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 pthread&OpenMP 编程

姓名：刘家骥

学号：2211437

专业：计算机科学与技术

2025 年 3 月 23 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. 按表求交算法的多线程编程，负责人：王俊杰
2. 按元素求交算法的多线程编程，负责人：刘家骥

本次实验由王俊杰和刘家骥两人共同完成。

本文为刘家骥负责在 arm 和 x86 平台上的 element-wise 串行算法、NEON 算法实现，pthread 和 openmp 的 query 间和 query 内两种并行策略的实现，以及探讨不同线程数和不同问题规模下，各算法的性能比较。

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] 加入-fopenmp

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 算法实现

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。

如图3.1所示

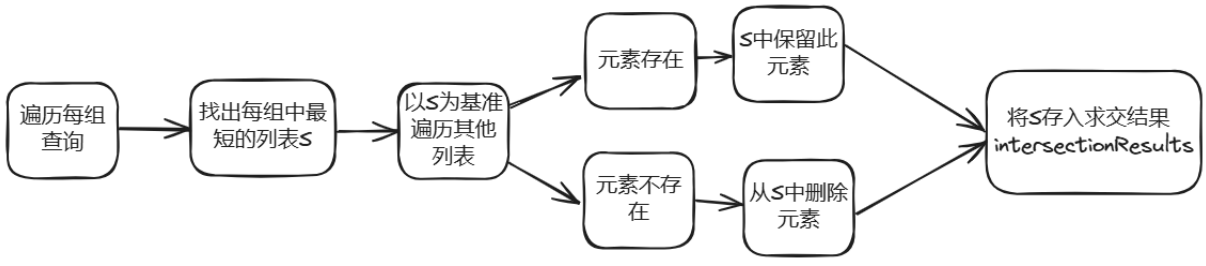


图 3.1: 倒排索引求交 element-wise 串行算法

3.3 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

4 倒排索引求交 Pthread 编程

4.1 Query 间动态并行算法

4.1.1 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。

4.1.2 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4.2所示

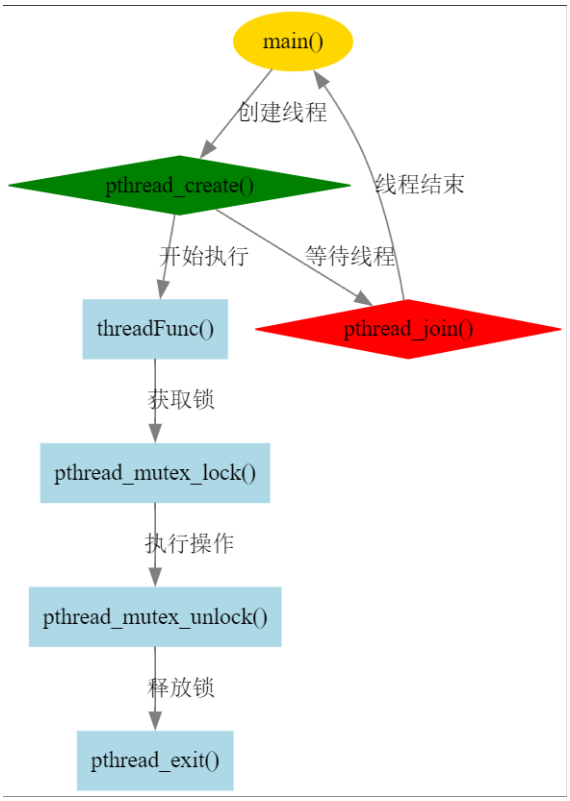


图 4.2: Query 间 Pthread 并行算法

4.1.3 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。

4.1.4 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 1: 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 2: arm

4.1.5 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。

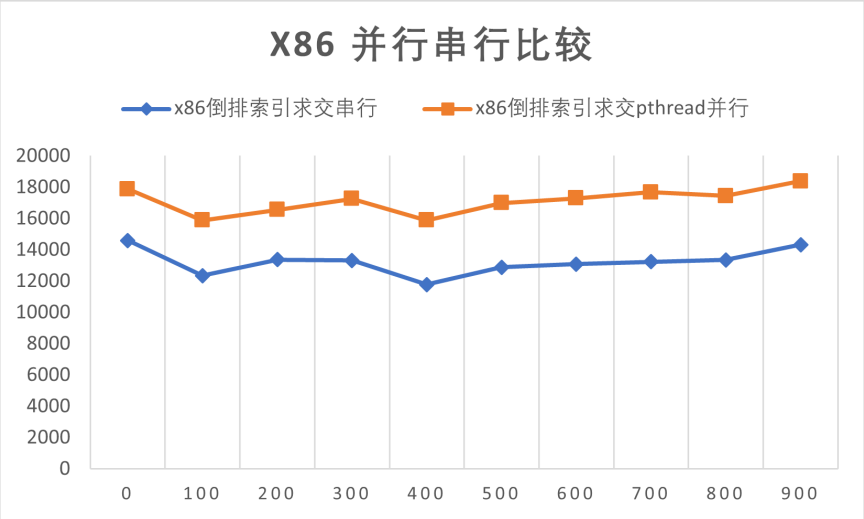


图 4.3: x86 并行串行比较

显然如图4.3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。

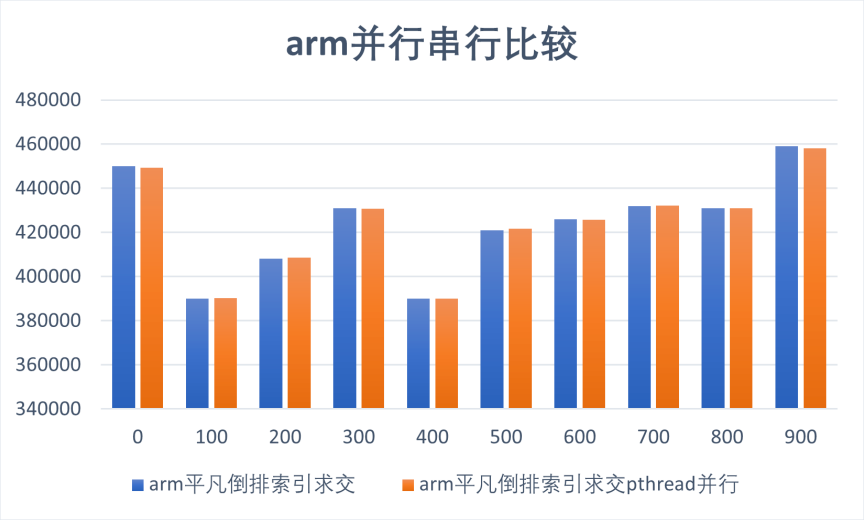
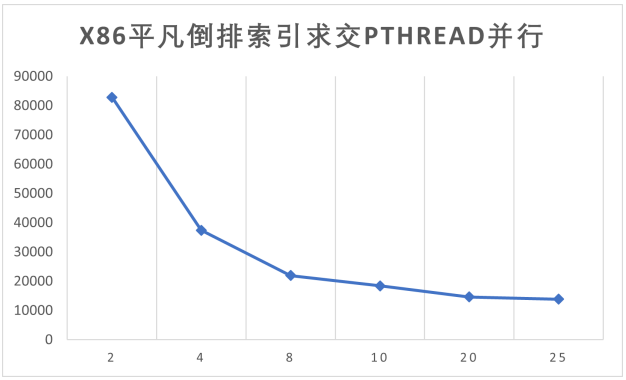


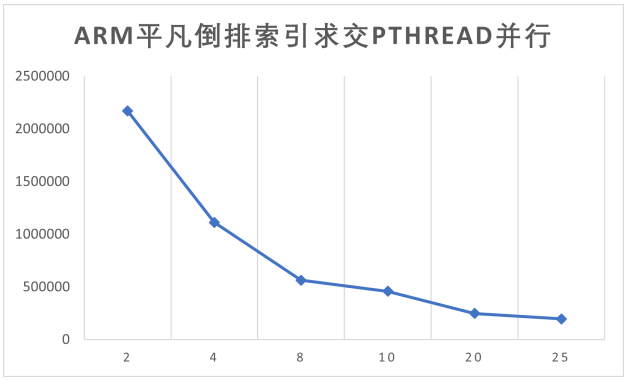
图 4.4: arm 并行串行比较

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



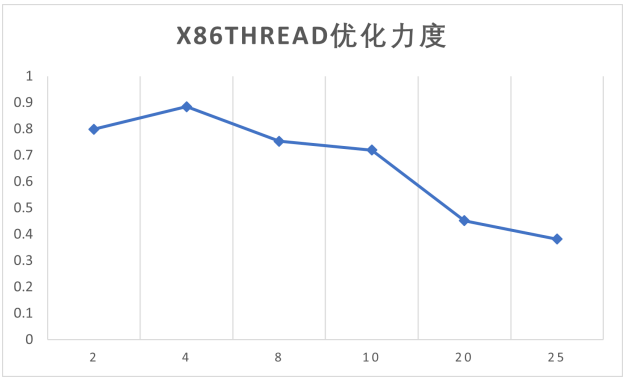
(b) arm

图 4.5: 不同平台串行优化算法的执行时间对比

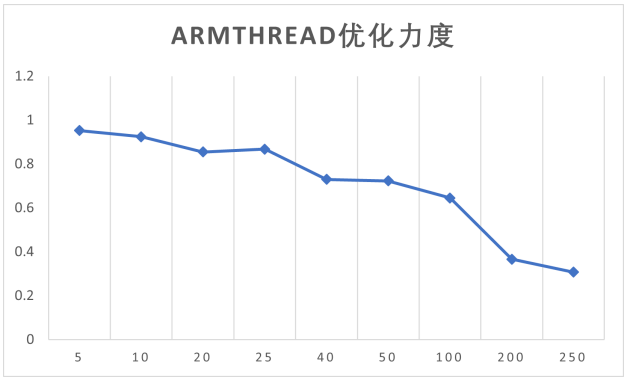
从理论上来说，线程数增加一倍，则总运行时间应该减少一倍，但是从上述不同线程的运行时间来看，显然不符合。所以为了直观看出线程数增加对程序性能优化真正的影响，引入：

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

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 4.6: 不同平台 pthread 并行线程数

显然如图6(a)和4.6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。

4.2 Query 内动态并行算法

4.2.1 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。

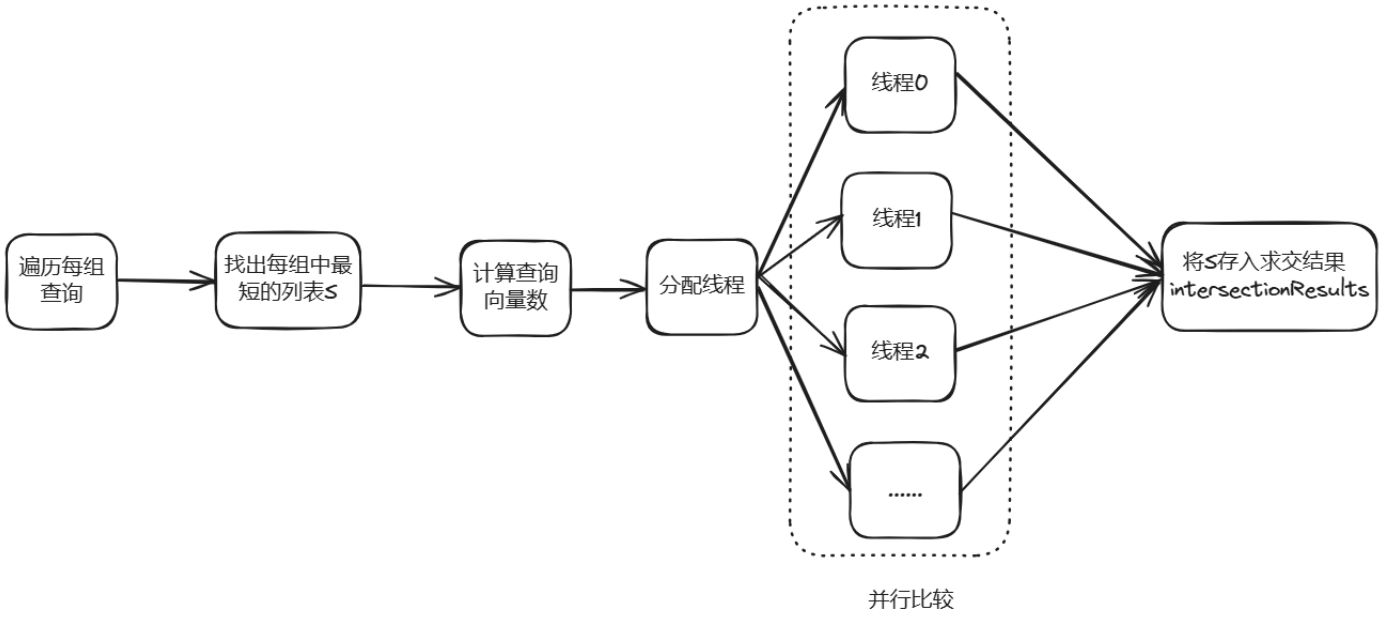


图 4.7: Query 内查询逻辑图

4.2.2 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;
}
```

4.2.3 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。

4.2.4 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 3: 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 4: 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。

4.2.5 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。

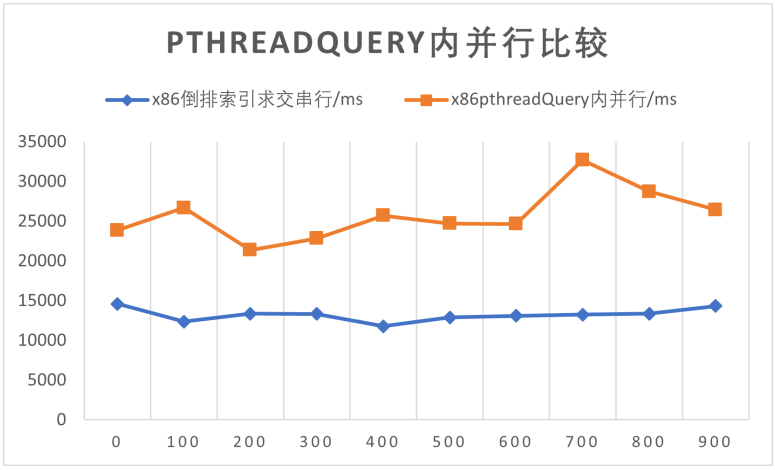


图 4.8: x86pthreadQuery 内并行比较

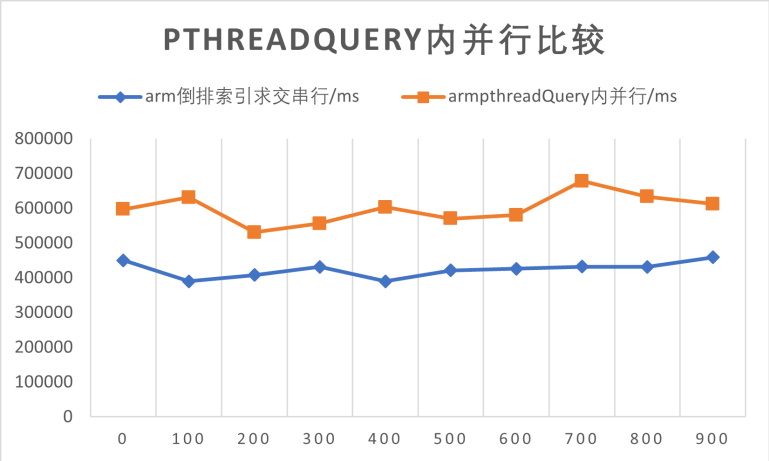


图 4.9: armpthreadQuery 内并行比较

观察图4.8和图4.9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。

5 倒排索引求交 OpenMP 编程

5.1 Query 间动态并行算法

5.1.1 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。

5.1.2 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 {
```

```
    #pragma omp parallel
    {
        // 这里的代码将由多个线程并行执行
    }
}
}
```

5.1.3 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

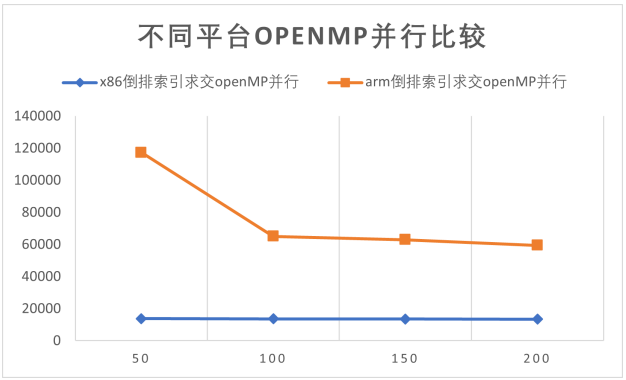
	arm 倒排索引求交	arm 倒排索引 openMP 并行	x86 倒排索引 pthread 并行
TOTAL TIME/ms	4238000	65090	458635

5.1.4 实验结果比较及分析

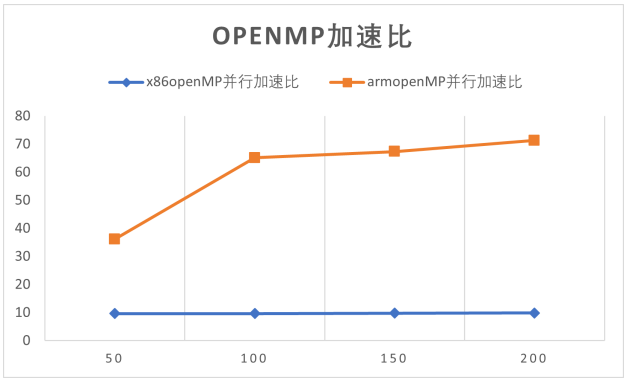
我们将线程数作为横坐标，运行时间作为纵坐标，画出如图所示的折线图。

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5.10, 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。

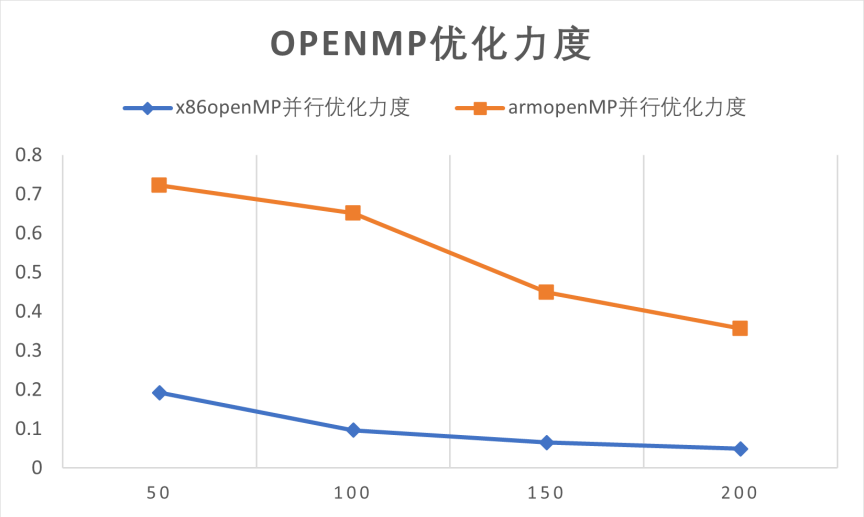


图 5.10: OpenMP 优化力度

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。

5.2 Query 内动态并行算法

5.2.1 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。

5.2.2 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 1 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
```

5.2.3 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

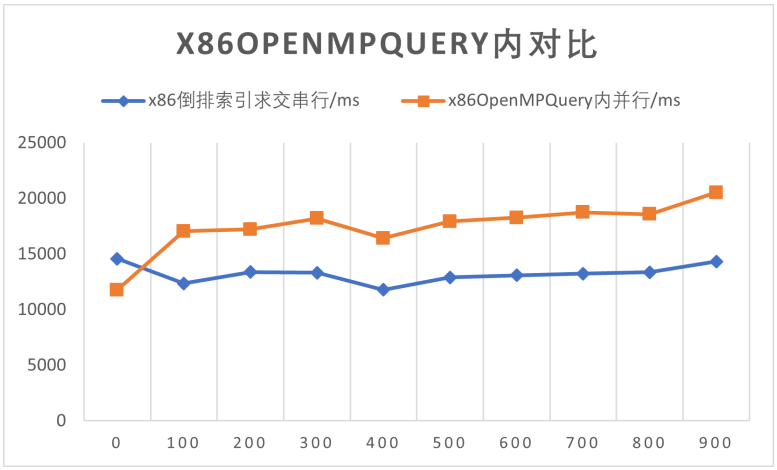
表 5: x86

	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

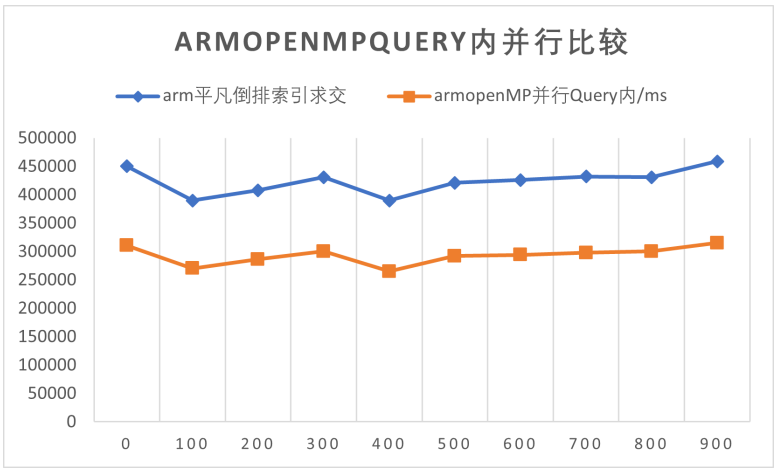
表 6: arm

5.2.4 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，



6 Pthread 和 OpenMP 并行效果对比

6.1 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。

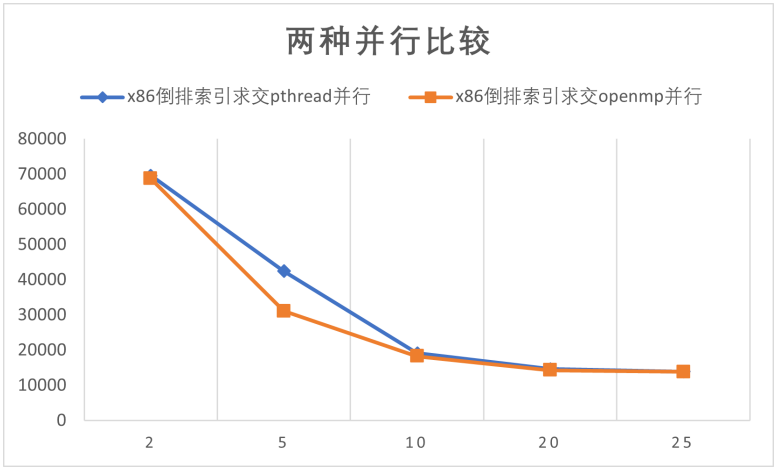
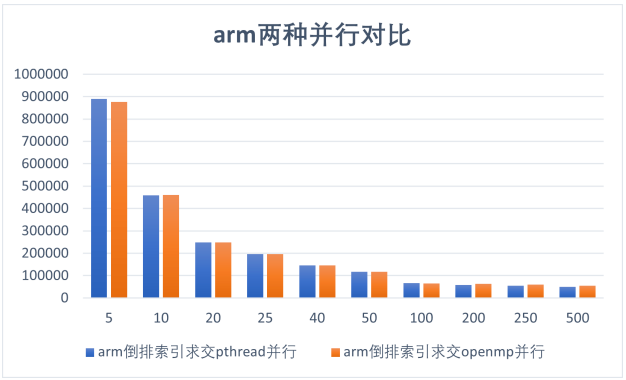


图 6.11: x86Query 间两种并行比较

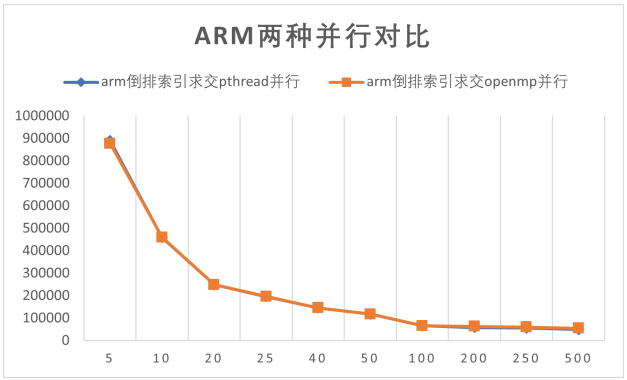
在 arm 平台上的测试数据如下

	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。

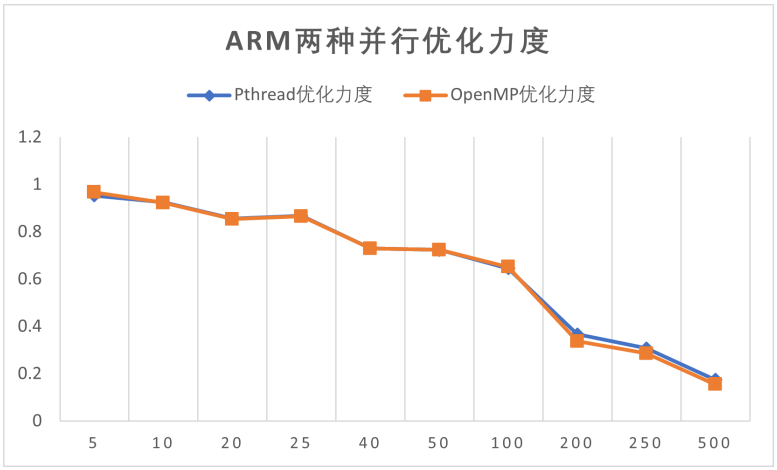


图 6.12: arm 两种并行优化力度

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。

6.2 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 7: 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

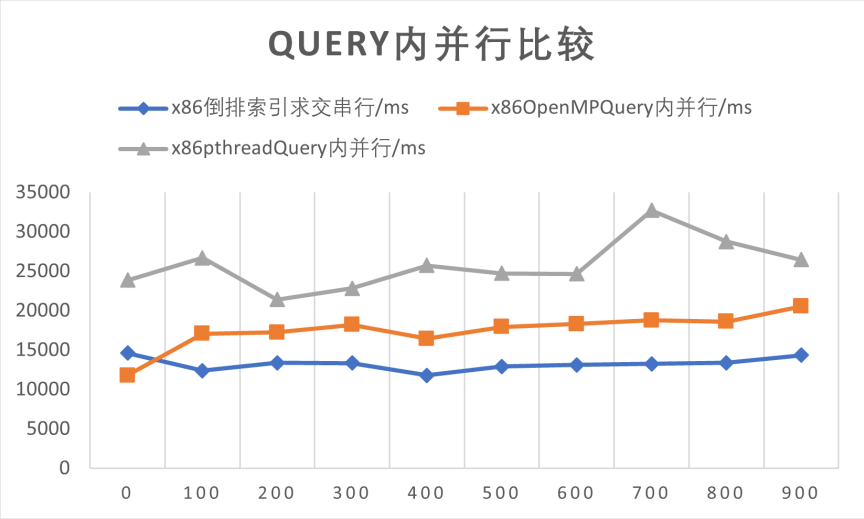


图 6.13: x86query 内并行比较

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

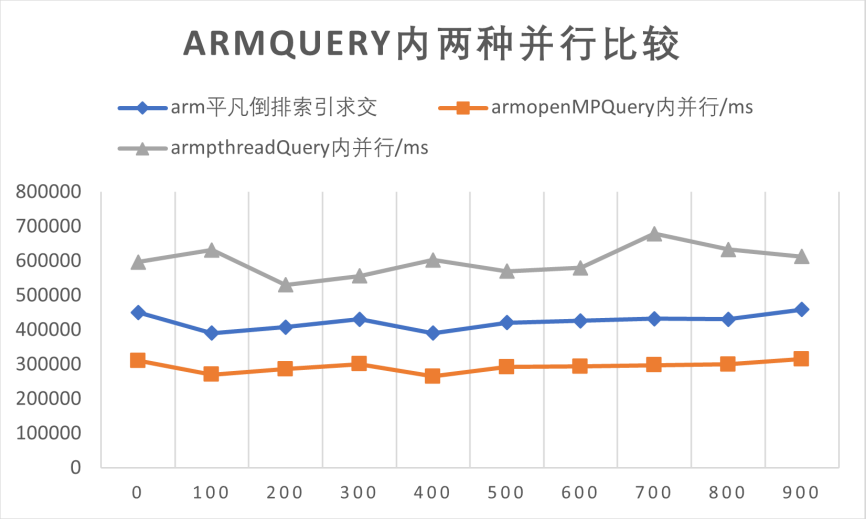


图 6.14: armquery 内两种并行比较

7 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。