南開大學
Nankai University

计算机学院

并行程序设计实验报告

Pthread & OpenMP 编程

姓名：张铭

学号：2211289

专业：计算机科学与技术

2024 年 5 月 23 日

目录

1 实验环境与流程

本次 Phread 和 OpenMP 实验涉及到了多种不同的多线程并行算法。分别根据 LIST-WISE 算法和 ELEMENT-WISE 算法的流程与步骤，分别从列表、位图和建立了二级索引的位图三种倒排索引存储方式出发，设计并实现了不同的任务分配策略和算法/编程策略，使用到了多种线程同步机制和线程管理代价优化方式，通过不同的 Pthread 编程范式和 OpenMP 范式进行编程实现。在此基础上，本次还实现了 pthread 多线程和 OpenMP 与 SIMD 相结合的并行方式。

实验中使用到了 c++ 提供的 pthread 多线程库函数以及 OpenMP 库函数，头文件分别为 pthread.h 和 omp.h。因此实验将分别在鲲鹏服务器（ARM 平台）和 IntelDevcloud 服务器（x86 平台）环境下进行实验与测试，还使用到了 Profiling 工具进行剖析。

实验结果对比与分析主要从以下几个方面展开：

github 仓库 Pthread & OpenMP 项目链接

2 选题描述和背景知识

本次 Phread 和 OpenMP 实验为自主选题，与期末研究报告结合，选择倒排索引求交进行 Phread 和 OpenMP 并行化算法求解。

2.1 倒排索引

对于一个有 U 个网页或文档（Document）的数据集，若想将其整理成一个可索引的数据集，则可以为数据集中的每篇文档选取一个文档编号（DocID），使其范围在 [1,U] 中。其中的每一篇文档，都可以看做是一组词（Term）的序列。则对于文档中出现的任意一个词，都会有一个对应的文档序列集合，该集合通常按文档编号升序排列为一个升序列表，即称为倒排列表（Posting List）。所有词项的倒排列表组合起来就构成了整个数据集的倒排索引。

2.2 倒排列表求交

在获得倒排索引列表的基础上，我们考虑倒排列表求交（List Intersection）。它指的是将多个倒排列表（即文档或记录中某个特定项的出现位置列表，比如某个关键词在不同文档中的出现位置列表）进行交集操作，以找到它们共同出现的位置或文档。

当用户提交了一个 k 个词的查询，查询词分别是 $t_1, t_2, ..., t_k$，表求交算法返回 $\cap_{1\leq i\leq k} l(t_i)$。

例如查询 “2014 NBA Final”，搜索引擎首先在索引中找到 “2014”，“NBA”，“Final” 对应的倒排列表，并按照列表长度进行排序：

$$l(2014) = (13, 16, 17, 40, 50) \tag{1}$$

$$l(NBA) = (4, 8, 11, 13, 14, 16, 17, 39, 40, 42, 50) \tag{2}$$

$$l(Final) = (1, 2, 3, 5, 9, 10, 13, 16, 18, 20, 40, 50) \tag{3}$$

求交操作返回三个倒排列表的公共元素，即：

$$l(2014) \cap l(NBA) \cap l(Final) = (13, 16, 40, 50) \tag{4}$$

2.3 位图

位图是一种数据结构，用于表示一组二进制位的集合。在位图中，每个位都只能是 0 或 1，分别表示集合中的元素是否存在或者是否被标记。使用位图来存储倒排列表时，位图的长度通常等于文档总数，每个位表示一个 DocID 是否包含在该 term 对应的倒排列表中。假定文档总数 U=10，一次搜索中的词组为 “the boy first”。我们通过以下示例来展现位运算操作实现倒排列表求交的过程。

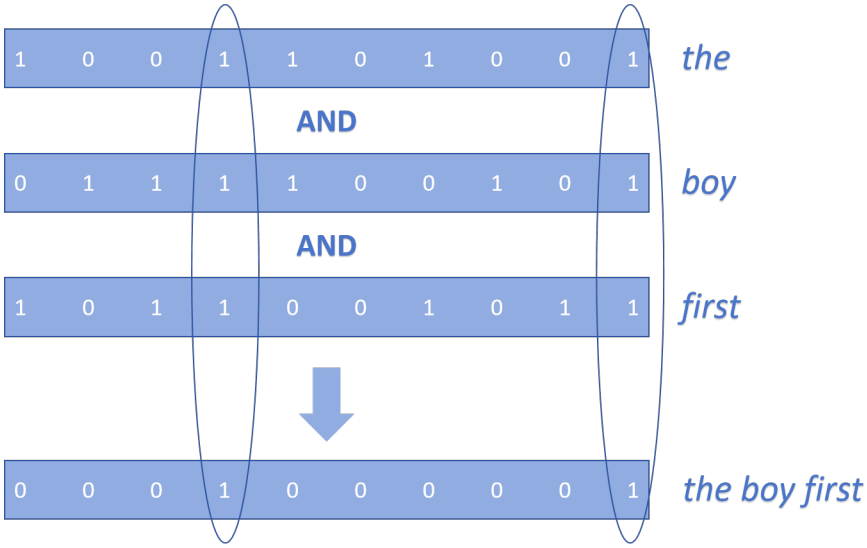


图 2.1: 位图存储的倒排列表通过 AND 位运算实现求交

3 LIST-WISE

3.1 任务划分

首先我们来回顾一下 list-wise 算法的整体流程，伪代码如下所示。对于该算法，存在内外两层 for 循环，外层循环由于每轮循环完成后都要进行 Delete 消除元素，因此存在数据依赖，不适合对此进行任务划分。由此我们将目光转移至内层循环。内存循环对 S 的每个元素 e 进行扫描，并在 l_i 中查找，因此内层循环只对 S 和 l_i 进行了读操作，且循环间不存在数据依赖性，并不存在冲突，因此我们可以对 S 进行元素划分，分配给不同线程，每个线程承担部分元素的查找任务。

Algorithm 1 list-wise 算法

Input: l(t1),l(t2),...,l(tk), 按升序排序，|l(t1)| |l(t2)| ... |l(tk)|

Output: $\cap_{1\leq i\leq k} l\,(t_i)$

```
for i=2 to k do
    for each element e ∈ S do
        found = find(e, (l_i))
        if found = FALSE then
            Delete e from S
        end if
    end for
end for
return S
```

虽然查找任务可以直接进行任务划分，但后续对元素的删除仍旧存在写操作，一种朴素的想法是使用线程锁限制只有单个线程可以进行删除操作，但这样会增大开销和线程等待时间，使得程序性能降低，且删除元素后 S 的长度会发生改变，每个线程承担元素的下标索引可能与删除前不一致，导致结果错误。

针对于上述问题，我们可以对 list-wise 算法进行修改，如下所示。每次查找结果为 false 时，将该元素暂时置为-1（倒排列表中不出现的元素值），标记其为待删除元素。当所有元素的查找任务全部完成时，将值为-1 的元素删除，这一部分删除工作我们可以分配给主线程来完成，也可以单独使用同步机制分配给 1 个线程（如 0 号线程）完成。由此我们避免了线程冲突等一系列问题。

Algorithm 2 list-wise 算法（改进）

Input: l(t1),l(t2),...,l(tk), 按升序排序，|l(t1)| |l(t2)| ... |l(tk)|

Output: $\cap_{1\leq i\leq k} l\left(t_i\right)$

```
for i=2 to k do
    for each element e ∈ S do
        found = find(e, (l_i))
        if found = FALSE then
            e = -1
        end if
    end for
    for each element e ∈ S do
        if e == -1 then
            Delete e from S
        end if
    end for
end for
return S
```

3.2 负载均衡

针对于对 S 的元素划分问题，我们可以考虑两种划分方式：

第一种方式考虑**平均划分**，将 S 列表的全部元素按照线程数量平均分配，例如 S 中存在 1000 个元素，线程数为 4，则我们将第 1-250 个元素的查找任务分配给 0 号线程，第 251-500 个元素分配给 1 号线程，以此类推。这样划分的思路虽然简单，但可能会存在负载不均的问题，如果某一个线程分配到的元素的查找结果存在大量 false，则该线程需要频繁的对 S 进行写操作，由此造成该线程的执行时间慢于其他线程，导致性能达不到理想预期。

针对上述负载不均问题，我们考虑**动态任务划分**，在 S 上维护一个索引指针。每个线程根据该索引动态的获取查找任务。此外考虑到动态划分会频繁地对 S 进行加锁和解锁，我们可以使用粗粒度进行划分，减少锁次数，进一步提高性能。

3.3 Pthread 编程范式

Pthread 编程存在两种范式，分别为动态线程和静态线程。针对与 list-wise 算法，我们考虑分别使用这两种范式进行 Pthread 编程。

对于动态线程而言，我们可以在内层循环创建线程，线程池中的所有线程只完成 l_0 和 l_i 的求交任务，完成后该线程立即被销毁。下一次 l_0 和 l_{i+1} 求交时重复创建线程——求交——销毁线程的步骤。使用静态线程范式的好处是在没有并行计算需求时（外层循环）不会占用系统资源，缺点是是有较大的线程创建和销毁开销。

对于静态线程而言，我们可以在程序开始时就创建好线程，每个线程读取索引列表的所有行。每个线程完成 l_0 和 l_i 的求交任务后，使用 pthread 同步机制等待其他线程完成对应行的求交任务，最终再由主线程或特定线程完成元素消除任务。使用静态线程范式的好处是没有频繁的线程创建、销毁开销，性能更优；缺点是线程一直保持，占用系统资源，可能造成资源浪费。

3.4 Pthread 编程实现

3.4.1 动态线程

在进行求交任务之前，我们需要定义线程数据结构，将动态线程待查找的列表（即 l_i）读取到数据结构中。

```
typedef struct threadParam_Dynamic{
    int t_id; //线程id
    const vector<uint32_t> *otherList; //待查找的列表
} threadParam_Dynamic;
```

接下来我们需要定义线程函数，这里我们主要对比展示使用平均划分方式和动态划分方式的代码实现:

```
void* threadFunc(void* param)
{
    //平均划分方式
    int n =Lists[0].size()/thread_count; //每个线程承担的任务规模
    int start = t_id * n;
    list-wise(otherlist, start, start + n); //索引求交

    //动态划分方式
    int task = 0; //待执行的任务
    while(true){
        pthread_mutex_lock(&mutex_task); //加锁
        task = next_arr;
        next_arr += size;
        pthread_mutex_unlock(&mutex_task); //解锁
        if(task + size >= Lists[0].size())
            break;
        list-wise(otherlist, task, task+size); //索引求交
```

后续主线程部分我们需要定义线程池和数据结构的数组，并在内层循环中启动线程，并在每次内层循环的结尾进行元素消除。代码具体实现不在报告中进行展示。

3.4.2 静态线程 + 信号量同步

进行静态线程范式编程时，使用信号量同步机制，通过信号量控制工作线程等待和主线程消除元素，具体的工作流程为：主线程首先唤醒工作线程完成并行求交任务，同时主线程进入睡眠，等待任务完成。完成求交任务后工作线程阻塞，主线程被唤醒，进行元素消除，消除完成后再次唤醒工作线程完成下一轮任务，循环往复直至结束。

首先我们需要定义信号量和数据结构：

```
sem_t sem_main;
sem_t* sem_workerstart = (sem_t*)malloc(thread_count*sizeof(sem_t));
sem_t* sem_workerend = (sem_t*)malloc(thread_count*sizeof(sem_t));
typedef struct threadParam_Static{
    int t_id;
} threadParam_Static;
```

接下来定义线程函数，我们主要配合代码中的注释展示有关信号量的代码部分，任务划分和按表求交的部分略过：

```
int next_arr = 0;
int size; //划分粒度
pthread_mutex_t mutex_task;
void* threadFunc(void* param) {
    for (int i = 1; i < Lists.size(); i++) {
        sem_wait(&sem_workerstart[t_id]);
        //...
        //任务划分与按表求交
        //...
        sem_post(&sem_workerend[t_id]); //唤醒主线程
        sem_wait(&sem_workerstart[t_id]); //阻塞，等待主线程唤醒进入下一轮
    }
    pthread_exit(NULL);
}
```

主线程信号量部分：

```
//初始化信号量
sem_init(&sem_main, 0, 0);
for (int t_id = 0; t_id < thread_count; t_id++) {
    sem_init(&sem_workerstart[t_id], 0, 0);
    sem_init(&sem_workerend[t_id], 0, 0);
}
//创建线程
pthread_t handles[thread_count];//创建对应的Handle
threadParam_ord_Static param[thread_count];//创建对应的线程数据结构
for (int t_id = 0; t_id < thread_count; t_id++) {
    param[t_id].t_id = t_id;
    pthread_create(&handles[t_id], NULL, threadFunc, &param[t_id]);
}
for (int i = 1; i < Lists.size(); i++) {
```

```
    //开始唤醒工作线程
    for (int t_id = 0; t_id < thread_count; t_id++)
        sem_post(&sem_workerstart[t_id]);
    //主线程睡眠（等待所有的工作线程完成此轮求交任务）
    for (int t_id = 0; t_id < thread_count; t_id++)
        sem_wait(&sem_workerend[t_id]);
    //主线程消除元素
    Delete_Element(Lists[0]);
    //主线程再次唤醒工作线程进入下一轮次的消去任务
    for(int t_id = 0;t_id < thread_count;t_id++)
        sem_post(&sem_workerstart[t_id]);
}
for(int t_id = 0;t_id < thread_count;t_id++)
    pthread_join(handles[t_id], NULL);
```

3.4.3 静态线程 + barrier 同步

除了使用信号量进行线程同步，我们还可以使用 barrier 同步。具体流程为：所有线程列表求交—使用 barrier 同步—0 号线程完成元素消除工作—barrier—下一轮列表求交。

定义线程函数，同样只展示使用到的 barrier 同步部分，其余部分省略

```
void* threadFunc(void* param) {
    threadParam_Static* p = static_cast<threadParam_Static*>(param);
    int t_id = p->t_id;
    for (int i = 1; i < Lists.size(); i++) {
        //...
        //任务划分与按表求交
        //...
        pthread_barrier_wait(&barrier_Intersection);
        //0号线程执行元素消除任务
        if(t_id == 0) Delete_Element(Lists[0]);
        pthread_barrier_wait(&barrier_Elimination);
    }
    pthread_exit(NULL);
}
```

主线程部分涉及到了 barrier 的初始化，创建线程池、数据结构数组、线程等待以及 barrier 销毁，代码编写较为简单，因此不在报告中进行展示。

3.5 OpenMP 编程范式

本次实验我们将使用两种 OpenMP 范式。第一种范式是在外循环之外创建线程，类似 Pthread 编程中的静态线程，避免线程反复创建销毁，需要注意的是每次循环后需要进行元素消除，因此还需要使用到 OpenMP 的同步机制，本次实验使用了 single 和 barrier + master 两种同步方式，single 将元素消除工作分配给一个线程完成，而 master 分配给主线程完成。

值得一提的是，single 代码块只会有一个线程执行，并且在 single 代码块最后会有一个同步点，只有 single 代码块执行完成之后，所有的线程才会继续往后执行。master 代码块最后并没有一个同步点，

而 single 会有一个隐藏的同步点，只有所有的线程到同步点之后线程才会继续往后执行，因此在使用 master 代码块进行元素消除时，需要配合 barrier 来完成任务。

第二种范式是在内循环创建线程，减少线程间通信开销，类似 Pthread 编程中的动态线程。同时考虑 static 和 dynamic 两种 schedule 方式进行任务划分，比对其性能差异。

3.6 OpenMP 编程实现

3.6.1 循环内创建线程

我们在内层循环中使用 omp parallel for 指令创建线程，类似于 Pthread 中的动态线程范式，具体的代码实现如下：

```
for(int i = 1; i < Lists.size(); i++){
    #pragma omp parallel for num_threads(thread_count) //schedule(dynamic, dsize)
    for(int j=0;j<Lists[0].size();j++){
        int element=Lists[0][j];
        bool found=std::binary_search(Lists[i].begin(), Lists[i].end(), element);
        if(!found) Lists[0][j] = -1;
        }
    Delete_Element(Lists[0]);     //主线程消除元素
}
```

3.6.2 循环外创建线程

我们在外层循环中创建线程，并在内层循环使用 omp for 进行并行化，这种方式类似于 Pthread 中的静态线程范式。为了节省篇幅，本小节将 master + barrier 同步方式和 single 同步方式放在一起进行展示：

```
#pragma omp parallel for num_threads(thread_count) //schedule(dynamic, dsize)
for(int i = 1; i < Lists.size(); i++){
    #pragma omp for
    for(int j=0;j<Lists[0].size();j++){
        int element=Lists[0][j];
        bool found=std::binary_search(Lists[i].begin(), Lists[i].end(), element);
        if(!found) Lists[0][j] = -1;
    }
    //single同步方式
    #pragma omp single          //单个线程执行
    Delete_Element(Lists[0]);   //单个线程消除元素
    //master + barrier同步方式
    #pragma omp master          //主线程执行
    Delete_Element(Lists[0]);   //主线程消除元素
    #pragma omp barrier         //barrier同步
}
```

4 LIST-WISE-BITMAP

4.1 任务划分与负载均衡

对于位图存储形式，求交运算就变为两个位向量的位与运算。因此我们也可以按照列表存储的任务划分方式，对每条位向量进行任务划分。。所有向量中所有元素都要进行运算，因此采用平均分配方式不存在负载不均问题，无需使用动态分配方式。

4.2 Pthread 编程范式

同样地，使用动态线程和静态线程两种方式进行 Pthread 编程实现。位图存储不存在元素删除操作，因此不需要使用到线程同步机制。编程思路与列表存储的实现方式类似，不再进行赘述。

4.3 Pthread 编程实现

4.3.1 动态线程

同样地，对于 Pthread 动态线程而言，我们先要定义相应的数据结构，将 t_id 和要求交的 bitmap 对应的行列表传入线程函数中。线程函数中涉及到平均任务划分和按位与求交两步操作，主线程中涉及到线程池和数据结构数组的定义，线程的创建和启动、线程等待等步骤，具体流程与 List 存储部分较为相似，具体代码不在报告中进行展示。

4.3.2 静态线程

对于静态线程而言，编程实现过程也与 List 静态线程的流程较为类似。不同的是使用 List 存储时，每次求交完成后需要进行元素消除，从而减少下一次求交时的冗余工作，但对于 Bitmap 存储而言，每个线程完成两行对应部分的运算后不需要与其他线程同步，直接与下一行进行运算，这使得 Bitmap 的性能要远优于 List 存储方式。Bitmap 静态线程的实现思路也较为简单，不在此统一展示。

4.4 OpenMP 编程实现

类似地，我们同样考虑内、外循环创建线程这两种 OpenMP 范式实现 Bitmap 求交，实现思路较为简单，不再进行赘述。

4.4.1 循环内创建线程

```
for(int i = 1; i < bitmap.size(); i++) {
    #pragma omp parallel for num_threads(thread_count) //schedule(dynamic, dsize)
    for(int j=0;j<bitmap[0].size();j++)
    bitmap[0][j]&=bitmap[i][j];
}
```

4.4.2 循环外创建线程

```
#pragma omp parallel num_threads(thread_count)
for(int i = 1; i < bitmap.size(); i++) {
```

```
    #pragma omp for //schedule(dynamic, dsize)
    for(int j=0;j<bitmap[0].size();j++)
    bitmap[0][j]&=bitmap[i][j];
}
```

4.4.3 OpenMP + SIMD

使用 OpenMP 与 SIMD 相结合的算法向量化思路与 SIMD 编程的思路类似，都是将位图元素按行进行向量化，再将两列向量进行按位与运算。因此我们只需要将 #pragma omp for 指令替换为 #pragma omp simd 即可，由于和上一节的代码基本类似，本节不再进行展示，后续将在实验结果分析部分对使用 SIMD 和不使用 SIMD 的 OpenMP 算法进行性能对比分析。

4.5 OpenMP 编程实现

OpenMP 实现思路为：将二级索引的逻辑与分配给多个线程分别执行，如果结果为 1，则完成对应的位图块的按位与求交，使用内、外循环创建线程两种范式进行编程实现。

4.5.1 循环内创建线程

```
int size = sqrt(bitmap[0].size());
for(int i=1;i<secondIndex.size();i++){
    #pragma omp parallel for num_threads(thread_count) //schedule(dynamic, dsize)
    for(int j=0;j<secondIndex[0].size();j++){
        if(secondIndex[0][j] && secondIndex[i][j])//该区块有相同元素
        for(int k = 0; k<size; k++)
        bitmap[0][j*size+k] &= bitmap[i][j*size+k];
    }
}
```

4.5.2 循环外创建线程

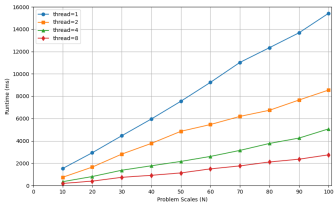
```
int size = sqrt(bitmap[0].size());
#pragma omp parallel num_threads(thread_count) //schedule(dynamic, dsize)
for(int i=1;i<secondIndex.size();i++){
    #pragma omp for
    for(int j=0;j<secondIndex[0].size();j++){
        if(secondIndex[0][j] && secondIndex[i][j])//该区块有相同元素
        for(int k = 0; k<size; k++)
        bitmap[0][j*size+k] &= bitmap[i][j*size+k];
    }
}
```

5 LIST-WISE Pthread 实验结果分析

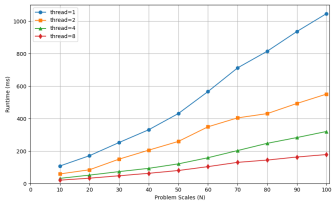
5.1 线程数对性能的影响

对于 List 存储方式，我们选取列表存储方式中的动态线程分析线程数对性能的影响，其中将线程数为 1 作为串行时间。根据实验结果得到的对比曲线如图2(a)所示。可以清晰地发现，当问题规模较小（N = 10，20）时，线程的创建和销毁次数和线程之间的同步次数较少，因此线程数的成倍提升对程序性能同样有着成倍的增长，二者关系基本呈正比例趋势；而随着查询数量的不断增加，线程的创建与销毁越发频繁，导致线程数对程序性能的提升幅度有所下降。

同样地，对于 Bitmap 存储方式，我们选取最优的静态线程范式版本进行分析，根据得到的实验结果绘制对比曲线如图2(b)所示。与 List 存储相比，使用 Bitmap 存储时虽然各线程不需要进行通信和同步，只需要对负责的数据完成按位与运算即可，但随着查询数量的增加，仍然存在频繁的线程创建和销毁。



(a) List



(b) Bitmap

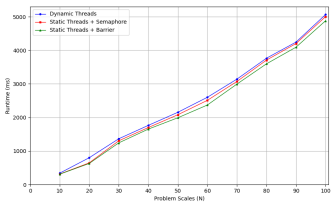
图 5.2: Pthread 线程数对性能的影响

上述实验结果说明在使用 Pthread 对问题规模较大的问题进行多线程并行求解时，除了要考虑任务划分和算法实现等问题，还要特别注意对线程的管理和线程间的通信这部分代价进行优化，以达到更好的程序性能。

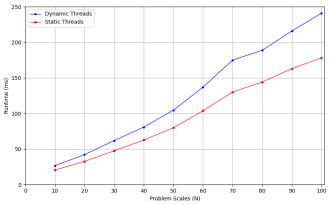
5.2 动、静态线程的性能差异

本节将分别对 List 存储和 Bitmap 存储进行动态线程与静态线程的对比分析。首先分析 List 存储方式。我们分别测量线程数为 4 时使用动态线程和静态线程不同查询数量下的运行时间，根据数据我们绘制出如图3(a)的折线图。图中可以看出使用静态线程的性能要稍优于动态线程。

接下来我们分析 Bitmap 存储方式。同样地，测量线程数为 4 时动态线程和静态线程不同查询数量下的运行时间，得到数据如图3(b)所示。我们发现使用 Bitmap 时静态线程比动态线程的性能更加优异，且随着问题规模的增长，静态线程的性能优势越来越显著。推测其原因为随着问题规模增长，线程的创建销毁次数不断增加，使用静态线程避免了频繁的线程创建和销毁，减少了部分开销。



(a) List



(b) Bitmap

图 5.3: Pthread 动态线程与静态线程间性能对比

5.3 不同同步机制的性能差异

本节我们对使用静态线程时用到的信号量同步机制和 barrier 同步机制进行性能对比。如图3(a)所示（上文），我们可以看到在使用静态线程范式时，相较于 sem 同步，使用 barrier 同步机制表现出的性能更加优异。分析其原因可能为：barrier 同步机制对线程进行统一管理，而信号量机制涉及到对单个线程进行启动和阻塞，从而造成了部分性能差异。

5.4 平均划分和动态划分的性能差异

本节我们 List 动态线程算法对平均划分和动态划分两种任务分配方式进行性能对比分析，颗粒度选择 $\sqrt{L/t}$（L: 每次求交要查询的元素总数，t: 线程数）。得到的数据如图5.4所示。我们发现在问题规模较小时，两种分配方式的性能基本一致，动态分配的优势不能够直观的体现。随着问题规模不断增加，动态分配的性能优势越发明显，能够更好的解决负载不均衡问题。

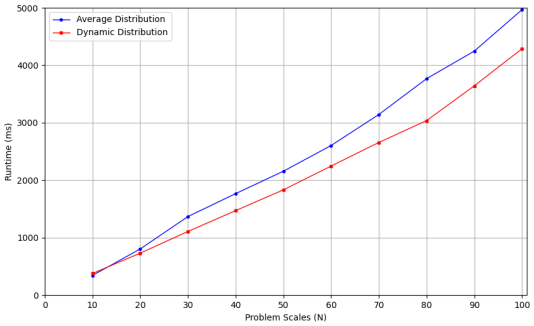


图 5.4: Pthread 平均划分与动态划分的性能差异

5.5 ARM 与 x86 平台性能对比

本节我们基于 Bitmap 静态线程算法对 ARM 和 x86 平台进行 Pthread 性能对比。线程数设定为 4，得到数据如图5.5所示。可以看到随着问题规模的增长，ARM 平台的时间增长较为平稳，整体呈现出一次函数增长趋势。而 x86 平台呈现出的时间增长趋势更加陡峭。我们初步推断与 ARM 和 x86 平台服务器的 cache 缓存大小相关。由于 x86 平台的 cache 较小，因此在问题规模达到较大规模时，cache 任务负载较重，处理的数据量极为庞大，从而造成运行时间大幅度增长；而 ARM 平台的 cache 相对较大，能够容纳更多数据进行运算，因此时间曲线比较平稳。

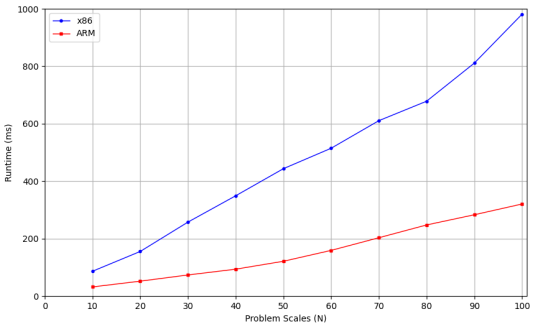


图 5.5: Pthread ARM 与 x86 性能对比

6 LIST-WISE OpenMP 结果分析

6.1 线程数对性能的影响

本节主要研究使用 Bitmap 存储时线程数量对程序性能的影响。我们选择循环内创建线程的 OpenMP 算法进行分析，根据得到的实验结果绘制对比曲线如图6.6所示。相较于 Pthread，使用 OpenMP 时增加线程数的性能提升幅度较小，并未达到理想预期。

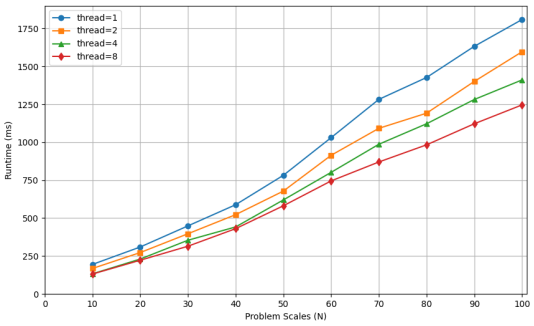


图 6.6: OpenMP-Bitmap 线程数对性能的影响

6.2 动态分配颗粒度对性能的影响

本节我们主要基于 List 存储方式分析 OpenMP 使用动态划分方式时采取不同颗粒度的程序性能进行对比分析。对于动态划分方式而言，颗粒度的选择对程序性能同样有着比较大的影响，如果设置较小颗粒度，可能会导致线程频繁阻塞和等待，若设置较大颗粒度，则可能存在局部负载不均等问题，这些都可能造成动态划分方式的性能较差，达不到理想预期。

因此我们对划分颗粒度设置不同梯度并进行实验，得到的数据如图6.7所示。我们可以看到，随着颗粒度的增加，程序运行时间呈先下降后上升的趋势，且颗粒度为 1 时的性能要优于颗粒度为 $\frac{L}{t}$，即平均划分时的性能。因此在设定颗粒度时，我们需要额外考虑阻塞和等待的开销，找到性能较好的颗粒度进行动态划分。

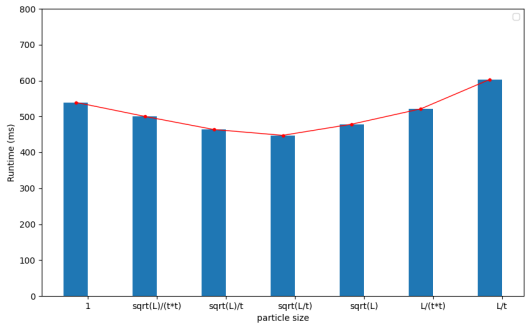


图 6.7: OpenMP 不同颗粒度下的程序性能

6.3 SIMD 对 OpenMP 性能的提升

本节主要研究 SIMD 对 OpenMP 性能的提升。我们对使用 SIMD 和不使用 SIMD 的 OpenMP 算法进行测试，线程数统一设定为 4，得到的数据如图6.8所示。从图中可以看出，SIMD 对程序的性能有着不错的优化，且相较于传统的 OpenMP，使用了 OpenMP+SIMD 的程序性能更加平稳，随着问题规模的增长，其时间性能保持在一个较为稳定的增长幅度上。

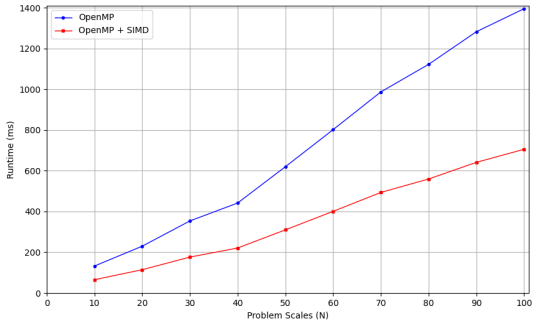


图 6.8: SIMD 对 OpenMP 性能的提升

6.4 ARM 与 x86 平台性能对比

本节主要基于 Bitmap 存储方式研究 OpenMP 在 ARM 和 x86 平台上的性能对比。我们将线程数设定为 4，分别在两台服务器上测量得到数据如图6.9所示。分析折线图可以看到，与 Pthread 类似，ARM 平台的性能要优于 x86。此外，OpenMP 在 ARM 平台上的性能依旧十分稳定，而在 x86 平台上，依旧是受到 cache 缓存的影响，程序运行时间在问题规模处于 30 80 之间时有较大波动，存在着不稳定性。

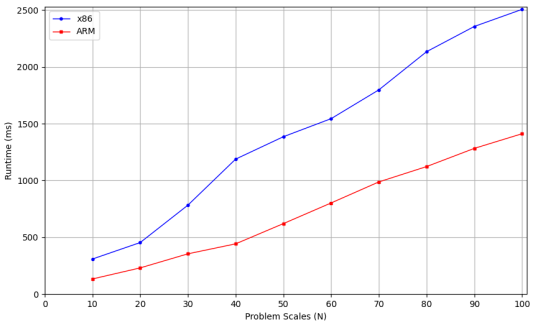


图 6.9: OpenMP ARM 与 x86 平台性能对比

7 LIST-WISE Pthread 与 OpenMP 性能对比

本节主要基于表现较优的 Bitmap 存储方式对 Pthread 和 OpenMP 进行性能对比分析。我们选择性能表现更加优异的静态线程方式，分别对 Pthread 和 OpenMP 进行测量，得到的数据如图7.10所示。我们可以看到 Pthread 的性能整体优于 OpenMP，推测原因可能为 Pthread 中我们人为地对任务进行了平均划分，且由于 Bitmap 算法的特殊性，我们并未使用到多线程同步，而 OpenMP 使用工作分割模型进行自动任务划分，具体的划分方式我们不可得知。上述这些因素造成 Pthread 和 OpenMP 在性能上有着比较大的差异。

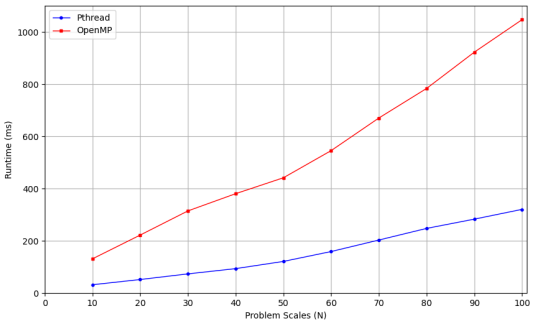


图 7.10: Pthread 与 OpenMP 性能对比

8 LIST-WISE Profiling

8.1 Pthread

8.1.1 动态线程与静态线程

本节主要基于 Bitmap 存储方式对动静态线程方式进行程序性能剖析。我们使用 perf 分别测量不同查询数量下动态线程与静态线程的 IPC，如表1所示。可以看到静态线程的 IPC 要高于动态线程，执行时间更短，程序运行更加高效，性能更加优异。

IPC			
查询数量	10	50	100
Dynamic thread	2.09	2.18	2.21
Static thread	2.25	2.27	2.28

表 1: Pthread 动态线程与静态线程 IPC 对比

8.1.2 线程数的影响

本节主要基于 Bitmap 存储方式对不同线程数进行程序性能剖析。我们使用 perf 分别测量不同线程数下程序的 IPC，如表2所示。我们看到使用更多线程可以提高程序运行的 IPC，加快指令执行，达到更好的性能。

IPC			
查询数量	10	50	100
1 线程	2.21	2.24	2.27
2 线程	2.23	2.27	2.29
4 线程	2.24	2.29	2.30
8 线程	2.26	2.33	2.35

表 2: Pthread 不同线程数 IPC 对比

8.2 OpenMP

8.2.1 循环内与循环外创建线程

本节主要基于 Bitmap 存储方式分别对循环内、外创建线程两种范式进行性能剖析。使用 perf 分别测量不同查询数量下程序 IPC，如表3所示。与 Pthread 类似，OpenMP 在循环外创建线程类似于 Pthread 中的静态线程范式，同样能够提高程序 IPC，达到更好的性能。

IPC			
查询数量	10	50	100
循环内	2.19	2.21	2.27
循环外	2.21	2.26	2.30

表 3: OpenMP 循环内、外创建线程 IPC 对比

8.2.2 动态划分与平均划分

本节主要对动态划分和平均划分两种任务划分方式进行性能剖析。使用 perf 分别测量不同查询数量下程序 cache 未命中率，如表4所示。发现动态划分在 cache 未命中率上要稍低于平均划分，对 cache 利用更加充分，程序性能有所提升。

cache miss rate			
查询数量	10	50	100
平均划分	0.32%	0.33%	0.32%
动态划分	0.17%	0.17%	0.16%

表 4: OpenMP 平均划分和动态划分 cache miss 对比

9 ELEMENT-WISE

在 SIMD 实验中，串行算法部分得到二分查找算法获得了很好的性能提升，因此，在本次实验我们选用二分查找求交算法作为串行基础算法。通过 pthread 以及 openMP 的多线程编程实现并行化，从而进一步探讨多线程的并行化对于程序性能的影响。

9.1 任务划分

element-wise 基础算法的伪代码如下所示。对于该算法，同样存在内外两层 for 循环。与 list-wise 不同的是，外层循环遍历第一个倒排列表 l_1 的所有元素，对于后面所有的倒排列表，如果都存在该元素，则将该元素加入到最终的结果 result 数组之中。因此，外层循环之间不存在明显的数据依赖（由于插入元素的顺序是随机的，最后还要进行一次排序操作），并不存在冲突，所以可以对外层循环进行任务划分。我们将 l_1 的所有元素分配给不同线程，每个线程承担部分元素的查找任务。

Algorithm 3 element-wise 算法

Input: l(t1),l(t2),...,l(tk), 按升序排序，|l(t1)| |l(t2)| ... |l(tk)|

Output: $\cap_{1\leq i\leq k} l\left(t_i\right)$

```
for each element e ∈ S do
    for i = 2 to k do
        found = find(e, (l_i))
        if found = FALSE then
            break
        end if
    end for
    if found = TRUE
        result ← result ∪ {e}
    end if
end for
return result
```

在生成线程所要执行函数的参数时，我们创建一个数据结构，将所需要的各种参数如线程 t_id，问题规模大小等打包起来通过指针转换来传递。同时，针对 element 算法需要特别注意的是，由于最终输出的 result 数组需要保持有序，因此，在不同的线程在执行时将相应元素加入到 result 数组中时，需要保证多个线程同步操作后得到的 result 数组是有序的。这似乎是一个挑战，但实际上有比较好的解决方法：在打包好的数据结构中声明一个局部的 result 数组，各个线程中的数组是有序的。所有线程并行执行完毕后，利用相关函数对几个局部的 result 数组作拼接，最终得到结果数组。

9.2 负载均衡

针对于对第一个倒排列表 l_1 的元素划分问题，我们可以考虑两种划分方式：

第一种方式考虑**平均划分**，将 l_1 列表的全部元素按照线程数量平均分配，例如 l_1 中存在 1000 个元素，线程数为 4，则我们将第 1-250 个元素的查找任务分配给 0 号线程，第 251-500 个元素分配给 1 号线程，以此类推。虽然这样的任务分配方式比较直观和简单，但是由于每一个线程中分配到的元素存在不确定性，某些线程可能要频繁地将元素加入结果数组，因此导致某个线程地操作时间过程，而其他线程处于空闲状态，从而出现负载不均的问题，性能达不到理想的结果。

针对上述负载不均问题，我们考虑**动态任务划分**，构建一个任务池（任务队列），在 l_1 上维护一个索引指针。每个线程根据该索引动态的获取查找任务。每当一个线程完成当前任务后，立即到任务池中再选择一个任务执行，知道左右任务被完成。此外考虑到动态划分会频繁地对 l_1 进行加锁和解锁，我们可以使用粗粒度进行划分，减少锁次数，进一步提高性能。

9.3 Pthread 编程范式

Pthread 编程存在两种范式，分别为动态线程和静态线程。针对于 element-wise 算法，我们考虑分别使用这两种范式进行 Pthread 编程。

对于动态线程而言，我们可以在外层循环创建线程，线程池中的所有线程只完成在除 l_1 的倒排列表中的寻找 l_1 中部分元素的求交任务，完成后该线程立即被销毁。下一次再分配 l_1 中元素时重复创建线程——寻找相应元素——销毁线程的步骤。使用动态线程范式的好处是在没有并行计算需求时（外层循环）不会占用系统资源，缺点是是有较大的线程创建和销毁开销。

对于静态线程而言，我们可以在程序初始化时就创建好线程。对于 l_1 中元素在剩余倒排列表中的寻找工作，将任务分配给线程执行。但执行完毕后并不结束线程，用 Pthread 同步机制等待其他线程完成任务之后再由主线程或其他线程完成结果的合并工作，直至整个程序结束。使用静态线程范式的好处是没有频繁的线程创建、销毁开销，性能更优；缺点是线程一直保持，占用系统资源，可能造成资源浪费。

9.4 Pthread 编程实现

9.4.1 动态线程

在进行求交任务之前，我们需要定义线程的数据结构，将动态线程的每一个子线程的线程编号、带求交的倒排列表等读取到数据结构中。

```
struct ThreadData {
    vector<vector<uint32_t>> vec;
    vector<int> result;
    int t_id;
};
```

定义线程函数，其中分别展示了使用平均划分方式和动态划分方式的代码实现:

```
int size; //划分粒度
int taskPointer = 0; //任务指针
pthread_mutex_t mutex_task; //锁
const int NUM_THREADS; //线程数量
const int NUM_TASKS; //任务数量

void* elementWiseThread(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    vector<vector<uint32_t>> vec = data->vec;
    int t_id = data->t_id;

    //平均划分方式
    int step = vec[0].size() / NUM_THREADS;
    int start = t_id * step;
```

```
    int end = (t_id == NUM_THREADS - 1) ? vec[0].size() : (t_id + 1) * step;
    element-wise(vec, start, end); //按元素求交

    //动态划分方式
    size = vec[0].size() / NUM_TASKS;
    while (true) {
        int start, end;
        pthread_mutex_lock(&mutex_task);
        start = taskPointer;
        taskPointer += size;
        pthread_mutex_unlock(&mutex_task);
        if (start >= vec[0].size())
            break; // 所有任务已完成
        end = min(start + size, (int)vec[0].size());
        element-wise(vec, start, end); //按元素求交
    }

    pthread_exit(NULL);
}
```

主线程部分：

```
vector<int> element_wise_pthread(const vector<vector<uint32_t>>& vec, int numThreads)
    {
    vector<int> result;
    pthread_t threads[numThreads];
    ThreadData threadData[numThreads];
    for (int i = 0; i < numThreads; ++i) {
        threadData[i].vec = vec;
        pthread_create(&threads[i], NULL, elementWiseThread, (void*)&threadData[i]);
    }
    for (int i = 0; i < numThreads; ++i) {
        pthread_join(threads[i], NULL);
        result.insert(result.end(), threadData[i].result.begin(),
            threadData[i].result.end());
    }
    return result;
}
```

9.4.2 静态线程 + 信号量同步

与 list-wise 算法不同的是，element-wise 算法对外层循环进行划分，能够一次性创建所有的线程并分配相应任务。因此，对其进行动态线程范式编程时，并不需要重复地进行线程创建和销毁，而这几乎和静态线程所达到的效果是相同的。可以预想到的是，如果采用 element-wise 算法进行倒排求交操作，对其实现 pthread 动态或者静态的并行化，这两者对于程序性能的影响没有明显差别。为了验证上述结论，我们需要进行静态线程范式编程，使用信号量同步机制，通过信号量来控制工作线程的开始和结束，这种情况下不需要用到主线程 sem_main。

首先定义信号量和数据结构，数据结构和上述一致：

```
sem_t sem_start[NUM_THREADS];
sem_t sem_end[NUM_THREADS];
```

定义线程函数，同样利用平均划分方式和动态划分方式，这里部分与上述重叠的部分不再在报告中进行展示：

```
void* elementWiseThread(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    vector<vector<uint32_t>> vec = data->vec;
    int t_id = data->t_id;
    sem_wait(&sem_start[t_id]);
    ...
    sem_post(&sem_end[t_id]);
    pthread_exit(NULL);
}
```

主线程部分：

```
vector<int> element_wise_pthread(const vector<vector<uint32_t>>& vec, int numThreads)
    {
    ...
    for (int i = 0; i < NUM_THREADS; ++i)
        sem_post(&sem_start[i]);
    for (int i = 0; i < NUM_THREADS; ++i)
        sem_wait(&sem_end[i]);
    ...
}
```

9.4.3 静态线程 + barrier 同步

除了使用信号量同步，element-list 算法同样可以使用 barrier 同步，我们只需要一个 barrier 即可。

首先定义 barrier 和数据结构，数据结构和上述一致：

```
pthread_barrier_t barrier;
```

定义线程函数，同样利用平均划分方式和动态划分方式，这里部分与上述重叠的部分不再在报告中进行展示：

```
void* elementWiseThread(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    vector<vector<uint32_t>> vec = data->vec;
    int t_id = data->t_id;
    ...
    pthread_barrier_wait(&barrier);
    pthread_exit(NULL);
}
```

主线程部分：

```
vector<int> element_wise_pthread(const vector<vector<uint32_t>>& vec, int numThreads)
    {
    //barrier初始化
    pthread_barrier_init(&barrier,NULL,NUM_THREADS);
    ...
    //barrier销毁
    pthread_barrier_destroy(&barrier);
    return result;
}
```

9.5 OpenMP 编程范式

OpenMP 编程范式相较于 Pthread 编程范式，有易用性和可读性方面具有明显优势。OpenMP 提供了高层次的编程接口，使用编译器指令 pragmas 来标记并行区域，使得代码更易读和编写。OpenMP 的编程模型简洁，大部分的管理以及各种各样的功能都由 OpenMP 运行时系统自动处理。这大大减少了代码的复杂性。OpenMP 的易用性和可读性主要表现在以下几个方面：

- 线程的创建、同步和销毁：使用编译器指令 pragmas 来标记并行区域，开发者不需要显式地管理线程的创建、同步和销毁，这些都由 OpenMP 运行时系统自动处理，因此调试和维护相对容易。
- 负载均衡和调度：OpenMP 提供了多种调度策略（如静态、动态、引导等）来优化线程负载均衡，开发者可以通过简单的编译器指令进行配置。
- 减少竞态条件：通过 OpenMP 的简洁指令，减少了由于手动管理线程引入的竞态条件风险，例如可以直接使用 pragma omp critical 实现临界区。
- 多种并行模式：OpenMP 支持多种并行模式，如数据并行、任务并行、循环并行等，能够适应不同类型的应用需求。

对于 element-wise 算法，由于它可以对外层循环进行直接划分，因此我们在外层循环前面编写 pragmas 指令形成相应并行区域，再声明相关参数确定算法的线程总数，调度策略等具体模式。我们主要考虑 static 和 dynamic 两种 schedule 方式进行任务划分，比对其性能差异。

9.6 OpenMP 编程实现

以 static 的任务划分方式为例，对 element-wise 的普通算法进行 OpenMP 编程。

```
vector<int> element_wise_openmp_static(vector<vector<uint32_t>> vec) {
    vector<int> result;
    bool found = false;
    #pragma omp parallel for num_threads(NUM_THREADS) schedule(static) private(found)
        //声明为静态编程
        for(int i = 0;i < vec[0].size();i++){
            found = false;
            for(int j = 1;j < vec.size();j++){
                found = binarySearch(vec[j],vec[0][i]);
                if(!found) break;
```

```
                if(found && j == vec.size() - 1) {
                    #pragma omp critical
                    result.push_back(vec[0][i]);
                }
            }
        }
    sort(result.begin(), result.end());
    return result;
}
```

10 ELEMENT-WISE-BITMAP

10.1 任务划分与负载均衡

对于 element-wise 的位图存储形式，我们将整个求交过程分为两个阶段：位图转置和判断。位图转置是将经过压缩得到倒排列表的位图形式的行列进行转换，判断是根据转置后的位图的每一行进行判断，如果这一行的 bit 全为 1，说明求交得到的结果包含这个数，把它加入结果数组。从上述描述不难发现，这两个阶段均可以实现 Pthread 的并行化，我们分开对两个阶段进行任务划分。

位图转置阶段的所有向量中所有元素都要进行运算，因此采用平均分配方式不存在负载不均问题，无需使用动态划分方式。而在判断阶段，每一行 bit 位数的情况不同，可能存在某个线程需要进行繁琐的加入结果数组的计算，因此在这个阶段中我们采用动态划分方式（以信号量为例）。

10.2 Pthread 编程范式

同样地，使用动态线程和静态线程两种方式进行 Pthread 编程实现。在判断阶段用信号量进行同步。编程思路与列表存储的实现方式类似，不再进行赘述。

10.3 Pthread 编程实现

10.3.1 动态线程

在位图转置阶段采用平均划分方式，而在判断阶段采用动态划分方式，具体代码如下：

```
//数据结构
struct ThreadData1 {
    vector<vector<uint32_t>> bitLists;
    vector<uint32_t> result;
    int t_id;
}; //位图转置阶段
struct ThreadData2 {
    vector<uint32_t> bitListsTranspose;
    vector<int> result;
    int testNumber; //由于判断的整数
    int t_id;
}; //判断阶段

//线程函数（位图转置阶段）
```

```
void* transposeBitmapThread(void* arg) {
    ThreadData1* data = (ThreadData1*)arg;
    vector<vector<uint32_t>> bitLists = data->bitLists;
    int t_id = data->t_id;
    //平均划分
    int num = bitLists[0].size()*sizeof(unsigned int)*8;
    int step = num / NUM_THREADS;
    int start = t_id * step;
    int end = (t_id == NUM_THREADS - 1) ? num : (t_id + 1) * step;
    for (int i = start; i < end; ++i)
        data->result.push_back(0);
    for (int i = start; i < end; ++i) {
        for (int j = 0; j < bitLists.size(); ++j) {
            bool bitSet = bitLists[j][i / (sizeof(uint32_t) * 8)] & (1 <<
                (sizeof(uint32_t) * 8 - 1 - (31 - i) % (sizeof(uint32_t) * 8)));
            data->result[i - start] = (data->result[i - start] << 1) + bitSet;
        }
    }
    pthread_exit(NULL);
}
//线程函数（判断阶段）
void* elementWiseBitmapThread(void* arg) {
    ThreadData2* data = (ThreadData2*)arg;
    vector<uint32_t> bitListsTranspose = data->bitListsTranspose;
    int testNumber = data->testNumber;
    //动态划分
    size = bitListsTranspose.size() / NUM_TASKS;
    while (true) {
        int start, end;
        pthread_mutex_lock(&mutex_task);
        start = taskPointer;
        taskPointer += size;
        pthread_mutex_unlock(&mutex_task);
        if (start >= bitListsTranspose.size())
            break; // 所有任务已完成
        end = min(start + size, (int)bitListsTranspose.size());
        for (int i = start; i < end; ++i) {
            if (bitListsTranspose[i] == testNumber)
                data->result.push_back(i);
        }
    }
    pthread_exit(NULL);
}
//相应主线程的函数与前面类似，因此不再赘述
```

10.3.2 静态线程

在判断阶段采用信号量进行同步，可以估计其程序性能与静态线程没有明显差别，这里只写出判断阶段线程函数的代码，其他部分与上述一致，重叠的部分也不再展示。

```
//线程函数（判断阶段）
void* elementWiseBitmapThreadSem(void* arg) {
    ThreadData2* data = (ThreadData2*)arg;
    vector<uint32_t> bitListsTranspose = data->bitListsTranspose;
    int testNumber = data->testNumber;
    int t_id = data->t_id;
    sem_wait(&sem_start[t_id]);
    ...
    sem_post(&sem_end[t_id]);
    pthread_exit(NULL);
}
//相应主线程的函数与前面类似，因此不再赘述
```

10.4 OpenMP 编程实现

类似地，同样以 static 的任务划分方式为例，我们考虑 OpenMP 范式编程来实现位图存储的 elemen-wise 求交。

```
//位图转置阶段
vector<uint32_t> transpose_bitmap_openmp_static(const vector<std::vector<uint32_t>>&
    bitLists) {
    vector<uint32_t> bitLists_transpose(bitLists[0].size()*sizeof(unsigned int)*8);
    int bit_count = sizeof(unsigned int)*8;
    #pragma omp parallel for num_threads(NUM_THREADS) schedule(static)
        //声明为静态编程
    for (int i = 0; i < bitLists_transpose.size(); i++) {
        uint32_t transposed_value = 0;
        for (int j = 0; j < bitLists.size(); j++) {
            bool bitSet = bitLists[j][i / bit_count] & (1 << (bit_count - 1 - (i %
                bit_count)));
            transposed_value = (transposed_value << 1) | bitSet;
        }
        bitLists_transpose[i] = transposed_value;
    }
    return bitLists_transpose;
}
//判断阶段
vector<int> element_wise_bitmap_openmp_static(const vector<uint32_t>&
    bitLists_transpose, int test_number) {
    std::vector<int> result;
    #pragma omp parallel num_threads(NUM_THREADS)
        #pragma omp for schedule(static) nowait //声明为静态编程
        for (int i = 0; i < bitLists_transpose.size(); i++) {
            if (bitLists_transpose[i] == test_number) {
```

```
                #pragma omp critical
                result.push_back(i);
            }
        }
    sort(result.begin(), result.end());
    return result;
}
```

11 ELEMENT-WISE Pthread 结果分析

11.1 线程数对性能的影响

对于普通存储方式，我们选取普通存储方式中的平均任务划分 + 动态线程方式分析线程数对性能的影响，其中将线程数为 1 作为串行时间。同样地，对于 Bitmap 存储方式，我们选取动态线程范式版本进行分析。根据实验结果得到的折线图如图11.11和11.12所示。

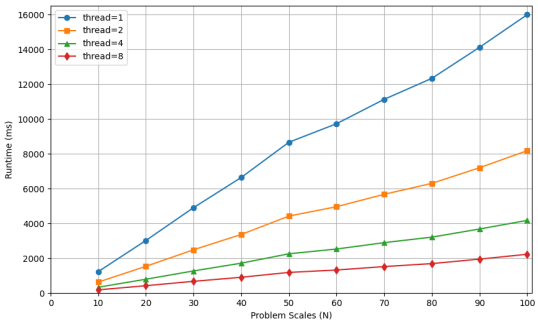


图 11.11: 普通存储：线程数对性能的影响

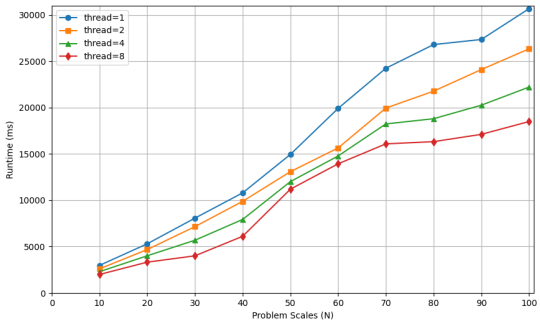


图 11.12: Bitmap 存储：线程数对性能的影响

可以发现，在不同的问题规模之下，与串行算法相比，随着线程数的增加，element-wise 普通算法的运行时间成倍减少，其比例与线程数几乎保持一致，能够验证理论的结果。但是对于 Bitmap 算法，虽然随着线程数的增加，其算法的运行时间也逐渐减少，但其下降幅度明显与线程数量不匹配，甚至在某些情况下仅仅出现了微小幅度的变化。

Bitmap 算法理论证明和实际结果的差异可能来自于其本身算法的复杂性。element-wise 算法的基本框架分为位图转置和判断两个阶段，而每一个阶段都涉及 Pthread 线程的销毁与创建过程，而且其计算任务比较庞大，增加了算法本身的复杂性。因此，通过增加线程数的方式优化性能，创建和管理线程本身需要一定的开销。如果线程数过多，这些开销可能会抵消多线程带来的性能提升；另一方面，有可能是平均划分方式所带来的任务划分不均问题。如果任务划分不均匀，有些线程可能会比其他线程多做一些工作，导致负载不均衡。这样一些线程会空闲等待其他线程完成工作，从而无法充分利用多线程的优势。

11.2 动态、静态线程以及不同同步机制的性能差异

Pthread 编程的范式有动态线程和静态线程两种。我们采取不同的编程范式可以得到不同的效果。同时在静态线程中，我们可以信号量同步、barrier 同步等同步方式避免频繁的创建和销毁线程。对于普通存储方式和 Bitmap 存储方式，我们选取总线程数为 4，考虑平均任务划分的方式分析动态线程

和静态线程对程序性能的性能差异、以及静态线程中不同同步机制的性能差异。根据实验结果得到的对比折线图如图11.13和11.14所示。

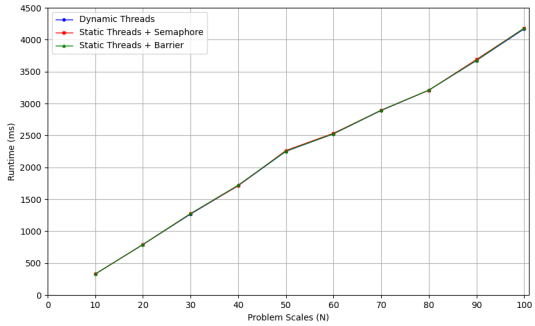


图 11.13: 普通存储：不同同步机制的性能差异

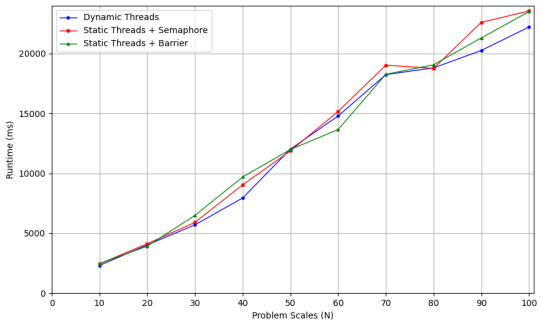


图 11.14: Bitmap 存储：不同同步机制的性能差异

在上述算法分析中我们提到过，由于 element-wise 算法只对外层循环进行任务划分的操作，不管利用哪一种 Pthread 的编程范式，线程创建和销毁的工作只在并行区域开头和末尾即可完成，因此，采用动态线程或者静态线程对 element-wise 算法的 Pthread 并行程序性能几乎不会产生任何影响。从两张折线图中不难得出，随着问题规模的变化，三种情况的折线始终重叠在一起或相互靠近，运行时间的差距在误差范围以内，可以忽略不计，能够验证上述结论。

11.3 两种划分策略对性能的影响

我们考虑到由平均划分方式所带来的负载不均的问题，引入动态任务划分：构建一个任务池，每个线程根获取相应大小的查找任务。每当一个线程完成当前任务后，立即到任务池中再选择一个任务执行直到所有任务被完成。更进一步地，我们可以使用不同粒度进行动态任务划分，探究动态划分中颗粒度对性能地影响，试图找到最佳地动态划分策略。

11.3.1 动态划分中颗粒度对性能的影响

由于程序运行的过程中，其任务总量会因具体问题而变化，因此，我们并不直接定义每个任务的大小（size），而是引入任务数（number of tasks），通过设置不同的任务数来控制每个任务的大小占任务总数的比例。本次实验我们动态线程方式，并设定线程数为 4，在设置任务数的时候将其默认为 4 的倍数。除此以外，单纯以 4 的倍数设置任务数可能无法得出完整的结论，我们尝试设置其他两组对照组，分别控制任务数的轻微浮动，使其上下浮动的值为 1（+1 和-1），例如，设定任务数为 20 时，同时设置对照组的任务数为 19 和 21。这样设置的目的是分析不同情况下的动态划分过程中各个线程的负载量以及执行情况。

任务数取 10 个梯度。我们以粗粒度（任务数为 4 40）和细粒度（任务数为 20 200）的划分分别进行两次实验，取实验规模 N=10。根据实验结果得到的对比折线图如图11.15和11.16所示，其中 Baseline 为平均划分方法的运行时间。

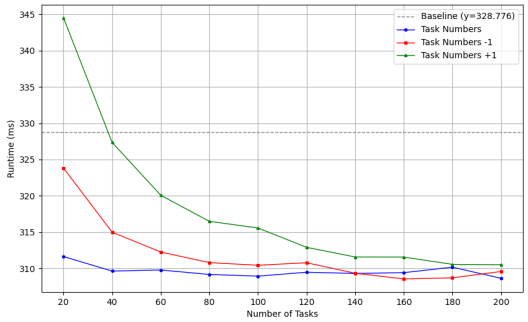


图 11.15: 细粒度划分对性能的影响

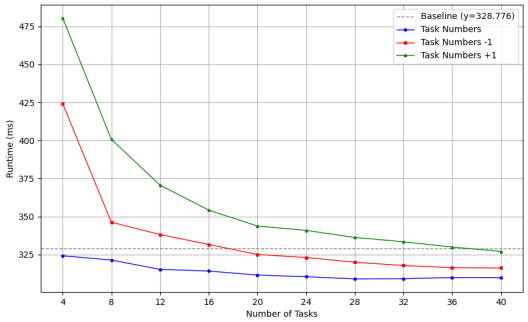


图 11.16: 粗粒度划分对性能的影响

综合两张折线图不难发现，任务数为 4 的倍数时，选取不同颗粒度的动态划分方式均能取得优化程序性能的效果，在其中任务数 100 的优化效果最好。而当任务数在原有基础上上下浮动（任务数 +1 或-1）时，程序性能下降，并且采取任务数 + 1 的动态划分方式的程序性能下降幅度更明显。性能之间的差异与不同线程的负载量和具体执行情况有关。在不同线程的负载量差异不大的情况下，如果任务数不是 4 的倍数（即任务数不是线程数的倍数），假设任务平均分给 4 个线程，将会有多余的任务未被分配，这可能会造成部分线程处于空闲状态而使性能降低。具体来说，如果浮动为-1，将可能会有 1 个线程处于空闲状态；如果浮动为 +1，将可能会有最多 3 个线程处于空闲状态。所以，按照上述分析，浮动 +1 的性能不及浮动-1 的性能。

将两张折线图对比来看，可以发现，粗粒度划分相较于细粒度划分，浮动造成的影响更加明显。原因在于，在两种划分都能较好地解决负载不均问题的情况下，细粒度划分中的浮动问题由于任务数较大而能被很好地控制，这是因为每一个线程单次的执行时间较少，对性能下降的影响也会更小。

11.3.2 平均、动态划分方法对性能的影响

我们选取动态划分中表现较好的情况（任务数为 100）来和平均划分方式进行性能的对比分析。在不同的问题规模下，设定线程数为 4，我们选取普通存储方式中的动态线程方式进行实验。根据实验结果得到的对比折线图如图11.17所示。

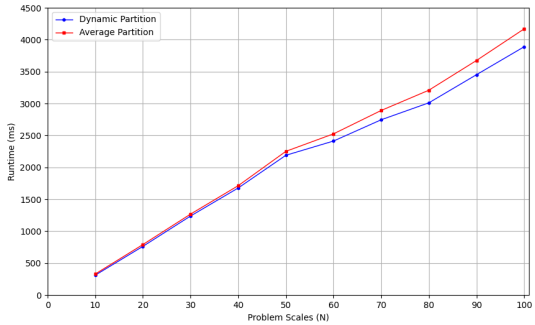


图 11.17: 不同划分方法对性能的影响

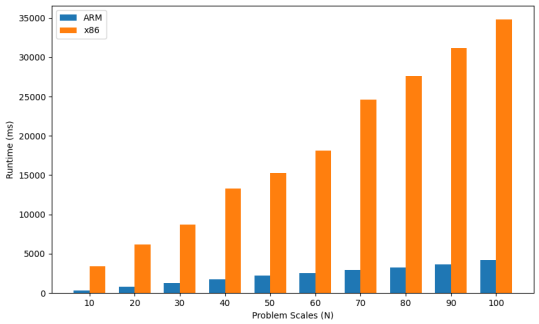


图 11.18: ARM 与 x86 平台性能对比

不难发现，动态划分方式相较于平均划分方式，其性能获得了一定程度的提升。性能提升受到限制的可能原因是频繁的加锁解锁操作和线程执行任务的不连贯性，

11.4 ARM 与 x86 平台性能对比

ARM 和 x86 是两种不同的计算机处理器架构，ARM 架构采用精简指令集（RISC）设计，而 x86 架构采用复杂指令集（CISC）设计，两种架构之间还存在着很多区别。因此，在两个平台上进行实验，其结果具有参考意义。我们选取普通存储方式中的的动态线程方式，考虑平均划分，设定线程数为 4，分析在 ARM 和 x86 平台下的程序性能差异。根据实验结果得到的对比柱状图如图11.18所示。

可以发现，对于 element-wise 的 Pthread 并行化编程实验，在 ARM 平台运行的性能明显优于 x86。

12 ELEMENT-WISE OpenMP 结果分析

12.1 线程数对性能的影响

与 Pthread 结果分析类似，这次我们选取普通存储方式中的动态线程方式分析线程数对性能的影响，其中将线程数为 1 作为串行时间。基于篇幅限制和其他因素，我们在此不再对 Bitmap 存储方式进行赘述。根据实验结果得到的折线图如图12.19所示。

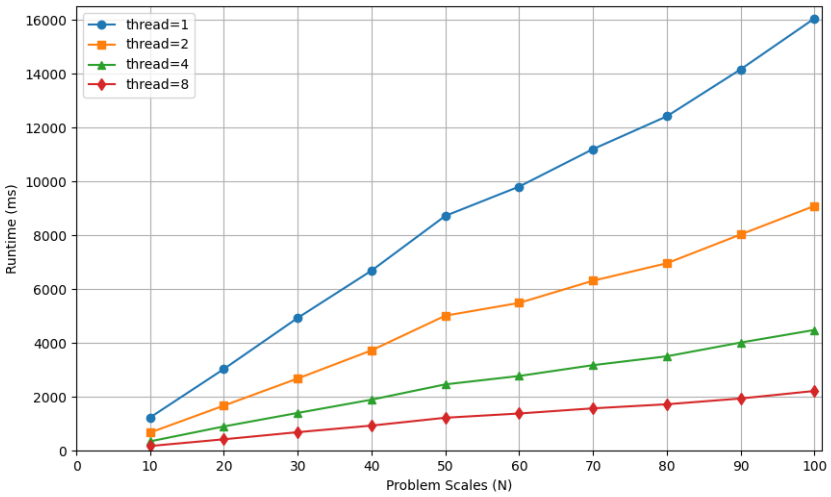


图 12.19: 线程数对性能的影响

基于 OpenMP 并行化编程得到的结论与 Pthread 一致：在不同的问题规模之下，与串行算法相比，随着线程数的增加，element-wise 普通算法的运行时间成倍减少，其比例与线程数几乎保持一致。

12.2 动态、静态线程的性能差异

同样地，与 Pthread 结果分析类似，对于普通存储方式，我们选取总线程数为 4，分析动态线程和静态线程对程序性能的性能差异。根据实验结果得到的对比折线图如图12.20所示。

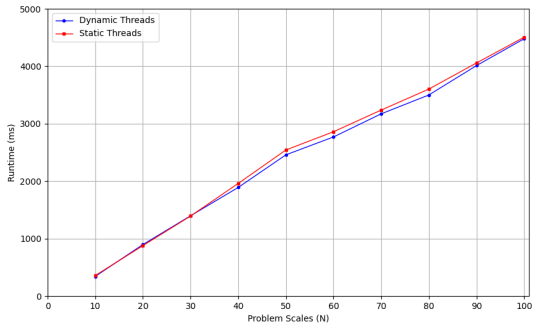


图 12.20: 动态、静态线程的性能差异

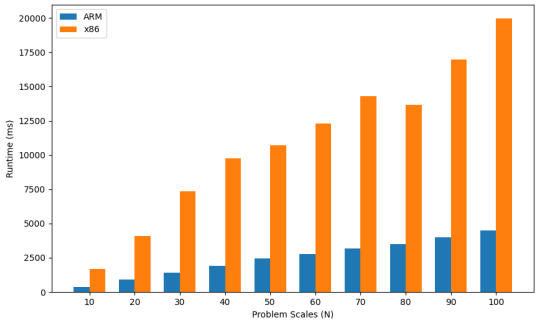


图 12.21: ARM 与 x86 平台性能对比

基于 OpenMP 并行化编程得到的结论与 Pthread 一致：采用动态线程或者静态线程对 element-wise 算法的 OpenMP 并行程序性能几乎不会产生任何影响。

12.3 ARM 与 x86 平台性能对比

与 Pthread 结果分析类似，使用 OpenMP 并行化编程，我们选取普通存储方式中的的动态线程方式，设定线程数为 4，分析在 ARM 和 x86 平台下的程序性能差异。根据实验结果得到的对比柱状图如图12.21所示。

可以发现，对于 element-wise 的 OpenMP 并行化编程实验，在 ARM 平台运行的性能明显优于 x86。

13 ELEMENT-WISE Pthread 与 OpenMP 性能对比

本节我们分别基于 Element-wise 的普通存储和 List-wise 的 Bitmap 存储对 Pthread 和 OpenMP 进行性能对比分析。对于 Element-wise 的普通存储方式，在不同的问题规模下，我们选取动态线程方式，设定线程数为 4，考虑平均划分，分析 Pthread 和 OpenMP 这两种并行化编程方式的性能差异。根据实验结果得到的对比折线图如图13.22所示。

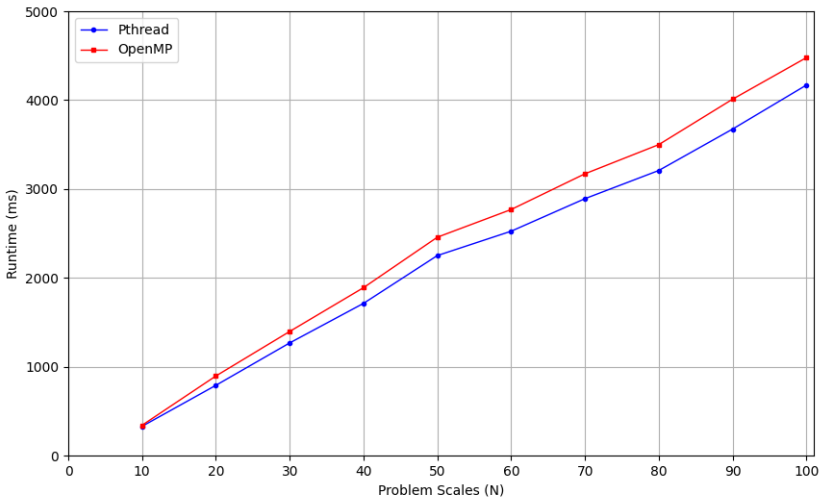


图 13.22: Pthread 与 OpenMP 性能对比

根据折线图我们可以发现，在两种不同的算法中，Pthread 编程的性能都优于 OpenMP 编程。我们推测可能的原因是 Pthread 编程中人为地进行了细致的划分，而 OpenMP 编程使用工作分割模型进行自动任务划分，可能划分的方法比较粗糙，从而造成了 Pthread 和 OpenMP 这两种并行化编程方式性能上的差异。

14 ELEMENT-WISE Profiling

14.1 Pthread

基于 element-wise 普通存储方式的对比分析，我们在华为鲲鹏服务器上使用 perf 进行性能剖析，问题规模设定为 **[10,50,100]**，分别测量 cache miss 率与 IPC。

14.1.1 动态线程与静态线程

在不同问题规模下，我们选择普通存储方式中的平均划分，对动态线程和静态线程作性能剖析，设置线程数为 4，测得的结果如表5：

查询数量	IPC			miss 率		
	动态	静态 (信号量)	静态 (barrier)	动态	静态 (信号量)	静态 (barrier)
10	1.99	1.99	1.99	0.37	0.38	0.38
50	1.98	1.98	1.98	0.39	0.39	0.39
100	1.98	1.98	1.98	0.39	0.37	0.39

表 5: 动态、静态线程对应的 IPC 和 miss 率

14.1.2 线程数的影响

在不同问题规模下，我们选取普通存储方式中的动态线程方式和平均划分方法，对不同的线程数进行剖析，测得的结果如表6：

查询数量	IPC				miss 率			
	thread=1	thread=2	thread=4	thread=8	thread=1	thread=2	thread=4	thread=8
10	1.99	1.99	1.99	1.99	0.38	0.38	0.37	0.45
50	1.99	1.99	1.98	1.98	0.38	0.38	0.39	0.40
100	1.99	1.99	1.98	1.98	0.37	0.38	0.39	0.46

表 6: 不同线程数对应的 IPC 和 miss 率

14.1.3 平均划分和动态划分

在不同问题规模下，我们选取普通存储方式中的动态线程，对平均划分和动态划分进行剖析，同时动态划分在粗粒度 (任务数为 20) 和细粒度 (任务数为 100) 中各选择一个值进行分析，设置线程数为 4，测得的结果如表7：

查询数量	IPC			miss 率		
	平均	动态 (粗粒度)	动态 (细粒度)	平均	动态 (粗粒度)	动态 (细粒度)
10	1.99	1.99	1.99	0.37	0.38	0.38
50	1.98	1.99	1.98	0.39	0.37	0.39
100	1.98	1.99	1.98	0.39	0.37	0.38

表 7: 平均划分和动态划分对应的 IPC 和 miss 率

观察上面三组表格，可以发现在 Pthread 编程环境下，element-wise 普通存储方式每一种情况下的 IPC 和缓存 miss 率几乎保持一致。这表明对于 element-wise 算法而言，采用 Pthread 不同的环境下的性能是稳定的。

14.2 OpenMP

与 Pthread 类似，我们在华为鲲鹏服务器上使用 perf 进行性能剖析，问题规模设定为 **[10,50,100]**，分别测量 cache miss 率与 IPC。

14.2.1 动态线程与静态线程

在不同问题规模下，我们选择普通存储方式中的平均划分，对动态线程和静态线程作性能剖析，设置线程数为 4，测得的结果如表8：

查询数量	IPC		miss 率	
	动态	静态	动态	静态
10	1.96	1.98	0.39	0.42
50	1.85	1.95	0.39	0.39
100	1.84	1.96	0.38	0.38

表 8: 动态、静态线程对应的 IPC 和 miss 率

14.2.2 线程数的影响

在不同问题规模下，我们选取普通存储方式中的动态线程方式和平均划分方法，对不同的线程数进行剖析，测得的结果如表9：

查询数量	IPC				miss 率			
	thread=1	thread=2	thread=4	thread=8	thread=1	thread=2	thread=4	thread=8
10	1.98	1.97	1.96	1.96	0.42	0.39	0.39	0.41
50	1.97	1.92	1.85	1.89	0.40	0.37	0.39	0.39
100	1.98	1.85	1.84	1.93	0.41	0.39	0.38	0.40

表 9: 不同线程数对应的 IPC 和 miss 率

观察上面两组表格，可以发现在 OpenMP 编程环境下，element-wise 普通存储方式随问题规模的增大，IPC 有降低的趋势。这表明对于 element-wise 算法而言，OpenMP 的负载负载增加时，会对程序性能产生一定的影响。

15 组内分工

本次实验由张铭 (2211289) 和张高（2213219）两人共同完成。张铭同学负责完成的任务包括：基于 Listwise 算法完成一系列 Pthread 编程算法和 OpenMP 编程算法实现；基于编程得到实验结果进行 Listwise 实验结果分析与对比，并对此使用 perf 进行程序性能剖析，绘制相关图表，撰写相关内容的实验报告。张高同学负责完成的任务包括：基于 Elementwise 算法完成一系列 Pthread 编程算法和 OpenMP 编程算法实现；基于编程得到实验结果进行 Elementwise 实验结果分析与对比，并对此使用 perf 进行程序性能剖析，绘制相关图表，撰写相关内容的实验报告。