

pthread 多线程编程

——以高斯消去为例

杜忱莹 周辰霏

2021 年 4 月

周浩

2022 年 4 月

陈静怡

2023 年 4 月

唐明昊

2024 年 4 月

华志远，张逸非，孔德嵘

2025 年 5 月

目录

| | |
|--|-----------|
| 1 实验介绍 | 3 |
| 1.1 实验选题 | 3 |
| 1.2 实验要求 | 3 |
| 2 实验设计指导 | 3 |
| 2.1 实验总体思路 | 3 |
| 2.2 Pthread 编程范式 | 4 |
| 2.3 作业注意要点及建议 | 5 |
| 2.4 算法描述 | 8 |
| 2.5 编译、运行 | 15 |
| 3 ANN 选题：多线程实验 | 15 |
| 3.1 IVF (Inverted File) | 15 |
| 3.2 HNSW (Hierarchical Navigable Small World graphs) | 17 |
| 4 口令猜测选题：多线程实验 | 18 |
| 4.1 口令猜测算法的并行化 | 18 |
| 4.2 基础要求 | 19 |
| 4.3 进阶要求 | 20 |
| 5 NTT 选题：多线程实验 | 21 |
| 5.1 朴素多线程优化 | 21 |
| 5.2 多分多线程优化 | 21 |
| 5.3 CRT 多线程优化 | 22 |
| 6 VTune 分析 (本地环境) | 22 |

1 实验介绍

1.1 实验选题

高斯消去（基础），ANN，NTT，口令猜测算法四选一。

1.2 实验要求

1. 基本要求（最高获得 90% 分数）：ARM 平台上普通高斯消去计算的基础 pthread 和基础 OpenMP 并行化实验：
 - 设计实现适合的任务分配算法，分析其性能；
 - 与 SIMD（Neon）算法相结合；
 - 在 ARM 平台上编程实现、进行实验，测试不同问题规模、不同线程数下的算法性能（串行和并行对比），讨论一些基本的算法/编程策略对性能的影响，讨论 Pthread 程序和 OpenMP 程序的性能差异。
2. * 进阶要求：ANN，NTT，口令猜测算法三选一，具体要求可参考各实验指导书。

2 实验设计指导

2.1 实验总体思路

以高斯消去法为例：

1. 首先初始化生成矩阵元素值，按照上次作业给出的伪代码实现高斯消去法串行算法。
2. 使用课堂所学 pthread 编程任务划分和同步机制，针对消去部分的两重循环，按照不同任务划分方式（如按行划分和按列划分），分别设计并实现 pthread 多线程算法。并考虑将其与 SIMD 算法结合。对各算法进行复杂度分析（基本的运行时间、加速比的分析以及更深入的伸缩性分析），思考能否继续改进。

3. 实验方面，改变矩阵的大小、线程数等参数，观测各算法运行时间的变化，对结果进行性能分析。借助 Vtune profiling 等工具分析算法过程中的同步开销和空闲等待等。

2.2 Pthread 编程范式

Pthread 编程相比于 OpenMP 编程相对更底层一些，有助于我们更好地对线程进行管理和协调，我们也可以更加灵活地分配并行任务以及管理任务调度。

Pthread 程序需要包含头文件：<pthread.h>

GCC 编译选项：-lpthread（新版本改为-pthread）

主要基础 API：

```
1 int pthread_create(pthread_t *, const pthread_attr_t *, void * (*)(void *), void *);
2 //用于创建线程
3 //第一个参数为线程 id 或句柄（用于停止线程等）
4 //第二个参数代表各种属性，一般为空（代表标准默认属性）
5 //第三个参数为创建出的线程执行工作所要调用的函数
6 //第四个参数为第三个参数调用函数所需传递的参数
7 int pthread_join(pthread_t *, void **value_ptr)
8 //除非目标线程已经结束，否则挂起调用线程直至目标线程结束
9 //第一个参数为目标线程的线程 id 或句柄
10 //第二个参数允许目标线程退出时返回信息给调用线程，一般为空
11 void pthread_exit(void *value_ptr);
12 //通过 value_ptr 返回结果给调用者
13 int pthread_cancel(pthread_t thread);
14 //取消线程 thread 执行
```

Pthread 编程有两种范式：

- **静态线程**：程序初始化时创建好线程（池）。对于需要并行计算的部分，将任务分配给线程执行。但执行完毕后并不结束线程，等待下一个并行部分继续为线程分配任务。直至整个程序结束，才结束线程。优点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。
- **动态线程**：在到达并行部分时，主线程才创建线程来进行并行计算，在这部分完成后，即销毁线程。在到达下一个并行部分时，再次重复创建线程——并行计算——销毁线程的步骤。优点是再没有并行计算需求时不会占用系统资源；缺点是有较大的线程创建和销毁开销。

程序的具体结构，通常是在主函数（主线程）中使用 `pthread_create` 函数创建工作线程，将并行部分抽取出来形成线程函数（将计算任务——通常体现为循环结构——划分给多个线程），工作线程执行线程函数进行并行计算，主函数调用 `pthread_join` 函数等待工作线程完成。

对于静态线程范式，在串行版本的基础上，通常是在主线程开始即创建所有工作线程，在主线程结束时等待线程结束，将串行程序的主体都纳入线程函数，在需要并行的地方进行任务划分。对于动态线程范式，在串行版本的基础上，在并行部分之前创建线程，将并行部分形成线程函数，紧接着等待线程结束。

注意：这里的并行部分考虑的是动态的程序执行，而非静态的程序结构。例如，程序主体是一个双重循环，外层循环内、内层循环之外基本没有实质性计算，主要计算操作都在内层循环内，我们选择将内层循环拆分配给工作线程。此时如果机械地从程序结构角度出发，**只将内层循环放入线程函数中**，外层循环还保留主函数中，则有两种情况：要么采用动态线程范式，**外层循环的每步执行都创建、销毁所有工作线程，带来严重的额外开销**；采用静态线程范式的话，**主线程和工作线程之间的通信（同步）就会很复杂**，程序易读性、可维护性大大下降。而好的方式是，从程序动态执行的角度看，其实外层循环执行的全过程都是处于并行部分中（因为内外层循环之间并无计算，外层循环的执行实际上只是在持续执行内层循环而已），因此**将双重循环都置于线程函数中，对外层循环保持不动，将内层循环拆分配工作线程即可**。本次作业即可采用这种思路。如内外层循环间还有其他代码，视具体情况处理一下，都可以采用上述思路实现多线程版本。

同步问题为 pthread 编程的重点与难点，方式也有很多：比如忙等待、互斥量和信号量、barrier 与条件变量等等。PPT 中已给出相关 API 及例子，请同学们复习参考 PPT，这里不再一一介绍讲解。

2.3 作业注意要点及建议

1. 矩阵数值初始化问题

根据有些同学们反映，自己初始化矩阵在计算过程中会出现 `inf` 或 `nan` 的问题。这是由于精度问题以及非满秩矩阵造成的。`inf` 或 `nan` 的情况无疑会影响结果正确性的判断，也会在并行计算性能上造成一定影响，而随机生成数据的方式很明显无法保证能避免该问题尤其在规模巨大的情况下。个人建议初始化矩阵时可以首先初始化一个上三角矩阵，然后随机的抽取若

并行去将它们相加减然后执行若干次，由于这些都是内部的线性组合，这样的初始数据可以保证进行高斯消去时矩阵不会有 inf 和 nan。

2. 生成线程所要执行函数的参数问题

由于只有一个 void 指针的参数，可以创建一个数据结构，将所需要的各种参数如线程 id，问题规模大小等打包起来通过指针转换来传递。注意对于均分任务时，当问题规模不等于线程数倍数时，对于分配最后一部分计算任务的线程不要直接使用 $\text{my_first} + \text{my_n}$ 计算 my_last ，可根据线程 id 将 n 作为该线程的 my_last 。对于推荐的范式来说，可直接在线程函数内完成这些工作，就无需传递参数了。

3. 尽量避免频繁的创建和销毁线程

以高斯消去问题为例，回顾一下其串行算法如下面伪代码所示：

```
1  procedure LU (A)
2  begin
3    for k := 1 to n do (外层循环)
4      //除法操作
5      for j := k+1 to n do (第一个内层循环)
6        A[k, j] := A[k, j]/A[k, k];
7      endfor;
8      A[k, k] := 1.0;
9
10     //消去操作
11     for i := k + 1 to n do (第二个内层循环)
12       for j := k + 1 to n do
13         A[i, j] := A[i, j] - A[i, k] × A[k, j];
14       endfor;
15       A[i, k] := 0;
16     endfor;
17   endfor;
18 end LU
```

一个简单直观的多线程思路就是：共进行 n 轮消去步骤（外层循环），第 k 轮执行完第 k 行除法后（第一个内层循环），对于后续的 $k+1$ 至 n 行进行减去第 k 行的操作（第二个内层循环），各行之间互不影响，可采用多线程执行。每轮除法完成后创建线程，该轮消去完成后销毁线程。这样的话创建和销毁线程的次数过多，而线程创建、销毁的代价是比较大的。

可以通过信号量同步、barrier 同步等同步方式避免频繁的创建和销毁线程。以信号量同步思路为例，主线程执行除法，工作线程执行消去。主线程开始时建立多个工作线程；在每一轮消去过程中，工作线程先进入睡眠，主线程完成除法后将它们唤醒，自己进入睡眠；工作线程进行消去操作，完

成后唤醒主线程，自己再进入睡眠；主线程被唤醒后进入下一轮消去过程，直至任务全部结束销毁工作线程。barrier 同步（有助于后面 openmp 编程的学习与理解）和其他同步方式请同学们自行思考，这里不再讲解。

但实际上，如上一节所述，第二种方法也存在程序**逻辑复杂**的问题。更好的方式是采用上一节所述范式，将多重循环都纳入线程函数中，消去操作对应的内层循环拆分，分配给工作线程。对于除法操作，可以只由一个线程执行，也可拆分由所有工作线程并行执行。需要注意的是，除法和消去两个步骤后都要进行**同步**，以保证进入下一步骤（下一轮）之前上一步骤的计算全部完成，保证一致性。

4. 不同任务划分策略

可以看到，高斯消去过程中的计算集中在 5-8 的第一个内层循环（除法）和 11-17 行的第二个内层循环（双重循环，消去），对应矩阵右下角 $(n-k+1) \times (n-k)$ 的子矩阵。因此，任务划分可以看作对此子矩阵的划分。对于除法部分，因为只涉及一行，只可能采用垂直划分（列划分）。而对于消去部分，即可采用水平划分（将其外层循环拆分，即每个线程分配若干行），也可采用垂直划分（将其内层循环拆分，即每个线程分配若干列）。两种划分策略在负载均衡上可能会有细微差异，而在同步方面会有差异，cache 利用方面也会有不同。

此外，在与 SIMD 结合时，SIMD 只能将行内连续元素的运算打包进行向量化，即只能对最内层循环进行展开、向量化。多线程不同任务划分策略要注意与 SIMD 的结合方式。

5. 计算误差与程序正确性

有关问题规模和并行计算由于重排了指令执行顺序和计算机浮点数所导致误差问题说明参考之前实验。

多线程编程一次结果正确并不代表算法的正确性，相比错误的结果，算法错误结果正确无法暴露问题才是最可怕的，需要多进行实验降低错误概率，不仅是同阶矩阵数据的多次实验，还需要设计多组数据进行对比测试，更好地验证并行优化算法对不同数据规模的加速效果，同学们也在代码逻辑上可多进行梳理。多线程编程不易调试 bug，错误也不好复现。需要大家更多的去思考各线程执行过程中的可能情况。

6. 临界区问题

在并行算法中，经常会遇到需要多个线程修改全局变量，而这些线程可能需要修改同一位置的情况，这时就需要考虑使用临界区：在多线程编程

中，临界区是指一段代码，在这段代码中共享资源的访问受到限制，只有一个线程可以进入临界区执行代码，而其他线程必须等待。这是为了避免多个线程同时对共享资源进行写操作而导致的竞争问题。

在使用 pthread 库进行多线程编程时，可以使用 pthread_mutex_t（互斥锁）来创建临界区。互斥锁可以保证在任意时刻只有一个线程可以获得锁，其他线程必须等待该线程释放锁后才能获得锁并进入临界区。这篇文章对 pthread 的互斥锁进行了简单的介绍，同学们可以参考。

当然，除了互斥锁以外，还可以使用条件变量、信号量等方法，请同学们参考课程 PPT 或查阅资料。

*7. 负载均衡问题

对于选择默认选题——高斯消元的同学们而言这一问题比较容易解决，但部分选择自主选题的同学可能面对这一问题比较棘手，因此在这里简单说一下。

我们以编译原理助教检查问题为例。假设约 100 组同学需要找助教检查作业，共 5 位助教。一种很坏的分配方式是同学们自由选择自己喜欢的助教，但可能某位助教极度受欢迎，有过多的同学找他，同时其余助教没有得到充分利用。一种稍微好一点的方法是规定每位助教检查 20 组，如同高斯消元的任务分配方式，但由于每组作业完成情况不同，检查时间不同，依然可能出现某位助教检查时间过长，后面的同学在排队，而其余助教闲置的情况。更好一点的方法是事先不分配，每位助教检查完一组立即随机选取一组未检查的同学进行检查。

回到 pthread 编程的问题，如果你不得不划分出一些计算量大小不一的任务，不便于手动分配给每个线程，可以考虑构建任务池（任务队列），把待完成的任务放进去。每当一个线程完成当前任务后，立刻到任务池中再选取一个任务执行，直到所有任务被完成。不过在这一过程中，一定要正确使用信号量、锁等方法来避免访存冲突。

2.4 算法描述

(1) 动态线程版本

在进行任务之前，我们需要定义线程数据结构以及线程函数：

```
1 typedef struct {  
2     int k; //消去的轮次  
3     int t_id; // 线程 id  
4 }threadParam_t;
```



```

5
6 void *threadFunc(void *param) {
7     threadParam_t *p = (threadParam_t*)param;
8     int k = p -> k; //消去的轮次
9     int t_id = p -> t_id; //线程编号
10    int i = k + t_id + 1; //获取自己的计算任务
11
12    For (int j = k + 1; j < n; ++j) do
13        A[i][j] = A[i][j] - A[i][k] * A[k][j];
14    end For
15    A[i][k] = 0;
16    pthread_exit(NULL);
17 }

```

基于上述线程数据结构以及线程函数的定义，高斯消元 Pthread 动态线程伪代码如下所示：

```

1 int main() {
2     For (int k = 0; k < n; ++k) do
3         //主线程做除法操作
4         For (int j = k+1; j < n; j++) do
5             A[k][j] = A[k][j] / A[k][k];
6         end For
7         A[k][k] = 1.0;
8
9         //创建工作线程，进行消去操作
10        int worker_count = n-1-k; //工作线程数量
11        pthread_t* handles = Malloc(); // 创建对应的 Handle
12        threadParam_t* param = Malloc(); // 创建对应的线程数据结构
13        //分配任务
14        For(int t_id = 0; t_id < worker_count; t_id++)
15            param[t_id].k = k;
16            param[t_id].t_id = t_id;
17        end For
18        //创建线程
19        For(int t_id = 0; t_id < worker_count; t_id++)
20            pthread_create();
21        end For
22        //主线程挂起等待所有的工作线程完成此轮消去工作
23        For(int t_id = 0; t_id < worker_count; t_id++)
24            pthread_join();
25        end For
26    end For
27
28 }

```

(2) 静态线程 + 信号量同步版本

```

1 //线程数据结构定义
2 typedef struct {
3     int t_id; //线程 id
4 }threadParam_t;
5
6 //信号量定义
7 sem_t sem_main;
8 sem_t sem_workerstart[NUM_THREADS]; // 每个线程有自己专属的信号量
9 sem_t sem_workerend[NUM_THREADS];
10
11 //线程函数定义
12 void *threadFunc(void *param) {
13     threadParam_t *p = (threadParam_t*)param;
14     int t_id = p -> t_id;
15
16     For (int k = 0; k < n; ++k) do
17         sem_wait(&sem_workerstart[t_id]); // 阻塞，等待主线完成除法操作（操作自己专属的信号量）
18
19         //循环划分任务
20         For(int i=k+1+t_id; i < n; i += NUM_THREADS) do
21             //消去
22             For (int j = k + 1; j < n; ++j) do
23                  $A[i][j] = A[i][j] - A[i][k] * A[k][j];$ 
24             end For;
25             A[i][k]=0.0;
26         end For;
27         sem_post(&sem_main); // 唤醒主线程
28         sem_wait(&sem_workerend[t_id]); //阻塞，等待主线程唤醒进入下一轮
29     end For;
30     pthread_exit(NULL);
31 }
32
33 int main() {
34     //初始化信号量
35     sem_init(&sem_main, 0, 0);
36     For (int i = 0; i < NUM_THREADS; ++i) do
37         sem_init(&sem_workerstart[i], 0, 0);
38         sem_init(&sem_workderend[i], 0, 0);
39     end for
40
41     //创建线程
42     pthread_t handles[NUM_THREADS]; // 创建对应的 Handle
43     threadParam_t param[NUM_THREADS]; // 创建对应的线程数据结构
44     For(int t_id = 0; t_id < NUM_THREADS; t_id++)
45         param[t_id].t_id = t_id;
46         pthread_create();

```

```

47     end For
48
49     For(int k = 0; k < n; ++k) do
50         //主线程做除法操作
51         For (int j = k+1; j < n; j++) do
52             A[k][j] = A[k][j] / A[k][k];
53         end For
54         A[k][k] = 1.0;
55
56         //开始唤醒工作线程
57         For (int t_id = 0; t_id < NUM_THREADS; ++t_id) do
58             sem_post(&sem_workerstart[t_id]);
59         end For
60
61         //主线程睡眠（等待所有的工作线程完成此轮消去任务）
62         For (int t_id = 0; t_id < NUM_THREADS; ++t_id) do
63             sem_wait(&sem_main);
64         end For
65
66         // 主线程再次唤醒工作线程进入下一轮次的消去任务
67         For (int t_id = 0; t_id < NUM_THREADS; ++t_id) do
68             sem_post(&sem_workerend[t_id]);
69         end For
70
71     end For
72
73     For(int t_id = 0; t_id < NUM_THREADS; t_id++) do
74         pthread_join();
75     end For
76
77     //销毁所有信号量
78     sem_destroy();
79
80
81     return 0;
82 }

```

(3) 静态线程 + 信号量同步版本 + 三重循环全部纳入线程函数

```

1
2 //线程数据结构定义
3 typedef struct {
4     int t_id; //线程 id
5 }threadParam_t;
6
7 //信号量定义
8 sem_t sem_leader

```

```

9  sem_t sem_Division[NUM_THREADS-1];
10 sem_t sem_Elimination[NUM_THREADS-1];
11
12 //线程函数定义
13 void *threadFunc(void *param) {
14     threadParam_t *p = (threadParam_t*)param;
15     int t_id = p -> t_id;
16
17     For (int k = 0; k < n; ++k) do
18         // t_id 为 0 的线程做除法操作，其它工作线程先等待
19         // 这里只采用了一个工作线程负责除法操作，同学们可以尝试采用多个工作线程完成除法操作
20         // 比信号量更简洁的同步方式是使用 barrier
21         if (t_id == 0)
22             For (int j = k+1; j < n; ++j) do
23                 A[k][j] = A[k][j] / A[k][k];
24             end For
25             A[k][k] = 1.0;
26         else
27             sem_wait(&sem_Division[t_id-1]); // 阻塞，等待完成除法操作
28         end if
29
30         // t_id 为 0 的线程唤醒其它工作线程，进行消去操作
31         if (t_id == 0)
32             For (int i = 0; i < NUM_THREADS-1; ++i) do
33                 sem_post(&sem_Division[i]);
34             end For
35         end if
36
37         //循环划分任务（同学们可以尝试多种任务划分方式）
38         For(int i=k+1+t_id; i < n; i += NUM_THREADS) do
39             //消去
40             For (int j = k + 1; j < n; ++j) do
41                 A[i][j] = A[i][j] -A[i][k] * A[k][j];
42             end For;
43             A[i][k]=0.0;
44         end For;
45
46
47         if(t_id == 0)
48             For (int i = 0; i < NUM_THREADS-1; ++i) do
49                 sem_wait(&sem_leader); // 等待其它 worker 完成消去
50             end For
51
52             For (int i = 0; i < NUM_THREADS-1; ++i) do
53                 sem_post(&sem_Elimination[i]); // 通知其它 worker 进入下一轮
54             end For

```

```

55         else
56             sem_post(&sem_leader); // 通知 leader, 已完成消灭任务
57             sem_wait(&sem_Elimination[t_id-1]); // 等待通知, 进入下一轮
58         end if
59     end For;
60     pthread_exit(NULL);
61 }
62
63 int main() {
64     // 初始化信号量
65     sem_init(&sem_leader, 0, 0)
66     For (int i = 0; i < NUM_THREADS-1; ++i) do
67         sem_init(&sem_Division, 0, 0);
68         sem_init(&sem_Elimination, 0, 0);
69     end For
70
71     // 创建线程
72     pthread_t handles[NUM_THREADS]; // 创建对应的 Handle
73     threadParam_t param[NUM_THREADS]; // 创建对应的线程数据结构
74     For(int t_id = 0; t_id < NUM_THREADS; t_id++)
75         param[t_id].t_id = t_id;
76         pthread_create();
77     end For
78
79     For(int t_id = 0; t_id < NUM_THREADS; t_id++) do
80         pthread_join();
81     end For
82
83     // 销毁所有信号量
84     sem_destroy();
85
86
87     return 0;
88 }

```

(4) 静态线程 + barrier 同步

```

1 // 线程数据结构定义
2
3 typedef struct {
4     int t_id; // 线程 id
5 } threadParam_t;
6
7 // barrier 定义
8 pthread_barrier_t barrier_Division;
9 pthread_barrier_t barrier_Elimination;
10

```

```

11 //线程函数定义
12 void *threadFunc(void *param) {
13     threadParam_t *p = (threadParam_t*)param;
14     int t_id = p -> t_id;
15
16     For (int k = 0; k < n; ++k) do
17         // t_id 为 0 的线程做除法操作，其它工作线程先等待
18         // 这里只采用了一个工作线程负责除法操作，同学们可以尝试采用多个工作线程完成除法操作
19         if(t_id == 0)
20             For (int j = k+1; j < n; ++j) do
21                 A[k][j] = A[k][j] / A[k][k];
22             end For
23             A[k][k] = 1.0;
24         end if
25
26         //第一个同步点
27         pthread_barrier_wait(&barrier_Division);
28
29         //循环划分任务（同学们可以尝试多种任务划分方式）
30         For(int i=k+1+t_id; i < n; i += NUM_THREADS) do
31             //消去
32             For (int j = k + 1; j < n; ++j) do
33                 A[i][j] = A[i][j] - A[i][k] * A[k][j];
34             end For;
35             A[i][k]=0.0;
36         end For;
37
38         // 第二个同步点
39         pthread_barrier_wait(&barrier_Elimination);
40
41     end For;
42     pthread_exit(NULL);
43 }
44
45 int main() {
46     //初始化 barrier
47     pthread_barrier_init(&barrier_Division, NULL, NUM_THREADS);
48     pthread_barrier_init(&barrier_Elimination, NULL, NUM_THREADS);
49
50     //创建线程
51     pthread_t handles[NUM_THREADS]; // 创建对应的 Handle
52     threadParam_t param[NUM_THREADS]; // 创建对应的线程数据结构
53     For(int t_id = 0; t_id < NUM_THREADS; t_id++)
54         param[t_id].t_id = t_id;
55         pthread_create();
56     end For

```

```
57  
58     For(int t_id = 0; t_id < NUM_THREADS; t_id++) do  
59         pthread_join();  
60     end For  
61  
62     //销毁所有的 barrier  
63     pthread_barrier_destroy();  
64  
65     return 0;  
66 }
```

2.5 编译、运行

参见 SIMD 编程实验教学指导书。注意：在鲲鹏服务器平台提交任务时，每个任务只能申请 1 个 CPU（8 核），这个核心执行你的实验程序，建议启动 8 个线程进行多线程实验（最佳方式是最多启动 7 个子线程，主线程在创建完工作线程后，自身也作为一个工作线程承担一部分计算任务，这样一共 8 个线程），线程数量并不是越多越好。

3 ANN 选题：多线程实验

3.1 IVF (Inverted File)

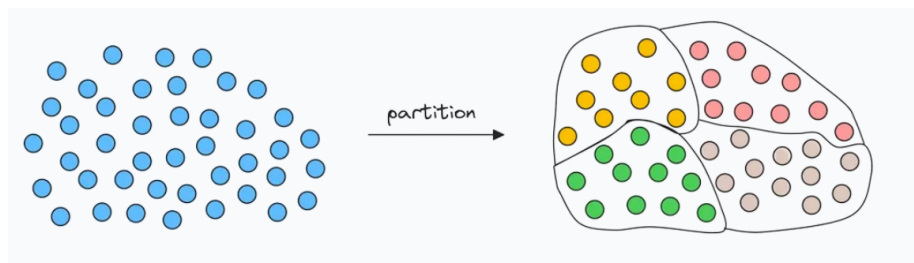


图 3.1: IVF 索引构建

IVF 的索引构建如图3.1所示，只需要使用任意一种聚类算法对 base data 进行聚类，并保存每个簇的质心和该簇中点的编号集合（簇的个数通常根据 base data 的大小决定，一般设置为 64 到 4096）。

IVF 的查询过程步骤一如图3.2和3.3所示，首先计算查询点到每个簇的距离，并选出距离查询点最近的前 nprobe 个簇，然后只计算这些簇中点到

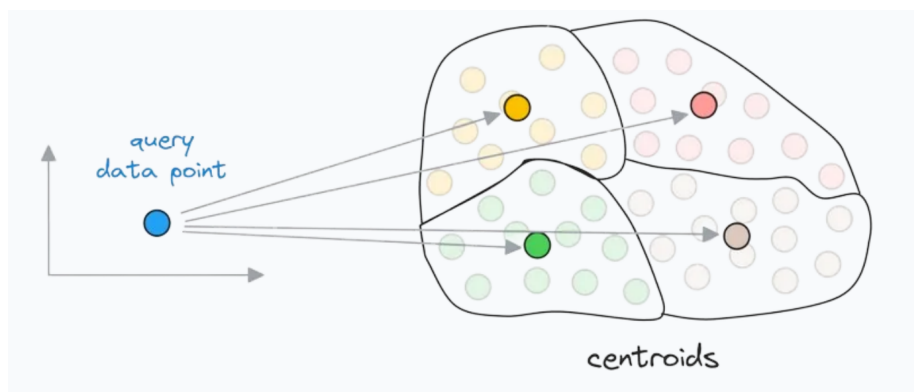


图 3.2: IVF 查询过程 I

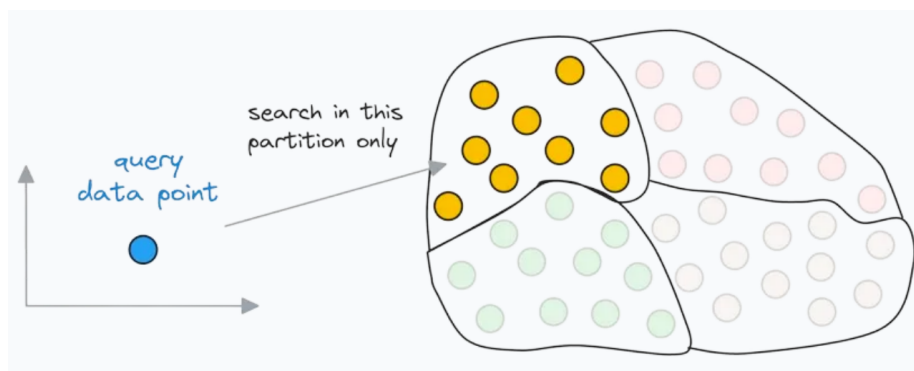


图 3.3: IVF 查询过程 II

查询的距离并进行排序。 $nprobe$ 为搜索时的参数，更高的 $nprobe$ 意味着更好的准确度，但也会导致延迟更高，实验结果中需要调整 $nprobe$ 的值来绘制算法的准确度-延迟曲线。对于 IVF 的并行优化，可以将前 $nprobe$ 个簇划分到不同线程中去计算，在第一步计算查询点到每个簇的距离时同样可以使用多线程计算。

在 IVF 的索引构建中，还可以对内存进行重排，将相同簇的向量存储在一起，降低查询时的 cache miss 率。

在实验中同样可以考虑将 IVF 和 PQ 结合，一般有两种方法：(1) 先对所有 base data 进行 PQ，再构建 IVF 索引。(2) 先构建 IVF 索引，再在每个簇中分别进行 PQ。两者的性能会有一些差异，同学们如果选择实现 IVF-PQ 算法的话可以尝试对该现象进行分析。

3.2 HNSW (Hierarchical Navigable Small World graphs)

对于 HNSW 图索引的并行实现，同学们可以直接在 hnsplib 的基础上进行开发，hnsplib 已经整理到了服务器的框架中并提供了索引构建的示例，这里便不再赘述。

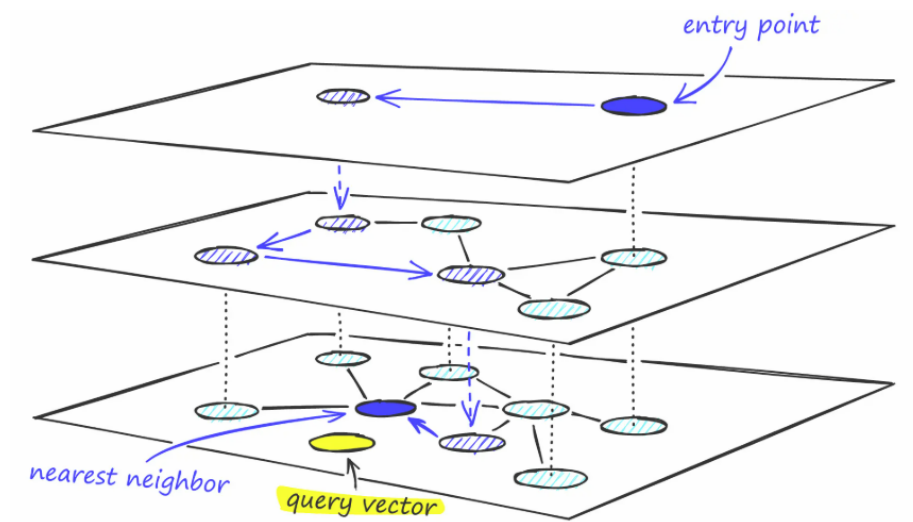


图 3.4: HNSW 查询过程

Algorithm 1 Greedy search (graph G , query q , entry point ep , ef)

```

1:  $pq$  = a priority queue with unlimited capacity, initialized with  $ep$ 
2:  $H$  = a max-heap with capacity  $ef$ 
3: while  $pq$  is not empty do
4:    $d_{v_c}, v_c$  = pop an element from  $pq$ 
5:    $d_{v_{top}}, v_{top}$  = the heap top of  $H$ 
6:   if  $d_{v_c} > d_{v_{top}}$  then
7:     break
8:   for each neighbor  $v$  of  $v_c$  which has not been accessed do
9:     if  $D(v, q) < d_{v_{top}}$  then
10:      Insert  $(D(v, q), v)$  into  $pq$  and  $H$ 
11:      mark  $v$  as accessed
12:   resize  $H$  to be  $ef$ 
13: return  $k$  smallest elements in  $H$ 

```

图 3.5: HNSW 查询过程伪代码

对于图索引上的查询过程，如图3.4所示，首先从 HNSW 的最顶层开

始,在该层中找到距离查询最近的点,然后在下一层进行搜索时,从上一层离查询最近的点开始搜索。当搜索到最底层时,再使用图3.5所示的算法进行搜索,该算法通过维护两个优先队列 pq 和 H ,优先队列 pq 表示候选队列, H 表示大小为 ef 的结果队列。在每次迭代中,选择 pq 中距离查询最近的点 v_c ,并探索点 v_c 的所有邻居。当算法找不到距离查询更近的点时,便停止搜索。

尽管 HNSW 采用了很符合直觉的层次结构来加速搜索,但是目前最新的研究表明 HNSW 的层次结构在高维空间中作用有限¹。所以同学们可以只使用 HNSW 的底层图并随机选择入口点,并且可以只关注底层图搜索算法的并行化。

对于图索引上查询内的并行化,实际上是一个非常困难的问题,目前学术界也只有很少的解决方案²。但是同学们仍然可以简单探索一些方法,例如(1)将搜索时探索一个点的邻居这一过程并行化;(2)采用 IVF+HNSW 的嵌套结构,建立多个 HNSW,每个线程负责搜索一部分 HNSW。上述的优化很可能导致负优化,同学们只要如实汇报实验结果并给出一定分析即可。

4 口令猜测选题:多线程实验

4.1 口令猜测算法的并行化

PCFG 算法的具体原理,以及其并行化过程,请参照学期初发给大家的“口令猜测算法并行化选题.pdf”,此处不再赘述。这里主要结合框架代码给大家介绍需要进行并行化的内容。

如图4.6、4.7所示,这里的两个循环,在本质上是给一个 PT 的最后一个 segment 进行具体的填充。例如,对于 L_8D_2 这个 PT 而言,在先前优先队列的初始化和不断迭代过程中,已经将其中的 L_8 进行实例化了,只需要在这里将 D_2 进行填充即可。这时,假设模型统计到了 12、11、23 这三个具体的值,那么这个循环就会逐一将这三个值填充到 D_2 里面,形成三个

¹实际上,HNSW 真正有效的部分在于底层图构建时使用三角不等式进行裁边,但是在论文中仅仅一笔代过,而没有给出详细讨论。如果对图索引构建理论感兴趣的同学可以参考论文Fast Approximate Nearest Neighbor Search With The Navigating Spreading-out Graph

²感兴趣的同学可以阅读论文iQAN: Fast and Accurate Vector Search with Efficient Intra-Query Parallelism on Multi-Core Architectures,但由于论文难度较大,这里并不要求同学们复现该论文,如果阅读后理解了该论文可以尝试把论文中的一些方法用到自己的并行实现中。

新的口令。

显而易见的是,这个过程是可以并行化的。你需要通过 pthread/OpenMP 将这个循环进行并行化,同时改变头文件、main 函数等文件,使不同线程的返回结果能够存起来,并且按照 main 函数里面的方式进行哈希和清理。

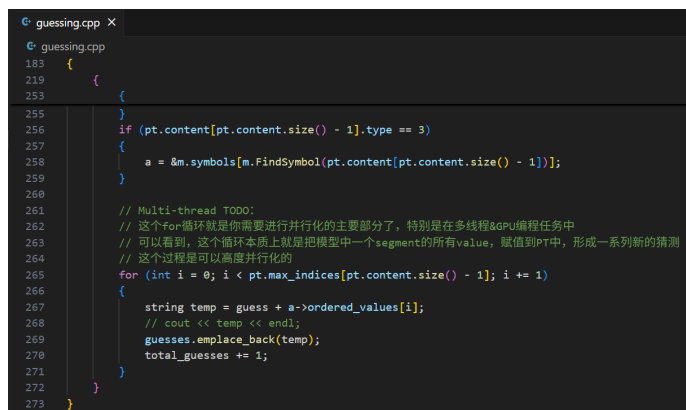


```

209 {
210     for (int i = 0; i < pt.max_indices[0]; i += 1)
211     {
212         string guess = a->ordered_values[i];
213         // cout << guess << endl;
214         guesses.emplace_back(guess);
215         total_guesses += 1;
216     }
217 }

```

图 4.6: 多线程并行化, guessing.cpp 文件, 第一处需要并行化的位置



```

265 for (int i = 0; i < pt.max_indices[pt.content.size() - 1]; i += 1)
266 {
267     string temp = guess + a->ordered_values[i];
268     // cout << temp << endl;
269     guesses.emplace_back(temp);
270     total_guesses += 1;
271 }
272 }
273 }

```

图 4.7: 多线程并行化, guessing.cpp 文件, 第二处需要并行化的位置

4.2 基础要求

你只需要将上述过程用 pthread 和 OpenMP 进行多线程并行化即可。需要注意的是,你应当保证这个过程“相对正确”。口令猜测并行化会在一定程度上牺牲口令概率的“颗粒度”(因为重新将同一批并行生成的口令按概率排序,效率太低),但总体上来讲口令的生成结果是基本不会有差别的。

和 SIMD 作业中哈希值的正确性不同，口令猜测很难直接检测“正确性”，实际操作中往往用相当长一段时间内的口令攻破成功率来评估口令猜测的并行算法。但是，由于服务器的计算资源相对有限，暂时不能支持千万级别的口令结果存储，所以你可以采用一个简单的方法来测试你并行生成的口令是否正确。你可以将飞书群里面的 `correctness_guess.cpp` 文件上传到服务器的 `guess` 目录下，将这个文件代替 `main` 函数进行编译。编译并运行之后，你会发现输出的信息多了一行“Cracked”，这就是猜测算法的攻破率了。你的并行猜测算法的攻破率应当和串行猜测算法的攻破率大致相同。

4.3 进阶要求

本次作业的进阶要求如下：

- 用 `pthread` 和 `OpenMP` 同时实现多线程的猜测并行化，并且对比二者性能上的差别，根据你所学的知识解释这种差别背后的原因。至少在 `OpenMP` 和 `pthread` 一种多线程编程上，实现猜测相对并行算法的加速。
- 我们回忆一下上次 SIMD 作业：不同时使用编译优化的时候，是很难实现 SIMD MD5 相对串行哈希的加速的。这次实验里面，如果你在多线程并行化猜测的基础上，同时采用 SIMD 对哈希过程进行加速，编译优化对二者的影响是否存在区别？尝试给出实验分析。
- 在上一个问题的基础上，如果你发现编译优化对二者的影响确实存在区别，那么你很可能不能实现多线程猜测和 SIMD 哈希的同时加速 (i.e., 猜测能加速的时候，哈希是负优化；哈希能加速的时候，猜测是负优化)。尝试优化你的多线程猜测，实现多线程猜测和 SIMD 哈希的同时加速。
- 一个并行设计中的常用思想是：并行一定会付出额外的代价，**并行化之后总的资源消耗往往是大于串行程序的**，并行化的过程实际上是充分利用串行程序所不能充分利用的设备资源。并行化适合处理什么样的问题？什么时候并行化更合适，什么时候串行更合适？思考上述问题，并尝试用你的思考和探索，优化多线程的口令猜测并行算法。

如果你完成了其它的工作，但并不在上述列表里，也同样可以按照工作量和难度进行相应给分

一些实验小提示：

- 你可以尝试调整生成的总猜测数，并探索加速比随总猜测数的变化。
- 你可以尝试调整使用的总线程数，并探索加速比随线程数的变化。
- 可供参考的论文之一：https://github.com/Ming-Xu-research/Ming-Xu-research.github.io/blob/master/_data/Parallel_PCFG_TDSC.pdf

5 NTT 选题：多线程实验

本次多线程实验分为三个方案，朴素算法，四分算法和 CRT 算法，朴素算法为基础要求，后两个为进阶要求，如果可以实现 CRT 算法，就不必再实现四分算法，但必须实现朴素算法，最终保障 CRT 算法得分不低于四分算法得分。

NTT 选题要求在本次实验不允许使用 SIMD 优化，但你可以把多线程和向量化的结合放在期末报告中。

5.1 朴素多线程优化

对于基础版的 NTT 多线程优化，实现方法较为简单，由于第二三层循环相当于遍历了一遍多项式数组，显然可以对第三层循环进行多线程优化，类似高斯消元，因此代码实现较为简单，只需要注意线程同步正确。

```
1
2 for(int mid = 1; mid < limit; mid <= 1) {
3     for(int j = 0; j < limit; j += (mid <= 1)) {
4         int w = 1; // 旋转因子
5         for(int k = 0; k < mid; k++, w = w * Wn) { // 对这层循环进行优化
6             // 运算主体
7         }
8     }
9 }
```

5.2 多分多线程优化

即使在 SIMD 实验中未实现四分的 NTT，也可以在多线程这里开始。

对于非二分的 NTT, 需要判断多项式长度, 如果多项式长度不等于 4^n (即多项式长度等于 2^{2k+1}), 由于每次枚举的步长乘积等于 4, 需要预处理第一层或最后一层的循环后, 才能进行四分。

如果线程数等于 4, 每个线程负责对应需要归并的四块数据的每一块。

5.3 CRT 多线程优化

提供一个全新的多线程优化思路, 在 SIMD 的实验指导书中提到了大模数的 NTT 方法, 其本质即任意模数 NTT, 如果你是在洛谷或 oiwiki 中学习的 NTT 基础知识, 也能看到任意模数 NTT 的原理, 即使用中国剩余定理 (CRT) 合并大模数。

对于任意模数 NTT 的模板, 通常采用三模数 NTT 合并, CRT 多线程优化的思想类似任意模数 NTT, 如果让每一个线程都使用不同的小模数, 最终使用 CRT 将结果合并。

如果要想实现此多线程优化, 需要在实现多模数 NTT 合并后才能实现 pthread 优化, 当模数越大时, 多线程的优化效果越明显。

你可以仿照任意模数 NTT (三模数合并) 的模板仿推出任意模数 NTT (多模数合并) 的公式, 也可以自己上网寻找相关博客, 另外以下给出了两篇应用了 CRT 合并 NTT 以加速大模数的同态加密论文, 有更详细的原理和优化的证明, 当然 CRT 合并 NTT 的重要性在超大模数 (往往远大于 64 bit) 时才明显, 对于同态加密库 (CKKS 等), 会经常使用超大模数进行加密, 因此如果实现了本优化, 要求最终需要测试一个大于 32 bit 的模数。

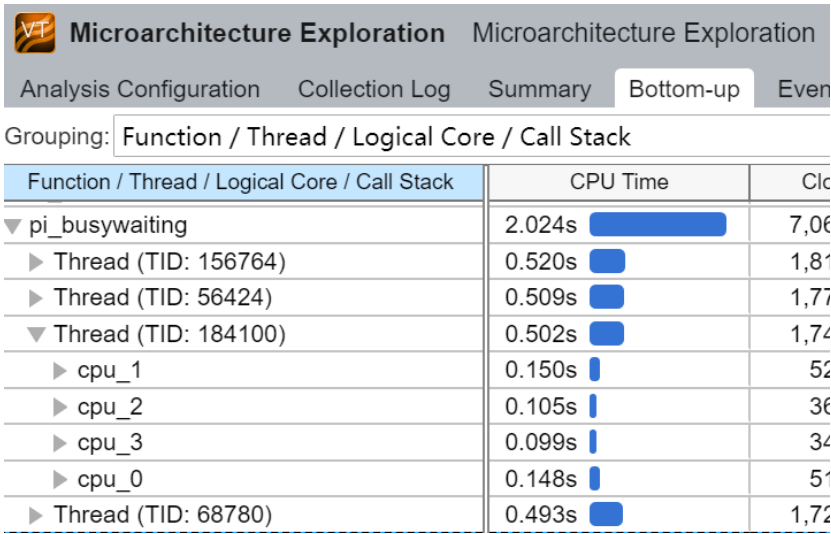
- A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes
- A Full RNS Variant of Approximate Homomorphic Encryption
- Approximate CRT-Based Gadget Decomposition and Application to TFHE Blind Rotation

有关多线程优化 NTT 的论文还有很多, 可以自行搜索。

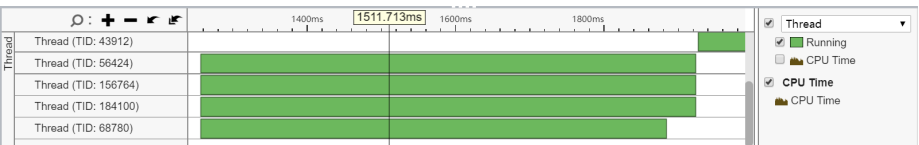
6 VTune 分析 (本地环境)

大家可以在 Intel 官网找到 VTune 的安装和使用教程。

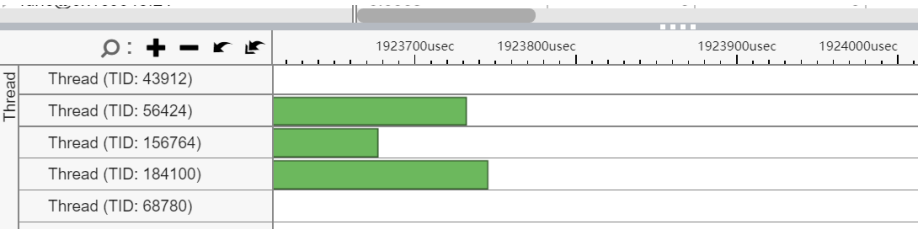
接下来简单举例说明如何使用 Vtune profiling 工具分析算法过程中的同步开销和空闲等待等问题。对于 Microarchitecture Exploration, 可以选择含有 thread 的 Grouping, 比如 Function/thread/Logical Core/Call Stack。我们就可以看到每个线程的 id 及其运行信息, 以忙等待方法为例如下图所示:



然后可以根据线程 id 找到各线程的运行时间以及对应事件状态:



很明显可以看出 68780 线程最先执行结束, 为代码中的 0 号线程, 其余线程结束时间十分接近, 进行放大:



可以看出 156764 线程第二个结束为 1 号线程; 56424 线程第三个线程

为 2 号线程，184100 线程最后一个结束为 3 号线程。了解线程顺序后可以查看到其他信息进行分析，比如 Sleep_Ex 函数，该函数用于中止当前线程等待恢复运行：

Grouping: **Function / Thread / Logical Core / Call Stack**

| Function / Thread / Logical Core / Call Stack | CPU Time | Clockticks | Instructions Retired | CF |
|---|----------|------------|----------------------|----|
| ▶ RtlGetCurrentUmsThread | 0.001s | 5,400,000 | 540,000 | |
| ▶ func@0x1401c93e0 | 0.003s | 7,020,000 | 1,080,000 | |
| ▼ SleepEx | 0.004s | 12,420,000 | 3,780,000 | |
| ▶ Thread (TID: 184100) | 0.001s | 3,780,000 | 3,780,000 | |
| ▶ Thread (TID: 56424) | 0.003s | 8,640,000 | 0 | |

可以看出 2 号线程和 3 号线程执行过该函数，结合 0 号线程相比其他三个线程很早就完成的情况，我们知道 1 号线程运行得很慢，由此造成了提早完成任务的 2 号线程和 3 号线程的空闲等待。这也是忙等待版本 take turn 相比于互斥量版本的缺陷。在线程数多的时候或是线程执行能力差异巨大的情况下尤为明显。

对于 CPI，指令数，Cache 命中等其他功能的分析参考体系结构相关及性能测试实验指导书，不再赘述。