南開大學
Nankai University

口令猜测并行化问题

——2025Spring 并行程序设计课程大作业选题之一

张逸非

2213218@mail.nankai.edu.cn

2025 年 3 月 28 日

目录

一、 结论

在解释口令猜测并行化这个选题的大致内容之前，先把大家最感兴趣的部分告诉大家比较合适。大家比较关心的问题，大致是难度、工作量、给分、助教能提供的支持这几个方面。

选择该选题和框架的优势：

- 并行算法原理的**理解难度上，本选题总体而言难度较低**。
- 相对基础要求的难度和工作量而言，**进阶要求的难度不高**（甚至只是在结果分析等科学问题上提出进一步的要求，**不会有较多的额外代码工作量**）。事实上，进阶要求和基础要求应当是一气呵成的，基础要求做好了，进阶要求就做完了。
- **基本不需要阅读英语论文**（进阶要求可能需要你总结前人方案的优缺点，**但不涉及额外的复现任务**）
- 你的方案最终实现的“加速比”和各类实验数据，**不会对分数造成根本性的影响**。

选择该选题和框架的劣势：

- 基础要求的工程难度相对较高。（主要难度并非代码量，而是**你认为有用的并行实现并不一定真的有效**）
- 助教对代码框架非常熟悉，可以快速定位到你的代码实现，并且**不介意逐行看你的代码**。**抄袭并且“洗代码”的风险非常高**。
- 用 AI 帮忙写代码是允许的——前提是你**看懂 AI 的每一行代码在干什么**，有什么效果。**口胡 AI 生成代码的作用，将会被视为抄袭**。

接下来是对该选题各方面特点的详细介绍。

(一) 选题难度

截至 3.22，本人（张逸非）已经完成了框架的代码编写工作，并且进行了 SIMD（x86）和多线程的相关实验，均实现了相对串行算法的加速。选题在难度上的特点大致如下：

1. 本选题的代码框架分为三个主要部分：模型训练/口令生成/MD5 哈希值生成。

2. 在三个主要部分中，MD5 哈希的原理理解难度较高，口令生成部分的原理理解难度较低。其余部分的理解比较简单。

3. 并行任务的主要难度，并不集中在原理的理解上，而是具体的代码实现。同一个并行算法，有的代码实现就是不能成功实现加速。你需要不断改进代码的细节，以求真正实现并行算法。

(二) 选题工作量

单就 SIMD 和多线程两次任务而言，本选题的代码量是很低的（MPI 和 CUDA 编程两次作业的代码量估计也不大）。但是代码量低的代价是，框架已经把串行算法完整地实现了一遍。这就意味着你需要至少理解框架关键部分的代码，才能继续对串行算法进行并行化。

(三) 给分/助教能够提供的支持

设计上，给分好坏与框架无关，将会控制本选题的平均分数与其余选题相对一致。但是本助教（张逸非）可以保证每次作业均会有非常详细的点评，扣分点/得分点/可以改进的地方均会在每次作业的批改中得到详细的解释。同时，本助教在批改过程中存在疑问/质疑的地方同样会给出理由。

对于选择本选题的同学，如对作业得分有不同意见（如助教看漏了某一部分、理解有误、质疑不合理），你可以根据助教的点评，针对性地进行反驳（rebuttal），并且联系助教。**不论反驳是否合理，反驳后的得分不会低于最初给分**。

助教保证能够提供的帮助包括：1. 分析你的代码实现有什么改进方法；2. 额外地为你讲解并行算法思路。

助教会尽力但不保证能提供的帮助：1. 帮你 debug 程序；2. 帮你解决本地的环境配置。

选择该选题的同学如对作业得分有意见或者需要该选题的相关帮助，请直接飞书联系张逸非同学（2213218）。

现在我们来具体看看口令猜测算法并行化选题的具体内容。接下来的部分将会分别介绍 PCFG 口令猜测算法及其并行化改造、MD5 哈希算法及其并行化

改造。

二、 PCFG 口令猜测算法

(一) 口令猜测背景知识（可以直接跳过）

口令，**俗称“密码”**，大多数网站、APP 等账号的身份认证环节都需要使用口令。大多数人可能会认为口令是不可猜测的：26 个字母，外加 10 个数字，以及特殊符号，暴力破解口令几乎是不可能的（更何况多数网站会有尝试次数限制）。事实上，用户的口令并不是随机的，而是有一定语义的，并且遵循一定规律。例如，用户选择”123456” 的概率，高于选择”123456!!!” 的概率；用户倾向于往口令中加入姓名缩写、生日等与个人信息相关的元素，如”zyf2025”；用户还有可能将口令进行重用，比如将 QQ 的口令”nankai114514” 稍微改改，将”Nankai114514” 用于淘宝。这些用户行为导致相当一部分口令是容易被猜出来的。

在本选题中，我们不考虑个人信息、口令重用这些额外的信息，只考虑非定向的口令猜测。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。

大家可能会发现，口令猜测问题，其实类似自然语言处理的任务。我们在这个选题中，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。

（本文和代码框架为了便于大家理解，简化了 PCFG 中各类专有名词和符号体系。如果大家直接看原论文 [2]，会发现这里的解释和原论文在用词和原理上有一些出入。）

(二) PCFG 串行训练（你需要看懂基本流程）

PCFG 的训练流程如图1所示，看懂训练流程，你就基本理解 PCFG 的运行原理了。

形式上来讲，任意给定的口令，我们可以将其分割为不同的字段（segments）。一共有三类字段：Letters（字母字段）、Digits（数字字段）、Symbols（特殊字符

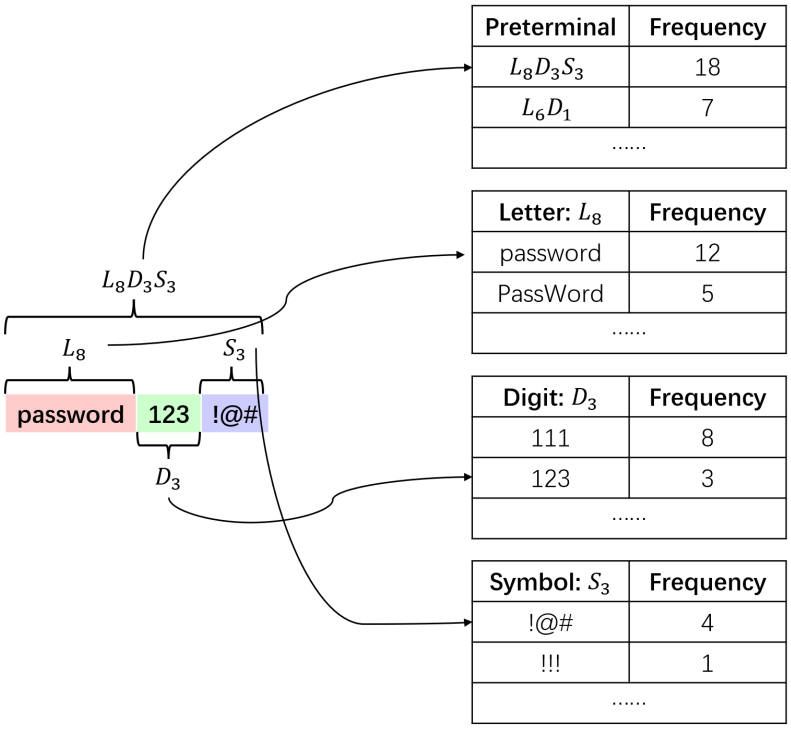

图 1: PCFG 模型的训练流程。每一个训练集里面的口令，都会被切分为不同的 segments，并且提取出一个 preterminal。模型对 preterminal 的不同类型、segment 的具体 value 进行统计，最终的统计结果就是我们的 PCFG 模型。

字段）。同一类字段，按照字段的长度进行区分；字段长度一样的，按照具体的 value 去进行区分。如图1所示，以”password123!!!” 为例，我们可以从中提取出 L_8（字母字段，长度为 8）、D_3（数字字段，长度为 3）、S_3（特殊字符字段，长度为 3）三个字段（segment）。在统计字段时，我们需要统计字段中各个 value 的值，如”password” 和”PassWord” 都是 L_8 这个 segment 的具体 value。

除了字段的 value 的频率需要进行统计之外，还需要统计 preterminal 的频率。其中，preterminal 就是一个口令中各 segment 组成的位置关系。比如，”password123!!!” 的 preterminal 是 $L_8D_3S_3$，”Alice1” 的 preterminal 是 L_6D_1，”nankai” 的 preterminal 是 L_6。

整个训练过程，其实就是对训练集里面出现的 segments 及其 values，以及 preterminals 的频率进行统计。训练完之后，就可以进行下一步的猜测生成了。

Preterminal	Probability
L_6S_1	0.52
L_6D_1	0.48

Digit: D_1	Probability
1	0.53
2	0.47

Letter: L_6	Probability
thomas	0.76
nankai	0.24

Symbol: S_1	Probability
!	0.8
@	0.2

图 2: 一个高度简化的 PCFG 模型

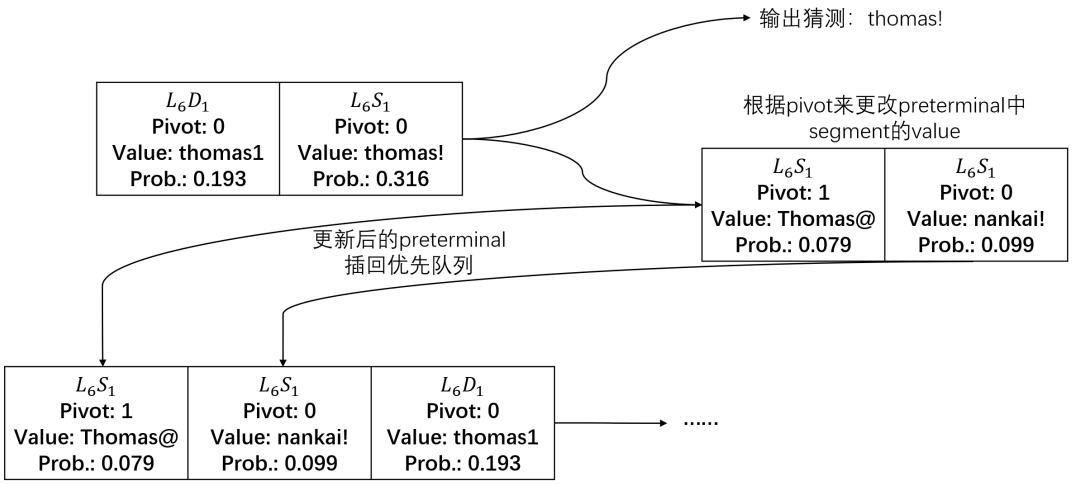

图 3: 利用优先队列进行猜测生成的流程

(三) PCFG 串行猜测生成（你需要仔细看）

利用优先队列和一些小技巧，就可以很方便地使用一个训练好的 PCFG 模型，生成大量口令猜测。我们以图2为例，说明如何用这个迷你 PCFG 模型进行猜测的生成（串行）。

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。例如，L_6D_1 中，L_6 概率最大的 value 是”thomas”，而 D_1 概率最大的 value 是”1”，所以初始化的时候，L_6D_1 的值就是”thomas1”。如图3所示，我们将模型中所有的 preterminal 进行初始化，并计算各自的概率，按照概率放入优先队列中，优先队列的初始化就完成了。

现在我们描述按概率降序生成猜测的过程。如图3所示，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”thomas!” 这

个口令。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。以 $L_6D_3S_1$ 为例，其 pivot 的取值范围为 0 到 2（即最大值为 preterminal 中 segment 的数目-1）。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。那么，假如 $L_6D_3S_1$ 的 pivot 值为 1，出队之后其将会生成两个新的 preterminals：

$$L_6D_3S_1, pivot = 1$$

$$L_6D_3S_1, pivot = 2,$$

其中，标红的 segments 是需要对 value 进行改变的 segments。那么，怎么改变某一个 segment 的 value 呢？我们从模型中找到这个 segment 对应的统计数据，取出其下一个概率最高的 value 即可。如图3所示，对于 pivot 值为 0 的 L_6S_1，当我们改变其 L_6 的值时，我们在模型中找到 L_6。由于当前 L_6 的值为”thomas”，其下一个概率最大的 value 为”nankai”，那么”nankai” 就是新的 value 值。这样一来，我们为这个 preterminal 生成了两个新的 value，并且改变了各自的 pivot 值。然后，我们将这两个新的 preterminal 放回优先队列即可。

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。

（四） PCFG 猜测生成并行化（你需要仔细看）

上述 PCFG 猜测生成的过程是不是有些复杂？没法完全看懂也不会影响对并行算法的理解！你只需要认识到，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。这个过程是不是不太好进行并行化？

其实并行化的最大阻碍，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，在这里我们介绍一个学术界的已有方案。

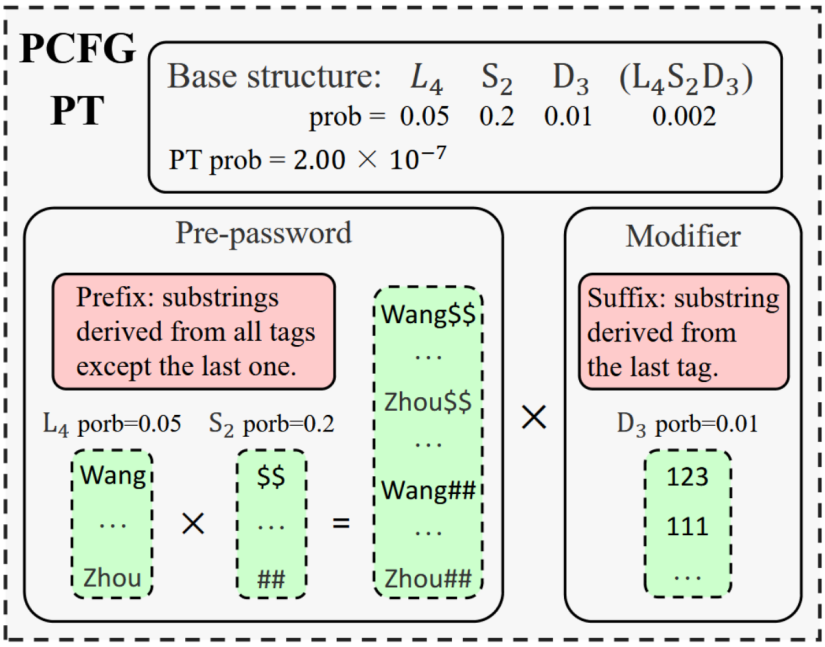

图 4: 本代码框架采用的并行化方案

学术界的一个并行化方案如图4所示。这种方案 [1] 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，$L_4S_2D_3$ 在出队时的 value 为 $Wang\#\#D_3$，那么此时将会把 D_3 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。本选题的代码框架，已经将这个并行算法用串行的方式进行了实现，你只需要将其用并行编程的方式，变成一个并行程序，就可以完成 PCFG 的并行化了。

恭喜你看完了整个 PCFG 并行化的大致流程！现在你会发现，其实前面一大堆 PCFG 的原理看似有点难以理解，但并行算法的思路并不需要你完全理解 PCFG 串行算法的全过程，只需要你思考如何让这个过程变快。

其实还有很多方法能够将 PCFG 的口令生成过程并行化，例如，采用多个优先队列并行地生成猜测。同时，PCFG 的训练过程也可以进行加速，比如将训练集分割成多个部分，然后并行地进行统计，最终再将多个部分的统计数据进行合并。大家可以大胆修改代码框架，甚至对代码框架的某些模块乃至整个框架进行重构，以尝试自己想到的新思路。

三、 MD5 哈希算法

(一) 口令哈希算法背景知识（可以直接跳过）

假设你注册了一个账号，然后将你的口令（密码）发给了这个网站的服务器。假如这个网站用明文直接存储你的口令，那么一旦服务器被攻破，你的口令就为天下人所知了。这时，即便服务器修好了，攻击者也可以立即用你的明文口令登录你的账户，乃至用这个口令攻击你的其他账户。

这个时候将口令用哈希的形式进行存储的重要性就体现出来了——即便攻击者攻破了服务器，也只能拿到你口令的哈希值，必须用口令猜测工具尝试破解这个哈希。我们的 PCFG 只是生成了一系列口令，但要想真正模拟攻击者的能力，还需要有对口令进行哈希的功能。在这个选题中，我们采用 MD5 对生成的一系列口令进行哈希。

(二) MD5 哈希算法原理（你需要看懂基本流程）

MD5 是一个常用的哈希算法（尽管很早以前就被证明不够安全）。对于任意长度的信息（字符串，文件，图像等），MD5 都能为其生成一个固定长度的“摘要”。给定一个消息，MD5 将能够生成一个确定性的“摘要”，并且对原始消息的任意改动（哪怕只是改变了一个 bit）都会改变 MD5 的哈希结果。不同的消息通过 MD5 产生相同输出的概率是极低的。

现在我们来大致描述 MD5 算法的运算过程。MD5 需要对消息进行预处理，将其变成比特串的形式，并将其长度附加到这个比特串的后面。然后，这个附加了消息长度的新消息会被分割成多个长度为 512bit 的切片。对于不足 512bit 的部分，将会填充值 512bit。实际的 MD5 预处理过程比上述过程复杂很多，但是代码框架已经为你解决了这些棘手的问题。

然后，我们将会维护一个长度为 256bit 的缓冲区。对于每一个长度为 512bit 的切片，我们会将其分为长度为 32bit 的 16 个部分。每个 32bit 都会分别用于参与一轮运算，也就是说每个长度为 512bit 的切片将会经过 16 轮运算。每轮运算使用的计算公式不尽相同，但你只需要知道一点：每次运算都会对 256bit 的缓冲区进行改变，并且这 16 轮运算是前后依赖的，顺序不可改变。

最终，当所有 512bit 切片都运算完毕后，得到的最终 256bit 缓冲区就是 MD5 的输出，也就是原始消息的哈希值。

（三） MD5 哈希算法并行化思路（大致看看即可）

MD5 的这个过程是不是不太好并行化？事实上，从单个信息的 MD5 哈希值计算过程来看，并行化计算是不可能的。但是我们可以并行地对多个消息（多个口令）进行 MD5 的运算。在设计这个选题的时候，PCFG 不太便于用 SIMD 进行并行化，所以 MD5 是为了 SIMD 的作业额外添加的。

事实上，MD5 的运算在数据上是高度对齐的，并且运算过程具有很高的确定性（不论消息长什么样，使用的算法是一致的，没有什么分支判断）。这就意味着其非常适合用 SIMD 同时进行多个数据的运算和处理。具体的计算过程和并行化方法，将在 SIMD 作业中进行具体的介绍和讲解。

参考文献

[1] Ziyi Huang, Ding Wang, and Yunkai Zou. Prob-hashcat: Accelerating probabilistic password guessing with hashcat by hundreds of times. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 674–692, 2024.

[2] Matt Weir, Sudhir Aggarwal, Breno De Medeiros, and Bill Glodek. Password cracking using probabilistic context-free grammars. In *2009 30th IEEE symposium on security and privacy*, pages 391–405. IEEE, 2009.