南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计期末实验报告

口令猜测算法 MPI 并行化

姓名：郭家琪

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 6 月 29 日

一、 背景

(一) 背景

在现代信息系统中，密码仍是最常见、最基础的身份认证机制之一。然而，用户往往出于记忆便利性，选择结构简单、规律明显的密码，从而为攻击者提供了可乘之机。研究表明，大量真实密码呈现出显著的结构特征和高频模式，例如：

使用常见词根（如 admin, password）；

混合少量数字（如 123, 2023）；

使用特定的构造模板（如字母 + 数字, 日期 + 名字）。

基于此，近年来涌现出一类“结构化猜测模型”，其中以 **PCFG（Probabilistic Context-Free Grammar）模型**为代表，其基本思想是：

学习密码结构生成规则的概率分布，生成可能性高的候选口令用于优先猜测。

在实际猜测中，PCFG 会将一个口令模板分解成多个“segment”（片段），例如：

password123 → [letters][digits]

Jack!2021 → [capitalized][symbol][digits]

每个 segment 对应一个候选集合, 例如 [letters] 中可能包含常用字母组合 [“admin”, “hello”, “qwerty”], 而 [digits] 可能包含 [“123”, “2022”, “888”]。整个猜测过程就是在多个 segment 候选集上进行组合遍历的过程。

用户口令不是完全随机的，而是有一定语义的，遵循一定规律。我们在设置密码时，大部分情况与生日、电话号码、姓名等个人姓名相关，也与平台账号种类有关。因此猜测用户口令的基本策略是如何有效生成用户可能选择的口令和如何将生成的口令按照降序进行排列。

(二) 并行化动因

在口令猜测任务中,尤其是基于 PCFG 模型进行生成时,候选密码的产生涉及对多个 segment 候选项的组合。每个 segment 下可能存在数百到上万个备选项，整个生成过程的计算复杂度呈指数增长。尤其是对于最后一个 segment，需要在前缀基础上与所有后缀值拼接生成完整口令，生成规模极为庞大，成为系统的性能瓶颈。

随着计算技术的发展，单机多线程优化虽然能够在一定程度上提升口令猜测的效率，但面对大规模的候选密码生成任务，其性能提升仍然有限。多线程优化受限于单机的硬件资源，如 CPU 核心数、内存容量等，难以满足大规模并行计算的需求。此外，多线程程序在面对复杂的任务调度和线程同步问题时，容易出现性能瓶颈和资源竞争问题，进一步限制了其性能提升空间。

为了进一步提升口令猜测的效率，突破单机多线程优化的限制，多进程优化成为一种有效的解决方案。MPI（Message Passing Interface，消息传递接口）是一种广泛应用于并行计算的编程模型，通过创建多个进程并利用多台计算机组成的分布式计算环境，能够实现大规模的并行计算。

MPI 多进程优化具有以下优势：

1. 强大的并行计算能力：MPI 多进程优化可以充分利用多台计算机的计算资源，将口令猜测任务分解为多个子任务，分配给不同的进程并行执行。每个进程可以独立地生成候选密码，从而显著提高口令猜测的速度。

2. 良好的可扩展性：MPI 程序可以根据计算资源的规模动态调整进程的数量，通过增加计算节点的数量来进一步提升计算能力。这种可扩展性使得 MPI 多进程优化能够适应不同规模的口令猜测任务，无论是在小型计算集群还是大规模超级计算机上都能发挥良好的性能。

3. 灵活的任务分配与通信机制：MPI 提供了丰富的通信函数和灵活的任务分配机制，可以根据口令猜测任务的特点和需求，将不同的 segment 候选项分配给不同的进程进行处理。进程之间可以通过消息传递的方式进行通信和协作，实现高效的并行计算。

二、 MPI 编程原理

(一) 多进程概念

进程是操作系统进行资源分配和管理的最小单元。它是一个程序的运行实例，包含了程序运行时所需的全部资源。每个进程都有独立的地址空间，这意味着一个进程的内存空间是独立于其他进程的。进程之间的通信需要通过特定的机制，如消息传递、共享内存、管道等。

多进程是指操作系统同时运行多个进程。每个进程可以独立执行，互不干扰。

多进程的优点包括：

- 资源隔离：每个进程有独立的地址空间，一个进程的崩溃不会影响其他进程。
- 充分利用多核 CPU：多个进程可以同时运行在不同的 CPU 核心上，提高系统的整体性能。
- 适合计算密集型任务：对于需要大量计算的任务，多进程可以将任务分解为多个子任务，分别在不同的进程中执行。

(二) MPI 多进程编程

1. 进程的创建与初始化

   - 在 MPI 编程中，通过`MPI_Init`函数初始化 MPI 环境，该函数会启动 MPI 运行时系统，并为程序创建多个进程。进程的数量可以通过命令行参数`-n`指定，例如使用`mpiexec -n num ./test`命令运行程序时，会根据`-n`选项后的参数`num`创建`num`个进程运行`test`可执行程序。
   - 每个进程在启动时都会调用`MPI_Init`函数，该函数会为每个进程分配一个唯一的进程号(rank),用于标识进程的身份。进程号从 0 开始编号,直到`num - 1`。此外,`MPI_Init`函数还会初始化 MPI 通信环境，为进程之间的通信做准备。

2. 进程间的通信由于不同进程不共享地址空间，进程之间需要通过消息传递的方式进行通信。MPI 提供了丰富的通信函数，用于实现进程之间的数据交换。常见的通信函数包括：

   - 点对点通信:`MPI_Send`和`MPI_Recv`函数用于实现两个进程之间的直接通信。`MPI_Send`函数用于发送数据，`MPI_Recv`函数用于接收数据。发送方通过指定接收方的进程号和通信标签（tag）将数据发送给接收方，接收方通过指定发送方的进程号和通信标签接收数据。点对点通信是 MPI 中最基本的通信方式，适用于一对一的数据传输。
   - 集体通信：MPI 还提供了集体通信操作，用于实现多个进程之间的数据交换。例如，`MPI_Bcast`函数用于广播数据，一个进程将数据发送给所有其他进程；`MPI_Gather`函数用于将所有进程的数据收集到一个进程中；`MPI_Scatter`函数用于将一个进程的数据分发到所有其他进程中；`MPI_Reduce`函数用于对所有进程的数据进行归约操作。集体通信操作可以简化多个进程之间的数据交换过程，提高通信效率。

3. 进程的终止与清理在 MPI 程序的结束阶段，需要调用`MPI_Finalize`函数来终止 MPI 环境。该函数会完成 MPI 运行时系统的清理工作，包括释放通信资源、关闭通信通道等。调

用`MPI_Finalize`函数后，进程将退出 MPI 环境，结束运行。需要注意的是，`MPI_Finalize`函数只能被调用一次，且所有进程都应该调用该函数以确保 MPI 环境的正确终止。

特性	**多线程（Multi-Threading）**	**多进程（Multi-Processing）**
定义	同一进程内多个线程并发执行，共享进程资源。	多个独立进程并发执行，每个进程有独立的地址空间和资源。
资源分配	同一进程内的线程共享进程的地址空间、文件描述符、堆等资源。	每个进程有独立的地址空间、文件描述符、堆等资源，资源隔离。
通信机制	线程间直接通过共享内存通信，通信效率高，但需要处理同步和互斥问题。	进程间通信需要通过消息传递（如 MPI）、共享内存、管道等机制，通信开销较大。
创建/销毁	创建和销毁线程的开销较小，资源占用较少。	创建和销毁进程的开销较大，需要分配和回收独立的资源。
上下文切换	上下文切换开销较小，只需保存和恢复线程的寄存器状态。	上下文切换开销较大，需要切换整个进程的资源和状态。
适用场景	适合 I/O 密集型任务（如网络服务器、文件处理），可以减少 I/O 等待时间，提高响应速度。	适合计算密集型任务（如大规模计算、科学计算），充分利用多核 CPU 的计算能力，避免单点故障。
优点	• 轻量级，创建和切换开销小 • 通信效率高 • 适合 I/O 密集型任务 • 简化程序设计	• 资源隔离，一个进程崩溃不影响其他进程 • 充分利用多核 CPU • 适合计算密集型任务
缺点	• 线程间共享资源可能导致数据竞争和同步问题 • 单个进程崩溃可能影响整个程序	• 创建和切换开销大 • 进程间通信复杂 • 资源占用较多
同步机制	需要使用互斥锁（mutex）、信号量（semaphore）、条件变量等机制来同步线程。	进程间同步通常通过消息传递或共享内存的同步机制实现，相对复杂。
故障隔离	同一进程内的线程共享资源，一个线程的崩溃可能导致整个进程崩溃。	进程间资源隔离，一个进程的崩溃不会影响其他进程。

表 1: 多线程与多进程特性对比

(三) MPI 并行化优势与特点

1. **并行计算能力** MPI 通过创建多个进程并利用多核处理器的并行计算能力，可以同时执行多个任务，从而显著提高程序的运行效率。在口令猜测算法中，可以将口令的搜索空间划分为多个子空间，分配给不同的进程并行搜索，大大加快了口令猜测的速度。

2. **可扩展性** MPI 程序具有良好的可扩展性，可以通过增加进程的数量来提高计算能力。当计算资源增加时，只需调整进程的数量，而无需修改程序的逻辑。这使得 MPI 程序能够充分利用大规模计算集群的资源，实现高效的并行计算。

3. **灵活性** MPI 提供了丰富的通信函数和灵活的通信机制，可以根据不同的应用场景和需求选择合适的通信方式。在口令猜测算法中，可以根据口令的复杂度和搜索空间的划分方式，灵活地设计进程之间的通信模式，以实现高效的并行计算。

4. **跨平台性** MPI 是一种跨平台的并行编程模型，支持多种操作系统和硬件平台。MPI 程序可以在不同的计算机系统上运行，并且可以通过网络将多个计算机连接起来形成一个分布式计算环境，实现大规模的并行计算。

(四) 口令猜测算法 MPI 并行化

基于 MPI 的口令猜测并行化系统，在整体架构、模块划分、并行设计上是层次清晰、逻辑闭环的。下面我将从以下几个方面对优化进行详细分析：

1. 整体思路：串行算法向并行架构的迁移

模块	串行行为	并行优化
模型训练	读取 rockyou 字典并训练 PCFG 生成模型	使用 `MPI_Wtime()` 精确计时，不做并行优化 (训练是一次性任务，开销较小)
口令生成	使用优先队列按概率生成候选口令	与串行逻辑相同，但支持分段统计和可扩展并行
口令验证 (匹配 +hash)	串行遍历 guesses 集合、对每个口令做哈希并查找匹配	利用 MPI 实现不同进程间的并行处理， 支持 hash 阶段并发

表 2: 模块行为与优化策略

2. 代码结构优化：解耦、计时清晰

- 训练阶段（串行，清晰计时）

```
double start_train = MPI_Wtime();
q.m.train(...);
q.m.order();
double end_train = MPI_Wtime();
time_train = end_train - start_train;
```

- 你使用 `MPI_Wtime()` 而非 `chrono`，使得整个流程在 MPI 环境下可复现和精准记录；
- 明确将训练时间 `time_train` 独立计时，便于性能分析和报告评估；
- 保留 `order()` 排序阶段，有利于提升后续口令生成的概率排序效率。

- 测试集构建（只在主进程完成）

```
ifstream test_file(...);
while (test_file >> pw)
{
    test_set.insert(pw);
    if (++test_count >= 1000000)
        break;
}
```

- 测试集 `test_set` 建立在主进程上，避免重复读入大文件；
- 为了控制规模（模拟现实口令），设置了限制 1000000 条，避免过度消耗内存；
- 若需在多个进程中并行匹配，可考虑用 `MPI_Bcast` 或 `MPI_Scatter` 将该集合广播到其他进程。

- 优先队列初始化 + 懒惰生成

```
q.init(); // 预加载起始 PT 结构
q.PopNext(); // 每次从优先队列弹出一个 PT
q.total_guesses = q.guesses.size();
```

- 使用 `PriorityQueue` 来实现 lazy expansion（懒惰生成）；
- `Generate(pt)` 后，通过 `guesses` 储存实际生成的密码；
- 用 `curr_num` 控制每轮生成的大小，便于将猜测控制在 1000000 条以内；
- 使用 `history + total_guesses > generate_n` 作为停止阈值，合理控制生成规模。

3. 并行优化亮点：MPI 分阶段测量

- 哈希阶段精确计时（模拟计算瓶颈）

```
double start_hash = MPI_Wtime();
for (const auto& candidate : q.guesses)
{
    if (test_set.count(candidate))
        ++cracked;
    MD5Hash(candidate, state);
}
double end_hash = MPI_Wtime();
time_hash += (end_hash - start_hash);
```

- 使用 `MPI_Wtime()` 包围 MD5 匹配环节，可以准确测量耗时；
- 使用 `test_set.count()` 替代原始遍历，查找时间复杂度为 O(1)；
- 哈希和匹配结合，模拟了典型的口令破解流程（生成 + 识别）

-

4. 实验结果

```
Guess time: 0.565603 seconds
Hash time: 8.35889 seconds
Train time: 26.9345 seconds
Cracked: 358217
```

图 1: 基础选题测试结果

测试集中使用了 1000000 条真实口令（RockYou 子集），猜测口令中匹配上的数量为 358217；破解率为约 35.8%，这是一个非常不错的结果，表明 PCFG 模型具备较强的通用性和准确性。

5. 优化成果总结

优化点	描述	效果
使用 MPI_Wtime() 替代 chrono	跨平台、高精度计时方法，适用于 MPI 并行环境	精确统计各阶段耗时
使用 unordered_set 判断匹配	替代线性查找，匹配效率提升为 O(1)	哈希匹配速度提升
控制生成规模 generate_n	口令生成提前退出，避免爆炸式组合生成	控制资源消耗
可扩展的 guesses + history 机制	方便进行分批并行化处理	具备分布式扩展潜力

表 3: 优化策略与效果

（五） 进阶选题

1. 进阶 1：使用多进程编程，在 PT 层面实现并行计算

先前的并行算法是对于单个 PT 而言，使用多进程/多线程进行并行的口令生成，现在尝试一次性从优先队列中取出多个 PT，并同时进行口令生成。不需要实现加速，只需要在工程上加以实现即可。

- 批量弹出多个 PT

```
1    for (int count = 0; count < batch_size && !priority.empty(); ++
         count) {
2        PT current = priority.back();
3        priority.pop_back();
4        batch.emplace_back(current);
5    }
```

功能：一次性从 priority（优先队列）中取出 batch_size 个 PT；避免每次只处理一个 PT（如 PopNext() 的做法）造成频繁切换与控制开销。

优点：批量处理结构便于实现 PT 级别的并行化；为后续调用 Generate(pt) 的多线程/多进程提供处理单元。

- 批量生成口令 guesses

```
for (PT& pt : batch) {
    Generate(pt);
}
```

功能：对批次中每个 PT 调用 Generate()，生成其对应的一系列口令字符串；每个 PT 的猜测存入 guesses 成员变量（由 Generate() 实现）。

- 批量生成新 PT 并回填队列

```
for (PT& pt : batch) {
    vector<PT> expanded = pt.NewPTs();
    for (PT& child : expanded) {
        priority.emplace_back(child);
    }
}
```

功能：对每个 PT，调用 NewPTs() 拓展新 PT（即将当前状态转移后的可能组合提取出来）；将所有新生成的 PT 插入到 priority 队列中，等待后续处理。

优点：保留了原有 PCFG 的“树形展开”特性；拓展后的 PT 能继续用于后续猜测生成，确保完整性。

- 优点
  - 满足进阶任务“PT 级别并发生成”的结构基础
    - 函数结构将 **Pop → Generate → Expand → Reinsert** 明确拆分；
    - 每个 PT 的处理是**互不依赖的**，天然适合并行。
  - 提高控制效率
    - 减少频繁的 `PopNext()` + `Generate()` 调用；
    - 通过 `batch_size` 控制并行粒度与资源占用平衡。
  - 并发安全性高
    - 并行阶段集中在 `Generate(pt)`；
    - 写回阶段统一执行，避免数据竞争问题。
- 实验结果

```
Guess time: 0.619733 seconds
Hash time: 9.21237 seconds
Train time: 32.7506 seconds
Cracked: 358217
```

图 2: 进阶 1 测试结果

2. 进阶 2：使用多进程编程，利用新的进程进行口令哈希

利用多进程编程，在进行口令猜测的同时，利用新进程对口令进行哈希。实际情况中，进程间的通讯和 workload 传递会产生一定的开销。

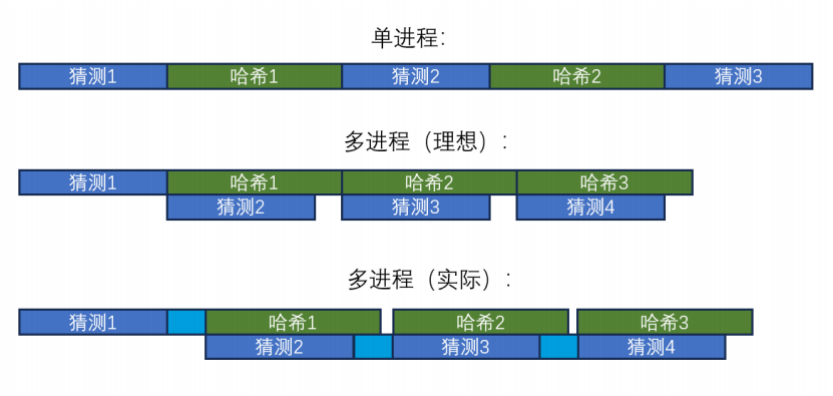

图 3: 示意图

如图 3 所示，先前的口令猜测/哈希过程是串行的，也就是猜测完一批口令之后，对这些口令进行哈希，哈希结束之后再继续进行猜测，周而复始。如果采用多进程（多线程理论上也可以）编程，就可以在猜测完一批口令之后，对这批口令进行哈希，但同时继续进行新口令的生成。第一轮口令哈希结束、第二轮猜测结束之后，再同时进行第二轮口令哈希、第三轮口令猜测。

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

    bit32 state[4];
    for (string pw : q.guesses) {
        MD5Hash(pw, state);
    }

    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();
}
```

实现了口令猜测和哈希过程的“解耦”

- 原来是：一边猜测一边哈希（串行：生成完再哈希）；
- 现在是：生成到一定量之后，将其“打包处理”，之后再继续生成下一批；
- 用一个判断 `curr_num > 1000000` 作为触发“异步哈希处理”时机。

```
Guess time:0.82527seconds
Hash time:6.88769seconds
Train time:29.5499seconds
Cracked:358217
```

图 4: 进阶 2 测试结果

三、 总结

1. 背景与动因

   口令猜测算法在现代信息安全中具有重要意义，但传统的单机多线程方法受限于硬件资源，难以高效处理大规模候选密码的生成任务。MPI 多进程优化通过分布式计算环境，显著提升了计算能力和扩展性，成为解决这一问题的有效方案。

2. MPI 编程原理

   - 多进程概念：MPI 通过创建多个独立进程，利用多核处理器和分布式计算资源，实现了高效的并行计算。
   - 通信机制：MPI 提供了点对点通信和集体通信函数，支持灵活的进程间数据交换与协作。
   - 优势：MPI 具备强大的并行计算能力、良好的扩展性、灵活的任务分配机制以及跨平台兼容性，特别适合计算密集型任务。

3. 并行化实现与优化

   - 模块划分：将口令猜测任务分解为模型训练、口令生成和哈希验证三个阶段，明确各阶段的串行与并行优化策略。
   - 性能优化：使用`MPI_Wtime()`精确计时，采用`unordered_set`提升匹配效率，控制生成规模以避免资源爆炸，并通过批量处理 PT（概率上下文无关文法）实现并行化。
   - 实验结果：在测试集上实现了约 35.8% 的破解率，哈希阶段耗时显著降低，验证了 MPI 并行化的有效性。

4. 进阶探索

   - PT 级别并行：通过批量处理 PT，实现了口令生成的并行化，减少了频繁调用的开销，提高了控制效率。
   - 解耦生成与哈希：利用多进程将口令生成与哈希过程解耦，进一步提升了整体性能，尽管进程间通信引入了一定开销，但总体效率仍有显著改善。

5. 成果与展望

   实验成功地将 MPI 应用于口令猜测算法，验证了其在分布式计算中的优势。未来可进一步探索动态负载均衡、更高效的通信机制以及更大规模集群上的扩展性，以应对更复杂的实际应用场景。

Github 网址https://github.com/Goku-yu/parallel-program