南開大學
Nankai University

计算机学院

并行程序设计报告

多线程编程

基于 pthread/openMP 加速的口令猜测算法

姓名：潘涛

学号：2314033

专业：计算机科学与技术

2025 年 5 月 29 日

目录

1 实验相关

本次实验通过多线程技术来实现对于口令猜测部分的加速实现。通过 pthread 和 openMP 对口令猜测实现了并行化。相关实现文件如下：

- correctness_guess.cpp 攻破率测试源码
- guessing.cpp 串行实现源码
- guessing_pthread.cpp pthread 优化源码
- guessing_opneMP.cpp openMP 优化源码

2 问题描述

对于一个用户的口令，对其进行猜测的基本策略是：生成一个按照概率降序排列的口令猜测词典，这个词典包括一系列用户可能选择的口令。那么，现在问题就变成了：1. 如何有效生成用户可能选择的口令；2. 如何将生成的口令按照降序进行排列。在这个选题中，使用最经典的 PCFG（Probabilistic Context-Free Grammar，概率上下文无关文法）模型来进行口令的生成，并且尝试将其并行化，以提升猜测的时间效率。口令猜测选题的框架主要分为三个部分：模型训练、口令生成、MD5 哈希值生成。在本篇中，我们将着重讨论口令生成部分的 Pthread 和 OpenMP 并行实现，以提高口令猜测的时间效率。

本次实验我们将利用训练好的 PCFG 模型进行口令猜测。猜测的生成需要使用优先队列，队列内的元素按照概率降序排列。首先，我们将每个 preterminal 中的各个 segments 用概率最高的 value 进行初始化。按照概率降序生成猜测的过程如下。只需要将优先队列队首的一个 preterminal 出队，并且取出其中的 value 值，就能生成一个对应的口令。但是，如何生成后续的口令呢？出队的 preterminal 并不会被直接删除，我们需要按照 pivot 的值，对该 preterminal 的 value 进行改造，再将生成的新的一系列（也可能没有）preterminal 按概率放回优先队列里。接下来我们来详细描述根据 pivot 值，为 preterminal 生成新 value 的过程。每次生成新的 value 时，我们规定，一次只更改一个 segment 的值，并且只更改位置下标大于 pivot 值的 segment。更改 segment 之后，还需要更新 pivot 的值。我们从模型中找到这个 segment 对应的统计据，取出其下一个概率最高的 value，实现改变 segment 的 value。重复上述过程，即可不断地按概率降序生成新的口令猜测。可以证明，这个过程不会生成重复的口令，生成的口令按概率降序，并且这个过程可以遍历模型中各 preterminal 所有可能的排列组合。

PCFG 生成口令的本质，就是按照概率降序不断地给 preterminal 及其各个 segments 填充具体的 value。并行化的最大阻碍，就是按照概率降序生成口令这一过程。但是，PCFG 往往用于没有猜测次数限制的场景中（例如哈希破解），我们并不需要严格按照降序生成口令。这样一来，并行化的思路就很显然了，我们可以一次取出多个 preterminal，也可以一次为某个 segment 分配多个 value... 并行的思路是很多样的。学术界的一个并行化方案如图所示2.1。这种方案和原始 PCFG 的区别是，其对 preterminal 中最后一个 segment 不进行初始化和改变。在 preterminal 从优先队列中弹出的时候，直接一次性将所有可能的 value 赋予这个 preterminal。例如，L4S2D3 在出队时的 value 为 WangD3，那么此时将会把 D3 所有可能的 value 都赋予这个 preterminal，以一次性生成多个口令。不难看出，这个过程是可以并行进行的。

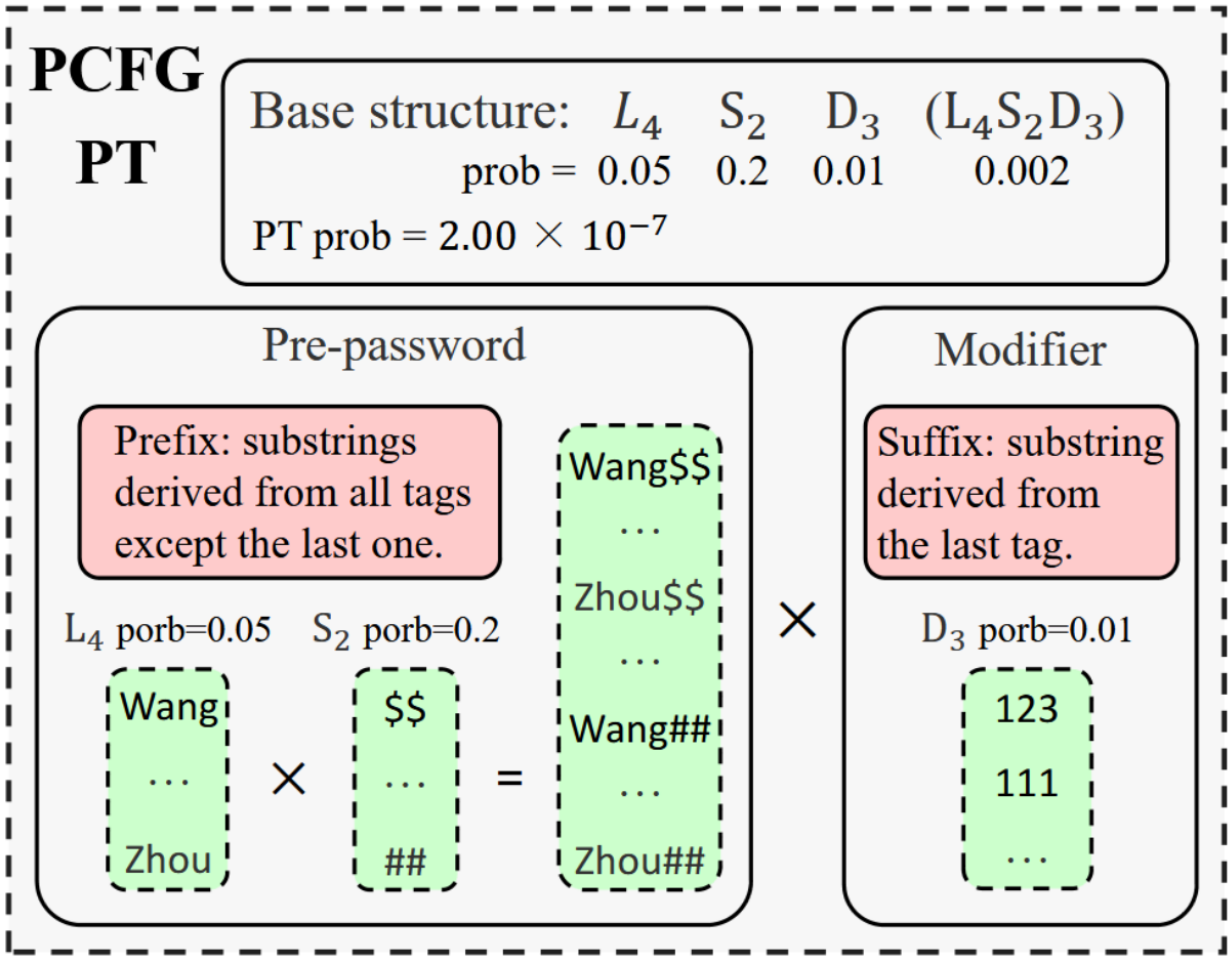

图 2.1: 并行化策略

3 多线程描述

3.1 Pthread

Pthread 是 POSIX 标准定义的一套多线程编程接口，提供在同一进程内创建、管理和同步多个“轻量级”线程的能力；每个线程拥有独立的调用栈和寄存器上下文，但共享进程的全局数据和堆空间。通过 pthread_create 可以启动新线程，pthread_join 或 pthread_detach 用于回收或分离线程资源；线程属性（pthread_attr_t）允许设置栈大小、调度策略和优先级等。Pthreads 提供多种同步原语：互斥锁 (pthread_mutex_t) 保护共享数据的互斥访问，条件变量 (pthread_cond_t) 支持线程间的等待和通知，读写锁 (pthread_rwlock_t) 实现多读单写并发控制，以及屏障 (pthread_barrier_t) 用于线程同步。这些机制能够精细控制并发执行流程、避免竞态和死锁。在本次实验中，我们将采用 phread 对口令猜测任务进行优化，以提高执行效率。

3.2 openMP

openMP 是一套基于共享内存的多线程并行编程标准，主要通过在 C/C++ 源码中插入编译指示（#pragma omp）来实现并行化，而不必显式管理线程的创建、同步和销毁。它遵循 Fork–Join 模型：当遇到并行区域（#pragma omp parallel）时，主线程会“分叉”出多个工作线程；并行区域结束后，这些线程再“汇合”回

主线程。OpenMP 提供了工作共享（parallel for）、数据划分（shared/private）、归约（reduction）、同步（critical/atomic/barrier）和调度策略（schedule）等机制，可以用最小的代码改动，实现循环并行、任务并行乃至复杂的嵌套并行。与 pthread 相比，OpenMP 开发复杂度更低但对线程调度的灵活度较低。

3.3 特性对比

两者的特性差异对比如下表：

维度	**pthread**	**OpenMP**
抽象层次	低级别线程库，需手动创建线程、管理同步与任务划分	基于编译指示，高层并行模型，自动生成底层线程调用
易用性	灵活但样板代码多，容易出错（死锁、竞态）	只需在循环或代码块上加注解即可并行化，改造成本低
可移植性	POSIX 标准，类 Unix 通用；Windows 需额外移植层	依赖编译器支持（GCC/Clang/MSVC 等），支持的平台上“一次注解，多处生效”
性能开销	无额外运行时开销，但需程序员自行管理同步粒度	有运行时调度和线程池管理开销，但自带多种调度策略和线程复用
调度与负载均衡	完全由程序员设计，适合不规则依赖场景	提供 `schedule(static|dynamic)` 等指令，自动化负载均衡
调试与可视化	GDB 直观跟踪调用，手动管理锁易产生 subtle bugs	隐藏线程管理，但大多数性能工具能识别 OpenMP 区段并报告并行效率
典型应用场景	性能敏感、自定义调度或实时系统	科学计算、数值模拟、大规模循环并行化、快速原型开发

表 1: pthread 与 OpenMP 特性对比

4 算法设计

4.1 串行实现

串行算法主要实现的思路是在 preterminal 从优先队列中弹出的时候，直接一次性将所有可能的 value 赋予这个 preterminal。在实现的过程中，我们可以分为两类进行讨论。对于只有一个 segment 的 PT，我们可以直接依次遍历生成其中所有 value 即可。对于有多个 segment 的 PT，需要给给当前 PT 的所有 segment 赋予实际的值（最后一个 segment 除外），得到一个前缀 guess，然后对于最后一个 segment 将所 segment 可能的 value 赋值到 PT 中，形成完整的口令。把生成的每条猜测保存到全局容器 guesses，并累加总猜测数 total_guesses 最终，guesses 中就存着所有从这个 PT 派生出来的完整密码候选，total_guesses 记录了总共生成了多少条。

4.2 Pthread 优化

在前面的串行算法中，我们可以知道在这个过程中最耗时的步骤便是两个循环的执行，因此我们并行化的主要操作便在于此处。我们采用多线程技术 pthread 来实现优化，我们将使用 8 个线程，将整个循环的任务划分为为 8 个部分，每个线程处理一部分任务，等到所有线程执行完毕后，再将结果进行合并操作，等到最终的结果。在这个过程中，每个线程独立处理一部分任务，实现 8 个线程并发处理口令猜测任务，从而将低循环所需要的时间，提高执行效率。

4.3 OpenMP 优化

在本次实验中，我们同样可以利用 OpenMP 对串行算法的循环任务进行并行化优化，从而提升算法的执行效率，降低执行时间。OpenMP 优化的基本思路就是，对 guess 进行空间的预分配写入空间，然后使用指令 #pragma omp parallel 自动给各个线程分配执行的任务，然后实现回合合并形成完整的输入。对口令写入的空间进行预分配的主要原因是可以避免线程在运行时频繁地进行动态内存分配或自动扩容（如 push_back 触发的重分配），从而减少锁竞争、系统调用和缓存不一致带来的开销；此外，预分配还能让每个线程在各自的内存区间内写入数据，降低伪共享风险，并且通过一次性调用 reserve 或 calloc 保证数据连续，从而提升并行写入的带宽和整体性能。

5 代码实现

5.1 串行算法

串行代码的实现如下图。其的核心步骤是：先调用 CalProb 初始化并计算当前 PT 的概率，然后检查 PT 是否只有一个 segment——若是，直接遍历该 segment 的所有可能值，依次将它们作为完整猜测加入 guesses 并累加计数；若 PT 包含多个 segment，则先根据 curr_indices 拼出除最后一段以外的固定前缀，再定位最后一段的模型数组，遍历其所有值，将每个后缀接到前缀上生成完整字符串，同样存入 guesses 并累加总猜测数。

对于单段处理，通过 content[0].type 来确定段的类型，然后根据类型取出所有可能的口令段。

```
        if (pt.content[0].type == 1)
        {
            a = &m.letters[m.FindLetter(pt.content[0])];
        }
        if (pt.content[0].type == 2)
        {
            a = &m.digits[m.FindDigit(pt.content[0])];
        }
        if (pt.content[0].type == 3)
        {
            a = &m.symbols[m.FindSymbol(pt.content[0])];
        }
```

下面的代码将直接遍历生成所有可能的值保存在猜测口令的集合中。

```
for (int i = 0; i < pt.max_indices[0]; i += 1) {
    string guess = a->ordered_values[i];
    guesses.emplace_back(guess);
    total_guesses += 1;
}
```

对于多段处理，我们首先给当前 PT 的所有 segment(除了最后一个 segment 除外) 赋予实际的值，segment 值根据 curr_indices 中对应的值加以确定。经过这一步，等到口令的前缀部分，转换为类似单段口令猜测，依次遍历所有可能的值，并加上前缀 guess 构成完整的密码。

```
    for (int idx : pt.curr_indices)
        {
```

```
            if (pt.content[seg_idx].type == 1)
            {
                guess +=
                    m.letters[m.FindLetter(pt.content[seg_idx])].ordered_values[idx];
            }
            if (pt.content[seg_idx].type == 2)
            {
                guess +=
                    m.digits[m.FindDigit(pt.content[seg_idx])].ordered_values[idx];
            }
            if (pt.content[seg_idx].type == 3)
            {
                guess +=
                    m.symbols[m.FindSymbol(pt.content[seg_idx])].ordered_values[idx];
            }
            seg_idx += 1;
            if (seg_idx == pt.content.size() - 1)
            {
                break;
            }
        }

        // 指向最后一个segment的指针，这个指针实际指向模型中的统计数据
        segment *a;
        if (pt.content[pt.content.size() - 1].type == 1)
        {
            a = &m.letters[m.FindLetter(pt.content[pt.content.size() - 1])];
        }
        if (pt.content[pt.content.size() - 1].type == 2)
        {
            a = &m.digits[m.FindDigit(pt.content[pt.content.size() - 1])];
        }
        if (pt.content[pt.content.size() - 1].type == 3)
        {
            a = &m.symbols[m.FindSymbol(pt.content[pt.content.size() - 1])];
        }
```

5.2 pthread 优化

首先，定义一个结构体 ThreadData，用于在主线程和工作线程之间传递参数：

- segment *a：指向当前要遍历的字符/数字/符号集合
- int start, end：当前线程负责处理的下标区间 [start, end)。
- vector<string> *guesses：指向存放最终所有猜测字符串的全局 vector。
- pthread_mutex_t *mutex：保护对全局 guesses 和 total_guesses 修改的互斥锁。
- int *total_guesses：全局猜测数计数器，所有线程合并时需要更新它。

- string prefix：前缀字符串——对于多段情况，先由主线程拼好前几段，再传给各子线程。

结构体 ThreadData

```
struct ThreadData {
    segment *a;                 // 指向segment的指针
    int start;                  // 线程处理的起始位置
    int end;                    // 线程处理的结束位置
    vector<string> *guesses;    // 全局猜测结果缓存
    pthread_mutex_t *mutex;     // 互斥锁
    int *total_guesses;         // 总猜测数
    string prefix;              // 前缀字符串
};
```

然后，定义线程函数 GenerateGuesses，实现每个线程分别计算猜测口令。使用局部缓存，先把所有结果放到一个线程私有的 local_guesses 中，避免对共享容器频繁加锁。实现加锁合并，一次性将整块结果插入共享 vector 并更新计数，以最小化锁的粒度。

线程函数

```
void *GenerateGuesses(void *arg) {
    // 转换参数类型
    ThreadData *data = (ThreadData *)arg;

    // 使用局部缓存减少锁竞争
    vector<string> local_guesses;

    // 生成猜测
    for (int i = data->start; i < data->end; i++) {
        string guess = data->prefix + data->a->ordered_values[i];
        local_guesses.emplace_back(guess);
    }

    // 合并结果时加锁
    pthread_mutex_lock(data->mutex);
    data->guesses->insert(data->guesses->end(), local_guesses.begin(),
        local_guesses.end());
    (*data->total_guesses) += local_guesses.size();
    pthread_mutex_unlock(data->mutex);

    return nullptr;
}
```

接下来是在主线程中划分任务。定义 num_threads 记录使用的线程数量，申请并定义 num_thread 个线程和每个线程对应的线程函数。使用 PTHREAD_MUTEX_INITIALIZER 静态初始化一个互斥锁。待处理元素总数和指定的线程数将工作均分成若干块 (chunk_size)，然后为每个线程填充对应的任务信息 (要遍历的 start/end 区间、共享结果容器和互斥锁等)，并通过 pthread_create 同时启动 8 个线程，让它们各自并行执行 GenerateGuesses；最后主线程用 pthread_join 等待所有子线程完成，将它们各自生成并合并好的猜测结果全部收集后再继续执行，从而实现了字符串拼接任务的高效并行化。

```
    // 初始化线程相关变量
    int num_threads = 8;
    pthread_t threads[num_threads];
    ThreadData thread_data[num_threads];
    pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

    // 计算每个线程的任务量
    int chunk_size = static_cast<int>(a->ordered_values.size()) / num_threads;

    // 创建线程
    for (int i = 0; i < num_threads; i++) {
        // 配置线程数据
        thread_data[i] = {
            a,
            i * chunk_size,
            (i == num_threads - 1) ? static_cast<int>(a->ordered_values.size()) : (i
                + 1) * chunk_size,
            &guesses,
            &mutex,
            &total_guesses,
            prefix
        };

        // 创建线程
        pthread_create(&threads[i], nullptr, GenerateGuesses, &thread_data[i]);
    }

    // 等待所有线程完成
    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], nullptr);
    }
}
```

generate 的完整实现如下，我们首先定义了一个 ThreadData 结构体，用于向每个工作线程传递它要遍历的数组区间、共享的结果容器、互斥锁及全局计数器等信息；然后在 GenerateGuesses 线程函数里，线程先在本地 vector 中批量生成拼接了前缀和后缀的字符串，避免频繁加锁，待本地结果全部就绪后一次性通过 pthread_mutex_lock 将它们插入到全局 guesses 容器并更新 total_guesses；主线程则根据数组总长度和线程数计算每个线程的 start/end 范围，初始化互斥锁与预留空间后调用 pthread_create 启动线程，最后用 pthread_join 等待所有线程完成合并，从而在多核环境下大幅提升字符串生成的吞吐率。

Generate 函数

```
void PriorityQueue::Generate(PT pt) {
    CalProb(pt);

    if (pt.content.size() == 1) {
        ......

```

```
        int num_threads = 8;
        pthread_t threads[num_threads];
        ThreadData thread_data[num_threads];
        pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

        int chunk_size = static_cast<int>(a->ordered_values.size()) / num_threads;
        for (int i = 0; i < num_threads; i++) {
            thread_data[i] = {
                a,
                i * chunk_size,
                (i == num_threads - 1) ? static_cast<int>(a->ordered_values.size()) :
                    (i + 1) * chunk_size,
                &guesses,
                &mutex,
                &total_guesses,
                ""
            };
            pthread_create(&threads[i], nullptr, GenerateGuesses, &thread_data[i]);
        }

        for (int i = 0; i < num_threads; i++) {
            pthread_join(threads[i], nullptr);
        }
    } else {
        .......

        int num_threads = 8; // Number of threads
        pthread_t threads[num_threads];
        ThreadData thread_data[num_threads];
        pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

        int chunk_size = static_cast<int>(a->ordered_values.size()) / num_threads;
        for (int i = 0; i < num_threads; i++) {
            thread_data[i] = {
                a,
                i * chunk_size,
                (i == num_threads - 1) ? static_cast<int>(a->ordered_values.size()) :
                    (i + 1) * chunk_size,
                &guesses,
                &mutex,
                &total_guesses,
                guess
            };
            pthread_create(&threads[i], nullptr, GenerateGuesses, &thread_data[i]);
        }

        for (int i = 0; i < num_threads; i++) {
            pthread_join(threads[i], nullptr);
```

```
        }
    }
}
```

5.3 openMP 优化

使用 openMP 优化算法的代码如下。对于只有一个 segment 的 PT，我们首先从 pt.max_indices[0] 中取出需要生成的候选数量 total_count，并通过 offset = guesses.size() 记录当前 guesses 容器末尾位置，然后一次性用 resize 扩容到足够大小；接着在 #pragma omp parallel for schedule(static) 并行循环中，每个线程将自己的区间 [offset, offset+total_count) 上对应的 a->ordered_values[i] 直接写入到 guesses[offset + i]，最后在串行区累加 total_guesses。这样做既避免了多线程环境下的 push_back 竞争，也通过预分配和静态调度利用多核并行加速数据填充，从而提高了整体生成效率。

openMP 优化

```
\\单segment优化
        int total_count = pt.max_indices[0];
        size_t offset = guesses.size();
        guesses.resize(offset + total_count);

        #pragma omp parallel for schedule(static)
        for (int i = 0; i < total_count; i++) {
            guesses[offset + i] = a->ordered_values[i];
        }
        total_guesses += total_count;
```

在多段 PT 情况下，首先从 pt.max_indices 里拿到要枚举的后缀个数 total_count，然后记录当前 guesses 容器的大小 offset，接着一次性用 resize(offset + total_count) 预分配好结果存储空间，避免在并行里动态扩容带来的锁竞争和内存碎片；随后把前面已拼好的不变前缀 prefix 提取并测量长度，进入 #pragma omp parallel for schedule(static) 并行循环，每个线程根据下标 i 拿到后缀字符串引用 suffix，通过一次性 resize + memcpy 把前缀和后缀拼到一起构造出完整猜测，再直接写入到 guesses[offset + i]；最后在串行区累加 total_guesses += total_count。这样既能利用多核并行大幅加速海量字符串拼接，也能保证线程安全和最小化内存分配开销。

openMP 优化

```
\\多segment优化
        // 计算需要生成的猜测数
        int total_count = pt.max_indices[pt.content.size() - 1];
        // 记录原始 guesses 的大小，后续新的猜测直接写入预分配好的区域
        size_t offset = guesses.size();
        // 预先分配好全局 vector 空间
        guesses.resize(offset + total_count);

        // 将不变的前缀提前保存，并计算其长度
        const string prefix = guess;
        const size_t prefix_len = prefix.size();

```

```
        #pragma omp parallel for schedule(static)
        for (int i = 0; i < total_count; i++) {
            // 取出后缀引用
            const string &suffix = a->ordered_values[i];
            const size_t suffix_len = suffix.size();
            // 构造新串，直接调整内存大小，避免额外分配
            string new_guess;
            new_guess.resize(prefix_len + suffix_len);
            // 直接将 prefix 和 suffix 内存拷贝到 new_guess 内
            memcpy(&new_guess[0], prefix.data(), prefix_len);
            memcpy(&new_guess[prefix_len], suffix.data(), suffix_len);
            // 将新拼接的字符串存入全局 vector 中
            guesses[offset + i] = std::move(new_guess);
        }
        total_guesses += total_count;
```

6 测试结果

6.1 正确性检验

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，所以在本次实验中采用一个简单的方法来测试并行生成的口令的正确性。将 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，输出的信息将多出一行”Cracked”，作为猜测算法的攻破率了，正确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。我们分别对三种方法的攻破率进行测试，等到如下数据：

	并行算法	pthread 优化	openMP 优化
Cracked	358217	358217	358217

表 2: 攻破率测试结果

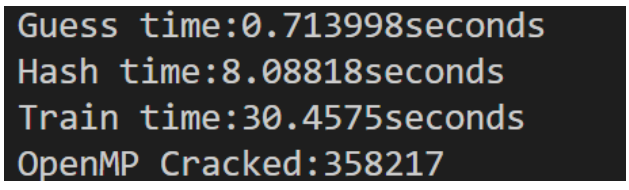

图 6.2: openMP

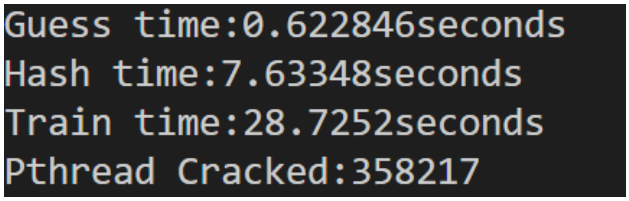

图 6.3: pthread

数据显示，使用 pthread 和 openMP 优化的算法的 Cracked 数据，同串行算法相比大致相同，说明采用 pthread 和 openMP 算法的口令猜测算法是正确的。

6.2 实验结果

本次实验主要在口令猜测部分修改，串行、pthread 优化、openMP 优化的代码分别保存在 guess.cpp、guess_pthread.cpp、guess_openMD.cpp 文件中。三个代码的编译指令分别为：

- 串行：g++ main.cpp train.cpp guessing.cpp md5.cpp -o main -O2
- pthread 优化：g++ main.cpp train.cpp guessing_pthread.cpp md5.cpp -o main -lpthread -O2
- openMP 优化：g++ main.cpp train.cpp guessing_openMP.cpp md5.cpp -o main -fopenmp -O2

本次实验的测试指标是 guess time，表示口令猜测过程中使用的时间。我们将三个代码分别在指定编译条件下运行，在提交 test.sh 时设置申请线程数为 8，重复实验 5 次，取平均值作为实验数据。实现结果如下表3所示：

guess time	串行实现	pthread 优化	openMP 优化
1	0.64949	0.76977	0.42699
2	0.59998	0.69255	0.41124
3	0.57217	0.78529	0.36381
4	0.63794	0.63052	0.41177
5	0.63352	0.66346	0.38689
平均用时	0.61862	0.70831	0.40014
加速比	＼	＼	1.5460

表 3: 性能测试结果 (8 线程)(单位:s)

表中的测试结果在小范围内波动，因此我们以 5 次测试结果的平均值作为我们的评教指标。通过表中 guess time 指标，我们可以看出串行实现的平均 guess time 为 0.61862s，而 pthread 的平均 guess time 为，openMP 的平均执行时间是 0.40014s。使用 openMP 优化的算法的执行时间明显低于串行实现，实现了相对于串行的加速，使用 openMp 的加速比为：1.5460 。openMP 实现了相较于串行算法的加速，明显看出使用 openMP 优化的循环执行的效率更高，执行时间更少。但是对于 phtread 的优化的性能表现不加，其执行时间于串行实现相差不大，没有很好的利用多线程的优势实现并行执行以获得加速。我们将在后续分析中详细的解释可能的原因。

7 结果分析

在前面的结果中显示使用 openMP 实现了相较于串行算法的加速，而 pthread 没有实现相较于串行的加速。下面将查阅相关资料，尝试在理论层面进行可能的原因分析。

1.OpenMP 能实现加速，可能是因为它在后台维护了一个线程池来复用线程、避免反复创建销毁的巨大开销，又通过 #pragma omp parallel for 自动为每个线程分配连续且均衡的写入区间，从而实现零锁并发写入和良好的缓存局部性；加上编译器对标记了并行指令的循环进行内联、展开和 SIMD 矢量化，以及运行时的线程亲和性优化，综合消除了手写 pthread 版中创建销毁开销、锁竞争、负载不均和编译器优化不足等瓶颈，最终获得了显著的并行加速效果。

2.pthread 未能实现加速，可能是由于每次都反复 pthread_create/pthread_join 带来的开销，诸如操作系统分配栈、调度上下文切换、线程启动和退出的成本，这些开销花费巨大，可能将本来可以并行的收益显著减少，甚至导致了负优化。与 openMP 相比较而言，OpenMP 在幕后维护了一个固定大小的线程池，后

续能够复用已有线程，几乎没有这种反复创建销毁的负担，所以能看到明显加速。要想实现 pthread 的加速优化，可能的方向是仿造 openMP 维护一个固定” 线程池”，让它们在后台循环等待任务、执行任务、再回到等待，而不是每次都 pthread_create / pthread_join。

8 探究线程数对加速比的影响

本次实验中我们已经通过 openMP 在 8 线程的基础上实现了相较于串行的加速，我们思考那么在多线程的实验中，线程数量是否会会影响加速比呢。为此，我们继续 openMP 的实验，使用不同的线程数，记录 guess time 的变化，来探究加速比的变化。由于实验平台最多只能申请 8 个线程，我们将测试在线程数为 1 到 8 的 openMP 优化算法。本次实验中，我们在 O2 优化的编译条件下，通过相平台申请不同的线程数，记录 guess time 的数据如下表：

线程数	1	2	3	4	5	6	7	8
串行	0.61547	0.60637	0.62121	0.58304	0.60906	0.57912	0.60282	0.58297
openMP	0.60547	0.57637	0.52121	0.48304	0.42906	0.44912	0.40282	0.37297
加速比	1.01096	1.04637	1.19221	1.20304	1.22906	1.33912	1.35282	1.47297

表 4: 不同线程数的执行时间

为了更加直观的展示数据之间的差异和变化趋势，我们绘制 guess time 和加速比随着线程数变化的折线图，如下图8.4和8.5所示。从数据中我们可以看出口令猜测的执行时间 guess time 随着线程数的增加存在下降的趋势，而相较于串行算法的加速比随着线程数的增加存在上升的趋势。由此，我们可以得出在一定的范围内，加速比随着线程数的增加而增加。从理论层面分析原因，是由于随着线程数的增加，使得每一个线程所需要完成的任务减少，所需要的时间减少。

由于实验平台一次提交最多申请 8 个线程，更多线程数的实验没有实实验，因此我们查阅资料可以得到：当线程数在 cpu 核心数以内时，多一个线程就多一份计算资源，每个线程都能独占或高效共享核心资源性能快速提升；一旦超出核心数，就进入 “超售” 模式，操作系统开始在更多线程间频繁进行上下文切换，并且缓存抖动、内存带宽饱和、TLB 未命中和锁竞争等开销迅速累积，此时新增线程无法带来额外并行度，反而因调度与同步开销而使性能增长停滞甚至下降。所以，在多线程数编程时，我们要选择合适的线程数，以实现最佳的加速。

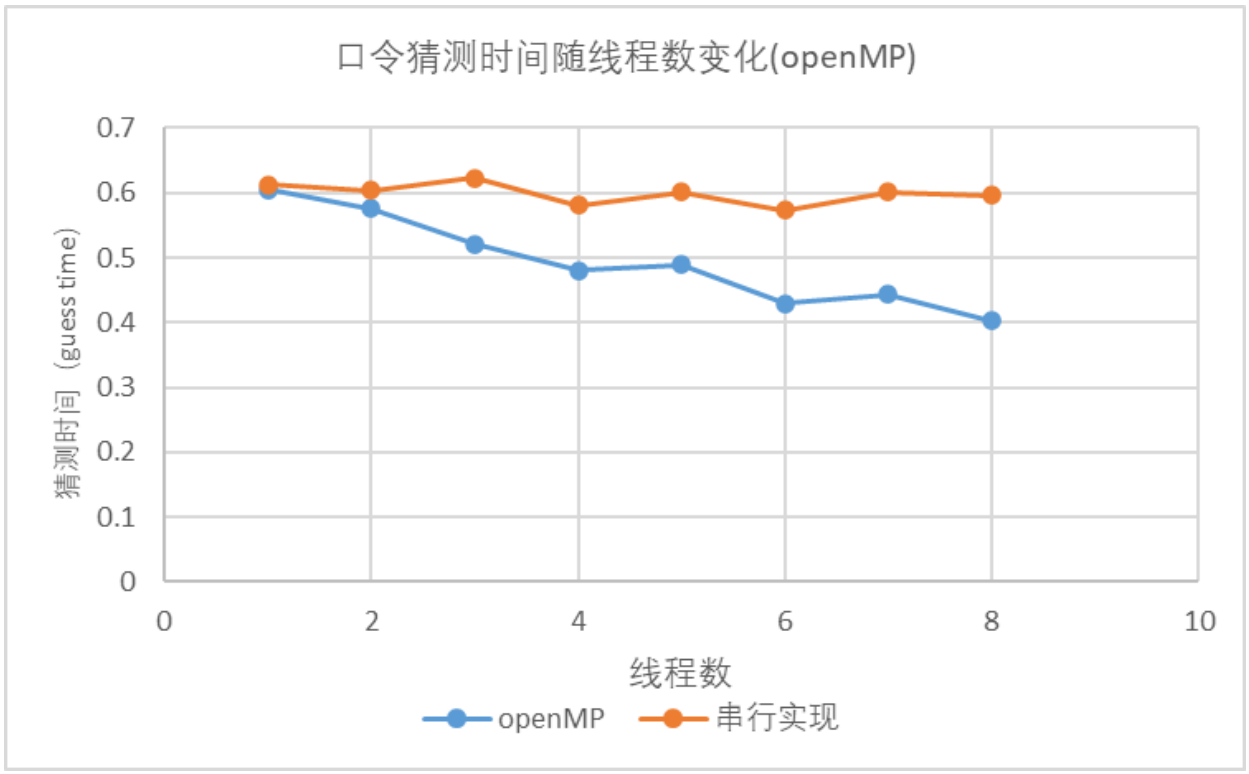

图 8.4: 时间变化

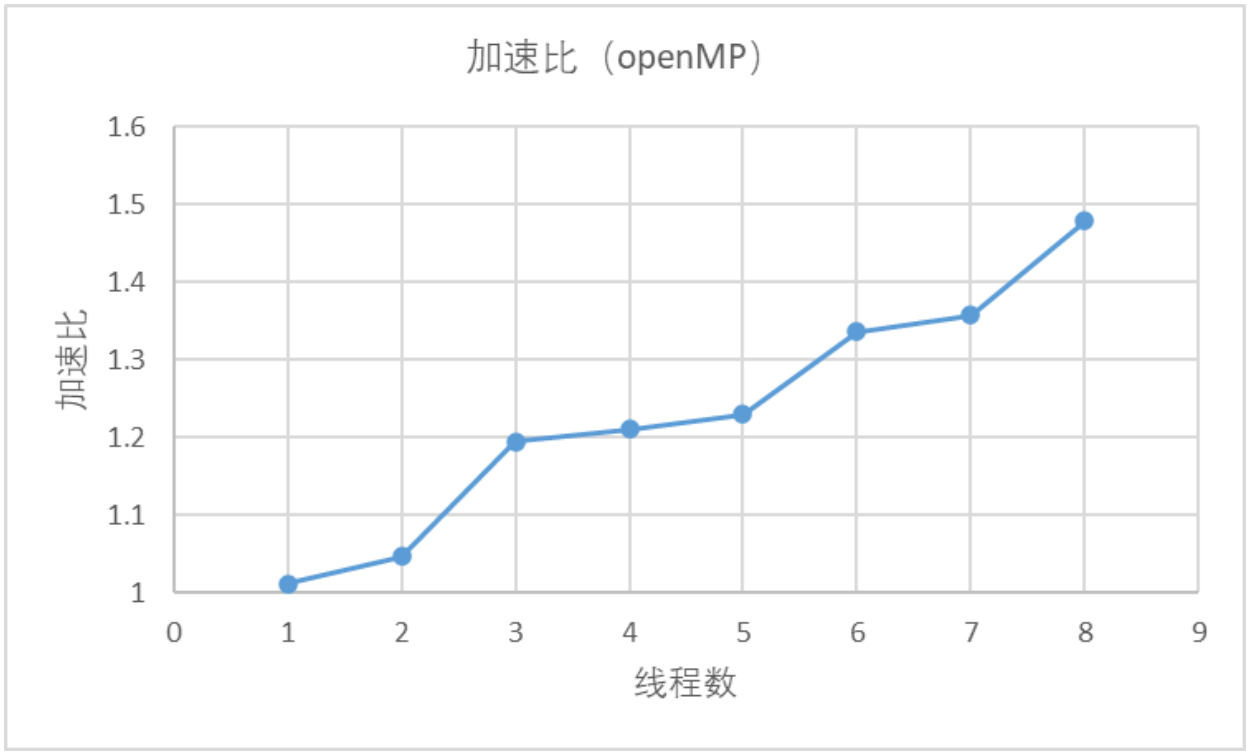

图 8.5: 加速比变化

9 SIMD+openMP 优化

9.1 测试说明

在 SIMD 编程实验中，我们利用 SIMD 的相关指令使口令猜测算法的 hash 过程能给实现同时对 4 个口令计算其 hash 值，实现了对于 MD5hash 函数的四路并行并行化，并且实现了相较于串行算法的加速。在本次实验中，我们同样利用 openMP 实现了针对口令猜测算法的并行化，利用 8 个线程实现了同时进行口令猜测的过程，并且同样实现了相较于串行的加速操作。接下来，我们将两个部分的并行化同时实现，探究是否能同时实现加速。SIMD 并行化影响 hash time,openMP 并行优化影响 guess time，在两者同时实现的基础上，我们将以 hash time 和 guess time 两者作为测试指标进行测试。

9.2 测试结果

实验测得数据如下表:

编译优化	time	串行实现	SIMD+openMP 优化
无	guess time	7.63870	7.32029
无	hash time	9.39410	15.26084
O1 优化	guess time	0.63910	0.45408
O1 优化	hash time	3.11583	2.69619
O2 优化	guess time	0.62261	0.39506
O2 优化	hash time	2.88532	2.55617

表 5: 性能测试结果 (8 线程)(单位:s)

1. 从上表可以看出，无论是串行实现还是 SIMD+openMP 优化，采用编译优化后的 guess time 和 hash time 都明显下降了，说明编译优化确实能够大幅提升程序的性能表现，能够明显降低同一个程序的执行时间，提高程序的运行效率。而针对不同的优化，提升的幅度较小，说明基础的编译优化部分确实起到了最大的作用，在已经有了编译优化后采用不同的编译优化的提升幅度不大。

2. 针对上述数据，我们可以发现在不使用编译优化时，串行的 guess time=7.63870 高于 openMP 优化的 guess time=7.32029, 说明在不使用编译优化的情况下，使用 openMP 能够实现相较于串行的口令猜测部分的加速，加速比为 1.04，处在一个较低的加速状态。而在不使用编译优化下，串行的 hash time=9.39410 远小于 SIMD 优化的 hash time = 15.26084，明显出现了负优化。

3. 采用 O1 或 O2 优化之后，我们可以发现此时优化后的算法无论是 guess time 还是 hash time 都明显低于串行实现的 guess time 和 hash time，此时 openMP 实现了对于口令猜测部分的加速，SIMD 优化算法实现了对于 MD5Hash 部分的加速，我们同时完成相交于串行的加速。

4. 通过编译优化之后，两者同时实现了加速，而不使用编译加速的情况下 SIMD 是无法实现加速的。我们可以明显看出出编译优化对与 SIMD 和 openMP 的影响确实存在差异。一般来说，编译优化对于 SIMD 的影响是更大的，能够将无法加速的算法编译优化之后实现了加速操作。

9.3 结果分析

为了更加深入的了解编译优化带来的影响。在查阅资料后，将相关的编译优化知识总结如下表。

1. 启用编译器优化后，编译器会在不改变程序功能的前提下，对源代码进行全局分析与变换：它能将热点变量常驻寄存器、消除无用代码与常量提前计算、在调用处展开小函数以去除调用开销，并为关键循环自动应用展开与向量化，最大化利用 CPU 流水线与 SIMD 单元；同时通过重排指令与合并分支减少流水线停顿与跳转开销，从而在运行时显著降低内存访问与分支预测失败带来的延迟，最终实现几时甚至数倍的性能提升。

优化级别	编译时间	代码大小	执行性能	主要优化特性
-O0	最快	最大（无任何删减 / 合并）	最慢（逐条生成，无寄存器重用）	• 仅展开宏、执行基本语法检查 • 保留全部调试信息 • 不做常量折叠或死代码消除
-O1	中等	略小于 -O0	中等（常见场景即可感知提升）	• 常量折叠、死代码消除 • 简单的循环展开与条件合并 • 限度内的函数内联（短小函数） • 基本寄存器分配与指令重排
-O2	较慢	中等偏大	较快（适合生产环境）	• 在 -O1 基础上更激进的循环优化 • 全局公共子表达式消除 • 跨函数内联（中等大小函数） • 指令调度与长度优化 • 简单的自动向量化（在支持的场景）

表 6: GCC 常用优化级别详细比较

2. 针对编译优化对于 SIMD 和 pthread 的性能能够大幅提升，可能的原因如下：当在串行编译优化的基础上引入 pthread 并行化和 SIMD 向量化时，程序不仅在指令层面获得了更高效的流水线调度和内联、常量折叠等好处，还在硬件层面同时动用了多核与矢量单元：pthread 并行化将大规模计算任务分摊到多核同时执行，SIMD 则让每条指令处理多组数据，二者本质上解决了“线程级”和“数据级”并行两大维度的性能瓶颈；之后再叠加编译器的高级优化，这共同作用使得整体加速比呈几何级增长，远超仅在单线程下使用编译优化所能达到的效果。

3. 在不开启编译器优化时，编译器会保留大量的函数调用开销、边界检查和不必要的内存访问，而且往往不会自动展开或对齐循环，也不会真正生成向量指令，于是即便我们手写了 SIMD 代码，也可能因为函数调用、未对齐的数据加载 / 存储、频繁的回退到标量路径等开销，把通过向量寄存器并行处理带来的收益全部抵消掉，导致看不到加速。而在打开了 -O2/-O3 这样的编译优化后，编译器会自动进行函数内联、循环展开、数据预取和内存对齐优化，并且启用自动向量化产生高效的 SIMD 指令序列，这些优化把函数调用开销和内存瓶颈大幅削减，让 SIMD 并行处理真正对齐数据、流水线饱满运行，从而在同一段代码上获得明显的性能提升。

10 总结与反思

1. 本次实验是实现基于 pthread 和 openMP 加速的口令猜测算法。在本次实验中，与串行算法相比，openMP 能够实现效率更高的口令猜测过程，但是并未实现 pthread 的加速，只能实现接近串行算法的执行时间。分析原因，个人对于多线程优化不够熟悉，只能实现基本的 pthread 优化，对于某些细节处的优化不够；在算法层面，可能是由于线程申请开销和线程合并的花销较大，与其带来的优化而言是不够的。可能优化的方向是首先一个线程池，重复利用线程，避免重复开销的代价。

2. 本次实现学习 pthread 的相关知识，通过 pthread 的练习和使用，我掌握了使用 pthread 对于串行加速的一般步骤：首先要找出最耗时且数据依赖最小的循环，将总任务按核数或负载均衡原则切分成若干份；然后定义线程入口函数，将每块数据的起止下标等必要信息打包为参数传给线程；在主线程中用 pthread_create 启动多个线程，让它们并行执行各自任务，结束后用 pthread_join 等待汇合；对于多线程可能同时访问的共享资源，需要用互斥锁、条件变量或屏障等同步机制保护。

3.openMP 对于串行加速的一般步骤是：首先定位程序中最耗时且迭代间无或极少依赖的循环；然后在代码开头 #include <omp.h>，并用 #pragma omp parallel for（可加 schedule、reduction、private、shared ）标注目标循环，把循环自动切分给线程；编译时加上 -fopenmp；运行后借助环境变量 OMP_NUM_THREADS 或 omp_set_num_threads() 控制线程数；最后通过调整调度策略、线程亲和性和减少同步开销等手段优化性能。

4. 通过实验分析，我们可以得出串行和并行的适用有所不同。一般来说，，当问题规模足够大、计算核间通信少、任务可均衡分配且硬件资源富余时并行化更合适；若任务依赖性高、需要频繁同步或硬件资源受限，则串行执行反而更加稳妥高效。

并行化最适合那些计算密集型、可拆分成大量独立子任务且数据依赖极少的问题，此时将工作划分给多个核心或线程，可以利用设备的并行计算单元，大幅缩短完成时间。而并行化的代价在于线程 / 进程的创建销毁、任务分配、同步和通信开销，以及可能的内存带宽竞争，因此在任务粒度过细、依赖关系复杂或数据交换频繁时，开销往往会抵消并行收益，反而不如串行高效；同样，如果问题规模较小、运行时间本身不足以掩盖并行初始化成本，也应保持串行实现。