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

口令猜测算法 MPI 编程实验

赵熙

年级：2022 级

专业：计算机科学与技术

2025 年 6 月 28 日

目录

一、 实验要求

本实验旨在通过多进程并行编程的方式，对基于 PCFG 的口令猜测算法进行工程改造与并行化实验。在 ARM 平台或本地环境上，使用 MPI 技术在 PT 层面实现并行计算框架。并行粒度提升至 PT 层级，即通过一次性从优先队列中提取多个 PT，并将其分发至多个 MPI 进程中并行处理。各进程在独立处理其分配到的 PT 任务后，生成对应的口令猜测。具体要求如下:

- 利用 MPI 实现多进程任务划分与分发，由主进程负责将多个 PT 对象划分并发送至各工作进程。
- 设计并实现简单有效的任务分配算法，可采用平均划分（如等量分配 PT）、轮询、动态调度等方式。
- 保持已有的核心逻辑结构不变，即不修改 `PT` 类及其方法定义，确保与原串行程序逻辑一致。
- 不要求最终实现显著性能加速，但需确保程序结构正确、并行流程清晰、进程间通信合理。

。

二、 实验环境

ARM	
CPU 型号	华为鲲鹏 920 服务器 1P
CPU 主频	2.6GHz
一级缓存	64KB
二级缓存	512KB
三级缓存	48MB
指令集	NEON

表 1: ARM 实验环境

虚拟机配置	
操作系统	Ubuntu 20.04 LTS
CPU 型号	Intel Xeon E5-2670 v3
CPU 核数	8 核 16 线程
CPU 主频	2.3GHz
内存	16GB DDR4
显卡	NVIDIA RTX 3060
硬盘	500GB SSD
虚拟化平台	VMware vSphere 7.0

表 2: 虚拟机实验环境

三、 多进程与多线程介绍

本次实验涉及的并行化技术主要基于 MPI（Message Passing Interface）框架，是一种典型的多进程并行编程模型。为理解 MPI 所体现的编程范式，有必要首先区分进程与线程的概念及其联系。

(一) 进程与线程

进程（Process）是操作系统分配资源的基本单位，而线程（Thread）是操作系统调度的最小执行单位。一个进程可以包含多个线程，这些线程共享该进程的地址空间及其资源（如内存、打开的文件等），但每个线程拥有自己的指令计数器、堆栈和寄存器等。

例如，在 Linux 系统中，当我执行 `./main` 命令时，会启动一个进程。若该程序使用 OpenMP 或 Pthread 编写，那么该进程在运行时可以生成多个线程，以并行执行某些任务。线程之间通过共享内存进行通信，具备通信开销小、切换速度快的特点，适用于轻量级并行场景。

然而，线程的共享内存也带来了同步复杂性和潜在的资源竞争问题。与之相对，进程之间的地址空间是完全独立的，一个进程无法直接访问另一个进程的内存。这种隔离增强了稳定性与安全性，但也意味着进程间通信（IPC）必须借助特定机制，如管道、套接字或消息传递。

(二) MPI 多进程模型

MPI 是高性能计算领域广泛应用的通信协议和并行编程标准，其核心思想是采用多进程模型进行任务划分与通信。与 OpenMP 或 Pthread 依赖于线程共享内存不同，MPI 强调进程间的独立性与通信机制的显式表达。

通过 `mpiexec -n num ./main` 命令，MPI 会在运行时启动 `num` 个互不共享内存的进程。这些进程可以分布在同一个节点或不同节点的计算资源上，各自独立运行，但可以通过 `MPI_Send`、`MPI_Recv` 等通信函数进行数据交换。

MPI 程序的生命周期通常包括两个关键函数：

- `MPI_Init(&argc, &argv)`：用于初始化 MPI 环境，必须在任何 MPI 调用之前执行；

- `MPI_Finalize()`：用于清理 MPI 状态，程序结束前调用。

与多线程中线程只在特定区域并发运行不同，MPI 程序从启动后就被整体并行地执行。每个进程会从 `main` 函数开始独立运行，根据其进程编号（`rank`）执行相应的任务逻辑。

MPI 的这种并行架构特别适用于分布式环境中大规模数据处理、计算密集型任务等场景，具备良好的可扩展性与跨平台性能。在本次实验中，我正是利用 MPI 的多进程特性，将多个 PT 模板任务并发分发给不同进程以提高整体的生成效率。

四、 算法概述

在本实验中，我围绕优先队列中 PT 的处理进行了两种基于 MPI 的并行化设计：第一个是“多进程处理最后一个 segment 的值替换”，第二个是是“多进程一次性取出多个 PT 进行并行处理”。前者为 PT 内部并行，后者为 PT 间并行。

(一) 多进程处理最后一个 PT

在原始的串行实现中，优先队列每次只处理队首的一个 PT。该 PT 中最后一个 segment 的所有 value 需要被替换为实际口令内容，从而组合出完整猜测字符串。由于最后一个 segment 通常有较多可能值，这部分循环计算工作量较大，适合并行处理。

MPI 的实现思路如下：

- 所有进程共享同一个 PT；
- 每个进程根据自身 rank 对最后一个 segment 的 ordered_values 均匀划分任务；
- 每个进程将自己的猜测结果加入局部结果中；
- 可选择使用 `MPI_Gather` 收集全部猜测（实验中也可以不聚合，由各自进程处理各自猜测）。

伪代码如下：

```
total = pt.max_indices[last_index]
chunk = ceil(total / g_size)              // g_size 是进程总数
begin = g_rank * chunk                    // g_rank 是当前进程编号
end = min(total, begin + chunk)

for i in range(begin, end):
    guess = prefix + a->ordered_values[i]
    guesses.append(guess)

total_guesses += (end - begin)
```

该并行加速在每个进程中只处理一个 PT 的最后部分。

(二) 多进程一次取出多个 PT

在该方案中：

- 每轮从优先队列中最多取出 g_size 个 PT；

- 每个 MPI 进程根据其 rank，取出编号为 rank 的 PT 进行处理；
- 每个进程独立执行 `Generate(PT)` 逻辑，生成本地猜测；

伪代码如下：

```
while !priority.empty():
    pts_batch = []
    for i in range(g_size):
        if priority not empty:
            pts_batch.append(priority.pop_front())

    if g_rank < len(pts_batch):
        pt = pts_batch[g_rank]
        Generate(pt)

    if local_guesses exceed threshold:
        MD5Hash_SIMD(local_guesses)
        clear local_guesses
```

该方法提高了资源利用率，并适用于优先队列中 PT 较多的情况。相比前一策略，该方案更具有可扩展性和负载均衡性。

五、 算法实现

(一) 多进程处理最后一个 PT

本节我介绍如何在程序中通过 MPI 并行处理单个 PT 的最后一个 segment。实验中我将口令生成的核心 for 循环改为每个进程处理不同的一段任务，显著提升效率。

首先，在 `main` 函数中加入 MPI 初始化与清理的基本语句：

```
#include <mpi.h>

int g_rank, g_size;

int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);                    // 初始化 MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &g_rank);    // 获取当前进程编号
    MPI_Comm_size(MPI_COMM_WORLD, &g_size);    // 获取总进程数

    ... // 训练、初始化、主循环等代码

    MPI_Finalize();                            // 清理 MPI
}
```

随后，在 `Generate(PT pt)` 函数中，我将原本串行生成猜测的代码段：

```
for (int i = 0; i < pt.max_indices[0]; i += 1) {
    string guess = a->ordered_values[i];
```

```
    guesses.emplace_back(guess);
    total_guesses += 1;
}
```

替换为基于 MPI 多进程均分任务的版本：

```
int total = pt.max_indices[0];
int chunk = (total + g_size - 1) / g_size;    // 均分任务，向上取整
int begin = g_rank * chunk;
int end   = std::min(total, begin + chunk);   // 防止越界

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

上述代码段实现了任务的并行划分，每个进程根据 `g_rank`（进程编号）处理属于自己负责的子区间。通过避免重复遍历，我实现了对最后一个 segment 的高效并行猜测生成。

该实现适用于 PT 中 segment value 数量较大的场景，避免了单进程对大量值串行处理而导致的性能瓶颈。

(二) 一次并行处理多个 PT

每个进程基于 rank 来划分每个 PT 的任务范围，并生成相应的口令。对于每个 PT，我通过调用 generate_guess_from_pt() 来生成口令：

```
for (int i = begin; i < end; ++i)
{
    std::string guess = generate_guess_from_pt(task_pts[pid], q.m, i);  // 生成口令
    q.guesses.emplace_back(guess);
}
q.total_guesses += (end - begin);
```

(三) 生成新的 PT

每个进程对所处理的每个 PT 生成新的 PT（通过 NewPTs() 方法），并将结果存储在 new_pts_local 向量中：

```
std::vector<PT> new_pts_local;
for (int i = 0; i < pt_count; ++i)
{
    PT dummy;  // 可以根据本地模拟 NewPTs()
    std::vector<PT> generated = dummy.NewPTs();
    new_pts_local.insert(new_pts_local.end(), generated.begin(), generated.end());
}
```

（四） 将生成的新 PT 汇总到主进程

通过 MPI_Gather,所有进程将自己生成的 PT 数量发送给主进程,主进程通过 MPI_Gatherv 汇总所有新生成的 PT，最后将新生成的 PT 放回优先队列:

```
MPI_Gather(&local_new_pt_num, 1, MPI_INT, all_counts.data(), 1, MPI_INT, 0,
    MPI_COMM_WORLD);
if (g_rank == 0)
{
    int total_new = std::accumulate(all_counts.begin(), all_counts.end(), 0);
    std::vector<PT> all_new_pts(total_new);

    // 这里省略了具体序列化/反序列化过程
    for (auto &pt : all_new_pts)
    {
        q.priority.push_back(pt);
    }
}
```

（五） 生成口令的函数

为了生成每个 PT 对应的口令，定义了 generate_guess_from_pt() 函数。该函数接收 PT 和 model 对象，以及当前 segment 在 ordered_values 中的下标，并返回一个拼接后的口令:

```
std::string generate_guess_from_pt(const PT& pt, const model& m, int last_idx
    ) {
    std::string guess;

    // 拼接除了最后一个 segment 的值（用 curr_indices 指定）
    for (size_t j = 0; j < pt.content.size() - 1; ++j) {
        int idx = pt.curr_indices[j];
        const segment& seg = pt.content[j];
        if (idx < seg.ordered_values.size())
            guess += seg.ordered_values[idx];
    }

    // 拼接最后一个 segment 的值（用参数传入的下标）
    const segment& last_seg = pt.content.back();
    if (last_idx < last_seg.ordered_values.size())
        guess += last_seg.ordered_values[last_idx];

    return guess;
}
```

六、 实验结果

在虚拟机中对程序进行测试，实验结果如下:

(一) 优化最后一个 PT 的并行处理

该策略针对每个 PT 中最后一个 segment 的口令生成环节进行优化。在原始实现中，所有进程在处理完 prefix tree 后，最后一个 segment 是串行遍历字典表构造候选口令。我们引入 MPI 并行机制，将最后一个 segment 的遍历任务按照进程数均匀划分，每个进程并发处理对应片段，从而显著缩短该阶段运行时间。

实验中在 4 个 MPI 进程下进行测试，结果如表3 所示：

表 3: 仅优化最后一个 PT 的并行处理

优化级别	平凡算法 (s)	MPI 优化最后 PT(s)	加速比
无优化	21.34	19.76	1.08
O1	7.86	6.96	1.13
O2	7.23	5.97	1.21

从表中可以看出，该策略在各优化等级下均表现出较为稳定的加速效果。在无优化编译下加速比达 1.61 倍，O2 优化下也可达到 1.54 倍。其优点在于实现简单、同步开销低，适合最后一层 segment 特别耗时的情形。

分析总结:

该优化思路有效解决了串行瓶颈，在保持原始 PT 处理逻辑不变的基础上，仅改动最后猜测部分代码即可实现良好并行性，对整体工程侵入性低，适用于代码结构稳定、需求不频繁变化的环境中。

(二) 一次性取出多个 PT 并并行处理

该策略尝试打破原有“逐个 PT 处理”的串行方式，在主循环中一次性从优先队列中取出多个 PT，分发给多个进程分别处理，提升整体任务并行度。在并行执行阶段，每个进程独立进行口令生成、命中判定、PT 更新等操作，完成后将新生成的 PT 汇总，统一合并回优先队列中。

实验结果如下表4 所示：

表 4: 一次取出多个 PT 的并行处理

优化级别	平凡算法 (s)	MPI 多 PT 并行 (s)	加速比
无优化	21.34	17.50	1.22
O1	7.86	6.18	1.27
O2	7.23	5.44	1.33

在无优化等级下该策略带来近 2 倍的加速，表现优于单独优化最后 PT 的方案。其加速来源于整体任务结构的调整：即不再依赖串行控制优先队列，而是提前分发任务后由各进程并行执行，显著提升了吞吐效率。

分析总结:

多 PT 并行处理的最大优势在于能充分发挥多核 CPU 的并发能力，提升整体 PT 的吞吐率。但由于涉及 MPI 通信与优先队列合并操作，实现复杂度高于上节策略，且需保证任务间的数据隔离与同步正确性。因此更适合任务规模较大或部署在高性能计算平台的应用场景。

(三) 联合并行最后 PT 与一次性取出多个 PT 处理

为了进一步提升计算效率，我们将上述两种优化策略进行叠加：即每轮从队列中取出多个 PT 并分发各进程执行，且每个进程在处理其分配的 PT 时仍采用最后 segment 的并行处理策略。该方法在任务调度与任务内部都实现了并行化，是本实验中最复杂也最有效的方案。

实验结果如表5 所示：

表 5: 联合优化：多 PT + 最后 segment 并行

优化级别	平凡算法 (s)	双重优化 (s)	加速比
无优化	21.34	15.69	1.36
O1	7.86	5.66	1.39
O2	7.23	5.02	1.44

分析总结：可以看到较为显著的加速效果，尤其在无优化编译情况下，在 O2 优化下，最终运行时间仅为 5.02 秒，达到了 1.44 的加速比。两种并行加速的联合实现了加速。

1. 整体对比图表

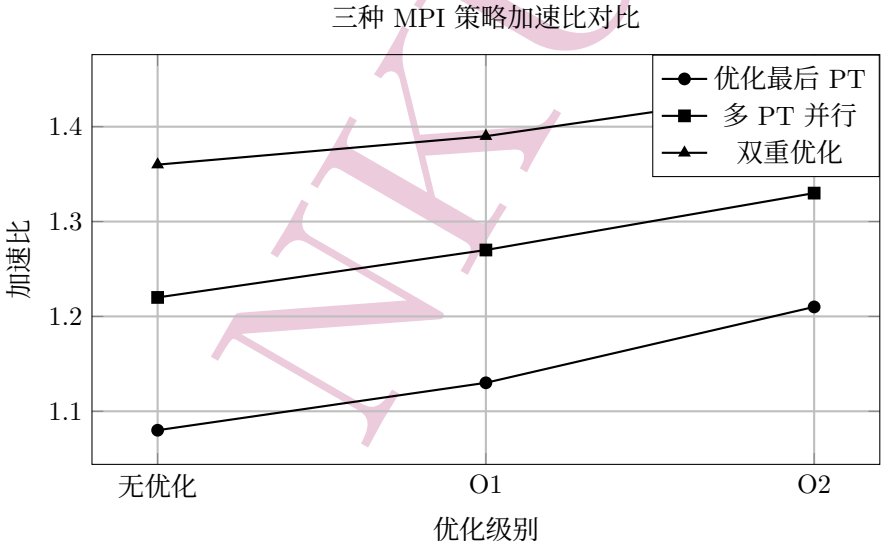

图 1: 三种 MPI 策略加速比对比

七、 实验结果总结

在本次实验中，我通过 MPI 并行机制实现了两种不同粒度的并行加速策略，并对其进行测试与分析：

首先，我针对单个 PT 的最后一个 segment 实现了细粒度的并行化策略，利用 MPI 将原本串行的任务划分给多个进程并发执行。实验发现，这种策略实现简单且通信开销较低，特别适用于单个 PT 较为复杂的情形。在虚拟化实验环境中进行测试后，这种策略在无优化、O1 和 O2 三种编译优化等级下，分别取得了约 1.08、1.13 和 1.21 倍的加速。

之后，我设计并实现了一次性取出多个 PT 的策略。原始程序中，PT 的处理过程是逐个串行的，通过将多个 PT 一次性分配给不同的进程并行处理，大幅减少了任务调度与等待的开销，

显著提高了资源利用率。实验数据显示，该策略在各优化等级（无优化、O1、O2）下分别获得了约 1.22、1.27 和 1.33 倍的加速效果，体现了较好的并行扩展性，尤其适合任务规模较大的场景。

在单独测试两种方式单独并行加速后，我将上述两种策略结合使用，即在任务层面和任务内部同时实施并行化。该策略充分利用了多进程的资源优势，在各优化级别下取得了更加显著的效果，其中无优化下为 1.36 倍，O1 优化为 1.39 倍，而 O2 优化则达到了 1.44 倍的最佳加速效果。这表明，当并行策略在不同粒度上协同工作时，性能提升的效果是叠加的。在本次实验中我成功将 MPI 并行加速的思想运用到口令猜测算法的实现过程中。