
南開大學
Nankai University

计算机学院

并行程序设计期末报告

# MPI 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 6 月 29 日

# 目录

# 1 问题描述

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。在这个选题中，使用最经典的 PCFG（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。口令猜测选题的框架主要分为三个部分：模型训练、口令生成、MD5 哈希值生成。在本篇中，我们将着重讨论口令生成部分的 MPI 的并行实现，以提高口令猜测的时间。

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。一个基本的思路是：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。此时将会把 D3 所有可能的 value 都赋予这个 preterminal，以一次性生成多个口令。不难看出，在这个过程中，存在两个循环是主要的瓶颈，我们可以利用 MPI 实现多进程的编程，从而实现并行化。

# 2 方法介绍

## 2.1 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。下面是一些相关的概念简介：

- **进程（Process）**：操作系统分配资源（内存、文件句柄、CPU 时间片等）的基本单位，拥有独立的虚拟地址空间、独立的堆栈和全局变量空间。
- **多进程**：在同一台机器或同一系统上同时运行多个进程实例，它们相互隔离、独立调度。不同进程间不能直接共享内存，需要借助操作系统提供的进程间通信（IPC）机制。
- **并发 vs 并行**：在单核 CPU 上，多进程通过操作系统的时间片轮转交替执行，表现为并发；在多核 CPU 上，不同进程可以被映射到不同核心并行运行，实现真正的并行。

## 2.2 mpi 编程

在本次实验中，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。

# 3 算法实现

## 3.1 基本思路

口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。主要的实现思路如下：

## 3.2 详细实现

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。

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

    /* ==== 获取 MPI 基本信息 ==== */
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    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。

```
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; }
        }
    }
}
```

然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 PopNext-MPI 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。

```
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);
    ......
    while (!q.priority.empty())
    {
        std::size_t before = q.guesses.size();
        q.PopNextMPI();
        std::size_t local_added = q.guesses.size() - before;

        long long global_added = 0;
        MPI_Allreduce(&local_added, &global_added, 1,
                      MPI_LONG_LONG, MPI_SUM, MPI_COMM_WORLD);

        total_generated += global_added;
        curr_num        += global_added;
    ......
    MPI_Finalize();
    return 0;
}
```

# 4 结果分析

申请进程数为 16，在 O1 优化，O2 优化的条件下，分别测试串行程序和 mpi 并行化程序，得到如下测试结果：

| guess time | 串行实现 | mpi 优化 |
|---|---|---|
| O1 优化 | 0.6927 | 1.99255 |
| O2 优化 | 0.6827 | 1.78529 |

表 1: 性能测试结果 (16 进程)(单位:s)

测试结果显示，使用 mpi 优化的程序出现了负优化。可能的原因如下：

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。

# 5 进阶要求

## 5.1 基本思路

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/多线程进行并行的口令生成，现在请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。需要每个 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。

## 5.2 具体实现

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。

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。

```
1 if (rank == 0)
2 {
3     std::vector<PT> global_batch;
4     for (int i = 0; i < batch_size && !priority.empty(); ++i)
5     {
```

```
        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]);
    }
}
```

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。

```
else
{
    // 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;
        pt.deserialize(pt_data);
```

```
        local_batch.push_back(pt);
    }

    // 每个进程处理分配到的PT
    std::vector<PT> new_pts;
    for (const auto& pt : local_batch)
    {
        Generate(pt);
        std::vector<PT> pt_new_pts = pt.NewPTs();
        for (PT& new_pt : pt_new_pts)
        {
            CalProb(new_pt);
            new_pts.push_back(new_pt);
        }
    }
}
```

然后是结果收集阶段，实现了 MPI 环境下主从模式的任务分配与结果收集机制：从进程（rank 0）生成新 PT 后，将其序列化并按”数量 → 大小 → 数据”的顺序发送给主进程；主进程（rank=0）接收所有从进程的数据，反序列化为 PT 对象后，按概率将其有序插入优先队列，确保队列始终按概率降序排列，从而实现并行生成与集中调度的协同工作。

```
if (rank != 0)
{
    // 从进程: 将生成的新PT发送回主进程
    int send_size = new_pts.size();
    MPI_Send(&send_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    for (const auto& new_pt : new_pts)
    {
        std::string pt_data = new_pt.serialize();
        int data_size = pt_data.size();
        MPI_Send(&data_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send(pt_data.c_str(), data_size, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
    }
}
else
{
    // 主进程: 收集从进程发送的新PT
    for (int i = 1; i < size; ++i)
    {
        int recv_size;
        MPI_Recv(&recv_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        for (int j = 0; j < recv_size; ++j)
        {
            int data_size;
            MPI_Recv(&data_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            std::string pt_data(data_size, '\0');
            MPI_Recv(&pt_data[0], data_size, MPI_CHAR, i, 0, MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
```

```
            PT new_pt;
            new_pt.deserialize(pt_data);
            new_pts.push_back(new_pt);
        }
    }

    // 将所有新的PT放回优先队列
    for (const auto& new_pt : new_pts)
    {
        bool inserted = false;
        for (auto iter = priority.begin(); iter != priority.end(); ++iter)
        {
            if (new_pt.prob <= iter->prob && (iter + 1 == priority.end() ||
                new_pt.prob > (iter + 1)->prob))
            {
                priority.insert(iter + 1, new_pt);
                inserted = true;
                break;
            }
        }
        if (!inserted)
        {
            if (priority.empty() || new_pt.prob > priority.front().prob)
            {
                priority.insert(priority.begin(), new_pt);
            }
            else
            {
                priority.push_back(new_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。

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。在 PT 类中，我们同样相应的序列化和反序列化函数。其中 serialize 方法将 PT 对象的成员变量信息存储到 stringstream 中；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 PT 对象，同时对 content 中的每个 segment 对象进行反序列化操作。segment 的相关实现如下，pt 的实现类似。

```
// 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) {
            int f;
            iss >> f;
            ordered_freqs.push_back(f);
        }
    }
```

在主函数中调用相关方法，实现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。

```
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);

    double time_hash = 0;  // 用于MD5哈希的时间
    double time_guess = 0; // 哈希和猜测的总时长
    double time_train = 0; // 模型训练的总时长
    PriorityQueue q;
    if (rank == 0)
    {
        auto start_train = system_clock::now();
        q.m.train("/guessdata/Rockyou-singleLined-full.txt");
```

```
        q.m.order();
        auto end_train = system_clock::now();
        auto duration_train = duration_cast<microseconds>(end_train - start_train);
        time_train = double(duration_train.count()) * microseconds::period::num /
            microseconds::period::den;

        q.init();
        cout << "here" << endl;
    }

    int curr_num = 0;
    auto start = system_clock::now();
    int history = 0;
    while (!q.priority.empty())
    {
        q.PopNextBatchMPI(16, rank, size);

        ......
    }
    MPI_Finalize();
    return 0;
}
```

# 6 总结与反思

本次实验通过使用 mpi 进行了多进程并行实验，我学会了使用 mpi 加速的一般步骤和相关方法。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。