南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

并行程序设计课程报告

多进程编程实验

孙沐赟

年级：2023级

专业：计算机科学与技术

指导教师：王刚

2025 年 6 月 10 日

目录

1 使用多进程方法实现并行化

本次实验需要将上一个实验中的多线程的并行化方法使用多进程来实现，其实听起来只是将多线程转换为多进程来实现，但是，线程与进程之间的区别、进程之间的通信等问题还是给我带来了不小的麻烦。

下面我来讲述一下我实现的第一版多进程的代码。

1.1 多线程实现 = 多进程实现？

在最初修改程序时，首先我就尝试直接按照多线程的逻辑，在generate函数中进行主进程向各个子进程分配任务，之后各个进程来独立执行自己的任务，但是我很快就面临了一个问题：子进程似乎并没有按照我想象的方式来运行，每个子进程也进行了模型的训练，优先队列的初始化等主进程进行的一系列操作，最后相当于每个进程都串行的执行了一遍完整的流程。

后来，我想到是否可以通过一些控制，限制主进程进行模型的训练、使用“主-从（Master-Worker）”模型，主进程（Rank 0）负责分发任务（PTs）给任何空闲的从进程，并汇总所有从进程的结果，从进程（Rank >0）：不断向主进程请求任务，执行任务，然后将结果报告给主进程，再请求下一个任务，实现动态负载……由此，在一步一步的代码修改之下，我有了一步一步的进展，最终实现了多进程的优化

1.2 初始化与数据分发

与多线程不同的是，在服务器中我们申请多进程处理，程序会主动的在每个进程中进行运行，所以我们要正确的实现各个进程的初始化与任务的分发。

在程序开始时，所有进程首先进行初始化和数据同步，以确保每个工作者都拥有执行任务所需的所有信息。我们以主进程作为主导。首先，只有主进程会读取密码文件并独立完成整个概率模型的训练过程。训练完成后，主进程会将包含字母、数字和符号等概率统计的完整模型序列化并广播给所有其他工作者进程。同样地，主进程也会读取一个用于测试的目标密码集，并通过也通过广播的方式将其分发给所有进程。这个步骤确保了每个进程都拥有一份模型的本地副本和一份完整的测试集，为后续的并行生成和本地验证做好准备。

具体代码实现如下：

1.2.1 主进程训练

```
if (world_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();
        time_train = duration_cast<microseconds>(end_train -
            start_train).count() * 1e-6;
        cout << "Training completed in " << time_train << "
            seconds" << endl
        << flush;
}
```

这部分代码只在 world_rank 为 0 的主进程上执行。它首先记录训练开始的时间。然后，调用 q.m.train() 方法加载 /guessdata/Rockyou-singleLined-full.txt 文件来训练 PCFG 模型。q.m.order() 方法用于对训练后的模型数据进行排序，例如按频率对字母、数字、符号和预终端模式进行排序，这有助于后续的概率计算和猜测生成。最后，计算并打印出训练所花费的时间。

1.2.2 主进程广播训练模型

```
        BroadcastModel(q, 0);
```

在主进程完成模型训练后，它需要将训练好的模型数据分发给所有其他工作进程。BroadcastModel 函数是实现这一功能的关键。

```
void BroadcastModel(PriorityQueue &q, int root_rank)
{
        BroadcastSegmentsOptimized(q.m.letters, root_rank);
        BroadcastSegmentsOptimized(q.m.digits, root_rank);
        BroadcastSegmentsOptimized(q.m.symbols, root_rank);
}
```

BroadcastModel 函数本身是一个高层封装，它通过调用三次 BroadcastSegmentsOptimized 函数来完成模型的广播。BroadcastSegmentsOptimized 是一个更底层的函数，专门用于广播 vector 类型的数据。

具体来说，BroadcastModel 函数的职责是：

广播字母段（letters）数据: BroadcastSegmentsOptimized(q.m.letters, root_rank); 这行代码会将 PriorityQueue 对象 q 中模型 m 的 letters 成员（一个 vector 类型）广播到所有进程。letters 存储了模型中关于字母模式（例如，'a', 'ab', 'abc' 等）的统计信息，包括它们的类型、长度、总频率、有序值和有序频率。

广播数字段（digits）数据: BroadcastSegmentsOptimized(q.m.digits, root_rank); 类似地，这行代码会将 q.m.digits 广播到所有进程。digits 存储了模型中关于数字模式（例如，'1', '12', '123' 等）的统计信息。

广播符号段（symbols）数据: BroadcastSegmentsOptimized(q.m.symbols, root_rank); 最后，这行代码会广播 q.m.symbols 到所有进程。symbols 存储了模型中关于特殊字符模式（例如，'!', '@', '#$' 等）的统计信息。

通过这三次广播，所有工作进程都能获得模型中关于字母、数字和特殊字符的完整统计数据. 这些数据对于每个进程独立。计算密码模式的概率和生成候选密码至关重要，因为它们包含了生成有效猜测所需的所有频率和排序信息。

其中 BroadcastSegmentsOptimized 函数来分别广播模型中的字母、数字和符号段的数据。BroadcastSegmentsOptimized 函数内部通过 MPI 的 MPI_Bcast 调用，将主进程（rank 0）上的序列化数据（包括段的数量、类型、长度、频率、有序值和有序频率等）广播到所有其他进程。由于BroadcastSegmentsOptimized函数过长，我们稍后解释。这样，每个工作进程都拥有了模型的一个本地副本，可以独立地进行猜测生成。

1.2.3 主进程广播测试集

```
if (world_rank == 0)
{
```

```
        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++;
        }
        cout << "Rank␣0:␣Test␣set␣size␣=␣" << test_set.size() << endl
        << flush;
}
BroadcastTestSet(test_set, 0);
```

这部分代码也只在主进程（world_rank == 0）上执行，负责读取用于测试的目标密码集。主进程打开 /guessdata/Rockyou-singleLined-full.txt 文件，并读取前1,000,000个密码，将它们存储在一个 unordered_set 中。这样做是为了后续快速查找和验证生成的猜测是否成功破解了目标密码。随后，BroadcastTestSet 函数被调用。这个函数会将主进程中加载的测试集通过 MPI 广播给所有其他工作进程。它首先广播测试集的大小，然后逐个广播每个密码的长度和内容。这样，每个工作进程都拥有了完整的测试集副本，可以在本地验证自己生成的猜测。

1.3 并行猜测循环

数据分发完成后，所有进程会同步进入一个主循环中，协同进行密码的生成和验证，直到满足设定的终止条件。在每一次循环中，管理者进程（rank 0）会从其独有的优先级队列中取出一个当前概率最高的密码模板（PT），然后调用 BroadcastPT 函数将这个模板广播给所有工作者。所有进程在接收到同一个模板后，会立即调用 Generate 函数来并行地生成具体的密码猜测。Generate 函数内部通过一个 for 循环（for (int i = rank; i < total_vals; i += size)）利用每个进程的唯一 rank 值来分割工作，确保了每个进程生成不重叠的密码子集。密码生成后，每个进程会独立地验证自己生成的 local_guesses，并将本地的猜测数量和破解数量通过 MPI_Reduce 操作归约到管理者。最后，管理者负责累加全局结果，并检查是否达到终止条件（如猜测总数达上限），若满足，则广播一个 should_exit 标志，通知所有进程退出循环，程序结束。

具体代码实现如下：

1.3.1 主循环的开始与PT广播

```
bool should_exit = 0;
long long history = 0;
int total_cracked = 0;
double time_guess = 0;
double time_hash = 0;

auto start = system_clock::now();
```

```
while (!should_exit)
{
        int keep_going = 1;
        PT top_pt;
        if (world_rank == 0)
        {
                if (q.priority.empty())
                {
                        keep_going = 0;
                }
                else
                {
                        top_pt = q.priority.top();
                        q.priority.pop();
                }
        }
        MPI_Bcast(&keep_going, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (!keep_going)
        {
                should_exit = 1;
                break;
        }

        BroadcastPT(top_pt, 0);
```

这段代码标志着密码生成和验证主循环的开始。循环由 should_exit 标志控制，初始为 false。在每次循环迭代的开始，主进程（world_rank == 0）会检查其优先级队列 q.priority 是否为空。如果队列为空，表示没有更多的密码模板可以生成，主进程会设置 keep_going 为 0；否则，它会从队列中取出当前概率最高的密码模板 top_pt 并将其从队列中移除。随后，keep_going 的值会通过 MPI_Bcast 广播给所有进程，所有进程会据此判断是否需要退出循环。如果 keep_going 为 0，所有进程会设置 should_exit 为 true 并跳出循环。如果循环需要继续，主进程会调用 BroadcastPT(top_pt, 0) 函数，将当前取出的 top_pt 广播给所有工作进程，确保所有进程都在同一轮中使用相同的模板来生成猜测。

1.3.2 密码生成与本地验证

```
vector<string> local_guesses;
auto start_guess = system_clock::now();
q.Generate(top_pt, local_guesses, world_rank, world_size);
auto end_guess = system_clock::now();

auto start_hash = system_clock::now();
int local_cracked = 0;
for (const string &guess : local_guesses)
{
```

```
        if (test_set.count(guess))
        {
                local_cracked++;
        }
}
auto end_hash = system_clock::now();
time_hash += duration_cast<microseconds>(end_hash - start_hash).
    count() * 1e-6;

cout << "Rank " << world_rank << ": local_cracked = " <<
    local_cracked << endl
<< flush;
```

在所有进程接收到相同的 top_pt 模板后，它们会并行地执行密码生成和本地验证。每个进程会创建一个 local_guesses 向量来存储自己生成的密码。接着，调用 q.Generate(top_pt, local_guesses, world_rank, world_size) 函数，该函数利用当前进程的 world_rank 和总进程数 world_size 来实现工作分割，确保每个进程生成不重叠的猜测子集，从而避免重复计算。密码生成完成后，每个进程会立即对 local_guesses 中的每个猜测进行本地验证。它们通过查找 test_set（预先广播的测试密码集合）来判断猜测是否成功破解了目标密码，并统计本地破解的密码数量 local_cracked。同时，这段代码也对哈希验证的时间进行了累计。

1.3.3 结果归约与终止条件检查

```
int local_count = local_guesses.size();
int global_count = 0;
int cracked = 0;
MPI_Reduce(&local_count, &global_count, 1, MPI_INT, MPI_SUM, 0,
    MPI_COMM_WORLD);
MPI_Reduce(&local_cracked, &cracked, 1, MPI_INT, MPI_SUM, 0,
    MPI_COMM_WORLD);

if (world_rank == 0)
{
        history += global_count;
        total_cracked += cracked;
        cout << "Guesses generated: " << history << endl
        << flush;
        if (history >= 10000000)
        {
                should_exit = 1;
        }
}
MPI_Bcast(&should_exit, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

在本地密码生成和验证完成后，所有进程会通过 MPI_Reduce 操作将各自的结果归约到主

进程。首先，每个进程将其本地生成的猜测数量 local_count 累加到主进程的 global_count 中。接着，将本地破解的密码数量 local_cracked 累加到主进程的 cracked 变量中，从而得到本次循环的全局破解数量。只有主进程（world_rank == 0）会累加 global_count 到 history（总猜测数量）以及 cracked 到 total_cracked（总破解数量）。主进程会持续打印当前生成的猜测总数。随后，主进程会检查 history 是否达到了预设的终止条件（例如，生成了超过 10,000,000 个猜测）。如果达到终止条件，主进程会设置 should_exit 标志为 true，并通过 MPI_Bcast 将此标志广播给所有进程。所有进程接收到 should_exit 为 true 的信号后，就会退出主循环，从而结束整个并行猜测过程。

1.3.4 最终时间统计与结果输出

```
auto end = system_clock::now();
time_guess = duration_cast<microseconds>(end - start).count() * 1e-6;

if (world_rank == 0)
{
        cout << "Guess time: " << time_guess - time_hash << " seconds" << endl
        << flush;
        cout << "Hash time: " << time_hash << " seconds" << endl
        << flush;
        cout << "Train time: " << time_train << " seconds" << endl
        << flush;
        cout << "Cracked: " << total_cracked << endl
        << flush;
}
```

在主循环结束后，所有进程都会记录程序结束的时间，并计算出总的猜测时间 (time_guess)。最终的结果输出仅在主进程 (world_rank == 0) 上执行。主进程会打印出几个关键的性能指标：实际用于生成猜测的时间（通过 time_guess - time_hash 计算，排除了哈希验证的时间），纯粹进行哈希验证的时间 (time_hash)，以及模型训练所花费的时间 (time_train)。最后，主进程会输出整个并行猜测过程中成功破解的密码总数 total_cracked。这些统计数据提供了对程序整体效率和效果的全面评估。

1.4 generate函数的多进程实现

在讲述了主函数的执行过程后，我们对整个代码的框架已经有了基础的了解，下面我们来看一些具体细节的实现。

首先时generate函数如何进行多进程优化：

1.4.1 函数定义与 MPI 初始化

```
void PriorityQueue::Generate(const PT &pt, vector<string> &local_guesses)
{
```

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

        local_guesses.clear();
```

这段代码定义了 PriorityQueue 类的一个成员函数 Generate，它接收一个常量引用 PT &pt 作为密码模板和一个 vector<string> &local_guesses 用于存储生成的密码猜测。在函数开始时，它通过 MPI_Comm_rank 和 MPI_Comm_size 获取当前进程的 rank（进程ID）和 MPI 通信器中的总进程数 size。这两个变量是实现并行化工作的关键，允许每个进程根据自己的 rank 来处理特定的数据子集。local_guesses 在每次调用前都会被清空，以确保只包含当前 PT 生成的密码。

1.4.2 处理只包含一个段的PT

```
if (pt.content.size() == 1)
{
        segment *a = nullptr;
        if (pt.content[0].type == 1)
        a = &m.letters[m.FindLetter(pt.content[0])];
        else if (pt.content[0].type == 2)
        a = &m.digits[m.FindDigit(pt.content[0])];
        else if (pt.content[0].type == 3)
        a = &m.symbols[m.FindSymbol(pt.content[0])];

        if (a && !pt.max_indices.empty())
        {
                int total_vals = pt.max_indices[0];
                local_guesses.reserve(total_vals / size + 1);
                for (int i = rank; i < total_vals; i += size)
                {
                        local_guesses.emplace_back(a->
                           ordered_values[i]);
                }
        }
}
```

这段代码处理 PT 只包含一个段（pt.content.size() == 1）的简单情况。它首先根据这个唯一段的类型（字母、数字或符号）从模型 m 中找到对应的 segment 数据。如果找到了有效的 segment 并且 pt.max_indices 不为空，它会获取该段可以生成的所有值的总数 total_vals. 接着，为了优化性能，local_guesses 会预分配内存。核心的并行化逻辑体现在 for 循环中：for (int i = rank; i < total_vals; i += size)。这个循环让每个进程以 size 为步长，从 rank 开始遍历索引，从而确保每个进程生成该 segment 所有可能值的一个不重叠的子集。最终，生成的具体值会被添加到 local_guesses 向量中。

1.4.3 处理包含多个段的PT

```
else
{
        string prefix;
        int seg_idx = 0;
        for (int idx : pt.curr_indices)
        {
                const auto &seg = pt.content[seg_idx];
                if (seg.type == 1)
                prefix += m.letters[m.FindLetter(seg)].
                    ordered_values[idx];
                else if (seg.type == 2)
                prefix += m.digits[m.FindDigit(seg)].
                    ordered_values[idx];
                else if (seg.type == 3)
                prefix += m.symbols[m.FindSymbol(seg)].
                    ordered_values[idx];
                seg_idx++;
                if (seg_idx == pt.content.size() - 1)
                break;
        }

        // ... 后续处理最后一个段的逻辑() ...
}
```

当 PT 包含多个段时，这段代码首先负责构建已确定部分的密码前缀。它遍历 pt.curr_indices，其中存储了每个已展开段在 ordered_values 中的具体索引。对于 pt.content 中的每个段，它根据其类型（字母、数字或符号）从模型 m 中找到对应的 segment，并取出 ordered_values 中由 idx 指定的具体值. 这些值被依次拼接起来，形成 prefix 字符串。循环会一直进行，直到处理到倒数第二个段，因为最后一个段的展开将通过接下来的并行化逻辑来完成。

1.4.4 处理最后一个段与并行化生成

```
else
{
        // ... 构建前缀的逻辑() ...

        segment *a = nullptr;
        int last = pt.content.size() - 1;
        if (pt.content[last].type == 1)
        a = &m.letters[m.FindLetter(pt.content[last])];
        else if (pt.content[last].type == 2)
        a = &m.digits[m.FindDigit(pt.content[last])];
        else if (pt.content[last].type == 3)
        a = &m.symbols[m.FindSymbol(pt.content[last])];
```

```
13
14          if (a && !pt.max_indices.empty())
15          {
16                  int total_vals = pt.max_indices[last];
17                  local_guesses.reserve(total_vals / size + 1);
18                  for (int i = rank; i < total_vals; i += size)
19                  {
20                          local_guesses.emplace_back(prefix + a->
                                ordered_values[i]);
21                  }
22          }
23  }
```

这段代码是针对包含多个段的 PT，在构建完前缀后，处理模板中最后一个段的并行化逻辑。它首先确定 PT 中的最后一个段 (pt.content[last])，并根据其类型从模型 m 中获取相应的 segment 数据 a. total_vals 存储了该最后一个 segment 可能生成的所有具体值的总数. 类似于只有一个段的情况，这里也进行内存预分配以优化性能。核心并行化循环 for (int i = rank; i < total_vals; i += size) 再次出现，它利用进程的 rank 和总进程数 size 来划分最后一个段的生成任务。每个进程负责生成其分配到的具体值，并将这些值与之前构建的 prefix 拼接，形成完整的密码猜测，最后添加到 local_guesses 向量中. 通过这种方式，Generate 函数能够有效地将复杂密码模板的最终展开工作分散到所有 MPI 进程上。

1.5 广播函数的实现

在上面的代码执行过程中，主进程向子进程分配广播模型、分配任务，子进程向主进程返回各自处理的结果等过程都需要依靠进程之间的广播来实现。下面我们来分析以下进程之间的广播是如何来实现的。

1.5.1 BroadcastSegmentsOptimized：主进程广播数据

函数的初始化部分：

```
1  void BroadcastSegmentsOptimized(vector<segment> &segments, int
       root_rank)
2  {
3          int world_rank;
4          MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
5
6          vector<int> int_buffer;
7          vector<char> char_buffer;
```

这段代码定义了 BroadcastSegmentsOptimized 函数，它接收一个 segment 向量的引用 segments 和一个整数 root_rank，表示执行广播的根进程的 rank。函数内部首先通过 MPI_Comm_rank 获取当前 MPI 进程的 world_rank，以便后续区分根进程与其他工作进程。接着，它声明了两个重要的缓冲区：int_buffer 是一个 int 类型的动态数组，用于临时存储 segment 对象中的所有整数数据；char_buffer 是一个 char 类型的动态数组，用于集中存储 segment 对象中所有字符串

（ordered_values 的内容）的字符数据。

数据序列化：

```
if (world_rank == root_rank)
{
        int_buffer.push_back(segments.size());
        for (const auto &seg : segments)
        {
                int_buffer.push_back(seg.type);
                int_buffer.push_back(seg.length);
                int_buffer.push_back(seg.total_freq);
                int_buffer.push_back(seg.ordered_values.size());
                int_buffer.push_back(seg.ordered_freqs.size());
                for (const auto &val : seg.ordered_values)
                {
                        int_buffer.push_back(val.length());
                        char_buffer.insert(char_buffer.end(), val.
                            begin(), val.end());
                }
                int_buffer.insert(int_buffer.end(), seg.
                    ordered_freqs.begin(), seg.ordered_freqs.end())
                    ;
        }
}
```

这段代码块仅在根进程 (world_rank == root_rank) 上执行(即，在主进程运行)，其核心任务是将复杂的 vector数据结构进行序列化。首先，根进程将 segments 向量中 segment 对象的总数量添加到 int_buffer，这有助于接收方正确地重建 segments 向量。然后，它遍历 segments 中的每一个 segment 对象，并将其内部的各个成员变量（包括 type、length、total_freq 以及 ordered_values 和 ordered_freqs 的大小）依次推入 int_buffer。特别地，对于 ordered_values 中包含的字符串，根进程会先将其长度推入 int_buffer，然后将其所有字符内容追加到 char_buffer 的末尾。这种集中存储字符串字符数据的方式可以优化后续的 MPI 通信效率。最后，ordered_freqs 向量的所有整数值也被直接插入到 int_buffer 中，完成了一个 segment 对象的完整序列化。

广播缓冲区大小：

```
long long int_buffer_size = (world_rank == root_rank) ? int_buffer
    .size() : 0;
MPI_Bcast(&int_buffer_size, 1, MPI_LONG_LONG, root_rank,
    MPI_COMM_WORLD);
if (world_rank != root_rank)
int_buffer.resize(int_buffer_size);

long long char_buffer_size = (world_rank == root_rank) ?
```

```
    char_buffer.size() : 0;
MPI_Bcast(&char_buffer_size, 1, MPI_LONG_LONG, root_rank,
    MPI_COMM_WORLD);
if (world_rank != root_rank)
char_buffer.resize(char_buffer_size);
```

此段代码负责在正式广播数据内容之前，先将两个缓冲区 (int_buffer 和 char_buffer) 的大小广播给所有进程。对于根进程，它会获取并初始化 int_buffer_size 和 char_buffer_size 为各自缓冲区的实际大小；而对于其他进程，这些大小初始为 0。随后，通过两次 MPI_Bcast 调用，根进程将这两个大小值分别发送给所有参与的 MPI 进程。非根进程在接收到这些大小信息后，会立即调用 resize() 方法来调整其本地 int_buffer 和 char_buffer 的容量，确保它们有足够的空间来接收即将到来的实际数据。这种两阶段广播（先广播大小，再广播内容）是 MPI 编程中的一种常见优化策略，可以避免接收进程在接收数据时进行多次内存重新分配，从而提高效率。

广播缓冲区内容：

```
MPI_Bcast(int_buffer.data(), int_buffer_size, MPI_INT, root_rank,
    MPI_COMM_WORLD);
MPI_Bcast(char_buffer.data(), char_buffer_size, MPI_CHAR,
    root_rank, MPI_COMM_WORLD);
```

在所有进程都已知道即将接收的数据大小并准备好缓冲区后，这两行代码执行了实际的数据内容广播。MPI_Bcast 函数的第一个参数 int_buffer.data() 和 char_buffer.data() 分别提供了 int_buffer 和 char_buffer 底层数组的指针，允许 MPI 直接访问和传输这些数据。第二个参数 int_buffer_size 和 char_buffer_size 指明了要传输的数据元素数量，而 MPI_INT 和 MPI_CHAR 则指定了数据类型。通过这两个 MPI_Bcast 调用，根进程将之前序列化好的所有整数数据和字符数据完整地传输到所有其他工作进程，此时所有进程的 int_buffer 和 char_buffer 都将包含相同的数据。

数据反序列化：

```
    if (world_rank != root_rank)
{
        segments.clear();
        if (int_buffer.empty())
        return;
        size_t int_idx = 0;
        size_t char_idx = 0;
        int num_segments = int_buffer[int_idx++];
        segments.resize(num_segments);
        for (int i = 0; i < num_segments; ++i)
        {
                segments[i].type = int_buffer[int_idx++];
                segments[i].length = int_buffer[int_idx++];
                segments[i].total_freq = int_buffer[int_idx++];
                int val_count = int_buffer[int_idx++];
```

```
                int freq_count = int_buffer[int_idx++];
                segments[i].ordered_values.resize(val_count);
                segments[i].ordered_freqs.resize(freq_count);
                for (int j = 0; j < val_count; ++j)
                {
                        int len = int_buffer[int_idx++];
                        if (len > 0)
                        {
                                if (char_idx + len > char_buffer.
                                    size())
                                {
                                        cerr << "Rank␣" <<
                                            world_rank << ":␣Fatal␣
                                            error␣during␣
                                            deserialization.␣Char␣
                                            buffer␣overflow." <<
                                            endl;
                                        MPI_Abort(MPI_COMM_WORLD,
                                            -1);
                                }
                                segments[i].ordered_values[j].
                                    assign(&char_buffer[char_idx],
                                    len);
                                char_idx += len;
                        }
                }
                for (int j = 0; j < freq_count; ++j)
                {
                        segments[i].ordered_freqs[j] = int_buffer[
                            int_idx++];
                }
        }
}
```

这段代码块仅在非根进程 (world_rank != root_rank) 上执行，负责将从根进程接收到的原始数据（存储在 int_buffer 和 char_buffer 中）反序列化并重建为 vector¡segment¿ 对象。首先，非根进程会清空其本地的 segments 向量，并检查 int_buffer 是否为空。然后，它使用两个索引 int_idx 和 char_idx 来追踪在两个缓冲区中的读取位置。它首先从 int_buffer 读取 segment 对象的总数量 num_segments，并根据此数量调整本地 segments 向量的大小。接着，进入一个循环，逐个重建 segment 对象。在每次循环中，它从 int_buffer 依次读取 segment 的基本属性、ordered_values 和 ordered_freqs 的大小，并调整对应的 vector 成员大小。对于 ordered_values 中的每个字符串，它会先从 int_buffer 读取其长度，然后从 char_buffer 中提取相应长度的字符来重构字符串。此处还包含了一个重要的缓冲区溢出检查，以防止因数据损坏导致的安全问题。最后，它从 int_buffer 中读取 ordered_freqs 的所有整数值，从而完整地重建了每个 segment 对

象，确保每个工作进程都拥有了与根进程一致的模型数据。

1.5.2 BroadcastTestSet：主进程广播测试集

函数定义与初始化：

```
void BroadcastTestSet(unordered_set<string> &test_set, int
    root_rank)
{
        int world_rank;
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
        vector<string> temp_vec;
        if (world_rank == root_rank)
        temp_vec = vector<string>(test_set.begin(), test_set.end()
            );
```

这段代码定义了 BroadcastTestSet 函数，它接收一个 unordered_set¡string¿ 类型的引用 test_set（用于在根进程存储原始测试集，在其他进程中接收广播数据），以及一个整数 root_rank（指定广播的源进程）。函数首先通过 MPI_Comm_rank 获取当前 MPI 进程的 world_rank。接着，它声明了一个临时的 vector¡string¿ temp_vec。这个 temp_vec 仅在根进程 (world_rank == root_rank) 中被初始化，将 test_set（一个无序集合）中的所有字符串复制到这个有序的向量中。将无序集合转换为有序向量是为了便于后续通过索引遍历并逐个广播字符串。

广播测试集大小：

```
int size = temp_vec.size();
MPI_Bcast(&size, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
```

在将字符串从 unordered_set 转换为 vector 后，这段代码将 temp_vec 的大小 (size) 广播给所有进程。对于根进程，size 就是 temp_vec 的实际大小；对于其他进程，size 初始为未定义值，但会通过 MPI_Bcast 从根进程接收到正确的大小。这个步骤非常关键，因为它允许所有非根进程知道即将接收多少个字符串，从而为后续的循环和内存分配做好准备。MPI_Bcast 确保了所有进程同步地获得测试集的总条目数。

循环广播每个字符串：

```
for (int i = 0; i < size; ++i)
{
        int len = (world_rank == root_rank) ? temp_vec[i].size() :
            0;
        MPI_Bcast(&len, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
        vector<char> buffer(len);
        if (world_rank == root_rank)
        copy(temp_vec[i].begin(), temp_vec[i].end(), buffer.begin
            ());
        MPI_Bcast(buffer.data(), len, MPI_CHAR, root_rank,
            MPI_COMM_WORLD);
        if (world_rank != root_rank)
```

```
10          test_set.insert(string(buffer.begin(), buffer.end()));
11  }
```

这段代码是 BroadcastTestSet 函数的核心，它在一个循环中逐个广播测试集中的每个字符串。循环会执行 size 次，其中 size 是测试集中字符串的总数量。在每一次循环迭代中，首先，根进程 (world_rank == root_rank) 会获取 temp_vec 中当前字符串（temp_vec[i]）的长度 len，而其他进程则将 len 初始化为 0。随后，通过一个 MPI_Bcast 调用，这个字符串的长度 len 会被根进程广播给所有参与的进程，确保所有进程都知道当前字符串的预期长度。

在进行完上述操作之后，所有进程都会根据接收到的 len 值创建一个 vector¡char¿ 类型的缓冲区 buffer，其大小恰好足以存储当前字符串的字符数据。如果当前进程是根进程，它会将 temp_vec[i] 的字符内容精确地复制到这个 buffer 中。完成字符数据准备后，另一个 MPI_Bcast 调用将 buffer 中包含的实际字符数据从根进程广播给所有其他进程。

最后，对于非根进程 (world_rank != root_rank)，在成功接收到字符数据后，它们会利用 string(buffer.begin(), buffer.end()) 构造函数从 buffer 中重建完整的字符串，并将其插入到各自本地的 test_set (一个 unordered_set¡string¿) 中。通过这种逐个字符串的广播和重构机制，BroadcastTestSet 函数确保了主进程的整个测试集能够被所有工作进程高效、准确地复制到其本地内存中，为后续的并行密码验证奠定基础。

1.5.3 BroadcastPT：主进程广播数据

BroadcastPT 函数的主要作用是将主进程（根进程）从优先级队列中取出的一个 PT (Preterm Template，密码模板) 对象广播给所有其他工作进程。PT 对象包含了密码模板的结构信息（如包含哪些类型的段、每个段的长度）、当前展开的状态（如已展开段的索引），以及与概率相关的数值。由于 PT 是一个自定义的复杂C++对象，它不能直接通过单个 MPI 调用进行广播，因此该函数需要手动序列化其关键成员数据，并通过一系列的 MPI_Bcast 调用将其分发给所有进程，确保所有工作进程都能获取到相同的、用于生成下一批密码猜测的模板。

函数定义与 content 向量大小的广播

```
1  void BroadcastPT(PT &pt, int root_rank)
2  {
3          int world_rank;
4          MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
5          int num_seg = (world_rank == root_rank) ? pt.content.size
               () : 0;
6          MPI_Bcast(&num_seg, 1, MPI_INT, root_rank, MPI_COMM_WORLD)
               ;
7          if (world_rank != root_rank)
8          pt.content.resize(num_seg);
```

这段代码定义了 BroadcastPT 函数，它接收一个 PT 对象的引用 pt 和一个整数 root_rank，后者指定了广播的源进程。函数首先通过 MPI_Comm_rank 获取当前进程的 world_rank。为了能够正确地重建 PT 对象的 content 向量（vector¡segment¿ 类型），根进程会获取其 pt.content.size() 作为 num_seg 的值，而其他进程则将 num_seg 初始化为 0。随后，MPI_Bcast 调用将这个 num_seg 值从根进程广播给所有其他进程。非根进程在接收到正确的 num_seg 后，会立即调用 pt.content.resize(num_seg) 来调整其本地 pt.content 向量的大小，为接收 segment 数据做好准

备。这个步骤是数据传输的第一个关键点，确保了接收方为后续的 segment 数据分配了正确的内存空间。

循环广播 content 向量中每个 segment 的 type 和 length

```
for (int i = 0; i < num_seg; ++i)
{
        int type = 0, length = 0;
        if (world_rank == root_rank)
        {
                type = pt.content[i].type;
                length = pt.content[i].length;
        }
        MPI_Bcast(&type, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
        MPI_Bcast(&length, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
        if (world_rank != root_rank)
        {
                pt.content[i].type = type;
                pt.content[i].length = length;
        }
}
```

在所有进程都知道 content 向量的正确大小之后，这段代码进入一个循环，逐个广播 pt.content 向量中每个 segment 对象的关键属性：type 和 length。在每次循环迭代中，根进程会从 pt.content[i] 中提取当前的 segment 的 type 和 length 值，而其他进程将这两个变量初始化为 0。接着，通过两次独立的 MPI_Bcast 调用，type 和 length 的值分别从根进程广播给所有其他进程。非根进程在接收到这些值后，会将其赋值给本地 pt.content[i].type 和 pt.content[i].length，从而重建 PT 对象中每个 segment 的基本结构信息。需要注意的是，segment 对象中的 ordered_values、ordered_freqs 和 total_freq 等详细统计数据无需在这里广播，因为它们是模型的一部分，已经在 BroadcastModel 阶段完成同步。

广播 curr_indices 和 max_indices 向量

```
auto broadcast_vec = [&](vector<int> &vec)
{
        int size = (world_rank == root_rank) ? vec.size() : 0;
        MPI_Bcast(&size, 1, MPI_INT, root_rank, MPI_COMM_WORLD);
        if (world_rank != root_rank)
        vec.resize(size);
        if (size > 0)
        MPI_Bcast(vec.data(), size, MPI_INT, root_rank,
            MPI_COMM_WORLD);
};
broadcast_vec(pt.curr_indices);
broadcast_vec(pt.max_indices);
```

为了简化和复用代码，我定义了一个 lambda 辅助函数 broadcast_vec，专门用于广播 vector<int> 类型的数据。这个 broadcast_vec 函数内部实现了标准的 MPI 向量广播模式：它首先广播向量的大小，然后非根进程根据接收到的大小调整其本地向量的容量，最后再广播向量的实际数据。在定义完 broadcast_vec 之后，它被两次调用，分别用于广播 pt.curr_indices 和 pt.max_indices 两个向量。pt.curr_indices 存储了当前 PT 已展开段的具体取值索引，而 pt.max_indices 存储了每个段可能取值的最大数量。通过这两个广播，PT 对象的当前状态和其可扩展的边界信息被完全同步到所有进程，使得它们能够基于相同的模板进行后续的密码生成操作。

此外，在进程之间的通信中也频繁的使用到了MPI库中自带的MPI_Bcast函数，通过 MPI_Bcast，所有工作进程都能获得执行其任务所需的相同数据和指令，从而实现并行计算和数据同步.

1.6 对于代码的进一步优化

在之前的代码中，如果使用 vector 来存储所有的 PT 结构，并且每次都需要找到概率最大的 PT，那么在每次迭代中都需要遍历整个 vector 来查找最大概率的元素。这会导致 O(N) 的时间复杂度。

我们能否替换数据结构，从而减少对于最大概率的 PT 的查找？我们可以使用最大堆的数据结构来替换现有的 vector 。定义一个 PTComparator 为了配合 std::priority_queue 使用，而 std::priority_queue 默认是一个最大堆（max-heap）。这意味着它总是将“最大”的元素放在顶部。

```
struct PTComparator
{
        bool operator()(const PT &a, const PT &b) const
        {
                return a.prob < b.prob;
        }
};
```

在 PTComparator 中，return a.prob ¡ b.prob; 表示当 a.prob 小于 b.prob 时，a 被认为是“小于” b 的。由于 std::priority_queue 是一个最大堆，它会根据这个比较规则来排列元素，使得概率值越大的 PT 结构越靠前（即被视为“更大”的元素），从而优先被处理。

通过将 PT 结构按照 prob（概率）从大到小排序，程序总是能够优先处理那些生成高质量（高概率）猜测的 PT。这意味着它能够更快地生成并检查更有可能匹配目标哈希值的密码，从而提高破解效率。使用最大堆的数据结构还有一个好处是，如果使用 std::vector 来存储所有的 PT 结构，并且每次都需要找到概率最大的 PT，那么在每次迭代中都需要遍历整个 vector 来查找最大概率的元素。这会导致 O(N) 的时间复杂度。但是在堆中，插入和删除操作的时间复杂度都是 O(logN)。通过这种方式，每次获取最高概率的 PT 只需要 O(1) 的时间（访问堆顶元素），而删除和重新插入的时间复杂度是 O(logN)，这比在 vector 中每次遍历查找和删除要高效得多。因此，优化数据结构（从 vector 到最大堆）能够加快程序的运行。

1.7 测试结果

下面，我们对程序进行运行结果的分析。

将使用多进程优化之后的函数与原串行算法分别在使用编译优化（mpic++）与不使用编译优化（强制在编译时输入-O0）进行对比，结果如下：

表 1: 多进程算法（版本1）与串行算法Guessing Time的比较

	不使用编译优化/单位(s)	编译优化/单位(s)
串行算法	8.00	1.26
多进程算法	0.27	0.11

表 2: 多进程算法（版本1）相比于串行算法Guessing Time的加速比

	不使用编译优化的加速比	编译优化的加速比
串行算法	1.00	1.00
多进程算法	29.63	11.45

从上述表格中可以看出，在经过多进程的优化与相关的数据从结构的优化之后，我们对于程序的优化效果还是很成功的!

但是为什么使用编译优化之后的加速比更低呢？经过对程序进行性能分析之后我发现，当开启编译优化之后，相比于不使用编译优化时cache率会下降，这会导致在程序运行时由于对于下一步的指令预测错误，这就使其虽然运行时间更短（可能对于执行的指令进行了其他的优化），但是cache率下降导致加速比反而有所下降。

2 另一种并行化思路

在实现了上述的并行方法之后，虽然实现效果已经较好，但是在实验进行到一半时，我产生了问题：现在我们限制主进程进行训练等操作，然后将结果广播到各个进程，之后我们对与单个 PT 的内部生成层面进行并行，但是，我们能否实现从一开始就将所有的 PT 划分给多个进程，在多个 PT 的处理层面进行并行？于是我基于上述的思想，完成了第二版代码的设计。

2.1 另一种并行化思路

1. 优先级队列的分布式管理

- **第一版:** 优先级队列 (`PriorityQueue::priority`) 仅存在于主进程（Rank 0）上。所有新的 `PT` 结构生成后，都由主进程维护并从其中取出进行处理。
- **第二版:** 每个 MPI 进程都拥有一个本地的优先级队列 (`PriorityQueue::local_priority`)。初始化时，主进程将最初的 `PT` 结构集（`m.ordered_pts`）根据进程数量均匀地分配到各个进程的本地队列中。
- **设计思想:** 在第一版中，随着猜测数量的增加，优先级队列可能会变得非常大，所有的 `push` 和 `pop` 操作都集中在 Rank 0，这会极大地限制可扩展性。第二版将队列操作分散到每个进程，让每个进程独立地管理自己的工作负载，从而实现更高效的并行。

2. 更细粒度的任务分配

- **第一版:** 主进程每次从队列中取出一个 `PT`，然后将其广播给所有工作进程。所有工作进程都基于这个相同的 `PT` 来并行生成其对应的猜测密码（通过 `Generate` 函数内的索引划分）。

- **第二版：**在主循环中，主进程不再广播单个 PT，而是将不同的 PT 结构直接发送给空闲的工作进程进行处理。每个工作进程接收到一个 PT 后，会独立地进行 Generate 操作，并将其产生的新 PT 结构推回到自己的本地队列中。
- **设计思想：**这种方式实现了在 PT 结构层面上的并行化。第一版中，所有进程同时处理一个 PT 的不同子任务，这可能导致负载不均（例如，某些 PT 结构生成的猜测数量很少）。第二版允许每个进程独立地处理完整的 PT 结构，只要有 PT 结构可供处理，工作进程就可以保持忙碌，从而提高整体的并行效率和负载均衡。

2.2 测试结果

在具体代码实现上其他思路大同小异，我们直接来看上述程序在的测试结果

表 3: 多进程算法（版本2）与串行算法Guessing Time的比较

	不使用编译优化/单位(s)	编译优化/单位(s)
串行算法	8.00	1.26
多进程算法	0.53	0.17

表 4: 多进程算法（版本2）相比于串行算法Guessing Time的加速比

	不使用编译优化的加速比	编译优化的加速比
串行算法	1.00	1.00
多进程算法	15.09	7.41

从上述结果的分析来看，我们后续实现的版本2的多进程实现也有较好的加速效果，但是为什么优化的结果没有版本1好呢？经过分析发现，主进程的运行速度使远小于子进程的运行速度的，为什么出现这一问题：我认为是因为主进程在运行时不需要进行额外的调用进程之间的通信函数，但是子进程在运行时会不停的向主进程进行任务的提交，因此会大量的调用通信函数，所以子进程将会运行的更慢，从而降低了程序整体的速度。还有一个原因就是为了减小子进程的通信的消耗，我设置子进程在处理100个PT的生成之后再将结果汇报给主进程，来判断是否达到结束条件，这就会使最终实际猜测的口令数可能略大于我们限制的10000000个目标猜测数，这也会导致程序的运行速度变慢。

3 对于问题规模的进程数量的思考

现在我们进行口令猜测的数量上限为10 000 000，那么如果将其调整为100 000 000，便会猜测除更多的口令，但是，其运行时间也会成倍增长。在此，我对已经实现的多线程并行化优化在不同问题规模与不同进程数量的场景下重新测试，来观察问题规模与线程数量对于Guessing Time会有怎样的影响（在mpic++编译优化的基础上进行测试）。

口令猜测上限为10 000 000

表 5: 多进程版本1与版本2优化与串行算法Guessing Time的比较

进程数	串行	2	3	4	5	6	7	8
版本一	1.26	0.17	0.13	0.11	0.10	0.09	0.11	0.12
版本二	1.26	0.40	0.22	0.17	0.14	0.13	0.12	0.12

将上述表格使用折线图绘制，以更直观的查看线程对于Guessing Time的影响：

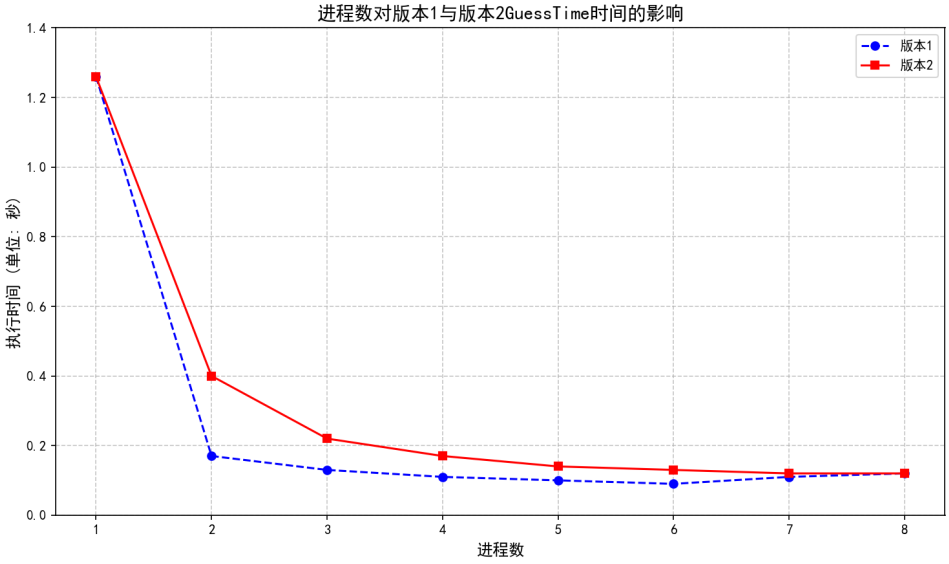

图 1: 口令猜测上限为10 000 000时Guessing Time随进程数的变化

从上面的分析中我们可以看出，总体而言，随着进程数的增多，Guessing Time也会呈现下降的趋势，这也符合我们对于实验结果的预期。而且，我们来横向对比两种问题规模的Guessing Time，其加速比大约都在11左右，说明我们的多线程优化有较好的效果。但是我们也发现，随着进程数的增大，每多加一个进程Guessing Time的时间减小的越来越慢，我认为这有可能是因为虽然使用了多进程进行任务的分配与处理，但是随着进程数的增多，进程之间的通信时间也会增加，所以测试结果体现为当进程数增多时Guessing Time不再下降甚至呈现上升趋势。

口令猜测上限为100 000 000

表 6: 多进程版本1与版本2优化与串行算法Guessing Time的比较

进程数	串行	2	3	4	5	6	7	8
版本一	5.30	2.92	2.39	2.11	2.19	1.65	1.72	1.96
版本二	5.30	8.22	4.08	3.74	2.93	2.36	1.77	1.11

将上述表格使用折线图绘制，以更直观的查看线程对于Guessing Time的影响：

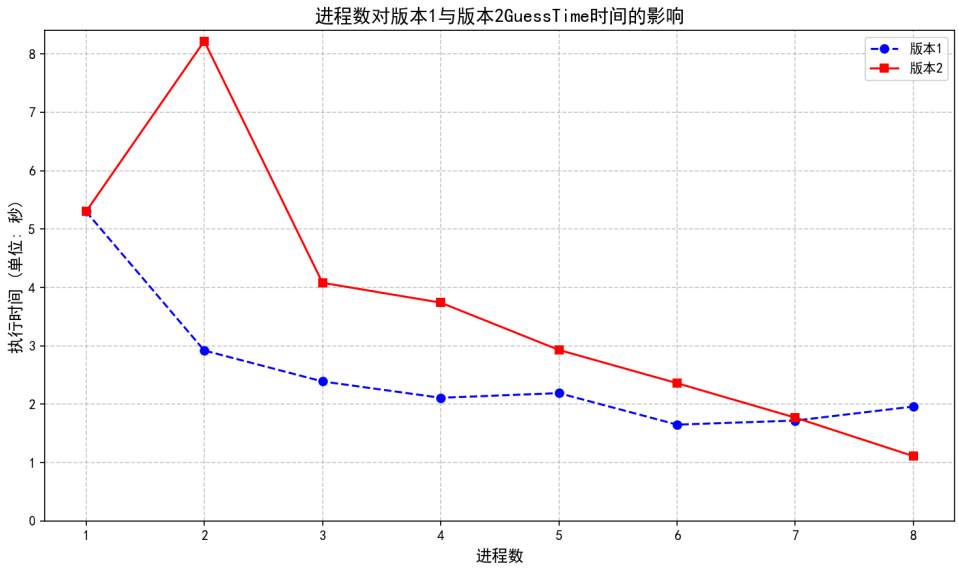

图 2: 口令猜测上限为100 000 000时Guessing Time随进程数的变化

在猜测口令上限来到100 000 000时，总体而言，随着进程数的增加，Guessing Time会随之降低，版本2的算法随进程增加的时间下降尤为明显，说明每个进程的通信时间的消耗是更小的，这也正好符合我最初的设计（设置每处理100个PT之后向主进程汇报任务，从而减少进程之间的通信）。而版本1在随着进程数增加，Guessing Time下降不明显，说明进程之间的通信是仍待向版本2的减少通信次数的方式进行优化。

4 实验总结

本次实验将多线程密码猜测并行化改造为多进程实现，在实验进行之初，我使用采用主-从进程模型来进行多进程的实现，其中主进程负责模型训练、数据分发和结果汇总，而从进程则请求并执行任务。为提升PT查找效率，我们将数据结构从std::vector优化为最大堆std::priority_queue。在后续实现的版本2中让每个子进程拥有本地队列，主进程分配不同的PT，每个子进程自行调用Generate函数，实现PT层面并行。经过在不同规模下对两个版本的代码进行测试，我发现版本一在小规模任务下表现更优，而版本二在更大规模任务下，其通信优化效果更为明显。实验结果清晰地表明，任务粒度和进程间通信开销是影响并行效率的关键因素，当进程数增加到一定程度时，通信开销可能抵消并行带来的收益，导致性能不再提升甚至下降。

这次实验让我深入理解了多进程并行化与线程化的根本区别，以及进程间通信在分布式环境中的重要性。通过手动序列化自定义数据结构并使用MPI进行广播，我掌握了在不同进程间有效传输数据的方法。同时，将数据结构从vector替换为最大堆显著提升了算法性能，这再次印证了选择合适数据结构对于解决特定问题的巨大影响。通过对两种并行策略的实现与对比，我更清晰地认识到在并行计算中如何权衡任务粒度与通信开销，这促使我思考未来如何更好地进行负载均衡和通信优化。此外，实验中观察到的编译器优化、缓存行为以及系统级通信瓶颈等因素对整体性能的复杂影响，也拓展了我对性能分析深度的理解。

我的本次实验的github仓库链接：https://github.com/fan-tuaner614/NKU-