南開大學
Nankai University

计算机学院

并行程序设计实验报告

PCFG 口令猜测并行化选题: MPI 多进程实验

姓名：陈语童

学号：2311887

专业：计算机科学与技术

2025 年 6 月 27 日

目录

1 实验环境 **2**

2 问题描述 **2**

2.1 选题：PCFG 口令猜测 2

2.1.1 口令猜测 2

2.1.2 PCFG 串行训练 2

2.1.3 PCFG 串行猜测生成 3

2.1.4 PCFG 猜测生成**并行化 *** 3

2.2 本次实验：MPI 多进程实验 4

2.2.1 MPI 4

2.2.2 PCFG 猜测循环的并行化 6

3 基础实验 **8**

3.1 并行度选择 8

3.2 MPI 的口令猜测并行化实现 8

3.2.1 并行化代码实现 8

3.3 正确性检验 12

3.4 加速效果测试 12

4 进阶实验 **13**

4.1 PT 层面的并行化 13

4.2 MPI+OpenMP 优化猜测生成 16

5 总结思考 **17**

1 实验环境

SSH 远程：OpenEuler ARM 服务器

课程提供的 OpenEuler 服务器集群，以物理机 + 虚拟机的形式搭建的，目前提供给并行课程使用的共有 1 个主节点 (master_ubss1) 和 17 个计算节点 (master_ubss1 9, node1_ubss1 8)，每个计算节点 8 核。

OpenEuler 是开源、免费的 Linux 发行版操作系统，使用 ARM 架构指令集。

通过 vscode （或 cmd/powershell）进行相关 SSH 配置并进行连接，登入服务器中为学生个人分配的节点，进行 ARM 架构上的并行代码测试。

2 问题描述

2.1 选题：PCFG 口令猜测

2.1.1 口令猜测

口令，就是俗称的“密码”。对于一定长度范围的口令，通过穷举进行暴力破解的效率是极低的。而本选题要做的，就是通过用户口令的语义和偏好规律，制定一定的策略来破解口令，并用**并行化**的方法对整个过程的各个部分进行优化与加速。

本选题中，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。那么问题就变成了：

- 如何有效生成用户可能选择的口令；
- 如何将生成的口令按照降序进行排列。

在本选题中，选择使用最经典的 **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。

2.1.2 PCFG 串行训练

PCFG 串行训练流程关键步骤如下：

1. **内容分类**。将口令内容分为三类不同字段（segments）——Letters（字母字段）、Digits（数字字段）、Symbols（特殊字符）
2. **解构口令**。然后将实际每条口令内容用这三种字段类型解构其组成，形成一系列的口令 preterminal（类似于 pattern）。
3. **统计频率**。对不同的 preterminal，以及每个 preterminal 中的每个 segment 的不同值均需统计出现频次。

经过上面三步，PCFG 模型生成。

2.1.3 PCFG 串行猜测生成

猜测生成的本质要求，是构建出一个出现概率降序排列的口令**优先队列**，对其动态更新和从中取猜测值。具体生成策略较为繁琐，此处只做概括：

1. **初始化**。所有 preterminal 中出现概率最高的口令组合先入队列，preterminal 本身的概率越高越优先入队列。同时，赋给一个初始 pivot 值为 0.
2. **出列**。只要队列有口令，就从先入队列的开始取，取到的即作为当前口令猜测值。
3. **变异**。取出的口令不立即放回，而是根据 pivot 值，依次单项更新大于 pivot 值的 segment 值，依然是概率降序入队列，并且新入列的要设置 pivot 值到发生变异的 segment.
4. **遍历**。接着向后遍历和动态更新即可。

通过这种策略生成的优先队列可尽可能保证口令是不重复地以概率降序排列的，从而提高遍历查找效率和命中率。

2.1.4 PCFG 猜测生成并行化 *

这是本选题的重点和最终目标。

PCFG 猜测生成过程似乎比较复杂，难以进行并行化。并行化的最大阻碍，是按照概率降序生成口令这一过程。但 PCFG 往往用于没有猜测次数限制的场景中（例如哈希破解），实际应用中并不需要严格按照降序生成口令。

由此产生的并行化思路很显然：一次取出多个 preterminal，或是一次为某一 segment 分配多个 value 等。

选题代码框架采用的并行化方案如图示：

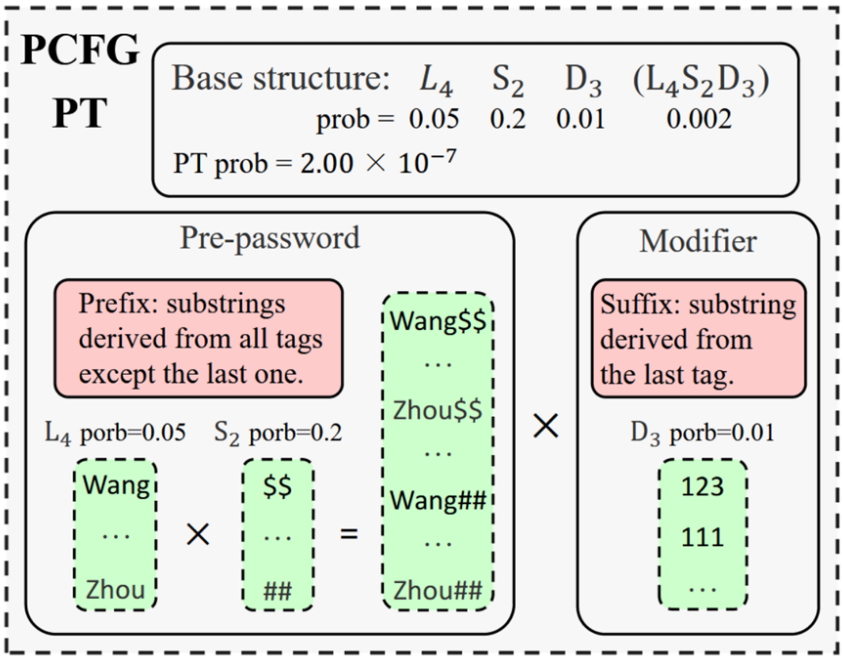


图 2.1: 本代码框架采用的并行化方案

这种方案和原始 PCFG 的区别在于，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. 不难看出，这个过程是可以并行进行的。

本选题的代码框架，已经将这个并行算法用串行的方式进行了实现，实际实验只需将其用并行编程的方式，变成一个并行程序，即可完成 PCFG 的并行化。

2.2 本次实验：MPI 多进程实验

本次实验使用 MPI 编程标准以及各种专用 API，对于 PCFG 的猜测生成过程进行并行化。

2.2.1 MPI

全称： Message Passing Interface，消息传递接口

平台： MPI 是标准，不依赖具体操作系统或硬件

语言： C/C++/Fortran 全面支持，其他部分语言可第三方调用

特性：

- **进程级并行：**每个进程有独立的内存空间，进程之间通过消息传递通信;
- **可扩展性强：**支持成千上万的进程同时运行，适用于大规模计算集群;
- **跨平台：** MPI 是标准，不依赖具体操作系统或硬件;
- **通信方式多样：**点对点通信（Send/Recv），广播（Broadcast），整体通信（Reduce、Scatter、Gather 等）。

MPI 是一种用于 **分布式内存并行计算**的 **编程模型**和 **通信协议**，广泛应用于高性能计算领域。它允许在多个**进程**之间进行高效的数据交换，适合多台计算节点（或者多核处理器）协同完成大规模计算任务。

值得注意的是，MPI 实现的是多**进程**编程，这与上次实验中用 pthread/OpenMP 实现的多**线程**编程时完全不同的两个概念。进程/线程之间的区别，可以体现在以下表中：

项目	进程	线程
定义	程序运行中的独立单位	单个进程内最小执行单位
地址空间	独立	同进程内线程共享地址空间
内存资源	各自有独立堆、栈和数据段	共享堆和数据段，但有各自的栈
通信方式	进程间通信，管道/消息队列/共享内存	线程间可直接读写共享内存
创建开销	大	小
执行效率	低	高

表 1: 进程和线程对比

所以进程和线程其实类似于包含关系，**一个进程包含多个线程，而一个线程仅属于一个进程**。进程相当于封闭容器，互不相干；线程相当于容器中的若干处理器，共享当前进程的资源。实际部署中，可以尝试使用多进程 + 多线程的融合实现，特定情形下可获得比单独实现多进程或单独实现多线程有更高的效率提升。

基础 API:

```
//预定义的数据类型
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_UNSIGNED_CHAR
MPI_FLOAT
MPI_DOUBLE

//函数接口
MPI_Comm_size //报告进程数
int MPI_Comm_size(MPI_Comm comm, int *size)

MPI_Comm_rank //报告识别调用进程的 rank，值从 0 size-1
int MPI_Comm_rank(MPI_Comm comm, int *rank);

MPI_Init //令 MPI 进行必要的初始化工作
int MPI_Init(
    int* argc_p /* 输入/输出参数 */,
    char *** argv_p /* 输入/输出参数 */);

MPI_Finalize //告诉 MPI 程序已结束，进行清理工作
int MPI_Finalize(void);

MPI_Send //基本（阻塞）发，向一个进程发送数据
int MPI_Send(
    void* buf /* 存储数据的缓冲区地址 */,
    int count /* 发送的数据量 */,
    MPI_Datatype datatype /* 数据类型 */,
    int dest /* 目的进程编号 */,
    int tag /* 标识向同一个目的进程发送的不同数据 */,
    MPI_Comm /* MPI 集群的通信域标识 */);

MPI_Recv //基本（阻塞）接收，从一个进程接收数据
int MPI_Recv(
    void* buf /* 存储数据的缓冲区地址 */,
    int count /* 接收的数据量 */,
    MPI_Datatype datatype /* 数据类型 */,
    int source /* 源头进程编号 */,
    int tag /* 标识从同一个源头进程接收的不同数据 */,
    MPI_Comm /* MPI 集群的通信域标识 */,
    MPI_Status *status /* 可以记录更多额外的信息 */)
```

简单示例:

```
#include <mpi.h>
#include <stdio.h>
```

```
int main(int argc, char** argv) {
    // 初始化 MPI 环境
    MPI_Init(&argc, &argv);

    int world_size; // 总进程数
    int world_rank; // 当前进程号

    // 获取总进程数
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // 获取当前进程号
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    if (world_rank == 0) {
        // 进程 0 发送消息
        const char* message = "Hello from Process 0!";
        for (int i = 1; i < world_size; i++) {
            MPI_Send(message, 25, MPI_CHAR, i, 0, MPI_COMM_WORLD);
        }
        printf("Process 0 sent messages to all processes.\n");
    } else {
        // 其他进程接收消息
        char recv_buffer[50];
        MPI_Recv(recv_buffer, 50, MPI_CHAR, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        printf("Process %d received message: %s\n", world_rank, recv_buffer);
    }

    // 结束 MPI 环境
    MPI_Finalize();
    return 0;
}
```

值得注意的是，程序并非只在 `MPI_Init()` 和 `MPI_Finalize()` 这两个函数调用之间的部分并行地执行，而是整体地并行执行，在两个函数之间的部分完成数据划分运算、通信等工作。

一种可能的输出：

```
Process 0 sent messages to all processes.
Process 1 received message: Hello from Process 0!
Process 2 received message: Hello from Process 0!
Process 3 received message: Hello from Process 0!
```

2.2.2 PCFG 猜测循环的并行化

PCFG 算法中生成猜测部分的串行算法通过循环来实现：

```
// ... ... ... ...
// Multi-thread TODO:
// 这个 for 循环就是你需要进行并行化的主要部分了，特别是在多线程 &GPU 编程任务中
// 可以看到，这个循环本质上就是把模型中一个 segment 的所有 value，赋值到 PT 中，形成一系列新的猜测
// 这个过程是可以高度并行化的
```

```
for (int i = 0; i < pt.max_indices[0]; i += 1)
{
    string guess = a->ordered_values[i];
    guesses.emplace_back(guess);
    total_guesses += 1;
}
// ... ... ... ...
// Multi-thread TODO:
// 这个 for 循环就是你需要进行并行化的主要部分了，特别是在多线程 &GPU 编程任务中
// 可以看到，这个循环本质上就是把模型中一个 segment 的所有 value，赋值到 PT 中，形成一系列新的猜测
// 这个过程是可以高度并行化的
for (int i = 0; i < pt.max_indices[pt.content.size() - 1]; i += 1)
{
    string temp = guess + a->ordered_values[i];
    guesses.emplace_back(temp);
    total_guesses += 1;
}
// ... ... ... ...
```

本此实验的主要目标是，通过 MPI 多进程 将以上两个循环并行优化（与上次 pthread/OpenMP 进行的优化本质是一样的）。

此外，进阶要求涉及到对于 PT 的获取和存储过程也尝试进行多进程并行化。观察到这个函数：

```
void PriorityQueue::PopNext() {
    // 对优先队列最前面的 PT，首先利用这个 PT 生成一系列猜测
    // <=== 串行方法 ===>
    Generate(priority.front());
    // <= pthread 方法 =>
    // PthreadGenerate(priority.front());
    // PthreadPoolGenerate(priority.front());
    // <== openmp 方法 =>
    // OpenMPGenerate(priority.front());

    // 然后需要根据即将出队的 PT，生成一系列新的 PT
    vector<PT> new_pts = priority.front().NewPTs();
    for (PT pt : new_pts)
    {
        // 计算概率
        CalProb(pt);
        // 接下来的这个循环，作用是根据概率，将新的 PT 插入到优先队列中
        for (auto iter = priority.begin(); iter != priority.end(); iter++)
        {
            // 对于非队首和队尾的特殊情况
            if (iter != priority.end() - 1 && iter != priority.begin())
            {
                // 判定概率
                if (pt.prob <= iter->prob && pt.prob > (iter + 1)->prob)
                {
                    priority.emplace(iter + 1, pt);
```

```
                    break;
                }
            }
            if (iter == priority.end() - 1)
            {
                priority.emplace_back(pt);
                break;
            }
            if (iter == priority.begin() && iter->prob < pt.prob)
            {
                priority.emplace(iter, pt);
                break;
            }
        }
    }

    // 现在队首的 PT 善后工作已经结束，将其出队（删除）
    priority.erase(priority.begin());
}
```

它的作用是从优先队列中取出一个 PT，基于它生成一些列的猜测，再将所有猜测加入优先队列。进阶要求尝试实现的是，一次性取多个 PT，分配给不同的 MPI 线程，分别用分配到的 PT 生成猜测，最后汇总再入列。

3 基础实验

基础要求： 将猜测生成过程用 MPI 进行多线程并行化。具体要求如下：

- 保证并行化后的猜测过程“相对正确”。口令猜测并行化会在一定程度上牺牲口令概率的“颗粒度”，但总体上口令的生成结果基本不会有差别。检查正确性的方法是：在主函数中新增给定代码检查口令破解数，需要保证并行算法破解数量在数量级上和串行算法保持一致。
- 不要求真正实现相对串行算法的加速，但是仍需要给出实验数据，并且分析为什么没有能够实现加速。

3.1 并行度选择

基础实验下未使用多线程优化，根据指导文档，默认使用的是 $nodes = 1 : ppn = 8$ 的脚本参数。

3.2 MPI 的口令猜测并行化实现

3.2.1 并行化代码实现

在 [PCFG.h] 中添加相应的优先队列成员函数 `MPIGenerate` 后，直接在 [guessing.cpp] 中进行实现。

首先初始化，获得当前进程的进程号 `rank` 以及总进程数 `size`. 使用 MPI 提供的接口获得：

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

提取 PT 待变异字段可取值列表的过程不变，接下来就是使用 MPI 并行化循环部分。

考虑到负载均衡的问题，采取**动态任务划分**的策略。首先将整体任务用总进程数进行整除，获得每个进程处理的任务量基数 `base_chunk`；整除后的余数不足进程数，作为 `remain_pack` 剩余量参数，那么前 `remain_pack` 就需要额外分配一个任务。这样可以尽可能保证进程之间所用时间不会相差太大，提高整体效率。动态任务划分过程的代码实现如下：

```
// 动态划分任务
int base_chunk = total_work / size;
int remain_pack = total_work % size;
int start, end;
// 设置各进程开始和结束位置
if (rank < remain_pack) {
    start = rank * (base_chunk + 1);
    end = start + base_chunk + 1;
}
else {
    start = remain_pack * (base_chunk + 1) + (rank - remain_pack) * base_chunk;
    end = start + base_chunk;
}
```

猜测的生成核心逻辑，由于未采用多线程进行优化，与串行方法保持一致即可：

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

以上的猜测生成过程是各个进程独立进行的，最终需要从局部提取汇总部分信息。使用 `MPI_Allreduce` 这一同步全局归约函数接口，统计所有进程得到总猜测数量 `global_count`，保证全局信息一致：

```
// 独立进程新增猜测数
int local_count = end - start;
// 总增量
int global_count = 0;
// 汇总猜测总数
MPI_Allreduce(&local_count, &global_count, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
total_guesses = global_count;
```

以上是以只有一个字段的 PT 为例的，多字段 PT 的猜测生成过程基本逻辑是一致的，不再详细推导。其代码如下：

```
else {
// ... ... ... ...
    // 总任务量
    int total_work = pt.max_indices[pt.content.size() - 1];
    // 动态划分任务
    int base_chunk = total_work / size;
    int remain_pack = total_work % size;
    int start, end;
```

```
    // 前 remain_pack 个进程多分配一个任务
    if (rank < remain_pack) {
        start = rank * (base_chunk + 1);
        end = start + base_chunk + 1;
    } else {
        start = remain_pack * (base_chunk + 1) + (rank - remain_pack) * base_chunk;
        end = start + base_chunk;
    }
    // 生成猜测
    for (int i = start; i < end; ++i) {
        guesses.push_back(prefix + a->ordered_values[i]);
    }
    // 进程内局部
    int local_count = end - start;
    // 全局
    int global_count = 0;
    // 汇总猜测总数
    MPI_Allreduce(&local_count, &global_count, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
    total_guesses = global_count;
}
// ... ... ... ...
```

MPI 测试时，无法使用原先代码架构的 `chrono` 时间戳，而要使用 `MPI_Wtime` 等专门针对 MPI 获取时间戳的函数来测运行时间。为此，在 [main.cpp] 重新编写专门适配于 MPI 方法的主函数，使用了 MPI 的计时方案、MPI 的资源调度方法，顺便使用 MPI 多进程优化了 MD5 过程：

```
// ... ... ... ... (#include)
int main(int argc, char *argv[])
{
    // MPI 多进程初始化
    MPI_Init(&argc, &argv);
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
// ... ... ... ...
    // MPI 计时：训练阶段
    double mpi_time_train_start = MPI_Wtime();
    if (rank == 0) {
        cout << "Starting model training..." << endl;
    }
// ... ... ... ...
    // MPI 计时：训练结束
    double mpi_time_train_end = MPI_Wtime();
    // MPI 计时：直接计算时间
// ... ... ... ...
    // MPI 阻塞：等待所有进程完成训练
    MPI_Barrier(MPI_COMM_WORLD);
// ... ... ... ...
    // MPI 计时：猜测生成阶段
```

```
24      double mpi_time_guess_start = MPI_Wtime();
25  // ... ... ... ...
26      while (!q.priority.empty())
27      {
28          q.PopNext();    // 调用 MPIGenerate 方法
29          // 收集所有进程的猜测总数
30          int local_guesses = q.guesses.size();
31          int global_guesses = 0;
32          MPI_Allreduce(&local_guesses, &global_guesses, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
33          q.total_guesses = global_guesses;
34  // ... ... ... ... (判断猜测数量上限以及输出运行时间)
35          // 到一定阈值进行哈希
36          if (curr_num > 1000000)
37          {
38              // MPI 计时: 哈希阶段
39              double start_hash = MPI_Wtime();
40              // 和 MPI 处理猜测生成一样类似的流程
41              const int batchSize = 4;
42              const int numFullBatches = q.guesses.size() / batchSize;
43              const int remainder = q.guesses.size() % batchSize;
44              // 每个哈希 4 个 bit32, 连续内存
45              bit32 state[batchSize * 4];
46              // 预分配字符串数组
47              string inputs[batchSize];
48              // 处理完整批次
49              for (int batch = 0; batch < numFullBatches; ++batch) {
50                  for (int i = 0; i < batchSize; ++i) {
51                      inputs[i] = q.guesses[batch * batchSize + i];
52                  }
53                  SIMDMD5Hash_4(inputs, state);
54              }
55              // 处理剩余项 (不足 4 个)
56              if (remainder > 0) {
57                  for (int i = 0; i < remainder; ++i) {
58                      inputs[i] = q.guesses[numFullBatches * batchSize + i];
59                  }
60                  // 剩余的用单个哈希函数处理
61                  for (int i = 0; i < remainder; ++i) {
62                      bit32 singleState[4];
63                      MD5Hash(inputs[i], singleState);
64                  }
65              }
66              // MPI 计时: 哈希结束
67              double end_hash = MPI_Wtime();
68              time_hash += end_hash - start_hash;
69  // ... ... ... ...
70          }
71  // ... ... ... ...
72      }
```

```
    // 终结
    MPI_Finalize();
    return 0
}
```

MPI 方法实现完整代码见 GitHub 仓库：[guessing.cpp]

main 函数的完整修改见 GitHub 仓库：[main.cpp]

3.3 正确性检验

测试正确性的代码和原理与上次实验是一致的，此处不再一一赘述。多次提交测试，结果稳定如下所示：

```
Guess time: 0.51328 seconds
Hash time: 0.222298seconds
Train time: 28.9123 seconds
Cracked: 95274
```

图 3.2: MPI 算法破解结果

串行算法：Cracked = 95274.

参考 上次实验（Lab3-pthread&OpenMP）报告 ，串行算法的 Cracked = 358217.

发现相比于串行，MPI 的破解效果大幅降低。但 95k+ 的 Cracked 量可以近似作 100k，仍可勉强满足同一数量级的要求。

推测破解效果大幅降低的可能性是多进程将猜测结果插入优先队列时与“按概率递减”的要求出入较大，或者其分配任务时有不可控的重叠、疏漏等错误，导致最终破解数量大幅降低。

此外还发现哈希的时间也大幅降低了，这是由于上面主函数中使用 MPI 对其加了速。

3.4 加速效果测试

在 $nodes = 1 : ppn = 8$ 的脚本参数下进行多次实验，测试有 MPI 的加速效果。口令猜测生成用时结果如下表所示（多组实验，剔除最高和最低；由于服务器不太稳定，串行方法也重新测了，为了做到严谨的对照）：

no.	1	2	3	4	5	**avg.**
serial	0.59663	0.613926	0.573981	0.603269	0.601928	**0.59794**
MPI	0.471947	0.493812	0.484105	0.502111	0.477537	**0.48590**

表 2: 串行和 MPI 优化下的猜测生成用时（单位：秒）

根据以上数据依次求得 MPI 在猜测生成上的**加速比**：

$$\text{Speedup(MPI)} = \frac{T_{avg}(\text{serial})}{T_{avg}(\text{MPI})} = \frac{0.59794}{0.48590} \approx 1.23 > 1$$

可见 MPI 对于口令猜测生成过程是有明显的加速效果的。参考 上次实验(Lab3-pthread&OpenMP)报告 中 pthread 和 OpenMP 的加速效果：

$$\text{Speedup(pthread(+))} = 1.30$$

$$\text{Speedup(OpenMP)} = 1.33$$

对比而言，理论上在等价的参数条件下（对于 pthread 和 OpenMP，$THREAD\_NUM = 8$），pthread 和 OpenMP 这两种线程并行化方案加速效果更加明显。这印证了表 1中对于进程/线程差异的分析，即进程操作相较于线程操作**开销较大**和**效率较低**的特点。

数据真实性证明见 GitHub 仓库：[speedup min data.md]

4 进阶实验

4.1 PT 层面的并行化

尝试猜测过程中在 PT 层面进行并行化。基本思路是，将现在每次从优先队列取出 1 个 PT 进行口令猜测生成的模式，改变为每次取多个 PT 进行口令猜测生成，将这多个 PT 分配给**不同线程**并行处理，**全部完成后**将新生成的猜测依次返回入列。

下面来依次实现，原 `PopNext` 函数的 MPI 并行版本 `MPIPopNext`.

首先初始化、基本参数设置并取出一定量的 PT：

```
// 初始化
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

// 每次批量划分 PT, batch_size 可调整
int batch_size = 4; // 一批处理 4 个 PT, 可调整
int actual_batch = min(batch_size, (int)priority.size());

// 提取前 actual_batch 个 PT
vector<PT> batch_pt(priority.begin(), priority.begin() + actual_batch);
```

接着动态划分 PT 任务给不同进程：

```
// 动态划分 PT 任务给各进程
int base_chunk = actual_batch / size;
int remain_pack = actual_batch % size;

int start, end;

if (rank < remain_pack) {
    start = rank * (base_chunk + 1);
    end = start + base_chunk + 1;
} else {
    start = remain_pack * (base_chunk + 1) + (rank - remain_pack) * base_chunk;
    end = start + base_chunk;
}
```

每个进程内，PT 生成猜测的过程使用串行方法即可：

```
// 每个进程串行处理自己负责的 PT
for (int i = start; i < end; ++i) {
    Generate(batch_pt[i]);
}
```

各进程进行完毕后，使用 MPI_Allreduce 接口跨进程同步，合并各进程中的局部量：

```
// 汇总每个进程的猜测总数
int local_count = total_guesses;
int global_count = 0;
MPI_Allreduce(&local_count, &global_count, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
total_guesses = global_count;
```

最后更新优先队列，删除此次取出进行猜测的 PT，并入列新生成的：

```
// 队列中删除已处理的 PT
priority.erase(priority.begin(), priority.begin() + actual_batch);

// 生成下一批的 PT 并插入队列
vector<PT> new_pts;
for (PT &pt : batch_pt) {
    vector<PT> generated_pts = pt.NewPTs();
    for (PT& gen_pt : generated_pts) {
        CalProb(pt);
        generated_pts.push_back(gen_pt);
    }
}

// 插入新生成的 PT
for (PT& pt : new_pts) {
    bool inserted = false;
    for (auto iter = priority.begin(); iter != priority.end(); iter++) {
        if (pt.prob > iter->prob) {
            priority.emplace(iter, pt);
            inserted = true;
            break;
        }
    }
    if (!inserted) {
        priority.emplace_back(pt);
    }
}
```

在主函数中调用 MPIPopNext 的时候，也需要预先修改一些逻辑。具体如下所示：

```
// ... ... ... ...
// 修改判断 Pop 结束的逻辑
bool local_not_empty = !q.priority.empty();
```

```
int global_not_empty = 0;
MPI_Allreduce(&local_not_empty, &global_not_empty, 1, MPI_INT, MPI_LOR, MPI_COMM_WORLD);
while (local_not_empty) {
    MPIPopNext();   // 调用 PT 层面的 MPI 多进程方法
    // ... ... ... ...
    // 更新判断
    local_not_empty = !q.priority.empty();
    MPI_Allreduce(&local_not_empty, &global_not_empty, 1, MPI_INT, MPI_LOR, MPI_COMM_WORLD);
}
// ... ... ... ...
```

MPI 方法实现 PT 并行化的完整代码见 GitHub 仓库：[guessing.cpp]

实现以上方法以后进行测试，发现并不能达到加速效果，运行速度相对较快的一次结果如下图示：

```
Guess time: 1.24791 seconds
Hash time: 0.368751seconds
Train time: 29.3681 seconds
```

此外，还有一个发现，是关于改变 `MPIPopNext` 中单次处理 PT 的数量对于 `Cracked` 参数的影响。具体统计如下表所示：

PT num.	1	2	3	4	5	6	7	8
Cracked	353278	167275	160299	107466	88429	83230	77292	57693

表 3: 不同单次处理 PT 数目下的 Cracked 参数值

这也提供了一些启发，可以尝试对于之前实现的 MPI 多进程并行化猜测过程调整进程数（本质上修改 `-np` 参数即可，没有线程并行的情况下这就代表实际的进程数），发现 `Cracked` 参数也会发生变化：

Pro. num.	1	2	3	4	5	6	7	8
Cracked	358217	220177	181062	156111	131717	115821	104498	95274

表 4: 不同进程数目下的 Cracked 参数值

根据以上测试得到的两组数据，绘制下面的综合折线图：

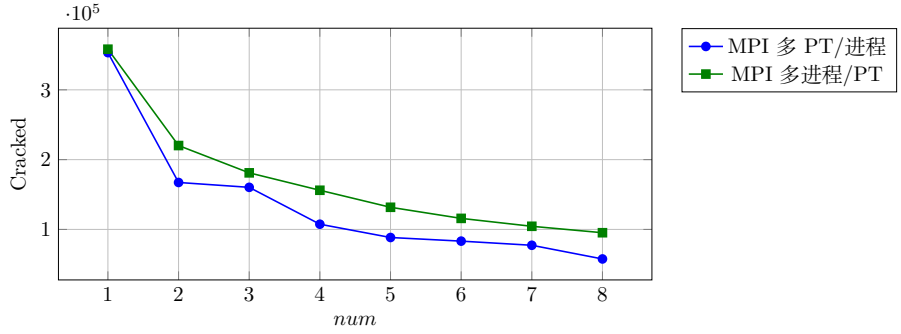


图 4.3: Cracked 值随 PT 数目或进程数目的变化趋势

接下来尝试分析由上面实验产生的一系列问题：

Q：为什么 Cracked 数随着线程数/PT 数增加而减少？

`Cracked` 表示总猜中的真实口令数。在原本纯串行的方法下，严格按照概率从高到低的顺序来处理，优先生成的猜测是猜中概率相对高的；而不论是 MPI 多线程处理同一 PT 的口令猜测生成任务，亦或是 MPI 多线程分配多个 PT 的口令猜测生成任务同步处理，由于各个线程的速度差异，可能导致大量的、被划分到**低概率**猜测生成的线程先完成任务并添加猜测到优先队列中，这样严重**降低了高概率猜测先进行验证**的优先度，从而导致 `Cracked` 参数降低。这是在原理层面上分析最主要的原因。

Q：为什么同一 PT 生成分配给不同进程的方法的 Cracked 数始终比多 PT 任务分配给不同进程的方法的 Cracked 数多？

同一 PT 的猜测生成分配给不同进程处理，至少保证了原始 PT 列表中的概率递减性的延续；而多个 PT 分配给不同进程处理时，有可能低概率 PT 的进程先处理完成并加入猜测到优先列表中，那么就导致了原始 PT 列表的概率地健星也被破坏，进一步牺牲“颗粒度”，能够正确猜中的可能性进一步降低。所以会产生两种方法始终有 `Cracked` 上的大小差异。

Q：为什么当同时处理的 PT 数为 1 时 MPI 多进程也无法到达原串行方法的 Cracked 数量？

这一问题尚未能够分析出合理解释.....

4.2 MPI+OpenMP 优化猜测生成

受到实验指导启发，尝试同时使用 MPI 多进程和 OpenMP 多线程对猜测生成过程进行加速，完整实现“多进程内多线程”的并行体系。

相对于普通的 MPI 方法 `MPIGenerate`，OpenMP 的加入只需要将串行实现的核心猜测过程修改成 OpenMP 方法即可：

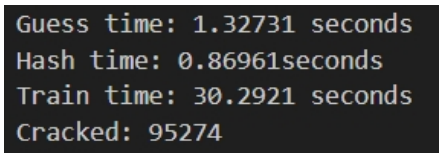
```
// ... ... ... ...
// OpenMP 并行部分
#pragma omp parallel
{
    std::vector<std::string> thread_guesses;

    #pragma omp for nowait schedule(static)
    for (int i = start; i < end; i++) {
        thread_guesses.push_back(a->ordered_values[i]);
    }

    #pragma omp critical
    {
        guesses.insert(guesses.end(), thread_guesses.begin(), thread_guesses.end());
    }
}
// ... ... ... ...
```

对于多个字段的情况，处理类似，代码略。

实现后，发现也不能达到加速效果，运行较快的一次结果如下图示（$nodes = 2 : ppn = 8, -np\ 8$）：



同时上图也展现了 `Cracked` 数目，和仅使用 MPI 多进程并行化猜测生成时的 `Cracked` 数目是一致的。

至于为什么添加 OpenMP 多线程无法实现加速，关键在于使用多进程划分后的任务已经足够轻量，如果再加入多线程，额外产生的线程启动、销毁以及同步等开销过大，对于数据本身处理的加速效果被掩盖。相对于仅进行 OpenMP 多线程实现的方法，这一实现下 OpenMP 多线程被多个进程包装，需要额外的进程同步；由于进程是多个的，原本 OpenMP 的单一同步需求也翻了若干倍。此外，也可能存在硬件方面的条件不足，不适合采用 MPI 多进程和 OpenMP 多线程联合优化的方法。

5 总结思考

多进程和多线程本质不同，多进程最大的一个特点是各个子任务执行时不能直接共享资源和数据，因而适合更加顶层层面的并行实现。本实验尝试用 MPI 多进程并行化 PCFG 的 PT 猜测生成过程，实际加速效果远不及 pthread/OpenMP，也不够稳定。这是符合多进程相对于多线程的各项差异的，说明这样的任务更适合通过多线程来实现。

本次实验所有代码均已上传至 GitHub：{OpenEuler 服务器中的 NEON 实现}