南開大學
Nankai University

计算机学院

并行程序设计实验报告

实验四：多线程编程（口令猜测）

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 5 月 23 日

目录

1 问题重述

PCFG（Probabilistic Context-Free Grammar，概率上下文无关文法）口令猜测是一种基于语言建模原理的密码猜测攻击技术，它结合了统计学习和文法规则，用于高效生成可能的密码候选列表，优于传统暴力破解或字典攻击。其核心思想是：利用密码样本学习密码结构和模式的概率分布，构建一个概率文法模型，再据此生成猜测结果，按概率从高到低进行尝试。

在理解了实验指导书上的内容之后，我们可以归纳出其基本需要实现以下几个过程：

1. 分析训练集：首先会对训练集进行分析，提取一些密码的结构并计算概率，比如 L8D3S3 这种，代表这个密码由 8 个字母 +3 个数字 +3 个符号组成，并且会计算出其在训练集中出现的概率。

2. 建立概率模型：为每种模式生成一个文法规则，并为每种模式及其组成部分（字母、数字、符号）计算出现概率。也就是计算某个密码出现的概率，分为上一过程提到的模式概率和组件概率（也就是向其中填入真正的字符信息）。比如说 L8D1S1 的概率是 0.2，在 L8 中，12345678 出现的概率为 0.5，在 D1 中 a 出现的概率为 0.1，在 S1 中！出现的概率为 0.1，那么密码 12345678a！的总概率就为 0.5*0.2*0.1*0.1（条件概率）。

3. 生成口令候选：根据概率文法生成新的口令，优先生成高概率组合。

4. 猜测过程：生成的口令候选从高概率到低概率依次尝试，大幅减少猜测空间，提高成功率。

而我们本次多线程编程实验，就需要重点研究后三个过程，来修改我们的 guessing.cpp，以实现相对串行的猜测性能优化。另外，PT 指的是一种拆分结构（如 L6S1），其中的每个部分叫 Segment。

2 串行算法分析

2.1 串行算法运行测试

编译程序并提交到服务器进行运行后，返回结果，可以看到程序正常运行并返回结果，在此基础上，我们开始之后的实验，以实现 guessing 模块的多线程并行优化。

```
101   // // g++ main.cpp train.cpp guessing.cpp md5.cpp -o main
```

问题 5　输出　调试控制台　终端　端口

```
Guesses generated: 8382935
Guesses generated: 8483085
Guesses generated: 8627212
Guesses generated: 8807999
Guesses generated: 8946318
Guesses generated: 9139387
Guesses generated: 9239569
Guesses generated: 9383796
Guesses generated: 9528822
Guesses generated: 9699507
Guesses generated: 9853408
Guesses generated: 10106852
Guess time:0.637762seconds
Hash time:2.96722seconds
Train time:25.575seconds

Authorized users only. All activities may be monitored and reported.
/home/s2312631/files: No such file or directory
```

图 2.1: Enter Caption

2.2 串行算法详细分析

2.2.1 整体流程

```
init() → 构建初始结构 → 插入优先队列（q）
    ↓
while q not empty:
    PopNext()
        ↓
    Generate(pt)        → 输出猜测字符串
        ↓
    NewPTs()            → 延展结构
        ↓
    CalProb() for each → 插入优先队列
```

2.2.2 CalProb() 伪代码：计算某个具体结构（PT）的生成概率

```
function CalProb(pt):
    pt.prob = pt.preterm_prob    // 初始化为PT本身的生成概率
    index = 0
    for idx in pt.curr_indices:
        seg = pt.content[index]
        if seg.type == letter:
            pt.prob *= get_letter_freq(seg, idx) / get_letter_total(seg)
        else if seg.type == digit:
            pt.prob *= get_digit_freq(seg, idx) / get_digit_total(seg)
        else if seg.type == symbol:
            pt.prob *= get_symbol_freq(seg, idx) / get_symbol_total(seg)
        index += 1
```

这段代码比较好理解，就是传入一个带有实际数值的 pt，计算结构概率 × 各 Segment 实际值的概率。pt.preterm_prob 指的是 PT 结构（如 L6S1）的概率（从训练集统计）。

2.2.3 init() 伪代码：初始化优先队列，将所有 PT 按结构概率加入队列

```
function init():
    for pt in all_ordered_PTs:
        pt.max_indices = []
        for seg in pt.content:
            if seg.type == letter:
                pt.max_indices.append(letter_value_count(seg))
            else if seg.type == digit:
                pt.max_indices.append(digit_value_count(seg))
            else if seg.type == symbol:
                pt.max_indices.append(symbol_value_count(seg))
        pt.preterm_prob = get_preterm_freq(pt) / total_preterm_freq
        CalProb(pt)
        priority_queue.append(pt)
```

这一段代码也相对比较好理解，就是遍历所有 pt 并通过调用 CalProb 函数计算每个 pt 结构的概率，并

加入到优先队列中，按照概率降序进行排序。每个结构中 segment 的最大可能数记录在 max_indices。

2.2.4 PopNext() 伪代码：弹出优先队列中概率最高的 PT 并扩展下一批 PT

```
function PopNext():
    best_pt = priority_queue.front()
    Generate(best_pt)
    new_pts = best_pt.NewPTs()
    for pt in new_pts:
        CalProb(pt)
        insert pt into priority_queue based on its probability (descending order)
    remove best_pt from priority_queue
```

这一部分也比较好理解，每一轮都只处理最优结构，确保穷举顺序是最优猜测顺序。首先拿出队首元素（也就是当前概率最大的 pt），通过调用 Generate() 函数，可以生成实际的猜测字符串，在后面对 Generate() 函数的分析再详细说明。然后就是传入概率最大的 pt，调用 NewPTs() 函数，获取一系列新的 pt，再对于获得的新的 pt，调用 CalProb(pt) 计算其概率，并加入到优先队列中，按照概率降序进行排序。

2.2.5 PT::NewPTs() 伪代码：生成从当前 PT 扩展出的所有新 PT

```
function NewPTs():
    if PT has only one segment:
        return empty_list
    result = []
    original_pivot = pivot
    for i from pivot to (curr_indices.size - 2):
        curr_indices[i] += 1
        if curr_indices[i] < max_indices[i]:
            pivot = i
            result.append(copy of current PT)
        curr_indices[i] -= 1  // restore
    pivot = original_pivot
    return result
```

这一部分通过修改给定 pt，一次只更改一个 segment，除去了最后一个 segment，生成一系列变体。在这一部分不会修改最后一个 segment（因为这部分在 Generate 中完整枚举，需要并行化）。

2.2.6 Generate() 伪代码：根据某个 PT，生成实际的猜测字符串

```
function Generate(pt):
    CalProb(pt)
    if pt has only one segment:
        seg = pt.content[0]
        value_list = get_values(seg)
        for value in value_list:
            guesses.append(value)
```

```
            total_guesses += 1
    else:
        guess_prefix = ""
        for i from 0 to pt.content.size - 2:
            seg = pt.content[i]
            idx = pt.curr_indices[i]
            guess_prefix += get_value_by_index(seg, idx)
        last_seg = pt.content.last()
        value_list = get_values(last_seg)
        for value in value_list:
            guess = guess_prefix + value
            guesses.append(guess)
            total_guesses += 1
```

这一段是我们需要进行并行化的核心部分。在这一部分，首先计算传入 pt 的概率（除了最后一个字段没计算）。对于只有一个 segment 的 PT，直接遍历生成其中的所有 value 即可。否则需要先获得前几个 segment，再把模型中最后一个 segment 的所有 value，赋值到 PT 中，形成一系列新的猜测。我们的任务就是完成对这两个 for 循环的并行化操作。

3 并行算法设计

3.1 并行算法思路

思考这个问题，我们需要先思考，为什么这两个 for 可以进行并行化？根本原因是因为这些数据是并行的，没有相互依赖，并且这个过程是一个读多写少的过程，不涉及全局状态改变，可以通过加锁操作，不会导致数据冲突。

我们并行化的思路，就是分配多个线程，把这个任务量比较大的任务细化，分配给多个线程，每个线程处理一部分数据，并在最后总和各个线程完成的任务。

3.2 并行算法具体实现

3.2.1 pthread 动态线程

首先定义了一个线程结构体，存储线程参数。

- start，end 指的是本线程要处理的 value 索引区间 [start, end)；
- prefix 指的是串联到每个 value 前面的那段字符串（只有一个 segment 的情况下就为空，多 segment 情况下就是前缀 guess）；
- vector<string>* values 是指向 a->ordered_values 的指针，线程据此取实际的 value；
- vector<string>* guesses_out 是指向全局 guesses 容器的指针，各线程往这里写结果；
- pthread_mutex_t* mutex; 是保护 guesses_out 和 total_guesses 的互斥锁；
- int* total_guesses 是指向全局的计数器，每生成一个 guess 就自增。

```
struct ThreadArgs {
    int start, end;
    string prefix;
    vector<string>* values;
    vector<string>* guesses_out;
    pthread_mutex_t* mutex;
    int* total_guesses;
};
```

GenerateGuesses 函数用于在多线程并行环境下生成密码猜测字符串，在 Generate_pthread 这部分里会进行调用，进行猜测，就是代替原来的猜测逻辑，封装成函数，方便多线程并行编程。本质上就是把前缀猜测与最后一个字段进行拼接。

- args->start, args->end：会限定该线程处理的 value 索引范围，避免多线程重复劳动。
- args->prefix：如果是多个 segment 密码，就先拼好除了最后一段之外的那串前缀；如果只有一个 segment，这里就是空字符串。
- *(args->values)：解引用指针，获取到 vector<string>，然后再取 [i]，就是取最后一个 segment 的值。
- **加锁/解锁：**
- pthread_mutex_lock：上锁，保证这一段对 guesses_out 容器的 emplace_back 调用和对 total_guesses 的修改，是一个原子操作，不会和别的线程冲突。
- pthread_mutex_unlock：释锁，其它线程才能进来写。

```
void* GenerateGuesses(void* arg) {
    ThreadArgs* args = (ThreadArgs*)arg;
    for (int i = args->start; i < args->end; ++i) {
        string guess = args->prefix + (*(args->values))[i];

        pthread_mutex_lock(args->mutex);
        args->guesses_out->emplace_back(guess);
        (*(args->total_guesses))++;
        pthread_mutex_unlock(args->mutex);
    }
    return nullptr;
}
```

在 Generate_pthread 函数中，对于多个 segment 和单个 segment 的操作类似，我们只粘出一段，可以看到基本逻辑还是和串行的一样。首先我们设置了 8 个线程，通过计算，把 total（总 value 数）平均分配给 8 个线程。然后填写了线程参数存储在 args 中。需要注意的是，很有可能某个线程处理的数据量小于平均值，所以我们需要额外处理，args[t].start = t*chunk，args[t].end = min((t+1)*chunk, total)，最后一个线程可能“凑不满”整块，用 min 截断到 total。然后要通过 pthread_create 创建线程，其四个参数分别代表线程句柄数组元素、线程属性（用 nullptr 表示默认）、线程函数指针 GenerateGuesses、传入

的参数结构体地址。接着这个 for 循环要等待每个线程结束，不然主线程可能在线程还没写完 guesses 就继续往下跑。最后记得销毁之前创建的互斥锁，释放占用的资源。

```
void PriorityQueue::Generate_pthread(PT pt)
.............
        {
            int num_threads = 8;
            int total = pt.max_indices[pt.content.size() - 1];
            int chunk = (total + num_threads - 1) / num_threads;
            pthread_t threads[num_threads];
            ThreadArgs args[num_threads];
            pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
            for (int t = 0; t < num_threads; ++t) {
                args[t] = {
                    t * chunk,
                    min((t + 1) * chunk, total),
                    guess,
                    &a->ordered_values,
                    &guesses,
                    &mutex,
                    &total_guesses
                };
                pthread_create(&threads[t], nullptr, GenerateGuesses, &args[t]);
            }
            for (int t = 0; t < num_threads; ++t) {
                pthread_join(threads[t], nullptr);
            }
            pthread_mutex_destroy(&mutex);
        }
```

但是当我们利用这样的算法运行之后，发现虽并行正确性达到了验证，但是时间性能上却变成了负优化。运行时间慢且不稳定，平均达到了 0.8s，说明我们的并行算法中的某些操作消耗的时间远远多于并行优化的时间。而这个操作一方面是进行加锁的操作导致的。

3.2.2 pthread 动态线程优化

为了解决这个问题，我们需要知道为什么之前的程序需要加锁。在之前的版本中，我们用到了两个共享变量 guesses_out 和 total_guesses，也就是说每个线程需要同时往同一个容器里写入数据，对同一个变量进行自增操作，这就需要加锁来保证程序的正确性。我们意图改变这种数据结构，各个线程进行自己的工作并保存在独立的一片内存里，这样就不需要加锁，大大减少了开销。

所以，在线程参数，我们修改原来的共享变量为独立的线程变量，results 中可以用于保存本线程的结果，count 用于保存该线程得到的结果的数量，不同于之前的容器需要使用 emplace_back 进行尾插，result 这一个大数组可以被多个线程同时访问，只要给定范围，限制访问地址不同。

```
// 线程参数结构
struct ThreadArgs {
    int start;
    int end;
```

```
    string prefix;
    vector<string>* values;
    string* results;
    int* count;
    int* total_count;
};
```

GenerateGuesses 自然要作修改，除了把逻辑改成保存本线程的结果给定范围的 result 数组外，还使用了局部变量接收指针指向的内容，减少频繁的指针解引用。

```
void* GenerateGuesses(void* arg) {
    ThreadArgs* args = static_cast<ThreadArgs*>(arg);

    // 使用局部变量减少指针解引用
    int start = args->start;
    int end = args->end;
    string prefix = args->prefix;
    vector<string>* values = args->values;
    string* results = args->results;

    // 直接写入预分配的结果数组
    for (int i = start; i < end; ++i) {
        results[i] = prefix + (*values)[i];
    }

    // 一次性更新计数，减少原子操作
    *(args->count) = end - start;

    return NULL;
}
```

在 Generate_pthread 函数中，首先我们添加了一个动态分配的逻辑，就是当任务量比较小的时候直接用串行算法执行，否则再用并行，if (total < 100000)，这个数值设置的越小越倾向于做并行操作。这部分核心逻辑基本上没有变，通过设置相关变量，传入 pthread_create 相关的参数、函数、线程数组等，主要变更的就是，在等待所以线程结束后，要把 result 中保存的结果返回给 guesses 中，原因是不同于之前的参数，ThreadArgs 存的是某个线程独立的结果。

```
void PriorityQueue::Generate_pthread(PT pt)
.....................................
        // 提高任务粒度阈值
        if (total < 100000) {
            // 直接使用串行处理
            for (int i = 0; i < total; i++) {
                guesses.push_back(guess + a->ordered_values[i]);
            }
            total_guesses += total;
            return;
        }
        int num_threads = 4;
```

```
        int chunk = (total + num_threads - 1) / num_threads;
        string* results = new string[total];
        int thread_counts[num_threads];
        memset(thread_counts, 0, sizeof(thread_counts));
        pthread_t threads[num_threads];
        ThreadArgs args[num_threads];
        for (int t = 0; t < num_threads; ++t) {
            args[t] = {
                t * chunk,
                min((t + 1) * chunk, total),
                guess,
                &a->ordered_values,
                results,
                &thread_counts[t],
                &total_guesses
            };
            pthread_create(&threads[t], NULL, GenerateGuesses, &args[t]);
        }
        for (int t = 0; t < num_threads; ++t) {
            pthread_join(threads[t], NULL);
        }
        // 批量添加结果 - 优化内存操作
        guesses.reserve(guesses.size() + total);
        for (int i = 0; i < total; ++i) {
            guesses.push_back(std::move(results[i]));
        }
        total_guesses += total;
        delete[] results;
```

这样修改之后，实现了相对串行的优化，可以达到 0.49s。另外，当我查看指导书时，发现对于线程创建，有两种做法，一种是静态线程，一种是动态线程。简要来说就是前者只创建一次线程，只有当程序结束时才会销毁线程，后者会在需要多线程并行计算的时候创建线程，但这样就会遭层多次创销线程，导致在创销线程上花费很多时间。很容易就可以看出，我们之前实现的并行算法是基于动态线程的，所以应该还可以进行优化。

为了解决这个问题，我们采用静态线程编程。

3.2.3 pthread 静态线程

首先这里定义了线程池任务结构体。每个任务会将 prefix + value[i] 拼接形成完整口令，这个结构体会被在线程池中被使用。

```
struct Task {// 线程池任务结构
    int start, end;
    string prefix;
    vector<string>* values;
    string* guesses_out;         // 可以为空，表示不使用共享结果数组
    int* guesses_count;  //结果数组
    int thread_id;               // 线程ID，用于标识写入哪个线程本地结果数组
```

```
    vector<string>* local_results; // 线程本地结果数组
};
```

在这里我们定义了一个线程池类，有一些变量结构，具体代表什么已在以下注释中给出。

```
class ThreadPool {// 线程池类
private:
    vector<pthread_t> workers;  //定义了线程容器
    queue<Task> tasks; //任务列表
    pthread_mutex_t queue_mutex;//互斥锁，用于保护 tasks 队列。
    pthread_cond_t condition;//任务可用的条件变量
    pthread_cond_t completion_cond;  // 任务完成条件变量
    int active_tasks;                // 表示当前正在执行的任务数
        配合stop和任务完成变量使用
    bool stop;//同于判断线程池是否处于终止状态。
```

这里进行线程池的初始化，通过 pthread_create 创建工作线程，并将 this 指针传给静态线程函数 WorkerThread，供其访问成员变量。

```
public:
    ThreadPool(size_t threads) : stop(false), active_tasks(0) {
        pthread_mutex_init(&queue_mutex, NULL);
        pthread_cond_init(&condition, NULL);
        pthread_cond_init(&completion_cond, NULL);  // 初始化完成条件变量

        workers.resize(threads);
        for (size_t i = 0; i < threads; ++i) {
            pthread_create(&workers[i], NULL, &ThreadPool::WorkerThread, this);
        }
    }
```

这里是析构函数，stop = true 的标志会通知所有线程退出，在等待所有线程完成操作之后，会销毁锁等。

```
    ~ThreadPool() {// 析构函数，清理线程池
        {
            pthread_mutex_lock(&queue_mutex);
            stop = true;
            pthread_mutex_unlock(&queue_mutex);
        }
        pthread_cond_broadcast(&condition);

        for (size_t i = 0; i < workers.size(); ++i) {
            pthread_join(workers[i], NULL);
        }

        pthread_mutex_destroy(&queue_mutex);
        pthread_cond_destroy(&condition);
        pthread_cond_destroy(&completion_cond);  // 销毁完成条件变量
```

```
    }
```

这里定义了一个函数 Enqueue，可以将任务压入队列，并唤醒一个等待中的线程处理任务。

```
    void Enqueue(Task task) {     // 添加任务到线程池
        pthread_mutex_lock(&queue_mutex);
        tasks.push(task);
        active_tasks++;  // 增加活跃任务计数
        pthread_mutex_unlock(&queue_mutex);
        pthread_cond_signal(&condition);
    }
```

这里的 WaitAll 函数会等待所有任务全部完成，所有任务完成后才返回。

```
    void WaitAll() {     // 等待所有任务完成
        pthread_mutex_lock(&queue_mutex);
        while (active_tasks > 0 || !tasks.empty()) {
            pthread_cond_wait(&completion_cond, &queue_mutex);
        }
        pthread_mutex_unlock(&queue_mutex);
    }
```

这里是进行口令猜测的核心部分，逻辑和之前相同，每个线程将其负责的 value 拼接上 prefix，写入结果数组 guesses_out 中，会在初始化时被调用于 pthread_create 中，也就会在 Generate_pthread_pool 函数中被调用，当接收到信号 stop=true 时，while 循环会停止，结束线程。

```
private:     // 工作线程函数
    static void* WorkerThread(void* arg) {
        ThreadPool* pool = static_cast<ThreadPool*>(arg);
        while (true) {
            Task task;
            bool got_task = false;
            {
                pthread_mutex_lock(&pool->queue_mutex);
                // 如果队列为空且线程池停止，则退出
                if (pool->stop && pool->tasks.empty()) {
                    pthread_mutex_unlock(&pool->queue_mutex);
                    return NULL;
                }
                // 如果队列不为空，取出一个任务
                if (!pool->tasks.empty()) {
                    task = pool->tasks.front();
                    pool->tasks.pop();
                    got_task = true;
                } else if (!pool->stop) {
                    // 如果队列为空且线程池未停止，等待条件变量
                    pthread_cond_wait(&pool->condition, &pool->queue_mutex);
                    pthread_mutex_unlock(&pool->queue_mutex);
                    continue;
```

```
                } else {
                    pthread_mutex_unlock(&pool->queue_mutex);
                    continue;
                }
                pthread_mutex_unlock(&pool->queue_mutex);
            }
            if (got_task) {      // 执行任务
                int local_count = 0;
                // 根据任务类型选择写入目标
                if (task.local_results != nullptr) {
                    // 写入线程本地结果数组（无锁）
                    for (int i = task.start; i < task.end; ++i) {
                        string guess = task.prefix + (*(task.values))[i];
                        task.local_results->push_back(std::move(guess));
                        local_count++;
                    }
                } else if (task.guesses_out != nullptr) {
                    // 写入共享结果数组（传统方式）
                    for (int i = task.start; i < task.end; ++i) {
                        string guess = task.prefix + (*(task.values))[i];
                        task.guesses_out[i] = std::move(guess);
                        local_count++;
                    }
                }
                // 更新线程局部计数并减少活跃任务计数
                pthread_mutex_lock(&pool->queue_mutex);
                *(task.guesses_count) = local_count;
                pool->active_tasks--;
                // 如果没有活跃任务，发送完成信号
                if (pool->active_tasks == 0 && pool->tasks.empty()) {
                    pthread_cond_signal(&pool->completion_cond);
                }
                pthread_mutex_unlock(&pool->queue_mutex);
            }
        }
        return NULL;
    }
};
```

这个函数用于初始化全局线程池，会在 main.cpp 中被调用，以创建静态线程。

```
// 全局线程池，在程序启动时创建
ThreadPool* global_thread_pool = nullptr;
// 初始化全局线程池
void InitThreadPool(int num_threads) {
    if (global_thread_pool == nullptr) {
        global_thread_pool = new ThreadPool(num_threads);
    }
}
```

同样，在主程序要结束时，会调用这个函数销毁线程。

```
// 清理全局线程池
void CleanupThreadPool() {
    if (global_thread_pool != nullptr) {
        delete global_thread_pool;
        global_thread_pool = nullptr;
    }
}
```

需要在 PCFG.h 中添加一下这两个声明语句，以便于 main 函数可以进行线程池的创建与销毁。

```
void InitThreadPool(int num_threads);
void CleanupThreadPool();
```

Generate_pthread_pool 函数跟 Generate_pthread 逻辑差不多，我们在这里不再进行展开分析，主要就是把函数调用改成了调用 Enqueue 函数。这样修改之后可以优化到 0.45s 左右（O2 编译 1000w 数据）。

3.2.4 openmp 并行

实现逻辑比较简单，我们的确实现了优化，在 O2 优化下 1000w 数据的 guess_time 约为 0.48s，代码链接：保存在 guessing_optimize.cpp 中

```
        #pragma omp parallel
        {
            std::vector<std::string> local_guesses;

            #pragma omp for nowait
            for (int i = 0; i < pt.max_indices[0]; i++)
            {
                string guess = a->ordered_values[i];
                local_guesses.emplace_back(guess);
            }

            // 合并局部结果
            #pragma omp critical
            {
                guesses.insert(guesses.end(), local_guesses.begin(),
                    local_guesses.end());
                total_guesses += local_guesses.size();
            }
        }
```

可以看到代码量很少，复杂性很低。我们来分析一下这些语句：

#pragma omp parallel 用于启动一个并行区域，openmp 将创建多个线程，所有线程执行这个大括号内的代码块。另外值得一提的是，这里让每个线程都创建一个自己的 local_guesses 向量，避免多个线程直接访问共享资源导致的冲突，这个向量将用于保存该线程生成的 guess 字符串列表。

#pragma omp for nowait 这个语句会告诉编译器，将接下来的 for 循环中的迭代任务分配给多个

线程并行执行，而 nowait 关键字表示：执行完该循环的线程无需等待其他线程完成，可以去执行别的任务，减少线程同步的等待时间，适合循环没有依赖关系的代码。

#pragma omp critical 定义了一个临界区，在这个代码块里，同一时刻只允许一个线程执行该代码段，以防止对共享资源的同时访问造成数据错误，用于将每个线程的结果合并到最终结果里，其实也就是在合并数据时进行了加锁解锁，保证了线程安全。

3.3 并行算法的正确性验证

可以看到我们实现的并行算法的猜测正确数与串行算法完全相同。

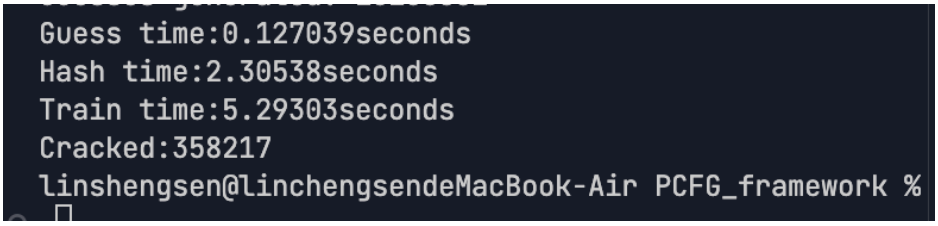

图 3.2: 并行算法的正确性验证

4 性能对比

由于服务器不稳定 guess_time 一直在大幅度跳，所以这里在我自己的 mac 电脑上跑（已征得助教同意）。

对于非 openmp 的程序，采用编译命令

```
g++ main.cpp train.cpp guessing.cpp md5.cpp -o test.exe
```

对于 openmp，采用编译命令为

```
g++ main.cpp train.cpp guessing.cpp md5.cpp -o test.exe -std=c++11 -Xpreprocessor
    -fopenmp -I/opt/homebrew/opt/libomp/include -L/opt/homebrew/opt/libomp/lib -lomp
```

4.1 不开启编译优化的情况下

表 1: 串行与 pthread

问题规模	串行 hash	并行 hash	串行 guess	并行 guess	hash 加速	guess 加速
500w	2.734s	4.861s	0.852s	0.826s	0.562	1.031
1000w	6.207s	10.994s	1.470s	1.351s	0.565	1.088
2000w	12.700s	22.096s	2.986s	2.696s	0.575	1.108
3000w	19.027s	32.859s	4.321s	3.785s	0.579	1.142

在不开启编译优化的情况下，对于我们实现的 pthread 动态编程来说，hash time 都不如串行（simd 编程），可能原因是，并行算法可能将数据分割到不同线程，导致内存访问模式不连续（如跨步访问），降低缓存利用率，而串行算法通常按顺序访问内存，更易被缓存预取优化。也有可能是因为 NEON 函数大量调用，没有进行内联展开，导致并行程序的执行时间更长。guess time 略优于串行算法，并且

随着问题规模的增大，guess time 加速比不断增大，这与我们的自然认知一致，随着问题规模的增大，多线程可以并行使用多个核，加速计算。

表 2: 串行与 openmp

问题规模	串行 hash	并行 hash	串行 guess	并行 guess	hash 加速	guess 加速
500w	2.734s	4.868s	0.852s	0.637s	0.562	1.338
1000w	6.207s	10.937s	1.470s	0.988s	0.568	1.488
2000w	12.700s	22.178s	2.986s	1.961s	0.573	1.523
3000w	19.027s	33.153s	4.321s	2.768s	0.574	1.561

对于 openmp 算法来说，guess time 的加速效果明显更好，最高达到了 1.561，这也体现了 openmp 多线程编程的优越性。并且随着问题规模的增大，guess time 加速比也在逐渐增大。

表 3: 串行与 pthread 线程池

问题规模	串行 hash	并行 hash	串行 guess	并行 guess	hash 加速	guess 加速
500w	2.734s	4.890s	0.852s	0.801s	0.559	1.064
1000w	6.207s	10.934s	1.470s	1.339s	0.568	1.098
2000w	12.700s	22.161s	2.986s	2.748s	0.573	1.087
3000w	19.027s	33.117s	4.321s	3.986s	0.575	1.084

对于 pthread 静态线程池算法来说，其也在一定程度上进行了 guess time 的优化。

4.2 O1 优化情况下

表 4: 串行与 pthread

问题规模	串行 hash	并行 hash	串行 guess	并行 guess	hash 加速	guess 加速
500w	0.624s	0.450s	0.090s	0.091s	1.387	0.989
1000w	1.396s	1.015s	0.149s	0.148s	1.375	1.007
2000w	2.847s	2.069s	0.295s	0.287s	1.376	1.028
3000w	4.308s	3.095s	0.437s	0.400s	1.392	1.093

在 O1 编译条件下，可以看到由于循环进行了展开，hash time 实现了加速，并且 guess time 也实现了加速，随着问题规模的增大加速效果越来越好。

表 5: 串行与 openmp

问题规模	串行 hash	并行 hash	串行 guess	并行 guess	hash 加速	guess 加速
500w	0.624s	0.451s	0.090	0.081s	1.384	1.111
1000w	1.396s	1.009s	0.148s	0.123s	1.384	1.203
2000w	2.847s	2.055s	0.295s	0.243s	1.385	1.214
3000w	4.308s	3.054s	0.437s	0.332s	1.411	1.316

openmp 算法也类似，但 guess time 加速效果更好一些。

表 6: 串行与 pthread 线程池

问题规模	串行 hash	并行 hash	串行 guess	并行 guess	hash 加速	guess 加速
500w	0.624s	0.455s	0.090s	0.085s	1.371	1.059
1000w	1.396s	1.016s	0.148s	0.140s	1.374	1.057
2000w	2.847s	2.045s	0.295s	0.275s	1.392	1.073
3000w	4.308s	3.12s	0.437s	0.396s	1.381	1.104

pthread 静态线程池算法也类似。

4.3 O2 优化情况下

表 7: 串行与 pthread

问题规模	串行 hash	并行 hash	串行 guess	并行 guess	hash 加速	guess 加速
500w	0.423s	0.435s	0.088s	0.081s	0.972	1.086
1000w	0.945s	0.969s	0.143s	0.129s	0.975	1.109
2000w	1.908s	1.965s	0.286s	0.255s	0.971	1.122
3000w	2.871s	2.939s	0.409s	0.355s	0.977	1.152

在 O2 编译情况下，比较反常的就是 hash time 变成了负优化，这也好理解，应该是 O2 编译对于串行算法的加速超越了对于 simd 并行算法的加速。guess time 还是依旧实现了加速，并随着问题规模增大加速效果越来越好。

表 8: 串行与 openmp

问题规模	串行 hash	并行 hash	串行 guess	并行 guess	hash 加速	guess 加速
500w	0.423s	0.461s	0.088s	0.077s	0.918	1.143
1000w	0.945s	0.976s	0.143s	0.117s	0.968	1.222
2000w	1.908s	1.981s	0.286s	0.227s	0.963	1.260
3000w	2.871s	2.971s	0.409s	0.329s	0.966	1.243

openmp 和 pthread 静态线程池算法也类似。

表 9: 串行与 pthread 线程池

问题规模	串行 hash	并行 hash	串行 guess	并行 guess	hash 加速	guess 加速
500w	0.423s	0.434s	0.088s	0.079s	0.975	1.114
1000w	0.945s	0.977s	0.143s	0.126s	0.967	1.135
2000w	1.908s	1.969s	0.286s	0.248s	0.969	1.153
3000w	2.871s	2.965s	0.409s	0.349s	0.968	1.172

基于以上的测试与观察，我发现随着问题规模的增大，多线程编程可以并行充分使用多个核，加速计算。另外，也见识到了 openmp 的优势，其使用的是线程池机制 + 编译器自动优化，可以避免频繁创建销毁线程并且编译器能自动安排线程和负载均衡。

但是，在服务器上测试时，明明我的 pthread 静态线程池算法优于 openmp 算法，但在我自己的电脑上测试时却恰好相反，这是为什么呢？我觉得可能是在服务器上创建或销毁线程的开销更大，导

致 openmp 算法劣于前者，而在我自己的电脑上，创建或销毁线程的开销比较小，所以 openmp 凭借着其编译器能自动安排线程和负载均衡，优于 pthread 静态线程池算法，而 pthread 静态线程池算法不用频繁创建和销毁线程的优势并没有体现出来。

5 进阶要求的实现

5.1 实现相对串行算法的加速

这几次结果是在服务器比较稳定的情况下截取的，可以看到，三种算法都实现了相对串行算法的加速，无论是 hash time 还是 guess time 都优化了。（O2 编译 1000w 数据）在运行 openmp 算法是要在编译时加上-fopenmp 命令。

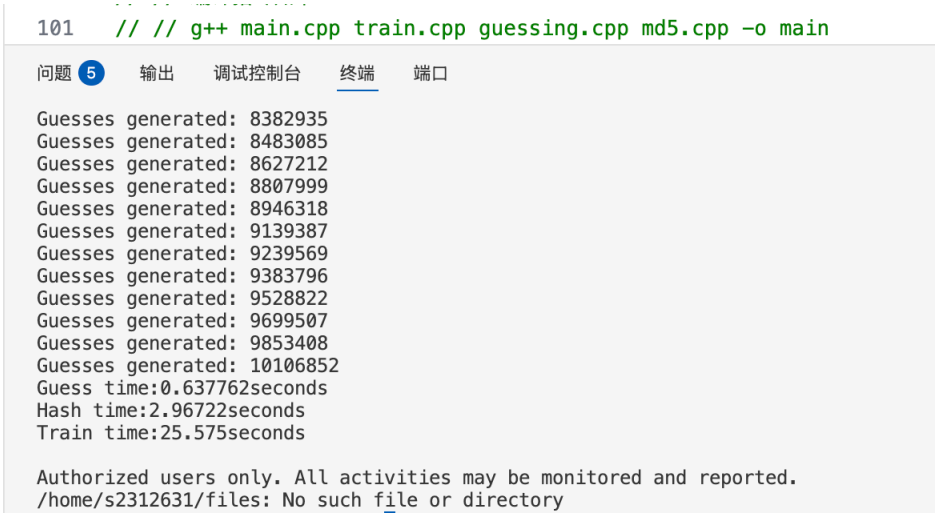

图 5.3: 串行 1000w

可以看到，串行算法的 hash time 为 2.96722s，guess time 为 0.637762s。

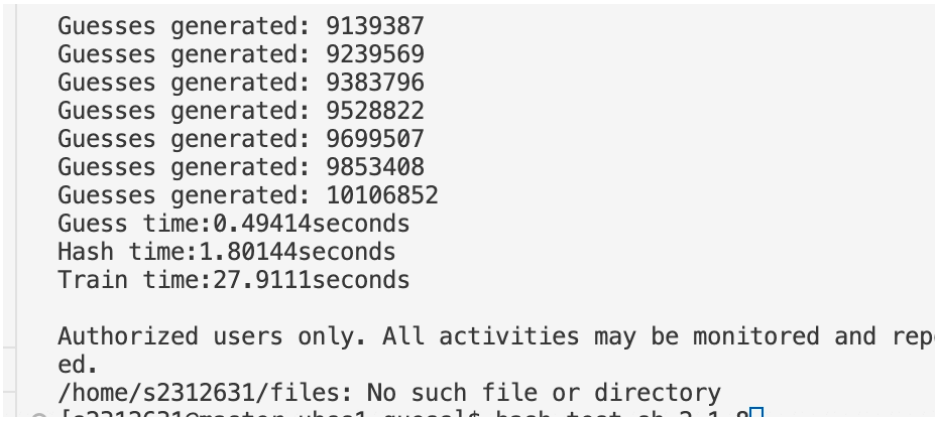

图 5.4: pthread1000w

pthread 动态编程 guess time 为 0.49414s，加速比为 1.291，hash time 为 1.80144s，加速比为 1.647。

```
Guesses generated: 10106852
Guess time:0.480814seconds
Hash time:1.77277seconds
Train time:27.3808seconds

Authorized users only. All activities may be monitored and reported.
/home/s2312631/files: No such file or directory
[s2312631@master_ubss1 guess]$
```

图 5.5: openmp1000w

openmp 编程 guess time 为 0.480814s，加速比为 1.326，hash time 为 1.77277s，加速比为 1.674。

```
Guesses generated: 9853408
Guesses generated: 10106852
Guess time:0.456535seconds
Hash time:1.74877seconds
Train time:27.8359seconds

Authorized users only. All activities may be monitored and reported.
/home/s2312631/files: No such file or directory
```

图 5.6: 线程池 1000w

pthread 静态线程池编程 guess time 为 0.456535s，加速比为 1.397，hash time 为 1.74877s，加速比为 1.697。

5.2 pthread 和 openmp 性能对比

优势	**原因说明**
更少的线程调度	避免频繁创建销毁线程，利用线程复用，由编译器分配任务，降低调度开销。
局部变量缓存友好	使用线程私有局部变量（如 ‘reduction‘），避免共享内存冲突，提高缓存命中率和执行效率。
自动负载均衡	使用线程池任务划分机制，根据任务量动态分配，防止线程空闲或过载。
少锁竞争	避免临界区（如 ‘critical‘）或使用 ‘reduction‘ ，减小线程间同步开销，提高并行效率。
错误率更低	使用封装好的内容，避免手动加锁和线程管理，减少并发 bug，提升程序健壮性。

表 10: openmp 优势

已在前面数据分析的时候说过，总结一下就是在我的电脑上，openmp 算法优于 pthread，原因是在我自己的电脑上，创建或销毁线程的开销比较小，所以 openmp 凭借着其编译器能自动安排线程和负载均衡，防止工作不均衡导致的资源浪费，优于 pthread 静态线程池算法，而 pthread 静态线程池算法不用频繁创建和销毁线程的优势并没有体现出来。在服务器上可能创建或销毁线程的开销更大，而 openmp 要实现更多的线程创建与销毁操作，导致 openmp 算法劣于 pthread。

而这也让我对于多线程编程的理解加深了，并不是同一套在任何环境下都好使，需要根据环境特性进行相应的调试与修改。

5.3 关于串行算法和并行算法的思考

回顾以往的学习，我发现并行化适合处理计算密集型或数据密集型的问题，尤其当任务可以被拆分为多个相互独立的子任务时，其效果尤为显著。这些任务在串行执行时无法充分利用多核 CPU 的并发能力，而通过并行化可极大提高处理速度和系统的资源利用率，从而实现更好的效能。

但是，并行算法并不总是最优选择。当问题规模较小、逻辑依赖强或线程之间共享数据频繁时，并行带来的线程调度、同步开销反而可能超过其带来的性能收益，甚至有可能改变程序的正确性。这类情况下，串行执行具有更低的资源消耗和更高的执行效率，代码也更易维护和调试。

以我们这次做的口令猜测实验为例，这是一个典型的数据并行问题。每个线程可以独立处理一段候选口令的生成与检验，很适合并行化。但在并行实现中也需注意优化策略，例如避免多个线程同时写入共享内存，而是使用线程本地缓存，并在最终阶段合并结果，或者用一个共享数组实现各个线程读写独立；对于线程计数变量应使用 reduction 避免锁竞争，这个 reduction 是 openmp 的一个指令；任务调度应采用动态划分策略，防止工作不均衡导致的资源浪费。

总之，要想设计一个并行化算法，应基于任务特性去权衡开销与收益。在任务独立性强、数据量大时并行化更合适；而在任务轻量或前后逻辑相互串联时，串行处理可能更高效。

6 心得体会

1. 通过这次实验，我对于多线程编程有了更深刻的认识，学会了 pthread 编程和 openmp 编程，pthread 编程需要自己创建线程并进行相关操作，比较麻烦且优化性能不一定好，openmp 编程易于理解，代码复杂性低，且往往会有很好的效果。

2. 通过将 pthread 动态编程优化成相对串行的加速版本，我对于锁有了新的认识。什么时候该用锁？什么时候可以不必用锁？这可能与我们选取的数据结构有关。在我的修改中，把不能共享访问的字符串容器改成了字符串数组，只要控制好访问的范围，字符串数组可以共享访问，不用加锁，不必额外消耗建锁解锁的时间，使得程序实现加速。

3. 对于多线程编程，并不一定是线程数越多越好，当问题规模较小时明显没必要，就算问题规模比较大时，可能会由于频繁地创建和销毁线程导致时间效益明显下降，正如我们动态编程，所以我设置的线程数为 4 而不是 8。但对于基于 pthread 的静态线程池编程，我们只在主程序的开始创建一次线程，在主程序的结束销毁一次线程，就不用考虑这个问题，可以把线程数设置为 8，使用多线程数带来的收益远远高于创建的损失。

4. 另外我还深刻理解了 PCFG 口令猜测的原理，就是通过训练拿到概率，计算具体一个 pt 的概率进行猜测，我还震惊于其解密功能。

7 代码链接

点击访问 GitHub 主页