南開大學
Nankai University

计算机学院

并行程序设计期末报告

倒排索引求交 element-wise 算法 SIMD 并行化

姓名：刘家骥

学号：2211437

专业：计算机科学与技术

2024 年 4 月 28 日

目录

1 问题描述

1.1 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。

1.2 本次实验子问题及分工

1. list-wise 串行算法实现，负责人：王俊杰
2. 位图索引方式的建立，负责人：王俊杰
3. 二级索引方式的建立，负责人：王俊杰
4. SSE 的 C/C++ 编程实现，负责人：王俊杰
5. 在 x86 平台上的并行优化，负责人：王俊杰
6. element-wise 串行算法实现，负责人：刘家骥
7. Neon 的 C/C++ 编程实现，负责人：刘家骥
8. 在 ARM 平台上的并行优化，负责人：刘家骥

本次实验主要由王俊杰和刘家骥两人共同完成，本文负责 list-wise 串行算法实现，位图索引方式的建立，二级索引方式的建立，NEON 的 C/C++ 编程实现，以及在 x86 平台上的并行优化。有关代码文件已经上传到Github上。

2 实验平台

2.1 硬件平台

x86 架构

- 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

ARM 架构

- 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

2.2 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]

3 倒排索引求交 element-wise 串行算法

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。

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
9   LARGE_INTEGER frequency;        // ticks per second
10  LARGE_INTEGER t1, t2;           // ticks
11  vector<double> elapsedTime(queryDataSize/step);
12
13  // get ticks per second
14  QueryPerformanceFrequency(&frequency);
15
16  QueryPerformanceCounter(&t1); // start timer at the beginning of the loop
17
18  for(size_t i=0;i<queryDataSize;i++)
19  {
20      size_t minSize = queryData[i][0].size();
21      size_t minIndex = 0;//最短列表的数组下标
22          for (size_t j = 0; j < queryData[i].size(); j++)
                //queryData[i].size()为每组列表的个数
23          {
24              if (queryData[i][j].size() < minSize)
25              {
26                  minSize = queryData[i][j].size();
27                  minIndex = j;
28              }
29          }
30      vector<unsigned int> S = queryData[i][minIndex];//最短的列表
31      for (size_t k = 0; k < queryData[i].size(); k++) //检查
32      {
33          auto it = S.begin();
34          while (it != S.end())
35          {
36              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()
37              {
38                  it = S.erase(it);//未找到即无交集，删除它
39              }
40              else
41              {
42                  it++;
43              }
44          }
45          cout<<i<<" k:"<<k<<endl;
46      }
47      intersectionResults.push_back(S);//S即为每组的交集
48      times++;
```

```
49          if(times%step==0)
50          {
51              // stop timer
52          QueryPerformanceCounter(&t2);
53
54          // 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      }
```

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。

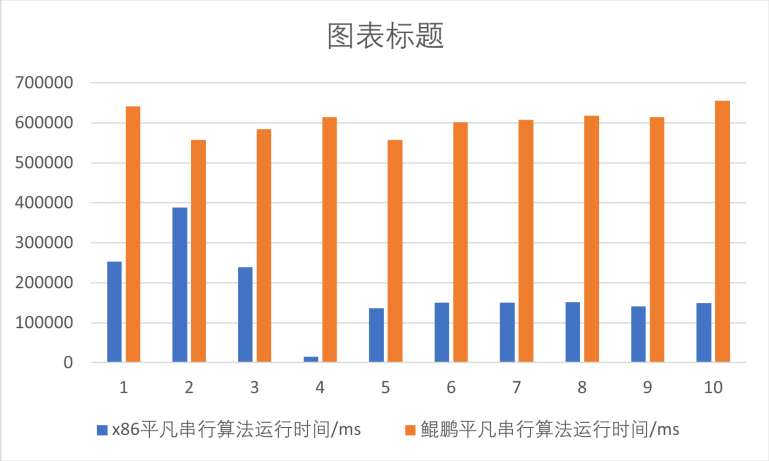


图 3.1: 平凡串行算法时间比较

4 倒排索引求交 element-wise 串行算法——以二级索引方式存储

4.1 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。

4.2 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 1 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
```

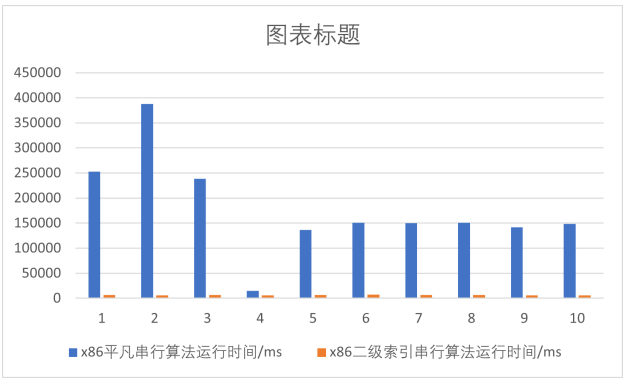
对于求交算法的实现，我们对 S 的二级索引 S_second 和所比较二级索引按位进行与运算即可，如果按位与结果为 0，对 S 即 S_second 进行 reset 置零操作。但是注意此处如果按位与结果均为 1，并不一定说明存在相交元素，还应该进入位图中继续逐位进行与运算，若此时按位与运算结果为 1，说明此处确实存在相交元素，这也是在编程过程中我遇到的一个找了很久的 bug。

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。

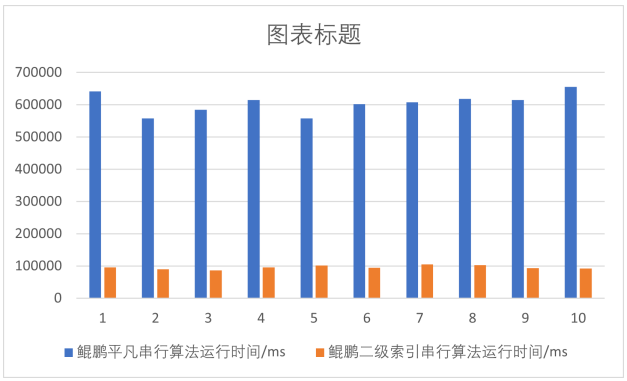
```

    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l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。

4.3 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 4.2: 不同平台串行优化算法的执行时间对比

5 对二级索引及位图存储下的 elemeng-wise 算法的 NEON 并行化

5.1 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。

5.2 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 2 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。

5.3 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。

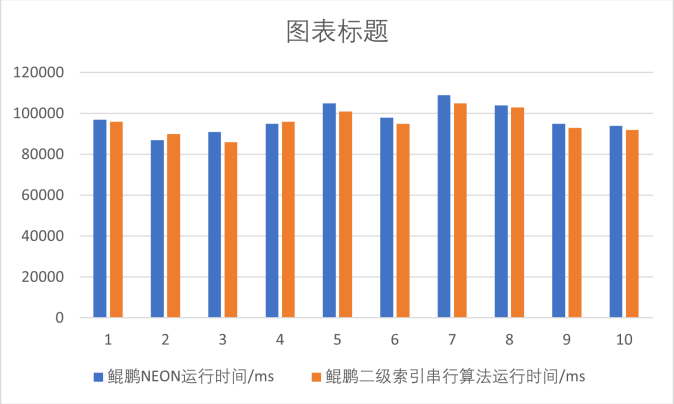


图 5.3: 并行算法与串行算法优化比较

6 实验结果分析

如图6.4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。

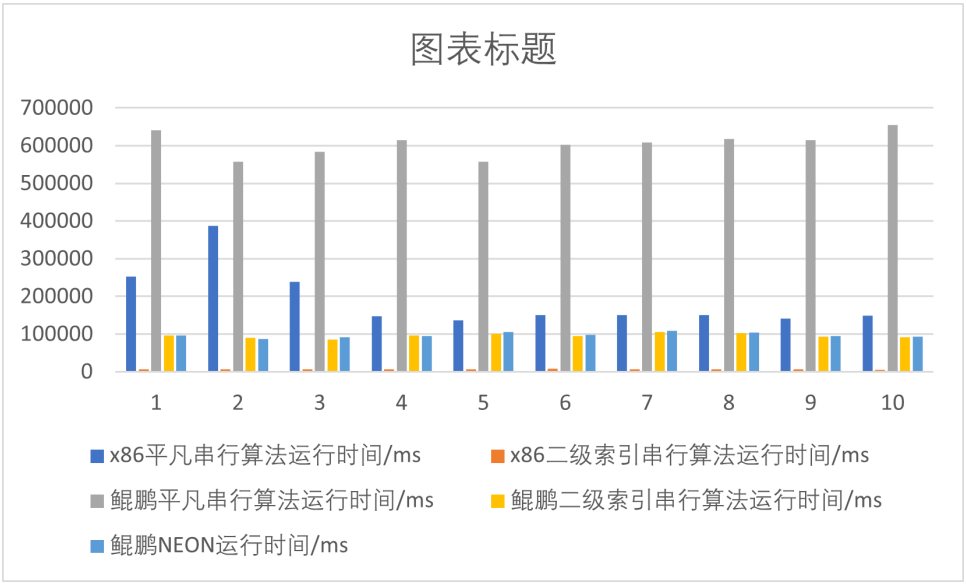


图 6.4: 各算法不同架构时间对比

如图6.5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。

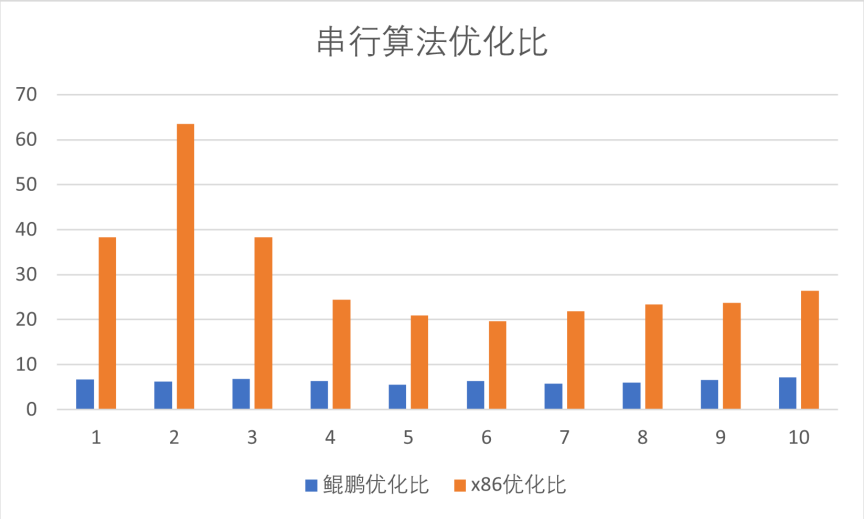


图 6.5: 串行算法优化比

7 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。