南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计实验报告

MPI 编程

许洋

年级：2023 级

专业：计算机科学与技术

指导教师：王刚

2025 年 6 月 11 日

摘要

关键字：Parallel

目录

一、 问题重述

PCFG（概率上下文无关文法）口令猜测算法是一种融合统计学习与文法规则的密码攻击技术，相比传统暴力破解和字典攻击能更高效生成密码候选列表。该算法通过分析密码数据集（如 Rockyou）提取密码结构模式（如 L8D3S3 表示 8 字母 +3 数字 +3 符号），并建立概率模型指导密码生成。其实验流程包含四个关键阶段：训练集结构分析、概率模型构建、候选密码生成以及按概率排序的密码验证。在本次 MPI 并行化实验中，重点优化后三个阶段的口令生成与验证性能。

随着密码规模增至 10^7 量级，串行实现面临显著性能瓶颈：MD5 哈希验证占据超过 70% 的计算时间；不同密码结构（PT）产生的候选密码量差异悬殊，如 L6S1 结构平均生成 52,000 个密码而 D4S2 仅 82 个；优先队列需要动态管理数千个 PT 结构的概率关系。这些特性使得 MPI 并行化成为提升效率的必然选择。

基础并行目标要求构建主从式任务分配架构。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。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。最后通过 MPI_Reduce 汇总各进程的破解结果，实现计算任务的分布式执行。

二、 实现思路

(一) main.cpp 的 MPI 实现思路

在 MPI 并行设计中，main 函数的核心思路是通过多进程协作实现高效的密码猜测生成与验证。首先，所有进程同步初始化 MPI 环境并加载相同的训练数据和测试集（Rockyou 前 100 万条），确保概率模型的一致性。随后进入主循环，各进程基于全局优先队列同步弹出预终结符 (PT)，调用 Generate 函数生成猜测密码。关键并行机制体现在： 进程 0 负责监控全局进度（每 10 万条输出）； 当累积 100 万条猜测时，各进程并行验证本地生成的密码是否命中测试集； 通过 MPI_Reduce 将各进程的破解数量汇总到进程 0 进行全局累加； 进程 0 在达到 1000 万条猜测上限时，统一输出总训练时间、猜测时间及破解密码数量。整个过程通过数据分片验证和结果归约实现高效并行，且无需进程间传输模型数据。

(二) guessing.cpp 的 MPI 实现思路

Generate 函数的核心思路是根据进程 ID 动态分配 segment 取值生成任务，实现无通信并行。对于单 segment 的预终结符 (PT)，各进程直接定位模型中的 segment 数据，通过公式 start=rank*(N/size) 和 end=(rank==size-1)?N:start+(N/size) 计算本进程处理的取值区间（N 为总取值数），在区间内生成猜测密码。对于多 segment 的 PT，所有进程先同步拼接前 N-1 个 segment 的固定前缀，再针对最后一个 segment 进行动态任务划分：各进程独立生成指定取值区间内的后缀，将其拼接到公共前缀后形成完整密码。这种设计通过末位进程处理余数（当 N 不能被 size 整除时）实现负载均衡，各进程独立写入本地猜测队列，全程无需跨进程通信，充分发挥了数据并行优势。

三、 MPI 实现

(一) main.cpp

主要功能：作为程序的入口点，负责协调整个 PCFG 密码猜测流程的并行执行，包括模型训练、猜测生成、哈希计算和结果统计。

MPI 初始化和变量声明

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
double time_hash = 0;    // 记录哈希计算总时间
double time_guess = 0;   // 记录口令生成总时间
double time_train = 0;   // 记录模型训练总时间
```

所有进程都会执行相同代码，通过 rank 区分不同角色。之后加载 rockyou 数据集训练概率上下文无关文法模型，同时对生成规则按概率降序排序，确保高概率口令优先生成。然后加载测试集，验证生成的猜测口令是否有效（是否在实际口令集中），数据结构 unordered_set 提供 O(1) 时间复杂度的查找

破解统计初始化

统计初始化

```
int cracked=0;           // 当前进程破解计数
int total_cracked = 0;   // 所有进程总破解数（仅rank 0使用）
int round_cracked = 0;   // 单轮汇总破解数
```

统计变量用于跟踪破解进度

口令生成初始化

口令生成初始化

```
q.init();  // 初始化优先级队列
if(rank == 0) cout << "here" << endl;  // 调试输出
int curr_num = 0;        // 当前批次口令计数
auto start = system_clock::now();  // 开始计时
int history = 0;         // 历史生成总数
```

初始化优先队列，并统计每批次的口令。

主循环结构

主循环

```
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
    }
}
```

批处理机制详解

批处理

```
if (curr_num > 1000000)
{
    //....
    // MPI破解数汇总
    MPI_Reduce(&cracked, &round_cracked, 1, MPI_INT, MPI_SUM, 0,
        MPI_COMM_WORLD);
    cracked = 0;  // 重置进程计数

    // Rank 0更新统计
    if(rank == 0) {
        total_cracked += round_cracked;  // 累加总破解数

        // 计算哈希时间
        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;
    }

    // 重置批次
    history += curr_num;  // 更新历史总数
```

```
20      curr_num = 0;           // 重置当前计数
21      q.guesses.clear();      // 清空口令列表
22  }
```

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。

MPI 终止

MPI 终止

```
1  MPI_Finalize();
2  return 0;
```

确保所有进程同步退出，正确释放 MPI 资源

设计中发现的问题

PCFG 模型训练

模型训练

```
1  PriorityQueue q;  // 口令生成优先级队列
2  auto start_train = system_clock::now();
3  q.m.train("/guessdata/Rockyou-singleLined-full.txt");  // 从文件训练模型
4  q.m.order();  // 按概率排序规则
5  auto end_train = system_clock::now();
6  auto duration_train = duration_cast<microseconds>(end_train - start_train);
7  time_train = double(duration_train.count()) * microseconds::period::num /
       microseconds::period::den;
```

在设计过程中模型的训练可以只在主进程完成，但是模型的广播需要进行调整，无法一次性广播给所有进程，而在我的修改中产生了冗余，在每个进程都进行了训练。

测试数据集加载

Generate 函数

```
1  unordered_set<std::string> test_set;  // 使用哈希集合存储测试口令
2  ifstream test_data("/guessdata/Rockyou-singleLined-full.txt");
3  int test_count=0;
4  string pw;
5  while(test_data>>pw) {
6      test_count+=1;
7      test_set.insert(pw);  // 插入哈希集合
8      if (test_count>=1000000) break;  // 只加载前100万个口令
9  }
```

在设计过程中加载测试集同样也可以只在主进程完成，但是广播同样会出现问题，每个进程都加载完整测试集，内存浪费（应共享）。

(二) guessing.cpp

MPI 并行化主要集中在遍历 segment 所有可能值的部分，通过任务划分实现并行加速。

MPI 初始化和变量获取

初始化

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);   // 当前进程的 ID (0 到 size-1)
MPI_Comm_size(MPI_COMM_WORLD, &size);   // MPI 进程总数
```

rank：当前进程的标识符（主进程通常为 0）

size：MPI 进程总数（总计算资源）

单段 PT 的并行处理

Generate 函数

```
// 计算任务划分
int chunk_size = pt.max_indices[0] / size;
int start = rank * chunk_size;
int end = (rank == size - 1) ? pt.max_indices[0] : (rank + 1) * chunk_size;

// 并行生成猜测
for (int i = start; i < end; i++) {
    string guess = a->ordered_values[i];
    guesses.emplace_back(guess);
    total_guesses += 1;
}
```

任务划分逻辑：将总任务量 pt.max_indices[0] 平均分给 size 个进程，每个进程处理连续的索引区间 [start, end)

负载均衡：最后一个进程处理剩余任务（当任务数不能整除时）

数据局部性：a->ordered_values 在每个进程中完整存储（广播或预加载），不需要进程间通信（无数据依赖）

结果存储：每个进程独立填充自己的 guesses 向量，total_guesses 为本地计数（需后续规约得到全局值）

多段 PT 的并行处理

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;
}
```

两阶段处理：串行部分使得所有进程生成相同的前缀字符串 guess，并行部分使得各自处理后段的不同索引范围，这样实现了前缀计算仅需一次（无并行开销），并行部分只处理变化的后段（数据并行）

零通信：不需要进程间同步（完全独立计算）

(三) 结果

最终未能实现对 guessing 的加速，Cracked 数量有 372214，与串行相比基本一致。

四、 进阶实现 1（有点问题）

(一) 实现目的

尝试使用多进程编程，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，使用多进程进行并行的口令生成，现在一次性从优先队列中取出多个 PT，并同时进行口令生成。

在实现过程中需要注意每个 PT 生成之后均需要将产生的新 PT 放回优先队列，如果一次性取出多个 PT，那么等待各 PT 生成完成后，再将一系列新的 PT 挨个放回优先队列。

(二) 代码实现

main.cpp

初始化

```
    // 初始化 MPI 环境
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // 获取当前进程的 rank
    MPI_Comm_size(MPI_COMM_WORLD, &size); // 获取总进程数
```

初始化 MPI 环境，获取当前进程编号 (rank) 和总进程数 (size)

训练

```

    if (rank == 0)  // 只有主进程执行训练
```

```
    {
        auto start_train = system_clock::now();
        q.m.train("/guessdata/Rockyou-singleLined-full.txt");
        q.m.order();
        auto end_train = system_clock::now();
        // ... 计算并输出训练时间 ...
        q.init();  // 初始化优先队列
    }
```

仅 rank 0 进程加载训练数据并初始化模型
初始化优先级队列

加载数据

```
    // 所有进程加载测试数据
    unordered_set<std::string> test_set;
    ifstream test_data("/guessdata/Rockyou-singleLined-full.txt");
    // ... 加载100万个测试密码 ...
    MPI_Barrier(MPI_COMM_WORLD);  // 同步所有进程
```

所有进程并行加载测试数据集 (用于密码验证)
使用 barrier 确保所有进程完成加载

处理多个 PT

```
    // 设置每次处理的PT数量
    int pts_per_batch = max(1, min(size/2, 2));

    while (should_continue)
    {
        if (rank == 0)  // 只有主进程管理队列
        {
            if (!q.priority.empty())
            {
                batch_size = min(pts_per_batch, (int)q.priority.size());
                q.guesses.clear();  // 清空历史猜测
                q.ProcessMultiplePTs(batch_size);  // 批量处理PT
            }
        }
```

计算每个批次的 PT 数量, 避免内存过大
主进程从队列取出多个 PT
调用 ProcessMultiplePTs 批量生成猜测

广播猜测

```
        // 广播队列状态
        MPI_Bcast(&queue_empty, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);
        if (queue_empty) break;

        // 广播猜测数量
        MPI_Bcast(&guesses_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
7
8           // 广播猜测内容
9           for (int i = 0; i < guesses_size; i++)
10          {
11              // ... 广播字符串长度 ...
12              // ... 广播字符串内容 ...
13          }
```

主进程向所有工作进程广播队列状态和猜测数量
逐个广播每个猜测密码，先广播长度，再广播字符串内容

任务分配

```
1
2           // 并行任务分配
3           int chunk_size = (guesses_size + size - 1) / size;
4           int start = rank * chunk_size;
5           int end = min(start + chunk_size, guesses_size);
6
7
8           // 各进程并行处理
9           for (int i = start; i < end; i++)
10          {
11              if (test_set.find(pw) != test_set.end()) local_cracked++;
12              MD5Hash(pw, state);  // 模拟哈希计算
13          }
```

将猜测列表均匀分块，每个进程处理自己的任务块 (start-end)
各进程独立处理分配到的猜测，验证密码是否正确 (查测试集)，计算 MD5 哈希 (模拟真实攻击)

汇总统计

```
1           // 汇总结果
2           MPI_Reduce(&local_cracked, &cracked_this_round, 1, MPI_INT, MPI_SUM,
                0, MPI_COMM_WORLD);
3
4           if (rank == 0)  // 主进程统计全局结果
5           {
6               total_cracked += cracked_this_round;
7               total_guesses += guesses_size;
8               // ... 检查终止条件(1000万猜测) ...
9           }
10          MPI_Bcast(&should_continue, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);
11      }
```

使用 Reduce 操作汇总所有进程的破解数量
主进程更新全局计数器
广播是否继续循环的决定

清理

```
    MPI_Finalize();
```

清理 MPI 环境

guessing.cpp

为了一次性取出多个 PT 我们增加了必要的函数

批量处理

```
void PriorityQueue::ProcessMultiplePTs(int batch_size)
{
    vector<PT> batch_pts;
    // 取前batch_size个PT
    for (int i = 0; i < batch_size; i++) {
        batch_pts.push_back(priority[i]);
    }
    vector<PT> all_new_pts;  // 存储所有新生成的PT

    for (PT& pt : batch_pts) {
        Generate(pt);  // 为当前PT生成猜测
        vector<PT> new_pts = pt.NewPTs();  // 生成新PT
        for (PT& new_pt : new_pts) {
            CalProb(new_pt);  // 计算新PT概率
            all_new_pts.push_back(new_pt);
        }
    }
    // 删除已处理的PT
    priority.erase(priority.begin(), priority.begin() + batch_size);
    // 对新PT按概率降序排序
    sort(all_new_pts.begin(), all_new_pts.end(),
        [](const PT& a, const PT& b) { return a.prob > b.prob; });
// 有序插入队列
    for (PT& pt : all_new_pts) {
        auto pos = upper_bound(priority.begin(), priority.end(), pt,
                               [](const PT& a, const PT& b){ return a.prob > b.
                                   prob; });
        priority.insert(pos, pt);
    }
}
```

批量获取多个 PT(指定的数量)，为每个 PT 生成猜测 (填充 guesses 列表)，为每个 PT 生成后续 PT(探索概率空间)，计算新 PT 的概率。

排序新 PT 确保概率顺序维护优先级, 通过二分查找插入位置，保持队列有序。

(三) 结果

同样未能实现对 guessing 的加速，Cracked 数量达到 396212，其中产生了一些重复计算，但是基本一致。

五、 进阶实现 2

(一) 实现目的

尝试使用多进程编程，在进行口令猜测的同时，利用新的进程进行口令哈希，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。

(二) 代码实现

main.cpp

MPI 初始化

初始化

```
int main(int argc, char **argv)
{
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    //....
}
```

MPI_Init：初始化 MPI 环境
MPI_Comm_rank：获取当前进程 ID（0 或 1）
MPI_Comm_size：获取总进程数

进程 0（口令生成进程）

批量处理

```

if (rank == 0) {
    // 1. 模型训练
    PriorityQueue q;
    auto start_train = system_clock::now();
    q.m.train("/guessdata/Rockyou-singleLined-full.txt"); // 加载训练数据
    q.m.order(); // 按概率排序
    auto end_train = system_clock::now();
    // ... 计算训练时间 ...

```

```
    // 2. 初始化优先队列
    q.init();

    // 3. 主生成循环
    int total_guesses = 0;
    const int MAX_GUESSES = 1000000;
    auto start_guess = system_clock::now();

    while (!q.priority.empty() && total_guesses < MAX_GUESSES) {
        // 4. 生成新批次口令
        q.PopNext();
        total_guesses += q.guesses.size();

        // 5. 接收上一轮哈希结果
        int cracked_count;
        MPI_Recv(&cracked_count, 1, MPI_INT, 1, 3, MPI_COMM_WORLD, &status);

        // 6. 进度报告
        if (total_guesses % 100000 == 0) {
            // ... 计算并输出统计信息 ...
        }
    }

    // 7. 发送结束信号
    int end_signal = -1;
    MPI_Send(&end_signal, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

    // 8. 最终统计输出
    // ... 计算总时间和性能指标 ...
}
```

模型训练：从 Rockyou 数据集加载训练数据，计算各 PT（Pre-Terminal）的概率分布，按概率降序排序 PT 列表。

队列初始化：初始化优先队列，包含所有 PT 及其概率信息。

生成循环：q.PopNext() 生成当前最高概率 PT 对应的口令批次，使用 MPI_Recv 等待上一批次的哈希计算结果，定期输出生成进度和破解统计。

进程 1（哈希计算进程）

批量处理

```
else if (rank == 1) {
    // 1. 加载测试数据集
    unordered_set<string> test_set;
    ifstream test_data("/guessdata/Rockyou-singleLined-full.txt");
    int test_count = 0;
    string pw;
    while (test_data >> pw && test_count < 1000000) {
        test_set.insert(pw);
```

```
        test_count++;
    }

    // 2. 主处理循环
    while (true) {
        // 3. 接收口令数量
        int guess_count;
        MPI_Recv(&guess_count, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

        // 4. 检查结束信号
        if (guess_count == -1) break;

        // 5. 计算哈希并验证
        int cracked_count = 0;
        auto start_hash = system_clock::now();

        for (int i = 0; i < guess_count; i++) {
            // 接收口令
            int len;
            MPI_Recv(&len, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
            vector<char> buffer(len + 1);
            MPI_Recv(buffer.data(), len, MPI_CHAR, 0, 2, MPI_COMM_WORLD, &
                status);
            buffer[len] = '\0';
            string guess(buffer.data());

            // 计算MD5哈希
            bit32 state[4];
            MD5Hash(guess, state);

            // 验证是否破解
            if (test_set.find(guess) != test_set.end()) {
                cracked_count++;
            }
        }

        // 6. 返回破解结果
        MPI_Send(&cracked_count, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);
    }

    // 7. 统计输出
    // ... 计算哈希总时间和破解率 ...
}
```

数据加载：创建测试集的哈希表，实现 O(1) 复杂度查询。

处理循环：接收口令批次大小，逐个接收口令内容，计算 MD5 哈希，并统计成功破解的口令数量

结果反馈：使用 MPI_Send 将破解结果返回给进程 0

guessing.cpp

批量处理

```cpp
void PriorityQueue::Generate(PT pt)
{
    CalProb(pt); // 重新计算概率
    guesses.clear(); // 清空前一批结果

    if (pt.content.size() == 1) {
        // 单segment处理
        segment *a = get_segment_ptr(pt.content[0]); // 获取segment指针
        for (int i = 0; i < pt.max_indices[0]; i++) {
            guesses.emplace_back(a->ordered_values[i]);
        }
    } else {
        // 多segment处理
        string guess;
        int seg_idx = 0;

        // 构造前N-1个segment的固定部分
        for (int idx : pt.curr_indices) {
            if (seg_idx == pt.content.size() - 1) break;
            guess += get_segment_value(pt.content[seg_idx], idx);
            seg_idx += 1;
        }

        // 添加最后一个segment的所有可能值
        segment *a = get_segment_ptr(pt.content.back());
        for (int i = 0; i < pt.max_indices.back(); i++) {
            guesses.emplace_back(guess + a->ordered_values[i]);
        }
    }

    // MPI发送批次
    int guess_count = guesses.size();
    MPI_Send(&guess_count, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);

    // 逐个发送口令
    for (const string& guess : guesses) {
        int len = guess.length();
        MPI_Send(&len, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
        MPI_Send(guess.c_str(), len, MPI_CHAR, 1, 2, MPI_COMM_WORLD);
    }
}
```

单 segment 直接遍历所有取值，多 segment 是固定前缀 + 变长后缀组合

高效通信：先发送批次大小，再逐个发送口令

长度前缀优化：接收方可精确分配内存

(三) 结果

同样未能实现对 guessing 的加速，Cracked 数量达到 303926，其中有小幅度的减少，但是基本一致。

六、 总结

实现了基础和进阶的实验，在基础实验中，主从模式下，主进程广播 PT 结构，工作进程按块划分验证密码，通过 chunk_size=(N+size-1)/size 实现负载均衡。PT 级批量并行实验中每次处理多个高概率 PT，新 PT 按概率排序插入队列，减少队列操作开销。但是实现有问题，需要进一步进行修改。流水线并行实验中，分离密码生成与哈希验证，Rank 0 生成批次、Rank 1 专职哈希，通过 MPI_Send/Recv 实现异步流水线。

后续需要进行改进，在修改的过程中遇到了模型以及训练数据的广播问题，模型训练和数据需要去冗余，主进程训练后广播模型参数。