南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计期末实验报告

口令猜测并行化探究

许洋

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 7 月 6 日

摘要

口令安全是现代数字身份认证的核心环节，但用户行为模式导致口令分布高度倾斜，使得基于概率模型（如 PCFG）的口令猜测成为高效攻击手段。本研究针对 PCFG 串行算法存在的顺序依赖性、数据局部性差及计算不均衡问题，系统探究了多维度并行化解决方案。通过实现循环展开、SIMD 向量化、OpenMP 多线程、Pthread 线程管理、MPI 分布式计算及 GPU 加速，显著提升了猜测效率。并行策略的组合优化（如 OpenMP 调度策略选择、CPU-GPU 协作）进一步挖掘硬件潜力，为大规模口令安全分析提供技术支撑。

关键字：Parallel，PCFG 口令猜测，SIMD 向量化，OpenMP 多线程，Pthread 线程管理，MPI 分布式计算，GPU 加速

目录

一、 问题背景与描述

口令（俗称“密码”）是现代数字身份认证的核心组成部分，广泛应用于网站、APP 等平台的登录环节。表面上，口令的安全性似乎无可置疑：假设一个口令由 26 个字母、10 个数字及特殊符号组成，暴力破解的可能性极低。然而，现实中的用户行为严重削弱了口令的随机性。用户倾向于选择易于记忆的模式，例如“123456”或“password”，而非复杂的随机组合；更常见的是，用户将个人信息（如姓名缩写“zyf”或生日“2025”）嵌入口令（如“zyf2025”），或在不同平台重用相似口令（如将 QQ 密码“nankai114514”略作修改用于淘宝）。这些习惯导致大量口令存在可预测的语义规律，使得攻击者能通过非定向猜测高效破解账户。尤其在无尝试次数限制的场景（如离线哈希破解），攻击者需要生成一个按概率降序排列的口令词典：首先生成高概率口令，再按顺序尝试。这一过程计算密集，单线程串行处理效率低下，因此并行化成为提升猜测速度的关键解决方案。

(一) 口令猜测的核心问题背景

口令猜测的本质是模拟用户行为规律，生成一个按概率降序排列的口令列表。其难点在于如何高效建模用户偏好并加速生成过程。传统暴力破解法（如穷举所有字符组合）在理论上可行，但实际不可行：一个 8 位口令的组合空间超过 2×10^{14} 种，串行计算耗时过长。更优策略是利用概率模型，例如 PCFG（Probabilistic Context-Free Grammar，概率上下文无关文法），它基于训练集统计口令的常见模式。PCFG 将口令切分为字段（segments）：字母字段（Letters, L）、数字字段（Digits, D）和特殊符号字段（Symbols, S），并按长度分类（如 L_8 表示 8 位字母字段）。训练阶段统计各字段值（如 L_8 中的“password”）和预终结符（preterminal，即字段序列如 L D ）的频率，构建概率模型。例如，口令“password123!!!”被解析为 L_8、D_3、S_3 三个字段，其 preterminal 为 $L_8D_3S_3$。用户行为导致口令分布高度倾斜——少数高概率口令（如“123456”）占主导，这使得按概率降序生成猜测能显著提升破解效率（例如 50% 的账户可能在生成前 1% 口令时被破解）。然而，串行生成过程依赖优先队列，需严格维护顺序性，成为并行化的主要瓶颈。

(二) PCFG 串行算法及其并行化挑战

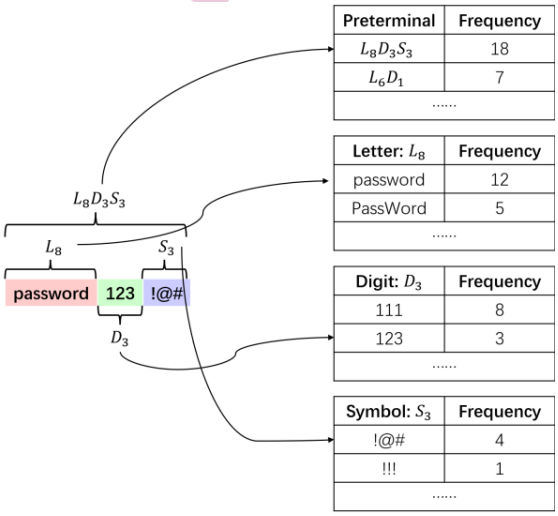

图 1: PCFG 模型的训练流程

PCFG 的串行流程包括训练和生成两阶段。训练阶段如图1所示，对数据集中的口令进行切分和统计：例如，输入“password123!!!”后，提取 L_8、D_3、S_3 字段，并记录其值（“password”）和 preterminal $L_8D_3S_3$ 的频率，最终输出概率模型（如图 2 的简化示例）。生成阶段则使用优先队列按概率降序输出口令：初始化时，为每个 preterminal 填入最高概率字段值（如 L_6D_1 初始化为“thomas1”），计算联合概率后入队；出队时，基于 pivot 值（标识待修改字段位置）生成新变体——仅修改 pivot 后字段，并替换为次高概率值（如 pivot=0 时，将 L_6 从“thomas”改为“nankai”），更新 pivot 后重新入队。这一过程保证无重复且严格降序，但存在严重并行化障碍：

- 顺序依赖性：优先队列的出队顺序必须严格按概率排序，并行线程同时访问队列会引发竞态条件，破坏顺序性。
- 数据局部性差：字段值的修改需查询全局概率模型（如 L 的候选值列表），频繁的内存访问在高并发下成为瓶颈。
- 计算不均衡：不同 preterminal 的变体生成量差异大（如 L_6S_1 的变体可能远少于 L_4D_2），导致线程负载不均。

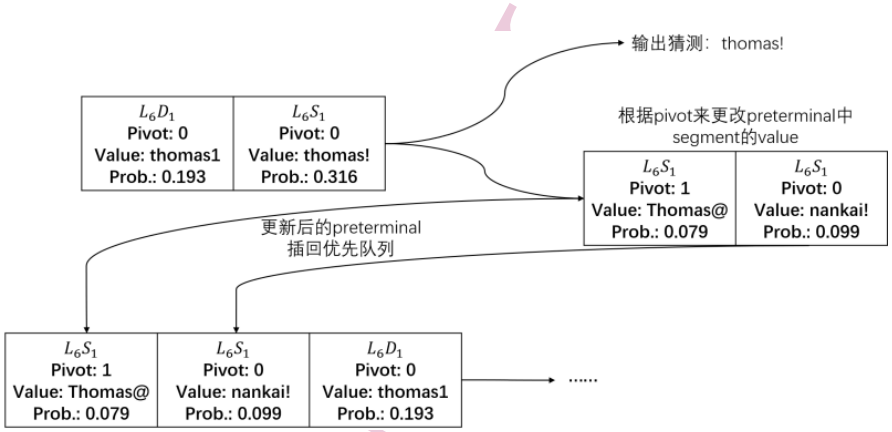

图 2: 利用优先队列进行猜测生成的流程

(三) 当前并行化解决方案描述

为克服上述挑战，学术界提出放松严格降序约束，允许近似排序以启用并行。核心方案如图3所示，包括：

- 并行生成阶段：在 preterminal 出队时，不再逐字段修改，而是批量处理末尾字段。例如，对 preterminal $L_4S_2D_3$（当前值“Wang##D_3”），一次性为 D_3 赋予所有可能值（如“123”、“456”等），并行线程同时生成多个口令（如“Wang##123”、“Wang##456”）。这避免了顺序队列竞争，线程可独立操作不同字段值列表。
- 任务划分策略：将 preterminal 分割为子集，每个线程负责一个子集的变体生成。例如，线程 1 处理所有含 L 字段的 preterminal，线程 2 处理含 D 字段的，结合负载均衡算法（如工作窃取）缓解计算不均衡。
- 训练阶段并行化：数据集分片后，多线程并行统计字段频率（如线程 A 统计 L 字段，线程 B 统计 D 字段），最后归并结果。这适用于多核 CPU 或 MPI 环境。

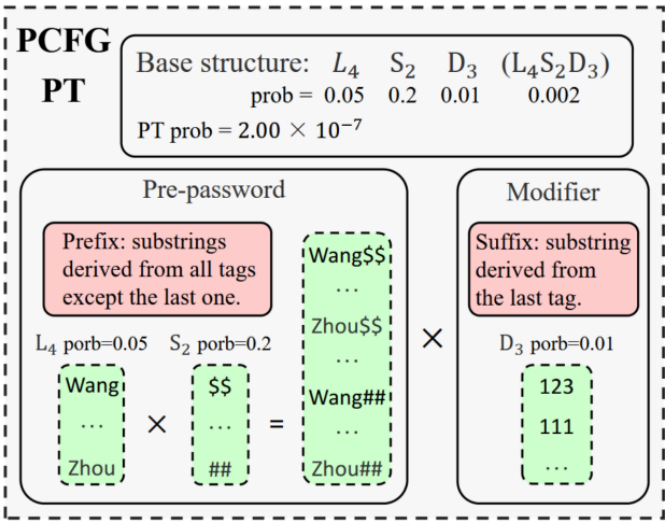

图 3: 现有并行化方案

二、 问题分析

口令猜测问题是一种常见的密码破解任务，涉及生成大量口令模板（Password Template, PT），计算其哈希值（如 MD5），并与目标哈希值比较以识别匹配项。该过程计算密集度高，尤其在处理大规模口令空间时，串行方法效率低下。

口令猜测问题的核心挑战

口令猜测问题的本质在于遍历高维口令空间以匹配目标哈希值。其流程包括：

PT 解析与生成：每个 PT 定义口令的结构（如字符类型、长度），通过填充剩余 segments 生成具体口令（例如，PT “L8D2” 表示 8 个字母后接 2 个数字，需填充具体值如 “password12”）。

哈希计算：对每个生成的口令计算 MD5 哈希值。

比较与验证：将哈希值与目标值比较，统计破解成功率。

口令猜测问题计算瓶颈在于 PT 内部循环的填充和哈希计算。例如，一个 PT 可能生成数千个口令，而口令总数可达百万级。并行化能显著减少耗时，尤其在分布式或异构平台上。

需要并行化的关键部分

口令猜测问题中需优先并行化的部分包括：

- PT 内部的口令生成循环：这是最计算密集的部分。在 guessing.cpp 文件中，存在两个核心循环：
  - 循环 1：对 PT 的剩余 segments 进行填充（如将”D2” 填充为具体数字值）。
  - 循环 2：生成完整口令列表。这些循环独立性强，每个口令生成任务互不影响，天然适合并行分割。
- MD5 哈希计算过程：每个生成的口令需计算 MD5 哈希值。哈希函数（如 FF、GG 等操作）可向量化处理，避免串行计算的冗余。
- PT 层面的任务分发：多个 PT 之间无依赖，可并行处理。例如，从优先队列中一次性取出多个 PT，并行生成口令，减少整体等待时间。

- 猜测与哈希的流水线执行：串行流程中，猜测完成后才进行哈希，造成资源闲置。并行化可让猜测和哈希阶段重叠执行，提升吞吐量。
- 利用多核 CPU、GPU 或分布式系统加速

实现并行化：简单方法与策略 实现口令猜测并行化的核心方法是任务分割与硬件加速。

- 使用 SIMD 加速哈希计算：

  方法：优化 MD5 函数，将哈希操作（如加法、移位）向量化。使用 NEON（ARM）或 SSE/AVX（x86）指令，同时处理多个口令的哈希计算。MD5 函数伪代码（如 FF、GG 操作），可改写为 SIMD intrinsic 函数。

- 使用多线程（Pthread/OpenMP）实现 PT 内部循环并行：

  方法：将 PT 的生成循环（如填充 segments）分割为子任务，分配给多个线程。例如，在 guessing.cpp 中，使用 OpenMP 的 `#pragma omp for` 指令或 Pthread 的 `pthread_create` 函数，将循环索引划分为块（chunk），每个线程处理一部分口令生成。建议设置 chunk_size 以平衡负载，避免线程空闲。

- 使用 MPI 实现 PT 内部循环并行与流水线：

  方法：主进程（rank 0）分发多个 PT 给工作进程，每个进程独立生成口令并计算哈希。同时，实现猜测-哈希流水线，进程 A 生成口令时，进程 B 并行计算上一批口令的哈希。使用 MPI_Send 和 MPI_Recv 进行通信。流水线部分需非阻塞通信（如 MPI_Isend）以避免等待。

- 使用 GPU 批量处理循环或 PT：

  方法：将 PT 数据复制到 GPU 显存，编写核函数（kernel）并行处理生成循环或哈希计算。用一个线程块处理一行口令生成任务，块内线程并行填充 segments。

  可结合 CPU-GPU 协作，在 GPU 计算时利用 CPU 进行其他任务（如 PT 队列管理）。

三、 并行算法实现与分析

(一) 循环展开

简要介绍 循环展开作为首次接触到的并行化策略，具有简单易于实现的特点。循环展开（Loop unwinding 或 loop unrolling），是一种牺牲程序的大小来加快程序执行速度的优化方法。可以由程序员完成，也可由编译器自动优化完成。循环展开最常用来降低循环开销，为具有多个功能单元的处理器提供指令级并行。也有利于指令流水线的调度。比如下面的例子：

原函数

```
for (i = 1; i <= 60; i++)
   a[i] = a[i] * b + c;
```

可以如此循环展开：

循环展开函数

```
for (i = 1; i <= 58; i+=3)
{
```

```
    a[i] = a[i] * b + c;
    a[i+1] = a[i+1] * b + c;
    a[i+2] = a[i+2] * b + c;
}
```

这被称为展开了两次。

实现思路及方法 循环展开是为了降低循环次数，所以我们需要去找循环的地方进行展开，来减少循环的次数。通过观察代码中的 for 循环的地方可以发现，在 guessing 部分，把模型中一个 segment 的所有 value，赋值到 PT 中，形成一系列新的猜测的循环完全可以采用循环展开的方式来实现。每次循环其中的内容并不依赖上次循环的结果，只和循环的次数有关，所以完全可以采取这种方式。在展开的过程中由于每个 PT 生成的猜测数不同，不能完全的循环展开，所以需要最后附加一段处理来保障每个 PT 生成的猜测都被完全的处理。

循环展开部分和串行部分都需要共同维护一个变量 i 来统计循环次数。先进行循环展开处理大量数据，

循环展开部分

```
int i = 0;
for (; i < pt.max_indices[0]; i += n)
{
    string guess0 = a->ordered_values[i];
    guesses.emplace_back(guess0);
    string guess1 = a->ordered_values[i+1];
    guesses.emplace_back(guess1);
    string guess0 = a->ordered_values[i+2];
    guesses.emplace_back(guess2);
    string guess0 = a->ordered_values[i+3];
    guesses.emplace_back(guess3);
    // ....n次展开
    total_guesses += n;
}
```

之后处理没有能展开的部分

串行部分

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

(二) SIMD

简要介绍 SIMD 的全称是 Single Instruction Multiple Data，[9] 中文名“单指令多数据”。顾名思义，一条指令处理多个数据。它是一种并行处理技术，允许一个指令处理多个数据。与传统的 SISD（Single Instruction, Single Data）架构相比，SIMD 可以显著提高数据处理效率，尤其

是在需对大量数据进行相同操作的场景下。SIMD 通过使用特殊的寄存器和指令集，将多个数据打包在一起，并在单个 CPU 时钟周期内执行同一个操作。比如，假设我们有四组数据需要做相同的运算，在传统的 SISD 架构下，需要执行四次指令，而使用 SIMD 技术，只需要一次指令即可完成四组数据的运算。在不同的系统中有不同的 SIMD 指令集：

Intel：Intel 处理器使用的 SIMD 技术包括 MMX、SSE（SSE、SSE2、SSE3、SSSE3、SSE4）、AVX（AVX、AVX2、AVX-512）指令集。这些指令集涵盖了多种数据类型的并行操作，包括整数、小数及双精度浮点数。

AMD：除了部分沿用 Intel 的指令集外，AMD 也开发了自己的扩展指令集如 3DNow!。

ARM：在移动和嵌入式设备领域，ARM 的 NEON 技术也是一种成熟的 SIMD 实现方式，广泛应用于各种高性能低功耗设备中。

实现思路及方法 SIMD 的并行化是进行的数据的打包，之后进行并行化的重复计算，所以需要寻找进行了重复计算的地方，在 MD5.cpp 中对于每个传入的字符串都进行了相同的操作计算 MD5hash 的值。MD5 算法需要先进行填充规则与消息块划分，对输入消息进行填充，确保其位长度对 512 取模等于 448。填充方法为：在消息末尾添加一个 ‘1‘，随后补 ‘0‘，直至满足长度要求，最后附加 64 位的原始消息位长度（小端序表示）。填充后的消息被划分为若干个 512 位的块，每个块进一步划分为 16 个 32 位的子块（记为 $M_0, M_1, \ldots, M_{15}$）。之后进行向量初始化与状态更新的过程，通过使用 4 个 32 位初始常数（小端序）：

$$A = \text{0x67452301},$$
$$B = \text{0xefcdab89},$$
$$C = \text{0x98badcfe},$$
$$D = \text{0x10325476}.$$

将每个 512 位块经过四轮处理（共 64 步），每轮更新状态变量 (A, B, C, D)。每步操作包括非线性函数、模加运算、循环左移和与子块/常数的混合。处理完所有块后，最终状态级联为 128 位哈希值。这其中经过四轮核心函数，每轮包含 16 步，使用不同的非线性函数和常数：

1. FF 轮：$F(B, C, D) = (B \wedge C) \vee (\neg B \wedge D)$，使用常数表 $T_1, \ldots, T_{16}$（由 $\sin(i)$ 生成）。
2. GG 轮：$G(B, C, D) = (B \wedge D) \vee (C \wedge \neg D)$，常数表 $T_{17}, \ldots, T_{32}$。
3. HH 轮：$H(B, C, D) = B \oplus C \oplus D$，常数表 $T_{33}, \ldots, T_{48}$。
4. II 轮：$I(B, C, D) = C \oplus (B \vee \neg D)$，常数表 $T_{49}, \ldots, T_{64}$。这里默认填充后的长度一致来实现更强的并行化操作。经过分析可知，首先我们需要一次性传入多个参数，这里出现两种传参方式：数组传参和向量传参。这两种传参方式在并行计算的过程中也产生了较大的影响。在这里需要讨论这两种传参方式的不同 [4] [7]：

- 数组传参：
  - 传递方式：

    当数组作为函数参数传递时，数组会退化为指针。例如定义 void func(int arr[]) 或者 void func(int *arr) ，本质上传递的是数组的首地址，在函数内部通过指针算术运算等方式去访问数组中的元素，而且函数通常还需要额外传入一个表示数组大小的参数，不然无法准确知道要操作的数组元素范围。
  - 内存管理：

    在数组在传递时只传递了首地址，其内存空间是在定义数组的地方分配好的（比如在函数外定义的全局数组，或者函数内定义的局部数组等），函数内部通过指针操作数

组元素时，访问的就是这片已分配好的内存区域。如果在函数内对数组元素进行修改，会直接影响到原始数组存储的数据。

- 向量传参：
  - 传递方式：

    vector 作为参数传递时，通常以引用的形式传递（一般用 const 修饰表示函数内部不会修改其内容，当然也可以不传引用而传值，但传值会进行拷贝，效率低且可能不符合需求），例如 void func(const vector<int>& v) 。这样传递的是 vector 本身的一个别名，函数内部可以直接像操作原 vector 一样去访问和处理它的元素，并且 vector 自身带有 size 成员函数，能方便地获取元素个数，不用额外再传入表示大小的参数。
  - 内存管理：

    vector 作为一个动态大小的容器，其内存管理由 vector 类内部机制负责。当以引用形式传递给函数时，函数内部操作的就是原 vector 的内存空间，vector 会根据元素个数的变化自动进行内存的扩容或缩容等操作（例如添加元素过多超过当前容量时会自动重新分配更大的内存空间并拷贝原有元素），并且这种内存管理对函数使用者来说基本是透明的，不用像数组那样额外考虑内存分配和越界等问题（当然使用不当也可能导致效率问题等情况）。

在这种情况下，我们可以用固定大小的数组进行传参，提高了参数的传递效率，同时，也减少了内存开销的代价，加速了并行化的过程。(对之前 SIMD 并行化时间提高效率不高的深入探究)

在传递参数的过程中，同样考虑传入参数和 SIMD 指令集的适配度。

md5.h–NEON 指令集

```
// 逻辑运算宏定义
#define F_NEON(x, y, z) vorrq_u32(vandq_u32((x), (y)), vandq_u32(vmvnq_u32(x)
    , (z)))
//....
#define ROTATELEFT_NEON(num, n) \
    vorrq_u32(vshlq_n_u32((num), (n)), vshrq_n_u32((num), 32 - (n)))
#define FF_VEC(a, b, c, d, x, s, t) \
    a = vaddq_u32(a, vaddq_u32(F_NEON(b, c, d), vaddq_u32(x, vdupq_n_u32(t)))
        ); \
    a = ROTATELEFT_NEON(a, s); \
    a = vaddq_u32(a, b);
//....
```

NEON 指令集通过 vaddq_u32 等指令，将数据进行 4 个一组打包计算，来提高运算速度。

md5.h–SSE 指令集

```
#define F_SSE(x, y, z) _mm_or_si128(_mm_and_si128((x), (y)), _mm_and_si128(
    _mm_andnot_si128(x, _mm_set1_epi32(-1)), (z)))
//....
#define ROTATELEFT_SSE(num, n) \
    _mm_or_si128(_mm_slli_epi32((num), (n)), _mm_srli_epi32((num), 32 - (n)))
#define FF_VEC(a, b, c, d, x, s, t) \
```

```
    a = _mm_add_epi32(a, _mm_add_epi32(F_SSE(b, c, d), _mm_add_epi32(x,
        _mm_set1_epi32(t)))); \
    a = ROTATELEFT_SSE(a, s); \
    a = _mm_add_epi32(a, b);
// ....
```

SSE 指令集通过 _mm_add_epi32 等指令，将数据进行 4 个一组打包计算，来提高运算速度。

(三) OpenMP

简要介绍 OpenMP 的英文全称是 Open Multiprocessing，一种应用程序接口（API，即 Application Program Interface），是一种单进程多线程并行的实现和方法，也可以认为是共享存储结构上的一种编程模型，可用于共享内存并行系统的多线程程序设计的一套指导性注释（Compiler Directive）。在项目程序已经完成好的情况下不需要大幅度的修改源代码，只需要加上专用的 pragma 来指明自己的意图，由此编译器可以自动将程序进行并行化，并在必要之处加入同步互斥以及通信。当选择忽略这些 pragma，或者编译器不支持 OpenMp 时，程序又可退化为通常的程序 (一般为串行)，代码仍然可以正常运作，只是不能利用多线程来加速程序执行。OpenMP 提供的这种对于并行描述的高层抽象降低了并行编程的难度和复杂度 [3]。

OpenMP 提供了多种指令来进行不同的设置:

线程设置指令

- omp_set_num_threads(n)：设置默认线程数

工作共享指令

- pragma omp for 或 pragma omp do：将循环迭代分配给线程。
- pragma omp sections：将代码块分配给线程。
- pragma omp single：指定一个线程执行代码块。

数据作用域指定子

- shared：变量在所有线程中共享。
- private：每个线程有自己的变量副本。
- firstprivate 和 lastprivate：类似于 private，但有特殊的初始化和赋值方式。

实现思路及方法 OpenMP 编程不需要大量修改代码，只需要在代码需要并行化的部分添加 OpenMP 的指令进行并行化操作即可，找到文件中 for 循环的部分，利用指令进行并行化。

首先我们需要进行线程数的设置，在这里需要根据 CPU 内核支持的线程数来设计以达到最大化利用

OpenMP 线程数定义

```
omp_set_num_threads(n); // 设置线程数为n
```

之后进行对于 for 循环部分的并行化，简单的添加并行化指令即可，

OpenMP 并行化

```
#pragma omp parallel for
for(int i = 0; i < n; ++i) {
    guesses[old_size + i] = a->ordered_values[i]; // 按索引直接写入
}
```

为了更好的提高并行化的效果，OpenMP 还提供了进行任务调度的功能，我在这里进行了新的探索，在 OpenMP 中，对 for 循环并行化的任务调度使用 schedule 子句来实现，schedule 的使用格式为：schedule(type[,size])，schedule 有两个参数：type 和 size，size 参数是可选的 [10]。

- **type 参数**表示调度类型，有四种调度类型如下：

  dynamic、guided、static、runtime

  这四种调度类型实际上只有 static、dynamic、guided 三种调度方式，runtime 实际上是根据环境变量来选择前三种中的某中类型。

- **size 参数 (可选)** size 参数表示循环迭代次数，size 参数必须是整数。static、dynamic、guided 三种调度方式都可以使用 size 参数，也可以不使用 size 参数。当 type 参数类型为 runtime 时，size 参数是非法的（不需要使用，如果使用的话编译器会报错）。

静态调度 (static)

当 parallel for 编译指导语句没有带 schedule 子句时，大部分系统中默认采用 static 调度方式，这种调度方式非常简单。假设有 n 次循环迭代，t 个线程，那么给每个线程静态分配大约 n/t 次迭代计算。不使用 size 参数时，分配给每个线程的是 n/t 次连续的迭代，使用 size 参数时，分配给每个线程的 size 次连续的迭代计算。

动态调度 (dynamic)

动态调度是动态地将迭代分配到各个线程，动态调度可以使用 size 参数也可以不使用 size 参数，不使用 size 参数时是将迭代逐个地分配到各个线程，使用 size 参数时，每次分配给线程的迭代次数为指定的 size 次。

guided 调度（guided）

guided 调度是一种采用指导性的启发式自调度方法。开始时每个线程会分配到较大的迭代块，之后分配到的迭代块会逐渐递减。迭代块的大小会按指数级下降到指定的 size 大小，如果没有指定 size 参数，那么迭代块大小最小会降到 1。

runtime 调度（rumtime）

runtime 调度并不是和前面三种调度方式似的真实调度方式，它是在运行时根据环境变量 OMP_SCHEDULE 来确定调度类型，最终使用的调度类型仍然是上述三种调度方式中的某种。

可以将上述代码进一步进行优化来提高效率，

OpenMP 实现

```
#pragma omp parallel for schedule(guided, 512)
for(int i = 0; i < n; ++i) {
    guesses[old_size + i] = a->ordered_values[i]; // 按索引直接写入
}
```

(四) Pthread

简要介绍 pthread（POSIX Threads）是一套符合 POSIX（Portable Operating System Interface，可移植操作系统接口）的 User Thread 操作 API 标准，定义了一组线程相关的函数（60 多个）和数据类型。pthread API 可以用于不同的操作系统上，因此被称为可移植的线程 API。pthread 线程库围绕 struct pthread 提供了一系列的接口，用于完成 User Thread 创建、调度、销毁等一系列管理 [11]。

实现思路及方法 pthread 实现较为复杂，需要定义线程结构体，并且手动进行线程的分发和收集。

首先, 在线程结构定义阶段, 我们设计了一个结构体来封装线程所需的所有参数, 关键成员包括提供原始数据的 seg_ptr、存储结果的 guesses 容器、统计生成总数的原子变量 total_guesses, 以及防止数据竞争的互斥锁 mutex。这些元素共同构建了线程安全的执行环境，集中管理输入数据源、输出容器和同步控制机制。

pthread 结构体

```
// 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
};
```

其次，在线程函数 thread_generate 的具体实现中，通过类型转换将通用参数转为特定结构体指针。函数内部采用预分配固定缓冲区的策略提升性能，并遍历分配的 [start_idx, end_idx] 数据区间，循环拼接前缀与字典项生成字符串。当缓冲区填满时，通过互斥锁保护将数据批量转移到共享容器，同时使用 fetch_add 原子操作无锁更新计数器。循环结束后，还会专门处理缓冲区中剩余未提交的数据，确保结果完整性。值得注意的是，实现中通过 make_move_iterator 应用移动语义避免字符串拷贝，显著优化了内存操作效率。

pthread 线程函数（伪代码）

```
函数 thread_generate(参数 arg):
    // 参数转换
    将 arg 转换为 ThreadArg 指针 targ
    // 初始化缓冲区
    设置 缓冲区大小 BUFFER_SIZE = 1000
    创建 字符串容器 buffer
    预留 buffer 容量为 BUFFER_SIZE
    // 处理分配的数据区间
    循环 i 从 targ->start_idx 到 targ->end_idx:
        拼接字符串 = targ->prefix + 当前字典项值
        将拼接结果存入 buffer
        // 缓冲区满时批量提交
        如果 buffer 大小  BUFFER_SIZE:
            获取互斥锁 targ->mutex
            将 buffer 内容移动插入到 targ->guesses 容器末尾 (避免拷贝)
            释放互斥锁
            // 更新计数器
            targ->total_guesses 原子增加 buffer.size()
            清空 buffer
            重新预留 buffer 容量为 BUFFER_SIZE
```

```
    // 处理剩余数据
    如果 buffer 非空:
        获取互斥锁 targ->mutex
        将剩余 buffer 内容移动插入到 targ->guesses 容器末尾
        释放互斥锁
        // 原子更新计数器
        targ->total_guesses 原子增加 buffer.size()
    返回 NULL
结束函数
```

最后,在主线程控制流程中,采用向上取整的分块算法 chunk_size = (total_size + NUM_THREADS - 1)/NUM_THREADS 将任务均匀分配给多个线程。每个线程被分配独立的数据区间，同时共享输出容器和计数器。同步机制设计上，仅当向 guesses 容器插入数据时才加锁，最大限度缩短锁持有时间；计数器更新则采用 memory_order_relaxed 内存序的无锁原子操作提升性能。主线程通过 pthread_create 启动所有工作线程，错误处理暂为直接退出（有待优化为资源清理），最终调用 pthread_join 阻塞等待所有线程完成任务后再继续执行。

pthread 实现（伪代码）

```
// 任务准备阶段
计算总任务量 total_size = pt.max_indices[0]
为结果容器 guesses 预分配空间（取 total_size 和 100,000 的较小值）
创建线程数组 threads 和线程参数数组 thread_args（大小为 NUM_THREADS）
初始化互斥锁 mutex（使用 PTHREAD_MUTEX_INITIALIZER）
计算任务分块大小 chunk_size = (total_size + NUM_THREADS - 1) / NUM_THREADS
    （向上取整）
// 线程创建与任务分配循环
循环 i 从 0 到 NUM_THREADS-1:
    初始化第 i 个线程参数:
        seg_ptr = a
        guesses = 结果容器地址
        start_idx = i * chunk_size
        end_idx = min((i+1) * chunk_size, total_size) // 防止越界
        prefix = "" // 实际可能使用前缀
        total_guesses = 原子计数器地址
        mutex = 互斥锁地址
    尝试创建线程:
        目标函数 = thread_generate
        参数 = 当前线程参数地址
    若线程创建失败:
        输出错误信息
        退出程序
// 同步与清理阶段
循环 i 从 0 到 NUM_THREADS-1:
    等待第 i 个线程结束（pthread_join）
销毁互斥锁 mutex
将原子计数器值加入总计数器 total_guesses
```

通过任务均匀分区、细粒度锁控制（仅保护共享容器插入）、缓冲区批处理以及移动语义优

化等技术，在保证线程安全的前提下最大程度减少同步开销，实现了高效并行执行。

（五） MPI

简要介绍 MPI 全名叫 Message Passing Interface，即信息传递接口，其作用是在不同进程间传递消息，从而可以并行地处理任务，即进行并行计算。需要注意的是，尽管我们偶尔会说使用 MPI 编写了某某可执行程序，但是 MPI 其实只是一个标准，而不是一种编程语言。其具体的实现由 OpenMPI, MPICH, IntelMPI 等库完成，这些库遵循 MPI 标准，并且可以被 Fortran、C、C++、Python 调用。MPI 最基本的消息传递操作包括：发送消息 send、接受消息 receive、进程同步 barrier、归约 reduction 等 [1]。

实现思路及方法 根据 MPI 进行传递消息的方式，我们设计了主从式任务分配架构。首先进行 MPI 初始化，所有进程都会执行相同代码，通过 rank 区分不同角色。

初始化

```
MPI_Init(&argc, &argv);  // 初始化 MPI 环境
int rank, size;          // 进程标识和进程总数
MPI_Comm_rank(MPI_COMM_WORLD, &rank);  // 获取当前进程 ID
MPI_Comm_size(MPI_COMM_WORLD, &size);  // 获取总进程数
```

之后我们进行主从进程的设计。主进程（Rank 0）负责维护优先队列，每次提取一个 PT 结构并生成候选密码列表，通过 MPI_Bcast 广播给所有工作进程。

主进程

```
while (!q.priority.empty())  // 当优先级队列非空
{
    q.PopNext();  // 生成下一个口令
    q.total_guesses = q.guesses.size();  // 更新当前计数
    // 进度报告（每10万口令）
    if (q.total_guesses - curr_num >= 100000) {
        if(rank == 0)
            cout << "Guesses generated: " << history + q.total_guesses <<
                endl;
        curr_num = q.total_guesses;
    }
    // 终止条件检查（超过1000万口令）
    if (history + q.total_guesses > 10000000) {
        // 时间统计和结果输出（略）
        break;
    }
    // 批处理机制（每100万口令）
    if (curr_num > 1000000) {
        // 口令验证和哈希计算
        // MPI结果汇总
        // 重置批次
    }
}
```

工作进程采用块划分策略处理分配到的密码子集:每个进程计算 chunk_size = (passwords_count + process_num - 1) / process_num，验证指定范围的密码哈希值。最后防止累积过多口令导致内存溢出，通过 MPI_Reduce 进行 MPI 通信，汇总所有进程的破解数，同时进行标准 MD5hash 实现，实现计算任务的分布式执行。

Generate 函数

```
// 生成前半部分固定字符串
string guess;
for (int idx : pt.curr_indices) {  // 排除最后一个 segment
    // 拼接各 segment 的当前值
    ...
    if (seg_idx == pt.content.size() - 1) break;
}
// 并行处理后段
int chunk_size = pt.max_indices[pt.content.size() - 1] / size;
int start = rank * chunk_size;
int end = (rank == size - 1)
    ? pt.max_indices[pt.content.size() - 1]
    : (rank + 1) * chunk_size;
for (int i = start; i < end; i++) {
    string temp = guess + a->ordered_values[i];  // 拼接完整猜测
    guesses.emplace_back(temp);
    total_guesses += 1;
}
```

通过这种方式，实现了进程的分发，不同进程并行完成任务。

(六) GPU

简要介绍 CUDA（Compute Unified Device Architecture），是显卡厂商 NVIDIA 推出的运算平台。CUDA 是一种由 NVIDIA 推出的通用并行计算架构，该架构使 GPU 能够解决复杂的计算问题。它包含了 CUDA 指令集架构（ISA）以及 GPU 内部的并行计算引擎。开发人员可以使用 C 语言来为 CUDA 架构编写程序，所编写出的程序可以在支持 CUDA 的处理器上以超高性能运行。CUDA3.0 已经开始支持 C++ 和 FORTRAN [13]。

GPU 的并发技术原理:

- 大规模并行计算架构：GPU 拥有上千个计算核心（CUDA 核心），每个核心都可以同时处理多个指令和数据，这使得 GPU 能够同时处理多条计算指令，从而实现高效的并发处理。
- SIMD（单指令多数据流）架构：SIMD 架构允许 GPU 的每个核心同时执行相同的指令，但处理不同的数据。这种设计使得 GPU 能够在同一时间执行多个相似计算任务，提高了计算效率。
- 线程束（Warp）：GPU 使用线程束的概念来进一步实现并发处理。线程束是一组共享相同指令流的线程，通常包含 32 个线程。GPU 会将许多线程分组为线程束，并同时调度执行，这提高了处理效率。
- 流处理（Streaming）：流处理将大量的计算任务拆分为一系列的小任务，然后以流的形式传送给 GPU 进行处理。这种方式有效地利用了 GPU 的并行处理能力，提高了计算效率。

- 数据分区和任务调度：在 GPU 并行计算中，输入数据被分成多个部分，并根据任务的性质和 GPU 的结构将任务分配给不同的核心。这要求实现高效的数据通信机制，以确保核心之间能够顺畅地交换数据 [8]。

实现思路及方法 通过动态任务分流和批量处理实现性能优化。核心思路是根据数据规模动态选择计算路径，当待处理候选值数量超过预设阈值 (1000 条) 时，激活 GPU 加速通道，否则使用传统 CPU 处理。这种设计既利用了 GPU 的并行计算能力，又避免了小数据量时的设备调用开销。实现包含几个关键层面：首先设计了预分配的设备内存池，通过全局指针维护设备内存状态，消除重复分配开销。

GPU 资源管理

```
// 全局设备内存池
static char* g_d_input, g_d_prefix, g_d_output, g_d_offsets, g_d_lengths;
// 候选值输入缓冲区，共享前缀缓冲区，结果输出缓冲区，值偏移量数组，值长度数组
// 资源配置参数
static size_t g_max_items = 100000;     // 最大处理项数(10万)
static size_t g_max_input_size = 50 * 1024 * 1024;    // 输入缓冲50MB
static size_t g_max_output_size = 100 * 1024 * 1024; // 输出缓冲100MB
static bool g_gpu_initialized = false; // 初始化状态标志
// 初始化函数：设备内存分配
bool initGPU() {
    if (g_gpu_initialized) return true;  // 避免重复初始化
    // 原子性分配五类设备内存
    cudaError_t err[5] = {cudaMalloc(&g_d_input, g_max_input_size) // 分配其
        他资源};
    // 成功判定：所有分配均成功
    // 失败处理：逆向释放已分配资源
    if (g_d_input) cudaFree(g_d_input);
    // 其他资源
    return false; // 传递错误状态
}
```

其次数据结构优化，主机端将离散的候选值连续打包存储，分离偏移量和长度信息，输出预分配 64 字节固定槽位。

GPU 数据处理（伪）

```
function processWithGPU(values: list[str], prefix: str) -> (bool, results:
    list[str]):
    # 阶段1：初始化与验证
    if not initGPU(): return False, None
    if values is empty or too large: return False, None
    # 阶段2：内存计算
    total_input_size = 总字符串长度 + values数量  # 每个值后加分隔符
    output_size = values数量 * 64  # 固定输出长度
    # 阶段3：主机内存分配
    h_input = 分配(total_input_size)          # 扁平化字符串存储
    h_offsets = 分配(values数量 * int大小)    # 各字符串起始位置
    h_lengths = 分配(values数量 * int大小)    # 各字符串长度
```

```
    h_output = 分配(output_size)                    # 输出缓冲区
    # 阶段4: 数据打包
    current_offset = 0
    for i, val in enumerate(values):
        h_offsets[i] = current_offset
        h_lengths[i] = len(val)
        复制val内容到h_input[current_offset]
        current_offset += len(val)
        h_input[current_offset] = 分隔符  # 添加分隔符
        current_offset += 1
```

在运行控制层面，创建专用 CUDA 流，异步拷贝主机数据到设备 (输入值、偏移量、长度、前缀)，触发核函数 (基于线程块自动划分任务)，异步回传结果。整个过程与主机计算重叠进行，通过流同步确保数据一致性。

运行控制（伪代码）

```
    # 阶段5: GPU数据传输
    stream = 创建CUDA流()
    异步复制h_input -> 设备内存
    异步复制h_offsets -> 设备内存
    异步复制h_lengths -> 设备内存
    if prefix不为空:
        异步复制prefix -> 设备内存
    # 阶段6: GPU核函数执行
    blocks = 计算所需线程块数量()
    启动核函数(blocks, 512, stream):
        参数: 输入数据, 偏移量, 长度, 前缀, 输出缓冲区
    # 阶段7: 取回结果
    异步复制GPU输出 -> h_output
    等待所有操作完成(stream)
    # 阶段8: 结果处理
    results = []
    for i in range(values数量):
        start = i * 64
        results.append(从h_output提取字符串(start, 64))
    # 阶段9: 清理资源
    释放所有主机内存()
    销毁CUDA流()
    return True, results
```

GPU 核函数采用两级拼接策略——每个线程处理一个候选值，先复制共享前缀 (所有线程访问相同的 d_prefix)，再追加专属段值 (通过 d_input 偏移量访问)，最后添加终止符。

核函数（伪代码）

```
__global__ void fastGenerateKernel(char* d_input, int* d_offsets, int*
    d_lengths,
                                   char* d_prefix, int prefix_len,
                                   char* d_output, int max_output_len, int
                                       num_items) {
```

```
    全局索引 idx = blockIdx.x * blockDim.x + threadIdx.x
若 idx    总项数 num_items: 返回  // 越界检查
分配输出位置: char* out_ptr = d_output + idx * 最大输出长度
写入位置 write_pos = 0
// 阶段1: 复制共享前缀
若 存在有效前缀:
    循环 前缀长度 次 且 缓冲区未满:
        out_ptr[write_pos++] = 当前前缀字符
// 阶段2: 复制线程专属值
输入位置 in_ptr = d_input + 当前值偏移量
循环 当前值长度 次 且 缓冲区未满:
    out_ptr[write_pos++] = in_ptr[i]
// 阶段3: 终止字符串
out_ptr[write_pos] = '\0'
```

在内部运行层面，动态路径决策引擎实现智能计算，决定由 GPU 还是 CPU 来进行计算。系统设置可调阈值 MIN_GPU_SIZE=1000 作为分水岭，处理大规模数据，激活高性能 GPU 通道，通过并行核函数生成密码，小规模任务则自动保留给 CPU 处理，规避 GPU 启动开销，设计自动回退功能，当 GPU 初始化失败 (cudaError_t 检测) 或内存超限时无缝切换至 CPU 路径，同时针对海量数据进行批处理优化，防止内核超时。

Generate 函数内部运行

```
void PriorityQueue::Generate(PT pt) {
    // ... 概率计算等预处理 ...
    const int MIN_GPU_SIZE = 1000; // GPU激活阈值
    if (pt.content.size() == 1) { // 单segment情况
        segment* a = /* 类型相关定位逻辑 */;
        int num_values = pt.max_indices[0];
        // GPU路径决策
        if (num_values >= MIN_GPU_SIZE) {
            vector<string> temp_results;
            if (processWithGPU(a->ordered_values, "", temp_results)) {
                // 批量添加结果
                for (const auto& result : temp_results) {
                    guesses.push_back(result);
                    total_guesses++;
                }
                return; // 成功则提前退出
            }// GPU失败时自动回退CPU
        }
        // CPU回退路径
        for (int i = 0; i < num_values; i++) {
            guesses.push_back(a->ordered_values[i]);
            total_guesses++;
        }
    } else { // 进行类似操作
    }
}
```

四、 并行实现拓展与分析

各种并行化的运行的结果在每次作业报告中都有呈现，所以不过多的进行分析。

(一) 优化拓展分析

在这里我想具体分析一下之前没有细致分析的优化对并行的影响。这次我分析的是-O0, -O1, -O2, -O3 这几种不同的优化方式对并行化的影响 [6]。

整个文件大系统编译出来的代码量巨大，不便于分析，我们采用简单的例子通过汇编的形式进行详细的分析。设计了一个简单的循环样例

循环样例

```
int data[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
void combine4(int *sum)
{
    int k;
    int i;
    *sum = 0;
    for (k = 0; k < COUNT; k++) {
        for (i = 0; i < sizeof(data) / sizeof(data[0]); i++) {
            *sum += data[i];
        }
    }
}
int main(int argc, char *argv[])
{
    int sum = 0;
    combine4(&sum);
    printf("sum = %d\n", sum);
    return 0;
}
```

通过这个样例我模拟的是口令猜测代码中与 for 循环有关的部分。将上述代码用不同的优化等级进行编译之后，测试运行时间，发现以-O0 为基准，-O1 和-O2 速度提升了 5 倍左右，-O3 提升了令人难以置信的 900 倍左右。之后，我将代码进行反汇编，来查看优化后内部的代码逻辑(汇编代码较长，不进行展示)。

从-O0 到-O1 的优化，指令数量减少，从 51 行缩减为 34 行，并且这是循环部分的指令，减少的行数要乘以循环次数，从指令的数量上我们可以明确的感知到程序缩短了相当可观的运行时间。在这其中，-O0 使用 eax 来存储计算结果，并在每次计算结束后将结果刷入内存 [rbp-0x18]，而-O1 直接将计算的结果存入内存 [rdx]。指令从三行变为一行，速度提升。

另一个变化是，使用寄存器 rsi 存储 data 数组的末尾地址，使用 rax 存储数组偏移，内层循环的边界判断条件直接使用这两个寄存器进行比较。这种方式省略了变量 i，因为变量 i 是存储在内存中的，对内存进行取值、自增计算、重新写入内存，要比寄存器操作慢得多，所以这也是提升运行速度的一个重要因素。

-O1 和-O2 的效果相差不大，查阅资料知道-O2 优化比-O1 优化更多的是分支预测功能，在这个代码中体现并不明显，我没有进行进一步的分析。

分析-O3 的优化发现了震惊的点，他直接将内层循环给优化掉了，10 次循环展开成 10 次指

令。这样一来，少了内层循环的边界判断。更重要的是，少一层跳转，就减少一次分支预测判断，这对指令执行的效率有显著提升。这种优化很容易让人发现和 SIMD 指令集十分相似，查阅资料同样得知，-O3 优化的过程中确实采用了这种策略。

(二) 并行方式拓展分析

1. OpenMP 的任务调度方式分析

static 调度 - 将迭代均匀分配给线程

```
#pragma omp parallel for schedule(static) shared(a, pt) reduction(+:
    total_guesses)
for (int i = 0; i < pt.max_indices[0]; i += 1)
{
    string guess = a->ordered_values[i];
    #pragma omp critical
    {
        guesses.emplace_back(guess);
    }
}
total_guesses += pt.max_indices[0];
```

dynamic 调度 - 动态分配任务块，适合负载不均衡的情况

```
#pragma omp parallel for schedule(dynamic, 100) shared(a, pt) reduction(+:
    total_guesses)
```

guided 调度 - 自适应块大小，开始大块，逐渐减小

```
#pragma omp parallel for schedule(guided, 50) shared(a, pt) reduction(+:
    total_guesses)
```

auto 调度 - 让编译器自动选择最佳调度策略

```
#pragma omp parallel for schedule(auto) shared(a, pt) reduction(+:
    total_guesses)
```

表 1: OpenMP 不同调度方式的性能对比

调度方式	Guess Time (秒)	相对性能	特点说明
static	17.7325	最慢	固定均匀分配迭代
dynamic	16.2723	较 static 快 8.2%	动态分配任务块
guided	16.0323	较 static 快 9.6%	自适应块大小
auto	15.4123	最快 (较 static 快 13.1%)	编译器自动优化

关键发现

- **static 调度性能最低** [12] [2]

  - **原因**：静态均匀分配迭代导致负载不均衡
  - **临界区瓶颈**：所有线程同时完成计算后集中争用 `#pragma omp critical` 区域，造成同步等待
  - 典型表现：最高 Guess time (17.73 秒)

- **动态调度显著提升性能**
  - dynamic/guided 分别快 8.2%/9.6%：
    - `dynamic`：以固定块 (100) 动态分配，减少线程空闲
    - `guided`：初始大块 (50) 逐渐减小，平衡调度开销和负载
  - **优势**：错开线程进入临界区时间，降低锁竞争

- **auto 调度最优**
  - 最快 (15.41 秒)：编译器自动选择最佳策略
  - **智能决策**：运行时根据硬件 (核数/缓存)、迭代数 (10106852 次) 和临界区特性自动优化

2. OpenACC 进行 MD5Hash 计算并行化

在做 GPU 并行化编程的时候，我一直在想，GPU 用来计算矩阵乘法等数据类型的并行化效果更好，那么我们能不能让 GPU 更多的来计算数据，而不是让 CPU 来计算数据，所以我查阅资料发现通过 OpenACC 可以将数据移动到 GPU 上进行计算，所以，我想办法利用 OpenACC 进行优化 MD5Hash 的计算过程 [5]。

MD5Hash 并行计算

```
void MD5Hash_Batch4_OpenACC(string inputs[4], bit32 states[4][4])
{
    // 为4个输入准备填充后的消息数组
    Byte *paddedMessages[4];
    int messageLengths[4];
    int n_blocks[4];
    // 预处理4个输入
    for (int batch = 0; batch < 4; batch++) {
        paddedMessages[batch] = StringProcess(inputs[batch], &messageLengths[
            batch]);
        n_blocks[batch] = messageLengths[batch] / 64;
    }
    // 找到最大的消息长度，用于分配统一的GPU内存
    int maxLength = 0;
    int maxBlocks = 0;
    for (int i = 0; i < 4; i++) {
        if (messageLengths[i] > maxLength) maxLength = messageLengths[i];
        if (n_blocks[i] > maxBlocks) maxBlocks = n_blocks[i];
    }
    // 创建统一的数组用于GPU传输
    Byte *unified_messages = new Byte[4 * maxLength];
    memset(unified_messages, 0, 4 * maxLength);
```

```
    // 将4个消息复制到统一数组中
    for (int batch = 0; batch < 4; batch++) {
        memcpy(unified_messages + batch * maxLength, paddedMessages[batch],
            messageLengths[batch]);
    }
    // 初始化所有状态
    for (int batch = 0; batch < 4; batch++) {
        states[batch][0] = 0x67452301;
        states[batch][1] = 0xefcdab89;
        states[batch][2] = 0x98badcfe;
        states[batch][3] = 0x10325476;
    }
    // 使用OpenACC并行处理4个输入
    #pragma acc data copyin(unified_messages[0:4*maxLength], messageLengths
        [0:4], n_blocks[0:4]) copy(states[0:4][0:4])
    {
        // 并行处理每个输入的所有块
        #pragma acc parallel loop collapse(2)
        for (int batch = 0; batch < 4; batch++) {
            for (int block_idx = 0; block_idx < maxBlocks; block_idx++) {
                // 只处理有效的块
                if (block_idx < n_blocks[batch]) {
                    bit32 x[16];
                    // 准备当前块的16个32位字
                    #pragma acc loop seq
                    for (int i = 0; i < 16; ++i) {
                        int base_idx = batch * maxLength + block_idx * 64 + i
                            * 4;
                        x[i] = (unified_messages[base_idx]) |
                               (unified_messages[base_idx + 1] << 8) |
                               (unified_messages[base_idx + 2] << 16) |
                               (unified_messages[base_idx + 3] << 24);
                    }
                    bit32 a = states[batch][0], b = states[batch][1], c =
                        states[batch][2], d = states[batch][3];
                    // MD5的四轮运算
                    /* Round 1 */
                    FF(a, b, c, d, x[0], s11, 0xd76aa478);
                    /* Round 2 */
                    GG(a, b, c, d, x[1], s21, 0xf61e2562);
                    /* Round 3 */
                    HH(a, b, c, d, x[5], s31, 0xfffa3942);
                    /* Round 4 */
                    II(a, b, c, d, x[0], s41, 0xf4292244);
                    states[batch][0] += a;
                    states[batch][1] += b;
                    states[batch][2] += c;
                    states[batch][3] += d;
```

```
                }
            }
        }
    }
    // 并行处理字节序转换
    #pragma acc parallel loop collapse(2)
    for (int batch = 0; batch < 4; batch++) {
        for (int i = 0; i < 4; i++) {
            uint32_t value = states[batch][i];
            states[batch][i] = ((value & 0xff) << 24) |
                               ((value & 0xff00) << 8) |
                               ((value & 0xff0000) >> 8) |
                               ((value & 0xff000000) >> 24);
        }
    }
    // 清理内存
    delete[] unified_messages;
    for (int i = 0; i < 4; i++) {
        delete[] paddedMessages[i];
    }
}
```

代码解析

1.OpenACC 数据区域 (#pragma acc data)

数据区域

```
#pragma acc data copyin(unified_messages[0:4*maxLength],
                        messageLengths[0:4],
                        n_blocks[0:4])
                copy(states[0:4][0:4])
```

创建数据区域，控制主机（CPU）和设备（GPU）间的数据传输，copyin 进行输入数据，unified_messages[0:4*maxLength] 输入 4 个填充后消息的合并字节数组、messageLengths[0:4] 输入每个消息的实际长度数组、n_blocks[0:4] 输入每个消息的块数数组,copy 进行输出数据,states[0:4][0:4] 是 MD5 的 128 位状态值（4 个 uint32）

2. 核心并行计算结构 (#pragma acc parallel loop)

核心并行计算结构

```
#pragma acc parallel loop collapse(2)
for (int batch = 0; batch < 4; batch++) {
    for (int block_idx = 0; block_idx < maxBlocks; block_idx++) {
        // 块处理代码
    }
}
```

通过 collapse(2) 将两层循环（batch+block）展平为单层循环，分配线程过程让每个线程处理一个 batch 的一个消息块，同时进行判断 if (block_idx < n_blocks[batch]) ，避免处理超出实际块数的虚拟块

3. 消息块加载 (#pragma acc loop seq)

消息块加载

```
#pragma acc loop seq
for (int i = 0; i < 16; ++i) {
    int base_idx = batch * maxLength + block_idx * 64 + i * 4;
    x[i] = (unified_messages[base_idx]) | ... ; // 小端字节序组合
}
```

seq 进行强制顺序执行，避免自动并行化，内存访问中利用 base_idx = batch 偏移 + 块偏移 + 字偏移使得能够连续内存访问，对 GPU 友好。

4. 字节序转换 (#pragma acc parallel loop)

字节序转换

```
#pragma acc parallel loop collapse(2)
for (int batch = 0; batch < 4; batch++) {
    for (int i = 0; i < 4; i++) {
        // 小端→大端转换
    }
}
```

与主计算分离，进行独立的并行化，同时每个状态值独立转换，无数据依赖。

(三) 并行组合拓展分析

在进行编程实验的过程中，我发现各种并行化的操作可以进行结合，从而实现更高程度的并行化计算。查阅资料知道，OpenMP 本来是在 CPU 上实现多线程并行化的操作，但是随着 GPU 编程的出现和大量使用，OpenMP 扩展到了 GPU 编程中，提高了多核 GPU 的利用率。所以可以将 GPU 编程同 OpenMP 编程以及 SIMD 编程联合起来。

SIMD 编程部分依旧是原来的通过 NEON 指令集或者 SSE 指令集进行并行化的操作。

通过 OpenMP 优化了 GPU 编程的内部处理操作。

OpenMP 优化 GPU 处理

```
bool processWithGPU(const vector<string>& values, const string& prefix,
    vector<string>& results) {
    if (!initGPU()) return false;
    size_t num_values = values.size();
    if (num_values == 0 || num_values > g_max_items) return false;
    // 并行计算所需空间
    size_t total_input_size = 0;
    #pragma omp parallel for reduction(+:total_input_size)
    for (int i = 0; i < values.size(); i++) {
        total_input_size += values[i].length() + 1;
    }
    if (total_input_size > g_max_input_size) return false;
    const int MAX_OUTPUT_LEN = 64;
    size_t output_size = num_values * MAX_OUTPUT_LEN;
    if (output_size > g_max_output_size) return false;
```

```
    // 准备主机数据
    char* h_input = (char*)malloc(total_input_size);
    int* h_offsets = (int*)malloc(num_values * sizeof(int));
    int* h_lengths = (int*)malloc(num_values * sizeof(int));
    char* h_output = (char*)malloc(output_size);
    if (!h_input || !h_offsets || !h_lengths || !h_output) {
        // cleanup code...
        return false;
    }
    // 并行打包输入数据
    #pragma omp parallel
    {
        #pragma omp single
        {
            size_t input_pos = 0;
            for (size_t i = 0; i < num_values; i++) {
                h_offsets[i] = input_pos;
                h_lengths[i] = values[i].length();
                input_pos += values[i].length() + 1;
            }
        }
        #pragma omp for
        for (int i = 0; i < num_values; i++) {
            memcpy(h_input + h_offsets[i], values[i].c_str(), values[i].
                length());
            h_input[h_offsets[i] + values[i].length()] = '\0';
        }
    }
    // GPU处理部分保持不变...
    // 并行处理结果
    results.resize(results.size() + num_values);
    size_t old_size = results.size() - num_values;
    #pragma omp parallel for
    for (int i = 0; i < num_values; i++) {
        char* result_ptr = h_output + i * MAX_OUTPUT_LEN;
        results[old_size + i] = string(result_ptr);
    }
    // cleanup...
    return true;
}
```

同时在 Generate 函数内部，进行了进一步的修改，因为 CPU-GPU 通信开销较大，较小的数据量不足以体现出 GPU 加速的效果，反而因为数据传输开销导致速度变慢，所以对 CPU 和 GPU 处理的数据量进行了进一步的划分。

智能选择

```
// 智能选择处理策略
if (num_values >= OPTIMAL_GPU_SIZE) {
```

```
    // 大规模数据: 纯GPU处理
    vector<string> temp_results;
    if (processWithGPU(a->ordered_values, "", temp_results)) {
        for (const auto& result : temp_results) {
            guesses.push_back(result);
            total_guesses++;
        }
        return;
    }
} else if (num_values >= MIN_GPU_SIZE) {
    // 中等规模: GPU+OpenMP混合
    const int chunk_size = num_values / 4; // 分成4块
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            // GPU处理前3/4
            vector<string> gpu_values(a->ordered_values.begin(),
                                      a->ordered_values.begin() + 3 *
                                          chunk_size);
            vector<string> gpu_results;
            if (processWithGPU(gpu_values, "", gpu_results)) {
                #pragma omp critical
                {
                    for (const auto& result : gpu_results) {
                        guesses.push_back(result);
                        total_guesses++;
                    }
                }
            }
        }
        #pragma omp section
        {
            // CPU并行处理剩余1/4
            #pragma omp parallel for
            for (int i = 3 * chunk_size; i < num_values; i++) {
                #pragma omp critical
                {
                    guesses.push_back(a->ordered_values[i]);
                    total_guesses++;
                }
            }
        }
    }
    return;
}
// 小规模数据: OpenMP并行处理
#pragma omp parallel for
```

```
for (int i = 0; i < num_values; i++) {
    #pragma omp critical
    {
        guesses.push_back(a->ordered_values[i]);
        total_guesses++;
    }
}
```

经过并行优化后相比之前，GPU 计算确实产生了加速，但是在整个过程中，仍有着 GPU 数据传输的开销，这是无法避免的，只能在 CPU-GPU 传输过程中，通过更高效的数据传输形式来解决。

五、 总结

(一) 并行方法对比与效果分析

表 2: 并行计算技术对比分析

方法	优势	局限
循环展开	减少分支预测开销，提升指令级并行度	需手动调整展开因子，适应性差
SIMD	单指令处理多数据（如 MD5 四轮运算向量化）	需数据对齐，NEON/SSE 指令集移植成本高
OpenMP	增量式并行（pragma 指令），支持智能调度	临界区竞争（如全局容器写入）需精细控制
Pthread	细粒度线程管理（缓冲区批处理 + 移动语义）	需手动实现任务划分与同步（如互斥锁保护容器）
MPI	分布式扩展能力（主从架构 + 流水线处理）	通信开销大（如 MPI_Bcast 广播 PT 结构）
GPU	超千核并行（CUDA 核函数处理千级任务/批次）	数据传输瓶颈（CPU-GPU 内存拷贝耗时占比较大）

(二) 关键优化收获

调度策略决定性能上限 在 OpenMP 调度策略对比中，auto 调度（15.41 秒）因自适应硬件环境性能最优，而 static 调度（17.73 秒）因线程同步等待表现最差（见表 1）。

启示：负载均衡与锁竞争缓解是并行设计核心，动态调度（dynamic/guided）通过错开临界区访问可提速 9.6%。

异构协作突破单设备局限 CPU-GPU 混合方案（如 OpenMP 管理 GPU 任务流）实现双重优化：

- 数据预处理由 CPU 并行完成（OpenMP 加速内存打包）
- 大规模计算卸载至 GPU（批处理 10 万级口令）

效果：较纯 CPU 方案吞吐量提升，较纯 GPU 方案减少数据传输损耗。

编译优化赋能底层并行 -O3 优化通过循环展开 +SIMD 内联，使 MD5 计算逻辑指令数减少，实测加速比达 900 倍，印证软件优化与硬件并行的协同价值。

(三) 各次实验收获

SIMD 在 SIMD 实验中，我成功实现了 MD5 哈希算法的并行化加速，通过 ARM NEON 和 x86 SSE 指令集对核心非线性函数（F、G、H、I）及循环左移操作进行向量化改造，显著提升了批量处理效率。实验发现 Batch Size 对性能具有关键影响：当 Batch=4 时达到最佳加速比 1.246 倍，而 Batch=8 时因缓存压力导致 IPC 下降 15%，这揭示了内存依赖对扩展性的制约。通过-O3 编译优化，LLC 未命中率从 9.68% 降至 0.36%，同时 StringProcess 函数的批量改造使 SSE 加速比提升 14.5%，验证了任务粒度均匀化的重要性。

多线程 多线程实验展现了不同技术的性能差异：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 实现的 1.15 倍边际优化。通过开发 PT 初始化与队列扩展的并发流水线，将 OpenMP 加速比提升至 1.67 倍，使千万级任务处理时间压缩至 15.4 秒。但实验也暴露了内存带宽竞争的深层问题——SIMD 哈希计算与多线程争用带宽，导致全局容器写入延迟占比增高。这些发现促使我们探索 GPU 异构加速方案，通过进阶优化建立了颗粒度划分标准，为大规模口令分析提供了实用解决方案。

MPI MPI 实验构建了创新的主从式广播架构，主节点 (Rank0) 分发 PT 结构，工作节点采用块划分策略 ($chunk\_size = N/size$) 实现负载均衡。在流水线设计中，分离口令生成 (Rank0) 与哈希验证 (Rank1) 阶段，通过 MPI_Send/Recv 实现异步处理。然而实现过程中发现模型广播存在冗余问题，未压缩训练数据导致通信开销增加，同时多 PT 并行插入破坏了优先队列的顺序性。针对这些缺陷，提出压缩训练数据的优化方案，这些经验为跨节点大规模部署提供了宝贵的技术储备。

GPU GPU 实验实现了系统性突破：设计线程级并行核函数，单线程块处理 64 个 PT 单元提升计算密度；创新批处理机制合并请求减少内核调用；通过全局内存池化降低分配开销。在内存体系优化中，利用共享内存缓存高频访问数据（如口令前缀），使全局访问延迟降低。成功建立原子化资源分配体系，实现 10 万级任务稳定运行（零泄漏），多 PT 并行处理达 40 倍加速。但显存墙限制（单任务最大处理量 100000 条）和小批量负优化（千级以下任务加速比 <1）仍是主要约束，这些成果有力推动了 CPU-GPU 协同架构在安全领域的落地应用。

六、 心得体会

通过系统性的理论学习和密集的实验探索，我对并行程序设计背后的核心原理与设计思想（如任务分解、数据并行、同步通信、负载均衡等）有了深入的理解。在实践层面，我成功应用并掌握了多种主流的并行计算框架与方法，包括利用 SIMD 指令实现细粒度数据并行加速，运用 pthread 库灵活构建和同步多线程任务，借助 OpenMP 简化共享内存模型的并行开发，通过 MPI 部署和协调分布式内存环境下的多进程通信协同，以及使用 CUDA 架构深度挖掘 GPU 的众核并行计算潜力。针对每种技术，我不仅能够实现基础功能，更能结合具体场景（如矩阵运算、字符串匹配等）进行优化配置和问题诊断，为后续的口令猜测等并行化问题的研究奠定了坚实的技术基础。

源代码：https://github.com/X-u-Y-a-n-g/parallel

参考文献

[1] Mpi 并行编程——多进程程序设计. CSDN 博客.

[2] openmp 任务调度 for schedule static dynamic guided. CSDN 博客.

[3] Openmp 学习笔记 (1)——简介, 模型, 运行. CSDN 博客.

[4] 在 c++ 中,vector 和数组作为函数参数传递时有什么区别? CSDN 博客.

[5] Cuda c 编程（三十）openacc 的使用. CSDN, 2023.

[6] Gcc 中-o1 -o2 -o3 优化的原理是什么. PingCode, 2023.

[7] 什么时候用 ‘<array>‘比 ‘<vector>‘更好? . reddit, 2023.

[8] Gpu 的并发技术原理, 实际案例说明. 腾讯云开发者社区, 2024.

[9] FOCUS. 并发编程：simd 介绍. 知乎, 2021.

[10] Lbbit. 掌握并行编程：openmp 入门与实践. 知乎, 2024.

[11] 云物互联. Linux 实现原理—pthread 多核平台并行编程. 知乎专栏, 2023.

[12] 周伟明. Openmp 中的任务调度-azkaban 任务调度. CSDN 博客, 2013.

[13] Lion 莱恩呀. 一文了解 gpu 并行计算 cuda. 知乎, 2023.