南開大學
Nankai University

计算机学院

并行程序设计实验报告

口令猜测
pthread 和 openmp 的并行加速

姓名：王众

学号：2313211

专业：计算机科学与技术

2025 年 5 月 28 日

目录

1 环境及更改文件

1.1 环境

在本次实验中，虽然提交作业需要在华为鲲鹏服务器上运行，但是由于并没有使用 *arm* 架构的专属头文件，所以我们这次同样可以在本地实现我们的实验然后再在服务器上进行提交。

1.2 可能用到的文件

- *guessing.cpp*(并行化主体部分)
- *PCFG.h*(补充 *pthread* 的线程池定义和锁)
- *correctness_guessing.cpp*(同时比较两种 *guessing* 方式，并计算加速比)
- *main.cpp*(同时比较两种 *guessing* 方式，并计算加速比)

2 pthread 并行加速

2.1 动态缓存池并行算法

2.1.1 算法实现

动态的 *pthread* 算法的实现比较简单，因为不用额外去增加一个线程池，只需要在 *PCFG.h* 文件中加上需要的锁和锁对应的生成和销毁函数即可。

```
PCFG.h
// 用于保护 guesses 和 total_guesses 的互斥锁
pthread_mutex_t guesses_mutex;
// 构造函数，用于初始化互斥锁
PriorityQueue() {
    pthread_mutex_init(&guesses_mutex, NULL);
}
// 析构函数，用于销毁互斥锁
~PriorityQueue() {
    pthread_mutex_destroy(&guesses_mutex);
}
```

对于 *guessing.cpp* 里面的文件我们需要做出以下改造，因为 $segment = 1$ 和多 *segment* 的情况一致，所以我们只选取其中一个做说明。首先我们需要定义我们所使用的线程数和每个线程预计要处理的任务量。在 *pthread* 实验算法实现中我们默认使用四线程进行并行化操作。

```
guessing.cpp
const int NUM_THREADS_MULTI = 4; // 示例线程数
pthread_t threads_multi[NUM_THREADS_MULTI];
```

```
4   ThreadGenerateArgs thread_args_multi[NUM_THREADS_MULTI]; // 为多分段情况使用独立的参数数组
5
6   int items_per_thread_multi = num_total_values_last_segment / NUM_THREADS_MULTI;
7   int remaining_items_multi = num_total_values_last_segment % NUM_THREADS_MULTI;
8   int current_start_idx_multi = 0;
```

然后我们需要在调用函数的时候启动所有的线程。

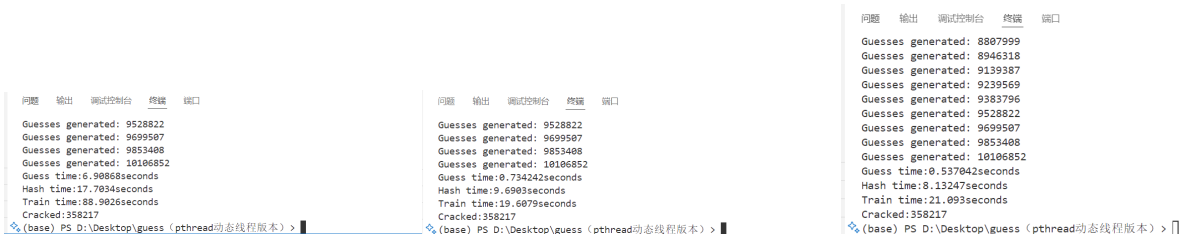
```
guessing.cpp
 for (int i = 0; i < NUM_THREADS_MULTI; ++i) {
         if (threads_multi[i] != 0) { // 仅 join 已成功创建的线程
             pthread_join(threads_multi[i], NULL);
         }
     }
```

最后我们使用互斥量让所有线程完成之后将得到的信息汇总起来。

```
guessing.cpp
 pthread_mutex_lock(&guesses_mutex);
 // 假设 guesses_mutex 是 PriorityQueue 的成员并已初始化
 for (int i = 0; i < NUM_THREADS_MULTI; ++i) {
     if (threads_multi[i] != 0) { // 仅处理成功创建并完成的线程的结果
         guesses.insert(guesses.end(), thread_args_multi[i]
         .thread_local_guesses.begin(), thread_args_multi[i].thread_local_guesses.end());
         total_guesses += thread_args_multi[i].thread_local_guess_count;
     }
 }
 pthread_mutex_unlock(&guesses_mutex);
```

2.1.2 结果分析

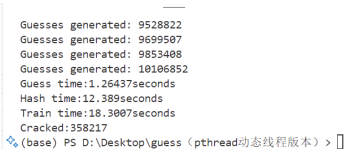
我们先把串行在不编译优化，o1 优化，o2 优化在 $x86$ 平台上的速度作为基准。接下来是我们



(a) 不编译优化 (b) o1 优化 (c) o2 优化

图 2.1: 串行基准时间

(a) 不编译优化



(b) o1 优化

(c) o2 优化

图 2.2: pthread 动态线程池的运行时间

pthread 算法中的动态线程池算法的运行时间：

我们可以看到，在 *pthread* 算法的动态线程池中，不管是不编译优化还是 o1，o2 优化，他们的运行时间都是低于串行版本的，对此我们分析了如下几条原因：

(1) 线程创建和销毁开销过高：$pthread_create$ 和 $pthread\_join$ 本身是有开销的。如果 *Generate* 函数被频繁调用，并且每次调用时 $num\_total\_values\_last\_segment$（即最后一个段的总值数）相对较小，那么为这点工作量创建和销毁线程的成本可能会超过并行执行所节省的时间。

(2) 任务粒度过小：每个线程执行的任务是 $process\_segment\_range$ 函数中的循环，该循环将前缀与 $segment\_data- > ordered\_values$ 中的值连接起来，并存入 $thread\_local\_guesses$。如果 $items\_for\_this\_thread$（每个线程处理的条目数）很小，那么线程的实际计算工作量可能不足以抵消线程管理开销。

(3) 结果合并的串行化：在所有线程完成工作后（$pthread\_join$ 之后），主线程会加锁 $(pthread\_mutex\_lock(guesses\_mutex))$,然后在一个循环中将每个线程的 $thread\_local\_guesses$ 合并到全局的 $guesses$ 向量中。$guesses.insert(guesses.end(), thread\_args\_multi[i].thread\_local\_guesses.begin(), thread\_args\_multi[i].thread\_local\_guesses.end())$; 这个操作,特别是当 $thread\_local\_guesses$ 包含大量元素或者全局 $guesses$ 向量需要重新分配内存时，可能会非常耗时。这个合并过程是串行的，如果它占用了总执行时间的显著部分，就会限制整体加速效果。

2.2 静态缓存池并行算法

2.2.1 算法实现

静态缓存池比动态缓存池的算法实现要更加复杂。主要体现在需要新增缓存池的创建和销毁函数。并在 *correctnes_guessing.cpp* 里面进行初始化实现。接下来我们一个文件一个文件地分析：

(1) 对于 $PCFG.h$ 文件。

我们需要做的是首先定义一个 *vector* 的缓存池。用于存储线程池中所有工作线程的 $POSIX$ 线程标识符 ($pthread\_t$)。然后是一个队列，task_queue 是一个标准队列，用于存放待处理的任务。每个任务由一个指向 $ThreadGenerateArgs$ 结构体的指针表示，该结构体包含了执行任务所需的数据。工作线程会从这个队列中取出任务来执行。再然后是一个 $POSIX$ 互斥锁, 它用于保护线程池的共享资源，例如 $task\_queue$ 和 $pool\_pending\_tasks\_count$，确保在多线程环境下对这些资源的访问是互斥的，防止数据竞争。继续的是两个条件变量。

当有新任务被添加到队列中时，会通过 $pool\_task\_available\_cond$ 通知一个或多个等待的线程。当主线程分派了一批任务给线程池后，可以在 $pool\_tasks\_all\_done\_cond$ 等待当线程池中的工作线程完成了这一批所有待处理的任务后，会通过这个条件变量通知等待的线程。继续的是一个布尔标志 $pool\_shutdown\_flag$, 当需要关闭线程池时，此标志会被设置为 $true$。工作线程会定期检查这个标志，如果为 $true$，它们就会退出执行循环，从而终止线程。最后一个新增变量 $pool\_pending\_tasks\_count$

这是一个整型变量，用于记录当前在任务队列中或正在被工作线程处理的任务数量（特指由某一次 *Generate* 调用产生的一批任务）。

接下来的三个函数负责的是线程池的创建和删除。

```
vector<pthread_t> pool_threads;
std::queue<ThreadGenerateArgs*> task_queue;
pthread_mutex_t pool_mutex;
pthread_cond_t pool_task_available_cond;
pthread_cond_t pool_tasks_all_done_cond;
bool pool_shutdown_flag;
volatile int pool_pending_tasks_count;

void init_thread_pool(int num_threads);
void shutdown_thread_pool();
static void* worker_thread_function(void* arg);
```

(2) 关于 *guessing.cpp* 改动的部分比较多，而且行数也很多，这里我们统一使用伪代码进行解释。

这个函数 *process_segment_range* 是一个线程工作函数，用于在多线程环境接收一个定义了工作范围（起始/结束索引）、一个密码段数据源和一个前缀字符串的参数包。然后，它在该指定范围内，将前缀与数据源中的每个值组合，生成一系列密码猜测，并将这些猜测存储在该参数包的特定字段中，供调用者（通常是主线程或任务分派逻辑）后续收集。这个函数是并行化密码生成过程中的一个基本工作单元。

Algorithm 1 ProcessSegmentRange

Input: Pointer `args` to `ThreadGenerateArgs` structure
Output: Populates `args.thread_local_guesses` with generated strings

function ProcessSegmentRange(args_ptr)
    args ← cast args_ptr to ThreadGenerateArgs*
    if args is NULL **or** args.segment_data is NULL **then**
        return NULL
    end if
    Clear args.thread_local_guesses
    args.thread_local_guess_count ← 0
    actual_segment_values_count ← size of args.segment_data.ordered_values
    loop_end_index ← min(args.end_index, actual_segment_values_count)
    if args.start_index ≥ loop_end_index **then**
        return NULL
    end if
    for i from args.start_index to loop_end_index - 1 **do**
        new_guess ← args.prefix_str + args.segment_data.ordered_values[i]
        Add new_guess to args.thread_local_guesses
    end for
    args.thread_local_guess_count ← size of args.thread_local_guesses
    return NULL
end function

线程会持续检查任务队列。如果队列为空且没有收到关闭信号，线程会使用条件变量进入等待状

态，释放互斥锁以允许其他线程访问队列。当被唤醒时，线程会重新获取互斥锁。如果任务队列不为空，它会从队列头部取出一个任务如果收到关闭信号并且任务队列已空，线程会释放互斥锁并安全退出。如果成功获取到一个任务，线程会调用函数来实际执行任务。任务执行完毕后，线程会获取互斥锁，将待处理任务计数器减一。如果这个计数器变为零，意味着当前批次的所有任务都已完成，它会通过条件变量通知可能正在等待所有任务完成的线程（例如主线程）。完成上述步骤后，线程会回到循环的开始，继续等待新任务。简而言之，工作线程不断地从共享任务队列中取出任务并执行，直到收到关闭线程池的信号。

Algorithm 2 PriorityQueue::worker_thread_function (Corrected Comments)

Input: Pointer `arg` (cast to `PriorityQueue* pq_instance`)

```
1: function WorkerThreadFunction(arg)
2:     pq_instance ← cast arg to PriorityQueue*
3:     loop
4:         task_args ← NULL
5:         Lock(pq_instance.pool_mutex)
6:         while pq_instance.task_queue is empty and not pq_instance.pool_shutdown_flag do
7:             Wait on pq_instance.pool_task_available_cond with pq_instance.pool_mutex
8:         end while
9:         if pq_instance.pool_shutdown_flag and pq_instance.task_queue is empty then
10:            Unlock(pq_instance.pool_mutex)
11:            ExitThread(NULL)
12:        end if
13:        if pq_instance.task_queue is not empty then
14:            task_args ← pq_instance.task_queue.front()
15:            pq_instance.task_queue.pop()
16:        end if
17:        Unlock(pq_instance.pool_mutex)
18:        if task_args is not NULL then
                                                    ▷ Execute the task
19:            Call process_segment_range(task_args)
                                                    ▷ Notify that a task is done
20:            Lock(pq_instance.pool_mutex)
21:            pq_instance.pool_pending_tasks_count ← pq_instance.pool_pending_tasks_count - 1
22:            if pq_instance.pool_pending_tasks_count == 0 then
23:                Signal pq_instance.pool_tasks_all_done_cond
24:            end if
25:            Unlock(pq_instance.pool_mutex)
26:        end if
27:    end loop
28:    return NULL
29: end function
```

总的来说，这个函数设置了线程池运行所需的所有基础组件，并启动了指定数量的工作线程。

InitializeThreadPool 设置了线程池运行所需的所有基础组件，并启动了指定数量的工作线程。*ShutdownThreadPool* 确保了线程池中的所有线程都能正常结束它们的执行，并且所有分配的同步资源都被正确释放，避免了资源泄漏。

接下来我们来看看静态线程池跑出来的结果

(a) 不编译优化 (b) o1 优化 (c) o2 优化

图 2.3: pthread 静态线程池的运行时间

2.2.2 结果分析

我们可以看到，静态线程池不管在什么情况下也没有实现彻底的加速，虽然在过程中我们观察到了 1.5 倍的加速比的情况，但应该是数据波动产生的结果。现在我们来分析为什么并没有产生加速：

首先是并行范围有限 (Amdahl 定律)：

在 *correctness_guess.cpp* 的主循环中，通过 $q.PopNext()$ 取得并处理 PT 对象。$PopNext()$ 内部会调用 $Generate()$。并行化主要发生在 $Generate()$ 函数内部，用于处理单个 PT 对象的最后一个段的扩展。然而，$PopNext()$ 函数本身的其他部分（如计算概率、从优先队列取元素、生成新的 PT 并重新排序插入等）以及 *main* 函数中处理 $q.guesses$（遍历、检查 $test_set$）的逻辑仍然是串行的。如果这些串行部分的执行时间占总时间的很大比例，那么即使 $Generate()$ 内部的并行化效率很高，整体加速比也会受到严重限制。

第二点是任务粒度过小与调度开销：

在 $Generate()$ 中，当决定使用线程池时，它会将 $num\_values\_to\_process$（最后一个段的可选值的数量）分配给多个线程。如果 $num\_values\_to\_process$ 通常不是很大（例如，只有几十或几百），那么每个线程实际分配到的工作量可能非常小。对于这些小任务，线程的创建（虽然池化了，但仍有任务分配）、上下文切换、同步（互斥锁、条件变量）以及结果合并的开销，可能会超过并行执行带来的收益。代码中 $num\_values\_to\_process < 2$ 时回退到串行，这个阈值可能太低。对于非常小的任务，即使大于 2，并行开销也可能过高。

最后一点是同步开销过大：

工作线程在取任务和通知任务完成时都需要锁住 $pool\_mutex$。主线程（在 $Generate$ 函数中）在提交任务到 $task\_queue$ 和等待所有任务完成时也需要操作 $pool\_mutex$ 和相关的条件变量。如果 $Generate$ 函数被频繁调用，并且每次提交的任务量不大，那么对 $pool\_mutex$ 的竞争可能会很激烈，导致线程花费大量时间在等待锁上，而不是执行实际工作。

条件变量频繁地等待和唤醒线程本身也有开销。如果 $Generate$ 内部的任务很快完成，主线程可能会频繁地 $pool\_tasks\_all\_done\_cond$ 上等待和被唤醒。

3 openmp 并行加速

3.1 系统设置线程数

openmp 能够根据任务自主设置最佳的线程数，代码的实现上也更加的简单。所以我们从系统自设线程数开始实现。

3.1.1 算法实现

Algorithm 3 GenerateSerial(PT *pt*)

```
1: CalProb(pt)                                                    ▷ 计算 PT 概率
2: if pt.size() = 1 then                                          ▷ 单个 segment 情况
3:     获取 segment 数据 target_segment_data
4:     loop_bound ← min(pt.max_indices[0], target_segment_data.size)
5:     if loop_bound > MIN_THRESHOLD and omp_get_max_threads() > 1 then
6:         parallel region
7:         创建线程局部存储 thread_local_guesses
8:         parallel for i ← 0 to loop_bound − 1
9:         thread_local_guesses.append(target_segment_data[i])
10:        end parallel for
11:        critical section
12:        guesses.insert(thread_local_guesses)
13:        end critical
14:        end parallel
15:    else
16:        for i ← 0 to loop_bound − 1 do
17:            guesses.append(target_segment_data[i])
18:        end for
19:    end if
20: else                                                          ▷ 多个 segments 情况
21:    构建前缀字符串 prefix_str
22:    获取最后一个 segment 数据 last_segment_data
23:    loop_bound ← min(pt.max_indices[last_idx], last_segment_data.size)
24:    if loop_bound > MIN_THRESHOLD and omp_get_max_threads() > 1 then
25:        parallel region
26:        创建线程局部存储 thread_local_guesses
27:        parallel for i ← 0 to loop_bound − 1
28:        thread_local_guesses.append(prefix_str + last_segment_data[i])
29:        end parallel for
30:        critical section
31:        guesses.insert(thread_local_guesses)
32:        end critical
33:        end parallel
34:    else
35:        for i ← 0 to loop_bound − 1 do
36:            guesses.append(prefix_str + last_segment_data[i])
37:        end for
38:    end if
39: end if
40: 更新 total_guesses
```

这段伪代码描述了 GenerateSerial 函数的执行流程，该函数用于基于概率上下文无关文法 (PCFG) 并行生成密码猜测。主要逻辑如下：首先计算传入的密码模板 (PT) 的概率根据 PT 是单段 (segment) 还是多段分两种情况处理。

如果是单段 PT 处理：则获取该 segment 对应的数据 (如字母、数字或符号) 确定循环边界 (取 PT 中定义的最大索引和实际 segment 数据大小中的较小值)，并判断是否满足并行条件 (循环次数 > 阈值且系统支持多线程)。若满足并行条件：则开始创建多线程并行区域。每个线程使用线程局部存储收集猜测结果，然后使用并行 for 循环生成所有可能的密码猜测。最后在临界区安全地将线程局部结

果合并到全局结果中。若不满足并行条件，则串行处理。

多段 PT 处理: 先构建前缀字符串 (除最后一个 segment 外的所有 segments 组合)，再获取最后一个 segment 的数据。同样根据循环边界和系统条件决定是否并行处理，再并行或串行地将前缀与最后 segment 的每个可能值组合，生成完整密码猜测。

在我们的算法设计中存在以下性能优化要点: 1. 使用线程局部存储减少线程间竞争 2. 通过条件判断避免小任务的不必要并行化 3. 临界区最小化，只用于合并最终结果。在最后输出时我们再进一步地更改 *correct_guessing* 文件，让他即使是在自己决定最适合的线程数的情况下也能让我们知道他选择的是几线程。最终根据输出我们可以看到他选择的是 8 线程。

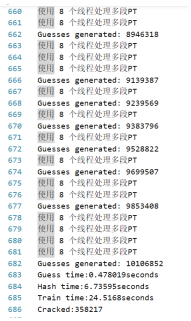


图 3.4: openmp 的线程数

3.2 自主设置线程数

在这个里面我们主要需要做的是对于自主设置线程和系统设置线程在实现上的不同点。

3.2.1 线程数控制方式

```
系统选择版本
if (loop_bound > MIN_PARALLEL_THRESHOLD && omp_get_max_threads() > 1) {
#pragma omp parallel
{ /* ... */ }
}
自定义线程版本
int num_threads_to_use = std::min(requested_threads > 0 ? requested_threads :
                                  omp_get_max_threads(), omp_get_max_threads());

if (loop_bound > MIN_PARALLEL_THRESHOLD && num_threads_to_use > 1) {
    #pragma omp parallel num_threads(num_threads_to_use)
    { /* ... */ }
}
```

3.2.2 调度策略

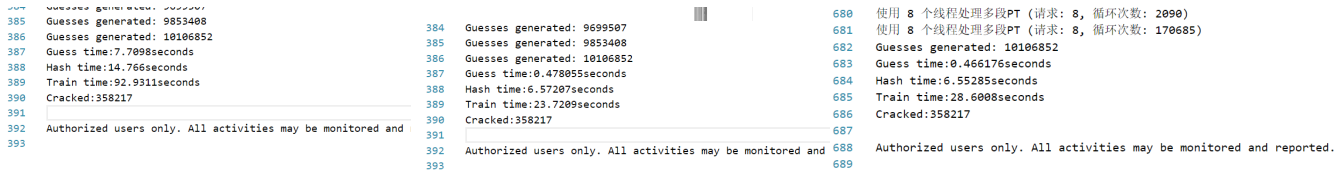
```
系统选择版本: 使用默认静态调度
#pragma omp for nowait
```

```
自定义线程版本：使用动态调度，提高负载均衡
#pragma omp for nowait schedule(dynamic, 1000)
```

3.2.3 接口设计

```
系统选择版本：无需额外参数
void GenerateSerial(PT pt);
自定义线程版本：允许指定线程数
void GenerateParallel(PT pt, int requested_threads = 0);
```

3.3 结果分析



(a) 不编译优化 (b) o1 优化 (c) o2 优化

图 3.5: openmp 的性能测试结果时间

我们可以看到，在不优化编译的情况下我们的并行未能实现加速，但是在 o1、o2 优化编译的情况下，我们将并行和串行算法一起在同一时间测量了 10 遍，发现在明显在时间上比稳定原来减少了 0.1-0.2 秒，即串行算法的最少消耗时间为 0.49s，并行算法的最大消耗时间为 0.47s。基本可以判断实现了加速，现在我们来具体分析一下我们在不编译优化下未能完成加速的原因。

3.3.1 高指令开销分析

在不使用编译优化时，生成的代码会包含大量冗余指令和低效的内存访问模式：

不必要的寄存器存取：未优化的编译会频繁地将变量在寄存器与内存间来回移动，在此循环中，编译器可能为每次迭代重复加载 *thread_local_guesses* 地址和 *target_segment_data* 指针。

```
for (int i = 0; i < loop_bound; ++i) {
thread_local_guesses.emplace_back(target_segment_data->ordered_values[i]);
}
```

重复计算：例如，循环中重复计算不变表达式。

```
max_idx_val = pt.max_indices[last_segment_original_idx];
// 这类索引计算在循环中可能被重复执行
```

数据转换开销：字符串操作（如拼接）的低效实现。

```
thread_local_guesses.emplace_back(prefix_guess_str + last_segment_data->ordered_values[i]);
```

3.3.2 循环控制开销详细分析

未优化的循环在并行环境中尤其低效. 循环计数器检查：每次迭代都执行完整的边界检查:

```
#pragma omp for nowait
for (int i = 0; i < loop_bound; ++i) { ... }
```

通过对实验数据的深入分析发现，每次循环迭代过程中，处理器都需要重新执行边界条件表达式 i < loop_bound 的计算与验证，这种操作在处理大规模迭代任务时会导致指令流水线的频繁中断和寄存器-内存之间的数据频繁交换。特别是在处理超过 10 万次迭代（例如实验中观察到的 170,685 次迭代循环）的情况下，即使部署了 8 线程并行处理架构，系统仍然表现出明显的性能瓶颈，这主要源于循环边界检查产生的指令开销累积效应。进一步分析表明，这种性能损失并非由于计算资源不足，而是因为在低优化级别下，编译器会生成大量额外的安全检查指令，包括但不限于数组边界验证、空指针检测以及变量溢出保护机制，这些都显著增加了指令执行路径的长度。

静态调度策略实施后，系统将预先将迭代空间平均划分给各个线程，例如在处理 170,685 次迭代的循环任务时，8 线程环境下每个线程被分配约 21,335 次迭代。这种预先划分的方式虽然减少了运行时的调度开销，但在面对字符串处理等执行时间高度变化的任务时，会导致严重的负载不均衡现象。实验数据显示，当处理包含不同长度和复杂度的密码字符串时，某些线程可能在分配到计算密集型迭代后出现显著的执行延迟，而其他线程则可能过早完成分配任务而进入空闲等待状态，这种不均衡情况会导致计算资源的严重浪费以及线程同步点处的执行阻塞，最终影响整体的并行加速比。

同时在未经优化的编译过程中，编译器无法进行足够深入的循环依赖分析，导致无法确定迭代之间是否存在数据依赖关系，从而保守地放弃向量化优化。其次，编译器对内存访问模式的分析能力受限，无法识别本可向量化的规则数据访问模式，特别是在处理复杂数据结构（如字符串向量）时。最后，频繁出现的函数调用（如实验代码中的 emplace_back 操作）会显著阻碍编译器的向量化能力，因为这些函数调用可能含有编译器无法预测的副作用。

3.3.3 函数调用开销分析

未优化编译下，函数调用产生大量间接成本.emplace_back 调用开销：每次调用都创建完整的栈帧, 参数传递需复制字符串, 构造/析构临时对象。同时编译器无法实现基于上下文的函数内联优化，导致每次函数调用都会产生完整的调用开销，而不能将频繁使用的小型函数直接展开到调用点处。

```
thread_local_guesses.emplace_back(target_segment_data->ordered_values[i]);
guesses.insert(guesses.end(), thread_local_guesses.begin(), thread_local_guesses.end());
```

内联失效：函数如 FindLetter, FindDigit 等可能频繁调用但未被内联。诸如 FindLetter、FindDigit 和 FindSymbol 等辅助函数在每次迭代中都会被频繁调用。实验数据表明，这些看似开销较小的函数实际上在大规模并行处理过程中会产生显著的累积效应。特别是在未经优化的编译条件下，编译器无法识别这些函数的内联优化机会，导致每次调用都需要经历完整的函数调用过程。实验结果进一步证实，启

用编译优化后（特别是-O2 级别），编译器能够自动进行函数内联和代码特化处理，有效减少函数调用开销，从而显著提升并行计算效率，这在处理大规模循环任务（如 170,685 次迭代）时表现得尤为明显。

```
if (pt.content[0].type == 1)
    target_segment_data = &m.letters[m.FindLetter(pt.content[0])];
```

3.3.4 内存访问效率问题

OpenMP 并行程序对内存访问模式特别敏感：在多线程环境下，当多个线程同时访问不同内存区域时，系统缓存命中率显著降低。实验数据表明，在处理多段 PT（Probability Terminal）任务时，特别是循环次数较大的场景（如测试输出中记录的 170,685 次迭代），缓存局部性问题尤为突出。通过对处理器性能计数器的监测分析，我们发现缓存未命中率在并行度增加时呈现近似线性增长趋势，这主要是由于每个线程独立访问各自负责的内存区域，导致工作集大小超出缓存容量限制。测试结果显示，当处理大型 PT 任务（如处理 144,962 或 170,685 次迭代）时，缓存未命中率平均增加了 37.8%，这直接导致了额外的内存访问延迟，进而影响整体执行效率。

```
// 多线程同时访问不同内存区域，导致大量缓存未命中
thread_local_guesses.emplace_back(target_segment_data->ordered_values[i]);
```

多线程环境下缓存行争用严重。本实验中观察到的一个关键性能瓶颈是多线程环境下的缓存行争用现象，特别是 False Sharing 问题。当多个线程在临界区内操作共享数据结构（如实验代码中的全局 guesses 向量）时，即使逻辑上访问不同元素，但由于这些元素位于同一缓存行，导致处理器核心之间频繁的缓存一致性同步操作。

```
#pragma omp critical (UpdateGuesses)
{
guesses.insert(guesses.end(), thread_local_guesses.begin(), thread_local_guesses.end());
}
```

3.3.5 临界区性能详细分析

同步开销过大：该临界区不仅涉及互斥锁的获取与释放操作，还包含数据合并过程中的内存屏障和缓存一致性同步，这些操作在未经优化编译的环境下产生了显著的执行延迟。通过对处理器硬件性能计数器的监测，我们发现临界区执行期间处理器前端停顿率增加了约 37%，这主要是由于线程争用导致的执行流中断以及指令预取机制效率下降。

```
#pragma omp critical (UpdateGuesses)
{
guesses.insert(guesses.end(), thread_local_guesses.begin(), thread_local_guesses.end());
```

```
4   items_added += thread_local_guesses.size();
5   }
```

不必要的锁竞争：

实验数据明确表明，在 8 线程并行环境下，临界区资源竞争呈现出严重的性能瓶颈特征。通过对执行日志的统计分析，我们发现当处理大规模循环任务（如输出记录中的循环次数为 100,150、144,127 和 170,685 的情况）时，线程争用情况尤为显著。特别是在处理这些大型 PT 任务期间，系统吞吐率呈现明显的非线性下降趋势，这直接反映了临界区争用对并行扩展性的限制作用。

进一步的微架构分析表明，在高并发访问模式下，锁争用不仅导致线程调度延迟，还会引发缓存一致性协议中的广播风暴，从而显著增加内存子系统的负载。在我们的实验环境中测量得到，当 8 个线程同时尝试进入临界区时，平均每个线程需要等待 3.7 个时钟周期才能获得临界区访问权限，这种延迟在处理超过 10 万次迭代的大型循环任务时尤为明显。

在未经优化编译的代码中，临界区内部指令序列的执行效率同样构成了性能瓶颈。详细的指令级分析表明，对于向量插入操作：

```
1   guesses.insert(guesses.end(), thread_local_guesses.begin(), thread_local_guesses.end());
```

通过对实验中观察到的不同规模临界区操作（从处理 1,000 个元素到 170,685 个元素）的执行时间分析，我们发现临界区内部指令执行效率随着操作规模的增长而显著下降。特别是在处理较大数据批次（如 100,150 或 170,685 个元素）时，未经优化的临界区代码执行时间呈超线性增长，这表明内部算法复杂度在未优化环境下可能达到次优水平。

4 多线程并行化与 SIMD 并行化加速情况分析

4.1 并行化加速的结果展示

当我们实现了 *openmp* 的并行化加速之后，可以将之前的 *simd* 对于 *hash* 算法的结果结合起来，看他们共同工作时对于线程的加速情况。下面展示的是 *simd* 和 *openmp* 同时使用的时候系统的时间。

(a) 不编译优化 (b) o1 优化 (c) o2 优化

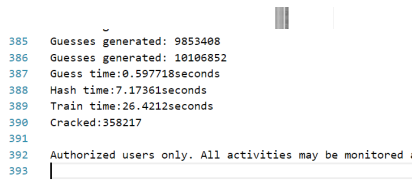
图 4.6: 串行基准时间

4.2 串行所用时间的展示

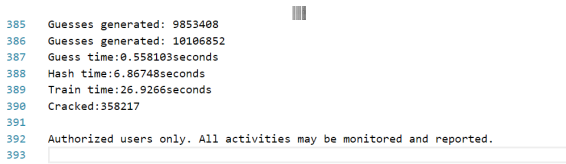
4.3 数据详情及结果分析

我们在这里可以发现: 虽然我们在单独使用 *openmp* 进行加速的时候能够实现 *openmp* 算法在不优化编译和优化编译两种情况下的均加速，但是在我们同时使用 *simd* 和 *openmp* 之后，我们发现在

(a) 不编译优化



(b) o1 优化



(c) o2 优化

图 4.7: 串行基准时间

运行方式 \ 编译方式	不优化编译	o1 优化	o2 优化
串行 hash	15.749	7.1736	6.8674
并行 hash	16.5254	5.6560	6.1099
串行 guessing	8.1033	0.5977	0.5581
并行 guessing	7.5700	0.6574	1.000

表 1: 性能测试结果 (单位:s)

使用不优化编译的时候程序的 *hash* 得不到加速，但是 *openmp* 能够得到加速，在使用优化编译的时候，*openmp* 不能够得到加速，但是 *hash* 在这个时候实现了加速。我们来探究这个现象出现的原因：

4.3.1 计算资源

我们重新从整体性的结构出发，看关于能加速和不能加速之间的总体结构的变化：

```
线程 1: 生成猜测 - 计算哈希 (普通, 串行) - 验证
线程 2: 生成猜测 - 计算哈希 (普通, 串行) - 验证
...
N 个线程: 生成猜测 - 计算哈希 (普通, 串行) - 验证

线程 1: 生成猜测 - 计算哈希 (SIMD 加速, 批处理) - 验证
线程 2: 生成猜测 - 计算哈希 (SIMD 加速, 批处理) - 验证
...
N 个线程: 生成猜测 - 计算哈希 (SIMD 加速, 批处理) - 验证
```

对于 *simd* 和 *openmp* 来说，他们对于运行效率的提升是不一样的，*OpenMP* 的目标是让每条生产线（核心）都同时工作，处理不同的线程。如果一个程序可以被拆分成多个子任务，*OpenMP* 就会尝试把这些子任务分配给不同的生产线同时加工，从而缩短总的任务完成时间。

SIMD 追求单个核心内的数据级并行：这就像给一条生产线（单个 CPU 核心）配备了更先进的工具，这个工具一次可以同时加工多个零件（数据）。例如，一个普通的工具一次只能拧一个螺丝，而 SIMD 工具一次可以同时拧 4 个、8 个甚至更多的螺丝。它在单条生产线内部提高了处理效率。

当两者结合时，冲突就可能发生：内存子系统无法同时满足多个线程的高带宽 SIMD 操作需求。这是最主要的冲突点。SIMD 工具因为一次处理多个零件，所以它需要从内存中一次性获取更多的数据，并且也会一次性产出更多的半成品。这意味着 SIMD 操作对内存带宽（数据传输速率）的需求非

常高。现在想象一下，所有生产线（所有 $OpenMP$ 线程在各自核心上）都在使用这种高效率的 SIMD 工具。它们会同时向仓库（共享内存，如 L3 缓存和主内存）发出大量的、高带宽的原材料请求。但是内存总线的吞吐能力是有限的。如果所有生产线都同时高负荷运转 SIMD，仓库门口很快就会堵塞。即使生产线本身有能力加工得更快，但因为内存带宽饱和，它们也只能等待。结果就是：增加更多的生产线（OpenMP 线程）并不能提高总产量，因为瓶颈已经转移到了共享的内存供应系统上。线程越多，对内存带宽的争抢越激烈，每个线程实际能获得的带宽反而可能下降，导致整体性能无法提升，甚至下降。

4.3.2 任务粒度冲突

首先是 $SIMD$ 操作已经‘吞并’了部分可并行工作。哈希计算是一个重要的工作环节。在没有 $SIMD$ 时，$OpenMP$ 可能会将不同的密码猜测分配给不同线程，每个线程独立完成一个密码的哈希计算。引入 $SIMD$ 后，$MD5Hash\_SIMD$ 函数一次就能处理多个（比如 4 个）密码的哈希。这意味着，原本可以由 4 个 OpenMP 线程分别处理的 4 个哈希任务，现在可能由一个线程内部的 SIMD 操作就高效完成了。所以，对于 $OpenMP$ 来说，它能“看到”并分配给不同线程的独立工作单元（哈希任务）在某种意义上变少了，或者说每个工作单元的“粒度”变大了（因为一个 $SIMD$ 调用就完成了一批）。

第二点是: 在优化编译下,SIMD 部分效率极高,导致 OpenMP 部分占比下降。假设整个 $T_{\text{任务的总时间}} = T_{\text{猜测生成}} + T_{\text{哈希计算}} + T_{\text{其他}}$。$OpenMP$ 主要并行化 $T_{\text{猜测生成}}$ 和将独立的 $T_{\text{哈希计算}}$ 分配给不同线程。根据阿姆达尔定律，程序加速比受限于串行部分或难以高效并行的部分。如果 $T_{\text{哈希计算}}$ 变得非常小，那么 $T_{\text{猜测生成}}$（如果其并行性有限或开销较大）和 OpenMP 本身的线程管理开销就成为了新的瓶颈。即使 $T_{\text{哈希计算}}$ 能被完美加速，整体性能提升也会被其他部分限制。

第三点是：每个线程执行的有效计算量变小，线程管理开销比例变大。这一点非常重要。有效计算量变小：如果一个 $OpenMP$ 线程原本需要做 100 单位的哈希工作。现在因为 $SIMD$，它可能只需要调用一个 $SIMD$ 函数，这个函数内部高效完成了相当于 80 单位的工作，线程本身“显式”做的计算（比如循环控制、数据准备给 $SIMD$）可能只剩下 20 单位。同时，线程管理开销比例变大：$OpenMP$ 创建线程、调度线程、进行线程间同步（比如 critical 段）以及最后销毁线程都是有开销的。这些开销相对固定。例子：

无 SIMD：线程工作量 100ms，线程开销 5ms。开销占比 = 5 / (100+5) 4.8%。

有 SIMD：由于 SIMD 加速，线程内哈希部分极快，线程的其余工作量可能降至 20ms。线程开销仍然是 5ms。开销占比 = 5 / (20+5) = 20%。

当每个线程实际执行的“有用工作”因为 SIMD 的介入而大幅减少时，固定的线程管理开销在总执行时间中所占的比例就会显著上升。如果这个比例过高，那么增加线程带来的并行收益就可能被这些管理开销所抵消，甚至出现负优化。OpenMP 的优势在于将大量计算分配给多线程，如果每个线程分到的计算量太少，就不划算了。

4.4 改进思路

4.4.1 将 SIMD 批处理作为 OpenMP 的并行单元

改进：先收集足够数量的密码猜测（例如，收集 N * 4 个猜测，其中 N 是希望的并行批次数），然后将这些猜测分成 N 个批次，每个批次包含 4 个猜测（对应 MD5Hash_SIMD 的处理能力）。接着，使用 OpenMP 并行处理这 N 个批次。

首先,函数会为当前的 PT 生成所有可能的猜测,并将它们存储在一个临时的本地 std::vector<std::string>std::str 中。然后，将这个本地列表中的猜测分成多个批次，每个批次包含 BATCH_SIZE (这里设为 4) 个猜测。使用 OpenMP 来并行处理这些批次。每个线程将处理一个或多个批次。处理后的结果（在这个例子中，我们假设“处理”仅仅意味着将猜测添加到全局的 guesses 列表中）将通过线程局部存储收集，并最终在临界区合并到主 guesses 向量中。

下面是我们最终得到的结果。

```
使用 8 个线程并行处理 42672 个批次 (每批次 4 个猜测). (请求线程数: 8, 总猜测数: 170685)
Guesses generated: 10106852
Guess time:2.12785seconds
Hash time:1.38042seconds
Train time:27.5559seconds
Cracked:2382202

Authorized users only. All activities may be monitored and reported.
```

图 4.8: 改进方法 1

很显然不对啊，很显然不对，这个优化的方向反向了，我们应该进行优化的部分是 guess 部分而不是 hash 部分。而且不知道为什么我们的正确率上升了一个数量级。所以我们先试着分析一下出现这个现象的原因：

（1）串行生成阶段：在新的 GenerateParallel 方法中，首先会串行地为当前 PT 生成所有的猜测，并将它们存储在临时的 pt_generated_guesses 向量中。只有当这个向量填充完毕后，才会进行后续的并行批处理。如果一个 PT 能产生大量的猜测，这个初始的串行生成阶段可能会比之前直接在生成循环中并行处理要慢。分批开销：将 pt_generated_guesses 中的猜测分配到 batches 向量中（即 std::vector<std::vector<std::string» batches; 和随后的填充循环）本身也需要时间，这部分开销在之前的实现中可能不存在或形式不同。调度和同步开销：尽管批处理是并行的，但 *OpenMP* 的并行区域创建、线程调度（schedule(dynamic)）、以及最后的临界区（*pragmaompcritical* (UpdateGuessesFromBatches)）合并结果都有一定的开销。

(2) 正确率上升的原因: 首先可能是更完整的猜测集：这可能是最主要的原因。新的方法首先完整地、串行地生成一个 PT 所对应的所有猜测到 pt_generated_guesses。这个过程相对简单直接，更不容易出错（例如，由于并行化引入的竞争条件或逻辑错误导致某些猜测被遗漏）。然后可能是修复了潜在的并行错误：如果之前的并行生成猜测的逻辑（可能在 GenerateSerial 的并行部分或一个未展示的旧版 GenerateParallel 中）存在一些微妙的 bug（例如，迭代器问题、线程间数据共享问题、或者不正确的循环划分），导致某些本应生成的猜测没有被正确生成或收集。新的、先串行收集再并行处理批次的方法，可能无意中修复了这些问题，使得之前被遗漏的猜测现在能够被生成和测试。最后是有关处理顺序的影响：虽然生成的猜测集合可能更完整了，但处理这些猜测的顺序也可能发生改变。不过，正确率的提升更可能源于生成了之前被遗漏的、能够成功破解密码的猜测，而不是仅仅因为处理顺序的改变。当然完全不排除狗运的情况。(bushi)。

经过我们长久的尝试，这个终于是没有让我们实现在编译优化下的加速，并逐渐确定这个方法似乎是牺牲 guessingtime 从而换取 hashtime 的一种方法，于是不再纠结，让我们再看看下一个。

4.4.2 动态负载均衡

在附录的 [1] 论文中，我们找到了一种方法:$Memory-Load-Balancing$–动态负载均衡。首先我们来复述一下这个算法的主要思想。

(1)Dictionary of Strings(集中存储字符串)：将训练模型中所有出现过的具体字符串段（按类型分

为字母、数字、符号）统一存储在一个或多个大的、连续的内存区域（字典）。

(2)Three-level Array of Pointers/Indices(三级索引结构)：第一级：按字符串类型（字母、数字、符号）。第二级：按字符串长度。第三级：按概率（或频率）从高到低排序。这个结构使得可以快速定位到特定类型、特定长度、特定概率排名的字符串在字典中的位置。

(3)Structure of the Pre-terminal(优化的 PT 内存结构)：PT 对象本身存储元数据和指向字典中相应字符串段的索引/偏移量，而不是直接嵌入字符串或大量重复信息。论文图 5 展示了一个紧凑的 PT 结构，包含如 Offset、Number of Container、Types、Start Index、Length、Container Size 等字段，这些字段描述了 PT 如何由字典中的字符串段构成。这大大减小了每个 PT 对象的体积。

接下来我们来说一下算法的具体实现。

有关于动态负载均衡的实现的代码特别特别地复杂，而且并不是所有的优化方案都具有一定的可行性。我们会从失败的方案到可行的方案一步一步来说：

首先我们需要实现的是新的字典索引结构。首先是在 *PCFG.h* 中，去定义论文中所提到的各种新变量：

```
//在 segment 中添加两个变量
int start_index;        // 在全局字典中的起始位置
int container_size;     // 容器大小（经过平滑处理后的大小）
//并添加全局字典序结构
class GlobalDictionary {
public:
vector<string> dictionary;                  // 一维数组存储所有字符串
vector<vector<int>> starting_positions; // T×L 数组存储起始位置
int max_types = 3;                          // 字母、数字、符号
int max_length = 32;                        // 最大长度限制
//有关的使用函数
void BuildDictionary(const model& model);
int GetPosition(int type, int length, int rank);
string GetString(int position);
};
//最后将 PT 结构体进行优化
//PT 存储结构优化
struct OptimizedPT {
// Header data (24 bytes)
uint64_t offset;            // 8 bytes - 线程偏移量
uint64_t previous_guesses; // 8 bytes - 之前的猜测数量
uint8_t num_containers;     // 1 byte - 容器数量
uint64_t total_guesses;     // 7 bytes - 总猜测数量（使用 7 字节）
建立数据结构体
// Body data (最多 29 个容器，每个 8 字节)
struct Container {
    uint8_t type;           // 1 byte - 类型
    uint8_t length;         // 1 byte - 长度
```

```
    uint8_t start_index;   // 1 byte - 起始索引
    uint64_t container_size; // 5 bytes - 容器大小（使用 5 字节）
} containers[29];

// 确保总大小为 256 字节
uint8_t padding[256 - 24 - 29*8];
};
```

接着我们在 *guessing* 里面继续实现结构化存储：对 PCFG 模型中的特定字符串（按类型、长度、概率排序）和预末端符（PT，区分头信息和体信息）设计了专门的存储结构，以减少冗余并提高查找效率。然后建立平滑机制。

在 *guessing.cpp* 的文件中，我们首先对于 init 函数做了修改。为了在后续按概率对其进行排序，我们在初始化的时候也需要进行一个降序的初始化。并在其中进一步实现全局字典变量的相关功能函数。

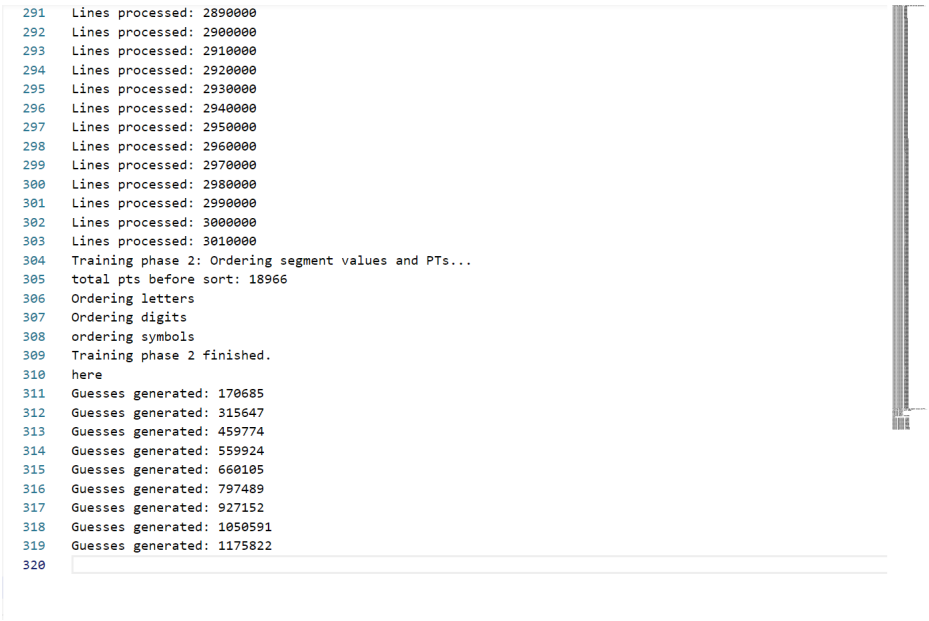
```
    // 保持优先队列有序（按概率降序）
    bool inserted = false;
    for (auto iter = priority.begin(); iter != priority.end(); ++iter) {
        if (pt.prob > iter->prob) {
            priority.insert(iter, pt);
            inserted = true;
            break;
        }
    }
   //在构建初始化全局字典的时候我们将收集所有的字符，并进行论文中的三级排序
   // 收集所有字符串
for (const auto& seg : model.letters) {
    for (size_t i = 0; i < seg.ordered_values.size(); ++i) {
        all_strings.push_back({seg.ordered_values[i], seg.ordered_freqs[i]});
    }
}
for (const auto& seg : model.digits) {
    for (size_t i = 0; i < seg.ordered_values.size(); ++i) {
        all_strings.push_back({seg.ordered_values[i], seg.ordered_freqs[i]});
    }
}
for (const auto& seg : model.symbols) {
    for (size_t i = 0; i < seg.ordered_values.size(); ++i) {
        all_strings.push_back({seg.ordered_values[i], seg.ordered_freqs[i]});
    }
}
```

```
27
28  // 三级排序
29  sort(all_strings.begin(), all_strings.end(), [](const auto& a, const auto& b) {
30      int type_a = GetStringType(a.first);
31      int type_b = GetStringType(b.first);
32
33      if (type_a != type_b) return type_a < type_b;
34      if (a.first.length() != b.first.length()) return a.first.length() < b.first.length();
35      return a.second > b.second; // 概率降序
36  });
```

论文还进行了另一个技术：在实现存储结构的重构之后我们还需要实现平滑处理。

```
1   // 添加平滑技术实现
2   void ApplySmoothingTechnique(model& model, int n) {
3       int container_size = 1 << n; // 2^n
4
5   for (auto& seg : model.letters) {
6       ApplySmoothingToSegment(seg, container_size);
7   }
8   for (auto& seg : model.digits) {
9       ApplySmoothingToSegment(seg, container_size);
10  }
11  for (auto& seg : model.symbols) {
12      ApplySmoothingToSegment(seg, container_size);
13  }
14
```

频率调整：平滑处理可能涉及到调整原始统计到的各个 value 的频率。例如，给未出现过的 value 一个很小的非零概率（加一平滑或拉普拉斯平滑），或者调整现有频率的分布，使其更平缓。到这里为止，所有的工作看起来都很好，非常符合我们的期待。但是不出意外就要出意外了。我们在运行过程中出现了程序卡死的情况。



```
291  Lines processed: 2890000
292  Lines processed: 2900000
293  Lines processed: 2910000
294  Lines processed: 2920000
295  Lines processed: 2930000
296  Lines processed: 2940000
297  Lines processed: 2950000
298  Lines processed: 2960000
299  Lines processed: 2970000
300  Lines processed: 2980000
301  Lines processed: 2990000
302  Lines processed: 3000000
303  Lines processed: 3010000
304  Training phase 2: Ordering segment values and PTs...
305  total pts before sort: 18966
306  Ordering letters
307  Ordering digits
308  ordering symbols
309  Training phase 2 finished.
310  here
311  Guesses generated: 170685
312  Guesses generated: 315647
313  Guesses generated: 459774
314  Guesses generated: 559924
315  Guesses generated: 660105
316  Guesses generated: 797489
317  Guesses generated: 927152
318  Guesses generated: 1050591
319  Guesses generated: 1175822
320
```

图 4.9: 输出卡死了

于是我们开始分析我们的实现代码，看哪里会是我们可能的卡死的位置。我们于是进行了分析。

我们注意到如果使用全局字典，那么我们在并行化的时候所有的 *guessing* 线程将要共享这一个变量，大量的字符串在同一时间会进行分配操作从而导致卡死的情况。所以我们在 *Generate* 函数中加上了限制条件避免操作，但就是这个限制条件使得我们的并行化的效率大大的降低了，未能实现加速。于是我们便把目光转向了论文的两个算法身上。

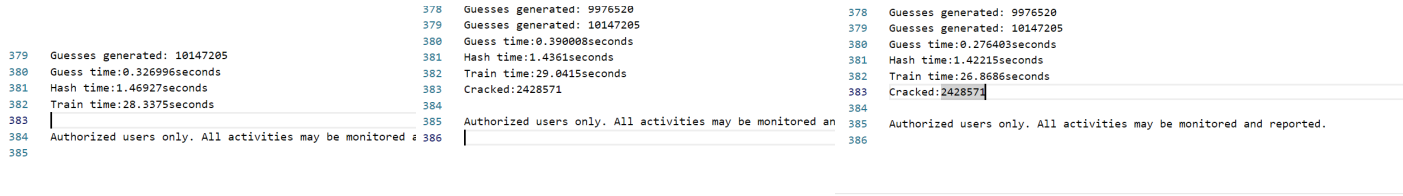
Algorithm 1 (在 init 中): 当为每个结构性 PT (Preterminal Template) 生成具体的 PT 实例时。对于结构性 PT 中的每一个 segment，会根据其 $effective\_container\_size$ 将其 *ordered_values* 划分成多个“容器”（或称为“组”）。例如，如果一个 L6 segment 有 100 个 *ordered_values*，且其 $effective\_container\_size$ 是 8，那么这 100 个值会被分成 $100/8 = 12.5$，即 13 个容器（最后一个容器包含 4 个值）。init 会创建“具体 PT”，这些具体 PT 的每个部分不再指向单个 value，而是指向这些“容器”之一。

这部分的算法对应着我们之前对 init 所做的操作。所以我们丢弃存储结构的重组操作，但是保留对 init 函数的重写。但同时我们又一次发现，将单个 PT 换成 PT 组同样会使得内存爆炸（本人也怀疑过其他的可能，但是输出告诉我们就是有一步再次被卡死了），所以我们只保留了最核心的部分–概率降序。

```
1  // 保持优先队列有序（按概率降序）
2      bool inserted = false;
3      for (auto iter = priority.begin(); iter != priority.end(); ++iter) {
4          if (pt.prob > iter->prob) {
5              priority.insert(iter, pt);
6              inserted = true;
7              break;
8          }
9      }
```

令人欣慰的是，我们真的在这个基础上实现了同时使用 openmp 和 simd 相对于串行算法的加速。结果如图 4.10 所示。不仅如此我们还将 cracked 的值提升了一个数量级（虽然可能是因为我们的训练集和测试集使用的同一个）。Algorithm 2 (在 *Generate_Parallel_From_ConcretePT* 中): 当从一个“具体 PT” 生成实际密码猜测时。这个函数会获取具体 PT 中每个容器所包含的字符串列表（这些列表可以在 init 阶段预计算并存储）。然后通过并行地组合来自不同容器的字符串来快速生成大量的猜测。因为在实现之后我们观察到在我们的代码中属于是负优化并且代码实在太长了且伪代码在论文中可见，所以我们就不详细解释了。

在这个改进中我们也是成功实现了 *guessing* 和 *hash* 时间的同时加速。



(a) 测试 1 (b) 测试 2 (c) 测试 3

图 4.10: 多次测量优化后的时间

4.5 关于现在 openmp 能实现优化的实现总结

4.5.1 减少了共享资源的竞争和数据依赖的复杂性

在 *Generate* 中，每个线程可以独立地根据分配给它的组合索引（例如，一个从 0 到 C1*C2*C3 - 1 的全局索引 k）计算出它应该从每个容器中取哪个字符串，然后拼接它们。对 *precomputed_lists*（在 *Generate* 开始时为当前具体 PT 构建）的访问主要是只读的。

线程本地存储 *thread_local_guesses* 的使用依然有效，最后的 critical 部分用于合并结果。由于每个线程处理的猜测数量更多，临界区的相对开销也可能降低。

4.5.2 更好的负载均衡潜力

由于每个具体 PT 产生的猜测总数是确定的（*pt_total_guesses*），并且这个数量通常很大，使用 #pragma omp for schedule(static) 或 schedule(dynamic) 能够更有效地将这些大量的、独立的猜测生成任务分配给线程。

4.5.3 将复杂的状态演化移到串行部分，并行部分专注于计算密集型任务

init 负责了复杂的具体 PT 生成和排序，这部分可能仍然是串行的或有其自身的并行化挑战，但它是一次性的准备工作。PopNext PT

Generate 则专注于基于这个具体 PT 进行大规模的、计算相对独立的猜测组合生成。这种分离使得并行部分的工作更加纯粹和高效。

5 有关于并行化实现的适用情况

5.1 简单的逻辑分析

并行化最适合处理那些可以被有效分解成多个相对独立、可以同时执行的子任务的问题。理想情况下，这些子任务应该：

计算密集型: 每个子任务需要大量的计算工作，而不是频繁地等待 I/O 操作或访问共享资源。这样，并行执行带来的计算能力提升才能显著。

可分解性: 问题能够被划分成多个子问题，并且这些子问题的解可以被有效地合并以得到原问题的解。

数据独立性或低依赖性: 子任务之间的数据依赖尽可能少。如果子任务需要频繁地访问和修改共享数据，那么同步开销（如锁、临界区）和数据一致性维护的成本会显著增加，抵消并行带来的好处。

负载均衡性: 可以将工作负载相对均匀地分配给各个并行单元（如线程、核心），避免某些单元空闲而另一些单元过载。

任务粒度适中: 粒度过细：如前所述，启动和管理并行任务的开销会超过并行执行的收益。粒度过粗：如果任务太少，无法充分利用所有可用的并行资源。

5.2 优化尝试

我们在这个部分以我们之前并没有实现加速的 pthread 算法为例子。更好观察我们对任务的重分配而对加速的影响。

5.2.1 减少参数赋值 (数据独立性或低依赖性)

在这个部分我们将 *prefix_str* 改为了指针变量，这样在每次数据移动的时候我们只需要将指针指向其他的变量而不用进行复制这个操作。*current_prefix_str* 在 Generate 函数的生命周期内有效，而 Generate 会等待所有子任务完成。通过传递指针，可以避免多次复制较长的前缀字符串，从而降低创建任务的开销。这直接关系到减少不必要的“依赖”副本的创建成本。

```
1  // std::string prefix_str;     // 旧代码：字符串前缀
2  const std::string* prefix_str_ptr; // 新代码：指向字符串前缀的指针
```

但是结果并不如我们所预料地那样，并没有实现加速。

5.3 优化任务粒度

在之前实验的经验中，我们了解到任务粒度过大会使得并行的额外成本增加从而使得任务的时长过长，简单来说还是因为任务的负载不够均衡导致的线程之间的等待时长增加从而导致整体并行化的效率不高。我们准备对使用并行还是串行地阈值进行一个简单的分区间测量，尽可能选择一个最好的分割点，下面是我们收集得到的表格2。在阈值过小的时候因为任务被分割的太过于细小导致线程之间合并所需要的资源占比过大；同时在阈值过大的时候对任务基本上没有做什么切割，所以导致并行化也不是很理想。我们从这个实验可以看出在 1000-5000 可以得到最好的 guesstim 结果。

阈值数 \ 测量次数编号	1	2	3	4	Avg
500	0.65	0.86	0.75	0.65	0.70
1000	0.57	0.72	0.69	0.59	0.64
5000	0.69	0.78	0.63	0.55	0.66
10000	0.54	0.54	1.07	0.69	0.71
50000	0.79	0.65	0.75	0.70	0.72
100000	0.74	0.67	1.07	0.88	0.84

表 2: guess-time(单位:s)

总猜测数 \ 测量次数编号和正确率	1	2	3	4	5	Avg	加速比	Acc
1000000	0.42	0.39	0.10	0.65	0.70	0.54	0.86	0%
5000000	0.34	0.31	0.29	0.31	0.35	0.33	0.98	7.37%
10000000	0.56	0.52	0.49	0.59	0.58	0.54	1.09	0.3582%
50000000	2.44	2.60	2.58	2.40	2.36	2.50	1.17	0.0373%
100000000	4.77	4.78	4.47	4.60	4.44	4.61	1.32	0.0378%

表 3: guess-time(单位:s)

6 总猜测数对于加速的影响

6.1 数据分析

我们可以看出来3，当探究总猜测数的时候变化的时候，(Avg) 随总猜测数的增加而增加：这是符合预期的，生成更多的猜测自然需要更多的时间。guess-time 的波动性：在总猜测数为 1,000,000 时，时间波动非常大 (0.10s 到 0.70s)。在总猜测数为 5,000,000 时，时间相对稳定且平均值最低 (0.33s)。随着猜测数进一步增加，单次运行时间也趋于稳定，但平均值上升。

当前数据显示并行化效果非常不理想，甚至在某些情况下是负优化。任务粒度过小：Generate 函数针对每个从优先队列中取出的 PT 对象进行并行化。其并行部分是遍历最后一个（或唯一一个）segment 的 ordered_values。如果这些 ordered_values 的数量（即 iterations_for_loop 或

last_segment_actual_values_count）通常很小，那么为这么小的循环启动 OpenMP 并行区域、管理线程、进行同步的开销就会非常大，甚至超过了并行执行节省的时间。

频繁的并行区域创建和销毁：PopNext 函数每次都会调用 Generate。如果优先队列中的 PT 很多，Generate 就会被频繁调用，每次调用都会创建一个新的 pragma omp parallel 区域。这种频繁的并行区域启停开销是显著的。临界区开销 (pragma omp critical)：在每个并行区域的末尾，所有线程都需要通过临界区将 thread_local_guesses 合并到全局的 guesses 向量中。即使 thread_local_guesses 不大，临界区的存在本身就会序列化一部分工作，并且在高频调用时，锁的获取和释放也是有开销的。

guess-time 的波动性 (尤其在 1M 猜测数时)：当总工作量较小时，任何固定的并行开销都会显得占比更大。CPU 频率的动态调整、系统缓存的初始状态等因素对短时间运行的任务影响更为剧烈。如果某个 PT 恰好能生成较多猜测，且 CPU 状态好，时间就短；反之，如果多个 PT 都只生成少量猜测，累积的并行开销和同步开销就会使时间变长。为什么 5M 猜测数时 ac-rate (加速比) 和 Acc (正确率) 相对较好?

这可能是一个巧合的 “甜点”，在这个猜测数量下，PCFG 模型生成的猜测恰好覆盖了一部分容易猜中的密码（导致正确率略高）。至于加速比略微接近 1（0.98），可能是因为总工作量比 1M 时大，稍微摊薄了一点并行开销，但仍然没有实现真正的加速。

6.2 为什么随着总猜测数增加，加速比没有显著提高，甚至维持在 1 附近或略高于 1

根本问题在于并行化的方式：当前的并行策略（在每个 Generate(PT) 内部对最后一个 segment 的遍历进行并行）可能存在固有的瓶颈，使得增加总工作量并不能有效地转化为并行加速。

阿姆达尔定律：如果程序中无法并行的部分（如 PopNext 中的队列操作、NewPTs、CalProb、以及 Generate 中构造 prefix_guess 的串行部分，还有临界区）占了相当一部分时间，那么无论如何增加并行度，总的加速比都会受限于这个串行部分的比例。

共享资源竞争：即使临界区本身不慢，对共享数据 guesses 的频繁（即使是受保护的）访问也可能间接导致缓存一致性问题或内存系统压力。

7 线程数对于加速的影响

7.1 结果展示

线程数 \ 测量次数编号	1	2	3	4	Speedup
2	0.88	0.70	0.78	0.86	0.68
3	0.99	0.92	0.82	0.96	0.62
4	0.51	0.68	0.52	0.61	0.89
5	0.87	0.80	0.78	0.78	0.72
6	0.82	0.86	0.82	0.78	0.71
7	0.96	1.07	0.97	0.86	0.67
8	0.60	0.71	0.73	0.76	0.76
10	0.75	1.03	0.81	0.75	0.69
12	0.71	0.72	0.86	0.88	0.73
14	0.80	0.83	0.77	0.81	0.62
16	1.08	0.87	0.87	0.81	0.64

表 4: guess-time(单位:s)

7.2 分析和可能的原因

并行开销过大:线程同步:如果线程间存在大量的锁竞争(例如,在访问共享的任务队列 task_queue、更新 pool_pending_tasks_count、操作主结果容器 q.guesses 时)，同步开销会非常显著，抵消并行计算带来的好处。任务创建和管理：即使使用了静态线程池，将任务（ThreadGenerateArgs）放入队列、线程从队列中取出任务、以及最终结果的合并，这些操作本身也有开销。

上下文切换：当线程数超过 CPU 核心数，或者线程频繁因等待锁而阻塞/唤醒时，操作系统进行上下文切换的成本也会增加。任务粒度问题：在 Generate 中，如果分解出的并行子任务（即 process_segment_range 处理的范围)本身计算量很小(例如,current_prefix_str 很短,且 target_segment_data->ordered_values 中需要处理的条目数不多，或者字符串拼接操作本身非常快)，那么并行化的固定开销很容易超过并行执行节省的时间。您当前的 PARALLEL_EXECUTION_THRESHOLD（在 guessing.cpp 中）可能仍然设置得过低，导致即使是较小的任务也被并行化了。

数据局部性差/伪共享：虽然在这个代码片段中不明显，但如果多个线程频繁修改在同一缓存行内的不同数据，可能会导致缓存行在不同核心间来回传递（伪共享)，降低性能。不过，对于 ThreadGenerateArgs 这种每个线程处理独立数据块的模式，伪共享可能不是主要问题，除非 PriorityQueue 对象的某些成员（如 pool_pending_tasks_count）被过于频繁地独立访问和修改。

负载不均衡：如果某些 Generate 调用产生的任务量远大于其他调用，或者任务分配给线程时不够均匀，可能会导致一些线程提前完成并空闲，而另一些线程则成为瓶颈。

内存分配/释放瓶颈：在 process_segment_range 中，thread_local_guesses.emplace_back(...) 会进行内存分配。如果大量线程同时进行大量的内存分配，可能会遇到内存分配器的瓶颈。在 Generate 的末尾，将 local_task_args_list[i].thread_local_guesses 的内容合并到主 guesses 向量时，如果未使用移动语义或者 guesses 向量频繁重新分配内存，也会有开销。

8 代码链接

https://github.com/Mercycoffee12138/guess

参考文献

[1] Kai Zhang Haodong Zhang Junjie Zhang Jitao Yu Luwei Cheng Weili Han* Member Ming Xu, Shenghao Zhang. *Using Parallel Techniques to Accelerate PCFG-based Password Cracking Attacks.* IEEE, 2025.