南開大學
Nankai University

计算机学院

并行程序设计期末报告

并行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 7 月 6 日

目录

1 问题描述

1.1 背景知识

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。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。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。整个口令猜测算法可以分为三个部分，分为模型训练过程、口令猜测部分以及 MD5 哈希过程，我们的并行优化主要集中于口令生成部分和 MD5 哈希部分。

1.2 PCFG 串行训练

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），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。在统计字段时，我们需要统计字段中各个 value 的值。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。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。

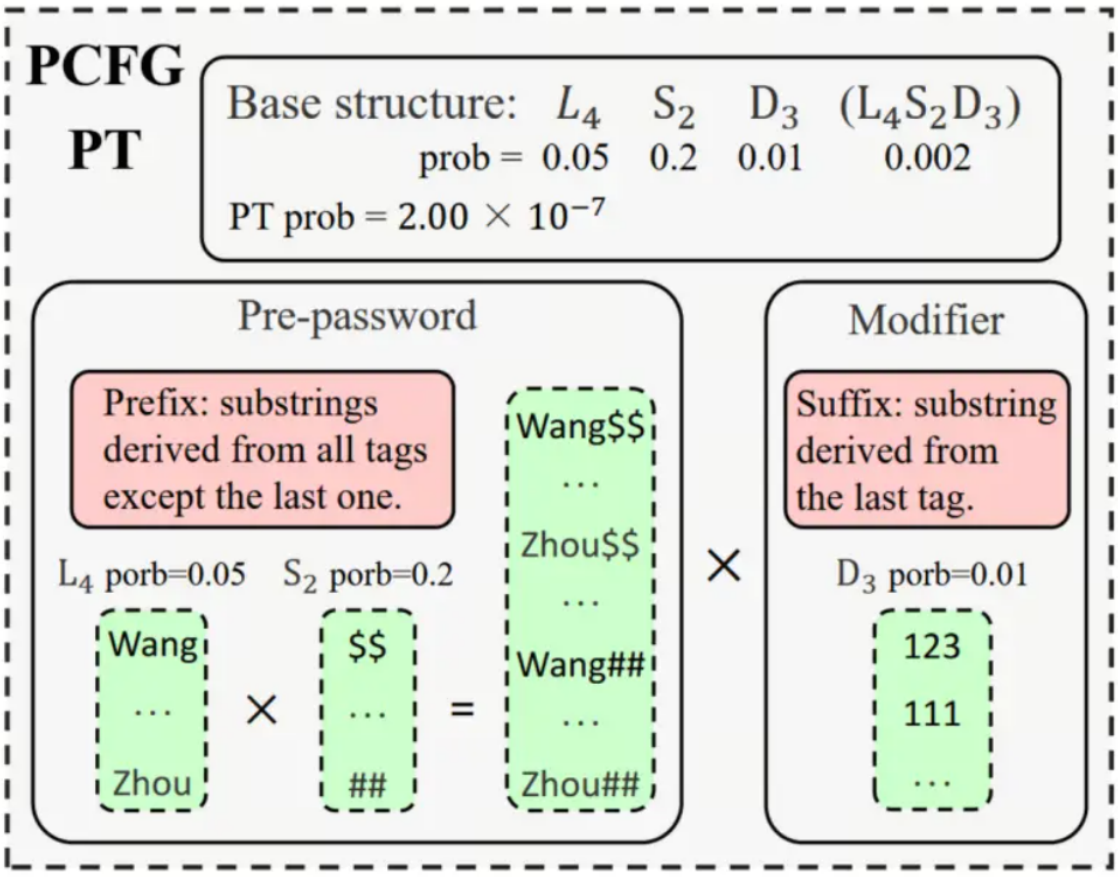
1.3 PCFG 串行生成

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。我们可以将模型中所有的 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。然后我们可以通过概率降序来生成猜测，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，就能生成相应的口令。对于后续的口令生成，则需要按照 pivot 的值，对该 preterminal 的 value 进行改造，再将生成的新的一系列 preterminal 按概率放回优先队列里。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。

1.4 PCFG 猜测生成并行化

通过上述串行化的实现过程，我们可以得知 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。考虑到 PCFG 往往用于没有猜测次数限制的场景中，我们并不需要严格按照降序生成口令。因此，我们可以通过可以一次取出多个口令猜测并行化问题，也可以一次为某个 segment 分配多个 value。一个基本的并行化方案如下图：

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。可



以看出，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。在本选题中，我们可以使用使用多线程、多进程、GPU 等技术实现口令猜测部分的并行化实现。

1.5 MD5 哈希算法

MD5 是一个常用的哈希算法。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。一个大致 MD5 算法的运算过程如下。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。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。通过使用 SIMD 并行地对多个消息（多个口令）进行 MD5 的运算，能够实现哈希过程的并行优化。

2 方法概述

2.1 SIMD

SIMD（单指令多数据）是一种并行计算技术，它允许一条指令同时对多个数据进行相同的操作。这种技术在现代处理器中广泛使用，可以显著提高数据并行任务的性能。SIMD 的核心思想是：

- 一条指令同时处理多个数据元素
- 数据被组织在特殊的宽寄存器中（如 128 位、256 位或 512 位）
- 相同的操作同时应用于寄存器中的所有数据

常见的 SIMD 指令集包括 Intel 的 SSE/AVX 和 ARM 的 NEON。在本实验中，我们将在 ARM 平台使用 NEON 指令集实现 MD5 哈希算法的并行化。NEON 是 ARM 架构的 SIMD 扩展指令集，广泛应用于移动设备和嵌入式系统。它提供 128 位宽向量寄存器，支持同时处理多个 8/16/32/64 位整数或 16/32 位浮点数，显著加速多媒体处理和数值计算任务。

2.2 多线程

2.2.1 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。

2.2.2 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（#pragmaomp）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。

2.2.3 pthread 与 openMP 对比

表 1: Pthread 与 OpenMP 对比分析

对比维度	Pthread (POSIX Threads)	OpenMP
编程模型	显式线程管理（手动创建/同步线程）	隐式并行（通过编译指令自动生成线程）
语法复杂度	需手动调用 API（如 `pthread_create`）	通过 `#pragma omp parallel` 指令简化
线程控制	精细控制（可指定线程属性、绑定 CPU 核）	高层抽象（依赖运行时调度，灵活性较低）
同步机制	需显式使用互斥锁（`pthread_mutex_t`）、条件变量	内置同步指令（如 `#pragma omp critical`）
数据共享	需手动管理共享/私有数据	通过 `shared/private` 子句自动划分
适用场景	需要低层控制的场景（如实时系统、自定义线程池）	快速并行化循环/代码块（如科学计算）
跨平台性	限于 POSIX 系统（Linux/Unix/macOS）	跨平台（支持 GCC、MSVC、ICC 等编译器）

2.3 多进程

多进程是在操作系统层面创建多个相互独立的进程，每个进程拥有自己的地址空间和资源副本，通过操作系统调度实现并发或并行。多进程编程的基本思路是将可并行或需隔离的工作拆分为相对独立的子任务，由主进程创建或复用子进程并发执行，子进程各自拥有独立地址空间，结果通过管道、套接字、共享内存等显式 IPC 机制传回主进程；主进程负责调度、监控、重启或回收子进程，并合理限制资源和并发度，以在保障稳定性和隔离性的同时充分利用多核优势。

本次实验中，我们将使用 mpi 完成多进程编程。MPI 是一种用于分布式并行计算的标准接口，用于在多进程（通常在不同节点或多核上）之间通过消息传递方式交换数据。其基本思路是：每个进程独立运行，有自己的地址空间，通过调用 MPI 提供的函数进行通信，而不共享内存，从而适合在集群或多节点环境下进行大规模并行计算。典型使用流程：

- 初始化与结束：在程序开始处调用 MPI_Init，在结束处调用 MPI_Finalize。
- 获取进程信息：通过 MPI_Comm_size(MPI_COMM_WORLD,&size) 得到总进程数，通过 MPI_Comm_rank(MPI_COMM_WORLD, &rank) 得到当前进程的编号（0 到 size-1）。
- 点对点通信：使用 MPI_Send/MPI_Recv 在两个进程之间传递数据，适合发送明确的消息或工作任务。需要指定目标/来源进程编号、消息标签（tag）等，以区分不同消息。
- 集合通信：使用 MPI_Bcast（广播）、MPI_Reduce（归约）、MPI_Allreduce、MPI_Scatter(分发)、MPI_Gather（收集）等函数，在所有进程或一组进程间进行常用模式的数据分发、汇总或归约操作，简化编程。
- 运行方式：在代码中包含 <mpi.h>，使用 MPIC++ 编译，指定启动的进程数及节点列表，MPI 运行时会启动多个进程并建立通信环境。

2.4 GPU 优化

CUDA 是 NVIDIA 开发的并行计算平台和编程模型，它允许开发者利用 GPU 的强大并行处理能力来加速计算密集型应用程序。通过 CUDA，程序员可以将部分代码从 CPU 卸载到 GPU 上执行，从而显著提升程序性能，特别是在处理大规模数据并行任务时。

CUDA 编程的核心概念是将计算任务分解为大量可以并行执行的线程。GPU 包含数千个轻量级处理核心，这些核心被组织成流式多处理器（SM），可以同时执行成千上万个线程。CUDA 程序通常由主机代码（在 CPU 上运行）和设备代码（在 GPU 上运行的内核函数）组成。内核函数定义了每个线程要执行的操作，而线程被组织成网格和线程块的层次结构。

CUDA 编程使用 C/C++ 语言的扩展，添加了特殊的关键字和函数来管理 GPU 内存和启动内核。程序员需要显式管理 CPU 和 GPU 之间的数据传输，包括分配 GPU 内存、复制数据到 GPU、执行内核函数，然后将结果复制回 CPU。CUDA 还提供了丰富的库和工具，如 cuBLAS、cuFFT、Thrust 等，帮助开发者更容易地实现高性能计算应用。

3 SIMD 优化

3.1 算法设计

SIMD 的优化主要体现在于对 MD5 哈希部分的优化。我们知道，对于串行的哈希算法的基本思路是通过 StringProcess 函数对输入字符串进行填充，使其长度符合 MD5 的要求，结果保存在 paddedMessage 中，并记录了处理后的总长度到 messageLength[0] 中。然后，将会对数据进行切分为多个 512bit 的切片，并使用 nblock 记录切片数量。对于每个 512bit 的切片，将会被进一步切分 16 个 32bit 的部分保存在下 x[i] 中。对于每个小块的长度是 32bit (4Byte)，即每个 x[i] 的值是数组 paddedMessage 的连续 4 个数据。接着是执行 4 轮 MD5 压缩变换（FF, GG, HH, II），每个回合使用不同的非线性函数（F/G/H/I）和不同常量，将消息 x 与当前变量 a, b, c, d 反复混合、旋转、累加。每一回合会调用 16 次对应的宏函数。

在这个过程中，我们着重关注函数 F、G、H、I、FF、GG、HH、II，这些函数所设计的运算不涉及条件判断，并且为较简单的运算（移位、与运算等等）。这就使得这些函数非常适合用 SIMD 进行并行化处理：一次性地让函数同时处理多个消息（口令），即可做到并行化。因此，我们的基本实现实思路是利用 NEON 指令集提供的它提供 128 位宽向量寄存器，同时处理 4 个条口令的 hash 过程。为此，我们需要在主函数中实现一次性取出 4 个口令，然后修改各个相关函数使得能够实现同时对 4 个口令执行相同的操作，以达成并行化的效果。

3.2 算法实现

首先是利用 neon 指令集中对应的针对 4 个 32 位向量的计算指令，实现 F、G、H、I、FF、GG、HH、II 的并行实现。

SIMD 优化

```
#define F_NEON(x, y, z) vorrq_u32(vandq_u32(x, y), vandq_u32(vmvnq_u32(x), z))
#define G_NEON(x, y, z) vorrq_u32(vandq_u32(x, z), vandq_u32(vmvnq_u32(z), y))
......
void MD5Hash_4(string inputs[4], uint32x4_t states[4]);
```

然后是 MD5 哈希函数的实现。我们使用同时输入四个口令保持在 inputs 中，然后我们利用 StringProcess 对 4 个口令依次进行预处理，并保存在 paddedMessages 中，接下来是逐个 block 的更新缓冲区 states，然后调用相关函数实现一次性进行 4 个 MD5 哈希的实现。

SIMD 优化

```
void MD5Hash_4(string inputs[4], uint32x4_t states[4])
{
  for (int i = 0; i < 4; i++)
  {
    paddedMessages[i] = StringProcess(inputs[i], &messageLengths[i]);
        if (i > 0 && messageLengths[i] != messageLengths[0]) {
            messageLengths[i] = messageLengths[0];
        }
  }

  states[0] = vdupq_n_u32(0x67452301);
  states[1] = vdupq_n_u32(0xefcdab89);
  states[2] = vdupq_n_u32(0x98badcfe);
  states[3] = vdupq_n_u32(0x10325476);

  for (int block = 0; block < n_blocks; block ++)
  {
    uint32x4_t a = states[0], b = states[1], c = states[2], d = states[3];

        uint32x4_t x[16];
        for(int i = 0; i < 16; i++) {
            uint32_t words[4];
            for(int j = 0; j < 4; j++) {
                const Byte* blockStart = paddedMessages[j] + block * 64;
                words[j] = (blockStart[i*4]) |
                           (blockStart[i*4 + 1] << 8) |
                           (blockStart[i*4 + 2] << 16) |
                           (blockStart[i*4 + 3] << 24);
            }
            x[i] = vld1q_u32(words);
        }
    ......

    states[0] = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(states[0])));
    states[1] = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(states[1])));
    states[2] = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(states[2])));
    states[3] = vreinterpretq_u32_u8(vrev32q_u8(vreinterpretq_u8_u32(states[3])));

}
```

3.3 实验结果与分析

通过测试串行和 SIMD 并行化在不同优化条件下的 hash time 作为评价指标，我们可以等到如下数据：

	无优化	O1	O2
串行实现	9.837	3.159	3.185
SIMD 优化	14.535	2.086	2.044
加速比	0.672	1.514	1.558

表 2: 性能测试结果 (单位:s)

从表中的数据我们可以看到，在不采取任何编译优化的前提下，SIMD 优化的算法明显劣于串行算法，出现了负优化；然而采用编译优化之后，我们可以看到 SIMD 优化后的 hash time 明显低于串行算法, 成功实现了相较于串行的加速，加速比达到 1.5。并且我们可以发现，随着编译优化的程度不断加深，加速的效果也有了明显的提升。

至此，我们可以初步等到一个结论，使用编译优化能够明显提高程序的执行效率，并且若实现了相对于串行的加速，加速比会有小幅提升。针对这一现象，我们可以得知出现这一现象的可能原因。

- 在未开启编译优化（-O0）的情况下，并行 MD5 实现未能实现预期的加速效果，这主要是因为 NEON 向量指令无法形成有效的指令流水线，频繁发生寄存器溢出到内存的情况，函数调用开销完全保留，内存访问模式保持源代码顺序，无法利用缓存预取等硬件特性。这些因素共同导致即使采用了并行计算策略，其性能提升也被底层实现的低效所抵消。
- 当采用-O1 基础优化级别时，编译器会启用一系列关键优化：首先进行基本的指令调度优化，使 NEON 指令能够形成较紧密的流水线；其次实施寄存器分配改进，显著减少 NEON 寄存器的内存溢出操作；同时对循环结构进行初步优化，并选择性内联小型 NEON 操作函数。这些优化使得计算密集型部分在整体执行时间中的占比显著提升，从而实现了可观测的加速效果。

4 多线程优化

4.1 算法设计

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。

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。

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。

4.2 算法实现

针对 pthread 和 openMP 的实现，我们着重关注 generate 中的两个循环操作。由于在这个过程中，不同循环时没有前后的依赖关系，比较独立，因此我们首先并行化的一个基本思路就是将整个生成口令任务的循环划分为不不同的部分，分别由不同的线程同时并行生成猜测，最后将不同线程生成的口令及进行汇总，进行后续的 hash 操作。

对于 openMp 的实现较为简单，当处理单段内容时，直接使用 #pragma omp parallel for 并行填充预分配内存的 guesses 数组；当处理多段组合内容时，先串行生成前缀字符串，然后并行拼接后缀并复制到结果数组，通过 schedule(static) 实现均匀任务划分，同时利用 memcpy 和预分配内存避免线程竞争，显著提升了密码猜测的生成效率。

openMP 实现

```
void PriorityQueue::Generate(PT pt)
{
    // 设置 OpenMP 使用的线程数量
    int num_threads = 8;
    omp_set_num_threads(num_threads);

    if (pt.content.size() == 1)
    {
        ......
        #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;
    }
    else
    {
        ......
        #pragma omp parallel for schedule(static)
        for (int i = 0; i < total_count; i++) {
            const std::string &suffix = a->ordered_values[i];
            const size_t suffix_len = suffix.size();
            std::string new_guess;
            new_guess.resize(prefix_len + suffix_len);
            memcpy(&new_guess[0], prefix.data(), prefix_len);
            memcpy(&new_guess[prefix_len], suffix.data(), suffix_len);
            guesses[offset + i] = std::move(new_guess);
```

```
        }
        total_guesses += total_count;
    }
}
```

对于 pthread 的实现，我们采取以下策略：

- 任务分块并行：将密码生成任务均匀分配给 8 个工作线程（通过 ThreadData 结构体传递参数），每个线程处理一个数据块（chunk_size），通过局部缓存 (local_guesses) 减少锁竞争。
- 智能任务粒度控制：当总密码数小于 5000 时自动回退到串行处理（避免了小任务并行化的额外开销），体现了并行编程中”任务粒度”的优化思想。
- 锁优化策略：采用”局部计算 + 全局加锁合并”的模式，线程先在无锁环境下构建局部结果，最后通过互斥锁 (pthread_mutex_t) 安全地合并到全局容器，大幅减少了锁争用。
- 内存预分配：在单段处理时直接 resize 目标容器，多段处理时使用 string 预分配和 memcpy 优化字符串拼接，提高了内存访问效率。

pthread 实现

```
struct ThreadData {
    segment *a;int start;
    int end;vector<string> *guesses;
    pthread_mutex_t *mutex;
    int *total_guesses;string prefix;
};

void *GenerateGuesses(void *arg) {
    ThreadData *data = (ThreadData *)arg;

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

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

    // 合并局部缓存到全局缓存时加锁
    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;
}
....
//后续再generate中调用相关的线程函数实现多线程并行口令生成
```

4.3 结果分析

探究 pthread/openMP 的实现效果

多线程并行的主要优化在于口令生成部分，因此我们使用 guess time 作为相关的测试指标。在线程数为 8，编译优化为 O2 的情况下，我们测得的实验结果如下表：

表 3: 性能测试结果（8 线程）（单位：s）

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。

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。

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：

表 4: 不同线程数的执行时间（单位：s）

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

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。

5 多进程优化

5.1 算法设计

在这部分，我们将使用 mpi 完成口令生成部分的多进程实现。针对并行化的需要，我们有两种优化思路，一种是将同一个 PT 的口令生成任务划分为不同部分，由不同的进程执行，最后汇总到主进程实现。令猜测部分最核心最关键的操作在于 generate 函数中的两处循环，我们的主要目标便是利用 mpi 对这两处循环进行并行化处理。MPI 并行化主要体现在两个关键步骤：口令生成和队列更新。通过将任务分配给多个进程，每个进程负责一部分工作，然后通过 MPI 通信进行数据汇总和同步，从而实现并行计算。

另一种实现优化的方法是同时取出多个 PT，由多个线程分别不同执行 PT 的猜测任务，从而实现在 PT 层面的并行。我们可以整体实现采用主从模式，进程 rank 0 作为主进程，负责模型训练、优先队列的初始化和任务的分发；其他进程作为从进程，负责接收主进程分发的任务并进行计算，最后将计算结果返回给主进程。通过函数 PopNextBatchMPI 实现了一次性从优先队列中取出多个 PT，然后将猜测任务分配给多个进程进行猜测实现，每个进程内部单独使用 generate 函数生成对应的口令，当生成新的 PT 再将其依次放回主进程，等待下一次任务的划分。

5.2 算法实现

单个 PT 并行实现

在 guessing_mpi.cpp 中，我们使用 GenerateMPI 方法实现了口令生成的并行化：计算每个进程需要处理的连续块大小 chunk_size，根据进程的 rank 计算该进程处理的起始索引 start 和结束索引 end，最后一个进程的 end 为 pt.max_indices[0]，以确保处理完所有元素，每个进程在其连续的索引范围内生成猜测，并将其添加到本地的 guesses 向量中，实现任务在不同进程中的划分。然后是数据的汇总，使用 MPI_Allreduce 函数将每个进程生成的口令数量进行汇总，得到全局生成的口令总数。

单 PT 并行实现

```
1 void PriorityQueue::GenerateMPI(PT pt)
2 {
3     CalProb(pt);
4
5     int rank, size;
6     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
7     MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```

    // 对于只有一个segment的PT，直接遍历生成其中的所有value即可
    if (pt.content.size() == 1)
    {
        .....
        int chunk_size = pt.max_indices[0] / size;
        int start = rank * chunk_size;
        int end = (rank == size - 1) ? pt.max_indices[0] : start + chunk_size;

        for (int i = start; i < end; i++)
        {
            string guess = a->ordered_values[i];
            guesses.emplace_back(guess);
        }
    }
    else
    {
        ......
        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] : start
            + chunk_size;

        for (int i = start; i < end; i++)
        {
            string temp = guess + a->ordered_values[i];
            guesses.emplace_back(temp);
        }
    }

    long long local_add = static_cast<long long>(guesses.size());
    long long global_add = 0;
    MPI_Allreduce(&local_add, &global_add,
                  1, MPI_LONG_LONG, MPI_SUM, MPI_COMM_WORLD);

    total_guesses += static_cast<int>(global_add);
}
```

PopNextMPI 方法是 PriorityQueue 类中的并行版本处理函数，借助 MPI 实现并行化。首先获取当前进程的排名 rank 和进程总数 size，记录 guesses 初始大小；接着从优先队列移除队首元素 PT，调用 GenerateMPI 并行生成猜测；通过 MPI_Allreduce 汇总各进程生成的猜测数量并更新 total_guesses；再由当前队首 PT 生成新的 PT 集合，计算新 PT 的概率并依据概率插入到优先队列合适位置，以此高效完成口令猜测任务的并行处理与优先队列的动态更新。

pthread 实现

```
void PriorityQueue::PopNextMPI()
{
    int rank, size;
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    std::size_t before_sz = guesses.size();

    PT pt = priority.front();
    priority.erase(priority.begin());
    GenerateMPI(pt);

    std::size_t local_added  = guesses.size() - before_sz;
    std::size_t global_added = 0;
    MPI_Allreduce(&local_added, &global_added, 1,
                  MPI_UNSIGNED_LONG_LONG, MPI_SUM, MPI_COMM_WORLD);

    total_guesses += static_cast<long long>(global_added);

    std::vector<PT> new_pts = pt.NewPTs();
    for (PT &child : new_pts)
    {
        CalProb(child);

        for (auto it = priority.begin(); it != priority.end(); ++it)
        {
            if (it != priority.end() - 1 && it != priority.begin())
            {
                if (child.prob <= it->prob && child.prob > (it + 1)->prob)
                {   priority.emplace(it + 1, child); break; }
            }
            if (it == priority.end() - 1)
            {   priority.emplace_back(child); break; }
            if (it == priority.begin() && it->prob < child.prob)
            {   priority.emplace(it, child);  break; }
        }
    }
}
```

最后是在主程序中完成相关方法的调用，借助 MPI_Init 对 MPI 环境进行初始化，同时使用 MPI_Comm_rank 和 MPI_Comm_size 获取当前进程的排名以及进程总数，在循环里调用 PopNextMPI 方法生成猜测，运用 MPI_Allreduce 对各进程生成的猜测数量进行汇总，并且依据条件对猜测进行哈希处理，最后调用 MPI_Finalize 结束 MPI 环境。

多 PT 并行实现

首先是主进程进行任务的分发。在 PopNextBatchMPI 方法中，主进程主进程从优先队列中取出 batch_size 个 PT 对象，存储在 global_batch 中，然后根据进程数量 size 对 global_batch 进行划分，计算每个进程应处理的 PT 数量，最后使用 MPI_Send 函数将每个从进程应处理的 PT 数量和 PT 对象的数据发送给对应的从进程。

pthread 实现

```
if (rank == 0)
    {
        // 主进程: 将优先队列中的PT分发给各个从进程
        std::vector<PT> global_batch;
        for (int i = 0; i < batch_size && !priority.empty(); ++i)
        {
            global_batch.push_back(priority.front());
            priority.erase(priority.begin());
        }

        int local_batch_size = global_batch.size() / size;
        int remainder = global_batch.size() % size;

        for (int i = 1; i < size; ++i)
        {
            int start = i * local_batch_size + std::min(i, remainder);
            int end = start + local_batch_size + (i < remainder ? 1 : 0);
            int send_size = end - start;
            MPI_Send(&send_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
            for (int j = start; j < end; ++j)
            {
                std::string pt_data = global_batch[j].serialize();
                int data_size = pt_data.size();
                MPI_Send(&data_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                MPI_Send(pt_data.c_str(), data_size, MPI_CHAR, i, 0, MPI_COMM_WORLD);
            }
        }

        // 主进程处理自己的任务
        int start = 0;
        int end = local_batch_size + (0 < remainder ? 1 : 0);
        for (int j = start; j < end; ++j)
        {
            local_batch.push_back(global_batch[j]);
        }
    }
```

然后是从进程接收主进程分发的 PT 对象，并进行猜测生成和新 PT 对象的生成。在这个阶段，将使用 MPI_Recv 函数接收主进程发送的 PT 数量和对象的数据，对每个接收到的 PT 对象，调用 Generate 函数生成猜测，调用 NewPTs 函数生成新的 PT 对象，并调用 CalProb 函数计算新 PT 对象的概率。

pthread 实现

```
    // 从进程: 接收主进程分发的PT
    int recv_size;
    MPI_Recv(&recv_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (int i = 0; i < recv_size; ++i)
    {
```

```
        int data_size;
        MPI_Recv(&data_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        std::string pt_data(data_size, '\0');
        MPI_Recv(&pt_data[0], data_size, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        PT pt;
        std::istringstream iss(pt_data);
        pt.deserialize(iss);
        local_batch.push_back(pt);
    }
```

在使用 MPI 进行并行编程时，需要在不同的进程之间传递数据。MPI 的通信函数只能处理基本数据类型和连续的内存块。而 segment 和 PT 是自定义的复杂数据结构，不能直接通过 MPI 函数进行传输。通过序列化，将这些复杂对象转换为连续的字节流，就可以使用 MPI 函数进行传输；接收方再通过反序列化将字节流转换回原来的对象。我们以 segment 为例实现序列化和反序列化。在 segment 中，我们实现序列化和反序列化函数。其中，serialize 方法将 segment 对象的各个成员变量信息存储到一个 stringstream 中，最后将其转换为字符串返回；deserialize 方法从输入的字符串中解析出各个成员变量的值，并将其赋给当前 segment 对象。

MPI 实现

```
    // 序列化为字符串
    std::string serialize() const {
        std::ostringstream oss;
        oss << type << " " << length << " " << total_freq << " ";
        oss << ordered_values.size() << " ";
        for (const auto& s : ordered_values) {
            oss << s.size() << " " << s;
        }
        oss << " " << ordered_freqs.size() << " ";
        for (int f : ordered_freqs) oss << f << " ";
        return oss.str();
    }

    // 反序列化
    void deserialize(std::istringstream& iss) {
        size_t n, m;
        iss >> type >> length >> total_freq >> n;
        ordered_values.clear();
        for (size_t i = 0; i < n; ++i) {
            size_t len;
            iss >> len;
            std::string s(len, ' ');
            iss.read(&s[0], len);
            ordered_values.push_back(s);
        }
        iss >> m;
        ordered_freqs.clear();
        for (size_t i = 0; i < m; ++i) {
```

```
29              int f;
30              iss >> f;
31              ordered_freqs.push_back(f);
32          }
33      }
```

5.3 结果实现

多线程并行的主要优化在于口令生成部分，因此我们使用 guess time 作为相关的测试指标。在线程数为 8，编译优化为 O2 的情况下，我们测得的实验结果如下表：

表 5: 不同的执行优化条件下的 guess time（单位：s）

guess time	串行实现	单 PT 并行	多 PT 并行
O1	0.649	1.787	0.3569
O2	0.599	1.696	0.3114

结果显示，在单 PT 实现 mpi 优化程序出现了负优化。其可能的产生原因如下：

1. 数据汇总频繁：在 PopNextMPI 中，每次调用 MPI_Allreduce 函数汇总各进程生成的猜测数量，会带来一定的通信开销。如果生成猜测的计算量较小，而通信操作频繁，通信开销可能会占据大部分运行时间，导致整体性能下降。

2. MPI_Allreduce 是一个全局同步操作，所有进程必须等待所有其他进程完成数据发送和接收才能继续执行。如果进程之间的执行速度差异较大，全局同步会导致部分进程长时间等待，降低并行效率。在多进程环境中，如果多个进程同时访问共享资源，可能会出现锁竞争问题。锁竞争会导致进程频繁进入阻塞状态，增加同步开销，降低性能。

而多 PT 实现了优化，通过 MPI 并行，口令生成任务被分解为多个子任务，分配给多个进程同时执行，每个进程独立处理自己分到的 PT（概率模板），大大提升了 CPU 资源的利用率，实现了多核/多节点的协同计算。相较于串行只能单线程逐步生成口令，MPI 并行让多个进程在同一时间并发生成大量口令，显著缩短了总运行时间，从而实现了整体任务的加速。

收获与反思通过本次实验，我们可以知道合理设计主从模式可有效分工—主进程负责全局调度，从进程专注并行计算，能够避免串行处理的瓶颈。通过 MPI 通信机制可以实现进程间高效协作，尤其在数据序列化与批量传输优化后，通信开销得到有效控制。然而，在这个实验中也发现了一些问题，诸如频繁同步操作会导致进程空闲等待，可通过非阻塞通信或流水线技术优化；各个进程之间的通信，也会代码新的开销。

6 GPU 优化

6.1 算法设计

口令猜测部分是整个算法的瓶颈，其中 generate 函数存在两个串行循环，我们可以将需要大量重复计算的任务分配给 GPU 的多个线程，每个线程独立处理一小部分数据，实现大规模并行。因此，我们将使用 cuda 实现这一过程，完成口令猜测的并行化。

实现并行化的过程与 mpi 类似，同样有两种思路实现。

- 其一，将一个的 PT 的口令生成任务分割为不同部分，交给 GPU 并行处理。
- 其二，一次性往 gpu 上一次装载多个 PT 进行生成。

6.2 算法实现

基础实现

首先是 cuda 内核函数的实现。使用内核函数 generate_guesses_kernel 处理只包含单个 segment 的密码生成。具体实现方式是：每个 GPU 线程根据其全局索引 $(blockDim.x*blockIdx.x+threadIdx.x)$ 从输入的字符数组 values 中提取对应的密码字符串片段，通过逐字符循环拷贝到输出缓冲区 out 的指定位置，并在末尾添加字符串结束符""，从而实现多个密码字符串的并行生成，其中每个线程独立处理一个完整的密码字符串。

cuda 实现

```
__global__ void generate_guesses_kernel(const char* values, int value_len, int
    value_count, char* out) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < value_count) {
        // 拷贝value到输出
        for (int j = 0; j < value_len; ++j)
            out[idx * MAX_LEN + j] = values[idx * value_len + j];
        out[idx * MAX_LEN + value_len] = '\0';
    }
}
```

使用内核函数 generate_guesses_with_preffx_kernel 处理包含处理包含多个 segment 的复合密码生成。具体实现方式是：每个 GPU 线程根据其全局索引获取一个独立的任务，先将共享的前缀字符串 (preffx) 拷贝到输出缓冲区的对应位置，然后将该线程负责的特定值 (values[idx]) 紧接着拼接在前缀之后，最后添加字符串结束符，从而实现多个" 前缀 + 后缀" 组合密码的并行生成，用于多段密码结构的高效批量生成。

cuda 实现

```
__global__ void generate_guesses_with_prefix_kernel(
    const char* prefix, int prefix_len,
    const char* values, int value_len,
    int value_count, char* out)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx < value_count) {
        // 拼接prefix
        for (int i = 0; i < prefix_len; ++i)
            out[idx * MAX_LEN + i] = prefix[i];
        // 拼接value
        for (int j = 0; j < value_len; ++j)
            out[idx * MAX_LEN + prefix_len + j] = values[idx * value_len + j];
        // 结尾符
        out[idx * MAX_LEN + prefix_len + value_len] = '\0';
```

```
    }
}
```

然后是在 generate 函数中实现 cuda 并行。首先将所有待处理的 value 打包成连续内存 (h_values)，便于 GPU 高效访问；然后进行内存复用检查，只在需要时分配或扩容 GPU 内存 (d_values, d_out, d_preffx)。然后使用 cudaMemcpyAsync 异步将数据传输到 GPU，配置合适的线程块大小和网格大小，通过 CUDA Stream 异步启动对应的内核函数 (generate_guesses_kernel 或 generate_guesses_with_preffx_kernel) 进行并行密码生成。最后，异步将生成结果从 GPU 传回主机内存，通过 cudaStreamSynchronize 等待所有操作完成，最后将结果逐一添加到 guesses 容器中，实现了从串行字符串拼接到并行批量生成的性能飞跃

cuda 实现

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

    static cudaStream_t stream = nullptr;
    if (!stream) cudaStreamCreate(&stream);
    ......
    int value_count = pt.max_indices[0];
    int value_len = 0;
    if (!a->ordered_values.empty())
        value_len = a->ordered_values[0].size();

    // 打包所有value为连续内存
    std::vector<char> h_values(value_count * value_len);
    for (int i = 0; i < value_count; ++i)
        memcpy(&h_values[i * value_len], a->ordered_values[i].c_str(), value_len);

    // 内存复用: 只在需要时分配或扩容
    size_t values_size = value_count * value_len;
    size_t out_size = value_count * MAX_LEN;
    if (!d_values || last_values_size < values_size) {
        if (d_values) cudaFree(d_values);
        cudaMalloc(&d_values, values_size);
        last_values_size = values_size;
    }
    if (!d_out || last_out_size < out_size) {
        if (d_out) cudaFree(d_out);
        cudaMalloc(&d_out, out_size);
        last_out_size = out_size;
    }

    cudaMemcpyAsync(d_values, h_values.data(), values_size,
        cudaMemcpyHostToDevice,stream);

    // 启动kernel
    int threads = 512;
```

```
    int blocks = (value_count + threads - 1) / threads;
    //generate_guesses_kernel<<<blocks, threads>>>(d_values, value_len, value_count,
        d_out);
    generate_guesses_kernel<<<blocks, threads, 0, stream>>>(d_values, value_len,
        value_count, d_out);

    //  拷贝结果回主机
    std::vector<char> h_out(out_size);
    //cudaMemcpy(h_out.data(), d_out, out_size, cudaMemcpyDeviceToHost);
    cudaMemcpyAsync(h_out.data(), d_out, out_size, cudaMemcpyDeviceToHost, stream);

    cudaStreamSynchronize(stream);

    //  插入guesses
    for (int i = 0; i < value_count; ++i)
        guesses.emplace_back(&h_out[i * MAX_LEN]);
    total_guesses += value_count;
```

进阶实现

在进阶实现中我们实现可以尝试往 gpu 上一次装载多个 PT 进行生成。基本的实现思路为在 CPU 端维护按概率排序的 PT 优先队列，每次从队列头部动态提取多个 PT 组成处理批次，通过计算每个 PT 得到可能组合数，并累加控制总组合数不超过预设阈值以避免内存溢出；将对应的 PT 批量传输至 GPU 后，由 CUDA 内核实施两级并行化处理——每个 PT 分配一个线程块，块内各线程并行生成该 PT 的不同组合变体，通过预计算的字符段偏移量表实现高效字符串拼接，最终将生成的猜测口令写回统一输出缓冲区。

首先，在 gpu_structs.h 中定义了 GPU 需要使用的数据结构，这些结构体是主机 (CPU) 和设备 (GPU) 共用的，用于在两者之间传递数据。

cuda 实现

```
struct PT_GPU {
    int seg_num;
    int curr_indices[MAX_SEG];
    int max_indices[MAX_SEG];
    int types[MAX_SEG];
    int type_indices[MAX_SEG];
};

struct Segment_GPU {
    int value_num;
    int value_offset;
};
```

然后是 CUDA 内核函数 batch_generate_kernel 的实现。它实现了高效的并行密码猜测生成，它通过将密码模板 (PT) 分解为多个字符段，利用 GPU 的并行计算能力批量生成所有可能的组合。每个 CUDA 线程处理一个独特的密码组合：首先根据全局索引确定所属的密码模板，然后解码出各字符段的索引位置，接着拼接对应的字符值形成完整密码，最后将结果写入输出缓冲区。

cuda 实现

```
__global__ void batch_generate_kernel(
    PT_GPU* pts, Segment_GPU* letters, Segment_GPU* digits, Segment_GPU* symbols,
    char* all_values,
    char* output, int* output_offsets, int* pt_offsets, int batch_size, int
        total_combos)
{
    int global_idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (global_idx >= total_combos) return;

    // 找到属于哪个PT
    int pt_idx = 0;
    while (pt_idx < batch_size && global_idx >= pt_offsets[pt_idx + 1]) ++pt_idx;
    if (pt_idx >= batch_size) return;

    int combo_idx = global_idx - pt_offsets[pt_idx];
    PT_GPU pt = pts[pt_idx];

    // combo_idx解码为curr_indices
    int curr_indices[MAX_SEG];
    int div = combo_idx;
    for (int i = pt.seg_num - 1; i >= 0; --i) {
        curr_indices[i] = div % pt.max_indices[i];
        div /= pt.max_indices[i];
    }
    ......
    // 写入输出
    int offset = output_offsets[global_idx];
    for (int i = 0; i < guess_len; ++i) {
        output[offset + i] = guess[i];
    }
    output[offset + guess_len] = '\0';
}
```

接着是在主机端实现的部分。在 guessing_new.cpp 中的 BatchGenerateGPU 函数处理主机与设备之间的数据传输和内核调用，主要的代码如下：

cuda 实现

```
void PriorityQueue::BatchGenerateGPU(const vector<PT>& batch)
{
    ......
    // 打包PT数据
    for (int i = 0; i < batch_size; ++i) {
        const PT& pt = batch[i];
        for (int j = 0; j < pt.content.size(); ++j) {
            h_pts[i].curr_indices[j] = pt.curr_indices[j];
            h_pts[i].max_indices[j] = pt.max_indices[j];
            h_pts[i].types[j] = pt.content[j].type;
```

```
        .....
        }
        h_pts[i].seg_num = pt.content.size();
    }
    ......
    // 打包segment数据
    for (int i = 0; i < letters_num; ++i) {
        h_letters[i].value_num = m.letters[i].ordered_values.size();
        h_letters[i].value_offset = offset;
        for (const auto& val : m.letters[i].ordered_values) {
            all_values.insert(all_values.end(), val.begin(), val.end());
            all_values.push_back('\0');
            offset += val.size() + 1;
        }
    }
    ......
    // 计算每个PT的组合数和全局偏移
    std::vector<int> pt_offsets(batch_size + 1, 0);
    int total_combos = 0;
    for (int i = 0; i < batch_size; ++i) {
        int combos = 1;
        for (int j = 0; j < h_pts[i].seg_num; ++j) {
            combos *= h_pts[i].max_indices[j];
        }
        pt_offsets[i] = total_combos;
        total_combos += combos;
    }
    pt_offsets[batch_size] = total_combos;
    ......
    // 拷贝到GPU
    cudaMalloc(&d_pts, batch_size * sizeof(PT_GPU));
    cudaMemcpy(d_pts, h_pts, batch_size * sizeof(PT_GPU), cudaMemcpyHostToDevice);
    ......
    // 分配输出空间和offsets
    std::vector<int> output_offsets(total_combos);
    for (int i = 0; i < total_combos; ++i) output_offsets[i] = i * max_guess_len;
    cudaMalloc(&d_output, total_combos * max_guess_len);
    cudaMalloc(&d_output_offsets, total_combos * sizeof(int));
    cudaMemcpy(d_output_offsets, output_offsets.data(), total_combos * sizeof(int),
        cudaMemcpyHostToDevice);

    // 启动kernel
    int threads = 256;
    int blocks = (total_combos + threads - 1) / threads;

    launch_batch_generate_kernel(d_pts, d_letters, d_digits, d_symbols, d_output,
        d_output_offsets, d_pt_offsets,
        batch_size, total_combos, threads, blocks);

```

```
58      //  拷贝结果回主机
59      char* h_output = new char[total_combos * max_guess_len];
60      cudaMemcpy(h_output, d_output, total_combos * max_guess_len,
            cudaMemcpyDeviceToHost);
61      ......
62  }
```

6.3 结果实现

在 gpu 共享平台测得实验数据，如下表：

表 6: 性能测试结果（单位：s）

guess time	串行实现	CUDA 优化
无优化	7.528	8.569
O1 优化	0.432	0.757
O2 优化	0.435	0.638

我们可以看到使用 cuda 优化的程序的 guess time 在 O2 优化下为 0.638s，明显大于串行的 0.435s。这表明在不同的优化条件下，使用了 cuda 进行 gpu 编程的程序，在 guess time 方面明显差于串行实现，出现了负优化。

针对出现负优化的现象，我们使用 nvprof 对当前处于 O2 优化的 cuda 程序进行性能测试。根据这个 nvprof 性能分析报告，可以看出 GPU 执行时间主要分布在：CUDA 内存拷贝从设备到主机 (DtoH) 耗时 95.382ms 占 84.52%，而单纯的密码生成内核仅耗时 683.48ffs 占 0.61%；API 调用方面，cudaMemcpyAsync 占用 65.35% 的时间达 345.90ms，cudaStreamCreate 占 32.95% 达 174.38ms。这个性能剖析显示了负优化的一个重要原因：实际 GPU 计算时间 (内核执行) 仅占总时间的约 10%，而数据传输时间占据了近 90%，表明当前的密码生成任务，被频繁的 CPU-GPU 数据传输拖累，加上 516 次高频的小批次内存拷贝调用，使得传输开销完全压倒了并行计算的收益。进一步的优化：1. 应该在于降低数据传输的开销，减少内存传输频率，尽量在 GPU 上保留数据，减少 CPUffGPU 传输，可以使用 GPU 内存池，避免频繁分配/释放；2. 增大批处理规模，不要每个 PT 都调用 GPU，而是在一定数量再处理。

7 其它优化（新增内容）

7.1 SIMD+openMP 优化

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。

实验测得数据如下：

表 7: 性能测试结果（8 线程）（单位：s）

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

- 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 hashtime 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。
- 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。
- 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。
- 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。

7.2 gpu 加速的 MD5hash 过程

在前面的过程中，我们已经实现了使用 SIMD 对 MD5 哈希过程实现了并行化，现在我们使用 gpu 进行该过程的加速，并于 SIMD 方法进行比较。由于单个字符串的 MD5 计算不适合用 GPU 加速（串行步骤多，数据量小），所以我们实现批量字符串 MD5 计算，让每个线程负责一个字符串，所有线程并行计算。

实现思路

- 数据准备：
  将所有待哈希的字符串打包为连续内存（如二维 char 数组），并记录每个字符串的长度。
  分配一块 GPU 内存用于输入字符串，一块用于输出 MD5 结果（每个结果 4 个 bit32）。
- 核函数设计
  每个线程处理一个字符串：
  读取自己的字符串和长度。

按 MD5 算法流程（padding、分块、循环、FGHI 等）独立计算。
将结果写入输出数组。

- 主机端流程：
  打包所有输入字符串和长度，拷贝到 GPU。
  启动核函数，每个线程处理一个字符串。
  拷贝所有 MD5 结果回主机。

7.2.1 具体实现过程

实现 CUDA 全局内核函数。实现了 GPU 上的批量并行 MD5 哈希计算：每个 CUDA 线程通过全局索引 ($blockDim.x * blockIdx.x + threadIdx.x$) 定位到一个特定的输入字符串，调用 md5_single 函数计算其 MD5 哈希值，并将 4 个 32 位的哈希结果写入到对应的输出位置，从而实现数千个字符串同时并行计算 MD5 的高效处理。

哈希过程的 cuda 实现

```
__global__ void md5_batch_kernel(const char* inputs, const int* lengths, int max_len,
    int count, bit32* out) {
    int idx = blockDim.x * blockIdx.x + threadIdx.x;
    if (idx >= count) return;
    const char* str = inputs + idx * max_len;
    int len = lengths[idx];
    bit32 result[4];
    md5_single(str, len, result);
    for (int i = 0; i < 4; ++i)
        out[idx * 4 + i] = result[i];
}
```

下面是批量 MD5 哈希计算的主机端接口函数，实现了完整的 CPU-GPU 协同处理流程：首先将变长的字符串向量转换为 GPU 友好的固定长度字符数组格式（每个字符串占用 64 字节），同时记录每个字符串的实际长度；然后在 GPU 上分配相应的内存空间用于存储输入数据、长度信息和输出结果；通过 cudaMemcpy 将预处理后的数据从主机内存传输到 GPU 显存；配置并启动 CUDA 内核（256 个线程 per 块，根据数据量计算所需块数）进行大规模并行 MD5 哈希计算；计算完成后将所有哈希结果从 GPU 传回 CPU 内存；最后释放所有 GPU 内存资源，从而实现了对数千个字符串的高效批量哈希处理，显著提升了密码破解系统中哈希计算的性能。

哈希过程的 cuda 实现

```
void md5_batch_host(const std::vector<std::string>& inputs, std::vector<bit32>&
    out_hash) {
    int N = inputs.size();
    int max_len = 64; // 假设最大64字节
    std::vector<char> h_inputs(N * max_len, 0);
    std::vector<int> h_lengths(N, 0);
    for (int i = 0; i < N; ++i) {
        memcpy(&h_inputs[i * max_len], inputs[i].c_str(), inputs[i].size());
        h_lengths[i] = inputs[i].size();
    }
```

```
    char *d_inputs;
    int *d_lengths;
    bit32 *d_out;
    cudaMalloc(&d_inputs, N * max_len);
    cudaMalloc(&d_lengths, N * sizeof(int));
    cudaMalloc(&d_out, N * 4 * sizeof(bit32));
    cudaMemcpy(d_inputs, h_inputs.data(), N * max_len, cudaMemcpyHostToDevice);
    cudaMemcpy(d_lengths, h_lengths.data(), N * sizeof(int), cudaMemcpyHostToDevice);

    int threads = 256;
    int blocks = (N + threads - 1) / threads;
    md5_batch_kernel<<<blocks, threads>>>(d_inputs, d_lengths, max_len, N, d_out);

    out_hash.resize(N * 4);
    cudaMemcpy(out_hash.data(), d_out, N * 4 * sizeof(bit32), cudaMemcpyDeviceToHost);

    cudaFree(d_inputs);
    cudaFree(d_lengths);
    cudaFree(d_out);
}
```

最后是在主函数 mian 中调用我们实现的并行化处理多批次的猜测数据的函数 md5_batch_host，从而实现 hash 过程的并行化。

哈希过程的 cuda 实现

```
if (curr_num > 1000000)
    {
        auto start_hash = system_clock::now();

        std::vector<bit32> out_hash;
        md5_batch_host(q.guesses, out_hash);

        auto end_hash = system_clock::now();
        auto duration = duration_cast<microseconds>(end_hash - start_hash);
        time_hash += double(duration.count()) * microseconds::period::num /
            microseconds::period::den;

        history += curr_num;
        curr_num = 0;
        q.guesses.clear();
    }
```

7.2.2 实验结果

我们将使用编译指令如下编译指令实现程序的测试：

- nvcc -O2 -DUSE_CUDA main.cpp train.cpp guessing.cpp md5.cpp md5_cuda.cu -o test.exe

同时，由于我们在前面的实验中已经实现了 hash 过程的 SIMD 并行化，为了避免不同测试环境

的影响，我们在相同的 gpu 平台重新测试实验，获得 hash time 数据，记录在下表，并与 cuda 并行的测试结果进行比较。

实验的测试结果如下表所示：

表 8: 性能测试结果 hash time（单位：s）

编译优化	串行实现	SIMD 优化	cuda 优化
无优化	6.197	10.234	1.041
O1 优化	2.249	1.485	0.8834
O2 优化	2.137	1.371	0.8176

计算各项优化的加速比，我们可以得到另外一个表格如下：

表 9: 加速比计算结果（相对于串行实现）

编译优化	SIMD 加速比	CUDA 加速比
无优化	0.61	5.95
O1 优化	1.51	2.55
O2 优化	1.56	2.61

测试结果描述：从性能测试结果可以看出，CUDA 优化在所有编译优化级别下都展现出了最佳的性能表现。在无编译优化的情况下，CUDA 实现的哈希时间仅为 1.041 秒，相比串行实现的 6.197 秒获得了 5.95 倍的显著加速；随着编译优化级别的提升，CUDA 优化的绝对性能进一步改善，在 O2 优化下达到 0.8176 秒的最佳表现。相比之下，SIMD 优化在无编译优化时甚至出现了性能倒退（加速比 0.61），但在启用编译优化后表现显著改善，在 O2 优化下获得了 1.56 倍的加速比。

7.2.3 结果分析

CUDA 加速原因分析：CUDA 能够实现显著加速的根本原因在于其大规模并行计算架构。MD5 哈希计算具有天然的并行性——每个密码字符串的哈希计算相互独立，无数据依赖关系。CUDA 利用 GPU 的数千个计算核心，可以同时处理数千个密码的哈希计算，实现了真正的大规模并行处理。此外，GPU 的高内存带宽和专门优化的并行计算架构，使得批量数据处理的效率远超传统 CPU，特别适合这种计算密集型且高度并行的任务场景。

CUDA 相比 SIMD 的优势分析：CUDA 相比 SIMD 展现出更优秀加速效果的主要原因在于并行规模和架构适配性的差异。SIMD 虽然能够实现向量化并行，但其并行度有限（通常一次处理 4-8 个数据），且在复杂的 MD5 算法中，指令级并行化的实现较为复杂，容易受到分支预测、内存访问模式等因素影响。而 CUDA 可以启动数千个线程同时工作，并行规模远超 SIMD；同时 GPU 架构专门针对大规模并行计算优化，具有更高的计算吞吐量和内存带宽。此外，SIMD 优化需要复杂的代码重构和向量化实现，而 CUDA 的编程模型更适合表达这种大规模并行计算模式，因此在处理大批量独立计算任务时表现出明显的性能优势。

7.3 进一步优化的思路

混合并行架构设计 (MPI+OpenMP+CUDA 三层并行模型)

架构层次设计：采用分层递进的并行策略，在最外层使用 MPI 实现集群节点间的分布式并行，每个 MPI 进程负责整个密码搜索空间的一个子集。在单节点内部，通过 OpenMP 实现 CPU 多核并行，

将密码生成任务分配给不同的 CPU 线程。同时，每个节点配备 GPU 作为协处理器，使用 CUDA 实现大规模哈希计算的加速。这种设计充分利用了现代 HPC 集群的异构计算资源，实现了从集群级到核心级的全方位并行化。

任务分工与协调机制：在具体实现中，MPI 主进程负责全局的任务调度和结果汇总，将 PCFG 模型的不同优先级队列分发给各个计算节点。每个节点内部，OpenMP 主线程负责 PCFG 队列的管理和新 PT 生成，工作线程并行执行密码候选的生成，专门的 GPU 调度线程负责将生成的密码批量传输到 GPU 进行哈希计算。各层级之间通过高效的消息队列和内存映射机制实现数据交换，确保计算流水线的连续性。GPU 计算完成后，结果通过 MPI 进行全局汇总和去重处理。

流水线并行处理设计

多阶段流水线架构：将口令猜测系统分解为五个主要阶段：PT 队列管理、密码生成、数据预处理、GPU 哈希计算和结果处理。每个阶段作为独立的处理单元，通过高效的缓冲队列连接。PT 队列管理阶段负责维护 PCFG 优先队列并生成新的预终结符；密码生成阶段并行产生密码候选；数据预处理阶段将变长字符串转换为 GPU 友好的固定长度格式；GPU 哈希阶段进行批量并行计算；结果处理阶段进行哈希比对和输出管理。

8 多种方法的比较分析（新增）

本学期的实验，我们主要使用了 simd，pthread，openMp，Mpi，cuda 几种并行化优化技术，在不断的练习中我们加深了对于不同方法的熟悉与认识，结合资料和个人使用，我们可以等到不同并行方法的不同特性。

表 10: 并行化技术基本特性对比

特性	**SIMD**	**Pthread**	**OpenMP**	**MPI**	**CUDA**
并行类型	指令级并行	线程级并行	共享内存并行	分布式并行	数据并行
并行规模	4-16 个元素	数十线程	数十线程	数万进程	数千线程
内存模型	向量寄存器	共享内存	共享内存	分布式内存	分层内存
硬件依赖	CPU 向量指令	多核 CPU	多核 CPU	集群网络	NVIDIA GPU
编程复杂度	中等	高	低	极高	中等
标准化程度	高	高	高	高	中等
启动开销	极低	中等	低	高	高
通信开销	无	低	低	高	中等
内存带宽	CPU 带宽	CPU 带宽	CPU 带宽	网络带宽	GPU 带宽
计算吞吐量	中等	中等	中等	高	极高
可扩展性	低	中等	中等	极高	高
能效比	高	中等	中等	低	中等
负载均衡	自动	手动	自动	手动	自动

具体而言，在本学期的口令猜测系统优化实践中，我们针对不同的并行化场景实现了四种优化方案，每种方案都体现了其独特的技术特点和适用场景。

- Pthread 优化采用了多线程并行的方式，通过手动创建和管理线程来并行处理密码生成。这种实现方式提供了最大的灵活性，允许我们精确控制每个线程的工作负载和同步机制，特别适合需要复杂线程间协作的场景。在实现中，我们可以根据 CPU 核心数动态调整线程数量，并通过互斥锁和条件变量来协调线程间的数据共享，实现了细粒度的并行控制。

- OpenMP 优化则通过编译指导语句实现了更为简洁的并行化方案。利用 #pragma omp parallel

for 等指令，我们能够快速将串行的密码生成循环转换为并行执行，OpenMP 运行时自动处理线程创建、负载均衡和同步等复杂操作。这种方式的最大优势在于代码修改量小，可以在现有串行代码基础上渐进式地添加并行性，特别适合快速原型开发和性能调优。

- CUDA 优化实现了异构计算架构下的大规模数据并行处理。我们将计算密集口令生成运算加载到 GPU 上，利用数千个 CUDA 核心同时处理大批量的密码计算。这种实现需要精心设计 CPU-GPU 之间的数据传输策略，通过批量处理来摊薄传输开销，并充分发挥 GPU 的高并行度和内存带宽优势。CUDA 优化在处理大规模数据并行任务时表现出色，但需要考虑 GPU 内存管理和核函数优化等技术细节。

- MPI 优化实现了分布式内存环境下的多进程并行计算。通过将密码搜索空间分割到不同的计算节点上，每个 MPI 进程独立处理自己的任务空间，进程间通过消息传递进行协调和结果汇总。这种方式特别适合大规模集群环境，能够将计算任务扩展到数百甚至数千个计算核心，实现了真正的大规模并行密码破解。MPI 实现需要仔细设计通信模式和负载分配策略。

这四种优化方案体现了从单机多核到集群分布式、从共享内存到异构计算的不同并行计算模式，为口令猜测系统在不同硬件环境和性能需求下提供了灵活的优化选择。

9 总结

通过本学期对口令猜测系统的 SIMD、pthread、OpenMP、CUDA 和 MPI 四种并行优化实践，我深刻体会到了不同并行技术的独特魅力和适用场景。从 pthread 的灵活性到 OpenMP 的便捷性，从 CUDA 获得加速到 MPI 的大规模可扩展性，每种技术都让我对并行计算有了更深入的认识。

这一学期的实践不仅让我掌握了多种并行编程技术，更重要的是培养了并行思维和系统优化的能力。我学会了如何分析问题的并行特性，如何根据硬件环境选择合适的并行策略，以及如何在性能、复杂度和可维护性之间找到平衡点。通过对比不同优化方案的性能表现，我认识到高性能计算是一个需要综合考虑算法、硬件和应用需求的复杂工程问题。

这些实践经历不仅提升了我的编程技能，更培养了解决复杂系统问题的工程思维，为今后在并行计算和高性能计算领域的深入学习奠定了坚实基础。

本学期，所有项目保存在 github 对应仓库中，仓库地址为：

https://github.com/pandar1223333666/experiment