南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

Pthread & OpenMP 编程

刘沿辰

年级：2020 级

专业：计算机科学与技术

指导教师：王刚

2023 年 5 月 13 日

摘要

本文对比了倒排索引求交的两种常见串行算法，并就其中的列表求交算法进行了二分查找与记录访问位置优化，达到了更好的串行性能。之后，本文分别对两种串行算法进行了 Pthread 并行优化，测试并分析了不同线程数下的算法运行表现，并结合 OpenMP 对比了各种并行策略的优劣。

关键字：倒排索引求交，串行优化，Pthread，OpenMP，负载均衡

目录

一、 概述

在 SIMD 编程作业中，我提到了倒排索引求交的两种并行算法，分别是按元素求交法和按列表求交法。两种算法各有优劣，其算法原理和适用场景已经在 SIMD 实验报告中给出，此处不做赘述。在本次实验中，我会对两种并行算法做进一步的分析与优化，得出更加优秀的串行算法。

本次实验要求使用 Pthread 和 OpenMP 对两种串行算法进行优化。Pthread 作为一个更加底层的并行方法，支持程序员编写更细化的并行调度，本次实验中，我将从程序的不同方向进行 Pthread 的任务划分并测试其性能。OpenMP 作为一个更易用也更自动的并行化工具，其基本逻辑和 Pthread 类似，都是将程序内可以并行处理的部分划分给不同的线程处理，本次实验中我也将就不同的串行算法，研究 OpenMP 带来的性能提升。

二、 串行算法性能优化

两种串行算法的实现思路在 SIMD 编程中已经给出，此处让我们来关注这些算法的优化方向。根据实验提供数据集，两算法的性能测试结果如下表所示，测试使用的请求数量为 1000 条。

	平均用时/ms	单次请求平均用时/ms	吞吐量/s
按元素求交	3012.76	3.013	331.9
按列表求交	423663.25	423.663	2.4

表 1: 串行算法测试结果

可以看到，虽然两方案的时间复杂度均为多项式级别，但是按元素求交算法的性能表现明显优于按列表求交。对于这个结果的成因，我有以下推断：

1. 经过观察，每个请求的关键词数量均为 2-5 个。按元素求交算法优势在于每次迭代后结果集合中 S 元素都大幅降低，但是由于关键词数量较少，其优势难以体现
2. 按列表求交涉及了大量的列表查找与删除操作。且大部分查询关键词数量较少，按元素求交算法可以迅速减少结果集中 S 的元素数量

两个算法复杂度都在 $O(n^2)$ 数量级，但是按列表求交算法此处优势没有得到体现。接下来，考虑借鉴按元素求交算法的思路来优化按列表求交算法。

(一) 二分查找调优

在常见的应用场景中，对数据集的排序操作是会被提前执行的一大重要操作。本次实验中的倒排列表也已经按照升序排列完毕。在按列表求交算法中，可以利用数据集各列表已经完成排序的特征来加快检索速度。

设系统的输入为：

- 数据集中的文档总数 D
- 倒排索引列表数 M
- 关键词数量 n

则算法原本的时间复杂度如下：

$$O(M + n * D^2)$$

由于数据集已经排列完毕，首先想到使用二分搜索来对数据进行查找。在二分查找中，每次查询的时间复杂度将会被优化为 $O(logD)$，因此每遍历一个列表的时间复杂度为 $O(D * logD)$，算法的总时间复杂度将被优化为：

$$O(M + n * DlogD)$$

(二) 记录访问位置调优

借鉴按元素求教算法的访问思想，按表求交算法可以继续得到优化。按表求交法在查找每个列表时都需要从头访问，带来了大量的性能浪费。由此考虑添加一个变量 *detected* 用于记录当前访问的位置，每次查询只需要从上一次查询的位置开始即可。使用该思路进行优化，遍历一个列表的时间复杂度被优化为 $O(D)$，算法的总时间复杂度优化为：

$$O(M + n * D)$$

(三) 优化实现

优化后的串行求交算法如下：

串行求交算法（优化）

```
for (int i = 2; i <= query[n][0];i++) {
    int detected = 0;
    for (int j = 0; j < S.size(); j++) {
        if (marks[j]) continue;
        bool found = false;
        for (;detected < index_lists[query[n][i]].size();detected++) {
            if (S[j] == index_lists[query[n][i]][detected]) {
                found = true;
                break;
            }
            if (S[j] < index_lists[query[n][i]][detected]) {
                break;
            }
        }
        if (!found) {
            marks[j] = true;
            continue;
        }
    }
}
```

实现改进程序后，测试串行程序性能如下表，测试用请求次数为 1000 次。

	平均总用时/ms	单次请求平均用时/ms	吞吐量/s
按列表求交（优化）	15212.70	15.212	65.73
按列表求交	423663.25	423.663	2.4

表 2: 按列表求交串行优化效果

该算法经过改进后，相较于原版效率有着二十倍以上的性能提升。从此处也能得出结论，原版的按列表求交算法时间被大量浪费在多次遍历每个列表上，并且搜索关键词多带来的性能增幅无法体现。但是该优化后的算法遇到列表元素不存在时，每次从结果集合删除操作的时间复杂度任为 $O(D)$。

三、 Pthread 并行优化

在考虑 Pthread 优化时，考虑不同请求的处理是可以分开单独执行的。因此可以每个线程完成一批请求，每个请求计算过程分别独立。该方法仅需为每个线程划分属于自己的请求即可，并仅需要划分完线程任务后进行一次同步即可。通信的额外开销小，编程也相较而言方便很多。

我才用循环来划分给每个线程安排的任务。设任务总数为 n，线程总数为 k，则第 i 号线程负责编号为 $i*n/k$ 到 $(i+1)*n/k$ 的线程，用区间表示为:

$$[i*n/k, (i+1)*n/k)$$

循环划分在一定程度上保证了请求结果的均匀访问，在写入结果时也不在使用 push_back，转而直接修改提前初始化好的向量集合下标即可。

(一) 按元素求交优化

此处用循环划分的方式来实现并行化:

按元素求交 Pthread 并行化算法

```
for (int n = t_id; n < QUERY_NUM; n += NUM_THREADS) {
    // 初始化变量
    // ......
    vector<unsigned int> S;
    sort(list_order, list_order + query[n][0], cmp);
    bool flag = false;
    while (list_order[0].first != 0 && !flag) {
        int element = 第一个未探查的元素
        for (int i = 1; i < query[n][0]; i++) {
            int now_list = list_order[i].second;
            bool found = false;
            //在其他表中未探查的元素开始, 寻找该元素是否存在
            for (; list_undetected[now_list] < list_length[now_list];
                list_undetected[now_list]++) {
                if (element == index_lists[list_to_index[now_list]][
                    list_undetected[now_list]]) {
                    found = true;
                    break;
```

```
                }
                if (element < index_lists[list_to_index[now_list]][
                    list_undetected[now_list]]) {
                    found = false;
                    break;
                }
            }
            if (found == false) break;
            if (i == query[n][0] - 1) {
                S.push_back(element);
            }
        }
        //更新未探查个数、列表长度排序
        // ......
    }
}
```

对按元素求交算法进行任务划分后，使用不同数量线程的测试结果如下表所示，测试使用请求个数为 1000 个。

线程数量	平均总用时/ms	单次请求平均用时/ms	吞吐量/s	加速比
串行	3012.76	3.013	331.9	-
2	1477.87	1.478	676.6	203.86%
4	899.46	0.900	1111.1	334.95%
8	630.33	0.630	1587.3	477.97%
10	610.70	0.611	1636.7	493.33%
12	520.67	0.521	1919.4	578.63%
16	560.23	0.560	1785.7	537.78%

表 3: Pthread 按元素求交优化效果

可以观察到，按元素求交的 Pthread 优化效果非常好，甚至在双线程的情况下达到了超过 200 的加速比（原因未知）。在实验结果中，最优的加速比出现在 12 线程时，到达了 570% 左右的加速比。但是当线程数更多（比如 16 线程）时，加速比反而出现了下降。我猜测原因有两方面：

1. 并行程序的实现依然依托于循环的固定划分，这导致部分线程处理完了而部分线程还在继续，由闲置的线程引发了并行效率较低
2. 当线程数目太多时，每个线程处理的运算变少，但是并行带来的通信代价上升，最终结果就是并行效率的下滑

当我们统计每个线程的运行时间时，可得结论如下表（此处以 8 线程为例）：

线程编号	执行时间/ms
0	494.2
1	562.3
2	501.3
3	547.5
4	538.7
5	550.0
6	493.8
7	511.9
总用时	571.3

表 4: 按列表求交 8 线程优化的各线程执行时间

由上表数据计算得线程平均运行时间为 525.0ms，但是程序得总用时却达到了 571.3ms，两者依然存在接近 10% 的性能差距。

(二) 按列表求交优化

类似地，按列表求交的优化也依托于循环划分实现：

按列表求交 Pthread 并行化算法

```
for (int n = t_id; n < QUERY_NUM; n += NUM_THREADS) {
    vector<unsigned int> S = index_lists[query[n][1]];
    for (int i = 2; i <= query[n][0]; i++) {
        int detected = 0;
        for (int j = 0; j < S.size(); j++) {
            bool found = false;
            for (; detected < index_lists[query[n][i]].size(); detected++) {
                if (S[j] == index_lists[query[n][i]][detected]) {
                    found = true;
                    break;
                }
                if (S[j] < index_lists[query[n][i]][detected]) {
                    break;
                }
            }
            if (!found) {
                S.erase(S.begin() + j);
                j--;
                continue;
            }
        }
    }
    res_lists[n] = S;
}
```

不同数量线程的测试结果如下：

线程数量	平均总用时/ms	单次请求平均用时/ms	吞吐量/s	加速比
串行	15212.70	15.212	65.73	-
2	10380.30	10.380	96.34	146.60%
4	8952.13	8.952	111.71	169.93%
6	13768.23	13.768	72.63	110.49%
8	17113.78	17.114	58.43	88.89%
16	29616.11	29.616	33.77	51.37%

表 5: Pthread 按列表求交优化效果

奇怪的就是，按列表求交时，在多线程数 6 及以下时还能取得一定的加速比（但是不成比例）；在线程数达到 8 或者以上时，多线程反而大幅度降低了运行速度。个人猜测是由于大量针对向量的删除导致每次循环都要进行一次进程间通信，通信成本大大上升，导致并行的负优化。

四、 OpenMP 并行优化

在并行优化按元素求交算法中，根据最后的统计结果可知线程平均运行时间比程序总耗时相差 10% 左右。实际上每次执行程序时，用时长的和用时短的线程无法确定，任务的执行始终存在误差，且该误差客观存在难以消除。考虑到 Pthread 程序任务划分时粒度很大，且任务早在执行前就划分完毕，尝试采用 OpenMP 来实现细粒度的运行时划分，达到更好的效果。

OpenMP 并行的操作很简单,只需要在处理请求的循环之前添加编译指令 $pragma omp parallel for ...$，然后在并行循环结束的位置添加 $pragma omp barrier$ 即可自动完成并行化。由于 OpenMP 的封装非常好，我们可以轻易地实现静态或动态分配任务，甚至可以动态地分配不同大小的任务块。

接下来我将测试不同任务划分与调度方式下 12 线程 OpenMP 带来的性能优化，具体测试指令包括:

- $pragma omp parallel for schedule(guided, 20) guided$
- $pragma omp parallel for schedule(dynamic, 10)$
- $pragma omp parallel for schedule(static, 1)$

第一条指令使用的是 guided 任务划分，分配的块的大小根据任务进程的推进而减小。该方法前期可以大块分配任务，在不空载的情况下减小并行通信代价；而到了任务末期，该方法将会分配的任务大小，以一定的并行通信代价换取负载均衡。

第二条指令使用了动态的队列式任务划分，每次划分五个任务，因此最后线程的空载时间不会超过五个任务的执行时间，限制了负载不均衡情况的出现。当然，此时程序的任务划分较为细致，程序每执行 5 个任务就会回来等待分配新的 5 个任务，可能会带来一定的并行开销。

最后一条指令就是静态的任务划分，每个线程一次划分一个任务。作为对比存在，与 pthread 的静态循环划分等价。

按列表求交算法经过 OpenMP 并行优化效果如下表所示，测试使用请求为 1000 个。

划分方式	平均总用时/ms	单次请求平均用时/ms	吞吐量/s	加速比
串行	3012.76	3.013	331.9	-
guided	481.26	0。481	2079.0	626.02%
dynamic	467.23	0.467	2141.3	644.81%
static	513.31	0.513	1949.3	586.93%

表 6: OpenMP 按元素求交优化效果

可以看到在 OpenMP 加速中，静态划分方法取得不错的效果，动态划分方法更加灵活，并且也取得了更大的加速比。值得注意的一点是，看似负载最均衡的 guided 方法效率却没能超过固定大小的 dynamic 划分。某种意义上，guided 算法牺牲了并行的信息交流代价来换取最高的负载均衡率，但是过度的牺牲系统的某种性质去追求另一种性质或许并没有办法带来好的效果。对比之下，dynamic 算法的粒度更粗一些，但是它做到了负载均衡和并行代价的平衡，因此获得了更好的效率。

五、 总结

作为最常见的并行方式，Pthread 和 OpenMP 无疑给我们程序的可并行部分带来了运行速度质的提升。在 Pthread 部分，我在划分好任务后测试了不同线程下的运行速度，确实取得了不错的结果。但可以观察到，程序的加速比在达到五百之后难以继续上升，其中原因便是负载均衡与并行代价的平衡问题。哪怕是效果不错的按元素求交 Pthread 加速，当我把线程数拉得太高时，程序依然因为并行代价产生了加速比降低的情况。而当我使用 OpenMP 时，OpenMP 特有的灵活与多线程造就了更好的并行效果。但整体来看，程序的共享内存使用部分较多，在并行过程中的各种读写都会造成并行效率的大幅降低。这告诉我们做并行程序时，线程间除开必要的信息交流之外，应该尽可能去解耦合，这样既有利于当下提升并行性能，也方便将来对程序进行修改。

当然，恰当的串行优化也是必不可少的，从列表求交的串行优化中可以看到，一个算法的串行优化也可以达到二十倍以上的优化效率，在并行之前打好良好的串行地基才能获取更高的效率。

此处附上 Github 仓库链接https://github.com/NKU-YuanRong/Parallal_Pthread_OpenMP，完整的代码存于仓库中。