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

许洋

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 5 月 29 日

摘要

实验针对概率上下文无关文法（PCFG）口令猜测技术中的候选生成环节，基于 pthread 和 OpenMP 实现了多线程并行优化。通过重构模板初始化、概率计算、队列扩展等核心模块的并行架构，系统分析了编译优化等级、线程数规模、口令总数对性能的影响。实验表明，OpenMP 凭借零锁设计、线程池管理和智能调度策略，在千万级口令规模下实现 1.4-1.5 倍加速，显著优于 pthread 的 1.15 倍边际优化。进一步通过全流程并行化改造，OpenMP 加速比提升至 1.67 倍。研究还揭示了 SIMD 向量化与多线程的资源竞争瓶颈，并探讨了 GPU 异构加速的可行性。

关键字：多线程优化、OpenMP、pthread、性能分析

目录

一、 问题重述

PCFG（概率上下文无关文法）口令猜测是结合统计学习与文法规则的密码猜测攻击技术，相比传统暴力破解和字典攻击，能更高效生成密码候选列表。其利用密码样本构建概率文法模型，按概率高低生成并尝试密码。

实验中，PCFG 实现包含四个过程：首先分析训练集，提取密码结构（如 L8D3S3，代表 8 个字母、3 个数字、3 个符号组成的密码）并计算其出现概率；接着建立概率模型，为密码模式和各组成部分计算概率，通过条件概率算出完整密码的出现概率；然后依据概率文法生成口令候选，优先输出高概率组合；最后按概率从高到低尝试候选口令，缩小猜测范围。

本次多线程编程实验聚焦后三个过程，需修改 guessing.cpp 文件，通过多线程技术优化口令候选生成与猜测过程，提升相比串行方式的猜测性能。其中，PT 代表密码拆分结构（如 L6S1），每个拆分部分称为 Segment。

二、 串行分析

1. void PriorityQueue::CalProb(PT pt)

计算某个 PT（密码模板）的整体概率。PT 是一个由多个 segment 构成的模板，每个 segment 是一种字符类型（字母、数字、符号）。PT 的概率 = preterm（模板）概率 × 每个 segment 的具体值的频率比例。

伪代码

```
function CalProb(pt):
    pt.prob = pt.preterm_prob
    for i from 0 to len(pt.curr_indices):
        idx = pt.curr_indices[i]
        seg = pt.content[i]
        if seg is Letter:
            freq = m.letters[m.FindLetter(seg)].ordered_freqs[idx]
            total = m.letters[m.FindLetter(seg)].total_freq
        else if seg is Digit:
            freq = m.digits[m.FindDigit(seg)].ordered_freqs[idx]
            total = m.digits[m.FindDigit(seg)].total_freq
        else if seg is Symbol:
            freq = m.symbols[m.FindSymbol(seg)].ordered_freqs[idx]
            total = m.symbols[m.FindSymbol(seg)].total_freq
        pt.prob *= (freq / total)
```

2.void PriorityQueue::init()

初始化优先队列，计算所有模板的初始概率并插入。遍历所有 PT 模板，计算概率后插入优先队列。概率由 preterm 概率和 segments 对应值的统计数量决定。

伪代码

```
function init():
    for pt in m.ordered_pts:
        for seg in pt.content:
            if seg is Letter:
```

```
                pt.max_indices.append(size of m.letters[m.FindLetter(seg)].
                    ordered_values)
            else if seg is Digit:
                pt.max_indices.append(size of m.digits[m.FindDigit(seg)].
                    ordered_values)
            else if seg is Symbol:
                pt.max_indices.append(size of m.symbols[m.FindSymbol(seg)].
                    ordered_values)
        pt.preterm_prob = m.preterm_freq[m.FindPT(pt)] / m.total_preterm
        CalProb(pt)
        priority.append(pt)
```

3.void PriorityQueue::PopNext()

处理优先队列中的当前最佳模板，生成猜测并生成其后继模板。生成当前最高优先级的猜测。生成后继模板，重新插入优先队列。

伪代码

```
function PopNext():
    pt = priority.front()
    Generate(pt)
    new_pts = pt.NewPTs()
    for new_pt in new_pts:
        CalProb(new_pt)
        insert new_pt into priority by descending prob
    remove pt from priority
```

4.vector<PT> PT::NewPTs()

基于当前 PT 的状态生成新模板,遍历除最后一个 segment 以外的组合。通过递增 curr_indices[i] 的方式生成新模板。除最后一个 segment 以外的每个 segment 都可以尝试增加当前的值。

伪代码

```
function NewPTs():
    res = []
    if content.size() == 1:
        return res
    original_pivot = pivot
    for i in range(pivot, curr_indices.size() - 1):
        curr_indices[i] += 1
        if curr_indices[i] < max_indices[i]:
            pivot = i
            res.append(copy of current PT)
        curr_indices[i] -= 1
    pivot = original_pivot
    return res
```

5.void PriorityQueue::Generate(PT pt)

为给定的模板 PT 生成所有可能的密码猜测（只处理最后一个 segment）。如果只有一个 segment，直接遍历所有 value。否则，只补齐最后一个 segment，前面的由 curr_indices 确定。

伪代码

```
function Generate(pt):
    CalProb(pt)
    if pt.content.size() == 1:
        seg = pt.content[0]
        a = reference to segment in model (letter/digit/symbol)
        for i in range(0, pt.max_indices[0]):
            guess = a.ordered_values[i]
            guesses.append(guess)
            total_guesses += 1
    else:
        guess = ""
        for i from 0 to pt.content.size() - 2:
            seg = pt.content[i]
            idx = pt.curr_indices[i]
            guess += model(seg).ordered_values[idx]
        last_seg = pt.content.back()
        a = model(last_seg)
        for i in range(0, pt.max_indices.back()):
            final_guess = guess + a.ordered_values[i]
            guesses.append(final_guess)
            total_guesses += 1
```


三、 实现思路

为了提升密码猜测引擎的整体性能，我们对串行代码中多个关键模块进行了并行化改造，主要包括模板生成、概率初始化、任务调度与模板扩展四个方面。以下分别阐述每一部分的串行逻辑与并行优化策略：

1. `PriorityQueue::Generate(PT pt)` 的并行化（核心优化）

串行逻辑：

若模板 `pt` 仅包含一个 segment，则直接遍历所有可能值，生成完整密码猜测。

若包含多个 segment，则固定前缀，仅对最后一个 segment 进行值枚举。

并行化策略：

由于候选值组合之间完全独立，可采用 OpenMP 对候选值空间并行遍历。各线程生成的猜测统一汇入结果集合。

2.`PriorityQueue::init()` 的并行化

串行逻辑：

遍历预排序的所有 PT (m.ordered_pts)，对每个 PT 根据 segment 类型，计算每个 segment 的最大枚举数(max_indices)，初始化当前枚举状态(curr_indices)，计算其概率值(CalProb(pt))，存入优先队列。

并行化策略：

以每个模板 PT 为最小单元进行并行计算，彼此无依赖。使用 #pragma omp parallel 启用并行区域；利用 #pragma omp for schedule(dynamic, 64) 对 m.ordered_pts 划分任务块，动态调度提高负载均衡。

每个线程写入 thread_local_results[tid]，避免共享写入冲突（data race）；主线程最后合并所有线程的结果进入 priority。

合并后调用 std::sort 对整个 priority 优先队列按概率降序排列，确保后续调度正确。

3.`PriorityQueue::PopNext()` 的并行化

串行逻辑：

从优先队列弹出概率最高的 PT；生成新的 PT 列表（通过 NewPTs()）；对每个新 PT 计算概率；插入到优先队列中，维持降序排列。

并行化策略：

每个新 PT 的概率计算互不依赖，天然适合并行。与 init() 类似，使用 #pragma omp parallel 并结合 #pragma omp for schedule(dynamic, 64)，对 new_pts 列表进行并行处理。

各线程将计算结果放入各自的 thread_local_results[tid] 中，避免写入冲突；主线程合并所有新 PT，统一排序并插入优先队列。

使用 std::lower_bound 找到每个新 PT 在当前 priority 中的插入位置；保证最终 priority 队列依然按概率降序。

4.`PT::NewPTs()` 的并行化串行逻辑：

对当前 PT 对象，从 pivot 位开始，向后依次尝试将某个 segment 的索引值加一；若增加后仍合法（未越界），则创建一个新的 PT（状态扩展）；记录所有合法的新状态。

并行化策略：

每个 segment 增加一的尝试互不干扰，适合并行。使用 #pragma omp for schedule(dynamic, 64) 并行遍历 curr_indices 中的后缀索引；每次操作前增加 curr_indices[i]，生成新 PT 后再恢复原值，确保线程安全。

与主流程一致，采用 thread_local_results[tid] 分别保存新状态；最终合并结果统一返回。

遍历过程中临时修改原始 PT，操作后立即恢复，保证在并发条件下不会影响主对象。

四、 多线程实现

(一) pthread

1. 代码实现

pthread 结构体以及线程函数

```
// 定义线程参数结构体
struct ThreadArg {
    segment* seg_ptr;
    vector<string>* guesses;
    size_t start_idx;
    size_t end_idx;
    string prefix;
    atomic<int>* total_guesses;
    pthread_mutex_t* mutex;
    size_t buffer_size;  // 添加缓冲区大小
};
// 线程函数
void* thread_generate(void* arg) {
    ThreadArg* targ = (ThreadArg*)arg;
    // 使用固定大小的缓冲区
```

```
    const size_t BUFFER_SIZE = 1000;
    vector<string> buffer;
    buffer.reserve(BUFFER_SIZE);
    size_t processed = 0;
    for(size_t i = targ->start_idx; i < targ->end_idx; i++) {
        buffer.push_back(targ->prefix + targ->seg_ptr->ordered_values[i]);
        processed++;
        // 当缓冲区满时批量提交
        if(buffer.size() >= BUFFER_SIZE) {
            pthread_mutex_lock(targ->mutex);
            targ->guesses->insert(targ->guesses->end(),
                                  make_move_iterator(buffer.begin()),
                                  make_move_iterator(buffer.end()));
            pthread_mutex_unlock(targ->mutex);
            // 更新计数并清空缓冲区
            targ->total_guesses->fetch_add(buffer.size(),
                memory_order_relaxed);
            buffer.clear();
            buffer.reserve(BUFFER_SIZE);
        }
    }
    // 处理剩余的数据
    if(!buffer.empty()) {
        pthread_mutex_lock(targ->mutex);
        targ->guesses->insert(targ->guesses->end(),
                              make_move_iterator(buffer.begin()),
                              make_move_iterator(buffer.end()));
        pthread_mutex_unlock(targ->mutex);
        targ->total_guesses->fetch_add(buffer.size(), memory_order_relaxed);
    }
    return NULL;
}
```

pthread 实现

```
        // 预分配空间
        size_t total_size = pt.max_indices[0];
        // 预分配较小的初始空间,后续按需扩展
        guesses.reserve(guesses.size() + min(total_size, size_t(100000)));
        vector<pthread_t> threads(NUM_THREADS);
        vector<ThreadArg> thread_args(NUM_THREADS);
        pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
        size_t chunk_size = (total_size + NUM_THREADS - 1) / NUM_THREADS;
        // 创建线程
        for(int i = 0; i < NUM_THREADS; i++) {
            thread_args[i] = {
                a,
                &guesses,
                i * chunk_size,
```

```
                min((i + 1) * chunk_size, total_size),
                "",
                &atomic_total_guesses,
                &mutex,
                chunk_size
            };
            if(pthread_create(&threads[i], NULL, thread_generate, &
                thread_args[i]) != 0) {
                cerr << "线程创建失败:␣" << i << endl;
                exit(1);
            }
        }
        // 等待所有线程完成
        for(int i = 0; i < NUM_THREADS; i++) {
            pthread_join(threads[i], NULL);
        }
        pthread_mutex_destroy(&mutex);
        total_guesses += atomic_total_guesses.load();
```

2. 代码解析

1. 线程结构定义解析:

在线程结构体中需要定义线程需要的相关变量:

seg_ptr：提供生成字符串所需的原始数据（如字典项）。

guesses 和 total_guesses：分别存储结果和统计总数。

mutex：防止多线程同时操作 guesses 导致数据竞争。

在这里封装了线程所需的全部参数，包括输入数据、输出容器、同步工具等。

2. 线程函数 thread_generate 定义解析:

参数解析：将 void 类型参数转换为 ThreadArg。

缓冲区初始化：预分配固定大小内存提升性能。

循环生成字符串：遍历 [start_idx, end_idx) 区间，拼接前缀和字典项。

批量提交：缓冲区满时，通过互斥锁保护将数据移动到共享容器。

原子计数：使用 fetch_add 无锁更新总数。

剩余数据处理：循环结束后提交未满缓冲区的数据。

3. 主线程实现

任务分块：通过 chunk_size 将总任务均匀分配给线程。

chunk_size = (total_size + NUM_THREADS - 1) / NUM_THREADS; 向上取整除法，确保所有数据被分配。

线程参数绑定：每个线程处理独立的数据区间，共享容器和计数器。互斥锁保护共享容器，仅在向 guesses 插入数据时加锁，最小化锁持有时间。

移动语义：make_move_iterator 避免字符串拷贝，提升性能。原子计数器

无锁操作：fetch_add 保证多线程并发更新时的原子性。

内存序：memory_order_relaxed 在无严格顺序要求时提升性能。

线程创建与错误处理：直接退出程序（需优化为资源清理）。

线程同步：pthread_join 确保所有线程完成后继续执行主线程。

(二) OpenMP

1. 代码实现

OpenMP 实现

```
1
2       // openmp code
3       // 使用OpenMP并行化for循环
4       omp_set_num_threads(5); // 设置线程数为5
5       const int n = pt.max_indices[0];
6       const size_t old_size = guesses.size();
7       guesses.resize(old_size + n); // 预分配空间
8
9       // OpenMP并行化循环, 直接写入预分配的空间
10      #pragma omp parallel for schedule(guided, 512)
11      for(int i = 0; i < n; ++i) {
12          guesses[old_size + i] = a->ordered_values[i]; // 按索引直接写入
13      }
14
15      // 一次性更新计数
16      total_guesses += n;
```

OpenMP 实现

```
1
2       // openmp code
3
4       // 使用OpenMP并行化for循环
5       omp_set_num_threads(5); // 设置线程数为5
6       const int n = pt.max_indices[pt.content.size() - 1];
7       const size_t old_size = guesses.size();
8       guesses.resize(old_size + n); // 预分配空间
9
10      // OpenMP并行化循环, 直接写入预分配的空间
11      #pragma omp parallel for schedule(guided, 512)
12      for(int i = 0; i < n; ++i) {
13          guesses[old_size + i] = guess + a->ordered_values[i]; // 按索引直
                接写入
14      }
15
16      // 一次性更新计数
17      total_guesses += n;
```

2. 代码解析

1. 设置线程数

omp_set_num_threads(x);

设置使用 x 个线程, 强制后续并行区域使用 x 个线程。

2 计算待处理元素数量 n

const int n = pt.max_indices[pt.content.size() - 1];

pt.content.size() - 1：获取 pt.content 容器的最后一个元素的索引。

pt.max_indices[...]：根据该索引从 pt.max_indices 数组中提取值，作为 n。

3. 预分配容器空间

const size_t old_size = guesses.size();

guesses.resize(old_size + n);

记录当前容器大小 old_size。为将 guesses 的大小扩展为 old_size + n，预留连续内存空间。避免并行写入时频繁调整容器大小，保证线程安全。

4. 并行化循环写入数据

#pragma omp parallel for schedule(guided, 512)

for(int i = 0; i < n; ++i) {

guesses[old_size + i] = guess + a->ordered_values[i];

}

schedule(guided, 512)：动态任务调度，初始分配大块任务（至少 512 次迭代），剩余任务逐步减小块大小。适合负载不均衡的场景。

guesses[old_size + i]：直接按索引写入预分配的内存位置。

guess + a->ordered_values[i]：将基准值 guess 与数组元素相加后写入。

每个 i 对应唯一的内存地址，无数据竞争。需确保 guess 在并行区域内为只读（无修改操作）。

5. 更新总数

total_guesses += n;

单线程更新全局计数器，无需同步操作（n 为常量）。

五、 性能分析

测试过程中我采用了多次测量求平均值的方法，来减小平台波动误差来使得结果分析更加准确。

(一) 优化分析

前置条件：口令总规模为 10000000，多线程优化线程数为 4

采用了 O0（不优化）、O1 优化和 O2 优化三种优化方式进行测量，每次至少测量 5 次求取平均值，得到以下表格数据：

优化等级	串行	pthread	OpenMP	pthread 加速比	OpenMP 加速比
O0	7.8418	7.8092	7.3727	1.0042	1.0636
O1	0.6995	0.7198	0.5080	0.9718	1.3770
O2	0.6692	0.6526	0.5350	1.0254	1.2509

表 1: 不同优化等级下的运行时间（单位：秒）和加速比

编译优化等级的影响 编译优化等级对程序性能具有关键影响。从表格数据可观察到：

- **-O0（无优化）**：三种方案性能差距最小，pthread 与 OpenMP 分别仅带来约 0.4% 和 6.3% 的加速。这是因为编译器未进行指令级优化，函数调用、内存访问、循环展开等皆保持原样，系统资源未被有效利用，导致多线程的并发优势难以发挥。
- **-O1（基础优化）**：此等级下 OpenMP 显著优于串行（加速比 1.377），而 pthread 反而略慢（加速比 < 1）。这一反直觉现象背后有两个原因：一是 pthread 实现中使用了共享 vector 写入与 mutex 锁，锁竞争严重；二是线程粒度较小，pthread 的线程启动与同步开销未能被任务量抵消，反而拖慢整体性能。
- **-O2（高级优化）**：此时各方案均达到了最优状态，OpenMP 仍维持稳定加速，而 pthread 表现有所改善。说明 O2 中更激进的内联、循环优化、寄存器分配策略对线程代码结构有积极影响。

pthread 实现的瓶颈分析 pthread 虽具备灵活控制线程生命周期的优势，但在本实验中暴露出几个关键性能瓶颈：

- **互斥锁开销大**：每个线程需要通过 `pthread_mutex_lock` 控制对 vector 的写入，导致多个线程频繁陷入阻塞状态。尤其在短任务中，锁操作相较实际工作内容比例过高，成为主要瓶颈。
- **缺乏线程局部缓存**：所有线程共享对同一个 vector 的访问，没有引入线程私有容器或预聚合机制，导致写操作时发生伪共享与缓存一致性问题，进一步拖慢性能。
- **调度负担高**：由于任务粒度较小，每个线程实际负责不到 1/NUM_THREADS 的工作量，线程上下文切换开销（尤其是频繁创建/销毁）未被摊销。

OpenMP 优势详解 相比 pthread，OpenMP 表现出更强的扩展性和调度效率，归因如下：

- **零锁设计**：OpenMP 实现采用预分配写入空间（`vector::resize`）+ 索引写入方式，避免了锁冲突。每个线程负责写入不重叠的段，天然避免数据竞争。
- **低开销线程池**：OpenMP 默认使用后台线程池进行任务分配，无需每次创建销毁线程，大幅降低调度成本，特别适合此类数据并行（data-parallel）任务。
- **自动调度策略**：实验中使用 `schedule(guided, 512)` 指令块自动控制负载平衡，避免线程空闲或负载倾斜问题。这种调度策略在任务量不均时尤为有效。

数据局部性与缓存友好性 值得注意的是，编译器优化等级提升不仅影响逻辑执行效率，还影响 CPU 缓存友好性：

- 在 O1/O2 中，内存访问模式更符合顺序访问原则，使得缓存命中率提升；
- OpenMP 实现中线程独立写入 vector 不重叠段，有效避免伪共享，提高缓存带宽利用；
- pthread 中多线程抢写同一个 vector 头部，可能导致缓存行争夺，增加内存总线压力；

小结 综合上述因素，在当前实验环境与任务设置下：

- **OpenMP 在性能与实现复杂度之间取得最佳平衡**，尤其在 O1 以上优化等级时，优势明显；
- **pthread 更适合控制精度高、任务粒度大或含状态管理的并行逻辑**，但在数据密集型任务中若无锁优化措施，性能将受限；
- **串行实现在 O2 下已接近其上限性能**，适合小规模调试或不易并行化任务；

(二) 线程数分析

前置条件：口令总规模为 10000000，采用 O1 优化

采用了多线程优化线程数从 4 到 7 进行测量，每次至少测量 5 次求平均值，得到以下表格数据：

表 2: 不同线程数下的运行时间与加速比（单位：秒）

线程数	串行	pthread	OpenMP	pthread 加速比	OpenMP 加速比
4	0.6995	0.7198	0.5080	0.9718	1.3770
5	0.6995	0.6565	0.5039	1.0655	1.3883
6	0.6995	0.6541	0.4999	1.0695	1.3992
7	0.6995	0.8433	0.5022	0.8295	1.3928

总体趋势分析 从表格与图像中可以观察到以下几个关键趋势：

- **OpenMP 随线程数增加，加速比稳步提升**：在 4 到 6 个线程之间，OpenMP 加速比从 1.377 提高至 1.399，表现出良好的可扩展性；
- **pthread 并不随线程数提升而线性加速**：pthread 在 5、6 线程时略优于串行，达到最大加速比 1.069，但 7 线程时性能大幅下降至 0.83；

原因分析 结合代码实现与硬件特性，造成上述现象的原因可归结为以下几点：

1. **线程管理开销**：pthread 每次运行需显式创建与销毁线程,涉及内核级系统调用。而 OpenMP 使用线程池，避免了重复创建，管理效率更高。
2. **负载不均衡与调度开销**：pthread 手动分配任务区间，若划分不均，会导致部分线程早早结束，其他线程仍在工作。而 OpenMP 支持如 `schedule(dynamic)` 之类的动态分配策略，使得线程能动态领取任务块，提高资源利用率。

最优线程数推测 由表中可知，OpenMP 在 6 个线程时达到最佳性能（1.399 倍加速），继续增加线程数并未显著提升性能。

而 pthread 在 6 线程时性能略优（1.069 倍），但在 7 线程时加速比急剧下降，说明其线程调度与管理在高并发下效率不足，暴露了其扩展性较差的特性。

结论 综上所述，线程数的选择对并行程序的性能具有显著影响。合理的线程数应根据目标平台的核心数、任务粒度与线程管理开销综合考量。OpenMP 由于其自动线程管理与调度机制，在多线程扩展性上明显优于 pthread；而 pthread 在线程数过多时反而引入性能损耗，需要更动态的规划。

(三) 总猜测数分析

前置条件：采用 O1 优化，多线程优化线程数为 5

测量了不同口令总规模的数据，小规模的口令至少进行 5 次测量求平均值，大规模的口令至少进行了 3 次测量求平均值，得到以下数据：

表 3: 不同口令数下运行时间与加速比

总口令数（万）	串行	pthread	OpenMP	pthread 加速比	OpenMP 加速比
200	0.1976	0.2580	0.1607	0.7662	1.2298
400	0.3263	0.3648	0.2671	0.8942	1.2216
600	0.4991	0.5251	0.3988	0.9505	1.2515
800	0.6145	0.6849	0.4173	0.8971	1.4725
1000	0.6995	0.6565	0.5039	1.0655	1.3883
1200	0.8327	0.8187	0.6065	1.0171	1.3729
1400	0.9603	0.9473	0.6305	1.0137	1.5230
1600	1.1598	1.0987	0.7559	1.0556	1.5344
1800	1.2737	1.2720	0.9207	1.0014	1.3835
2000	1.5522	1.4497	1.0124	1.0707	1.5332
3000	2.1066	1.9160	1.6194	1.0995	1.3009
4000	2.7260	2.4184	2.2404	1.1272	1.2167
5000	3.4596	3.0325	2.6781	1.1408	1.2918

趋势观察

- **OpenMP 随口令数增长加速比提升明显**：当总口令数从 200 万增至 1600 万时，加速比由 1.22 提高至 1.53，表明任务规模的增大有效摊平了 OpenMP 的并行开销；
- **pthread 初期低效，但规模增大后改善**：在任务较小时，pthread 加速比小于 1，表明并行反而带来负担；但在 2000 万以上逐渐提升至 1.14；
- **任务规模越大并行优势越明显**：这体现了“并行粒度”的概念，即每个线程所承担的工作越多，线程调度、同步等开销相对越小，程序并行效率越高。

结合代码分析原因 结合具体实现，造成上述现象的核心原因有：

1. **初始化与调度开销占比**：在小任务规模下，pthread 的线程创建与销毁、内存拷贝、工作划分等固定成本相对较高。具体来看，代码中：

Listing 1: pthread 主调函数片段

```
for (int i = 0; i < thread_count; i++) {
```

```
    pthread_create(&threads[i], NULL, crack_range, &args[i]);
}
for (int i = 0; i < thread_count; i++) {
    pthread_join(threads[i], NULL);
}
```

每次都需要大量 ‘pthread_create‘ 与 ‘join‘，造成开销；而在任务量提升后，每个线程处理数据量更多，单位工作量的固定开销比例下降，因此加速比提升。

2. **OpenMP 的线程池与调度优化**：OpenMP 使用线程池 + 任务划分策略如 `static` 或 `dynamic`，具有显著的调度效率。例如：

Listing 2: OpenMP 并行循环

```
#pragma omp parallel for
for (int i = 0; i < total; i++) {
    if (check_password(i)) cracked++;
}
```

OpenMP 能直接利用已有线程池对 ‘for‘ 循环自动划分，调度粒度小、均衡性好；因此即使任务较小时也能获得可观加速效果。

3. **串行版本缓存友好但无法并行利用资源**：串行版本虽然避免了线程调度和数据共享问题，但由于未能利用多核并行能力，执行时间随任务线性增长，效率逐渐劣于并行版本。

六、 进阶修改

在前面的修改过程中，我们已经实现了 pthread 和 openmp 算法对 guess time 的加速，同时对 MD5hash 的时间进行了测量，发现 SIMD 编程的 MD5hash 算法同样也实现了加速。以下是测量数据。

表 4: SIMD 在并行下的优化加速（-O1）

线程数	实现方式	串行时间 (s)	并行时间 (s)	加速比 (x)
	串行	7.23828	2.69479	1.12876
3	pthread	7.51917	7.08008	1.06202
	openmp	7.59873	7.1816	1.05808
4	pthread	7.56529	6.90653	1.09538
	openmp	7.92418	7.60034	1.04261
5	pthread	7.2606	6.78429	1.07021
	openmp	7.18172	6.7095	1.07038
6	pthread	7.56679	6.84566	1.10534
	openmp	7.44705	6.67199	1.11617
7	pthread	7.54208	6.93717	1.0872
	openmp	7.32609	6.95552	1.05328

补充：在测量过程中我发现了 SIMD 加速比比上一次报告中变小了，但是一直都还是加速状态，这里我研究了原因，了解了其他同学 SIMD 方法进行 MD5hash 计算的方法后得知，我采

用的计算方法是将全部猜测一次性传入，并且进行消息的填充，这里大量传参，以及消息填充，同时应用 vector 向量都会减慢加速效果，而学习了其他人的方法我了解到在参数传递过程我们按照数组的形式传入参数，同时提前计算好传入参数个数，这样会提高加速效果，在这次实验中虽然衰减但是也看起来没有这么低。我也尝试进行了代码的修改，发现确实提高了不少，但是基于之前的实验继承的结果，保证单一变量的对比没有体现在此。

之后我们以 OpenMP 为例进行多线程进一步修改。

(一) 进阶修改 1

在这里我进一步对并行化的函数进行了修改，将修改思路中除了 Generate 函数之外的其他函数也进行了修改，发现效果显著。在这里给出伪代码以及测量数据。

Listing 3: PriorityQueue::init() 并行化伪代码

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp for schedule(dynamic, 64)
    for (int i = 0; i < ordered_pts.size(); ++i) {
        PT pt = ordered_pts[i];
        // 初始化 pt 的 max_indices、curr_indices、概率等
        CalProb(pt);
        // 写入线程本地结果
        thread_local_results[tid].push_back(pt);
    }
}
// 合并所有线程本地结果
for (auto& local_vec : thread_local_results)
    priority.insert(priority.end(), local_vec.begin(), local_vec.end());
// 按概率排序
std::sort(priority.begin(), priority.end(), 比较函数);
```

Listing 4: PriorityQueue::PopNext() 并行化伪代码

```
std::vector<PT> new_pts = priority.front().NewPTs();

#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp for schedule(dynamic, 64)
    for (int i = 0; i < new_pts.size(); ++i) {
        CalProb(new_pts[i]);
        // 写入线程本地结果
        thread_local_results[tid].push_back(new_pts[i]);
    }
}
// 合并所有线程本地结果
for (auto& local_vec : thread_local_results)
    processed_pts.insert(processed_pts.end(), local_vec.begin(), local_vec.
        end());
```

```
// 排序 + 插入优先队列中合适位置
std::sort(processed_pts.begin(), processed_pts.end(), 比较函数);
for (const PT& pt : processed_pts)
    priority.insert(合适位置, pt);
// 删除原先队首
priority.erase(priority.begin());
```

Listing 5: PT::NewPTs() 并行化伪代码

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    #pragma omp for schedule(dynamic, 64)
    for (int i = pivot; i < curr_indices.size() - 1; ++i) {
        curr_indices[i] += 1;
        if (curr_indices[i] < max_indices[i]) {
            PT new_pt = *this;
            new_pt.pivot = i;
            thread_local_results[tid].push_back(new_pt);
        }
        curr_indices[i] -= 1;
    }
}
// 合并所有线程本地结果
for (auto& local_vec : thread_local_results)
    result.insert(result.end(), local_vec.begin(), local_vec.end());
```

测量数据与对比:

```
Guess time: 0.395906 seconds
Serial Hash time: 8.0902 seconds
Parallel Hash time: 7.39198 seconds
Parallel Speedup ratio: 1.09446x
Train time: 27.4403 seconds
Cracked:353982
```

图 1: 优化 =-O1、口令数 =10000000、线程数 =5

相比于之前的单独一部分用 openmp 并行得到的 1.35 的加速比，现在的加速比提升到了 1.5+，在更大量的数据上会有更好的结果，同时 SIMD 优化从平均 1.05 的加速比提升到了平均 1.1，依旧有了小幅度的提升，但是，由于并行化的颗粒度原因，导致口令猜测的准确性产生了少量下降，不过数量级并没有产生大幅度的变化，说明加速很成功。这里产生的对于 SIMD 的加速可能是由于并行的加速，使得 SIMD 资源等待时间变短，从而产生了小幅度的加速。

(二) 进阶修改 2

通过阅读文章，我了解了新的并行策略，想尝试修改，不过没有成功。从文章中学习到优化策略：[3]

1. 内存存储结构的优化（GPU)

字典存储优化：字符串按类型、长度、概率三级排序存储为一维数组，预计算起始位置数组（T×L）实现快速定位，减少内存碎片，提升 GPU 访问效率。

PT 结构优化：每个 PT 固定 256 字节（头 24 字节 + 体 232 字节），严格匹配 GPU 缓存行，减少 GPU 内存读取次数，提升吞吐量。

容器平滑技术：调整容器大小为 2 的幂次（如 8），重复填充保证任务均衡，线程任务分配均匀，避免执行等待。

2. 并行生成算法的实现

二维索引机制：通过 (PT_id, Guess_id) 定位密码，线程独立计算索引, 完全并行化，GPU 线程无冲突.

反向索引算法：通过模运算逆向计算字符串索引, 快速定位字符，减少计算开销。

异步处理机制：CPU 生成 PT 与 GPU 处理密码异步执行, 提升 CPU-GPU 协作效率，减少空闲时间。

3. 负载均衡策略

容器分组：将概率相近的字符串分组为 2 的幂次大小, 避免线程资源浪费（如某些线程空闲）。

动态线程分配: 根据 PT 的密码数动态分配线程块(如 1024 线程/块)。这个应用在了 openmp 中设置了 dynamic 的分配。

(三) 进一步探究

在这一步探究之前我想阐述一下我的错误:

认真读代码是个好习惯,在没有认真看代码的前提下盲目进行了修改,直接加入了 MD5hashbatch 函数的测量，忽略了在测量 m 到 hash 串行时间的过程中产生了记录 cracked 的时间变化，导致 MD5hash 串行时间边长，而只测量了 MD5hashbatch 的计算时间。再次进行修改过程中发现依旧没有遇到不能同时优化的问题。但是相比于之前完全串行的时候，加速比变小了，所以需要进行探究原因。当然我觉得这个问题和不能同时加速的问题有共通之处。

这里回答提出的多线程和 simd 不能同时优化的问题（我没有遇到，是因为在改的过程中这两者优化和使用的地方不同，但是这个问题感觉很有意思，所以参考一下网络上以及 AI 的回答）[2] [1]

原因一：资源竞争与瓶颈转移

SIMD 加速哈希会大量占用 CPU 内部的 SIMD 单元和寄存器。而多线程猜测会用满多个 CPU 核心, 每个线程也需要执行哈希操作。当每个线程都调用 SIMD 哈希时：每个核心的 SIMD 单元就成了共享资源，出现资源冲突和上下文切换开销。

结果：多线程同时 SIMD 时，每个线程的 SIMD 调用效率下降，线程调度频繁，导致加速“打架”，反而不如只用其中一种优化策略高效。[4]

原因二：内存带宽成为瓶颈

多线程和 SIMD 都是吞吐量型优化：它们对内存的需求增大。如果哈希处理的是大批量数据（如批量字符串），SIMD 会快速消耗内存带宽。多线程的“猜测”线程需要频繁读写内存，也依赖缓存和内存。

结果：L1/L2 cache_miss 增多，主存访问频率升高，内存成为瓶颈，造成延迟升高和效率下降。

原因三：数据依赖性

多线程的分块策略可能与 SIMD 所需的数据连续性冲突。多线程划分的数据块若未对齐或存在伪共享，会破坏 SIMD 的向量化效率, 可能导致负载不均衡，使得加速效果不显著。[5]

原因四：线程调度与 SIMD 非兼容

在某些平台上（尤其是嵌入式 ARM 系统）：SIMD 单元可能共享（比如一对核心共享 NEON 单元）。而你的线程调度器（如 OpenMP）并不知道这些硬件共享情况。所以你以为开了 8 个线程，每个都在跑 NEON，实际上只有一半线程真的在做事，另外一半在等资源。

七、 总结

从实验结果和分析可得以下结论：

OpenMP 凭借线程池管理和零锁设计，在千万级任务中实现 1.4-1.5 倍加速，显著优于 pthread 的边际优化（1.15 倍）。优化编译等级、线程数、口令总数都会对并行优化产生不同的结果。

进阶优化通过全流程并行化（PT 初始化、队列扩展）进一步调高了性能，将 OpenMP 加速比提升至 1.67 倍，同时提高了 SIMD 并行效果。

进一步通过查阅资料了解了并行-向量化协同瓶颈：SIMD 哈希与多线程会产生内存带宽竞争可能会影响加速效果，需要进行合理的颗粒度划分，以及探索 GPU 异构加速的可能。

GitHub 代码仓库：https://github.com/X-u-Y-a-n-g/parallel

参考文献

[1] BLAKE3 Development Team. Performance degradation with many threads, 2021. Accessed: 2025-05-26.

[2] Stack Overflow User. SIMD hurts performance when running with multiple threads, 2023. Accessed: 2025-05-26.

[3] Ming Xu, Shenghao Zhang, Kai Zhang, Haodong Zhang, Junjie Zhang, Jitao Yu, Luwei Cheng, and Weili Han. Using parallel techniques to accelerate pcfg-based password cracking attacks. *IEEE Transactions on Dependable and Secure Computing*, 22(1):123–137, 2025. Accessed: 2025-05-26.

[4] 强化学习曾小健. Simd 实现条件分支的并行计算及其局限. CSDN 博客, 2025.

[5] 腾讯云小微. 并行嵌套循环中的数据竞争, Jul 2022. [Online; accessed 2025 年 5 月 29 日].