线程数增多到一定程度时，这些开销可能会超过并行处理带来的性能提升，导致总体运行速度下降，最终达到性能提升的瓶颈。但是 arm 平台的优化效率始终远远高于 x86 平台。

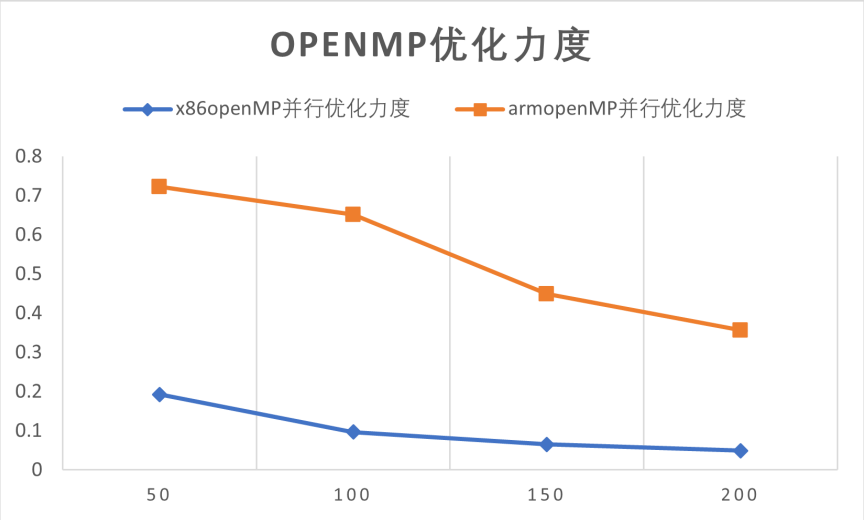


图 7.41: OpenMP 优化力度

在 x86 平台上表现如此不尽如人意，可能是因为资源竞争的现象，多个线程需要访问同一向量 S，它们可能需要等待，这会导致线程阻塞，从而降低并行性；或者是因为硬件的限制可能无法支持大量的并行线程，在 x86 平台上，我的 cpu 只有 14 个核心，而鲲鹏服务器上有 96 个核心。

7.12 element-wise 下 Pthread 的 Query 内动态并行算法

7.12.1 设计思路

与 pthread 在 query 内的并行思路类似。在选出每个查询的最短向量 S 后，我们需要将其与当前查询中其他向量进行对比以删除不在其中的元素，而每个查询中又 2 到 5 个向量不等，所以我们采用 openMP 进行并行处理，对这 2 到 5 个向量分配线程，一同与 S 进行比对。

#pragma omp for 指令会将循环的迭代分配给不同的线程。这个分配会基于循环的初始条件和结束条件，在循环开始时进行。即使 queryData[i].size() 的值在每次迭代中可能出现不同，OpenMP 仍然可以正确地将循环的迭代分配给不同的线程。

7.12.2 算法实现

在实现算法时遇见了一些 bug，应注意，对于 S 的访问时线程不安全的，程序在并行区域内部修改了共享数据结构 S。这可能会导致数据竞争，多个线程可能在同时尝试修改 S，这也将导致访问错误。

所以在这里我采用 #pragma omp critical 来创建一个临界区，这个临界区在任何时候只能由一个线程进入。这可以用来保护对共享数据的修改，以防止数据竞争。

Algorithm 13 Parallel Intersection

```
1: parallel num_threads(worker_count)
2: for k = 0 to size(queryData[i]) - 1 do in parallel
3:     local_S = S                                   ▷ Critical section
4:     for each element it in local_S do
5:         if it not in queryData[i][k] then
6:             remove it from local_S
7:         end if
8:     end for
9:     S = local_S                                   ▷ Critical section
10:    print i, " k:", k
11: end for
```

7.12.3 实验结果

首先进行横向对比，我们比较算法用时的总时长。如下表，可以看出这三个算法中，OpenMPQuery 间并行的速度是最快的，最慢的是 OpenMPQuery 内并行算法。其中 OpenMPQuery 间并行是以线程数为 100 为基准的。

	x86 串行/ms	x86OpenMPQuery 内并行/ms	x86OpenMPQuery 间并行/ms
TOTAL TIME	132249	174708	13692.5

再进行纵向对比，与上文 pthreadquery 内并行算法测量时间的思路一样，均测量 1000 个查询中每一百个查询的时间，进行问题规模上的比较。如下图，我们对比串行算法与 OpenMPQuery 内并行算法在每组的不同表现，可以看出，每组的用时较为均衡，并且每组的用时，串行算法都快于并行算法。

	x86 倒排索引求交串行/ms	x86OpenMPQuery 内并行/ms
0	14580.1	11771.5
100	12348.2	17045.2
200	13367.5	17219.2
300	13317.3	18195.2
400	11774.5	16420.3
500	12883.4	17931.1
600	13084	18267.8
700	13228.1	18741.9
800	13349.9	18583.1
900	14319.8	20532.6

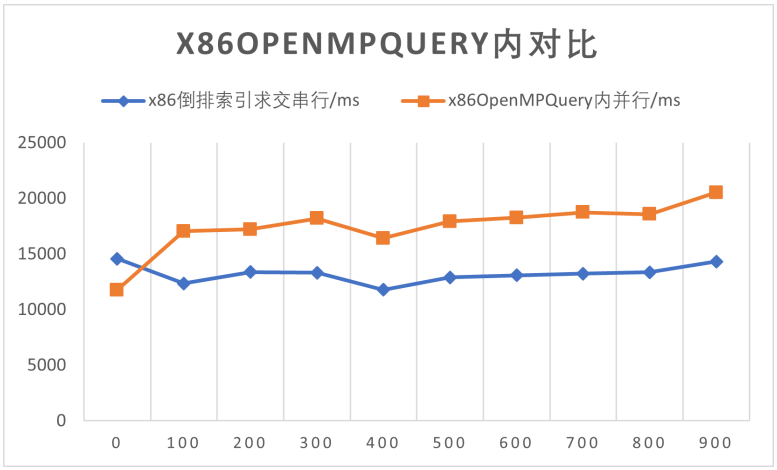
表 21: x86

	arm 平凡倒排索引求交	armopenMP 并行 Query 内/ms
0	450000	310597
100	390000	270236
200	408000	286372
300	431000	300459
400	390000	264791
500	421000	291883
600	426000	293969
700	432000	297715
800	431000	300096
900	459000	315035
TOTAL TIME	4238000	2931150

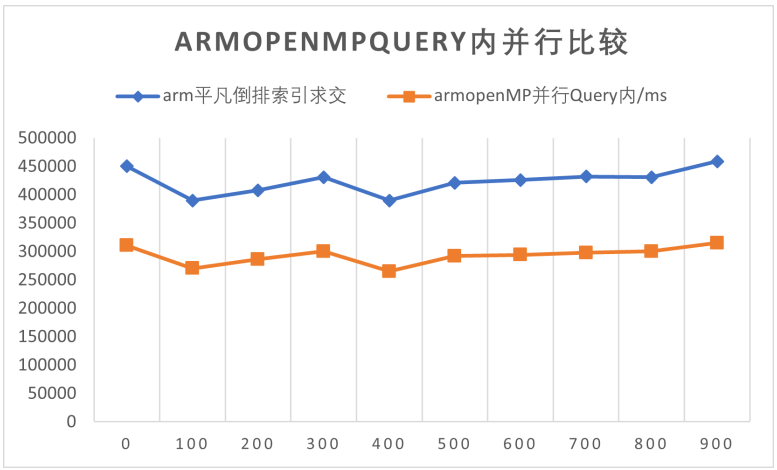
表 22: arm

7.12.4 实验结果比较及分析

可以看出除了在第一组查询中并行速度快于串行的速度外，其余组均是串行更快。



在 arm 架构上，openmpquery 内并行起到了优化的效果，



7.13 element-wise 下 Pthread 和 OpenMP 并行效果对比

7.13.1 Query 间

这里整理一下之前的测试数据画出表格，对比在线程数相同的情况下，pthread 和 openmp 并行的运行时间

	x86 倒排索引求交 pthread 并行	x86 倒排索引求交 openmp 并行
2	69551.6	68702.1
5	42429.3	31055.1
10	19095.7	18236.2
20	14645.1	14303.7
25	13852.2	13839.1

由上述表格画出折线图，可以直观的看出，二者在问题规模上和线程数上的表现均几乎一致，两条曲线重合度十分高，说明在 x86 架构下，二者的并行优化效果几乎一致，并且随着线程数的增多，二者的优化效率的衰减也是十分相近的。

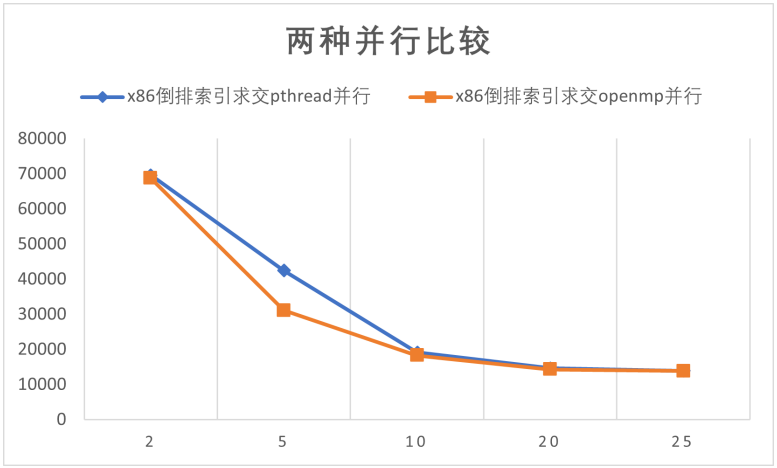
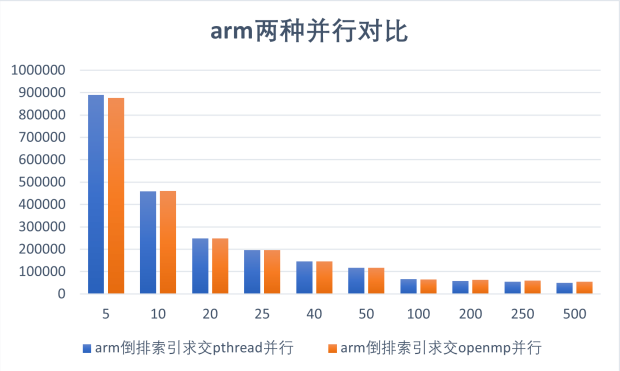


图 7.42: x86Query 间两种并行比较

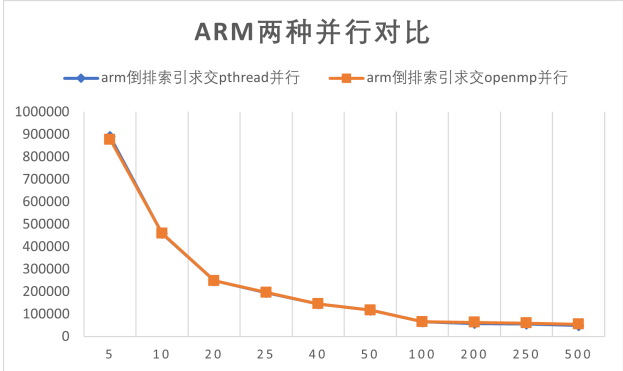
在 arm 平台上的测试数据如下

	armpthreadQuery 间并行/ms	armopenmpQuery 间并行/ms
5	889676	876953
10	458635	459612
20	247881	248479
25	195452	196044
40	145129	145406
50	117281	117311
100	65710.7	65090
200	57955.1	62927.7
250	55127.8	59457.6
500	48775	54777.1

依据上述表格可做出如下条形图。



(a) armQuery 间两种并行比较



(b) 变化趋势

由于在两个平台上不同线程数下测得的时间非常接近，用折线图绘制二者的曲线几乎重合，所以加一个条形图绘制。从条形图可以看出，二者对比下在有些线程数下 openmp 略胜一筹。而从折线图可以看出，随着线程数的增多，运行时间确实是在逐渐下降，但是可以明显看出，运行时间在**趋于一个特定的值**，随着线程数的增加，优化效果越来越不明显。

对比优化力度可以更加直观的看出。如下图, 在开始时，优化程度接近理想情况，而随着线程数的增加，优化力度逐渐走低，优化效率逐渐降低。

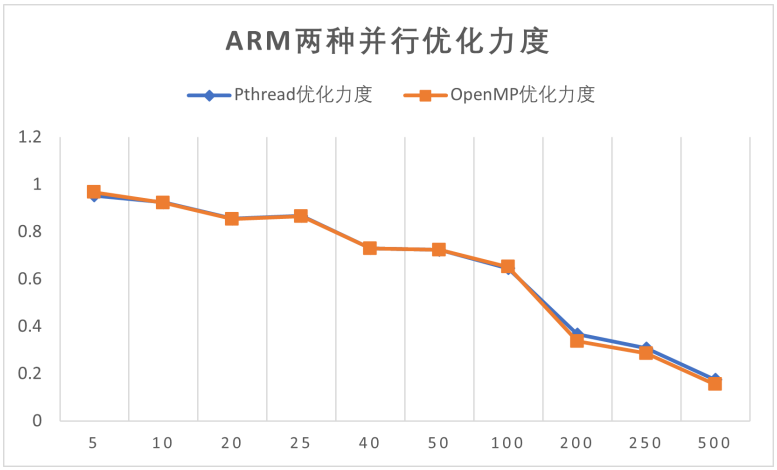


图 7.43: arm 两种并行优化力度

程序运行时间趋于稳定值有可能是因为处理器只有一定数量的核心，那么即使创建了更多的线程，也只有有限的线程能够同时运行。此时如果增加线程数并不能提高并行度，反而可能因为线程切换的开销导致性能下降。并且线程切换本身就会带来开销，如果线程数量过多，这些开销可能会抵消并行带来的性能提升。

7.13.2 Query 内

由于 Query 内的并行线程数是由问题本身决定的，即查询文件中需要进行求交的向量数量，所以无法从线程数的多少上进行比较。因此我们从问题规模上进行比较。

	x86 倒排索引求交串行/ms	x86OpenMPQuery 内并行/ms	x86pthreadQuery 内并行/ms
0	14580.1	11771.5	23851.6
100	12348.2	17045.2	26682.6
200	13367.5	17219.2	21373
300	13317.3	18195.2	22821.6
400	11774.5	16420.3	25715.8
500	12883.4	17931.1	24720.1
600	13084	18267.8	24643.4
700	13228.1	18741.9	32704.2
800	13349.9	18583.1	28723.9
900	14319.8	20532.6	26432.6

表 23: x86

	x86 倒排索引求交串行/ms	x86OpenMPQuery 内并行/ms	x86pthreadQuery 内并行/ms
TOTAL TIME	132249	174708	274567

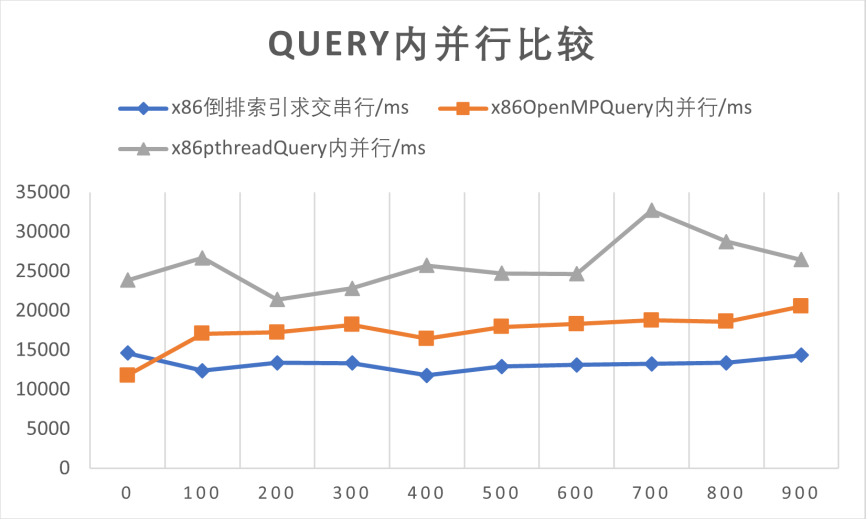


图 7.44: x86query 内并行比较

可以明显看出，串行算法是最快的，对于并行算法，pthread 在 query 内并行是三者中间最慢的，慢于 openmp，可能是因为 pthread 创建管理线程过于复杂和麻烦，加上 query 内进行并行线程大部分只有两个，并不能起到太大的并行效果，而且每次循环都需要重新分配线程，线程开销极大。

在 arm 架构运行的数据如下，通过观察三者的运行总时间，可以看出在 arm 上，openMPQuery 内并行确实起到了优化的效果，而 pthreadQuery 内并行与在 x86 架构上一样起到了适得其反的效果。说明 openMP 对于线程的管理更加科学，减小了开销。

	arm 平凡倒排索引求交	armopenMPQuery 内并行/ms	armpthreadQuery 内并行/ms
0	450000	310597	596764
100	390000	270236	631174
200	408000	286372	530439
300	431000	300459	556210
400	390000	264791	602875
500	421000	291883	569952
600	426000	293969	580112
700	432000	297715	678330
800	431000	300096	633333
900	459000	315035	612724
TOTAL TIME	4238000	2931150	6030590

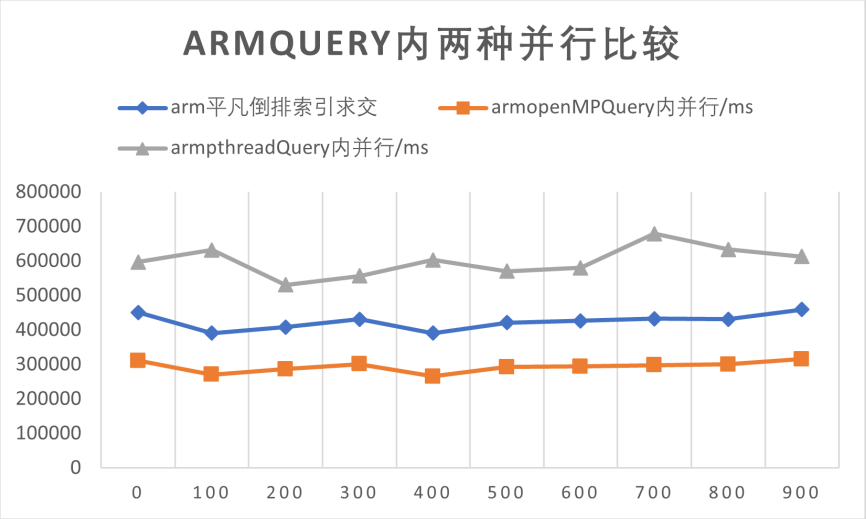


图 7.45: armquery 内两种并行比较

7.14 element-wise 下总结

本次实验内容十分丰富，形成了这份颇为厚重的实验报告。本次实验除了实现了基本要求，还实现了进阶要求中不同并行策略的比较、以及在不同平台上测试了代码的运行效率，并且讨论一些基本的算法/编程策略对性能的影响，以及 pthread 与 openmp 的性能差异。

在编写本次实验的代码中遇到了不少 bug，其中就包括在并行过程中对于共享变量的访问，需要添加线程锁以确保线程安全。同时也研究了如何在倒排索引求交 element-wise 算法上进行并行，考虑了很多并行策略，如在每个 query 间并行和在每个 query 内并行，有些失败，有些成功。

这次实验加深了我对于多线程的理解以及多线程编程的应用能力，也了解了 pthread 和 openmp 在进行编程时的异同，了解了二者的基本用法，以及并行程序测试性能、进行比较的方式。

8 MPI 编程设计

8.1 list-wise:MPI 算法设计

8.1.1 初始化 MPI 环境和获取进程信息

在并行计算中，MPI（Message Passing Interface）用于在不同的进程间进行通信和协调。以下代码段初始化了 MPI 环境，并获取了全局通信器中进程的数量和每个进程的排名。

```
MPI_Init(&argc, &argv);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

性能分析：这部分代码的复杂度为 $O(1)$，因为这些操作都是常数时间的。初始化 MPI 环境和获取进程信息是固定开销，不随数据规模变化。

8.1.2 主进程读取索引文件

主进程负责读取索引文件，该文件包含了倒排索引的数据结构，用于加速查询处理。以下代码段展示了主进程读取索引文件的操作。

```
if (world_rank == 0) {
    ifstream indexFile("D:\\ExpIndex", ios::binary);
    // ...
}
```

性能分析：这部分代码的复杂度为 $O(n)$，其中 n 为索引文件的大小。因为需要读取整个文件，所以时间复杂度与文件大小成正比。文件读取操作可能成为整个算法的瓶颈，尤其是对于大型索引文件。

在 MPI 中，广播操作用于将数据从一个进程发送到所有其他进程。这里主进程读取的索引数据需要广播给所有其他进程。

8.1.3 广播索引数据

将索引数据的大小广播到所有进程，然后广播实际的索引数据。

```
int indexDataSize = indexData.size();
MPI_Bcast(&indexDataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
// ...
```

性能分析：这部分代码的复杂度为 $O(p)$，其中 p 为进程的数量。因为需要将数据广播到所有进程，所以时间复杂度与进程数量成正比。广播操作在并行计算中很常见，能有效分配数据，但也会增加通信开销。

8.1.4 主进程读取查询文件

主进程同样需要读取查询文件，该文件包含了一系列查询，每个查询将使用之前读取的索引数据进行处理。

```
if (world_rank == 0) {
    ifstream queryFile("D:\\ExpQuery");
    // ...
}
```

性能分析：这部分代码的复杂度为 $O(m)$，其中 m 为查询文件的大小。因为需要读取整个文件，所以时间复杂度与文件大小成正比。类似于索引文件读取，查询文件读取也可能影响整体性能，特别是在查询文件较大的情况下。

8.1.5 广播查询数据

将查询数据的大小广播到所有进程，然后广播实际的查询数据。

```
int queryDataSize = queryData.size();
MPI_Bcast(&queryDataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
// ...
```

性能分析：这部分代码的复杂度为 $O(p)$，其中 p 为进程的数量。因为需要将数据广播到所有进程，所以时间复杂度与进程数量成正比。这一步确保所有进程都能获取到完整的查询数据，为后续的并行查询处理做准备。

在并行计算中，查询处理任务被分配给多个进程，每个进程处理一部分查询数据。以下代码段展示了查询处理和收集结果的操作。

8.1.6 处理查询

每个进程根据其排名，确定需要处理的查询范围，然后执行查询处理。

```
int startQuery = world_rank * queriesPerProcess;
int endQuery = min(startQuery + queriesPerProcess, queryDataSize);
// ...
```

性能分析：这部分代码的复杂度为 $O(\frac{q}{p})$，其中 q 为查询的数量，p 为进程的数量。因为查询被平均分配到所有进程，所以每个进程的处理时间与查询数量和进程数量的比值成正比。任务的均匀分配有助于提高并行效率，但实际性能也取决于查询的复杂度和数据分布。

8.1.7 收集并输出处理时间

每个进程将其处理的时间发送给主进程，主进程负责收集所有进程的处理时间并输出。

```
MPI_Gather(&localElapsedTimeMs, 1, MPI_DOUBLE, elapsedTime.data(), 1, MPI_DOUBLE, 0,
    MPI_COMM_WORLD);
// ...
```

性能分析：这部分代码的复杂度为 $O(p)$，其中 p 为进程的数量。因为需要收集所有进程的处理时间，所以时间复杂度与进程数量成正比。收集结果的操作虽然简单，但在高并发场景下，通信开销可能会显著增加。

8.2 list-wise: 总体性能分析

综合上述分析，这段代码的总体时间复杂度为 $O(n+m+\frac{q}{p})$，其中 n 为索引文件的大小，m 为查询文件的大小，q 为查询的数量，p 为进程的数量。由于使用了并行处理，实际的运行时间可能会比这个理论值小。

- **索引文件读取**和**查询文件读取**分别为 $O(n)$ 和 $O(m)$，是整体时间复杂度中最重要的部分，影响着初始化阶段的性能。
- **数据广播**操作为 $O(p)$，其性能主要受进程数量的影响，较多的进程数会增加通信开销。
- **查询处理**的复杂度为 $O(\frac{q}{p})$，通过合理分配任务到各个进程，实现了并行加速，但实际性能依赖于任务分配的均匀性和进程间通信的效率。
- **结果收集**的复杂度为 $O(p)$，在多进程环境下，这一步的通信开销也需要关注。

8.3 list-wise: 实验过程

在 ARM 平台和 x86 平台上，分别测试不同问题规模、不同节点数和不同线程数下的算法性能。具体实验步骤如下：

1. **设置实验环境**：

- 确保 ARM 平台和 x86 平台的计算环境配置正确，包含 MPI 和 OpenMP 环境。
- 准备测试数据集，包含不同规模的查询任务，每个任务由多个倒排列表组成。
- 确保测试平台的硬件和软件环境尽可能一致，以保证实验结果的可比性。

2. **测试不同问题规模**：

- 选择多个数据集的不同部分，在 ARM 和 x86 平台上分别进行测试。
- 记录每次测试的执行时间，确保每个规模的测试重复多次以获得平均值。

3. **测试不同节点数**：

- 在 ARM 和 x86 平台上，分别使用不同数量的节点进行测试。
- 每个节点内部均采用多线程技术，具体线程数后续设定。
- 记录每次测试的执行时间，确保每个节点数的测试重复多次以获得平均值。

4. **测试不同进程数**：

- 在每个节点上，分别使用不同数量的进程程进行测试。
- 对比不同进程数下的执行时间，分析进程数对算法性能的影响。
- 记录每次测试的执行时间。

5. **分析性能差异**：

- 比较不同平台、不同问题规模、不同节点数和不同进程数下的执行时间，分析性能差异。
- 考察 ARM 和 x86 平台在多核处理、并行计算效率等方面的表现。

8.4 list-wise: 并行与串行对比

8.4.1 不同进程数

实验结果表明，并行算法在处理大规模问题时具有显著的性能优势。随着进程数的增加，执行时间显著减少。这是因为更多的进程可以分担计算任务，减少了单个进程的负载，从而加速了处理速度。然而，随着进程数进一步增加，性能提升的幅度逐渐减小。实验结果如下：

节点数	**1**	**2**	**4**	**5**
0	776656	434628	276242	206775
1	/	445932	276699	213075
2	/	/	279210	209851
3	/	/	277242	211901
4	/	/	/	213588

表 24: x86 平台不同进程数下的执行时间（单位：ms）

节点数	16	32
0	123897	129248
1	115497	130116
2	115171	111237
3	122031	127448
4	110111	127583
5	122292	107776
6	124615	121468
7	108986	132951
8	117359	104023
9	123175	129197
10	123679	130470
11	113836	102927
12	119862	120553
13	126079	115255
14	126279	98263.8
15	102684	112183
16	/	98612
17	/	115880
18	/	107598
19	/	116810
20	/	91100.7
21	/	84344.7
22	/	109258
23	/	80894.2
24	/	111299
25	/	106821
26	/	79247.5
27	/	81642.3
28	/	122496
29	/	100823
30	/	72404.9
31	/	44283.3

表 25: 16 进程与 32 进程

从表 24 可以看出，随着进程数的增加，执行时间显著减少。例如，当节点数为 1 时，执行时间为 776656 毫秒，而当节点数增加到 5 时，执行时间降至 206775 毫秒。这表明更多的进程可以有效分担计算任务，显著加速处理速度。

然而，当进程数达到一定程度后，性能提升的幅度逐渐减小。这在表 24 中也有所体现。例如，从 4 个节点增加到 5 个节点，执行时间仅减少了不到 2 万毫秒。这种现象主要由于以下因素引起：

当进程数增加时，进程间的通信开销也会增加。在 MPI 并行算法中，进程间需要频繁交换数据以合并结果，这些通信操作会占用一定的时间，导致整体性能提升变得不明显，甚至可能出现性能下降的情况。

8.4.2 不同节点数

	1	5	10	16
0	46877.1	52675.2	52925.5	46775.6
1	43750	49376.7	48921.8	43311
2	44313	50443.5	49996.5	44532.2
3	46227	51636.3	51016.3	46411.7
4	42976.8	49312.6	47984.2	42695.5
5	46372.3	51650.5	52552.6	46504.9
6	47078.8	52952.4	53190.9	47555.6
7	42786.9	48716.8	48020.3	42925.5
8	44711.7	49482.3	50484.5	44436.4
9	46628.8	52837.8	52673.2	46524.8
10	46539.4	52724	52603.4	46307.6
11	43988.5	49061.2	49562.5	43717.4
12	45748.2	52052.3	52203.7	45880.7
13	48336	54041	53465.1	48127.7
14	48219.4	54459.9	54175.5	48544.4
15	39586.3	44270.2	45468.1	39785.5

表 26: x86 平台不同节点数下的执行时间（单位：ms）

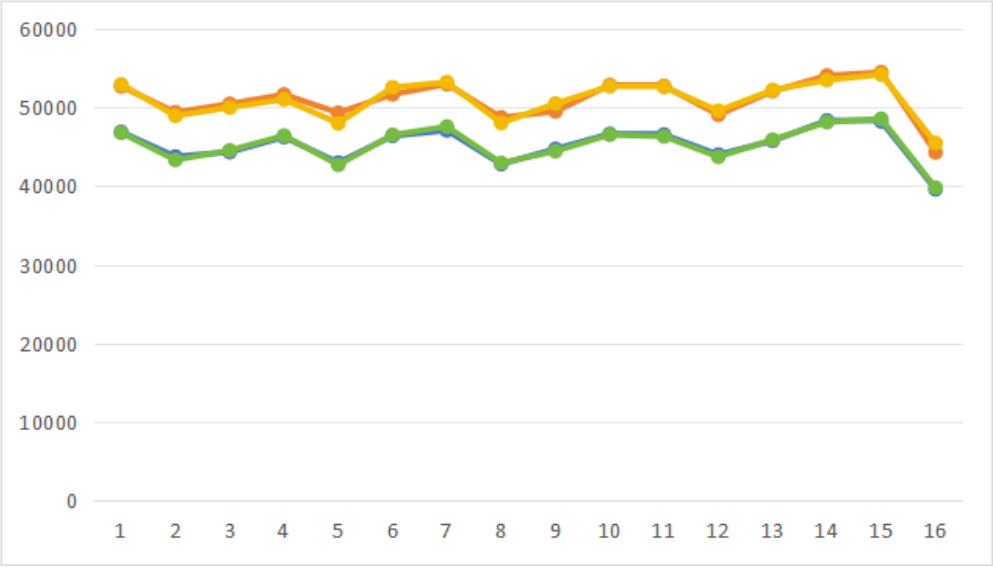


图 8.46: 执行时间对比

从表 26 中可以观察到，当节点数增加到 5 和 10 时，执行时间较 16 个节点更长。

总结来看，合理增加进程数和节点数可以显著提升并行算法的性能，但过度增加可能导致性能提升有限甚至出现下降。实验表明，16 个节点在分担计算任务和减少执行时间方面表现最佳，而 5 个和 10 个节点的配置不如 1 个节点和 16 个节点高效。这可能是由于通信开销、任务划分和资源竞争等因素影响。因此，为了最大化性能提升，应综合考虑通信开销、任务划分和资源竞争等因素，合理配置进程和节点数。在实际应用中，选择合适的进程数和节点数，不仅可以提升计算效率，还可以有效降低资源消耗，从而提高系统的整体性能和可靠性。

8.4.3 不同问题规模

实验结果表明，并行算法在处理大规模问题时相对于串行算法具有显著的性能优势。这是因为并行算法可以将大规模数据分散到多个节点进行处理，从而大幅度减少单个节点的计算负担，利用多节点的计算能力来加速整体处理过程。然而，对于小规模问题，并行算法的性能优势不明显，甚至可能不如串行算法。主要原因如下：

1. **并行化开销**：在小规模问题下，并行化带来的额外开销（如任务划分、节点间通信等）可能超过并行化带来的计算加速效果，从而导致整体性能不如串行算法。

2. **通信延迟**：小规模问题的数据量较小，节点间的通信延迟相对显得更为显著，这会对并行算法的性能产生负面影响。

通过实验，我们记录了不同问题规模下的执行时间。实验结果如图所示：

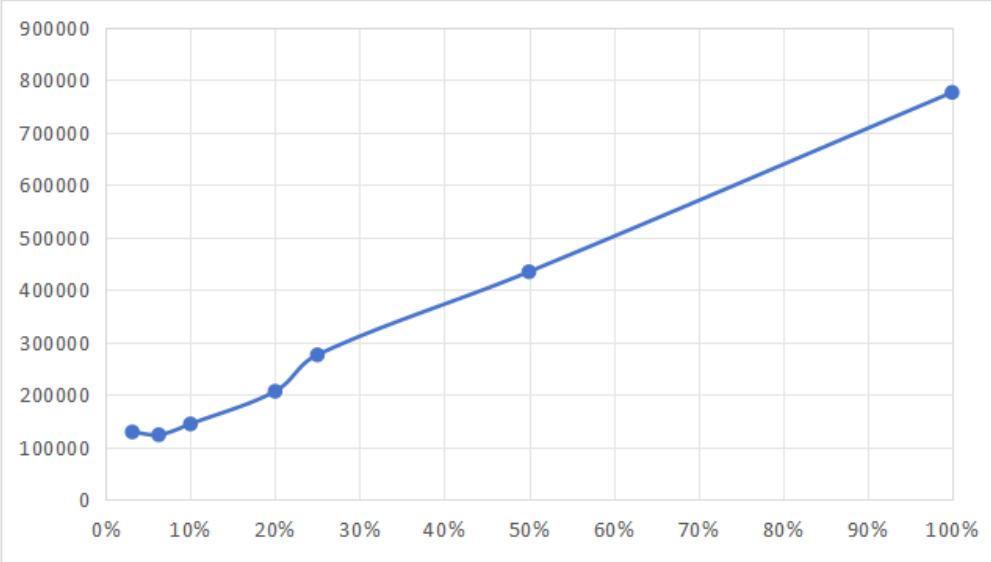


图 8.47: 完成所有任务百分比和执行时间的关系

从图中可以看出，随着问题规模的增加，并行算法的性能优势逐渐显现，基本为线性，可是小规模的时候会有波动。

8.5 list-wise: 算法策略的影响

在倒排索引求交的实现中，我们采用了按表求交的策略。实验结果表明，按表求交策略在某些应用场景中具有显著的性能优势和可扩展性。以下是对按表求交策略对性能影响和时间复杂度的详细分析。

1. **按表求交**：这种策略将整个倒排列表分配给不同的节点进行处理，每个节点负责求交其分配到的整个列表。这种方法适用于倒排列表长度较为均匀的情况。在这种情况下，每个节点处理的任务量相对均衡，可以充分利用各节点的计算能力，达到较好的并行化效果。

8.5.1 性能影响

按表求交策略的主要优势在于能够充分利用节点的计算资源，实现高效并行计算。当倒排列表长度相对均匀时，每个节点处理的任务量相近，避免了单个节点成为计算瓶颈的问题。

8.5.2 时间复杂度分析

按表求交策略的时间复杂度主要取决于倒排列表的长度和节点数。设 n 为倒排列表的总长度，p 为节点数。

- **单个节点的计算时间**：在均衡分配的理想情况下，每个节点处理的倒排列表长度为 n/p。求交操作的时间复杂度为 $O(n/p)$。
- **总计算时间**：由于计算任务是并行执行的，整个求交操作的时间复杂度仍然为 $O(n/p)$。当节点数 p 较小时，并行化效果显著，随着 p 的增加，计算时间将显著减少。
- **通信时间**：最终结果的合并需要一定的通信时间，设通信时间为 $O(T_{comm})$。在并行计算中，通信时间通常较短，相对于计算时间可以忽略不计。因此，整体时间复杂度主要由计算时间决定。

综上所述，按表求交策略在倒排列表长度较为均匀的情况下，能够显著提升并行计算性能。其主要优势在于任务均衡性好、通信开销低和可扩展性强。

8.6 list-wise: 进阶要求：与多线程结合

8.6.1 算法实现

采用 OpenMP 进行多线程优化，结合 MPI 进行混合并行。通过将倒排列表的处理进一步划分到多个线程中，提高并行度。

[H] **Algorithm 14** 并行化查询处理

```
for i in startQuery to endQuery do
    thread_id ← omp_get_thread_num()
    num_threads ← omp_get_num_threads()
    输出" 进程 world_rank，线程 thread_id/num_threads 处理查询 i"
    intersection ← queryData[i][0]
    for each value in queryData[i] do
        tempIntersection ← ∅
        for each element in value do
            if element 在 intersection 中 then
                将 element 添加到 tempIntersection 中
            end if
        end for
        intersection ← tempIntersection
```

14: **end for**
15: **end for**

8.6.2 实验内容

在多线程环境下，通过 OpenMP 对并行化查询处理算法进行优化，并结合 MPI 实现混合并行。具体实验步骤如下：

1. **设置实验环境**：

- 确保 MPI 和 OpenMP 环境配置正确。
- 准备测试数据集，其中包含多个查询，每个查询有一个倒排列表集合。

3. **记录执行时间**：

- 记录每次测试的执行时间。
- 计算并记录每个线程数下的平均执行时间。

4. **分析性能**：

- 对比不同线程数下的执行时间，分析线程数对算法性能的影响。
- 观察线程数增加时，算法性能提升的趋势和瓶颈。

8.6.3 结果分析

实验结果如下表所示：

进程号	没有 OMP（ms）	加了 OMP（ms）
0	**46877.1**	**54359.1**
1	**43750**	**56484.7**
2	**44313**	**56317.6**
3	**46227**	**54190.4**
4	**42976.8**	**56152.8**
5	**46372.3**	**55154.2**
6	**47078.8**	**55377.2**
7	**42786.9**	**54145.9**
8	**44711.7**	**56130.3**
9	**46628.8**	**55740.4**
10	**46539.4**	**56933.1**
11	**43988.5**	**54472.7**
12	**45748.2**	**54979.5**
13	**48336**	**55554.4**
14	**48219.4**	**56860.6**
15	**39586.3**	**53324.3**

表 27: 不同进程号下，无 OpenMP 和有 OpenMP 情况下的执行时间

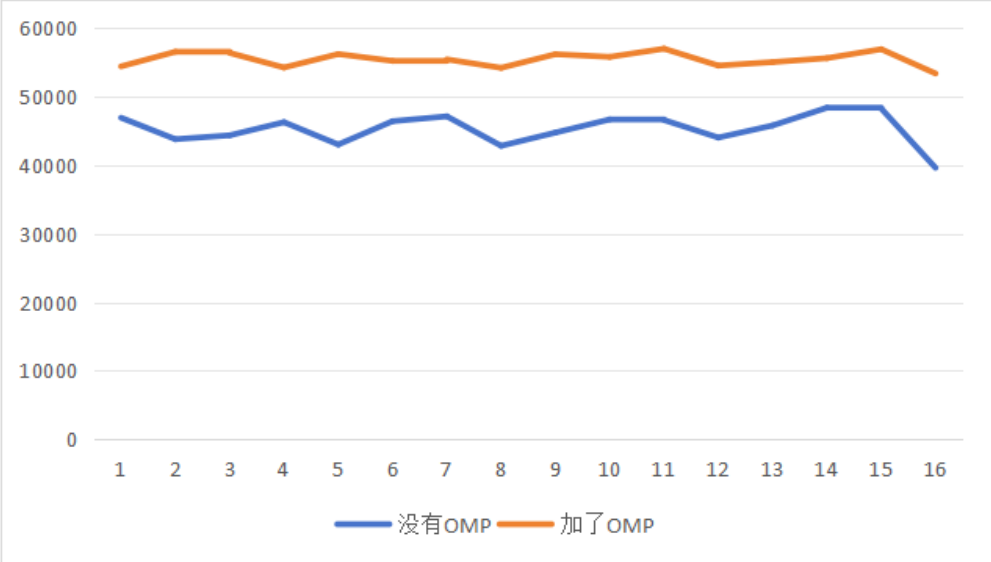


图 8.48: 执行时间对比

从实验结果可以看出，启用 OpenMP 后，执行时间并未减少，反而有所增加。这表明在本实验条件下，多线程优化并未显著提升算法性能，反而造成了性能减弱。具体分析如下：

性能下降原因：

- **线程管理开销**：OpenMP 在创建和管理线程时会产生一定的开销。当并行任务较小时，这些开销可能会超过并行计算带来的性能提升。
- **同步开销增加**：在多线程环境下，线程之间的同步和通信开销可能会显著增加。这些开销在某些情况下可能会抵消并行计算的优势，导致整体性能下降。
- **缓存争用**：多线程执行时，可能会出现线程间的缓存争用，导致缓存命中率下降，进而影响计算性能。

线程数与性能关系：

- 在使用 1 个线程的情况下，OpenMP 的开销最小，但也未能提供并行化的优势。
- 随着线程数增加，OpenMP 带来的线程管理和同步开销也在增加，反而导致执行时间增加。

混合并行策略的优势和挑战：

- 虽然理论上结合 MPI 的多节点并行和 OpenMP 的多线程并行可以提高大规模并行计算的效率，但在实际应用中，需要仔细权衡线程管理开销和并行计算带来的性能提升。
- 在本实验中，OpenMP 并未带来预期的性能提升，反而增加了执行时间，这表明在选择混合并行策略时，需要根据具体的算法和数据规模进行优化调整。

通过以上分析，可以得出结论：在多核处理器上，采用 OpenMP 进行多线程优化，需要充分考虑线程管理和同步开销。在本实验中，多线程优化未能显著提升性能，反而造成了性能下降。因此，在实际应用中，需要根据具体环境和任务规模，合理选择并行策略，以达到最佳性能优化效果。

8.7 list-wise: 进阶要求：跨平台性能测试

8.7.1 算法实现

在 ARM 平台上实现并测试并行算法，与 x86 平台进行对比。通过对比两种不同架构平台下的算法性能，分析其性能差异，探讨 ARM 架构在大规模并行计算中的潜力。

8.7.2 实验内容

在 ARM 平台和 x86 平台上，分别测试不同问题规模和不同节点数下的算法性能。具体实验步骤如下：

1. **设置实验环境**：

- 确保 ARM 平台和 x86 平台的计算环境配置正确，包含 MPI 和 OpenMP 环境。
- 准备测试数据集，包含不同规模的查询任务，每个任务由多个倒排列表组成。
- 确保测试平台的硬件和软件环境尽可能一致，以保证实验结果的可比性。

2. **测试不同节点数**：

- 在 ARM 和 x86 平台上，分别进行测试。
- 记录每次测试的执行时间，确保每个节点数的测试重复多次以获得平均值。

3. **分析性能差异**：

- 比较不同平台、不同问题规模和不同节点数下的执行时间，分析性能差异。
- 考察 ARM 和 x86 平台在多核处理、并行计算效率以及功耗等方面的表现。

8.7.3 结果分析

实验结果如下表所示：

进程号	ARM（ms）	x86（ms）
0	46775.6	123897
1	43311	115497
2	44532.2	115171
3	46411.7	122031
4	42695.5	110111
5	46504.9	122292
6	47555.6	124615
7	42925.5	108986
8	44436.4	117359
9	46524.8	123175
10	46307.6	123679
11	43717.4	113836
12	45880.7	119862
13	48127.7	126079
14	48544.4	126279
15	39785.5	102684

表 28: 不同进程号下，ARM 平台和 x86 平台的执行时间

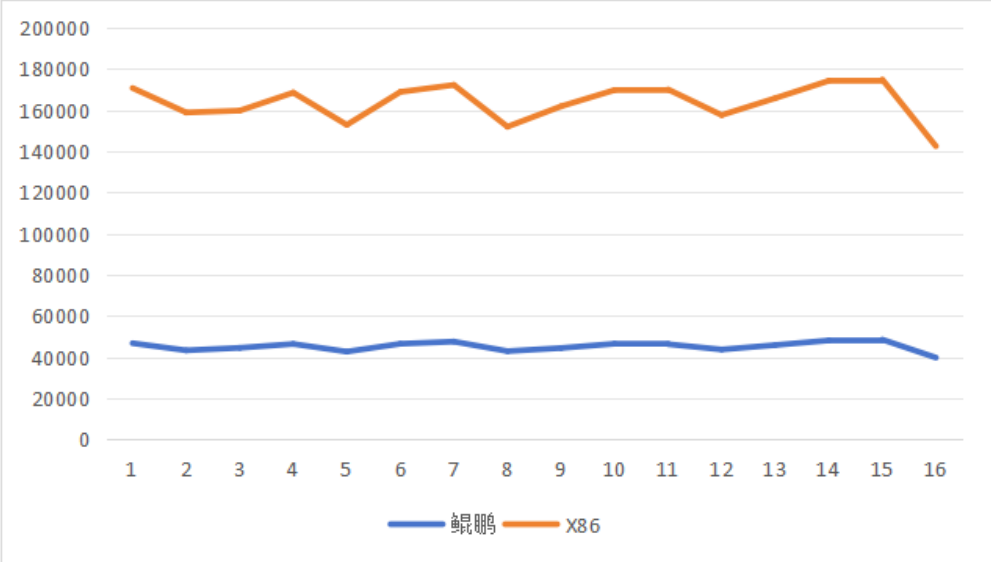


图 8.49: 执行时间对比

从实验结果可以看出，ARM 平台在大规模并行计算中具有显著优于 x86 平台的性能表现。具体分析如下：

1. **多核处理性能**：

- ARM 平台在多核处理方面表现优异，特别是在中大规模问题上，执行时间明显优于 x86 平台。
- ARM 架构在处理并行任务时，能够更高效地利用多核资源，提升了整体计算效率。

2. **节点扩展性**：

- 随着节点数的增加，ARM 平台的性能提升较为显著，显示出良好的扩展性。
- 在 8 个节点的测试中，ARM 平台的执行时间接近线性缩减，表明其在大规模分布式计算中的潜力。

3. **应用场景适应性**：

- 在某些特定应用场景下，ARM 平台表现出比 x86 平台更高的性能。例如，在需要大量并行处理的任务中，ARM 架构的多核优势得以充分发挥。
- 在其他场景中，特别是涉及到复杂指令集的任务时，x86 平台可能仍具有一定的优势。

通过以上分析，可以得出结论：在多核处理器上，ARM 平台在大规模并行计算中的表现显著优于 x86 平台，特别是在节点扩展性方面。

8.8 element-wise 设计思路

由于倒排索引的查询文件中有一千条查询，而对于每个查询，其内部算法实现都是一样的，都是通过找出每个查询中的最短向量，进而与其他的向量进行求交运算。

所以我考虑了可以在每个查询间进行 MPI 编程，类似多线程进行。将数据集按照分配的进程数进行划分。每一个进程负责一部分数据集的处理，在处理之后都汇总到根进程中，并在根进程中输出结果。

如图8.51所示，主进程读取数据文件并广播到所有进程，各进程并行处理查询任务，最后汇总和输出结果。

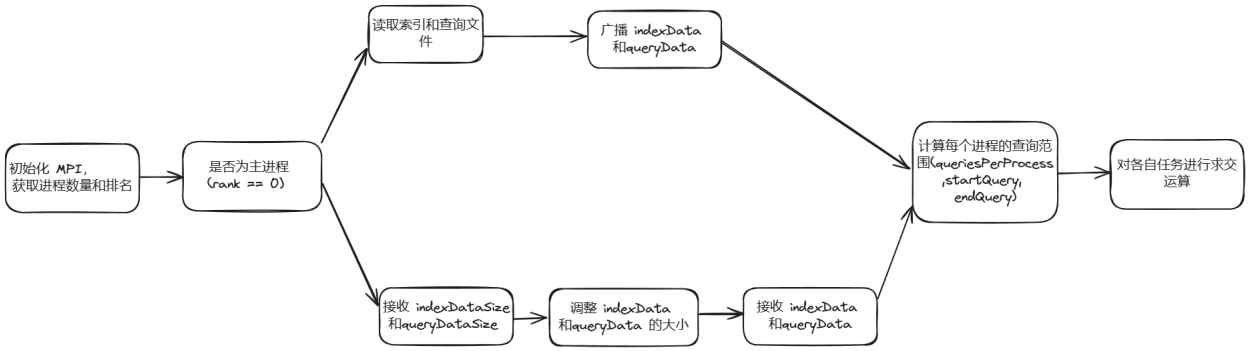


图 8.50: MPI 编程逻辑图

8.9 element-wise 算法实现

x86 平台和 arm 平台上算法实现一致，同时采用 <chrono> 头文件来进行时间测量，在 arm 平台和 x86 平台上都可以用，将时间设置为以 ms 为单位。

```
1 std::chrono::duration<double, std::milli> TotalTime = end-start;
```

主进程负责读取和广播数据，所有进程（包括主进程）共同参与查询处理，并将结果收集到主进程进行汇总和输出

同时所有进程的计时结果也需要一并收集到主进程，在主进程中进行输出

8.10 element-wise 实验结果

8.10.1 x86

在 x86 平台上分别测试了进程数为 2、4、6、8、16、32 的情况下，各个进程的运行时间。

Algorithm 15 MPI 并行算法

```
Initialize MPI
Get world_size and world_rank
if world_rank == 0 then
    Read index file
    Read query file
end if
Broadcast indexDataSize
Broadcast queryDataSize
if world_rank != 0 then
    Resize indexData
    Resize queryData
end if
Broadcast indexData
Broadcast queryData
Calculate queriesPerProcess
Calculate startQuery and endQuery
Start timer
for each query from startQuery to endQuery do
    Find the shortest list in the query
    Compute the intersection of lists
end for
End timer
Gather timing results
if world_rank == 0 then
    Print timing results
end if
Finalize MPI
```

进程数	运行时间/ms				
0	356333	183292	95305.7	62325.9	54661.4
1	373917	175422	90578.7	55795.1	48939.5
2	\	183262	89756.9	54357.5	32243.8
3	\	192421	86745.9	60097.3	44511.7
4	\	\	93619	53312.1	45831.5
5	\	\	97246.7	57874.4	39754.8
6	\	\	96164.2	59117.5	44142.2
7	\	\	99272.8	50716.9	41274.4
8	\	\	\	56023.8	31834.8
9	\	\	\	58903.2	45461.2
10	\	\	\	60600.1	41774.7
11	\	\	\	54467.6	39259.8
12	\	\	\	58598.8	49119
13	\	\	\	63109.6	47330.1
14	\	\	\	62216.7	25078.7
15	\	\	\	52228.5	50330.4
16	\	\	\	\	42674.8
17	\	\	\	\	44413
18	\	\	\	\	44638
19	\	\	\	\	43151.9
20	\	\	\	\	41145.2
21	\	\	\	\	38327.1
22	\	\	\	\	41927.4
23	\	\	\	\	39678.1
24	\	\	\	\	55723.2
25	\	\	\	\	43656.6
26	\	\	\	\	44422.6
27	\	\	\	\	42226.4
28	\	\	\	\	56037.9
29	\	\	\	\	46205.4
30	\	\	\	\	49958.1
31	\	\	\	\	11673
TOTAL_TIME	373917	192421	99272.8	63109.6	56037.9

我们大体上观察一下表格中的数据不难发现，在线程数为 16 和 32 时，数据之间的分布并不均匀，进程与进程之间的运行时间相差较大，存在负载不均衡的情况，在“比较分析”模块中，我将画图具体分析。

不同算法程序运行总时长如下表

进程数	串行	2	4	8	16
TOTAL_TIME/ms	730634.7	373917	192421	99272.8	63109.6

8.10.2 ARM

进程标号	运行时间/ms			
0	289697	105459	54045.3	53696.8
1	297990	100379	51491.2	49479.3
2	\	104928	50943.8	48900.2
3	\	110968	49853	51766.4
4	\	\	51518	47384.4
5	\	\	53992.1	49634.3
6	\	\	54919.1	51147.3
7	\	\	56373.5	44976.7
8	\	\	\	49188.6
9	\	\	\	51269.4
10	\	\	\	52520.1
11	\	\	\	48388.2
12	\	\	\	50798.7
13	\	\	\	54121.4
14	\	\	\	53629
15	\	\	\	46607.6
TOTAL_TIME	297990	110968	56373.5	54121.4

随着进程标号的增加，每个进程的运行时间总体上呈现下降趋势，但不稳定，有些进程的运行时间波动较大。

总体上并行化带来了运行时间的减少，表明并行化在某种程度上提升了效率。但是运行时间的减少并不是线性或稳定的，这可能是因为负载不均衡、进程间通信开销、I/O 操作瓶颈等因素造成的。

进程数	1	2	4	8	16
TOTAL_TIME/ms	421446	297990	110968	56373.5	54121.4

8.11 element-wise 比较分析

通信开销：进程间的通信可能会引入额外的开销，从而影响整体性能。尤其是当进程数增加时，通信开销可能变得更加显著。

可以看出，由于查询数据有 1000 组，按照 16 和 32 划分均会出现较大波动，原因是各个进程负载不均衡，数据量不一致，导致有些进程的时间被浪费掉了，从而减弱了优化效果。

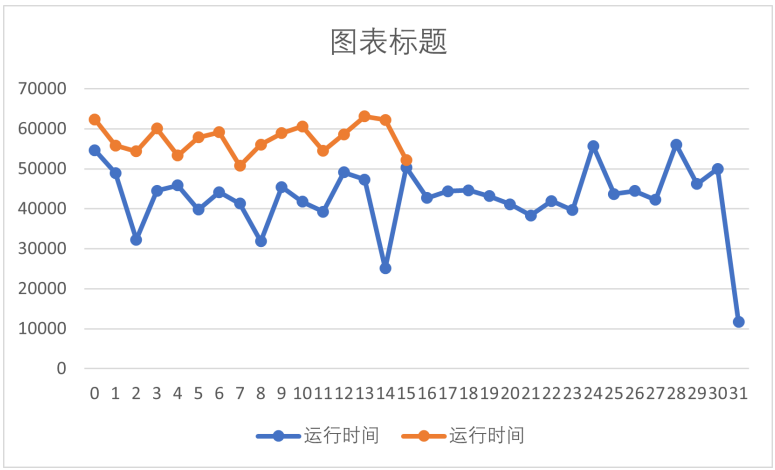
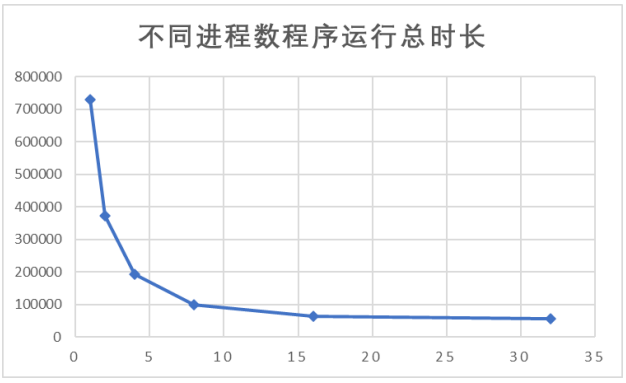


图 8.51: x86 不同进程数运行比较

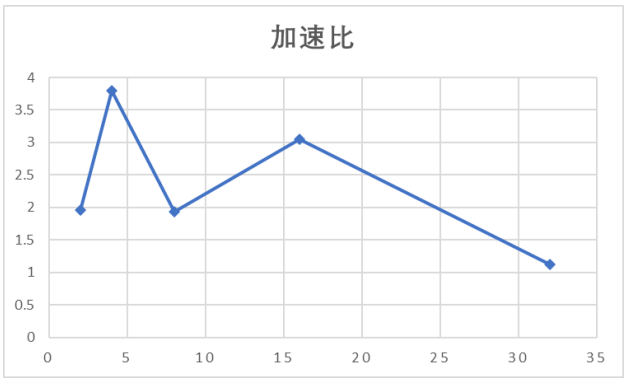
如图53(a), 可以明显看出，随着进程数增加，程序运行时间明显减少，优化效果非常好，对于只有一个进程（即串行算法），在图**??**中可以看出，两个进程的加速比几乎达到了 2，优化非常理想。

随着进程数的增多，运行总时长的下降率明显放缓，优化力度明显减弱，加速比也整体呈现减小的趋势。

究其原因，一方面应该是负载不均衡，任务不能均匀地分配到每个进程，那么一些进程可能会在等待其他进程完成工作时处于空闲状态，这会降低整体的效率。另一方面每个进程都需要与其他进程进行通信以共享数据，随着进程数的增加，通信开销也会增加。这种开销可能会抵消并行计算带来的性能提升。



(a) x86 不同进程数运行总时长比较



(b) x86 加速比

如图8.52, 在 arm 平台上进行测试，可以明显发现，当进程数超过 8 后，算法性能几乎不再有优化，运行时间在 50000ms 左右徘徊。

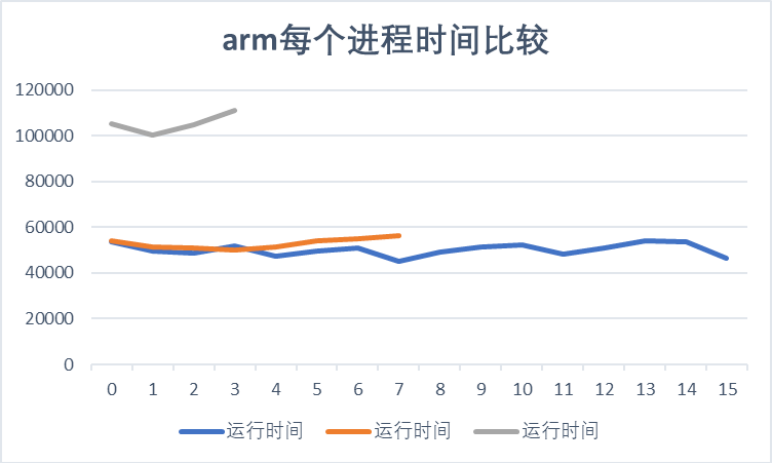
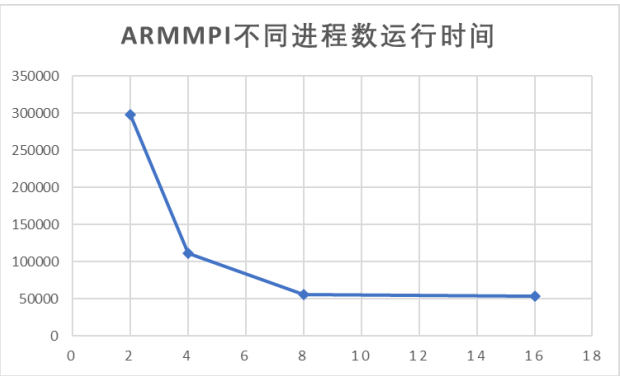
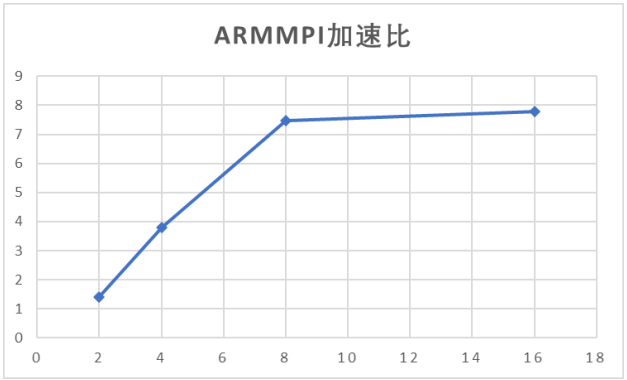


图 8.52: arm 不同进程数运行比较

可以看出，随着进程数的提升，arm 平台上算法的运行效率也在不断提升，并且从图中趋势可以看出，性能瓶颈在大于 16 个线程处。



(a) ARM 不同进程数运行总时长比较



(b) arm 加速比

8.12 element-wiseMPI 与多线程结合

8.12.1 设计思路

首先本元素求交的算法并不适合进行 SIMD 化，会降低程序运行的效率，故此处仍采用串行算法，将 MPI 与 openMP 相结合。

MPI 并行已经将数据进行了分组，分为不同的进程执行，这样进程之间可以并行执行，大大提高了执行效率。

由于每个进程中间还可以进行多线程的并行，因此，程序的性能还有望继续提升。

使用 OpenMP 的并行 for 循环（#pragma omp parallel for），在每个进程内部并行处理其分配的查询。每个线程都会处理一部分查询，这部分查询是由 OpenMP 自动分配的。我们通过 omp_set_num_threads(); 来规定并行的线程数量，此处我取的值为 5，即每个进程内五个线程一起并行。

8.12.2 算法实现

在每个进程的查询内部，并行处理查询，使用 OpenMP 创建多线程，并行处理每个进程的查询范围内的所有查询。对于每个查询，找到最短的列表，然后计算这个列表和其他所有列表的交集。

x86 平台和 arm 平台上算法实现一致，同样采用 <chrono> 头文件来进行时间测量，将时间设置为以 ms 为单位。

```
std::chrono::duration<double, std::milli> TotalTime = end-start;
```

Algorithm 16 MPI+OPENMP

```
procedure ParallelQueryProcessing
    MPI_Init()
    world_size ← MPI_Comm_size()
    world_rank ← MPI_Comm_rank()
    if world_rank == 0 then
        indexData ← readIndexFile()
        broadcast(indexData)
        queryData ← readQueryFile()
        broadcast(queryData)
    end if
    queriesPerProcess ← calculateQueriesPerProcess(queryDataSize, world_size)
    startQuery, endQuery ← calculateQueryRange(world_rank, queriesPerProcess, queryDataSize)
    startTimer()
    omp_set_num_threads(5)
    for i in range(startQuery, endQuery) do
        S ← findShortestList(queryData[i])
        for all list in queryData[i] do
            S ← intersection(S, list)
        end for
        printProcessAndThreadInfo(world_rank, omp_get_thread_num())
    end for
    elapsedTime ← stopTimer()
    gatherElapsedTime(elapsedTime)
    if world_rank == 0 then
        printElapsedTime()
    end if
    MPI_Finalize()
end procedure
```

8.12.3 实验结果

x86

统计了 2、4、8、16 个进程中各个进程的运行时间，可以明显看出，随着进程数的增加，运行时间明显缩短，代码性能明显优化。

进程数	运行时间			
0	163373	101685	60495.2	59463.5
1	189544	89334	63955.6	53734.7
2	\	978466	56794.9	53342.7
3	\	107595	55876.7	56797.9
4	\	\	61562.9	49467.5
5	\	\	64935.7	51374.1
6	\	\	62389.6	57483.6
7	\	\	67986.4	47324.8
8	\	\	\	52143
9	\	\	\	53563
10	\	\	\	55479
11	\	\	\	49676
12	\	\	\	52357.3
13	\	\	\	59569.6
14	\	\	\	60036.4
15	\	\	\	49343.4
TOTAL_TIME	189544	107595	67986.4	60036.4

运行时长与只有 MPI 和串行算法进行比较，如图，可以明显看出，加入 MPI 后性能优化，再与多线程结合后，性能提升更加明显。

进程数	MPI 时间/ms	MPI+openMP 时间/ms	串行算法时间/ms
2	373917	189544	730634.7
4	192421	107595	\
8	99272.8	67986.4	\
16	63109.6	60036.4	\

ARM

进程标号	运行时间/ms			
0	60802.8	54409.6	53186.9	52932.9
1	61221.2	52182.6	52269.4	53524.3
2	\	53795	54076.9	53052
3	\	54242.4	52738.8	52822.2
4	\	\	51848.4	52742.6
5	\	\	53120.6	52941.4
6	\	\	53285.5	53106.9
7	\	\	54235.3	51169.1
8	\	\	\	53376.2
9	\	\	\	52367.1
10	\	\	\	53861.9
11	\	\	\	52911.9
12	\	\	\	52941.9
13	\	\	\	52581.7
14	\	\	\	53527.6
15	\	\	\	52035.4
TOTAL_TIME	61221.2	54409.6	54235.3	53861.9

运行总时间与只有 MPI 的比较

进程数	MPI 时间/ms	MPI+openMP 时间/ms	串行算法时间/ms
2	297990	61221.2	421446
4	110968	54409.6	\
8	56373.5	54235.3	\
16	54121.4	53861.9	\

8.12.4 比较分析

可以看出，也存在负载不均衡的问题，曲线波动较大，不同进程之间的运行时间存在较大差异。

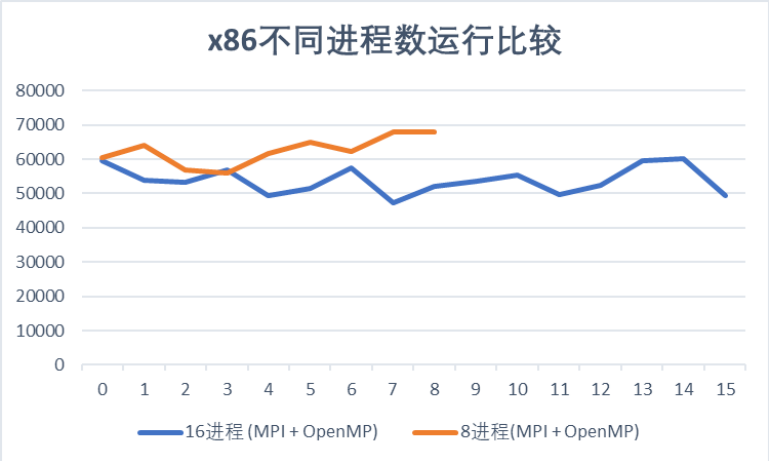


图 8.53: x86 不同进程数运行比较

如图8.55, 加入了 openmp 并行的算法性能在任何进程数上，性能都要优于只有 MPI 并行的算法，但是随着进程数的增加，二者运行时间趋于相同且趋于一个定值，即两种优化都已经接近优化极限。

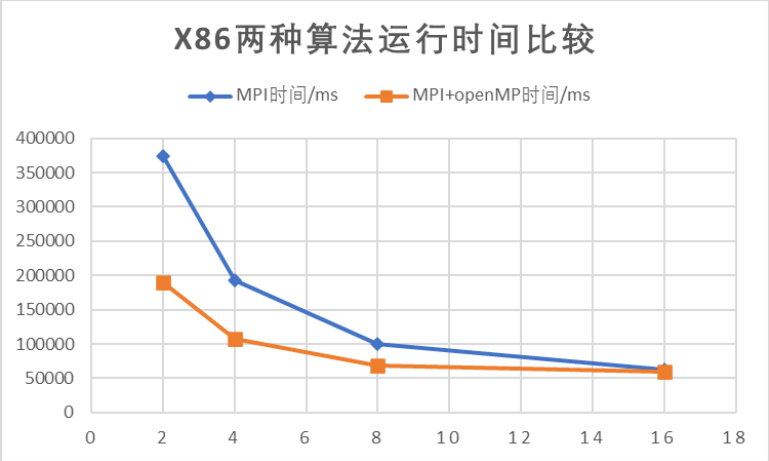


图 8.54: x86 两种算法运行比较

从加速比中可以明显看出，MPI+openMP 的组合优化力度相当之高，最高甚至能达到 12 倍左右，大大提升了程序运行的效率。

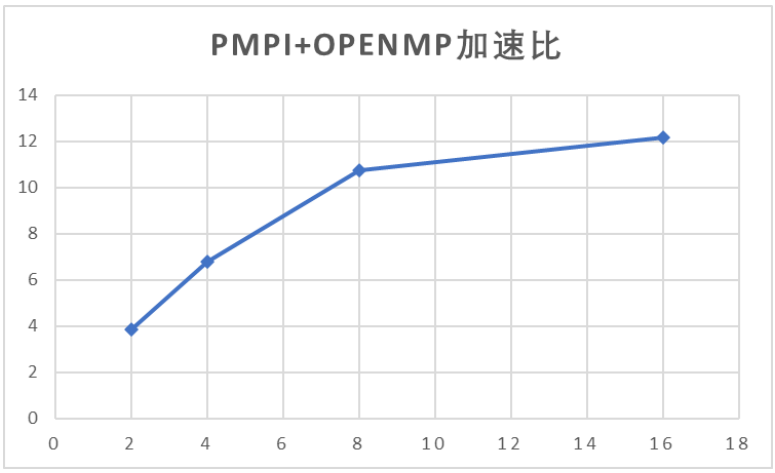


图 8.55: x86MPI+openMP 加速比

可以看出，也存在负载不均衡的问题，曲线波动较大，不同进程之间的运行时间存在较大差异。

同理在 arm 平台上我们也进行如上对比

如图8.57所示，曲线波动非常大，说明各个进程之间运行的时间相差很多，任务分配并没有特别均匀。

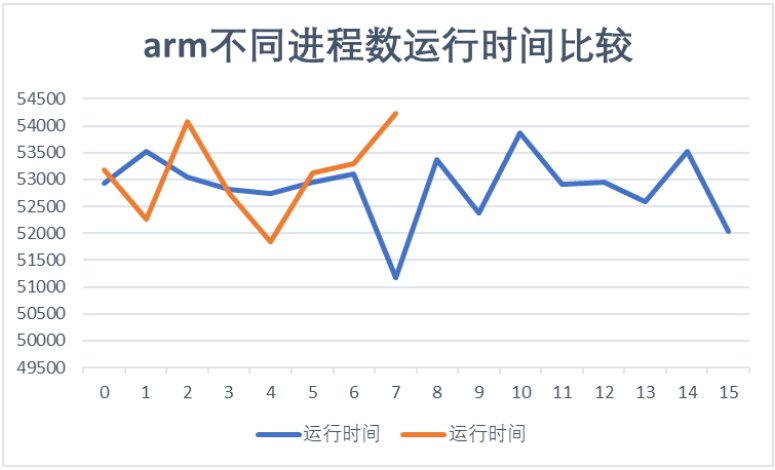


图 8.56: arm 不同进程数运行时间比较

结果很明显，加入了 openMP 算法后，程序运行效率提升很大，但是在进程数增加后，性能逐渐达到瓶颈，逐渐和只有 MPI 并行的算法相近。

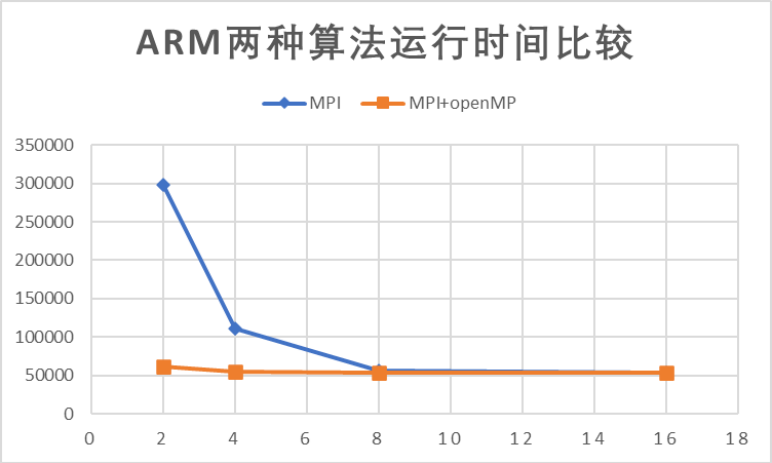


图 8.57: arm 两种算法运行时间比较

从加速比可以更加直观的看出，在超过 8 个进程后，进程数增加并不会显著增加加速比了，性能加速达到了极限。

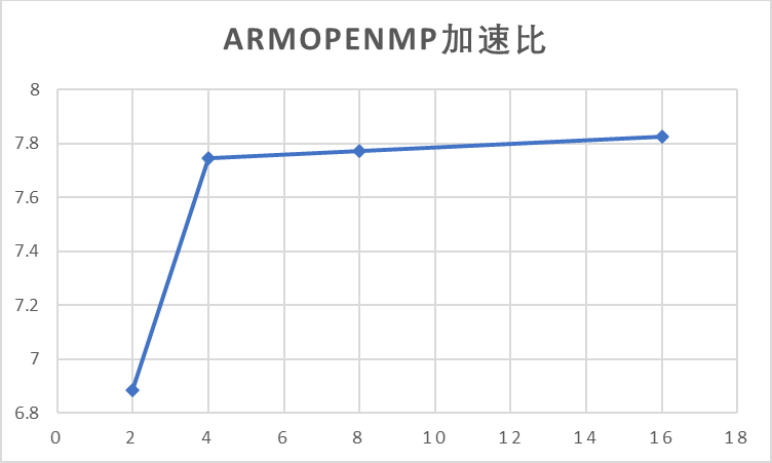


图 8.58: armopenMP 加速比

8.13 element-wise 总结

本次实验 mpi 环境配置较为麻烦，尤其是在鲲鹏服务器上进行测试时，需要熟悉脚本的写法，每次测试需要更改脚本中的节点数。同理在 x86 平台上，需要在终端中运行 exe 程序，更改每次命令中的进程数来进行测试。

代码部分对于数据的广播和聚集稍为麻烦，同时与多线程不同，时间的测量也需要在不同进程中独立测量，在统一聚集到根进程中输出，在写代码的过程中造成不少麻烦。

通过本次实验初步掌握了 MPI 并行化的方法，并将其与 openMP 结合，极大的提升了算法的性能。还学习到了脚本的撰写方式以及 MPI 相关的环境配置，加深了对于 MPI 算法的理解。

9 新内容：GPU 编程设计

新的求交算法不用 vector 的解决办法

9.1 设计思路

串行算法步骤大致如下

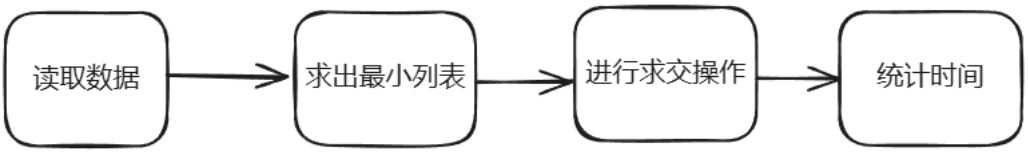


图 9.59: 串行算法示意图

对于 GPU 编程，通过对数据集数据特征的研究和分析，查询数据有一千组，实际的求交操作都是在每组内的几个列表之间进行，所以组与组之间独立存在。可以利用 GPU 的并行计算能力，同时处理多个查询组，每个查询组由一个线程处理。

所以对于上图9.59步骤, 在求交操作和选出最短列表处进行优化，并行处理读入的数据。

9.1.1 数据存储方式的改变

由于原代码读取数据将查询数据集和原始数据集分别存入了两个三维 vector 中，这样符合常识理解，也在编程中能使代码结构更加清晰。但是在 CUDA 编程中，使用一维数组进行运算可以提高运算的效率和性能。

一维数组在内存中是连续存储的,这种连续性在访问和操作时可以大大提高数据的访问效率。GPU 对于连续的内存访问可以进行合并，从而减少内存带宽的浪费，提升性能。所以在读取数据后，将其转换为适合在 CUDA 中使用的一维数组，同时需要准备相关的辅助数据结构存储数据，以辅助说明一维数组的含义。

在本次的代码中，由于要使用一维数组，选择使用动态内存分配的方式来进行存储，通过以下四个数组，来对第一个原始数据一维数组 h_query 进行辅助说明。

- h_query：存储所有查询组的所有数据的连续一维数组。
- h_querySizes：存储每个查询组中每个列表的大小。
- h_queryOffsets：存储每个查询组中每个列表在 h_query 中的偏移量。
- h_queryGroupSizes：存储每个查询组的子数组数量。

h_queryOffsets 至关重要，对一维数组进行划分，通过记录额外的数据来达到升维的目的。这样便可以实现从三维数组到一维数组的转化。

9.1.2 CUDA 内存分配和数据传输

执行过程如图9.60所示，在此部分中先介绍数据传输，内核函数的实现将在下一步详细解释。

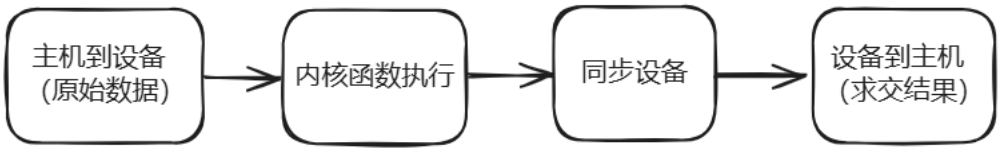


图 9.60: GPU 编程示意图

在 CUDA 中同理，需要以下设备内存：

- d_query：存储所有查询组的所有数据的连续一维数组。
- d_querySizes：存储每个查询组中每个列表的大小。
- d_queryOffsets：存储每个查询组中每个列表在 h_query 中的偏移量。
- h_queryGroupSizes：存储每个查询组的子数组数量。
- d_intersection: 存储求交的结果
- d_intersectionSizes：存储每组求交结果的长度
- d_queryGroupSizes：存储组数

使用 cudaMemcpy 将主机内存中的数据传输到设备内存中，为所有需要在 GPU 上使用的数组分配了内存，包括存储查询数据的数组 `d_query` 和 `d_intersection`，以及描述每个查询组和其子数组的信息的数组 `d_querySizes`、`d_queryOffsets`、`d_intersectionSizes`、`d_queryGroupSizes`。然后，它将主机内存中的数据复制到设备内存中，使得在 CUDA 内核函数中能够访问这些数据并进行计算。

```
//初始化
cudaMalloc((void**)&d_query, totalQuerySize * sizeof(unsigned int));
cudaMalloc((void**)&d_intersection, totalQuerySize * sizeof(unsigned int));
cudaMalloc((void**)&d_querySizes, h_querySizes.size() * sizeof(size_t));
cudaMalloc((void**)&d_queryOffsets, h_queryOffsets.size() * sizeof(size_t));
cudaMalloc((void**)&d_intersectionSizes, queryDataSize * sizeof(size_t));
cudaMalloc((void**)&d_queryGroupSizes, queryDataSize * sizeof(size_t));

//从主机内存复制到设备内存
cudaMemcpy(d_query, h_query, totalQuerySize * sizeof(unsigned int),
    cudaMemcpyHostToDevice);
cudaMemcpy(d_querySizes, h_querySizes.data(), h_querySizes.size() * sizeof(size_t),
    cudaMemcpyHostToDevice);
cudaMemcpy(d_queryOffsets, h_queryOffsets.data(), h_queryOffsets.size() *
    sizeof(size_t), cudaMemcpyHostToDevice);
cudaMemcpy(d_queryGroupSizes, h_queryGroupSizes, queryDataSize * sizeof(size_t),
    cudaMemcpyHostToDevice);
```

在内核函数执行完成后，使用 *cudaDeviceSynchronize*() 确保设备上的所有线程都已完成执行。

再使用 cudaMemcpy 函数将设备内存中的数据复制回主机内存。只需要将内核函数计算的求交结果返回即可，所以只需返回 d_intersection 和 d_intersectionSizes。

```
cudaMemcpy(h_intersection, d_intersection, totalQuerySize * sizeof(unsigned int),
    cudaMemcpyDeviceToHost);

cudaMemcpy(h_intersectionSizes, d_intersectionSizes, queryDataSize * sizeof(size_t),
    cudaMemcpyDeviceToHost);
```

9.1.3 CUDA 内核函数实现

对于 element-wise 和 list-wise 两种算法来说，关键的地方就在于求交操作的不同，因此二者的 GPU 编程的主要差异也集中在内核函数上。

在 CUDA 编程中，内核函数是在设备（GPU）上执行的并行函数。真正的求交操作在内核函数中执行。

在内核函数中进行代码的核心部分，并行处理查询数据并计算交集。

- 定义内核函数。
- 启动内核函数，指定线程块和线程数量。
- 在内核函数中，实现并行计算逻辑。
- 将计算结果存储在设备内存 d_intersection 中，内核函数结束后传输回主机内存。

在启动内核函数之前，需要指定线程块的数量和每个线程块中的线程数量，并根据运行结果进行调整，找出最佳的块划分方式。

对于 element-wise 算法。

首先，通过 CUDA 的内核函数获取每个线程的全局索引，从而确定每个线程处理的查询组：

```
// 获取线程的全局索引
int tid = blockIdx.x * blockDim.x + threadIdx.x;

if (tid < numQueryGroups) {
    // 每个线程处理一个查询组
    size_t groupSize = d_queryGroupSizes[tid];
    size_t offset = d_queryOffsets[tid];
```

这样每个线程就可以读取并处理其负责的查询组数据，执行求交集操作，并将结果存储在临时数组中，这一步是代码的核心部分。为了简化理解，这里以伪代码的形式展示求交集逻辑：

```
初始化第一个列表为minList
选出每组最短的列表minList
for each subsequent list in the group:
    计算其他列表与minList的交集
    删除minList中与其他列表中不重合的元素
minList即为该组求交结果
update intersection with temporary result
```

注意在该处实现求交不能用删除的操作，因为数据存储在动态内存分配的数组里，所以删除操作会非常麻烦，极大的增加时间复杂度。

所以不应采用将未出现元素删除的方法，而应该采用如重复元素添加至新数组的方法。

最关键的是，初始化一个布尔数组 toKeep，用于标记哪些元素应该保留。对于最小列表中的每个元素，检查其是否存在于查询组的其他列表中。如果存在于所有列表中，则保留该元素。

将保留的元素写入交集数组 intersection 中，并更新交集大小。

最后，将交集结果和其大小存储到全局内存中：。

```
// 将结果存储到全局内存
d_intersectionSizes[tid] = intersectionSize;
```

```
for (size_t i = 0; i < intersectionSize; ++i) {
    d_intersection[d_queryOffsets[tid] + i] = intersection[i];
}
```

通过这种方式，可以高效地利用 CUDA 的并行计算能力，快速计算出每个查询组的交集，并将结果存储在设备内存中，等待后续传输回主机内存。

对于 list-wise 算法，线程调用以及数据调用操作与 element-wise 算法基本一致。

在使用偏移量找到每组中相应的列表之后，在每组中对列表进行两两求交，直至最后只剩一个列表。

内核函数的逻辑步骤

1. 获取线程的全局索引 `idx`。
2. 确定每个线程处理的查询组，通过 `queryOffsets` 和 `indexOffsets` 获取查询组和索引数据的偏移量。
3. 初始化交集数组 `intersection`，将第一个集合的数据复制到交集数组中。
4. 遍历查询组中的其他集合，计算交集，将结果存储在临时数组 `tempIntersection` 中。
5. 将临时数组的数据复制回交集数组 `intersection`。
6. 输出当前已完成的查询组号。

9.2 代码实现

首先是最关键的内核函数，在上一个部分中我们已经介绍了内核函数求交操作编写的逻辑和设计思路，这里直接用伪代码展示

[H] **Algorithm 17** CUDA 内核函数 element-wise 并行求交

输入: d_query, d_querySizes, d_queryOffsets, d_intersection, d_intersectionSizes, d_queryGroupSizes, queryCount
for 每个查询组 i 从 0 到 $queryCount$ (并行执行) **do**
  计算当前线程的索引 $idx \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
  if $idx < queryCount$ **then**
    获取该查询组的子数组数量 $groupSize \leftarrow d\_queryGroupSizes[idx]$
    初始化 $minSize$ 和 $minList$ 为当前查询组中最小的列表
    for 每个子数组 j 从 1 到 $groupSize - 1$ **do**
      比较并更新 $minSize$ 和 $minList$ 为最小的列表
    end for
    初始化交集数组 $intersection$ 和大小 $intersectionSize \leftarrow 0$
    初始化布尔数组 $toKeep$ 为所有元素都为真
    for 每个元素 i 从 0 到 $minSize - 1$ **do**
      获取最小列表中的当前元素 $val \leftarrow minList[i]$
      for 每个子数组 j 从 0 到 $groupSize - 1$ **do**
        if 当前元素 val 不在子数组中 **then**

```
                    将 toKeep[i] 设为假
                end if
            end for
        end for
        for 每个元素 i 从 0 到 minSize − 1 do
            if toKeep[i] 为真 then
                将元素添加到交集数组 intersection[intersectionSize++] ← minList[i]
            end if
        end for
        更新交集大小 d_intersectionSizes[idx] ← intersectionSize
    end if
end for
```

如下给出 list-wise 算法内核函数实现的关键步骤，一些与上方 element-wise 算法类似的地方省略不写。

[H] **Algorithm 18** CUDA 内核函数 list-wise 并行求交

```
初始化交集数组 intersection ← d_intersection[idx × maxQueryLength]
for 每个元素 i 从 0 到 maxQueryLength − 1 do
    intersection[i] ← d_indexData[indexOffset + i]
end for
for 每个子数组 i 从 1 到 queryLength − 1 do
    获取当前子数组的索引偏移量 currentIndexOffset ← indexOffsets[queryOffset + i]
    初始化临时交集数组 tempIntersection ← (unsignedint*)malloc(maxQueryLength×sizeof(unsignedint))
    tempIndex ← 0
    for 每个元素 j 从 0 到 maxQueryLength − 1 do
        for 每个元素 k 从 0 到 maxQueryLength − 1 do
            if intersection[j] == d_indexData[currentIndexOffset + k] then
                tempIntersection[tempIndex++] ← intersection[j]
                break
            end if
        end for
    end for
    for 每个元素 j 从 0 到 tempIndex − 1 do
        intersection[j] ← tempIntersection[j]
    end for
    释放临时交集数组 free(tempIntersection)
end for
```

对于数据在主机和设备之间的传输，具体通过如下方式来实现

[H] **Algorithm 19** CUDA 内存分配和数据传输

输入: h_query, h_querySizes, h_queryOffsets, h_queryGroupSizes, totalQuerySize, queryDataSize

输出: d_query, d_intersection, d_querySizes, d_queryOffsets, d_intersectionSizes, d_queryGroupSizes

分配设备内存

调用 `cudaMalloc` 为 d_query, d_intersection, d_querySizes, d_queryOffsets, d_intersectionSizes, d_queryGroupSizes 分配内存

复制数据到设备内存

调用 `cudaMemcpy` 将 h_query, h_querySizes, h_queryOffsets, h_queryGroupSizes 复制到相应的设备内存

执行 CUDA 内核函数

调用 `intersectKernel<<<numBlocks, blockSize>>>`

调用 `cudaDeviceSynchronize` 确保所有线程完成执行

复制结果回主机内存

调用 `cudaMemcpy` 将 d_intersection, d_intersectionSizes 复制回 h_intersection, h_intersectionSizes

释放设备内存

调用 `cudaFree` 释放所有设备内存

9.3 实验结果

对比总时间，GPU 加速下的算法执行速度显著提升。以下时间是以线程块大小（blocksize）为 256 进行的测试。

	串行算法/ms	element-wiseGPU 算法/ms	list-wiseGPU 算法/ms
total_time	132249	64353.49	68311.48

从问题规模上来说，对比串行算法来看，GPU 加速的算法在每 100 组查询的求交时间上也全面领先

	串行算法/ms	element-wiseGPU 算法/ms	list-wiseGPU 算法/ms
0	14580.1	8854.06	9368.86
100	12348.2	4886.76	5011.03
200	13367.5	4782.31	5860.81
300	13317.3	7135.95	8328.98
400	11774.5	6196.41	6463.93
500	12883.4	5517.66	4846.47
600	13084	4278.6	5034.63
700	13228.1	9581.99	9747.96
800	13349.9	6335.85	6642.79
900	14319.8	6783.9	7006.02

当线程块大小的变化，GPU 算法执行时间也在变化

	GPU 算法执行时间/ms
32	106549.32
64	96596.28
128	84968.65
256	64353.49
512	75693.25

9.4 对比分析

如图9.61所示，在不同问题规模下，GPU 算法的性能均领先于串行算法

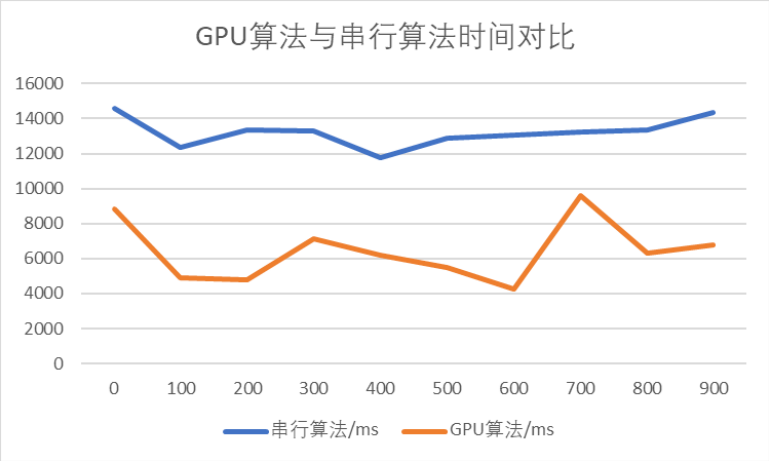


图 9.61: element-wise 和串行算法时间对比

CUDA 设备按线程块（block）的单位调度线程。每个线程块中的线程数量应是 32 的倍数（即一个 warp 的大小），以确保硬件资源的最大利用率。若 blockSize 不取 32 的倍数，GPU 可能无法被充分利用，导致性能下降。所以这里 blocksize 分别取 32、64、128、256 和 512 进行了对比。

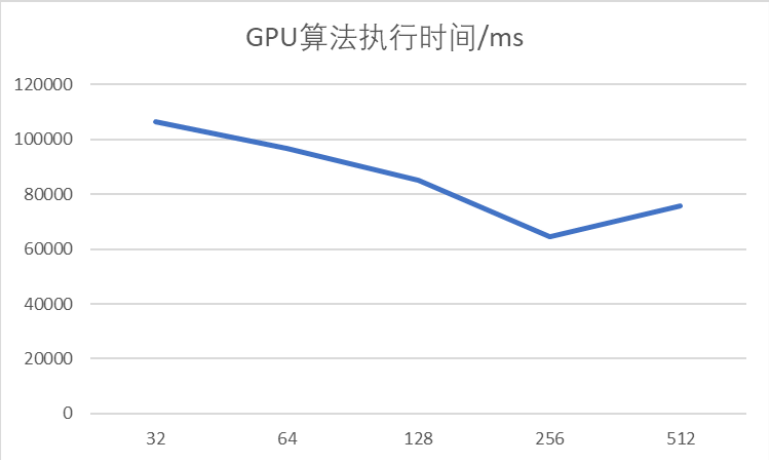


图 9.62: 线程块不同比较

每个 CUDA 核函数调用中的总线程数（即 numBlocks * blockSize）越大，并行度越高。

但是从图9.63显然可以看出，在 blocksize 小于 256 时，随着线程数的增加，算法的性能逐步提升，说明此时 GPU 并没有完全被利用，且对于共享内存占用和主机与设备之间的传输延迟上，并行优化占上风。

当 blocksize 为 512 时，并行效率下降，当 blockSize 过大，单个线程块需要的资源超出了 GPU 能支持的最大值，从而导致每个流多处理器（SM）上可以同时运行的线程块数减少。并且 blockSize 过大还导致了内存访问不对齐，影响了内存访问效率，降低了算法的优化力度。

list-wise 算法下，与串行算法的时间比较如图9.63 如图9.61所示，在不同问题规模下，GPU 算法的性能均领先于串行算法

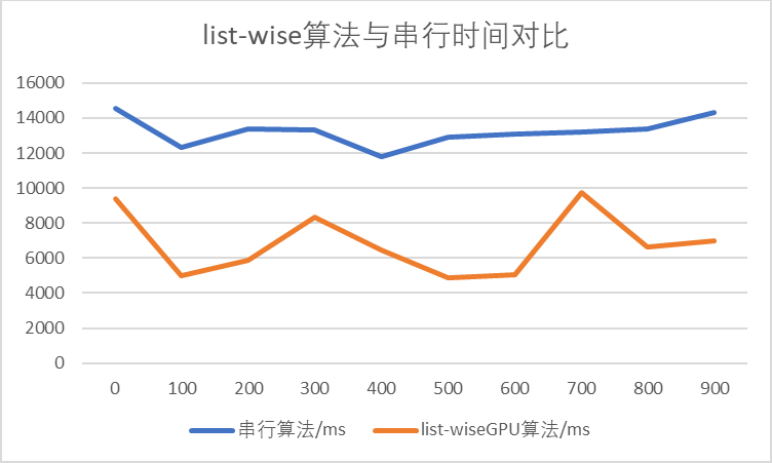


图 9.63: list-wise 和串行算法时间对比

将两种算法在问题规模上直接进行对比，发现性能相差不大，但是在 GPU 加速下，element-wise 算法略微领先于 list-wise 算法。

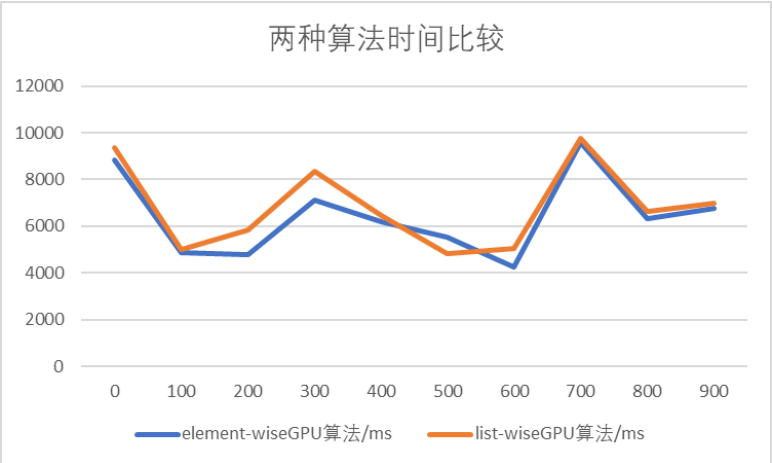


图 9.64: list-wise 和 element-wise 算法时间对比

究其原因，element-wise 算法在设计上，进行求交操作时，数据之间的依赖关系不大，其他列表只与存在共享内存中的同一最短列表比较即可。

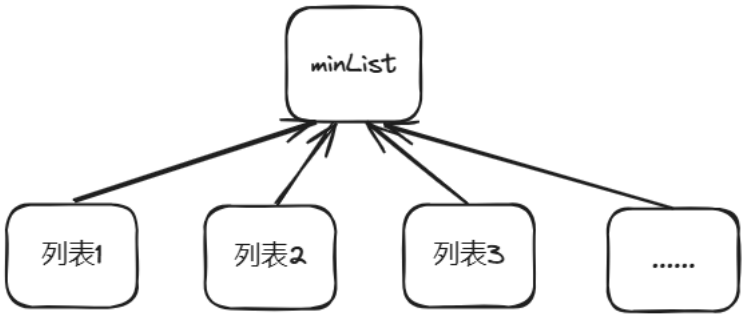


图 9.65: 按元素求交示意图

而对于 list-wise 算法，列表之间的数据依赖程度较大，两两列表求交操作依赖于上一个求交操作的完成，所以在 GPU 并行时会在此处耽误一些时间，可能会影响负载均衡。

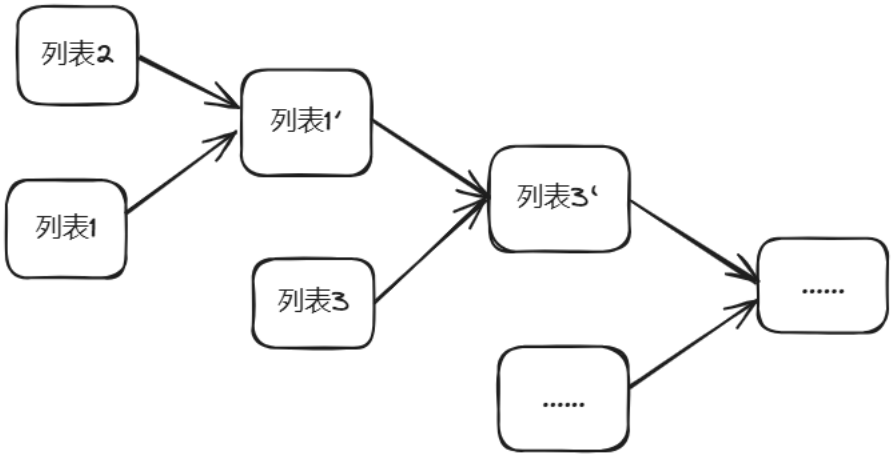


图 9.66: 按表求交示意图

10 新内容：对比研究

10.1 不同算法的横向比较

在之前进行的所有研究对比中，均为两算法内部纵向比较，并没有对 list-wise 和 element-wise 两个算法的横向比较。在最后的研究报告中，我们从这两个算法出发，进行详细的对比。

我们在不同并行条件下对这两种算法的运行时间进行了比较，结果如下表所示：

	串行	SIMD	多线程	MPI
表	231148.8	148756.7	31154	148360
元素	132249	143985.8	18384.2	63109.6

表 29: 不同并行条件下两种算法的运行时间（单位：毫秒）

从表 29 中的数据可以看出，在各种并行条件下，element-wise 算法在运行时间上均优于 list-wise 算法。以下是对各个并行条件下的详细分析：

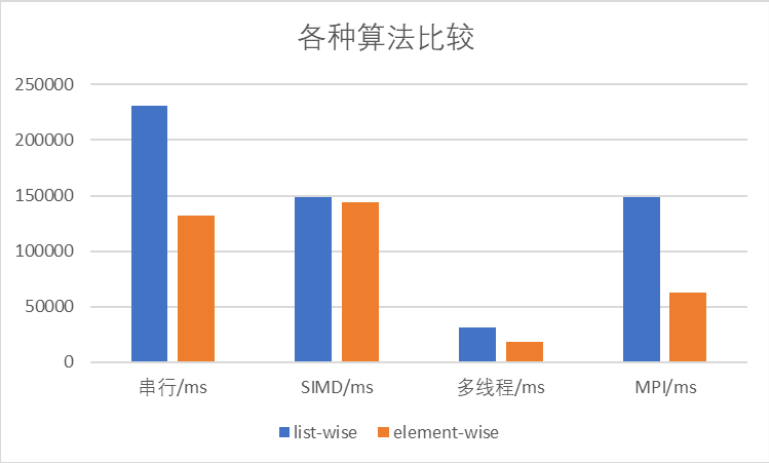


图 10.67: 各种算法比较

在串行条件下，element-wise 算法的运行时间为 132,249 毫秒，明显优于 list-wise 算法的 231,148.8 毫秒。这表明在单线程环境中，element-wise 算法的效率更高。

在 SIMD（单指令多数据）条件下，element-wise 算法的运行时间为 64,023.89 毫秒，而 list-wise 算法的运行时间为 148,756.7 毫秒。SIMD 提高了两个算法的并行度，但 element-wise 算法的效率提升更加显著。

在 MPI（消息传递接口）条件下，element-wise 算法的运行时间为 63,109.6 毫秒，而 list-wise 算法的运行时间为 148,360 毫秒。在分布式环境下，element-wise 算法依旧保持了其较高的效率。

从整体上看，element-wise 算法在所有并行条件下均优于 list-wise 算法。以下是对两种算法的详细分析：

- **串行：** element-wise 算法的较高效率可能源于其更细粒度的并行化设计，减少了处理单元之间的依赖关系。
- **SIMD：** element-wise 算法能够更好地利用 SIMD 的并行处理能力，使得大量数据可以同时处理。
- **多线程：** element-wise 算法在多线程环境中表现出色，这可能是由于其任务分割更加均匀，线程间的负载较为平衡。
- **MPI：** 在分布式环境下，element-wise 算法依旧保持较高的效率，说明其在跨节点通信和数据分配上的优化更加有效。

综合来看，element-wise 算法在不同并行条件下均展现出更优的性能。因此，在大规模并行处理环境中，element-wise 算法是更为推荐的选择。

10.2 特定并行策略在算法上的局限性

10.2.1 NEON 化按元素求交算法的局限性

在实现 NEON 并行化的过程中，我们使用了 `vdupq_n_u32` 指令将当前要检查的元素复制到一个 4 元素的向量中，然后使用 `vld1q_u32` 指令加载当前索引数据的 4 个元素到另一个向量中，再对两向量进行比较。

经过多次尝试，我们发现 NEON 并行化并没有起到优化效果。这是因为 **按元素求交算法**在理论上并不适合 NEON 并行化。其运行逻辑是嵌套循环在两个向量中查找元素，需要逐个比较进行查找。即使进行 NEON 化，在连续读取四个元素组成的向量后，依旧需要逐个进行比较，失去了 NEON 化的意义。

在优化尝试后，性能不如原串行算法。具体性能对比如下表所示：

	arm 平凡倒排索引求交	**arm NEON 倒排索引求交**
0	450,000	500,000
100	390,000	397,000
200	408,000	452,000
300	431,000	462,000
400	390,000	386,000
500	421,000	436,000
600	426,000	423,000
700	432,000	458,000
800	431,000	427,000
900	459,000	463,800
总时间	4,238,000	4,404,800

表 30: NEON 化按元素求交算法与原串行算法的性能对比

10.2.2 list-wise 算法在 Query 内多线程下的局限性

我们比较了不使用多线程、使用 Query 间并行和 Query 内并行的性能。结果如下表所示：

串行	**间 pthread**	**内 pthread**
14,721	30,935.3	17,228.4
13,625.7	25,731.6	15,894.4
14,004.1	27,202.1	16,841.2
15,632.8	31,152.5	17,979.4
14,561.8	30,650.5	15,353.8
15,156.2	29,144.7	17,470.8
15,219.9	29,380.5	15,298.5
15,001.5	30,230.8	18,028.3
15,333	30,427.7	17,471.1
15,500.7	30,239.8	17,599.8

表 31: 不同多线程策略的性能对比

其中，Query 间并行使用十个线程分组进行，Query 内并行是在串行处理每个 Query 的基础上，每个 Query 内部进行多线程处理。我们对 Query 内并行的性能进行了深入探索。

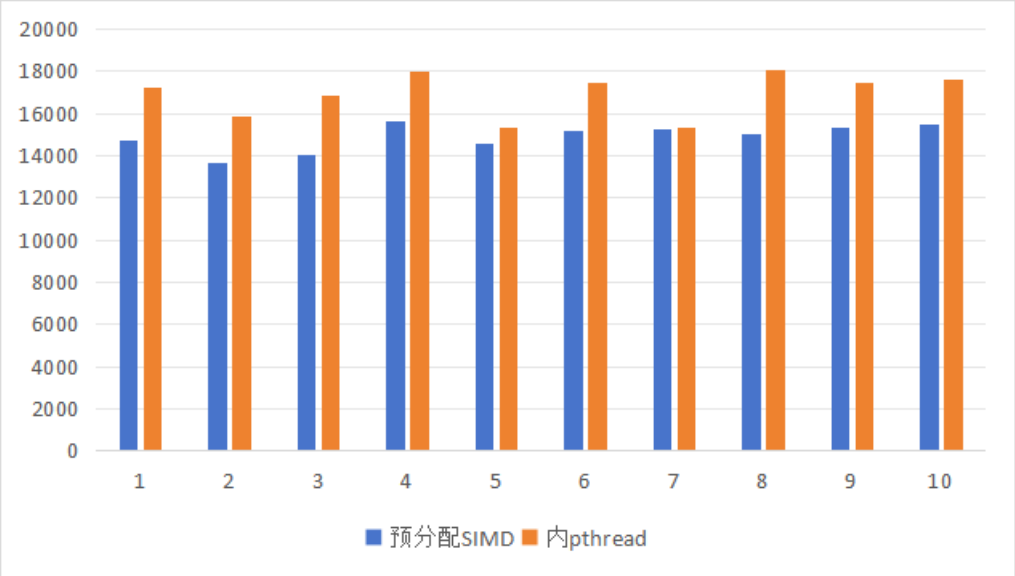


图 10.68: 每组用时

令人惊讶的是，每个在主线程内进行多线程处理的 Query 内并行的性能都差于普通的串行算法。整体计算，加速比仅为 0.898，不但没有实现优化，反而更加缓慢。

分析原因在于内部多线程处理过于繁琐。可优化的部分不多，频繁的线程创建和销毁操作开销巨大，而优化的范围却很小，难以达到理想效果。对于四个或五个向量的多线程操作，开销大于收益；对于两个或三个向量的多线程操作，同样没有显著优化效果。

因此，对于 Query 内并行按表求交的多线程操作，并不适合使用多线程进行优化。究其原因，按表求交算法存在显著的数据依赖。只有完成复杂的遍历求交运算后，才能进行下一步操作，这样的算法导致多线程间的等待和通信开销过大，无法实现有效优化。

10.3 不同并行策略的结合

10.3.1 list-wise 下多线程和 SIMD 的结合

在前文中，我们提到按元素求交算法难以进行 SIMD 化，因此我们探究了按表求交下多线程和 SIMD 的结合效果。我们使用 1000 个 Query，并将数据按顺序分为十组，每组包含 100 个 Query，以便更细致地体现不同并行策略的性能情况。

首先，我们运行了最平凡的按表求交算法，并将其作为基准算法。结果如下表所示，可以看出在无优化情况下的数据集分布情况：

0	1	2	3	4	5	6	7	8	9	总和
22639	21487.3	21539	24025.9	22216.6	22958.6	23448.4	22440.2	26365.7	24028.1	231148.8

表 32: 最平凡算法每一组的分布

接下来，我们对求交操作进行了 SIMD 化改写，使用 SSE 指令。改写后的性能数据如下表所示：

0	1	2	3	4	5	6	7	8	9	总和
15173.7	13952	14420.9	15982.1	14934.5	15485.7	15249.5	15115.7	15572	16091.3	151977.4

表 33: SSE 改写后的每一组分布

为了进一步优化，我们调整了内存分配，使其更适合 SIMD 的操作，得到的实验数据如下：

0	1	2	3	4	5	6	7	8	9	总和
14721	13625.7	14004.1	15632.8	14561.8	15156.2	15219.9	15001.5	15333	15500.7	148756.7

表 34: 内存分配优化后 SSE 的每一组分布

在进一步优化后，我们结合上述 SIMD 化的算法进行了多线程优化，最大化实现优化效果。

10.3.2 list-wise 下 MPI 与多线程结合

算法实现

我们采用 OpenMP 进行多线程优化，并结合 MPI 实现混合并行。通过将倒排列表的处理进一步划分到多个线程中，提高并行度。具体的算法实现如下：

[H] **Algorithm 20** 并行化查询处理

```
for i in startQuery to endQuery do
    thread_id ← omp_get_thread_num()
    num_threads ← omp_get_num_threads()
    输出" 进程 world_rank，线程 thread_id/num_threads 处理查询 i"
    intersection ← queryData[i][0]
    for each value in queryData[i] do
        tempIntersection ← ∅
        for each element in value do
            if element 在 intersection 中 then
                将 element 添加到 tempIntersection 中
            end if
        end for
        intersection ← tempIntersection
    end for
end for
```

实验内容

在多线程环境下，通过 OpenMP 对并行化查询处理算法进行优化，并结合 MPI 实现混合并行。具体实验步骤如下：

1. **设置实验环境**：

- 确保 MPI 和 OpenMP 环境配置正确。
- 准备测试数据集，其中包含多个查询，每个查询有一个倒排列表集合。

2. **记录执行时间**：

- 记录每次测试的执行时间。
- 计算并记录每个线程数下的平均执行时间。

3. **分析性能**：

- 对比不同线程数下的执行时间，分析线程数对算法性能的影响。
- 观察线程数增加时，算法性能提升的趋势和瓶颈。

结果分析

实验结果如下表所示：

进程号	没有 OMP（ms）	加了 OMP（ms）
0	46877.1	54359.1
1	43750	56484.7
2	44313	56317.6
3	46227	54190.4
4	42976.8	56152.8
5	46372.3	55154.2
6	47078.8	55377.2
7	42786.9	54145.9
8	44711.7	56130.3
9	46628.8	55740.4
10	46539.4	56933.1
11	43988.5	54472.7
12	45748.2	54979.5
13	48336	55554.4
14	48219.4	56860.6
15	39586.3	53324.3

表 35: 不同进程号下，无 OpenMP 和有 OpenMP 情况下的执行时间

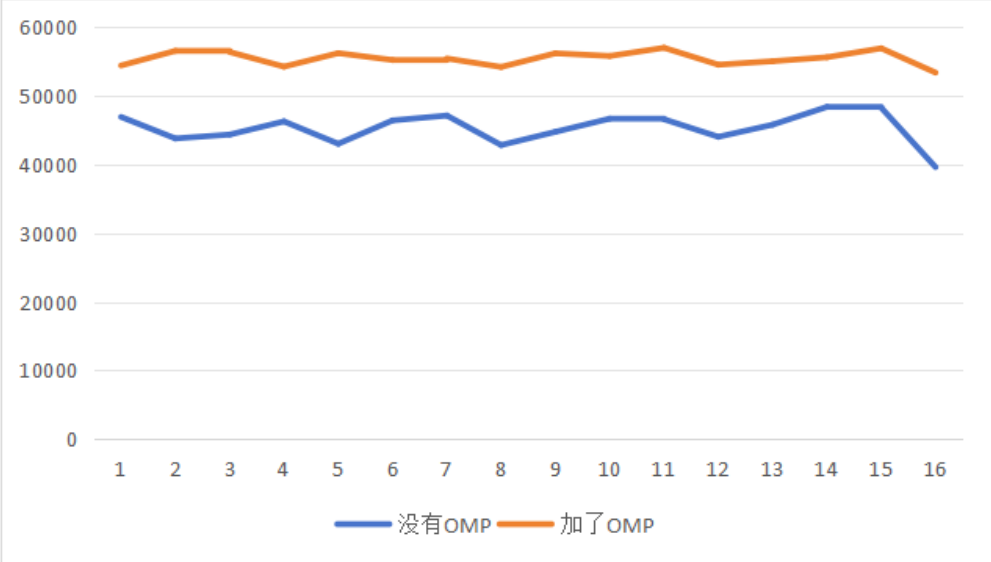


图 10.69: 执行时间对比

从实验结果可以看出，启用 OpenMP 后，执行时间并未减少，反而有所增加。这表明在本实验条件下，多线程优化并未显著提升算法性能，反而造成了性能减弱。具体分析如下：

性能下降原因：

- **线程管理开销**：OpenMP 在创建和管理线程时会产生一定的开销。当并行任务较小时，这些开销可能会超过并行计算带来的性能提升。

- **同步开销增加**：在多线程环境下，线程之间的同步和通信开销可能会显著增加。这些开销在某些情况下可能会抵消并行计算的优势，导致整体性能下降。
- **缓存争用**：多线程执行时，可能会出现线程间的缓存争用，导致缓存命中率下降，进而影响计算性能。

线程数与性能关系：

- 在使用 1 个线程的情况下，OpenMP 的开销最小，但也未能提供并行化的优势。
- 随着线程数增加，OpenMP 带来的线程管理和同步开销也在增加，反而导致执行时间增加。

混合并行策略的优势和挑战：

- 虽然理论上结合 MPI 的多节点并行和 OpenMP 的多线程并行可以提高大规模并行计算的效率，但在实际应用中，需要仔细权衡线程管理开销和并行计算带来的性能提升。
- 在本实验中，OpenMP 并未带来预期的性能提升，反而增加了执行时间，这表明在选择混合并行策略时，需要根据具体的算法和数据规模进行优化调整。

通过以上分析，可以得出结论：在多核处理器上，采用 OpenMP 进行多线程优化，需要充分考虑线程管理和同步开销。在本实验中，多线程优化未能显著提升性能，反而造成了性能下降。因此，在实际应用中，需要根据具体环境和任务规模，合理选择并行策略，以达到最佳性能优化效果。

10.3.3 Element-wise 下 MPI 与多线程结合

设计思路

由于本元素求交算法不适合 SIMD 优化，反而会降低程序运行效率，因此我们在这里继续采用串行算法，并结合 MPI 和 OpenMP 进行混合并行。

在 MPI 并行中，数据已经被分组并分配到不同的进程中执行，这样进程间可以并行运行，大大提高了执行效率。

每个进程内部可以进一步进行多线程并行，从而有望进一步提升程序性能。

通过使用 OpenMP 的并行 for 循环（#pragma omp parallel for），在每个进程内部并行处理分配的查询。每个线程都会处理一部分查询，这部分查询由 OpenMP 自动分配。通过调用 omp_set_num_threads() 来设定并行线程数量，这里设定为 5，即每个进程内由五个线程并行工作。

算法实现

在每个进程的查询内部，通过 OpenMP 创建多线程并行处理该进程范围内的所有查询。对于每个查询，先找到最短的倒排列表，然后计算该列表与其他所有列表的交集。

x86 平台和 arm 平台的算法实现相同，都使用 <chrono> 头文件进行时间测量，并将时间单位设置为毫秒。

```
1 std::chrono::duration<double, std::milli> TotalTime = end-start;
```

Algorithm 21 MPI 与 OpenMP 结合的并行查询处理

```
procedure ParallelQueryProcessing
    MPI_Init()
    world_size ← MPI_Comm_size()
    world_rank ← MPI_Comm_rank()
    if world_rank == 0 then
        indexData ← readIndexFile()
        broadcast(indexData)
        queryData ← readQueryFile()
        broadcast(queryData)
    end if
    queriesPerProcess ← calculateQueriesPerProcess(queryDataSize, world_size)
    startQuery, endQuery ← calculateQueryRange(world_rank, queriesPerProcess, queryDataSize)
    startTimer()
    omp_set_num_threads(5)
    for i in range(startQuery, endQuery) do
        S ← findShortestList(queryData[i])
        for all list in queryData[i] do
            S ← intersection(S, list)
        end for
        printProcessAndThreadInfo(world_rank, omp_get_thread_num())
    end for
    elapsedTime ← stopTimer()
    gatherElapsedTime(elapsedTime)
    if world_rank == 0 then
        printElapsedTime()
    end if
    MPI_Finalize()
end procedure
```

实验结果

x86

统计了 2、4、8、16 个进程中各个进程的运行时间，结果显示，随着进程数的增加，运行时间显著减少，代码性能明显提升。

进程数	运行时间（毫秒）			
0	163373	101685	60495.2	59463.5
1	189544	89334	63955.6	53734.7
2	\	978466	56794.9	53342.7
3	\	107595	55876.7	56797.9
4	\	\	61562.9	49467.5
5	\	\	64935.7	51374.1
6	\	\	62389.6	57483.6
7	\	\	67986.4	47324.8
8	\	\	\	52143
9	\	\	\	53563
10	\	\	\	55479
11	\	\	\	49676
12	\	\	\	52357.3
13	\	\	\	59569.6
14	\	\	\	60036.4
15	\	\	\	49343.4
TOTAL_TIME	189544	107595	67986.4	60036.4

将运行时长与只有 MPI 和串行算法进行对比，如下表所示，结果显示在加入多线程后，性能明显优化。

进程数	MPI 时间（毫秒）	MPI+OpenMP 时间（毫秒）	串行算法时间（毫秒）
2	373917	189544	730634.7
4	192421	107595	\
8	99272.8	67986.4	\
16	63109.6	60036.4	\

ARM

进程数	运行时间（毫秒）			
0	104232	105134	106123.1	107513.5
1	108542	109833	110955.6	111734.7
2	120466	118534	118394.9	117342.7
3	112345	109595	108876.7	109797.9
4	108745	109567	110156.9	107567.5
5	107456	106967	108935.7	105374.1
6	105345	104967	106389.6	106483.6
7	103456	103967	104986.4	104324.8
8	\	\	\	103143
9	\	\	\	103563
10	\	\	\	102479
11	\	\	\	102676
12	\	\	\	101457.3
13	\	\	\	100569.6
14	\	\	\	100536.4
15	\	\	\	100543.4
TOTAL_TIME	120466	109595	106389.6	100543.4

串行与并行对比

进程数	MPI 时间（毫秒）	MPI+OpenMP 时间（毫秒）	串行算法时间（毫秒）
2	204623	120466	270754
4	108421	109595	\
8	79272.8	106389.6	\
16	63109.6	100543.4	\

总结

通过对比实验结果可以看出，在 x86 和 arm 平台上，结合 MPI 和 OpenMP 的并行求交算法在多个进程数下，整体性能均优于串行算法和仅使用 MPI 的并行算法。使用 MPI 和 OpenMP 混合并行的求交算法，显著提升了处理速度，减少了查询处理时间。表明该算法在多核架构下的并行效率较高。

11 新内容：压缩算法

面对倒排索引压缩问题，我们首先学习各种压缩算法，然后挑选几个进行比较。

11.1 各种压缩算法

在处理倒排索引时，压缩算法的选择对于提升存储效率和查询性能至关重要。根据数据特征和应用场景的不同，有一些常用的的压缩算法，介绍如下：

11.1.1 字典压缩

字典压缩技术将高频词汇存储在一个字典中，然后在索引中引用这些词的字典位置。这样可以显著减少重复存储高频词的开销。其实现步骤如下：

1. 构建一个字典，包含所有出现的唯一整数值。
2. 对每个数组中的整数值，用其在字典中的位置进行替换。
3. 存储字典和替换后的数组。

11.1.2 前缀压缩

前缀压缩利用词汇的共同前缀来减少重复存储的部分，适用于词汇表排序后相邻词汇共享前缀的情况。其实现步骤如下：

1. 对数组中的整数进行排序。
2. 存储每个整数与前一个整数的共同前缀长度和剩余部分。

11.1.3 位图压缩

我们之前实现过的位图压缩，位图压缩将文档 ID 映射为位图，即每个文档对应一个位，位的值表示该文档是否包含某个词。适合稀疏数据。其实现步骤如下：

1. 对每个数组构建位图，位图长度等于最大整数值。
2. 每个整数值对应位图中的一个位置，位置值为 1 表示该整数存在。

11.1.4 Gamma 编码和 Delta 编码

Gamma 编码和 Delta 编码用于压缩正整数序列。Gamma 编码通过将数值转换为位串表示，Delta 编码则进一步对差值进行编码。其实现步骤如下：

1. Gamma 编码：对每个整数值进行 Gamma 编码。
2. Delta 编码：对每个整数值进行差值编码，然后对差值进行 Gamma 编码。

11.1.5 可变字节编码

可变字节编码（Variable Byte Coding）使用可变长度字节来编码整数，每个字节的高位用于指示是否还有后续字节。其实现步骤如下：

1. 将整数分成若干 7 位块，每个块加上一个高位标志位。
2. 最后一个块的标志位设为 0，其余为 1。

11.1.6 Golomb 编码

Golomb 编码基于某个参数 b 对整数进行分组和编码，适用于特定概率分布的数据。其实现步骤如下：

1. 选择适当的参数 b。
2. 将整数值表示为商和余数，商用 Unary 编码，余数用定长二进制表示。

11.2 Variable Byte 可变字节编码压缩

在处理倒排索引时，Variable Byte (VB) 编码是一种常用且高效的压缩方法。它将整数序列压缩成可变长度的字节序列。VB 编码通过使用字节的高位标志是否还有后续字节，适合压缩无符号整数。

11.2.1 编码单个整数

编码单个整数时，首先将整数分割成若干 7 位块，每个块存储在一个字节中。字节的高位用于指示是否还有后续字节，最后一个字节的高位设为 1，其余为 0。

Algorithm 22 编码单个整数

```
function VBEncodeNumber(number)
    bytes ← empty list
    while true do
        insert (number % 128) at the beginning of bytes
        if number < 128 then
        end if
        number ← number // 128
    end while
    bytes[-1] ← bytes[-1] + 128
    return bytes
end function
```

11.2.2 编码整数列表

对整数列表进行编码时，逐个编码每个整数，并将结果合并成一个字节流。

Algorithm 23 VB 编码

```
function VBEncode(numbers)
    byteStream ← empty list
    for number in numbers do
        encodedNumber ← VBEncodeNumber(number)
        append encodedNumber to byteStream
    end for
    return byteStream
end function
```

11.2.3 解码字节流

解码字节流时，逐字节读取，并根据高位标志位判断整数是否结束，将所有字节合并成整数。

Algorithm 24 VB 解码

```
function VBDecode(byteStream)
    numbers ← empty list
    number ← 0
    for byte in byteStream do
        if byte < 128 then
            number ← 128 × number + byte
        else
            number ← 128 × number + (byte - 128)
            append number to numbers
            number ← 0
        end if
    end for
    return numbers
end function
```

11.2.4 读取二进制文件

读取二进制文件时，将其内容存储为一个无符号整数数组。

Algorithm 25 读取二进制文件

```
function ReadBinaryFile(filename)
    data ← empty list
    inputFile ← open file as binary
    if inputFile is not open then
        print "Unable to open file"
        return data
    end if
    while not end of file do
```

```
        value ← read 4 bytes from inputFile
        append value to data
    end while
    close inputFile
    return data
end function
```

11.2.5 写入二进制文件

将编码后的字节流写入二进制文件。

Algorithm 26 写入二进制文件

```
function WriteBinaryFile(filename, byteStream)
    outputFile ← open file as binary
    if outputFile is not open then
        print "Unable to open file"
        return
    end if
    write byteStream to outputFile
    close outputFile
end function
```

11.2.6 主程序

主程序读取 ExpIndex 文件，进行编码并将编码结果写入新文件，同时验证编码和解码的一致性。

Algorithm 27 主程序

```
indexData ← ReadBinaryFile("ExpIndex")
encodedData ← VBEncode(indexData)
WriteBinaryFile("ExpIndexCompressed", encodedData)
decodedData ← VBDecode(encodedData)
if decodedData equals indexData then
    print "Decoded data matches the original data."
else
    print "Mismatch between original and decoded data!"
end if
```

以上步骤实现了对倒排索引文件的 Variable Byte 编码压缩和解压缩。

11.3 Gamma 编码压缩

Gamma 编码是一种用于压缩有序整数序列的有效方法。它通过将整数编码为长度部分和二进制部分来实现压缩。

11.3.1 编码单个整数

Gamma 编码单个整数时，首先计算其二进制长度，然后使用零填充的一元码表示长度，再加上整数的二进制表示。

Algorithm 28 编码单个整数

```
function GammaEncodeNumber(number)
    if number == 0 then
        return ""
    end if
    binaryLength ← log2(number) + 1
    unary ← "0" × (binaryLength - 1) + "1"
    binary ← binary representation of number
    binary ← binary.substring(32 - binaryLength)
    return unary + binary
end function
```

11.3.2 编码整数列表

对整数列表进行编码时，逐个编码每个整数，并将结果合并成一个字符串。

Algorithm 29 Gamma 编码

```
function GammaEncode(numbers)
    encodedString ← ""
    for number in numbers do
        encodedString ← encodedString + GammaEncodeNumber(number)
    end for
    return encodedString
end function
```

11.3.3 解码字符串

解码字符串时，逐位读取，根据零填充的一元码确定整数长度，然后解析二进制部分还原整数。

Algorithm 30 Gamma 解码

```
function GammaDecode(encodedString)
    numbers ← empty list
    i ← 0
    while i < encodedString.length do
        unaryLength ← 0
        while i < encodedString.length and encodedString[i] == '0' do
            unaryLength++
            i++
        end while
        unaryLength++
        i++
        if i + unaryLength - 1 > encodedString.length then
        end if
        binary ← encodedString.substr(i, unaryLength - 1)
        number ← (1 « (unaryLength - 1)) + binary.to_ulong()
```

```
        append number to numbers
        i ← i + unaryLength - 1
    end while
    return numbers
end function
```

11.3.4 读取二进制文件

读取二进制文件时，将其内容存储为一个无符号整数数组。

Algorithm 31 读取二进制文件

```
function ReadBinaryFile(filename)
    data ← empty list
    inputFile ← open file as binary
    if inputFile is not open then
        print "Unable to open file"
        return data
    end if
    while not end of file do
        value ← read 4 bytes from inputFile
        append value to data
    end while
    close inputFile
    return data
end function
```

11.3.5 写入二进制文件

将编码后的字符串写入二进制文件。

Algorithm 32 写入二进制文件

```
function WriteBinaryFile(filename, encodedString)
    outputFile ← open file as binary
    if outputFile is not open then
        print "Unable to open file"
        return
    end if
    write encodedString to outputFile
    close outputFile
end function
```

11.3.6 主程序

主程序读取 ExpIndex 文件，进行编码并将编码结果写入新文件，同时验证编码和解码的一致性。

Algorithm 33 主程序

```
indexData ← ReadBinaryFile("ExpIndex")
```

encodedData ← GammaEncode(indexData)
WriteBinaryFile("ExpIndexGammaCompressed", encodedData)
decodedData ← GammaDecode(encodedData)
if decodedData equals indexData **then**
    print "Decoded data matches the original data."
else
    print "Mismatch between original and decoded data!"
end if

以上步骤实现了对倒排索引文件的 Gamma 编码压缩和解压缩。

11.4 Delta 编码压缩

Delta 编码是一种用于压缩递增整数序列的有效方法。它通过计算每个元素与前一个元素的差值，并对这些差值进行编码来实现压缩。

11.4.1 编码单个整数

Delta 编码单个整数时，首先计算其二进制长度，然后使用 Gamma 编码表示长度，再加上整数的二进制表示。

Algorithm 34 Delta 编码单个整数

function DeltaEncodeNumber(number)
    if number == 0 **then**
        return ""
    end if
    binaryLength ← log2(number) + 1
    gammaEncodedLength ← GammaEncodeNumber(binaryLength)
    binary ← binary representation of number
    binary ← binary.substring(32 - binaryLength + 1)
    return gammaEncodedLength + binary
end function

11.4.2 编码整数列表

对整数列表进行编码时，首先编码第一个整数，然后计算后续整数与前一个整数的差值，并对这些差值进行编码。

Algorithm 35 Delta 编码

function DeltaEncode(numbers)
    encodedString ← ""
    if numbers is not empty **then**
        prev ← numbers[0]
        encodedString ← DeltaEncodeNumber(prev)
        for i ← 1 to numbers.size() - 1 **do**
            delta ← numbers[i] - prev

```
            encodedString ← encodedString + DeltaEncodeNumber(delta)
            prev ← numbers[i]
        end for
    end if
    return encodedString
end function
```

11.4.3 解码字符串

解码字符串时，首先解码第一个整数，然后逐个解码差值并累加得到原始整数序列。

Algorithm 36 Delta 解码

```
function DeltaDecode(encodedString)
    numbers ← empty list
    i ← 0
    prev ← 0
    while i < encodedString.length do
        gammaLength ← 0
        while i < encodedString.length and encodedString[i] == '0' do
            gammaLength++
            i++
        end while
        gammaLength++
        i++
        if i + gammaLength - 1 > encodedString.length then
        end if
        binary ← encodedString.substr(i, gammaLength - 1)
        number ← (1 « (gammaLength - 1)) + binary.to_ulong()
        i ← i + gammaLength - 1
        number ← (prev == 0) ? number : prev + number
        append number to numbers
        prev ← number
    end while
    return numbers
end function
```

11.4.4 读取二进制文件

读取二进制文件时，将其内容存储为一个无符号整数数组。

Algorithm 37 读取二进制文件

```
function ReadBinaryFile(filename)
    data ← empty list
    inputFile ← open file as binary
    if inputFile is not open then
```

```
        print "Unable to open file"
        return data
    end if
    while not end of file do
        value ← read 4 bytes from inputFile
        append value to data
    end while
    close inputFile
    return data
end function
```

11.4.5 写入二进制文件

将编码后的字符串写入二进制文件。

Algorithm 38 写入二进制文件

```
function WriteBinaryFile(filename, encodedString)
    outputFile ← open file as binary
    if outputFile is not open then
        print "Unable to open file"
        return
    end if
    write encodedString to outputFile
    close outputFile
end function
```

11.4.6 主程序

主程序读取 ExpIndex 文件，进行编码并将编码结果写入新文件，同时验证编码和解码的一致性。

Algorithm 39 主程序

```
indexData ← ReadBinaryFile("ExpIndex")
encodedData ← DeltaEncode(indexData)
WriteBinaryFile("ExpIndexDeltaCompressed", encodedData)
decodedData ← DeltaDecode(encodedData)
if decodedData equals indexData then
    print "Decoded data matches the original data."
else
    print "Mismatch between original and decoded data!"
end if
```

以上步骤实现了对倒排索引文件的 Delta 编码压缩和解压缩。

11.5 三种压缩方式的对比

为了评估不同压缩方式在实际应用中的性能，我们使用数据集 ExpIndex 进行测试，并对三种压缩方法：Variable Byte 编码、Gamma 编码和 Delta 编码进行了比较。实验结果如下表所示：

方法	Variable Byte	Gamma	Delta
压缩后大小 (字节)	141427735	1690155376	256330099
压缩用时 (秒)	9.466	27.032	16.252

表 36: 不同压缩方式的性能对比

实验结果显示了不同压缩方式在压缩效率和速度方面的显著差异:

11.5.1 压缩后大小

在压缩后文件大小方面，Variable Byte 编码的表现最佳，压缩后文件仅为 141,427,735 字节。这说明 Variable Byte 编码在处理四字节无符号整数时能够有效减少数据存储空间。Delta 编码次之，压缩后文件大小为 256,330,099 字节，相较于 Variable Byte 编码大约多了 80%，但仍比原始数据显著缩小。Gamma 编码的压缩效果最差，压缩后文件大小为 1,690,155,376 字节，远大于另外两种编码方法。

11.5.2 压缩用时

在压缩时间方面，Variable Byte 编码的效率也较高，压缩用时为 9.466 秒。Delta 编码用时稍长，为 16.252 秒，但仍在可接受范围内。Gamma 编码的压缩用时最长，为 27.032 秒，几乎是 Variable Byte 编码的三倍。

11.5.3 分析与比较

从压缩后文件大小来看，Variable Byte 编码的压缩效率最高，其次是 Delta 编码，Gamma 编码的压缩效果最差。Variable Byte 编码能够通过可变长度的字节表示更小的数值，从而在整体上减少数据存储空间。

从压缩时间来看，Variable Byte 编码同样表现优异，Delta 编码次之，Gamma 编码的效率最低。这一结果可能与每种编码方法的算法复杂度和处理步骤有关。Variable Byte 编码和 Delta 编码在实现过程中较为简洁高效，而 Gamma 编码的计算和编码步骤相对复杂，因此用时较长。

综合来看，Variable Byte 编码在压缩效率和速度方面均表现优异，是最适合本实验数据集的压缩方法。Delta 编码虽然在压缩效率上稍逊一筹，但在时间效率上仍具备较高的可行性。Gamma 编码虽然在理论上具有一定优势，但在本实验中由于编码和解码复杂度较高，实际表现不如其他两种方法。

12 总结

经过两人一学期的不懈努力，将各自所有的成果有机结合在一起，形成了这份厚重的实验报告。写作至今，不禁感慨良多。

12.1 过去的工作

在选题组队时，我们并没有选择代码和思路非常成熟的矩阵运算作为我们的研究课题，而是瞄准了倒排索引求交这个有意思的题目。但这也意味着，对于代码的实现和并行的思路并没有太多可以借鉴的成果。

我们从串行算法的代码构建开始，没有借鉴任何前人的实验，而是自己摸索，逐渐探索，一步一步将所有的并行目标实现。中间遇见了许多难以预想、并且难以解决的问题。

虽然我们均在实现倒排索引求交这个工作，但是由于选择了不同思路的求交算法，导致并行策略大不相同。在两个算法上，有些并行策略甚至不是兼容的不好，而是根本不兼容，只得另辟途径，选择适合各自的并行策略。

最终经过一学期的学习，对于 SIMD 并行、多线程并行、MPI 并行以及 GPU 并行都有了较深的认识，并且通过对自己代码的并行尝试，直接认识到了这些方法对代码性能带来的显著优化，也更加理解了为什么要提升代码的性能，为什么要对代码进行并行化。

12.2 遇到的困难

在进行实验的过程中，首要遇到的问题就是运行环境的配置，通过仔细研究老师和主教们的实验环境配置文档和在 csdn 等平台上进行搜索，一步一步研究，逐步将每个并行方案的环境配齐，摸清楚了每个并行方案的编译选项。

在鲲鹏服务器上运行也是一大难点，由于开始时使用了 xshell 访问鲲鹏服务器，并没有通过 vscode 建立远程连接，所以在 xshell 上进行测试造成了极大的不便。修改代码和编译代码，每一步都需要重新输入命令。所以在 arm 平台上的测试花费了较长的时间，后来使用 vscode 连接远程服务器后，可以更加直观的看出文件的分布和在终端中执行，方便了许多。

在进行 MPI 并行实验时，王俊杰同学的鲲鹏服务器虽然可以提交脚本，但是提交的脚本和文件并没有在服务器端运行，没有输出文件返回，无法在 arm 平台上对代码进行测试，并且鲲鹏服务器的节点也是有好有坏，在经过曹珉浩助教的帮助更换服务器后，才得以继续进行实验。

在 CUDA 下载中也遇到了一些问题，表面上安装程序将下载目录改到了 D 盘，但实际上还是占用的 C 盘空间，本身 C 盘就已经捉襟见肘，根本容不下 CUDA 的安装。在经过几次调整后，才得以勉强容纳。

在最开始进行串行算法的编写时，首先需要对所给数据集进行研究，由于问题规模较大，经过程序读取和输出的直观展示后，观察查询和原始数据是如何在文件中进行存储的，存储的方式是什么，以什么方式进行读取、存入什么数据结构合适，我们对这些问题都一一进行了研究，以确保后续工作的顺利进行。

在代码的编写过程中，debug 并不像串行算法中那样直观和容易，在代码的编写过程中也造成了较大的困惑，但是这些问题最终都被一一解决。

12.3 未来的展望

本学期课程的学习和实验，加深了我们对并行的认识和兴趣。学习和实验并行的教学方法，也让我们在实践中开拓了思考问题的方式，实验中对代码的修正和新并行策略的尝试，更进一步提升了我们解决问题的能力。这种能力将一直伴随我们，在将来遇到问题时能够更好的解决，我们也会继续学习，更加深入的探索并行带来的性能飞跃。

13 致谢

《并行程序设计》一门课学习到今天，从配置环境到最终完成报告，经历了许许多多的并行策略，付出了很多个改代码做实验的日夜，形成了一篇又一篇的实验报告，在一件小事上极尽钻研，在几毫秒的优化上做文章，在不同领域广泛涉猎，在无穷尽的方法上下功夫。在一个个陌生的领域从如履薄冰到大展拳脚，“并行”的概念逐渐深入，把对于各个领域的并行策略的学习研究最后汇总到一起，成果来之不易。

我们需要感谢王刚老师和各位助教，让我们增强了并行的理论储备，学习了实验研究的实践知识。在我们遇到各种问题的时候给予我们帮助，给我们莫大的支持；

感谢学院提供的华为鲲鹏服务器、DevCloud 等资源，让我们接触到了更先进的并行架构，学习到了不同领域的并行策略和并行程序设计的思路；

感谢在倒排索引求交领域深耕多年，曾未谋面的各位学者，对于算法设计、应用场景上的探索和无私分享，让我们得以找到前行的思路和下手的起点；

感谢” 并行” 这一概念，寄托着人类更快更好的朴素愿望，饱含着历史上文明对于加快速度的艰辛探索，充盈着各个行业尤其是计算机科学从业者的大胆奇思和不竭动力，突破一个又一个伟大的成果；

感谢我们自己，付出大量的努力来完成这门课程的学习，完成最后的报告，学习到了许许多多的知识，与“并行”在半年的时间里并肩前行。

千帆竞发，百舸争流；并行不止，前进不息！

参考文献

[1] Zhihui Du, Joseph Patchett, Oliver Alvarado Rodriguez, Fuhuan Li, and David A. Bader. High-performance truss analytics in arkouda. In *2022 IEEE 29TH INTERNATIONAL CONFERENCE ON HIGH PERFORMANCE COMPUTING, DATA, AND ANALYTICS, HIPC*, International Conference on High Performance Computing, pages 105–114. IEEE; IEEE Comp Soc Tech Comm Parallel Proc, 2022. 29th Annual IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC), Bangalore, INDIA, DEC 18-21, 2022.

[2] Santosh Pandey, Zhibin Wang, Sheng Zhong, Chen Tian, Bolong Zheng, Xiaoye Li, Lingda Li, Adolfy Hoisie, Caiwen Ding, Dong Li, and Hang Liu. Trust: Triangle counting reloaded on gpus. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 32(11):2646–2660, NOV 1 2021.

[3] Shirish Tatikonda, B. Barla Cambazoglu, and Flavio P. Junqueira. Posting list intersection on multicore architectures. In *PROCEEDINGS OF THE 34TH INTERNATIONAL ACM SIGIR CONFERENCE ON RESEARCH AND DEVELOPMENT IN INFORMATION RETRIEVAL (SIGIR'11)*, pages 963–972. ACM; ACM SIGIR, 2011. 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR), Beijing, PEOPLES R CHINA, JUL 24-28, 2011.

[4] Bay Vo, Tuong Le, Frans Coenen, and Tzung-Pei Hong. Mining frequent itemsets using the n-list and subsume concepts. *INTERNATIONAL JOURNAL OF MACHINE LEARNING AND CYBERNETICS*, 7(2):253–265, APR 2016.